Selective Protection for Sparse Iterative Solvers to Reduce the Resilience Overhead

Hongyang Sun*, Ana Gainaru*, Manu Shantharam†, Padma Raghavan*
*Vanderbilt University, Nashville, TN, USA
†San Diego Supercomputer Center, San Diego, CA, USA

***Abstract*—The increasing scale and complexity of today's high-performance computing (HPC) systems demand a renewed focus on enhancing the resilience of long-running scientific applications in the presence of faults. Many of these applications are iterative in nature as they operate on sparse matrices that concern the simulation of partial differential equations (PDEs) which numerically capture the physical properties on discretized spatial domains. While these applications currently benefit from many application-agnostic resilience techniques at the system level, such as checkpointing and replication, there is significant overhead in deploying these techniques. In this paper, we seek to develop application-aware resilience techniques that leverage an iterative application's intrinsic resiliency to faults and selectively protect certain elements, thereby reducing the resilience overhead. Specifically, we investigate the impact of soft errors on the widely used Preconditioned Conjugate Gradient (PCG) method, whose reliability depends heavily on the error propagation through the sparse matrix-vector multiplication (SpMV) operation. By characterizing the performance of PCG in correlation with a numerical property of the underlying sparse matrix, we propose a selective protection scheme that protects only certain critical elements of the operation based on an analytical model. An experimental evaluation using 20 sparse matrices from the SuiteSparse Matrix Collection shows that our proposed scheme is able to reduce the resilience overhead by as much as 70.2% and an average of 32.6% compared to the baseline techniques with full-protection or zero-protection.**

***Index Terms*—Resilience, soft errors, selective protection, iterative solvers, preconditioned conjugate gradient.**

I. Introduction

Many scientific applications concern the simulation of partial differential equations (PDEs) that are discretized in space and time, and solved using explicit, implicit or semi-implicit methods [24], [25]. While the explicit method offers simplicity, implicit and semi-implicit methods provide more attractive numerical and convergence properties. Their underlying solution schemes are typically iterative, with outer iterations corresponding to time-steps and inner iterations corresponding to how linear equations related to the spatial domain are solved. Many of these applications are long-running, as high resolutions are often needed in space and/or time to characterize the dynamics at multiple scales and components for the phenomenon under study (e.g., structural properties, materials defects, turbulence) [22].

Currently, scientific applications executing on large HPC systems face serious challenges from both hard failures and soft faults, which are occurring at increasing rates due to the combined effects of hardware feature miniaturization and increased system scale [9], [10], [41]. To protect these applications, resilience techniques have been proposed at the application level by using algorithm-based fault tolerance (ABFT) [11], [12], [28], [42] for matrix algorithms (dense or sparse) or iterative methods for linear system solutions. These techniques, however, require tapping into the underlying numerical library and are often not easy to implement. Alternatively, application-agnostic techniques at the system level, such as checkpointing and replication [3], [6], [18], [26], are available as general-purpose and non-intrusive resilience mechanisms to protect the applications. While these techniques effectively enable application recovery in the event of faults, there is significant overhead associated with them, especially when the application is inherently resilient to faults.

In this paper, we seek to develop low-overhead resilience techniques against soft errors at the system level while leveraging an iterative application's intrinsic resiliency to faults. We focus on the widely used Preconditioned Conjugate Gradient (PCG) method to solve a sparse linear system $Ax = b$, where A is an $N \times N$ symmetric positive definite matrix, and both x and b are $N \times 1$ dense vectors. Many prior studies [8], [31], [37] have characterized the impact of soft errors on the performance of PCG, which depends heavily on the error propagation through the sparse matrix-vector multiplication (SpMV) operation $q \leftarrow A \cdot p$, a key step performed in every iteration of PCG. We show that soft errors striking certain elements of SpMV could significantly degrade the performance of PCG, while errors striking other elements have negligible impact. Hence, fully protecting the operation could potentially incur unnecessarily high overhead while not protecting it at all could cause severe performance degradations.

This observation calls for a selective scheme, which we propose in this paper, to judiciously protect certain critical elements of SpMV for a given linear system in order to reduce the resilience overhead while minimizing the overall performance degradation. Our scheme is based on a characterization of the performance of PCG in correlation with a numerical property of the sparse matrix A, and is further guided by an analytical model that predicts the overhead with only a small number of profiling runs. The selected elements are then protected at the system level via duplication, and soft errors are detected by comparing the results of the duplicated computation and mitigated by re-running the last iteration.

An experimental evaluation using 20 sparse matrices from the SuiteSparse Matrix Collection [1] shows that our proposed

scheme is able to reduce the resilience overhead by as much as 70.2% and an average of 32.6% compared to the baseline techniques with full-protection or zero-protection.

Our main contributions are summarized as follows:

- A characterization of the impact of soft errors on the performance of PCG in correlation with a numerical property of the sparse matrix A;
- A selective protection scheme for certain critical components of the key SpMV operation in PCG based on an analytical performance model;
- An experimental validation to demonstrate the benefit of the proposed scheme as a low-overhead resilience technique at the system level.

The rest of this paper is organized as follows. Section II introduces the background of sparse linear systems and the PCG algorithm. Section III characterizes the impact of soft errors on the performance of PCG, and presents our selective protection scheme based on an analytical model. Section IV evaluates the performance of our proposed scheme using a sample of the SuiteSparse Matrix Collection. Section V reviews some related work on protecting sparse iterative solvers and resilience techniques in general. Finally, Section VI concludes the paper and discusses future work.

II. Background

In this section, we provide a brief background on using sparse linear systems to numerically solve PDEs and give an overview of the PCG algorithm.

