New-Sum: A Novel Online ABFT Scheme For General Iterative Methods

Dingwen Tao
University of California, Riverside
dtao001@cs.ucr.edu

Shuaiwen Leon Song
Pacific Northwest National Laboratory
Shuaiwen.Song@pnnl.gov

Sriram Krishnamoorthy
Pacific Northwest National Laboratory
sriram@pnnl.gov

Panruo Wu
University of California, Riverside
pwu011@cs.ucr.edu

Xin Liang
University of California, Riverside
xlian007@cs.ucr.edu

Eddy Z. Zhang
Rutgers University
eddy.zhengzhang@cs.rutgers.edu

Darren Kerbyson
Pacific Northwest National Laboratory
Darren.Kerbyson@pnnl.gov

Zizhong Chen
University of California, Riverside
chen@cs.ucr.edu

ABSTRACT

Emerging high-performance computing platforms, with large component counts and lower power margins, are anticipated to be more susceptible to soft errors in both logic circuits and memory subsystems. We present an online algorithm-based fault tolerance (ABFT) approach to efficiently detect and recover soft errors for general iterative methods. We design a novel checksum-based encoding scheme for matrix-vector multiplication that is resilient to both arithmetic and memory errors. Our design decouples the checksum updating process from the actual computation, and allows adaptive checksum overhead control. Building on this new encoding mechanism, we propose two online ABFT designs that can effectively recover from errors when combined with a checkpoint/rollback scheme. These designs are capable of addressing scenarios under different error rates. Our ABFT approaches apply to a wide range of iterative solvers that primarily rely on matrix-vector multiplication and vector linear operations. We evaluate our designs through comprehensive analytical and empirical analysis. Experimental evaluation on the Stampede supercomputer demonstrates the low performance overheads incurred by our two ABFT schemes for preconditioned CG (0.4% and 2.2%) and preconditioned BiCGSTAB (1.0% and 4.0%) for the largest SPD matrix from UFL Sparse Matrix Collection. The evaluation also demonstrates the flexibility and effectiveness of our proposed designs for detecting and recovering various types of soft errors in general iterative methods.

HPDC'16, May 31-June 04, 2016, Kyoto, Japan

DOI: http://dx.doi.org/10.1145/2907294.2907306

Keywords

Algorithm-Based Fault Tolerance (ABFT); Resilience; Iterative Methods; Online Error Detection; Silent Data Corruption (SDC); Checksum; Checkpoint; Rollback Recovery

1. INTRODUCTION

Supercomputers are being built with an increasing number of complex components, each of which has growing on-chip transistor density [20]. Together with a renewed emphasis on limiting power and energy consumption, this is anticipated to result in these systems being increasingly susceptible to soft errors [5, 15], errors that do not lead to noticeable system crashes, but to silent data corruption (SDC). This phenomenon has already been observed on several real-world leadership-class supercomputers [5, 15].

Algorithm-based fault tolerance (ABFT) is an approach to detect and possibly correct errors at a lower cost than double- or triple-modular redundancy. These approaches exploit the characteristics of an algorithm to encode a small amount of redundancy into the computation. This redundancy is later used to detect and correct errors. In this paper, we present an ABFT approach to tolerate soft errors in general iterative methods. These methods, used in a wide variety of applications [1], primarily consist of matrix-vector multiplication (MVM) and vector linear operations (VLOs).

Novel checksum-encoding scheme. The effectiveness and efficiency of an ABFT scheme depends on the checksum-encoding mechanism employed and its coverage in terms of number and type of errors detected. Much existing work on ABFT strategies is built on a checksum-encoding scheme designed for matrix-matrix multiplication [11]. While this scheme has been extended to cover a wide range of related algorithms [18, 19, 23], we show that it is not sufficient to construct ABFT schemes for matrix-vector multiplication (MVM) due to its inability to detect soft errors if the input vector is corrupted. We present a novel checksum-encoding scheme that can tolerate soft errors in both logic circuits (e.g., arithmetic operations) and the memory subsystems (e.g., memory, cache and register bit-flips). Our new scheme

separates the checksums from their corresponding encoded matrix and vectors, enabling checksum updates across multiple operations to improve overall performance. We show that our checksum scheme also can be used in the context of preconditioners employed in iterative methods.

Flexible detection latency. Error detection latency refers to the latency between the manifestation of an error and its detection. In general, longer error detection latencies enable reduced detection costs but might lead to increased recovery overhead due to the error corrupting a larger fraction of the application state. ABFT schemes for iterative methods often require error detection after each matrix-vector multiplication (MVM) or each iteration. We show that our checksum scheme supports eager (immediate) and lazy (after several iterations) error detection by detecting an arbitrary number of errors across multiple operations.

Two-level ABFT. ABFT schemes often employ redundancy proportional to the number of errors to be detected and corrected. This requires careful consideration of anticipated error rates and performance penalties proportional to the number of errors that need to be corrected. Alternatively, checkpoint-rollback incurs significant recovery penalties to recover even from one error, say impacting one arithmetic operation. We present a two-level ABFT algorithm that combines the best aspects of both strategies. In the most compute-intensive component of iterative methods, the matrix-vector multiplication (MVM), we employ a low-cost inner-level recovery scheme to efficiently correct one error and detect multiple errors. When multiple errors are detected, the algorithm resorts to immediate rollback. Multiple errors in an MVM as well as errors in the VLOs are protected by an outer-level rollback strategy that is invoked every few iterations. This two-level approach protects the most compute-intensive parts efficiently while ensuring sufficient coverage for other parts of the computation.

Contributions. The proposed ABFT schemes in this paper are applicable to all the iterative methods constructed from matrix-vector multiplication and vector linear operations. In particular, all the Krylov solvers, including Richardson, Chebyshev, CG variants, quasi-minimal residual (QMR), conjugate residuals (CR), generalized conjugate residuals (GCR), variations of GMRES, minimum residual (MINRES), SYMMLQ, and LSQR, can be protected using the presented ABFT approaches.

We compare our online ABFT designs with the state-of-the-art techniques and demonstrate benefits in terms of coverage for different types of soft errors, generality for addressing iterative methods, and overhead introduced. We also evaluate the overall performance of our two schemes (i.e., *basic online ABFT* and *two-level online ABFT*) under various error scenarios on a leadership-class supercomputer. Experimental results show that our proposed designs encounter trivial overhead for both erroneous (single error or multiple errors) and error-free execution. Additionally, we compare the two schemes through theoretical and empirical analysis, demonstrating the scenario under which each scheme should be applied to achieve the better overall performance.

The primary contributions of this paper are:

- A novel checksum encoding scheme for matrix-vector multiplication and preconditioners, separating the checksums from their corresponding encoded matrix and vectors, leading to schemes that compute the output checksum directly from the inputs' checksums;
- Proof that the new checksum scheme can detect soft errors in both the logic circuits and the memory subsystem;
- An ABFT scheme for iterative methods that allows the errors to be detected eagerly or lazily;
- A technique to efficiently correct one error and detect the presence of multiple errors in a vector;
- Two online ABFT designs based on the new encoding mechanism;
- Detailed theoretical and empirical comparison between the proposed designs and state-of-the-art approaches.

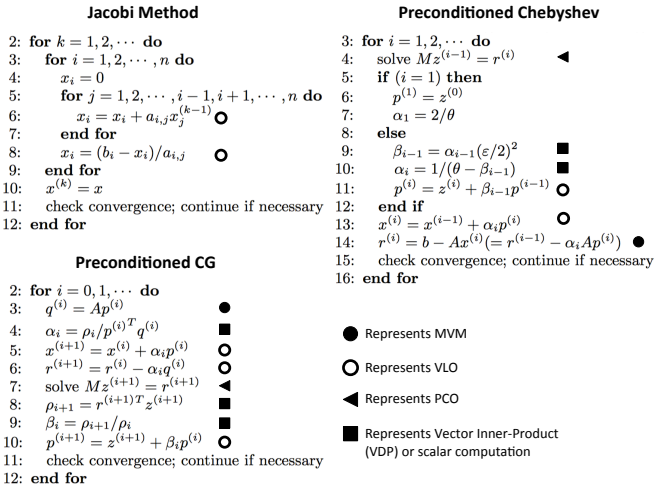

Figure 1: Main loops of several representative iterative methods: Jacobi, preconditioned conjugate gradient (PCG), preconditioned Chebyshev.

2. ALGORITHM-BASED FAULT TOLERANCE FOR ITERATIVE METHODS

Iterative methods are widely used for solving systems of equations or computing eigenvalues of large sparse matrices. The key feature of iterative methods is the use of matrix-vector multiplication (MVM) to iteratively compute approximations to the solution vector until desired accuracy is achieved. Figure 1 shows three representative iterative methods: Jacobi, preconditioned CG, and preconditioned Chebyshev. As illustrated, iterative methods consist of a few key operations: matrix-vector multiplications (MVM), vector linear-operations (VLOs), and solving preconditioned systems (PCO). Among them, MVM and PCO consume the largest fraction of the total computation time, making them particularly vulnerable to soft errors.

Algorithm-based fault tolerance (ABFT) techniques exploit specific algorithmic properties of a given computation to detect and possibly locate/correct errors. More commonly, ABFT techniques for matrix computations augment the input matrices with a *checksum* computed from the rows or columns of the matrices. It is then shown that performing a matrix operation on this augmented matrix automatically computes the checksum for the output matrix as part of the computation, as shown in Figure 2(a). Any error in the computation will result in the encoding relationship between the

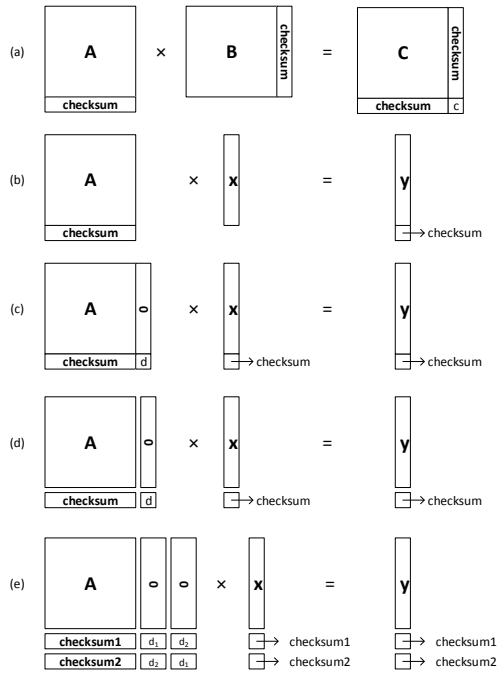

Figure 2: Checksum encoding mechanisms discussed in this paper: (a) Traditional checksum; (b) Traditional checksum applied to matrix-vector multiplication; (c) Our proposed checksum encoding; (d) Separation scheme of our new checksum; (e) Multiple-checksums encoding for our new checksum.

output matrix and its checksum being violated. Variants of this scheme have been designed to verify the encoding relationship online—at intermediate steps within the execution rather than at the end.

