A backward/forward recovery approach for the preconditioned conjugate gradient method

Massimiliano Fasi, Julien Langou, Yves Robert, Bora Uçar

A Backward/Forward Recovery Approach for the Preconditioned Conjugate Gradient Method☆

Massimiliano Fasi[a], Julien Langou[b], Yves Robert[c], Bora Uçar[d,*]

[a]*The University of Manchester, UK*
[b]*University Colorado Denver, USA*
[c]*ENS Lyon, France & University Tennessee Knoxville, USA*
[d]*LIP, UMR5668 (CNRS - ENS Lyon - UCBL - Université de Lyon - INRIA), Lyon, France*

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 algorithm-based fault tolerance (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 introduce an abstract performance model to compute the performance of all schemes, and we instantiate it using the preconditioned conjugate gradient algorithm. Finally, we validate our new approach through a set of simulations.

Keywords: Fault-tolerance, Silent errors, Algorithm-based fault tolerance, Checkpointing, Sparse matrix-vector multiplication,, Preconditioned conjugate gradient method.
2010 MSC: 68M15, 94C12, 65F08

1. Introduction

Silent errors (or silent data corruptions) have become a significant concern in HPC environments [2]. There are many sources of silent errors, from bit

---

flips in cache caused by cosmic radiations, to wrong results produced by the arithmetic logic unit. The latter source becomes relevant when the computation is performed in the low voltage mode to reduce the energy consumption in large-scale computations. But the low levels of voltage dramatically reduces the dependability of the system.

The key problem with silent errors is the *detection latency*: when a silent error strikes, the corrupted data is not identified immediately, but instead only when some numerical anomaly is detected in the application behavior. It is clear that this detection can occur with an arbitrary delay. As a consequence, the de facto standard method for resilience, namely checkpointing and recovery, cannot be used directly. Indeed, the method of checkpointing and recovery applies to fail-stop errors (e.g., hardware crashes): such errors are detected immediately, and one can safely recover from the last saved snapshot of the application state. On the contrary, because of the detection latency induced by silent errors, it is often impossible to know when the error struck, and hence to determine which checkpoint (if any) is valid to safely restore the application state. Even if an unlimited number of checkpoints could be kept in memory, there would remain the problem of identifying 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 is 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 large-scale HPC application [3, 4, 5, 6]. The cost of re-executing the application one or more times becomes prohibitive, and other approaches need to be considered.

Several recent papers have proposed to introduce a verification mechanism to be applied periodically in order to detect silent errors. These papers mostly target iterative methods to solve sparse linear systems, which 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 iteration of the application, 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. Note that in all these approaches a *selective reliability* model is enforced, where the parts of the application that are not protected are assumed to execute in a reliable mode.

While verification mechanisms speed up the detection of silent errors, they cannot provide correction, and thus they cannot avoid the re-execution of the application from scratch. 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. A checkpoint is always valid because it is being preceded by a verification. If an error occurs, it will be detected by one of the c verifications performed before

the next checkpoint. This is exactly the approach proposed by Chen [7] for a variety of methods based on Krylov subspaces, including the widely used conjugate Gradient (CG) algorithm. Chen [7] gives an equation for the overhead incurred by checkpointing and verification, and determines the best values of c and d by finding a numerical solution of the equation. In fact, computing the optimal verification and checkpoint intervals is a hard problem. In the case of pure periodic checkpointing, closed-form approximations of the optimal period have been given by Young [8] and Daly [9]. However, when combining checkpointing and verification, the complexity of the problem grows. To the best of our knowledge, there is no known closed-form formula, although a dynamic programming algorithm to compute the optimal repartition of checkpoints and verifications is available [10].

For linear algebra kernels, another widely used technique for silent error detection is algorithm-based fault tolerance (ABFT). The pioneering paper of Huang and Abraham [11] 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 [12], LU with partial pivoting [13], Cholesky [14] and QR [15] factorizations, and more recently to sparse kernels like SpMxV (matrix-vector product) and triangular solve [16]. 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 an increased overhead, because several checksums are needed to detect and correct, while a single checksum is enough when just detection is required. 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 the execution can be safely resumed from that very same iteration. Only when two or more silent errors strike within an iteration we do need to rollback to the last checkpoint. In many practical situations, it is reasonable to expect no more than one error per iteration, which means that most roll-back operations can be avoided. In turn, this leads to less frequent checkpoints, and hence less overhead.

The major contributions of this paper are an ABFT framework to detect multiple errors striking the computation and a performance model that allows to compare methods that combine verification and checkpointing. The verification mechanism is capable of error detection, or of both error detection and correction. The model tries to determine the optimal intervals for verification and checkpointing, given the cost of an iteration, the overhead associated to verification, 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 a CG kernel, preconditioned with a sparse approximate inverse [17], and compare the performance of two ABFT-based verification mechanisms. We call the first scheme, capable of error detection only,

3

ABFT-Detection and the second scheme, which enhances the first one by providing single error correction as well, ABFT-Correction. Through numerical simulations, we compare the performance of both schemes with Online-Detection, the approach of Chen [7] (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.

Our discussion focuses on the sequential execution of iterative methods. Yet, all our techniques extend to parallel implementation based on the message passing paradigm (with using, e.g., MPI). In an implementation of SpMxV in such a setting, the processing elements (or processors) hold a part of the matrix and the input vector, and hold a part of the output vector at the end. A recent exposition of different algorithms can be found elsewhere [18]. Typically, the processors perform scalar multiply operations on the local matrix and the input vector elements, when all required vector elements have been received from other processors. The implementations of the MPI standard guarantees correct message delivery, i.e., checksums are incorporated into the message so as to prevent transmission errors (the receives can be done in-place and hence are protected). However, the receiver will obviously get corrupted data if the sender sends corrupted data. Silent errors can indeed strike at a given processor during local scalar multiply operations. Performing error detection and correction locally implies global error detection and correction for the SpMxV. Note that, in this case, the local matrix elements can form a matrix which cannot be assumed to be square in general (for some iterative solvers they can be). Furthermore, the mean time between failures (MTBF) reduces linearly with the number of processors. This is well-known for memoryless distributions of fault inter-arrival times and remains true for arbitrary continuous distributions of finite mean [19]. Therefore, resilient local matrix vector multiplies are required for resiliency in a parallel setting.

The rest of the paper is organized as follows. Section 2 provides an overview of related work. Section 3 provides background on ABFT techniques for the PCG algorithm, and presents both the ABFT-Detection and ABFT-Correction approaches. Section 4 discusses the proposed ABFT method for the SpMxV. Section 5 is devoted to the abstract performance model. Section 6 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 7.

2. Related work

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

4

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 [20], which induces a 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 [21]. Elliot et al. [22] 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 et al. [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 [24] and ABFT techniques (see below).

As already stated, most papers assume on a selective reliability setting [25, 26, 27, 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 energy consumption 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 et al. [29] suggest coupling a higher-order with a lower-order scheme for PDEs. 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 [30] design a fault-tolerant GMRES capable of converging despite silent errors. Bronevetsky and de Supinski [31] 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 [7], who deals with sparse iterative solvers. For PCG, the verification amounts to checking the orthogonality of two vectors and to recomputing and checking the residual (see Section 3 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 [11]. They describe an algorithm capable of detecting and correcting a single silent error striking a matrix-matrix

5

multiplication by means of row and column checksums. This germinal idea is then elaborated by Anfinson and Luk [32], 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 to relatively small matrices, and is hence out of our scope. Bosilca et al. [33] and Du et al. [12] present two relatively recent survey.

The problem of algorithm-based fault-tolerance for sparse matrices is investigated by Shantharam et al. [16], who suggest a way to detect a single error in an SpMxV at the cost of a few additional dot products. Sloan et al. [34] 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.

A preliminary version of this work appeared [1]. In the preliminary version, we dealt with CG only, without preconditioning. In practice, CG is most often used with preconditioning to accelerate the convergence. In the current work, we focus on preconditioned CG (PCG for short). We provide new results and experiments, together with several extensions on single error correction and multiple error detection.

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. In particular, we consider a PCG variant where the application of the preconditioner reduces to the computation of two SpMxV with triangular matrices [17], which are a sparse factorization of an approximate inverse of the coefficient matrix. In fact, the model discussed in this paper can be profitably employed for any sparse inverse preconditioner that can be applied by means of one or more SpMxV.

We first provide a background on the CG method and overview both Chen's stability tests [7] and our ABFT protection schemes.