A. Sparse Linear Systems

Many large-scale scientific applications involve solving PDE-based systems, such as those found in heat diffusion, computational fluid dynamics (CFD) and structural mechanics [24], [25]. Such applications compute solutions using explicit, implicit or semi-implicit methods, with the latter two requiring the use of sparse linear solvers.

As a concrete example, consider a PDE-based application such as a heat equation [24] applied to a set of N discretized grid points in space. Let u^k, an $N \times 1$ vector, denote the temperature at time step k on all the grid points, and A, an $N \times N$ sparse matrix, represent the dependencies in space and physical properties. To compute the temperatures at next time step $k+1$, an *explicit* method calculates $u^{k+1} = u^k + Au^k$, whereas an *implicit* method calculates $Au^{k+1} = u^k$. While both methods consider the temperature evolution across time, the implicit methods requires a sparse linear system solution as an important inner step.

B. The PCG Algorithm

The *Preconditioned Conjugate Gradient (PCG)* algorithm [32] is one of the most widely used solvers for a sparse linear system of the form $Ax = b$, where A is an $N \times N$ symmetric positive definite (SPD) matrix, and both x and b are $N \times 1$ dense vectors. Algorithm 1 shows the pseudocode of PCG. The algorithm takes, as input, the coefficient matrix A, a known vector b, an initial guess of the solution vector x_0, a preconditioner matrix M, the maximum number of iterations allowed $maxit$, and a tolerance tol to detect the convergence of the solution. At each iteration i, the algorithm computes a new search direction (p_i) that is A-orthogonal to the previous search directions, and uses it to update the approximate solution (x_{i+1}) and the residual (r_{i+1}).

Algorithm 1: Preconditioned Conjugate Gradient (PCG)

```
   Input: A, M, b, x_0, tol, maxit
1  begin
2      r_0 ← b − Ax_0;                          // Initial residual
3      z_0 ← M^{-1} r_0;                        // Preconditioning
4      p_0 ← z_0;
5      i ← 0;
6      while i < maxit and ||r_i||/||b|| > tol do
7          q_i ← Ap_i;
8          v_i ← r_i^T z_i;
9          α ← v_i/(p_i^T q_i);
10         x_{i+1} ← x_i + αp_i;                // Improve approximation
11         r_{i+1} ← r_i − αq_i;                // Update residual
12         z_{i+1} ← M^{-1} r_{i+1};            // Preconditioning
13         v_{i+1} ← r_{i+1}^T z_{i+1};
14         β ← v_{i+1}/v_i;
15         p_{i+1} ← z_{i+1} + βp_i;            // New search direction
16         i ← i + 1;
17     end
18 end
```

A key operation in each iteration of the PCG algorithm is the sparse matrix-vector multiplication (SpMV) (Line 7). It is also the most compute-intensive operation taking $O(nnz)$ time, where nnz represents the number of nonzeros in matrix A. Preconditioning (Line 12) is another important part of PCG, as it ensures faster convergence of the algorithm. Although computing the inverse of a matrix is generally expensive, simple yet effective preconditioners are typically chosen to speed up the computation[1]. Other operations in PCG, including the inner products (Lines 8 and 13), vector additions (Lines 10, 11 and 15), and scalar operations (Lines 9 and 14), are relatively inexpensive, taking $O(N)$ time.

III. A Selective Protection Scheme

In this section, we introduce a selective protection scheme to reduce the resilience overhead of the PCG algorithm. The scheme is motivated by characterizing the impact of soft errors on the performance of PCG (Section III-A) and its correlation with a numerical property of the sparse matrix (Section III-B). Based on these, we build an analytical model to estimate the optimal selective protection pattern (Section III-C).

A. Impact of Soft Errors

Many prior works (e.g., [8], [31], [37]) have studied the impact of soft errors on the performance of PCG. In particular, it has been shown [37] that SpMV, which is the most expensive operation performed in every iteration of the algorithm, is most prone to errors. Furthermore, any error in the other operations will eventually show up in an element of the vector p in the

[1]For instance, computing the inverse of a diagonal preconditioner takes only linear time, and for preconditioners that use the incomplete Cholesky or LU factorization, the inverse can be computed using triangular solve, whose complexity is proportional to the number of nonzeros in M.

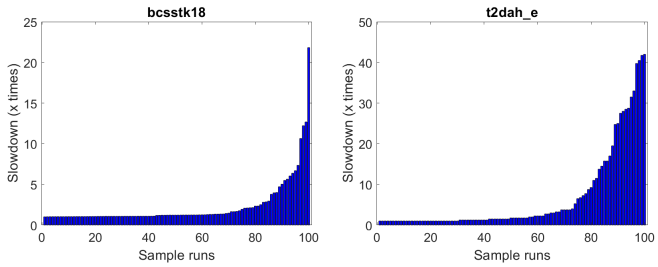

Fig. 1. Slowdowns in convergence of PCG for two matrices (*bcsstk18* and *t2dah_e*) in 100 sample runs, where each run has an error injected in a random element of vector p in the SpMV operation.

following iteration. Hence, we focus on soft errors that occur in the SpMV operation, in particular in the vector p.

However, errors that strike different elements of the vector, might have very different impacts. This is illustrated in Figure 1, which shows the slowdowns in convergence of PCG in 100 sample runs for two matrices (*bcsstk18* and *t2dah_e*) from the SuiteSparse Matrix Collection [1]. In each of the 100 runs, an error (of the same magnitude) is injected in a random element of vector p. The *slowdown* is measured against the error-free run as follows:

$$slowdown = I_e / I_o \ , \tag{1}$$

where I_e and I_o denote the number of iterations for the algorithm to converge with and without errors, respectively. In Figure 1, the slowdowns are sorted in ascending order. As we can see, a soft error striking certain elements of p significantly slows down the convergence (by more than $20\times$ for *bcsstk18* and $40\times$ for *t2dah_e*), while an error striking many other elements of p has negligible impact on the convergence.

