RESILIENCE FOR ASYNCHRONOUS ITERATIVE METHODS FOR SPARSE LINEAR SYSTEMS

by

Evan Coleman
B.S., March 2009, Oregon State University
M.S., May 2011, Syracuse University

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

MODELING & SIMULATION

OLD DOMINION UNIVERSITY
May 2019

Approved by:

Masha Sosonkina (Director)

Ross Gore (Member)

Duc Nguyen (Member)

Hong Yang (Member)

ABSTRACT

RESILIENCE FOR ASYNCHRONOUS ITERATIVE METHODS FOR SPARSE LINEAR SYSTEMS

Evan Coleman
Old Dominion University, 2019
Director: Dr. Masha Sosonkina

Large scale simulations are used in a variety of application areas in science and engineering to help forward the progress of innovation. Many spend the vast majority of their computational time attempting to solve large systems of linear equations; typically arising from discretizations of partial differential equations that are used to mathematically model various phenomena. The algorithms used to solve these problems are typically iterative in nature, and making efficient use of computational time on High Performance Computing (HPC) clusters involves constantly improving these iterative algorithms. Future HPC platforms are expected to encounter three main problem areas: scalability of code, reliability of hardware, and energy efficiency of the platform. The HPC resources that are expected to run the large programs are planned to consist of billions of processing units that come from more traditional multicore processors as well as a variety of different hardware accelerators. This growth in parallelism leads to the presence of all three problems.

Previously, work on algorithm development has focused primarily on creating fault tolerance mechanisms for traditional iterative solvers. Recent work has begun to revisit using asynchronous methods for solving large scale applications, and this dissertation presents research into fault tolerance for fine-grained methods that are asynchronous in nature. Classical convergence results for asynchronous methods are revisited and modified to account for the possible occurrence of a fault, and a variety of techniques for recovery from the effects of a fault are proposed. Examples of how these techniques can be used are shown for various algorithms, including an analysis of a fine-grained algorithm for computing incomplete factorizations. Lastly, numerous modeling and simulation tools for the further construction of iterative algorithms for HPC applications are developed, including numerical models for simulating faults and a simulation framework that can be used to extrapolate the performance of algorithms towards future HPC systems.

Copyright, 2019, by Evan Coleman, All Rights Reserved.

This thesis is dedicated to my family; especially Sara, for always being there to support me.

ACKNOWLEDGEMENTS

There are many people who deserve recognition for the help they have provided me. First, my committee members: Dr. Masha Sosonkina, Dr. Ross Gore, Dr. Duc Nguyen, and Dr. Hong Yang. Second to my family and friends. Thank you for all of your understanding and support.

Throughout the course of my studies I've had the opportunity to work and have interesting discussions with a large number of wonderful colleagues located at many different institutions; many of whom I've been lucky enough to collaborate with on various studies and papers. The experiences have been overwhelmingly positive and have helped me grow immensely as a researcher.

A special thanks to my major advisor, Dr. Masha Sosonkina, for all of the support and encouragement that she has provided over the last several years. I appreciate all of the opportunities and guidance that you've given to me; thank you.

Throughout my research, I have benefited from the use of several different computing facilities. My work was primarily conducted on the computers at Old Dominion University, primarily the Turing cluster but additionally on the Borges and Rulfo computers; however, some experiments were also conducted at the National Energy Research Scientific Computing Center (NERSC) on both the Hopper and Edison supercomputers.

Lastly, I would also like to acknowledge the support I've received from the Naval Surface Warfare Center, Dahlgren Division. In addition to the direct funding of my graduate studies at Old Dominion University, I have also received support from the Academic Fellowship Program and the In-house Laboratory Independent Research Program. The support from these programs helped me greatly during my pursuit of higher education.

TABLE OF CONTENTS

Page

LIST OF TABLES . . . viii

LIST OF FIGURES . . . xi

Chapter

1. INTRODUCTION . . . 1
   1.1 MOTIVATION . . . 2
   1.2 PROBLEM . . . 7
   1.3 CONTRIBUTIONS . . . 8
   1.4 OUTLINE . . . 10

2. BACKGROUND MATERIAL . . . 12
   2.1 LITERATURE REVIEW . . . 12
   2.2 KRYLOV SUBSPACE SOLVERS . . . 25
   2.3 PRECONDITIONERS . . . 30
   2.4 ASYNCHRONOUS ITERATIVE METHODS . . . 39

3. TECHNIQUES FOR RESILIENCE TO SOFT FAULTS . . . 50
   3.1 MODELING THE IMPACT OF SOFT FAULTS . . . 52
   3.2 RESILIENCE STRATEGIES . . . 76
   3.3 SUMMARY . . . 83

4. NUMERICAL MODELING OF FAULTS . . . 84
   4.1 SIMULATING SOFT FAULTS . . . 86
   4.2 NUMERICAL RESULTS . . . 98
   4.3 SUMMARY . . . 123

5. USE CASE: FINE-GRAINED INCOMPLETE FACTORIZATIONS . . . 125
   5.1 FINE-GRAINED PARALLEL ALGORITHM . . . 127
   5.2 CONVERGENCE OF THE ALGORITHM . . . 129
   5.3 SOFT FAULT RESILIENCE . . . 138
   5.4 NUMERICAL RESULTS . . . 156
   5.5 SUMMARY . . . 176

6. FRAMEWORK FOR MODELING AND ANALYSIS . . . 182
   6.1 DESIGN OF SIMULATION FRAMEWORK . . . 183
   6.2 FRAMEWORK EXTENSION FOR FAULT-TOLERANCE . . . 203
   6.3 NUMERICAL EXPERIMENTS . . . 204
   6.4 SUMMARY . . . 207

7. CONCLUSIONS . . . 209
7.1 THEORETICAL RESULTS AND TECHNIQUES . . . 209
7.2 NUMERICAL SOFT FAULT MODELING . . . 209
7.3 FAULT TOLERANT FINE-GRAINED INCOMPLETE FACTORIZATIONS . 210
7.4 FRAMEWORK FOR MODELING AND ANALYSIS . . . 210
7.5 FUTURE WORK . . . 211

REFERENCES . . . 213

VITA . . . 236

LIST OF TABLES

Table Page

1 Comparison of different fault injection techniques . . . 93

2 Comparison of SBSFM and PBSFM . . . 94

3 Comparison of fault effects, LAPLACE2D . . . 96

4 Full fault simulation results, sticky faults . . . 114

5 Input parameters the value of which varied in the experiments . . . 116

6 Summary of beneficial results - persistent fault injection . . . 122

7 Progression of residual norms . . . 144

8 Example of progression of nonlinear residual norm . . . 145

9 Fault model comparison . . . 159

10 Symmetric problem summary . . . 160

11 Symmetric condition numbers . . . 161

12 Non-symmetric problem descriptions . . . 171

13 Effect of increasing α . . . 172

14 Results for non-symmetric problem set . . . 173

15 Increase in non-zeros for different levels of ILU fill-in . . . 174

16 Krylov subspace solver resilience - non-symmetric problem set . . . 175

17 Mean times for copy, compute, and update operations . . . 196

18 Boundary conditions for the second implementation of the Laplacian . . . 199

19 Mean iteration time and standard deviation by thread count . . . 200

20 Comparisons of run times between parallel executions and simulation . . . 203

LIST OF FIGURES

Figure Page

1 Nominal HPC program flow 4

2 Potentially faulty HPC program flow 5

3 Breakdown of Types of Soft Faults 7

4 Example of the effects of fill-in 32

5 Demonstration of ARMS block factorization 35

6 Second level ARMS factorization 36

7 Final Schur complement for ARMS demonstration 36

8 Convergence rate comparison for FGMRES 39

9 Component-wise progression of common termination conditions for the asynchronous Jacobi algorithm 74

10 Simple fault injection baseline example 87

11 BFSFM injection example 88

12 PBSFM injection example 89

13 SBSFM injection example 91

14 Breakdown of Types of Soft Faults 91

15 Distribution of run times in a fault-free environment 101

16 Effect of bit-flip faults in the exponent and sign bits 102

17 Effect of bit-flip faults in the mantissa bits 103

18 Effect of faults injected using the SBSFM 104

19 Effect of faults injected using the PBSFM 105

20 Effect of recovery with bit-flip faults in the exponent and sign bits 106

21 Effect of recovery with bit-flip faults in the mantissa bits 107

22 Effect of fault recovery with the SBSFM 108

23 Effect of fault recovery with the PBSFM 109

24 Effect of soft faults, sticky faults, small problem 113

25 Effect of soft faults, sticky faults, large problem 114

26 Effect of soft faults, persistent faults, outer matvec, varied l^2-norm 117

27 Effect of soft faults, persistent faults, outer matvec, decreasing l^2-norm 118

28 Effect of soft faults, persistent faults, application of preconditioner 120

29 Effect of soft faults, persistent faults, outer matvec and application of preconditioner, decreasing l^2-norm 121

30 Effect of soft faults, persistent faults, outer matvec and application of preconditioner, small faults 122

31 Sparsity plots, symmetric matrices, unordered 162

32 Sparsity plots, symmetric matrices, Reverse Cuthill-Mckee ordering 178

33 Example of impact of a fault 179

34 Success rates - Perturbation-based faults, symmetric problems 179

35 Success rates - Bit-flip faults, symmetric problems 180

36 Krylov subspace solver performance - Perturbation-based faults, symmetric problems 180

37 Krylov subspace solver performance - Bit-flip faults, symmetric problems 181

38 Results of resilience testing, non-symmetric problems 181

39 Stages in the proposed framework development 184

40 Block diagram of the simulation framework 185

41 Example of nominal performance of the synchronous Jacobi iteration. 189

42 Example of experiments within the simulation framework. Each line shows the effect of slowing down a single processor to some factor of the (synchronous) performance of the other processors. 190

43 Example of experiments within the simulation framework. Each line shows the effect of increasing the variance in processor performance from 1 to 5 to 10. 191

44 Performance variations between Safe and Race as a function of thread count. . . 195

45 Safe copy, compute, and update histograms with kernel fits. 197

46 Race copy, compute, and update histograms with kernel fits. 198

47 Iteration time histograms with kernel fits. 201

48 Block diagram of the simulation framework with added support for fault tolerance mechanisms 204

49 Effect of differing values of γ on the progression of the residual 206

CHAPTER 1

INTRODUCTION

Many advancements in science in engineering stem from high fidelity simulations that are very large scale in nature. The computational requirements of such simulations naturally lend them to be executed on the largest High Performance Computing (HPC) platforms available. As the requirements of these simulations continue to grow, increasingly larger scales of computation must be provided by the HPC resources.

Future HPC platforms continue to scale in the number of processing elements as they progress towards performing exascale levels of computation. Several reports from various sources including the U.S. Department of Energy (e.g., [1]–[3]) and collaborations between academic, industrial, and government entities (e.g., [4]–[9]) have identified major challenge areas as HPC platforms progress towards exascale. At a high level, these challenges can be broadly categorized as follows:

1. Scalability: designing applications that are able to efficiently make use of the greatly increased level of parallelism that will be present on exascale capable machines.

2. Resilience: exascale level HPC platforms are expected to experience hardware errors at an increased rate and algorithms and applications will need to be designed to deal with such errors.

3. Energy: current techniques and methodologies would create exascale level systems that consumed over 100 MW of power [10]; this is not sustainable, and energy-aware computing needs to be employed to help drive the total amount of energy consumed down.

Asynchronous iterative methods are a class of parallel algorithm in which a processor does not need to wait upon input from other processors before proceeding in the computations assigned to it. Removing synchronization between the processors offers a means to increase performance. More traditional synchronous algorithms may have the following drawbacks [11]: synchronism may be hard or computationally expensive to enforce in practice, communication delays may introduce computational bottlenecks, the act of synchronizing may cause far more communication to occur than is necessary for convergence of the algorithm, and natural variance among processors (even in a homogeneous computing environment) may lead to having many processors idle for large amounts of time. The ability for an algorithm to function in an asynchronous manner allows it to help tolerate latency in HPC environments.

1.1 MOTIVATION

Looking forward to the future of HPC, it is important to develop algorithms that are resilient to faults. On future platforms, the rate at which faults occur is expected to decrease dramatically [4]–[7], which will cause the mean time between failures (MTBF) to continue to decrease. Analysis of components has led to the conclusion that future components will suffer similar failure rates [12], [13], which, when combined with the drastic increase in the number of components, could cause MTBF to be on the order of tens of hours [14]. This calls for the design of fault-tolerant computational algorithms that are robust in the face of these failures. Development of such algorithms has become one of many priorities on the road towards exascale.

Many large scale simulations spend the vast majority of their computational time attempting to solve large sparse systems of linear algebraic equations, typically arising from discretizations of (systems of) partial differential equations that are used to mathematically model various phenomena. Examples of such applications include tools for computational fluid dynamics such as `FUN3D` [15] and `OpenFOAM` [16], simulations of shock hydrodynamics

like `LULESH` [17], as well as libraries designed to help with solving partial differential equations (e.g., `FEniCS` [18]) or finite elements problems (e.g., `deal.II` [19]), or even multiphysics simulations such as those built around the `MOOSE` framework [20]. The algorithms used to solve these problems are typically iterative in nature, and making efficient use of computational time on high performance computing clusters involves constantly improving these iterative algorithms.

Fine-grained parallel methods decompose a problem into a large number of small tasks. If these tasks can be performed independently, then the computation can be done asynchronously. These methods are beginning to be used more prevalently due to their ability to be parallelized naturally on modern co-processors such as GPUs and Intel Xeon Phis®[1]. Many examples of recent work using fine-grained parallel methods are available including work on fine-grained relaxation methods [21], [22], studies concerning the use of fine-grained methods as preconditioners [23], [24], and fine-grained solvers for triangular systems [25].

Asynchronous iterative methods describe the more specific class of fine-grained parallel iterative algorithms where each computing element is allowed to perform its task without waiting for updates from any of the other processes. This dissertation aims to examine the class of algorithms that are captured by fixed point iterative methods for solving linear systems of equations. This class of algorithms encompasses many recent techniques that are of great use in both solving and preconditioning sparse linear systems. A fixed point iteration,

$$x^{(k+1)} = G\left(x^{(k)}\right), \tag{1}$$

can be updated in an asynchronous manner, with the ultimate goal of finding a fixed point, i.e. a location x^* in the domain such that $x^* = G(x^*)$. This class of problems has been used in a wide variety of applications including: the solution of linear systems [21], [26], [27], the preconditioning of linear solvers [23], [24], optimization [28], [29], and techniques for

[1] Intel Corp., Santa Clara, CA

solving partial differential equations [30], among many others. The prevalence of fixed point iterations, especially within the realm of asynchronous algorithms, provides a focal point for much of the analysis presented throughout this dissertation; however, many of the techniques extend naturally to other domains.