The code for the variant of the PCG method we use is shown in Algorithm 1. The main loop features three 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*. Two of the sparse matrix-vector multiplications are carried out at line 10 where we compute $\mathbf{z}_{i+1} \leftarrow \mathbf{M}^\mathsf{T}\mathbf{M}\mathbf{r}_{i+1}$. Here the matrix $\mathbf{M}$ is the upper triangular factor of the approximate inverse $\mathbf{M}^\mathsf{T}\mathbf{M}$, where $\mathbf{M}^\mathsf{T}\mathbf{M}$ well approximates $\mathbf{A}^{-1}$ [17].

Chen's stability tests [7] 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.

Algorithm 1 The PCG algorithm.

Require: $\mathbf{A}, \mathbf{M} \in \mathbb{R}^{n \times n}, \mathbf{b}, \mathbf{x}_0 \in \mathbb{R}^n, \varepsilon \in \mathbb{R}$
Ensure: $\mathbf{x} \in \mathbb{R}^n \;:\; \|\mathbf{A}\mathbf{x} - \mathbf{b}\| \leq \varepsilon$
1: $\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_0$;
2: $\mathbf{z}_0 \leftarrow \mathbf{M}^\intercal \mathbf{M} \mathbf{r}_0$;
3: $\mathbf{p}_0 \leftarrow \mathbf{z}_0$;
4: $i \leftarrow 0$;
5: **while** $\|\mathbf{r}_i\| > \varepsilon \left(\|\mathbf{A}\| \cdot \|\mathbf{r}_0\| + \|b\| \right)$ **do**
6:      $\mathbf{q} \leftarrow \mathbf{A}\mathbf{p}_i$;
7:      $\alpha_i \leftarrow \|\mathbf{r}_i\|^2 / \mathbf{p}_i^\intercal \mathbf{q}$;
8:      $\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + \alpha\, \mathbf{p}_i$;
9:      $\mathbf{r}_{i+1} \leftarrow \mathbf{r}_i - \alpha\, \mathbf{q}$;
10:      $\mathbf{z}_{i+1} \leftarrow \mathbf{M}^\intercal \mathbf{M} \mathbf{r}_{i+1}$;
11:      $\beta \leftarrow \|\mathbf{r}_{i+1}\|^2 / \|\mathbf{r}_i\|^2$;
12:      $\mathbf{p}_{i+1} \leftarrow \mathbf{z}_{i+1} + \beta\, \mathbf{p}_i$;
13:      $i \leftarrow i + 1$;
14: **end whilereturn** $\mathbf{x}_i$;

Our only modification 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 give an overview of 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 $\mathbf{p}_i$. Since ABFT protection for vector operations is as costly as repeated computation, we use triple modular redundancy (TMR) for them for simplicity.

Although theoretically possible, constructing an 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 PCG versions 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 correctable.

We assume the selective reliability model in which all checksums and checksum related operations are non-faulty, also the tests for the orthogonality checks are non-faulty.

4. ABFT-SpMxV

Here, we discuss the proposed ABFT method for the SpMxV (combining ABFT with checkpointing is described later in Section 5.2). The proposed ABFT mechanisms are described for detecting single errors (Section 4.1), multiple errors (Section 4.2), and correcting a single error (Section 4.3).

4.1. Single error detection

The overhead of the standard single error correcting ABFT technique is too high for the sparse matrix-vector product case. Shantharam et al. [16] 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). Because their results depend 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 $Colid \in \mathbb{N}^{\mathrm{nnz}(\mathbf{A})}$, $Val \in \mathbb{R}^{\mathrm{nnz}(\mathbf{A})}$ and $Rowidx \in \mathbb{N}^{n+1}$; these arrays correspond to the arrays JA, AA, and IA in the more standard notation [35, Sec. 3.4]). Here $\mathrm{nnz}(\mathbf{A})$ is the number of non-zero entries in $\mathbf{A}$.

Shantharam et al. can protect $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$, 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

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

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. It can be shown [16] that, in 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, a condition that seems to restrict too much the practical applicability of their method. Shantharam et al. need this condition to ensure 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. 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 (1), 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 $\mathcal{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ of $\mathbb{R}^n$ – the rows of the input matrix. Each of these n vectors is perpendicular to a hyperplane h_i of $\mathbb{R}^n$, and $\mathbf{w}$ does not verify the condition

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

for any i, if and only if it lies on h_i. Since the Lebesgue measure in $\mathbb{R}^n$ of an hyperplane of $\mathbb{R}^n$ itself is zero, the union of these hyperplanes is measurable and has measure 0. Therefore, the probability that a vector $\mathbf{w}$ randomly picked in $\mathbb{R}^n$ 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 $\mathbb{R}^n$ one can have is finite, and the probability of randomly picking a vector that is orthogonal to a given one could be larger 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 6). 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 [36, Ch. 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 entries a constant value chosen so that all elements of the new vector are different from zero. We give the generalized result in Theorem 1.

Theorem 1 (Correctness of Algorithm 2). *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*

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

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

$$\mathbf{c} = \left[\sum_{i=1}^{n} a_{i,1} + m, \ldots, \sum_{i=1}^{n} a_{i,n} + m \right] ,$$

where m is such that

$$c_j = \sum_{i=1}^{n} a_{i,j} + m \ne 0 , \tag{4}$$

9

Algorithm 2 Shifting checksum algorithm.

Require: $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\mathbf{x} \in \mathbb{R}^n$
Ensure: $\mathbf{y} \in \mathbb{R}^n$ such that $\mathbf{y} = \mathbf{Ax}$ or the detection of a single error
 1: $\mathbf{x'} \leftarrow \mathbf{x}$;
 2: $[\underline{\mathbf{w}}, \mathbf{c}, m, c_r] = \textsc{ComputeChecksum}(\mathbf{A})$; **return** $\textsc{SpMxV}(\mathbf{A}, \mathbf{x}, \mathbf{x'}, \underline{\mathbf{w}}, \mathbf{c}, m, c_r)$;

 3: **function** $\textsc{ComputeChecksum}(\mathbf{A})$
 4:     Generate $\underline{\mathbf{w}} \in \mathbb{R}^{n+1}$;
 5:     $\mathbf{w} \leftarrow \underline{\mathbf{w}}_{1:n}$;
 6:     $\mathbf{c} \leftarrow \mathbf{w}^\mathsf{T} \mathbf{A}$;
 7:     **if** $\min(|\mathbf{c}|) = 0$ **then**;
 8:         Find m that verifies (4);
 9:         $\mathbf{c} \leftarrow \mathbf{c} + m$;
10:     **end if**
11:     $c_r \leftarrow \underline{\mathbf{w}}^\mathsf{T} Rowindex$; **return** $\underline{\mathbf{w}}, \mathbf{c}, m, c_r$;
12: **end function**

13: **function** $\textsc{SpMxV}(\mathbf{A}, \mathbf{x}, \mathbf{x'}, \underline{\mathbf{w}}, \mathbf{c}, m, c_r)$
14:     $\mathbf{w} \leftarrow \underline{\mathbf{w}}_{1:n}$;
15:     $s_r \leftarrow 0$;
16:     **for** $i \leftarrow 1$ to n **do**
17:         $y_i \leftarrow 0$;
18:         $s_r \leftarrow s_r + Rowindex_i$;
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:         **end for**
23:     **end for**
24:     $y_{n+1} \leftarrow m\, \mathbf{w}^\mathsf{T} \mathbf{x'}$;
25:     $c_y \leftarrow \underline{\mathbf{w}}^\mathsf{T} \mathbf{y}$;
26:     $d_x \leftarrow \mathbf{c}^\mathsf{T} \mathbf{x}$;
27:     $d_{x'} \leftarrow \mathbf{c}^\mathsf{T} \mathbf{x'}$;
28:     $d_r \leftarrow c_r - s_r$;
29:     **if** $d_x = 0 \wedge d_{x'} = 0 \wedge d_r = 0$ **then return** $\mathbf{y}_{1:n}$;
30:     **else**
31:         **error** ("Soft error is detected");
32:     **end if**
33: **end function**

---

for $1 \leq j \leq n$,

2. *an auxiliary checksum* $y_{n+1} = m \sum_{i=i}^{n} \widetilde{x}_i$,

3. *an auxiliary counter* s_r *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* $c_r = \sum_{i=1}^{n} Rowindex_i \in \mathbb{N}$.

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

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

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

iii. $s_r = c_r$.

Proof. There are three possible error sources when computing Equation (3), 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 $\mathbf{x}$.

We consider these three possible cases in the following discussion.