Hence, to protect the SpMV operation and the PCG algorithm from soft errors, neither *full-protection* nor *zero-protection* would be ideal: the former will likely incur unnecessarily high overheads during the majority of runs when the impact of error is small, and the latter will cause severe performance degradations when the impact of error is high. This motivates the design of a *selective protection* scheme, which we will discuss in the rest of this section.

B. Performance Characterization

As soft errors striking different elements of the vector will degrade the performance of PCG to different degrees, one promising approach is to protect only those elements that will cause more severe performance degradations. It is, however, a challenge to dynamically identify such elements in an online manner as the execution of the algorithm progresses.

To address this challenge, we characterize the performance degradation of PCG by correlating it with a numerical property of the sparse matrix A, namely, its row 2-norms, which is static information that can be pre-computed offline. Specifically, the *2-norm* for the i-th row of matrix A is given by:

$$\|A_{i*}\|_2 = \sqrt{\sum_{j=1}^{N} A_{i,j}^2} \ . \tag{2}$$

It has been shown [37] that, for repeated SpMV operations to compute a sequence of vectors $\{y_0, y_1, y_2, \ldots, y_k, \ldots\}$, where $y_k \leftarrow A y_{k-1}$ (e.g., when it is used as an explicit method to numerically solve a PDE), the magnitude of an error in the i-th element of the initial vector y_0 will propagate and grow non-linearly with the 2-norm of the i-th row of matrix A. Since PCG contains an SpMV operation in a similar repeated fashion over iterations, we posit that the row 2-norms of matrix A could serve as a good indicator on the performance degradation of PCG as well, when an error occurs in the corresponding elements of the vector p.

Figure 2 illustrates this correlation for four representative matrices (*bcsstk18*, *t2dah_e*, *cvxbqp1* and *Trefethen_20000*) from the SuiteSparse Matrix Collection [1]. We run 100 experiments with random error injections in the vector p and each point in the figure represents one run. The figure also indicates the Pearson correlation coefficient r (in red). It can be seen that errors in positions with larger row 2-norms of matrix A indeed lead to more slowdowns in the convergence of the PCG algorithm. Similar results have also been observed for many other matrices in the collection.

We further correlate the row 2-norms of matrix A with two additional metrics on the intermediate performance of the algorithm to confirm that it can indeed be used as a reliable performance indicator. The two metrics are the *relative residual norm* and *A-norm of errors*, which, for the i-th iteration, are defined as follows:

$$\textit{relative residual norm} = \|r_i\| / \|b\| \ , \tag{3}$$

$$\textit{A-norm of errors} = \sqrt{(x_i - \hat{x})^T A (x_i - \hat{x})} \ , \tag{4}$$

where x_i and r_i denote the intermediate solution and residual at the i-th iteration, respectively, and $\hat{x}$ denotes the true solution to the system. Both metrics indicate the quality of the solution at an intermediate step. In particular, the relative residual norm is directly used as a convergence criterium (in Algorithm 1), and the A-norm of errors is known as an important indicator on the algorithm's convergence [30], [37].

Figures 3 and 4 show strong correlations of both metrics with the row 2-norms of matrix A for the same four matrices. Again, the Pearson's r values are indicated in red, and similar strong correlations have also been observed for many other matrices. All the results are measured at the I_o-th iteration of PCG (when an error-free run converges). These strong correlation results indicate that we can indeed use the row 2-norms of matrix A to reliably estimate the impact of soft errors on the performance of the PCG algorithm.

C. Predicting the Optimal Protection Pattern

The previous characterizations on the impact of soft errors and performance correlation suggest that we could selectively protect the elements in the SpMV operations corresponding to those rows with larger 2-norms in matrix A. The question remains to determine how many elements to protect in order to minimize the overall resilience overhead. To this end, we develop an analytical model to predict the optimal protection

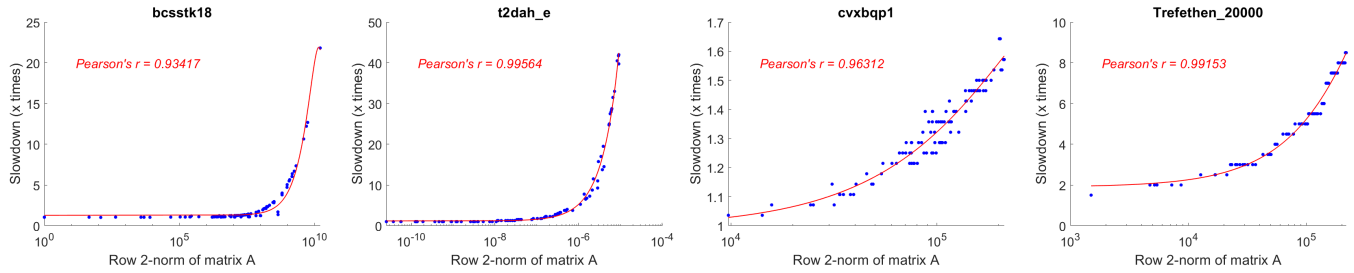

Fig. 2. Correlation between the row 2-norm of matrix A and the slowdown in convergence of PCG for four matrices with 100 runs.

