Adaptive Erasure Coded Fault Tolerant Linear System Solver

XUEJIAO KANG, DAVID F. GLEICH, AHMED SAMEH, and ANANTH GRAMA,
Purdue University

As parallel and distributed systems scale, fault tolerance is an increasingly important problem—particularly on systems with limited I/O capacity and bandwidth. Erasure coded computations address this problem by augmenting a given problem instance with redundant data and then solving the augmented problem in a fault oblivious manner in a faulty parallel environment. In the event of faults, a computationally inexpensive procedure is used to compute the true solution from a potentially fault-prone solution. These techniques are significantly more efficient than conventional solutions to the fault tolerance problem.

In this article, we show how we can minimize, to optimality, the overhead associated with our problem augmentation techniques for linear system solvers. Specifically, we present a technique that adaptively augments the problem only when faults are detected. At any point in execution, we only solve a system whose size is identical to the original input system. This has several advantages in terms of maintaining the size and conditioning of the system, as well as in only adding the minimal amount of computation needed to tolerate observed faults. We present, in detail, the augmentation process, the parallel formulation, and evaluation of performance of our technique. Specifically, we show that the proposed adaptive fault tolerance mechanism has minimal overhead in terms of FLOP counts with respect to the original solver executing in a non-faulty environment, has good convergence properties, and yields excellent parallel performance. We also demonstrate that our approach significantly outperforms an optimized application-level checkpointing scheme that only checkpoints needed data structures.

CCS Concepts: • **Computing methodologies** → *Parallel computing methodologies*; *Parallel algorithms*

Additional Key Words and Phrases: Fault tolerance, linear solver, adaptive fault tolerance

ACM Reference format:
Xuejiao Kang, David F. Gleich, Ahmed Sameh, and Ananth Grama. 2021. Adaptive Erasure Coded Fault Tolerant Linear System Solver. *ACM Trans. Parallel Comput.* 8, 4, Article 21 (December 2021), 19 pages.
https://doi.org/10.1145/3490557

1 INTRODUCTION

Fault tolerance for parallel and distributed computations is a significant challenge, particularly at scale. The corresponding problem for storage has been effectively addressed with development of erasure codes, which augment data with redundant blocks so that in the event of an erasure (e.g., disk failure), data can be reconstituted using remaining (non-erased) coded blocks. A number of

This work was funded by the U.S. Department of Energy, DE-SC0014543.
Authors' address: X. Kang, D. F. Gleich, A. Sameh, and A. Grama, Purdue University, Department of Computer Science, 305 North University Street, West Lafayette, IN 47907; emails: {kang138, dgleich}@purdue.edu, {sameh, ayg}@cs.purdue.edu.

2329-4949/2021/12-ART21 $15.00
https://doi.org/10.1145/3490557

codes have been developed that minimize storage overhead, computational cost for coding and reconstitution of data, and communication in distributed environments.

In a recent set of results [26], we have developed a novel concept called *erasure coded computations* that generalizes erasure codes from storage to computations. The theory underlying erasure coded computations has been developed in the context of linear system solvers and eigenvalue problems. The basic idea is to augment an input matrix with a suitable set of coded rows and columns to generate an augmented problem. This augmented problem is then solved using conventional solvers (e.g., **Conjugate Gradient (CG)**) on a faulty ensemble of parallel processors in a fault-oblivious manner. For instance, when a processor fails (we assume a fail stop failure), the rest of the processors simply continue with their computation. In this manner, a solution is computed for the augmented problem. In the event of a fault, a computationally inexpensive reconstruction procedure is used to recover the original solution (to the original problem instance) from the augmented solution. In the work of Zhu et al. [26], we derive a number of theoretical results relating to necessary conditions for augmentation blocks, recovery algorithm, and associated costs.

While providing a proof of concept for erasure coded computations, past results [16] make several assumptions that are problematic in practice. They rely on static coding blocks, based on the assumption of a known maximum number of faults. The solver fails if the actual number of faults exceeds this bound. Second, the overhead of the coding blocks is paid, irrespective of the number of errors. Stated otherwise, even if the actual number of faults is lower, the cost associated with redundant blocks corresponds to the worst case. Third, the addition of a redundant blocks irrespective of erasures adds a null space into the coded matrix. This often manifests in slower convergence rates for the solver operating on the coded matrix. Finally, the coding blocks induce communication in distributed execution. This communication pattern does not follow the communication induced by the original matrix. Having a large coding block complicates parallel formulations of the solver and degrades parallel performance.

Motivated by these shortcomings, in this article we present the next step in the development of practical and efficient erasure coded computations, which we refer to as **Adaptive Erasure Coded Computations** (AECC). The idea behind AECC is that coded rows and columns are added only as faults are detected. Specifically, the solver operates with no coding blocks until a fault is detected. In the event of faults, only the required number of rows and columns are added. In this manner, the solver only ever operates on a system of size $n \times n$. This has a number of desirable properties: (i) the method is computation-optimal in the sense that the system size stays the same as the input system; (ii) the convergence properties are maintained (i.e., the system is always full rank, and if the input is **symmetric positive definite (SPD)**, the coded system is also SPD); and (iii) the coded block adds negligible computational and parallel overhead to the base solver. We argue that a combination of these properties makes AECC an ideal fault tolerant solver.

We present the theoretical underpinnings of AECC, coding and solution reconstruction techniques, as well as parallel formulation of the AECC solver. We support all of our theoretical constructions using a parallel implementation and validate various desirable features of our AECC solver. We also present comparisons to a highly optimized checkpointing scheme, which only stores necessary data structures, at optimally tuned intervals. We show significant improvements in efficiency and scalability for our erasure coded scheme over checkpointing.

We make the following specific contributions in the article:

- We present a novel AECC scheme for linear system solvers that is computation-optimal with respect to base solver.
- We derive theoretical underpinnings of AECC, including coding and reconstruction techniques.

- We present a parallel implementation of AECC and demonstrate excellent performance in terms of convergence rates, scalability, and robustness to different fault arrival models.
- We present a comparison with an optimized checkpointing scheme and show significant performance benefits of AECC.

2 BACKGROUND

Techniques such as checkpoint-restart and active replicas are commonly used for fault tolerance in parallel systems. Checkpoint-restart relies on consistent checkpoints, which are often hard to find without significant rollbacks, particularly in scalable parallel programs that attempt to minimize global synchronizations. They also require significant bandwidth (to store checkpoints in persistent storage, or across multiple nodes for in-memory checkpoints [12, 18]) and available disk or extra memory capacity. Scalable systems with hundreds of thousands of cores and beyond are often constrained in all of these resources. Active replicas [22], however, are typically used in real-time systems, where worst-case execution times have to be guaranteed. In this case, critical computations are replicated and a consensus protocol detects and corrects errors. Active replicas have significant resource overheads.

In the domain of distributed computations, faults are often handled through techniques such as deterministic replay in systems such as MapReduce [4, 10]. In such systems, the reduce step provides intermediate checkpoints. Computations (maps) are executed and monitored by a runtime system. Failed maps are rescheduled to ensure fault tolerance. Such techniques are dependent on two critical aspects: periodic reduce phases that act as consistent checkpoints (global synchronizations) so that maps are only rescheduled from the last reduce point, and committing output of reduce phases to persistent storage (typically to a redundant distributed file system). The two overheads of these schemes are also apparent. First, the makespan of jobs increases as faults increase, since the runtime system must reschedule maps, thus slowing down the entire computation. Second, the overhead of committing output of reduce phases can itself be significant.

An alternate class of techniques, broadly classified as algorithm-based fault tolerance (ABFT), redesigns the algorithm to make it resilient to faults [1, 2, 5–8, 11, 15]. As an example, for linear system solvers, a faulty solver may be viewed as a preconditioner for an outer solve, which validates the solution of the potentially faulty inner solve. Although these techniques alleviate many of the performance and resource constraints of system-supported fault tolerance techniques, they require intricate design of the algorithm, along with associated proofs of correctness and characterization of performance.