The expected flow of any program that is executed on a high-performance computing (HPC) environment is provided in Fig. 1. Generally, the program starts computing, some sequence of operations is executed and then program execution terminates as expected.

Fig. 1: Nominal HPC program flow

However, if faults are introduced during the operations step, there are a range of potential outcomes that become possible. In total, the following scenarios may occur:

- the computations involved in the operations step could enter an infinite loop and continue indefinitely;
- the computations could suffer a failure such that the operations step is aborted entirely and the program exits;
- the operations step could complete successfully, but the failure that is encountered could corrupt the output; or
- the operations step could complete successfully, and the output could be within the initial tolerance.

This range of potential outcomes is depicted in Fig. 2.

Faults can broadly be divided into two categories: hard faults and soft faults [31], [32].

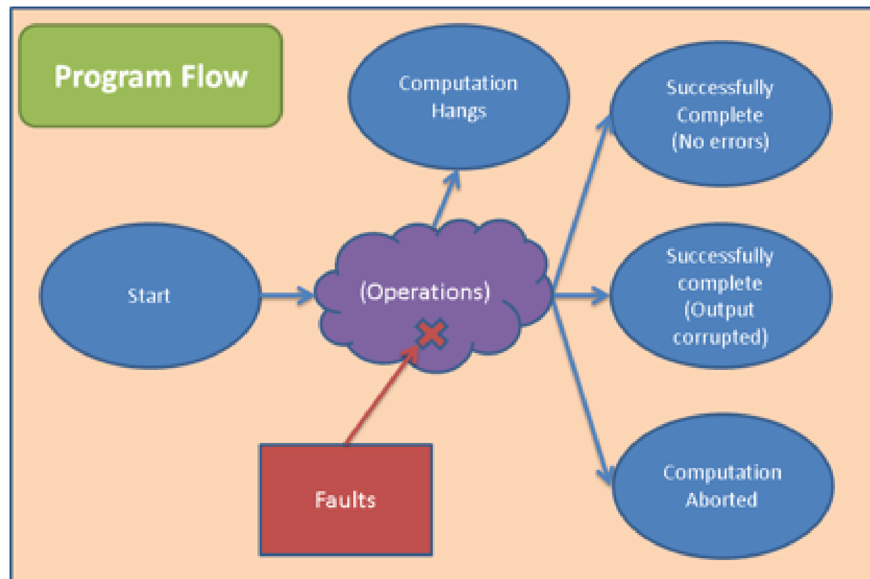


Fig. 2: Potentially faulty HPC program flow

- **Hard faults:** cause immediate program interruption and typically come from negative effects on the physical hardware components of the system or on the operating system itself.
- **Soft faults:** represent all faults that do not cause the executing program to stop and are the primary focus of this work. Most often, soft faults refer to some form of data corruption that is occurring either directly inside of, or as a result of, the algorithm that is being executed.

At a high level, successful fault tolerance for asynchronous iterative methods with respect to hard faults relies chiefly upon successful detection of the fault itself. This will most likely be handled by the HPC platform; however, the successful recovery of the iterative method in question requires the iterative algorithm itself to have the knowledge that a hard fault has occurred. This could be achieved internally in the algorithm by declaring a hard fault if components belonging to a particular block (corresponding to some specific processing element) fail to be updated within some stated time bound.

Similar to the case of a hard fault, the most important aspect to recovering from a soft

fault is successful fault detection. However, this is often more difficult in the case of a soft fault since – though it corrupts data – it does not cause direct interruption to the flow of the iterative process. Many detection techniques rely on choosing an appropriate tolerance to check a property of the algorithm that has predictable behavior (e.g. a residual that is monotonically decreasing, a known property concerning a vector/matrix norm, etc); a tolerance that is too loose will allow potentially harmful errors to go undetected, while a tolerance that is too strict may report a fault when none actually occurred ("false positive") and cause the program to do extra work to recover from a non-existent problem.

Historically, one of the first approaches towards this goal was the implementation of checksums in the computation of matrix operations [33]. In this approach, each matrix or vector is extended with additional memory that encodes a checksum that can be used to detect and correct single upset faults caused by the hardware. This was the beginning of an algorithmic based approach towards fault tolerance. This style of fault tolerance can allow for more efficient recovery from soft faults, but oftentimes it is harder to detect the fault initially. The balance in choosing the correct fault tolerance method to recover from soft faults is typically application dependent.

Following the taxonomy presented in [31], [32], soft faults can be further divided into three categories:

- **Transient Faults**: temporary and can be viewed as faults that occur only once.
  Example: Changing a single floating point number at a single point during execution

- **Sticky Faults**: linger and can affect multiple operations. It is always possible to remedy the cause of these faults if the root cause can be determined.
  Example: There is a fault in a data copy, and the incorrect data will be used (resulting in faulty results) until the data is recopied

- **Persistent Faults**: persist during a large part of the computing time. It is not always possible to remedy the cause of these faults even if it can be detected.

Example: A hardware problem (processor or RAM) that causes results to be computed incorrectly, but does not terminate program execution

A pictorial representation of this taxonomy is provided in Fig. 3.

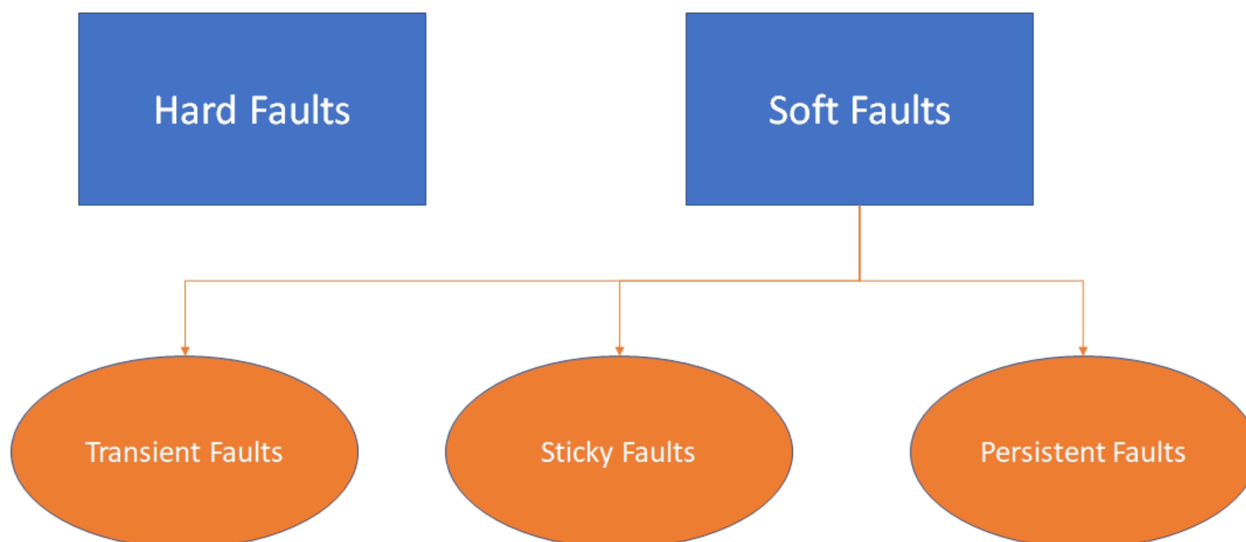


Fig. 3: Breakdown of Types of Soft Faults

1.2 PROBLEM

Being able to improve the iterative algorithms in use requires analyzing the combination of the algorithm itself, the problem domain, and future HPC platforms. The combination of future high performance computing platforms and current algorithms and applications has led to three main problem areas being identified in the community: scalability of existing and future code, reliability of the hardware environment, and the energy efficiency of the computing cluster. The future high performance computing resources that are expected to run the large programs discussed above are planned to consist of billions of processing units that come from more traditional multicore processors as well as a variety of different hardware accelerators. This expected growth in parallelism and scale leads to the presence

of all three of the identified problems.

Recent work in the community has hinted that asynchronous iterative algorithms may be able to address the scalability concern. However, with this new direction of research comes a new question: *how can the resiliency of these asynchronous algorithms be guaranteed?* Existing work on algorithm development for the next generation HPC systems tends to focus primarily on developing fault tolerance mechanizations for traditional iterative solvers. While some recent work has begun to revisit the tractability of using asynchronous methods for solving large scale problems, this dissertation presents research into fault tolerance for (fine-grained) asynchronous methods. These methods tend to scale well to both large systems and to systems of differing architectures since they attempt to completely remove the synchronization delay present in all traditional iterative methods. Additionally, asynchronous methods tends to be more energy efficient since they are able to take advantage of energy saving mechanisms present in the hardware and tend to maximize time spent computing, consequently minimizing the idle time of the system. As this dissertation aims to address the question of resilience of asynchronous methods directly, all three of the main challenges facing future large scale simulation will be addressed by the results presented in this work.

1.3 CONTRIBUTIONS

From a high-level point of view, this dissertation makes the following contributions to the field:

- The theory and mathematical analysis needed to make general statements about the convergence of fine-grained methods subject to the potential impact of a fault. These developments provide a better understanding of the behavior of the algorithms under study, facilitating development of fault tolerant variants.
- Examples of using the theory in applications. Specifically an examination of fault tolerant variants of fine-grained preconditioning algorithms for preparing preconditioners

for solving linear systems. A realistic use case helps establish the efficacy of the theory that is developed, and provides a model for how other similar algorithms can be modified to become resilient.

- Generalized soft fault models that can be used to simulate the occurrence of a fault in a generic sense that is not tied to a specific manifestation of an error. The use of these fault models will be used in the development of fault tolerant algorithms. The use of numerical soft fault models abstracts away any connection to a particular method that a fault may manifest, and allows experiments to carefully control the amount of corruption induced to ensure that they consistently force sufficiently bad behavior to drive a reaction from the algorithm.

- Simulation tools capable of modeling the performance of asynchronous variants of fine-grained algorithms for various hardware architectures that can be utilized in algorithm development. Specifically, the simulation framework that is developed allows for quick, easy experimentation to be conducted on a variety of potential algorithmic modifications.

Developing algorithms that are resilient to faults is of paramount importance and fine-grained parallel methods are no exception. This dissertation aims to offer a means of generating resilient fine-grained asynchronous methods by putting forth: a simulation framework for experimenting with these methods, predictive performance models capable of extrapolating the performance of algorithms to future HPC hardware, a generalized fault model for use in designing resilient algorithms, mathematical theory for dealing with this class of methods, and a detailed example of how the theory can be used to develop fault tolerant variants of existing fine-grained algorithms.

1.4 OUTLINE

This work is organized into seven chapters in total. Chapter 2 provides a literature review of all recent related studies and introduces the background material necessary for understanding the later studies. In particular, an overview of classical and asynchronous iterative methods is provided and the mathematical framework for studying asynchronous iterative methods is introduced.

Next, Chapter 3 discusses in greater detail existing results concerning the convergence of asynchronous iterative methods and introduces several new results that cover the convergence of asynchronous iterative methods in the case that a soft fault occurs. Additionally, this chapter proposes several different techniques that can be used to recover from the occurrence of a fault and provides quick examples that show how each strategy can be used to ensure resilience of the asynchronous Jacobi algorithm.

Chapter 4 develops the novel numerical soft fault model that is used in many of the subsequent experiments that attempt to judge the impact a soft fault may have an on algorithm. A comparison and analysis of this numerical soft fault model with similar techniques developed in the community as well as with the direct effects of injecting a bit flip directly are provided, as well as experiments and analysis of the impact that the fault model may have on asynchronous iterative methods. These results are used to help create efficient fault tolerant algorithmic variants. This chapter also conducts similar analysis for traditional iterative methods used in HPC applications (specifically FGMRES) in order to demonstrate that this fault model can also be used in other environments.

Chapter 5 takes the techniques and results that were proposed in Chapter 3 and applies them to a popular fine-grained algorithm: the fine-grained parallel incomplete LU factorization. This algorithm represents a more complicated alternative to the asynchronous Jacobi algorithm that is quickly explored in Chapter 3, and the techniques are examined in much greater detail in order to provide an example of how they can be used in the most general sense. The fault tolerant variants that are created are then tested using both the numerical

soft fault model developed in Chapter 4 as well as direct injection of bit-flips, and the results are explored.

The development of a simulation framework that can be used for modeling and analysis is then detailed in Chapter 6. This includes the development and validation of the framework based on empirical data generated from shared memory experiments. A series of numerical experiments demonstrating the utility of the simulation tools developed is presented, including a use case demonstrating how the simulation framework can be used to help generate efficient fault tolerant algorithms.

Lastly, Chapter 7 provides summaries of all experiments and several different possible directions for future work.

CHAPTER 2

BACKGROUND MATERIAL

This chapter provides a literature review that covers recent related studies being conducted around the HPC community, as well as introductory background material that is common to, and needed for a full understanding of, later chapters.

2.1 LITERATURE REVIEW

This section presents a review of the literature that is divided further into three parts:

1. iterative methods,
2. fault tolerance, and
3. performance modeling.

The studies that are discussed in the subsection on iterative methods covers recent work for both classical (i.e. synchronous) and asynchronous iterative methods. Again, these studies are important to build a foundation for all of the work presented in this thesis. Particular care is taken to highlight the historical development of asynchronous iterative methods including the resurgence in interest in these methods in recent years, and to show how the results that have been obtained relate to the results that are presented here. Chapters 3 and 5 most directly relate to these studies, but the results do carry significant weight to all of the work presented in this dissertation.

The section on fault tolerance is designed to capture results from the recent literature that detail the increased expectation of faults to occur and to discuss some high level techniques that have been employed for mitigating faults. Additionally, a subsection is included that

details recent work in the numerical modeling of faults. This work is relevant throughout all of this dissertation, but also directly applies to the work shown in Chapter 4.

The next section on performance modeling covers a wide variety of techniques and frameworks that have been used to attempt to extrapolate results forward to future hardware and software environments. This work is pertinent to the new work presented in Chapter 6.

2.1.1 ITERATIVE METHODS

2.1.1.1 Classical Iterative Methods

The work done in this dissertation to show the effectiveness of many of the linear solvers is built upon one of the more popular Krylov subspace methods (see the classic text by Saad [34] for an overview). This class of methods includes two of the main methods that are focused on here. The first is the Conjugate Gradient method, used when the associated linear system is symmetric positive-definite (SPD), and the second is GMRES [35], developed by Saad and Schultz and useful for non-symmetric or highly indefinite linear systems. The "flexible" variant of GMRES, i.e. FGMRES – also developed by Saad, is used when appropriate as well [36]. FGMRES is very similar to GMRES with the notable exception of allowing the preconditioner to change adaptively at each iteration. More details are provided in the pertinent background section (Section 2.2).

