An Algorithmic Approach to Error Localization and Partial Recomputation for Low-Overhead Fault Tolerance

Joseph Sloan, Rakesh Kumar
University of Illinois,
Urbana-Champaign
jsloan,rakeshk@illinois.edu

Greg Bronevetsky
Lawrence Livermore National Laboratory,
Livermore,CA
bronevetsky@llnl.gov

***Abstract*—The increasing size and complexity of massively parallel systems (e.g. HPC systems) is making it increasingly likely that individual circuits will produce erroneous results. For this reason, novel fault tolerance approaches are increasingly needed. Prior fault tolerance approaches often rely on checkpoint-rollback based schemes. Unfortunately, such schemes are primarily limited to rare error event scenarios as the overheads of such schemes become prohibitive if faults are common. In this paper, we propose a novel approach for algorithmic correction of faulty application outputs. The key insight for this approach is that even under high error scenarios, even if the result of an algorithm is erroneous, most of it is correct. Instead of simply rolling back to the most recent checkpoint and repeating the entire segment of computation, our novel resilience approach uses algorithmic *error localization* and *partial recomputation* to efficiently correct the corrupted results. We evaluate our approach in the specific algorithmic scenario of linear algebra operations, focusing on matrix-vector multiplication (MVM) and iterative linear solvers. We develop a novel technique for localizing errors in MVM and show how to achieve partial recomputation within this algorithm, and demonstrate that this approach both improves the performance of the Conjugate Gradient solver in high error scenarios by 3x-4x and increases the probability that it completes successfully by up to 60% with parallel experiments up to 100 nodes.**

Keywords*—*algorithmic error correction, partial recomputation, error localization, numerical methods, sparse linear algebra

I. Introduction

As High-Performance Computing (HPC) and other massively parallel systems grow more capable, they also grow larger and more complex. This means that as the number of components in the systems rises, so does the probability that one of them will suffer from a fault. Soft faults in chip circuitry are among the most worrying for system designers and application developers because they can corrupt the application's computations and produce incorrect output. Tera-scale systems are already vulnerable to soft errors, with ASCI Q experiencing 26.1 CPU failures per week [16] and a L1 cache soft error occurs about once every five hours on the 104K node BlueGene/L system at Lawrence Livermore National Laboratory [9]. Looking into the future, according to the International Technology Roadmap for Semiconductors, the soft error rates (SER) will grow with smaller chip feature sizes, with SRAM SER growing linearly with the number of transistors on a chip [2], which grows exponentially over time. This and the fact that many parallel systems of the foreseeable future will have hundreds of thousands to millions of electronic chips with feature sizes as low as 12nm [2] has led several recent studies [8] to warn that "traditional resiliency solutions will not be sufficient". Hardware-based approaches for fault tolerance have been proposed for many computing systems. However, their reliance on redundancy makes them impractical for future massively parallel systems because they will be severely power-constrained [8]. In fact, evolutionary extensions of today's high performance computing (HPC) systems (CrayXT, BlueGene) will be unable to reach exaFLOP performance by 2020 within a power budget of 20MW, the typical limit of modern computing centers [8].

The traditional approach for dealing with errors in systems is to roll the application back to a prior checkpoint whenever a fault is detected. This approach incurs a high cost in transferring checkpoint data [23]. Further, since expensive checkpoints result in long checkpointing periods [23], each rollback incurs a large cost in recomputing lost work. While this may be acceptable in scenarios where faults are rare, as fault rates increase with rising node counts and finer circuit features, the cost of full-application rollback may become prohibitive. Figure 1 shows the the performance of parallel linear solver, CG, using a traditional checkpoint-restart approach in the face of increasing fault rates.The results in Figure 1 assume that faults can be detected perfectly when they occur, thereby isolating the overhead due to application-level rollback. We observe that the overhead of application-level rollback reduces the performance of the solver by 2x-10x as fault rates increase.

In this paper, we propose a novel approach for fault tolerance that uses algorithmic correction of faulty application outputs based on *error localization* and *partial recomputation*. The key insight of our approach is that even under high error scenarios, a large fraction of the output is correct even if a portion of it is erroneous. Therefore, instead of simply rolling back to the most recent checkpoint and repeating the entire segment of computation, our approach identifies and corrects the actual subsegments of the output which are faulty. The correction technique we propose is algorithmic and leverages the properties of individual algorithms of interest to identify fault locations and limit the scope of recomputation. For example, errors in the output of a sorting algorithm can be localized by scanning through the results [28] and noting the ones that are mis-ordered, missing or new. A small number of

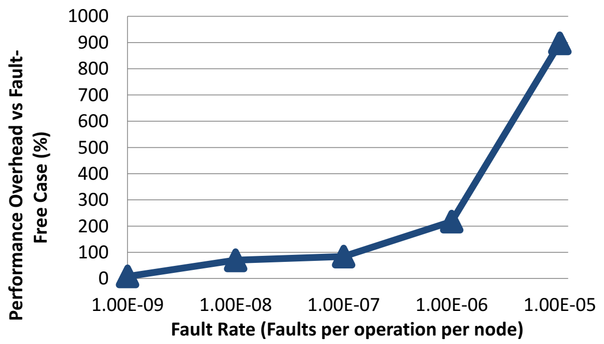

Fig. 1: Parallel CG Performance for different fault rates, when using traditional checkpoint-restart approach and assuming perfect fault detection, (Number of processor nodes=10, accuracy target=1e-6).

such errors can be corrected efficiently without repeating the entire sort.

This paper explores this concept in the context of numerical linear algebra in high error scenarios on parallel systems. It focuses on the matrix-vector multiplication (MVM) operation as well as iterative linear solvers. MVMs often dominate computation in many HPC and Recognition, Mining, and Synthesis (RMS) applications (see Section IV-E). We make the following contributions:

- We propose a *partial recomputation*- based approach for algorithmic correction, that is much more suited for high error rate scenarios than more traditional fault tolerance approaches, such as checkpoint/restart, which incurs high recovery costs.
- We propose a novel algorithmic technique for *error localization* (the process of identifying partitions of faulty and non-faulty outputs) for MVM operations.
- We show that the proposed techniques scale much better than traditional parallel fault tolerance approaches because they alleviate the performance bottlenecks that arise from high recovery costs.
- We quantify the performance benefits of *partial recomputation* and *error localization* in the context of parallel MVM and the parallel Conjugate Gradient (CG) iterative solver, which uses MVM internally under varying magnitude error rates. Our experiments show that while traditional detection/rollback has 2x-3x overhead under high fault rates, *Partial recomputation* is 2x cheaper while maintaining similar accuracy as ideal detection/rollback approaches. With more realistic detection schemes using dynamic thresholds, *partial recomputation*-based approaches are significantly more efficient (CG converges 70% more often and performance overheads 2x-3x smaller).
- We study the scalability of these techniques within the context of a parallel linear solver application for different parallel scales. For a fixed moderate fault rate, partial recomputation-based approaches with complete error localization reduce the overhead by up to 32% on average when scaled up to 10 nodes and 320% when scaled up to 100 nodes. Similarly, with the relaxed error localization routine the overhead is reduced by up to 77% at 10 nodes and up to 390% at 100 nodes.

