

Combining backward and forward recovery to cope with silent errors in iterative solvers

Massimiliano Fasi, Yves Robert, Bora Uçar

► **To cite this version:**

Massimiliano Fasi, Yves Robert, Bora Uçar. Combining backward and forward recovery to cope with silent errors in iterative solvers. PDSEC2015, May 2015, Hyderabad, India. IEEE CPS, pp.980–989, IPDPSW 2015: 29th IEEE International Parallel & Distributed Processing Symposium Workshops 2015. <hal-01159679>

HAL Id: hal-01159679

<https://hal.inria.fr/hal-01159679>

Submitted on 3 Jun 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Combining backward and forward recovery to cope with silent errors in iterative solvers

Massimiliano Fasi^{*†}, Yves Robert^{*‡} and Bora Uçar^{*}

^{*}École Normale Supérieure de Lyon, CNRS & INRIA, France

Email: {yves.robert|bora.ucar}@ens-lyon.fr

[†]University of Bologna, Italy

Email: massimiliano.fasi@studio.unibo.it

[‡]University of Knoxville, USA

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 introduce an abstract 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.

Keywords

Fault-tolerance, Silent errors, Algorithm-based fault tolerance, Checkpointing, Sparse matrix-vector multiplication, Performance model.

1. Introduction

Silent errors (or silent data corruption) have become a significant concern in HPC environments [28]. 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 condition known to dramatically reduce 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 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 taken.

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 execute in reliable mode.

While verification mechanisms speed up the detection of silent errors, they are not able to provide correction, hence cannot 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 [9] for a variety of Krylov-based methods, including the widely used Conjugate Gradient (CG) algorithm. Chen [9] 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 [35] and Daly [10]. 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 [3].

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 [22] 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 [31]. 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 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 [9] (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.

All our discussion focuses on 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. Typically (a recent exposition of different algorithms can be found elsewhere [24]), the processors perform scalar multiply operations on the local matrix elements and the input vector elements, when all of the required vector elements are received from others. 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 error can indeed strike at a given processor during local scalar multiply operations. Performing error detection and correction locally imply 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 MTBF (Mean Time Between Failures) 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 [2].

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 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

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

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 [26], 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 [16]. Elliot et al. [14] 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. [25].

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 [23] and ABFT techniques (see below).

As already pointed out, most papers focus on a selective reliability setting [7], [20], [21], [30]. 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.

Iterative methods. Iterative methods offer a wide range of ad-hoc approaches. For instance, instead of duplicating the computation, Benson et al. suggest coupling a higher-order with a lower-order scheme for PDEs [4]. 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 [30]. Heroux and Hoemmen [17] design a fault-tolerant GMRES capable of converging despite silent errors. Bronevetsky and de Supinski [8] 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 [9], 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.

ABFT. The very first idea of algorithm-based fault tolerance for linear algebra kernels is given by Huang and Abraham [22]. 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 to 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. [31], who suggest a way to detect a single error in an SpMxV at the cost of a few additional dot products. Sloan et al. [32] 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 [5], [12].

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

3.1. CG and verification mechanisms

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

Chen’s stability tests [9] amount to checking the orthogonality of vectors \mathbf{p}_{i+1} and \mathbf{q} , at the price of computing $\frac{\mathbf{p}_{i+1}^\top \mathbf{q}}{\|\mathbf{p}_{i+1}\| \|\mathbf{q}_i\|}$, and to checking the residual at the price of an additional SpMxV operation $\mathbf{Ax}_i - \mathbf{b}$. 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 them for simplicity. That is 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. [31] 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 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 $Colid \in \mathbb{N}^{\text{nnz}(\mathbf{A})}$, $Val \in \mathbb{R}^{\text{nnz}(\mathbf{A})}$ and $Rowidx \in \mathbb{N}^{n+1}$ [29, Sec. 3.4].