The notion of ABFT was introduced by Huang and Abraham [11] in the form of a checksum-based approach to verify matrix-matrix multiplication and LU decomposition. Several subsequent ABFT techniques employ the same encoding strategy [16, 18, 19, 23] illustrated for matrix $\boldsymbol{A}$ as follows:

$$\boldsymbol{A} \xrightarrow{encode} \boldsymbol{A}^* := \begin{bmatrix} \boldsymbol{A} \\ \boldsymbol{c^T A} \end{bmatrix}$$

where $\boldsymbol{c}$ is a predefined vector with all non-zero entries and $\boldsymbol{c^T A}$ is matrix $\boldsymbol{A}$'s checksum. A matrix-vector multiplication $\boldsymbol{y} = \boldsymbol{Ax}$ is replaced with a multiplication of the encoded operands, $\boldsymbol{y}^* = \boldsymbol{A}^*\boldsymbol{x}$, shown in Figure 2(b). We use $checksum(\boldsymbol{y})$ to denote the checksum computed in $\boldsymbol{y}^*$ as part of the operation. In this example, $checksum(\boldsymbol{y})$ is the last entry of the vector $\boldsymbol{y}^*$. In the absence of faults, $checksum(\boldsymbol{y}) = \boldsymbol{c^T y}$[1]. This checksum relationship can be used to detect errors in the computation of $\boldsymbol{y}$. For example, consider an error in an arithmetic operation resulting in $\boldsymbol{y'}^*$ that is not equal to $\boldsymbol{y}^*$. In this case, it can be shown that $checksum(\boldsymbol{y'}) \neq \boldsymbol{c^T y}$.

This encoding was designed for matrix-matrix multiplication and has some limitations when applied to iterative methods. For example, consider an error in $\boldsymbol{x}$ before the operation, resulting in an erroneous vector $\boldsymbol{x'}$. The encoded matrix-vector multiplication $\boldsymbol{y} = \boldsymbol{Ax}$ becomes:

$$\boldsymbol{y'}^* = \boldsymbol{A}^*\boldsymbol{x'} = \begin{pmatrix} \boldsymbol{A} \\ \boldsymbol{c^T A} \end{pmatrix} \boldsymbol{x'} = \begin{pmatrix} \boldsymbol{Ax'} \\ \boldsymbol{c^T Ax'} \end{pmatrix}$$

We observe that the checksum relationship for $\boldsymbol{y}$ holds even in the presence of an error in $\boldsymbol{x}$. In the absence of additional protection, this can lead to silent data corruption, making the encoding scheme unusable in detecting faults in the input vector. In other words, under this encoding scheme, the output vector's checksum relationship cannot be used to identify all soft errors.

Dealing with cache errors. Although adding additional checksum verification cost for $\boldsymbol{x}$ may capture the erroneous output vector sometimes, a more insidious case is a cache error in $\boldsymbol{x}$. Consider an error that affects a value of $\boldsymbol{x}$ in cache (e.g., a cache bit-flip while memory still holds the correct value). This erroneous value then resides in cache for the duration of the calculation, corrupting the computation of both the matrix-vector multiplication and the associated checksums. Given that the entire calculation consistently used the incorrect input value, the output checksums will be consistent, which makes verifying output checksum useless. Now if we add extra cost to verify $\boldsymbol{x}$, it may still not detect the error because cache lines can be evicted and the erroneous value $\boldsymbol{x}$ is replaced by the correct value loaded from the main memory. This error will now escape detection. Manifestation of this error depends on various factors such as compiler optimizations that reorder instructions, cache replacement and eviction policies, hardware cache configurations, etc. Tolerating faults in cache or registers in this scheme requires verifying every access to the vectors, which could be much more expensive. Because matrix-vector multiplication is invoked in every iteration of the iterative methods, and the traditional encoding scheme may lead to undetected errors, these errors need to be effectively detected every iteration.

Protecting preconditioners. Another limitation of the preceding checksum scheme is its inability to deal with preconditioned systems (PCOs) used to accelerate convergence in iterative solvers. For example, when solving preconditioned system $\boldsymbol{Mz} = \boldsymbol{r}$ in CG, where matrix $\boldsymbol{M}$ and vector $\boldsymbol{r}$ are input operands, the above encoding scheme cannot compute the checksum for $\boldsymbol{z}$ as part of the encoded computation.

State-of-the-art online ABFT schemes. The two most related state-of-the-art online ABFT schemes are from Chen [6] and Sloan et al. [19]. Chen proposed to exploit the vectors' orthogonal relationship to detect soft errors for *a subset of algorithms* in Krylov subspace methods. Every several iterations, the algorithm will check if the orthogonality relationship or the residual relationship $\boldsymbol{r}^{(i+1)} = \boldsymbol{b} - \boldsymbol{Ax}^{(i+1)}$ is valid. If either relationship is broken, execution is rolled back to the nearest checkpoint. Although good in terms of coverage for many operations (MVM, VLO, PCO, vector dot product, etc), this method has several limitations. First, it is not general enough to cover all iterative methods because some of them do not have orthogonal relationship of vectors, e.g., Jacobi and Chebyshev methods shown in Figure 1. Even if there are orthogonality relations between vectors, it still cannot detect soft errors that do not propagate to these vectors. Checking the residual requires an expensive MVM operation. This higher error detection overhead necessitates less frequent error checking, leading to a higher rollback recovery cost when errors are detected.

[1]This holds subject to the inexactness of floating point arithmetic, which we account for in the overall algorithm.

Sloan et al. [19] apply the traditional checksum discussed above to identify only arithmetic errors in the MVM operation (detection), and then use binary search to locate and partially correct the erroneous element(s) in the output vector (recovery). However, several issues can occur when this approach is used in the context of iterative methods. First, they focus on MVM and assume that no soft errors occur in other operations such as VLOs and PCOs. Second, they assume the input vector to the MVM is correct. As demonstrated previously, the traditional checksum cannot detect the error(s) by verifying the output vector's checksum relationship if the input vector is corrupted (e.g., memory bit-flips before MVM or arithmetic errors carried from the previous iteration). If the detection technique fails, all the benefits from the recovery scheme disappear. Third, to avoid such undetectable propagation of errors, Sloan's method needs to conduct expensive checksum verification for error detection every iteration, and then apply binary search to locate and correct the errors. This might be only beneficial under high error rates. While soft errors are more likely to occur in future systems, errors affecting the computation every few iterations is not a common or practically anticipated scenario. We present theoretical and empirical evaluation of Sloan's approach in Sections 6.2 and 6.3.

These issues motivate the design of new online ABFTs scheme that can be applied to general iterative methods to effectively tolerate various types of soft errors.

3. ERROR MODEL

We focus on errors affecting matrices and vectors employed in iterative methods. Specifically, we focus on errors in matrix-vector multiplication (MVM), vector linear operations (VLOs)—addition, scaling, assignment, etc.—and solving preconditioned systems (PCOs). We assume other low-computation operations (e.g., scalar operations, vector dot-product, etc.) not amenable to general ABFT checksum-encoding are protected using other schemes (e.g., duplicated execution or definition-use checksums [22]).

We consider errors in arithmetic operations or values used in these operations. These correspond to soft errors in the ALU or the memory subsystem. We consider errors in any part of the memory subsystem—main memory, caches, registers, etc.—that can affect the result of multiple (but not necessarily all) arithmetic operations.

We only consider errors that affect the data in the matrices and vectors used in the iterative method. As is the case with other ABFT schemes, we assume that errors do not affect scalar variables, control flow, program stack, etc. We model an error as a random additive contribution $\boldsymbol{e}$ to a value. For example, an error in $\boldsymbol{x}$ before computing $\boldsymbol{Ax}$ can be represented as $\boldsymbol{A}(\boldsymbol{x}+\boldsymbol{e})$ where $\boldsymbol{e}$ represents the error introduced. We assume that errors do not get canceled or get hidden during the algorithm execution. This notion is specified in the form of the following assumptions:

Non-zero scaling factor assumption: In any operation $\boldsymbol{y} = \boldsymbol{\alpha x}$, we assume that $\boldsymbol{\alpha} \neq \vec{\mathbf{0}}$. We exploit this property to ensure that any error in $\boldsymbol{x}$ is reflected in the output of the operation, enabling efficient detection by checking only $\boldsymbol{y}$. A complete solution needs to check, at runtime, the scaling factor α and detect errors in $\boldsymbol{x}$ if α is close to zero.

Cancellation-less error assumption: $\forall \boldsymbol{x}, \boldsymbol{e_1}, \boldsymbol{e_2} : \boldsymbol{x} + \boldsymbol{e_1} \neq \vec{\mathbf{0}}$ and $\boldsymbol{x} \cdot \boldsymbol{e_1} \neq \vec{\mathbf{0}}$ and $\boldsymbol{e_1} + \boldsymbol{e_2} \neq \vec{\mathbf{0}}$, where $\boldsymbol{x}$ is an arbitrary program variable and $\boldsymbol{e_1}$ and $\boldsymbol{e_2}$ are the errors introduced. This assumption ensures that existing errors in a variable do not get canceled out by subsequent errors.

4. ERROR PRESERVING CHECKSUM FOR MATRIX-VECTOR MULTIPLICATION

In this section, we present our checksum scheme that addresses the aforementioned limitations. Given an $N \times N$ matrix $\boldsymbol{A}$, we define $checksum(\boldsymbol{A})$ (Figure 2(c)) as:

$$checksum(\boldsymbol{A}) = \boldsymbol{c^T A} - d\boldsymbol{c^T},$$

where $\boldsymbol{c}$ is a predefined $N \times 1$ vector and d is a predefined non-zero scalar larger than $n||\boldsymbol{c}||_\infty||\boldsymbol{A}||_\infty / min(\boldsymbol{c})$ (see Lemma 2). Here $min(\boldsymbol{c}) = \min_{1,\cdots,n} |c_i|$ and c_i is the i-th element of the vector $\boldsymbol{c}$.