By showing the utility of *partial recomputation* in the context of popular numerical algorithms we hope to demonstrate the value of research into *partial recomputation* in the context of a wider range of algorithms. Since this approach is significantly more efficient than whole-application recomputation and also significantly simpler than algorithmic correction techniques, we expect that this line of work will be extremely productive in ensuring cheap and effective resilience on future HPC systems and other massively parallel systems.

The paper is organized as follows. Section II describes related work and explains the limitations of prior checkpoint and rollback techniques. Section III describes the opportunities and approaches for low overhead algorithmic fault correction. Section IV discusses the methodology for evaluating the effectiveness of the techniques. Section V presents the results. SectionVI concludes.

II. Related Work

Checkpoint and rollback mechanisms have been the dominant approach for providing fault tolerance for HPC and massively parallel systems for decades [1, 19, 30, 5]. These approaches all rely on periodically saving the application and system state (i.e. address space, message buffers, and architectural state), so that if a fault is later detected, the system can simply restart from the saved prior state, rather than from the beginning of the application. Although this approach is exceptionally general, it can also incur prohibitive performance overheads under high error rate scenarios, due to large recovery costs.

For checkpoint-rollback based techniques, there does exist a tradeoff between the detection latency and the checkpoint frequency. For example, a system can increase the checkpointing frequency in order to reduce the detection latency and recovery costs [29]. However, this also significantly increases the checkpointing overhead in terms of performance, storage, and bandwidth, limiting the efficiency of this type of tradeoff. Our proposed approach instead uses algorithmic correction to partially recompute localized regions of output which are identified as being erroneous and reduce the recovery cost associated with rolling back and recomputation

Some previous studies have also identified checkpoint-rollback as significant limitation for future systems, mainly due to the storage and bandwidth overheads, and have proposed alternatives to checkpointing. In the context of permanent failures, Chen et. al. study the use of erasure-codes to recover lost data and eliminate more traditional checkpoints [6]. In this paper, we focus on transient computation faults which require an active detector. Also, rather than eliminating checkpoints entirely we propose the use of *error localization* to guide the algorithmic correction of errors by partial recomputation.

There is also much related work on algorithmic fault tolerance approaches. For linear solvers, some algorithmic techniques [25, 17] have been proposed that add additional inner/outer optimization loops to account for noisy computations. Researchers have also studied the use of linear error correcting codes [18] with algorithmic techniques for fault tolerance. The check for a linear operation, such as the matrix vector product (Ax where matrix A and vector x are inputs) detects faults by verifying that the following identity holds:

$$c^T(Ax) = (c^T A)x$$

Intuitively, the check computes the projection of the result Ax onto the vector c in two different ways. If there are any computation errors, the two projections will very likely be unequal (e.g. the difference between projections surpasses a given threshold, τ) In the common case where $c = \bar{1}$ (a vector of all 1's), the projection is equivalent to multiplying x by the vector containing the sums of matrix A's columns.

Consider for example, a $5x5$ matrix A and an input vector x:

$$A = \begin{pmatrix} 3 & 0 & 2 & 3 & 4 \\ 2 & 1 & 0 & 2 & 5 \\ 0 & 3 & 2 & 1 & 6 \\ 1 & 0 & 3 & 2 & 2 \\ 3 & 1 & 0 & 0 & 2 \end{pmatrix}, \; x = \begin{pmatrix} 5 \\ 5 \\ 7 \\ 1 \\ 2 \end{pmatrix}$$

The correct output of the matrix vector product is:

$$y = Ax = \begin{pmatrix} 40 \\ 27 \\ 42 \\ 32 \\ 24 \end{pmatrix}$$

Let say that an error (e) perturbs the correct output y into y':

$$y' = y + e, e = \begin{pmatrix} 3 \\ 5 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

In order to detect whether output y' is correct, take a vector c:

$$c = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Performing the error detection involves checking whether the checksum invariant holds ($c^T y' = c^T Ax$). In practice, the check invariant can be verified by computing the difference between the checksums (i.e. the syndrome) and comparing it to a threshold ($\tau = 0$). In this example, if the syndrome is not equal to zero an error is detected, otherwise the computation is deemed correct:

$$c^T y - (c^T A)x = 0 \quad \text{(OK, check invariant holds)}$$
$$c^T y' - (c^T A)x = 8 \quad \text{(Error(s) in the output!)}$$

The check works similarly for other linear operations as well.

Results from these checks can also be used for correction, however this can be expensive. At least d code vectors are required in order to correct at most $\lfloor d/2 \rfloor$ errors. Moreover, correction is heavily code dependent and not a trivial problem. (i.e. In order to correct multiple faults linear codes usually use a set of vectors each with unique non-binary codes and then the problem results in having to solve a non-trivial system of nonlinear equations [3]).

For this reason, traditional approaches commonly rely on checksum-based techniques for detection, but still rely on high overhead checkpoint-rollback for correction. In the above example, for example, if an error is detected (e.g. $c^T y' - (c^T A)x = 8$ signals an error), it will trigger a restart since the last checkpoint, which may result in high total overhead of recovery over the entire execution. Even recent work [27] on reducing the overhead of checksum-based techniques for detecting faults of sparse linear algebra, still relies on checkpoint-rollback for the correction of errors. This paper proposes an entirely different approach for correction - correction based on targeted algorithmic recomputation which, in turn, is enabled by error localization through a checksum-based technique. I.e., checksums are used to localize errors rather than just detect errors which allows for low overhead partial recomputation to be employed instead of high overhead full restart. To the best of our knowledge, this is the first work on algorithmic error localization and partial recomputation.

The ability to localize faults has been studied in the context of parallel program where researchers attempted to locate the cause of anomalies of parallel applications [22, 4]. Similarly, researchers have proposed approaches for identifying the physical location of detected hardware bugs [24]. This paper proposes the use of *error localization* during the runtime of the application to better guide partial recomputation.

Finally, there has been some work on algorithmic techniques to avoid conventional checkpoint-restart mechanisms in applications by transforming them into a form which is naturally more tolerant to errors. In [26], arbitrary applications are transformed into numeric optimizations problems due to the fact that the solvers for these problems are inherently robust to errors. A numeric optimization methodology provides a general path for making applications robust. The performance overhead incurred from this transformation varies across different applications. In general, if the complexity per iteration of the transformed application is less than the complexity of the original application the overhead is also low.

III. Partial Recomputation

The traditional approach of rolling back and repeating entire portions of applications upon the detection of faults can be prohibitively expensive under high error rates. Instead, we propose the use fine-grained *partial recomputation* to enable efficient forward progress. To identify segments of an application for fine-grained *partial recomputation* we will need to efficiently find the location of errors. These operations are possible in many algorithms and in this paper we demonstrate how it can be applied in the context of linear operations, focusing on matrix vector products (MVMs). MVMs often dominate computation in many HPC and RMS applications (see Section IV-E).

A. *Error Localization for Linear Operations*

Suppose that the MVM operation Ax (A is a matrix, x is a vector) outputs the vector $\hat{y}$ which may be equal to the correct result y or may contain errors. Errors in $\hat{y}$ can be detected by simply multiplying both Ax and result $\hat{y}$ by a check vector c that contains all 1s. The difference between $(c^T A)x$ and $c^T \hat{y}$ (identical to $c^T(Ax)$) is close to zero if there is no error (accounting for round-off error) and notably larger than zero if there is an error. Further, because the quantity $(c^T A)$ can be pre-computed and reused for all multiplications of A by a vector, this check is efficient, employing two dense dot-products (good memory locality) to check a sparse MVM (poor locality).