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

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

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_r 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 $\varepsilon \in \mathbb{R} \setminus \{0\}$. Noting that $\mathbf{x} = \mathbf{x}'$, the sum of the elements of $\widetilde{\mathbf{y}}$ gives

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

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

Let us remark that COMPUTECHECKSUM in Algorithm 2 does not require the input vector $\mathbf{x}$ of SpMxV as an argument. Therefore, in the common scenario of many SpMxV 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 these checksumming techniques.

Shifting the sum checksum vector by an amount is probably the simplest deterministic approach to relax the strictly diagonal dominance hypothesis, but it is not the only one. An alternative solution is described in Algorithm 3, which basically exploits the distributive property of matrix multiplication over matrix addition. The idea is to split the original matrix $\mathbf{A}$ into two matrices of the same size, $\mathbf{A}$ and $\widehat{\mathbf{A}}$, such that no column of either matrix has a zero checksum. Two standard ABFT multiplications, namely $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ and $\widehat{\mathbf{y}} \leftarrow \mathbf{A}\widehat{\mathbf{x}}$, are then performed. If no error occurs neither in the first nor in the second computation, the sum of $\mathbf{y}$ and $\widehat{\mathbf{y}}$ is computed in reliable mode and then returned. Let us note that, as we expect the number of non-zeros of $\widehat{\mathbf{A}}$ to be much smaller than n, we store sparsely both the checksum vector of $\widehat{\mathbf{A}}$ and the $\widehat{\mathbf{y}}$ vector.

We do not write down an extended proof of the correctness of this algorithm, and limit ourselves to a short sketch. We consider the same three cases as in the proof of Theorem 1, without introducing any new idea. An error in the computation of $\mathbf{y}$ or $\widehat{\mathbf{y}}$ can be detected using the dot product between the corresponding column checksum and the $\mathbf{x}$ error. An error in $\mathbf{A}$ can be detected by either c_r or an erroneous entry in $\mathbf{y}$ or $\widehat{\mathbf{y}}$, as the matrix loop structure of the sparse multiplication algorithm has not been changed. Finally, an error in the

Table 1: Overhead comparison for Algorithm 2 and Algorithm 3. Here n denotes the size of the matrix and n' the number of null sum columns.

	Algorithm 2	Algorithm 3
initialization of $\mathbf{y}$	n	n
computation of y_{n+1}	n	-
SpMxV overhead	-	n'
checksum check	$2n$	$2n + 2n'$
computation of c_y and $c_{\widehat{y}}$	n	$n + n'$
computation of $\mathbf{y} + \widehat{\mathbf{y}}$	-	n'
Total SpMxV overhead	$5n$	$4n + 5n'$

pth component of $\mathbf{x}$ would make the sum of the entries of $\mathbf{y}$ and $\widehat{\mathbf{y}}$ differ from $\mathbf{c}^\mathsf{T}\mathbf{x}'$ and $\widehat{\mathbf{c}}^\mathsf{T}\mathbf{x}'$, respectively.

The evaluation of the performance of the two algorithms, though straightforward from the point of view of the computational cost, has to be carefully assessed in order to devise a valid and practical trade-off between Algorithm 2 and Algorithm 3.

In both cases COMPUTECHECKSUM introduces an overhead of $\mathcal{O}\left(\text{nnz}(\mathbf{A})\right)$, but the shift version should in general be faster containing less assignments than its counterpart, and this changes the constant factor hidden by the asymptotic notation. Nevertheless, as we are interested in performing many SpMxV with the same matrix, this pre-processing overhead becomes negligible.

The function SpMxV has to be invoked once for each multiplication, and hence more care is needed. Copying $\mathbf{x}$ and initializing $\mathbf{y}$ both require n operations, and the multiplication is performed in time $\mathcal{O}\left(\text{nnz}(A)\right)$, but the split version pays an n' more to read the values of the sparse vector $\mathbf{b}$. The cost of the verification step depends instead on the number of zeroes of the original checksum vector, that is also the number of non-zero elements of the sparse vector $\widehat{\mathbf{c}}$. Let us call this quantity n'. Then the overhead is $4n$ for the shifting and $3n + 3n'$ for the splitting, that requires also the sum of two sparse vectors to return the result. Hence, as summarized in Table 1, the two methods bring different overheads into the computation. Comparing them, it is immediate to see that the shifting method is cheaper when

$$n' > \frac{n}{5} \, ,$$

while it has more operations to do when the opposite inequality holds. For the equality case, we can just choose to use the first method, because of the cheaper preprocessing phase. In view of this observation, it is possible to devise a simple algorithm that exploits this trade-off to achieve better performance. It suffices to compute the checksum vector of the input matrix, count the number of non-zeros and choose which detection method to use accordingly.

We also note that by splitting the matrix $\mathbf{A}$ into say ℓ pieces and checksumming each piece separately, we can possibly protect $\mathbf{A}$ from up to ℓ errors, by

---

Algorithm 3 Splitting checksum algorithm.

Require: $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\mathbf{x} \in \mathbb{R}^n$
Ensure: $\mathbf{y} \in \mathbb{R}^n$ such that $\mathbf{y} = \mathbf{Ax}$ or the detection of a single error

 1: $[\mathbf{c}, k, c_r] = \text{COMPUTECHECKSUM}(\mathbf{A})$; **return** $\text{SPMXV}(\mathbf{A}, \mathbf{x}, \mathbf{c}, \widehat{\mathbf{c}}, \mathbf{b}, c_r)$;

 2: **function** COMPUTECHECKSUM(A)
 3:    $\mathbf{c}, \mathbf{m} \leftarrow 0$;
 4:    **for** $i \leftarrow 1$ to $\text{nnz}(\mathbf{A})$ **do**
 5:        $ind \leftarrow Colid_i$;
 6:        $c_{ind} \leftarrow c_{ind} + Val_i$;
 7:        $m_{ind} \leftarrow i$;
 8:    **end for**
 9:    $k \leftarrow 0$;
10:    **for** $i \leftarrow 1$ to n **do**
11:        **if** $c_i = 0 \wedge m_i \neq 0$ **then**
12:            $b_{m_i} \leftarrow$ **true**;
13:            $\widehat{c}_i \leftarrow Val_{m_i}$;
14:            $c_i \leftarrow c_i - \widehat{c}_i$;
15:        **end if**
16:    **end for**
17:    $c_r \leftarrow \sum_{i=1}^{n} Rowindex_i$; **return** $\mathbf{c}, \widehat{\mathbf{c}}, \mathbf{b}, c_r$;
18: **end function**

19: **function** SPMXV($\mathbf{A}, \mathbf{x}, \mathbf{c}, \widehat{\mathbf{c}}, \mathbf{b}, c_r$)
20:    $\mathbf{x}' \leftarrow \mathbf{x}$;
21:    **for** $i \leftarrow 1$ to n **do**
22:        $y_i \leftarrow 0$;
23:    **end for**
24:    $s_r \leftarrow 0$;
25:    **for** $i \leftarrow 1$ to n **do**
26:        $s_r \leftarrow s_r + Rowindex_i$;
27:        **for** $j \leftarrow Rowindex_i$ to $Rowindex_{i+1} - 1$ **do**
28:            $ind \leftarrow Colid_j$;
29:            **if** b_j **then**
30:                $\widehat{y}_i \leftarrow \widehat{y}_i + Val_j \cdot x_{ind}$;
31:            **else**
32:                $y_i \leftarrow y_i + Val_j \cdot x_{ind}$;
33:            **end if**
34:        **end for**
35:    **end for**
36:    $c_y \leftarrow \sum_{i=1}^{n} y_i$; $c_{\widehat{y}} \leftarrow \sum_{i=1}^{n} \widehat{y}_i$;
37:    $d_x \leftarrow \mathbf{c}^{\mathsf{T}}\mathbf{x} - c_y$; $d_{\widehat{x}} \leftarrow \widehat{\mathbf{c}}^{\mathsf{T}}\mathbf{x} - c_{\widehat{y}}$;
38:    $d_{x'} \leftarrow \mathbf{c}^{\mathsf{T}}\mathbf{x}' - c_y$; $d_{\widehat{x}'} \leftarrow \widetilde{\mathbf{c}}^{\mathsf{T}}\mathbf{x}' - c_{\widehat{y}}$;
39:    $d_r \leftarrow c_r - s_r$;
40:    **if** $d_x = 0 \wedge d_{x'} = 0 \wedge d_{\widehat{x}} = 0 \wedge d_{\widehat{x}'} = 0 \wedge d_r = 0$ **then return** $\mathbf{y} + \widehat{\mathbf{y}}$;
41:    **else**
42:        **error** ("Soft error is detected");
43:    **end if**
44: **end function**

---

protecting each piece against a single error (obviously the multiple errors should hit different pieces).

