FAULT TOLERANCE IN LINEAR ALGEBRAIC METHODS USING ERASURE CODED COMPUTATIONS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Xuejiao Kang

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2018

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Ananth Grama, Chair

Department of Computer Science, Purdue University–West Lafayette

Dr. Ahmed Sameh

Department of Computer Science, Purdue University–West Lafayette

Dr. David F. Gleich

Department of Computer Science, Purdue University–West Lafayette

Dr. Alex Pothen

Department of Computer Science, Purdue University–West Lafayette

Dr. Hemanta K. Maji

Department of Computer Science, Purdue University–West Lafayette

Approved by:

Dr. Voicu Popescu

Head of the Department Graduate Program

To my husband and son,

Xin Cheng, Jason Cheng

ACKNOWLEDGMENTS

Working with Prof. Grama, who is knowledgeable, helpful and patient, has been a memorable experience for me. The knowledge I gained from him went beyond my research to other aspects of life as well. He was always patient with me, listened to my concerns, was ready to help, and always gave me confidence in my ability. He welcomed new ideas and was open to approaches different from his own. His attitude to research, work, and life impressed me, and gave me a broader perspective, which will benefit me in my future.

I would like to thank my committee members: Prof. Ahemed Sameh, Prof. David F. Gleich, Prof. Alex Pothen and Prof. Hemanta K. Maji. Prof. Sameh guided me when I was confused in my research and brought me into the field of linear algebra, which I have always been interested in. Discussions with him always enhanced my knowledge and motivated me to learn more. Collaboration with Prof. Gleich was such an enjoyable experience for me. He always showed me different views of the problems and inspired me with new ideas. Prof. Pothen helped me a lot during my Ph.D. studies, especially during the time when I was unclear about my future. He gave me a number of helpful suggestions about my future and possible solutions to my research problems. Prof. Maji was patient when listening to my problems and was always happy to discuss different ways to solve my problems. He showed me new directions to approach my problems, and the beauty and power of mathematics.

I would also like to say thanks to my friends and colleagues. Chi-Hao Fang, Vikram Ravindra, Shahin Mohammadi, Sudhir Kylasa provided useful discussion and comments on my research. Yu-Hong Yeung, Zhengyi Zhang and Yongyang Yu spent their time discussing and sharing their thoughts and comments with me. Yu-hong and Kristen M. Johnson also gave me many useful comments and suggestions on my final dissertation.

I would like to thank Prof. Daisuke Kihara whose spirit and passion for research impressed and motivated me. He showed me how to start and finish a research project and always welcomed discussions and different ideas. My colleagues in his lab were always friendly and helpful.

I would also like to say thanks to all the staff in the CS department. Monica Shively, Pam Graf, Tammy Muthig, Charles Fultz, Ron Castongia, Kathy Moore, Steve Plite, Kelsey Vance, Shelley Straley and William J. Gorman provided so much help during my Ph.D. studies and the kind atmosphere in CS department made my time there enjoyable.

The journey towards my Ph.D. has been interesting and remarkable. Although I encountered many problems, luckily, I have my husband – Xin, who is always beside me regardless of my mood. He gave me confidence when I nearly gave up and took care of me in my daily life. We also have our angel–Jason, entering our lives during my Ph.D. studies. Jason is such a good boy, who changes my perspectives on many things, and gives me courage to face new challenges. He helps me overcome difficulties and develops my patience. Xin and Jason give me a peaceful and happy life, which helps me focus on my studies. I am thankful for the happiness and support they give me every day.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Kang, Xuejiao. Ph.D., Purdue University, December 2018. Fault Tolerance in Linear Algebraic Methods using Erasure Coded Computations . Major Professor: Ananth Grama.

As parallel and distributed systems scale to hundreds of thousands of cores and beyond, fault tolerance becomes increasingly important – particularly on systems with limited I/O capacity and bandwidth. Error correcting codes (ECCs) are used in communication systems where errors arise when bits are corrupted silently in a message. Error correcting codes can detect and correct erroneous bits. Erasure codes, an instance of error correcting codes that deal with data erasures, are widely used in storage systems. An erasure code adds redundancy to the data to tolerate erasures.

In this thesis, erasure coded computations are proposed as a novel approach to dealing with processor faults in parallel and distributed systems. We first give a brief review of traditional fault tolerance methods, error correcting codes, and erasure coded storage in Chapter 1. The benefits and challenges of erasure coded computations with respect to coding scheme, fault models and system support are also presented.

In the first part of my thesis, I demonstrate the novel concept of erasure coded computations for linear system solvers. Erasure coding augments a given problem instance with redundant data. This augmented problem is executed in a fault oblivious manner in a faulty parallel environment. In the event of faults, we show how we can compute the true solution from potentially fault-prone solutions using a computationally inexpensive procedure. The results on diverse linear systems show that our technique has several important advantages: (i) as the hardware platform scales in size and in number of faults, our scheme yields increasing improvement in resource

utilization, compared to traditional schemes; (ii) the proposed scheme is easy to code as the core algorithm remains the same; (iii) the general scheme is flexible to accommodate a range of computation and communication trade-offs.

We propose a new coding scheme for augmenting the input matrix that satisfies the recovery equations of erasure coding with high probability in the event of random failures. This coding scheme also minimizes fill (non-zero elements introduced by the coding block), while being amenable to efficient partitioning across processing nodes. Our experimental results show that the scheme adds minimal overhead for fault tolerance, yields excellent parallel efficiency and scalability, and is robust to different fault arrival models and fault rates.

Building on these results, 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 during execution, we only solve a system with the same size as 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 the observed faults. We present, in details, the augmentation process, the parallel formulation, and the performance of our method. 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.

Based on the promising results for linear system solvers, we apply the concept of erasure coded computation to eigenvalue problems, which arise in many applications including machine learning and scientific simulations. Erasure coded computation is used to design a fault tolerant eigenvalue solver. The original eigenvalue problem is reformulated into a generalized eigenvalue problem defined on appropriate augmented matrices. We present the augmentation scheme, the necessary conditions for augmentation blocks, and the proofs of equivalence of the original eigenvalue problem and the

reformulated generalized eigenvalue problem. Finally, we show how the eigenvalues can be derived from the augmented system in the event of faults.

We present detailed experiments, which demonstrate the excellent convergence properties of our fault tolerant TraceMin eigensolver in the average case. In the worst case where the row-column pairs that have the most impact on eigenvalues are erased, we present a novel scheme that computes the augmentation blocks as the computation proceeds, using the estimates of leverage scores of row-column pairs as they are computed by the iterative process. We demonstrate low overhead, excellent scalability in terms of the number of faults, and the robustness to different fault arrival models and fault rates for our method.

In summary, this thesis presents a novel approach to fault tolerance based on erasure coded computations, demonstrates it in the context of important linear algebra kernels, and validates its performance on a diverse set of problems on scalable parallel computing platforms. As parallel systems scale to hundreds of thousands of processing cores and beyond, these techniques present the most scalable fault tolerant mechanisms currently available.

1 BACKGROUND AND INTRODUCTION

In this chapter, we provide a brief review of the state of the art in fault tolerance techniques and three related, yet critically distinct concepts – error correcting codes, erasure coded storage, and our proposed erasure coded computations. Then, we will introduce fault tolerant linear algebraic methods based on erasure coded computations. We conclude by summarizing the organization of this dissertation.

1.1 Background on Fault Tolerance Techniques

Techniques for fault tolerance can be classified into two broad categories – system-supported methods and algorithm-based methods. System-supported methods include checkpoint-restart, active replicas, and deterministic replay [1–4]. Checkpoint-restart techniques periodically save application state into persistent storage (either disks or in-memory). This requires identification of consistent checkpoints, along with capacity for persistent storage in terms of space and bandwidth (to store checkpoints). Scalable systems with hundreds of thousands of cores and beyond are often constrained in all of these resources. Without significant rollbacks, it is hard to find the restart point from the consistent checkpoints, particularly in scalable parallel programs that attempt to minimize global synchronizations.

Active replicas execute computations at multiple sites – these replicas are monitored for potential faults, and a consensus protocol identifies fault-free executions. In general, a computation must be executed on $k+1$ replicas to tolerate k faults. As k increases, replicated execution involves significant resource overhead. The most common instantiation of deterministic replay is in MapReduce environments [5]. Maps are scheduled at different execution units and are monitored for successful completion. If a map task times out, it is rescheduled at an alternate execution unit. The

presence of intermediate reduce points ensures that maps have to be replayed only back to the previous reduce point. Such techniques depend 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 may itself be significant.

Algorithm-based fault tolerance (ABFT) methods modify the base algorithm to embed redundant computations, so as to render the overall computation resilient to faults [6–13]. ABFT methods must typically be supplemented by correctness proofs, limits on fault tolerance, and bounds on computational overhead. The fault tolerant linear solver proposed by Chen et al. [10] views the faulty solver as a preconditioner for an outer solver, which validates the solution of the potentially faulty inner solver. The eigenvalue solver of Chen [11] relies on checksum coding along with fault location and reconfiguration to detect an erroneous signal. Different error detection methods that utilize properties of eigenvalue systems have been proposed. The checksum rows and columns added to detect soft errors are used in conjunction with a checkpoint procedure for rolling back state to the last correct state. In contrast, our method does not require checkpoint and restart procedure. By carefully reformulating the problem and suitably coding the input matrix, our method runs in a completely fault oblivious manner. A checksum method for QR decomposition (which can be used to solve eigenvalues) was proposed by Luk and Park [13]. Although this method does not need to roll back computation to correct errors, it can only detect and correct one transient error. ABFT techniques often have advantages over system-supported methods in terms of resource overheads. However, they must be specifically designed for each algorithm, leverage specific aspects of the algorithm as well as fault characteristics of underlying platforms, and typically require intricate correctness proofs.

1.2 Error Correcting Codes, Erasure Coded Storage and Computations

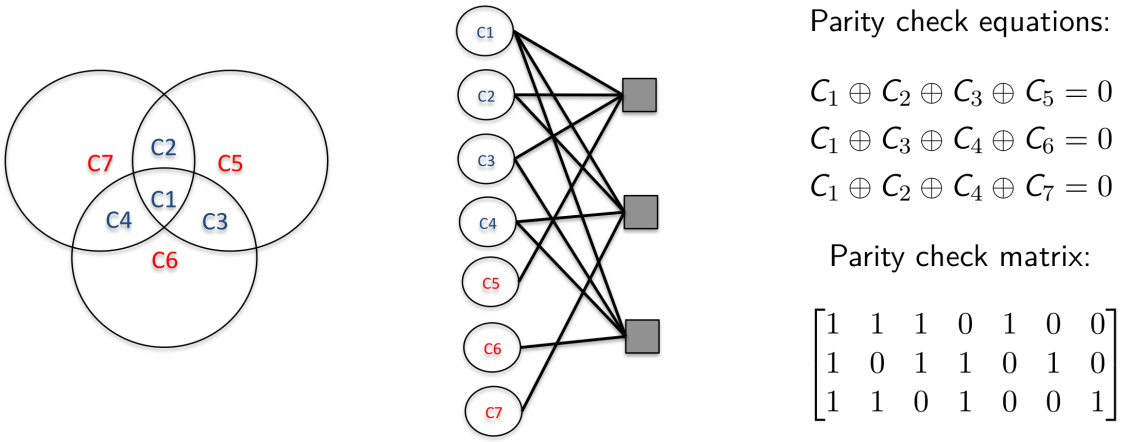


Figure 1.1.: Example of Hamming Code.

Error correcting codes (ECCs) are widely used in communication and storage systems, where noise may exist [14]. Hamming Codes [15] and low density parity codes (LDPC) [16] are commonly used instances of ECCs. An example of Hamming Code is shown in Figure 1.1 (from left to right: hamming code, bi-partite graph representation, parity-check equations and matrix). The code can detect and correct one bit error by computing $\boldsymbol{H} \times \mathbf{c}$ ($\boldsymbol{H}$ is the parity check matrix and $\mathbf{c}$ is the codeword). For example, if we receive a codeword $\mathbf{c} = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}^T$, applying $\boldsymbol{H}$ to $\mathbf{c}$ will give us $\mathbf{s} = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}^T$, which is the same as the second column of $\boldsymbol{H}$. This implies that the location of the error is at the second bit (therefore flip the bit to correct). Compared to Hamming codes, LDPC is a sparse code, where very few bits are set to 1 relative to the number of bits in the original word. The sparsity property of LDPC makes it attractive for use in *Fault Tolerant Storage*.

Erasure Coded Storage (ECS) Consider a code that augments m units of data with k redundant units, for a total of $n = k + m$ stored items. If such a code is capable of correcting k errors, then the code is space-optimal and is referred to as

Maximum Distance Separable (MDS) [17]. Figure 1.2 presents two examples of an MDS systematic code. The data vector is premultiplied by a distribution matrix to compute an augmented vector. We can recover the data vector from a single element erasure by solving the linear system corresponding to the remaining rows of the distribution vector, since they are linearly independent.

In general, a coding matrix (referred to as a distribution (or coding) matrix) has $m + k$ rows and m columns, with the property that any subset of m rows is linearly independent. The distribution matrix is multiplied by the data vector (of dimension m) to yield an $n = m + k$ items coded vector. If k elements of the coded vector are erased, one can recover the data vector by solving the linear system corresponding to the remaining m rows of the distribution matrix.

For example, the left distribution coding matrix in Figure 1.2 can help us tolerate one failure ($k = 1$). If one element of coded vector is erased, we can recover the data vector by subtracting the sum of all other non-faulty original rows from the redundant rows. By contrast, although the coding matrix on the right in Figure 1.2 has two redundant rows in coded data vector, it can only tolerate one fault. Any erasure of the odd numbered rows can be recovered from the first redundant row by linear operation, while any erasure of the even numbered rows can be recovered from the second redundant row.

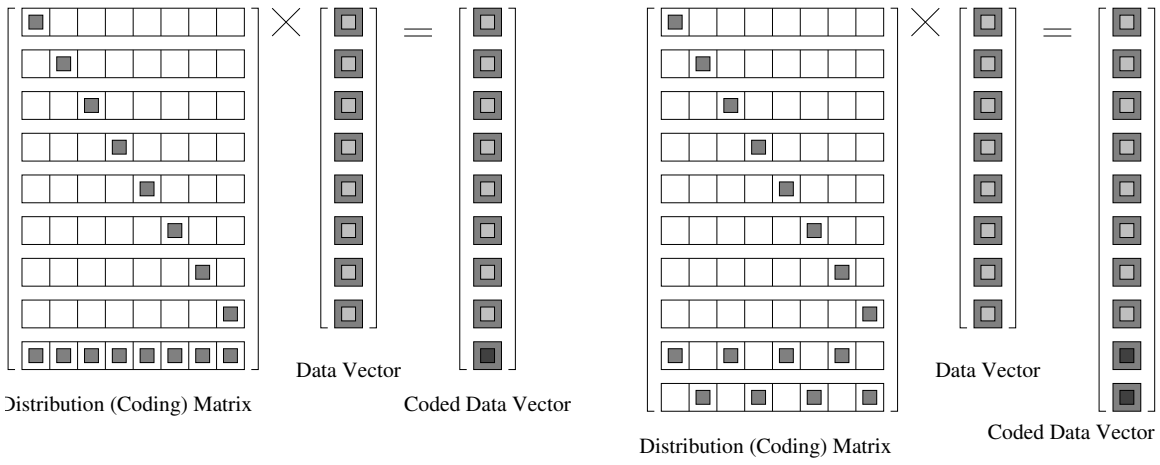


Figure 1.2.: Basic principles of erasure coding.

Erasure Coded Computation (ECC) While similar in spirit, our proposed concept of erasure coded computations in linear algebra has several significant differences from error correcting codes (ECCs) and erasure coded storage (ECS). First, unlike ECCs and ECS schemes, erasure coded computations aim to *code the results of a computation* in a fault tolerant manner, *as opposed to the data*. Second, data structures used are not linear bit strings (or blocks), but rather sparse matrices. Third, recovery is numerical in nature; i.e., we do not need to perform our computations over a finite field; instead, we use real arithmetic, which can greatly relieve the algorithmic burden of coding/decoding. Finally, erasure coded computations must deal with failure modes other than erasures. All of these issues pose significant challenges, which are the foci of recent work [18].

Basic Kernels of ECC We illustrate the idea for erasure coded computations in a simple form for a sparse matrix-vector product (mat–vec). We assume that this matrix is partitioned row-wise across processing nodes.

Figure 1.3 shows an example of such a matrix, partitioned across eight nodes. Figure 1.3(a) shows the sparse matrix and its (sparse) graph representation; Figure 1.3(b) illustrates the augmentation process when the given sparse matrix is premultiplied by a distribution matrix (also called the coding matrix), resulting in a new matrix with one extra row, which is the sum of all other rows shown in Figure 1.3(c); Figure 1.3(d) presents the graph view of the augmented matrix with a new node added; Figure 1.3(e) illustrates that using an alternate distribution matrix allows us to control the fills in the augmentation rows. The augmented block now consists of two (sparse) rows. This computation is still resilient to one process failure. The augmentation block can itself be distributed among processors for load balance. Consider the first simple case of coding the mat-vec using a parity distribution matrix (Figure 1.2). We multiply the distribution matrix by the given sparse matrix (Figure 1.3(b)) to get an augmented matrix, in which the first m rows remain unchanged because of the identity block of rows in the distribution matrix, whereas the last (augmented) row is

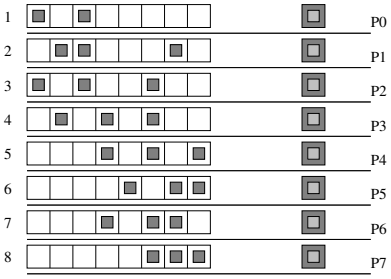
the negative sum of all other rows. In a graph view, this augmentation corresponds to the addition of a new node (node 9) that is connected to all other nodes in the original graph with a negative weight. The operation is now executed using nine processing nodes (as opposed to the original case of eight nodes). If one of the processes has a fail-stop failure, the corresponding vector element is lost. However, due to the structure on the row-sums, the missing element can be inferred because the sum of the output vector must be zero.

In linear algebraic techniques, we aim to encode the input problem by a coding (distribution) matrix and input the coded data into traditional methods based on the concept of erasure coded computations. The true solution can be extracted by a decoding procedure, which does not involve complicated/expensive computations under a potentially faulty environment. Hence, for traditional methods, the process is fault oblivious, and does not require rollbacks or restarts.

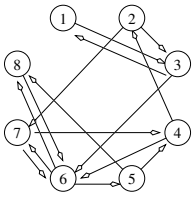
1.3 Benefits and Challenges in ECC

In Section 1.2, we show three related, yet critically distinct concepts – error correcting codes, erasure coded storage, and the proposed erasure coded computations. In this section, we will show the benefits and challenges, including the design of the coding schemes, the fault models, and the system support, of erasure coded computations.

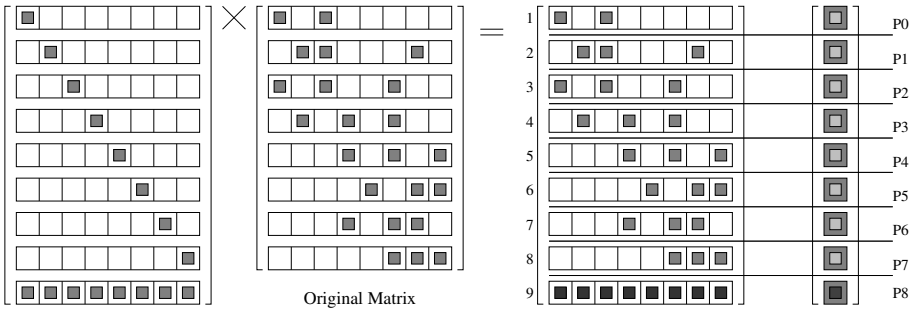
We note several important points related to this general scheme of fault oblivious computations: (i) Tolerating a single fault using traditional replication requires twice as much computer power and tolerating a single fault using deterministic replay doubles the makespan of the job, while our proposed scheme achieves fault tolerance to a single fault by increasing the computation only fractionally, depending on the code used. We show that the encoding, the execution, and the decoding procedures for our technique introduce little overhead in terms of excess computation and communication during a parallel execution. We show that our linear solvers are particularly



(a) Original Matrix



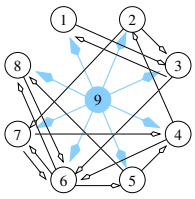
(b) Graph of Original Matrix



Distribution (Coding) Matrix

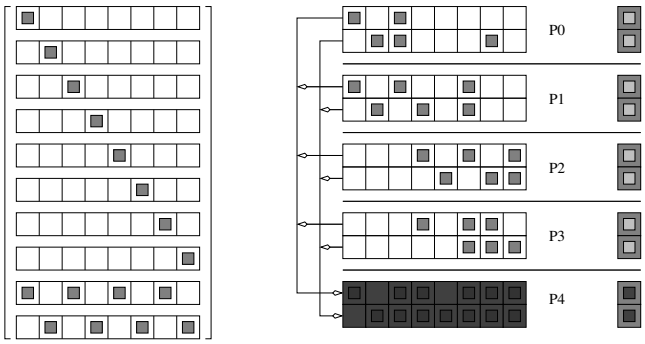
Original Matrix

Augmented Matrix



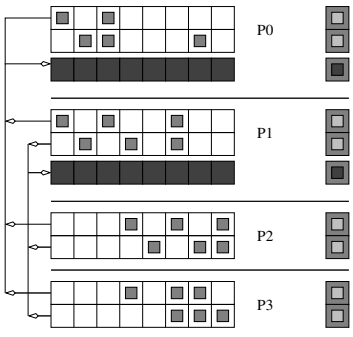
(d) Graph of augmented matı

(c) Augmentation Process: Multiplication by a distribution/ coding matrix.
Augmented matrix is tolerant to one node (row) failure.



Distribution (Coding) Matrix

Controlling density of augmented blocks by combining blocks (first row at each processor is combined into first row of the augmented block). Note the reduced fill. Matvec is still tolerant to one process failure.



Augmentation block is now distributed across processors. This addresses problem of load imbalance. Note higher communication cost of this scheme. This can be amortized through coarse−grained processor partitions.

(e) An alternate distribution (coding) matrix that reduces density at the cost of increased augmentation for the same fault tolerance.

Figure 1.3.: Illustration of the concept of erasure coded computations

attractive for systems with high fault rates, since they do not require checkpoints, are adaptive, in the sense that augmented blocks are introduced into the input matrix only when faults are detected, and that the recovery procedure, in the event of faults, is computationally simple and inexpensive [18–20]. (ii) The benefits of our scheme are significantly amplified as higher levels of fault tolerance are desired. (iii) Controlling fill in augmentation rows and associated communication overhead can be achieved using suitable codes. (iv) Load balancing considerations can be achieved by distributing the augmentation rows across processes. These benefits are validated by our recent work [19–21], which successfully solves the following challenges.

Deriving Sparse Erasure Coding Schemes The example in Figure 1.3 illustrates two codes – a parity code and a block parity code. In erasure coded storage, Vandermonde and Cauchy matrices are often used for generating codes [22,23]. While space efficient, these codes induce significant communication overhead in our setting. For instance, a Vandermonde matrix induces a dense augmentation block. The block parity code illustrated in Figure 1.3 induces augmentation blocks with $O(n)$ non-zeros (when using n processing nodes), assuming an $O(1)$ non-zeros in each row of the input matrix. While this is more desirable than the $O(r)$ communication and computation of the Vandermonde block (here r is the number of rows in the input matrix), the dense block imposes constraints on scaling.

Sparse matrices are generated from different applications ranging from mechanical engineering to biology systems. We need to solve these sparse matrices using linear algebraic methods. The dense coding matrix also imposes computation overheads for these problems, since the original problems are sparse. Hence, a sparse distribution (coding) matrix is more desirable. Sparse coding matrices should satisfy the properties required by the coding and decoding procedure. We tackle this problem using distribution matrices similar to structured low density parity codes (LDPC) [16]. These codes are described in Chapter 2.

ECC under Different Fault Models We have, thus far, considered only fail-stop failures [24]; i.e., in the event of a fault, a process will change to a state that permits other components to detect that a failure has occurred and then stops. Indeed, there are other fault models as well, ranging from transient (soft) faults to Byzantine behavior [25, 26]. Soft/transient faults manifest themselves in the form of erroneous data, which show up silently without any indication from the hardware/operating system. In the Byzantine model, a component can exhibit arbitrary and malicious behavior, perhaps involving collusion with other faulty components.

Our proposed method can be extended to these other fault classes using an application provided local fault detection scheme. These schemes are presented in the form of asserts (predicates whose violation signifies an error). When such a fault is locally detected, the process is killed and other processes are notified of the failure – thus emulating a fail-stop failure. Asserts work similarly when Byzantine failures are detected. Thus, a combination of tolerance to fail-stop failures with user-specified predicates for local fault detection allows us to handle a broader class of faults. Our methods are tested under different fault arrival models and fault rates for fail-stop failures. The results demonstrate the robustness of our methods, as shown in Chapter 2, Chapter 3 and Chapter 4.

Systems Support for ECC There are important issues related to system support and programming models that are associated with the proposed fault oblivious paradigm. Specifically, a single process failure often causes the entire program to crash. In yet other scenarios, a crashed process can cause group communication operations (reductions, broadcasts, etc.) to block. These kinds of behaviors would not allow the leveraging of our proposed scheme. Specifically, we assume that the faulty processes simply drop out of the ensemble, while the remaining processes continue. In this dissertation, we do not undertake the task of developing such software infrastructure. Rather, to demonstrate the feasibility and performance of our erasure coded computations, we code it entirely using asynchronous non-blocking communi-

cation operations. In other words, we simply ignore the faulty process, and it does not affect the operations on other processes. Although this is not a direct comparison with their synchronous counterparts, it allows us to demonstrate the performance and fault tolerance characteristics of our method.

1.4 Related Research

Our approach is best characterized as adapting the ideas (erasure coded storage) behind fault tolerant storage to linear algebraic problems to achieve fault tolerant computation. Our work can be considered as a hybrid of algorithm-based methods and system-supported methods, in which the original problem is modified for fault tolerance, and then executed by a standard algorithm in a fault oblivious manner to compute a solution. Compared to a checkpoint-restart method, our method does not need consistent checkpoints and significant rollback to find the restart point. In comparison to active replica, which needs k replicas to tolerate k faults, our method's resource overhead is low and does not need replicated executions. MapReduce, for example, has drawbacks of staged execution and increased job makespan, particularly when the number of faults is large. Our method does not need to rollback to restart from the point of last fault, and does not incur significant overhead with increasing number of faults.

In the context of linear system solvers using erasure coded computations, the results most closely related to ours originated by the work of Huang and Abraham [12]. They added a checksum row and column to detect soft errors in a variety of dense matrix computations. Their result motivated a line of research [6, 8–10, 27, 28] on generalizing the concept, dealing with fail-stop failures, and parallel computation. However, the focus of their work on iterative methods for linear systems involves efficiently emulating a checkpoint-restart system, where efficiently computed checksums are used instead of the checkpoints [28]. The solver restarts when faults are detected. Their efficiency is significantly degraded when the number of faults increases, or when

the system's fault rate is high. Our method simultaneously encodes the entire problem. The whole execution is fault oblivious and there is no restart. We exploit coding schemes in iterative algorithms differently, where we establish a general faremwork of erasure coded computations in iterative methods. This involves concepts related to the Kruskal rank [6, 29]. Prior work on fault tolerant iterative methods using coding involves some similar ideas, but in the context of algorithms that restore the state on a per-iteration basis, instead of solution recovery after the entire computation, as in our case. Alternative hybrid strategies for solving linear systems of equations with faults include the selective reliability framework, where algorithms are programmed to be tolerant to faults in certain regions of the computations [7]. In the context of linear solvers, these methods prescribe a set of *critical* work that must be done reliably and other sections of *fault tolerant* work that could proceed with a variety of soft errors. For instance, in linear system solvers, residual computations must be done reliably to gauge convergence, whereas preconditioning applications can tolerate a variety of faults. This setting then involves flexible Krylov subspace methods, such as flexible GMRES [30–33]. Generating synthetic but realistic soft errors [34] is a challenge for such methods. In comparison, our erasure coded approach only requires the encoding and final decoding to be reliable. The scope of critical work on encoding and decoding is small compared to the whole iterative procedure. Selective reliability for standard iterative methods usually requires sequences of reliable and fault tolerant sections of code.

In the context of eigenvalue solvers using erasure coded computations, the results most closely related to ours originated in the work [11–13]. The eigenvalue solver of Chen [11] relies on checksum coding, along with fault location and reconfiguration on detecting an erroneous signal. Different error detection schemes are implemented based on checksum rows and the property of problems and algorithms. When a fault is detected, the solver will roll back state to the last correct state. This approach involves consistent checkpoints and rollback procedures. Consistent checkpoints need significant bandwidth to I/O or extra memory for in-memory checkpoints. The roll-

back procedure decreases efficiency of the whole program, especially for the system with high fault rates.

In contrast, our method does not require checkpoints and rollback. A checksum method for QR decomposition (which can be used to solve for eigenvalues) was proposed by Luk and Park [13]. Although this method does not need to rollback computation to correct errors, it can only detect and correct one single (transient) error of the ideal case that only a single element of the matrix is changed at the k^{th} step of the computation. This is a very limited single transient error model. However, in a real system, it is very common that one or more processors become faulty and several faults happen at different stages of the execution. Even for an instantaneous fault model, where faults happen instantaneously, more than one fault may happen across multiple processors. In contrast, with precomputed coded blocks, we can tolerate as many faults as the size of coded block faults. Moreover, our method is robust to different fault models including the instantaneous model and the exponential model, where faults arrive randomly, following an exponential distribution. This is especially useful and applicable to real distributed systems with high fault rates.

1.5 Our Fault Oblivious Linear Algebraic Algorithms based on ECC

The concept of fault oblivious execution was introduced based on erasure coded computations [18]. Fault oblivious executions suitably augment the input to a parallel program and execute the program on this augmented input in a potentially faulty environment. For a class of faults (fail-stop), the program executes obliviously of the faults (i.e., stopped processes are not restarted), and generates an augmented output. The true output of the program is generated from an inexpensive operation on the augmented output of the faulty execution. This technique is different from traditional system-supported fault tolerance methods, for which resource overheads are a big concern. However, as shown in Section 1.3, erasure coded computations also

pose challenges including coding schemes, robustness to different fault models, and system requirements. We will present solutions for these problems in Chapter 2.