In the domain of storage, fault tolerance has been addressed using replication and erasure coding techniques. Replication techniques store data at $s+1$ distinct sites to tolerate s failures. The resource overheads of such techniques are significant. Erasure coding techniques [20], however, augment data with codes that enable recovery in the event of a failure. Erasure codes may be viewed as multiplication of the data (viewed as a vector) by a matrix that satisfies certain rank properties. Given an n dimensional data vector, erasure codes multiply the data vector by an $(n + k) \times n$ matrix (i.e., a matrix with $n+k$ rows and n columns) to generate an $n+k$ dimensional coded vector. The coding matrix is designed in such a way that any subset of n rows of the matrix is linearly independent. In the event of up to k failures, one simply inverts the rest of the matrix and multiplies with the available data items to extract the original n dimensional data vector. Such techniques have been used with great success in systems such as RAID (Redundant Array of Inexpensive Disks) and wide area file systems such as Google's Colossus. The precise structure of the coding matrix is determined by desired level of fault tolerance, as well as the overhead of constituting the coded data and computing original data in the event of a failure. The key benefit of erasure coding

is that it requires $n + k$ storage to tolerate k faults, as compared to $n(k + 1)$ storage in the case of replication. We realize the same benefit for computation, through our erasure coded computation analog.

Our overall approach is best characterized as adapting the ideas behind erasure coded fault-tolerant storage to create fault tolerant problem formulations, and using the same base algorithm to solve these augmented problems. As a further advance, this article presents a novel technique that adaptively codes problems as faults occur, thus minimizing overhead of fault tolerance.

3 AN ERASURE CODED LINEAR SYSTEM SOLVER

We begin with a brief description of the base erasure coded linear solver to provide necessary background. We refer readers to the work of Zhu et al. [26] for the theoretical underpinnings of erasure coded computations. The purpose here is simply to provide necessary background and motivation for the parallel adaptive fault tolerant solver, which is the topic of our current work.

Given a linear system, $\boldsymbol{A}\mathbf{x} = \mathbf{b}$, we first construct a coding matrix $\boldsymbol{E}$ so that $\boldsymbol{E}^T$ has Kruskal rank k, to tolerate a maximum of k faults. Note that Kruskal rank of k implies that any subset of k columns of $\boldsymbol{E}^T$ is guaranteed to be linearly independent. Such coding matrices can be constructed using traditional coding techniques, using Vandermonde matrices or sparse low-density parity codes (LDPC). Using matrices $\boldsymbol{A}$ and $\boldsymbol{E}$, we build an encoded system to provide fault tolerance. The augmented or encoded system, one solution to the augmented system, and the augmented right hand side, are given by the following equation.

$$\tilde{\boldsymbol{A}} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{AE} \\ \boldsymbol{E}^T\boldsymbol{A} & \boldsymbol{E}^T\boldsymbol{AE} \end{bmatrix} \qquad \tilde{\mathbf{x}}^* = \begin{bmatrix} \mathbf{x}^* \\ 0 \end{bmatrix} \qquad \tilde{\mathbf{b}} = \tilde{\boldsymbol{A}}\tilde{\mathbf{x}}^* = \begin{bmatrix} \mathbf{b} \\ \boldsymbol{E}^T\mathbf{b} \end{bmatrix}$$

Note that this system is singular, and there are multiple solutions. With these augmented data structures, we find any solution of the new system,

$$\tilde{\boldsymbol{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}, \tag{1}$$

using traditional parallel techniques. In the event of a fault (we assume a fail-stop failure), the non-faulty processors continue their computations in a fault-oblivious manner until the solver converges.

When $\boldsymbol{E}^T$ has Kruskal rank k, Zhu et al. [26] prove a number of properties of this augmented system that guarantee that solution recovery is possible. If a subset of up to k rows and columns of matrix $\tilde{\boldsymbol{A}}$ were to be erased (due to a fail-stop processor failure), along with associated elements of intermediate solution $\tilde{\mathbf{x}}$ and right-hand side $\tilde{\mathbf{b}}$, the solver continues computations on remaining parts of the augmented system. If the original matrix $\boldsymbol{A}$ is SPD, it can be shown that the remaining system that survives erasures is symmetric semi-definite, and that Krylov subspace methods converge to a solution for this partially erased augmented system. We can recover true solution $\mathbf{x}^*$ from the solution returned by the fault-oblivious solver. Specifically, let $\left[\begin{smallmatrix} \mathbf{y} \\ \mathbf{z} \end{smallmatrix}\right]$ be any solution to the augmented system, where $\mathbf{y} \in \mathbb{R}^n$. To recover the true solution $\mathbf{x}^*$ in the presence of errors, we only need to compute [26]

$$\mathbf{x}^* = \mathbf{y} + \boldsymbol{E}\mathbf{z}. \tag{2}$$

This gives us a straightforward and computationally inexpensive recovery algorithm.

Note that the solver operates on the augmented system of size $(n + k) \times (n + k)$. For large values of k, this can add significant overhead, particularly when the augmentation blocks have significant fill (non-zeros). This overhead is incurred by the solver, independent of the number of faults encountered in the execution, since the value of k is chosen based on the worst-case estimate of number of faults.

4 CODING MATRIX FOR PARALLEL IMPLEMENTATION

The results from Zhu et al. [26] are designed such that *any combination* of failures can be handled. Existing simple techniques to achieve this involve matrices $\boldsymbol{E}$ with Kruskal rank-k, which then cause large degrees of *fill-in* for the coded system $\boldsymbol{AE}$. Given that over short-enough time windows, we expect computations to be successful, we investigate strategies to relax this strict requirement to enable methods that will work better in practice. This motivates the following weaker definition used in the work of Kang et al. [16].

Definition. A n-by-k matrix $\boldsymbol{E}$ satisfies the recovery-at-random property if a random subset of k rows (selected uniformly with replacement) is rank k with probability $\geq 1 - 1/n^c, c > 1$.

This definition admits a much wider class of coding matrices. A simple construction for one such matrix is to use staggered matrices as in Figure 2(a). These have k columns and runs of length p elements that wrap-around. An example with $k = 6, p = 3$ is as follows.

$$\boldsymbol{E} = \begin{bmatrix} x & x & x & 0 & 0 & 0 \\ 0 & x & x & x & 0 & 0 \\ 0 & 0 & x & x & x & 0 \\ 0 & 0 & 0 & x & x & x \\ x & 0 & 0 & 0 & x & x \\ x & x & 0 & 0 & 0 & x \\ x & x & x & 0 & 0 & 0 \end{bmatrix}$$

These are motivated by a random coding of a small window of the system to promote sparsity in the final result. Formally, for the i^{th} row, the $j = (i-1+s) \mod k$ elements for s ranging from 1 to p are set to random reals in the range $(0, 1)$. We call this n-by-k matrix a p-staggered distribution.

These matrices have a number of useful properties including the the recovery-at-random property.

Proposition 4.1. *Let $\boldsymbol{E}'$ be a submatrix of $\boldsymbol{E}$ formed by selecting at most p rows of matrix $\boldsymbol{E}$, then the $\boldsymbol{E}'$ matrix has full row rank. This shows that any collection of up to p rows of matrix $\boldsymbol{E}$ are linearly independent.*

Proof. The two keys to our proof are that *wide matrices of random entries (more columns than rows)* with the same non-zero structure are non-singular and that matrices with distinct non-zero patterns are non-singular if any subset of the rows with the same non-zero pattern are non-singular. The full argument simply combines these two pieces. Consider a matrix of up to p rows where all of the non-zero structures are the same. This full row rank r-by-p matrix of random uniform $[0, 1]$ entries is rank r if $r \leq p$. If there are distinct non-zero patterns, we simply repeat the argument on each distinct non-zero pattern, which shows that it is non-zero. The base case is a single row, which is the simplest case of a full row-rank matrix. □

Proposition 4.1 shows that any submatrix of at most p rows of matrix $\boldsymbol{E}$ has full rank. But because we want to use this to correct failure, to use theory similar to Zhu et al. [26], we would need that any submatrix of k rows of matrix $\boldsymbol{E}$ must have rank k. Clearly, there exist degenerate cases where this is not true—specifically, if we select $k > p$ rows, each with the same non-zero structure, we end up with a submatrix of rank only p, which is less than k. Here, it is important to appreciate that when this result is used, the rows chosen correspond to failures or fault. We now show that random collections are likely to be full rank, which will help us prove the recovery at random result.

