# Combining algorithm-based fault tolerance and checkpointing for iterative solvers

Massimiliano Fasi, Yves Robert, Bora Uçar

RESEARCH REPORT
N° 8675
January 2015
Project-Team ROMA

ISSN 0249-6399    ISRN INRIA/RR--8675--FR+ENG

# Combining algorithm-based fault tolerance and checkpointing for iterative solvers

Massimiliano Fasi[*], Yves Robert[†], Bora Uçar[‡]

Project-Team ROMA

Research Report n° 8675 — January 2015 — 26 pages

**Abstract:** Several recent papers have introduced a periodic verification mechanism to detect silent errors in iterative solvers. Chen [PPoPP'13, pp. 167–176] has shown how to combine such a verification mechanism (a stability test checking the orthogonality of two vectors and recomputing the residual) with checkpointing: the idea is to verify every $d$ iterations, and to checkpoint every $c \times d$ iterations. When a silent error is detected by the verification mechanism, one can rollback to and re-execute from the last checkpoint. In this paper, we also propose to combine checkpointing and verification, but we use ABFT rather than stability tests. ABFT can be used for error detection, but also for error detection and correction, allowing a forward recovery (and no rollback nor re-execution) when a single error is detected. We present a performance model to compute the performance of all schemes, and we instantiate it using the Conjugate Gradient algorithm. Finally, we validate our new approach through a set of simulations.

**Key-words:** Fault-tolerant computing, Silent errors, Algorithm-based fault tolerance, Checkpointing, Sparse matrix-vector multiplication, Conjugate gradient method, Error detection, Error correction, Performance model.

[*] École Normale Supérieure de Lyon, France and Università di Bologna, Italy
[†] Laboratoire LIP, ENS Lyon, France & University of Tennesse Knoxville, USA
[‡] CNRS and LIP (CNRS-ENS Lyon-INRIA-UCBL), 46, allée d'Italie, ENS Lyon, Lyon F-69364, France.

**RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES**

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

# Une approche mixant les techniques ABFT et le checkpoint périodique pour les solveurs itératifs

**Résumé :** Plusieurs travaux récents ont introduit un mécanisme de vérification pour la détection des erreurs silencieuses dans l'exécution de solveurs itératifs. Chen [PPoPP'13, pp. 167–176] a montré comment combiner un tel mécanisme (un test de stabilité vérifiant l'orthogonalité de deus vectors et recalculate de résidu) avec le checkpoint périodique: dans son approche, on vérifie toutes les $d$ itérations, et on prend un checkpoint tous les $c \times d$ iterations. Quand une erreur silencieuse est détectée par le mécanisme de vérification, on peut revenuer au fernier checkpoint. Dans ce rapport, nous proposons également de combiner vérification et checkpoint, mais nous utilisons les techniques ABFT à la place du test de stabilité pour a vérification. Ces techniques ABFT peuvent server à détecter les errors, mais aussi à les corriger, ce qui permet de continuer l'exécution sans retour-arrière quand une seule erreur est détectée puis corrigée. Nous présentons un modèle de performance pour analyser notre approche, et nous l'instancions pour l'algorithme du Gradient Conjugué. Enfin, nous validons notre approche à l'aide d'un ensemble de simulations.

**Mots-clés :** checkpoint, vérification, erreur silencieuse, produit matrice-vecteur creux, algorithme du Gradient Conjugué, détection d'erreur, correction d'erreur, modèle de performance

# 1 Introduction

Silent errors (also known as silent data corruption) have become a significant concern in HPC environments [26]. There are many sources of silent errors, from bit flips in cache caused by cosmic radiations, to wrong results produced by the arithmetic and logic unit. The latter source becomes extremely important, because large computations are usually performed in low voltage mode, to reduce energy consumption, a setting that is known to dramatically reduce the dependability of the system.

The key problem with silent errors is *detection latency*: when a silent error strikes, the corrupted data is not identified immediately, but instead only when some anomaly is detected in the application behavior. This detection can occur with an arbitrary delay. As a consequence, the de-facto standard method for resilience, namely checkpoint and recovery, cannot be used any longer. Indeed, checkpoint and recovery applies to fail-stop errors (such as hardware crashes): such errors are detected immediately, and one can safely recover from the last checkpoint. On the contrary, because of the detection latency induced by silent errors, it is impossible to know when the error did strike, and hence to determine which checkpoint (if any) is valid and can be safely used to restore the application state. Even for the unrealistic scenario where an unlimited number of checkpoints could be kept in memory, there would remain the problem to identify a valid one!

In the absence of a resilience method, the only known remedy to silent errors is to re-execute the application from scratch as soon as a silent error are detected. On large-scale systems, the silent error rate grows linearly with the number of components, and several silent errors are expected to strike during the execution of a typical HPC application. The cost of re-executing the application one or more times becomes prohibitive, and another approach must be found.

Several recent papers have proposed to introduce a verification mechanism that could be applied periodically to detect silent errors. These papers mostly target iterative methods to solve sparse linear systems, because such methods are natural candidates to periodic detection. If we apply the verification mechanism every, say, $d$ iterations, then we have the opportunity to detect the error earlier, namely at most $d-1$ iterations after the actual faulty iteration, thereby stopping the progress of a flawed execution much earlier than without detection. However, the cost of the verification may be non-negligible in front of the cost of one application iteration, hence the need to trade-off for an adequate value of $d$. Verification can consist in testing the orthogonality of two vectors (cheap) or recomputing the residual (cost of a sparse matrix-vector product, more expensive). We survey several verification mechanisms in Section 2. An important caveat is that each approach applies only to a given type of silent errors: in other words, a *selective reliability* model is enforced, and those parts of the application that are not protected are assumed to be executed in reliable mode.

While verification mechanisms speed up the detection of silent errors, they are not able to provide their correction, hence to avoid a full re-execution of the application. A solution is to combine checkpointing with verification. If we apply the verification mechanism every $d$ iterations, we can checkpoint every $c \times d$ iterations, thereby limiting the amount of re-execution considerably. The last checkpoint is always valid, because it is preceded by a verification, and the error, if any, is always detected by one of the $c$ verifications applied before the next checkpoint is taken. This is exactly the approach proposed by Chen [8] for a variety of Krylov-based methods, including the widely used Conjugate Gradient (CG) algorithm. Chen [8] gives an equation for the overhead incurred by checkpointing and verification, and determines the best values of $c$ and $d$ by a numerical

resolution of the equation. In fact, computing the optimal values of $c$ and $d$ is a difficult problem. In the case of pure periodic checkpointing, closed-form approximations of the optimal period have been given by Young [33] and Daly [9]. However, when combining checkpointing and verification, the problem becomes much more difficult. To the best of our knowledge, there is no known closed-form formula, but a dynamic programming algorithm to compute the optimal repartition of checkpoints and verifications is available [2].

For linear algebra kernels, another widely used technique for silent error detection is the so-called *algorithm-based fault tolerance* (ABFT). The pioneering paper of Huang and Abraham [21] describes an algorithm capable of detecting and correcting a single silent error striking a dense matrix-matrix multiplication by means of row and column checksums. ABFT protection has been successfully applied to dense LU and QR factorizations, and more recently to sparse kernels, SpMxV (matrix-vector product) and triangular solve [29]. The overhead induced by ABFT is usually small, which makes it a good candidate for error detection at each iteration of the CG algorithm.

The beauty of ABFT is that it can *correct* errors in addition to detecting them. This comes at the price of more overhead, because several checksums are needed to detect and correct, while a single checksum was needed for sole detection. Still, being able to correct a silent error on the fly allows for *forward recovery.* No rollback, recovery nor re-execution are needed when a single silent error is detected at some iteration, because ABFT can correct it, and execution is safely resumed from that very same iteration. Only when two or more silent errors have struck within an iteration we do need to rollback to the last checkpoint. In many practical situations, only single errors will occur within an iteration, and most of the roll-back will be avoided. In turn, this will lead to less frequent checkpoints, and hence less overhead.

The major contribution of this paper is to build a performance model that allows to compare methods that combine verification and checkpointing. The verification mechanism is capable either of error detection, or of both error detection and correction. What are the optimal intervals for verifying and checkpointing, given the cost of an iteration, the overhead associated to each verification mechanism, the overhead associated to checkpoint and recovery, and the rate of silent errors? Our abstract model provides the optimal answer to this question, as a function of the cost of all application and resilience parameters.

We instantiate the model using the CG kernel, and compare the performance of two ABFT-based verification mechanisms The first ABFT-based scheme is called ABFT-Detection and is capable of error detection only, while the second scheme, ABFT-Correction, performs both detection and single error correction. Through numerical simulations, we compare the performance of both schemes with Online-Detection, the approach of Chen [8] (which we extend to recover from memory errors by checkpointing the sparse matrix in addition to the current iteration vectors). These simulations show 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.