4.2. Multiple error detection

With some effort, the shifting idea in Algorithm 2 can be extended to detect errors striking a single SpMxV. Let us consider the problem of detecting up to k errors in the computation of $\mathbf{y} \leftarrow \mathbf{Ax}$ introducing an overhead of $\mathcal{O}(kn)$. Let k weight vectors $\mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(k)} \in \mathbb{R}^n$ be such that any sub-matrix of

$$\mathbf{W} = \left[\mathbf{w}^{(1)} \ \mathbf{w}^{(2)} \ \cdots \ \mathbf{w}^{(k)} \right]$$

has full rank. For $k = 2$ and $k = 3$, we give a possible choice for the $\mathbf{W}$ matrix explicitly. For larger values of k, we suggest the use of rectangular random matrices, which besides having full rank sub-matrices are also well conditioned with high probability [37, 38].

To build our ABFT scheme let us note that, if no error occurs, for each weight vector $\mathbf{w}^{(\ell)}$ it holds that

$$\mathbf{w}^{(\ell)\intercal} \mathbf{A} = \left[\sum_{i=1}^n w_i^{(\ell)} a_{i,1}, \ldots, \sum_{i=1}^n w_i^{(\ell)} a_{i,n} \right],$$

and hence that

$$\mathbf{w}^{(\ell)\intercal} \mathbf{Ax} = \sum_{i=1}^n w_i^{(\ell)} a_{i,1} x_1 + \cdots + \sum_{i=1}^n w_i^{(\ell)} a_{i,n} x_n = \sum_{i=1}^n \sum_{j=1}^n w_i^{(\ell)} a_{i,j} x_j .$$

Similarly, the sum of the entries of $\mathbf{y}$ weighted with the same $\mathbf{w}^{(\ell)}$ is

$$\sum_{i=1}^n w_i^{(\ell)} y_i = w_1^{(\ell)} y_1 + \cdots + w_n^{(\ell)} y_n$$

$$= w_1^{(\ell)} \sum_{j=1}^n a_{1,j} x_j + \cdots + w_n^{(\ell)} \sum_{j=1}^n a_{n,j} x_j = \sum_{i=1}^n \sum_{j=1}^n w_i^{(\ell)} a_{i,j} x_j ,$$

and we can conclude that

$$\sum_{i=1}^n w_i^{(\ell)} y_i = \left(\mathbf{w}^{(\ell)\intercal} \mathbf{A} \right) \mathbf{x} ,$$

for any $\mathbf{w}^{(\ell)}$ with $1 \leq \ell \leq k$.

To convince ourself that with these checksums it is actually possible to detect up to k errors, let us suppose that k' errors, with $k' \leq k$, occur in positions $p_1, \ldots, p_{k'}$, and let us denote by $\widetilde{\mathbf{y}}$ the faulty vector where $\widetilde{y}_{p_i} = y_{p_i} + \varepsilon_{p_i}$ for $\varepsilon_{p_i} \in \mathbb{R} \setminus \{0\}$ and $1 \leq i \leq k'$ and $\widetilde{y}_i = y_i$ otherwise. Then for each weight vector we have

$$\sum_{i=1}^n w_i^{(\ell)} \widetilde{y}_i - \sum_{i=1}^n w_i^{(\ell)} y_i = \sum_{j=1}^{k'} w_{p_j}^{(\ell)} \varepsilon_{p_j} .$$

15

Said otherwise, the occurrence of the k' errors is not detected if and only if, for $1 \leq \ell \leq k$, all the ε_{p_i} respect

$$\sum_{j=1}^{k'} w_{p_j}^{(\ell)} \varepsilon_{p_j} = 0 \ . \tag{5}$$

We claim that there cannot exist a vector $(\varepsilon_{p_1}, \ldots, \varepsilon_{p'_k})^\intercal \in \mathbb{R}^{k'} \setminus \{0\}$ such that all the conditions in (5) are simultaneously verified. These conditions can be expressed in a more compact way as a linear system

$$\begin{pmatrix} w_{p_1}^{(1)} & \cdots & w_{p_{k'}}^{(1)} \\ \vdots & \ddots & \vdots \\ w_{p_1}^{(k)} & \cdots & w_{p_{k'}}^{(k)} \end{pmatrix} \begin{pmatrix} \varepsilon_{p_1} \\ \vdots \\ \varepsilon_{p_{k'}} \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix} .$$

Denoting by $\mathbf{W}^*$ the coefficient matrix of this system, it is clear that the errors cannot be detected if only if $(\varepsilon_{p_1}, \ldots \varepsilon_{p_{k'}})^\intercal \in \ker(\mathbf{W}^*) \setminus \{0\}$. Because of the properties of $\mathbf{W}$, we have that $\operatorname{rk}(\mathbf{W}^*) = k$. Moreover, it is clear that the rank of the augmented matrix

$$\begin{pmatrix} w_{p_1}^{(1)} & \cdots & w_{p_{k'}}^{(1)} & \bigg| & 0 \\ \vdots & \ddots & \vdots & \bigg| & \vdots \\ w_{p_1}^{(k)} & \cdots & w_{p_{k'}}^{(k)} & \bigg| & 0 \end{pmatrix}$$

is k as well. Hence, by means of the Rouché–Capelli theorem, the solution of the system is unique and the null space of $\mathbf{W}^*$ is trivial. Therefore, this construction can detect the occurrence of k' errors during the computation of $\mathbf{y}$ by comparing the values of the weighted sums $\mathbf{y}^\intercal \mathbf{w}^{(\ell)}$ with the result of the dot product $(\mathbf{w}^{(\ell)\intercal} \mathbf{A})\mathbf{x}$, for $1 \leq \ell \leq k$.

However, to get a true extension of the algorithm described in the previous section, we also need to make it able to detect errors that strike the sparse representation of $\mathbf{A}$ and that of $\mathbf{x}$. The first case is simple, as the k errors can strike the *Val* or *Colid* arrays, leading to at most k errors in $\widetilde{\mathbf{y}}$, or in *Rowindex*, where they can be caught using k weighted checksums of the *Rowindex* vector.

Detection in $\mathbf{x}$ is much trickier, since neither the algorithm just described nor a direct generalization of Algorithm 2 can manage this case. Nevertheless, a proper extension of the shifting technique is still possible. Let us note that there exists a matrix $\mathbf{M} \in \mathbb{R}^{k \times n}$ such that

$$\mathbf{W}^\intercal \mathbf{A} + \mathbf{M} = \mathbf{W} \ .$$

The elements of such an $\mathbf{M}$ can be easily computed, once that the checksum rows are known. Let $\widetilde{\mathbf{x}} \in \mathbb{R}^n$ be the faulty vector, defined by

$$\widetilde{x}_i = \begin{cases} x_i + \varepsilon_{p_i}, & 1 \leq i \leq k', \\ x_i & otherwise. \end{cases}$$

for some $k' \leq k$, and let us define $\widetilde{\mathbf{y}} = \mathbf{A}\widetilde{\mathbf{x}}$. Now, let us consider a checksum vector $\mathbf{x}' \in \mathbb{R}^n$ such that $\mathbf{x}' = \mathbf{x}$ and let assume that it cannot be modified by a transient error. For $1 \leq \ell \leq k$, it holds that

$$
\sum_{i=1}^{n} w_i^{(\ell)} \widetilde{y}_i + \sum_{j=1}^{n} m_{\ell,j} \widetilde{x}_j = \sum_{i=1}^{n}\sum_{j=1}^{n} w_i^{(\ell)} a_{i,j} x_j + \sum_{i=1}^{k'} \varepsilon_{p_i} \left(\sum_{j=1}^{n} w_j^{(\ell)} a_{j,p_i} \right) + \sum_{j=1}^{n} m_{\ell,j} x_j + \sum_{i=1}^{k'} \varepsilon_{p_i} m_{l,p_i}
$$

$$
= \sum_{j=1}^{n}\sum_{i=1}^{n} w_i^{(\ell)} a_{i,j} x_j + \sum_{j=1}^{n} m_{\ell,j} x_j + \sum_{i=1}^{k'} \varepsilon_{p_i} \left(\sum_{j=1}^{n} w_j^{(\ell)} a_{j,p_i} + m_{l,p_i} \right)
$$

$$
= \sum_{j=1}^{n} \left(\sum_{i=1}^{n} w_i^{(\ell)} a_{i,j} \right) x_j + \sum_{j=1}^{n} m_{\ell,j} x_j + \sum_{i=1}^{k'} \varepsilon_{p_i} w_{p_i}^{(\ell)}
$$

$$
= \sum_{j=1}^{n} \left(\sum_{i=1}^{n} w_i^{(\ell)} a_{i,j} + m_{\ell,j} \right) x_j + \sum_{i=1}^{k'} \varepsilon_{p_i} w_{p_i}^{(\ell)}
$$

$$
= \mathbf{w}^{(\ell)\mathsf{T}} \mathbf{x} + \sum_{i=1}^{k'} \varepsilon_{p_i} w_{p_i}^{(\ell)}
$$

$$
= \mathbf{w}^{(\ell)\mathsf{T}} \mathbf{x}' + \sum_{i=1}^{k'} \varepsilon_{p_i} w_{p_i}^{(\ell)} .
$$

