2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

Efficient Algorithm-Based Fault Tolerance for Sparse Matrix Operations

Alexander Schöll, Claus Braun, Michael A. Kochte and Hans-Joachim Wunderlich
Institute of Computer Architecture and Computer Engineering, University of Stuttgart
Pfaffenwaldring 47, D-70569, Germany, Email: {schoell,braun,kochte,wu}@informatik.uni-stuttgart.de

***Abstract*—We propose a fault tolerance approach for sparse matrix operations that detects and implicitly locates errors in the results for efficient local correction. This approach reduces the runtime overhead for fault tolerance and provides high error coverage. Existing algorithm-based fault tolerance approaches for sparse matrix operations detect and correct errors, but they often rely on expensive error localization steps. General checkpointing schemes can induce large recovery cost for high error rates. For sparse matrix-vector multiplications, experimental results show an average reduction in runtime overhead of 43.8%, while the error coverage is on average improved by 52.2% compared to related work. The practical applicability is demonstrated in a case study using the iterative Preconditioned Conjugate Gradient solver. When scaling the error rate by four orders of magnitude, the average runtime overhead increases only by 31.3% compared to low error rates.**

Keywords*—*Fault Tolerance, Sparse Linear Algebra, ABFT, Online Error Localization

I. Introduction

Large-scale applications in science and engineering benefit from the increasing compute power of heterogeneous computing systems comprising multi-core CPUs and many-core GPUs. Such systems became widespread in the high performance computing domain as they allow to substantially accelerate compute-intensive tasks from different fields like molecular biology [1, 2], electronic design automation [3], thermodynamics [4], as well as data analysis and machine learning [5, 6].

However, such modern nano-scaled CMOS devices become increasingly vulnerable to a growing spectrum of reliability threats including transient events, latent defects, stress and aging mechanisms, as well as marginal hardware due to process variations [7, 8]. Transient effects that are exposed to the applications on these systems may cause crashes or Silent Data Corruptions (SDC). SDCs are commonly considered more severe, as there is no indication that errors corrupted the result [9]. Future manufacturing processes will allow even smaller chip feature sizes resulting in an increased vulnerability, which makes the integration of fault tolerance mandatory. For example, foreseeable high-performance computing systems are predicted to encounter errors every couple of minutes [10].

Different hardware-based fault tolerance approaches such as redundancy or guard banding are applied to mask errors. However, such techniques dissipate much power and are not suitable for highly integrated solutions [11]. Therefore, a growing number of transient effects will be exposed to the software which has to tolerate them. Future software applications must therefore be capable of detecting errors and recovering from them [12, 13].

Traditional checkpointing techniques are a mature approach to tolerate errors in such applications. These techniques write the state of an application periodically to a reliable storage and restart the application from a prior state if an error is detected. However, this approach can induce high costs in both transferring checkpoint data and recomputing lost results. These costs might be acceptable when errors are rare, but become infeasible in the near future when error rates increase with smaller chip feature sizes. Therefore, checkpointing techniques will become increasingly impractical as they induce significant bottlenecks for the execution of applications [10, 14].

Algorithmic error detection and correction approaches have been proposed which correct erroneous application outputs. Such approaches avoid the cost induced by rolling back to a prior state. Typically, only a small portion of the output is corrupted by errors, even under high error rates. Therefore, the error correction cost can be reduced, if only the erroneous part is corrected. This *partial recomputation* approach, however, requires the localization of errors in the output to avoid unnecessary correction cost. Different error detection techniques, such as assertions [15], check for the presence of errors in application outputs. However, if only the error presence is known, then either a complete recomputation has to be performed or the location of the errors has to be identified to allow partial recomputation. Additional error localization costs may be acceptable for small outputs, but become unacceptable for large output sizes.

We propose a fault tolerance approach that allows the efficient algorithmic detection and correction of erroneous application outputs with high error coverage. Instead of only detecting errors, our approach instruments error detection steps to implicitly provide error locations. This enables partial recomputations just for erroneous outputs directly after error detection, which avoids both expensive error localization steps as well as repeating entire computations. The proposed fault tolerance technique is algorithm-based and exploits the properties of specific algorithms to identify error locations during error detection steps. Since our approach avoids both redundant recomputations and expensive error localization steps, it significantly reduces the runtime overhead for fault tolerance.

In this work, we apply this scheme to sparse matrix operations on heterogeneous computing systems under high error rates. We evaluate our scheme for the sparse matrix-vector multiplication (SpMV) and present a case study for iterative linear solvers in which SpMV constitutes a dominating subroutine. Both the SpMV operation as well as iterative linear solvers are crucial tasks in many high performance computing applications in science and engineering [16–18]. As the SpMV operation is a numerical task, which is typically performed using floating point arithmetic, additional challenges arise to detect errors reliably due to rounding errors. We address this issue and present a novel analytical error function that provides suitable rounding error bounds for the SpMV operation. This paper presents the following contributions:

- A fault tolerance approach that allows the efficient algorithmic detection and correction of corrupted application outputs, i.e. the result vector of SpMV operations. This approach instruments error detection steps to identify corrupted output parts. Therefore, this approach is more favorable for high error rates compared to related works that perform error detection and localization separately.
- An analytical error function for SpMV operations that provides suitable rounding error bounds to distinguish errors in the magnitude of the rounding error and errors that may be harmful to the application.
- Evaluation of our fault tolerance approach and presentation of experimental results for the SpMV operation in terms of *runtime overhead* for both *error detection* and *error correction* as well as achievable *error coverage*. Experimental results show that the runtime overhead for fault tolerance (i.e. error detection and correction) ranges from 13.6% to 155.7%. While the runtime overhead is reduced by 43.8% compared to related work approaches, the error coverage is on average improved by 52.2% (cf. Section V).
- A case study, in which our fault tolerance approach for the SpMV operation is evaluated for a widely-used iterative solver, the *Preconditioned Conjugate Gradient* method [19]. In this case study, we evaluate runtime overheads required for finding correct results as well as the error coverage. Experimental results are presented for different error scenarios ranging from low to high error rates. Under high error rates, our approach reduces the runtime overhead on average by 40.1% compared to related work. At the same time, the number of successful solver runs (i.e. that provided a correct result) is on average increased by 61.6%. Compared to traditional checkpoint-rollback techniques, our approach allows on average 3.6 times more successful solver runs.

The remainder of this paper is organized as follows. Section II presents an overview of related work and discusses prior fault tolerance techniques. Section III explains our proposed fault tolerance approach for sparse matrix operations. The experimental setup is presented in Section IV. Section V presents the results for the evaluation of the SpMV operation. The case study in Section VI demonstrates the practical applicability of our approach.

II. Related Work

Traditional fault tolerance techniques like modular redundancy and checkpointing are widely used in the scientific computing domain [20]. Redundant execution techniques such as triple modular redundancy (TMR) are applied to provide fault tolerance for highly critical applications [21]. However, duplication or even triplication of procedures induce high costs in power, energy, and throughput.

Over the past decades, different checkpointing and rollback approaches were proposed and are nowadays commonly used to provide fault tolerance in high-performance computing systems [10, 14, 22]. These approaches periodically write the application state to a storage that is considered to be reliable [23]. If an error is detected within a checkpointing interval, then the system restarts the application and restores the prior application state. However, such checkpointing approaches can induce large runtime overheads under high error rates since the recomputation of lost results has to be performed frequently. If such errors persist in the application and become detectable in succeeding checkpointing intervals, then restarts from prior *corrupted* checkpoints may lead to endless loops [24]. In [25], shortening of checkpointing intervals is proposed to reduce recomputation cost under high error rates. However, this leads to increasing checkpointing overheads, as additional runtime and bandwidth is required to store the application state more frequently.

Different algorithm-based fault tolerance approaches (ABFT) were proposed for *dense* linear algebra operations such as matrix multiplications or decompositions [26–28]. ABFT techniques encode input data by adding checksums before performing a linear operation and calculate new checksums for the results. The results are checked for errors by evaluating invariants between checksums that were processed by the operation and the checksums computed for the results.

For the matrix-vector multiplication $Ab = r$, the checksum invariant can be expressed as

$$w^T(Ab) = (w^T A)b \tag{1}$$

which is based on the associative property for matrix multiplication. Here, w^T is a weight vector, A is the input matrix and b is the operand vector. In case of errors, equation 1 will most likely no longer hold. This *dense check* can be divided into two parts. The first part comprises the setup which is executed once for each matrix A to obtain a *dense column checksum vector* (i.e. $c = (w^T A)$). The second part comprises the operations for error checking, which requires two inner products. With matrix size n, $\mathcal{O}(n^2)$ computations are required for setup, but only $\mathcal{O}(n)$ computations must be performed to check for errors. This is efficient for dense matrices since $\mathcal{O}(n^2)$ computations are induced by the multiplication itself.