Based on the concept of erasure coded computations, the erasure coded linear system $\tilde{\boldsymbol{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$ is constructed, given the original system $\boldsymbol{A}\mathbf{x}^* = \mathbf{b}$ ($\mathbf{x}^*$ is the true solution of original linear system) and a coding matrix $\boldsymbol{E}$ as follows:

$$\underbrace{\begin{bmatrix} \boldsymbol{A} & \boldsymbol{AE} \\ \boldsymbol{E}^T\boldsymbol{A} & \boldsymbol{E}^T\boldsymbol{AE} \end{bmatrix}}_{\tilde{\boldsymbol{A}}} \underbrace{\begin{bmatrix} \mathbf{x}^* \\ 0 \end{bmatrix}}_{\tilde{\mathbf{x}}} = \underbrace{\begin{bmatrix} \mathbf{b} \\ \boldsymbol{E}^T\mathbf{b} \end{bmatrix}}_{\tilde{\mathbf{b}}} \tag{1.1}$$

The augmented system ($\tilde{\boldsymbol{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$) is solved in a potential faulty environment to tolerate faults and the true solution of original system ($\boldsymbol{A}\mathbf{x} = \mathbf{b}$) is recovered from $\tilde{\mathbf{x}}$. Properties of the augmented system will be shown in Chapter 2.

We address issues that arise in coding and executing erasure coded computations in distributed environments, including the sparsity of the erasure code and the partitioning considerations of augmented parts, when using a conjugate gradient (CG) solver [35]. We describe the construction of the sparse coding matrix $\boldsymbol{E}$ and prove the Kruskal rank property [29] of the proposed coding matrix $\boldsymbol{E}$. Compared to other fault tolerant algorithms, our erasure coded fault tolerant scheme separates fault tolerance from the algorithm itself, and enables the use of a simple distributed algorithm.

The details of parallel implementation, including matrix reordering, augmentation block partitioning, and erasure of data on a distributed platform are also presented. Our experimental results demonstrate the following, in the context of a distributed erasure coded conjugate gradient solver: (i) the overhead of input augmentation is low – orders of magnitude smaller than corresponding overhead for replicated execution; (ii) the conditioning of the augmented system is comparable to the original system as evidenced by its convergence rates; (iii) the increase in the number of iterations due to errors is low, as evidenced in comparison to a regular solver; (iv) the augmented input solver yields excellent parallel performance in the presence of errors; (v) the augmented input solver is robust to different fault arrival rates; and (vi) for three different fault arrival models – uniform, instantaneous, and random, our fault tolerant

solver yields excellent performance on convergence. All of these results are validated on small to medium sized systems through an MPI-based implementation.

The distributed fault tolerant linear system solver in Chapter 2 shows good performance in convergence, time overhead, parallel performance and robustness to different fault arrival models. The solver assumes an estimated maximum number of faults and augments the system prior to execution for this worst-case estimation. The overhead of this augmentation is paid in the solver, even if fewer faults are encountered. Hence, in the worst case scenario when there are no faults during execution, the computational overhead for the augmentation parts is paid at each iteration even when they do not contribute to the final results.

In Chapter 3, we propose an adaptive augmentation approach that augments the input matrix only when faults are encountered. When a set of rows is erased during execution, the same number of precomputed augmentation rows will be added to the input matrix. Consequently, the system being solved is identical in size to the input matrix. We will show how we can minimize the overhead associated with the augmentation in detail. The augmented system in Chapter 2 can be written as follows:

$$\begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} & \boldsymbol{Z}_1 \\ \boldsymbol{A}_{12}{}^T & \boldsymbol{A}_{22} & \boldsymbol{Z}_2 \\ \boldsymbol{Z}_1{}^T & \boldsymbol{Z}_2{}^T & \boldsymbol{E}^T\boldsymbol{A}\boldsymbol{E} \end{bmatrix} \begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_f \\ \mathbf{x}_r \end{bmatrix} = \begin{bmatrix} \mathbf{b}_c \\ \mathbf{b}_f \\ \boldsymbol{E}^T b \end{bmatrix}$$

Here $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{12}{}^T & \boldsymbol{A}_{22} \end{bmatrix}$, $\boldsymbol{E} = \begin{bmatrix} \boldsymbol{E}_1 \\ \boldsymbol{E}_2 \end{bmatrix}$ and $\begin{cases} \boldsymbol{Z}_1 = \boldsymbol{A}_{11}\boldsymbol{E}_1 + \boldsymbol{A}_{12}\boldsymbol{E}_2 \\ \boldsymbol{Z}_2 = \boldsymbol{A}_{12}^T\boldsymbol{E}_1 + \boldsymbol{A}_{22}\boldsymbol{E}_2 \end{cases}$. The size of matrix $\boldsymbol{E}$ is $n \times k$, which lets us handle up to k erasures. Here $\mathbf{x}_c$ and $\mathbf{x}_f$ correspond to the correct (fault free) and faulty parts, respectively, and $\mathbf{x}_r$ is the redundant part for the augmented system ($\mathbf{x}_c$ is an $(n-k)\times 1$ vector, $\mathbf{x}_f$ and $\mathbf{x}_r$ are $k \times 1$ vectors). In comparison, the adaptive fault tolerant linear system solver in this chapter will solve the following system when faults happen:

$$\begin{bmatrix} \boldsymbol{A}_{11} & \tilde{\boldsymbol{Z}}_1 \\ \tilde{\boldsymbol{Z}}_1^T & \tilde{\boldsymbol{E}}^T\boldsymbol{A}\tilde{\boldsymbol{E}} \end{bmatrix} \begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_r \end{bmatrix} = \begin{bmatrix} \mathbf{b}_c \\ \tilde{\boldsymbol{E}}^T\mathbf{b} \end{bmatrix} - \begin{bmatrix} \boldsymbol{A}_{12} \\ \tilde{\boldsymbol{Z}}_2^T \end{bmatrix} \mathbf{x}_f. \tag{1.2}$$

Here $\tilde{\boldsymbol{Z}}_1$, $\tilde{\boldsymbol{Z}}_2$, $\tilde{\boldsymbol{E}}^T\boldsymbol{A}\tilde{\boldsymbol{E}}^T$, $\tilde{\boldsymbol{E}}^T\mathbf{b}$ are the submatrices selected from the precomputed coding data ($\boldsymbol{Z}_1$, $\boldsymbol{Z}_2$, $\boldsymbol{E}^T\boldsymbol{A}\boldsymbol{E}$ and $\boldsymbol{E}^T\mathbf{b}$) respectively when an erasure occurs. This system has a number of desirable properties – (i) the method is computationally optimal in the sense that the system size stays the same as the input system, especially when the number of faults that occurred during execution is much less than k; (ii) the convergence property is 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 present, in detail, the augmentation process and the parallel formulation. The experimental results on sparse matrices 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 under different fault arrival models and rates.

Building on the erasure coded linear system solver, we apply the concept of erasure coding computations to an eigenvalue solver for $\boldsymbol{A}\mathbf{x} = \lambda\mathbf{x}$. There are three major differences between linear systems and eigenvalue systems. First, compared to the n unknowns in a linear system of the same dimension, an eigenvalue system has $n+1$ unknowns (an $n-$vector and a scalar λ); second, naively adding a coding block to a given matrix $\boldsymbol{A}$ changes its eigenvalues; lastly, there are no known computationally inexpensive ways of recovering the original eigenvalues from the perturbed eigenvalues. To address this, we transform the original eigenvalue problem to an equivalent (in terms of eigenvalues) generalized eigenvalue problem $\tilde{\boldsymbol{A}}\tilde{\mathbf{x}} = \lambda\tilde{\boldsymbol{B}}\tilde{\mathbf{x}}$, where $\tilde{\boldsymbol{A}}$ is an augmented form of input matrix $\boldsymbol{A}$ and $\tilde{\boldsymbol{B}}$ is a suitably constructed sparse matrix as follows:

$$\underbrace{\begin{bmatrix} \boldsymbol{A} & \boldsymbol{A}\boldsymbol{E} \\ \boldsymbol{E}^T\boldsymbol{A} & \boldsymbol{E}^T\boldsymbol{A}\boldsymbol{E} \end{bmatrix}}_{\tilde{\boldsymbol{A}}} \underbrace{\begin{bmatrix} \mathbf{x} \\ 0 \end{bmatrix}}_{\tilde{\mathbf{x}}} = \lambda \underbrace{\begin{bmatrix} \boldsymbol{I} & \boldsymbol{E} \\ \boldsymbol{E}^T & \boldsymbol{E}^T\boldsymbol{E} \end{bmatrix}}_{\tilde{\boldsymbol{B}}} \underbrace{\begin{bmatrix} \mathbf{x} \\ 0 \end{bmatrix}}_{\tilde{\mathbf{x}}}. \tag{1.3}$$

We prove that the eigenvalues of the original problem ($\boldsymbol{A}\mathbf{x} = \lambda\mathbf{x}$) and the new generalized eigenvalue problem ($\tilde{\boldsymbol{A}}\tilde{\mathbf{x}} = \lambda\tilde{\boldsymbol{B}}\tilde{\mathbf{x}}$) are identical in exact arithmetic. We characterize the impact of faults on the solution of the generalized eigenvalue problem, and present the reconstruction of the eigenvalues in the event of faults (recovery procedure). The properties of the generalized eigenvalue system inspires a perturbation and purification approach to construct a robust eigensolver.

We solve the reformulated generalized eigenvalue problem using TraceMin [36,37], which can simultaneously compute the top k smallest (largest) eigenvalues and corresponding eigenvectors of large sparse generalized eigenvalue problem $\boldsymbol{A}\mathbf{x} = \lambda\boldsymbol{B}\mathbf{x}$. We refer to this variant as the fault oblivious TraceMin algorithm. Our experimental results on sparse matrices show that our solver has excellent convergence properties (eigenvalues are very close to those of the original system) in the event of random faults (a set of randomly chosen row-column pairs are erased). We also evaluate the performance of our eigensolver under the worst case scenarios. It is well known that different rows of a matrix have different impacts on the eigenspectrum of the matrix [38, 39]. One way to characterize the importance of a row-column pair with respect to its eigenspectrum is using leverage scores [40–42]. We construct worst case scenarios for our solver by erasing row-column pairs with the top k highest leverage scores obtained from a prior SVD decomposition. We notice that our fault tolerant scheme needs more iterations to converge. This is due to the fact that our coding scheme is oblivious to leverage scores, which are not considered during construction of augmentation blocks. Since it is infeasible to precompute the leverage scores before constructing our coding blocks, we estimate the leverage scores from eigenvectors as the TraceMin computation proceeds, and dynamically update the code based on the estimated leverage scores. We demonstrate that by using this adaptive scheme, we can handle the worst case scenarios with minimal convergence degradation. We perform experiments to compare our erasure coded eigensolver against the base eigensolver in terms of: (i) the average case convergence rate; (ii) the worst case convergence properties; (iii) the effectiveness (in terms of convergence) of our updated algorithm using

adaptive coding technique on the worst case; (iv) the impact of using approximations of leverage scores (as opposed to exact leverage scores) to establish bounds; and (v) the impact of different fault arrival models (instantaneous versus random arrivals) and fault rates. Our experimental results show that our solver has fast convergence, can effectively handle worst case erasures, and is robust to different fault arrival models and fault rates.

We discuss avenues for future work in Chapter 5 and summarize this dissertation in Chapter 6.

2 DISTRIBUTED FAULT TOLERANT LINEAR SYSTEM SOLVER

A version of this chapter has been published in the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS) [19](DOI: 10.1109/ICDCS.2017.261).
In this chapter, our idea of erasure coded computations is applied to a linear system solver. We code the original system including the matrix $\boldsymbol{A}$ and right-hand-side vector $\mathbf{b}$ based on erasure coded computations. A solver will run on the coded system. Details of the erasure coded linear system solver will be given in Section 2.1. To minimize the fill in augmentation rows, we design a novel sparse coding matrix $\boldsymbol{E}$, which satisfies the recovery-at-random property. The structure of $\boldsymbol{E}$ and the proof of its properties are shown in Section 2.2. The partitioning considerations for data are given in Section 2.4 to achieve better parallel performance. The experimental results in Section 2.5 demonstrate that our distributed fault tolerant linear solver converges under non-faulty and faulty environments, exhibits good speedup and low computational overheads on distributed platforms, and adapts to different fault arrival rates and different fault arrival models.

2.1 Erasure Coded Linear System Solver

We now summarize the work in [18] that describes how to extend the idea of erasure coded computations to solving a linear system of equations. This forms the basis for the distributed solver proposed in this chapter. Let $\mathbf{x}^*$ be the true solution of the original linear system

$$\boldsymbol{A}\mathbf{x}^* = \mathbf{b} \tag{2.1}$$

We set $k \leq n$ to be the allowed number of faults. The encoding matrix $\boldsymbol{E}$ is an n-by-k matrix that has the property that any subset of k rows are linearly independent. This condition is equivalent to stating that $\boldsymbol{E}^T$ should have Kruskal rank k [29]. Given matrices $\boldsymbol{A}$ and $\boldsymbol{E}$, the augmented or encoded system $\tilde{\boldsymbol{A}}$, the solution to the augmented system $\tilde{\mathbf{x}}$, and the augmented right-hand-side $\tilde{\mathbf{b}}$, are given by:

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

With these augmented structures ($\tilde{\boldsymbol{A}}$ is $(n+k) \times (n+k)$ symmetric matrix and $\tilde{\mathbf{b}}$ is $n+k$ vector), we solve:

$$\tilde{\boldsymbol{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}} \tag{2.2}$$

Computing the new blocks of this system (such as $\boldsymbol{AE}$, $\boldsymbol{E}^T\boldsymbol{A}$ and $\boldsymbol{E}^T\boldsymbol{AE}$) must be done reliably through some type of existing fault tolerance scheme, although this is a small amount of work. When $\boldsymbol{E}^T$ has Kruskal rank k, Gleich et al. [18] prove a number of properties of $\boldsymbol{E}$ including the null space basis of $\tilde{\boldsymbol{A}}$ (the null space of $\tilde{\boldsymbol{A}}$ is $\begin{bmatrix} \boldsymbol{E} \\ -\boldsymbol{I}_k \end{bmatrix}$) and the symmetric positive semi-definiteness (SPSD). These properties guarantee that solution recovery is possible. In particular, let $\begin{bmatrix} \mathbf{y} \\ \mathbf{z} \end{bmatrix}$ be any solution of the augmented system, where $\mathbf{y} \in \Re^n$ and $\mathbf{z} \in \Re^k$, then it can be written as $\begin{bmatrix} \mathbf{y} \\ \mathbf{z} \end{bmatrix} = \begin{bmatrix} \mathbf{x}^* \\ 0 \end{bmatrix} + \begin{bmatrix} \boldsymbol{E} \\ -\boldsymbol{I}_k \end{bmatrix} \mathbf{a}$ for a unique $\mathbf{a} \in \Re^k$. To recover $\mathbf{x}^*$ in the presence of errors, we can use $\mathbf{y} = \mathbf{x}^* + \boldsymbol{E}\mathbf{a}$ and $\mathbf{z} = -\mathbf{a}$. These two equations lead to $\mathbf{x}^* = \mathbf{y} + \boldsymbol{E}\mathbf{z}$ for recovery [18]. This gives us a straightforward recovery algorithm. The solver needs a condition on matrix $\boldsymbol{E}$ such that: (i) there is always a solution to the augmented system for any faults as long as the number of faults is less than or equal to k (the column size of $\boldsymbol{E}$); (ii) given any solution obtained from the augmented system with faults, we can extract a solution for the original system ($\boldsymbol{A}\mathbf{x} = \mathbf{b}$) from it. This condition is equivalent to a full Kruskal rank of matrix $\boldsymbol{E}^T$ [29]. The vandermond matrix [23] is a matrix with full Kruskal rank and the random matrix used in [18]

also satisfies the full Kruskal rank requirement with probability tends to 1. The main drawback of these coding schemes is that the matrices are dense, which will increase the fills of $(1,2),(2,1),(2,2)$ blocks of $\tilde{\boldsymbol{A}}$. For a linear system with sparse matrix, computations resulting from the augmented parts can be as comparably expensive as the computation corresponding to the $(1,1)$ block of $\tilde{\boldsymbol{A}}$. With increasing number of faults (k), the computation overhead will skyrocket and the scheme is infeasible for a large distributed system with high fault rates. Hence, we use a sparse coding scheme that satisfies the random-recovery-property.

2.2 Coding Matrix

Having established the algebraic basis for our method (Section 2.1), we now focus on the problem of developing a suitable coding matrix $\boldsymbol{E}$ designed to work with distributed sparse matrices. This poses two challenges: first, the matrix $\boldsymbol{E}$ must satisfy certain algebraic properties of Kruskal rank in order to utilize the existing theory. Second, it must simultaneously minimize fills in the augmented matrix $\tilde{\boldsymbol{A}}$ for performance, both in terms of operation counts and communication. In order to achieve these dual objectives, we weaken the requirements on Kruskal rank k. In the proofs from Gleich et al. [18], having Kruskal rank k is necessary to guarantee that *any possible* subset of components can fail and we can always recover the true solution. This is obviously the ideal scenario. However, it is a worst-case analysis. In this chapter, we relax the requirement to meet only the *recovery-at-random* property.

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 $\geq 1 - 1/n^c, c > 1$.

We construct a matrix $\boldsymbol{E}$ by setting p successive elements in each row to non-zero elements. These p non-zero elements are selected in a staggered manner, i.e., the first p elements in Row 1, one zero followed by p non-zero elements in Row 2, two zeros

followed by p non-zero elements in Row 3, and so on. More generally, 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 $(0, 1)$. An example with $k = 6$, $p = 3$ is:

$$E = \begin{bmatrix} \bullet & \bullet & \bullet & 0 & 0 & 0 \\ 0 & \bullet & \bullet & \bullet & 0 & 0 \\ 0 & 0 & \bullet & \bullet & \bullet & 0 \\ 0 & 0 & 0 & \bullet & \bullet & \bullet \\ \bullet & 0 & 0 & 0 & \bullet & \bullet \\ \bullet & \bullet & 0 & 0 & 0 & \bullet \\ \bullet & \bullet & \bullet & 0 & 0 & 0 \end{bmatrix}$$

This choice of matrix $\boldsymbol{E}$ satisfies the recovery-at-random property and it is inspired by low density parity check (LDPC) codes which are a class of linear block codes [16]. LDPC codes have the characteristic of their parity-check matrix which contains only a few 1s in comparison to the amount of 0s, while still have the capacity very close to the theoretical maximum of allowing noise [14]. We will prove the property of our choice of matrix $\boldsymbol{E}$ by the following proposition and theorems.

Proposition 2.2.1 *Let $\boldsymbol{E}'$ be a submatrix of $\boldsymbol{E}$ formed by selecting any p rows of matrix $\boldsymbol{E}$. The matrix $\boldsymbol{E}'^T$ has rank p (alternately, any p rows of matrix $\boldsymbol{E}$ are linearly independent).*

Case 1

$$\begin{bmatrix} \bullet & \bullet & \bullet & 0 & 0 & 0 \\ 0 & \bullet & \bullet & \bullet & 0 & 0 \\ 0 & 0 & \bullet & \bullet & \bullet & 0 \end{bmatrix}$$

Case 2

$$\begin{bmatrix} \bullet & \bullet & \bullet & 0 & 0 & 0 \\ \bullet & \bullet & \bullet & 0 & 0 & 0 \\ \bullet & \bullet & \bullet & 0 & 0 & 0 \end{bmatrix}$$

Combination of Case 1 and Case 2

$$\begin{bmatrix} \bullet & \bullet & \bullet & 0 & 0 & 0 \\ 0 & \bullet & \bullet & \bullet & 0 & 0 \\ 0 & \bullet & \bullet & \bullet & 0 & 0 \end{bmatrix}$$

Proof. We consider different cases as shown above: (i) all rows of matrix $\boldsymbol{E}'$ have distinct non-zero structure; in this case, all rows of $\boldsymbol{E}'$ are trivially linearly independent; and (ii) all rows of matrix $\boldsymbol{E}'$ have the same non-zero structure; i.e., the non-zeros form

a $p \times p$ non-zero block in matrix $\boldsymbol{E}'$. In this case, we know that the $p \times p$ block of randomly generated entries is non-singular with probability tends to 1; and (iii) some rows have the same non-zero structure and some rows have distinct non-zero structures; this case is the combination of the above two cases. The rows with identical non-zeros structure have full rank as the number of non-zeros on each row is larger than the number of rows, and rows with distinct non-zeros will be linearly independent as in the first case. Therefore, all rows of matrix $\boldsymbol{E}'$ are linearly independent in this case as well. Hence, the matrix $\boldsymbol{E}'$ has rank p, which means any p rows of matrix $\boldsymbol{E}$ are linearly independent. ■

Proposition 3.1 shows that any submatrix of p rows of matrix $\boldsymbol{E}$ has full rank. We 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 rows, each with the same non-zero structure, we end up with a submatrix of rank p, which is less than k. It is important to note that this row-selection process is determined by the location of failures in the system. We show in Theorem 3.2 that such degenerate selections are highly unlikely.

Theorem 2.2.1 *The probability that a random set of k rows of matrix $\boldsymbol{E}$ is linearly dependent is less than $(\frac{e}{p+1})^{p+1}$.*

Proof. 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 $(\frac{1}{k})^{p+1}$. 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:

$$Pr(F) \leq \binom{k}{p+1} \cdot \left(\frac{1}{k}\right)^{p+1}$$

Here the random variable F representing the k rows of matrix $\boldsymbol{E}$ that are linearly dependent. For $\binom{k}{p+1}$, we have the following inequality based on Stirling approximation [43]:

$$\begin{aligned}
\binom{k}{p+1} &= \frac{k!}{(p+1)!(k-p-1)!} \\
&= \frac{(k/e)^k\sqrt{2\pi k}}{(\frac{p+1}{e})^{p+1}\sqrt{2\pi(p+1)}(\frac{k-p-1}{e})^{k-p-1}\sqrt{2\pi(k-p-1)}} \\
&= \frac{k^k\sqrt{k}}{(p+1)^{p+1}(k-p-1)^{k-p-1}\sqrt{2\pi}\sqrt{(p+1)(k-p-1)}} \\
&\leq \frac{k^k}{(p+1)^{p+1}(k-p-1)^{k-p-1}}
\end{aligned}$$

Based on the inequality, we can get the upper bound of $Pr(F)$ as follows:

$$\begin{aligned}
Pr(F) &\leq \binom{k}{p+1}\cdot\left(\frac{1}{k}\right)^{p+1} \\
&\leq \frac{k^k}{(p+1)^{p+1}(k-p-1)^{k-p-1}}\cdot\left(\frac{1}{k}\right)^{p+1} \\
&\leq \frac{k^{k-p-1}}{(p+1)^{p+1}(k-p-1)^{k-p-1}} \\
&\leq (\frac{k}{k-p-1})^{k-p-1}\frac{1}{(p+1)^{p+1}} \\
&\leq ((1+\frac{p+1}{k-p-1})^{\frac{k-p-1}{p+1}})^{p+1}\frac{1}{(p+1)^{p+1}} \\
&\leq (\frac{e}{p+1})^{p+1}
\end{aligned}$$

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. ■

We now focus on the problem of selecting the smallest value of p in our coding matrix $\boldsymbol{E}$ to guarantee that we are unlikely to have too many rows with the same non-zero pattern. We investigate the expected maximum number of rows that share the same non-zero structure. If this expected number exceeds p, our recovery-at-random condition fails. Therefore, we must select p greater than this expected maximum number.

Theorem 2.2.2 *The expected maximum number of rows from among k randomly selected rows of matrix $\boldsymbol{E}$ that have the same nonzero structure is $O(\frac{\ln k}{\ln \ln k})$.*

Proof. Define a random variable M to be the maximum number of rows that have the same non-zero structure when we select k rows uniformly at random from matrix $\boldsymbol{E}$ and $Pr(M = t)$ be the probability that the random variable takes value t. For a given non-zero structure, we fashion each row selection as a Bernoulli trial, with success corresponding to the selection of the specified row structure, and failure otherwise. Since there are k distinct row structures, all with identical probability of selection, the probability of exactly t successes is given by the Bernoulli distribution as $\binom{k}{t}(\frac{1}{k})^t(1-\frac{1}{k})^{k-t}$. Since there are k different row structures from which we select the one common structure, the probability:

$$Pr(M = t) = \binom{k}{1}\binom{k}{t}\left(\frac{1}{k}\right)^t\left(1-\frac{1}{k}\right)^{k-t} \leq k\left(\frac{e}{t}\right)^t$$

For $t = c\frac{\ln k}{\ln \ln k}$, where c is some constant, we can show $Pr(M = t) \leq \frac{1}{k^2}$ as follows:

$$\begin{aligned} Pr(M = t) &\leq k\left(\frac{e}{t}\right)^t \\ &= k\left(\frac{e \ln \ln k}{c \ln k}\right)^{\left(\frac{c \ln k}{\ln \ln k}\right)} \\ &\leq exp\left(\ln k + \frac{c \ln k}{\ln \ln k}\left(\ln \ln \ln k - \ln \ln k\right)\right) \\ &= exp\left((1-c)\ln k + \frac{c \ln k \ln \ln \ln k}{\ln \ln k}\right) \end{aligned}$$

For sufficiently large k, we have $\ln \ln k \geq 2 \ln \ln \ln k$. Therefore,

$$\begin{aligned} Pr(M = t) &\leq exp\left(\left(1-\frac{c}{2}\right)\ln k\right) \\ &= \frac{1}{k^{(c/2-1)}} \end{aligned}$$

For $c = 4$, we have $Pr(M = t) \leq \frac{1}{k}$ and for $c = 6$, we have $Pr(M = t) \leq \frac{1}{k^2}$. Hence, the expected maximum number of rows sharing a row structure $E(M)$ from k randomly selected rows of matrix $\boldsymbol{E}$ is given by:

$$
\begin{aligned}
E(M) &= \sum_{t=1}^{k} t \cdot Pr(M = t) \\
&= \sum_{t=1}^{c\frac{\ln k}{\ln\ln k}} t \cdot Pr(M = t) + \sum_{t=\frac{c\ln k}{\ln\ln k}}^{k} t \cdot Pr(M = t) \\
&\leq \sum_{t=1}^{\frac{c\ln k}{\ln\ln k}} \frac{c\ln k}{\ln\ln k} \cdot Pr(M = t) + \sum_{t=\frac{c\ln k}{\ln\ln k}}^{k} k \cdot Pr(M = t) \\
&= \frac{c\ln k}{\ln\ln k} \sum_{t=1}^{\frac{c\ln k}{\ln\ln k}} Pr(M = t) + k \sum_{t=\frac{c\ln k}{\ln\ln k}}^{k} Pr(M = t) \\
&\leq \frac{c\ln k}{\ln\ln k} \times 1 + k \sum_{t=\frac{c\ln k}{\ln\ln k}}^{k} Pr(M = t) \\
&\leq \frac{c\ln k}{\ln\ln k} + k \cdot \frac{1}{k^{c/2-1}} \\
&\leq O(\frac{\ln k}{\ln\ln k}).
\end{aligned}
$$

■

The last two inequalities hold as $k \cdot \frac{1}{k^{c/2-1}} \leq \frac{1}{k^{c/2-2}} \leq 1$ when $c \geq 4$, and the last inequality holds based on $k \cdot \frac{1}{k^{c/2-1}} \leq 1$. Theorems 3.2 and 3.3 provide bounds on values of p for a given value of k. In our experiments, we select $k = 4, 8, 16$, and the number of non-zeros per row $p = \min(k, 5)$. For these selections, our choice of p exceeds the expected number of rows that share the same row structure.

2.3 Parallel implementation of erasure coded CG

Gleich et al. show that $\tilde{\boldsymbol{A}}$ is symmetric-positive-semi-definite when $\boldsymbol{A}$ is symmetric positive definite and we can apply CG in this setting when $\boldsymbol{A}$ and $\mathbf{b}$ are consistent [18, 44]. This requires the following two-term recurrence form of CG [45]. The augmented matrix $\tilde{\boldsymbol{A}}$ and the augmented vectors are distributed among multiple processes by

Algorithm 1 Fault Oblivious CG with a two-term recurrence.

1: Let x_0 be the initial guess and $r_0 = b - Ax_0$, $\beta_0 = 0$.
2: **for** $t = 0, 1, \ldots$ until convergence **do**
3: $\beta_{t=} \begin{cases} 0 & t = 0 \text{ or after a fault} \\ \langle r_t, r_t \rangle / \langle r_{t-1}, r_{t-1} \rangle & t > 0 \end{cases}$
4: $p_t = r_t + \beta_t p_{t-1}$
5: $q_t = Ap_t$
6: $\alpha_t = \langle r_t, r_t \rangle / \langle q_t, p_t \rangle$
7: $x_{t+1} = x_t + \alpha_t p_t$
8: $r_{t+1} = r_t - \alpha_t q_t$
9: **end for**

rows. The operations of Algorithm 1 affected by faults in a distributed environment are the aggregation operations – inner products and matrix-vector multiplication. Let the index set associated with process i be I_i, then $[n+k] = \bigcup_i I_i$ and the set of faulty indices is $F_t = \bigcup_{i \in P_t} I_i$. P_t is the set of indices of erased rows from process P at time t.

- Inner products $\langle \mathbf{r}_t, \mathbf{r}_t \rangle$ and $\langle \mathbf{q}_t, \mathbf{p}_t \rangle$. The viable processes carry out the all-reduce operation by skipping the faulty components F_t in the vectors.

$$\langle \mathbf{r}_t, \mathbf{r}_t \rangle = \langle (\mathbf{r}_t)_{[n+k]\setminus F_t}, (\mathbf{r}_t)_{[n+k]\setminus F_t} \rangle$$
$$\langle \mathbf{q}_t, \mathbf{p}_t \rangle = \langle (\mathbf{q}_t)_{[n+k]\setminus F_t}, (\mathbf{p}_t)_{[n+k]\setminus F_t} \rangle$$

- Matrix-vector multiplication $\mathbf{q}_t = \boldsymbol{A}\mathbf{p}_t$. A viable process carries out its local aggregation operation for computing $\boldsymbol{A}_{I_i,:}\mathbf{p}_t$ by skipping the faulty components F_t in $\mathbf{p}_t$.

$$\boldsymbol{A}_{I_i,:}\mathbf{p}_t = \boldsymbol{A}_{I_i,[n+k]\setminus F_t}(\mathbf{p}_t)_{[n+k]\setminus F_t}$$

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 $\check{\mathbf{p}}_t = \mathbf{r}_t + \beta_t \mathbf{p}_{t-1}$ to be $\mathbf{p}_t = \mathbf{r}_t$. This corresponds to a reset of the Krylov process.