We encode matrix $\boldsymbol{A}$ to matrix $\boldsymbol{A}^*$ as

$$\boldsymbol{A}^* = \begin{pmatrix} \boldsymbol{A} & \vec{\mathbf{0}} \\ \boldsymbol{c^T A} - d\boldsymbol{c^T} & d \end{pmatrix}. \tag{1}$$

As shown in Figure 2(c), we encode all the vectors $\boldsymbol{x}$ with their own column checksums:

$$\boldsymbol{x}^* = \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{c^T x} \end{bmatrix}$$

The ABFT form of a given operation, such as matrix-matrix multiplication, often performs the same operation on the encoded matrix (Figure 2(c)). For example, $\boldsymbol{A} \cdot \boldsymbol{B}$ is replaced with $\boldsymbol{A}^* \cdot \boldsymbol{B}^*$. In the case of our checksum scheme, encoding the symmetric matrix $\boldsymbol{A}$ leads to $\boldsymbol{A}^*$ that is no longer symmetric. This will cause some iterative methods (e.g., CG) that solve symmetric and positive-definite (SPD) systems to converge slowly, or even diverge. Therefore, we develop a different scheme on ABFT that separates the checksum(s) from the encoded input matrix and vectors, shown in Figure 2(d). This allows the original operation to proceed unchanged while the checksum of the output vector computed directly from the checksums of the input operands. For example,

$$\begin{aligned} \boldsymbol{y}^* = \boldsymbol{A}^*\boldsymbol{x}^* &= \begin{pmatrix} \boldsymbol{A} & \vec{\mathbf{0}} \\ \boldsymbol{c^T A} - d\boldsymbol{c^T} & d \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ checksum(\boldsymbol{x}) \end{pmatrix} \\ &= \begin{pmatrix} \boldsymbol{Ax} \\ (\boldsymbol{c^T A} - d\boldsymbol{c^T})\boldsymbol{x} + d \cdot checksum(\boldsymbol{x}) \end{pmatrix} \\ &= \begin{pmatrix} \boldsymbol{Ax} \\ checksum(\boldsymbol{A}) \cdot \boldsymbol{x} + d \cdot checksum(\boldsymbol{x}) \end{pmatrix} \end{aligned}$$

As can be seen, the output checksum $checksum(\boldsymbol{y})$ can be computed as $(checksum(\boldsymbol{A})\boldsymbol{x} + d \cdot checksum(\boldsymbol{x}))$. In this way, computing the output vector's checksum does not rely on the operations of the encoded input operands (i.e., $checksum(\boldsymbol{y})$ and $\boldsymbol{y}^* = \boldsymbol{A}^*\boldsymbol{x}$ can be computed separately). The output checksum for MVM, VLO and PCO operations are computed as follows:

Matrix-vector multiplication $\boldsymbol{y} = \boldsymbol{Ax}$:

$$checksum(\boldsymbol{y}) = checksum(\boldsymbol{A})\boldsymbol{x} + d \cdot checksum(\boldsymbol{x}) \tag{2}$$

Vector linear-operation $\boldsymbol{z} = \alpha\boldsymbol{x} + \beta\boldsymbol{y}$:

$$checksum(\boldsymbol{z}) = \alpha \cdot checksum(\boldsymbol{x}) + \beta \cdot checksum(\boldsymbol{y}) \tag{3}$$

Preconditioner Say the preconditioned system $\boldsymbol{M}\boldsymbol{z}^{(i)} = \boldsymbol{r}^{(i)}$ needs to be solved to compute the preconditioned system's residual vector $\boldsymbol{z}^{(i)}$, where $\boldsymbol{M}$ is a preconditioner and $\boldsymbol{r}$ is the original system's residual. The preconditioner $\boldsymbol{M}$ can be expressed either explicitly as a matrix or implicitly as a sequence of operations. If $\boldsymbol{M}$ is expressed explicitly, according to the Equation (2), the checksum of $\boldsymbol{z}$ can be computed from $\boldsymbol{r}$ as:

$$checksum(\boldsymbol{z}) = \frac{(checksum(\boldsymbol{M})^T \boldsymbol{z}^{(i)} - checksum(\boldsymbol{r}))}{d} \tag{4}$$

If the preconditioner $\boldsymbol{M}$ is expressed implicitly, e.g., incomplete factors or algebraic multigrid, it will be composed of several matrix-vector multiplications (MVMs) and vector-linear operations (VLOs). Thus, even if the preconditioner $\boldsymbol{M}$ cannot be directly encoded, the checksum of $\boldsymbol{z}$ can be computed through updating each checksum of the output vector after these MVMs and VLOs using Equations (2) and (3).

Since the implicitly expressed preconditioner can be encoded as a composition of encoded MVMs and VLOs, we only consider the explicit expression of $\boldsymbol{M}$ in the following discussion. However, we note that implicit preconditioners composed of MVMs and VLOs can also be efficiently protected by the schemes described in this paper.

We now prove that this checksum scheme can detect soft errors for the key operations in iterative methods, even if the input vectors of these operations are corrupted. We refer to the following operations that can generate a vector in iterative methods as *vector-generating operations*:

Table 1: Vector-generating operations and their expression

Vector-generating operations	Expression
MVM	$\boldsymbol{w} := \boldsymbol{Au}$
PCO	$\boldsymbol{Mw} := \boldsymbol{u}$
VLO scaling	$\boldsymbol{w} := \alpha\boldsymbol{u}$
VLO addition	$\boldsymbol{w} := \boldsymbol{u} + \boldsymbol{v}$

$\boldsymbol{M}$ is a preconditioner and ':=' is assignment. Note that iterative methods are primarily composed of these vector-generating operations (e.g., shown in Figure 1).

Lemma 1. *For any vector-generating operation, if the checksum relationship holds for the input operands and there is no soft error during the operation, the checksum relationship of the output vector is maintained.*

Proof. We prove for each vector-generating operation:

1. Consider an MVM, $\boldsymbol{w} := \boldsymbol{Au}$. We have $\boldsymbol{w} = \boldsymbol{Au}$ and, from Equation (2), $checksum(\boldsymbol{w}) = \boldsymbol{c^T Au} + d(checksum(\boldsymbol{u}) - \boldsymbol{c^T u})$. Combining the two:

   $$checksum(\boldsymbol{w}) - \boldsymbol{c^T w} = d(checksum(\boldsymbol{u}) - \boldsymbol{c^T u}).$$

   If $checksum(\boldsymbol{u}) = \boldsymbol{c^T u}$, $checksum(\boldsymbol{w}) = \boldsymbol{c^T w}$.
2. Consider a PCO, $\boldsymbol{Mw} := \boldsymbol{u}$. Together with Equation (4), we have

   $$checksum(\boldsymbol{w}) - \boldsymbol{c^T w} = \frac{checksum(\boldsymbol{u}) - \boldsymbol{c^T u}}{d}.$$

   If $checksum(\boldsymbol{u}) = \boldsymbol{c^T u}$, $checksum(\boldsymbol{w}) = \boldsymbol{c^T w}$.
3. Consider a VLO, $\boldsymbol{w} = \alpha\boldsymbol{u}$. We have $\boldsymbol{w} = \alpha\boldsymbol{u}$ and, from Equation (2), $checksum(\boldsymbol{w}) = \alpha \cdot checksum(\boldsymbol{u})$. Therefore,

   $$checksum(\boldsymbol{w}) - \boldsymbol{c^T w} = \alpha(checksum(\boldsymbol{u}) - \boldsymbol{c^T u}).$$

   If $checksum(\boldsymbol{u}) = \boldsymbol{c^T u}$, $checksum(\boldsymbol{w}) = \boldsymbol{c^T w}$.
4. Consider a VLO, $\boldsymbol{w} := \boldsymbol{u} + \boldsymbol{v}$. We have $\boldsymbol{w} = \boldsymbol{u}+\boldsymbol{v}$ and, from Equation (3), $checksum(\boldsymbol{w}) = checksum(\boldsymbol{u}) + checksum(\boldsymbol{v})$, thus,

   $$\begin{aligned} checksum(\boldsymbol{w}) - \boldsymbol{c^T w} = &(checksum(\boldsymbol{u}) - \boldsymbol{c^T u}) \\ &+ (checksum(\boldsymbol{v}) - \boldsymbol{c^T v}) \end{aligned}$$

   If $checksum(\boldsymbol{u}) = \boldsymbol{c^T u}$ and $checksum(\boldsymbol{v}) = \boldsymbol{c^T v}$, $checksum(\boldsymbol{w}) = \boldsymbol{c^T w}$. □

Lemma 2. *For any vector-generating operation, any composition of the following soft errors results in the checksum relationship of output vector being broken:*

1. *Arithmetic error affecting the operation;*
2. *Memory bit flips or arithmetic errors in input vectors carried from the previous operations;*
3. *Cache or register bit flips that affect the input vector(s) during the operation.*

Proof. Due to space constraints, we only present the proof for the MVM operation. The proof of the PCO operation is similar. The proof for the VLO operations is the same as in the case of traditional checksum schemes.

Using our new checksum mechanism, we perform an MVM $\boldsymbol{w} := \boldsymbol{Au}$ with checksum update.

1. If there are arithmetic errors during the operation and $\boldsymbol{e_a}$ represents arithmetic errors, the erroneous output vector $\boldsymbol{w}$ can be represented as $\boldsymbol{Au} + \boldsymbol{e_a}$.
2. If the input vector is corrupted by memory bit flips or arithmetic errors (possibly carried from previous operations) before its first use in this operation and $\boldsymbol{e_m}$ represents the errors, the erroneous output vector can be represented as $\boldsymbol{w} = \boldsymbol{A}(\boldsymbol{u} + \boldsymbol{e_m})$.
3. If cache or register bit flips corrupt the input vector during the operation and $\boldsymbol{e_{c1}}, \cdots, \boldsymbol{e_{ck}}$ are cache or register errors, assume $\boldsymbol{A} = \boldsymbol{A_{ne}} + \boldsymbol{A_{e1}} + \cdots + \boldsymbol{A_{ek}}$, where $\boldsymbol{A_{ne}}$ represents the rows that are used in the computation without being affected by soft errors and $\boldsymbol{A_{ei}}$ $(i = 1, \cdots, k)$ represents the i-th row of $\boldsymbol{A}$ if it is affected by a combination of $\boldsymbol{e_{c1}}, \cdots, \boldsymbol{e_{ck}}$ in the computation. Then the erroneous output vector $\boldsymbol{w}$ can be represented as $\boldsymbol{A_{ne}u} + \sum_{i=1}^{k} \boldsymbol{A_{ei}}(\boldsymbol{u} + \sum_{j=1}^{k} \alpha_{ij}\boldsymbol{e_{cj}}) = \boldsymbol{Au} + \sum_{i=1}^{k}\sum_{j=1}^{k} \alpha_{ij}\boldsymbol{A_{ei}e_{cj}}$, where $\alpha_{ij} \in \{0, 1\}$.