Besides dense linear operations, sparse systems and sparse linear algebra are important operations for many applications in scientific computing and in engineering [16–18]. The direct use of such dense checks for sparse operations induces significant runtime overheads as, for instance, the SpMV operation requires only $\mathcal{O}(n)$ computations. Since the SpMV operation and the dense error check are in same order of time complexity (i.e. $\mathcal{O}(n)$), the runtime overhead for error detection may exceed the runtime of the original operation.

Nonetheless, this dense check is applied in different related works to detect errors in sparse linear operations such as the SpMV operation [29, 30] or triangular solvers [31]. Different approaches were proposed to reduce the overhead for sparse linear operations. A checksum encoding is proposed in [32] for the SpMV operation that omits a fraction of the matrix columns in the checksum generation. This approach reduces the runtime overhead for error detection, but also reduces the error coverage. In [30], an error localization scheme is proposed for the SpMV operation that reduces the runtime overhead for error correction by avoiding complete recomputations. After an error is detected by the dense check, errors are located by an iterative bisection technique, which repeatedly divides the matrix and checks for errors in the submatrices. However, the runtime overhead to provide fault tolerance still depends on the dense ABFT check, since the result of this check is required before any error localization steps can be performed.

Since sparse linear operations are typically performed using floating-point arithmetic, rounding errors occur. Therefore, error checking approaches need to check invariants under consideration of a threshold τ (i.e. the rounding error bound) to avoid false positive error detections. Rounding errors typically cause small differences in these checksums which makes a direct comparison of checksums impossible. One approach is to let the user determine such thresholds manually [26]. However, this approach requires both deep knowledge of the input data and re-calibration for each new problem set. Different approaches were proposed that determine such rounding error bounds with respect to the input data at runtime. Analytical bounds for rounding errors were derived for different linear operations [33, 34]. These were used in [35] to obtain an analytical rounding error bound to protect dense matrix vector multiplications. For dense matrix operations, probabilistic rounding error functions were proposed in [28].

In summary, traditional checkpointing approaches induce large runtime overheads since recomputation of lost results is performed frequently under high error rates. For sparse linear operations, the direct application of traditional dense checks is highly inefficient. The approaches [30–32] reduce runtime overheads for error detection and correction steps separately. However, the utilized error detection schemes only indicate the existence of errors and not their location. Therefore, either complete recomputations or additional error localization steps need to be performed to correct errors.

III. Efficient Algorithm-Based Fault Tolerance for Sparse Matrix Operations

In this work, we propose an efficient fault tolerance approach that detects and corrects erroneous application outputs with low runtime overhead. This approach is based on the observation that even under high error rates, errors typically do not affect complete application outputs, but only small parts. Instead of performing expensive error localization steps, we propose novel error detection steps that implicitly and highly efficiently provide the error locations. This approach allows to correct erroneous outputs directly, which reduces the overall runtime overhead to provide fault tolerance. At the same time, this approach does not rely on checkpointing techniques, which makes it favorable for high error rates.

We explore this approach with respect to matrix operations, focusing on the sparse matrix-vector multiplication (SpMV). SpMV operations are crucial parts for many compute-intensive tasks in the high-performance computing domain (cf. Section III-E).

A. Algorithm-Based Fault Tolerance for Sparse Matrix-Vector Multiplications

Our fault tolerance scheme divides the SpMV operation into *small blocks* and performs checksum-based error detection for each block separately. This technique was evaluated for dense matrix multiplications in [36]. Instead of locating the affected result element exactly, our approach delimits error locations to blocks of result elements. Errors are corrected by recomputing the SpMV operation *partially* for erroneous blocks. Therefore, both complete recomputation and error localization steps are avoided since error locations are already determined during error detection. At the same time, the runtime overhead to detect and locate errors is significantly reduced since this block-based approach exploits the sparsity of the underlying input matrix.

Our approach encodes the input matrix A to obtain a *sparse checksum matrix* C. The checksum matrix C inherits the sparsity of the input matrix A. Each row of a checksum matrix C contains the column checksums for a specific block of the input matrix A. By exploiting this sparsity, our block-based approach reduces the runtime overhead to detect errors compared to prior approaches [30–32] that apply dense checksum vectors. To detect errors in the results of a SpMV operation, new checksums are computed for the results and compared to the checksums that where processed by the SpMV operation. Our error detection approach also determines *analytical error bounds* for each SpMV operation to distinguish errors in the magnitude of the rounding error and errors that may be harmful to the application. Therefore, we significantly increase the error coverage compared to dense checks.

Figure 1 shows the overview of the algorithmic steps that are performed for each fault-tolerant sparse matrix-vector multiplication. This figure indicates operations that can be executed in parallel to each other by depicting such operations in common rows. In the first step, the SpMV operation $Ab = r$ itself is computed to obtain the result vector r. Parallel to the SpMV operation, the operand checksum vector t_1 is computed as $Cb = t_1$.

The second and third steps calculate error detection criteria that are required to check the results. In the second step, the operand norm β is computed which is required to determine the rounding error bounds for error detection. Parallel to this step, new checksums are obtained for the results by computing the result checksum vector t_2 as $t_{2_k} = w_k^T r_k$. As explained below, w_k and r_k correspond to the k-th block in weight vector w and in result vector r. In the third step, the syndrome vector is computed by calculating the differences between the result checksum vector t_1 and the operand checksum vector t_2. Errors are detected in the fourth step by comparing each element of the syndrome vector s against the rounding error bound. The location of errors is provided during this error detection step by determining the portion of blocks for which the syndrome exceeds rounding error bounds. Such result blocks contain at least one erroneous element and are therefore separately recomputed in the fifth step to correct these errors. Therefore, the SpMV operation is recomputed only *partially* in case of errors.

B. Error Detection

The SpMV operation is divided into blocks of size b_s and error detection steps are performed for each block separately. To divide the SpMV operation into blocks, the underlying input matrix A is decomposed into rows of block matrices A_k (i.e. the k-th row block in A). After decomposition, each block matrix A_k comprises at most b_s rows. In the example below, the 6×6 sparse matrix A is decomposed into three block

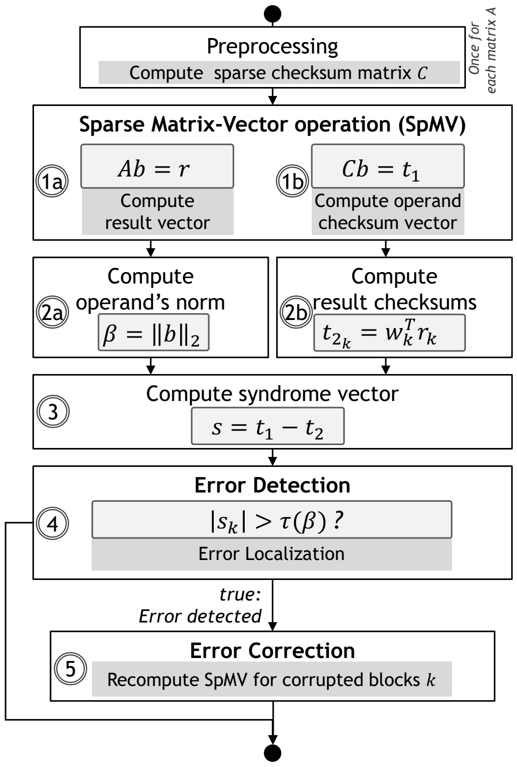


Fig. 1: Overview of the proposed fault-tolerant sparse matrix-vector multiplication.

matrices A_1, A_2 and A_3 with block size $b_s = 2$. [1]

$$A = \begin{bmatrix} 1 & & & & & 2 \\ & 5 & 2 & & & \\ & 2 & 3 & 5 & & \\ & & 5 & 4 & & 1 \\ & & & & 2 & \\ 2 & & & 1 & & 9 \end{bmatrix} \Rightarrow A_k: \quad A_1 = \begin{bmatrix} 1 & & & & & 2 \\ & 5 & 2 & & & \end{bmatrix}, \quad A_2 = \begin{bmatrix} & 2 & 3 & 5 & & \\ & & 5 & 4 & & 1 \end{bmatrix}, \quad A_3 = \begin{bmatrix} & & & & 2 & \\ 2 & & & 1 & & 9 \end{bmatrix}$$

To detect errors in the result of SpMV operations, the following checksum invariant is evaluated for each block k:

$$w_k^T(A_k b) \approx (w_k^T A_k)b \qquad (2)$$

The right-hand side of the equation encodes the block matrices A_k using weight vectors w_k to obtain column checksums (i.e. $c_k = w_k^T A_k$) for each block. Checksums are computed for the SpMV operation by multiplying the column checksums c_k by the original operand b. In our error detection scheme, we combine those vectors to obtain the sparse checksum matrix C. Instead of multiplying each column checksum vector c_k by the original operand b separately, we multiply the checksum matrix C with b to obtain the *operand* checksum vector t_1.