The rest of the report is organized as follows. Section 2 provides an overview of related work. Section 3 provides background on ABFT techniques for the CG algorithm, and presents both the ABFT-Detection and ABFT-Correction approaches. Section 4 is devoted to the abstract performance model. Section 5 reports numerical simulations comparing the performance of ABFT-Detection, ABFT-Correction and Online-Detection. Finally, we outline main conclusions and directions for future work in Section 6.

# 2 Related work

In this section, we classify related work along the following topics: silent errors in general, verification mechanisms for iterative methods, and ABFT techniques.

## 2.1 Silent errors

Considerable efforts have been directed at error-checking to reveal silent errors. Error detection is usually very costly. Hardware mechanisms, such as ECC memory, can detect and even correct a fraction of errors, but in practice they are complemented with software techniques. The simplest technique is triple modular redundancy and voting [24], which induces a highly costly verification. For high-performance scientific applications, process replication (each process is equipped with a replica, and messages are quadruplicated) is proposed in the RedMPI library [15]. Elliot et al. [13] 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 comprehensive list of general-purpose techniques and references is provided by Lu, Zheng and Chien [23].

Application-specific information can be very useful to enable ad-hoc solutions, which dramatically decrease the cost of detection. Many techniques have been advocated. They include memory scrubbing [22] and ABFT techniques (see below).

As already pointed out, most papers focus on a selective reliability setting [6, 19, 20, 28]. It essentially means that one can choose to perform any operation in reliable or unreliable mode, assuming the former to be error-free but energy consuming and the latter to be error-prone but preferable from an energetic point of view.

## 2.2 Iterative methods

Iterative methods offer a wide range of ad-hoc approaches. For instance, instead of duplicating the computation, Benson, Schmit and Schreiber suggest coupling a higher-order with a lower-order scheme for PDEs [3]. Their method only detects an error but does not correct it. Self-stabilizing corrections after error detection in the CG method are investigated by Sao and Vuduc [28]. Heroux and Hoemmen [16] design a fault-tolerant GMRES capable of converging despite silent errors. Bronevetsky and de Supinski [7] provide a comparative study of detection costs for iterative methods.

As already mentioned, a nice instantiation of the checkpoint and verification mechanism that we study in this paper is provided by Chen [8], who deals with sparse iterative solvers. For CG, the verification amounts to checking the orthogonality of two vectors and to recomputing and checking the residual (see Section 3.1 for further details).

As already mentioned, our abstract performance model is agnostic of the underlying error-detection technique and takes the cost of verification as an input parameter to the model.

## 2.3 ABFT

The very first idea of algorithm-based fault tolerance for linear algebra kernels is given by Huang and Abraham [21]. They describe an algorithm capable of detecting and correcting a single silent error striking a matrix-matrix multiplication by means of row and column checksums. This germinal

idea is then elaborated by Anfinson and Luk [1], who propose a method to detect and correct up to two errors in a matrix representation using just four column checksums. Despite its theoretical merit, the idea presented in their paper is actually applicable only with relatively small matrices, and is hence out of our scope.

The problem of algorithm-based fault-tolerance for sparse matrices is investigated by Shantharam et al. [29], who suggest a way to detect a single error in an SpMxV at the cost of a few additional dot products. Sloan et al. [30] suggest that this approach can be relaxed using randomization schemes, and propose several checksumming techniques for sparse matrices. These techniques are less effective than the previous ones, not being able to protect the computation from faults striking the memory, but provide an interesting theoretical insight. Surveys of ABFT schemes are provided in [4, 11].

# 3 CG-ABFT

We streamline our discussion on the CG method, however, the techniques that we describe are applicable to any iterative solver that use sparse matrix vector multiplies and vector operations. This list includes many of the non-stationary iterative solvers such as CGNE, BiCG, BiCGstab where sparse matrix transpose vector multiply operations also take place. Furthermore, preconditioned variants of these solvers with an approximate inverse preconditioner (applied as SpMxV) can also be made fault-tolerant with the proposed scheme.

In Section 3.1, we first provide a background on the CG method and overview both Chen's stability tests [8] and ABFT protection schemes. Then we detail ABFT techniques for the SpMxV kernel.

**Algorithm 1** The Conjugate Gradient algorithm for a sparse positive definite matrix.

**Input:** $\mathbf{A} \in \mathbf{R}^{n \times n}, \mathbf{b}, \mathbf{x}_0 \in \mathbf{R}^n, \varepsilon \in \mathbf{R}$
**Output:** $\mathbf{x} \in \mathbf{R}^n \ : \ \|\mathbf{A}\mathbf{x} - \mathbf{b}\| \leq \varepsilon$

1: $\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_0$;
2: $\mathbf{p}_0 \leftarrow \mathbf{r}_0$;
3: $i \leftarrow 0$;
4: **while** $\|\mathbf{r}_i\| > \varepsilon \left(\|\mathbf{A}\| \cdot \|\mathbf{r}_0\| + \|b\|\right)$ **do**
5: $\quad\mathbf{q} \leftarrow \mathbf{A}\mathbf{p}_i$;
6: $\quad\alpha_i \leftarrow \|\mathbf{r}_i\|^2 / \mathbf{p}_i^\intercal \mathbf{q}$;
7: $\quad\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + \alpha\, \mathbf{p}_i$;
8: $\quad\mathbf{r}_{i+1} \leftarrow \mathbf{r}_i - \alpha\, \mathbf{q}$;
9: $\quad\beta \leftarrow \|\mathbf{r}_{i+1}\|^2 / \|\mathbf{r}_i\|^2$;
10: $\quad\mathbf{p}_{i+1} \leftarrow \mathbf{r}_{i+1} + \beta\, \mathbf{p}_i$;
11: $\quad i \leftarrow i + 1$;
12: **return** $\mathbf{x}_i$;

## 3.1 CG and verification mechanisms

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

Chen's stability tests [8] amount to checking the orthogonality of vectors $\mathbf{p}_{i+1}$ and $\mathbf{q}$, at the price of computing $\frac{\mathbf{p}_{i+1}^{\mathsf{T}}\mathbf{q}}{\|\mathbf{p}_{i+1}\|\|\mathbf{q}_i\|}$, 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.

The only modification made to Chen's original approach is that we also save the sparse matrix $\mathbf{A}$ in addition to the current iteration vectors. This is needed when a silent error is detected: if this error comes for a corruption in data memory, we need to recover with a valid copy of the data matrix $\mathbf{A}$. This holds for the three methods under study, Online-Detection, ABFT-Detection and ABFT-Correction, which have exactly the same checkpoint cost.

We now introduce our own protection and verification mechanisms. We use ABFT techniques to protect the SpMxV, its computations (hence the vector $\mathbf{q}$), the matrix $\mathbf{A}$ and the input vector $\mathbf{p}_i$. As ABFT methods for vector operations is as costly as a repeated computation, we use triple modular redundancy (TMR) for simplicity. In other words, we do not protect $\mathbf{p}_i$, $\mathbf{q}$, $\mathbf{r}_i$, and $\mathbf{x}_i$ of the $i$th loop beyond the SpMxV at line 5 with ABFT, but we compute the dots, norms and *axpy* operations in the 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 $\lfloor k/2 \rfloor$. Therefore, we focus on detecting up to two errors and correcting the error if there was only one. That is, we detect up to two errors in the computation $\mathbf{q} \leftarrow \mathbf{A}\mathbf{p}_i$ (two entries in $\mathbf{q}$ are faulty), or in $\mathbf{p}_i$, or in the sparse representation of the matrix $\mathbf{A}$. With TMR, we assume that the errors in the computation are not overly frequent so that two out of three are correct (we assume errors do not strike the vector data here). Our fault-tolerant CG version thus have the following ingredients: ABFT to detect up to two errors in the SpMxV and correct the error, if there was only 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 4.2).

### 3.2 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. [29] propose a cheaper ABFT SpMxV algorithm that guarantees the detection of a single error striking either the computation or the memory representation of the two input operands (matrix and vector). As their results depends on the sparse storage format adopted, throughout the paper we will assume that sparse matrices are stored in the compressed storage format by rows (CSR), that is by means of three distinct arrays, namely *Colid* $\in \mathbf{N}^{\mathrm{nnz}(\mathbf{A})}$, *Val* $\in \mathbf{R}^{\mathrm{nnz}(\mathbf{A})}$ and *Rowindex* $\in \mathbf{N}^{n+1}$ [27, Sec. 3.4].