We now consider the recovery of the solution to the original (raw) system (2.1). Suppose the erasure-coded CG converges on the augmented system (2.2) after T iterations. Let the returned approximate solution be $\tilde{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ \mathbf{r} \end{bmatrix}$. We can recover the intended solution to the original (raw) system (2.1) through the equation

$$\begin{bmatrix} \mathbf{x}^* \\ 0 \end{bmatrix} = \tilde{\mathbf{x}} + \begin{bmatrix} \boldsymbol{E} \\ -\boldsymbol{I}_k \end{bmatrix} \mathbf{r}.$$

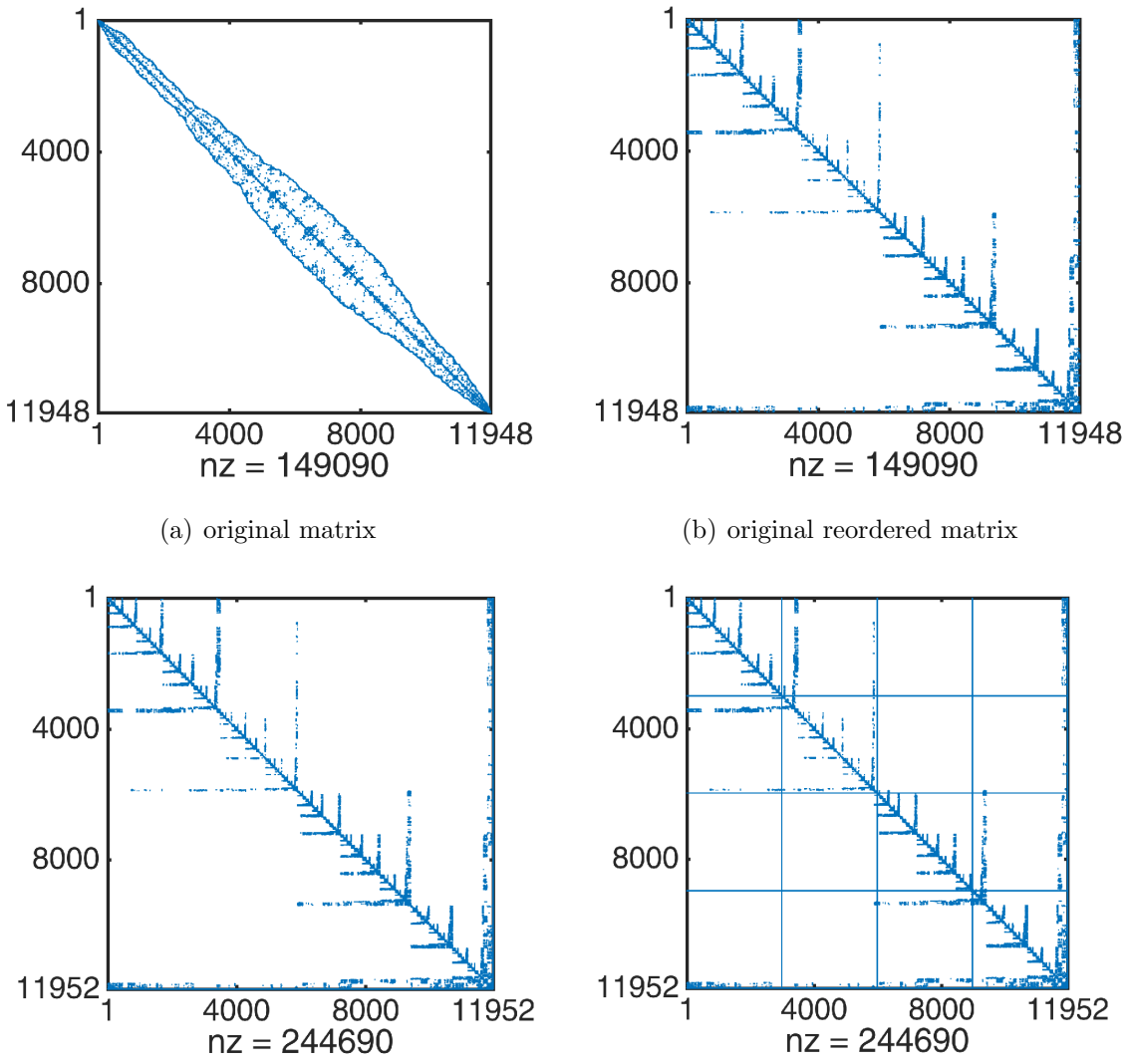
which means $\mathbf{x}^* = \mathbf{x} + \boldsymbol{E}\mathbf{r}$.

2.4 Partitioning Considerations

Even with the sparse matrix $\boldsymbol{E}$ described in Section 2.2, the augmentation blocks ($(1,2), (2,1), (2,2)$ 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 reorder the input matrix (Figure 2.1(a)) using a traditional matrix/graph partitioning technique such as Metis (Figure 2.1(b)) [46]. We then use this reordered matrix to compute $\tilde{\boldsymbol{A}}$ (Figure 2.1(c)). The resulting matrix is then reordered once again to partition $\tilde{\boldsymbol{A}}$ across various nodes in a parallel/distributed platform (Figure 2.1(d)) manually, which means we distribute the augmentation part into each processor according to Round-Robin strategy [47]. The first reordering minimizes non-zeros in $\tilde{\boldsymbol{A}}$, and the second partitioning step minimizes communication for the solver applied to $\tilde{\boldsymbol{A}}$ and balances the computation load for each processor.

2.5 Experimental Results

We report comprehensive results from our MPI implementation tested on up to 16 processing cores on four selected matrices. We benchmark the parallel performance of our solver on a 192 core (8 sockets) Intel(R) Xeon(R) Platinum 8168 processor operating at 2.70GHz.



(a) original matrix

(b) original reordered matrix

(c) augmented matrix

(d) augment reordered matrix

Figure 2.1.: Illustration of computing and repartitioning augmented matrix $\tilde{A}$.

Input Matrices, Augmented Systems, and Execution Parameters We select matrices from the University of Florida Matrix Collection for our experiments [48]. Matrix sizes and number of non-zeros are summarized in Table 2.1.

Table 2.1.: Matrices Used in Testing

Matrix	Rows	Non-zeros
bcsstk18	**11,948**	**149,090**
consph	**83,334**	**6,010,480**
inline_1	**503,712**	**36,816,170**
ldoor	**952,203**	**42,493,817**

All of the test matrices are symmetric positive definite (SPD), i.e., both CG on the original and augmented system converge on these matrices. The right-hand-side vectors $\mathbf{b}$ are firstly normalized (which means $||\mathbf{b}||_2 = 1$). Hence, the final relative error $rtol = \frac{||\boldsymbol{A}\mathbf{x}-\mathbf{b}||_2}{||\mathbf{b}||_2}$ equals to $||\boldsymbol{A}\mathbf{x} - \mathbf{b}||_2$ which is the same as the residual. During CG, we monitor residual $||\mathbf{r}||_2 = ||\boldsymbol{A}\mathbf{x} - \mathbf{b}||_2$ at each step and set the termination condition as $||\mathbf{r}||_2 < 10^{-06}$ for all matrices. We also set the maximum number of iterations for CG as 10000. All matrices are first reordered using Metis [46]. To construct the augmented system, we first build a suitable encoding matrix $\boldsymbol{E}$ as described in Section 2.2, and use this coding matrix to generate the augmented system. After generating the augmented system, we will manually distribute the augmentation parts to each processor based on a Round-Robin scheme [47].

We test the solver with a number of different fault models. In our initial set of experiments we use an instant fault arrival model – faults arrive at iteration 1000 instantaneously (all faults happen at the same time). Faults are distributed uniformly across processors. Once faults happen, the search direction of CG is set as the residual from the last iteration before faults happened. We also test it with a uniform fault arrival model, which means faults happen every certain number of iterations. Different fault arrival rates (fault steps) are tested to show the effect of fault step on convergence. Later in this section, we describe experiments with an alternate fault

model (random faults following exponential distribution) and quantify its effect on convergence.

2.5.1 Convergence of Raw System and Augmented System

Our first set of experiments is designed to compare the convergence rates of the original system and the augmented system. Figure 2.2 to Figure 2.5 show convergence of CG for the matrices for the original and augmented system without and with faults happening during execution. Here, K is the size of the redundant part which is also the number of faults which will happen during execution. $K = 0$ represents the original system with no faults happening. In the situation where faults happen, the faults happen at the 1000^{th} iteration instantaneously.

Without faults To evaluate the convergence of the augmented system, we execute the solver on the augmented matrix under the situation that there are no faults happening. The relative residual for the augmented system is computed with respect to the original system. Figure 2.2 and Figure 2.3 plot the results.

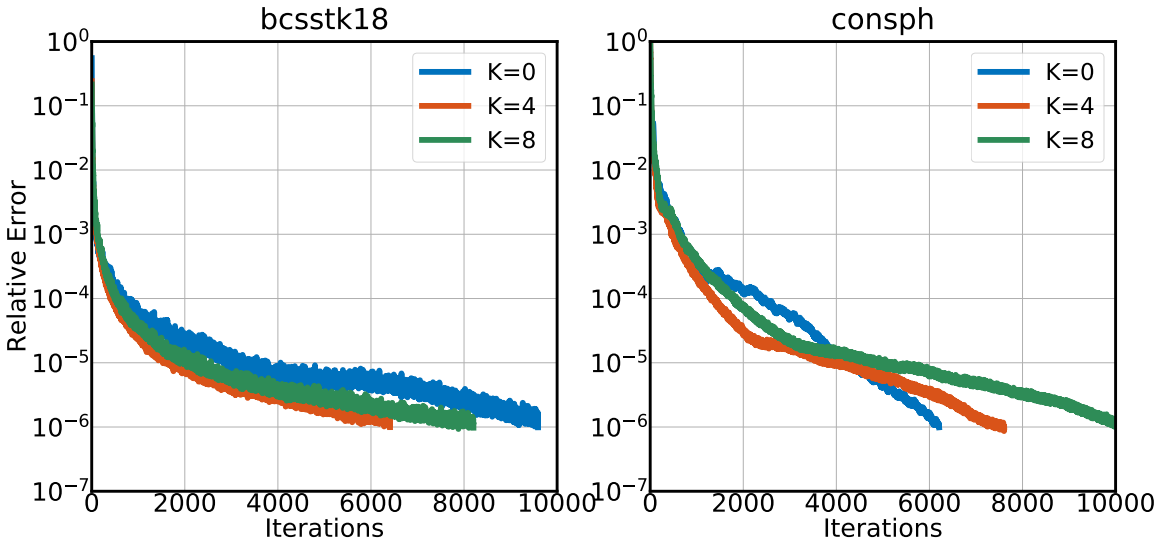


Figure 2.2.: Convergence of original and augmented system on small matrices(fault-free).

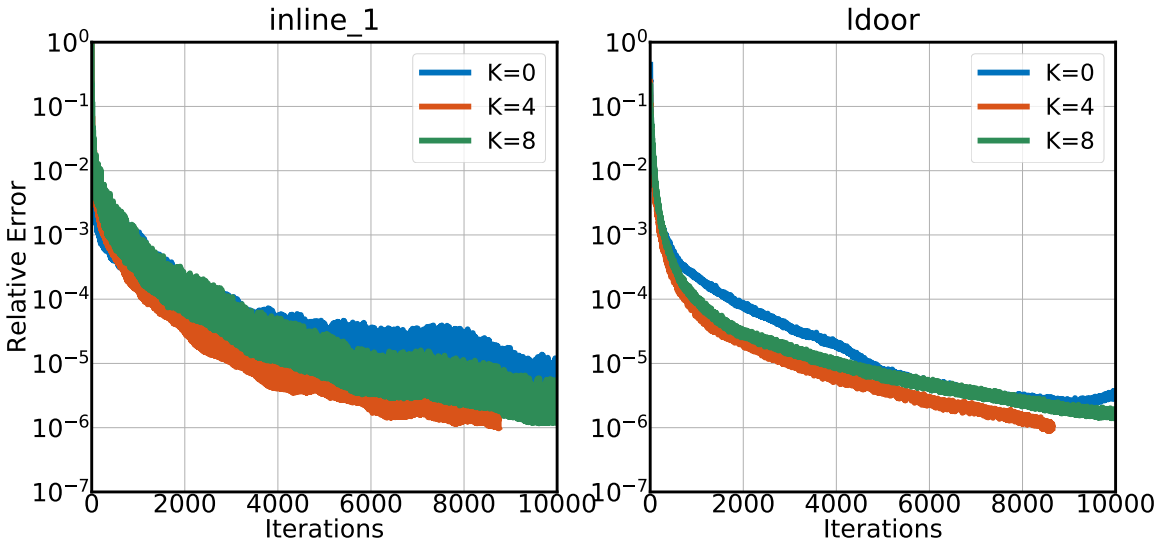


Figure 2.3.: Convergence of original and augmented system on large matrices(fault-free).

We note from these experiments that the augmented system can always achieve the same level of relative error as the original system. For matrices `bcsstk18`, `inline_1` and `ldoor`, the augmented systems with $K = 4$ converge even faster than the original system. With a larger number of faults ($K = 8$), the augmented systems give the same level of relative error with acceptable (less than 50% more iterations required by original system) convergence delay. This implies that the input augmentation does not adversely or significantly impact convergence of the base iterative solver.

With faults To evaluate the convergence of the solution to the original system, we execute the solver on the augmented matrix, except, in this case a number of faults happen during the execution. This set of results demonstrates how well the augmented system converges to the true solution. Figure 2.4 and Figure 2.5 plot the convergence. For matrices `bcsstk18`, `inline_1` and `ldoor`, we can achieve nearly the same level of relative tolerance as the original system. For matrix `consph`, although we have small delay of convergence, we can still reach the same level of relative error as the original system without faults. The convergence of the solution with respect to

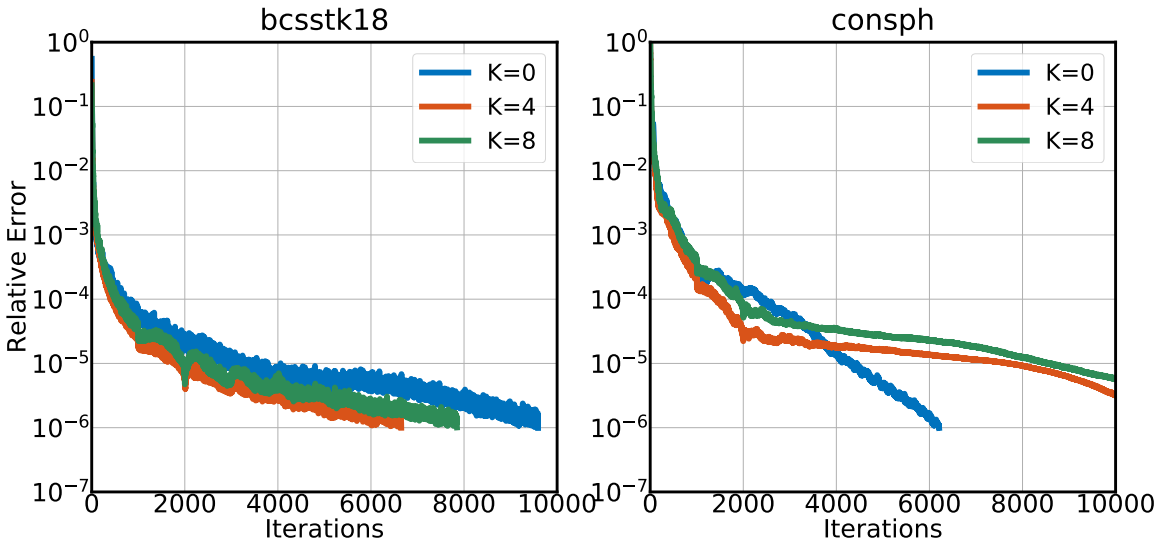


Figure 2.4.: Convergence of original and augmented system on small matrices(faulty).

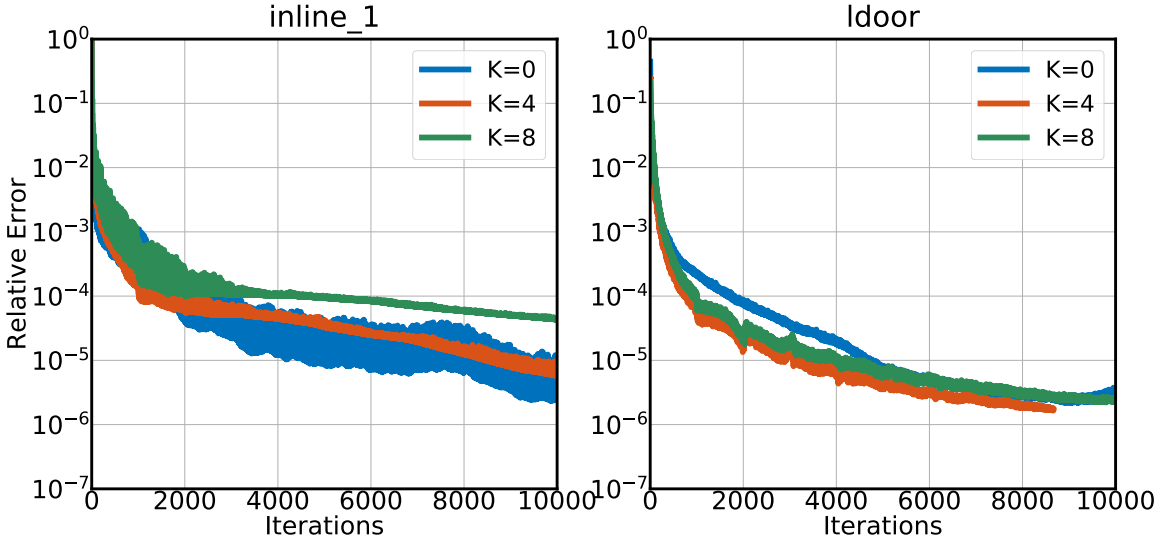


Figure 2.5.: Convergence of original and augmented system on large matrices(faulty).

the original system is comparable to the base (error free solver applied to the original system) method.

2.5.2 Parallel Performance of the Solver on Augmented Systems

In this section, we will show the parallel performance of our fault tolerant solver on a distributed platform. The tests were done on the platform with Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz with 384 cores.

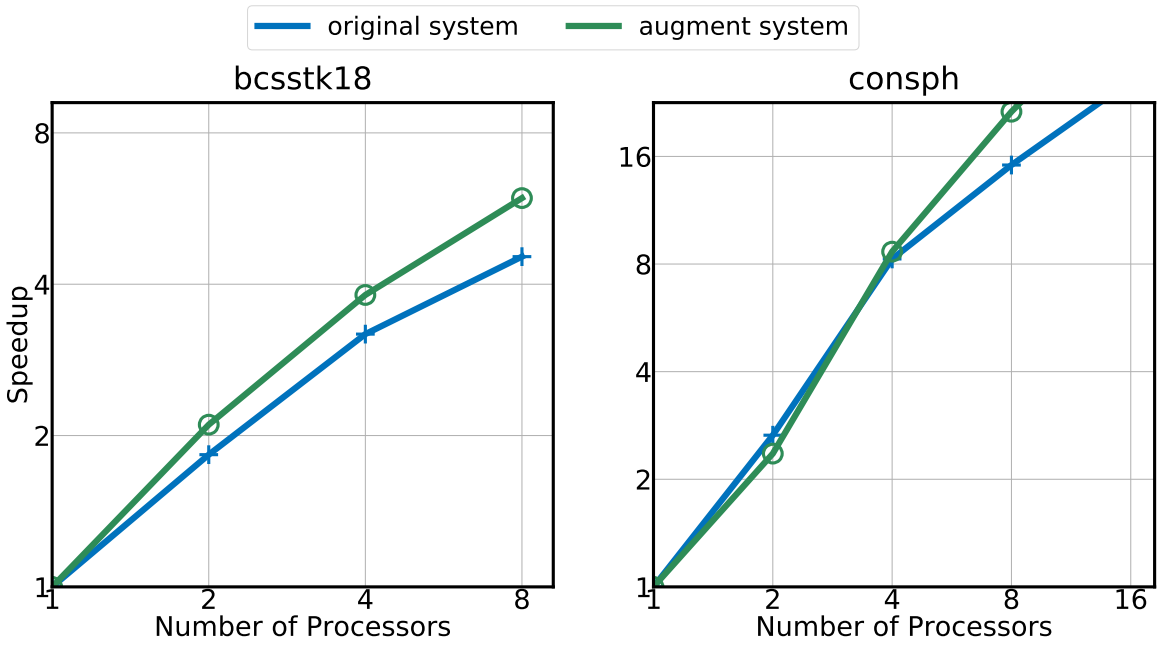


Figure 2.6.: Parallel performance of small matrices with $K = 16$.

Figure 2.6 and Figure 2.7 plot speedups from parallel execution of the solver on the original and augmented systems. For the augmented systems, we use $K = 16$ for all experiments. We notice here that in all cases, the solver yields good speedups; however, we also notice that the augmented system yields superior speedups compared to the original system. This can be explained by the fact that the augmented system is slightly larger, and therefore, has more computation associated with it. This, combined with similarly low communication overhead, yields excellent speedups for the augmented systems.

Besides the parallel performance, we also show the time overheads of our solver with different sizes of augment parts. Figure 2.8 and Figure 2.9 show the time mag-

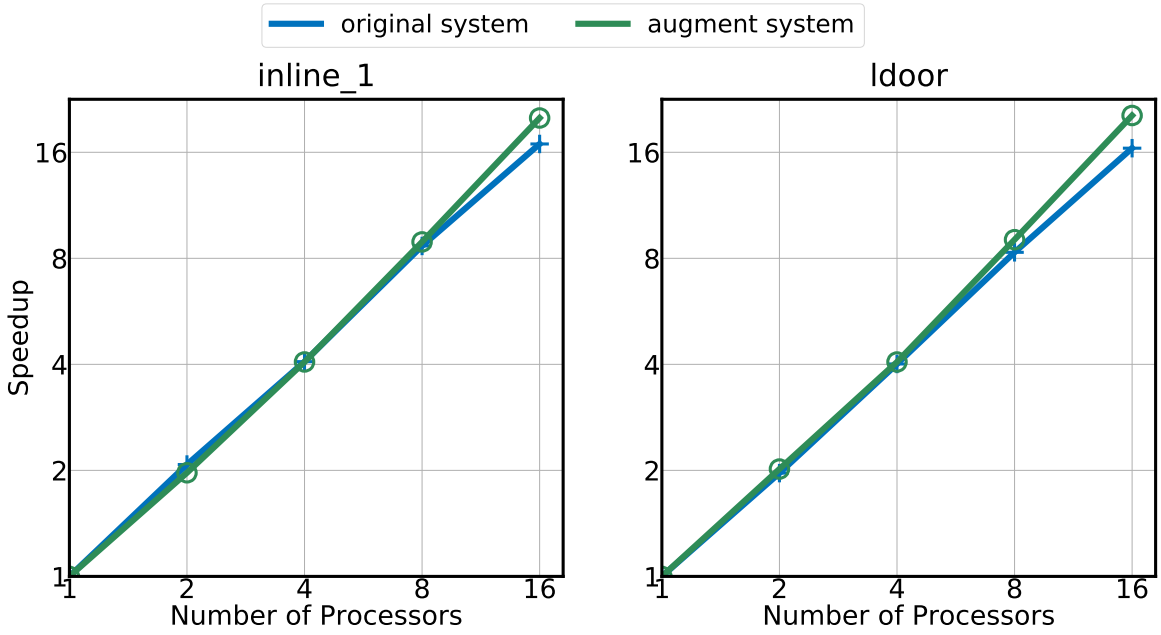


Figure 2.7.: Parallel performance of large matrices with $K = 16$.

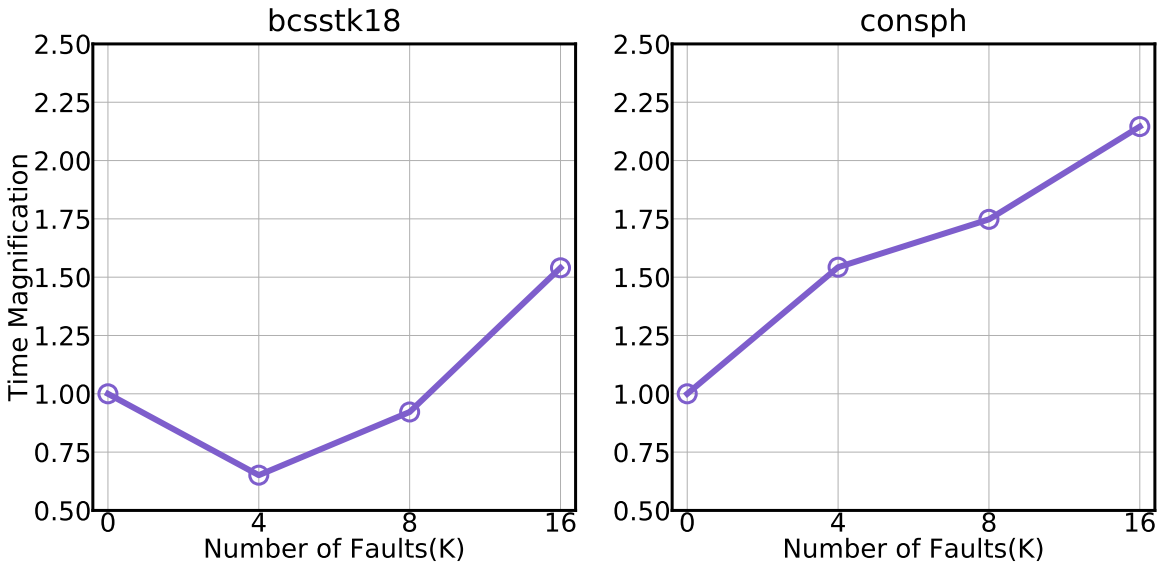


Figure 2.8.: Time overhead of augmented system on small matrices.

nification of the augmented system with different augmented sized compared to the original system. As the augmented size increases, the running time also increases.

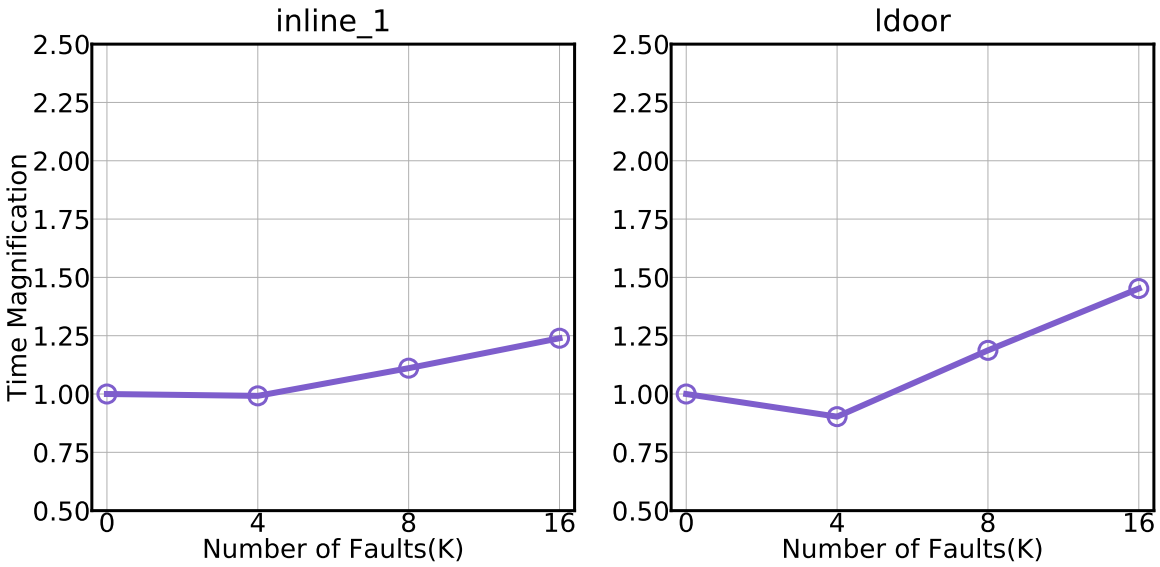


Figure 2.9.: Time overhead of augmented system on large matrices.

The increased time comes from computation overheads caused by the augmentation parts and the convergence delay (more iterations are needed to achieve the same level of relative errors). However, the increase in runtime is much lower, compared with the use of active replicas, for instance. With $K = 16$, the time magnification is just around 2.0, which is much smaller than the time of 16 needed by the active replicas method.

2.5.3 Convergence of Different Fault Arrival Rates

All of our previous results assumed a simple fault arrival model in which faults arrived instantaneously. In reality, the faults may come at different times and one by one. In the next set of results, we vary the fault arrival rate (number of iterations between two consecutive faults) and test the convergence property of our solver.

Figure 2.10 to Figure 2.13 plot the convergence rate of the solver for varying fault arrival rates with number of faults $K = 4$ and $K = 8$ on small and large matrices. We observe faster convergence when arrival rate is slow (fault step between

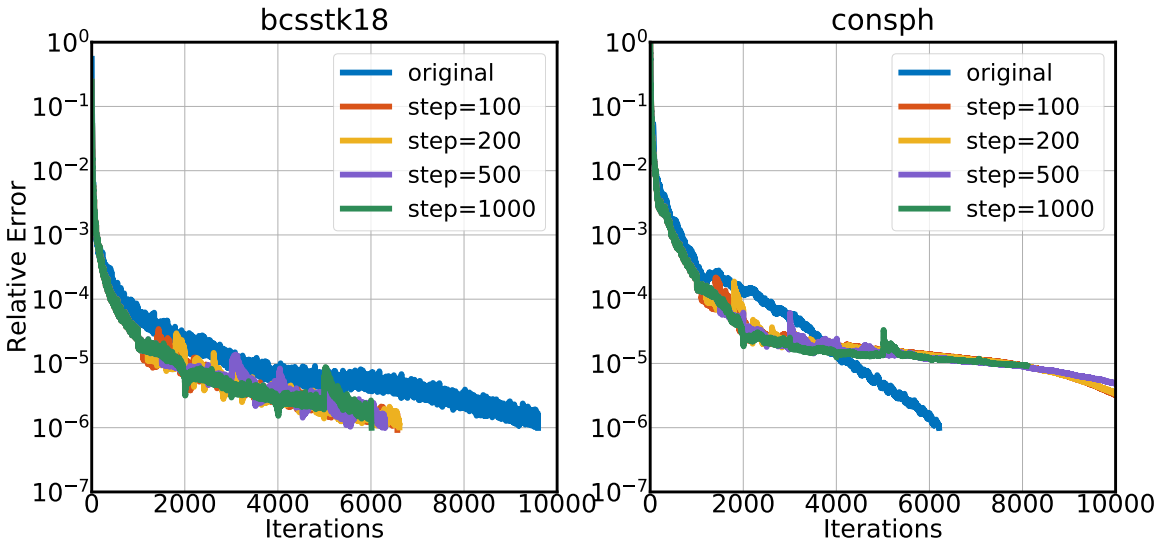


Figure 2.10.: Convergence of different fault rate on small matrices(K=4)

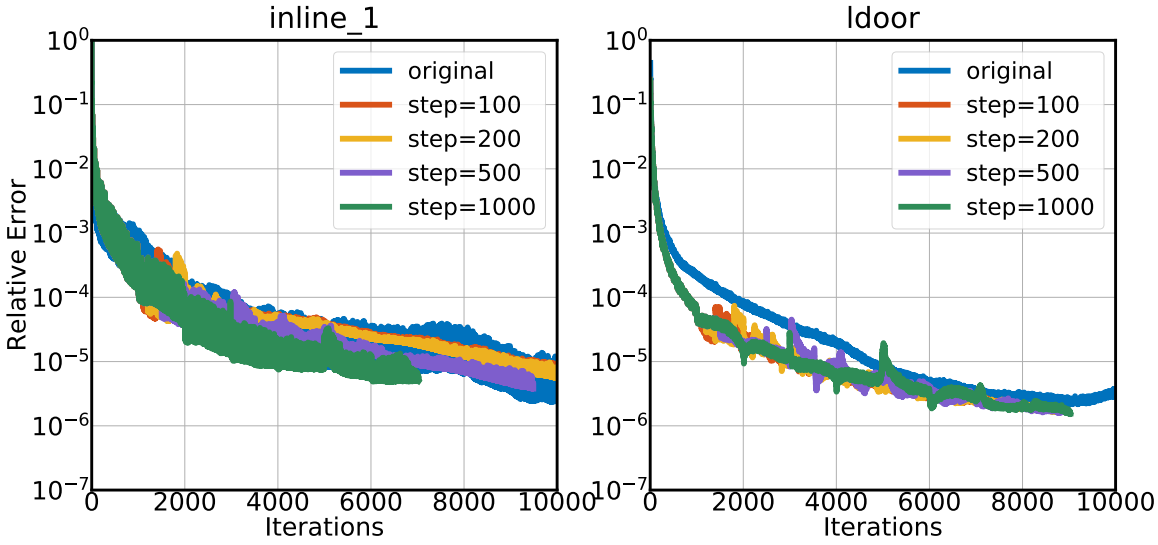


Figure 2.11.: Convergence of different fault rate on large matrices(K=4)

two consecutive faults is large), and vice versa. This can be explained by the fact that as faults happen, we set the search direction of CG as the previous residual. Consequently, the higher the fault rate, the more frequently we change the search direction and the associated Krylov subspace.