Krylov subspace solvers have been studied extensively. Van der Vorst provides an overview of how these solvers can be nested for improved convergence [37], a similar approach in spirit to the FGMRES solver [36] that is used in the studies shown in Chapter 4. Applications in computational fluid dynamics (CFD) have been considered extensively in the past (see [38]–[41]); this application domain is a great example of problems that could benefit from the resilient solvers developed here and the work in Chapter 5 includes an example of a problem from the CFD domain. A study by Sosonkina, Saad, and Cai [42] provides examples of the use of Krylov subspace solvers in many other realistic domain areas and introduces

several computational manipulations that can be used to help with convergence such as the idea of a positive diagonal shift. This last idea is experimented with in Chapter 5 to see the effect it has on the use of nonlinear fixed point iterations for non-symmetric problems.

Perhaps most relevant to the work presented in this dissertation is the work on inexact Krylov subspace methods. These are a class of iterative methods that examine whether the computationally expensive operations during a Krylov subspace method can be replaced with approximations. For example, one of the most computationally expensive operations during the iteration is the matrix-vector multiply that is used to expand the basis for the Krylov subspace. If instead of performing the multiplication exactly the computation is performed approximately, i.e. instead of computing

$$w = Av \tag{2}$$

one can instead compute the matrix-vector multiplication approximately which results in an equation of the following form,

$$w \approx Av + Ev \tag{3}$$

where E is an appropriate error matrix. In essence, the computation may be sped up by allowing computationally expensive routines to be computed approximately without any significant degrade in convergence. An overview of this idea is given by van den Eshof [43]. The study presented by Simoncini and Szyld [44] investigates what properties the matrix E must have, and their later paper [45] looks into the instances when using approximate computation can actually speed up convergence as well as what these approximate computations can have on the Krylov subspace itself. This is relevant to a study on fault tolerance for Krylov subspace solvers since a fault can be viewed by some fault models as a perturbation that causes an operation, such a matrix-vector multiply, to become inexact even if the routine is computed exactly. Further, instance of superlinear convergence such as those studied in [45]

were seen in the work on the effect of persistent faults on Krylov subspace solvers that is detailed in Section 4.2.2.

Another area of interest is in the use of accelerators (e.g., Intel Xeon Phis®or GPUs) for iterative methods. The study by Li and Saad [46] shows how GPU-acceleration can be used for preconditioned iterative linear solvers, including both the Conjugate Gradient and Flexible GMRES algorithm which are both featured prominently in the work presented in this dissertation. The work shown by Jamal et al. [47] presents a technique for creating a hybrid parallel iterative linear solver based upon the pARMS library and solver [48]–[50] which is also featured heavily in the experiments that are showcased in Chapter 4.

Research into the convergence of iterative methods when solving non-symmetric problems has been studied previously as well. Chow and Saad [51] present an experimental study on the convergence of Krylov subspace solvers with various incomplete LU factorizations, with a focus on the performance of the algorithm used to generate the incomplete factorization. They enumerate a large number of problems that cannot be solved with the baseline incomplete LU factorization (i.e. ILU(0)) and investigate methods for improving the performance of both the algorithm to generate the incomplete factorization and the associated Krylov Subspace solver.

Benzi et al. [52] also study convergence of non-symmetric and indefinite matrices. They were motivated by problems from a wide variety of application areas through science and engineering (e.g. chemical engineering, economic modeling, circuit simulation, etc) where the structure of the discretized system is not as nice as the structure of problems arising from the discretization of many common elliptic partial differential equations. Similar to [51], the authors present experimental results for a wide variety of problems. Their experiments use a variety of different preconditioning techniques (including an approximate inverse algorithm not similar to anything used in [51]) and they analyze the results to attempt to find beneficial modifications that can be made to the problem (e.g., reorderings, scalings, etc). Both of these studies [51], [52] were used as a starting point for the work presented here on non-symmetric

problems.

The effect of matrix reordering on convergence was studied in the previously mentioned works on non-symmetric problems (i.e. [51], [52]) and has been focused on exclusively in papers such as a study of the effect of reorderings on incomplete factorizations by Benzi, Szyld, and Van Duin [53]. That work focuses on how different reorderings affect the convergence of Krylov subspace solvers for non-symmetric problems. In their study, a variety of different solvers and incomplete LU factorizations are used. The analysis presented there inspired some of the work done here to judge how matrix reorderings affect the nonlinear fixed point iteration for generating incomplete LU factorizations (see Chapter 5).

2.1.1.2 Asynchronous Iterative Methods

Fine-grained parallel methods, specifically parallel fixed point methods, are an area of increased research activity due to the practical use of these methods on HPC resources. An initial exploration of fault tolerance for stationary iterative linear solvers (i.e. Jacobi) is given by Anzt, Dongarra, and Quintana-Ortí [26] and expanded on in their later study [21]. Fault tolerance for synchronous fixed point algorithms from a numerical analysis has been investigated by Stoyanov and Webster [54]. Error correction and mixed precision techniques for GPU based oriented asynchronous methods were investigated by Anzt et al. [55].

The general convergence of parallel fixed point methods has also been explored extensively. Addou and Benahmed present an overview of parallel nonlinear fixed point iterations with an emphasis on synchronous results [56], and Benahmed later described the specific extensions needed to ensure the results extend to the scenario of asynchronous updates [57]. A survey that presents a wide range of different methods is given by Frommer and Szyld [58].

The general theory of parallel fixed point methods is captured well by the seminal textbook by Bertsekas and Tsitsiklis [59]. Results specific to parallel nonlinear fixed point methods can be obtained from the class text by Ortega and Rheiboldt [60].

The class of asynchronous iterative problems that the simulation framework proposed in

this dissertation (see Chapter 6) addresses are stationary solvers (also referred to as relaxation methods). The focus is on the behavior of these methods in asynchronous computing environments; however, the framework also easily admits synchronous updates; the key is the fine-grained nature of the algorithm. Fine-grained parallel methods, specifically parallel fixed point methods, are an area of increased research activity due to the practical use of these methods on HPC resources.

While many recent research results for asynchronous iterative methods are focused on implementations that utilize a shared memory architecture, one area of asynchronous iterative methods that has seen significant development using a distributed memory architecture is optimization. Cheung and Cole provide an asynchronous coordinate descent algorithm [29], Hong developed a distributed asynchronous ADMM routine specific to nonconvex problems [61], and Boyd et al. present an asynchronous approach to the alternating direction method of multipliers (ADMM) routine designed to apply to machine learning problems [62]. Rechet et al. [27] and Tsitsiklis et al. [11] present investigations into changing the nature of steepest descent (i.e. gradient descent) optimization routines to make them suited for asynchronous behavior. Lastly, both Zhong and Cassandras [63] and Srivastava and Nedic [64] focus on the communication patterns needed for distributed asynchronous optimizations.

Nonlinear fixed point iterations have also found a use in modern practical applications. For example, the fine-grained parallel incomplete LU (FGPILU) factorization uses a nonlinear fixed point iteration to compute incomplete factorizations that can be used as preconditioners for more traditional linear solvers; Chow and Patel describes the algorithm itself [24], while Chow, Anzt, and Dongarra describe how the algorithm may be implemented efficiently on GPUs using what is referred to as a block-asynchronous method [23]. Later, Anzt et al. showed how this algorithm could be used efficiently when solving a series of related linear systems in model order reduction applied to physical problems [65]. Note that this algorithm is the focus of the chapter of this dissertation (see Chapter 5) that shows how the theory developed earlier can be used to make the necessary algorithmic modifications to an

asynchronous algorithm to ensure fault tolerance.

Asynchronous methods themselves have a long and storied history. The development initially began in earnest in the late 1960's and continued into the 1970's. The paper that started investigation into asynchronous iterative methods was by Chazan and Miranker [66] and investigated whether the updates when solving a linear system via a relaxation method (e.g., Jacobi) could occur in a random, uncoordinated manner that they termed "chaotic". This work was immediately expanded on by other authors, see the work on developing periodic chaotic relaxations documented by Donnelly [67].

Development continued throughout the 1970's. Work of note includes a generalization of the results presented by Chazan and Miranker to nonlinear operators by Miellou [68] and the paper by Kung [69] that provides an overview of research effort on parallel synchronous and asynchronous algorithms to that date (1976). Much of the work on nonlinear asynchronous iterative methods stems from the work by Ortega and Rheinboldt [60] that analyzed (synchronous) nonlinear operators of several variables in great depth. Later, work continued towards further generalizing results in the seminal work by Baudet [70], where a framework for analyzing asynchronous iterative methods was proposed that can still be seen in the modern frameworks (including the one used throughout this dissertation). The results for linear and nonlinear operators were reframed so that the assumption of bounded delay was unnecessary, and some initial work to analyze and bound the convergence rate of asynchronous iterative methods was documented. A related work by the same author provides an initial set of guidelines that can be used in the development of asynchronous iterative methods [71].

In the 1980's there came a focus on detailing criteria for convergence (i.e. establishing the necessary conditions for proposed algorithms to have eventual convergence) [72], [73] and termination conditions [74], [75]. Of note due to the relation of the subject matter to the algorithms studied here, Tsitsiklis analyzed differences in the convergence rate of Jacobi and Gauss-Seidel in the asynchronous paradigm [76]. Additional work during this time fo-

cused on adapting the general results put forth previously towards specific algorithms: for example, Bojańczyk analyzed an asynchronous implementation of Newton's method for optimization [77], Anwar and El Tarazi examined equations specific to the solution of Poisson's equation [78], Spiteri looked into asynchronous methods for boundary value problems [79], and Smart and White showed results for circuit problems [80]. Asynchronous implementations of gradient algorithms also began to be explored for the first time, see [11], [81], [82].

General results continued to be put forth throughout the 1980's in an effort to provide incremental results to continually try to further the theory. The two main techniques used to prove convergence of an asynchronous algorithm were both introduced in the early 1980's: Bertsekas introduced the idea of finding a suitable sequence of nested sets [83] while El Tarazi brought forth the idea of using an appropriate weighted maximum norm [84]. General results for distributed networks were developed by Bertsekas [85], generalized asynchronous iterations were studied by Uresin and Dubois [86], and results for non-negative matrices were introduced by Lubachevsky and Mitra [87] during this time period. The last study to be noted during this time is a study on the stability of this class of methods by Tsitsiklis [88] that provided an investigation into the sufficient conditions needed for convergence that is similar to Lyapunov stability theory.

Work on asynchronous methods picked up during the 1990's and early 2000's on both the theoretical and applied results. A survey was provided in the early 1990's by Bertsekas and Tsitsiklis that captured results to that point [89], general convergence results were examined [90], [91], studies were conducted for linear (or almost linear) equations [92]–[95], two-stage iterations were introduced [96], [97], and termination conditions were examined [98].

A few of the different mathematical models that have been used to analyze asynchronous iterative methods historically were compared by Szyld [99], who also investigated "the mystery of asynchronous iterations convergence when the spectral radius is one," i.e. for operators that are not strictly contractive. This leads into the work that was done to expand existing results to broader classes of iterative methods. Some of the strong results

for contractive linear systems were expanded to apply to nonexpansive linear systems by Bahi [100], to parallel synchronous algorithms by Addou and Benahmed [56] and then finally to fully asynchronous algorithms by Benahmed [57]. Additionally, asynchronous iterative methods specific to the solution of partial differential equations began to be examined more [101], [102], and stability of the solutions of these methods was revisited [103]. A probabilistic approach governing which component each processor should update was introduced by Strikwerda [104], which served as a precursor to the much more recent work by Avron, Druinsky, and Gupta that proved bounds on convergence rates for this stochastic class of asynchronous iterative methods [105]. A few final works of particular note from this era are the great survey of classical and modern results provided by Frommer and Szyld [58]), and the text by Bahi [106] that captures many results on parallel iterative algorithms, including a large section on asynchronous iterative methods.

While the incredible amount of work contributed to the field from the first paper in 1969 until the mid 2000's provides a firm foundation that will be referenced and called upon frequently in both this work and other modern work in the field of asynchronous methods, the development of asynchronous results become more pertinent to the work of this dissertation with the influx of work performed in roughly the last 10 years. The more recent studies dedicated to this field tend to be related to the goal of removing the synchronization delay from high performance computing clusters that are comprised of heterogeneous components, e.g. systems that make use of co-processors such as GPUs or Intel Xeon Phi®'s.

Additional recent efforts include performance analysis of asynchronous methods by Bethune et al. [107], [108], as well as similar analysis presented by Hook and Dingle [109]. Anzt et al. explored the suitability of GPUs for use with asynchronous iterative methods [110], [111], as well as performance tuning of block-asynchronous methods [112]. Asynchronous methods have also been modified to efficiently execute triangular solves [25], [113], very commonly used computations in numerical linear algebra. Asynchronous optimization methods have also been explored; Recht et al. proposed a gradient descent method [27], Cheung and Cole stud-

ied coordinate descent [29], and Aybat et al. studied methods for convex optimization [28]. Magoules et al. investigated techniques for solving partial differential equations [30]. In an effort to help ease the development of asynchronous iterative methods, recent years have seen several modifications to existing parallel programming paradigms (such as the Message Passing Interface, MPI). These include Casper [114], JACK [115], and a direct modification to MPI [116].

2.1.2 FAULT TOLERANCE

The expected increase in faults for future HPC systems is detailed in a variety of sources. The general expectation is that as HPC platforms continue to evolve towards exascale levels of computation, they will become more prone to errors. Geist published a famous article detailing the expected increase in failure rate from a reasonably non-technical point of view that is available in the various versions of the “Monster in the Closet” talk [117] and paper [118], as well as an article for a more general audience [119].

More technical and detailed reports that speak to the increase in faults are given in a variety of sources composed of groups of different researchers from both academia and industry. A report from the University of California, Berkeley led by Asanovic et al. famously identifies the “seven dwarfs” facing modern HPC [6]. A group of academics from various institutions put together a report in 2009 that listed the challenges that would need to be overcome as the HPC community moved towards exascale levels of computation [4] and then provided an update in 2014 [5]. Geist and Lucas provide a detailed list and discussion of the major challenges that will need to be overcome on the road to exascale [7], and Snir et al. specifically address the manner in which faults and failures in the computing environment will need to be dealt with [8].