Shantaram et al. can protect an SpMxV of the form

$$\mathbf{y} \leftarrow \mathbf{A}\mathbf{x},$$

where $\mathbf{A} \in \mathbf{R}^{n \times n}$ and $\mathbf{x}, \mathbf{y} \in \mathbf{R}^n$. To perform error detection, they rely on a column checksum vector $\mathbf{c}$ defined by

$$c_j = \sum_{i=0}^{n} a_{i,j} \tag{1}$$

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. In fact, it

can be shown [29] that in case of no error these three quantities carry the same value, whereas if a single error strikes either the memory or the computation, then 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 method. 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

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

where $\mathbf{w} \in \mathbf{R}^n$ 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 $\langle \cdot, \cdot \rangle : \mathbf{R}^n \times \mathbf{R}^n \to \mathbf{R}$ defined by $\langle \mathbf{u}, \mathbf{v} \rangle \mapsto \mathbf{u}^\intercal \mathbf{v}$. 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 (1), we have chosen $\mathbf{w}$ to be such that $w_i = 1$ for $1 \leq i \leq n$, in order to make the computation easier. Let us see now what happens removing this restriction.

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

$$\langle \mathbf{w}, \mathbf{b}_i \rangle \neq 0, \tag{2}$$

for any $i$, if and only if it lies on $h_i$. As the Lebesgue measure in $\mathbf{R}^n$ of an hyperplane of $\mathbf{R}^n$ 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 $\mathbf{R}^n$. Therefore, the probability that a vector $\mathbf{w}$ randomly picked in $\mathbf{R}^n$ does not satisfy condition (2) 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 $\mathbf{R}^n$ 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 the execution time and the number of rounding errors (see Section 5). 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 (in the context of some other iterative methods than CG). There are many matrices with this property, for example the Laplacian matrices of graphs [32].

In Algorithm 2, we propose an ABFT SpMxV method that uses weighted checksums to perform error detection 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 1.

**Theorem 1** (Correctness of Algorithm 2). *Let $\mathbf{A} \in \mathbf{R}^{n\times n}$ be a square matrix, let $\mathbf{x}, \mathbf{y} \in \mathbf{R}^n$ be the input and output vector respectively, and let $\mathbf{x}' = \mathbf{x}$. Let us assume that the algorithm performs the computation*

$$\widetilde{\mathbf{y}} \leftarrow \widetilde{\mathbf{A}}\widetilde{\mathbf{x}}, \tag{3}$$

*where $\widetilde{\mathbf{A}} \in \mathbf{R}^{n\times n}$ and $\widetilde{\mathbf{x}} \in \mathbf{R}^n$ are the possibly faulty representations of $\mathbf{A}$ and $\mathbf{x}$ respectively, while $\widetilde{\mathbf{y}} \in \mathbf{R}^n$ 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 $\mathbf{c} = [\sum_{i=1}^n a_{i,1} + k, \ldots, \sum_{i=1}^n a_{i,n} + k]$, where $k$ is such that $\sum_{i=1}^n a_{i,j} + k \neq 0$ for $1 \leq j \leq n$,*
2. *an auxiliary checksum $y_{n+1} = k \sum_{i=i}^n \widetilde{x}_i$,*
3. *an auxiliary counter $s_{Rowindex}$ initialized to 0 and updated at runtime by adding the value of the hit element each time the Rowindex array is accessed (line 20 of Algorithm 2),*
4. *an auxiliary checksum $c_{Rowindex} = \sum_{i=1}^n Rowindex_i \in \mathbf{N}$.*

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

*i.* $\mathbf{c}^\mathsf{T}\widetilde{\mathbf{x}} = \sum_{i=1}^{n+1} \widetilde{y}_i$,

*ii.* $\mathbf{c}^\mathsf{T}\mathbf{x}' = \sum_{i=1}^{n+1} \widetilde{y}_i$,

*iii.* $s_{Rowindex} = c_{Rowindex}$.

**Proof**. We will consider three possible cases, namely

a. a faulty arithmetic operation during the computation of $\mathbf{y}$,

b. a bit flip in the sparse representation of $\mathbf{A}$,

c. a bit flip in an element of of $\mathbf{x}$.

**Case a.** Let us assume, without loss of generality, that the error has struck at the $p-$th position of $\mathbf{y}$, that implies $\widetilde{y}_i = y_i$ for $1 \leq i \leq n$ with $i \neq p$ and $\widetilde{y}_p = y_p + \varepsilon$, where $\varepsilon \in \mathbf{R} \setminus \{0\}$ represents the value of the error that has occurred. Summing up the elements of $\widetilde{\mathbf{y}}$ gives

$$\begin{aligned}
\sum_{i=1}^{n+1} \widetilde{y}_i &= \sum_{i=1}^{n+1} y_i + \varepsilon \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n} a_{i,j}\widetilde{x}_j + k\sum_{j=1}^{n} \widetilde{x}_j + \varepsilon \\
&= \sum_{j=1}^{n} \left(\sum_{i=1}^{n} a_{i,j} + k\right) \widetilde{x}_j + \varepsilon \\
&= \sum_{j=1}^{n} c_j \widetilde{x}_j + \varepsilon \\
&= \mathbf{c}^\mathsf{T}\widetilde{\mathbf{x}} + \varepsilon,
\end{aligned}$$

that violates condition (i).

**Case b.** A single error in the $\mathbf{A}$ matrix can strike one of the three vectors that constitute its sparse representation:

- a fault in *Val* that alters the value of an element $a_{i,j}$ implies an error in the computation of $\widetilde{y}_i$, which leads to the violation of the safety condition (i) because of (a),
- a variation in *Colid* can zero out an element in position $a_{i,j}$ shifting its value in position $a_{i,j'}$, leading again to an erroneous computation of $\widetilde{y}_i$,
- a transient fault in *Rowindex* entails an incorrect value of $s_{Rowindex}$ and hence a violation of condition (iii).

**Case c.** Let us assume, without loss of generality, an error in position $p$ of $\mathbf{x}$. Hence we have that $\widetilde{x}_i = x_i$ for $1 \leq i \leq n$ with $i \neq p$ and $\widetilde{x}_p = x_p + \varepsilon$, for some $\varepsilon \in \mathbf{R} \setminus \{0\}$. Noting that $\mathbf{x} = \mathbf{x}'$, the sum of the elements of $\widetilde{\mathbf{y}}$ gives

$$
\begin{aligned}
\sum_{i=1}^{n+1} \widetilde{y}_i &= \sum_{i=1}^{n}\sum_{j=1}^{n} a_{i,j}\widetilde{x}_j + k\sum_{j=1}^{n}\widetilde{x}_j \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n} a_{i,j}x_j + k\sum_{j=1}^{n} x_j + \varepsilon\sum_{i=1}^{n} a_{i,p} + \varepsilon k \\
&= \sum_{j=1}^{n}\left(\sum_{i=1}^{n} a_{i,j} + k\right) x_j + \varepsilon\left(\sum_{i=1}^{n} a_{i,p} + k\right) \\
&= \sum_{j=1}^{n} c_j x_j + \varepsilon\left(\sum_{i=1}^{n} a_{i,p} + k\right) \\
&= \mathbf{c}^{\mathsf{T}}\mathbf{x}' + \varepsilon\left(\sum_{i=1}^{n} a_{i,p} + k\right),
\end{aligned}
$$

that violates (ii) since $\sum_{i=1}^{n} a_{i,p} + k \neq 0$ by the definition of $k$. □

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 talking about the performances of the checksumming techniques. Let us also note that building up the necessary structures requires $\mathcal{O}(k\,\mathrm{nnz}(\mathbf{A}))$ time, and the overhead per SpMxV is $\mathcal{O}(kn)$.

We now discuss error correction, referring to Algorithm 3 which performs double error detection and single error correction. It at least one of the tests at line 23 fails, the algorithm invokes CORRECTERRORS in order to determine whether just one error struck either the computation or the memory and, in case, correct it. Whenever a single error is detected, disregarding its location (i.e. computation or memory vector) it is corrected by means of a succession of various steps. Once the presence of errors is detected, the correction mechanism tries to determine the number of striking errors and, in case of single error, its position. At this point the errors are corrected using the values of the checksums and if need be partial recomputations of the result are performed.

Specifically, we proceed as follows. To detect errors striking *Rowindex* we compute the quantity $d =\mid r_2/r_1 \mid$, 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 $\varepsilon$. If it is so, the algorithm concludes that the $d$-th element of *Rowindex* is faulty, performs the correction $Rowindex_d = Rowindex_d - r_1$ and recomputes $y_d$ and $y_{d-1}$ if the error in $Rowindex_d$ 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* corresponing to the entry at row $d$ and column $f$ of $A$ and corrects it by using the column checksums much like as described for *Rowindex*. 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 $\varepsilon$. 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 $\mathbf{x}^\tau \in \mathbf{R}^{n\times n}$ is such that $x_d^\tau = \tau$ and $x_i^\tau = 0$ otherwise.