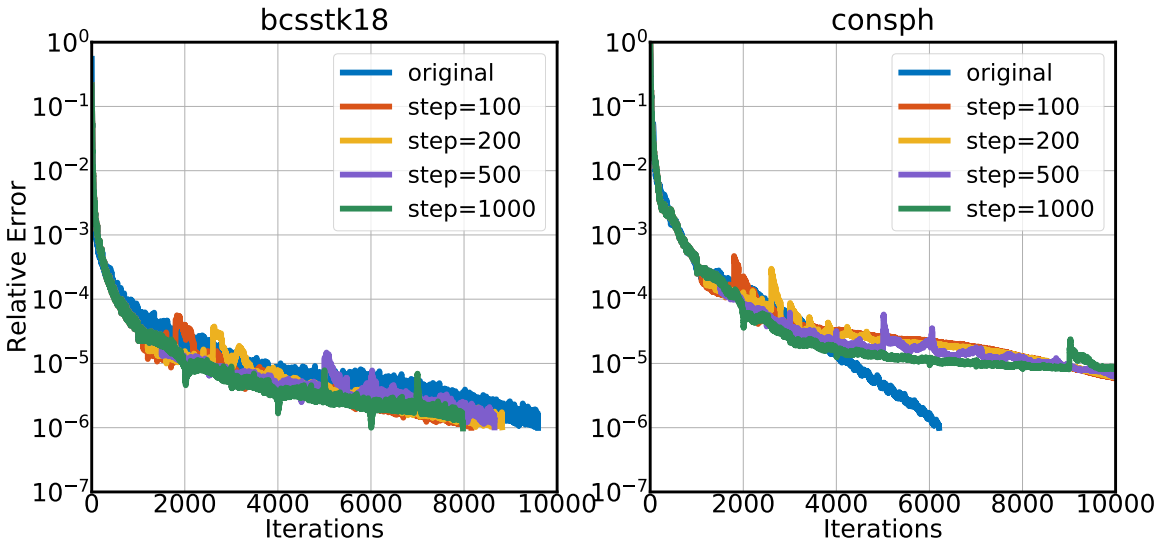


Figure 2.12.: Convergence of different fault rate on small matrices(K=8)

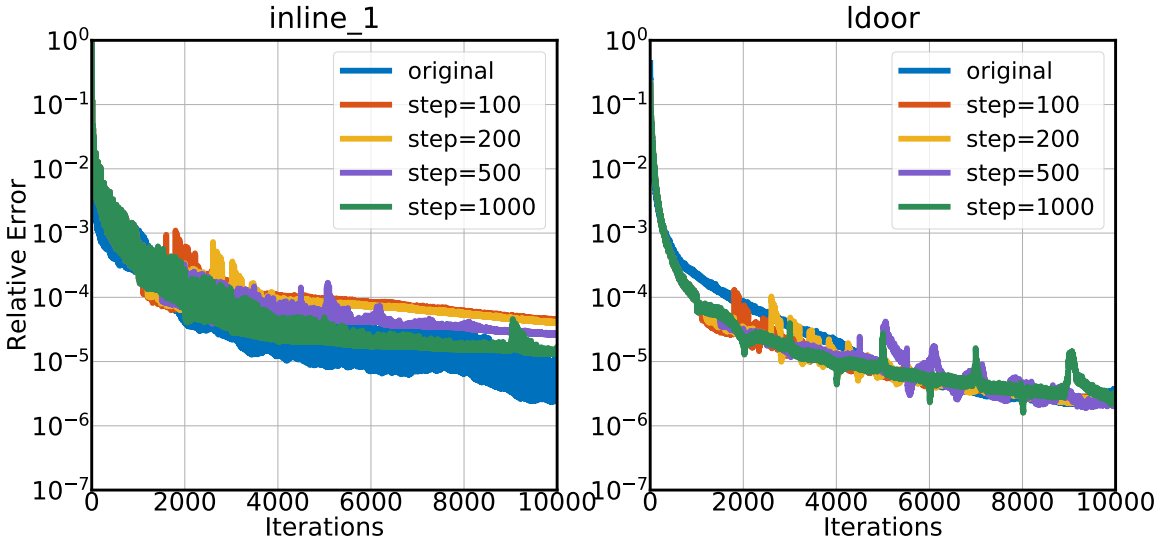


Figure 2.13.: Convergence of different fault rate on large matrices(K=8)

2.5.4 Convergence of Different Fault Arrival Models

In Section 2.5.1 and Section 2.5.3 we present the results from the instant fault arrival model and uniform fault arrival model (faults happen at a fixed rate), respectively. We now investigate alternate fault arrival patterns and evaluate their impact

on the convergence of the solver. We specifically experiment with three fault patterns – instantaneous (all faults happen at the same time), uniform (faults happen at a uniform rate), and random (faults happen at random intervals with a defined mean and variance). Instantaneous faults happen when partitions of machines fail (or multicore nodes fail), resulting in multiple instantaneous failures. Uniform faults correspond to a single core fault. The exponential distribution is the most commonly used random fault arrival model [49]. It assumes the time to failure to be exponentially distributed. The probability distribution function (PDF) of the time (τ) to failure is given by:

$$P_e(t < \tau) = 1 - e^{-r_e \tau} \tag{2.3}$$

Here r_e is the failure rate. In each of our experiments, faults are initiated at iteration 1000 and the mean of the random fault arrival model (for exponential distribution, the mean of the distribution equals $\frac{1}{\text{rate parameter}}$, hence, the mean number of iterations between two consecutive faults is $\frac{1}{r_e}$) is set as 1000, which means the fault rate = $\frac{1}{1000}$. The convergence for different numbers of faults – $K = 4$ and $K = 8$ are tested.

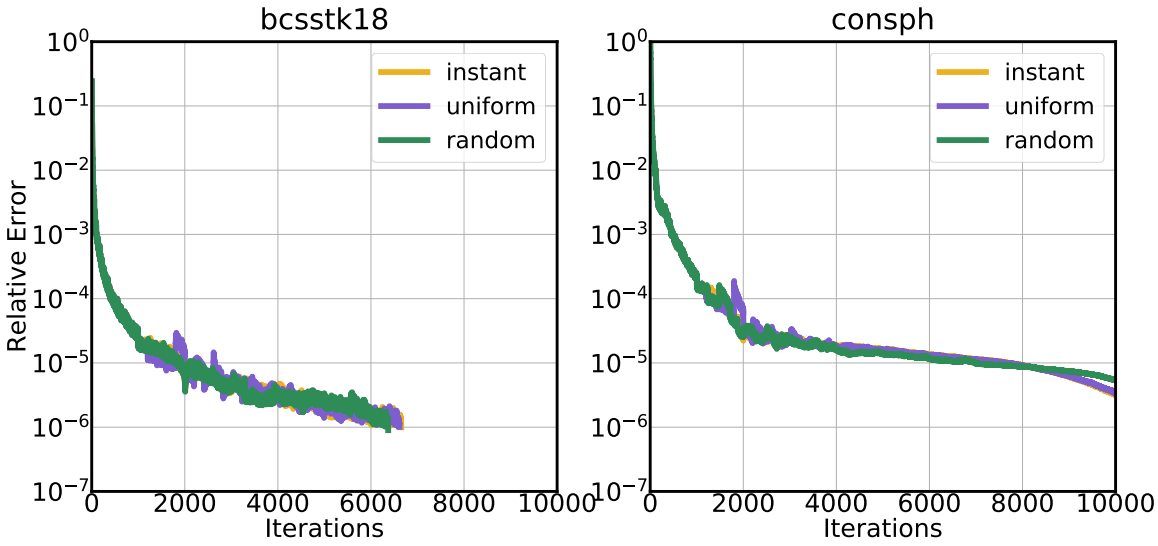


Figure 2.14.: Convergence of different fault arrival models on small matrices(K=4)

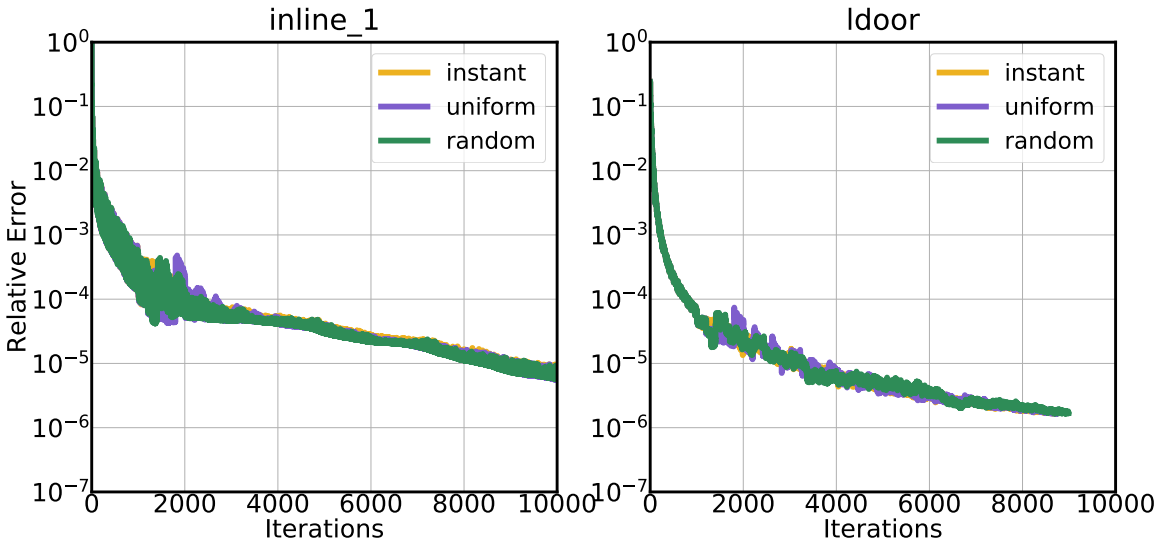


Figure 2.15.: Convergence of different fault arrival models on large matrices(K=4)

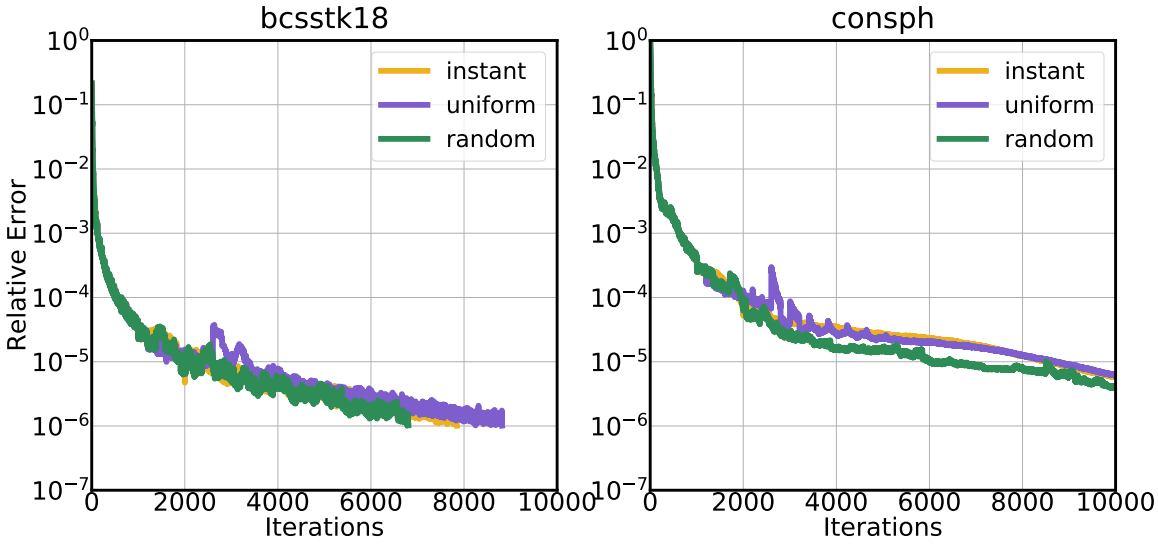


Figure 2.16.: Convergence of different fault arrival models on small matrices(K=8)

Our results in Figure 2.14 to Figure 2.17 show that the best convergence rate is achieved for the random fault arrival model. Under this model, the solver achieves lower relative error or faster convergence rate. The corresponding rate for instantaneous arrivals is close to the random arrival case. These results show that our method is particularly attractive for realistic fault arrival models.

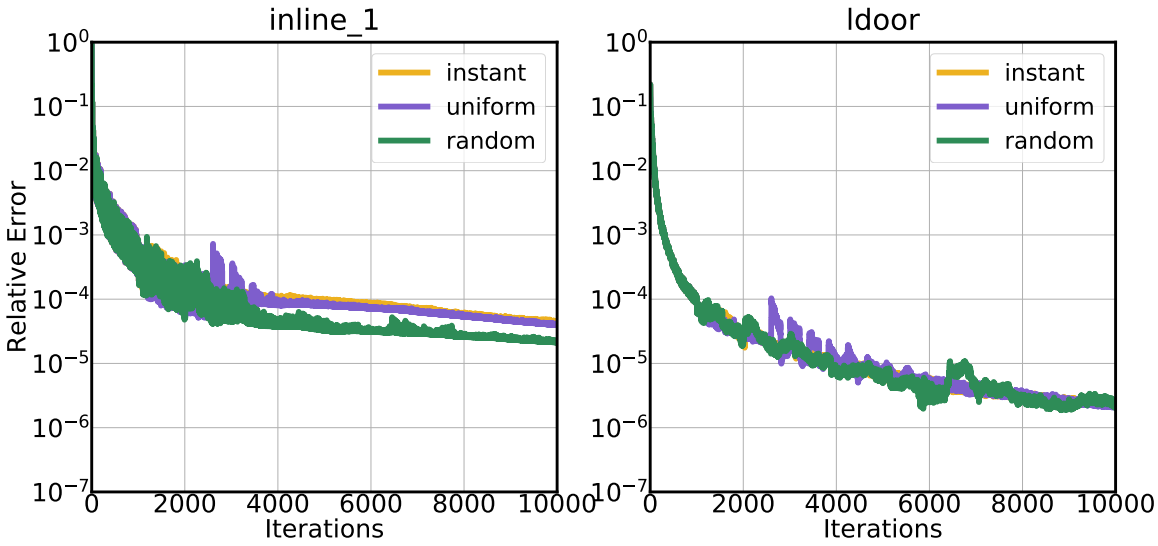


Figure 2.17.: Convergence of different fault arrival models on large matrices(K=8)

2.6 Conclusion

Fault tolerance is an important problem for scalable parallel and distributed systems. We show how to take recently proposed erasure coding computation schemes and adapt them to a scenario suitable for distributed computation. This involves creating a new encoding matrix that satisfies the recovery equations for almost all sets of failing components. We show how to reorder and partition matrices to realize better parallel performance on up to 16 cores. The convergence under a faulty/non-faulty environment shows the effectiveness of our proposed scheme. The high speedup and low time overheads prove the applicability of our scheme on large distributed systems. Performance with different fault arrival models and fault arrival rates further show the robustness of our proposed scheme. These advantageous properties motivate us to apply this scheme to other problems such as eigenvalue problems, graph analyses, and other machine learning kernels.

3 ADAPTIVE FAULT TOLERANT LINEAR SYSTEM SOLVER

Our past work estimates maximum number of faults and augments the system prior to execution for this worst case estimate. The overhead of this augmentation is paid in the solver, even if fewer faults are encountered. In this chapter, we significantly improve the performance by developing an adaptive augmentation technique. This technique augments the input matrix only when faults are encountered. As a result, the system dimension remains n (the dimension of the input matrix) throughout the execution. We demonstrate a range of desirable features of our adaptive augmentation technique in this chapter.

We call this method as *Adaptive Erasure Coded Computation*(AECC). AECC coded rows and columns are added only when faults are detected. This leads to 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 in 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 a detailed theoretical framework underlying AECC in Section 3.1, coding and solution reconstruction techniques in Section 3.2; the parallel formulation of the AECC solver and the running time overhead are also shown. We support our theoretical results through a parallel implementation, and validate all of the desirable features of our AECC solver in Section 3.3.

3.1 Adaptive Linear System Solver

In Chapter 2, 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 schemes, from Vandermonde matrices to sparse low density parity codes (LDPC) [16,23]. With matrices $\boldsymbol{A}$ and $\boldsymbol{E}$, the augmented or encoded system $\tilde{\boldsymbol{A}}$, a solution to the augmented system $\tilde{\mathbf{x}}$, and the augmented right-hand-side $\tilde{\mathbf{b}}$, are given by:

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

Chapter 2 demonstrates that erasure coded linear system solvers with suitably chosen coding matrices are significantly more efficient than state of the art fault tolerance techniques, particularly when the number of faults is large. Note that the solver operates on the augmented system of size $(n + k) \times (n + k)$. For large values of k (large distributed systems or high fault rate systems), the computation overhead can be significant, particularly when the augmentation blocks have significant number of fills. 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. For linear system with sparse left-hand-side, the computation overheads produced by the large augmentation blocks may be a dominating factor of the whole computation.

The underlying principle of our new adaptive solver is to add redundant blocks into the matrix only as errors are encountered. The solver only ever operates on $n \times n$ matrices, which are guaranteed to be symmetric positive definite (SPD) if the input matrix is symmetric positive definite. Redundant blocks ($\boldsymbol{AE}$,$\boldsymbol{E}^T\boldsymbol{A}$,$\boldsymbol{E}^T\boldsymbol{AE}$) are precomputed and stored, but only utilized in the event of faults. We now describe our adaptive fault tolerant solver built using a conjugate gradient solver.

We assume that the original matrix $\boldsymbol{A}$ and right-hand-side vector $\mathbf{b}$ 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 stated, we assume a fail-stop failure model (other fault models can be reduced to fail-stop models through predicated checks to detect failures), in which 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 $[\boldsymbol{E}^T\boldsymbol{A}, \boldsymbol{E}^T\boldsymbol{A}\boldsymbol{E}]$. 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.

We describe the overall algorithm/method here. Without loss of generality, we permute the system with some permutation matrix such that the erased rows correspond to a block:

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

where x_c is the correct (faulty-free) part of solution vector with size $(n-k) \times 1$, while x_f corresponds to the faulty part with size $k \times 1$. When we lose elements to erasures, we lose the rows associated with ${\boldsymbol{A}_{12}}^T$ and $\boldsymbol{A}_{22}$ along with the elements of $\mathbf{b}_f$. We also assume that other processors have cached the most recent values from $\mathbf{x}_f$ so that this information is not lost. We assume that the precomputed matrices $\boldsymbol{Z}_1, \boldsymbol{Z}_2, \boldsymbol{E}^T\boldsymbol{A}\boldsymbol{E}$ and vector $\boldsymbol{E}^T\mathbf{b}$ involved in block partitioned form of the augmented system (2.2) are available:

$$\begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} & \boldsymbol{Z}_1 \\ {\boldsymbol{A}_{12}}^T & \boldsymbol{A}_{22} & \boldsymbol{Z}_2 \\ {\boldsymbol{Z}_1}^T & {\boldsymbol{Z}_2}^T & \boldsymbol{E}^T\boldsymbol{A}\boldsymbol{E} \end{bmatrix} \begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_f \\ \mathbf{x}_r \end{bmatrix} = \begin{bmatrix} \mathbf{b}_c \\ \mathbf{b}_f \\ \boldsymbol{E}^T b \end{bmatrix}$$

Here, $\begin{cases} \boldsymbol{Z}_1 = \boldsymbol{A}_{11}\boldsymbol{E}_1 + \boldsymbol{A}_{12}\boldsymbol{E}_2 \\ \boldsymbol{Z}_2 = \boldsymbol{A}_{12}^T\boldsymbol{E}_1 + \boldsymbol{A}_{22}\boldsymbol{E}_2 \end{cases}$ and $\boldsymbol{E} = \begin{bmatrix} \boldsymbol{E}_1 \\ \boldsymbol{E}_2 \end{bmatrix}$. We never form this matrix explicitly, however. The size of this augmented system is $(n+k) \times (n+k)$, which will let us handle up to k erasures.

When an erasure occurs, then $m \leq k$ rows of the system are lost. We select an arbitrary set of m rows-columns from the precomputed coding data corresponding to an $n \times m$ matrix $\tilde{\boldsymbol{E}}$. (In the event of a sequence of erasures, then a column can only be selected once.) Let $\tilde{\boldsymbol{Z}}_1, \tilde{\boldsymbol{Z}}_2, \tilde{\boldsymbol{E}}^T\boldsymbol{A}\tilde{\boldsymbol{E}}^T, \tilde{\boldsymbol{E}}^T\mathbf{b}$ be the data selected. Then we form and solve the new system:

$$\begin{bmatrix} \boldsymbol{A}_{11} & \tilde{\boldsymbol{Z}}_1 \\ \tilde{\boldsymbol{Z}}_1^T & \tilde{\boldsymbol{E}}^T\boldsymbol{A}\tilde{\boldsymbol{E}} \end{bmatrix} \begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_r \end{bmatrix} = \begin{bmatrix} \mathbf{b}_c \\ \tilde{\boldsymbol{E}}^T\mathbf{b} \end{bmatrix} - \begin{bmatrix} \boldsymbol{A}_{12} \\ \tilde{\boldsymbol{Z}}_2^T \end{bmatrix} \mathbf{x}_f. \tag{3.1}$$

Given solution of this problem, we can recreate the true solution as $\mathbf{x}^* = \mathbf{x}_c + \boldsymbol{E} * \mathbf{x}_r$ [18].

The full methodology and algorithm is described in Algorithm 2. Note that in Algorithm 2 above, Steps 3 to 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 vector $\mathbf{b}$ to account for the fault. To compute $\mathbf{b}^{(\text{cur})}$, we use the saved vector $\mathbf{x}_f$. 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 lookup 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 $\boldsymbol{A}$ 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.

Algorithm 2 Adaptive Fault Oblivious CG.

1: (Reliably) Compute and save the entries $\boldsymbol{Z}_1, \boldsymbol{Z}_2, \boldsymbol{E}^T\boldsymbol{A}\boldsymbol{E}, \boldsymbol{E}^T\mathbf{b}$ for an $n \times k$ coding matrix $\boldsymbol{E}$.

2: Let $\boldsymbol{A}^{(\text{cur})} = \boldsymbol{A}$ and $\mathbf{b}^{(\text{cur})} = \mathbf{b}$

3: Let $\mathbf{x}_0$ be the initial guess, $\mathbf{r}_0 = \mathbf{b}^{(\text{cur})} - \boldsymbol{A}^{(\text{cur})}\mathbf{x}_0$, and $\beta_0 = 0$.

4: **for** $t = 1, \ldots$ until convergence **do**

5: $$\mathbf{p}_t = \begin{cases} \mathbf{r}_{t-1} & t = 1 \text{ or after a fault} \\ \mathbf{r}_{t-1} + \frac{\langle \mathbf{r}_{t-1}, \mathbf{r}_{t-1} \rangle}{\langle \mathbf{r}_{t-2}, \mathbf{r}_{t-2} \rangle} \mathbf{p}_{t-1} & t > 1 \end{cases}$$

6: $\mathbf{q}_t = \boldsymbol{A}^{(\text{cur})}\mathbf{p}_t$

7: $\alpha_t = \langle \mathbf{r}_{t-1}, \mathbf{r}_{t-1} \rangle / \langle \mathbf{q}_t, \mathbf{p}_t \rangle$

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 $\tilde{\boldsymbol{Z}}_1, \tilde{\boldsymbol{Z}}_2, \tilde{\boldsymbol{E}}^T\boldsymbol{A}\tilde{\boldsymbol{E}}, \tilde{\boldsymbol{E}}^T\mathbf{b}$ correspond the columns used.

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

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

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

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

11: **end for**

3.2 Coding Matrix

The structure and numerical properties of the coding matrix $\boldsymbol{E}$ have significant impact on the performance of our method. The simplest coding matrix, one that is often used in redundant storage, is a Vandermonde matrix. However, this matrix is dense, leading to dense augmentation blocks. This increases both FLOP counts, as well as communication overhead in computing the matrix vector products in CG. Furthermore, Vandermonde matrices are ill-conditioned, and they significantly degrade convergence of the augmented system. An alternate choice for a coding matrix is a random dense matrix. It is known that a random dense matrix has full rank with probability 1. Such matrices also have the adverse effect of making the augmentation blocks dense.

An alternate coding structure, also used in Chapter 2 is to use a structured sparse matrix. We construct a matrix $\boldsymbol{E}$ by setting s successive elements in each row to non-zero elements. These s non-zero elements are selected in a staggered manner, i.e., the first s elements in Row 1, one zero followed by s non-zero elements in Row 2, two zeros, followed by s non-zero elements in Row 3, and so on. More generally, for the i^{th} row, the $j = (i - 1 + v) \mod k$ elements for v ranging from 1 to s are set to random reals in the range $(0, 1)$. This matrix is illustrated in Figure 3.1(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 3.1 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 3.1(c), we show how parts of the augmentation blocks are swapped in, to compensate for erasures. Finally, Figure 3.1(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

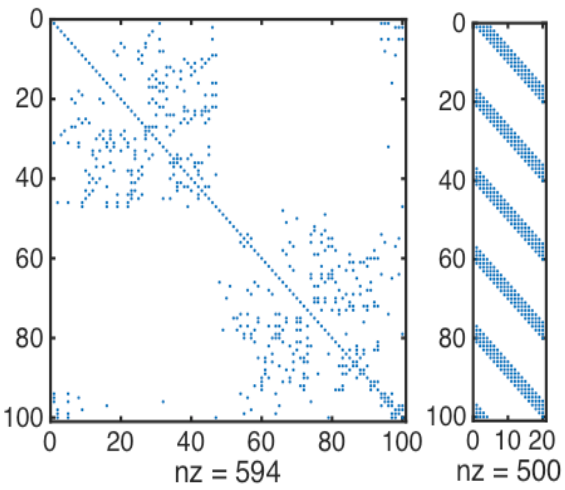
detected. If the coding blocks are dense, we can select arbitrary columns from $\boldsymbol{AE}$, corresponding to columns of the $\boldsymbol{Z}$ matrices from the last section, to compensate for erasures. However, if the coding blocks are sparse, we cannot arbitrarily select columns from the coding 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 $\boldsymbol{E}_{i,j}$. This is easily done, since by assumption, all processors are aware of the indicies of erased elements and the elements of $\boldsymbol{E}$ are structured.

Again, let us reiterate that we do not add the erased row itself because then we would have to maintain multiple copies of the matrix (one more than the number of node failures we wish to tolerate) in order 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. Since solvers typically operate at the limit of memory (or a meaningful fraction thereof) for scaling, it is unrealistic to assume any significant replication for tolerating faults.

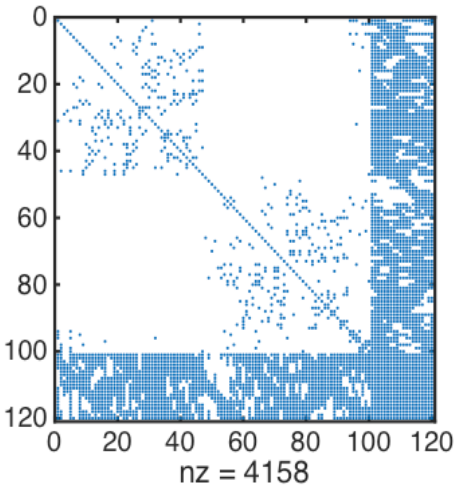
3.3 Experimental Validation

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) the adaptive fault tolerant linear solver converges to solution at low relative residual; (ii) the number of iterations and time of our fault tolerant solver do not increase much, compared to state of the art techniques; and (iii) the parallel communications induced by our augmentation rows is small – leading to highly efficient and scalable performance.

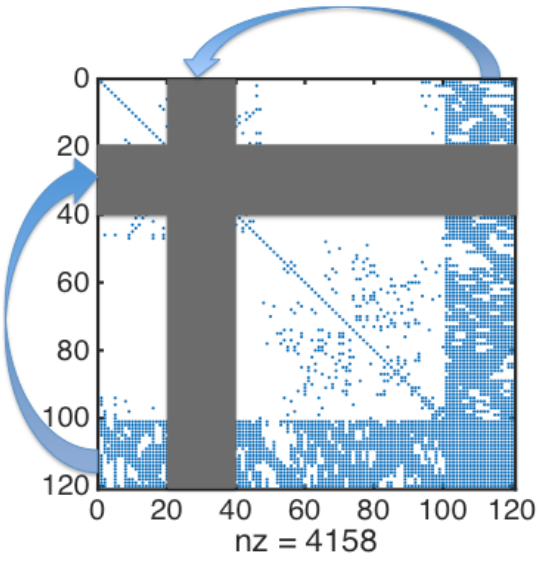
We select a few matrices from the University of Florida Matrix Collection for our experiments – `cbuckle` and `gyro_m` are used to test the convergence of adaptive fault



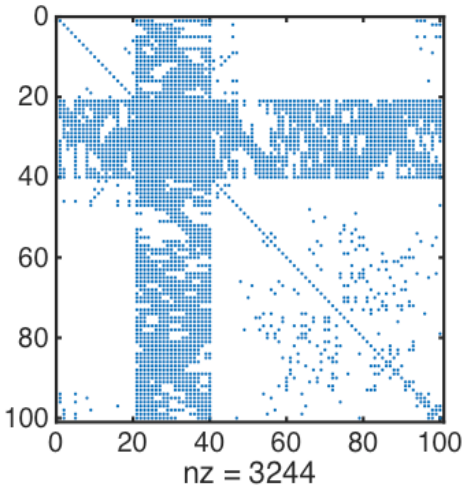
(a) Input matrix $\boldsymbol{A}$ (left) and Coding matrix $\boldsymbol{E}$ (right)



(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

Figure 3.1.: Illustration of the coding and compensation process for erasures.

tolerant linear solver; `consph` and `ldoor` are used to evalute the parallel scalability and robustness under different fault arrival models. The matrix sizes and sparsities are shown in Table 3.1.