Under any composition of these soft errors, the erroneous output vector $\boldsymbol{w}$ can be represented as

$$\boldsymbol{w} = \boldsymbol{A}(\boldsymbol{u} + \boldsymbol{e_m}) + \boldsymbol{e_a} + \sum_{i=1}^{k}\sum_{j=1}^{k} \alpha_{ij}\boldsymbol{A_{ei}e_{cj}}$$

Because the checksum update is calculated after a soft error occurs, we have two scenarios:

1. If the checksum update is not affected by the soft error,

   $$\begin{aligned} checksum(\boldsymbol{w}) &= checksum(\boldsymbol{A})(\boldsymbol{u} + \boldsymbol{e_m}) + d \cdot checksum(\boldsymbol{u}) \\ &= \boldsymbol{c^T A}(\boldsymbol{u} + \boldsymbol{e_m}) - d\boldsymbol{c^T e_m} \end{aligned}$$

   Thus, $checksum(\boldsymbol{w}) - \boldsymbol{c^T w} = -\boldsymbol{c^T}(\sum_{i=1}^{k}\sum_{j=1}^{k} \alpha_{ij}\boldsymbol{A_{ei}e_{cj}}) - d\boldsymbol{c^T e_m} - \boldsymbol{c^T e_a}$.

2. If checksum update is affected by the soft error,

$$checksum(\boldsymbol{w}) =$$
$$checksum(\boldsymbol{A})(\boldsymbol{u} + \boldsymbol{e_m} + \sum_{i=1}^{k} \boldsymbol{e_{ci}}) + d \cdot checksum(\boldsymbol{u})$$
$$= \boldsymbol{c^T A}(\boldsymbol{u} + \boldsymbol{e_m} + \sum_{i=1}^{k} \boldsymbol{e_{ci}}) - d\boldsymbol{c^T}(\sum_{i=1}^{k} \boldsymbol{e_{ci}} + \boldsymbol{e_m})$$

Thus, $checksum(\boldsymbol{w}) - \boldsymbol{c^T w}$ is

$$\sum_{i=1}^{k}(\boldsymbol{c^T A} - d\boldsymbol{c^T} - \boldsymbol{c}^T \sum_{j=1}^{k} \alpha_{ji} \boldsymbol{A_{ej}})\boldsymbol{e_{ci}} - d\boldsymbol{c^T e_m} - \boldsymbol{c^T e_a}$$
$$= \sum_{i=1}^{k}(\boldsymbol{c^T A_e} - d\boldsymbol{c^T})\boldsymbol{e_{ci}} - d\boldsymbol{c^T e_m} - \boldsymbol{c^T e_a}$$

Since $\boldsymbol{A_e}$ is a part of $\boldsymbol{A}$, the absolute value of each element in vector $\boldsymbol{c^T A_e}$ is no larger than $n||\boldsymbol{c}||_\infty||\boldsymbol{A}||_\infty$, which means $||\boldsymbol{c^T A_e}||_\infty \leq n||\boldsymbol{c}||_\infty||\boldsymbol{A}||_\infty$. Moreover, we choose d to be larger than $n||\boldsymbol{c}||_\infty||\boldsymbol{A}||_\infty / min(\boldsymbol{c})$, thus, the absolute value of each element in vector $d\boldsymbol{c^T}$ is larger than $n||\boldsymbol{c}||_\infty||\boldsymbol{A}||_\infty$, which means $min(d\boldsymbol{c^T}) > n||\boldsymbol{c}||_\infty||\boldsymbol{A}||_\infty$. Therefore, $min(d\boldsymbol{c^T}) > ||\boldsymbol{c^T A_e}||_\infty$. It demonstrates that $\boldsymbol{c^T A_e}$ can not be equal to $d\boldsymbol{c^T}$, thus, $\boldsymbol{c^T A_e} - d\boldsymbol{c^T} \neq \vec{\boldsymbol{0}}$.

Therefore, if any of $\boldsymbol{e_a}, \boldsymbol{e_m}, \boldsymbol{e_{c1}}, \cdots, \boldsymbol{e_{ck}}$ is not equal to $\vec{\boldsymbol{0}}$, according to our error model, $checksum(\boldsymbol{w}) \neq \boldsymbol{c^T w}$. □

Theorem 3. *For any vector-generating operation, the checksum relationship of the output vector is preserved if and only if there are no soft errors before or during the operation.*

Proof. The proof follows from Lemma 1 (if part) and Lemma 2 (only-if part). □

Based on Theorem 3, if the checksum relationship of the output vector from a vector-generating operation is broken, soft errors must have occurred before (memory bit flips or arithmetic errors carried from the previous operations) or during (memory bit flips of the input vector or arithmetic errors) the operation. On the other hand, if the checksum relationship of the output vector is maintained, the checksum relationship of any input vector is held and this can guarantee that no soft error (arithmetic errors, or bit flips in memory, caches or registers) happened before or during the operation. *This provides an efficient approach to soft error detection for all the vector-generating operations: we only need to identify if the checksum relationship of the output vector is broken.*

As mentioned in Section 2, the traditional checksum and its encoding for MVM and PCO does not propagate the inconsistency of the input vector to the output vector when the input vector is corrupted before the operation. In order to get better coverage for error detection, ABFT approaches based on prior checksum schemes need to check every input and output vector in all operations, incurring large detection overheads. This is evaluated in greater detail in Section 6.

5. NEW ONLINE ABFT SCHEMES

We now use the checksum encoding scheme described above to design efficient online ABFT solutions for iterative methods. While widely applicable to iterative solvers, for simplicity and clarity, we will illustrate the designs in the context of the widely-used preconditioned conjugate gradient (PCG).

5.1 "Lazy" Detection: Low-Cost Online ABFT Algorithm Using Checksum Update

The *preconditioned conjugate gradient (PCG)* method is one of the most commonly used iterative methods to solve the sparse linear system $\boldsymbol{Ax} = \boldsymbol{b}$ when the coefficient matrix $\boldsymbol{A}$ is symmetric and positive definite (SPD). PCG consists of three major computation components: successive approximations to the solution, residuals corresponding to the approximate solution, and search directions used to update both the approximate solutions and the residuals [13]. Each iteration consists of one sparse MVM, three vector updates, and two vector inner-products (Figure 1). Figure 3 outlines our first proposed online ABFT algorithm (**Algorithm 1**) for PCG based on the proposed checksum mechanism. The figure also illustrates the cost (in terms of operation count) for the added code lines. In **Algorithm 1**, after each vector-generating operation (i.e., MVM, VLO and PCO), we efficiently update the checksum for each output vector according to Equations (2), (3), and (4).

Table 2: Computation relationships among various vectors in PCG.

Output Vector	Input Vector(s)	Operation
$\boldsymbol{z}$	$\boldsymbol{r}$	$\boldsymbol{z} = \boldsymbol{M^{-1}r}$
$\boldsymbol{p}$	$\boldsymbol{z}$, $\boldsymbol{p}$	$\boldsymbol{p} = \boldsymbol{z} + \beta\boldsymbol{p}$
$\boldsymbol{q}$	$\boldsymbol{p}$	$\boldsymbol{q} = \boldsymbol{Ap}$
$\boldsymbol{x}$	$\boldsymbol{x}$, $\boldsymbol{p}$	$\boldsymbol{x} = \boldsymbol{x} + \alpha\boldsymbol{p}$
$\boldsymbol{r}$	$\boldsymbol{r}$, $\boldsymbol{q}$	$\boldsymbol{r} = \boldsymbol{r} - \alpha\boldsymbol{q}$

To detect soft errors, the simplest method is to *verify the checksum relationship* of each output vector after every vector-generating operation. However, this incurs high detection overhead. The most practical recovery strategy for iterative solvers involves checkpoint/rollback. When any soft error is detected, the program will be rolled back to the nearest checkpoint. Minimizing the fault tolerance cost requires balancing the checkpointing overheads with the potential to lose significant amount of work in the event of an error-induced rollback. To reduce the overall error checking and recovery overhead, we analyze the computational relationships among all the involved vectors in the vector-generating operations and see if their checksums really need to be verified after every operation.

We make the following three observations based on the summarized computational relationships between the vectors in PCG, shown in **Table 2**:

- Soft errors, if present, in vectors $\boldsymbol{z}$, $\boldsymbol{p}$, or $\boldsymbol{q}$ will eventually propagate to the vectors $\boldsymbol{x}$ and $\boldsymbol{r}$. Therefore, verifying the checksum relationship of the vector $\boldsymbol{x}$ and $\boldsymbol{r}$ is adequate to cover all the other vectors.
- Computing vectors $\boldsymbol{p}$, $\boldsymbol{x}$, and $\boldsymbol{r}$ requires their results from the previous iteration, which means that soft errors in $\boldsymbol{p}$, $\boldsymbol{x}$ and $\boldsymbol{r}$, if presented in an iteration, will propagate to the subsequent iterations.
- At each iteration, we can use vectors $\boldsymbol{p}$ and $\boldsymbol{x}$ to compute the other three vectors $\boldsymbol{q}$, $\boldsymbol{r}$, and $\boldsymbol{z}$.

Corresponding to these three observations, we identify **three optimizations** to significantly reduce the overhead of the error detection and recovery for PCG:

1. Rather than verifying each output vector's checksum relationship after every vector-generating operation, we only need to **verify** two checksum relationships, namely $checksum(\boldsymbol{x}) = \boldsymbol{c^T x^{(i)}}$ and $checksum(\boldsymbol{r}) = \boldsymbol{c^T r^{(i)}}$, to detect the soft errors in any vector (line 6 in Algorithm 1);
2. We only need to **verify** the checksum relationship between $\boldsymbol{x}$ and $\boldsymbol{r}$ *every several iterations* rather than every iteration (line 5 in Algorithm 1);
3. We only need to **checkpoint** two vectors, $\boldsymbol{p}$ and $\boldsymbol{x}$. In

Algorithm 1 Online-ABFT Preconditioned Conjugate Gradient Algorithm Based on New Checksum with checkpoint/restart Technique