Shantaram 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

$$c_j = \sum_{i=0}^n a_{i,j} \quad (1)$$

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 $\sum_{i=1}^n y_i$, $\mathbf{c}^\top \mathbf{x}$ and $\mathbf{c}^\top \mathbf{x}'$, and to compare their values. It can be shown [31] that in the case of no errors, these three quantities carry the same value, whereas if a single error strikes either the memory or the computation, one of them must differ from the other two. Nevertheless, this method requires \mathbf{A} to be strictly diagonally dominant, that seems to restrict too much the practical applicability of their 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

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

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 $\langle \cdot, \cdot \rangle : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$ defined by $\langle \mathbf{u}, \mathbf{v} \rangle \mapsto \mathbf{u}^\top \mathbf{v}$. 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 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 $\mathcal{B} = \{\mathbf{b}_1, \dots, \mathbf{b}_n\}$ of \mathbb{R}^n – the rows of the input matrix. Each one of these n vectors is perpendicular to a hyperplane h_i of \mathbb{R}^n , and \mathbf{u} does not verify the condition

$$\langle \mathbf{w}, \mathbf{b}_i \rangle \neq 0, \quad (2)$$

for any i , if and only if it lies on h_i . As the Lebesgue measure in \mathbb{R}^n of an hyperplane of \mathbb{R}^n itself is zero, the union of these hyperplanes is measurable with $m_n(\bigcup_{i=1}^n h_i) = 0$, where m_n denotes the Lebesgue measure of \mathbb{R}^n . Therefore, the probability that a vector \mathbf{w} randomly picked in \mathbb{R}^n does not satisfy condition (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 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 $(1, \dots, 1)^\top$ 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 $\mathbf{w} = (1, \dots, 1)^\top$ 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 [34].

In Algorithm 2, we propose an ABFT SpMxV method that uses weighted checksums and does not require the matrix to be strictly diagonally dominant. The idea is to compute the checksum vector and then shift it by adding to all of its entries a constant value chosen so that all of the elements of the new vector are different from zero. We give the result in Theorem 1 for the simpler case of single error detection without correction, in which case Algorithm 2 has $\mathbf{W} = (1, \dots, 1)^\top$ at line 1 and raises an error at line 26 if the tests at line 23 are not passed. The cases of multiple error detection and single error correction are proved in the accompanying technical report [15, Section 3.2].

Theorem 1 (Correctness of Algorithm 2 for single error detection). *Let $\mathbf{A} \in \mathbb{R}^{n \times n}$ be a square matrix, let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$*

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

Input: $\mathbf{A} \in \mathbb{R}^{n \times n}$ (as $Val \in \mathbb{R}^{\text{nnz}(\mathbf{A})}$, $Colid \in \mathbb{N}^{\text{nnz}(\mathbf{A})}$, $Rowidx \in \mathbb{R}^n$), $\mathbf{x} \in \mathbb{R}^n$

Output: $\mathbf{y} = \mathbf{A}\mathbf{x}$, correction of single error or detection of double error

```

1: global  $\mathbf{W}^\top \leftarrow \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & 2 & \dots & n \end{bmatrix} \in \mathbb{R}^{2 \times n}$ ;
2: global  $\mathbf{W}^\top \leftarrow \begin{bmatrix} \mathbf{W}^\top \\ \mathbf{1}^\top \end{bmatrix} \in \mathbb{R}^{2 \times n+1}$ ;
3:  $\mathbf{x}' \leftarrow \mathbf{x}$ ;
4:  $[\mathbf{C}, \mathbf{M}, \mathbf{c}_r, \mathbf{c}_x] = \text{COMPUTECHECKSUMS}(Val, Colid, Rowidx)$ ;
5: return SPMXV( $Val, Colid, Rowidx, \mathbf{x}, \mathbf{x}', \mathbf{M}, \mathbf{c}_r, \mathbf{c}_x$ );

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

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

```

be the input and output vector respectively, and let $\mathbf{x}' = \mathbf{x}$. Let us assume that the algorithm performs the computation

$$\tilde{\mathbf{y}} \leftarrow \tilde{\mathbf{A}}\tilde{\mathbf{x}}, \quad (3)$$

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

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

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

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

The proof of this theorem is technical and is available in the extended version of this paper [15, Theorem 1].

The function COMPUTECHECKSUM in Algorithm 2 requires just the knowledge of the matrix. Hence in the common scenario of many SpMxVs with the same matrix, it is enough

to invoke it once to protect several matrix-vector multiplications. This observation will be crucial when talking about the performances of the checksumming techniques.