Table 3.1.: Matrices Used in Testing.

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

All of the test matrices are symmetric positive definite (SPD). Hence, both CG on the original and augmented system converge on these matrices. The right-hand-side vector $\mathbf{b}$ 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 norm $\frac{||\boldsymbol{A}\mathbf{x}-\mathbf{b}||_2}{||\mathbf{b}||_2}$ is equal to the absolute residual norm $||\mathbf{r}||_2 = ||\boldsymbol{A}\mathbf{x} - \mathbf{b}||_2$ for our problems. During the 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 10000. 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 [46]. To construct the augmented system, we precompute 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. An exponential distribution is the most commonly used fault arrival model [49] as shown in Eq. 2.3. 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 (r_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 $\frac{1}{r_e}$. This means the average number of faults in `orig_iter` iterations is 1, 2, and 3 for $r_e = \frac{1}{\texttt{orig\_iter}}, \frac{2}{\texttt{orig\_iter}}$ 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 $\frac{\texttt{orig\_iter}}{1+\texttt{orig\_iter}/r_e}$. This is to ensure that the fault process starts. Otherwise, in some runs, there are no faults at all for the exponential fault process.

3.3.1 Convergence Rates

Our first set of experiments focus 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.

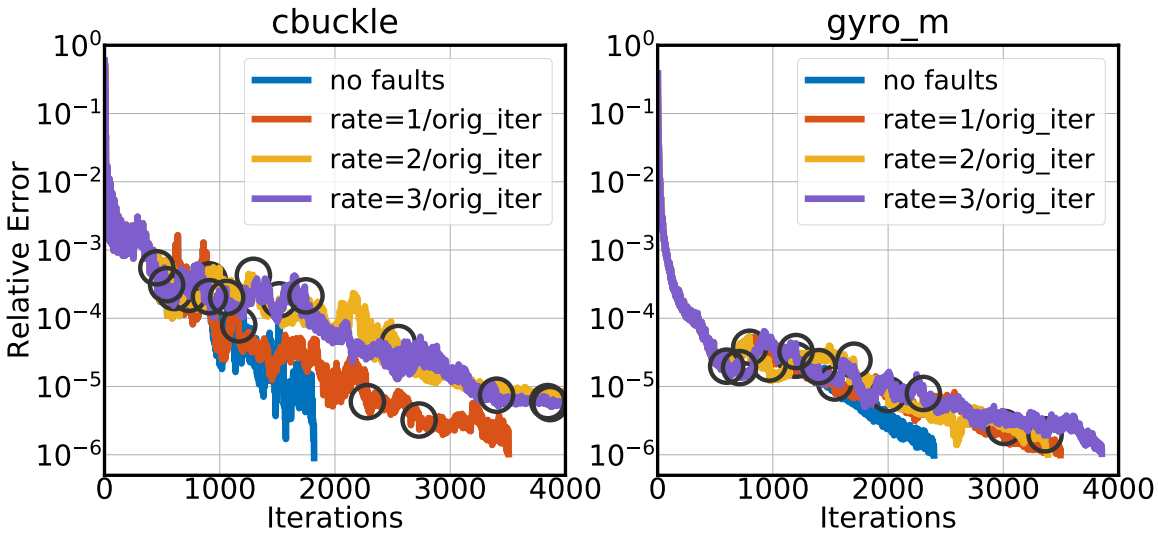


Figure 3.2.: Convergence Rate of `cbuckle` and `gyro_m` with different fault rates.

Figure 3.2 shows the convergence rate of the adaptive fault tolerant linear solver with different fault rates. For the test matrices, `cbuckle` and `gyro_m`, we observe

that our solver can reach the same relative error as 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 that of using comparable fault tolerance techniques based on replicated execution or deterministic replay.

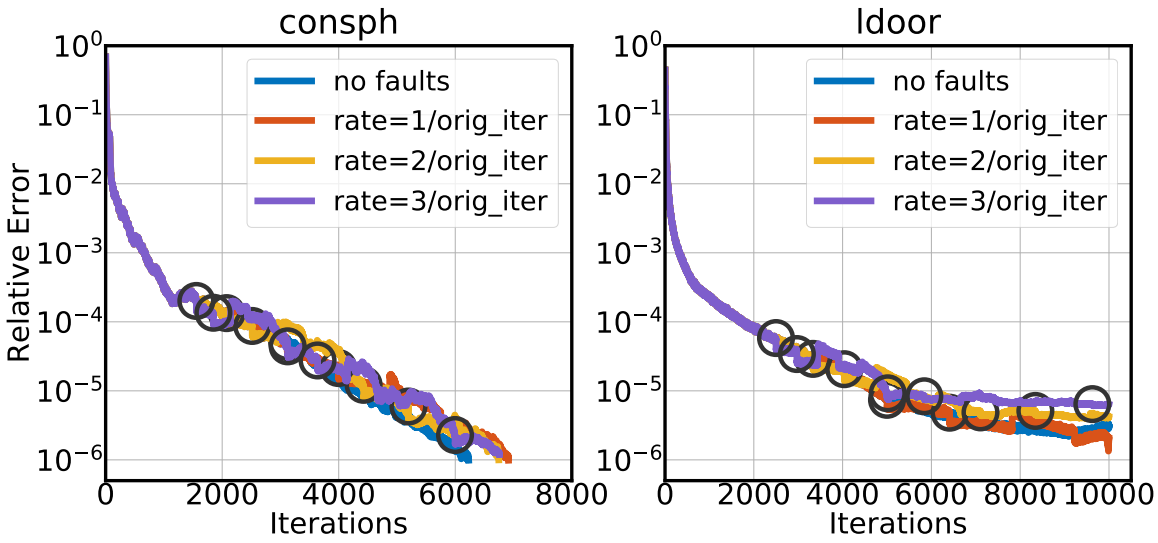


Figure 3.3.: Convergence Rate for `consph` and `ldoor` with different fault rates.

Figure 3.3 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 convergences of the faulty and no-fault cases are very close with each other – indicating that using the system after it has been augmented does not impact solver convergence adversely.

3.3.2 Parallel Performance

We now evalute 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(R) Xeon(R) Platinum 8168 processor operating at 2.70GHz. We use

MPI to implement our solver. We simulate faults in the system by inducing a selected number of erasures. Processes communicate via 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 – exponential fault model and instantaneous fault model. Upon detecting a fault, the solver updates the erased rows, and continues with the solve, as described in our Algorithm 2. 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). Here, parallel speedup is defined as the ratio of the time taken by one processor (to convergence or to reach the 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=$\frac{2}{\texttt{orig\_iter}}$ for all matrices. Figure 3.4 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 introduce negligible parallel overhead (over and above the base solver).

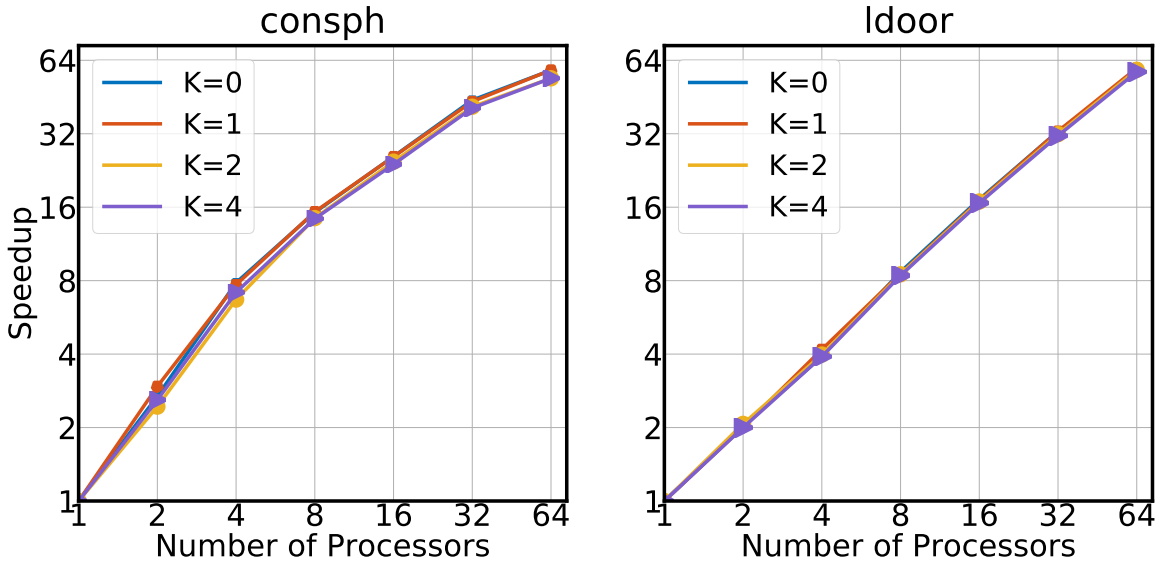


Figure 3.4.: Parallel Performance of `consph` and `ldoor` for exponential fault model

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 fault rate=$\frac{2}{\texttt{orig\_iter}}$.

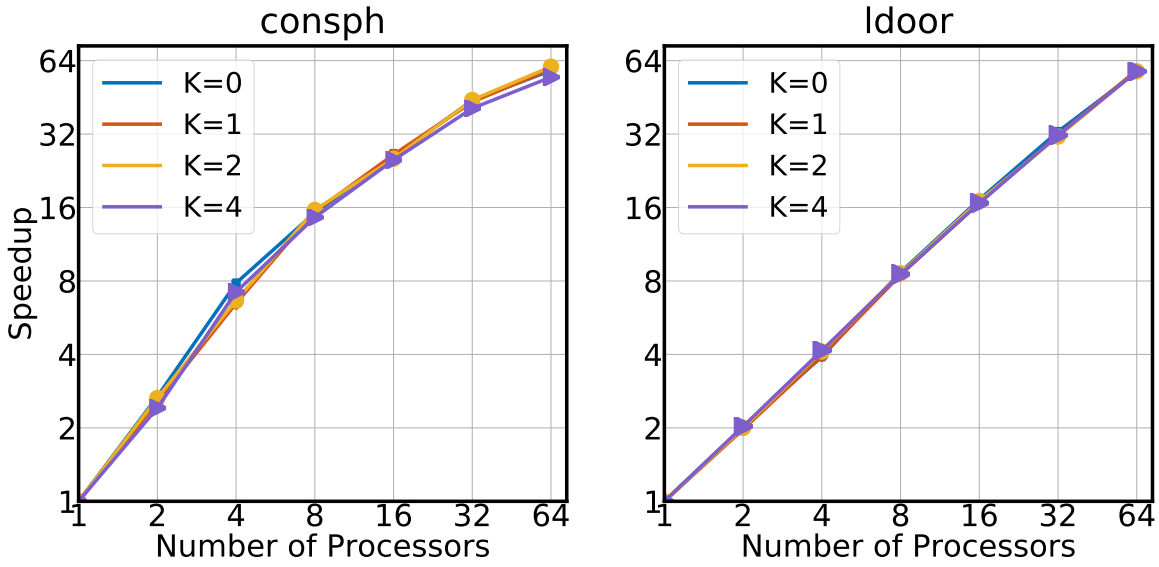


Figure 3.5.: Parallel performance of `consph` and `ldoor` for instantaneous fault model.

Figure 3.5 shows the speedup of 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..

3.3.3 Time Overheads

We finally 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 solution time of the augmented system with faults to that of the original system with no faults. As stated before, solution time corresponds to either the time to convergence or that to reach the iteration bound. Since we are interested in quantifying the computation 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.

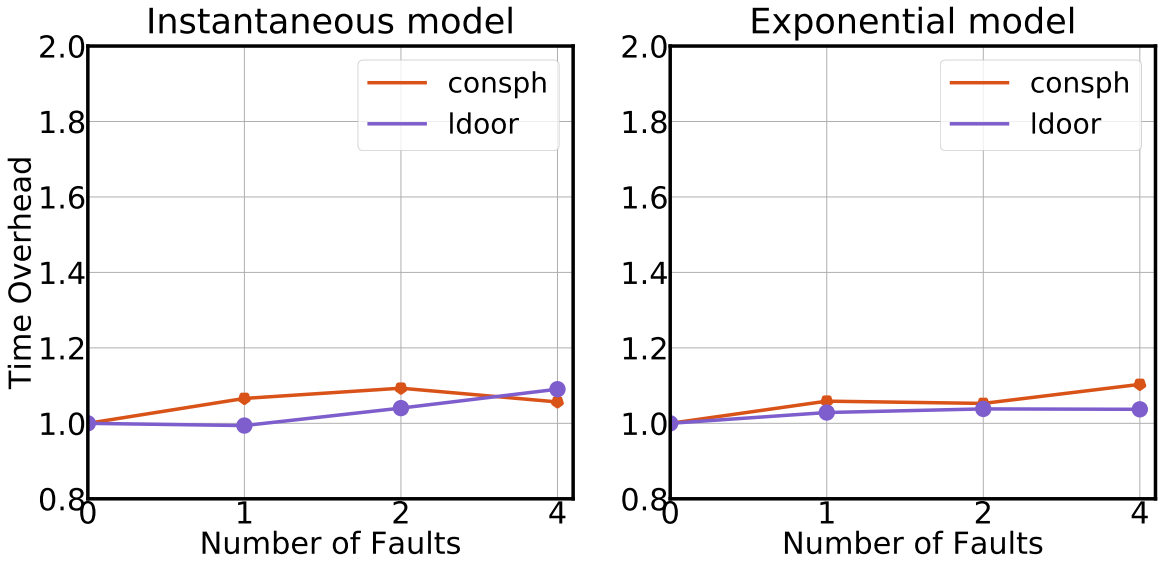


Figure 3.6.: Time overhead with different number of faults under different fault models.

Figure 3.6 shows the time overhead of our solver for different numbers of faults. As expected, With increasing number of faults, more time is needed to converge. This time overhead comes from two factors – the impact of denser coding rows in the augmentation blocks, and 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. Compared to the results in Figure 2.8 (matrix `consph`), the time overhead for $K = 4$ decreased from 1.5 to less than 1.1. This proves the effectiveness of reducing computation using *Adaptive Erasure Coded Computations*(AECC). For different fault arrival model where the fault happens one after another, the time overhead is similar to that of instantaneous fault arrival model. This demonstrates the applicability of our solver to real practice large distributed systems.

3.4 Conclusion

In this chapter, we presented an adaptive fault tolerant linear system solver capable of scaling to large number of processors and associated faults. The solver works by augmenting the input matrix by erasure coded blocks when faults are detected. We derived constraints on coding blocks, and presented sparse coding techniques that satisfy these constraints with high probability. Our proposed 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 processes. We evaluted the effect of fault rates and fault arrival patterns (instantaneous versus exponential) and showed that our scheme is robust to a wide range of fault characteristics.

4 ERASURE CODED FAULT TOLERANT EIGENVALUE SOLVER

In this chapter, we present a novel erasure coded computation scheme for solving eigenvalue problem, $\boldsymbol{A}\mathbf{x} = \lambda\mathbf{x}$. Unlike linear systems, naively adding a coding block to a given matrix $\boldsymbol{A}$ changes its eigenvalues, and there are no known computationally inexpensive ways of recovering the original eigenvalues from these perturbed eigenvalues. To address this, we transform the original eigenvalue problem to an equivalent (in terms of eigenvalues) generalized eigenvalue problem $\tilde{\boldsymbol{A}}\tilde{\mathbf{x}} = \lambda\tilde{\boldsymbol{B}}\tilde{\mathbf{x}}$, where $\tilde{\boldsymbol{A}}$ is an augmented form of input matrix $\boldsymbol{A}$ and $\tilde{\boldsymbol{B}}$ is a suitably constructed sparse matrix. We present detailed proofs of the equivalence of eigenvalues of the original problem and the new generalized eigenvalue problem. We characterize the impact of faults on the solution of the generalized eigenvalue problem, and present the procedure to recover the eigenvalues of the original problem in the event of faults.

We solve the reformulated generalized eigenvalue problem using TraceMin [36, 37] and show that our solver has excellent convergence properties (eigenvalues are very close to those of the original system) in the event of random faults (a set of randomly chosen row-column pairs are erased). We also evaluate the performance of our eigensolver in worst case scenarios. It is well known that, different rows of a matrix have different impact on the eigenspectrum of the matrix [38,39]. One way to characterize the importance of a row-column pair with respect to its eigenspectrum is using leverage scores [40–42]. We construct worst case scenarios for our solver by erasing row-column pairs with the top k highest leverage scores. In this case, we note that our fault tolerance scheme has increased overheads in terms of convergence rate. This is due to the fact that our coding scheme is oblivious to leverage scores. Since it is infeasible to compute leverage scores a-priori in order to construct our coding blocks, we present a novel adaptive scheme, where leverage scores are estimated from the eigenvectors as the TraceMin computation proceeds, and the code is suitably

updated. We demonstrate that using this adaptive scheme, we can handle worst case scenarios with minimal convergence overheads.

The rest of this chapter is organized as follows. We present our erasure coded eigensolver in Section 4.1, along with details of the coding matrix and transformed generalized eigenvalue system, and the properties of the generalized eigenvalue system, and prove the equivalence of the original system and the augmented (generalized) system. We present the implementation details in Section 4.2, along with details of the perturbation and purification strategy. In Section 4.3, we present our experimental results on convergence rate and robustness including (i) average case convergence rate of base algorithm; (ii) worst case convergence properties of the base algorithm and the adaptive coding technique; (iii) the impact of using approximations of leverage scores (as opposed to exact leverage scores) to establish bounds; and (iv) the impact of different fault arrival models (instantaneous v.s. random arrivals). We draw conclusions and summarize our contributions in Section 4.4.

4.1 Formulating the Erasure Coded Eigensolver

Given an eigenvalue problem:

$$\boldsymbol{A}\mathbf{x} = \lambda\mathbf{x} \tag{4.1}$$

where $\boldsymbol{A}$ is an $n \times n$ matrix, λ is the eigenvalue, and $\mathbf{x}$ is the corresponding eigenvector with size $n \times 1$, we aim to formulate an erasure coded eigenvalue problem such that the solution is computed either directly, or through an inexpensive computation, yields the dominant eigenvalues of the given problem (Eq. 4.1). Eq. 4.1 can also be written as $\boldsymbol{A}\mathbf{x} = \lambda\boldsymbol{B}\mathbf{x}$ where $\boldsymbol{B}$ equals to the identity matrix $\boldsymbol{I}_{n\times n}$. The erasure coded formulation must be capable of executing in a parallel environment with faults in the form of erasures (corresponding to fail-stop failures in parallel platforms).

Our solution reformulates the given problem as the following generalized eigenvalue problem:

$$\tilde{\boldsymbol{A}}\tilde{\mathbf{x}} = \lambda\tilde{\boldsymbol{B}}\tilde{\mathbf{x}}, \tag{4.2}$$

where

$$\tilde{\boldsymbol{A}} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{AE} \\ \boldsymbol{E}^T\boldsymbol{A} & \boldsymbol{E}^T\boldsymbol{AE} \end{bmatrix} \qquad \tilde{\boldsymbol{B}} = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{E} \\ \boldsymbol{E}^T & \boldsymbol{E}^T\boldsymbol{E} \end{bmatrix} \qquad \tilde{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ \mathbf{r} \end{bmatrix}$$

Here, $\boldsymbol{E}$ is the coding matrix of dimension $n \times k$, x and r are $n \times 1$ vector and $k \times 1$ vector, respectively. The coding matrix $\boldsymbol{E}$ must satisfy certain conditions with respect to its Kruskal Rank [29] and must be as sparse as possible to minimize fills in the augmentation blocks (the (1,2), (2,1), and (2,2) blocks in matrices $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{B}}$). We describe these later in this chapter.

Multiplying the row block of Eq. 4.2, we have

$$\boldsymbol{A}(\mathbf{x} + \boldsymbol{E}\mathbf{r}) = \lambda(\mathbf{x} + \boldsymbol{E}\mathbf{r}) \tag{4.3}$$

and

$$\boldsymbol{E}^T\boldsymbol{A}(\mathbf{x} + \boldsymbol{E}\mathbf{r}) = \lambda\boldsymbol{E}^T(\mathbf{x} + \boldsymbol{E}\mathbf{r}) \tag{4.4}$$

Let $\mathbf{v} \equiv \mathbf{x} + \boldsymbol{E}\mathbf{r}$. Then Eq. 4.3 becomes $\boldsymbol{A}\mathbf{v} = \lambda\mathbf{v}$, which is the standard eigenvalue problem (Eq. 4.1). Here $\mathbf{v}$ is an eigenvector of $\boldsymbol{A}$ and λ is the corresponding eigenvalue. The generalized eigenvalue system has the same eigenvalues as the original eigenvalue system. However, as we show in the next section, it can recover the true eigenvector of Eq. 4.1 when faults happen during the execution. We formalize this in the following theorem:

Theorem 4.1.1 *Given eigenvalue problem $\boldsymbol{A}\mathbf{x}^\star = \lambda\mathbf{x}^\star$, we can construct the generalized eigenvalue system $\tilde{\boldsymbol{A}}\tilde{\mathbf{x}} = \lambda\tilde{\boldsymbol{B}}\tilde{\mathbf{x}}$ such that $\mathbf{x}^\star = \mathbf{x} + \boldsymbol{E}\mathbf{r}$, where $\boldsymbol{E}$ is the coding matrix and $\tilde{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ \mathbf{r} \end{bmatrix}$ is the eigenvector of the generalized eigenvalue system.*

We will prove the theorem based on the properties of the generalized eigenvalue system in Section 4.1.1 and equivalence of original eigenvalue system and generalized eigenvalue system in Section 4.1.2.

4.1.1 Properties of Generalized Eigenvalue System

We state and prove a number of important properties of the generalized eigenvalue system (Eq. 4.2). These properties provide the basis for the solution recovery method in faulty execution environments. They are also useful in characterizing failure modes and motivate efficient algorithms for recovery. We provide details on implementation strategies and algorithms in Section 4.2.

Theorem 4.1.2 *The generalized eigenvalue system* $\tilde{\boldsymbol{A}}$ *and* $\tilde{\boldsymbol{B}}$ *has null space* $\begin{bmatrix} \boldsymbol{E} \\ -\boldsymbol{I}_k \end{bmatrix}$.

Proof: From construction of the generalized system, $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{B}}$ have rank n. Therefore, the null space must have rank k. The matrix $\begin{bmatrix} \boldsymbol{E} \\ -\boldsymbol{I}_k \end{bmatrix}$ has rank k. Furthermore, we have

$$\tilde{\boldsymbol{A}} \begin{bmatrix} \boldsymbol{E} \\ -\boldsymbol{I}_k \end{bmatrix} = 0, \qquad \textit{and} \qquad \tilde{\boldsymbol{B}} \begin{bmatrix} \boldsymbol{E} \\ -\boldsymbol{I}_k \end{bmatrix} = 0.$$

It therefore follows that $\begin{bmatrix} \boldsymbol{E} \\ -\boldsymbol{I}_k \end{bmatrix}$ is the null space for $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{B}}$. ■

Theorem 4.1.3 *If* $\begin{bmatrix} \mathbf{y} \\ \mathbf{z} \end{bmatrix}$ *is a solution to the generalized eigenvalue system in Eq. 4.2, the component* $\mathbf{y}$ *is uniquely determined once* $\mathbf{z}$ *is specified. Furthermore, if* $\mathbf{z} = 0$, *then* $\mathbf{y} = \mathbf{x}^*$.

Proof: It is obvious that $\begin{bmatrix} \mathbf{x}^* \\ 0 \end{bmatrix}$ is a solution for the generalized eigenvalue system in Eq. 4.2 by substituting it in the equation. As proved in Theorem 4.1.2, $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{B}}$ have null space $\begin{bmatrix} \boldsymbol{E} \\ -\boldsymbol{I}_k \end{bmatrix}$, then any solution of the generalized eigenvalue system in

Eq. 4.2 can be written as:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{z} \end{bmatrix} = \begin{bmatrix} \mathbf{x}^* \\ 0 \end{bmatrix} + \begin{bmatrix} \boldsymbol{E} \\ -\boldsymbol{I}_k \end{bmatrix} \mathbf{a} \tag{4.5}$$

for some unique vector $\mathbf{a}$. The equation holds since:

$$\tilde{\boldsymbol{A}} \begin{bmatrix} \mathbf{y} \\ \mathbf{z} \end{bmatrix} = \tilde{\boldsymbol{A}} \left(\begin{bmatrix} \mathbf{x}^* \\ 0 \end{bmatrix} + \begin{bmatrix} \boldsymbol{E} \\ -\boldsymbol{I}_k \end{bmatrix} \mathbf{a} \right) = \lambda \tilde{\boldsymbol{B}} \left(\begin{bmatrix} \mathbf{x}* \\ 0 \end{bmatrix} + \begin{bmatrix} \boldsymbol{E} \\ -\boldsymbol{I}_k \end{bmatrix} \mathbf{a} \right) = \lambda \tilde{\boldsymbol{B}} \begin{bmatrix} \mathbf{y} \\ \mathbf{z} \end{bmatrix}$$

Therefore, we can let $\mathbf{a} = -\mathbf{z}$, and $\mathbf{y}$ is determined as $\mathbf{y} = \mathbf{x}^* - \boldsymbol{E}\mathbf{z}$, which implies that $\mathbf{x}^* = \mathbf{y} + \boldsymbol{E}\mathbf{z}$. This provides the basis for recovery of solution. If $\mathbf{z} = 0$, then we have $\mathbf{y} = \mathbf{x}^*$. ■

Theorem 4.1.4 *If matrix $\boldsymbol{A}$ is SPD,the generalized eigenvalue system $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{B}}$ is SPSD (Symmetric Positive Semi-Definite).*

Proof: We express Cholesky factorization of $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{B}}$ as follows: if the Cholesky factorization of matrix $\boldsymbol{A}$ is given $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T$, then

$$\tilde{\boldsymbol{A}} = \begin{bmatrix} \boldsymbol{L} \\ \boldsymbol{E}^T \boldsymbol{A} \boldsymbol{L}^{-T} \end{bmatrix} \begin{bmatrix} \boldsymbol{L} \\ \boldsymbol{E}^T \boldsymbol{A} \boldsymbol{L}^{-T} \end{bmatrix}^T$$

and $\tilde{\boldsymbol{B}}$ has the Cholesky factorization:

$$\tilde{\boldsymbol{B}} = \begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{E}^T \end{bmatrix} \begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{E}^T \end{bmatrix}^T$$

For any non-trivial vector $\begin{bmatrix} \mathbf{g}_1 \\ \mathbf{g}_2 \end{bmatrix}$, we have

$$\begin{aligned} \begin{bmatrix} \mathbf{g}_1 \\ \mathbf{g}_2 \end{bmatrix}^T \tilde{\boldsymbol{A}} \begin{bmatrix} \mathbf{g}_1 \\ \mathbf{g}_2 \end{bmatrix} &= \begin{bmatrix} \mathbf{g}_1 \\ \mathbf{g}_2 \end{bmatrix}^T \begin{bmatrix} \boldsymbol{L} \\ \boldsymbol{E}^T \boldsymbol{A} \boldsymbol{L}^{-T} \end{bmatrix} \begin{bmatrix} \boldsymbol{L} \\ \boldsymbol{E}^T \boldsymbol{A} \boldsymbol{L}^{-T} \end{bmatrix}^T \begin{bmatrix} \mathbf{g}_1 \\ \mathbf{g}_2 \end{bmatrix} \\ &= (\mathbf{g}_1^T \boldsymbol{L} + \mathbf{g}_2^T \boldsymbol{E}^T \boldsymbol{A} \boldsymbol{L}^{-T})(\boldsymbol{L}^T \mathbf{g}_1 + \boldsymbol{L}^{-1} \boldsymbol{A} \boldsymbol{E} \mathbf{g}_2) \\ &= \mathbf{g}_1^T \boldsymbol{A} \mathbf{g}_1 + \mathbf{g}_1^T \boldsymbol{A} \boldsymbol{E} \mathbf{g}_2 + \mathbf{g}_2^T \boldsymbol{E}^T \boldsymbol{A} \mathbf{g}_1 + \mathbf{g}_2^T \boldsymbol{E}^T \boldsymbol{A} \boldsymbol{E} \mathbf{g}_2 \\ &= (\mathbf{g}_1 + \boldsymbol{E}\mathbf{g}_2)^T \boldsymbol{A} (\mathbf{g}_1 + \boldsymbol{E}\mathbf{g}_2) \\ &\geq 0 \text{ (The quality holds when } \mathbf{g}_1 + \boldsymbol{E}\mathbf{g}_2 = 0.) \end{aligned} \tag{4.6}$$

Hence, $\tilde{\boldsymbol{A}}$ is symmetric positive semi-definite (SPSD). We can prove $\tilde{\boldsymbol{B}}$ is also SPSD using the similar way. Later we will show how we perturb this SPSD system into a SPD system and use TraceMin for solving the generalized eigenvalue problem. ■

4.1.2 Faulty Execution and Solution Recovery from Generalized Eigenvalue System

Without loss of generality, we permute the input system $\boldsymbol{A}$ and $\boldsymbol{B}$ as $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ {\boldsymbol{A}_{12}}^T & \boldsymbol{A}_{22} \end{bmatrix}$ and $\boldsymbol{B} = \begin{bmatrix} \boldsymbol{B}_{11} & \boldsymbol{B}_{12} \\ {\boldsymbol{B}_{12}}^T & \boldsymbol{B}_{22} \end{bmatrix}$ with some permutation matrix such that the erased rows correspond to the row block $[\boldsymbol{A}_{12}, \boldsymbol{A}_{22}]$ and $[\boldsymbol{B}_{12}, \boldsymbol{B}_{22}]$, then the generalized eigenvalue system can be written as:

$$\begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} & \boldsymbol{Z}_1 \\ {\boldsymbol{A}_{12}}^T & \boldsymbol{A}_{22} & \boldsymbol{Z}_2 \\ {\boldsymbol{Z}_1}^T & {\boldsymbol{Z}_2}^T & \boldsymbol{R} \end{bmatrix} \begin{bmatrix} \mathbf{c} \\ \mathbf{f} \\ \mathbf{r} \end{bmatrix} = \lambda \begin{bmatrix} \boldsymbol{B}_{11} & \boldsymbol{B}_{12} & \boldsymbol{Q}_1 \\ {\boldsymbol{B}_{12}}^T & \boldsymbol{B}_{22} & \boldsymbol{Q}_2 \\ {\boldsymbol{Q}_1}^T & {\boldsymbol{Q}_2}^T & \boldsymbol{S} \end{bmatrix} \begin{bmatrix} \mathbf{c} \\ \mathbf{f} \\ \mathbf{r} \end{bmatrix} \tag{4.7}$$

Here, $\mathbf{c}$ is the correct (fault-free) part of the eigenvector with size $(n-k) \times 1$, $\mathbf{f}$ is the part that encounters faults (erasures, or fail-stop failures) with size $k \times 1$, and $\mathbf{r}$ (size is $k \times 1$) is the redundant part. We can write $\boldsymbol{E}$ as $\begin{bmatrix} \boldsymbol{E}_1 \\ \boldsymbol{E}_2 \end{bmatrix}$, then we have

$$\begin{cases} \boldsymbol{Z}_1 &= \boldsymbol{A}_{11}\boldsymbol{E}_1 + \boldsymbol{A}_{12}\boldsymbol{E}_2 \\ \boldsymbol{Z}_2 &= {\boldsymbol{A}_{12}}^T\boldsymbol{E}_1 + \boldsymbol{A}_{22}\boldsymbol{E}_2 \\ \boldsymbol{R} &= \boldsymbol{E}^T\boldsymbol{A}\boldsymbol{E} \\ \boldsymbol{Q}_1 &= \boldsymbol{B}_{11}\boldsymbol{E}_1 + \boldsymbol{B}_{12}\boldsymbol{E}_2 \\ \boldsymbol{Q}_2 &= {\boldsymbol{Y}_{12}}^T\boldsymbol{E}_1 + \boldsymbol{Y}_{22}\boldsymbol{E}_2 \\ \boldsymbol{S} &= \boldsymbol{E}^T\boldsymbol{B}\boldsymbol{E} \end{cases} \tag{4.8}$$

When faults happen, the generalized eigenvalue system is reduced to the following $n \times n$ system:

$$\begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{Z}_1 \\ {\boldsymbol{Z}_1}^T & \boldsymbol{R} \end{bmatrix} \begin{bmatrix} \mathbf{c} \\ \mathbf{r} \end{bmatrix} = \lambda \begin{bmatrix} \boldsymbol{B}_{11} & \boldsymbol{Q}_1 \\ {\boldsymbol{Q}_1}^T & \boldsymbol{S} \end{bmatrix} \begin{bmatrix} \mathbf{c} \\ \mathbf{r} \end{bmatrix} + \begin{bmatrix} \lambda\boldsymbol{B}_{12} - \boldsymbol{A}_{12} \\ \lambda{\boldsymbol{Q}_2}^T - {\boldsymbol{Z}_2}^T \end{bmatrix} \mathbf{f} \tag{4.9}$$

Following from the results of Gleich et al. [18], the purified system Eq. 4.9 has a solution when the coding matrix $\boldsymbol{E}$ has Kruskal rank k. We also need to show that if $\begin{bmatrix}\mathbf{c}\\ \mathbf{r}\end{bmatrix}$ is the solution of the purified system, then $\begin{bmatrix}\mathbf{c}\\ \mathbf{f}\\ \mathbf{r}\end{bmatrix}$ is the solution of the generalized eigenvalue system.