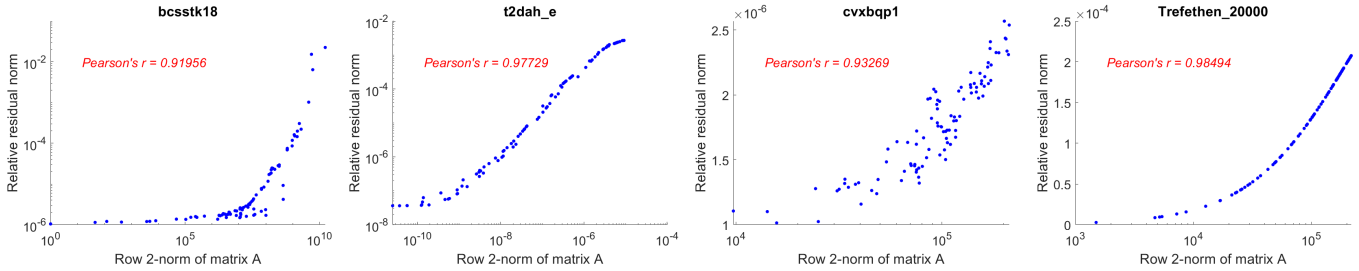

Fig. 3. Correlation between the row 2-norm of matrix A and the relative residual norm of PCG for four matrices with 100 runs at the I_o-th iteration.

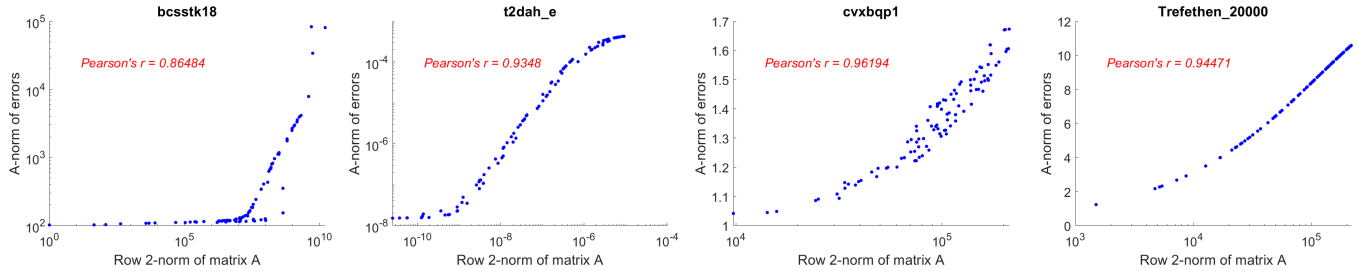

Fig. 4. Correlation between the row 2-norm of matrix A and the A-norm of errors of PCG for four matrices with 100 runs at the I_o-th iteration.

pattern. This is done in two steps, which are performance prediction and overhead optimization.

1) Performance Prediction: The first step aims at learning a function F that predicts the performance degradation of the algorithm (in terms of slowdown), given the row in which an error would occur. Due to its strong correlation with the row 2-norms of matrix A, we can learn such a function by profiling the algorithm's performance with a small number of sample runs followed by curve fitting.

Specifically, we inject errors in $m \ll N$ randomly selected elements, whose corresponding 2-norm values in matrix A are evenly distributed across the range. Note that the row 2-norms of A, and hence its range, are static, so they only need to be pre-computed once offline. We then measure the slowdown of the algorithm corresponding to each injected error. Since the function F may not be linear, we predict it by using *polynomial regression* to fit the data. In the performance evaluation (Section IV), we use $m = 20$ samples and fit a polynomial function of degree 2 or 3 (using higher degrees tends to overfit the data with high variance).

2) Overhead Optimization: After predicting the performance degradation function F, the second step builds an analytical model to optimize the resilience overhead. For convenience, we use a_i to denote the 2-norm of the i-th row of matrix A, i.e., $a_i = \|A_{i*}\|_2$, and let σ denote a permutation that arranges the a_i's in non-increasing order, i.e., $a_{\sigma(1)} \geq a_{\sigma(2)} \geq \cdots \geq a_{\sigma(N)}$.

Suppose soft errors strike all elements of the vector with equal probability (i.e., $1/N$). Then, by protecting k elements with the largest a_i values, the expected performance degradation (or slowdown) when an error strikes is given by:

$$D_e(k) = \frac{1}{N}\left(k + \sum_{i=k+1}^{N} F(a_{\sigma(i)})\right) , \tag{5}$$

and the normalized computational cost per iteration is:

$$C_e(k) = 1 + \frac{k}{N} . \tag{6}$$

Note that both quantities in Equations (5) and (6) are normalized with respect to an error-free execution without protection, which is assumed to have no slowdown, i.e., $D_o = 1$, and a unit computational cost per iteration, i.e.,

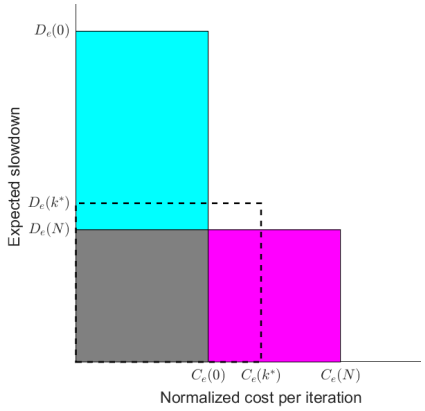

Fig. 5. Trade-off between the normalized cost per iteration and expected slowdown when optimizing the overall resilience overhead.

$C_o = 1$. The expected overhead when protecting these k elements is therefore given by:

$$H_e(k) = D_e(k) \cdot C_e(k) - D_o \cdot C_o$$
$$= \frac{1}{N}\left(k + \sum_{i=k+1}^{N} F(a_{\sigma(i)})\right)\left(1 + \frac{k}{N}\right) - 1\ . \quad (7)$$

Equation (7) provides an analytical model for estimating the expected overhead of a selective protection scheme. The goal is to find the optimal protection pattern (i.e., the optimal number k^* of protected elements) that minimizes the expected overhead. Given all row 2-norms and a predicted function F for a particular matrix A, the solution can be computed in $O(N)$ time by evaluating all values of $k \in \{1, 2, \ldots, N\}$.