1:	Compute $r^{(0)} = b - Ax^{(0)}, z^{(0)} = M^{-1}r^{(0)}, p^{(0)} = z^{(0)}$ and $\rho_0 = {r^{(0)}}^T z^{(0)}$ for some initial guess $x^{(0)}$	
2:	Compute $checksum(A) = c^T A - dc^T$, $checksum(M) = c^T M - dc^T$, $checksum(b) = c^T b^{(0)}$, $checksum(x) = c^T x^{(0)}$, $checksum(r) = c^T r^{(0)}$, $checksum(z) = [(checksum(r) - checksum(M)z^{(0)})]/d$, $checksum(p) = checksum(z)$	
3:	Checkpoint A, M	
4:	**for** $i = 0, 1, \cdots$ **do**	
5:	**if** $((i > 0)$ and $(i\%d = 0))$ **then**	
6:	**if** $(\|\|checksum(r) - c^T r^{(i)}\|\|/n > \theta)$ or $(\|\|checksum(x) - c^T x^{(i)}\|\|/n > \theta)$ **then**	2 VDP (4n FLOPS)
7:	Rollback: recover $A, M, i, \rho_i, p^{(i)}, x^{(i)}, r^{(i)}$	2 Mcpy + 4 Vcpy
8:	**else if** $(i\%(cd) = 0)$ **then**	
9:	Checkpoint: $i, \rho_i, p^{(i)}, x^{(i)}$	2 Vcpy
10:	**end if**	
11:	**end if**	
12:	$q^{(i)} = Ap^{(i)}$	1 MVM (nnz FLOPS)
13:	$checksum(q) = checksum(A)p^{(i)} + d \cdot checksum(p)$	1 VDP (2n FLOPS)
14:	$\alpha_i = \rho_i / {p^{(i)}}^T q^{(i)}$	1 VDP (2n FLOPS)
15:	$x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$	1 VLO (2n FLOPS)
16:	$checksum(x) = checksum(x) + \alpha_i \cdot checksum(p)$	2 FLOPS
17:	$r^{(i+1)} = r^{(i)} - \alpha_i q^{(i)}$	1 VLO (2n FLOPS)
18:	$checksum(r) = checksum(r) - \alpha_i \cdot checksum(q)$	2 FLOPS
19:	solve $Mz^{(i+1)} = r^{(i+1)}$	1 PCO
20:	$checksum(z) = \frac{(checksum(r) - checksum(M)z^{(i+1)})}{d}$	1 VDP (2n FLOPS)
21:	$\rho_{i+1} = {r^{(i+1)}}^T z^{(i+1)}$	1 VDP (2n FLOPS)
22:	$\beta_i = \rho_{i+1} / \rho_i$	1 FLOPS
23:	$p^{(i+1)} = z^{(i+1)} + \beta_i p^{(i)}$	1 VLO (2n FLOPS)
24:	$checksum(p) = checksum(z) + \beta_i \cdot checksum(p)$	2 FLOPS
25:	check convergence; continue if necessary	
26:	**end for**	

MVM: Matrix-Matrix Multiplication
VDP: Vector Dot-Product
VLO: Vector Linear Operation
PCO: Preconditioned Operation
Mcpy: Matrix Copy
Vcpy: Vector Copy

Figure 3: online ABFT algorithm for preconditioned conjugate gradient (PCG) based on our new checksum mechanism. The operation count for checksum updates, error detection, and recovery (checkpoint/restart) are also listed.

the event of an error, we can use the checkpointed version of the two vectors to recover all the other vectors and checksums (line 9 in Algorithm 1) .

Algorithm 1 in Figure 3 shows a low-cost online ABFT-based PCG algorithm that includes these optimizations. In Figure 3 and 4, black represents the original code of PCG; pink represents the checkpoints; red represents the checksum updates; blue represents the error detection and rollback. Pink, red, and blue show the extra operations introduced over the original code. This scheme enables "lazy" error detection mode that only checks errors every several iterations based on the assumption that under a lower error rate, immediate checksum verification after every vector-generating operation is too expensive. Note that we still update the checksums of the output vector after each operation using the low-cost scheme described earlier. For error detection in Algorithm 1, we verify the checksum relationship of $\boldsymbol{x}$ and $\boldsymbol{r}$ every error detection interval (d), based on the optimization (1) and (2) above. The overhead of such detection is only the checksum verification for two VLOs (line 6), which is $O(n)$ FLOPS. For error recovery, according to optimization (3), we only need to checkpoint the two vectors $\boldsymbol{p}$ and $\boldsymbol{x}$ every checkpoint interval (cd), which can significantly reduce the checkpointing overhead and the large memory space requirements. Note that for the purpose of high scalability in parallel computation, all the checkpoints and checksums are saved locally in our proposed designs. We will discuss error detection interval d and checkpoint interval cd in the later section.

We use $\theta = 10^{-10}$ as threshold in our experimental evaluations. In reality, we perform all the operations using floating-point arithmetics with round-off errors. When the checksum relationships of $\boldsymbol{x}$ and $\boldsymbol{r}$ are used to detect errors, the effects of round-off errors need to be carefully investigated. In Algorithm 1, as the problem size n increases, the accuracy of the round-off error (i.e., $checksum(\boldsymbol{x}) - \boldsymbol{c^T x}$) decreases. When verifying checksum relationship, we apply $(checksum(\boldsymbol{x}) - \boldsymbol{c^T x})/n$ to reduce the accuracy loss for round-off errors. When the errors are close to the machine accuracy ϵ, we cannot detect them. However, because we only focus on numerically stable solvers and well-conditioned problems, these errors do not need to be detected since they will not significantly impact the performance of numerically stable algorithms and well-conditioned problems.

5.2 "Eager" Online Recovery for MVM Using Triple Checksums

In order to reduce the chances of a rollback, we would like to correct errors as soon as possible without requiring rollback. In iterative methods, since MVM operations are the most computation-intensive, and therefore the most vulnerable operations, they could benefit from a faster recovery under a high error rate.

According to coding theory, $2m + 1$ checksums (i.e., independent equations) can be used to locate and correct m errors. For instance, Figure 2(e) shows the case of two separate checksums in our scheme for detection and correction. However, this requires a strong assumption of a bound m on the maximum number of soft errors in an MVM. A larger number of errors than m can lead to the recovery mechanism mis-identifying some locations as erroneous and correcting them, resulting in cases of "fake correction". We illustrate this scenario and present a solution.

Consider a double-checksum used to detect errors and correct up to one error in the output vector. Shown in Figure 2(e), let the output vector be $\boldsymbol{y} = (y_1, y_2, \cdots, y_n)^T$. We encode it with double checksums as $\boldsymbol{y}^* = (\boldsymbol{y}, \boldsymbol{c_1^T y}, \boldsymbol{c_2^T y})^T$. In this example, we use $\boldsymbol{c_1} = (1, 1, \cdots, 1)^T$ and $\boldsymbol{c_2} = (1, 2, \cdots, n)^T$. The double checksums $checksum_1(\boldsymbol{y})$ and $checksum_2(\boldsymbol{y})$ can be represented as

$$checksum_1(\boldsymbol{y}) = \boldsymbol{c_1^T y} = \sum_{i=1}^{n} y_i$$
$$checksum_2(\boldsymbol{y}) = \boldsymbol{c_2^T y} = \sum_{i=1}^{n} i y_i$$

Now, say the output vector $\boldsymbol{y'} = (y'_1, y'_2, \cdots, y'_n)^T$ has one erroneous element. Specifically, $y'_j \neq y_j$, where the error position j is to be determined to locate the error. The presence of the error can be detected as:

$$\delta_1 = \sum_{i=1}^{n} y'_i - checksum_1(\boldsymbol{y}) = y'_j - y_j \neq 0$$
$$\delta_2 = \sum_{i=1}^{n} i y'_i - checksum_1(\boldsymbol{y}) = j(y'_j - y_j) \neq 0$$

One common way to locate the error position j is to calculate a simple division $\delta_2/\delta_1 = j$, and then apply $\delta_1 = y'_j - y_j$ to correct the erroneous computed value y'_j through $y'_j = y'_j - \delta_1$. In reality, we do not know the number of soft errors that has affected a given MVM. For instance, *one error could occur in the input vector and propagate to the output, forming multiple errors.* Using the two checksums, we can only tell if the MVM result is erroneous, but cannot know if it has only one soft error. If there are actually k erroneous elements $y'_{j_1}, \cdots, y'_{j_k}$ $(k > 1)$, but the assumption is that

there is only one error, we calculate δ_1 and δ_2 as

$$\delta_1 = \sum_{i=1}^{k}(y'_{j_i} - y_{j_i}), \delta_2 = \sum_{i=1}^{k} j_i(y'_{j_i} - y_{j_i})$$

If $y'_{j_1} - y_{j_1} = \cdots = y'_{j_k} - y_{j_k}$ and $j_1 + \cdots + j_k$ is multiples of k, $\delta_2/\delta_1 = (j_1 + \cdots + j_2)/k$ is an integer. Therefore, the double-checksum mechanism will lead to a fake correction of $\boldsymbol{y}'$, which may result in slow convergence or divergence of the underlying iterative method.

Inspired by Fasi et al. [10], we propose a triple-checksum detection and correction mechanism to identify if there is only one error, and if so, correct it. Specifically, we introduce a different third checksum with vector $\boldsymbol{c_3} = (1, \frac{1}{2}, \cdots, \frac{1}{n})^T$. Assuming that there are k erroneous elements $y'_{j_1} \cdots y'_{j_k}$. δ_1, δ_2 and δ_3 are computed as:

$$\delta_1 = \sum_{i=1}^{k}(y'_{j_i} - y_{j_i}), \delta_2 = \sum_{i=1}^{k} j_i(y'_{j_i} - y_{j_i}), \delta_3 = \sum_{i=1}^{k} \frac{y'_{j_i} - y_{j_i}}{j_i}$$

Now we can use the relationship between δ_1, δ_2 and δ_3 to identify if there is only one error, eliminating the fake correction case above. Since in the fake correction case, $y'_{j_1} - y_{j_1} = \cdots = y'_{j_k} - y_{j_k}$, we have

$$\delta_2/\delta_1 = \frac{j_1 + \cdots + j_k}{n}, \delta_1/\delta_3 = \frac{n}{\frac{1}{j_1} + \cdots + \frac{1}{j_k}}$$

Therefore, δ_2/δ_1 and δ_1/δ_3 are the arithmetic mean and harmonic mean, respectively, of $j_1, \cdots, j_k$. The relation between the two means requires that $\delta_2/\delta_1 = \delta_1/\delta_3$ **if and only if** $j_1 = \cdots = j_k$. However, since the position indices $j_1, \cdots, j_k$ are all different, $\delta_2\delta_3 = \delta_1^2$ **if and only if** $k = 1$. Therefore, we are able to use this simple verification $\delta_2\delta_3 = \delta_1^2$ to identify if the output vector is erroneous when there is only a single error. If it is a single error, we can locate and correct the error right away using this triple-checksum mechanism.