Theorem 4.1.5 *If* $\begin{bmatrix}\mathbf{c}\\ \mathbf{r}\end{bmatrix}$ *is the solution of the purified system, then* $\begin{bmatrix}\mathbf{c}\\ \mathbf{f}\\ \mathbf{r}\end{bmatrix}$ *is the solution of the generalized eigenvalue system.*

Proof: The purified system gives us

$$\boldsymbol{A}_{11}\mathbf{c} + \boldsymbol{Z}_1\mathbf{r} - \lambda\boldsymbol{B}_{11}\mathbf{c} - \lambda\boldsymbol{Q}_1\mathbf{r} = \lambda\boldsymbol{B}_{12}\mathbf{f} - \lambda\boldsymbol{A}_{12}\mathbf{f} \tag{4.10}$$

$$\boldsymbol{Z}_1{}^T\mathbf{c} + \boldsymbol{R}\mathbf{r} - \lambda\boldsymbol{Q}_1{}^T\mathbf{c} - \lambda\boldsymbol{S}\mathbf{r} = \lambda\boldsymbol{Q}_2{}^T\mathbf{f} - \lambda\boldsymbol{Z}_2{}^T\mathbf{f} \tag{4.11}$$

From equations: Eq. 4.11-$E_1{}^T\times$ Eq. 4.10, we get:

$$\boldsymbol{E}_2{}^T\boldsymbol{A}_{12}\mathbf{c} + \boldsymbol{E}_2{}^T\boldsymbol{Z}_2\mathbf{r} - \lambda\boldsymbol{E}_2{}^T\boldsymbol{B}_{12}\mathbf{c} - \lambda\boldsymbol{E}_2{}^T\boldsymbol{Q}_2\mathbf{r} = \boldsymbol{E}_2{}^T(\lambda\boldsymbol{B}_{22} - \boldsymbol{A}_{22})\mathbf{f} \tag{4.12}$$

Since $\boldsymbol{E}_2$ is $k \times k$ and the kruskal rank of $\boldsymbol{E}$ is k, $\boldsymbol{E}_2$ is rank k and nonsigular. Ee can multiply the pseudo-inverse of $\boldsymbol{E}_2$ to the left side of Eq. 4.12 and get:

$$\boldsymbol{A}_{12}\mathbf{c} + +\boldsymbol{A}_{22}\mathbf{f} + \boldsymbol{Z}_2\mathbf{r} = \lambda\boldsymbol{B}_{12}\mathbf{c} + \lambda\boldsymbol{B}_{22}\mathbf{f} + \lambda\boldsymbol{Q}_2\mathbf{r}, \tag{4.13}$$

which is the second row block of Eq. 4.7. ■

Theorem 4.1.6 *In the event of up to k faults, the eigenvalues of eigensystem in Eq. 4.2 are identical to the eigenvalues of the original system.*

Proof: When faults happen, we will solve the purified system

$$\begin{bmatrix}\boldsymbol{A}_{11} & \boldsymbol{Z}_1\\ \boldsymbol{Z}_1{}^T & \boldsymbol{R}\end{bmatrix}\begin{bmatrix}\mathbf{c}'\\ \mathbf{r}'\end{bmatrix} = \lambda'\begin{bmatrix}\boldsymbol{B}_{11} & \boldsymbol{Q}_1\\ \boldsymbol{Q}_1{}^T & \boldsymbol{S}\end{bmatrix}\begin{bmatrix}\mathbf{c}'\\ \mathbf{r}'\end{bmatrix} \tag{4.14}$$

$\lambda^{'}$ is the eigenvalue for the purified system and $\begin{bmatrix}\mathbf{c}'\\ \mathbf{r}'\end{bmatrix}$ is the eigenvector of corresponding eigenvalue. According Eq. 4.14 and Eq. 4.8, we have the following equations:

$$\boldsymbol{A}_{11}\mathbf{c}' + (\boldsymbol{A}_{11}\boldsymbol{E}_1 + \boldsymbol{A}_{12}\boldsymbol{E}_2)\mathbf{r}' = \lambda^{'}[\boldsymbol{B}_{11}\mathbf{c}' + (\boldsymbol{B}_{11}\boldsymbol{E}_1 + \boldsymbol{B}_{12}\boldsymbol{E}_2)\mathbf{r}'] \tag{4.15}$$

$$(\boldsymbol{E}_1^T\boldsymbol{A}_{11} + \boldsymbol{E}_2^T\boldsymbol{A}_{12}^T)\mathbf{c}' + \boldsymbol{E}^T\boldsymbol{A}\boldsymbol{E}\mathbf{r}' = \lambda^{'}[(\boldsymbol{E}_1^T\boldsymbol{B}_{11} + \boldsymbol{E}_2^T\boldsymbol{B}_{12}^T)\mathbf{c}' + \boldsymbol{E}^T\boldsymbol{B}\boldsymbol{E}\mathbf{r}'] \tag{4.16}$$

Eq. 4.16 - $\boldsymbol{E}_1^T\times$ Eq. 4.15 gives us

$$\begin{aligned}&\boldsymbol{E}_2^T\boldsymbol{A}_{12}^T\mathbf{c}' + \boldsymbol{E}^T\boldsymbol{A}\boldsymbol{E}\mathbf{r}' - (\boldsymbol{E}_1^T\boldsymbol{A}_{11}\boldsymbol{E}_1 + \boldsymbol{E}_1^T\boldsymbol{A}_{12}\boldsymbol{E}_2)\mathbf{r}'\\ =&\lambda^{'}[\boldsymbol{E}_2^T\boldsymbol{B}_{12}^T\mathbf{c}' + \boldsymbol{E}^T\boldsymbol{B}\boldsymbol{E}\mathbf{r}' - (\boldsymbol{E}_1^T\boldsymbol{B}_{11}\boldsymbol{E}_1 + \boldsymbol{E}_1^T\boldsymbol{B}_{12}\boldsymbol{E}_2)\mathbf{r}']\end{aligned}$$

As

$$\begin{cases}\boldsymbol{E}^T\boldsymbol{A}\boldsymbol{E} &= \boldsymbol{E}_1^T\boldsymbol{A}_{11}\boldsymbol{E}_1 + \boldsymbol{E}_1^T\boldsymbol{A}_{12}\boldsymbol{E}_2 + \boldsymbol{E}_2^T\boldsymbol{A}_{12}^T\boldsymbol{E}_1 + \boldsymbol{E}_2^T\boldsymbol{A}_{22}\boldsymbol{E}_2\\ \boldsymbol{E}^T\boldsymbol{B}\boldsymbol{E} &= \boldsymbol{E}_1^T\boldsymbol{B}_{11}\boldsymbol{E}_1 + \boldsymbol{E}_1^T\boldsymbol{B}_{12}\boldsymbol{E}_2 + \boldsymbol{E}_2^T\boldsymbol{B}_{12}^T\boldsymbol{E}_1 + \boldsymbol{E}_2^T\boldsymbol{B}_{22}\boldsymbol{E}_2\end{cases},$$

thus Eq. 4.16 - $\boldsymbol{E}_1^T\times$ Eq. 4.15 becomes

$$\boldsymbol{E}_2^T(\boldsymbol{A}_{12}^T\mathbf{c}' + \boldsymbol{A}_{12}^T\boldsymbol{E}_1\mathbf{r}' + \boldsymbol{A}_{12}\boldsymbol{E}_2\mathbf{r}') = \lambda^{'}\boldsymbol{E}_2^T(\boldsymbol{B}_{12}^T\mathbf{c}' + \boldsymbol{B}_{12}\boldsymbol{E}_1\mathbf{r}' + \boldsymbol{B}_{22}\boldsymbol{E}_2\mathbf{r}') \tag{4.17}$$

Multiplying pesuo-inverse of $\boldsymbol{E}_2^T$, we will have

$$\boldsymbol{A}_{12}^T\mathbf{c}' + \boldsymbol{A}_{12}^T\boldsymbol{E}_1\mathbf{r}' + \boldsymbol{A}_{12}\boldsymbol{E}_2\mathbf{r}' = \lambda^{'}(\boldsymbol{B}_{12}^T\mathbf{c}' + \boldsymbol{B}_{12}\boldsymbol{E}_1\mathbf{r}' + \boldsymbol{B}_{22}\boldsymbol{E}_2\mathbf{r}') \tag{4.18}$$

Let $\mathbf{t}_1 \equiv \mathbf{c}' + \boldsymbol{E}_1\mathbf{r}'$ and $\mathbf{t}_2 \equiv \boldsymbol{E}_2\mathbf{r}'$, then we can write Eq.4.15 as

$$\boldsymbol{A}_{11}\mathbf{t}_1 + \boldsymbol{A}_{12}\mathbf{t}_2 = \lambda^{'}(\boldsymbol{B}_{11}\mathbf{t}_1 + \boldsymbol{B}_{12}\mathbf{t}_2) \tag{4.19}$$

and Eq.4.18 as

$$\boldsymbol{A}_{12}\mathbf{t}_1 + \boldsymbol{A}_{22}\mathbf{t}_2 = \lambda^{'}(\boldsymbol{B}_{12}\mathbf{t}_1 + \boldsymbol{B}_{22}\mathbf{t}_2) \tag{4.20}$$

Combing Eq.4.19 and 4.20, we can get the following eigenvalue system:

$$\begin{bmatrix}\boldsymbol{A}_{11} & \boldsymbol{A}_{12}\\ \boldsymbol{A}_{12}{}^T & \boldsymbol{A}_{22}\end{bmatrix}\begin{bmatrix}\mathbf{t}_1\\ \mathbf{t}_2\end{bmatrix} = \lambda^{'}\begin{bmatrix}\boldsymbol{B}_{11} & \boldsymbol{B}_{12}\\ \boldsymbol{B}_{12}{}^T & \boldsymbol{B}_{22}\end{bmatrix}\begin{bmatrix}\mathbf{t}_1\\ \mathbf{t}_2\end{bmatrix} \tag{4.21}$$

According to Eq.4.7, Eq.4.21 corresponds to the original egienvalue system and $\lambda^{'}$ is the eigenvalue of original system in Eq.4.1. Hence, we have $\lambda = \lambda^{'}$, which means the eigenvalues of eigensystem in Eq.4.2 are identical to the eigenvalues of the original system in the event of up to k faults. ■

4.1.3 Encoding Matrix

Following the result of Gleich et al. [18], we require the matrix $\boldsymbol{E}$ to have full Kruskal rank (Kruskal rank is k) [29]. We use a dense random matrix as our choice of $\boldsymbol{E}$. This coding matrix has full Kruskal rank with probability tends to 1. We note that for scalable parallel execution, one would select matrix $\boldsymbol{E}$ to be sparse [19]. However, our purpose here is to demonstrate convergence and fault tolerance properties of our eigensolver – leaving issues of parallel performance and coding matrix design to future work.

4.2 Implementation Details

We provide details of implementation of our erasure coded eigensolver. We use the TraceMin algorithm, with a few modifications, for solving the generalized eigenvalue problem. As shown in Section 4.1, the generalized eigenvalue problem has a null space of dimension k. To avoid the case that the intermediate eigenvectors fall into the null space, perturbation and purification are introduced in Section 4.2.1. The resulting erasure coded TraceMin algorithm is shown in Algorithm 3.

4.2.1 Perturbation and Purification

Since $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{B}}$ are SPSD, intermediate eigenvectors may fall in the null space of $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{B}}$. To avoid this problem, we add perturbation to the augmented systems and perform purification once the trace is sufficiently reduced [50].

Perturbation

Since we have

$$\tilde{\boldsymbol{A}} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{AE} \\ \boldsymbol{E}^T\boldsymbol{A} & \boldsymbol{E}^T\boldsymbol{AE} \end{bmatrix}, \tilde{\boldsymbol{B}} = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{E} \\ \boldsymbol{E}^T & \boldsymbol{E}^T\boldsymbol{E} \end{bmatrix}$$

We can get the perturbed augment system

$$\tilde{\boldsymbol{A}}_p = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{AE} \\ \boldsymbol{E}^T & \epsilon \boldsymbol{I}_k + \boldsymbol{E}^T \boldsymbol{AE} \end{bmatrix}, \tilde{\boldsymbol{B}}_p = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{E} \\ \boldsymbol{E}^T & \epsilon \boldsymbol{I}_k + \boldsymbol{E}^T \boldsymbol{E} \end{bmatrix} \tag{4.22}$$

For the augmented system, we perturb the lower-right $k \times k$ block. The perturbation is $10^{-06} \times \boldsymbol{I}$, where $\boldsymbol{I}$ is a $k \times k$ identity matrix. After perturbation, $\tilde{\boldsymbol{A}}_p$ and $\tilde{\boldsymbol{B}}_p$ are SPD, and trace minimization can be used to solve the generalized eigenvalue problem.

Purification

During execution, we monitor the trace at each iteration, and purification is performed when the relative difference of two consecutive trace values is less than 10^{-03}. For the system $\tilde{\boldsymbol{A}}_p \tilde{\mathbf{x}} = \lambda \tilde{\boldsymbol{B}}_p \tilde{\mathbf{x}}$, from TraceMin, we obtain the eigenpair $(\mu, \mathbf{u})$, which approximates the true eigenpairs of $\boldsymbol{A}$ ($\boldsymbol{A}\mathbf{v} = \lambda \mathbf{v}$). Therefore, we have

$$\boldsymbol{A}(\mathbf{u} + \delta \mathbf{u}) = (\mu + \delta \mu)(\mathbf{u} + \delta \mathbf{u}) \tag{4.23}$$

Furthermore, we have

$$\mathbf{u}^T \delta \mu = 0 \tag{4.24}$$

Combining Eq. 4.23 and Eq. 4.24, we get the linear system:

$$\begin{bmatrix} \boldsymbol{A} - \mu \boldsymbol{I}_n & -\mathbf{u} \\ -\mathbf{u}^T & 0 \end{bmatrix} \begin{bmatrix} \delta \mathbf{u} \\ \delta \mu \end{bmatrix} = \begin{bmatrix} -(\boldsymbol{A}\mathbf{u} - \mu \mathbf{u}) \\ 0 \end{bmatrix} \tag{4.25}$$

After computing $\delta \mathbf{u}$ and $\delta \mu$ using symmetric linear system solver [30, 44], we update the approximation of the eigenpairs and continue the TraceMin procedure. Since purification is performed infrequently (*once during entire execution*), it does not add significant running time overhead to the solution procedure.

4.2.2 Erasure Coded TraceMin

Our erasure coded TraceMin based eigensolver is shown in Algorithm 3. The algorithm also provides details on how to deal with faults.

Algorithm 3 Fault Oblivious Trace Minimization

1: Choose a block size $s = 2p$ and an $n \times s$ random matrix V_1 of full rank such that $V_1^T \tilde{B}_p V_1 = I$.

2: **for** $t = 0, 1, \ldots$ until convergence **do**

3: Compute $W_t = \tilde{A}_p V_t$ and the interaction matrix $H_t = V_t^T W_t$.

4: Compute the eigenpairs of (Y_t, Θ_t) for H_t.

5: Check the trace and do the purification (only once during the whole algorithm) if the purification condition is satisfied. Update the eigenpairs if the purification executed.

6: Sort the eigenvalue in ascending order and rearrange the corresponding eigenvectors.

7: Compute the corresponding Ritz Vectors $X_t = V_t Y_t$.

8: Compute the residual $R_t = \tilde{A}_p X_t - \tilde{B}_p X_t \Theta_t$.

9: Test for Convergence.

10: Solve the following linear system approximately via the CG to get Δ_t.

$$\begin{bmatrix} \tilde{A}_p & \tilde{B}_p X_t \\ X_t^T \tilde{B}_p & 0 \end{bmatrix} \begin{bmatrix} \Delta_t \\ L_t \end{bmatrix} = \begin{bmatrix} \tilde{A}_p X_t \\ 0 \end{bmatrix} \tag{4.26}$$

11: $\tilde{B}_p$-orthonormalize $X_t - \Delta_t$ into V_{t+1} by the Gram-Schmidt process with reorthogonalization.

12: **end for**

13: **return** Θ.

Compared to original TraceMin algorithm, our fault oblivious TraceMin adds purification procedure in Step 5. This helps us avoid the null space of the generalized system, and is only executed once during the entire execution. It does not incur significant time overhead. Assuming that the augmented matrices $\tilde{\boldsymbol{A}}_p$, $\tilde{\boldsymbol{B}}_p$, and the augmented eigenvectors $\tilde{\boldsymbol{X}}_k$ are distributed among multiple processes by rows, the operations of Algorithm 3 affected by faults in a distributed environment are the matrix-matrix and matrix-vector products. Let the index set associated with process i be I_i, then $[n+k] = \bigcup_i I_i$ (n is the size of original input matrix and k is the size of augmentation part), and the set of faulty indices at time (iteraition) t is $F_t = \bigcup_{i \in P_t} I_i$ (P_t is the set of faulty indices for process P at time (iteration) t).
The viable processes carry out the all-reduce operation by skipping the faulty components F_t in the matrix.

- The Matrix-Matrix Operation:

$$
\begin{aligned}
(\tilde{\boldsymbol{A}}_p, \boldsymbol{V}_t) &= \left((\tilde{\boldsymbol{A}}_p)_{[n+k]\setminus F_t}, (\boldsymbol{V}_t)_{[n+k]\setminus F_t}\right) \\
(\tilde{\boldsymbol{A}}_p, \tilde{\boldsymbol{X}}_t) &= \left((\tilde{\boldsymbol{A}}_p)_{[n+k]\setminus F_t}, (\boldsymbol{V}_t)_{[n+k]\setminus F_t}\right) \\
(\tilde{\boldsymbol{B}}_p, \tilde{\boldsymbol{X}}_t) &= \left(\tilde{\boldsymbol{B}}_p)_{[n+k]\setminus F_t}, (\boldsymbol{X}_t)_{[n+k]\setminus F_t}\right)
\end{aligned}
$$

- The Matrix-Vector Operation in Step 10:

$$
(\tilde{\boldsymbol{A}}_p, \Delta_t) = \left((\tilde{\boldsymbol{A}}_p)_{[n+k]\setminus F_t}, (\Delta_t)_{[n+k]\setminus F_t}\right)
$$

4.3 Experimental results

We present detailed experimental evaluation of our fault tolerant eigensolver. We demonstrate the following key aspects of our solver: (i) our solver is highly effective in tolerating faults as it introduces minimal convergence overhead both in average case, where a randomly selected set of row-column pairs deleted, as well as in the worst case,

where row-column pairs with top k highest leverage scores are deleted. (ii) focusing on the worst case, we analyze two coding schemes – the first scheme is oblivious to leverage scores of row-column pairs when the code is constructed; the second scheme updates the code, as we get successively refined estimates of leverage scores through the iterative TraceMin procedure. We show that our adaptive code yields much better performance compared to the leverage-score-oblivious coding scheme. (iii) recognizing the superiority of the adaptive coding scheme, we characterize the loss in performance from the use of the approximated leverage scores, as compared to exact leverage scores (recall that it is not feasible to compute exact leverage scores a-priori for constructing the code). We show that the use of the approximated leverage scores adds minimal overhead, compared to use of the exact leverage scores; (iv) finally, we consider different fault arrival models – instantaneous, where all faults arrive simultaneously, and the random model, where faults arrive according to an exponential distribution with prescribed mean. We show that our method is robust to different fault arrival models and fault rates.

4.3.1 Input Matrices and Execution Parameters

For our experimental studies, we use two matrices from the University of Florida Matrix Collection [48]. The details of these matrices are shown in Table 4.1. The matrices selected are SPD, sufficiently ill conditioned so as to require a larger number of TraceMin iterations to converge (otherwise the fault tolerance problem may not be vital), and have distinct non-zero structure and application of origin.

Table 4.1.: Matrices Used in Testing

Matrix	**Rows**	**Nonzeros**	**Kind**
`minsurfo`	**40,806**	**203,622**	**Optimization Problem**
`s3dkq4m2`	**90,449**	**4,427,725**	**Structural Problem**

Experimental Setup

- To evaluate the convergence, we monitor the quantity $\frac{||\mathbf{r}_1||_2}{\lambda_1}$ (here, $\mathbf{r}_1 = \tilde{\boldsymbol{A}}_p\mathbf{x}_1 - \lambda_1\tilde{\boldsymbol{B}}_p\mathbf{x}_1$ is the residual for the first eigenpair – λ_1 and $\mathbf{x}_1$, which are the smallest eigenvalue and corresponding eigenvector, respectively). The stopping criteria is set to 10^{-8} for all matrices.

- To characterize the accuracy of our results (the eigenvalues computed by our fault tolerant eigensolver), we first compute the normalized error for each eigenvalue $t_i = \frac{aug_{\lambda_i} - orig_{\lambda_i}}{orig_{\lambda_i}}$. Here, $orig_{\lambda_i}$ and aug_{λ_i} correspond to eigenvalues of the original system and those computed by our method on the augmented system, respectively. The 2-norm of the vector with the first ten normalized error components $\mathbf{t} = [t_1, t_2, \cdots, t_{10}]$ is used to compute the final relative error $rtol = ||\mathbf{t}||_2$.

- To test the robustness of our fault tolerant eigensolver, we add augmented blocks of different sizes to the original system, and simulate different number of faults. Here, $K = 0$ corresponds to the original system; $K = d$ corresponds to an augmented block size of d, and d faults happen during the execution. In our experiments, we set $d = 1, 8, 16$ for convergence tests.

- To characterize average case and worst case performance, we simulate erasures of row-column pairs in different ways. Average case is simulated by erasing a random set of row-column pairs. To evaluate the worst case performance, we rely on leverage scores of row-column pairs [42]. Leverage scores can be computed using an SVD decomposition [40, 41]. For matrix $\boldsymbol{A} = \boldsymbol{U}\Sigma\boldsymbol{V}$($\boldsymbol{A}$ is an $n \times n$ matrix and $\boldsymbol{U}$,Σ, and $\boldsymbol{V}$ are matrices corresponding to the SVD decomposition of $\boldsymbol{A}$), the leverage score for each row of $\boldsymbol{A}$ is given by:

$$l(i) = \sum_{j=1}^{n} U(i,j)^2$$

  For characterizing worst case performance of our method, the k rows with highest leverage scores are erased.

- We use two different fault arrival models – faults arriving instantaneously and faults arriving at different points in time according to an exponential distribution. An exponential distribution is the most commonly used fault arrival model [49] as shown in Eq. 2.3. To simulate the real practice scenario, the number of iterations is used to represent time τ. We use `orig_iter` to represent the number of iterations the original system needs to converge (achieve the termination condition). Different fault rates(r_e) ranging from $\frac{1}{\texttt{orig\_iter}}$ to $\frac{4}{\texttt{orig\_iter}}$ are tested. Note that for an exponential distribution, the mean number of iterations between two consecutive faults is $\frac{1}{r_e}$. This means the average number of faults in `orig_iter` iterations is 1 and 2 for $r_e = \frac{1}{\texttt{orig\_iter}}$ and $\frac{2}{\texttt{orig\_iter}}$, 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 iteration $\frac{\texttt{orig\_iter}}{1+\texttt{orig\_iter}/r_e}$. This is to ensure that the fault process starts, otherwise, in some runs, there are no faults at all for the exponential fault process.

4.3.2 Effectiveness of basic erasure coded eigensolver

In our first set of experiments, we erase a specified number of row-column pairs from the augmented matrix and compare the convergence rate of our method with the case in which there are no erasures. This corresponds to the average-case fault tolerance behavior for our method. We set the augmentation block sizes to be $K = 1, 8$, and 16. In each case, we assume instantaneous faults, and report results for different fault points (iteration count at which faults occur). When faults happen, the relative error increases, and then reduces in the following iterations. This is due to the fact that the corresponding components of the eigenvectors are erased from intermediate computations. However, as evident from Figures 4.1 and 4.2 our solver rapidly recomputes these components and the convergence tracks the non-fault case closely. The following observations can be made from the figures: (i) the overhead in

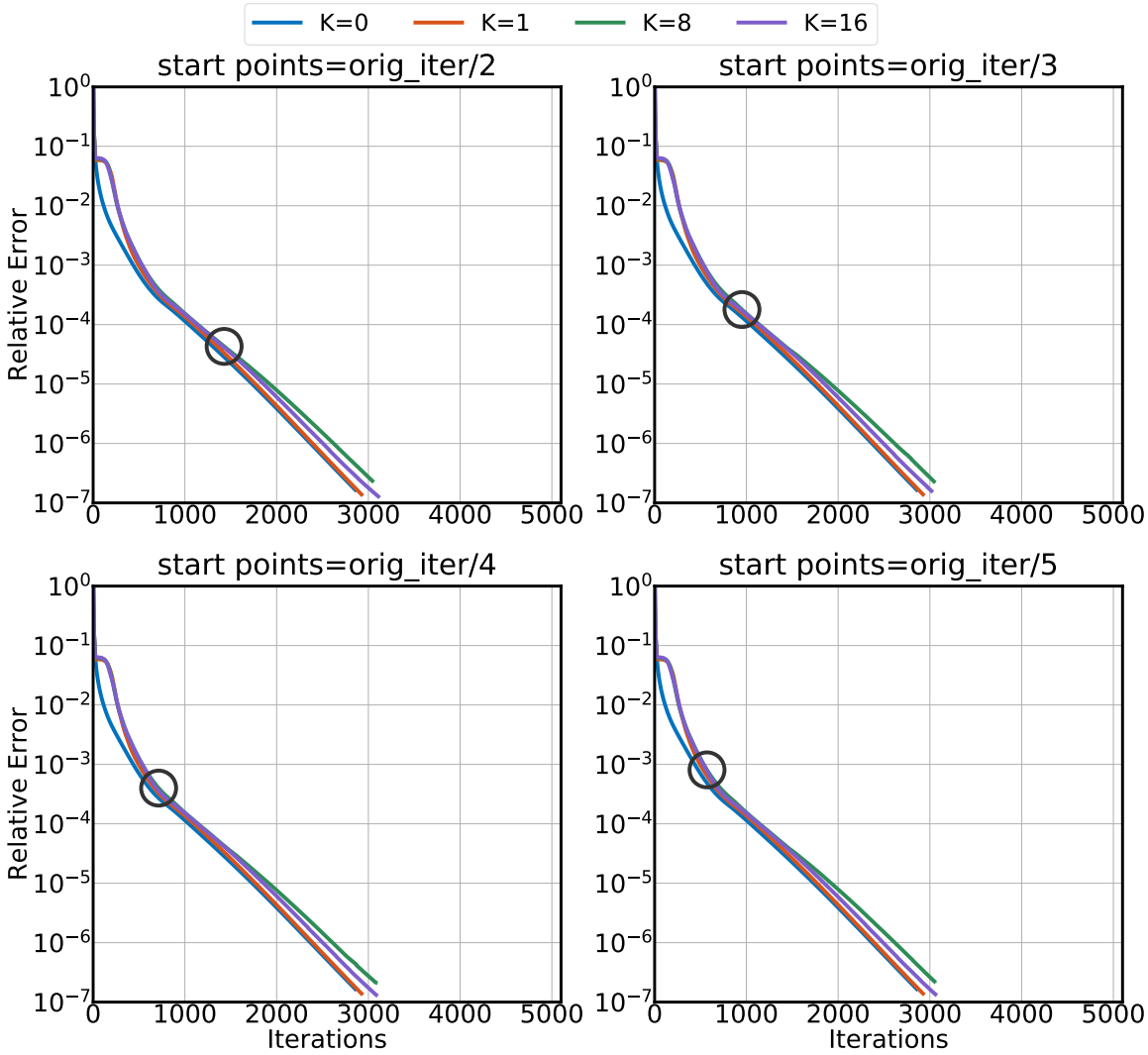


Figure 4.1.: Effectiveness of basic algorithm with instantaneous fault model (`minsurfo`).

terms of iterations to converge for our method is minimal; and (ii) the performance of our method scales very well in terms of increasing number of faults. Note that these excellent results for the average case performance are aided by the fact that random erasures in matrices do not perturb eigenvalues significantly, with high probability. This motivates our next set of experiments on the worst case behavior of our method.