Figure 5 illustrates the trade-off involved in this optimization. In particular, protecting all elements ($k = N$) doubles the computational cost per iteration, i.e., $C_e(N) = 2C_e(0) = 2C_o$, while causing almost no slowdown, i.e., $D_e(N) = D_o$. This results in 100% overhead (indicated by the magenta area) compared to the cost of an error-free run (indicated by the grey area). On the other hand, protecting no element ($k = 0$) incurs no extra cost per iteration, i.e., $C_e(0) = C_o$, but could lead to a large slowdown $D_e(0)$ and hence a large expected overhead (indicated by the cyan area). An optimal scheme protects a certain number of k^* elements that minimizes the overhead (corresponding to the difference between the area of the dashed rectangle and the grey area).

We point out that the model and approach proposed above apply to system-level protection of the iterative solver, hence a full protection corresponds to duplicating the entire computation. Soft errors are detected by comparing the results of the duplicated computation and mitigated by re-running the last iteration. Despite the availability of application-level techniques (e.g., ABFT [11], [38], [42]), which could further reduce the overhead by tapping into the numerical libraries, system-level protection remains the most transparent, least intrusive, and easy-to-implement technique to protect an application [2], [3], especially when accessing the internals of numerical libraries is limited. Investigating selective protection schemes at the application level will be part of our future work.

Table I. 20 matrices from the SuiteSparse Matrix Collection [1].

Id	*Matrix*	N	nnz	*Density*
1	t2dah_e	11445	176117	0.13%
2	bcsstk18	11948	149090	0.1%
3	cbuckle	13681	676515	0.36%
4	Pres_Poisson	14822	715804	0.33%
5	gyro_m	17361	340431	0.11%
6	nd6k	18000	6897316	2.1%
7	bodyy5	18589	128853	0.037%
8	raefsky4	19779	1316789	0.34%
9	Trefethen_20000	20000	554466	0.14%
10	msc23052	23052	1142686	0.22%
11	bcsstk36	23052	1143140	0.22%
12	wathen100	30401	471601	0.051%
13	vanbody	47072	2329056	0.11%
14	cvxbqp1	50000	349968	0.014%
15	ct20stif	52329	2600295	0.095%
16	thermal1	82654	574458	0.0084%
17	m_t1	97578	9753570	0.1%
18	2cubes_sphere	101492	1647264	0.016%
19	G2_circuit	150102	726674	0.0032%
20	pwtk	217918	11524432	0.024%

IV. Performance Evaluation

In this section, we evaluate the performance of our proposed selective protection scheme, and compare its resilience overhead against those of some baseline techniques.

A. Experimental Setup

In the experiments, the sparse matrix A is selected from 20 symmetric positive definite (SPD) matrices in the SuiteSparse Matrix Collection [1] (formerly known as the University of Florida Sparse Matrix Collection). Table I lists these matrices and their properties, including matrix dimension (N), number of nonzeros (nnz), and matrix density (nnz/N^2).

We run the PCG algorithm with incomplete Cholesky factorization as the preconditioner [29], where threshold dropping is used with the threshold set to be 10^{-3}, and the convergence tolerance is set to be 10^{-6}. The vector b is set as $A \cdot \mathbf{1}$, where $\mathbf{1}$ denotes an all-one vector, and the initial guess x_0 of the solution is set as an all-zero vector $\mathbf{0}$. The same setting has been used in some prior works [8], [31].

As stated in Section III-A, we focus on soft errors that manifest in the SpMV operation due to the important role it plays in the PCG algorithm. Instead of performing actual bit-flips, we simulate a soft error by perturbing the value of a randomly selected element in the vector p. This approach has been proposed in some previous studies [37], [42] as a systematic approach to investigate the impact of soft errors. In particular, the value in a random position k of the vector p is perturbed in the very first iteration by an error e, i.e., $p_1[k] = p_1[k] + e$. The magnitude of the error is generated uniformly at random from the range of all values in vector p in that iteration. Our experiments show the same results if the error is injected in later iterations. All results are obtained by averaging over 100 experiments and error injections.

B. Experimental Results

We first evaluate the performance prediction model (the first step of our scheme), where we used $m = 20$ random points

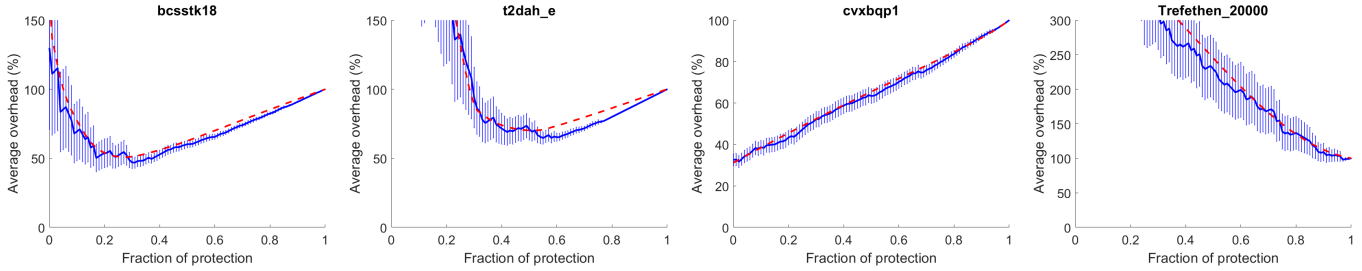

Fig. 6. Average overhead and 95% confidence interval (in blue) for four matrices when varying the fraction of protected elements using our selective protection scheme. The red dashed line represents the predicted overhead using our analytical model.