The longer version of this paper [15, Section 3.2] discusses extensions to $k \geq 2$ errors, where the following are detailed. The method just described can be extended to detect up to a total of k errors anywhere in the computation, in the representation of \mathbf{A} , or in the vector \mathbf{x} . Building up the necessary structures requires $\mathcal{O}(k \text{nnz}(\mathbf{A}))$ time, and the overhead per SpMxV is $\mathcal{O}(kn)$. The particular case of $k = 2$ is also detailed [15, Section 3.2], where a result similar to that in Theorem 1 is shown.

We now discuss error correction. If at least one of the tests at line 23 of Algorithm 2 fails, the algorithm invokes CORRECTERRORS in order to determine whether just one error struck either the computation or the memory and, in case, correct it. Whenever a single error is detected, disregarding its location (i.e., computation or memory) 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 $Rowidx$, we compute the ratio d of the second component of \mathbf{d}_r to the first one, and check whether its distance from an integer is smaller than a certain threshold parameter ε . If it is so, the algorithm concludes that the d -th element of $Rowidx$ is faulty, performs the correction by subtracting the first component of \mathbf{d}_r to $Rowidx_d$, and recomputes y_d and y_{d-1} , if the error in $Rowidx_d$ is a decrement; or y_{d+1} if it was an increment. Otherwise, it just emits an error.

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

- If $z_{\tilde{\mathbf{C}}} = 0$, then the error is in the computation of y_d , and can be corrected by simply recomputing this value.
- If $z_{\tilde{\mathbf{C}}} = 1$, then the error concerns an element of Val . Let us call f the index of the non-zero column of $\tilde{\mathbf{C}}$. The algorithm finds the element of Val corresponding to the entry at row d and column f of A and corrects it by using the column checksums much like as described for $Rowidx$. Afterwards, y_d is recomputed to fix the result.
- If $z_{\tilde{\mathbf{C}}} = 2$, 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 $z_{\tilde{c}} > 2$, then errors can be detected but not corrected, and an error is emitted.

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

Finally, note that double errors could be shadowed when using Algorithm 2, but the probability of such an event is negligible. Still, there exists an improved version which avoids this issue by adding a third checksum [15, Section 3.2].

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

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 $\mathbb{E}(s, T)$ 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 [6], we derive the following recursive equation to compute the expected completion time of a single frame:

$$\mathbb{E}(s, T) = q^s(s(T + T_{verif})) + T_{cp} + (1 - q^s)(\mathbb{E}(T_{lost}) + T_{rec} + \mathbb{E}(s, T)). \quad (4)$$

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 $1 \leq i \leq s$, 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

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

We have $\sum_{i=1}^s f_i = \frac{(1-q)h(q)}{1-q^s}$ where $h(q) = 1 + 2q + 3q^2 + \dots + sq^{s-1}$. If $m(q) = q + q^2 + \dots + q^s = \frac{1-q^{s+1}}{1-q} - 1$, 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 (4), 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

$$\mathbb{E}(s, T) = T_{cp} + (q^{-s} - 1)T_{rec} + (T + T_{verif}) \frac{1 - q^s}{q^s(1 - q)}. \quad (5)$$

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

$$s = \underset{s \geq 1}{\operatorname{argmin}} \left(\frac{\mathbb{E}(s, T)}{sT} \right). \quad (6)$$

The minimization is complicated and should be conducted numerically (because T , the size of a chunk, is still unknown). Note that a dynamic programming algorithm to compute the optimal value of T and s is available in [3].

4.2. Instantiation to CG

For each of the three methods, ONLINE-DETECTION, ABFT-DETECTION and ABFT-CORRECTION, we instantiate the previous model and discuss how to solve Equation (6).

4.2.1. ONLINE-DETECTION. For Chen's method [9], 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 λ_a be the rate of arithmetic errors, and λ_m be the rate of memory errors. For the latter, we have $\lambda_m = M\lambda_{word}$ if the data memory consists of M words, each susceptible to be corrupted with rate λ_{word} . Altogether, since the two error sources are independent, they have a cumulated rate of $\lambda = \lambda_a + \lambda_m$, and the success probability for a chunk is $q = e^{-\lambda T}$.