Additionally, the Department of Energy has commissioned two very detailed reports about the progression towards exascale level computing, one from a general computing standpoint [1] (summarized in [2]), and a report aimed specifically at applied mathemat-

ics for exascale computing [3]. Both reports provide detailed road maps that the authors believe will need to followed in order to achieve successful use of exascale level supercomputers. Changes to the underlying parallel framework (e.g. MPI) have been considered as an alternative to direct modification of the algorithm under analysis. These include work by Fagg et al. on Fault Tolerant MPI extensions (FT-MPI) [120], [121], which subsequently evolved into an effort by Bland et al. to extend the MPI standard in an effort called User Level Failure Mitigation (ULFM) [122], [123], as well as work similar to that by Zheng et al. on fault tolerant extensions to MPI as it interfaces with Charm++ [124].

The variants of the fine-grained algorithms that are discussed in this dissertation build upon ideas from various methods for fault tolerance. Sao and Vuduc proposed the idea of self-stabilizing iterative algorithms [125], where the process naturally corrects any faults that occur. Bridges et al. put forth the idea of selective reliability whereby certain computations are computed in “high reliability” mode with the expectation that they will not be negatively effected by a fault and others are computed in a faster “low reliability” mode where computations are not protected, but may be able to be executed faster [31], [32].

2.1.2.1 Fault Tolerance for Classical Iterative Methods

Fault tolerance for traditional iterative methods (i.e. both stationary solvers and Krylov subspace solvers) has been studied extensively in recent years. Characterization of the effects of the faults on such solvers has been conducted by both Bronvetsky and de Supinski [126] and Shantharam et al. [127], and fault detection for iterative methods in linear algebra has been studied as well by Chen [128] and Sloan, Kumar, and Bronevetsky [129]. Fault tolerance for specific iterative methods has also been studied; see for example work on modifications to the FGMRES algorithm by Hoemmen and Heroux [31] and modifications to the Conjugate Gradient algorithm by Shantharam et al. [127]. Elliott, Hoemmen, and Mueller conducted a study on the development of reliable fault detectors [130], as well as an investigation into how much data corruption (i.e. due to a fault) can be tolerated while still ensuring convergence

of the iterative method [131].

The use of a periodic correction step is one alternative class of methods for fault tolerance proposed by Sao and Vuduc [125] that offers several advantages. First, these methods provide a way to avoid the cost of checkpointing which has been suggested to be prohibitively high on future exascale platforms [4], [5], [7]. Second, they do not necessarily rely on any sort of fault detection. If a fault is not detected successfully in a traditional checkpointing algorithm it can cause catastrophic effects, a self-stabilizing method based upon a periodic correction step should be designed in such a way that it will return a valid answer without falling back on traditional fault detection mechanisms.

2.1.2.2 Numerical Soft Fault Modeling

Traditionally, when performing experiments to analyze the potential impact of soft faults upon a computing environment, researchers have relied primarily upon the injection of bit flips into a particular portion of the routine (see, for example [126], [132]). The idea of treating faults as numerical corruption as opposed to attempting emulate the manner that a fault currently occurs has gained momentum over the last decade. A general position paper by Elliot, Hoemmen, and Mueller on the efficacy of treating soft faults has numerical corruption was provided [133] that outlines several reasons for adopting this approach, and several numerically based fault models have been utilized in recent studies. These include a "numerical" fault model that is predicated on shuffling the components of an important data structure by Elliot et al. [134], a perturbation based model put forth by Stoyanov and Webster [54], and an approach that induces a small shift to a single component of a vector [31], [32]. This numerical approach models the impact of soft faults by disregarding the actual source of the fault and allowing the fault injector to create as large or as small a fault as necessary for the experiment.

Whether a given study chooses to model faults using bit flips or adopt a more numerical analysis style approach, much of the previous work on the impact of silent data corruption

(SDC) has to do with modeling transient errors. One of the goals of this effort is to present a fault model that can accurately predict the impact of either sticky or persistent soft faults. The models presented here is general enough that they can be adapted to simulate the impact of any persistent error, including those caused by hardware malfunction. Traditional analysis of potential persistent type errors has rested more in the hardware domain than in the algorithmic domain, with analysis of both processor based faults by both Li et al. [135] and Bower et al. [136] and memory based faults by Schroeder et al. [12], though the impact of persistent faults on iterative methods does not seem to have been explored to a great extent.

2.1.3 PERFORMANCE MODELING

Development of computational frameworks for the purposes of simulating performance has a long history in the literature. Examples of such frameworks include SimGrid by Casanova et al. [137], [138] which focuses on modeling the performance of distributed experiments, GangSim by Dumitrescu and Foster [139] and GridSim by Buyya and Murshed [140] which both focus on grid scheduling, as well as CloudSim [141], [142] which models performance of cloud computing environments. These environments focus on specific HPC implementation features, such as job scheduling and data movement, and attempt to provide a view of how the systems themselves behave in HPC-like scenarios.

Bahi et al. demonstrate the efficacy of asynchronous methods, especially for grid systems, and propose a system for classifying parallel iterative algorithms, based on computational and communication strategies [143], [144]. Jager and Bradley demonstrate superior performance of asynchronous methods for solving large sparse linear fixed-point problems [145]. Voronin compares three parallel implementations using MPI and OpenMP®[1], with asynchronous threads, and finds that utilizing a “postman” thread within each computational node to perform communications delivers superior performance, compared to the alternative hybrid

[1]OpenMP Architecture Review Board, Austin, TX

MPI-OpenMP®implementation [146].

2.2 KRYLOV SUBSPACE SOLVERS

Krylov subspace solvers are a popular class of iterative methods for solving the sparse linear systems that arise naturally in many domain areas of science and engineering. An in-depth tutorial is beyond the scope of this dissertation; however, a brief introduction is warranted since two different Krylov subspace solvers are used in the various studies presented in this dissertation (see Chapters 4 and 5).

It is possible to derive this class of solver entirely from an optimization point of view, but the approach here is to follow the text [34] and introduce them as projection based methods. When solving the linear system,

$$Ax = b \tag{4}$$

where $A \in \mathbb{R}^{n \times n}$ the idea behind projection based techniques is to search for the solution in a subspace $\mathcal{K}$ of dimension m where $m < n$. In order to uniquely define a solution in the search subspace, a total of m constraints must be imposed. This can be done by choosing m independent orthogonality constraints on the solution vector. These m orthogonality constraints define another subspace, $\mathcal{L}$, that contains the m vectors that the solution must be orthogonal to. These orthogonality requirements are known as the Petrov-Galerkin condition.

The idea behind projection based methods is that the solution $x^* \in \mathcal{K}$ is sought such that the residual, $r^* = b - Ax^*$, is orthogonal to $\mathcal{L}$. If an initial guess x^0 is to be used, then the solution x^* is searched for in the affine space $x^0 + \mathcal{K}$.

Krylov subspace methods have in common that they all search for the solution vector x^* in the Krylov subspace, i.e. the subspace defined by,

$$\mathcal{K} = \mathcal{K}_m\left(A, r^{(0)}\right) = span\{r^{(0)}, Ar^{(0)}, A^2r^{(0)}, \ldots, A^{m-1}r^{(0)}\}. \tag{5}$$

Broadly speaking, where the various methods differ is in how they define the subspace $\mathcal{L}$.

To help improve the performance of sparse linear solvers solvers, such as Krylov subspace solvers, a preconditioner is often used to help accelerate convergence [34], [147]. The use of a preconditioner transforms the system of linear algebraic equations into a preconditioned system. A preconditioned system writes the general linear system of equations

$$Ax = b \tag{6}$$

in the form

$$M^{-1}Ax = M^{-1}b, \tag{7}$$

when preconditioning is applied from the left, and

$$AM^{-1}y = b \tag{8}$$

with $x = M^{-1}y$, when preconditioning is applied from the right. The matrix M is a nonsingular approximation to A, and is called the *preconditioner*.

2.2.1 CONJUGATE GRADIENT

The Conjugate Gradient (CG) method selects the subspace of orthogonality constraints to be the Krylov subspace itself, i.e., $\mathcal{L} = \mathcal{K}_m\left(A, r^{(0)}\right)$. If the matrix A is symmetric and positive-definite (SPD) then this choice of $\mathcal{L}$ minimizes the A-norm of the error [34], [148]. Because of this, the CG method is typically used when A is SPD. In the case that A is non-symmetric, this choice of $\mathcal{L}$ defines the full orthogonalization method (FOM) which is mathematically equivalent to CG [34]; however, the symmetry of A can be used to help lessen the memory requirements of the algorithm.

The right-preconditioned CG algorithm, as described in [34], p. 263 is provided in Algorithm 1.

Algorithm 1: Conjugate Gradient algorithm

Input: A linear system $Ax = b$, a preconditioner M, and an initial guess at the solution, x_0

Output: An approximate solution x_j for some $j \geq 0$

1 $r_0 := b - Ax_0$,
2 $z_0 := M^{-1}r_0$
3 $p_0 := z_0$
4 **for** $j = 1, 2, \ldots$ **do**
5 $\quad\alpha_j = (r_j, z_j)/(Ap_j, p_j)$
6 $\quad x_{j+1} = x_j + \alpha_j p_j$
7 $\quad r_{j+1} = r_j - \alpha_j Ap_j$
8 $\quad z_{j+1} = M^{-1}r_{j+1}$
9 $\quad\beta_j = (r_{j+1}, z_{j+1})/(r_j, z_j)$
10 $\quad p_{j+1} = z_{j+1} + \beta_j p_j$

2.2.2 FLEXIBLE GMRES

The General Minimal Residual method (GMRES) and its variants are all defined by the choice of $\mathcal{L} = A\mathcal{K}$ where $\mathcal{K}$ is the Krylov subspace. This choice of $\mathcal{L}$ minimizes the 2-norm of the residual. This method is very popular for general (i.e. not necessarily SPD) systems. The right-preconditioned GMRES algorithm, as described in [34], p. 269 is provided in Algorithm 2.

The flexible variant of GMRES (FGMRES) is similar in its nature to the standard GMRES with the notable exception of allowing the preconditioner to change at each iteration by storing the result of each preconditioning operation (cf. matrix Z_m in Line 11 of Algorithm 3). In the studies conducted here, FGMRES was selected because it is a robust solver proven to converge under variable preconditioning, including converging in situations where the variability comes as a result of some sort of anomaly in the preconditioning operation. Here, such an anomaly may be due to a fault injected during the experiments. The right-preconditioned FGMRES algorithm, as described in [34], p. 273 is provided in Algorithm 3.

In particular, in the studies presented throughout this dissertation, faults were injected

Algorithm 2: GMRES algorithm

Input: A Linear system $Ax = b$ and an initial guess at the solution, x_0
Output: An approximate solution x_n for some $n \geq 0$

1 $r_0 := b - Ax_0,$
2 $\beta := ||r_0||_2, v_1 := r_0/\beta$
3 **for** $j = 1, 2, \ldots, m$ **do**
4 $\quad z_j = M^{-1}v_j$
5 $\quad w = Az_j$
6 $\quad$**for** i = *1, 2, . . . , j* **do**
7 $\quad\quad h_{i,j} := w \cdot v_i$
8 $\quad\quad w := w - h_{i,j}v_i$
9 $\quad h_{j+1,j} := ||w||_2, v_{j+1} := w/h_{j+1,j}$
10 $\quad V_m := [v_1, \ldots, v_m], \bar{H}_m := h_{i,j_{1\leq i\leq j+1;1\leq j\leq m}}$
11 $y_m := argmin_y||\bar{H}_m y - \beta e_1||_2, x_m := x_0 + M^{-1}V_m y_m$
12 **if** *Convergence was reached* **then**
13 $\quad$**return** x_m
14 **else**
15 $\quad$go to to Line 1

Algorithm 3: Flexible GMRES algorithm

Input: A linear system $Ax = b$ and an initial guess at the solution, x_0
Output: An approximate solution x_m for some $m \geq 0$

1 $r_0 := b - Ax_0,$
2 $\beta := ||r_0||_2, v_1 := r_0/\beta$
3 **for** $j = 1, 2, \ldots, m$ **do**
4 $\quad z_j = M_j^{-1}v_j$
5 $\quad w = Az_j$
6 $\quad$**for** i = *1, 2, . . . , j* **do**
7 $\quad\quad h_{i,j} := w \cdot v_i$
8 $\quad\quad w := w - h_{i,j}v_i$
9 $\quad h_{j+1,j} := ||w||_2$
10 $\quad v_{j+1} := w/h_{j+1,j}$
11 $\quad Z_m := [z_1, \ldots, z_m]$
12 $\quad \bar{H}_m := h_{i,j_{1\leq i\leq j+1;1\leq j\leq m}}$
13 $y_m := argmin_y||\bar{H}_m y - \beta e_1||_2$
14 $x_m := x_0 + Z_m y_m$
15 **if** *Convergence was reached* **then**
16 $\quad$**return** x_m
17 **else**
18 $\quad$go to line 1

at two distinct locations inside the FGMRES algorithm: Line 1, called here the *outer matvec* operation, and Line 4, which is the application of the preconditioner M. These locations were selected since they are two of the most computationally expensive operations inside of the algorithm, and therefore the risk of a fault occurring during these operations is naturally higher since the algorithm will spend more time executing them.

2.2.2.1 Fault Detection and Resilience in FGMRES

Fault detection inside of FGMRES can be achieved in many different ways. Upon each restart of FGMRES, the norm of the residual is computed, and in a fault-free environment these norms should be monotonically decreasing. A cheap fault detector could be implemented to check that the progression of the norm of the residual is in fact non-increasing [34]. This would be an intuitive way to attempt to detect faults that occur during the outer sparse matrix-vector multiply. Another way to detect errors was proposed in [130] and consists of bounding the entries of the upper Hessenberg matrix, H by either $||A||_2$ or $||A||_F$ in an attempt to detect faults that cause the values of H to be larger than is theoretically possible.

It will be shown experimentally (see Section 4.2.2) that if the fault that is injected into the outer sparse matrix-vector multiply does not increase the norm of the initial residual, then it has a significantly less negative effect on the convergence of FGMRES than a similar fault injected into the preconditioning operation. Generally speaking, the effect that a fault may have on the norm of the data structure inside the Krylov subspace solver is indicative of the effect that it may have on the overall performance of the solver [130], [131], [149].

One of the key observations made in [32] was that since the preconditioner is allowed to change on every iteration in the FGMRES algorithm, faults that occur during the preconditioning operation (Line 4 in Algorithm 3) can be modeled as different preconditioners. As such, if a fault were to occur anywhere inside of the preconditioning operation it can be modeled by injecting a fault into the result of the preconditioning operation (z_j in Algorithm 3). The numerical soft fault models used in these studies allow the size of the fault to

be controlled by offering direct control on the size of the perturbation that is injected.

Further, it will be shown that FGMRES is capable of proceeding through many faults occurring in the preconditioning operation by accepting the faulty output as a different preconditioner. This natural adaptive response in the FGMRES algorithm to faults that occur during preconditioning should also cause faults that occur during the outer sparse matrix-vector multiply to have more of an impact on the convergence of FGMRES. This was also shown experimentally, and results are provided in Section 4.2.2.