In order to match the triple-checksum encoding in the vectors, the matrices are encoded in the following fashion:

$$\boldsymbol{A}^* = \begin{pmatrix} \boldsymbol{A} & \vec{\mathbf{0}} & \vec{\mathbf{0}} & \vec{\mathbf{0}} \\ \boldsymbol{c_1^T A} - d_1\boldsymbol{c_1^T} - d_2\boldsymbol{c_2^T} - d_3\boldsymbol{c_3^T} & d_1 & d_2 & d_3 \\ \boldsymbol{c_2^T A} - d_2\boldsymbol{c_1^T} - d_3\boldsymbol{c_2^T} - d_1\boldsymbol{c_3^T} & d_2 & d_3 & d_1 \\ \boldsymbol{c_2^T A} - d_3\boldsymbol{c_1^T} - d_1\boldsymbol{c_2^T} - d_2\boldsymbol{c_3^T} & d_3 & d_1 & d_2 \end{pmatrix}$$

Therefore, applying triple-checksum encoding, we can: (1) detect if there is any error, (2) identify whether or not there is more than one error, and (3) if there is only one error, locate and correct it.

5.3 "Hybrid" Detection: Two-Level Online ABFT Algorithm Using Triple-Checksum

Based on this triple-checksum protection mechanism and **Algorithm 1**, we present a *two-level online ABFT* scheme for iterative methods that uses the triple-checksum for immediate single error detection and recovery in MVM (inner-level), and the checksum relationship verification to recover from multiple errors (outer-level). The general procedure to construct a two-level online ABFT version of a given iterative solver is as follows:

1. Encode matrices and vectors using triple-checksum;
2. Form the checksum update formulas for the output vectors based on the encoded matrices and vectors;
3. Compute these checksum updates after each vector-generating operation (MVM, VLO, PCO);
4. Analyze the dependency relationships between vectors;
5. Every d iterations, invoke the outer-level protection: verify the checksum relationships of the vectors that the other vectors will eventually propagate to every d iterations;
6. At the beginning of every cd iterations, checkpoint the minimum number of the vectors based on the dependency relationships from **Step 4** and scalars that can calculate the other vectors;
7. After each MVM, add the inner-level protection:
   (a) Use one out of the three checksums of the output vector to identify if there is any error in MVM
   (b) If there are no errors, update the vector checksums and continue the execution
   (c) If the output vector is erroneous, use the tripe-checksum mechanism to identify if it is one error or multiple errors
   (d) If there is one error in MVM, apply triple-checksum to correct immediately
   (e) If there are multiple errors, rollback to the previous checkpoint

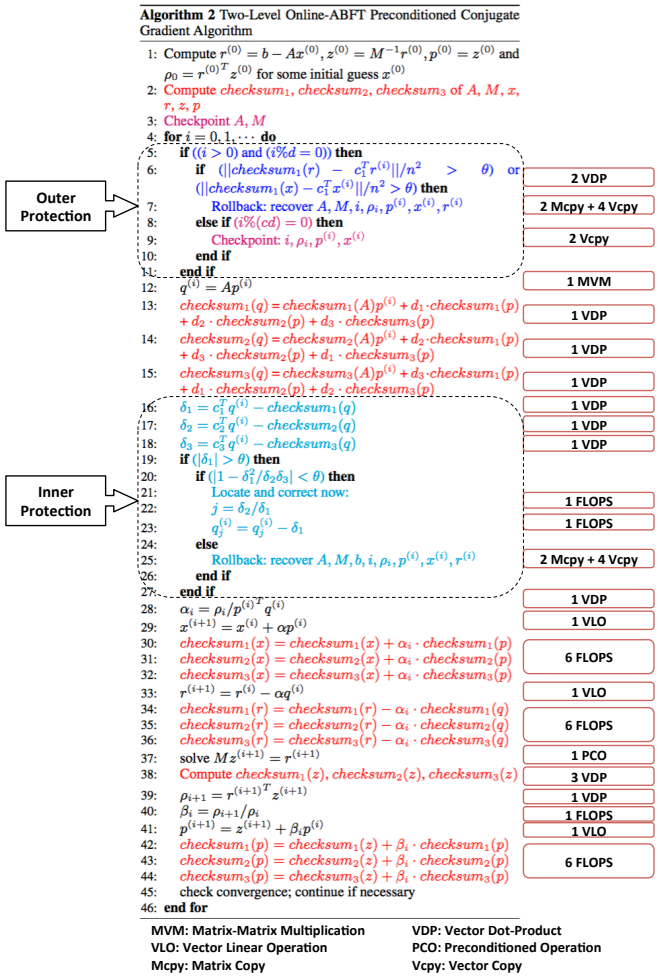

Figure 4: Two-level online ABFT algorithm for preconditioned conjugate gradient (PCG)

We illustrate this procedure for the PCG solver in **Algorithm 2**. For the **outer-level** protection (lines 5–11), we verify the checksum relationship of the vectors $\boldsymbol{x}$ and $\boldsymbol{r}$ for detecting the potential error(s) in VLOs and multiple errors that cannot be recovered in MVM, and then rollback to the latest secure state if the condition is not satisfied (recov-

ery). For the **inner-level** protection (lines 16–27), after the MVM operation $\boldsymbol{q} = \boldsymbol{Ap}$, we verify the checksum relationship of the output vector $\boldsymbol{q}$ (line 19). If it does not hold, it indicates that soft error has affected the MVM. Then we use the simple check $\delta_2\delta_3 = \delta_1^2$ (line 20) to determine the cause of the inconsistency in $\boldsymbol{q}$: a single error or multiple errors. If the verification fails, it indicates that there are multiple errors. Since our triple-checksum mechanism cannot correct more than one error, the program is rolled back to the closest checkpoint immediately for recovery, in order to avoid the waiting till the beginning of the next d interval. If the verification passes, it indicates that there is only one error and the error is corrected immediately via triple-checksum.

This two-level algorithm effectively protects all the vector-generating operations in an iterative method. Specifically, we identify the essential vectors that need to be checkpointed, recover from the most common case of single error by locating and correcting the error rather than rolling back, while ensuring that the outer-level detector catches possible errors from the other operations. In the common case, where there are no errors, the inner-level detector checks for errors with a single checksum. This operation incurs $O(n)$ cost per MVM operation per time step. For sparse matrices with number of non-zeros being much larger than vector length, say $nnz > 10n$, this check adds very little overhead. The efficient check and correction employed by the inner-level scheme enables us to reduce the frequency of the outer-level protection (i.e., increasing the outer protection interval d). Additionally, since the outer protection is using the higher-cost checkpoint/restart technique for recovery, at a high error rate, our two-level protection mechanism may significantly lower the overhead introduced by checkpoint/restart and, therefore, the overall overhead (see Section 6.2 and 6.3).

6. RESULTS AND ANALYSIS

In this section, we start by comparing the following five soft-error tolerating schemes in terms of features and coverage: (1) the offline residual checking at the end of computation (denoted by **offline residual**), (2) the state-of-the-art online matrix-vector multiplication (MVM) scheme using the traditional checksum proposed in [11] (denoted by **online MV**), (3) the online orthogonality checking proposed in [6] (denoted by **online orthogonality**), (4) our proposed "lazy" online ABFT scheme using checksum updates and checkpoint/rollback (**Algorithm 1** in Section 5.1, denoted by **basic online ABFT**), and (5) our proposed two-level online ABFT algorithm using triple-checksum mechanism (**Algorithm 2** in Section 5.3, denoted by **two-level online ABFT**). After that, we theoretically compare the cost of applying online-MV scheme with that of applying our two schemes, assuming they have the same error coverage. Finally, we empirically evaluate our two schemes on a leadership supercomputer under different error scenarios, and compare their costs with those using online MV.

We conduct our evaluation with the widely used preconditioned conjugate gradient (PCG) solver and preconditioned biconjugate gradient stabilized (PBiCGSTAB) solver [17]. The former has the orthogonality relations between essential vectors, but the latter does not. For input, we use the sparse matrix 'G3_circuit', the largest SPD matrix available from the University of Florida Sparse Matrix Collection [7]. It contains 1,585,478 rows and columns with 7,660,826 non-zero elements. We implement our proposed online ABFT schemes in PETSc [1], one of the most popular toolkits providing parallel solutions of scientific applications modeled by PDEs. All of our experiments are conducted using 2048 cores (i.e., 256 nodes, each node with two Intel Xeon E5-2680 processors) on the Stampede supercomputer at Texas Advanced Computer Center (TACC).

Table 3: Comparison of features and error coverage among different fault-tolerance techniques for iterative methods

	Offline residual	Online MV	Online orthogonality	Basic/two-level online ABFT method
Can protect arithmetic error	Yes	Yes	Yes	Yes
Can protect memory bit flips	Yes	Yes	Yes	Yes
Can protect cache or register bit flips	Yes	No	No	Yes
Can be applied to all iterative methods	Yes	Yes	No	Yes
Not necessary to check every iteration	Yes	No	Yes	Yes
Not necessary to check every operation	Yes	No	Yes	Yes

6.1 Error Coverage and Feature Comparison

Table 3 compares features and error coverage of different fault tolerance schemes for iterative methods. The table clearly shows that only the offline residual scheme and our proposed two designs have all the six soft-error tolerance features. The offline-residual scheme simply verifies the residual at the end of computation to identify if there is any error. If there is, the offline-residual scheme has to recompute the entire program. Under the best-case scenario where the convergence iteration number is set to that of the correct execution, the offline-residual scheme incurs 100% overhead. Therefore, the offline-residual scheme will most likely perform worse than the well-designed online schemes if any soft error occurs during the program execution [6]. Therefore, it is excluded from the following performance evaluation. For the online-MV scheme, we will evaluate it both theoretically (Section 6.2) and empirically (Section 6.3) against our approaches. To further demonstrate better coverage of our schemes over the online-orthogonality approach, we will use the preconditioned biconjugate gradient stabilized (PBiCGSTAB) solver in addition to PCG, to evaluate our designs in Section 6.3.

6.2 Theoretical Performance Comparison

As previously discussed in Section 4 and Table 3, although the **online MV** approach (i.e., using the traditional checksum encoding schemes) cannot protect cache or register bit-flips, we still would like to explore the performance of applying it to iterative methods when those error scenarios are absent. We also want to conduct theoretical performance comparison between online MV and our two approaches under different error rates. We select the recent online ABFT scheme from Sloan et al. [19] that uses traditional checksum encoding method for comparison. Note that Sloan's method only considers soft errors in matrix-vector multiplication (MVM). For the purpose of a fair comparison, we apply the standard triple modular redundancy (TMR) to protect other operations in iterative methods such as VLOs and PCOs, since the traditional encoding mechanism cannot encode PCOs. We also exclude cache or register bit-flips from the error scenarios for this comparison because the online-MV approach cannot protect them. To be consistent with the use case in [19], we use PCG as the candidate iterative method. The algorithm cost analysis for our approaches are

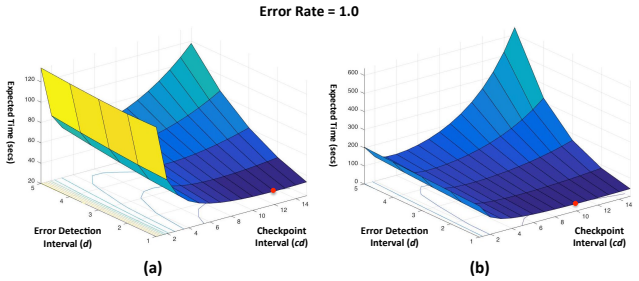

Figure 5: Expected execution time of our basic online ABFT scheme for (a) PCG and (b) PBiCGSTAB for G3_circuit with error rate $\lambda = 1.0$.

shown in Figures 3 and 4. Due to space limitation, we refer the reader to [19] for the details of Sloan's algorithm.