Theorem 4.2. *Let $k \leq e\sqrt{p+1}$. The probability that a random set of k rows of a matrix $\boldsymbol{E}$ drawn from the n-by-k p-staggered distribution is linearly dependent is less than $(\frac{e}{p+1})^{p+1}$.*

Proof. A necessary and sufficient condition for k rows to be linearly dependent is that some selection of $p+1$ of these k rows have the same non-zero structure.

Note that there are k distinct non-zero structures in the n rows of matrix $\boldsymbol{E}$. Furthermore, since rows are uniformly assigned one of these k distinct row structures, the probability that a row has a specific row structure is $1/k$ and the probability that $p+1$ rows have the same row structure is $(\frac{1}{k})^p$. Since there are $\binom{k}{p+1}$ ways to select $p+1$ rows out of the selected block of k rows, the probability that a selected block of k rows is linearly dependent is given by

$$\binom{k}{p+1} \cdot \left(\frac{1}{k}\right)^p \leq \frac{k}{e\sqrt{p+1}} \left(\frac{e}{p+1}\right)^{p+1}. \qquad \square$$

As p increases, it is easy to see that this probability rapidly approaches 0. Stated otherwise, matrix $\boldsymbol{E}$ satisfies recovery-at-random for p chosen as a suitable function of n. The following analysis generalizes the situations. Note that there are k distinct non-zero structures in the p-staggered distribution, and this was key to the proof of the previous result. We can analyze the expected maximum number of duplicate non-zero structures from a set of k in the following result.

Theorem 4.3. *Let $\boldsymbol{E}$ be a matrix where each row has one of k non-zero patterns and the pattern in a random row occurs with probability $1/k$. The maximum number of rows from among k randomly selected rows of matrix $\boldsymbol{E}$ that have same non-zero structure is $(\frac{\ln k}{\ln \ln k})(1 + o(1))$ with high probability.*

Proof. This result is equivalent to existing arguments for *balls-and-bins* problems that arise in the study of hashing. In this equivalent setup there are k bins (one for each type of row non-zero pattern). Then our selection of k random rows consists of "dropping" k balls into these k bins, where each ball chooses a bin with probability $1/k$. Our question is then what is the maximum number of balls in any bin—the maximum number of rows with the same non-zero structure. This was originally shown in the work of Gonnet [14] and later re-derived in the work of Raab and Steger [21]. Both show that the maximum is $(\frac{\ln k}{\ln \ln k})(1 + o(1))$ with probability $1 - o(1)$. $\square$

These results show that we will have recovery at random for up to k faults using an n-by-k coding matrix constructed using a staggered non-zero pattern with p-non-zeros, when p is larger than $\log k / \log \log k$. Notably, since $p \leq k$, we need $\log k / (\log \log k) \leq k$, which occurs when $k \geq 5$. Consequently, we use $p = \min(k, 5)$, which, with high probability, guarantees recovery at random for k smaller than a few hundred thousand.

5 PARALLEL IMPLEMENTATION OF ERASURE CODED CG

A simple approach to solve the augmented system with $\tilde{\boldsymbol{A}}$ in parallel is to use a version of CG that will enable us to reset the recurrence whenever a fault is detected. The two-term CG [19], instead of the three-term CG, is such a method. This was used by Zhu et al. [26] to solve systems where $\boldsymbol{A}$ is SPD. The algorithm and the reset procedure are given in Algorithm 1.

When this is executed in parallel [16], there are a few relevant details that motivate our new work on adaptive solvers. First, consider the case where the augmented matrix $\tilde{\boldsymbol{A}}$ and the augmented vectors can be distributed among multiple processes by rows. Alternate formulations with 2D partitioning are also feasible within our coding framework, although we limit our discussion for simplicity. Let the index set associated with process i be $\mathcal{I}_i$, then $[n+k] = \bigcup_i \mathcal{I}_i$ and the set of

ALGORITHM 1: Fault oblivious CG with a two-term recurrence. When we notice a fault, we set $\beta_t = 0$ at that iteration.

Let $\mathbf{x}_0$ be the initial guess and $\mathbf{r}_0 = \mathbf{b} - \boldsymbol{A}\mathbf{x}_0$, $\beta_0 = 0$.
for $t = 0, 1, \ldots$ until convergence **do**
  $\beta_t = (\mathbf{r}_t, \mathbf{r}_t)/(\mathbf{r}_{t-1}, \mathbf{r}_{t-1})$
  $\mathbf{p}_t = \mathbf{r}_t + \beta_t \mathbf{p}_{t-1}$
  $\mathbf{q}_t = \boldsymbol{A}\mathbf{p}_t$
  $\alpha_t = (\mathbf{r}_t, \mathbf{r}_t)/(\mathbf{q}_t, \mathbf{p}_t)$
  $\mathbf{x}_{t+1} = \mathbf{x}_t + \alpha_t \mathbf{p}_t$
  $\mathbf{r}_{t+1} = \mathbf{r}_t - \alpha_t \mathbf{q}_t$
end for

faulty indices is $\mathcal{F}_t$. Now consider what happens to the operations of Algorithm 1 and how they are affected by faults in a distributed environment. The two non-local steps are the aggregation operations: inner products and the matrix-vector multiplication. After a fault, the viable processes carry out the all-reduce operation in an inner product $(\mathbf{r}_t, \mathbf{r}_t)$ or $(\mathbf{q}_t, \mathbf{p}_t)$ by simply skipping the faulty components $\mathcal{F}_t$ in the vectors; see other works [16, 26] for additional details. For the matrix-vector multiplication $\mathbf{q}_t = \boldsymbol{A}\mathbf{p}_t$, each process has a block of the matrix $\boldsymbol{A}_{\mathcal{I}_i,:}$ (using Matlab notation), a viable process carries out its local aggregation operation for computing $\boldsymbol{A}_{\mathcal{I}_i,:}\mathbf{p}_t$ by again skipping the faulty components $\mathcal{F}_t$ in $\mathbf{p}_t$.

Another technical issue we need to consider when faults happen is the update to the search direction $\mathbf{p}_t$. Here, when we observe a fault, we truncate the update $\mathbf{p}_t = \mathbf{r}_t + \beta_t \mathbf{p}_{t-1}$ to be $\mathbf{p}_t = \mathbf{r}_t$. This corresponds to a reset of the Krylov process and is described in the caption of Algorithm 1.

We now reiterate the recovery of the solution to the original system. Suppose the erasure-coded CG converges on the augmented system (1). Then, we simply compute the expression given by the recovery Equation (2) on that solution.

Partitioning considerations. Even with the sparse matrix $\boldsymbol{E}$ described in Section 4, the augmentation blocks in $\tilde{\boldsymbol{A}}$ potentially introduce dense blocks if not suitably computed. For this reason, we use a two-step process. In the first step, we order the input matrix (Figure 1(a)) using a traditional matrix/ graph partitioning technique such as Metis (Figure 1(b)). We then use this ordered matrix to compute $\tilde{\boldsymbol{A}}$ (Figure 1(c)). The resulting matrix is then reordered once again to partition $\tilde{\boldsymbol{A}}$ across various nodes in a parallel/ distributed platform (Figure 1(d)). The first step minimizes non-zeros in $\tilde{\boldsymbol{A}}$, and the second partitioning step minimizes communication for the solver applied to $\tilde{\boldsymbol{A}}$.

6 AN ADAPTIVE FAULT TOLERANT CONJUGATE GRADIENT SOLVER

The underlying principle of our adaptive solver is to add redundant blocks into the matrix only when errors are encountered. (Solvers from Section 5 operate on $n + k \times n + k$ matrices with up to k errors handled.) The solver always operates on $n \times n$ matrices, which are guaranteed to be SPD if the input matrix is SPD. Redundant blocks are precomputed and stored, but only utilized in the event of faults. We now describe our adaptive fault tolerant scheme built on top of the CG solver described earlier.