2.3 PRECONDITIONERS

2.3.1 INCOMPLETE FACTORIZATIONS

Incomplete LU factorization methods (ILUs) are effective preconditioning techniques for solving linear systems. In this case, the preconditioning matrix M has the form,

$$M = \bar{L}\bar{U}, \tag{9}$$

where $\bar{L}$ and $\bar{U}$ are approximations to the L and U factors of the standard LU decomposition of A. The incomplete factorization may be computed using the Gaussian elimination algorithm, by discarding some entries in the L and U factors. In the ILUT preconditioner used throughout the experiments, a dual non-zero threshold (τ, ρ) is used. In particular, this causes all computed values that are smaller than $\tau||a_i||_2$ to be dropped, where $||a_i||_2$ is the norm of a given row of the matrix A, and only the largest ρ elements of each row are kept.

Typically, to generate a complete LU factorization of a given matrix A such that

$$A = LU, \tag{10}$$

a Gaussian elimination process is used. However, when this process is carried out, fill-in will usually occur. This causes the triangular factors L and U to tend to have significantly more

non-zero elements. This destruction of sparsity can be prohibitive when solving large sparse problems (for example, those arising from three-dimensional boundary value problems [147]) due to space and time constraints. An example of the amount of fill-in that is possible during the process of finding complete L and U factors is provided by Fig. 4. In this example, the initial matrix A is taken to be a three dimensional finite-difference approximation of the Laplacian,

$$\Delta u = f, \tag{11}$$

over a $50 \times 50 \times 50$ grid using a nine-point stencil. Note that the number of non-zero terms increases from 3.3 million elements in A to 312.9 million elements in *both* L and U, respectively, after the factorization is performed. Even more drastic examples of fill-in are possible for many other problems throughout science and engineering.

To avoid this effect, an incomplete LU factorization is typically computed instead. An incomplete factorization process generates an approximate factorization of the matrix A such that,

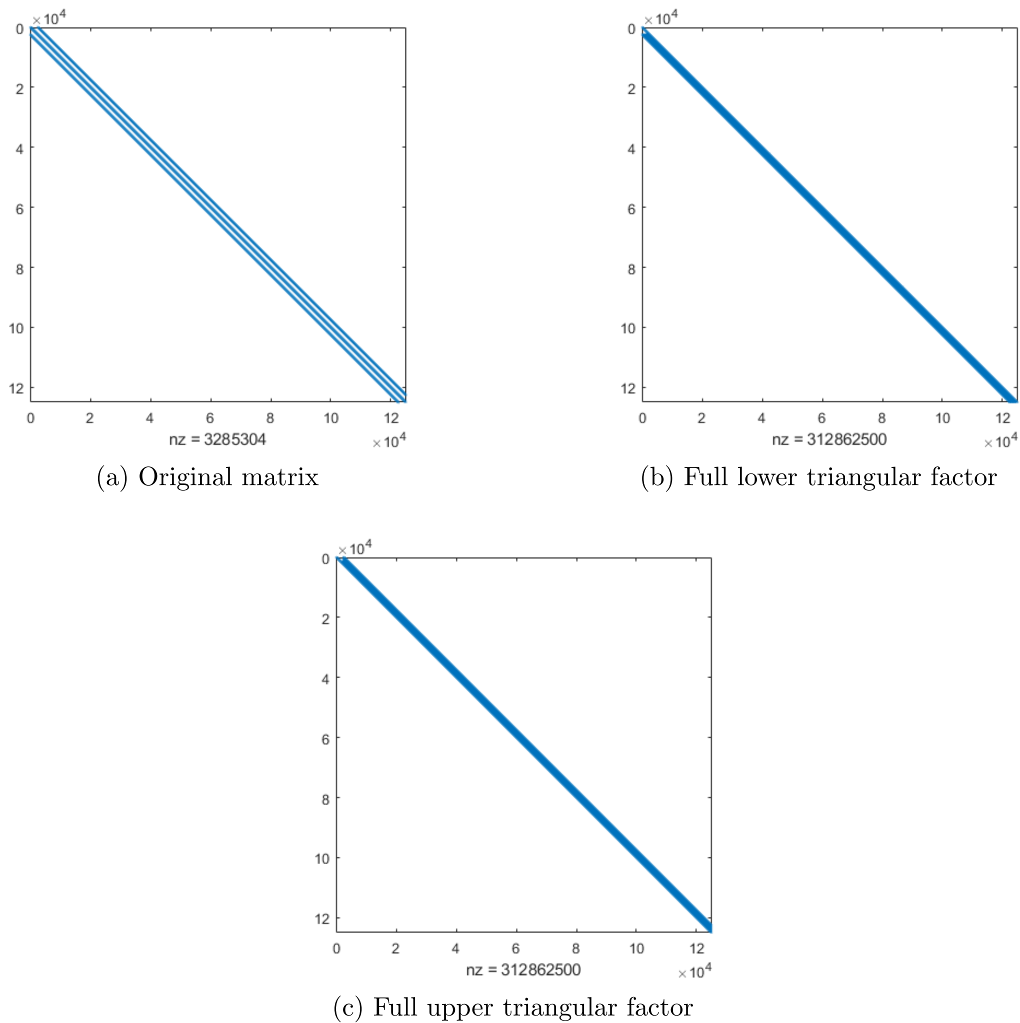
$$A \approx LU. \tag{12}$$

While this incomplete factorization cannot be used to solve a linear system directly (as the exact LU factorization can), it can be used as a *preconditioner* that helps to accelerate the convergence of an iterative method for solving linear systems. For example, when solving a linear system,

$$Ax = b, \tag{13}$$

the full LU factorization can be used to reduce the system to,

$$LUx = b, \tag{14}$$



(a) Original matrix

(b) Full lower triangular factor

(c) Full upper triangular factor

Fig. 4: Example of the effects of fill-in during the Gaussian Elimination process. Note the relative sparsity of the input matrix A (Fig. 4a) compared to the factors L and U (Figs. 4b and 4c respectively), especially in the total number of non-zero elements in each matrix.

which can be solved completely with two triangular solves. In the case of an incomplete factorization, a nonsingular approximation to A can be used to transform the given linear system, described by Eq. (13), into one that is easier to solve. In particular, the linear system

$$M^{-1}Ax = M^{-1}b \tag{15}$$

will have the same solution as the original system but may be easier to solve, especially when

used in conjunction with an iterative method. In the case of an incomplete LU factorization, the incomplete LU factors that are obtained can be used to create this approximation, i.e., $M = LU$.

In order to create an incomplete factorization, first define a set S which specifies the locations of the non-zero elements in the incomplete factorization. Specifically, if $(i, j) \in S$ then there will be a non-zero at the corresponding location in either the factor L if $i > j$, or U if $i < j$. Given this set, an algorithm that provides incomplete factorization of a matrix A is given by Algorithm 4. Note that the set S can be defined before the start of the algorithm, or can be updated dynamically over the course of the algorithm.

Algorithm 4: Conventional Incomplete LU Factorization

```
   Input: Input matrix A
1  for i = 1, 2, ..., n do
2      for k = 1, 2, ..., i − 1 and (i, k) ∈ S do
3          a_ik = a_ik / a_kk
4          for j = k + 1, k + 2, ..., n and (i, j) ∈ S do
5              a_ij = a_ij − a_ik a_kj
```

The major problem with this type of algorithm is the difficulty in parallelizing it. Reordering the matrix can introduce more parallelism, although often parallelism is limited below a level that would be desired for the scale of problems that are considered. Alternatively, several variants of conventional incomplete LU factorization have been proposed in an attempt to increase the benefit of the preconditioner (see e.g., ILUT [150], ILUM [151], BILUTM [152], among many others).

2.3.1.1 Algebraic Recursive Multilevel Solver (ARMS)

The next preconditioner to be introduced is the Algebraic Recursive Multilevel Solver (ARMS). This preconditioner is considered since it serves as an example of a type of precon-

ditioning that is more powerful (i.e., able to accelerate the convergence of a Krylov subspace method more effectively) than the traditional incomplete factorization preconditioners introduced in the previous subsection.

Several different multi-level ILU preconditioners take advantage of the fact that sets of unknowns that are not coupled to each other can be eliminated simultaneously in Gaussian elimination; these collections of unknowns are commonly referred to as independent sets. A block independent set is a set of groups (blocks) of unknowns such that there is no coupling between unknowns of any two different groups (blocks) [151], [153]. For a given linear system, represented by the matrix A, that contains n linear algebraic equations (and therefore n unknowns), if m of the independent unknowns are numbered first, and the other $n-m$ unknowns last, the coefficient matrix of the system is permuted in a 2×2 block structure.

$$PAP^T = \begin{pmatrix} D & F \\ E & C \end{pmatrix}, \tag{16}$$

where D is a diagonal matrix of dimension m that contains the m independent unknowns, C is a square matrix of dimension $n-m$ that contains the remaining variables, and P is the appropriately chosen permutation matrix.

In multi-elimination methods [34], p. 392, a reduced system is recursively constructed from the permuted system performing a block LU factorization of PAP^T as follows

$$PAP^T = \begin{pmatrix} D & F \\ E & C \end{pmatrix} \approx \begin{pmatrix} L & 0 \\ G & I_{n-m} \end{pmatrix} \times \begin{pmatrix} U & W \\ 0 & A_1 \end{pmatrix}, \tag{17}$$

where P is a permutation matrix, D is a diagonal matrix (or block-diagonal if independent sets of unknowns are used), L and U are the triangular factors of the LU factorization of D, and

$$A_1 = C - ED^{-1}F \tag{18}$$

is the Schur complement with respect to C, I_{n-m} is the identity matrix of dimension $n-m$, $G = EU^{-1}$ and $W = L^{-1}F$.

As a visual example of this process, consider the 3D convection-diffusion problem discretized using finite differences. A sparsity plot of the original matrix is shown next to an image of this first level of decomposition, which is generated by software tools [154] in Fig. 5.

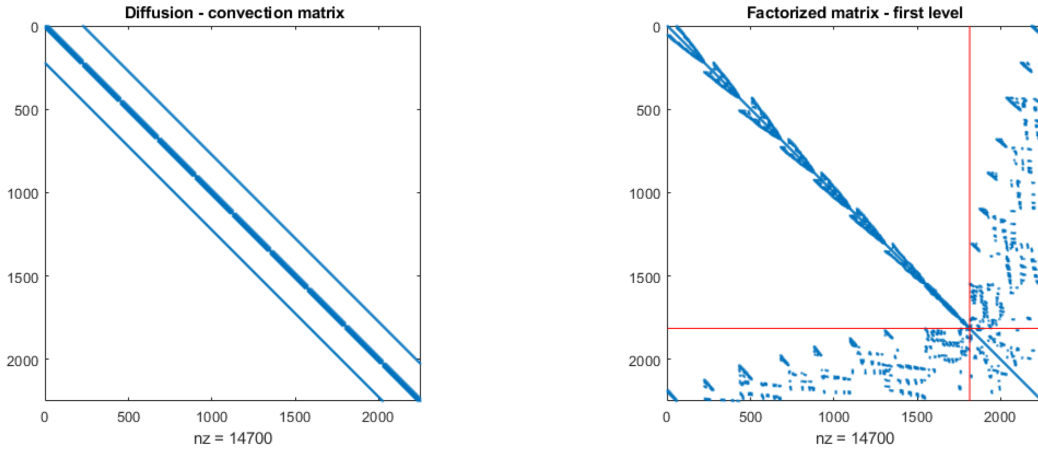


Fig. 5: Demonstration of ARMS block factorization.

Note that in this factorization the m independent variables make up the large block diagonal matrix D in the upper left of the image, and the matrix C should have a size significantly smaller than the original matrix A. The reduction process can be applied another time to the reduced system, A_1, by performing the same factorization for the next lower level. An image of the second level factorization, i.e., the block decomposition of A_1 as shown in Fig. 5 is given in Fig. 6.

In general, this recursion follows the pattern,

$$P_l A_l P_l^T = \begin{pmatrix} D_l & F_l \\ E_l & C_l \end{pmatrix} \approx \begin{pmatrix} L_l & 0 \\ G_l & I_{n-m} \end{pmatrix} \times \begin{pmatrix} U_l & W_l \\ 0 & A_{l+1} \end{pmatrix}, \tag{19}$$

and this process can then be applied recursively to each consecutively reduced system until

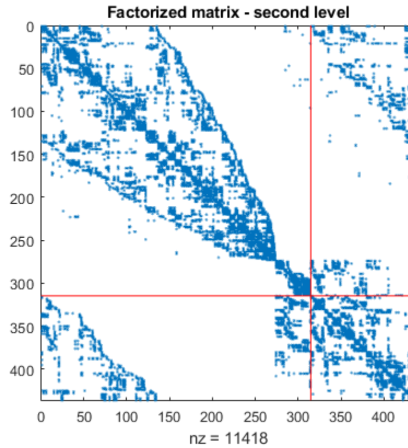


Fig. 6: Second level ARMS factorization.

the Schur complement is small enough to be solved with a standard method. This can be determined a priori by specifying a number of levels of recursion, a desired size of the Schur complement, a desired ratio of the size of the final Schur complement to the original matrix, etc. For the example given, the process of the Algebraic Recursive Multilevel Solver (ARMS) exits after the second factorization since the size of the next block diagonal matrix (e.g., D_3) constitutes the entirety of the matrix A_2. An image showing the location of the non-zeros of the final Schur complement is given in Fig. 7.

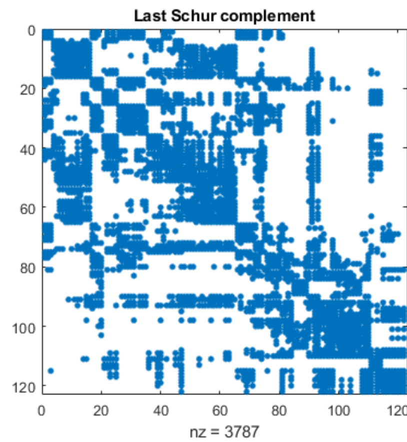


Fig. 7: Final Schur complement for ARMS demonstration.

The ARMS preconditioner [153] uses block independent sets to discover sets of independent unknowns and computes them by using the greedy algorithm. In the ARMS implementation used here, the incomplete triangular factors $\bar{L}$, $\bar{U}$ of D are computed by one sweep of ILU using dual non-zero thresholds (ILUT) [34]. In the second loop, an approximation $\bar{G}$ to $E\bar{U}^{-1}$ and an approximate Schur complement matrix $\bar{A}_l$ are derived. This holds at each reduction level while the recursive process is employed.