We denote the error detection interval as d (i.e., outer-loop error detection happens every d iterations in our approaches) and checkpoint interval as cd (i.e., checkpointing necessary data every cd iterations). d and cd are integers and $cd > d$. We use I to denote the total number of iterations and assume $I = k \times cd$ where $k \geqslant 1$. Since Sloan's approach is not based on checkpoint/rollback technique, d does not exist in Online-MV. The following three error-rate scenarios are explored in this comparison:

- Scenario 1: One error in MVM during the entire execution (I iterations).
- Scenario 2: One error in MVM every cd iterations.
- Scenario 3: One error in MVM every iteration.

These three scenarios correspond to low error rate, medium or high error rate, and extremely high error rate, respectively. Table 4 shows the performance overhead per iteration for basic online ABFT (O_1), two-level online ABFT (O_2) and online MV (O_3) under the three scenarios in PCG. Note that $c_0 = nnz/n$ represents the sparsity of the matrix $\boldsymbol{A}$; and ∞ illustrates that the execution will not terminate due to the repeated rollbacks. Also, all the costs shown in Table 4 are the average values from different error locations in cd. From Table 4, we can make the following conclusions:

1. Under low or extremely low error rate (Scenario 1), **basic online ABFT (O_1)** approach has the lowest overhead: $O_1 < O_2 < O_3$. This is because PCO consumes much more time than vector-vector multiplication (VDP) and vector-linear operation (VLO) in CG.
2. Under medium/high error rate (Scenario 2), the performance overhead of **two-level online ABFT (O_2)** is the lowest among the three. However, the performance comparison between our basic online ABFT and online MV is unclear, depending on if PCO is more time consuming than MVM. For example, if the matrix $\boldsymbol{A}$ is highly sparse, $O_1 < O_3$ because PCO in this case consumes more time. Otherwise, if PCO is less time-consuming, e.g., the preconditioner $\boldsymbol{M}$ is well selected, $O_3 < O_1$.
3. Under extremely high error rate (Scenario 3), **two-level online ABFT (O_2)** wins: $O_2 < O_3 < O_1$. The basic online ABFT (O_1) will not terminate.

To summarize, no matter the error rate, one of our approaches will outperform the online MV method implemented with [19] for PCG. The methodology used here can also be applied to analyze other iterative methods.

6.3 Empirical Performance Evaluation

We implement our proposed online ABFT schemes in PETSc, and use its default preconditioner (block Jacobi with ILU/IC) and convergence tolerance. We simulate an arithmetic or storage error by significantly increasing the value of a random element in matrices or vectors.

6.3.1 *Determining Optimal cd and d*

As discussed previously, the expected execution time of an application depends on the error detection interval (d) and checkpoint interval (cd), which are commonly determined by the system's error rate (λ). However, it is often difficult to determine their optimal values for different designs. In order to conduct a fair evaluation and comparison, we first estimate their optimal values under a certain error rate. Because both our proposed designs are based on (or partially based on) loop-level checkpoint/rollback, it is similar to Chen's online orthogonality approach [6]. Therefore, we modify the expected execution time formula from that work to accommodate our scenarios. The following equation shows the expected time for basic online ABFT:

$$E(c,d) = \min_{c,d} \frac{I}{cd}[(e^{\lambda cd(t+t_u+t_d/d)} - 1)(\frac{d\cdot(t+t_u)+t_d}{1-e^{-\lambda cd(t+t_u+t_d/d)}} + t_r) + t_c] \quad (5)$$

where λ denotes the error rate, t_d denotes the time to detect an error (i.e., loop-level detection), t_c denotes the overhead for one checkpoint, t_r denotes checkpoint recovery overhead, t denotes the time for each iteration, d denotes the error detection interval (in iterations), cd denotes the checkpoint interval (in iterations), and I is the total number of iterations. Additionally, we add t_u to represent the overhead for checksum updates per iteration in basic online ABFT. We assume the time to failure for all processes follows an independent and identically distributed exponential distribution. Due to space limitations, please refer to [6] for details on how the equation is derived.

Based on Equation (5), Figure 5 demonstrates the simulated correlations between error detection interval (d), checkpoint interval (cd) and the expected execution time under a medium/high error rate, denoted as $\lambda = 1.0$. All the input parameters used to construct Figure 5, such as t_d, t_c , t_r, and t_u, are the average measurements from 50 runs on Stampede. Based on Figure 5, we can estimate the optimal pairs of (cd, d) for both PCG and PBiCGSTAB algorithms implemented with basic online ABFT as (12, 1) and (10, 1). Similarly, when the error rates are low and extremely high, e.g., $\lambda = 10^{-2}$ and $\lambda = 10$, the optimal (cd, d) pairs for PCG and PBiCGSTAB implemented with basic online ABFT can be found in Table 5. Since the goal of the empirical analysis is to compare the overhead induced by different techniques, optimal (cd, d) is not required for two-level online ABFT as long as they are consistent between our two designs (see Table 4). Thus, for simplicity, we use the same (cd, d) from basic online ABFT for error injection in two-level online ABFT in the following comparative analysis. Note that, since the proposed ABFT algorithms conduct a low-cost error detection every d iterations for only two vector-dot products, our detection overhead is small under these scenarios, therefore, the optimal strategy is to detect errors at every iteration.

6.3.2 *Overhead Comparison Between Techniques*

In order to validate the conclusions from the theoretical comparison in Section 6.2, we will use the same error

Table 4: Theoretical cost analysis for three schemes under different error-rate scenarios.

	Basic online ABFT (O_1)	Two-level online ABFT (O_2)	Online MV (O_3)
Scenario 1	(2/d+2)VDP+2VLO/cd	(2/d+9)VDP+2VLO/cd	1PCO+2VDP+3VLO
Scenario 2	0.5MVM+(2/d+5)VDP+0.5PCO+(6(1+c_0)/cd + 1.5)VLO	(2/d+9)VDP+2VLO/cd	1PCO+(5/cd+2)VDP+3VLO
Scenario 3	$+\infty$	(2/d+9)VDP+2VLO/cd	1PCO+7VDP+3VLO

Table 5: Optimal pairs of (cd, d) for PCG and PBiCGSTAB algorithms implemented with basic online ABFT.

	PCG	PBiCGSTAB
$\lambda = 10^{-2}$	(1000, 1)	(1000, 1)
$\lambda = 1$	(12, 1)	(10, 1)
$\lambda = 10$	(1, 1)	(1, 1)

scenarios to conduct our empirical analysis for PCG and PBiCGSTAB, denoting them as O_1, O_2, and O_3. As discussed in Section 6.3.1, in corresponding to the three error scenarios, we will use the optimal (cd, d) for three error rates (i.e., low, medium/high, extremely high) from Table 5 to setup the experiments. It is worth noting that, unlike PCG, PBiCGSTAB does not exhibit the orthogonal property within its essential vectors and it is more compute-intensive than PCG (i.e., two MVMs and two PCOs every iteration). The implementations of the basic PCG and PBiCGSTAB algorithms are from PETSc. We use their default preconditioners. All the results are the average values from different error locations in cd.

Figure 6 shows the overall performance comparison between three online ABFT implementations of PCG under different error scenarios. We make the following observations. (1) The performance overhead for all three designs is low (i.e., 0.4%, 2.2% and 1.3% respectively) when the execution is error-free. (2) Under low error rate (Scenario 1), basic online ABFT has the lowest overhead: $O_1 < O_2 < O_3$. (3) Under medium/high error rate (Scenario 2), two-level online ABFT performs the best: $O_2 < O_3 < O_1$. As the preconditioner M is well-selected for PCG, PCO is less time-consuming than MVM, causing $O_3 < O_1$. (4) Under extremely high error rates (Scenario 3), two-level online ABFT outperforms the other two again and our basic online ABFT is unable to terminate. The overhead of online MV is 48% higher than two-level online ABFT. Observations (2), (3) and (4) are consistent with our theoretical analysis from Section 6.2.

For the performance comparison among PBiCGSTAB implementations shown in Figure 7, we observe behaviors different from those in PCG. (1) When the execution is error-free, our proposed two techniques still incur low overhead (1.0% and 4.0%, respectively), but much higher than those of PCG. This is because the overhead of checksum updates increases with more involved vectors in PBiCGSTAB. Also, our techniques' overhead is much lower than that of online MV (29% lower), indicating that the triple-checksum updates encounter much lower overhead than the more expensive binary search and partial computation. (2) In Scenario 1, two-level online ABFT has the lowest overhead: $O_2 < O_1 < O_3$, which is different from the case for PCG. This is because average execution time per iteration of PBiCGSTAB (9.1×10^{-2} seconds) is much higher than that of PCG (4.8×10^{-2} seconds). Therefore, the rollback recovery overhead in PBiCGSTAB becomes more significant than when using the triple-checksum scheme. (3) Unlike the case with PCG, basic online ABFT outperforms online MV in Scenario 2 of PBiCGSTAB. This is because the choice of $\boldsymbol{M}$

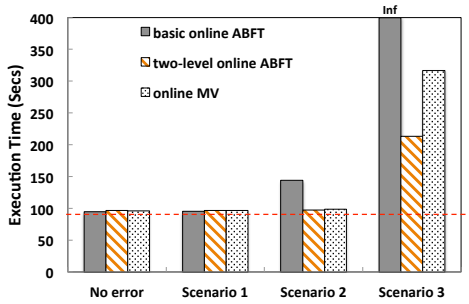

Figure 6: Performance comparison of PCG and (b) PBiCGSTAB implemented with three online ABFT techniques on Stampede. 'Inf' means it doesn't terminate. Red dotted lines represent the baseline cases.