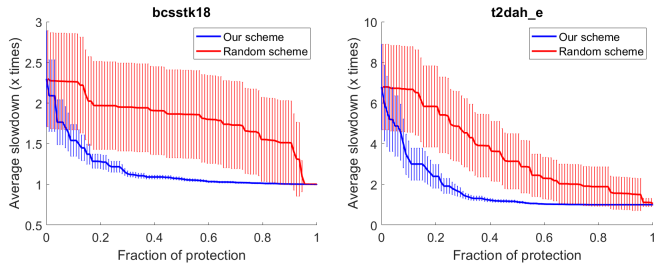

Fig. 7. Average slowdown with 95% confidence interval for two matrices when varying the fraction of protected elements using our selective protection scheme (in blue) and the random selective protection scheme (in red).

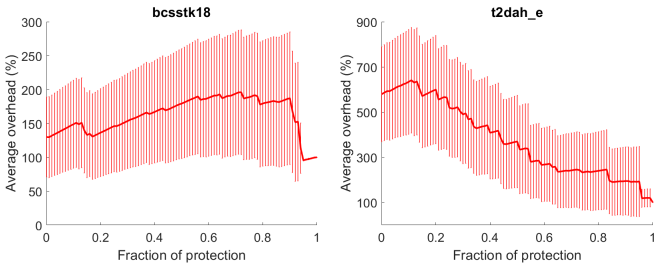

Fig. 8. Average overhead and 95% confidence interval (in red) for two matrices when varying the fraction of protected elements using the random selective protection scheme.

to do curve fitting. Despite the use of only a small number of points, the model is able to accurately predict the slowdown, thanks to its strong correlation with the row 2-norms of matrix A. Figure 2 shows the fitted curves (in red lines) for the four presented matrices. Furthermore, for all the 20 matrices, the r^2-score (i.e., the coefficient of determination that measures the fitness of the function) obtained from the 100 points, is above 0.9, which validates the accuracy of the performance prediction model.

To evaluate the analytical model on the resilience overhead (Equation (7)) and our overhead optimization procedure (the second step), we plot in Figure 6 the average overhead measured from the 100 runs (in blue with the 95% confidence interval) along with the model-predicted overhead (in red dashed line) for the four matrices, as the fraction of protected elements is varied from 0 (i.e., zero-protection) to 1 (i.e., full-protection). First, we can observe that the predicted overhead matches closely the average overhead obtained from the experiments. Furthermore, our optimization procedure suggests protecting about 28% and 51% of all elements for matrices *bcsstk18* and *t2dah_e*, resulting in an average overhead of 52% and 69%, respectively. The results represent significant performance improvements over the zero-protection scheme (whose overhead is more than 100% for both matrices) and the full-protection scheme (with 100% overhead).

For matrix *cvxbqp1*, the optimal strategy is to protect no element (zero-protection), and this is because all of them will induce a relatively mild slowdown (<2), as shown in Figure 2. On the other hand, the optimal strategy for matrix *Trefethen_20000* is to protect all elements (full-protection), because most of the elements will induce a large slowdown (>2), again as shown in Figure 2. In both cases, however, our selective protection scheme is able to find the optimal protection pattern based on the established analytical model.

To further illustrate the benefit of our selective protection scheme, we plot in Figure 7 its average slowdown for two matrices (*bcsstk18* and *t2dah_e*) in comparison with that of a random selective protection scheme [39], which protects a certain fraction of randomly selected elements[2]. The figure shows that, as we increase the fraction of protected elements, our scheme is able to reduce the slowdown much faster and with less variation compared to the random scheme. This is a direct consequence of our scheme's more targeted approach, which judiciously protects those elements that would cause larger slowdowns. As a result, the optimal random protection (among all protection fractions) still yields an average overhead close to 100% for the two matrices (as shown in Figure 8), while our scheme is able to reduce the overheads to 52% and 69%, respectively (as shown in Figure 6).

Finally, Figure 9 compares the average overheads of our selective protection scheme with those of the optimal random protection, zero-protection and full-protection for all the 20 matrices. For matrices *cvxbqp1*, *thermal1* and *nd6k* (left-most three), the minimal overhead is achieved by the zero-protection scheme, due to the small impact of soft errors on all the elements. On the other hand, for matrices *Trefethen_20000*, *vanbody* and *wathen100* (right-most three), the minimal overhead is achieved by the full-protection scheme, due to the large impact of soft errors on most of the elements. For all of

[2]This random protection scheme was originally proposed in [39] to augment the traditional ABFT technique applied at the application level to reduce the resilience overhead of sparse linear solvers.