The left-hand side of the equation decomposes the result vector r into row blocks r_k (i.e. $r_k = A_k b$) with block size b_s and computes new checksums for the results using the weight vectors w_k. The resulting checksums are combined in the *result* checksum vector t_2. In the error-free case, the checksum invariant holds for all blocks, i.e. the checksum vectors t_1 and t_2 are equal aside from rounding errors. Errors are detected by evaluating the difference in the invariant against a certain rounding error threshold τ.

Parallel to the detection of errors, the locations of errors in the result vector are delimited by such blocks for which the checksum invariant does not hold. For example, consider the 6×6 sparse matrix A that is multiplied by the operand vector b to obtain the result vector r.

$$\underset{A}{\begin{bmatrix} 1 & & & & & 2 \\ & 5 & 2 & & & \\ & 2 & 3 & 5 & & \\ & & 5 & 4 & & 1 \\ & & & & 2 & \\ 2 & & & 1 & & 9 \end{bmatrix}} \cdot \underset{b}{\begin{bmatrix} 2 \\ 4 \\ 3 \\ 5 \\ 4 \\ 1 \end{bmatrix}} = \underset{r}{\begin{bmatrix} 4 \\ 26 \\ 42 \\ 36 \\ 8 \\ 18 \end{bmatrix}}$$

To detect errors in this SpMV operation, the matrix A is decomposed into three row blocks of block size $b_s = 2$ and encoded by weight vectors w_k for each block. In this example, the weight vectors are set to $(1, 1)$.

$$\begin{array}{ccccc} w_k & & A_k & & c_k \\ \begin{bmatrix} 1 & 1 \end{bmatrix} & \cdot & \begin{bmatrix} 1 & & & & & 2 \\ & 5 & 2 & & & \end{bmatrix} & = & \begin{bmatrix} 1 & 5 & 2 & & & 2 \end{bmatrix} \\ \begin{bmatrix} 1 & 1 \end{bmatrix} & \cdot & \begin{bmatrix} & 2 & 3 & 5 & & \\ & & 5 & 4 & & 1 \end{bmatrix} & = & \begin{bmatrix} & 2 & 8 & 9 & & 1 \end{bmatrix} \\ \begin{bmatrix} 1 & 1 \end{bmatrix} & \cdot & \begin{bmatrix} & & & & 2 & \\ 2 & & & 1 & & 9 \end{bmatrix} & = & \begin{bmatrix} 2 & & & 1 & 2 & 9 \end{bmatrix} \end{array}$$

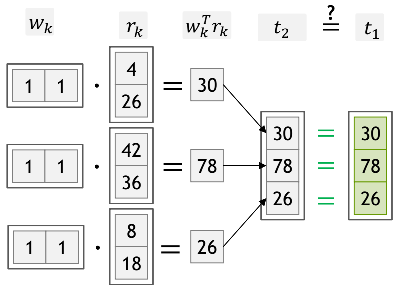
The resulting three column checksum vectors c_k have to be computed once for each input matrix A. For weight vectors $(1, 1)$, each element of the checksum vectors c_k contains the sum of a column of A_k. The checksum matrix C is obtained by combining the checksum vectors c_k. The sparsity of the input matrix A is passed to the checksum matrix C. To compute the operand checksum vector t_1, the checksum matrix C is multiplied by the operand b.

$$\underset{C}{\begin{bmatrix} 1 & 5 & 2 & & & 2 \\ & 2 & 8 & 9 & & 1 \\ 2 & & & 1 & 2 & 9 \end{bmatrix}} \cdot \underset{b}{\begin{bmatrix} 2 \\ 4 \\ 3 \\ 5 \\ 4 \\ 1 \end{bmatrix}} = \underset{t_1}{\begin{bmatrix} 30 \\ 78 \\ 26 \end{bmatrix}}$$

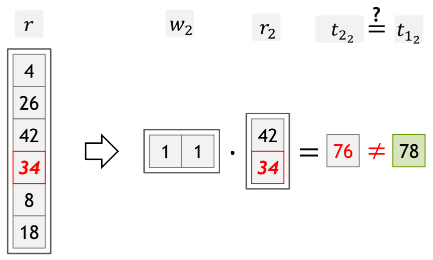
New checksums are computed for the results by decomposing the result vector r into row blocks r_k of block size $b_s = 2$ and encoding those blocks using the weight vectors

[1]The example uses integer values for illustration, but typically floating-point values are used in SpMV operations.

w_k. For each block, one checksum $w_k^T r_k$ is computed. The different result checksums are combined in the result checksum vector t_2 which is compared to the operand checksum vector t_1 to detect errors. In this example, no error occurred, and the checksum invariant holds for all blocks.



Assume an error corrupted the fourth element in the result vector r. Here, the corruption induced an offset of 2. This error is both detected and located by evaluating the checksum invariant for the second block A_2. The result checksum for the second block t_{2_2} differs to the operand checksum t_{1_2}.



Errors are not only detected by our error detection scheme but also located. In this example, the error location is delimited to the second block since the checksum invariant does not hold for this block. To correct this error, only the results for the second block have to be recomputed by performing an SpMV operation partially for rows 3 and 4 of input matrix A.

As mentioned above, by exploiting the sparsity of the checksum matrix C on heterogeneous computing systems the runtime overhead to detect and locate errors is significantly reduced compared to prior approaches that utilize dense checksum vectors to protect the SpMV operation [30–32] .

The sparse checksum matrix C needs to be computed for each matrix A only once. Before any checksum values can be derived, the data structure for matrix C needs to be derived from the data structure of input matrix A. Figure 2 presents the necessary steps to derive the data structure at the example of an input matrix A of size 20×20.

In the first step, the data structure of A is decomposed into row blocks A_k. In this example, the block size b_s is chosen to be 5. For each row block, non-empty columns are identified in the second step. Checksum elements are stored in a row of C, only if the corresponding column of A contains at least one element. The resulting data structure for the checksum matrix C is depicted in the third step.

The size of the blocks b_s determines the runtime overhead to detect and correct errors. For the error detection steps, the *block size* b_s trades-off the runtime for computing the operand

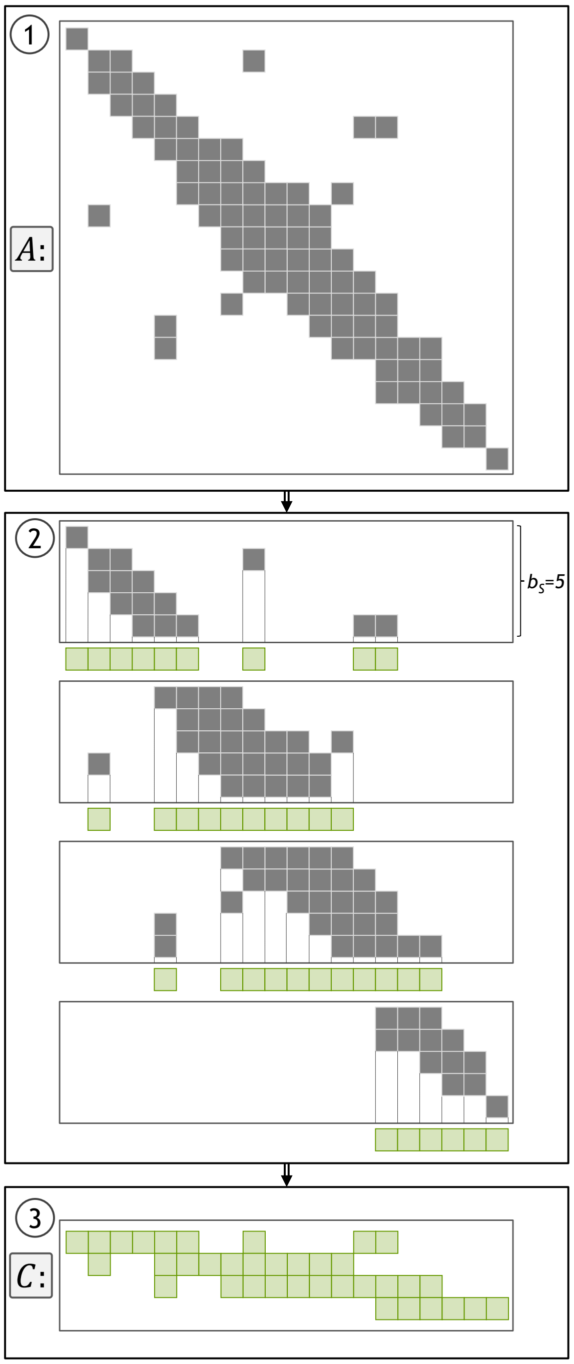


Fig. 2: Decomposition of a sparse matrix A to obtain the data structure of the sparse checksum matrix C.

checksum vector t_1 against the runtime for the computing the result checksum vector t_2. For large blocks, the checksum matrix C contains fewer rows which reduces the runtime for computing t_1 (i.e. $t_1 = Cb$). However, the number of elements that have to be processed for each element of t_2 increases. Each element is computed as an inner product which is typically implemented as a *reduction* on parallel computer architectures [37]. With increasing number of elements in each inner product, the number of sequential reduction steps increases. Smaller blocks result in larger checksum matrices C but reduce the runtime for computing the result checksum vector t_2.