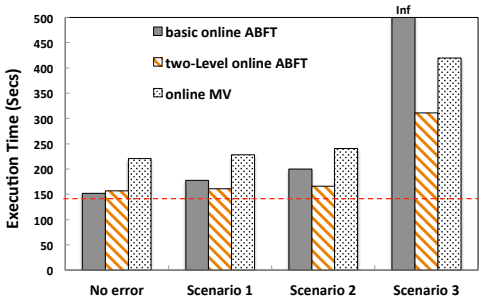

Figure 7: Performance comparison of PBiCGSTAB implemented with three online ABFT techniques on Stampede.

in PBiCGSTAB is not optimal, making PCOs to consume much higher time than MVMs. (4) The performance improvement of our two designs over online MV is more significant than that in PCG. This is because the fraction of time involving in updating the checksums shrinks as the computation intensity of the algorithm increases. This indicates that our designs will benefit computation intensive solvers with more MVMs and PCOs.

Figure 8 and 9 show the overall performance comparison between the three online ABFT implementations of PCG and PBiCGSTAB, respectively, under those three different error scenarios on the supercomputer Tianhe-2. Figure 8 and 9 indicate that the peformance overhead on Tianhe-2 is similar to Stampede.

6.3.3 *Scenario with Multiple Errors*

We evaluate our two techniques under a relatively high error-rate scenario, where multiple errors occur in different MVMs within different cd and one error occurs in a randomly selected VLO during the entire execution. This scenario is built on the rationale that MVM dominates most of the execution time in PCG, so it is highly likely that several soft errors arrive one after another, spreading out in different checkpoint intervals (cd) during the execution when error rate is high [23]. Figure 10 shows three scenarios of errors: (1) 4 errors occur one after another in four different MVMs of different checkpoint intervals; (2) 2 errors occur one after

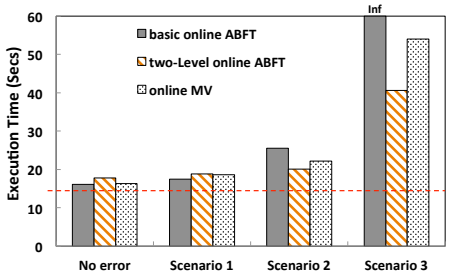

Figure 8: Performance comparison of PCG implemented with three online ABFT techniques on Tianhe-2.

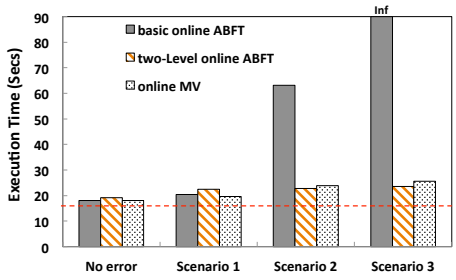

Figure 9: Performance comparison of PBiCGSTAB implemented with three online ABFT techniques on Tianhe-2.

another in two different MVMs of different checkpoint intervals; and (3) 1 error occurs in a randomly selected MVM. For the purpose of fair comparison, each of these three cases will be accompanied with a scenario in which an error occurred in a randomly selected VLO. As shown, the two-level online ABFT, on average, outperforms the basic online ABFT by 32.1% under the high error rate scenario, even though it suffers from a one-time rollback cost from the outer-level protection on the VLO's error. Furthermore, because of the inner-level protection, we can reduce the frequency of the error detection in the outer-level (i.e., increasing d), which may even gain further performance improvement. Determining d to achieve global optimal performance will be our future work.

7. RELATED WORK

Bronevetsky et al. [5], Shantharam et al. [15] and Li et al. [21] characterized and predicted the impact of soft errors on scientific applications. Di et al. [8] and Bautista-Gomez et al. [2] focused on combating SDC problems for general HPC applications. The most straightforward method to tolerate soft errors is triple modular redundancy (TMR) [12]. While generally applicable, it incurs high overheads.

Algorithm-based fault tolerance (ABFT) is a checksum-based technique developed by Huang and Abraham [11], commonly used to locate and sometimes correct soft errors for matrix operations. The basic check for absence of soft errors involves verifying if the checksum relationships in output are maintained in the final results. Using their checksum encoding mechanism, Wu et al.[23] presented a design and implementation of a fault tolerant version of the ScaLAPACK[3] to support online error detection and possibly recovery for *dense linear algebra* routines such as Cholesky, QR, and LU factorizations. This traditional checksum encoding mechanism has also been applied to the realm of iterative methods. Sloan et al. [18, 19] proposed techniques specifically for soft error detection and correction in matrix-vector multiplications (MVM), which can be applied to iterative methods under some strong assumptions (discussed in Section 2). As discussed in Sections 2 and 6, the traditional checksum-encoding mechanism cannot correctly detect soft errors in the input vectors in MVM and PCO without additional expensive verification. Even with such verification, it cannot protect the computation from input vector corruption due to cache or register errors. Chen [6] developed a highly efficient online ABFT approach for soft error detection by leveraging the orthogonality relationship of two vectors. This approach, however, only covers a subset of the Krylov methods that can offer such orthogonality. Shantharam et al. [16] proposed an ABFT-SpMxV algorithm for PCG that guarantees the detection of a single error striking in either computation or memory representation of the two input operands. This method requires $\boldsymbol{A}$ to be strictly diagonally dominant, a condition that will restrict the practical applicability of this techniques. Unlike these approaches, our proposed designs can be applied to general iterative methods without the aforementioned limitations while providing significantly improved error coverage and efficient online error detection and recovery.

In [4] and [9], the authors target GMRES based on its special characteristics and proposed a fault tolerant version via selective reliability, which can run through soft errors in the preconditioning phase without the need for detection and recovery. Similar to theirs, Sao and Vuduc[14] studied self-stabilizing corrections after error detection for CG algorithm. However, these approaches potentially affect the speed of the convergence for the underlying algorithms in an error- and input-specific fashion. Moreover, these solutions target specific features of certain iterative algorithms and are not general enough to address the larger class of iterative methods addressing by the our work.

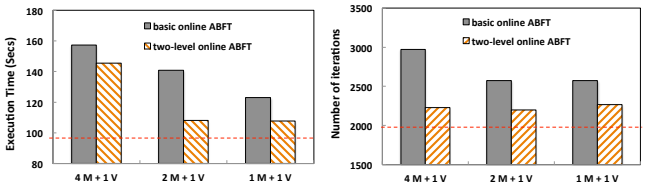

Figure 10: Performance comparison of our proposed techniques under a relatively high error-rate scenario, where multiple errors occur in different MVMs at different checkpoint intervals, and one error occurs in a randomly selected VLO.

8. CONCLUSION

To enable high error-detection coverage and low overhead, we proposed a new checksum encoding mechanism that guarantees that only the checksum relationship of the output vector needs to be verified to detect any soft error in MVM or VLO operations in iterative methods. Unlike traditional checksum schemes, our design can tolerate cache and register bit-flips and does not require additional checksum verifications after every vector-generating operation. Based on this new checksum encoding scheme, we developed two online ABFT algorithms—basic and two-level—for general iterative methods, allowing errors to be detected eagerly or lazily based on system error rates. Experimental results demonstrated that our proposed designs are efficient and effective in tolerating various error scenarios in general iterative methods.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments and valuable suggestions. This work is partially supported by the NSF grants CCF-1305622, ACI-1305624, CCF-1513201, the SZSTI basic research program JCYJ20150630114942313, and the Special Program for Applied Research on Super Computation of the NSFC-Guangdong Joint Fund (the second phase). This work was also supported in part by the U.S. Department of Energy's (DOE) Office of Science, Office of Advanced Scientific Computing Research, under awards 66905 and 59921. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830.

9. REFERENCES

[1] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, et al. PETSc Web page. http://www.mcs.anl.gov/petsc, 2015.

[2] L. A. Bautista-Gomez and F. Cappello. Detecting and correcting data corruption in stencil applications through multivariate interpolation. In *CLUSTER, Chicago, IL, USA, September 8-11, 2015*.

[3] L. S. Blackford, J. Choi, A. J. Cleary, J. Demmel, I. S. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley. Scalapack: A portable linear algebra library for distributed memory computers - design issues and performance. In *SC, November 17-22, 1996, Pittsburgh, PA, USA*.

[4] P. G. Bridges, K. B. Ferreira, M. A. Heroux, and M. Hoemmen. Fault-tolerant linear solvers via selective reliability. *CoRR*, abs/1206.1390, 2012.

[5] G. Bronevetsky and B. R. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *ICS, Island of Kos, Greece, June 7-12, 2008*.

[6] Z. Chen. Online-abft: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 167–176, 2013.

[7] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1, 2011.

[8] S. Di and F. Cappello. Adaptive impact-driven detection of silent data corruption for hpc applications. *TPDS*, to appear, 2016.

[9] J. Elliott, M. Hoemmen, and F. Mueller. Evaluating the impact of SDC on the GMRES iterative solver. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 1193–1202, 2014.

[10] M. Fasi, Y. Robert, and B. Uçar. Combining backward and forward recovery to cope with silent errors in iterative solvers. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 980–989, 2015.

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

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

[13] Y. Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[14] P. Sao and R. W. Vuduc. Self-stabilizing iterative solvers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA 2013, Denver, Colorado, USA, November 17-21, 2013*, pages 4:1–4:8, 2013.

[15] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Characterizing the impact of soft errors on iterative methods in scientific computing. In *Proceedings of the 25th International Conference on Supercomputing, 2011, Tucson, AZ, USA, May 31 - June 04, 2011*, pages 152–161, 2011.

[16] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *International Conference on Supercomputing, ICS'12, Venice, Italy, June 25-29, 2012*, pages 69–78, 2012.

[17] G. L. Sleijpen and D. R. Fokkema. Bicgstab (l) for linear equations involving unsymmetric matrices with complex spectrum. *Electronic Transactions on Numerical Analysis*, 1(11):2000, 1993.

[18] J. Sloan, R. Kumar, and G. Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012, Boston, MA, USA, June 25-28, 2012*, pages 1–12, 2012.

[19] J. Sloan, R. Kumar, and G. Bronevetsky. An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*, pages 1–12, 2013.

[20] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Fingerprinting: Bounding soft-error detection latency and bandwidth. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, pages 224–234, 2004.

[21] L. Tan, S. L. Song, P. Wu, Z. Chen, R. Ge, and D. J. Kerbyson. Investigating the interplay between energy efficiency and resilience in high performance computing. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 786–796, 2015.

[22] S. Tavarageri, S. Krishnamoorthy, and P. Sadayappan. Compiler-assisted detection of transient memory errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 24, 2014.

[23] P. Wu and Z. Chen. FT-ScaLAPACK: correcting soft errors on-line for ScaLAPACK cholesky, QR, and LU factorization routines. In *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, pages 49–60, 2014.