Plugging these values in Equation (6) gives an optimisation formula very similar to that of Chen [9, 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 λ , where $\lambda = \lambda_a + \lambda_m$ as for ONLINE-DETECTION and ABFT-DETECTION. The probability of exactly k errors in time T is $\frac{(\lambda T)^k}{k!} e^{-\lambda T}$ [27], hence the probability of no error is $e^{-\lambda T}$ and the probability of exactly one error is $\lambda T e^{-\lambda T}$, so that $q = e^{-\lambda T} + \lambda T e^{-\lambda T}$.

5. Experiments

5.1. Setup

There are two different sources of advantages in combining ABFT and checkpointing. First, 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. 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 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 [11], 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 λ , as detailed in Section 4.2. These bit flips can strike either the matrix (the elements of Val , $Colid$ and $Rowidx$), or any entry of the CG vectors \mathbf{r}_i , \mathbf{q} , \mathbf{p}_i or \mathbf{x}_i . We choose not to inject errors striking the computation 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 [30]. 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 $\alpha \in (0, 1)$. 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 (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). A detailed discussion of the implementation of this mixed strategy can be found in the technical report [15, Section 4.2].

Implementing Algorithm 2, in particular the tests at line 23, poses a challenge. The comparison $\mathbf{d}_r = \mathbf{c}_r - \mathbf{s}_r$ is between two integers, and can be correctly evaluated by any programming language using the equality check. 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 [18, Section 2.2] so that the errors caught by the tests at line 23 of Algorithm 2 are really 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 2 (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 [18, Chapter 4], it holds that*

$$|fl((\mathbf{c}^T \mathbf{A}) \mathbf{x}) - fl(\mathbf{c}^T (\mathbf{A} \mathbf{x}))| \leq 2 \gamma_{2n} |\mathbf{c}^T| |\mathbf{A}| |\mathbf{x}|. \quad (7)$$

We refer the reader to the technical report for the proof [15, 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 (7) can be safely replaced with \mathbf{c} itself. It is also clear that these bounds are not computable, since $\mathbf{c}^T |\mathbf{A}| |\mathbf{x}|$ is not, in general, a floating point number. This problem can be alleviated by overestimating the bound by means of matrix and vector norms.

Recalling that it can be shown [19, Section B.7] that

$$\|\mathbf{A}\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^n |a_{i,j}|, \quad (8)$$

we can upper bound the right hand side in (7) so that

$$|fl((\mathbf{c}^T \mathbf{A}) \mathbf{x}) - fl(\mathbf{c}^T (\mathbf{A} \mathbf{x}))| \leq 2 \gamma_{2n} n \|\mathbf{c}^T\|_\infty \|\mathbf{A}\|_1 \|\mathbf{x}\|_\infty. \quad (9)$$

Though the right hand side of (9) is not exactly computable in floating point arithmetic, it requires an amount of operations dramatically smaller than the one in (7), just a few sums for the norm of \mathbf{A} . As this norm is usually computed using the identity in (8), any kind of summation yields a relative error of at most $n' \mathbf{u}$ [18, Section 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, the right hand side in (9) does not depend on \mathbf{x} , it can be computed just once for a given matrix and weight vector.

Clearly, using (9) 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. [13], [33] 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 $\lambda = \frac{1}{16M}$ 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 checkpointing interval s_i^* , that achieves the shortest execution time $E_t(s_i^*)$, and the checkpointing interval \tilde{s}_i that is the best stepsize according to our method, along with its execution time $E_t(\tilde{s}_i)$. Finally, we evaluate the performance of our guess by means of the quantity

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

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 \tilde{s}_i and s_i^* are close, even though the time lost 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 λ .

We also compare the execution time of the three algorithms to empirically assess 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 4.1 for ABFT-DETECTION and ABFT-CORRECTION, and by Chen [9, Eq. 10] for ONLINE-DETECTION. In terms of the 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, though being slightly faster than ABFT-CORRECTION just for the last matrix. For lower fault rates, however, ONLINE-DETECTION seems to be the slowest one: this is plausibly 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 λ .

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.