Finally, let us note that double errors could be shadowed when using Algorithm 3, although the probability of such an event is negligible. Nevertheless, an implementation that takes such an issue into account is possible, at the cost of an additional checksum. Let us note that using the weight matrix

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ \vdots & \vdots \\ 1 & n \end{bmatrix}$$

is enough to guarantee on the one hand single error correction and, on the other, two errors correction, but not to achieve both correction and detection at the same time. Indeed, let us restrict ourselves to an easy case, without considering errors in $\mathbf{x}$. As usual, we compute the column checksums matrix

$$\mathbf{C} = (\mathbf{W}^\intercal \mathbf{A})^\intercal ,$$

and then compare the two entries of $\mathbf{d} = \mathbf{C}^\intercal \mathbf{x} \in \mathbf{R}^2$ with the weighted sums

$$\widetilde{y}_1^c = \sum_{i=1}^{n} \widetilde{y}_i$$

and

$$\widetilde{y}_2^c = \sum_{i=1}^{n} i\widetilde{y}_i$$

where $\widetilde{\mathbf{y}}$ is the possibly faulty vector computed by the algorithm. It is clear that if no error occurs, the computation verifies the condition $\delta = \widetilde{\mathbf{y}}^c - \mathbf{c} = 0$. Furthermore, if exactly one error occurs, clearly we have that $\delta_1, \delta_2 \neq 0$ and $\frac{\delta_2}{\delta_1} \in \mathbf{N}$, and if two errors strike the vectors protected by the checksum $\mathbf{c}$, the algorithm is able to detect them by verifying that $\delta \neq 0$.

At this point it is natural to ask whether this information is enough to build a working algorithm or some border cases can bias its behaviour. In particular when $\frac{\delta_2}{\delta_1} = p \in \mathbf{N}$ it is not clear how to discern whether there have been one or two errors.

Let $\varepsilon_1, \varepsilon_2 \in \mathbf{R} \setminus \{0\}$ be the value of two errors occurring at position $p_1$ and $p_2$ respectively, and let $\widetilde{\mathbf{y}} \in \mathbf{R}^n$ be such that

$$\widetilde{y}_i = \begin{cases} y_i, & 1 \leq i \leq n, i \neq p_1, p_2 \\ y_i + \varepsilon_1, & i = p_1 \\ y_i + \varepsilon_2, & i = p_2 \end{cases}.$$

Then the following conditions

$$\delta_1 = \varepsilon_1 + \varepsilon_2, \tag{4}$$

$$\delta_2 = p_1\varepsilon_1 + p_2\varepsilon_2, \tag{5}$$

hold. Therefore, it is clear that if $\varepsilon_1$ and $\varepsilon_2$ are such that

$$p\left(\varepsilon_1 + \varepsilon_2\right) = p_1\varepsilon_1 + p_2\varepsilon_2, \tag{6}$$

it is not possible to distinguish these two errors from a single error of value $\varepsilon_1 + \varepsilon_2$ occurring in position $p$. This issue can be solved by introducing a new set of weights and hence a new row of column checksums. Indeed, let us consider a weight matrix $\widehat{\mathbf{W}} \in \mathbf{R}^{n\times 3}$ that includes a quadratic weight vector

$$\widehat{\mathbf{W}} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ \vdots & \vdots & \vdots \\ 1 & n & n^2 \end{bmatrix}.$$

Then let us consider the vector

$$\hat{\delta} = \left(\widehat{\mathbf{W}}^\intercal \mathbf{A}\right)\mathbf{x} - \widehat{\mathbf{W}}^\intercal \widetilde{\mathbf{y}},$$

whose components can be expressed as

$$\delta_1 = \varepsilon_1 + \varepsilon_2 \tag{7}$$

$$\delta_2 = p_1\varepsilon_1 + p_2\varepsilon_2 \tag{8}$$

$$\delta_2 = p_1^2\varepsilon_1 + p_2^2\varepsilon_2. \tag{9}$$

To be confused with a single error in position $p$, $\varepsilon_1$ and $\varepsilon_2$ have to be such that

$$p(\varepsilon_1 + \varepsilon_2) = p_1\varepsilon_1 + p_2\varepsilon_2$$

and

$$p^2(\varepsilon_1 + \varepsilon_2) = p_1^2\varepsilon_1 + p_2^2\varepsilon_2$$

hold simultaneously for some $p \in \mathbf{N}$. In other words, possible values of the errors are the solution of the linear system

$$\begin{pmatrix} (p-p_1) & (p-p_2) \\ (p^2-p_1^2) & (p^2-p_2^2) \end{pmatrix} \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

It can be easily computed that the determinant of the coefficient matrix is

$$(p-p_1)(p-p_2)(p_2-p_1),$$

that always differs from zero, as long as $p$, $p_1$ and $p_s$ differ pairwise. Thus, the matrix is invertible and the solution space of the linear system is the trivial kernel $(\varepsilon_1, \varepsilon_2) = (0, 0)$. Thus using $\widehat{\mathbf{W}}$ as weight matrix guarantees that it is always possible to distinguish single from double errors.

# 4 Performance model

In Section 4.1, we introduce the general performance model. Then in Section 4.2 we instantiate it for the three methods that we are considering, namely Online-Detection, ABFT-Detection and ABFT-Correction.

## 4.1 General approach

In this section, we introduce an abstract performance model to compute the best combination of checkpoints and verifications for iterative methods. We execute $T$ time-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 checkpoint, recovery and verification are error-free operations. Let $T_{cp}$, $T_{rec}$ and $T_{verif}$ be the respective cost of these operations. Finally, assume an Exponential distribution of errors and let $q$ be the probability of successful execution for each chunk: $q = e^{-\lambda T}$, where $\lambda$ is the fault rate.

The goal of this section is to compute the expected time $\mathbb{E}(s, T)$ needed to execute a frame composed of $s$ chunks of size $T$. We derive the best value of $s$ as a function of $T$ and of the resilience parameters $T_{cp}$, $T_{rec}$, $T_{verif}$, and $q$, the success probability of a chunk. Each frame is preceded by a checkpoint, except maybe the first one (for which we recover by reading initial data again). Following [5], we derive the following recursive equation to compute the expected completion time of a single frame:

$$\mathbb{E}(s, T) = q^s(s(T + T_{verif})) + T_{cp}) + (1 - q^s)(\mathbb{E}(T_{lost}) + T_{rec} + \mathbb{E}(s, T)). \tag{10}$$

Indeed, the execution is successful if all chunks are successful, which happens with probability $q^s$, and in this case the execution time simply is the sum of the execution times of each chunk plus the final checkpoint. Otherwise, with probability $1 - q^s$, we have an error, which we detect after some

time $T_{lost}$, and that forces us to recover and restart the frame anew, hence in time $\mathbb{E}(s,T)$. The difficult part is to compute $\mathbb{E}(T_{lost})$.

For $1 \le i \le s$, let $f_i$ be the following conditional probability:

$$f_i = \mathbb{P}(\text{error strikes at chunk } i | \text{there is an error in the frame})$$

Given the success probability $q$ of a chunk, we obtain that

$$f_i = \frac{q^{i-1}(1-q)}{1-q^s} .$$

Indeed, the first $i-1$ chunks were successful (probability $q^{i-1}$), the $i$th one had an error (probability $1-q$), and we condition by the probability of an error within the frame, namely $1-q^s$. With probability $f_i$, we detect the error at the end of the $i$th chunk, and we have lost the time spent executing the $i$ first chunks We derive that

$$\mathbb{E}(T_{lost}) = \sum_{i=1}^{s} f_i \left(i(T+T_{verif})\right) .$$

We have $\sum_{i=1}^{s} f_i = \frac{(1-q)h(q)}{1-q^s}$ where $h(q) = 1+2q+3q^2+\cdots+sq^{s-1}$. If $m(q) = q+q^2+\cdots+q^s = \frac{1-q^{s+1}}{1-q} - 1$, we get by differentiation that $m'(q) = h(q)$, hence $h(q) = \frac{-(s+1)q^s}{1-q} + \frac{1-q^{s+1}}{(1-q)^2}$ and finally

$$\mathbb{E}(T_{lost}) = (T+T_{verif})\frac{sq^{s+1}-(s+1)q^s+1}{(1-q^s)(1-q)} .$$

Plugging the expression of $\mathbb{E}(T_{lost})$ back into Equation (10), we obtain

$$\mathbb{E}(s,T) = s(T+T_{verif})) + T_{cp} + (q^{-s}-1)T_{rec} + T\frac{sq^{s+1}-(s+1)q^s+1}{q^s(1-q)} , \tag{11}$$

which simplifies into

$$\mathbb{E}(s,T) = T_{cp} + (q^{-s}-1)T_{rec} + (T+T_{verif})\frac{1-q^s}{q^s(1-q)} . \tag{12}$$

We have to determine the value of $s$ that minimizes the overhead of a frame:

$$s = \underset{s\ge 1}{\operatorname{argmin}} \left(\frac{\mathbb{E}(s,T)}{sT}\right) . \tag{13}$$

The minimization is complicated, because $T$, the size of a chunk, is still unknown, and should be conducted numerically. Note that a dynamic programming algorithm to compute the optimal value of $T$ and $s$ is available in [2].