In this dissertation, two distinct variants of ARMS are used. For the studies on sticky faults (see Section 4.2.2.2) an implementation of ARMS called ARMS_RBT [155] is used, where the last Schur complement system is small enough to be converted into a dense matrix and randomized using Random Butterfly Transformations [156], [157] to avoid pivoting in the Gaussian elimination. Pivoting during the Gaussian Elimination process is one of the more computationally expensive steps in the process, and there has been prior research on the use of Random Butterfly Transformations to avoid pivoting in direct methods such as Gaussian Elimination [158].

After the transformation, the linear system that results from the randomization step is then solved via an LAPACK-like [159] routine that performs Gaussian elimination with no pivoting, followed by two triangular solves. The ARMS_RBT version has shown satisfactory numerical behavior [155] and can potentially benefit from GPU computing [47]. It appeared also in the experiments conducted in our study that the convergence results with ARMS and ARMS_RBT have been quite similar.

Conversely, in the studies on persistent faults (see Section 4.2.2.3) the final system is solved according to the original methodology proposed in [153] and implemented in [50]. In this implementation, at the last level, another final sweep of ILUT is applied to the last reduced system which is maintained in sparse format throughout.

2.3.2 PRECONDITIONING EXAMPLE

As an example of the effect that preconditioning can have on the solution of a linear system, consider here the 3D convection-diffusion problem discretized over a $15 \times 15 \times 10$ domain using finite differences with a nine-point stencil. This results in a matrix, A, that is symmetric positive-definite (SPD) and has size $2,250 \times 2,250$. The solution to the linear system of algebraic equations,

$$Ax = b, \tag{20}$$

is sought using FGMRES and the two preconditioning techniques discussed in the previous subsections. The right-hand side of the equation, b, is initialized as,

$$b = Av \tag{21}$$

where $v = (1, 2, 3, \ldots, n)^T$ with n equal to the size of matrix, $2,250$. The initial guess, x_0, is set to random numbers sampled from a uniform distribution over the interval $[-1, 1]$.

The FGMRES routine uses a restart parameter of 40 (i.e., the Krylov subspace is restarted every 40 iterations), and the routine exits after the residual is reduced below the threshold value of 10^{-6}. Progress of the solver routine over the first 100 iterations is given in Fig. 8 for the case of: preconditioning FGMRES with ARMS (`ARMS-FGMRES`), preconditioning FGMRES with ILUT (`ILUT-FGMRES`), and the base GMRES with no preconditioning (`NoPC-GMRES`).

Note the superior performance of ARMS preconditioning as compared to ILUT preconditioning. The ARMS preconditioning routine is more computationally expensive than most common incomplete factorization preconditioners, but can often provide better overall performance by decreasing the total number of iterations required more than enough to compensate for the increased computational cost. Performance of GMRES with no preconditioning is included as a baseline to help motivate the use of preconditioners in the solution

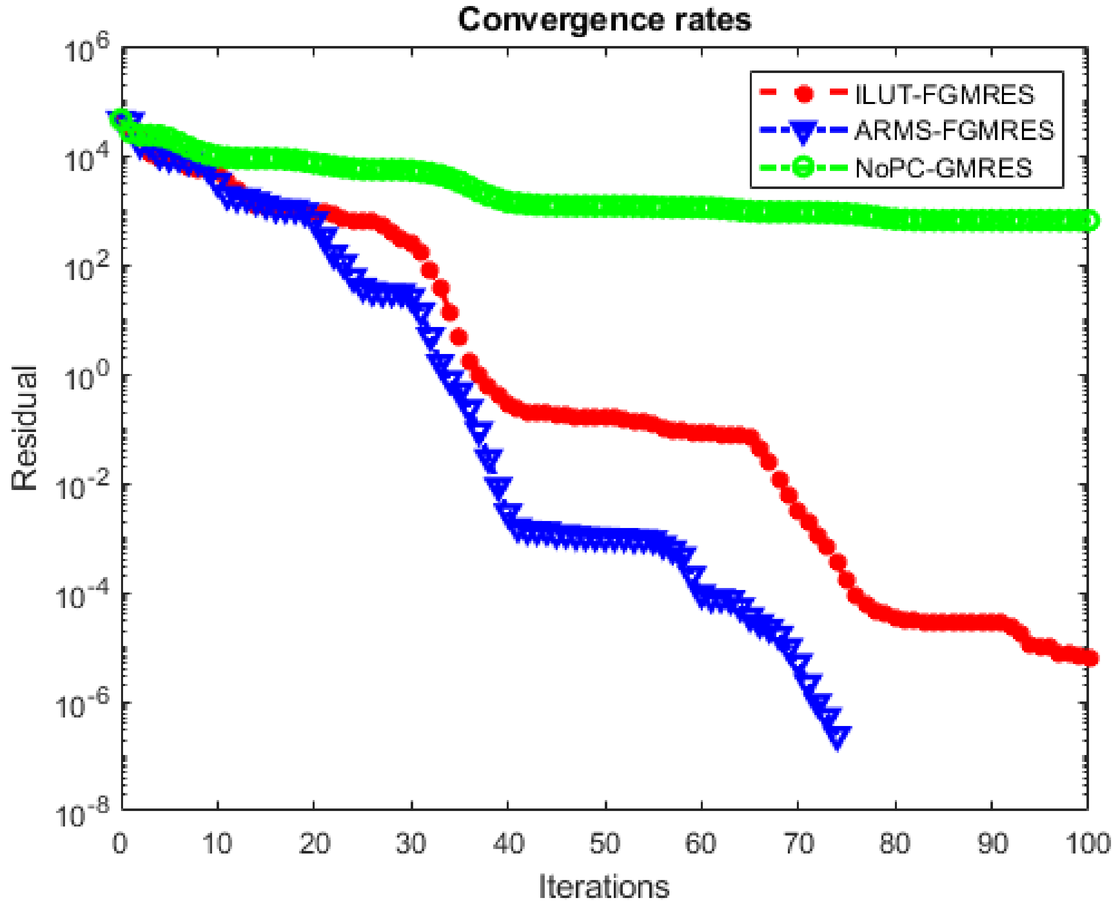


Fig. 8: Convergence rate comparison for FGMRES.

of linear systems of algebraic equations.

2.4 ASYNCHRONOUS ITERATIVE METHODS

In fine-grained parallel computation, each component of the problem, i.e. a matrix or vector entry, is updated in a manner that does not require information from the computations involving other components while the update is being made. This allows for each computing element (i.e. a single processor, CUDA®[2] core or Xeon Phi®core) to act independently from all other computing elements. Depending on the size of both the problem and the computing environment, each computing element may be responsible for updating a single entry, or may

[2]NVIDIA Corp., Santa Clara, CA

be assigned a block that contains multiple components.

This section provides first a review of synchronous iterations, wherein all components are updated at the same time, followed by an introduction to the asynchronous case, which allows component updates to occur at different times. The discussion surrounding the asynchronous case develops the mathematical machinery that will be used to develop results concerning the convergence of asynchronous iterative methods in environments that are susceptible to faults.

2.4.1 REVIEW OF SYNCHRONOUS ITERATIONS

Fixed point iterations are concerned with finding solutions to the iteration

$$x^{(k+1)} = G(x^{(k)}) \tag{22}$$

where $G : \mathbb{R}^n \to \mathbb{R}^n$ is composed of component functions G_i such that

$$\begin{aligned} x_1 &= G_1(x) \\ x_2 &= G_2(x) \\ &\vdots \\ x_n &= G_n(x) \end{aligned} \tag{23}$$

where the subscript represents the *component*, the iteration superscripts have been removed, and the vector notation is added to emphasize that each individual component function used to update a specific component can (potentially) rely on all other components.

In a parallel computing environment, the task of finding the update for an individual (or set of) component(s) can be assigned to an individual processing element. In a system that relies on synchronous updates, the component functions all utilize the same components of

x. In particular,

$$x_i^{(k+1)} = G_i(x^{(k)}) \tag{24}$$

for all components $i \in \{1, 2, \ldots, n\}$, or, breaking this equation into the individual component functions,

$$\begin{aligned} x_1^{(k+1)} &= G_1\left(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \cdots, x_n^{(k)}\right) \\ x_2^{(k+1)} &= G_2\left(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \cdots, x_n^{(k)}\right) \\ x_3^{(k+1)} &= G_3\left(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \cdots, x_n^{(k)}\right) \\ &\vdots \\ x_n^{(k+1)} &= G_n\left(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \cdots, x_n^{(k)}\right). \end{aligned} \tag{25}$$

2.4.2 FRAMEWORK FOR ASYNCHRONOUS ITERATIONS

Asynchronous computation of fixed point iterations follows similarly to the synchronous scenario; however, in the asynchronous case processors will use the latest local information available to them without waiting for global synchronization. There are several ways to define this mathematically (see, for example [30], [56], [58], [66], [70], [100] among many others); informally, the data for each component, x_i, may or may not be from the iteration that just occurred. Following standard assumptions about the amount of allowable delay on updates for the different components [30], [58], [59], convergence of many iterative algorithms is preserved.

This type of updating will lead to different update patterns for the individual component functions, each of which will be utilizing components that may be updated a different number of times. The convergence of parallel fixed point iterations is discussed in the literature for both the synchronous [56] and asynchronous [58] cases among many other sources [57], [59],

[60], [70]. Note that there are many combinations of synchronous and asynchronous updates possible. For example, blocks of components could be scheduled for updates asynchronously, but the individual component updates could be made in a synchronous manner inside of the blocks.

The generalized mathematical model that is used throughout this dissertation comes primarily from [58], which in turn has evolved from sources such as [66], [70], [84], [99]. Small differences exist between the mathematical models proposed for asynchronous iterative methods, but most share the same core tenets. A detailed comparison of the different mathematical models used for asynchronous linear operators (with an emphasis on models that allow for overlapping subdomains) is provided by [99].

To keep the mathematical model as general as possible, consider a function $G : D \to D$ where D is a domain that represents a product space $D = D_1 \times D_2 \times \cdots \times D_m$. The goal is to find a fixed point of the function G inside of the domain D. To this end, a fixed point iteration is performed such that, $x^{(k+1)} = G(x^{(k)})$, and a fixed point is declared if $x^{(k+1)} \approx x^{(k)}$. Note that the function G has internal component functions G_i, for each sub-domain, D_i inside of the product space, D. In particular, $G_i : D \to D_i$, which gives that

$$\begin{aligned} x = (x_1, x_2, \ldots, x_m) \in D \longrightarrow G(x) &= G(x_1, x_2, \ldots x_m) \\ &= (G_1(x), G_2(x), \ldots, G_m(x)) \in D. \end{aligned} \tag{26}$$

Example:

Let each $D_i = \mathbb{R}$. Forming the product space of each of these D_i's gives that $D = \mathbb{R}^m$. This leads to the more formal component function mapping, $f : \mathbb{R}^m \to \mathbb{R}^m$. Additionally, let $f(\mathbf{x}) = 2\mathbf{x}$. In this case, each of the individual f_i component functions is defined by $f_i(\mathbf{x}) = 2x_i$. Note that each component function operates on *all* of the vector $\mathbf{x}$ even if the individual function definition does not require all of the components of $\mathbf{x}$ to perform its specific update.

Further, the assumption is also made that there is some finite number of processing elements $P_1, P_2, \ldots, P_r$ each of which is assigned to a block B of components $B_1, B_2, \ldots, B_m$ to update. Note that the number of processing elements, r, will typically be significantly smaller than the number of blocks, m, to update. With these assumptions, the computational model can be stated in Algorithm 5.

Algorithm 5: General Computational Model

1 **for** *each processing element* P_l **do**
2   **for** $i = 1, 2, \ldots$ *until convergence* **do**
3   Read x from memory
4   Compute $x_j^{(i+1)} = G_j(x)$ for all $j \in \mathcal{B}_l$
5   Update x_j in memory with $x_j^{(i+1)}$ for all $j \in \mathcal{B}_l$

Note that the computational model presented in Algorithm 5 allows for either synchronous or asynchronous computation; it only prescribes that an update has to be made as an "atomic" operation (in line 5), i.e., without interleaving of its result. If each processing element P_l is to wait for the other processors to finish each update, then the model describes a parallel synchronous form of computation. On the other hand, if no order is established for P_ls, then an asynchronous form of computation arises.

To continue formalizing this computational model a few more definitions are necessary. First, set a global iteration counter k that increases every time any processor reads $\mathbf{x}$ from common memory. At the end of the work done by any individual processor, p, the components associated with the block B_p will be updated. This results in a vector, $\mathbf{x} = (x_1^{s_1(k)}, x_2^{s_2(k)}, \ldots, x_m^{s_m(k)})$ where the function $s_i(k)$ indicates how many times an specific component has been updated. Finally, a set of individual components can be grouped into a set, I^k, that contains all of the components that were updated on the kth iteration. Given these basic definitions, the three following conditions (along with the model presented in Algorithm 5) provide a working mathematical framework for fine-grained asynchronous

computation.

Definition 1. If the following three conditions hold:

1. $s_i(k) \leq k$, *i.e. only components that have finished computing are used in the current approximation.*

2. $\lim_{k\to\infty} s_i(k) = \infty$, *i.e. the newest updates for each component are used.*

3. $|k \in \mathbb{N} : i \in I^k| = \infty$, *i.e all components will continue to be updated.*

Then given an initial $\mathbf{x}^{(0)} \in D$, the iterative update process defined by,

$$x_i^k = \begin{cases} x_i^{k-1} & i \notin I^k \\ G_i(\mathbf{x}) & i \in I^k \end{cases}$$

where the function $G_i(\mathbf{x})$ uses the latest updates available is called an asynchronous iteration.

This basic computational model (i.e. the combination of Algorithm 5 and Definition 1 together) allows for many different results on fine-grained iterative methods that are both synchronous and asynchronous, though the three conditions given in Definition 1 are unnecessary in the synchronous case.

Using these definitions, the iterative updates expressed in Eq. (25) can be expressed in a possibly asynchronous format using the functions $s_i(k)$ that keep track of how many updates have occurred for each individual component.

$$\begin{aligned} x_1^{(k+1)} &= G_1\left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_m^{(s_m(k))}\right) \\ x_2^{(k+1)} &= G_2\left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_m^{(s_m(k))}\right) \\ x_3^{(k+1)} &= G_3\left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_m^{(s_m(k))}\right) \\ &\vdots \\ x_n^{(k+1)} &= G_n\left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_m^{(s_m(k))}\right). \end{aligned} \tag{27}$$

The following two examples show how the framework detailed above can be used to express common fixed point iterations.

Example:
Using the terminology adopted in Section 2.4, synchronous iterations are given by enforcing the additional condition that $s_i(k) = k$ for all i and for each iteration k.