Therefore, an error is not detected if and only if the linear system

$$
\begin{pmatrix} w_{p_1}^{(1)} & \cdots & w_{p_{k'}}^{(1)} \\ \vdots & \ddots & \vdots \\ w_{p_1}^{(k)} & \cdots & w_{p_{k'}}^{(k)} \end{pmatrix} \begin{pmatrix} \varepsilon_{p_1} \\ \vdots \\ \varepsilon_{p_{k'}} \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}
$$

has a non-trivial solution. But we have already seen that such a situation can never happen, and we can thus conclude that our method, whose pseudocode we give in Algorithm 4, can also detect up to k errors occurring in $\mathbf{x}$. Therefore, we have proven the following theorem.

Theorem 2 (Correctness of Algorithm 4). *Let us consider the same notation as in Theorem 1. Let $\underline{\mathbf{W}} \in \mathbb{R}^{n+1 \times n}$ be a matrix such that any square submatrix is full rank, and let us denote by $\mathbf{W} \in \mathbb{R}^{n \times n}$ the matrix of its first n rows. Let us assume an encoding scheme that relies on*

1. *an auxiliary checksum matrix $\mathbf{C} = (\mathbf{W}^\mathsf{T} \mathbf{A})^\mathsf{T}$,*

2. *an auxiliary checksum matrix $\mathbf{M} = \mathbf{W} - \mathbf{C}$,*

3. *a vector of auxiliary counters $\mathbf{s}_{Rowindex}$ initialized to the null vector and updated at runtime as in lines $16 - 17$ of Algorithm 4),*

4. *an auxiliary checksum vector $\mathbf{c}_{Rowindex} = \underline{\mathbf{W}}^\mathsf{T} Rowindex$.*

Then, up to k errors striking the computation of $\mathbf{y}$ or the memory locations that store $\mathbf{A}$ or $\mathbf{x}$, cause one of the following conditions to fail:

i. *$\mathbf{W}^\mathsf{T} \mathbf{y} = \mathbf{C}^\mathsf{T} \mathbf{x}$,*

ii. $\mathbf{W}^{\mathsf{T}}\left(\mathbf{x}' - \mathbf{y}\right),$

iii. $\mathbf{s}_{Rowindex} = \mathbf{c}_{Rowindex}.$

Let us note that we have just shown that our algorithm can detect up to k errors striking only $\mathbf{A}$, or only $\mathbf{x}$ or only the computation. Nevertheless, this result holds even when the errors are distributed among the possible cases, as long as at most k errors rely on the same checkpoint.

It is clear that the execution time of the algorithm depends on both $\mathrm{nnz}(\mathbf{A})$ and k. For the COMPUTECHECKSUM function, the cost is, assuming that the weight matrix $\mathbf{W}$ is already known, $\mathcal{O}\left(k\,\mathrm{nnz}(\mathbf{A})\right)$ for the computation of $\mathbf{C}$, and $\mathcal{O}\left(kn\right)$ for the computation of $\mathbf{M}$ and $\mathbf{c}_{Rowindex}$. Hence the number of performed operations is $\mathcal{O}\left(k\,\mathrm{nnz}(\mathbf{A})\right)$. The overhead added to the SpMxV depends instead on the computation of four checksum matrices that lead to a number of operations that grows asymptotically as kn.

4.3. Single error correction

We now discuss single error correction, using Algorithm 4 as a reference. We describe how a single error striking either memory or computation can be not only detected but also corrected at line 27. We use only two checksum vectors, that is, we describe correction of single errors assuming that two errors cannot strike the same SpMxV. By the end of the section, we will generalize this approach and discuss how single error correction and double error detection can be performed concurrently by exploiting three linearly independent checksum vectors. Whenever a single error is detected, regardless of its location (computation or memory), it is corrected by means of a succession of various steps. When one or more errors are detected, the correction mechanism tries to determine their multiplicity and, in case of a single error, what memory locations have been corrupted or what computation has been miscarried. Errors are then corrected using the values of the checksums and, if need be, partial recomputations of the result are performed.

As we did for multiple error detection, we require that any 2×2 submatrix of $\mathbf{W} \in \mathbb{R}^{n \times 2}$ has full rank. The simplest example of weight matrix having this property is probably

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

To detect errors striking $Rowidx$, we compute the ratio ρ 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 this distance is smaller, the algorithm concludes that the σth element, where $\sigma = Round(\rho)$ is the nearest integer to ρ, of $Rowidx$ is faulty, performs the correction by subtracting the first component of $\mathbf{d}_r$ from $Rowidx_{\sigma}$, and recomputes y_{σ} and $y_{\sigma-1}$, if the error in

18

Algorithm 4 Shifting checksum algorithm for k errors detection.

Require: $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\mathbf{x} \in \mathbb{R}^n$

Ensure: $\mathbf{y} \in \mathbb{R}^n$ such that $\mathbf{y} = \mathbf{A}\mathbf{x}$ or the detection of up to k errors

1: $\mathbf{x}' \leftarrow \mathbf{x}$
2: $[\underline{\mathbf{W}}, \mathbf{C}, \mathbf{M}, \widetilde{\mathbf{c}}] = \textsc{computeChecksums}(\mathbf{A}, k)$; **return** $\textsc{SpMxV}(\mathbf{A}, \mathbf{x}, \mathbf{x}', \underline{\mathbf{W}}, \mathbf{C}, \mathbf{M}, k, \widetilde{\mathbf{c}})$;

3: **function** $\textsc{computeChecksums}(\mathbf{A}, k)$
4:      Generate $\underline{\mathbf{W}} = \begin{bmatrix} \mathbf{w}^{(1)} \dots \mathbf{w}^{(k)} \end{bmatrix} \in \mathbb{R}^{n+1 \times n}$;
5:      $\mathbf{W} \leftarrow \underline{\mathbf{W}}_{1:n,*} \in \mathbb{R}^{n \times k}$
6:      $\mathbf{C}^{\intercal} \leftarrow \mathbf{W}^{\intercal} \mathbf{A}$;
7:      $\mathbf{M} \leftarrow \mathbf{W} - \mathbf{C}$;
8:      $\mathbf{c}_{Rowindex} \leftarrow \underline{\mathbf{W}}^{\intercal} Rowindex$; **return** $\underline{\mathbf{W}}, \mathbf{C}, \mathbf{M}, \mathbf{c}_{Rowindex}$;
9: **end function**

10: **function** $\textsc{SpMxV}(\mathbf{A}, \mathbf{x}, \mathbf{x}', \underline{\mathbf{W}}, \mathbf{C}, \mathbf{M}, k, \widetilde{\mathbf{c}})$
11:      $\mathbf{W} \leftarrow \underline{\mathbf{W}}_{1:n,*} \in \mathbb{R}^{n \times k}$
12:      $\widetilde{\mathbf{s}} \leftarrow [0, \dots, 0]$;
13:      **for** $i \leftarrow 1$ to n **do**
14:          $y_i \leftarrow 0$;
15:          **for** $j = 1$ to k **do**
16:              $\widetilde{s}_j \leftarrow \widetilde{s}_j + w_{ij} Rowindex_i$;
17:          **end for**
18:          **for** $j \leftarrow Rowindex_i$ to $Rowindex_{i+1} - 1$ **do**
19:              $ind \leftarrow Colid_j$;
20:              $y_i \leftarrow y_i + Val_j \cdot x_{ind}$;
21:          **end for**
22:      **end for**
23:      $\mathbf{d}_x \leftarrow \mathbf{W}^{\intercal} \mathbf{y} - \mathbf{C}^{\intercal} \mathbf{x}$;
24:      $\mathbf{d}_{x'} \leftarrow \mathbf{W}^{\intercal} (\mathbf{x}' - \mathbf{y}) - \mathbf{M}^{\intercal} \mathbf{x}$;
25:      $\mathbf{d}_r \leftarrow \widetilde{\mathbf{c}} - \widetilde{\mathbf{s}}$;
26:      **if** $\mathbf{d}_x = 0 \wedge \mathbf{d}_{x'} = 0 \wedge \mathbf{d}_r = 0$ **then return** $\mathbf{y}$;
27:      **else**
28:          **error** ("Soft errors are detected");
29:      **end if**
30: **end function**

Rowindex$_\sigma$ is a decrement; or $y_{\sigma+1}$ if it was an increment. Otherwise, it just emits an error.