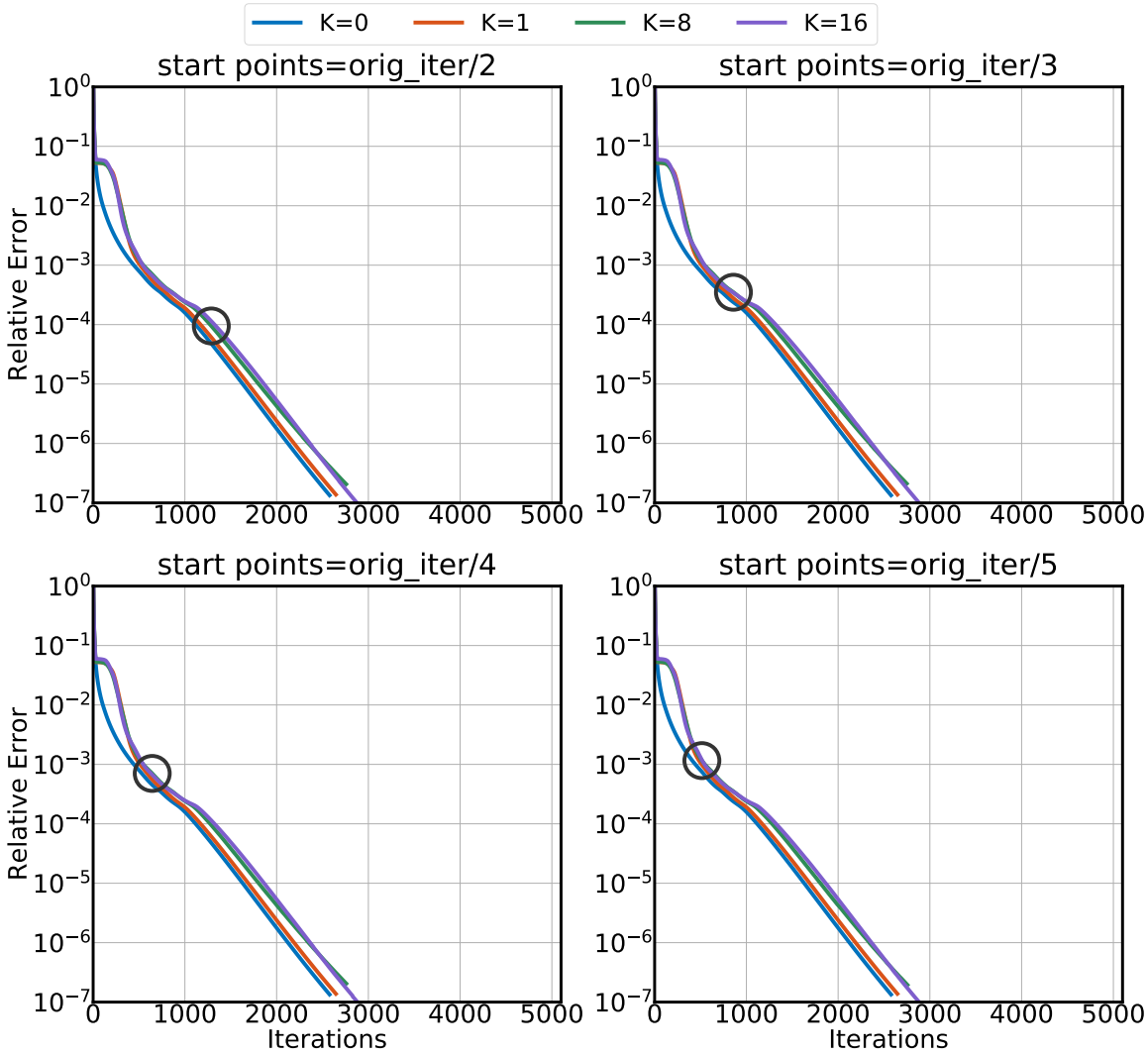


Figure 4.2.: Effectiveness of basic algorithm with instantaneous fault model (`s3dkq4m2`).

4.3.3 Worst Case behavior of basic erasure coded eigensolver

In this set of experiments, we erase K row-column pairs with the largest leverage scores. Note that leverage scores indicate the importance of a row-column pair with respect to the eigenvalues of the matrix.

From our results in Figure 4.1, Figure 4.2, Figure 4.3 and Figure 4.4, we observe the following phenomena: (i) erasing K rows and columns with highest leverage scores

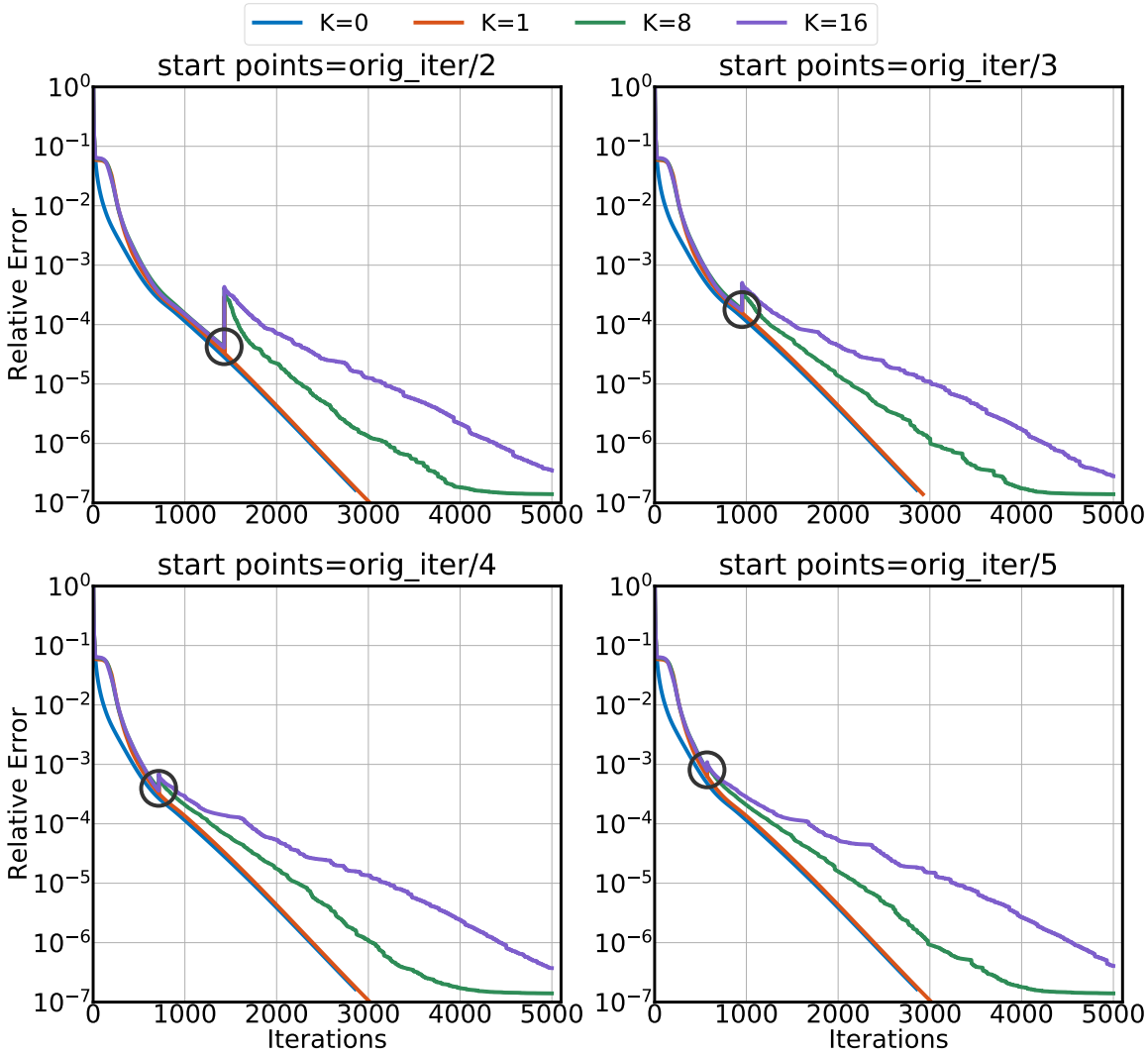


Figure 4.3.: Worst case Performance of basic algorithm with instantaneous fault model (`minsurfo`).

has a more significant impact on the intermediate eigenvalues than random erasures; (ii) our solver is able to achieve convergence even under the worst case fault model, although, the performance in terms of convergence rate is not as good as that of the average case; (iii) the iteration overhead in all cases, even for $K = 16$ is within a factor of two of the non-faulty case; and (iv) the error increases with increasing number of faults (i.e., with increasing K). Our results demonstrate the robustness of our solver in this worst case experiment.

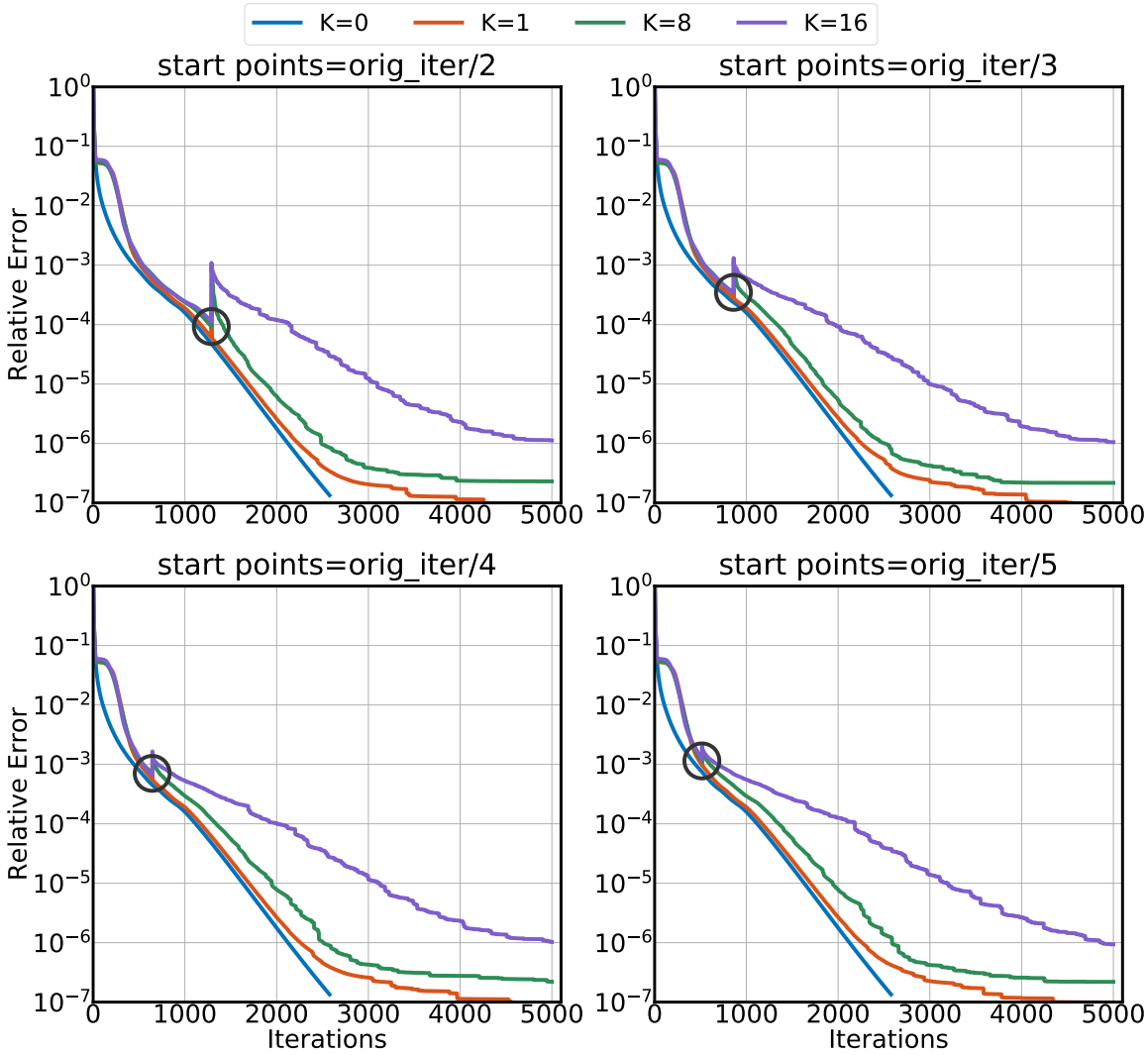


Figure 4.4.: Worst case Performance of basic algorithm with instantaneous fault model (`s3dkq4m2`).

4.3.4 Estimated Leverage Score Updating Method

Recognizing the relative loss of performance in worst case erasures, we implement an adaptive coding scheme that relies on the estimation of leverage scores. Since computing exact leverage scores requires the computation of an SVD [40, 41], we approximate leverage scores using the eigenvectors estimated during TraceMin exe-

cution. Coding blocks are periodically updated using the estimation of leverage scores got from the previous iteration. The coding matrix $\boldsymbol{E}$ is adaptively updated as follows

$$E(i,:) = E(i,:) * \frac{l(i)}{\bar{l}}$$

Here $E(i,:)$ is the i^{th} row of coding matrix $\boldsymbol{E}$, $l(i)$ is the leverage score of i^{th} row and $\bar{l}$ is the average leverage score of all rows. By doing this, we will assign high weights to the rows with high leverage scores, and low weights to the rows with low leverage scores.

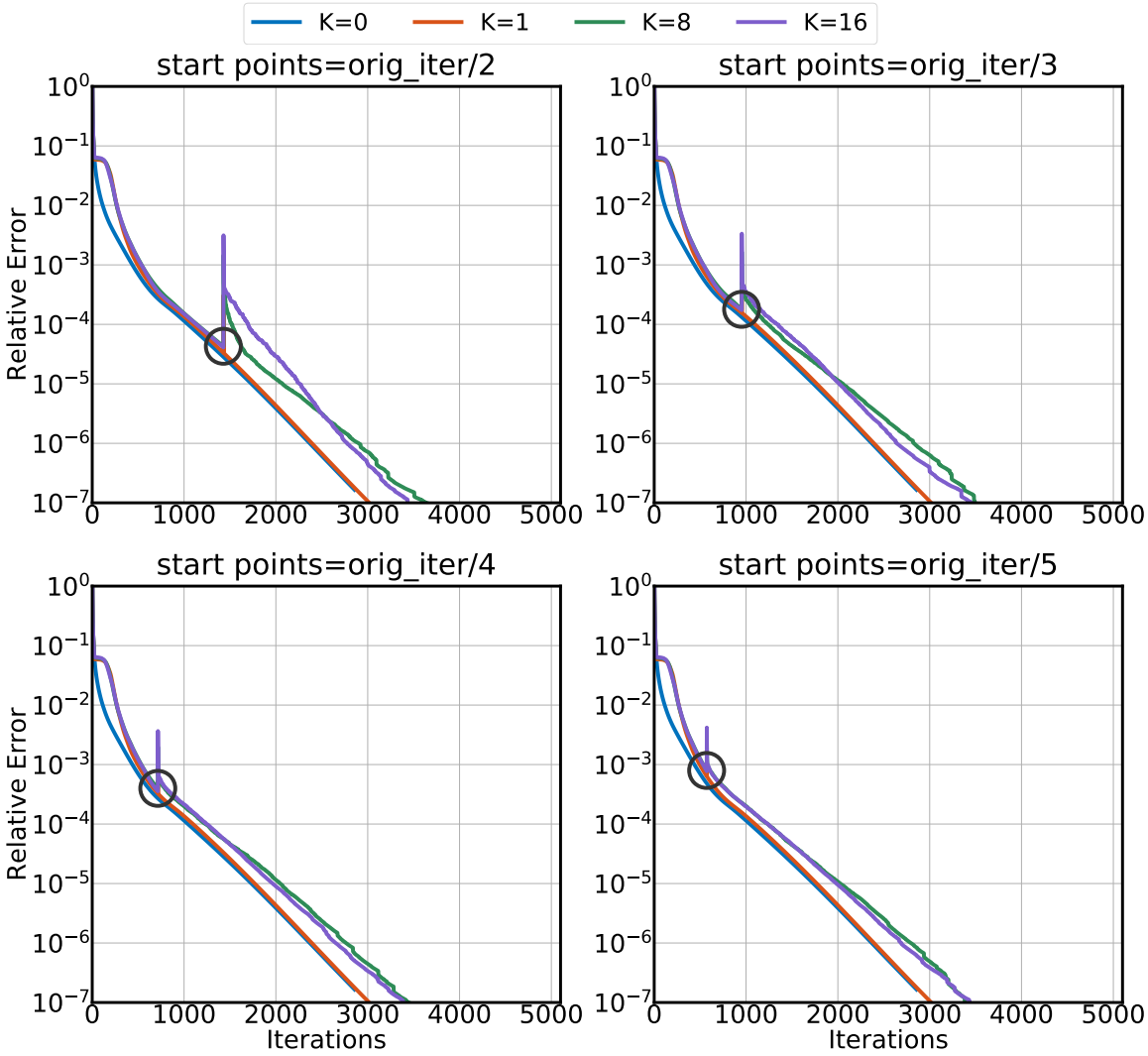


Figure 4.5.: Effectiveness of updated algorithm with instantaneous fault model(`minsurfo`).

For erasures occured in high leverage scores rows, the adaptive coding scheme can better compensate the erasures as it assigns higher weights to the corresponding rows in coding matrix $\boldsymbol{E}$. This is different from the static coding scheme, which assigns static weight to each row no matter where the erasures happen. The results of this adaptive augmentation scheme are presented in Figure 4.5 and Figure 4.6. These plots show excellent performances of our adaptive coding schemes, particularly in comparison with the static coding technique.

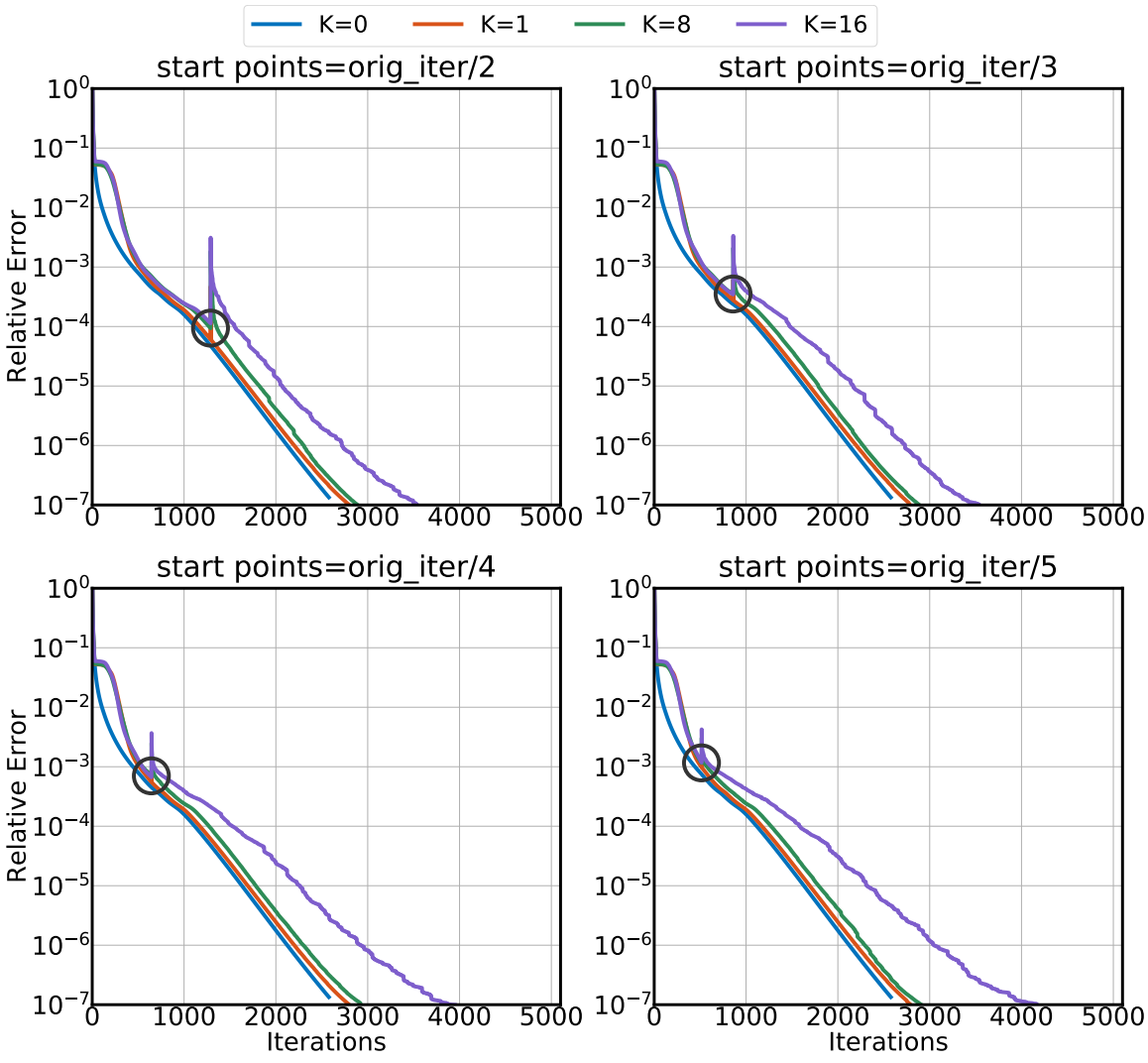


Figure 4.6.: Effectiveness of updated algorithm with instantaneous fault model(`s3dkq4m2`).

4.3.5 Comparative Performance Benefit of Estimated Leverage Score Based Adaptive Coding Scheme

We compare the performance of the non-adaptive and adaptive coding schemes. We test different fault rates for $K = 16$ with the basic and adaptive coding schemes. Results in Figure 4.7 and Figure 4.8 show that the adaptive coding scheme always yields lower relative error than the static (non-adaptive) algorithm and converges faster.

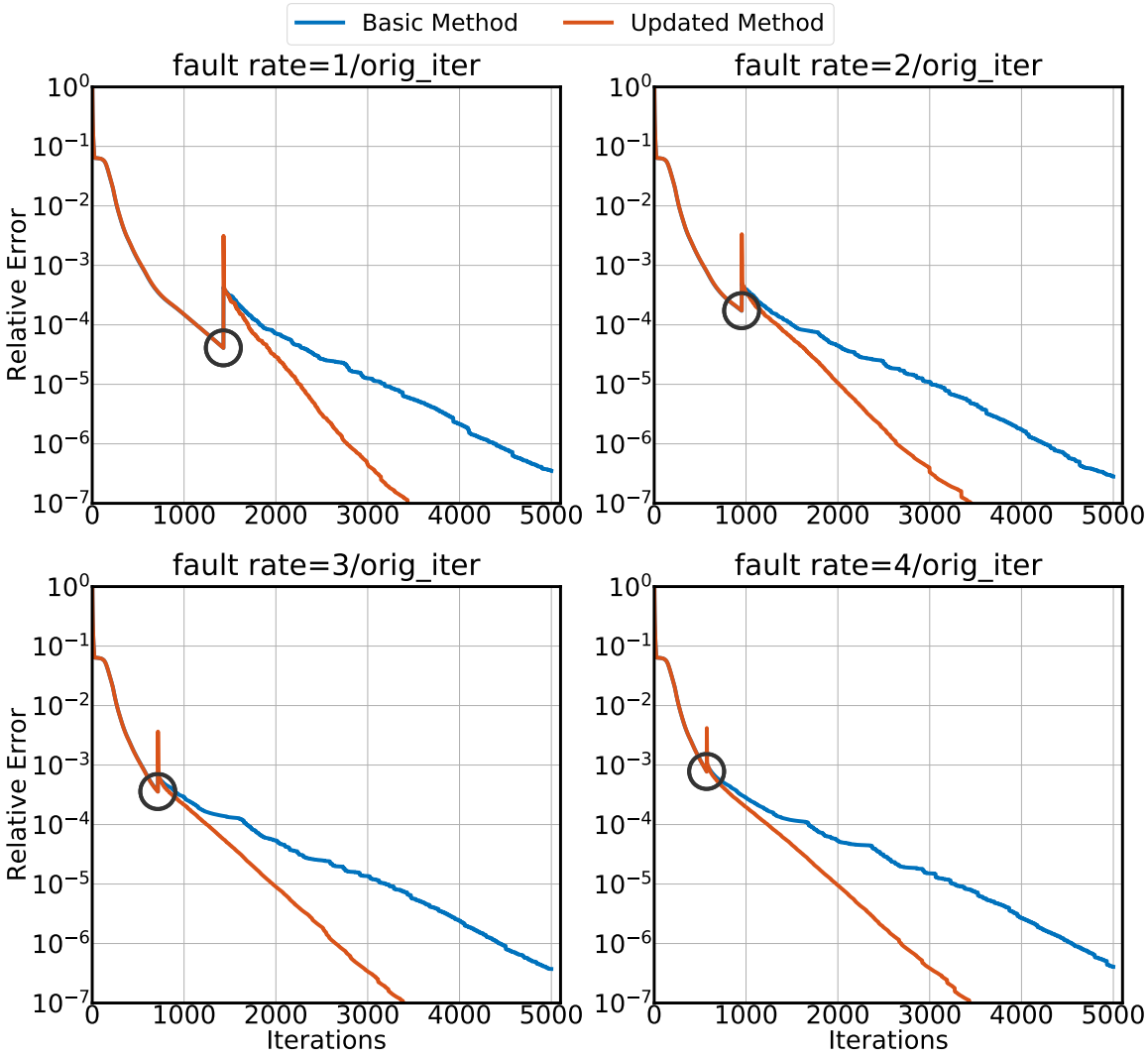


Figure 4.7.: Comparison of Basic Algorithm and Updated Algorithm(`minsurfo`)

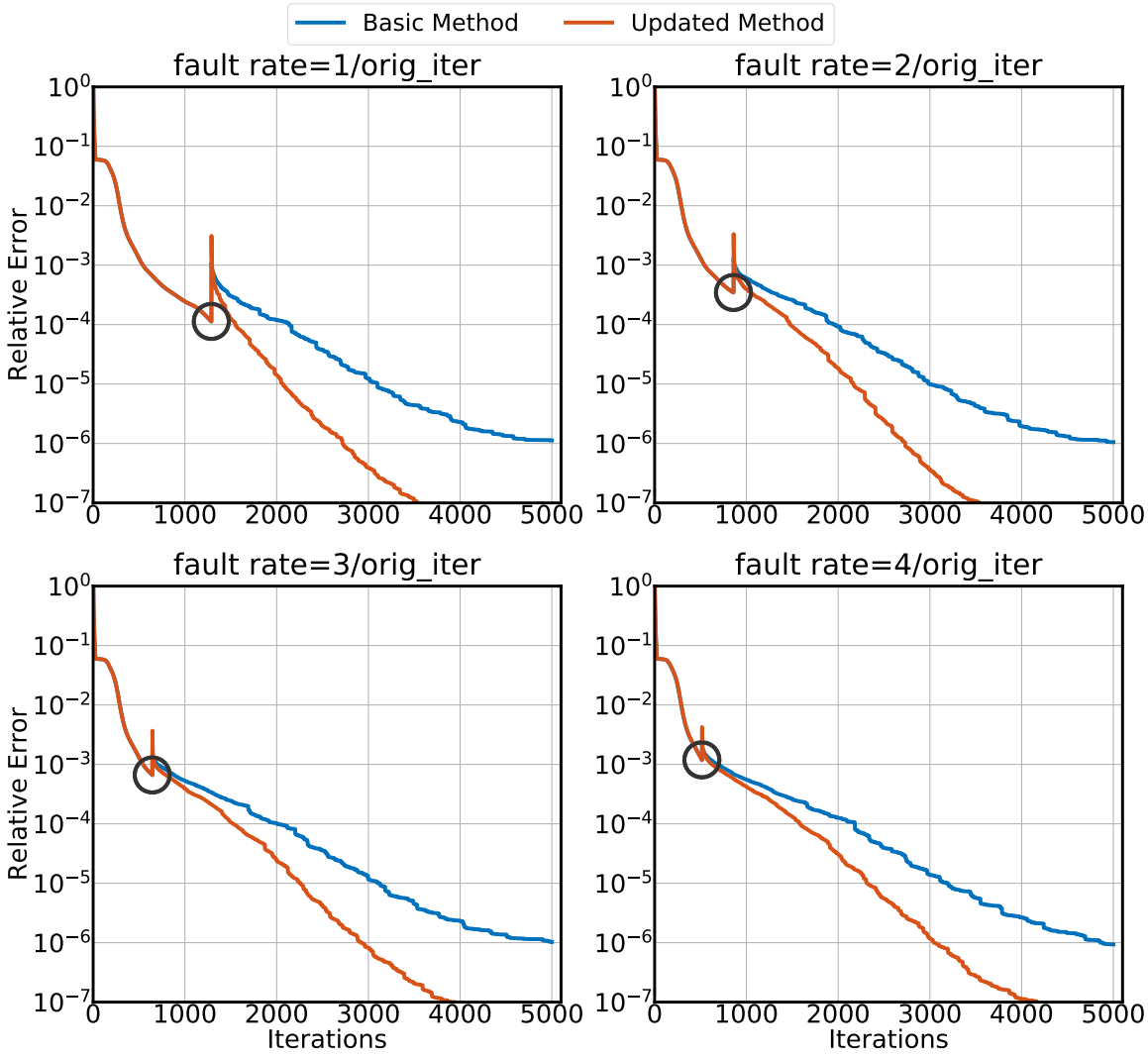


Figure 4.8.: Comparison of Basic Algorithm and Updated Algorithm(`s3dkq4m2`)

4.3.6 Comparison of Exact and Estimated Leverage Score Updating

We notice the infeasibility of computing the exact leverage scores. Instead we rely on estimated leverage scores to adaptively update our coding matrix. We now assess the loss in performance due to this approximation. In this set of experiments, we compare results obtained from coding matrices using estimated leverage scores as compared to those using the exact leverage scores computed a priori using an SVD decomposition.