Example:
The Jacobi method, both the synchronous and asynchronous case, is given by letting $I^k = \{1, 2, \ldots p\}$ for all k, i.e., all components are updated on every iteration. Further, the synchronous case is given by enforcing $s_i(k) = k$ (as noted in Line 5). The Gauss-Seidel method, specifically the synchronous case, can be defined by letting $s_i(k) = k$ and $I^k = \{k \mod p + 1\}$.

2.4.3 ASYNCHRONOUS RELAXATION METHODS

Relaxation methods have been the focus of many of the works mentioned in Section 2.1 such as [66] and [70]; a much more detailed description can be found in [59] among many other sources. This section provides an introduction that will serve as a reference for the later work in this dissertation.

Relaxation methods can be expressed as general fixed point iterations of the form

$$x^{k+1} = Cx^k + d \ , \tag{28}$$

where C is the $n \times n$ iteration matrix, x is an n-dimensional vector that represents the solution, and d is another n-dimensional vector that can be used to help define the particular problem at hand.

The Jacobi method is an asynchronous relaxation method built for solving linear systems of the form,

$$Ax = b, \tag{29}$$

and following the methodology put forth in [59], this can be broken down to view a specific row – say the ith – of the matrix A,

$$\sum_{j=1}^{n} a_{ij}x_j = b_i, \tag{30}$$

and this equation can be solved for the ith component of the solution, x_i, to give,

$$x_i = \frac{-1}{a_{ii}} \left[\sum_{j \neq i} a_{ij}x_j - b_i \right]. \tag{31}$$

This equation can then be computed in an iterative manner in order to give successive updates to the solution vector. In synchronous computing environments, each update to an element of the solution vector, x_i, is computed sequentially using the same data for the other components of the solution vector (i.e. the x_j in Eq. (31)). Conversely, in an asynchronous computing environment, each update to an element of the solution vector occurs when the computing element responsible for updating that component is ready to write the update to memory and the other components used are simply the latest ones available to the computing element.

Expressing Eq. (31) in a block matrix form more similar to the original form of the iteration expressed in Eq. (28),

$$x = -D^{-1}((L+U)x - b) \tag{32}$$

$$= -D^{-1}(L+U)x + D^{-1}b \ , \tag{33}$$

where D is the diagonal portion of A, and L and U are the strictly lower and upper triangular portions of A respectively. This gives an iteration matrix of $C = -D^{-1}(L+U)$.

Convergence of asynchronous fixed point methods of the form presented in Eq. (28) is determined by the spectral radius of the iteration matrix, C, and dates back to the pioneering work done by both [66] and [70]:

Theorem 1. *For a fixed point iteration of the form given in Eq.* (28) *that adheres to the asynchronous computational model provided by Algorithm 5 and Definition 1, if the spectral radius of* C*,* $\rho(|C|)$*, is less than one, then the iterative method will converge to the fixed point solution.*

As noted in [160], the iteration matrix C that is used in the Jacobi relaxation method serves as a worst case for relaxation methods of the form discussed here. However, because of the ubiquitous use of the Jacobi method in parallel solutions of large problems in many different domains in science and engineering, the asynchronous (block) Jacobi method is used predominantly throughout the remainder of this dissertation. However, many of the concepts and ideas expressed in this dissertation can be easily adapted to more complex algorithms.

2.4.3.1 Asynchronous Jacobi

In science and engineering, partial differential equations (PDEs) mathematically model systems in which continuous variables, such as temperature or pressure, change with respect to two or more independent variables, such as time, length, or angle [161]. Laplace's equation in two dimensions,

$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = b, \tag{34}$$

is fundamental for modeling equilibrium and steady state problems, such as incompressible fluid flow or heat transfer, and maintains that the rate at which a fluid enters a domain is equal to the rate at which a fluid leaves the domain. In practice, the partial differential equation is discretized such that a finite difference operator computes difference quotients over a discretized domain. For example, the two-dimensional discrete Laplace operator,

$$\left(\nabla^2 f\right)(x,y) = f(x-1,y) + f(x+1,y) + f(x,y-1) \\ + f(x,y+1) - 4f(x,y), \quad (35)$$

approximates the two-dimensional continuous Laplacian using a five-point stencil [162]. A special case of the Jacobi algorithm,

$$\left(v_{l,m}^{k+1} = \frac{1}{4}(v_{l+1,m}^{k} + v_{l-1,m}^{k} + v_{l,m+1}^{k} + v_{l,m-1}^{k}\right), \quad (36)$$

may be applied to solve a two-dimensional sparse linear system of equations [102]. This work uses the Jacobi algorithm to solve a two-dimensional finite-difference discretization of the Laplacian with Dirichlet boundary conditions. This can be viewed as a heat diffusion problem, in which a plate is held to specific temperatures along the boundary [163]. Pseudocode for this algorithm is provided below in Algorithm 6. Note that each processor, P_l, may not

Algorithm 6: Asynchronous Jacobi

Input: $a_{ij} \in A$, initial guess for $x^{(0)}$
Output: Solution vector x
1 Assign elements $x_i \in x$ to each processing element
2 **for** $t = 1, 2, \ldots$ *until convergence* **do**
3   **for** *each processor* P_l **do**
4   **if** P_l *is ready to compute updates* **then**
5   **for** *each element* $x_i \in x$ *assigned to* P_l **do**
6                 $x_i = \frac{-1}{a_{ii}}\left[\sum_{j \neq i} a_{ij}x_j - b_i\right]$
7   Calculate the residual, $b - Ax^{(t)}$
8   Check termination conditions

be available to compute updates at the same time. This lack of determinism in the update order (i.e. the amount of time it will take a processor to perform the Jacobi relaxation for

the components that are assigned to it) leads to the asynchronous nature of the algorithm.

In more powerful solvers for solving PDEs, including Krylov subspace solvers such as GMRES and Flexible GMRES, the level of parallelism can be limited compared to asynchronous solvers such as Jacobi, and the penalty associated with having multiple spots in the algorithm that require synchronization (as is the case in all Krylov subspace solvers) could provide reasons to not use them on future HPC platforms.

CHAPTER 3

TECHNIQUES FOR RESILIENCE TO SOFT FAULTS

Asynchronous iterative methods describe a class of parallel iterative algorithms where each computing element is allowed to perform its task without waiting for updates from other processes. Asynchronous iteration is often applied to the parallel solution of fixed point problems, whereby a fixed point iteration,

$$x^{(k+1)} = G(x^{(k)}), \tag{37}$$

is updated in an asynchronous manner with the ultimate goal of finding a fixed point, i.e. a location x^* in the domain such that $x^* = G(x^*)$. This class of problems has been used in a wide variety of applications including: the solution of linear systems [21], [26], [27], the preconditioning of linear solvers [23], [24], optimization [28], [29], and techniques for solving partial differential equations [30], among many others. Note that the analysis of fixed point iterations presented here consider both linear and nonlinear maps.

Asynchronous linear solvers tend not to converge to high precision as quickly as their Krylov subspace counterparts [105]; however, they can approach a low level of accuracy very quickly. This loss of accuracy may cause the use of asynchronous linear solvers to be suboptimal for some applications, but may increase their utility in certain scenarios. For example, possible use cases include using the asynchronous linear solver as a preconditioner to a traditional Krylov subspace solver (e.g. using an asynchronous stationary method solve with low accuracy to precondition a flexible Krylov subspace solver) or to solve systems that only require lower accuracy solutions (e.g. big data, machine learning) [105]. The amount of computational work done during an asynchronous iteration can be greater compared to a synchronous solver; however, the cost of synchronization can cause the asynchronous variant

to be faster. Convergence rate and related theoretical results are not very developed in the asynchronous case relative to the synchronous case; however, there has been a resurgence in interest in asynchronous iterative methods in recent years which has caused new results to emerge.

At a high level, fault tolerance may be divided into efforts to compensate for the effects of hard faults, and efforts to mitigate the impact of soft faults. Successful fault tolerance for hard faults relies chiefly upon detection of the fault itself. This will most likely be handled by the HPC platform; however, the successful recovery of the iterative method in question requires the algorithm to have the knowledge that a hard fault has occurred. This could be achieved internally in the algorithm by declaring a hard fault if components belonging to a particular block (corresponding to some specific processing element) fail to be updated within some stated bound. An alternative to this approach is to adopt a randomized approach towards which processors are assigned to work on specific tasks. Utilizing randomization in this manner allows progress on any given task to continue if a processor or node experiences a fault, since as long as a single processor is still capable of completing tasks it will eventually complete all required work by virtue of being randomly assigned each iteration. Work evaluating the performance of various methods of weighting the random selection was performed in [164], but is not included in this dissertation.

Similar to the case of a hard fault, the most important aspect to recovering from a soft fault is successful detection. This is often more difficult in the case of a soft fault since, though it corrupts data, it does not cause direct interruption to the flow of the iterative process. Many detection techniques rely on choosing an appropriate tolerance to check a property of the algorithm that has predictable behavior (e.g. a residual that is monotonically decreasing, a known property concerning a vector/matrix norm); a tolerance that is too loose will allow potentially harmful errors to go undetected, while a tolerance that is too strict may report a fault when none actually occurred ("false positive") and cause the program to do extra work to recover from a non-existent problem. The balance in choosing the correct fault tolerance

method to recover from soft faults can be application or problem dependent.

Most often, soft faults refer to some form of data corruption in the algorithm being executed. The focus of the theoretical analysis presented in this chapter is to analyze the effect that soft faults might have on fine-grained iterative methods used for calculating a fixed point, specifically the effect of faults that are transient in nature (i.e. faults whose impact is generated over a very short period of time). A common example of such a fault is a bit-flip that causes one bit of data in unprotected memory to become corrupted. Additionally, a variety of recovery techniques are discussed that may be able mitigate the effect of a computing fault. Several examples are provided for popular classes of algorithms, and a more in-depth analysis of an algorithm will be presented in Chapter 5. Portions of this chapter have been captured in other papers: [165]–[167].

The structure of this chapter is organized as follows: Section 3.1 discusses the impact a soft fault may have on an asynchronous iterative method and presents new results regarding the convergence of asynchronous iterative methods based upon given observations. Section 3.2 goes over techniques for the resilience of asynchronous iterative methods, while Section 3.3 provides a summary.

3.1 MODELING THE IMPACT OF SOFT FAULTS

Before providing results, it is necessary to make some further definitions and assumptions to clarify the scope that the presented results are intended to apply to. This begins with clarifying the expected impact of a soft fault. A soft fault is an error that is undetected by the algorithm and the operating system and introduces silent data corruption into the result of the operation that it occurs on. This section takes two distinct views on how a fault may manifest. The first is given in the following definition and the second is a probabilistic viewpoint (based on the work in [54]) that is detailed in Section 3.1.2.

Definition 2. A fault experienced by the computing element P_j is realized as a delay, d_f, in the computation of the update for the component function, G_j, that P_j was assigned to

update at the time that the fault occurred.

This definition implies that if a component function, G_i, were to perform an update at the kth iteration, it would updated at the $(k + d_f)^{th}$ iteration.

Remark 1. *The definition of a fault given in Definition 2 also applies to hardware malfunction or hard faults. That is, if the fault is detected and the underlying hardware is able to correct the associated error (e.g., by restarting the work assigned to a specific computing element), the result will naturally manifest as described in Definition 2.*

The limitation of a fault to an effect that is felt by the algorithm as a delay may be hard to enforce in practice; however, adopting a *selective reliability* computational model [31], [32] – which allows certain computations to be executed safely in a high reliability mode, while allowing other calculations to occur in a potentially faster lower reliability mode (also referred to as a *fast* mode of computation) – may allow restriction of the occurrence of faults to data locations that only affect allowable portions of the data structures. In conjunction with scanning for extreme values such as `NaN`, `INF`, as well as values over some prespecified extreme value threshold, it is not unreasonable to expect that faults may be restricted in such a way that they only manifest as delays. Additionally, if fault tolerance strategies such as checkpointing are allowed, it becomes even more reasonable to expect that certain components may just be set farther back along their individual iterative path. Examples of potential applications of the theory developed in this subsection as well as strategies for detecting and correcting the effects of a fault are discussed in Section 3.2.

Section 3.1.1 extends classical results that are often used as building blocks for new asynchronous algorithms to cope with the introduction of faults as defined in Definition 2. The second subsection, Section 3.1.2, takes a much more general viewpoint and attempts to find ways to bound the amount of total error introduced when very few assumptions are made regarding the impact that the fault may have.

3.1.1 GENERAL CONVERGENCE RESULTS

The following standard assumption (see [56], [58], [59], [85], [100] among many others) specifies a few further conditions on the fixed point operation that are needed in order to proceed with proving results.

Assumption 1. *Given a domain D and an operator G as outlined by Definition 1, there is a sequence of nonempty sets $\{D^k\}$ such that the following three conditions hold:*

1. (Nested set condition) The sequence of sets $\{D^k\}$ satisfy the condition that $D^{k+1} \subset D^k$ for all k. Further, the image of each set under G is contained in the next set in the sequence, i.e.,

$$G(D^k) \subset D^{k+1} \tag{38}$$

2. (Synchronous convergence condition) There exists a point x^* such that a sequence $\{y^k\}$ that satisfies $y^k \in D^k$ for all k has the property that,

$$\lim_{k \to \infty} y^(k) = x^* \tag{39}$$

i.e., that every limit point of a sequence that is taken with one element from each of the nested sets is a fixed point of the operator G.

3. (Box condition) For every D^k in the collection $\{D^k\}$ there are sets $D_i^k \subset D^k$ such that,

$$D^k = D_1^k \times D_2^k \times \cdots \times D_m^k \tag{40}$$

The general convergence theorem for traditional, fault-free fixed point algorithms, is provided below in Theorem 2.

Theorem 2. *If all of the conditions from both Definition 1 and Assumption 1 hold, and the initial guess x^0 is in the set D^0, then the sequence of iterates $\{x^{(k)}\}$ given by Algorithm 5 converge to x^*, the fixed point of G.*

The proof from [59] proceeds by induction and showing that there will be a time (e.g., an iteration count, t) where eventually all of the individual components of $x^{(k)}$ (i.e., $x_1^{(k)}, x_2^{(k)}, \ldots$) will be in D^k.

The next result provides convergence of the asynchronous fixed point algorithm subject to a single fault that is realized as a delay.