We assume that the original matrix $\boldsymbol{A}$ and right-hand side $\mathbf{b}$ are initially distributed among multiple processes by rows. This assumption does not restrict parallelization to 1D; rather, the assumption is merely for exposition. The solver runs on the input system until a fault occurs. As before, we assume a fail-stop failure model (other fault models can be reduced to fail-stop models

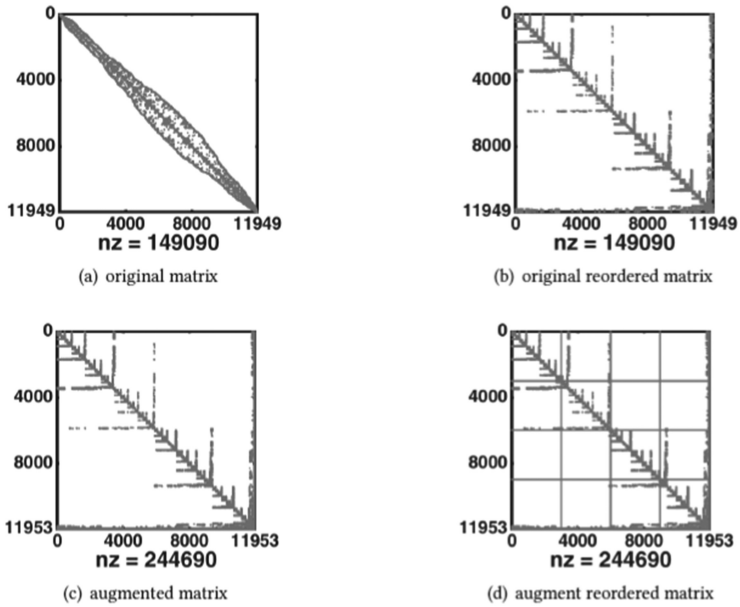


Fig. 1. Illustration of the process of computing and repartitioning augmented matrix $\tilde{A}$. From Kang et al. [16].

through predicate checks to detect failures). We also assume that other processors can detect the identity of the failed processor. This is typically implemented in systems through periodic heartbeats. When a fault happens, the rows (and columns) assigned to the processor are erased. These erased blocks are compensated for by the addition of an identical number of rows (and columns) selected from the precomputed coding blocks $[\boldsymbol{E}^T\boldsymbol{A}, \boldsymbol{E}^T\boldsymbol{A}\boldsymbol{E}]$ (Figure 2). We elaborate on this selection strategy in the next section. For now, we assume that the coding blocks are dense and that the selection of which rows/columns to add from the coding blocks is arbitrary.

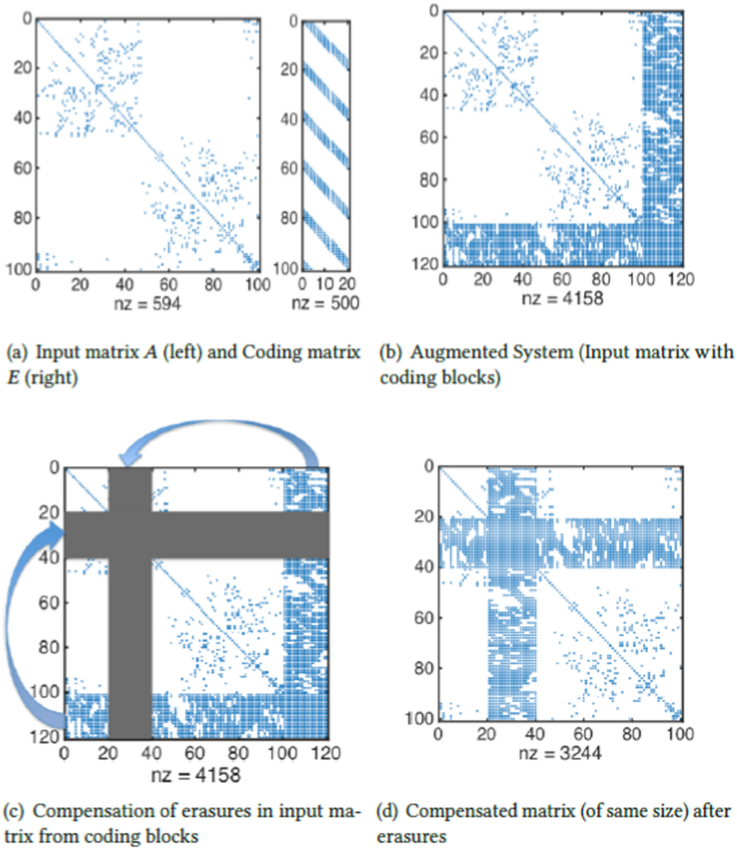
To describe the overall methodology, let $\boldsymbol{Z} = \boldsymbol{A}\boldsymbol{E}$ so that we can write the augmented matrix as

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{Z} \\ \boldsymbol{Z}^T & \boldsymbol{E}^T\boldsymbol{A}\boldsymbol{E} \end{bmatrix}.$$

We assume that the matrix $\boldsymbol{Z}$ and the vector $\boldsymbol{E}^T\mathbf{b}$ are available at all processors. In our adaptive solver, the matrix $\boldsymbol{Z}$ and vector are held in reserve and initially unused. Consequently, on the first fault, which we represent as erasures, we conceptually permute the matrix as follows to identify the correct and faulty entries in blocks:

$$\begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{12}^T & \boldsymbol{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_f \end{bmatrix} = \begin{bmatrix} \mathbf{b}_c \\ \mathbf{b}_f \end{bmatrix}.$$

Thus, when we lose elements to erasures, we lose the rows associated with ${\boldsymbol{A}_{12}}^T$ and $\boldsymbol{A}_{22}$ along with the elements $\mathbf{b}_f$. We also assume that other processors have cached the most recent values from $\mathbf{x}_f$ so that this information is not lost. The idea is that we are going to use information from $\boldsymbol{Z}$ and $\boldsymbol{E}^T\mathbf{b}$ to quickly add information back to the matrix. Consider, again conceptually, the full



(a) Input matrix A (left) and Coding matrix E (right)

(b) Augmented System (Input matrix with coding blocks)

(c) Compensation of erasures in input matrix from coding blocks

(d) Compensated matrix (of same size) after erasures

Fig. 2. Illustration of the coding and compensation process for erasures (fail-stop failures).

augmented system (1) at this point with the same partitioning:

$$\begin{bmatrix} A_{11} & A_{12} & Z_1 \\ {A_{12}}^T & A_{22} & Z_2 \\ {Z_1}^T & {Z_2}^T & E^TAE \end{bmatrix} \begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_f \\ \mathbf{x}_r \end{bmatrix} = \begin{bmatrix} \mathbf{b}_c \\ \mathbf{b}_f \\ E^T\mathbf{b} \end{bmatrix}. \tag{3}$$

Here,

$$\begin{aligned} Z_1 &= A_{11}E_1 + A_{12}E_2 \\ Z_2 &= A_{12}^TE_1 + A_{22}E_2 \end{aligned}$$

and

$$E = \begin{bmatrix} E_1 \\ E_2 \end{bmatrix}.$$

By assumption, the still-running processors have available the precomputed matrices Z_1, Z_2, E^TAE and vector $E^T\mathbf{b}$ involved in block partitioned form of the augmented system (1). The vector $\mathbf{x}_r$ has not been used at this point, so its values are zero. To be abundantly clear, we do not form the system in (3) explicitly. This is handled algorithmically. When an erasure occurs, $m < k$ rows of the system are lost. We select an arbitrary set of m columns from the precomputed coding data corresponding to an $n \times m$ matrix $\tilde{E}$. (In the event of a sequence of erasures, a column can only be selected once.) Let $\tilde{Z}_1, \tilde{Z}_2, \tilde{E}^TA\tilde{E}^T, \tilde{E}^T\mathbf{b}$ be the data selected. Then, we form and solve the new

ALGORITHM 2: Adaptive fault oblivious CG.