C. Rounding Error Bounds

The SpMV operation is typically performed using floating point arithmetic which is prone to rounding errors. Error detection schemes need to consider the impact of rounding errors as they typically cause small differences in the checksums. Therefore, direct comparisons of checksums can cause false positives. Instead, suitable rounding error bounds τ have to be determined that cope with differences caused by rounding errors. Error bounds that were chosen smaller than the actual rounding error lead to false positive error detections, which will trigger unnecessary corrections. Too large bounds increase the number of undetected errors (false negatives) that may propagate to the final result in the application. For instance, in the case of iterative solvers, such errors may significantly increase execution times or lead to silent data corruption [29, 38].

In [35], an analytical rounding error bound is presented to protect dense matrix vector multiplications:

$$\begin{aligned}|t_1 - t_2| < \tau(\beta) :=& \\ \Big((n + 2m - 2) \cdot \sum_{i=1}^{m} ||a_i||_2 +& \\ n \cdot ||c||_2\Big) \cdot \epsilon_M \cdot \beta&\end{aligned}$$

with $\beta := ||b||_2$. This bound estimates the maximum difference between the operand and result checksums for a dense matrix A with m rows and n columns with a machine precision of ϵ_M (i.e. $\epsilon_M = 2^{-53}$ for double precision floating-point values). Such maximum differences are constituted by the norms of operand b, checksum vector c as well as the rows in input matrix A (i.e. $||a_i||_2$). Sparse matrices contain a large portion of zero elements which do not contribute to the rounding error. Therefore, the maximum difference between the operand and result checksums is typically smaller and the error bounds derived by this approach are too loose for sparse problems.

We propose the following analytical rounding error bound for sparse matrix-vector multiplications:

$$\begin{aligned}|t_{1_k} - t_{2_k}| < \tau(\beta) :=& \\ \Big((n_k + 2b_s - 2) \cdot \sum_{i=(k-1)\cdot b_s+1}^{k\cdot b_s} ||a_i||_2 +& \\ n_k \cdot ||c_k||_2\Big) \cdot \epsilon_M \cdot \beta&\end{aligned}$$

with $\beta := ||b||_2$. Instead of assuming each block A_k to cover all n columns, the actual number of non-empty columns n_k is utilized to estimate the maximum difference between the operand and result checksums. Since $n_k < n$ for sparse matrices, this estimation provides tighter error bounds since it considers the actual number of elements that contribute to the rounding error. Therefore, our approach provides a significantly increased error coverage (cf. Section V).

D. Parallel Implementation

Our proposed fault-tolerant SpMV operation relies on SpMV operations itself and on inner products. Such linear algebra operations are parallelizable, which allows the execution of such operations on heterogeneous systems with multi-core CPUs and many-core GPUs. Different parallelization techniques exist for both inner products and the SpMV operation [37]. Typically, inner products are implemented as parallel reductions which have a runtime complexity of $\mathcal{O}(log_2(N))$. Different operations of our approach can be parallelized to each other. As explained before, Figure 1 shows such operations that are independent and therefore parallelizable (e.g. by using task parallel programming techniques).

The computation of the checksum matrix C requires additional operations that are not covered by inner products or the SpMV operation. To compute the checksum matrix C, two steps are required as depicted in Figure 3.

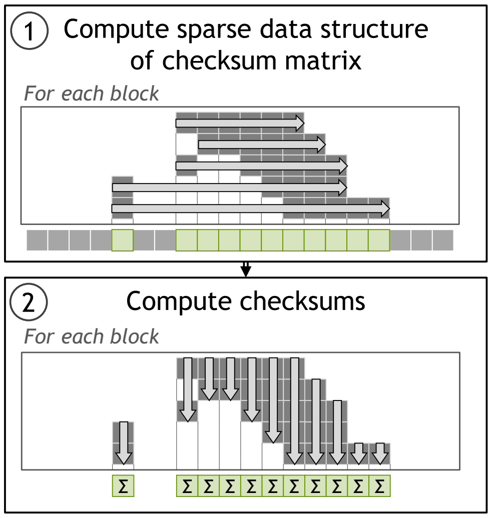


Fig. 3: Computation of data structure and checksums for each block in the checksum matrix C.

In the first step, the sparse data structure of matrix C has to be determined. For each block, the data structure of input matrix A to obtain the number and the indexes of non-zero columns. This step can be parallelized for each row in input matrix A.

After this step, the data structure of matrix C is used to compute the checksums for each block and each checksum element. The checksums are computed by multiplying the elements in row block A_k with the corresponding elements in the weight vector w_k. The computation of checksums can be parallelized for each column in A_k.

E. Generality

Both the SpMV operation as well as iterative linear solvers are important tasks for many large-scale applications in science and engineering such as structural mechanics [16], computational fluid dynamics [17], or the study of electromagnetic fields [18]. Our proposed fault tolerance technique exploits both the associativity of the SpMV operation as well as its decomposability into suboperations to apply online error localization. For this reason, our approach can be applied to any application that relies on associative linear operations which are decomposable and comprise multiple result elements. Since many different applications in science and engineering include such linear operations, this approach can be widely adopted to exploit its benefits.

Applications that reuse data repeatedly achieve even larger benefits since preprocessing only needs to be performed once for each input data (i.e. computation of checksum matrices). An important class of such applications are iterative solvers that typically dominate the runtime for many scientific applications. Such iterative solvers are also used in different computing domains besides high-performance computing, for instance, to process video streams in real time [39]. We evaluate such benefits for the iterative PCG solver in Section VI.

A different important class of applications which heavily rely on both linear systems and linear operations are data analysis and machine learning applications. In machine learning, *support vector machines* utilize linear operations such as the SpMV operation for classification and regression analysis [6]. Different graph-based applications such as *PageRank* and *Random Walk with Restart* rely on linear operations as well [40]. Since these applications use linear operations at their core, our approach can be directly applied to provide efficient fault tolerance.

IV. Experimental Setup

The proposed fault tolerance technique was evaluated for the SpMV operation itself and for iterative linear solvers which use SpMV as an internal subroutine. The experimental setup including the error model, utilized benchmarks and the system setup are presented below. Section V presents the experimental results with respect to the SpMV operation. A case study is presented in Section VI, in which our fault tolerance approach for the SpMV operation is evaluated for one of the most important iterative solvers, the Preconditioned Conjugate Gradient method (PCG) [19].

A. Error Model

The experimental evaluation focuses on transient events that constitute errors in outputs of arithmetic computations. Erroneous arithmetic outputs may lead to Silent Data Corruptions (SDC) if they remain undetected. Such corruptions of outputs may occur in the arithmetic components of a processor due to the manifestation of faults in form of soft errors. We do not consider errors in memory elements since hardware fault tolerance techniques like ECC [20] are typically used in high-performance systems to protect main memories, caches and register files. We also do not consider errors in the control logic or in the encoding of instructions as we assume them to be protected by appropriate measures (e.g. error-detecting/correcting codes, signature monitoring, or assertion techniques [41–43]).

Different implementations of floating-point units exist that may have different error propagation patterns for transient events. In accordance to related works [24, 30–32], errors are injected into computations at runtime by instrumenting the application to determine random error injections. During execution, errors are injected into a randomly selected element within the result vector of the SpMV operation. The results of the underlying floating point instructions are modified by randomized bursts of bidirectional bit flips. The position of a burst is randomly chosen from a uniform distribution within the 64 bits of the floating-point values. The number of bits affected by such bursts are randomly chosen from a normal distribution with mean = 3 and variance = 2. Bit flips were also injected into operations that perform error detection.

B. Benchmarks

As benchmarks, 25 matrices from the Florida Sparse Matrix Collection [44] were evaluated which are shown in Table I. Besides the names and sizes of the matrices ($N \times N$), the number of nonzero elements (NNZ) is presented. As a side information, the portion of 0s within the matrices is shown. The matrices have the following properties: square, symmetric, real and positive definite. The evaluated matrices were stored in the compressed sparse row storage format [45].