This basic algorithm can be extended to also identify the fault's location. Suppose that we replace all the 1's in the bottom half of c with 0's and repeat the above check. If there exist errors in the top half of $\hat{y}$, the difference $(c^T A)x) - c^T \hat{y}$ will be larger than zero but errors in the bottom half of $\hat{y}$ will have no effect on the result. This is true for any such variant of c. Let $c_{i,j}$ = { vector with 1's between indices i and j }. We can check if any entry i of $\hat{y}$ is erroneous by performing the above check using $c_{i,i}$ instead of c. Because $\hat{y}$ is large it is significantly more efficient to detect the location of each error hierarchically, checking for errors in each half of $\hat{y}$, then "zooming in" on the half of each region found to be erroneous. This procedure uses the tree of check vectors shown in Figure 2 and operates by starting at the top of the tree and proceeding downward. At each step the algorithm computes $(c_{i,j}^T A)x - c_{i,j}^T \hat{y}$, where $c_{i,j}$ is the vector at the current tree node. If a difference is detected, the algorithm recurses to each of the node's children. Any leaf nodes reached by this algorithm correspond to precisely detected errors in $\hat{y}$.

In the worst case scenario (i.e. every element of the result $\hat{y}$ is erroneous), the algorithm would perform a computation for every node, requiring $2n-1$ dot-products, where n is the size of $\hat{y}$. Fortunately, even under higher error rate scenarios, only a small fraction of output entries are likely to be corrupted. As such, only a fraction of the tree (O($log_2(N)$) will typically need to be traversed for any check. For this reason, our proposed *error localization* algorithm allows for much lower average overhead as opposed to prior approaches which use a fixed set of pre-designed codes for exact detection and correction properties.

Once the locations of errors in $\hat{y}$ are identified, they can be corrected via targeted correction. The MVM operation has the property that the element at index i in result vector y of Ax is the dot-product of row i of A by the vector x. This property makes it possible to correct the erroneous entries by simply recomputing them from x and the corresponding rows of A. Further, it can be observed that while the cost of identifying that an error lies within a given index of $\hat{y}$ requires two dot products of matrix row by a vector, while recomputing a given the same element of $\hat{y}$ requires just one dot product. This suggests that it may be more efficient to stop the fault localization procedure early and recompute a larger region of $\hat{y}$. In general, if our localization algorithm stops k levels short from the bottom then 2^k entries of $\hat{y}$ need to be recomputed. Section V experimentally explores this tradeoff. Another observation is that, in many scenarios small errors have little effect on the correctness of the algorithm that uses MVM. For instance, many iterative algorithms converge from a poor estimate of their result to an accurate estimate. As such, small errors in intermediate estimates will have little effect on convergence and it is thus more cost-effective to allow such errors than correct them. Our algorithm can be adjusted to meet the resilience needs of applications by using a threshold τ where the localization and correction procedure is only employed if the difference between $(c^T A)x)$ and $c^T \hat{y}$ is larger than τ.

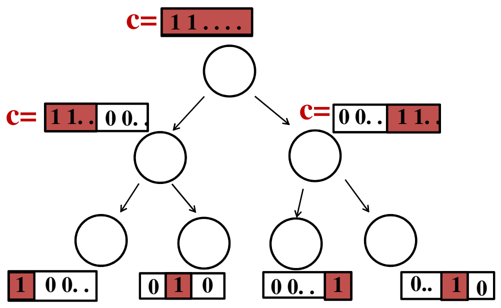

Fig. 2: The complete set of codes with $\|c\|^2 = 1$ (i.e. the basis of subspace) corrects all faults exactly by means of simply computing all the syndromes for this set.

1) *Example:* Let's consider the same input matrix A, input x, and error e from Section II. In order to construct a binary tree which can be utilized in the process of *error localization*, we need to construct a set of c vectors. As described in Section III-A, these vectors take the form:

$$c_{i,j} = \{ \text{ vector with 1's between indices i and j } \}$$
$$c_{i,i} = \{ \text{ vector with exactly one 1 at index i } \}$$

For the prior $5x5$ example, these c vectors are:

$$c_{0,4} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, c_{0,2} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}, c_{3,4} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix},$$
$$c_{0,1} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, c_{0,0} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, c_{1,1} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix},$$
$$c_{2,2} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, c_{3,3} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, c_{4,4} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Each of these codes are used to evaluate the check invariant ($c^T y' = c^T Ax$) at a node in tree the binary tree. Part of the second checksum ($c^T Ax$) may be precomputed or cached during the execution. The precomputed products ($c_i^T A = A^T c_i$) associated with the codes in this example are:

$$A^T c_{0,4} = \begin{pmatrix} 9 \\ 5 \\ 7 \\ 8 \\ 19 \end{pmatrix}, A^T c_{0,2} = \begin{pmatrix} 5 \\ 4 \\ 4 \\ 6 \\ 15 \end{pmatrix}, A^T c_{3,4} = \begin{pmatrix} 4 \\ 1 \\ 3 \\ 2 \\ 4 \end{pmatrix},$$

$$A^T c_{0,1} = \begin{pmatrix} 5 \\ 1 \\ 2 \\ 5 \\ 9 \end{pmatrix}, A^T c_{0,0} = \begin{pmatrix} 3 \\ 0 \\ 2 \\ 3 \\ 4 \end{pmatrix}, A^T c_{1,1} = \begin{pmatrix} 2 \\ 1 \\ 0 \\ 2 \\ 5 \end{pmatrix},$$

$$A^T c_{2,2} = \begin{pmatrix} 0 \\ 3 \\ 2 \\ 1 \\ 6 \end{pmatrix}, A^T c_{3,3} = \begin{pmatrix} 1 \\ 0 \\ 3 \\ 2 \\ 2 \end{pmatrix}, A^T c_{4,4} = \begin{pmatrix} 2 \\ 1 \\ 0 \\ 0 \\ 2 \end{pmatrix}$$

Performing *error localization* involves starting at the top of the tree (i.e. with the vector $c_{0,4}$) and evaluating the check invariant in order to detect if any errors occurred during the computation. By computing the difference in the checksums (i.e. the syndrome) at the top node of tree reveals that at least one error exists in the output:

$$c_{0,3}^T y' - (c_{0,3}^T A)x = 8 \quad \text{(Error(s) in the output (segment } [0,3])$$

Therefore, the algorithm proceeds to the next level of the tree which now considers the same check invariant, but with codes $c_{0,2}$ and $c_{3,4}$, which represent the first and second halves of the output, we can further narrow down that the error(s) must be located within the first half of the output.

$$\begin{aligned} c_{0,2}^T y' - (c_{0,2}^T A)x = 8 &\quad \text{(Error(s) in segment } [0,2]) \\ c_{3,4}^T y' - (c_{3,4}^T A)x = 0 &\quad \text{(OK, No errors in segment}[3,4]) \end{aligned}$$

With the error(s) localized to the first half of the output, we can ignore part of the binary tree corresponding to the second and proceed to isolate the errors within the first half. The codes $c_{0,1} and c_{2,2}$ locate the errors(s) within the first two elements of the output.