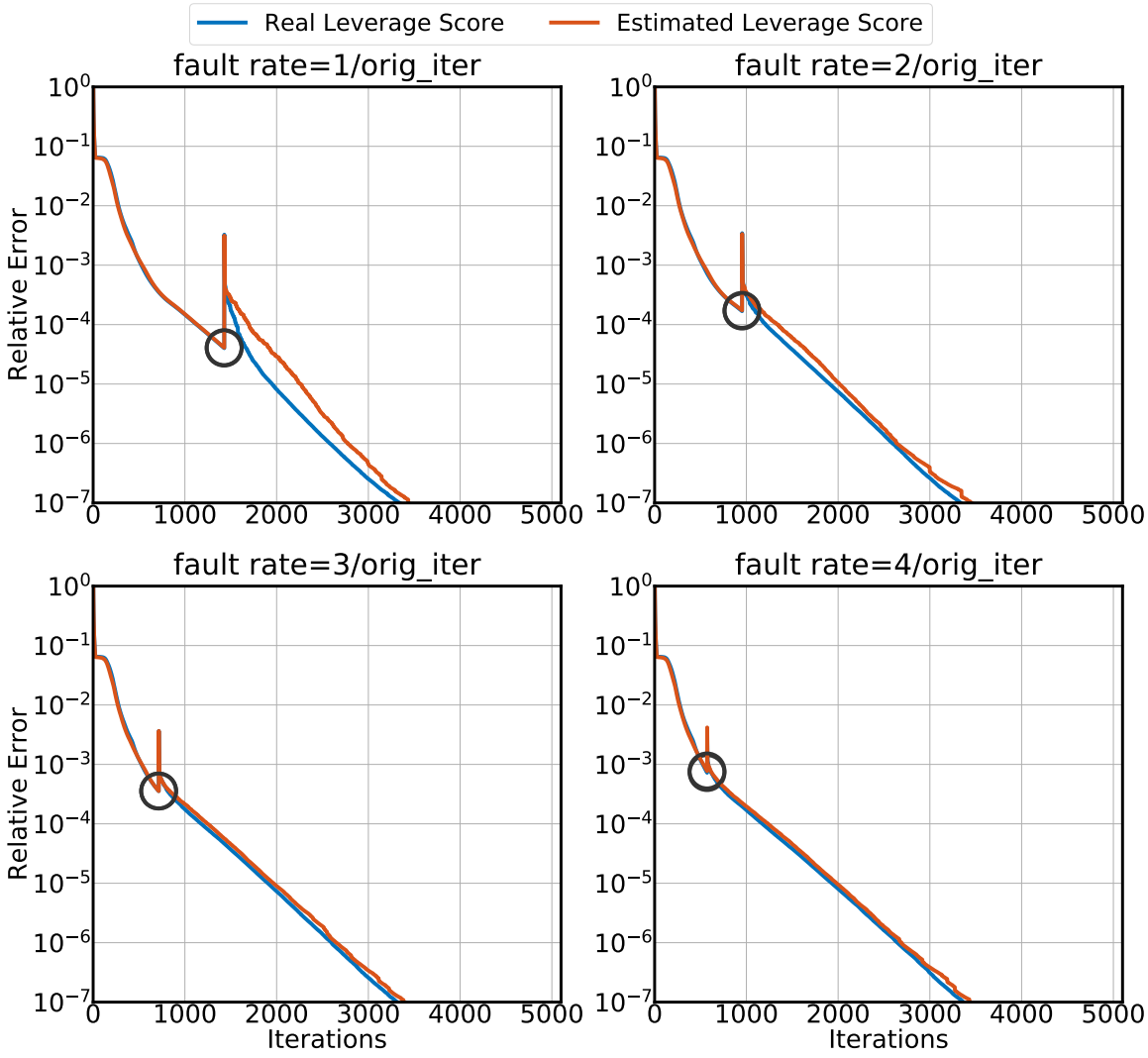


Figure 4.9.: Effect Comparison of Real and Estimated Leverage Score(`minsurfo`)

Figure 4.9 and Figure 4.10 show convergence rates while using the exact and the estimated leverage scores under different fault rates for $K = 16$. For the matrix `minsurfo`, the relative error curves for the two cases are nearly overlapping. For matrix `s3dkq4m2`, with increasing fault rates, the augmented system using the exact leverage scores converges faster than that using the estimated leverage scores. However, both of them achieve similar levels of relative error. These results demonstrate the effectiveness of the estimated leverage scores as a proxy for the exact leverage scores while constructing/updating augmentation blocks.

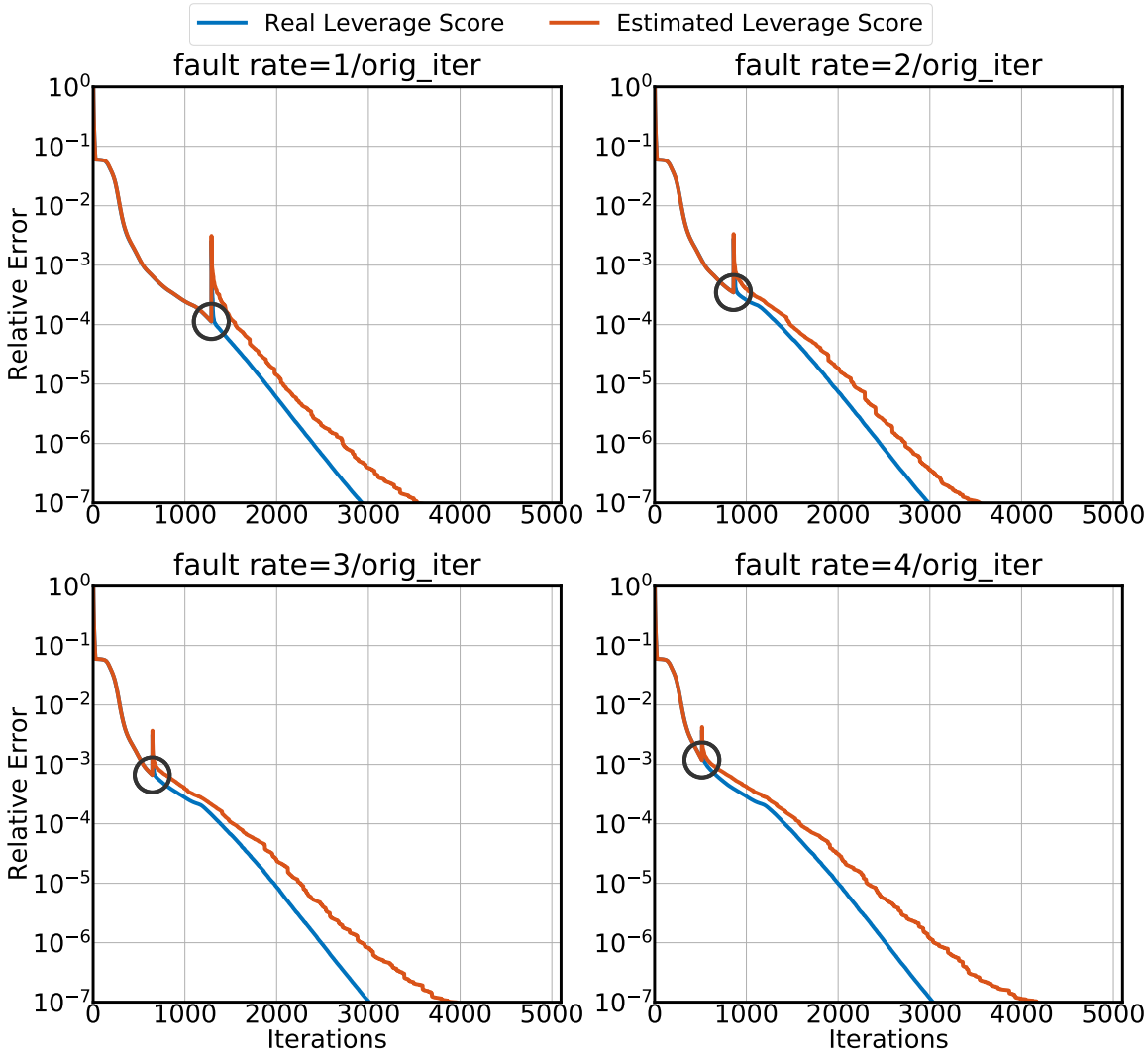


Figure 4.10.: Effect Comparison of Real and Estimated Leverage Score(`s3dkq4m2`)

4.3.7 Time Overheads of Erasure Coded Eigensolver

In this part, we analyze the time overhead of the generalized eigenvalue system with respect to the original system. We let a fixed number of faults happen during the execution ($K = 1, 8, 16$) and calculate the ratio of the solution time for the generalized eigenvalue system with faults to the original eigenvalue system without faults. Solution time corresponds to the time needed to converge to 10^{-7}. Since we are interested in quantifying the computation overhead of our coding block, all results

here are obtained on a single CPU core. Ideally, we want this ratio to be as small as possible.

Figure 4.11 shows the time overheads of erasure coded eigensolver. With increasing

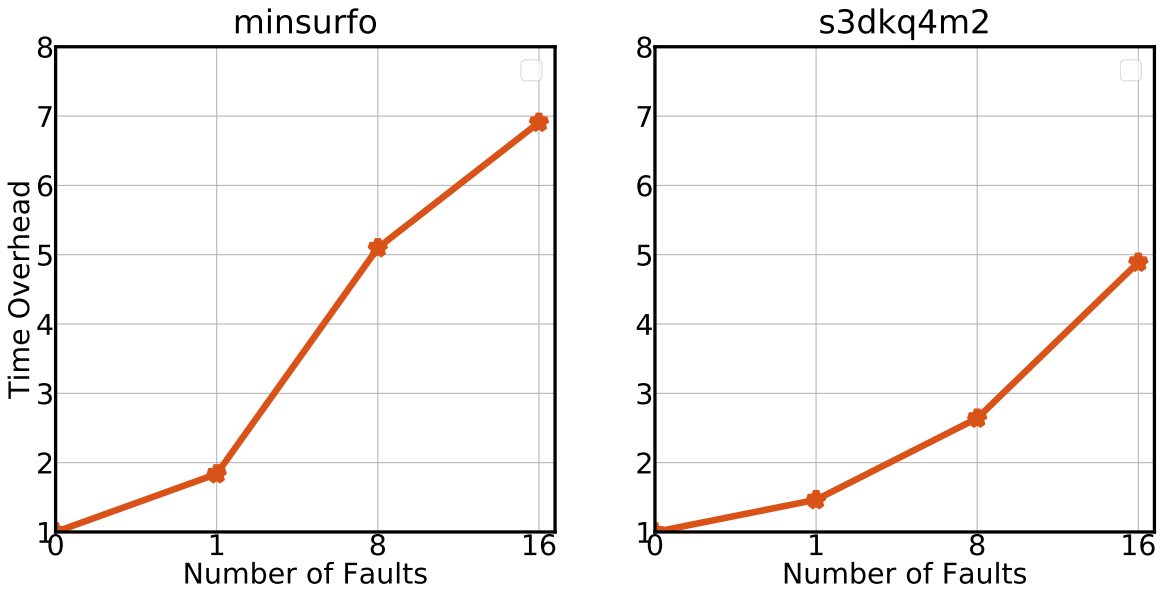


Figure 4.11.: Time Magnification of Different number of faults.

number of faults, the time overhead increases. This comes from two reasons: (1) the dense blocks in the generalized eigenvalue system; and (2) the number of iterations to converge. When the number of faults increases, the size of dense blocks increases and it brings more non-zeros, and thus the number of iterations to converge also increases. Time overhead of `minsurfo` is larger than that of matrix `s3dkq4m2` as the sparsity of `minsurfo` is higher than that of `s3dkq4m2`. Hence, the added dense blocks play a more important role during computation.

4.3.8 Impact of Different Fault Arrival Model

We evaluate the effectiveness of our adaptive coding scheme under the exponential fault arrival model. In this model, faults arrive randomly following an exponential distribution. We test different fault arrival rates ranging from $\frac{1}{\texttt{orig\_iter}}$ to $\frac{4}{\texttt{orig\_iter}}$.

In this experiment, rows with the top-k largest leverage scores are erased one at a time until convergence (or reaching maximum iteration bound). Figure 4.12 and Figure 4.13 demonstrate excellent convergence properties of our eigensolver. We also note that the exponential fault model is easier to handle than the instantaneous fault model.

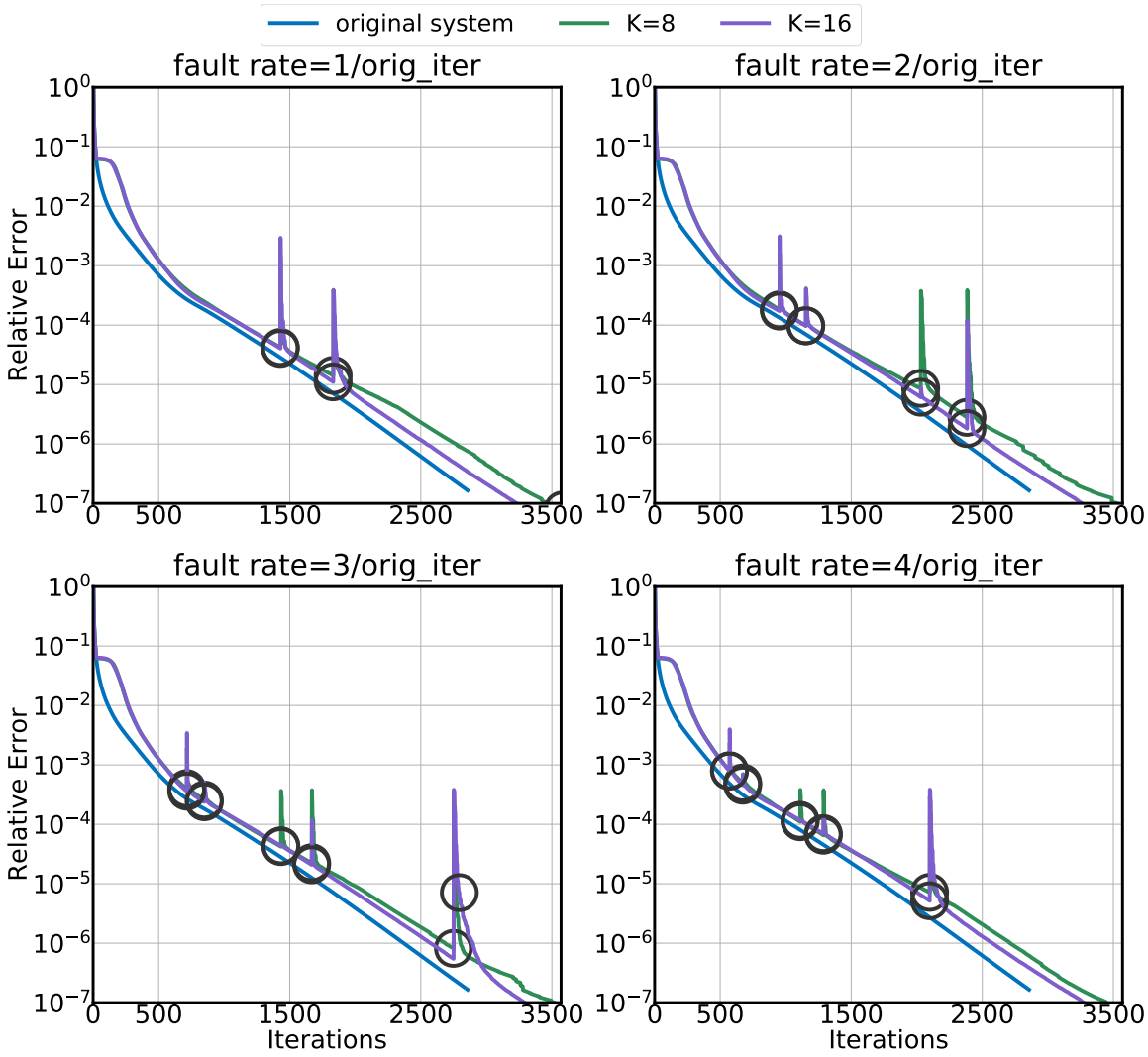


Figure 4.12.: Exponential Fault Arrival Model(`minsurfo`).

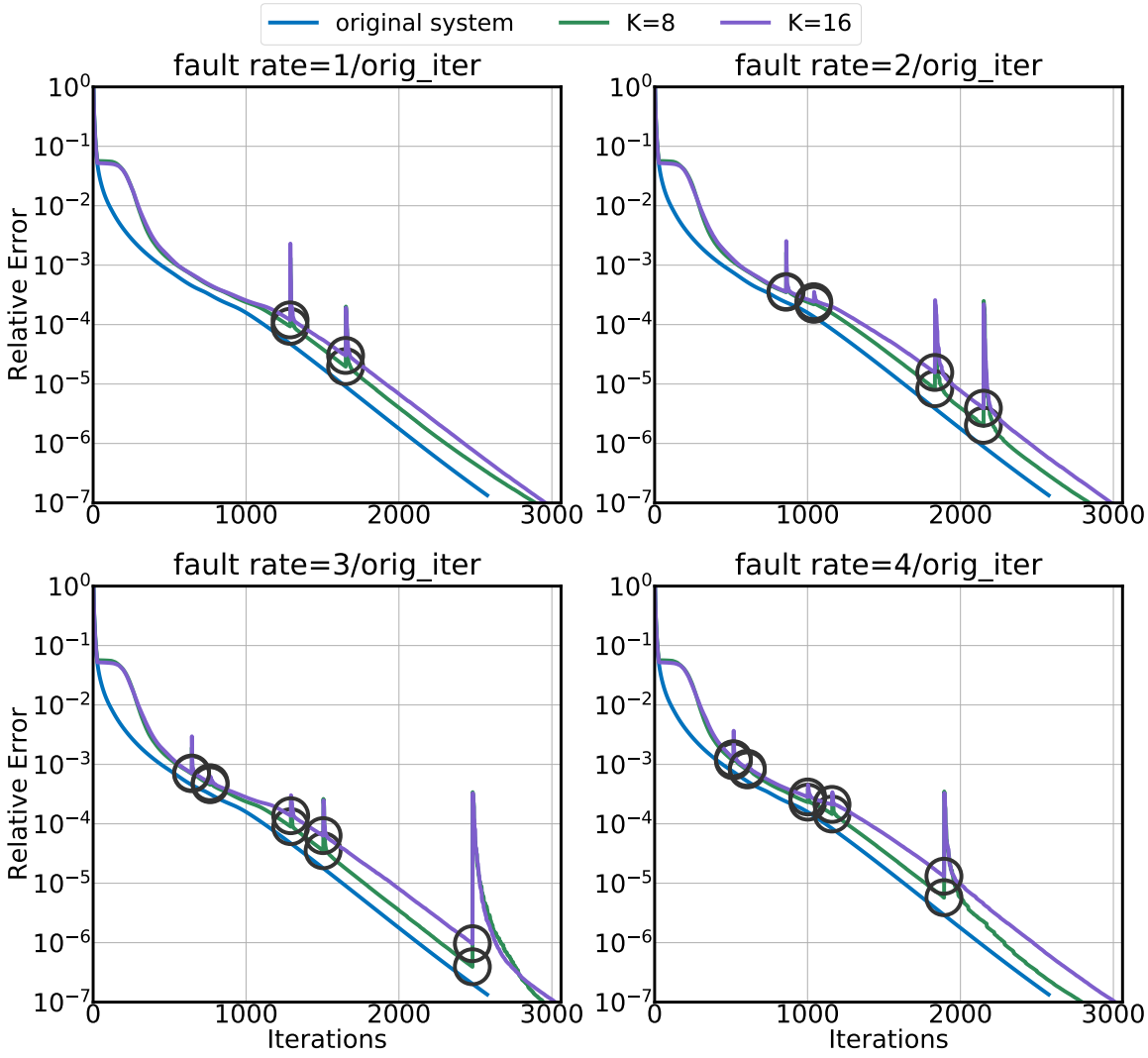


Figure 4.13.: Exponential Fault Arrival Model(`s3dkq4m2`).

4.4 Conclusions

In this chapter, we presentd a novel erasure coded fault tolerant eigenvalue solver. We demonstrated that our solver introduced minimal overhead in terms of iterations to convergence for instantaneous and random faults. Motivated by the loss of performance in worst case erasures, we presented an adaptive coding scheme that updates the augmentation blocks using successively refined estimates of eigenvectors, as our iterative computation proceeds. We demonstrate excellent robustness and performance

of this solver across a range of fault rates and arrival models. Our methods provide the basic computational substrate for massively parallel eigenvalue computations in potentially faulty environments. The success on eigenvalue problem further proves the applicability of erasure coded computations in linear algebraic methods.

5 PLAN OF RESEARCH

In this chapter, we discuss future research directions based on current results, further improvement of erasure coded solvers, the application of current solvers in different domains, and the development of erasure coded computations for other linear algebraic problems.

Distributed Fault Tolerant Eigensolver In Chapter 4, the effectiveness of erasure coded computation for eigensolvers is demonstrated. Besides the fault tolerance property shown in Chapter 4, the challenges for distributed fault tolerant eigensolver include good parallel performance (speedup), minimal time overheads, and the robustness to different fault arrival models and fault rates on distributed platforms. The results in Chapter 2 showed the effectiveness of the sparse coding matrix in distributed computation and applying sparse coding matrix to our eigensolver is effective in improving parallel performance and reducing time overhead.

Adaptive Fault Tolerant Eigensolver For the erasure coded eigensolver in Chapter 4, we augmented the input matrix before importing it to TraceMin. The computation overhead is paid at each iteration even if there are no faults happen during the execution. The adapative linear solver in Chapter 3 gives an adaptive scheme for updating input matrix according the current situation (faulty or non-faulty) and input matrix is augmented only when faults are encountered. The results on the linear solver show that the adaptive scheme has good convergence rate and parallel performance, and lower overhead. We plan to develop a similar adaptive scheme for the fault tolerant eigensolver. The challenges for an adaptive fault tolerant eigensolver include a correctness proof of equivalent eigenvalues of the original system

and the adaptive system, demonstration of convergence rates, and evaluation of the robustness for different fault arrival models and fault rates.

Applications of Fault Tolerant Linearsolver and Eigensolver In many applications such as Newton method for optimization and Principal Component Analysis (PCA) [51, 52], the linear system ($f''(x)\delta x = f'(x)$, $f'(x)$ and $f''(x)$ are first order and second order derivation of $f(x)$, where we want to get the root of it ($f(x) = 0$)) needs to be solved in Newton iterations, and the top k largest eigenvalues (principal components) are needed for PCA. We plan to apply our fault tolerant linear solver to the Newton method and fault tolerant eigensolver to the PCA method. The challenges include the augmentation strategy of the input matrix, and the development of recovery operation for execution under potentially faulty environments.

Erasure Coded Matrix Decomposition LU decomposition and QR decomposition [53, 54] are widely used in linear system solvers and linear least square problems, respectively. Singular Value Decomposition [40, 41] is useful in signal processing and statistics. The success of linear solver and eigensolver based on erasure coded computations motivates us to apply the concept of erasure coded computations to matrix decomposition procedures. Based on erasure coded computation, we plan to augment the input matrix and perform decomposition on the augmented matrix. Then the decomposition results for the input matrix can be recovered from the decomposition results of the augmented matrix through inexpensive operations. The challenges for erasure coded decomposition include construction of the augmented matrix, the proof of properties of the augmented matrix, and the feasibility of recovery.

6 SUMMARY

In this dissertation, we show how the concept of erasure coded computation, which is derived from the erasure coded storage, can be used to tolerate faults. The erasure coded computation is applied to linear algebraic problems– solving a system of linear equations and computing eigenvalues.

Error correcting codes and the erasure coded storage are introduced and used to motivate the concept of *erasure coded computation* (ECC). The challenges and benefits of erasure coded computation are presented for general problems.

The erasure coded linear system is introduced in Chapter 2. To achieve good parallel performance and minimize the fill in augmentation rows, we present a sparse coding scheme and prove necessary properties of the coding matrix (random-at-recovery property). For distributed implementation, we consider a balanced partition of matrices across different processes. Our experimental results on the convergence rates, speedup, time overhead and robustness under different fault arrival models and different fault arrival rates show the effectiveness of the *distributed fault tolerant linear system solver*.

Based on the distributed fault tolerant linear system solver, we present an *adaptive fault tolerant linear system solver* (ADLS) in Chapter 3. The solver runs on the original system until faults happen. The erased rows are then replaced by the precomputed augmentation blocks. This guarantees that the size of system we solve remains the same, and that there is no null space of the system. Another benefit of the adaptive linear solver is the number of computations during the execution. Since we execute on the original system, and only replace erased rows by augmentation rows when faults happen, any increase in computation is a result of increased iterations to convergence. We evaluated the solver for its convergence rates on small and large

matrices, its parallel performance, and its time overhead under different fault arrival models and fault arrival rates.

Success in the erasure coded linear solver motivated us to apply the concept of erasure coded computations to the eigenvalue problem. We proposed the *erasure coded eigensolver* in Chapter 4. The solver first reformulated the original eigenvalue system to a generalized eigenvalue system. Properties of the generalized eigenvalue system are shown and equivalence of the original eigenvalue system and the generalized eigenvalue system are proved. We introduced a perturbation and purification procedure in the erasure coded trace minimization algorithm. From the experimental results, we demonstrated the effectiveness of our solver under random case and the worst case, when the rows with top leverage scores are erased. We also presented an adaptive coding scheme and illustrated the importance of updating coding matrix based on the estimated leverage scores obtained from the trace minimization algorithm. The robustness of the erasure coded eigensolver under different fault arrival models and fault arrival rates was also presented and validated.

REFERENCES

REFERENCES

[1] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[2] Flavin Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):56–78, February 1991.

[3] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. In *Proceedings of 1986 ACM Fall Joint Computer Conference*, ACM '86, pages 1150–1158, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[4] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. Deterministic replay: A survey. *ACM Comput. Surv.*, 48(2):17:1–17:47, September 2015.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, 2004.

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

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

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

[9] Zizhong Chen and Jack Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Trans. Parallel Distrib. Syst.*, 19(12):1628–1641, 2008.

[10] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. *Fault tolerant high performance computing by a coding approach*, pages 213–223. 2005.

[11] Jacob A. Abraham Chien-Yi Chen. Fault-tolerant systems for the computation of eigenvalues and singular values, 1986.

[12] Kuang-Hua Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, June 1984.

[13] Franklin T. Luk and Haesun Park. Fault-tolerant matrix triangularizations on systolic arrays. *IEEE Trans. Computers*, 37(11):1434–1438, 1988.

[14] Claude Elwood Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 7 1948.

[15] Richard Wesley Hamming. Error-detecting and error-correcting codes. 29(2):147–160, 1950.

[16] Robert G. Gallager. Low-density parity-check codes, 1963.

[17] 11 mds codes. In F.J. MacWilliams and N.J.A. Sloane, editors, *The Theory of Error-Correcting Codes*, volume 16 of *North-Holland Mathematical Library*, pages 317 – 331. Elsevier, 1977.

[18] Yao Zhu, Ananth Grama, and David F. Gleich. Erasure coding for fault oblivious linear system solvers. *SIAM J. of Scientific Computing*, 39(1):C48–C64, 2017.

[19] Xuejiao Kang, David F. Gleich, Ahmed Sameh, and Ananth Grama. Distributed fault tolerant linear system solvers based on erasure coding. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2478–2485, 2017.

[20] Xuejiao Kang, David Gleich, Ahmed Sameh, and Ananth Grama. Adaptive fault tolerant linear system solver. Technical report, West Lafayette, IN, USA, 2018.

[21] Xuejiao Kang, Xin Cheng, David Gleich, Ahmed Sameh, and Ananth Grama. Erasure coded eigensolver. Technical report, West Lafayette, IN, USA, 2018.

[22] J. S. Plank. Erasure codes for storage systems: A brief primer. *;login: the Usenix magazine*, 38(6), December 2013.

[23] Dimitri Pertin, Alexandre Van Kempen, Benoît Parrein, and Nicolas Normand. Comparison of RAID-6 Erasure Codes. In *The third Sino-French Workshop on Information and Communication Technologies, SIFWICT 2015*, Nantes, France, June 2015.

[24] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[25] George Kola, Tevfik Kosar, and Miron Livny. Faults in large distributed systems and what we can do about them. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, pages 442–453, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[26] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[27] Zizhong Chen and Jack Dongarra. Numerically stable real number codes based on random matrices. In *Computational Science(ICCS)*, pages 115–122, 2005.

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

[29] J.B. Kruskal. Three-way arrays: Rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics. *Linear Algebra Appl.*, 18(2):95–138, 1977.

[30] Youcef Saad. A flexible inner-outer preconditioned gmres algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, March 1993.

[31] Valeria Simoncini and Daniel B. Szyld. Flexible inner-outer Krylov subspace methods. *SIAM Journal on Numerical Analysis*, 40:2219–2239, 2003.

[32] Jasper vanden Eshof and Gerard L. G. Sleijpen. Inexact krylov subspace methods for linear systems. *SIAM J. Matrix Anal. Appl.*, 26(1):125–153, January 2004.

[33] Valeria Simoncini and Daniel B. Szyld. Theory of inexact Krylov subspace methods and applications to scientific computing. *SIAM Journal on Scientific Computing*, 25:454–477, 2003.

[34] James Elliott, Mark Hoemmen, and Frank Mueller. A numerical soft fault model for iterative linear solvers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 271–274. ACM, 2015.

[35] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA, 1994.

[36] A. Sameh and Z. Tong. The trace minimization method for the symmetric generalized eigenvalue problem. *Journal of Computational and Applied Mathematics*, 123:155–175, November 2000.

[37] Ahmed H. Sameh. A trace minimization algorithm for the generalized eigenvalue problem. *SIAM J. Numer. Anal.*, 19(6):1243–1259, 1982.

[38] Michael W. Mahoney. Randomized algorithms for matrices and data. *Foundations and Trends in Machine Learning*, 3(2):123–224, 2011.

[39] Mark Rudelson and Roman Vershynin. Sampling from large matrices: An approach through geometric functional analysis. *J. ACM*, 54(4), July 2007.

[40] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. Fast monte carlo algorithms for matrices i: Approximating matrix multiplication. *SIAM J. Comput.*, 36(1):132–157, July 2006.

[41] Petros Drineas, Ravi Kannan, and Michael W.Mahoney. Fast monte carlo algorithms for matrices ii: Computing a low-rank approximation to a matrix. *SIAM J. Comput.*, 36(1):158–183, July 2006.

[42] Petros Drineas, Malik Magdon-Ismail, Michael W. Mahoney, and David P. Woodruff. Fast approximation of matrix coherence and statistical leverage. *J. Mach. Learn. Res.*, 13(1):3475–3506, December 2012.

[43] H. Robbins. A remark of stirling's formula. *Amer. Math.*, Monthly 62, 1955.

[44] Pascal Joly and Gérard Meurant. Complex conjugate gradient methods. *Numerical Algorithms*, 4(3):379–406, Oct 1993.

[45] Gerard Meurant. *The Lanczos and Conjugate Gradient Algorithms: From Theory to Finite Precision Computations (Software, Environments, and Tools)*. 2006.

[46] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[47] Alban Karapici, Enri Feka, Igli Tafa, and Alban Allko. The simulation of round robin and priority scheduling algorithm. In *2015 12th International Conference on Information Technology - New Generations*, Las Vegas, NV, USA, 2015.

[48] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(6):1–25, November 2011.

[49] Xavier Castillo, Stephen McConnel, and Daniel Siewiorek. Derivation and calibration of a transient error reliability model. *IEEE Transactions on Computers*, C-31:658–671, 1982.

[50] G. W. Stewart and Ji guang Sun. *Matrix Perturbation Theory*, pages 165–217. 1990.

[51] Tjalling J. Ypma. Historical development of the newton-raphson method. *SIAM Rev.*, 37(4):531–551, December 1995.

[52] H. Hotelling. Analysis of a complex of statistical variables into principal components. *J. Educ. Psych.*, 24, 1933.

[53] A. M. TURING. Rounding-off errors in matrix processes. *The Quarterly Journal of Mechanics and Applied Mathematics*, 1(1):287–308, 1948.

[54] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.