1: (Reliably) Compute and save the entries $Z_1, Z_2, E^T AE, E^T \mathbf{b}$ for an $n \times k$ coding matrix E.
2: Let $A^{(\text{cur})} = A$ and $\mathbf{b}^{(\text{cur})} = \mathbf{b}$
3: Let $\mathbf{x}_0$ be the initial guess, $\mathbf{r}_0 = \mathbf{b}^{(\text{cur})} - A^{(\text{cur})}\mathbf{x}_0$, and $\beta_0 = 0$.
4: **for** $t = 1, \ldots$ until convergence **do**
5: $\mathbf{p}_t = \begin{cases} \mathbf{r}_{t-1} & t = 1 \text{ or after a fault} \\ \mathbf{r}_{t-1} + \frac{\mathbf{r}_{t-1}, \mathbf{r}_{t-1}}{\mathbf{r}_{t-2}, \mathbf{r}_{t-2}} \mathbf{p}_{t-1} & t > 1 \end{cases}$
6: $\mathbf{q}_t = A^{(\text{cur})}\mathbf{p}_t$
7: $\alpha_t = (\mathbf{r}_{t-1}, \mathbf{r}_{t-1})/(\mathbf{q}_t, \mathbf{p}_t)$
8: $\mathbf{x}_t = \mathbf{x}_{t-1} + \alpha_t \mathbf{p}_t$
9: $\mathbf{r}_t = \mathbf{r}_{t-1} - \alpha_t \mathbf{q}_t$
10: When there are m faults detected, create a new system with m unused columns of coding data. Let $\tilde{Z}_1, \tilde{Z}_2, \tilde{E}^T A\tilde{E}, \tilde{E}^T \mathbf{b}$ correspond to the columns used.

$$A^{(\text{cur})} \leftarrow \begin{bmatrix} A_{1,1} & \tilde{Z}_1 \\ \tilde{Z}_1^T & \tilde{E}^T A\tilde{E} \end{bmatrix}$$

$$\mathbf{b}^{(\text{cur})} \leftarrow \begin{bmatrix} \mathbf{b}_c - A_{1,2}\mathbf{x}_f \\ \tilde{E}^T \mathbf{b} - \tilde{Z}_2^T \mathbf{x}_f \end{bmatrix}$$

$$\mathbf{x}_t = \begin{bmatrix} \mathbf{x}_c \\ 0 \end{bmatrix}$$

$$\mathbf{r}_t = \mathbf{b}^{(\text{cur})} - A^{(\text{cur})}\mathbf{x}_t$$

11: **end for**

system:

$$\begin{bmatrix} A_{11} & \tilde{Z}_1 \\ \tilde{Z}_1^T & \tilde{E}^T A\tilde{E} \end{bmatrix} \begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_r \end{bmatrix} = \begin{bmatrix} \mathbf{b}_c \\ \tilde{E}^T \mathbf{b} \end{bmatrix} - \begin{bmatrix} A_{12} \\ \tilde{Z}_2^T \end{bmatrix} \mathbf{x}_f. \tag{4}$$

Note that all of these matrix blocks are available to us. Given any solution of this problem, we can recreate the true solution to the original system following the reconstruction procedure from Zhu et al. [26].

The corresponding procedure is described in Algorithm 2 in terms of explicit algorithmic primitives. Steps 3 through 9 correspond to the standard CG method. Step 10 corresponds to the adaptive fault tolerance mechanism. After erasures, we need to update the corresponding right-hand side to account for the fault. To compute $\mathbf{b}^{(\text{cur})}$, we use the saved vector $\mathbf{x}_f$. We also set the initial value of $\mathbf{x}_r = 0$. Then, we reset the Krylov subspace process for this new system to ensure that the computed recurrences represent the changes to the system.

How this differs from reforming the original system. One subtle aspect of this idea is that the precomputed coding data allows us to easily look up data that will render the system non-singular and equivalent for *any* possible erasure. In contrast, without this information, we would have to recreate precisely those elements of the matrix A that were erased. Given that forming linear systems can itself be a complicated process, where we are unlikely to maintain random access to all the data in future, this represents a distinct advantage to our approach. Even if such a random access structure were to be available, large-scale solvers typically run at the limit of memory, and storing multiple copies of the matrix (as many copies as faults to be tolerated) would typically not be feasible.

Coding matrix. We use the structured sparse coding matrix described in Section 4. The coding matrix is illustrated in Figure 2(a). Furthermore, owing to its structure and sparsity, it does not induce dense blocks into the augmented matrix. Please note that the relative dimensions of matrices $\boldsymbol{A}$ and $\boldsymbol{E}$ in Figure 2 are selected for illustration purposes. The choices of values of k and s relative to the size of the matrix n make the augmentation blocks appear dense. In real experiments, the augmentation blocks constitute a small fraction of the total matrix, and are much sparser. In Figure 2(c), we show how parts of the augmentation blocks are swapped in, to compensate for erasures. Finally, Figure 2(d) illustrates the compensated matrix.

Using a sparse coding matrix has important implications for the adaptive fault tolerant solver. Recall that coding rows (or blocks) are added only as faults are detected. If the coding blocks are dense and Kruskal rank-k, we can select arbitrary columns from matrix $\boldsymbol{AE}$, corresponding to columns of the $\boldsymbol{Z}$ matrices from the last section, to compensate for erasures (see detailed discussion in the work of Zhu et al. [26]). However, if the coding blocks are sparse, we cannot arbitrarily select columns from the coding blocks, because column j of $\boldsymbol{E}$ may not involve row or column i from the matrix $\boldsymbol{A}$. Consequently, we need to ensure that if an element x_i is erased, then the column j we select from $\boldsymbol{AE}$ must have a non-zero entry $E_{i,j}$. This is easily done, since by assumption, all processors are aware of the indices of erased elements and the elements of $\boldsymbol{E}$ are structured.

We reiterate that we do not add the erased row of matrix A itself because we would have to maintain multiple copies of the matrix (one more than the number of node failures we wish to tolerate) to be able to replace erased rows with original rows in the matrix. In contrast, using coded rows, we can significantly reduce the storage requirement for coded blocks. Note that this reduction in required memory is identical to that of storing erasure coded data in storage systems, as compared to replication.

Communication overhead of coded rows. An important question on parallel performance of our method relates to the communication overhead and computational cost introduced by the coded rows. The volume of communication and consequently the communication overhead are determined by the structure and density of the coding block. Rows in the coding block are constructed as scaled summations of sparse subsets of columns of the input matrix. For this reason, the coding block structure, fill in coded row, and resulting communication overheads are highly dependent on the matrix structure. We note, however, that for minimizing communication and maximizing cache performance, sparse matrices are ordered in such a way that rows with similar non-zero structure are ordered to be in proximity of each other. Since our coding matrix is constructed as scaled sum of selected contiguous rows, the fill structure of the coding block is similar to that of other rows in the partition (other rows assigned to the processor). What this implies is that the coded rows typically do not increase the number of processors that need to exchange data over the base matrix-vector product. The increase in volume of communication is minimal, as we demonstrate in our experiments, resulting in minimal loss in efficiency, compared to the solver on the original uncoded matrix.

Comparison with the static coding scheme. The base erasure coded linear solver of Kang et al. [16] estimates worst-case execution faults and augments the input matrix to account for this worst case. Although this static coding scheme outperforms other fault tolerance mechanisms, it has two performance drawbacks:

- The added computational/communication overhead of the static coding block that must be paid even when there are fewer faults.
- The slower convergence rate from the null space that is added to the system.

Table 1. Matrices Used in Testing

Matrix	Rows	Non-Zeros
cbuckle	13,681	676,515
gyro_m	17,361	340,431
consph	83,334	6,010,480
ldoor	952,203	42,493,817

These performance characteristics are observed in the experimental results in the work of Kang et al. [16]. In contrast, experimental results from our adaptive coding scheme clearly show that the convergence of our solver closely tracks the base (uncoded matrix) case (i.e., there is no loss in convergence, since there is no null space), and that the parallel performance is indistinguishable from the base (uncoded matrix) solver.

7 EXPERIMENTAL VALIDATION

We present a comprehensive experimental validation of our proposed adaptive fault tolerance scheme. We aim to demonstrate the following key aspect of our scheme: (i) we characterize the fault tolerance of the CG solver, demonstrating that the solver converges to solution at low relative residual; (ii) the iteration and time overhead of our fault tolerant solver is low, compared to state of the art techniques; and (iii) the parallel overhead induced by our augmentation rows is small—leading to highly efficient and scalable performance.

We select matrices from the University of Florida Matrix Collection for our tests, in which `cbuckle` and `gyro_m` are used to validate the convergence of adaptive fault tolerant linear solver; larger matrices `consph` and `ldoor` are used to validate parallel scalability and robustness to different fault arrival models. The matrix sizes and sparsity are shown in Table 1.