The correction of errors striking *Val*, *Colid* and the computation of y are performed together. Let now ρ be the ratio of the second component of $\mathbf{d}_x$ to the first one. If ρ is near enough to an integer σ, 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 $\widetilde{\mathbf{C}} = |\mathbf{C} - \mathbf{C}'|$. At this stage, three cases are possible:

- If $z_{\widetilde{\mathbf{C}}} = 0$, then the error is in the computation of y_σ, and can be corrected by simply recomputing this value.

- If $z_{\widetilde{\mathbf{C}}} = 1$, then the error has struck 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 σ 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 σ. 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_σ 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 ρ, 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 σ is smaller than ε. Provided that this condition is verified, the algorithm computes the value of the error $\tau = \sum_{i=1}^{n} x_i - cx_\sigma$ and corrects $x_\sigma = x_\sigma - \tau$. 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 $x_\sigma^\tau = \tau$ and $x_i^\tau = 0$ otherwise.

Let us now investigate how detection and correction can be combined and let us give some details about the implementation of ABFT-CORRECTION as defined in Section 3. Indeed, note that double errors could be shadowed when using Algorithm 2, although the probability of such an event is negligible.

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 $\mathbf{C}^\intercal \mathbf{x} \in \mathbb{R}^2$ 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, we have $\delta_1, \delta_2 \neq 0$ and $\frac{\delta_2}{\delta_1} \in \mathbb{N}$, 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 behavior. In particular, when $\frac{\delta_2}{\delta_1} = p \in \mathbb{N}$, it is not clear how to discern between single and double errors. Let $\varepsilon_1, \varepsilon_2 \in \mathbb{R} \setminus \{0\}$ be the value of two errors occurring at position p_1 and p_2 respectively, and let $\widetilde{\mathbf{y}} \in \mathbb{R}^n$ 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 conditions

$$\delta_1 = \varepsilon_1 + \varepsilon_2, \tag{6}$$
$$\delta_2 = p_1 \varepsilon_1 + p_2 \varepsilon_2 , \tag{7}$$

hold. Therefore, if ε_1 and ε_2 are such that

$$p(\varepsilon_1 + \varepsilon_2) = p_1 \varepsilon_1 + p_2 \varepsilon_2 , \tag{8}$$

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. Let us consider a weight matrix $\widehat{\mathbf{W}} \in \mathbb{R}^{n \times 3}$ that includes a quadratic weight vector

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

and the tridimensional 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

$$\delta_1 = \varepsilon_1 + \varepsilon_2$$
$$\delta_2 = p_1 \varepsilon_1 + p_2 \varepsilon_2$$
$$\delta_3 = p_1^2 \varepsilon_1 + p_2^2 \varepsilon_2 .$$

To be confused with a single error in position p, ε_1 and ε_2 have to be such that

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

and

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

hold simultaneously for some $p \in \mathbb{N}$. 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 is easy to see that the determinant of the coefficient matrix is

$$\left(p - p_1\right)\left(p - p_2\right)\left(p_2 - p_1\right),$$

which 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 a single error from double errors.

5. Performance model

In Section 5.1, we introduce the general performance model. Then in Section 5.2 we instantiate it for the three methods that we are considering, namely ONLINE-DETECTION, ABFT-DETECTION and ABFT-CORRECTION.

5.1. General approach

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 λ is the fault rate.

The goal of this section is to compute the expected time $\mathbb{E}\left(s, T\right)$ 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 earlier work [39], we derive the following recursive equation to compute the expected completion time of a single frame:

$$\begin{aligned} \mathbb{E}\left(s, T\right) = {} & q^s(s(T + T_{verif})) + T_{cp}) \\ & + (1 - q^s)\left(\mathbb{E}\left(T_{lost}\right) + T_{rec} + \mathbb{E}\left(s, T\right)\right). \end{aligned} \tag{9}$$

22

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 $\mathbb{E}(T_{lost})$, and that forces us to recover (in time T_{rec}) 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:

$$f_i \quad = \quad \mathbb{P}(\text{error strikes at chunk } i | \text{there is an error in the frame}) \quad . \quad (10)$$

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 ith 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 ith chunk, and we have lost the time spent executing the first i 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 (9), we obtain

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

which simplifies into

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

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

$$s = \underset{s \ge 1}{\operatorname{argmin}} \left(\frac{\mathbb{E}(s, T)}{sT} \right) . \quad (11)$$

The minimization is complicated and should be conducted numerically (because T, the size of a chunk, is still unknown). Luckily, a dynamic programming algorithm to compute the optimal value of T and s is available [10].

5.2. Instantiation to PCG

For each of the three methods, ONLINE-DETECTION, ABFT-DETECTION and ABFT-CORRECTION, we instantiate the previous model and discuss how to solve (11).

5.2.1. ONLINE-DETECTION

For Chen's method [7], we have chunks of d iterations, hence $T = dT_{iter}$, where T_{iter} is the raw cost of a PCG iteration without any resilience method. The verification time T_{verif} is the cost of the orthogonality check operations performed as described in Section 3. 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 cumulative rate of $\lambda = \lambda_a + \lambda_m$, and the success probability for a chunk is $q = e^{-\lambda T}$.

Plugging these values in (11) gives an optimization formula very similar to that of Chen [7, Sec. 5.2], the only difference being that we assume that the verification is error-free, which is needed for the correctness of the approach.

5.2.2. 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 $q = e^{-\lambda T}$.

5.2.3. ABFT-CORRECTION

In addition to detection, we now correct single errors at every chunk. 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 λ, where $\lambda = \lambda_a + \lambda_m$ as for ONLINE-DETECTION and ABFT-DETECTION. The probability of exactly k errors in time T is $\frac{(\lambda T)^k}{k!}e^{-\lambda T}$ [40], 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 $q = e^{-\lambda T} + \lambda T e^{-\lambda T}$.

6. Experiments

6.1. Setup

There are two different sources of advantages in combining ABFT and check-pointing. First, the error detection capability lets us perform a cheap validation of the partial result of each PCG step, recovering as soon as an error strikes. Second, being able to correct single errors makes each step more resilient and increases the expected number of consecutive valid iterations. We say an iteration is valid if it is non-faulty, or if it suffers from a single error that is corrected via ABFT.

For our experiments, we use a set of positive definite matrices from the UFL Sparse Matrix Collection [41], with size between 17456 and 74752 and density lower than 10^{-2}. These matrices are listed in Table 2. Among these matrices, "Andrews/Andrews" and "GHS_psdef/jnlbrng1" are diagonally dominant, others are not. We perform the experiments under Matlab and use the factored approximate inverse preconditioners [17, 42] in the PCG. The application of these preconditioners requires two SpMxV, which are protected against error using the methods proposed in Section 4 (in all methods ONLINE-DETECTION, ABFT-DETECTION, and ABFT-CORRECTION).

At each iteration of PCG, faults are injected during vector and matrix-vector operations but, since we are assuming selective reliability, all 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 λ, as detailed in Section 5.2. These bit flips can strike either the matrix (the elements of *Val*, *Colid* and *Rowidx*), or any entry of the PCG vectors $\mathbf{r}_i$, $\mathbf{z}_i$, $\mathbf{q}$, $\mathbf{p}_i$ or $\mathbf{x}_i$. We chose not to inject errors during the computation explicitly, as they are just a special case of error 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 λ is chosen to be inversely proportional to M (memory size) with a proportionality constant $\alpha \in (0, 1)$; this makes sense as larger the memory used by an application, larger is the chance to have an error. It follows that the expected number of PCG iterations 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 (single detection and rolling back as soon as an error is detected), and ABFT-CORRECTION (correcting single errors during a given step and rolling back only if two errors strike a single operation). We instantiate them by limiting the maximum number of PCG steps to 50 (20 for #924, whose convergence is sublinear) and setting the tolerance parameter ϵ at line 5 of Algorithm 1 to 10^{-14}. The number of iterations for a non-faulty execution and the achieved accuracy are detailed in Table 2.