$$\begin{aligned} c_{0,1}^T y' - (c_{0,1}^T A)x = 8 &\quad \text{(Error(s) in segment } c_{0,1}) \\ c_{2,2}^T y' - (c_{2,2}^T A)x = 0 &\quad \text{(OK, No errors in y'[2])} \end{aligned}$$

Finally, unary codes $c_{0,0}$ and $c_{1,1}$ each locate and identify the specific magnitudes of both of the actual faults since they contain a c vector containing exactly one 1 (i.e. $\|c_4\| = \|c_5\| = 1$). Note that at any point in the traversal of the tree, the *error localization* process could be stopped and the segment of the output corresponding where faults are potentially located is recomputed (*partial recomputation*).

$$\begin{aligned} c_{0,0}^T y' - (c_{0,0}^T A)x = 3 &\quad \text{(Error of 3 at y'[0])} \\ c_{1,1}^T y' - (c_{1,1}^T A)x = 5 &\quad \text{(Error of 5 at y'[1])} \end{aligned}$$

B. Exploiting Sparsely Structured Products

Due to the sparse and structured nature of codes used within the *error localization* process, the runtime overhead of locating faults can be further reduced. Each code is vector comprised mostly of zeros except for one contiguous segment of ones. The feature can be exploited in both matrix vector products and the dot products that are used to compute the checksums. The first two procedures described in Algorithm III.1, illustrate how this structure can be exploited to reduce the overhead of both the matrix vector products and dot products used multiple times at every level of the tree. By passing pointers indicating the segment $start$ and end points, only the rows of matrix A for the matrix vector product and the rows of x, y for the dot product will be read. Algorithm III.1 assumes that the matrix A is in CSR format, which is represented by a row pointer array (pntr), an array of the values within the matrix (val), and an array of the column indices corresponding to those values (indx).

Based on the observation that the sparse matrix vector product of a sparse and structured vector is also typically sparse and structured (i.e. a sparse vector which is densely concentrated), we can also compute the dot product on the RHS of the check as a sparse dot product. This requires that we store pointers indicating the location of the nonzero elements in the vector. The sparse matrix vector product procedure in Algorithm III.1 illustrates how the pointers for the output vector can be easily computed as the inputs are scanned.

Note that on the LHS of the check invariant, we can also exploit the structure of the tree and compute half of the checksums, corresponding to the coded output (c_{ij}^y), from prior coded output calculations. However, a tradeoff exists by doing this since the accuracy of checksum can be significantly reduced (i.e. $3-5$ decimal points or $\pm 1e-2$ as opposed to $< 1e-6$ for the rounding error incurred from the typical checksum calculation). For our experiments in Section V we did not use this optimization due to the increase in floating point roundoff error.

The *error localization* routine can also exploit the associativity/distributivity of the linear operations which compose the checks to amortize the cost of the computation that pertains to static inputs (i.e. the matrix A). Similar to [27], the coded column sums ($c^T A$) can be precomputed beforehand and amortized over the execution of the entire application. Based on the observation that in the average case the binary tree does not need to be traversed entirely over the lifetime of the application, we can significantly reduce the cost of having to precompute all $2N-1$ coded column sum vectors ($c^T A$). Instead, we can simply compute the coded products "on the fly", as they are needed and then cache them in case they are needed in a later instance of the *error localization* routine.

1) Example: Performing the *error localization* process as described in Section III-A involves traversing a binary tree and computing a sequence of dot and matrix vector products at each level. The performance of these products is primarily dictated by the size of the inputs. With the prior example in Section III-A for example, the input A is a $5x5$ matrix, so each dot product computation scans over two vectors of length 5 and each matrix vector product scans over every element of the matrix and c vector.

Instead, we can perform the *error localization* process as described in Section III-B and exploit the sparse and binary nature of the operations within *error localization* to further reduce the overhead.

At the top level node of tree, the computation of the check invariant is similar to Section III-A, and uses the full matrix vector and dot product computation (superscripts represent elements of vectors, e.g. $c_{0,4}^i$ is the ith element of the vector $c_{0,4}$):

$$c_{0,4}^T y' - (c_{0,4}^T A)x = \sum_{i=0}^{4} c_{0,4}^i y'^i - \sum_{j=0}^{4} x^j \sum_{i=0}^{4} c_{0,4}^i A^{ij} = 8$$

Upon moving to the next level of the tree however, *error localization* can now compute the syndromes sparsely rather than using full matrix vector products and dot products. This new version of the process uses the equivalent of 2 dot and 1 matrix vector product as opposed to the original implementation from Section III-A which uses 4 dot products and 2 matrix vector products.

$$c_{0,2}^T y' - (c_{0,2}^T A)x = \sum_{i=0}^{2} c_{0,2}^i y'^i - \sum_{j=0}^{2} x^j \sum_{i=0}^{2} c_{0,2}^i A^{ij} = 8$$
$$c_{3,4}^T y' - (c_{3,4}^T A)x = \sum_{i=3}^{4} c_{3,4}^i y'_i - \sum_{j=3}^{4} x^j \sum_{i=3}^{4} c_{3,4}^i A^{ij} = 0$$

Similarly, as the algorithm proceeds to traverse the tree, the nodes associated with $c_{0,1}$ and $c_{2,2}$ now use the equivalent of about 2 dot products and half of a matrix vector product to compute the syndromes which originally cost 4 dot products and 2 matrix vector products.

$$c_{0,1}^T y' - (c_{0,1}^T A)x = \sum_{i=0}^{1} c_{0,1}^i y'^i - \sum_{j=0}^{1} x^j \sum_{i=0}^{1} c_{0,1}^i A^{ij} = 8$$
$$c_{2,2}^T y' - (c_{2,2}^T A)x = y'^2 - \sum_{i=0}^{4} x^i A^{2i} = 0$$

At the last level of the tree, the syndromes associated with these final nodes, which provide the actual correction due to having exactly one in their c vectors, are computed using the equivalent of roughly 2 dot products, which is again significantly less than the original 4 dot products and 2 matrix vector products required.

$$c_{0,0}^T y' - (c_{0,0}^T A)x = y'^0 - \sum_{i=0}^{4} x^i A^{0i} = 3$$
$$c_{1,1}^T y' - (c_{1,1}^T A)x = y'^1 - \sum_{i=0}^{4} x^i A^{1i} = 5$$

The sparsity of the codes used in the *error localization process*can be exploited as shown above to reduce the overhead of scanning over excess elements in the input vector (x), matrix A, and the code (c) that do not impact the syndrome output. At every level of binary search tree the number of dimensions that are scanned is typically reduced by a factor of two assuming that the error(s) are not pervasive and instead are spatially isolated.