## 4.2 Instantiation to CG

For each of the three methods we have named in the introduction, that are Online-Detection, ABFT-Detection and ABFT-Correction, we instantiate the previous model and discuss how to solve Equation (13).

#### 4.2.1 Online-Detection:

For Chen's method [8], 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 $T_{verif}$ is the cost of the operations described in Section 3.1.
- 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 $\lambda_a$ be the rate of arithmetic errors, and $\lambda_m$ 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 $\lambda_{word}$. 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 $q = e^{-\lambda T}$.

Plugging these values in Equation (13) gives an optimisation formula very similar to that of [8, Section 5.2]. The only difference is that we have assumed that the verification is error-free, which is needed for the correctness of the approach.

#### 4.2.2 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, $T_{verif}$ 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 $q = e^{-\lambda T}$.

#### 4.2.3 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 $T_{verif}$ 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 $\lambda$, where $\lambda = \lambda_a + \lambda_m$ as for Online-Detection and ABFT-Detection. The probability of exactly $k$ errors in time $T$ is $\frac{(\lambda T)^k}{k!} e^{-\lambda T}$ [25], 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 Experiments

### 5.1 Setup

Intuitively, there are two different sources of advantages in combining ABFT and checkpointing. On the one hand, the error detection capability lets us perform a cheap validation of the partial result of each CG step, recovering as soon as an error strikes. On the other hand, being able to correct single errors makes each step more resilient and thus lets us increase the expected number

of consecutive *valid* iteration. Here we consider *valid* a step which either is non-faulty, or suffers from a single error that is corrected via ABFT.

For our experiments, we use a bunch of positive definite test matrices from the Harwell-Boeing Sparse Matrix Collection [10, 12], with size between 17456 and 74752 and density lower than $10^{-2}$.

At each step, faults are injected during vector and matrix-vector operations but, since we are assuming selective reliability, all of the checksums and checksum operations are considered non-faulty. Faults are modeled as bit flips occurring independently at each step, under an exponential distribution of parameter $\lambda$, as detailed in Section 4.2. These bit flips can strike either the matrix, i.e., the elements of *Val*, *Colid* and *Rowindex*, or any entry of the CG vectors $\mathbf{r}_i, \mathbf{q}, \mathbf{p}_i$ or $\mathbf{x}_i$. We choose not to take errors striking the computation into account because they are just a special case of the kind of issues we are considering. Moreover, to simplify the injection mechanism, $T_{iter}$ is normalized to be one, meaning that each memory location or operation is given the chance to fail just once per iteration [28]. Finally, to get data that are homogeneous among the test matrices, the fault rate $\lambda$ is chosen to be inversely proportional to $M$ with a proportionality constant $\alpha \in (0, 1)$. It follows that the expected number of CG steps between two distinct fault occurrences does not depend either on the size or on the sparsity ratio of the matrix.

We compare the performance of three algorithms, namely Online-Detection, ABFT-Detection, that performs single detection and rolls back as soon as an error is detected, and ABFT-Correction, which is capable of correcting single errors during a given step and rolls back only if two errors strike a single operation.

Implementing the equality tests in Algorithm 2 and Algorithm 3 poses a challenge. The comparison $\mathbf{r} = \mathbf{c}_r - \mathbf{s}_r$ is between two integers, and can be correctly evaluated by any programming language using the equality sign, even in case of overflow. However, the other two, having floating point operands, are problematic. Since the floating point operations are not associative and the distributive property does not hold, we need a tolerance parameter that takes into account the rounding operations that are performed by each floating point operation. Here, we give an upper bound on the difference between the two floating point checksums, using the standard model [17, Section 2.2]. Our goal is to guarantee that only real errors are detected, while mere inaccuracies due to floating point operations tolerate, since they would be generated by non-faulty executions as well.

**Theorem 2** (Accuracy of the floating point weighted checksums)**.** *Let* $\mathbf{A} \in \mathbf{R}^{n \times n}$*,* $\mathbf{x} \in \mathbf{R}^n$*,* $\mathbf{c} \in \mathbf{R}^n$*. Then, if all of the sums involved into the matrix operations are performed using some flavour of recursive summation [17, Chapter 4], it holds that*

$$\left| fl\left(\left(\mathbf{c}^\intercal \mathbf{A}\right) \mathbf{x}\right) - fl\left(\mathbf{c}^\intercal \left(\mathbf{A}\mathbf{x}\right)\right) \right| \leq 2\, \gamma_{2n} \; \left| \mathbf{c}^\intercal \right| \left| \mathbf{A} \right| \left| \mathbf{x} \right| . \tag{14}$$

**Proof**. To prove the result in (14) we bound the left hand side of the inequality with the sum of two classical floating point bounds in the form $| x - fl(x) |$ and then bound them independently. For the first step, let us note that the absolute value is a norm on the real numbers and hence

$$\begin{aligned} \left| fl\left(\left(\mathbf{c}^\intercal \mathbf{A}\right) \mathbf{x}\right) - fl\left(\mathbf{c}^\intercal \left(\mathbf{A}\mathbf{x}\right)\right) \right| &= \left| fl\left(\left(\mathbf{c}^\intercal \mathbf{A}\right) \mathbf{x}\right) - fl\left(\mathbf{c}^\intercal \left(\mathbf{A}\mathbf{x}\right)\right) + \mathbf{c}^\intercal \mathbf{A}\mathbf{x} - \mathbf{c}^\intercal \mathbf{A}\mathbf{x} \right| \\ &\leq \left| fl\left(\left(\mathbf{c}^\intercal \mathbf{A}\right) \mathbf{x}\right) - \left(\mathbf{c}^\intercal \mathbf{A}\right) \mathbf{x} \right| \\ &\quad + \left| fl\left(\mathbf{c}^\intercal \left(\mathbf{A}\mathbf{x}\right)\right) - \mathbf{c}^\intercal \left(\mathbf{A}\mathbf{x}\right) \right| . \end{aligned} \tag{15}$$

To bound the absolute value of the dot product of the checksum of $\mathbf{A}$ and $\mathbf{x}$, we proceed by levels. First of all, we can upper-bound the value of a single entry of $\mathbf{c}^\intercal \mathbf{A}$ with

$$fl\,(\mathbf{c}^\intercal \mathbf{A})_j \leq a_{1,j}(1+\delta_1)^{n-1} + a_{2,j}(1+\delta_2)^{n-1} + a_{3,j}(1+\delta_2)^{n-2} + \cdots + a_{n,j}(1+\delta_n), \qquad (16)$$

where $\mid \delta_i \mid \leq \mathbf{u}$ for $1 \leq i \leq n$, where $\mathbf{u}$ is the machine epsilon, otherwise called unit roundoff. As it is standard in classical error analysis, we drop the subscript and simplify the expression by writing $1+\delta_i = 1 \pm \delta$. Now, multiplying the $j$-th entry of the matrix checksum with the corresponding $\mathbf{x}$ entry, gives that

$$\begin{aligned} fl\left((\mathbf{c}^\intercal \mathbf{A})_j\, x_j\right) &\leq \left((a_{1,j}+a_{2,j})(1\pm\delta)^{n-1} + a_{3,j}(1\pm\delta)^{n-2} + \cdots + a_{n,j}(1\pm\delta)\right) x_j(1\pm\delta) \\ &= (a_{1,j}+a_{2,j})x_j(1\pm\delta)^{n} + a_{3,j}x_j(1\pm\delta)^{n-1} + \cdots + a_{n,j}x_j(1\pm\delta)^2. \end{aligned}$$