Implementing the null checks in Algorithm 2, Algorithm 3 and Algorithm 4 poses a challenge. The comparison $\mathbf{d}_r = 0$ is between two integers, and can be correctly evaluated by any programming language using the equality check.

Table 2: Test matrices used in the experiments. Name and id are from the University of Florida Sparse Matrix Collection.

name	id	size	density	steps	residual
Boeing/bcsstk36	341	23052	2.15e-03	50	6.41e-04
Mulvey/finan512	752	74752	1.07e-04	25	2.19e-14
Andrews/Andrews	924	60000	2.11e-04	20	1.59e-04
GHS_psdef/wathen100	1288	30401	5.10e-04	50	2.55e-13
GHS_psdef/wathen120	1289	36441	4.26e-04	50	9.16e-14
GHS_psdef/gridgena	1311	48962	2.14e-04	50	5.61e-05
GHS_psdef/jnlbrng1	1312	40000	1.24e-04	50	5.83e-13
UTEP/Dubcova2	1848	65025	2.44e-04	50	1.16e-05
JGD_Trefethen/Trefethen_20000	2213	20000	1.39e-03	10	6.00e-16

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 [43, Sec. 2.2] to make sure that errors caught by our algorithms really are errors and not merely inaccuracies due to floating point operations (which is tolerable, as non-faulty executions can give rise to the same inaccuracy).

Theorem 3 (Accuracy of the floating point weighted checksums). *Let $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{c} \in \mathbb{R}^n$. If all of the sums involved into the matrix operations are performed using some flavor of recursive summation [43, Ch. 4], it holds that*

$$\left| fl\left((\mathbf{c}^{\mathsf{T}} \mathbf{A}) \, \mathbf{x} \right) - fl\left(\mathbf{c}^{\mathsf{T}} (\mathbf{A} \mathbf{x}) \right) \right| \leq 2 \, \gamma_{2n} \, |\mathbf{c}^{\mathsf{T}}| \, |\mathbf{A}| \, |\mathbf{x}| \; . \tag{12}$$

We refer the reader to the technical report for the proof [44, Theorem 2]. 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 (12) can be safely replaced with $\mathbf{c}$ itself. It is also clear that these bounds are not computable, since $\mathbf{c}^{\mathsf{T}} \, | \, \mathbf{A} \, | \, | \, \mathbf{x} \, |$ is not, in general, a floating point number. This problem can be alleviated by overestimating the bound by means of matrix and vector norms.

Since we are interested in actually computing the bound at runtime, we consider a weaker bound. Recalling that [45, Sec. B.7]

$$\|\mathbf{A}\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{n} |a_{i,j}| \; . \tag{13}$$

we can upper bound the right hand side in so that

$$\left| fl\left((\mathbf{c}^{\mathsf{T}} \mathbf{A}) \, \mathbf{x} \right) - fl\left(\mathbf{c}^{\mathsf{T}} (\mathbf{A} \mathbf{x}) \right) \right| \leq 2 \, \gamma_{2n} \, n \, \|\mathbf{c}^{\mathsf{T}}\|_{\infty} \, \|\mathbf{A}\|_1 \, \|\mathbf{x}\|_{\infty} \; . \tag{14}$$

Table 3: 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 $E_t\left(\widetilde{s_i}\right)$ and $E_t\left(s_i^*\right)$ stand for the execution time of the algorithm using these checkpointing intervals.

id	$\widetilde{s_1}$	$E_t\left(\widetilde{s_1}\right)$	s_1^*	$E_t\left(s_1^*\right)$	l_1	$\widetilde{s_2}$	$E_t\left(\widetilde{s_2}\right)$	s_2^*	$E_t\left(s_2^*\right)$	l_2
341	4	305.42	1	293.22	**4.16**	4	305.45	1	293.16	**4.19**
752	30	13.81	24	13.34	**3.57**	24	12.17	23	11.93	**2.01**
924	23	49.82	30	47.53	**4.82**	23	44.52	26	42.42	**4.96**
1288	22	11.12	19	10.82	**2.72**	22	11.32	19	11.03	**2.58**
1289	16	16.56	13	16.38	**1.07**	23	13.49	23	13.49	**0.00**
1311	4	216.70	1	207.97	**4.19**	4	220.20	1	208.19	**5.77**
1312	25	14.41	22	13.86	**3.97**	23	12.30	22	12.06	**1.96**
1848	4	321.70	1	309.28	**4.01**	4	366.03	1	314.20	**16.49**
2213	19	2.31	12	2.19	**5.58**	24	2.33	23	2.20	**5.94**

Though the right hand side of (14) is not exactly computable in floating point arithmetic, it requires an amount of operations dramatically smaller than (12); just a few sums for the norm of $\mathbf{A}$. As this norm is usually computed using the identity in (13), any kind of summation yields a relative error of at most $n'\mathbf{u}$ [43, Sec. 4.6], where n' is the maximum number of nonzeros in a column of $\mathbf{A}$, and $\mathbf{u}$ is the machine epsilon. 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, since the right hand side in (14) does not depend on $\mathbf{x}$, it can be computed just once for a given matrix and weight vector.

Using (14) as tolerance parameter guarantees no false positives (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. [46, 47] 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.

6.2. Simulations

To validate the model, we perform the simulation whose results are illustrated in Table 3. For each matrix, we set $\lambda = \frac{1}{16\,M}$ and consider the average execution time of 100 repetitions of both ABFT-DETECTION (columns 5-8) and ABFT-CORRECTION (columns 6-9). In the table we record the checkpointing interval s_i^* which achieves the shortest execution time $E_t\left(s_1^*\right)$, and the checkpointing interval $\widetilde{s_i}$ which is the best stepsize according to our method, along

with its execution time $E_t\left(\widetilde{s_i}\right)$. Finally, we evaluate the performance of our guess by means of the quantity

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

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, we clearly see that the values of $\widetilde{s_i}$ and s_i^* are close, since the time loss reaches just above 5% for l_1 and just below 15% for l_2. This sometimes 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 other values of λ.

We also compare the execution time of the three algorithms to empirically asses how much their relative performance depend on the fault rate. The results on our test matrices are shown in Fig. 1, where the y-axis is the execution time (in seconds), and the x-axis is the normalized mean time between failure (the reciprocal of α). Here, 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 λ, we draw the average execution time of 50 runs of the three algorithms, using the best checkpointing interval predicted in Section 5.1 for ABFT-DETECTION and ABFT-CORRECTION, and by Chen [7, Eq. 10] for ONLINE-DETECTION. In terms of execution time, Chen's method is comparable to ours for middle to high fault rates, since it clearly outperforms ABFT-DETECTION in five out of nine cases, being slightly faster than ABFT-CORRECTION for lower fault rates.