All of the test matrices are SPD (i.e., both CG on the original and augmented system converge on these matrices). The right-hand side $\mathbf{b}$ is initialized as $\mathbf{b} = A\mathbf{e}$, where $\mathbf{e}$ is the column vector with all 1s and normalized (i.e., $||\mathbf{b}||_2 = 1$). Therefore, the relative residual $\frac{||A\mathbf{x}-\mathbf{b}||_2}{||\mathbf{b}||_2}$ is equal to the residual $||\mathbf{r}||_2 = ||A\mathbf{x} - \mathbf{b}||_2$ for our problems. During execution, we compute the residual at each iteration and set the termination condition as $||\mathbf{r}||_2 < 10^{-6}$ for all matrices. The maximum number of iterations of CG is set to 10,000. Of our test matrices, only `ldoor` does not achieve a residual of 10^{-6} before reaching the iteration bound, for either the fault-free or faulty cases.

For parallel performance, the matrices are first reordered using Metis [17]. To construct the augmented system, we use an encoding matrix $\boldsymbol{E}$ as described in the previous section and use this matrix to generate the augmented system.

In our tests, we use two different fault arrival models: faults arriving instantaneously and faults arriving at different points in time according to an exponential distribution.

The motivation for the instantaneous fault model derives from coordinated failures. For instance, there have been studies that show that the components with three highest failures rates in data centers are disk, memory, and power supply [23]. In each of these cases, one or more sockets (i.e., the processes allocated to those cores) may fail instantaneously. This is modeled by our instantaneous fault model, where at a selected time, a specific number of processes (i.e., those on a single socket or a blade) fail.

An exponential distribution is a commonly used fault arrival model [3]. The probability distribution function (PDF) of the time to failure is given by

$$P_e(t < \tau) = 1 - e^{-\lambda_e \tau}. \tag{5}$$

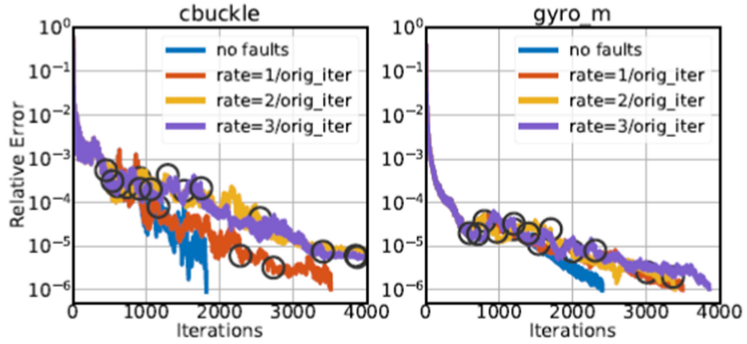


Fig. 3. Convergence rate for cbuckle and gyro_m with different fault rates and fault starting points. The black circles indicate instances where faults are encountered in execution.

Here, λ_e is the failure rate. We define orig_iter as the number of iterations the original system needs to converge to a residual norm of less than 10^{-6} or to reach the iteration bound. Different fault rates (λ_e) ranging from $\frac{1}{\texttt{orig\_iter}}$ to $\frac{3}{\texttt{orig\_iter}}$ are tested. Note that for an exponential distribution, the mean number of iterations between two consecutive faults is $\frac{1}{\lambda_e}$. This means the average number of faults in orig_iter iterations is 1, 2, or 3 for $\lambda_e = \frac{1}{\texttt{orig\_iter}}, \frac{2}{\texttt{orig\_iter}}$ and $\frac{3}{\texttt{orig\_iter}}$, respectively. However, since the number of iterations to convergence may be increased by the addition of coding rows, the actual number of faults (even on average) may be higher. In our tests, we set the first fault to happen at $\frac{\texttt{orig\_iter}}{1+\texttt{orig\_iter}/\lambda_e}$. This is to ensure that the fault process starts; otherwise, in some runs, there are no faults at all for the exponential fault process. Finally, in our experiments, for the exponential fault model, we report the number of faults K to be the mean of this distribution.

7.1 Convergence Rates

Our first set of experiments focuses on the convergence rate of the adaptive linear system solver in the presence of faults. We plot convergence rates and compare them with the no-fault case. The relative error is calculated with respect to the original system.

Figure 3 shows the convergence rate of the adaptive fault tolerant linear solver with different fault rates. For test matrices cbuckle and gyro_m, we observe that our solver can reach the same relative error as the original system (without any faults) while tolerating a number of faults. As the fault rate increases, the solver needs more iterations to converge. However, we note that the overhead in terms of increased iterations is significantly lower than comparable fault tolerance techniques based on replicated execution or deterministic replay for the same number of faults.

Figure 4 shows the convergence of the solver on larger matrices, consph and ldoor, for different fault rates. We observe that our solver achieves good convergence rates for both matrices with different fault rates. Furthermore, we note that the convergence for the faulty and no-fault cases track very closely with each other, indicating that use of the augmented system does not impact solver convergence adversely.

7.2 Parallel Performance

We now demonstrate the parallel performance and the time overhead of our augmented system solver. We benchmark the parallel performance of our solver on a 192-core (8 sockets) Intel Xeon Platinum 8168 processor operating at 2.70 GHz. We use MPI to implement our solver. We simulate faults in the system by inducing a selected number of erasures. Processors communicate via

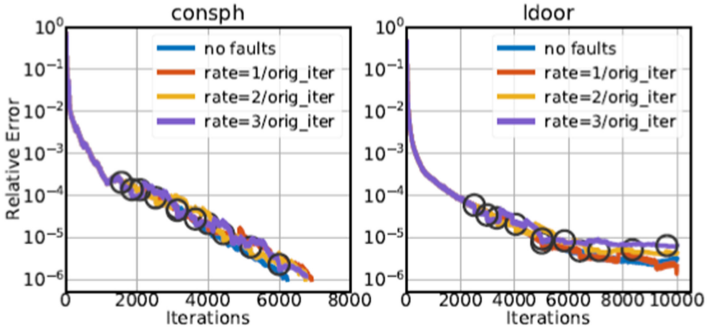


Fig. 4. Convergence rate for consph and ldoor with different fault rates and fault starting points.

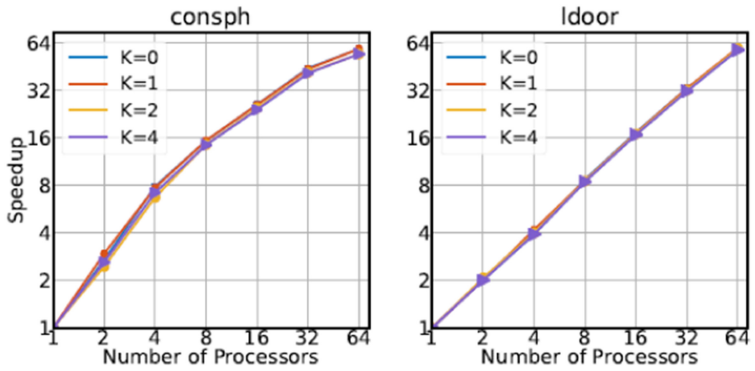


Fig. 5. Parallel performance of consph and ldoor for the exponential fault model.

non-blocking communications, and processors where faults have been induced stop communicating from the time of induced fault. In our experiments, we use two fault models: the exponential fault model and the instantaneous fault model. Upon detecting a fault, the solver updates the erased rows and continues with the solve, as described in our algorithm. We report on the parallel performance for different sizes of augmenting blocks (varying number of faults, $K = 0, 1, 2, 4$; here, $K = 0$ corresponds to the original system without faults during execution). The parallel speedup is defined as the ratio of the time taken by one processor (to convergence or to reach iteration bound) to the corresponding time taken by the parallel execution. Since we aim to quantify the parallel overhead of the coding blocks, both serial and parallel executions are assumed to have the same number of faults.

Exponential fault model. For the exponential fault model, we use the fault arrival rate of $\frac{2}{\texttt{orig\_iter}}$ for all matrices. Figure 5 shows the speedup for large matrices with different number of faults. With increasing number of processors, the speedup increases nearly linearly for all augmentation sizes. Note that the speedup starts to saturate for matrix consph as the number of processors increases. However, this saturation also happens for $K = 0$, indicating that our augmentation blocks add negligible parallel overhead (over and above the base solver).