TABLE I: Overview of evaluated matrices from the Florida Sparse Matrix Collection [44].

Name	N	NNZ	Portion of 0s
nos3	960	15844	98.28%
bcsstk21	3600	26600	99.79%
bcsstk11	1473	34241	98.42%
ex3	2410	54840	99.06%
ex10hs	2548	57308	99.12%
nasa2146	2146	72250	98.43%
sts4098	4098	72356	99.57%
bcsstk13	2003	83883	97.91%
msc04515	4515	97707	99.52%
ex9	3363	99471	99.12%
aft01	8205	125567	99.81%
bodyy6	19366	134208	99.96%
Muu	7102	170134	99.66%
s3rmt3m3	5357	207123	99.28%
s3rmt3m1	5489	217669	99.28%
bcsstk28	4410	219024	98.87%
s3rmq4m1	5489	262943	99.13%
bcsstk16	4884	290378	98.78%
bcsstk38	8032	355460	99.45%
msc23052	23052	1142686	99.78%
msc10848	10848	1229776	98.95%
nd3k	9000	3279690	95.95%
ship_001	34920	3896496	99.68%
hood	220542	9895422	99.98%
crankseg_1	52804	10614210	99.62%

For the experiments, both the SpMV operation and the PCG algorithm were tailored to a heterogeneous computing system comprising multi-core CPUs and many-core GPUs. All parallelizable linear algebra operations were mapped to a GPU architecture and GPU-accelerated linear algebra libraries were utilized. All experiments have been performed in double precision.

The hardware platform consists of two Intel Xeon E5-2623 with 3.00 GHz and 128 GB RAM. The system hosts three Nvidia Tesla K80 GPUs with 2×2496 processing cores at 745MHz and 24 GB GDDR5 RAM per device. ECC protection was enabled for all experiments protecting the register files, cache and DRAM. The machine runs a linux operating system with CUDA version 7.5 and a GNU GCC 4.4.7 compiler tool chain. Each experiment was executed using a combination of a single CPU core and a single GPU.

V. Results

We evaluated our fault tolerance approach for the SpMV operation with respect to the *runtime overhead* for both *error detection* and *correction* as well as the achievable *error coverage*. Our error detection scheme is compared to the *dense check* approach as proposed in [30, 31]. Our error correction scheme is compared to *complete* [31] and *partial* recomputation approaches [30]. For the partial recomputation approach, we adopted the error localization steps described in [30] (i.e. *iterative bisection technique* with early stop at 40% of complete localization traversal).

A. Runtime Overhead

To investigate the runtime overheads induced by the different methods, we compared the runtime of protected SpMV operations to the runtime of original SpMV operations:

$$\text{Runtime overhead} = \frac{\text{Runtime for protected SpMV}}{\text{Runtime for original SpMV}} - 1$$

The *runtime overhead for error detection* that is induced by our method depends on the block size b_s. We evaluated the runtime overhead for different block sizes ranging from 1 to 512. Figure 4 shows the results of this investigation. Each data point represents a result for a specific matrix. The red graph depicts the average runtime overhead over all evaluated matrices. For each matrix, 100,000 experiments were performed. The average runtime overhead is ranging from 43.0% for block size 32 to 83.7% for block size 1. For block size 1, the checksum matrix C is equal to the input matrix A. Therefore, the computation of both the original SpMV (Ab) and the checksums (Cb) have equal complexity. Runtime overheads below 100% can be explained by the parallel execution of those two operations. All evaluations below were performed using block size 32 since this block size provides the minimum error detection overhead.

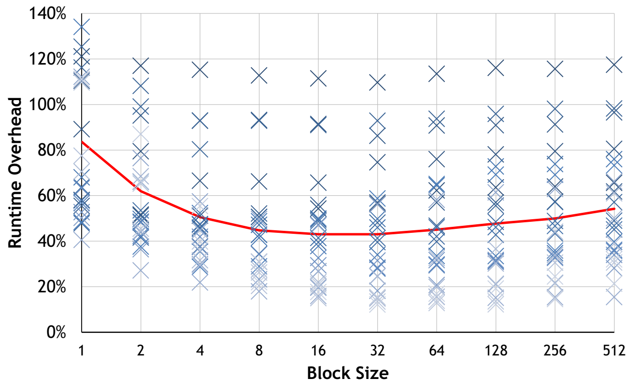


Fig. 4: Runtime overhead of SpMV for different block sizes compared to unprotected executions.

The error detection overhead is shown for each matrix in Figure 5. We compare the error detection overhead of our method using block size 32 against the *dense check* approach [30, 31]. For fair comparison, the generation of checksums in the dense technique ($c^T b$) was also parallelized with the original SpMV operation (Ab). The matrices are ordered by increasing size of NNZ (cf. Table I). With increasing matrix sizes, the overhead of all methods decreases. The error detection overhead for our method ranges from 12.1% to 109.6%. Compared to dense checks, the runtime overhead is on average reduced by 50.79%. The minimum reduction is 19.3% with matrix *s3rmq4m1* and the maximum reduction is 82.1% with matrix *msc10848*.

We evaluated the runtime overhead for both *detecting and correcting errors* by injecting errors into SpMV operation and comparing the runtime to the unprotected SpMV operation. We compare our method to *complete* [31] and *partial* recomputation approaches [30] which both rely on dense checks to detect errors. For each matrix and each evaluated method, 100,000 error injection experiments were performed which triggered error corrections in all evaluated methods. Figure 6 shows the results of this investigation. The overhead for both error detection and correction ranges from 13.6% to 155.7% for our method. The runtime overhead is on average reduced by 43.8% compared to the error localization and partial recomputation technique [30] and by 55.7% compared to complete recomputations [31]. The minimum reduction is 19.9% with matrix *Muu* and the maximum reduction is 69.9% with matrix *ship_001* compared to the related work methods.

B. Error Coverage

The error coverage of a method is constituted by the number of successfully detected errors (*true positives*), the number of undetected errors (*false negatives*) and the number of mistakenly identified errors (*false positives*). To evaluate and compare the effectiveness of our method against dense checks, we performed 100,000 error injection experiments for each matrix and method. In each injection experiment, a random result element was selected and errors were injected into the underlying floating-point value as described in Section IV-A. We apply the rounding error bound as presented in Section III-C. For the dense check approach, the *norm-based rounding error bound* $\tau = ||b||_2$ is applied as proposed in [30] (i.e. $|w^T r - w^T Ab| < ||b||_2$). From the experiment results, we compute the balanced F$_1$-score [46]:

$$F_1 = \frac{2 \cdot \text{true positives}}{2 \cdot \text{true positives} + \text{false negatives} + \text{false positives}}$$

Figure 7 shows the F$_1$-Scores for the different methods distinguished by different minimal error significances σ (i.e. 1e-8, 1e-10, and 1e-12). Only errors were injected that corrupted a result element r_k to become r_{kERR} with

$$|r_{kERR}| > |r_k|(1+\sigma) \ \lor \ |r_{kERR}| < |r_k|(1-\sigma)$$

where σ is the minimal error significance. For a minimal error significance σ = 1e-12, the F$_1$-Score for our method ranges from 0.68 for matrix *nos3* to 0.88 for matrix *ex3*. The average F$_1$-Score is 0.81. The F$_1$-Score is on average improved by 52.2% compared to dense checks. The minimum improvement is 2.9% with matrix *bcsstk13*. The maximum improvement is reported with matrix *Muu*, for which the F$_1$-Score is 17.6 times larger compared to dense checks. With increasing minimal error significance σ, the average F$_1$-Score is 0.88 for σ = 1e-10 and 0.95 for σ = 1e-8.

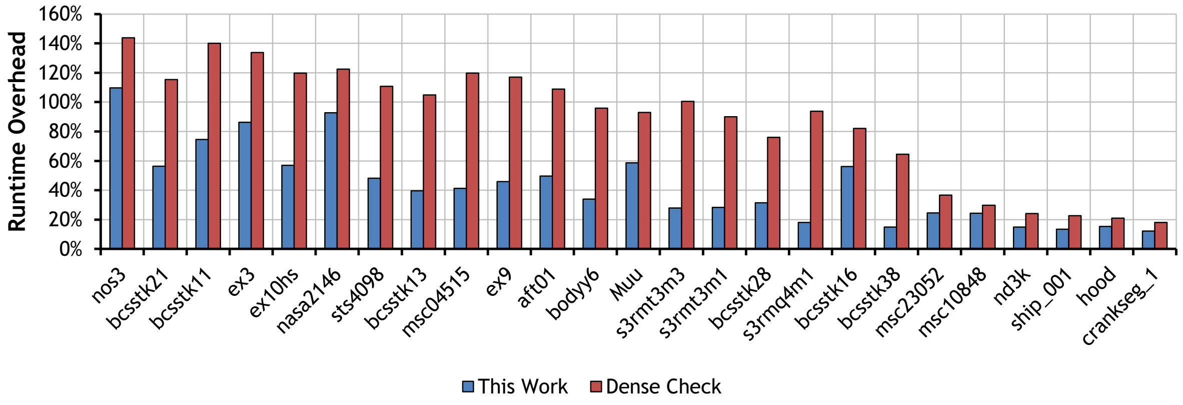


Fig. 5: Runtime overhead for error detection.