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

7. 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 PCG kernel that can protect axpy's and
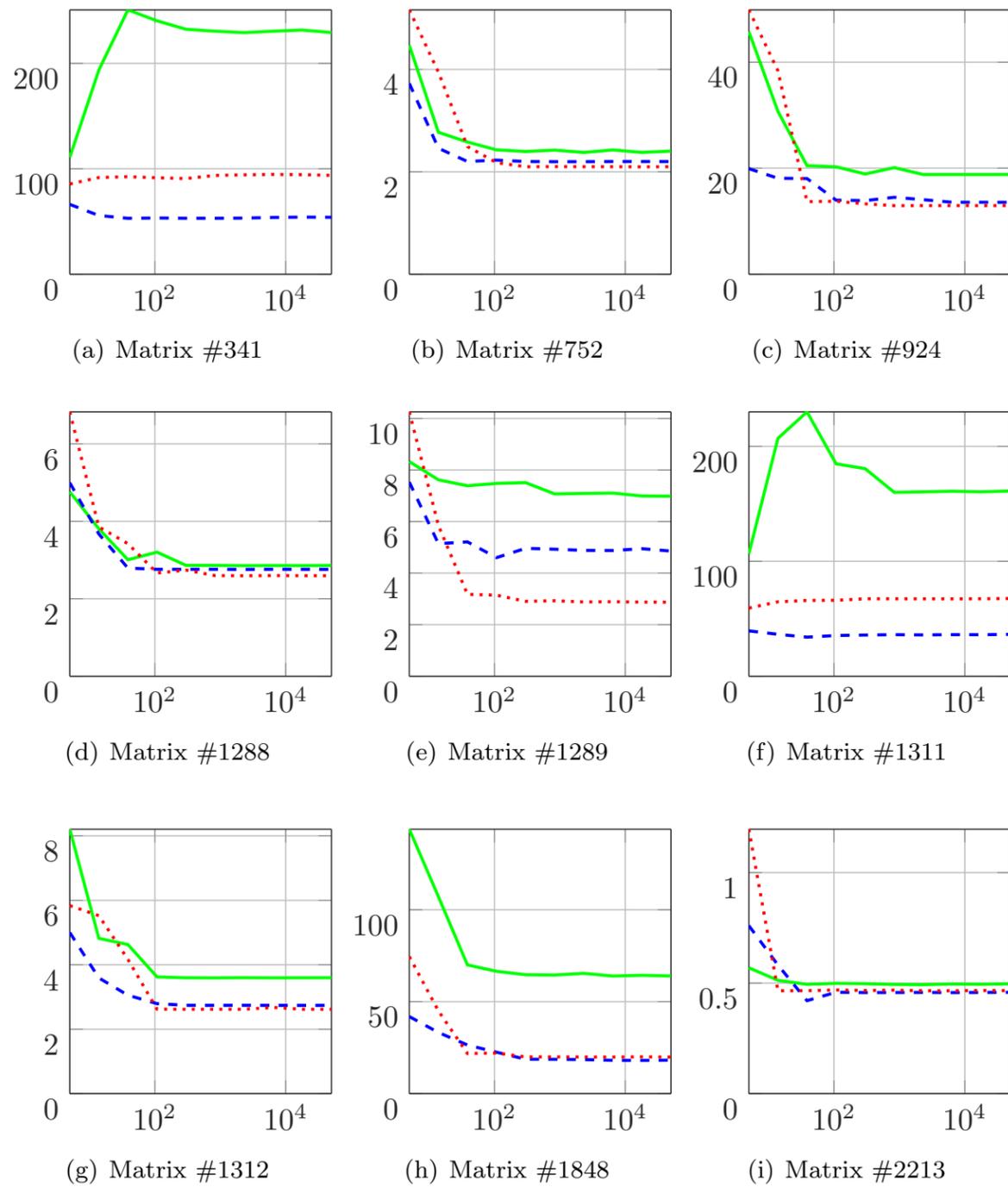
(a) Matrix #341 (b) Matrix #752 (c) Matrix #924

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

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

Figure 1: Execution time in seconds (y axis) of ONLINE-DETECTION (dotted), ABFT-DETEC-
TION (solid line) and ABFT-CORRECTION (dashed) with respect to the normalized MTBF
(mean time between failures), x-axis). The matrix number is in the subcaption.

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 only checksum computations need to be performed reliably.

In addition, we examine checkpointing techniques as a tool to improve the resilience of our ABFT PCG 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 errors 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 this combined approach to be interesting for other variants of the preconditioned conjugate gradient algorithm [35]. Triangular preconditioners seem to be particularly attracting, in that it looks possible to treat them by adapting the techniques described in this paper (Shantharam et al. [16] addressed the triangular case).

We discussed the adequacy of the proposed techniques in the single core and distributed memory settings, but did not address shared memory parallel systems with many-cores. In these latter systems, the task-level parallelism necessitates application of accuracy theorems (in the spirit of Theorem 3) to avoid false positives. We leave the resilience of task-parallel executions as a future work.

Acknowledgements

References

[1] M. Fasi, Y. Robert, B. Uçar, Combining Backward and Forward Recovery to Cope with Silent Errors in Iterative Solvers, Proceedings of the 16th Workshop on Parallel and Distributed Scientific and Engineering Computing 2015 (PDSEC 2015) (2015) 980–989.

[2] A. Moody, G. Bronevetsky, K. Mohror, B. de Supinski, Design, modeling, and evaluation of a scalable multi-level checkpointing system, in: High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for, 2010, pp. 1–11.

[3] F. Cappello, Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities, International Journal of High Performance Computing Applications 23 (3) (2009) 212–226.

[4] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, M. Snir, Toward exascale resilience, International Journal of High Performance Computing Applications 23 (4) (2009) 374–388.

[5] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, M. Snir, Toward exascale resilience: 2014 update, Supercomputing frontiers and innovations 1 (1), open Access, `http://superfri.org/superfri/article/view/14`.

[6] B. Schroeder, G. A. Gibson, Understanding failures in petascale computers, Journal of Physics: Conference Series 78 (012022).

[7] 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, ACM, 2013, pp. 167–176.

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

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

[10] A. Benoit, A. Cavelan, Y. Robert, 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.

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

[12] P. Du, A. Bouteiller, G. Bosilca, T. Herault, J. Dongarra, Algorithm-based fault tolerance for dense matrix factorizations, in: PPoPP, ACM, 2012, pp. 225–234.

[13] E. Yao, J. Zhang, M. Chen, G. Tan, N. Sun, Detection of soft errors in LU decomposition with partial pivoting using algorithm-based fault tolerance, International Journal of High Performance Computing Applications 29 (4) (2015) 422–436.

[14] D. Hakkarinen, P. Wu, Z. Chen, Fail-stop failure algorithm-based fault tolerance for cholesky decomposition, Parallel and Distributed Systems, IEEE Transactions on 26 (5) (2015) 1323–1335.

[15] P. Du, P. Luszczek, S. Tomov, J. Dongarra, Soft error resilient QR factorization for hybrid system with GPGPU, Journal of Computational Science 4 (6) (2013) 457 – 464, scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011.

[16] M. Shantharam, S. Srinivasmurthy, P. Raghavan, Fault Tolerant Preconditioned Conjugate Gradient for Sparse Linear System Solution, in: Proceedings of the 26th ACM International Conference on Supercomputing, ICS '12, ACM, 2012, pp. 69–78.

[17] M. Benzi, M. Tuma, A sparse approximate inverse preconditioner for nonsymmetric linear systems, SIAM Journal on Scientific Computing 19 (3) (1998) 968–994.

[18] K. Kaya, B. Uçar, U. V. Çatalyürek, Analysis of partitioning models and metrics in parallel sparse matrix-vector multiplication, in: Parallel Processing and Applied Mathematics (PPAM2014), Springer LNCS, Warsaw, Poland, 2014, pp. 174–184.

[19] G. Aupy, Y. Robert, F. Vivien, D. Zaidouni, Checkpointing algorithms and fault prediction, Journal of Parallel and Distributed Computing 74 (2) (2014) 2048–2064.

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

[21] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, 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.

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

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

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

[25] M. Hoemmen, M. A. Heroux, Fault-tolerant iterative methods via selective reliability, Tech. rep., Sandia Corporation (2011).

[26] M. Hoemmen, 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, Vol. 3, 2011, p. 9.

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

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

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

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

[31] G. Bronevetsky, B. de Supinski, Soft error vulnerability of iterative linear algebra methods, in: Proc. 22nd Int. Conf. on Supercomputing, ICS '08, ACM, 2008, pp. 155–164.

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

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

[34] J. Sloan, R. Kumar, 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, 2012, pp. 1–12.

[35] Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd Edition, SIAM Press, 2003.

[36] F. R. K. Chung, Spectral Graph Theory, American Mathematical Society, 1997.

[37] A. Edelman, Eigenvalues and condition numbers of random matrices, SIAM Journal on Matrix Analysis and Applications 9 (4) (1988) 543–560.

[38] Z. Chen, J. J. Dongarra, Condition numbers of Gaussian random matrices, SIAM Journal on Matrix Analysis and Applications 27 (3) (2005) 603–620.

[39] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, F. Vivien, Checkpointing strategies for parallel jobs, in: SC'2011, IEEE, 2011, pp. 1–11.

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

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

[42] M. Benzi, R. Kouhia, M. Tůma, Stabilized and block approximate inverse preconditioners for problems in solid and structural mechanics, Computer Methods in Applied Mechanics and Engineering 190 (49–50) (2001) 6533 – 6554.

[43] N. J. Higham, Accuracy and Stability of Numerical Algorithms, 2nd Edition, SIAM Press, 2002.

[44] M. Fasi, Y. Robert, B. Uçar, Combining Algorithm-based Fault Tolerance and Checkpointing for Iterative Solvers, Research Report RR-8675, INRIA (2015).

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

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

[47] M. Stoyanov, C. Webster, Quantifying the impact of single bit flips on floating point arithmetic, Tech. rep., Oak Ridge National Laboratory (2013).