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

id	n	density	\tilde{s}_1	$E_t(\tilde{s}_1)$	s_1^*	$E_t(s_1^*)$	l_1	\tilde{s}_2	$E_t(\tilde{s}_2)$	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 \tilde{s}_i and s_i^* represent the best checkpointing interval according to our model and to our simulations respectively, whereas $E_t(\tilde{s}_i)$ and $E_t(s_i^*)$ stand for the execution time of the algorithm using these checkpointing intervals.

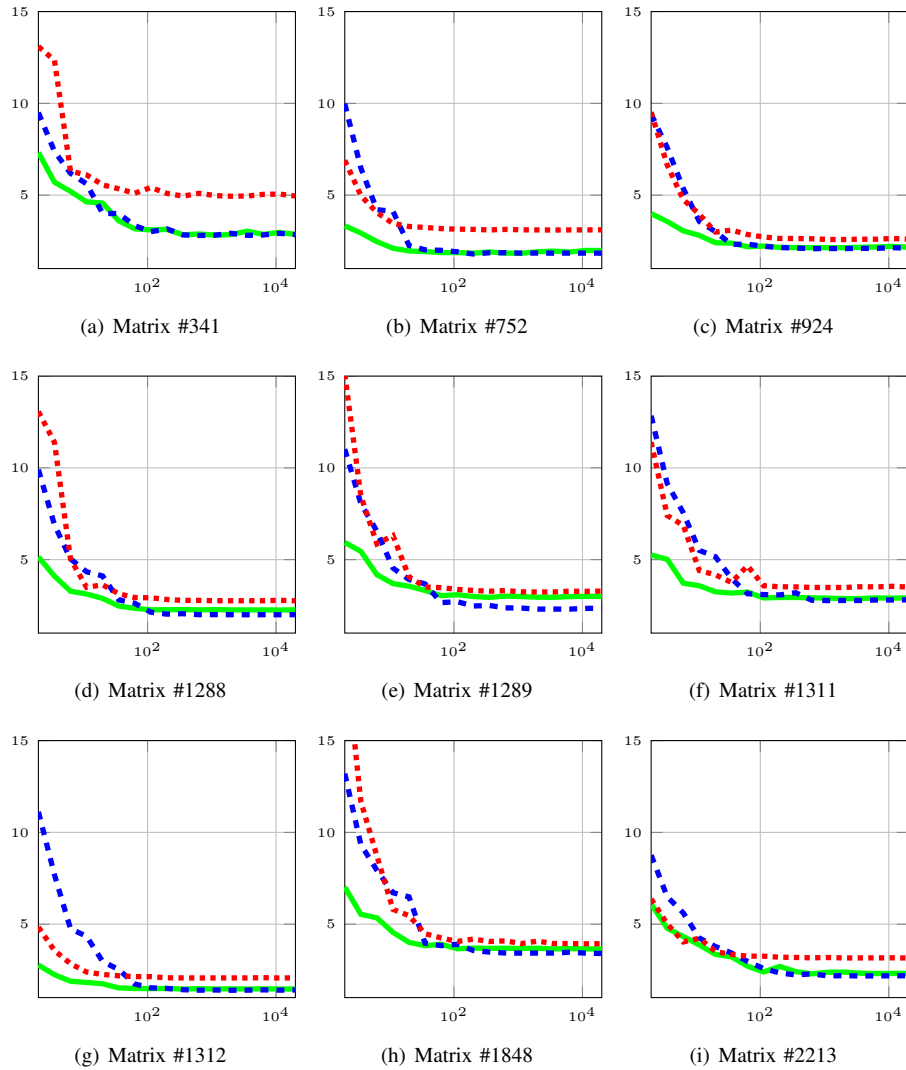


Figure 1. Execution time in seconds (y axis) of ONLINE-DETECTION (dotted), ABFT-DETECTION (dashed) and ABFT-CORRECTION (solid line) with respect to the normalized MTBF (x -axis). The matrix number is in the subcaption.

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 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 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 useful even when dealing with the preconditioned CG Algorithm [29], 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, since it should be possible to treat them by adapting the techniques described in this paper (Shantaram et al. [31] 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 this work.

Acknowledgements

M. Fasi was partly supported by a grant of the Collegio Superiore of Bologna. Y. Robert and B. Uçar were partly supported by the French Research Agency (ANR) through the Rescue and SOLHAR (ANR MONU-13-0007) projects. Y. Robert is with Institut Universitaire de France.