Algorithm III.1: LOCATEFAULTS(x, y)

```
procedure MATVEC_SPARSE_BIN($A, start, end$)
  comment: A is in CSR format (pntr,val,indx)
  for $i \leftarrow start$ to $end$
    do for $j \leftarrow pntr[i]$ to $pntr[i+1]$
         do $y[indx[j]] += val[j]$
            if $indx[j] < start_{out}$
              then $start_{out} \leftarrow indx[j]$
            if $indx[j] > end_{out}$
              then $end_{out} \leftarrow indx[j]$
  return $(y, start_{out}, end_{out})$

procedure DOT_SPARSE($x, y, start, end$)
  $temp \leftarrow 0$
  for $i \leftarrow start$ to $end$
    do $temp += x[i]y[i]$
  return $(temp)$

main
  $sub_{cy}[0] = \sum_{i=0}^{N} y_i$
  for $k \leftarrow 0$ to $tree\_size$
    do if $finished[k] == 0$
         then
           $sub_{cy}[k] = \sum_{i=start[k]}^{end[k]} y_i$

           if $valid[k]$
             then $sub_{cA}[k] =$ MATVEC_SPARSE_BIN(
                                  $A, start[k], end[k]$)
                  $valid[k] = 1$

           $c_k^T Ax \leftarrow$ DOT_SPARSE($sub_{cA}[k], x,$
                                     $start[k], end[k]$)
           $syndrome = sub_{cy}[k] - c_k^T Ax$

           if $|syndrome| < \tau$
             then mark all children of k as finished
             else comment: Recompute segment k
                  $y_k \leftarrow A_k x$ (MATVEC_SPARSE())

           $finished[k] \leftarrow 1$
```

IV. METHODOLOGY

A. Fault Model

Our evaluation focuses on transient faults that affect the outputs of numerical computations. Other manifestations of transient faults, such as memory corruption, deviations of control flow or memory access errors are assumed to be accounted for by using simple low overhead techniques [12, 13], unless they manifest as numerical data errors, which the proposed techniques cover. This is a widely addressed fault model [18, 14, 11].

Faults are injected into the computation by instrumenting the application directly and calculating the random faults online. During execution the instrumentation code adds a random numeric error to the output of individual multiply/addition

Variable	Description
N	Number of Rows and Columns in A
$tree\_size$	Total number of segments in tree: $2N - 1$
$sub_{cA}[k]$	Storage of coded checksums (i.e. c^TA products
$valid[k]$	Indicates if $sub_{cA}[k]$ has been computed
$sub_{cy}[k]$	Storage of c^Ty products
$finished[k]$	Indicates if kth node completed (eliminated)
$syndrome[k]$	Syndrome computed for segment/node k
$c_{start}[k]$	Index of first element of segment k
$c_{end}[k]$	Index of one past the last element of segment k

TABLE I: Fault Localization Variables

operations used within matrix vector multiplication [27]. Faults are also similarly injected into the arithmetic operations that compose the checks themselves.

Over 50k runs were executed in order to get the statistical results shown in Section V. These experiments were run in parallel on HPC systems at LLNL (Sierra, Hera, and Atlas), where the distribution was analyzed and plotted using matlab and ggplot2 [31].

Since the timing of each fault is assumed to be independent, fault times are sampled from an exponential distribution with a rate λ. $\frac{1}{\lambda}$ is the expected number of arithmetic operations between consecutive faults. The fault rate of a system is defined as the probability that a given arithmetic operation has a fault and is equivalent to λ in our methodology.

Our experiments examine different fault rates that model phenomena ranging from physical faults arising from infrequent particle strikes (3-4 soft errors per day) to frequent errors arising from the use of aggressively designed (error-prone) technologies at large scales (multiple errors per second). A fault rate of 1e-8 corresponding roughly to soft errors which occur about 3 per day, and 1e-4 corresponds to aggressively designed systems, which trade off accuracy for energy for example, and exhibit multiple errors per second.

When a fault occurs, it is modeled by drawing a value from a fault distribution that models the arithmetic effects of circuit-level faults at a high level (i.e. a symmetric distribution with two Gaussian modes, centered at $1e5$ and $-1e5$ and with variance $1e2$) and adding it to the target operation. These error magnitudes distributions are representative of faults arising in arithmetic units from timing errors due to voltage over-scaling [27, 20].

Parameter	Description
Fault Rates	1e-8, 1e-7,4e-7,1e-6,4e-6,1e-5,1e-4
Fault Model	Symmetric distribution w/ two Gaussian modes
Parallel Nodes (N)	1,2,10,20,100

TABLE II: Fault Parameters

B. Benchmarks

The algorithmic fault detection techniques were implemented within the SparseLib library [10] of core sparse linear algebra operations, including matrix vector multiplication. To understand the effectiveness of our technique in a wide range of practical contexts, we evaluated them on 30 randomly chosen square linear systems from the University of Florida Sparse Matrix Collection and Matrix Market [7, 15] with the following properties: matrix size $\in [1000, 100000]$, symmetric, positive definite, and real. These are the most common parameters for matrices used in scientific computing applications and those used with linear solvers due to good convergence properties. These matrices represent a variety of physical phenomena and real algorithms, including model reductions, computational fluid dynamics, and circuit simulation. Table III lists the chosen matrices and their properties (nnz represents the number of non-zero elements, N represents the size of the problem, and Sparsity represents relative number of non-zeros per row, nnz/N).

We evaluate our fault detection techniques both in the context of an applications which utilizes matrix vector multiplication as a subroutine. The class of applications we have focused on in this study are sparse linear solvers since they are very common in computational science and make extensive use of matrix vector products. Sparse linear systems are commonly found in many different types of applications including HPC, Graph-based, and data-mining algorithms.

We consider one of the most common solvers, the Conjugate Gradient (CG) method.

CG is a popular solver well suited for very large and sparse symmetric positive definite problems. It expresses x as a linear function of n vectors $p_1, p_2, ...p_n$, with each pair of vectors conjugate in A ($p_iAp_j = 0$). Although the p_i's can be computed directly, in practice a small subset of the p_i's is needed to achieve accuracy within machine precision. As such, CG approximates the solution $x = q_1p_1 + ... + q_np_n$ with only a few vectors. The initial approximation is x_0; the residual $r_0 = b - Ax_0$, which is the direction of the error in x_0, serves as the first conjugate vector, p_0. Subsequent iterations compute the residual r_k and use it to compute the next conjugate vector p_k. To ensure that p_k is conjugate to prior p_i's, $p_k = r_k - \frac{r_{k-1}^T - r_{k-1}}{r_{k-2}^T r_{k-2}} p_{k-1}$. The coefficients α_k are computed as $\frac{r_k^T r_k}{p_k^T A p_k}$. This process is repeated until r_k falls below some threshold.

Solvers are run till a common fixed accuracy target of (1e-6) which is close to machine precision.