Instantaneous fault model. For the instantaneous fault model, all faults happen at the same time—we select this to be $\frac{\texttt{orig\_iter}}{3}$ for all matrices. This point is the same as the occurrence of the first fault in exponential fault model with a fault rate equal to $\frac{2}{\texttt{orig\_iter}}$.

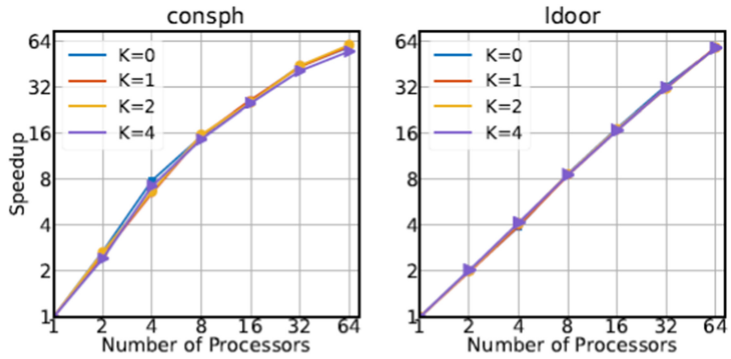


Fig. 6. Parallel performance for matrices consph and ldoor for the instantaneous fault model.

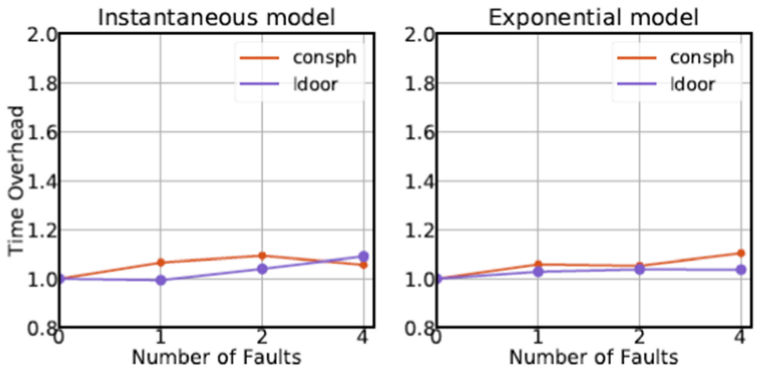


Fig. 7. Time overhead with different numbers of faults under different fault arrival models.

Figure 6 shows the speedup for large matrices with different number of faults under the instantaneous fault model. We observe near linear speedup for our solver—and most importantly, the speedup is very close to the base solver with no faults—indicating that under this fault model as well, we do not introduce any significant parallel overhead..

7.3 Time Overhead Using Coding Blocks

We analyze the time overhead of the augmented system solver with respect to original system. We let a fixed number of faults happen during execution ($K = 1, 2, 4$) and calculate the ratio of the time to solution for the augmented system with faults and the original system with no faults. As before, time to solution either corresponds to the time to convergence or to reach the iteration bound. Since we are interested in quantifying the compute overhead of our coding block, all results here are obtained on a single core. Ideally, we want this ratio to be as close to 1 as possible.

Figure 7 shows the time overhead of our solver for different numbers of faults. As expected, with increasing number of faults, we need more time to converge. This time overhead comes from two factors: (i) the impact of denser coding rows in the augmentation blocks and (ii) the increased number of iterations to convergence. We observe for our test matrices that the time overhead for each case tested is less than a factor of 1.2. This is highly efficient, particularly when the number of faults increases, especially in comparison to competing replicated execution or deterministic replay schemes.

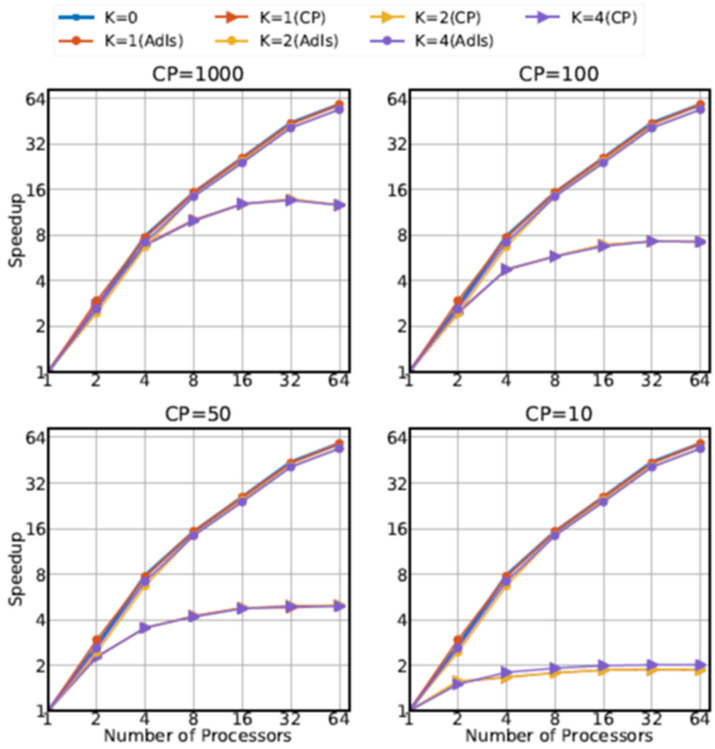


Fig. 8. Comparison of AECC with checkpointing for different fault rates on matrix consph. Each figure shows a different checkpoint interval—indicated in the title. We see the best speedup for checkpointing with infrequent checkpointing, but this still lags our method. None of the methods greatly differ in the number of faults.

7.4 Comparison with Checkpointing

We compare AECC with traditional checkpointing [12, 24]. Checkpointing periodically saves the intermediate program state and rolls back to the last consistent checkpoint when it detects a fault. We implement an optimized form of application checkpointing, where we only store data structures that are needed for restarting the iterative solver, as opposed to all ancillary data structures that may have been updated. Specifically, we save vectors $\mathbf{x}$, $\mathbf{p}$, α, and β into persistent storage (temporary files) periodically. We collect this data at a designated storage node using a gather operation and save it to disk. Checkpointing intervals are important, because infrequent checkpoints result in significant rollback, whereas frequent checkpoints in the presence of infrequent faults results in significant checkpointing overhead [9, 13, 25]. For this reason, we experiment with different checkpointing intervals including every 1,000, 100, 50, or 10 iterations of the solver. When a fault is detected, the solver reads from the file containing the most recent intermediate results and continues.

7.4.1 Comparison of Speedup. Figures 8 and 9 show speedup comparison of AECC with checkpointing for different checkpoint intervals (CP) and fault rates. With increasing number of processors, AECC achieves nearly linear speedup, whereas the speedup of checkpointing saturates. This is particularly true for smaller checkpoint intervals, where the overhead of checkpointing becomes a dominant bottleneck.

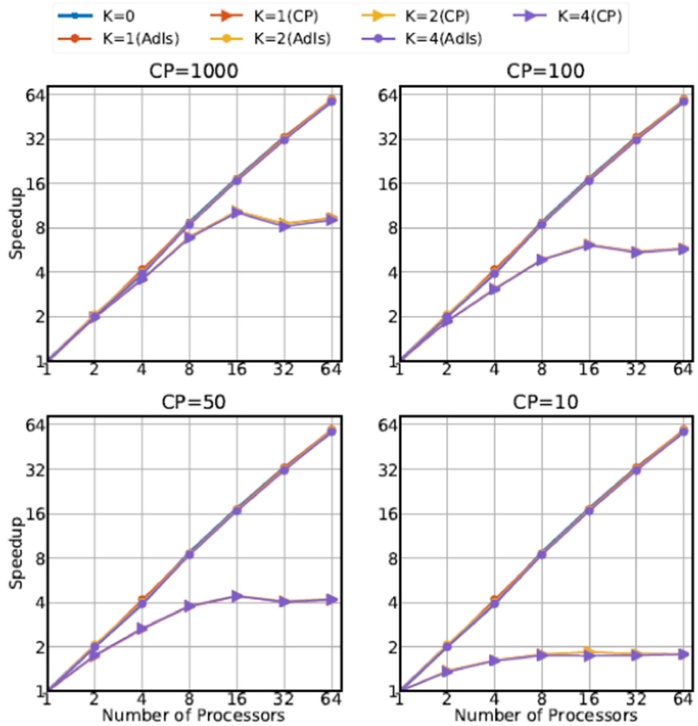


Fig. 9. Comparison of AECC with checkpointing for different fault rates on matrix `ldoor`. Each figure shows a different checkpoint interval—indicated in the title. We see the best speedup for checkpointing with infrequent checkpointing, but this still lags our method. None of the methods greatly differ in the number of faults.