Finally, we can upper-bound the whole dot product with a sum

$$\begin{aligned} fl\,((\mathbf{c}^\intercal \mathbf{A})\,\mathbf{x}) \leq & \left((a_{1,1}+a_{2,1})x_1\,(1\pm\delta)^{n} + a_{3,1}x_1\,(1\pm\delta)^{n-1} + \cdots + a_{n,1}x_1\,(1\pm\delta)^2\right)(1\pm\delta)^{n-1} \\ & + \left((a_{1,2}+a_{2,2})x_2\,(1\pm\delta)^{n} + a_{3,2}x_2\,(1\pm\delta)^{n-1} + \cdots + a_{n,2}x_2\,(1\pm\delta)^2\right)(1\pm\delta)^{n-1} \\ & + \left((a_{1,3}+a_{2,3})x_3\,(1\pm\delta)^{n} + a_{3,3}x_3\,(1\pm\delta)^{n-1} + \cdots + a_{n,3}x_3\,(1\pm\delta)^2\right)(1\pm\delta)^{n-2} \\ & + \qquad \cdots \\ & + \left((a_{1,n}+a_{2,n})x_n\,(1\pm\delta)^{n} + a_{3,n}x_n\,(1\pm\delta)^{n-1} + \cdots + a_{n,n}x_n\,(1\pm\delta)^2\right)(1\pm\delta)\,, \end{aligned}$$

that distributing becomes

$$\begin{aligned} fl\,((\mathbf{c}^\intercal \mathbf{A})\,\mathbf{x}) \leq\; & (a_{1,1}+a_{2,1})x_1\,(1\pm\delta)^{2n-1} + a_{3,1}x_1\,(1\pm\delta)^{2n-2} + \cdots + a_{n,1}x_1\,(1\pm\delta)^{n+1} \\ + \; & (a_{1,2}+a_{2,2})x_2\,(1\pm\delta)^{2n-1} + a_{3,2}x_2\,(1\pm\delta)^{2n-2} + \cdots + a_{n,2}x_2\,(1\pm\delta)^{n+1} \\ + \; & (a_{1,3}+a_{2,3})x_3\,(1\pm\delta)^{2n-2} + a_{3,3}x_3\,(1\pm\delta)^{2n-3} + \cdots + a_{n,3}x_3\,(1\pm\delta)^{n} \\ + \; & \qquad \cdots \\ + \; & (a_{1,n}+a_{2,n})x_n\,(1\pm\delta)^{n+1} + a_{3,n}x_n\,(1\pm\delta)^{n} + \cdots + a_{n,n}x_n\,(1\pm\delta)^{3}\,. \end{aligned}$$

This long expression can be simplified [17, Lemma 3.1] to get

$$\begin{aligned} fl\,((\mathbf{c}^\intercal \mathbf{A})\,\mathbf{x}) \leq\; & (a_{1,1}+a_{2,1})x_1\left(1+\theta_{2n-1}^{(1,1)}\right) + a_{3,1}x_1\left(1+\theta_{2n-2}^{(3,1)}\right) + \cdots + a_{n,1}x_1\left(1+\theta_{n+1}^{(n,1)}\right) \\ + \; & (a_{1,2}+a_{2,2})x_2\left(1+\theta_{2n-1}^{(1,2)}\right) + a_{3,2}x_2\left(1+\theta_{2n-2}^{(3,2)}\right) + \cdots + a_{n,2}x_2\left(1+\theta_{n+1}^{(n,2)}\right) \\ + \; & (a_{1,3}+a_{2,3})x_3\left(1+\theta_{2n-2}^{(1,3)}\right) + a_{3,3}x_3\left(1+\theta_{2n-3}^{(3,3)}\right) + \cdots + a_{n,3}x_3\left(1+\theta_{n}^{(n,3)}\right) \\ + \; & \qquad \cdots \\ + \; & (a_{1,n}+a_{2,n})x_n\left(1+\theta_{n+1}^{(1,4)}\right) + a_{3,n}x_n\left(1+\theta_{n}^{(3,4)}\right) + \cdots + a_{n,n}x_n\left(1+\theta_{3}^{(n,4)}\right), \end{aligned}$$

where $\mid \theta_n^{i,j} \mid \leq \gamma_n$, for $1 \leq i,j \leq n$. This last inequality can be interpreted as a backward error result, i.e.,

$$fl\,((\mathbf{c}^\intercal \mathbf{A})\,\mathbf{x}) = \mathbf{c}^\intercal\,(\mathbf{A}+\Delta\mathbf{A})\,\mathbf{x}, \qquad \mid \Delta\mathbf{A} \mid \leq \gamma_{2n-1} \mid \mathbf{A} \mid,$$

where both the matrix absolute value and the inequality between matrices hold componentwise. To get a forward error bound, we proceed as follows:

$$\begin{aligned} \mid fl\,((\mathbf{c}^\mathsf{T}\mathbf{A})\,\mathbf{x}) - (\mathbf{c}^\mathsf{T}\mathbf{A})\,\mathbf{x} \mid &= \mid \mathbf{c}^\mathsf{T}\,(\mathbf{A}+\Delta\mathbf{A})\,\mathbf{x} - (\mathbf{c}^\mathsf{T}\mathbf{A})\,\mathbf{x} \mid \\ &\leq \mid \mathbf{c}^\mathsf{T} \mid \mid \Delta\mathbf{A} \mid \mid \mathbf{x} \mid \\ &\leq \gamma_{2n-1}\,\mathbf{c}^\mathsf{T} \mid \mathbf{A} \mid \mid \mathbf{x} \mid . \end{aligned} \tag{17}$$

Along the same lines, it can also be shown that

$$\mid fl\,(\mathbf{c}^\mathsf{T}\,(\mathbf{A}\mathbf{x})) - \mathbf{c}^\mathsf{T}\,(\mathbf{A}\mathbf{x}) \mid \leq \gamma_{2n-1}\,\mathbf{c}^\mathsf{T} \mid \mathbf{A} \mid \mid \mathbf{x} \mid, \tag{18}$$

and thus substituting (17) and (18) into (15) concludes the proof. □

Let us note that if all of the entries of $\mathbf{c}$ are positive, as it is often the case in our setting, the absolute value of $\mathbf{c}$ in (14) can be safely replaced with $\mathbf{c}$ itself. It is also clear that these bounds are not computable, since $\mathbf{c}^\mathsf{T} \mid \mathbf{A} \mid \mid \mathbf{x} \mid$ is not, in general, a floating point number. This problem can be alleviated by overestimating the bound by means of matrix and vector norms.

Recalling that it can be shown [18, Section B.7] that

$$\|\mathbf{A}\|_1 = \max_{1\leq j\leq n} \sum_{i=1}^{n} \mid a_{i,j} \mid , \tag{19}$$

it is possible to upper bound the right hand side in (14) so to get that

$$\mid fl\,((\mathbf{c}^\mathsf{T}\mathbf{A})\,\mathbf{x}) - fl\,(\mathbf{c}^\mathsf{T}\,(\mathbf{A}\mathbf{x})) \mid \leq 2\,\gamma_{2n}\,n\;\|\mathbf{c}^\mathsf{T}\|_\infty\,\|\mathbf{A}\|_1\,\|\mathbf{x}\|_\infty . \tag{20}$$

Though the right hand side of (20) is not exactly computable in floating point arithmetic, it requires an amount of operations dramatically smaller than the one in (14), just a few sums for the norm of $\mathbf{A}$. As this norm is usually computed using the identity in (19), any kind of summation yields a relative error of at most $n'\mathbf{u}$ [17, Section 4.6], where $n'$ is the maximum number of nonzeros in a column of $\mathbf{A}$. Since we are dealing with sparse matrices, we expect $n'$ to be very small, and hence the computation of the norm to be accurate. Moreover, the right hand side in (20) does not depend on $\mathbf{x}$, it can be computed just once for a given matrix and weight vector.

Clearly, using (20) as tolerance parameter guarantees no false positive (a computation without any error that is considered as faulty), but allows false negatives (an iteration during which an error occurs without being detected) when the perturbations of the result are small. Nonetheless, this solution works almost perfectly in practice, meaning that though the convergence rate can be slowed down, the algorithms still converges towards the "correct" answer. Though such an outcome could be surprising at first, Elliott et al. [14, 31] showed that bit flips that strike the less significant digits of the floating point representation of vector elements during a dot product create small perturbations of the results, and that the magnitude of this perturbation gets smaller as the size of the vectors increases. Hence, we expect errors that are not detected by our tolerance threshold to be too small to impact the solution of the linear solver.

### 5.2 Simulations

To validate the model, we perform the simulation whose results are illustrated in Table 1. For each matrix, we set $\lambda = \frac{1}{16\,M}$ and consider the average execution time of 50 repetitions of both ABFT-Detection (columns 5-8) and ABFT-Correction (columns 6-9). In the table we record the

| id | n | density | $\widetilde{s_1}$ | $E_t(\widetilde{s_1})$ | $s_1^*$ | $E_t(s_1^*)$ | $l_1$ | $\widetilde{s_2}$ | $E_t(\widetilde{s_2})$ | $s_2^*$ | $E_t(s_2^*)$ | $l_2$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 341 | 23052 | 2.15e-03 | 18 | 8.52 | 17 | 8.50 | **6.19** | 14 | 5.68 | 12 | 5.60 | **0.19** |
| 752 | 74752 | 1.07e-04 | 15 | 6.52 | 10 | 5.61 | **16.21** | 13 | 5.72 | 10 | 5.68 | **0.73** |
| 924 | 60000 | 2.11e-04 | 10 | 8.92 | 7 | 8.30 | **7.44** | 13 | 4.31 | 14 | 4.17 | **3.36** |
| 1288 | 30401 | 5.10e-04 | 15 | 6.81 | 12 | 6.31 | **7.98** | 16 | 6.95 | 13 | 6.68 | **4.02** |
| 1289 | 36441 | 4.26e-04 | 16 | 8.20 | 13 | 7.42 | **10.49** | 16 | 7.69 | 13 | 7.58 | **1.50** |
| 1311 | 48962 | 2.14e-04 | 16 | 8.75 | 13 | 7.70 | **13.66** | 16 | 7.52 | 16 | 7.52 | **0.00** |
| 1312 | 40000 | 1.24e-04 | 14 | 4.86 | 11 | 4.18 | **16.23** | 15 | 3.96 | 16 | 3.17 | **25.01** |
| 1848 | 65025 | 2.44e-04 | 17 | 9.56 | 15 | 9.40 | **2.09** | 16 | 12.78 | 14 | 9.31 | **37.22** |
| 2213 | 20000 | 1.39e-03 | 17 | 7.10 | 15 | 6.71 | **5.92** | 16 | 6.60 | 25 | 5.42 | **21.77** |