C. Parallel CG

The baseline CG implementation [10] was parallelized with MPI. The algorithm was parallelized by dividing the

Name	nnz	N	Sparsity	Type
nd3k	3279690	9000	364.41	3D Mesh Problem
bcsstk38	355460	8032	44.2555	Airplane Engine Component
Kuu	340200	7102	47.902	Mathworks Test Matrix
bcsstk16	290378	4884	59.455	Corp. of Engineers Dam
s2rmq4m1	263351	5489	47.978	FEM cylindrical shell
s3rmq4m1	262943	5489	47.9036	FEM cylindrical shell
s1rmq4m1	262411	5489	47.8067	FEM cylindrical shell
bcsstk28	219024	4410	49.6653	Solid Element Model
s2rmt3m1	217681	5489	39.657	FEM cylindrical shell
s3rmt3m1	217669	5489	39.6555	FEM cylindrical shell
s1rmt3m1	217651	5489	39.6522	FEM cylindrical shell
s3rmt3m3	207123	5357	38.664	FEM cylindrical shell
nasa2910	174296	2910	59.8955	Structure from NASA
Muu	170134	7102	23.9558	Mathworks Test Matrix
bcsstk24	159910	3562	44.8933	Winter Sports Arena
aft01	125567	8205	15.3037	Acoustic Radiation
bcsstk15	117816	3948	29.8419	Offshore platform
crystm01	105339	4875	21.608	FEM Vibration
nasa4704	104756	4704	22.2696	Structure from Nasa
ex9	99471	3363	29.5781	Test Matrix from FIDAP
ex15	98671	6867	14.3689	Test Matrix from FIDAP
msc04515	97707	4515	21.6405	Symmetric Test Mat
bcsstk13	83883	2003	41.8787	Fluid Flow
ex13	75628	2568	29.4502	Test Matrix from FIDAP
sts4098	72356	4098	17.6564	Structural Engineering Mat
nasa2146	72250	2146	33.6673	Structure from NASA
bcsstk14	63454	1806	35.1351	Roof of OMNI Coliseum
ex10hs	57308	2548	22.4914	Test Matrix from FIDAP
bcsstk27	56126	1224	45.8546	matrix Buckling Problem
ex3	54840	2410	22.7552	Sym. Powers of Graphs

TABLE III: List of Matrices and Properties

rows of the linear problem across a given set of N nodes, as shown in Figure 3. All of the internal state within CG (e.g. search direction p and residual r) were also divided in a similar manner so that every node contained a fraction of the entire linear system. By decomposing the algorithm in this manner, much of the actual CG code did not change, and instead only the implementations of the matrix vector product and dot product needed to be modified to account for the decomposition. Although a given node will only need it's local segment of the matrix A to compute the corresponding output segment, it will also need the entire input state (x). Similarly, the dot product operation requires that the nodes add the sum of all smaller local dot products to compute actual dot product. Therefore both the beginning of the matrix vector product and end of the dot products represent synchronization points for the parallel implementation of CG. For the parallel matrix vector product, this synchronization was implemented in MPI by gathering the x segments from all of the processes. For the dot products a sum reduction of all the locally computed dot products was used to gather the global dot product result.

All our experiments involved full application runs of the CG solver from the Iterative Method Library [10] that utilizes SparseLib for linear operations. Our techniques were integrated into the matrix vector routine in SparseLib. Over 50k runs were used to get the statistical results shown in Section V. These runs were executed on LLNL machines (system specs: Sierra: Intel Xeon 5660, 2.8GHz, 24GB memory per node, Hera: AMD Quad-Core Opteron 2.3GHz, 32GB Memory per node, Atlas: AMD Opteron, 2.4GHz, 16GB Memory per node).

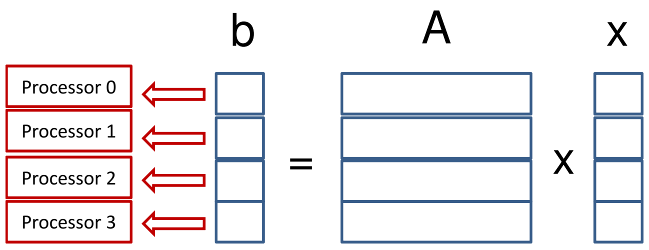

Fig. 3: Example Decomposition of Linear System ($y = Ax$) for Parallel CG implementation (N=4)

D. Other Implementation Details

Due to the nature of *partial recomputation* and *error localization* based approach, there exist several potential parameters for trading off accuracy and performance within the technique. This is advantageous for providing an interface for more application-specific algorithmic corrections. Some of these parameters include:

- Threshold (τ) which is used determine if branches of tree (segments of output) are pursued for further error localization can be adjusted according to application characteristics and requirements.
- Traversal of the tree can be stopped at any point (i.e. depth=d, which is the fraction of the entire tree's height), and correction can be applied to the segment.
- Roll forward correction as opposed to the typical rollback correction, using the projection of the error onto the code space, instead of roll back computation can be used to correct the errors in the identified segment.

Similar to the approach in [27], we can use decision trees to learn the best thresholds for the techniques. The first two are considered in the evaluations of partial recomputation in Section V, while the third is the focus of future work.

E. Generality

The proposed approach for algorithmic correction applies to any application which uses linear operations that are associative and has multiple outputs and/or intermediate states. This means that many scientific computing applications can benefit from these approaches since they rely heavily on mathematical operations at their core. Our proposed techniques rely on knowledge of the linear operations, not CG. In order to see the greatest benefits, the techniques do require that the application have some reuse of data in order to amortize the setup costs. However, because many HPC applications (PDE, ODE, Multigrid solvers, etc.) all exhibit this characteristic exactly and use linear operations iteratively, we expect the technique to apply to a large class of iterative methods dominated by linear operations. We also note that iterative solvers can take up to 80%-99% of the full application runtime for many HPC and scientific applications.

As another example, consider the FFT application, which is also a linear transformation similar to MVM. FFT produces an output vector X_k, such that $X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N}$. Due to linearity, part of the algorithm can also be expressed as a MVM operation. The matrix (M) used in the product is a symmetric matrix which represents: $M_{st} = e^{-i2\pi st/N}$. Using M, the FFT application can then be expressed as $X = Mx$ Similar to other MVMs, we can utilize a linear code to detect faults ($c^T(X) = (c^T M)x$) and the only precomputation that is required for localizing errors in the output of FFT ($c^T M$), is entirely independent of the input (x)

Linear systems and the linear operations that operate on them are at the core of many non-scientific applications as well. For example, many emerging applications (Recognition, Mining, and Synthesis (RMS) applications are increasingly being dominated by linear operations. RMS applications are increasingly being dominated by linear operations, such as dot products and matrix vector products, in order process large data sets in the most efficient manner. For example, the winner of the Netflix Prize competition 2009, which focused on analyzing large data sets of movie preferences and synthesizing unknown preferences, was primarily a numerical solution using matrix factorizations at it's core [21]. Other RMS applications which process large data sets are also heavily dominated by linear operations with which the proposed techniques for fault tolerance in this paper are directly applicable.

V. Results and Analysis

This section evaluates the performance and scalability of *partial recomputation* and *error localization* in the context of a common linear solver application, CG (Section IV-B).

We compare the performance of *partial recomputation* (PR) with traditional detect and rollback (DR) methods in this section. The baseline (DR) uses the common ABFT encoding scheme (a linear code) for detection. We also consider two instances of *partial recomputation*, one with full *error localization* (binary search depth of d=1) and one instance with a partial traversal of the localization tree ($d = 0.4$). We also consider two baseline detect and rollback (DR) approaches. One that assumes an ideal (oracle-based) detector and another with a more realistic detection-based approach that uses the traditional checksum/threshold comparison ($c^T y - c^T Ax \lessgtr \tau$). In this case, the threshold is chosen to dynamically adjust to the scale of the input ($\tau = t\|x\|$) [27]. In our experiments, we used a fixed scaling factor of $t = 1$ for the dynamically adjusted threshold. Both baseline implementations (DR) apply detection and rollback at the level of matrix vector operations. The input vector of the matrix vector product constitutes the application checkpoint.

Each technique is applied to an instance of the Parallel CG algorithm (N=10) on linear problems, defined by the matrices described in Section IV-B. These solvers are run until a fixed accuracy of 1e-6.