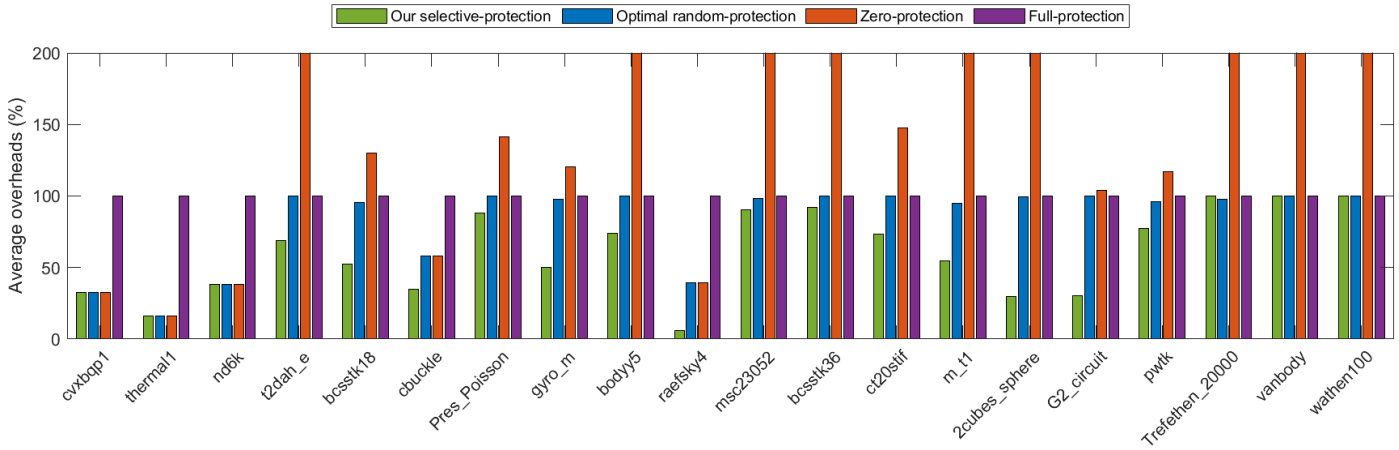

Fig. 9. Comparison of the average overheads for all 20 matrices using our selective protection scheme, the optimal random protection scheme, the zero-protection scheme and the full-protection scheme.

the other matrices, our scheme leads to the best performance, with an average overhead that improves upon that of the best of these baseline schemes by as much as 70.2% and an average of 32.6%. The results demonstrate the benefit of our proposed scheme as a general resilience technique for PCG that can be applied at the system level to reduce the overhead.

V. Related work

In this section, we review some related work on application-level techniques to protecting sparse iterative solvers, as well as general resilience techniques for scientific applications.

A. Resilience Techniques for Sparse Iterative Solvers

There has been considerable interest in developing resilient sparse iterative solvers, including multiple variants of CG or PCG, and different flavors of GMRES. Most of these works aim at designing low-overhead resilience techniques at the application level to make the solvers robust against faults.

One widely used technique is ABFT-enabled checksums and checkpointing. Bronevestky et al. [8] used a combination of matrix encoding schemes and checkpointing methods to build resilient CG solvers. Shantharam et al. [38] relied on the symmetric positive definite and diagonally dominant properties of the sparse matrices to develop a checksum encoded fault-tolerant PCG. Tao et al. [42] combined roll-back and roll-forward techniques with both lazy and eager checksums to provide complete protection of the PCG solver. A similar checksum technique was developed by Fasi et al. [17] to protect CG with both error detection and correction capabilities. Without using checkpointing, Sloan et al. [40] proposed partial recomputation by performing a binary search on the location of errors. Schöll et al. [35] applied a similar technique while relying on a blocked checksum approach for PCG.

In addition to checksum encoding and checkpointing techniques, many researchers have also explored the numerical and convergence properties of the underlying solvers to make them resilient to faults. Chen [11] designed an error detection and checkpointing technique by verifying the orthogonality and residual properties of the Krylov subspace iterative methods. Sao and Vuduc [33] proposed self-stabilizing iterative solvers by periodically checking and restoring the orthogonality property. Schöll et al. [34] combined orthogonality restoration and periodic checkpointing to achieve both roll-back and roll-forward recoveries for PCG. Bridges et al. [7] and Elliott et al. [15] designed fault-tolerant GMRES by applying selective reliability to its inner and outer loops.

B. General Resilience Techniques for Scientific Applications

Resilience solutions seek effective and efficient management of faults or errors to ensure the reliable outcomes of scientific applications. In general, resilience techniques for HPC systems are based on one or several methods of detection [5], [43], containment [23], [36] and recovery [16].

The most popular general-purpose resilience technique is checkpointing and rollback-recovery (C/R), which enables a scientific application to capture a snapshot of its state, save the checkpoint file, and recover from it in case of a fault. A comprehensive survey of application and system-level C/R methods can be found in [14], [26]. In recent years, incremental or differential checkpointing [19] has been proposed to avoid re-writing identical data between two consecutive checkpoints, either by tracking dirty memory pages in the system and only updating those within the checkpoint or by partitioning the application datasets (not memory pages) in blocks and keep track of the changes of each block [27]. Another general resilience technique is replication [2], [3], [26], which runs two or more copies of the application and compares their results to detect errors and/or to correct them via majority voting. Different flavors of replication have been proposed, including replicating individual processes of the application [18] and collectively replicating a group of processes [6]. Dichev and Nikolopoulos [13] applied process duplication to detect errors for the PCG solver.

For scientific applications using MPI, an addition layer of fault tolerance can be included within the MPI runtime system through the ULFM interface [4]. ULFM is a low-level API that provides resilience constructs to support a variety of fault tolerance models, either specific to one application [20]

or a set [21]. Our work is complementary to these general resilience methods, since it can inform the MPI runtime, the C/R protocol, or the replication process on the application specific data that needs protection.

VI. Conclusion and Future Work

In this paper, we have proposed a selective protection scheme for the widely used PCG solver to reduce the resilience overhead at the system level while leveraging the application-level resiliency characteristics. Our scheme is motivated by the different impacts of soft errors on the performance of PCG and its strong correlation with the row 2-norms of the underlying sparse matrix. We have built an analytical model that accurately predicts the convergence performance, and used it to estimate the optimal protection pattern. Experimental results based on a sample of the SuiteSparse Matrix Collection have confirmed the benefit of our selective protection scheme with significant reduction in overhead compared to some baseline techniques. Future work will be devoted to the design of selective protection schemes for other iterative solvers and the use of selective protection at the application level to further reduce the resilience overhead.

References

[1] The suitesparse matrix collection. https://sparse.tamu.edu/.

[2] A. Benoit, A. Cavelan, F. Cappello, P. Raghavan, Y. Robert, and H. Sun. Identifying the right replication level to detect and correct silent errors at scale. In *FTXS*, page 31–38, 2017.

[3] A. Benoit, A. Cavelan, F. Cappello, P. Raghavan, Y. Robert, and H. Sun. Coping with silent and fail-stop errors at scale by combining replication and checkpointing. *J. Parallel Distrib. Comput.*, 122:209–225, 2018.