Table 1: Experimental validation of the model. Here $\widetilde{s_i}$ and $s_i^*$ represent the best checkpointing interval according to our model and to our simulations respectively, whereas $E_t(\widetilde{s_i})$ and $E_t(s_i^*)$ stand for the execution time of the algorithm using these checkpointing intervals.

checkpointing interval $s_i^*$, that achieves the shortest execution time $E_t(s_1^*)$, and the checkpointing interval $\widetilde{s_i}$ that is the best stepsize according to our method, along with its execution time $E_t(\widetilde{s_i})$. Finally, we evaluate the performances of our guess by means of the quantity

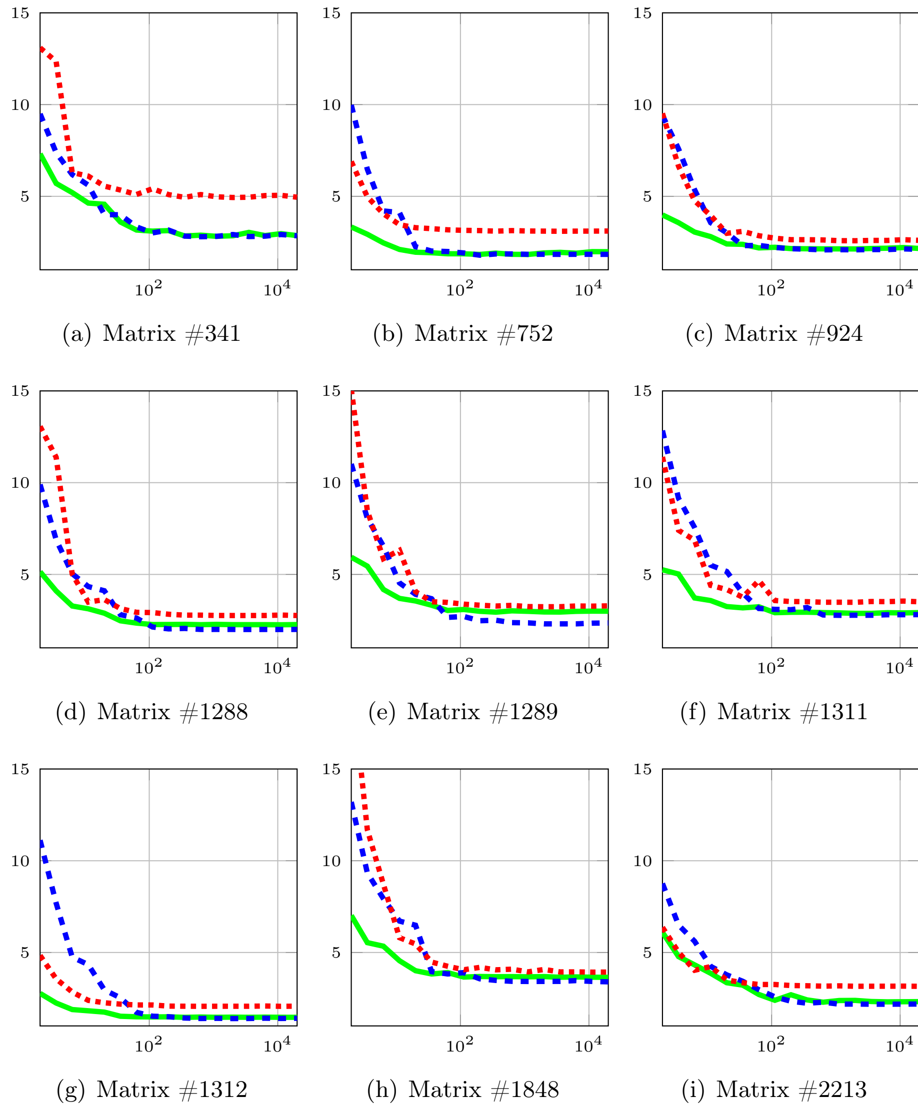
$$l_i = \frac{E_t(\widetilde{s_i}) - E_t(s_i^*)}{E_t(s_i^*)} \cdot 100,$$

that expresses the loss, in terms of execution time, of executing with the checkpointing interval given by our model with respect to the best possible choice.

From the table, it is clear that the values of $\widetilde{s_i}$ and $s_i^*$ are close, even though the time loss is something considerably high, reaching peaks above 15% for $l_1$ and just below 40% for $l_2$. This seemingly poor result depends just on the small number of repetitions we are considering, that leads to the presence of outliers, lucky runs in which a small number of errors occur and the computation is carried on in a much quicker way. Similar results hold for smaller values of $\lambda$.

Furthermore, we compare the execution time of the three algorithms, trying to empirically asses how much their relative performances depend on the fault rate. The results on our test matrices are depicted in Figure 1, where the y-axis represents the execution time (in seconds) and the x-axis the normalized mean time between failure, i.e., the reciprocal of $\alpha$. In other words, the larger $x = \frac{1}{\alpha}$, the smaller the corresponding value of $\lambda = \frac{\alpha}{M}$, hence the smaller the expected number of errors. For each value of $\lambda$, we draw the average execution time of 50 runs of the three algorithms, using the best checkpointing interval predicted by Section 4.1 for ABFT-Detection and ABFT-Correction, and by [8, Eq. 10] for Online-Detection. By the point of view of the execution time, Chen's method is comparable with ours for middle to high fault rates, since it clearly outperforms ABFT-Detection in five out of nine cases, though being slightly faster than ABFT-Correction just for the last test matrix. For lower fault rates, however, Online-Detection seems to be the slowest one: this is probably due to the fact that its detection/correction mechanism implies a greater overhead compared to ABFT mechanisms. Moreover, the algorithm that relies on both ABFT correction and checkpointing becomes slightly slower than the detection/recovery mechanism for very small values of $\lambda$.

Intuitively, this behavior is not surprising. When $\lambda$ is large, many errors occur but, since $\alpha$ is between zero and one, we always have, in expectation, less than one error per iteration. Thus




(a) Matrix #341 (b) Matrix #752 (c) Matrix #924

(d) Matrix #1288 (e) Matrix #1289 (f) Matrix #1311

(g) Matrix #1312 (h) Matrix #1848 (i) Matrix #2213

Figure 1: Execution time (in seconds along the $y$ axis) of Online-Detection (dotted), ABFT-Detection (dashed) and ABFT-Correction (solid line) with respect to the normalized MTBF (along the $x$-axis). The matrix number is in the subcaption.

ABFT-Correction requires fewer checkpoints than ABFT-Detection and almost no rollback, and this compensates for the slightly longer execution time of a single step. When the fault rate is very low, instead, the algorithms perform almost the same number of iterations, but ABFT-Correction takes slightly longer due to the additional dot products at each step.

Altogether, the results show 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

We consider the problem of silent errors in iterative linear systems solvers. At first, we focus our attention on ABFT methods for SpMxV, developing algorithms able to detect and correct errors in both memory and computation using various checksumming techniques. Then, we combine ABFT with replication, in order to develop a resilient CG kernel that can protect axpy's and dot products as well. We also discuss how to take numerical issues into account when dealing with actual implementations. These methods are a worthy choice for a selective reliability model, since most of the operations can be performed in unreliable mode, whereas just checksum computations need to be performed reliably.

In addition, we examine checkpointing techniques as a tool to improve the resilience of our ABFT CG and develop a model to trade-off the checkpointing interval so to achieve the shortest execution time in expectation. We implement two of the possible combinations, namely an algorithm that relies on roll back as soon as an error is detected, and one that is able to correct a single error and recovers from a checkpoint just when two of them strike. We validate the model by means of simulations and finally compare our algorithms with Chen's approach, empirically showing that ABFT overhead is usually smaller than Chen's verification cost.

We expect the combined approach we propose to be useful even when dealing with the preconditioned CG Algorithm [27], a modified version of Algorithm 1 that requires, at each step, an additional matrix-vector operation. In this case, diagonal, approximate inverse, and triangular preconditioners seem to be particularly attracting, as it should be possible to treat them by adapting the techniques described in these pages (Shantaram et al. [29] addressed the triangular case). However, not having a practical and general decoding algorithm for multiple error correction is a major obstacle. We are therefore convinced that finding such an algorithm would widen the applicability of our work.