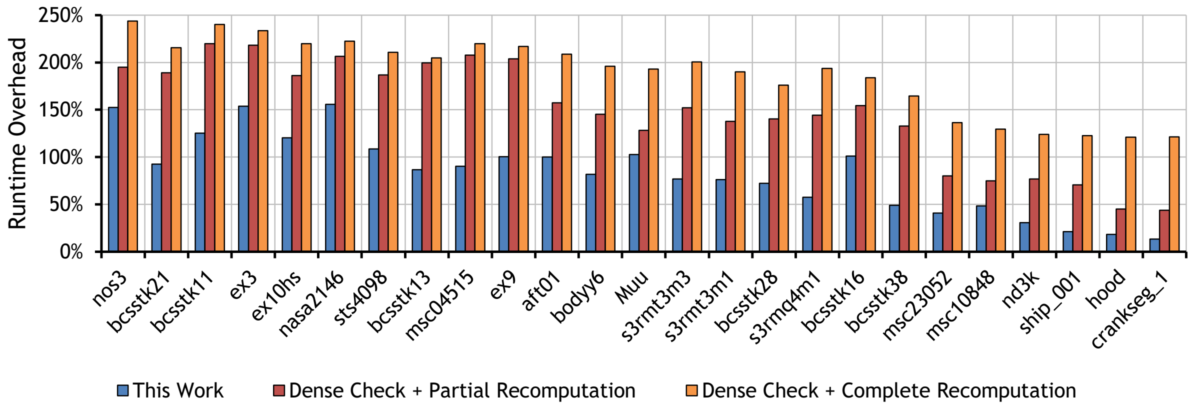


Fig. 6: Runtime overhead for error detection and correction.

VI. Case Study: Preconditioned Conjugate Gradient Solver

We evaluate the runtime overhead and success rates for our proposed fault tolerance approach by focusing one of the most common solvers, the Preconditioned Conjugate Gradient (PCG) method [19], which is an iterative solver. The solution of linear systems is an important task in many large-scale applications in science and engineering including structural mechanics [16] and computational fluid dynamics [17]. We compare our approach with both a traditional *checkpointing (i.e. error detection and rollback)* approach and the *error detection, localization and partial recomputation* approach from related work [30]. More than 200,000 error injection experiments were performed to obtain the results below. Each experiment comprises both a complete run of the PCG solver and a number of error injections with respect to different error rates λ. The runtime overhead corresponds to the runtime of protected PCG executions compared to the runtime of original unprotected PCG executions. The success rate is defined as the portion of error injection experiments in which the PCG solver provided a correct solution after a maximum limit of $10 \cdot N$ iterations.

We evaluate runtime overheads and success rates for different error rate scenarios. Typically, the timing of occurring errors is assumed to be independent. Therefore, we determine the error events from an exponential distribution with an error rate λ. We define $\frac{1}{\lambda}$ to be the expected number of arithmetic operations between two consecutive error events. In particular, the error rate λ corresponds to the probability that an arbitrary arithmetic operation will return an erroneous result. We consider different error rates in our experiments that range from low to high error rates. Low error rates may correspond to natural phenomena such as charged particle strikes that occur a couple of times per day (i.e. λ = 1e-8). The error rate is scaled up to λ = 1e-4 which may correspond to specially designed hardware that trades off accuracy against power consumption and might therefore induce multiple errors per second [12, 13]. Error injections are performed in sparse matrix vector multiplications according to the error model described in Section IV-A. Errors were also injected into operations that perform error detection.

We apply the block size $b_s = 32$ that was determined in Section V-A as well as the analytical rounding error bound described in Section III-C to our method. Dense checks are performed using the norm-based error bound (cf. Section V-B). The partial recomputation approach utilizes the error localization steps that were also utilized for the evaluation of the SpMV itself (cf. Section V). The *checkpointing* approach

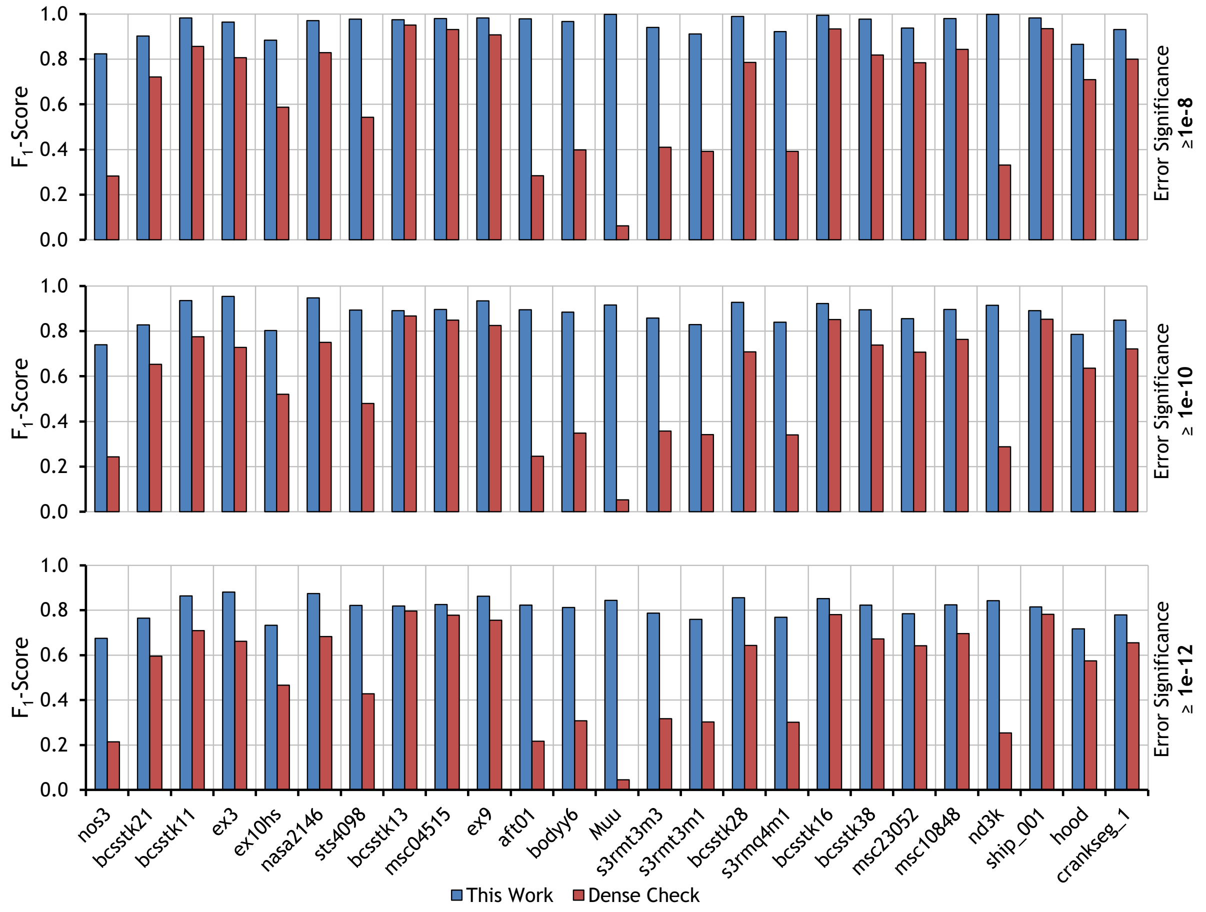


Fig. 7: Comparison of error coverage (F_1-Score) between our approach and dense checks.

relies on dense checks to detect errors. This approach samples the application state periodically and writes it to a ECC-protected memory after every 20 iterations of the solver.