Figure 4 shows the performance overheads of each technique across varying fault rates ranging from 1e-8 to 1e-4 on the axis. The y-axis shows the performance overhead that is calculated for each matrix by:

$$\left(\frac{Time_{tech_i}}{Time_{NONE} \text{ with zero faults}} - 1 \right)$$

$Time_{tech_i}$ is the execution time of a given technique and $Time_{NONE}$ is execution time of the solver without any fault tolerance techniques applied. Each dot in the Figure 4 represents the average performance of a particular technique applied to a specific problem. The overhead is calculated in relation to runs with no fault tolerance and zero faults (as discussed in Section IV. Each line represents the mean overhead across all of the problems at each fault rate. The first observation from these results is that the detection and rollback techniques have a high cost ($> 200\%$) as the fault rate increases. For low fault rates (e.g., between 1e-8 and 1e-7), both the PR and DR based techniques have a similar mean overhead around 50%, but after a fault rate of 1e-7, the overheads of DR-based approaches increase to over 100% on average. This is expected due to the higher cost of recovery for these traditional approaches. If we observe the PR-based techniques, however, we see that the mean overheads increase much more slowly. In fact, PR-based techniques may have as much as 2x-3x less overhead. In particular, the PR based approach that only locates faults to within segments located at 0.4 the total height of the binary search tree, shows only a

30% average overhead from fault rates ranging from 1e-8 to 1e-4 (as opposed to up to 3x with DR - perfect detection).

If the fault rates increase past a certain point, nearly every entry of the output will end up being erroneous for the matrices we chose. For such fault rates, the benefits of PR-based approaches will diminish versus DR based approaches. However, for most real world problems, the range of fault rates before this "saturating point" is hit encompasses even the worst-case expected operating points for future HPC systems. For example, with the sparse matrices considered in this evaluation, the high fault rate scenarios (1e-6 – 1e-4) correspond to only 0.0001% to 0.01% of the output being corrupted. As systems and problems scale to even larger size (e.g. millions and billions of nodes), increasingly smaller fraction of output will be corrupted for any reasonable fault rate and as such the value of the PR-based approaches will only increase.

Figure 4 also shows that the overheads at any given fault rate can vary significantly based on the properties of the matrix. For example, for problems *bcsstk16* and *nd3k* the overheads are significantly larger for the techniques using practical detection schemes (PR-based and DR (t=1)). This is because the average magnitude of entries within these problems is very large, making it difficult to detect faults with a fixed threshold (*bcsstk16* and *nd3k* contain average magnitudes of $1e6$ and $1e-4$ respectively). Therefore, the detection and fault localization process is less accurate and incurs greater overheads from false negatives and false positives. Other problems are simply poorly conditioned and incur high overheads across all techniques (e.g. *nasa2910* has a condition number of 1e64). In general, the overheads for the different matrices are near their representative means ($\pm 30\%$).

Each of the solver instances is run under faults until it either hits the accuracy target or the limit on the maximum number of iterations ($10 x number of rows$). If the solver does not meet the accuracy target by the maximum number of iterations, it is considered as a failure. Figure 5 shows the success rate for each of the techniques over the same set of matrices and fault rates as Figure 4, on the x-axis. The solvers are run until convergence and for a maximum number of iterations (10*dimension of problem). As the fault rate increases, DR is less likely to make forward progress, and increasingly not able to meet the accuracy target by the maximum number of iterations. All the techniques, except DR using a realistic threshold, complete nearly 80% of the tested matrices. For solver instances using $DR(t = 1)$, the success rate drops off quickly going from 1e-8 to 1e-4 due to the high overhead of rollbacks and the overhead of extra iterations incurred due to missed faults. Solver instances using DR with perfect detection show good success rates until a fault rate of 1e-4 where the high overhead of rollback-based recoveries prohibits the solver from meeting the accuracy target by the iteration limit.

A. Scalability of Techniques

As the number of nodes in the system increases, we also expect the benefits of a *partial computation* based approach vs traditional detection to increase. In order to evaluate the scalability of the techniques, we fix the fault rate (1e-6) and run the same experiments with different numbers of nodes ($\{1, 2, 10, 20, 100\}$). A description of the parallel solver

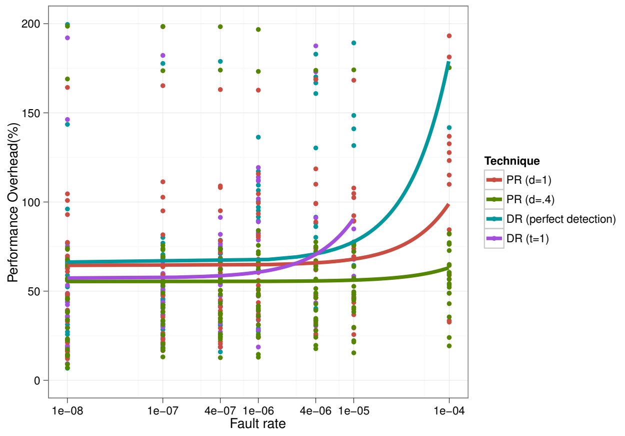

Fig. 4: Parallel CG Performance of techniques when scaling the fault rate, (N=10). At a fault of 1e-4 none of the DR experiments completed successfully (i.e. reached the accuracy target in maximum number of iterations).

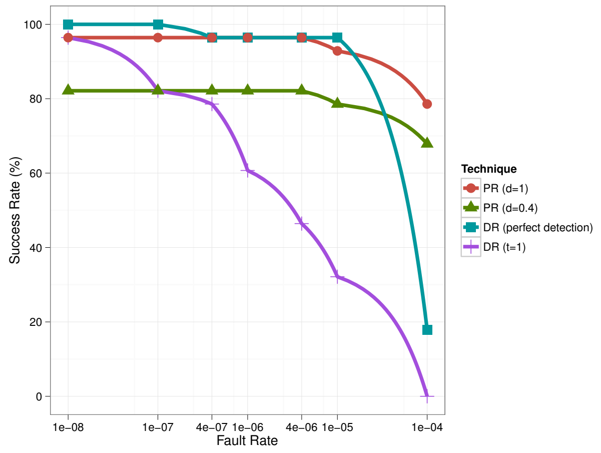

Fig. 5: Parallel CG Success Rate of techniques when scaling the fault rate, (N=10).

implemented with MPI is included in Section IV-C. Figure 6 illustrates the results of the experiments. Again, dots correspond to individual experiments and lines corresponding to the mean overheads. For these experiments, we calculated the overhead for each matrix by:

$$\left(\frac{Time_{tech_i} \text{ with N nodes and 1e-6 faults}}{Time_{NONE} \text{ with N nodes and zero faults}} - 1\right)$$

We see in Figure 6 that the benefit of PR-based approaches increases as the number of nodes scales up. At N=10, the average overhead of DR-based approaches over PR is 50%. As the number of nodes used in the solver is increased to 100, the overheads of DR over PR are even more pronounced, increasing over $300\% - 350\%$. These results indicate that

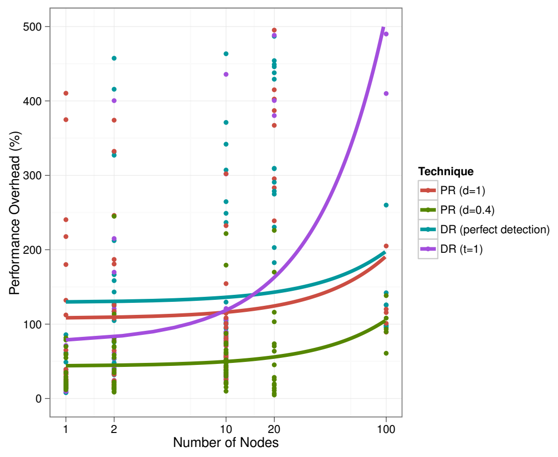

Fig. 6: Parallel CG Performance of techniques when scaling up the number of nodes from 1 to 100, with a fixed fault rate of 1e-6

the rollback recovery mechanism represents a significant bottleneck for traditional parallel fault tolerance mechanisms. Additionally, it shows that the scalability of these applications can be greatly approved by utilizing *error localization* and *partial recomputation* to alleviate this bottleneck.