7.4.2 Comparison of Time Overhead. We compare the time overhead of AECC and checkpointing in Figure 10. Compared to AECC, whose time overheads primarily come from increased computation on dense augmentation blocks and larger number of iterations to converge, the overheads of checkpointing primarily come from storing and retrieving intermediate results to/ from persistent storage. When the checkpoint interval, CP, is small, checkpointing incurs significant storage overhead but does not lose much in wasted computation, since it does not roll back too far. Conversely, when checkpoint interval is large, the storage overhead is low, but the roll back overhead is high. The optimal checkpoint interval is therefore a function of the storage cost (which in turn depends on the size of program state, as well as performance of the underlying I/O subsystem), fault rate, and the cost of finding a consistent checkpoint (the latter is not a significant problem in our case, since the iterative synchronized nature of our solver makes it easy to find consistent checkpoints).

Figure 10 shows a comparison of time overheads of AECC and checkpointing. The figure clearly demonstrates that the overhead of AECC is significantly lower than checkpointing across a range of fault models and checkpoint intervals. This is particularly true for the larger benchmark matrix, `ldoor`.

8 CONCLUSION

In this article, we present an adaptive fault tolerant linear system solver capable of scaling to large numbers of processors and associated faults. The solver works by augmenting the input

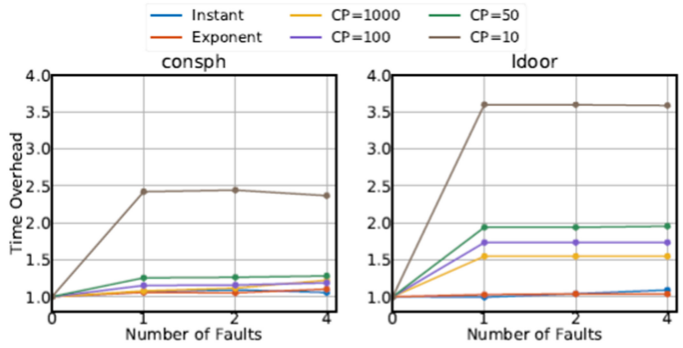


Fig. 10. Comparison of time overhead of AECC and checkpointing for matrices consph and ldoor demonstrating significantly lower overhead of AECC. The AECC results are the same as Figure 7.

matrix using erasure coded blocks as faults are detected, instead of augmenting the input matrix *a priori*. Our technique has the following significant advantages: (i) coding blocks are only used when faults are detected—at any time, the linear system is always identical in size to the input system; (ii) the convergence properties of the augmented system closely follow those of the original (input) matrix; and (iii) the parallel performance of the solver scales well with increasing numbers of processors. We investigate the effect of fault rates and fault arrival patterns (instantaneous versus exponential) and show that our scheme is robust to a wide range of fault characteristics and significantly outperforms traditional fault tolerance mechanisms.

In terms of avenues for future research, the proof of concept presented in this work strongly establishes adaptive erasure coded fault tolerance as a powerful new technique, particularly at scale. An important question that arises in this context is the design of the coding matrix. Although a structured sparse matrix is used in our experiments, an open question relates to the best coding matrix structure that optimally trades off fill in augmentation block, communication in parallel execution, and conditioning of augmented matrix.

REFERENCES

[1] George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. 2009. Algorithm-based fault tolerance applied to high performance computing. *J. Parallel Distrib. Comput.* 69, 4 (April 2009), 410–416.

[2] Patrick G. Bridges, Kurt B. Ferreira, Michael A. Heroux, and Mark Hoemmen. 2012. Fault-tolerant linear solvers via selective reliability. *CoRR* abs/1206.1390 (2012).

[3] Xavier Castillo, Stephen McConnel, and Daniel Siewiorek. 1982. Derivation and calibration of a transient error reliability model. *IEEE Trans. Comput.* C-31 (1982), 658–671.

[4] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. 2015. Deterministic replay: A survey. *ACM Comput. Surv.* 48, 2 (Sept. 2015), Article 17, 47 pages.

[5] Zizhong Chen. 2009. Optimal real number codes for fault tolerant matrix operations. In *Proceedings of the ACM/IEEE Conference on High Performance Computing*.

[6] Zizhong Chen. 2011. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing*. 73–84.

[7] Zizhong Chen and Jack Dongarra. 2005. Numerically stable real number codes based on random matrices. In *Proceedings of the International Conference on Computational Science (ICCS'05)*. 115–122.

[8] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. 2005. Fault tolerant high performance computing by a coding approach. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*. 213–223.

[9] J. T. Daly. 2006. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.* 22, 3 (Feb. 2006), 303–312.

[10] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation—Volume 6*. 10.

[11] Jack Dongarra and Zizhong Chen. 2008. Algorithm-based fault tolerance for fail-stop failures. *IEEE Trans. Parallel Distrib. Syst.* 19, 12 (2008), 1628–1641.
[12] Ifeanyi P. Egwutuoha, David Levy, Bran Selic, and Shiping Chen. 2013. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *J. Supercomput.* 65, 3 (Sept. 2013), 1302–1326.
[13] Nosayba El-Sayed and Bianca Schroeder. 2016. Understanding practical tradeoffs in HPC checkpoint-scheduling policies. *IEEE Trans. Dependable Secure Comput.* PP (March 2016), 1. https://doi.org/10.1109/TDSC.2016.2548463
[14] Gaston H. Gonnet. 1981. Expected length of the longest probe sequence in hash code searching. *J. ACM* 28, 2 (April 1981), 289–304. https://doi.org/10.1145/322248.322254
[15] Kuang-Hua Huang and J. A. Abraham. 1984. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.* 33, 6 (June 1984), 518–528.
[16] Xuejiao Kang, David F. Gleich, Ahmed Sameh, and Ananth Grama. 2017. Distributed fault tolerant linear system solvers based on erasure coding. In *Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS'17)*. 2478–2485.
[17] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.* 20, 1 (Dec. 1998), 359–392.
[18] Richard Koo and Sam Toueg. 1986. Checkpointing and rollback-recovery for distributed systems. In *Proceedings of 1986 ACM Fall Joint Computer Conference (ACM'86)*. IEEE, Los Alamitos, CA, 1150–1158.
[19] Gerard Meurant. 2006. *The Lanczos and Conjugate Gradient Algorithms: From Theory to Finite Precision Computations (Software, Environments, and Tools)*. SIAM.
[20] J. S. Plank. 2013. Erasure codes for storage systems: A brief primer. *Login* 38, 6 (Dec. 2013).
[21] Martin Raab and Angelika Steger. 1998. "Balls into bins"—A simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science (RANDOM)*. Lecture Notes in Computer Science, Vol. 1518. Springer, 159–170. https://doi.org/10.1007/3-540-49543-6_13
[22] Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319.
[23] Guosai Wang, Lifei Zhang, and Wei Xu. 2017. What can we learn from four years of data center hardware failures? In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'17)*. IEEE, Los Alamitos, CA, 25–36. https://doi.org/10.1109/DSN.2017.26
[24] Tang Xiongchao, Jidong Zhai, Bowen Yu, Wenguang Chen, Weiming Zheng, and Keqin Li. 2017. An efficient in-memory checkpoint method and its practice on fault-tolerant HPL. *IEEE Trans. Parallel. Distrib. Systems* PP (Dec. 2017), 1. https://doi.org/10.1109/TPDS.2017.2781257
[25] John W. Young. 1974. A first order approximation to the optimum checkpoint interval. *Commun. ACM* 17, 9 (Sept. 1974), 530–531. https://doi.org/10.1145/361147.361115
[26] Yao Zhu, Ananth Grama, and David F. Gleich. 2017. Erasure coding for fault oblivious linear system solvers. *SIAM J. Sci. Comput.* 39, 1 (2017), C48–C64.

Received February 2021; revised August 2021; accepted September 2021