A. The Preconditioned Conjugate Gradient Solver

The *Preconditioned Conjugate Gradient* method (PCG)[19] is commonly used to solve linear systems $Ax = b$, when the underlying matrix A is symmetric and positive-definite. Beginning with the approximation x_0, PCG computes new approximations $(x_1, x_2, x_3, ...)$ for every successive iteration k. Each successive iteration of *PCG* provides an improved approximation x_k with respect to the exact solution x. PCG composes the solution x as a linear combination of *search directions* $p_0, p_1, p_2, ..., p_N$ and $x = x_0 + \sum_{k \leq N} \alpha_k p_k$. Iterations are performed until the approximation error falls below a predefined threshold ϵ, which corresponds to the comparison between the euclidean norm of the residual r_k and ϵ (i.e. $\|r_k\| = r_k^T r_k < \epsilon$). The time complexity of PCG depends on both the size and the condition number of the matrix A [19]. A suitable *preconditioner* M is able to diminish the condition number of the matrix A, which improves the rate of convergence.

The *Jacobi-Preconditioner* was applied for preconditioning [19] in the experiments. In addition to these results, we conducted experiments with other preconditioners such as *SSOR* and *Incomplete Cholesky* [19]. The results do not show significant differences. For all experiments, a random vector was generated for the initial guess x_0, If the right-hand side b was not available for a matrix, then a random solution x was generated. Using x, the right-hand side b was computed with $Ax = b$. We set the the error tolerance ϵ in our experiments to 10^{-6} as proposed in [30].

B. Runtime Overhead

The runtime overhead is obtained by comparing the runtime of protected PCG executions for a given error rate λ to the runtime of original unprotected PCG executions:

$$\text{Runtime overhead} = \frac{\text{Runtime for protected PCG}}{\text{Runtime for unprotected PCG}} - 1$$

For this evaluation, only runtimes of PCG executions were considered that provided a correct result. Figure 8 shows the average runtime overhead for all evaluated matrices under error rates ranging from λ = 1e-8 to λ = 1e-12. For low error rates λ = 1e-8, the runtime overhead of our method is 39.8%. The runtime overhead for the *partial recomputation* approach [30] is 58.4% and the runtime overhead for *checkpointing* is 62.9%. In [30], the reported overhead for the *partial recomputation* approach is roughly 55% under low error rates. This small difference (i.e. 3.4%) between the reported results can be explained by the different systems setups, i.e. many-core GPU system versus multi-core HPC system. For low error rates λ = 1e-8, our method reduces the average runtime overhead by 31.7% compared to the *partial recomputation* approach [30] and by 36.7% compared to the *checkpointing* approach.

When the error rate is increased from λ=1e-8 to λ=1e-6, the average runtime overhead of our method increases by only 0.6%. At the same time, the runtime overhead for the partial recomputation approach increases by 6.7% and by 4.3% for the checkpointing approach.

For high error rates, i.e. λ=1e-4, the runtime overhead of our method is 52.3%. At the same time, the runtime overhead for the *partial recomputation* approach is 87.4%, while the average runtime overhead for *checkpointing* is 162.9%. For such high error rates, our method reduces the average runtime overhead by 40.1% with respect to the *partial recomputation* approach [30] and by 67.6% with respect to the *checkpointing* approach. While the error rate is increased by four orders of magnitude, i.e. from 1e-8 to 1e-4, the average runtime overhead induced by our method only increases by 31.3%.

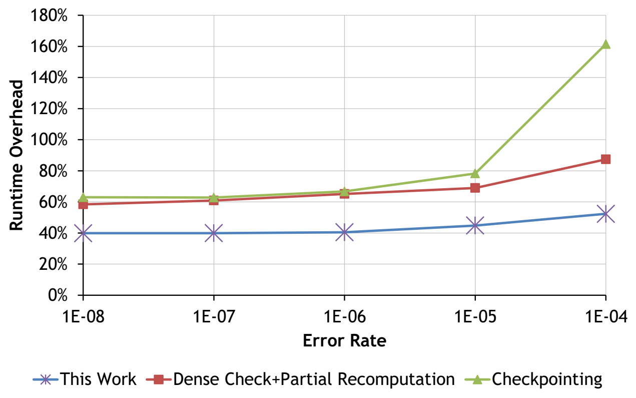


Fig. 8: Runtime overhead under different error rates compared to unprotected PCG execution.

C. Error Coverage

We evaluate the error coverage by identifying the success rates for the different fault tolerance techniques. As mentioned above, we define the success rate by the portion of error injection experiments in which PCG converged to a correct result within $10 \cdot N$ iterations. Figure 9 shows the portion of successful PCG executions for different error rates λ ranging from λ=1e-8 to λ=1e-6. Under low error rates (i.e. λ = 1e-8), all considered methods achieve roughly 100% success rates. With increasing error rates, the number of successful PCG executions decreases. For the error scenario λ = 1e-8, our method achieves a success rate of 88.4%. The partial recomputation approach achieves a success rate of 84.8% while the checkpointing approach achieves 81.3%. Differences between the different methods significantly increase for higher error rates. For the error scenario λ = 1e-4, our method achieves a success rate of 55.5%. Compared to the partial recomputation and checkpointing approaches, the success rate is improved by 1.61x and 3.6x respectively.

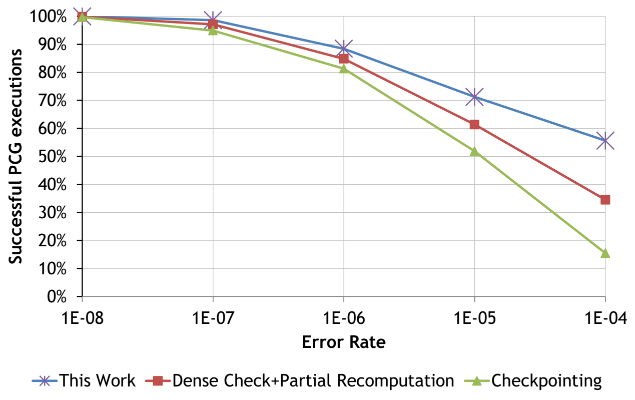


Fig. 9: Successful PCG executions under different error rates.

VII. Conclusion

In this work, we presented a fault tolerance approach for sparse matrix operations that allows the efficient algorithmic detection and correction of erroneous application outputs. Instead of only detecting errors, our approach instruments error detection steps to provide error locations. This enables partial recomputations just for erroneous results directly after error detection. Therefore, both expensive error localization steps as well as repeating entire recomputations are avoided. Existing algorithm-based fault tolerance approaches for sparse matrix operations often rely on expensive error localization steps. General checkpointing techniques restart the application from periodically saved states to correct errors. However, such techniques can induce large recovery cost for high error rates.

We evaluated our approach for sparse matrix-vector multiplications (SpMV) and for iterative linear solvers which use SpMV as an internal subroutine. While the runtime overhead to provide fault tolerance for SpMV is on average reduced by 43.8%, the error coverage is on average improved by 52.2% compared to related work approaches that apply dense checks to detect errors. The runtime overhead scales very well with increasing problem sizes.

In a case study, we evaluated our approach for the Preconditioned Conjugate Gradient Solver (PCG). For low error rates, our method reduces the average runtime overhead to provide fault tolerance for PCG by 31.7% compared to related algorithmic detection and correction approaches and by 36.7% compared to checkpointing. Under high error rates, experimental results show an average reduction in runtime overhead for fault tolerance of 40.1% and 67.6% respectively. At the same time, the number of successful PCG solver runs is on average increased by 61.6% with respect to related algorithmic detection and correction approaches. Compared to traditional checkpointing techniques, the average number of successful solver executions is increased by 3.6 times. Our method scales favorably with increasing error rates since the average runtime overhead increases by only 31.3% while the error rate is scaled by four orders of magnitude.

Acknowledgment

The authors thank the German Research Foundation (DFG) for financial support of the project within the Cluster of Excellence in Simulation Technology (EXC 310/2) at the University of Stuttgart.

BIBLIOGRAPHY

[1] M. Orobitg *et al.*, "High Performance Computing Improvements on Bioinformatics consistency-based Multiple Sequence Alignment Tools", *Parallel Computing*, vol. 42, pp. 18–34, 2015.

[2] A. Schöll *et al.*, "Adaptive Parallel Simulation of a Two-Timescale-Model for Apoptotic Receptor-Clustering on GPUs", in *Proc. of the IEEE Intl. Conference on Bioinformatics and Biomedicine (BIBM'14)*, Belfast, UK, Nov. 2014, pp. 424–432.

[3] E. Schneider *et al.*, "GPU-Accelerated Small Delay Fault Simulation", in *Proc. of the ACM/IEEE Conference on Design, Automation Test in Europe (DATE'15)*, Grenoble, France, Mar. 2015, pp. 1174–1179.

[4] C. Braun *et al.*, "Acceleration of Monte-Carlo Molecular Simulations on Hybrid Computing Architectures", in *IEEE Intl. Conference on Computer Design (ICCD)*, Montreal, Canada, Oct. 2012, pp. 207–212.