VI. Conclusions

Future HPC and massively parallel systems will be prone to errors and severely energy constrained. For these systems, it will be critical for errors to be efficiently tolerated in order to ensure good forward progress. The traditional approach for dealing with errors in massively parallel systems is to roll the application back to a prior checkpoint whenever a fault is detected. However, this approach incurs a high cost in transferring checkpoint data [23] and a large cost in recomputing lost work. While this may be acceptable in scenarios where faults are rare, the cost of full-application rollback can be prohibitive for error prone HPC systems.

We propose a novel approach for algorithmic correction of faulty application outputs based on *error localization* and *partial recomputation*. The key insight of our approach is that even under high error scenarios, a large fraction of the output is correct even if a portion of it is erroneous. Therefore, instead of simply rolling back to the most recent checkpoint and repeating the entire segment of computation, our approach identifies and corrects the actual subsegments of the output which are faulty. By alleviating a key bottleneck associated with recovery, the parallel applications employing our fault tolerance techniques are able to scale significantly better. We explore this concept in the context of numerical linear algebra – the matrix-vector multiplication (MVM) operation as well as iterative linear solvers, in high error scenarios on parallel systems. Numerical linear algebra dominates computation in many HPC and RMS applications. Our experiments show that while traditional detection/rollback has 2x-3x overhead under high fault rates, partial recomputation is 2x cheaper while maintaining similar accuracy as ideal detection/rollback approaches. With more realistic detection schemes using dynamic thresholds, partial recomputation-based approaches are significantly more efficient (CG converges 70% more often and performance overheads 2x-3x smaller). For a fixed moderate fault rate, partial recomputation-based approaches with complete error localization reduce the overhead by up to 32% on average when scaled up to 10 nodes and 320% when scaled up to 100 nodes. Similarly, with the relaxed error localization routine the overhead is reduced by up to 77% at 10 nodes and up to 390% at 100 nodes.

Our results demonstrate the value of research into *partial recomputation* in the context of a wider range of algorithms. Since this approach is significantly more efficient than whole-application recomputation and also significantly simpler than algorithmic correction techniques, we expect that this line of work will be extremely productive in ensuring cheap and effective resilience on future massively parallel systems.

VII. Acknowledgments

This work was supported in part by NSF, ARO, GSRC, and the Department of Energy Early Career award program. It was also partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. We thank Tzanio Kolev and anonymous reviewers for their helpful feedback.

References

[1] Mpich-v. http://mpich-v.lri.fr.

[2] International Technology Roadmap for Semiconductors. White Paper, ITRS, 2010.

[3] J. Anfinson and F. T. Luk. A linear algebraic model of algorithm-based fault tolerance. *IEEE Trans. Comput.*, 37:1599–1604, December 1988.

[4] Greg Bronevetsky, Ignacio Laguna, Saurabh Bagchi, Bronis R. de Supinski, Dong H. Ahn, and Martin Schulz. AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks. In *2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 231 –240, Chicago, IL, 2010. resilience.

[5] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of mpi programs. *SIGPLAN Not.*, 38(10):84–94, June 2003.

[6] Zizhong Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 73–84, New York, NY, USA, 2011. ACM.

[7] Timothy A. Davis. University of florida sparse matrix collection. *NA Digest*, 92, 1994.

[8] Berman et. al. Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead. Technical report, DARPA IPTO, SEP 2008.

[9] J. N. Glosli et. al. Extending stability beyond cpu millennium: a micron-scale atomistic simulation of kelvin-helmholtz instability. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 58:1–58:11, New York, NY, USA, 2007. ACM.

[10] Jack Dongarra Andrew et. al. A sparse matrix library in c++ for high performance architectures, 1994.

[11] Keun Soo Yim et. al. Hauberk: Lightweight silent data corruption error detector for gpgpu. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, 2011.

[12] Man-lap Li et. al. Swat: An error resilient system, 2008.

[13] Nahmsuk Oh et. al. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51:111–122, 2002.

[14] Nithin Nakka et. al. An architectural framework for providing reliability and security support. In *In DSN*, pages 585–594. IEEE Computer Society, 2004.

[15] Ronald F. Boisvert et. al. Matrix market: A web resource for test matrix collections. In *The Quality of Numerical Software: Assessment and Enhancement*, pages 125–137. Chapman and Hall, 1997.

[16] Sarah Michalak et. al. Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratorys ASC Q Supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3), 2005.

[17] Michael A. Heroux and Mark Hoemmen. Fault-tolerant iterative methods via selective reliability. Technical report, Sandia National Laboratories Technical Report, 2011.

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

[19] P. Hargrove J. Duell and E. Roman. The design and implementation of berkeley labs linux checkpoint/restart. Technical report, Berkeley lab Technical Report, 2002.

[20] C. Kong. Study of voltage and process variation's impact on the path delays of arithmetic units. In *UIUC Master's Thesis*, 2008.

[21] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems, 2009.

[22] A.V. Mirgorodskiy, N. Maruyama, and B.P. Miller. Problem diagnosis in large-scale computing environments. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, page 11, nov. 2006.

[23] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *ACM/IEEE Conference on Supercomputing*, 2010.

[24] Sung-Boem Park, Anne Bracy, Hong Wang, and Subhasish Mitra. Blog: post-silicon bug localization in processors using bug localization graphs. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 368–373, New York, NY, USA, 2010. ACM.

[25] Valeria Simoncini, Daniel, and B. Szyld. Theory of inexact krylov subspace methods and applications to scientific computing. Technical report, 2002.

[26] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi. A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance. In *Dependable Systems and Networks (DSN), 2010*, June 2010.

[27] J. Sloan, R. Kumar, and G. Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *Dependable Systems and Networks (DSN), 2012*, 2012-july 1 2012.

[28] Gregory F. Sullivan, Dwight S. Wilson, and Gerald M. Masson. Certification of computational results. *IEEE Transactions on Computers*, 44:833–847, 1995.

[29] Diana Szentivanyi, Simin Nadjm-Tehrani, and John M. Noble. Optimal choice of checkpointing interval for high availability. In *Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing*, PRDC '05, pages 159–166, Washington, DC, USA, 2005. IEEE Computer Society.

[30] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. A job pause service under lam/mpi+blcr for transparent fault tolerance. In *In International Parallel and Distributed Processing Symposium*, pages 26–30, 2007.

[31] Hadley Wickham. *ggplot2: elegant graphics for data analysis*. Springer New York, 2009.