References

- [1] C. Anfinson and F. Luk, "A Linear Algebraic Model of Algorithm-Based Fault Tolerance," *IEEE Trans. Computers*, vol. 37, no. 12, pp. 1599–1604, 1988.
- [2] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni, "Checkpointing algorithms and fault prediction," *Journal of Parallel and Distributed Computing*, vol. 74, no. 2, pp. 2048–2064, 2014.
- [3] 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.
- [4] A. R. Benson, S. Schmit, and R. Schreiber, "Silent error detection in numerical time-stepping schemes," *CoRR*, vol. abs/1312.2674, 2013.
- [5] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based fault tolerance applied to high performance computing," *J. Parallel and Distributed Computing*, vol. 69, no. 4, pp. 410–416, 2009.
- [6] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien, "Checkpointing strategies for parallel jobs," in *SC'2011*. IEEE, 2011, pp. 1–11.
- [7] P. G. Bridges, K. B. Ferreira, M. A. Heroux, and M. Hoemmen, "Fault-tolerant linear solvers via selective reliability," preprint, 2012.
- [8] G. Bronevetsky and B. de Supinski, "Soft error vulnerability of iterative linear algebra methods," in *Proc. 22nd Int. Conf. on Supercomputing*, ser. ICS '08. ACM, 2008, pp. 155–164.
- [9] 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*, ser. PPOPP '13. ACM, 2013, pp. 167–176.
- [10] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *FGCS*, vol. 22, no. 3, pp. 303–312, 2004.
- [11] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, 2011.
- [12] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," in *PPOPP*. ACM, 2012, pp. 225–234.
- [13] J. Elliott, F. Mueller, M. Stoyanov, and C. Webster, "Quantifying the impact of single bit flips on floating point arithmetic," preprint, 2013.
- [14] 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.
- [15] M. Fasi, Y. Robert, and B. Uçar, "Combining Algorithm-based Fault Tolerance and Checkpointing for Iterative Solvers," INRIA, Research Report RR-8675, 2015. [Online]. Available: <https://hal.inria.fr/hal-01111707>
- [16] 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.*, ser. SC '12. IEEE Computer Society Press, 2012.
- [17] M. Heroux and M. Hoemmen, "Fault-tolerant iterative methods via selective reliability," Sandia National Laboratories, Research report SAND2011-3915 C, 2011.
- [18] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. SIAM Press, 2002.
- [19] —, *Functions of Matrices: Theory and Computation*. SIAM Press, 2008.
- [20] M. Hoemmen and M. A. Heroux, "Fault-tolerant iterative methods via selective reliability," Sandia Corporation, Tech. Rep., 2011.
- [21] —, "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.
- [22] K.-H. Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *Computers, IEEE Transactions on*, vol. C-33, no. 6, pp. 518–528, 1984.
- [23] 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*, vol. 40, no. 1, pp. 111–122, 2012.
- [24] K. Kaya, B. Uçar, and U. V. Çatalyürek, "Analysis of partitioning models and metrics in parallel sparse matrix-vector multiplication," in *Parallel Processing and Applied Mathematics (PPAM2014)*. Warsaw, Poland: Springer LNCS, 2014, pp. 174–184.
- [25] 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.
- [26] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM J. Res. Dev.*, vol. 6, no. 2, pp. 200–209, 1962.
- [27] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.
- [28] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proc. of the ACM/IEEE SC Conf.*, 2010, pp. 1–11.
- [29] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. SIAM Press, 2003.
- [30] P. Sao and R. Vuduc, "Self-stabilizing iterative solvers," in *Proc. SCALA '13*. ACM, 2013.
- [31] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Fault tolerant preconditioned conjugate gradient for sparse linear system solution," in *Proc. ICS '12*. ACM, 2012.
- [32] 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*, 2012, pp. 1–12.
- [33] M. Stoyanov and C. Webster, "Quantifying the impact of single bit flips on floating point arithmetic," Oak Ridge National Laboratory, Tech. Rep., 2013.
- [34] E. W. Weisstein, "Laplacian matrix," From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/LaplacianMatrix.html>, last accessed October 2014.
- [35] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Comm. of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.