[4] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. *Int. J. High Perform. Comput. Appl.*, 27(3):244–254, Aug. 2013.

[5] G. Bosilca, A. Bouteiller, A. Guermouche, T. Herault, Y. Robert, P. Sens, and J. Dongarra. Failure detection and propagation in HPC systems. In *SC*, 2016.

[6] M. Bougeret, H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni. Using group replication for resilience on exascale systems. *Int. J. High Perform. Comput. Appl.*, 28(2):210–224, May 2014.

[7] P. G. Bridges, K. B. Ferreira, M. A. Heroux, and M. Hoemmen. Fault-tolerant linear solvers via selective reliablity. *Sandia National Laboratories*, 2012.

[8] G. Bronevetsky and B. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *ICS*, pages 155–164, 2008.

[9] F. Cappello, G. Al, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomput. Front. Innov.: Int. J.*, 1(1):5–28, Apr. 2014.

[10] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388, Nov. 2009.

[11] Z. Chen. Online-abft: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *PPoPP*, 2013.

[12] Z. Chen and J. Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems*, 19(12):1628–1641, Dec 2008.

[13] K. Dichev and D. S. Nikolopoulos. TwinPCG: Dual thread redundancy with forward recovery for preconditioned conjugate gradient methods. In *CLUSTER*, 2016.

[14] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. 65(3):1302–1326, 2013.

[15] J. Elliott, M. Hoemmen, and F. Mueller. Evaluating the impact of SDC on the GMRES iterative solver. In *IPDPS*, 2014.

[16] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[17] M. Fasi, Y. Robert, and B. Uçar. Combining backward and forward recovery to cope with silent errors in iterative solvers. In *PDSEC*, 2015.

[18] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *SC*, pages 44:1–44:12, 2011.

[19] K. B. Ferreira, R. Riesen, P. G. Bridges, D. C. Arnold, and R. Brightwell. Accelerating incremental checkpointing for extreme-scale computing. *Future Gener. Comput. Syst.*, 30:66–77, 2014.

[20] M. Gamell, D. S. Katz, K. Teranishi, M. A. Heroux, R. F. Van der Wijngaart, T. G. Mattson, and M. Parashar. Evaluating online global recovery with Fenix using application-aware in-memory checkpointing techniques. In *ICPPW*, 2016.

[21] M. Gamell, K. Teranishi, J. Mayo, H. Kolla, M. A. Heroux, J. Chen, and M. Parashar. Modeling and simulating multiple failure masking enabled by local recovery for stencil-based applications at extreme scales. *IEEE Trans. on Parallel Distrib. Syst.*, 28(10):2881–2895, 2017.

[22] G. Golub and J. M. Ortega. *Scientific Computing: An Introduction with Parallel Computing.* Academic Press, 1993.

[23] A. Guermouche, T. Ropars, M. Snir, and F. Cappello. HydEE: Failure containment without event logging for large scale send-deterministic MPI applications. In *IPDPS*, 2012.

[24] M. T. Heath. *Scientific Computing: An Introductory Survey*. McGraw-Hill Higher Education, 2nd edition, 1996.

[25] M. A. Heroux, P. Raghavan, and H. D. Simon. *Parallel Processing for Scientific Computing (Software, Environments and Tools)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[26] T. Hérault and Y. Robert, editors. *Fault-Tolerance Techniques for High-Performance Computing*, Computer Communications and Networks. Springer Verlag, 2015.

[27] K. Keller and L. Bautista-Gomez. Application-level differential checkpointing for HPC applications with dynamic datasets. In *CCGRID*, 2019.

[28] Kuang-Hua Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518–528, 1984.

[29] I. Lee, P. Raghavan, and E. G. Ng. Effective preconditioning through ordering interleaved with incomplete factorization. *SIAM J. Matrix Anal. Appl.*, 27(4):1069–1088, Dec. 2005.

[30] G. Meurant. *The Lanczos and Conjugate Gradient Algorithms: From Theory to Finite Precision Computations (Software, Environments, and Tools)*. Society for Industrial and Applied Mathematics, USA, 2006.

[31] B. Ozcelik Mutlu, G. Kestor, J. Manzano, O. Unsal, S. Chatterjee, and S. Krishnamoorthy. Characterization of the impact of soft errors on iterative methods. In *HiPC*, pages 203–214, 2018.

[32] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, USA, 2nd edition, 2003.

[33] P. Sao and R. Vuduc. Self-stabilizing iterative solvers. In *ScalA*, 2013.

[34] A. Schöll, C. Braun, M. A. Kochte, and H. Wunderlich. Low-overhead fault-tolerance for the preconditioned conjugate gradient solver. In *DFTS*, 2015.

[35] A. Schöll, C. Braun, M. A. Kochte, and H. Wunderlich. Efficient algorithm-based fault tolerance for sparse matrix operations. In *DSN*, 2016.

[36] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78:012022, 2007.

[37] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Characterizing the impact of soft errors on iterative methods in scientific computing. In *ICS*, pages 152–161, 2011.

[38] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *ICS*, pages 69–78, 2012.

[39] J. Sloan, R. Kumar, and G. Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *DSN*, 2012.

[40] J. Sloan, R. Kumar, and G. Bronevetsky. An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance. In *DSN*, 2013.

[41] M. Snir and et al. Addressing failures in exascale computing. *Int. J. High Perform. Comput. Appl.*, 28(2), 2014.

[42] D. Tao, S. L. Song, S. Krishnamoorthy, P. Wu, X. Liang, E. Z. Zhang, D. Kerbyson, and Z. Chen. New-sum: A novel online ABFT scheme for general iterative methods. In *HPDC*, 2016.

[43] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun. Diagnosing performance variations in HPC applications using machine learning. In *ISC*, 2017.