**Algorithm 2** Shifting checksum algorithm.

**Input:** $\mathbf{A} \in \mathbf{R}^{n \times n}$ $\big($ as *Val* $\in \mathbf{R}^{\mathrm{nnz}(\mathbf{A})}$, *Colid* $\in \mathbf{N}^{\mathrm{nnz}(\mathbf{A})}$, *Rowindex* $\in \mathbf{R}^{n}\big)$, $\mathbf{x} \in \mathbf{R}^n$
**Output:** $\mathbf{y} \in \mathbf{R}^n$ such that $\mathbf{y} = \mathbf{A}\mathbf{x}$ or the detection of a single error

1: $\mathbf{x}' \leftarrow \mathbf{x}$;
2: $[\underline{\mathbf{w}}, \mathbf{c}, k, c_{Rowindex}] = $ computeChecksum(*Val*, *Colid*, *Rowindex*);
3: **return** SpMxV(*Val*, *Colid*, *Rowindex*, $\mathbf{x}$, $\mathbf{x}'$, $\underline{\mathbf{w}}$, $\mathbf{c}$, $k$, $c_{Rowindex}$);

4: **function** computeChecksum(*Val*, *Colid*, *Rowindex*)
5: Generate $\underline{\mathbf{w}} \in \mathbf{R}^{n+1}$;
6: $\mathbf{w} \leftarrow \underline{\mathbf{w}}_{1:n}$;
7: $\mathbf{c} \leftarrow \mathbf{w}^\intercal \mathbf{A}$;
8: **if** $\min(|\ \mathbf{c}\ |) = 0$ **then**;
9: Find $k$ such that $\sum_{i=1}^{n} a_{i,j} + k \neq 0$ for $1 \leq j \leq n$;
10: $\mathbf{c} \leftarrow \mathbf{c} + k\ \mathbf{w}$;
11: $c_{Rowindex} \leftarrow \underline{\mathbf{w}}^\intercal Rowindex$;
12: **return** $\underline{\mathbf{w}}, \mathbf{c}, k, c_{Rowindex}$;

13: **function** SpMxV(*Val*, *Colid*, *Rowindex*, $\mathbf{x}$, $\mathbf{x}'$, $\underline{\mathbf{w}}$, $\mathbf{c}$, $k$, $c_{Rowindex}$)
14: $\mathbf{w} \leftarrow \underline{\mathbf{w}}_{1:n}$;
15: $s_{Rowindex} \leftarrow 0$;
16: **for** $i \leftarrow 1$ to $n$ **do**
17: $y_i \leftarrow 0$;
18: $s_{Rowindex} \leftarrow s_{Rowindex} + Rowindex_i$;
19: **for** $j \leftarrow Rowindex_i$ to $Rowindex_{i+1} - 1$ **do**
20: $ind \leftarrow Colid_j$;
21: $y_i \leftarrow y_i + Val_j \cdot x_{ind}$;
22: $y_{n+1} \leftarrow k\ \mathbf{w}^\intercal \mathbf{x}'$;
23: $c_y \leftarrow \underline{\mathbf{w}}^\intercal \mathbf{y}$;
24: **if** $\mathbf{c}^\intercal \mathbf{x} = c_y \wedge \mathbf{c}^\intercal \mathbf{x}' = c_y \wedge c_{Rowindex} = s_{Rowindex}$ **then**
25: **return** $\mathbf{y}_{1:n}$;
26: **else**
27: **error** ("Soft fault detected");

**Algorithm 3** ABFT-protected SpMxV, detection of 2 errors, correction of 1 error

**Input:** $\mathbf{A} \in \mathbf{R}^{n\times n}$ $\big($ as $\mathit{Val} \in \mathbf{R}^{\mathrm{nnz}(\mathbf{A})}$, $\mathit{Colid} \in \mathbf{N}^{\mathrm{nnz}(\mathbf{A})}$, $\mathit{Rowindex} \in \mathbf{R}^{n}\big)$, $\mathbf{x} \in \mathbf{R}^n$
**Output:** $\mathbf{y} = \mathbf{Ax}$, correction of single error or detection of double error

1: global $\mathbf{W}^\intercal \leftarrow \left[\begin{smallmatrix}1 & 1 & \cdots & 1\\ 1 & 2 & \cdots & n\end{smallmatrix}\right] \in \mathbf{R}^{2\times n}$;
2: global $\underline{\mathbf{W}}^\intercal \leftarrow \left[\mathbf{W}^\intercal \begin{smallmatrix}1\\ n+1\end{smallmatrix}\right] \in \mathbf{R}^{2\times n+1}$;
3: $\mathbf{x}' \leftarrow \mathbf{x}$;
4: $[\mathbf{C}, \mathbf{M}, \mathbf{c}_r, \mathbf{c}_x] =$ computeChecksums($\mathit{Val}$, $\mathit{Colid}$, $\mathit{Rowindex}$);
5: **return** SpMxV($\mathit{Val}$, $\mathit{Colid}$, $\mathit{Rowindex}$, $\mathbf{x}$, $\mathbf{x}'$, $\mathbf{M}$, $\mathbf{c}_r$, $\mathbf{c}_x$);

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

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

## References


[1] C. Anfinson and F. Luk. A Linear Algebraic Model of Algorithm-Based Fault Tolerance. *IEEE Trans. Computers*, 37(12):1599–1604, 1988.

[2] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. Assessing general-purpose algorithms to cope with fail-stop and silent errors. In *Workshop on Performance Modeling, Benchmarking and Simulation ( PMBS)*, 2014. Extended version available as INRIA Research Report RR-8599.

[3] A. R. Benson, S. Schmit, and R. Schreiber. Silent error detection in numerical time-stepping schemes. *CoRR*, abs/1312.2674, 2013.

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

[5] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. Checkpointing strategies for parallel jobs. In *SC'2011*, pages 1–11. IEEE, 2011.

[6] P. G. Bridges, K. B. Ferreira, M. A. Heroux, and M. Hoemmen. Fault-tolerant linear solvers via selective reliability. preprint, 2012.

[7] G. Bronevetsky and B. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proc. 22nd Int. Conf. on Supercomputing*, ICS '08, pages 155–164. ACM, 2008.

[8] Z. Chen. Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods. In *Proc. 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 167–176. ACM, 2013.

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

[10] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011.

[11] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. In *PPoPP*, pages 225–234. ACM, 2012.

[12] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse Matrix Test Problems. *ACM Trans. Math. Softw.*, 15(1):1–14, 1989.

[13] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. Combining partial redundancy and checkpointing for HPC. In *Proc. ICDCS '12*. IEEE Computer Society, 2012.

[14] J. Elliott, F. Mueller, M. Stoyanov, and C. Webster. Quantifying the impact of single bit flips on floating point arithmetic. preprint, 2013.

[15] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proc. of the ACM/IEEE SC Int. Conf.*, SC '12. IEEE Computer Society Press, 2012.

[16] M. Heroux and M. Hoemmen. Fault-tolerant iterative methods via selective reliability. Research report SAND2011-3915 C, Sandia National Laboratories, 2011.

[17] N. J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM Press, 2nd edition, 2002.

[18] N. J. Higham. *Functions of Matrices: Theory and Computation.* SIAM Press, 2008.

[19] M. Hoemmen and M. A. Heroux. Fault-tolerant iterative methods via selective reliability. Technical report, Sandia Corporation, 2011.

[20] M. Hoemmen and M. A. Heroux. Fault-Tolerant Iterative Methods via Selective Reliability. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE Computer Society*, volume 3, page 9, 2011.

[21] K.-H. Huang and J. A. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *Computers, IEEE Transactions on*, C–33(6):518–528, 1984.

[22] 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.

[23] G. Lu, Z. Zheng, and A. A. Chien. When is multi-version checkpointing needed. In *3rd Workshop for Fault-tolerance at Extreme Scale (FTXS).* ACM Press, 2013.

[24] 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.

[25] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, 2005.

[26] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proc. of the ACM/IEEE SC Conf.*, pages 1–11, 2010.

[27] Y. Saad. *Iterative Methods for Sparse Linear Systems.* SIAM Press, 2nd edition, 2003.

[28] P. Sao and R. Vuduc. Self-stabilizing iterative solvers. In *Proc. ScalA '13.* ACM, 2013.

[29] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *Proc. ICS '12.* ACM, 2012.

[30] J. Sloan, R. Kumar, and G. Bronevetsky. Algorithmic Approaches to Low Overhead Fault Detection for Sparse Linear Algebra. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12, 2012.

[31] M. Stoyanov and C. Webster. Quantifying the impact of single bit flips on floating point arithmetic. Technical report, Oak Ridge National Laboratory, 2013.

[32] E. W. Weisstein. Laplacian matrix. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/LaplacianMatrix.html`, last accessed October 2014.

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

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

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

ISSN 0249-6399