[5] D. A. Reed and J. Dongarra, "Exascale Computing and Big Data", *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, Jul. 2015.

[6] E. Nurvitadhi, A. Mishra, and D. Marr, "A Sparse Matrix Vector Multiply Accelerator for Support Vector Machine", in *Proc. Intl. Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '15, Amsterdam, The Netherlands, Oct. 2015, pp. 109–116.

[7] I. Haque and V. Pande, "Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU", in *Proc. of the IEEE/ACM Intl. Conference on Cluster, Cloud and Grid Computing (CCGrid'10)*, Melbourne, Australia, May 2010, pp. 691–696.

[8] "The International Technology Roadmap for Semiconductors 2013 Edition". [Online]. Available: http://www.itrs.net/Links/2013ITRS/Home2013.htm

[9] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-Cost Program-Level Detectors for Reducing Silent Data Corruptions", in *Proc. of the 42nd IEEE/IFIP Intl. Conference on Dependable Systems and Networks (DSN'12)*, Boston, MA, USA, Jun. 2012, pp. 1–12.

[10] T. Z. Islam *et al.*, "McrEngine: A Scalable Checkpointing System using Data-Aware Aggregation and Compression", in *Intl. Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, Salt Lake City, UT, USA, Nov. 2012, pp. 1–11.

[11] H. Esmaeilzadeh *et al.*, "Dark Silicon and the End of Multicore Scaling", *IEEE Micro*, no. 3, pp. 122–134, 2012.

[12] A. Sampson *et al.*, "EnerJ: Approximate Data Types for Safe and General Low-power Computation", in *Proc. 32nd ACM Conference on Programming Language Design and Implementation (PLDI'11)*, San Jose, CA, USA, Jun., pp. 164–174.

[13] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware", in *Proc. Intl. Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '13)*, Indianapolis, IA, USA, Oct. 2013, pp. 33–52.

[14] M. P. R. Suraana and N. Thoutam, "A Review on Evaluation of Multi-level Checkpointing System in Distributed Environment", *Intl. Journal of Electronics, Communication and Soft Computing Science & Engineering (IJECSCSE)*, vol. 3, no. 7, pp. 25–32, 2014.

[15] R. Vemu and J. Abraham, "CEDA: Control-flow Error Detection using Assertions", *IEEE Transactions on Computers*, vol. 60, no. 9, pp. 1233–1245, 2011.

[16] I. Smith, D. Griffiths, and L. Margetts, *Programming the Finite Element Method*, 4th ed. Wiley, Oct 2013.

[17] D. Yuen *et al.*, *GPU Solutions to Multi-scale Problems in Science and Engineering*, 8th ed., ser. In Earth System Sciences. Springer, 2013.

[18] A. Peixoto de Camargos *et al.*, "Efficient Parallel Preconditioned Conjugate Gradient Solver on GPU for FE Modeling of Electromagnetic Fields in Highly Dissipative Media", *IEEE Trans. on Magnetics*, vol. 50, no. 2, pp. 569–572, Feb. 2014.

[19] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Siam, 2003.

[20] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*, M. Kaufmann, Ed. Elsevier, 2010.

[21] C. Engelmann, H. Ong, and S. L. Scott, "The Case for Modular Redundancy in Large-Scale High-Performance Computing Systems", in *Proc. 8th IASTED Intl. Conference on Parallel and Distributed Computing and Networks (PDCN'09)*, Innsbruck, Austria, Feb. 2009, pp. 189–194.

[22] D. Ibtesham *et al.*, "Coarse-Grained Energy Modeling of Rollback/Recovery Mechanisms", in *Proc. of the 44th Annual IEEE/IFIP Intl. Conference on Dependable Systems and Networks (DSN'14)*, Atlanta, GA, USA, Jun. 2014, pp. 708–713.

[23] D. Hakkarinen and Z. Chen, "Multilevel Diskless Checkpointing", *IEEE Transactions on Computers*, vol. 62, no. 4, pp. 772–783, 2013.

[24] A. Schöll *et al.*, "Low-Overhead Fault-Tolerance for the Preconditioned Conjugate Gradient Solver", in *Proc. of the Intl. Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT'15)*, Amherst, MA, Oct. 2015, pp. 60–66.

[25] J. T. Daly, "A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps", *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303–312, 2006.

[26] K.-H. Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations", *IEEE Trans. on Computers*, vol. 33, no. 6, pp. 518–528, Jun. 1984.

[27] A. Bouteiller *et al.*, "Algorithm-Based Fault Tolerance for Dense Matrix Factorizations, Multiple Failures and Accuracy", *ACM Trans. Parallel Computing*, vol. 1, no. 2, pp. 10:1–10:28, Feb. 2015.

[28] C. Braun, S. Halder, and H.-J. Wunderlich, "A-ABFT: Autonomous Algorithm-Based Fault Tolerance for Matrix Multiplications on Graphics Processing Units", in *Proc. of The 44th IEEE/IFIP Intl. Conference on Dependable Systems and Networks (DSN'14)*, Atlanta, GA, USA, Jun. 2014, pp. 443–454.

[29] G. Bronevetsky and B. de Supinski, "Soft Error Vulnerability of Iterative Linear Algebra Methods", in *Proc. of the Intl. Conference on Supercomputing*, Island of Kos, Greece, Nov. 2008, pp. 155–164.

[30] J. Sloan, R. Kumar, and G. Bronevetsky, "An Algorithmic Approach to Error Localization and Partial Recomputation for Low-Overhead Fault Tolerance", in *Proc. of the 43rd IEEE/IFIP Intl. Conference on Dependable Systems and Networks (DSN'13)*, Budapest, Hungary, Jun. 2013, pp. 1–12.

[31] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Fault Tolerant Preconditioned Conjugate Gradient for Sparse Linear System Solution", in *Proc. of the ACM Intl. Conference on Supercomputing*, Venice, Italy, Jun. 2012, pp. 69–78.

[32] J. Sloan, R. Kumar, and G. Bronevetsky, "Algorithmic Approaches to Low Overhead Fault Detection for Sparse Linear Algebra", in *Proc. of the 42nd IEEE/IFIP Intl. Conference on Dependable Systems and Networks (DSN'12)*, Boston, MA, USA, 2012, pp. 1–12.

[33] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*. Siam, 1996, no. 48.

[34] Å. Björck, *Numerical Methods in Matrix Computations*, ser. Texts in Applied Mathematics. Springer International Publishing, 2014.

[35] A. Roy-Chowdhury and P. Banerjee, "Tolerance Determination for Algorithm-Based Checks Using Simplified Error Analysis Techniques", in *Proc. of the Intl. Symposium on Fault-Tolerant Computing*, Toulouse, France, Jun. 1993, pp. 290–298.

[36] J. Rexford and N. Jha, "Partitioned Encoding Schemes for Algorithm-Based Fault Tolerance in Massively Parallel Systems", *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 6, pp. 649–653, 1994.

[37] E. Gallopoulos, B. Philippe, and A. Sameh, *Parallelism in Matrix Computations*, ser. Scientific Computation. Springer, 2015.

[38] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Characterizing the Impact of Soft Errors on Iterative Methods in Scientific Computing", in *Proc. of the Intl. Conference on Supercomputing*, Seattle, WA, USA, Nov. 2011, pp. 152–161.

[39] P. Greisen *et al.*, "Evaluation and FPGA implementation of sparse linear solvers for video processing applications", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 23, no. 8, pp. 1402–1407, 2013.

[40] A. Ashari *et al.*, "Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications", in *The Intl. Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*, New Orleans, LA, USA, Nov. 2014, pp. 781–792.

[41] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-Flow Checking by Software Signatures", *IEEE Trans. on Reliability*, vol. 51, no. 1, pp. 111–122, Aug. 2002.

[42] K. Wilken and J. P. Shen, "Continuous Signature Monitoring: Low-cost Concurrent Detection of Processor Control Errors", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 6, pp. 629–641, Jun. 1990.

[43] A. Sanchez-Macian, P. Reviriego, and J. A. Maestro, "Hamming SEC-DAED and Extended Hamming SEC-DED-TAED Codes Through Selective Shortening and Bit Placement", *IEEE Transactions on Device and Materials Reliability*, vol. 14, no. 1, pp. 574–576, 2014.

[44] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection", *ACM Trans. on Mathematical Software*, vol. 38, no. 1, pp. 1:1–1:25, Nov. 2011.

[45] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills, "Vectorized Sparse Matrix Multiply for Compressed Row Storage Format", in *Computational Science*, ser. Lecture Notes in Computer Science, 2005, vol. 3514, pp. 99–106.

[46] C. Van Rijsbergen, *Information Retrieval*. Butterworths, 1979.