Theorem 3. *If all of the conditions from both Definition 1 and Assumption 1 hold, the initial guess $x^{(0)}$ is in the set D^0, and a single fault is encountered at some unspecified time t_f during the execution of the algorithm that manifest according to Definition 2, then the sequence of iterates $\{x^{(k)}\}$ given by Algorithm 5 converge to x^*, the fixed point of G.*

Proof. The proof proceeds by induction. Similar to the proof of Theorem 2 provided in [59], the goal is to show that there is a time, t_k, such that for sufficiently large k, the values of $s_i(k)$ will all be large enough to guarantee that the individual components inside of the respective domains D_i, and thus $G(x^{(t)})$, are in D^k for all $t \geq t_k$.

The base case of the induction, i.e. the case corresponding to $k = 0$, is true from the assumption that $x^0 \in D^0$ which is made in the statement of Theorem 3.

Next, the proof proceeds by assuming that the statement is true for a given k and establishing that it is true for $k+1$. In particular, a time, t_{k+1} is sought such that for all times t that are larger than t_{k+1}. To this end, introduce a new collection of sets, T^i, that, matching the notation in [59], represent the set of times that x_i is updated. Note that this can be recovered from the sequence of sets $\{I_k\}$ that represent the collection of components that are updated at each iteration, k.

Using this, for each domain index, $i = 1, 2, 3, \ldots, m$, let $\bar{t}^i$ represent the first time that each component i would be updated in a fault-free environment after the time t_k. Equivalently, this will be the smallest element of the set T^i that is larger than t_k. Assume without loss of generality that the single fault effects component i_f. Because of condition 2 of Definition 1 (i.e. that all components will continue to be updated) and the fact that a fault is realized as a delay d_f, set t^i to be the smallest entry in the set T^i that is larger than $\bar{t}^i + d_f$.

Then, the synchronous box condition gives that,

$$G_i\left(x^{(t^i)}\right) \in D^{k+1}, \tag{41}$$

or that if all components have been updated sufficiently to ensure there individual update counts are larger than t_k – accounting for the occurrence of a fault – the operator G will move the domain to the next subdomain in the sequence of nested subdomains.

Next, using the box condition, this gives that,

$$x_i^{(t^i+1)} = G_i\left(x^{(t^i)}\right) \in D_i^{k+1}, \tag{42}$$

i.e., that each component resides in the appropriate subdomain in the decomposition of the nested set, D^k. Set $t'_k = \max_i\{t^i\} + 1$. Then for all $t \geq t'_k$ each component function, G_i, will have moved the components it is responsible for into the subdomain, D_i^{k+1}, and therefore using the box condition it can be said that,

$$x^{(t)} \in D^{k+1}, \tag{43}$$

which shows that the iterates will continue progressing through the nested sets until they arrive at the fixed point, completing the proof. □

Theorem 3 addresses only the case of a single fault. While faults on next generation HPC platforms are expected to be rare occurrences, there is certainly no guarantee that only a single soft fault will be experienced during the runtime of an application. In fact, large-scale, long running simulations may encounter many soft faults based on the predicted worst case MTBF [1]. The next result addresses the case of a general fixed point algorithm encountering multiple faults during a single execution.

Theorem 4. *If all of the conditions from both Definition 1 and Assumption 1 hold, the initial guess $x^{(0)}$ is in the set D^0, and a finite number of faults are encountered at some unspecified*

time t_f during the execution of the algorithm that manifest according to Definition 2, then the sequence of iterates $\{x^{(k)}\}$ given by Algorithm 5 converge to x^*, the fixed point of G.

Proof. Proceed again by induction. Base case is given by Theorem 3. Assume true for k faults, and the modification for $k+1$ faults follows from the same logic that is used in the proof of Theorem 3. In particular, it is possible to pick a sufficiently large time to ensure that all components have been updated sufficiently to guarantee regardless of how many faults occur, under the critical assumption that faults manifest as delays to the updates of individual components. □

3.1.1.1 Nonexpansive Operators

There are many special cases of the general theory presented in the previous subsection that merit further attention. One in particular that will be examined here is the case where G is a nonexpansive operator. Before proceeding further, however, several more concepts need to be defined more precisely. The goal in this subsection is to examine results for nonexpansive operators, which represent a slight generalization to the standard results, and to adjust the results as necessary to apply to the case where a fault occurs. Since results concerning nonexpansive operators are a little more general than the results presented in the previous subsection, additional assumptions need to be defined to ensure that the algorithm converges properly.

The theory behind asynchronous iterative methods for nonexpansive operators was first detailed in [59], and then generalized for the case of linear operators in [100]. Next, [56] examined parallel fixed point iterations and expanded the class of operators to include non-linear nonexpansive operators; however, the paper [56] restricted the parallelism to only include synchronous updates; the subsequent paper [57] generalized this further and allowed for the case of asynchronous updates. The additional assumptions (relative to the conditions assumed for Theorem 2) made by all four of these works are similar, but not identical.

To develop theoretical results related to nonexpansive operators, a few more concepts must be clearly defined. First, it must be noted that the convergence results presented that relate to nonexpansive operators all require the concept of bounded delay. This concept has also been referred to as partial asynchronism [59], and is made more precise in the following definition.

Definition 3. An asynchronous iteration with bounded delay is one that enforces a bound on how far behind the updates to one component can lag behind another. This can be viewed as a constant, t_b, such that for any given iteration count, k, the following inequality holds:

$$\left(\max_i s_i(k) - \min_i s_i(k)\right) \leq t_b. \tag{44}$$

Next, the concepts of nonexpansive and paracontracting operators need to be clearly defined.

Definition 4. An operator, $G : D \to D$ is said to be nonexpansive with respect to the norm $||\cdot||$ if it satisfies,

$$||G(x) - G(y)|| \leq ||x - y|| \tag{45}$$

for all $x, y \in D$. Note that if the operator G is linear, it can be expressed as a matrix, and that this identity can then be written more simply as,

$$||Gx|| \leq ||x|| \tag{46}$$

for all $x \in D$.

In order to examine this class of operators, certain restrictions need to be placed upon the subdomains, D^i, that make up the entire product space, D. In particular, each subdomain D^i needs to be assumed to be a normed linear space, $(D^i, ||\cdot||_i)$. With this assumption, the weighted maximum norm can be defined. Note that the definition used here follows from [100] and [58].

Definition 5. The weighted maximum norm of a vector $x \in D$, denoted $||x||_w$ is given by,

$$||x||_w = \max_{1 \leq i \leq m} \frac{||x||_i}{w_i} \tag{47}$$

where $||x||_i$ is the norm that exists on the normed subdomain, D^i, and $w = (w_1, w_2, \ldots, w_m)$ is a positive vector that satisfies $w_i > 0$ for all $i = 1, 2, \ldots, m$.

Before proceeding to the development of results on nonexpansive results, [58] provides a result, originally due to El Tarazi [84], that is related to the nested set result presented in Theorem 2, but that makes use of the idea of weighted norms.

Theorem 5. *Assume that $G : D \to D$ has a fixed point x^*, and that there exists a constant, γ with $0 \leq \gamma < 1$ such that for all iterations, k, the following is satisfied,*

$$||G(x) - x^*||_w \leq \gamma \cdot ||x - x^*||_w. \tag{48}$$

Then the asynchronous iterates defined by the sequence $\{x^{(k)}\}$ converge to x^*.

This is just a slightly more specific instance (i.e., one where each of the subdomains is required to have a norm) of Theorem 2. As pointed out in [59], one possible motivation for examining contractive mappings with respect to weighted norms is that the unit sphere defined by such a weighted norm satisfies the nested set criteria imposed in Assumption 1.

In general, showing that an asynchronous iterative algorithm converges requires showing one of two things:

1. there is an appropriate sequence of nested sets (see Assumption 1) or
2. the operator that generates the sequence is contractive under an appropriate weighted maximum norm.

Here, the additional assumptions are taken from [57] and are adopted in Theorem 6 that provides convergence in the fault-free case (also taken from [57]) below.

Theorem 6. *Let the following hypotheses hold:*

h0 There exists a subsequence, $\{p_k\}$ *such that for all* $i \in \{1, 2, 3, \ldots, p\}$*, the conditions* $i \in I(p_k)$ *and* $s_i(p_k) = p_k$ *for all* i

h1 The asynchronous iterative procedure has bounded delay with constant t_b

h2 There exists a fixed point x^* *in the domain* D *such that* $G(x^*) = x^*$

h3 The operator G *is nonexpansive with respect to the maximum norm*

h4 The condition,

$$||G(x) - G(y)||^2 \leq \langle G(x) - G(y), x - y\rangle \tag{49}$$

for all $x, y \in D$

Then the asynchronous iteration defined by Algorithm 5 and Definition 1 converges to a fixed point $x^* \in D$.

The hypothesis $h0$ dictates that there are a set of times that all processors are synchronized, a hypothesis that can be enforced by the programmer, while the condition in $h4$ is satisfied by a large number of operators (e.g. G is linear, symmetric, and positive; G is a maximal monotone operator; G is strongly convex; etc [56], [57]).

Extending this analysis to the case of a fault follows a similar pattern to the development from Theorem 2 to Theorem 3. In particular, the new result will assume the occurrence of a single, undetected computing fault to one of the processors at some point throughout the iterative procedure. This result can be stated as follows.

Theorem 7. *Let the following hypotheses hold:*

h0 There exists a subsequence, $\{p_k\}$ *such that for all* $i \in \{1, 2, 3, \ldots, p\}$*, the conditions* $i \in I(p_k)$ *and* $s_i(p_k) = p_k$ *for all* i

h1 The asynchronous iterative procedure has bounded delay with constant t_b

h2 There exists a fixed point x^* *in the domain* D *such that* $G(x^*) = x^*$

h3 The operator G is nonexpansive with respect to the maximum norm

h4 The condition,

$$||G(x) - G(y)||^2 \leq \langle G(x) - G(y), x - y \rangle \tag{50}$$

for all $x, y \in D$

h5 A single soft fault, as defined by Definition 2, occurs at an undetermined time during execution but before convergence is reached

Then the asynchronous iteration defined by Algorithm 5 and Definition 1 converges to a fixed point $x^* \in D$.

Proof. Similar to the proofs in both [56] and [57], this proof proceeds in three steps. Each of the three steps is very similar to those presented in [57] with small additions made to account for the occurrence of a fault.

Step 1 The first step is to show that the sequence generated by the iterates $\{x^p\}$ is bounded. This is done by considering that the sequence generated by $\{||x^p - x^*||_\infty\}_{p \in \mathbb{N}}$, for some fixed point x^* of the operator G is convergent.

To do this, consider a collection of iterates, $x^{(p)}, x^{(p-1)}, x^{(p-2)}, \ldots, x^{(p-t_b)}$, and define $z^{(p)}$ such that,

$$z^{(p)} = \max_{0 \leq l \leq t_b} ||x^{(p-l)} - x^*||_\infty = \max_{p-t_b \leq l \leq p} ||x^{(l)} - x^*||_\infty \tag{51}$$

Then, for each processor $i \in \{1, 2, \ldots, r\}$ one of two conditions much be true: either $i \in I^p$, meaning that processor i will perform an update at the pth iteration, or $i \notin I^p$ so that processor i will not perform an update at the pth iteration; either way, the distance between the ith block of components (assigned to the ith processor) and the corresponding components

in the fixed point, x^*, can be bounded. In the case that $i \in I^p$,

$$\begin{aligned} ||x_i^{(p+1)} - x_i^*||_i &= ||x^{(p_i)} - x_i^*||_i \\ &\leq ||x^{(p)} - x^*||_\infty \\ &\leq \max_{0 \leq l \leq t_b} ||x^{(p-l)} - x^*||_\infty \\ &= z^{(p)} \end{aligned} \tag{52}$$

Or, in the case that $i \notin I^p$,

$$||x_i^{(p+1)} - x_i^*||_i = ||G_i(x_1^{(s_1(p))}, x_2^{(s_2(p)}), \ldots, x_m^{(s_m(p))}) - G_i(x^*)||_i \tag{53}$$

$$\leq ||G(x_1^{(s_1(p))}, x_2^{(s_2(p))}, \ldots, x_m^{(s_m(p))}) - G(x^*)||_\infty$$

$$\leq ||(x_1^{(s_1(p))}, x_2^{(s_2(p))}, \ldots, x_m^{(s_m(p))}) - x^*||_\infty \tag{54}$$

$$= ||x_j^{(s_j(p))} - x_j^*||_j \tag{55}$$

$$\leq ||x^{(s_j(p))} - x^*||_\infty$$

$$\leq \max_{p-t_b \leq l \leq p} ||x^{(l)} - x^*||_\infty \tag{56}$$

$$= z^{(p)} \tag{57}$$

where Eq. (54) follows from the assumption that the operator G is nonexpansive (i.e. $h3$), Eq. (55) follows from the previous line since the equality must hold for some $j \in \{1, 2, \ldots, r\}$, and Eq. (56) follows from the assumption of bounded delay.

In either the case of $i \in I^p$ or $i \notin I^p$, there is a chance that a fault occurs during the update process that is considered in the inequalities. In the first case, $i \in I^p$, this only affects the range that the maximum is taken over; instead of taking the maximum over the range $0 \leq l \leq t_b$, the maximum needs to be taken over the (possibly) extended range, $0 \leq l \leq \max(t_b, d_f)$. Similar to the extensions made to the proof in Theorem 3, this ensures that the delay caused by potential hardware malfunction has time to be corrected by the

natural iterative update process in the algorithm. In the case $i \notin I^p$, by definition no update is being made, so therefore no adjustments need to be made.

With this, for all $i \in \{1, 2, \ldots, r\}$,

$$||x_i^{(p+1)} - x_i^*||_i \leq z^p \tag{58}$$

which establishes that,

$$||x^{(p+1)} - x^*||_\infty \leq z^p, \tag{59}$$

and therefore,

$$\begin{aligned} z^{(p+1)} &= \max_{0 \leq l \leq t_b} ||x^{(p+1-l)} - x^*||_\infty \\ &= \max\{\max_{0 \leq l \leq t_b - 1} ||x^{(p-l)} - x^*||_\infty, ||x^{(p+1)} - x^*||_\infty\} \\ &\leq z^{(p)}. \end{aligned} \tag{60}$$

This shows that the sequence defined by $\{z_p\}$ is decreasing, and since it is also positive as it is defined to be a norm, this combines to show that the sequence is convergent.

Expanding the definition of $\{z^{(p)}\}$,

$$\begin{aligned} \lim_{p \to \infty} z^p &= \lim_{p \to \infty} \max_{0 \leq l \leq t_b} ||x^{(p-l)} - x^*||_\infty \\ &= \lim_{p \to \infty} ||x^{(p-j(p))} - x^*||_\infty \text{ for } 0 \leq j(p) \leq t_b \\ &= \lim_{p \to \infty} ||x^{(p)} - x^*||_\infty, \end{aligned} \tag{61}$$

which shows that the sequence defined by $\{||x^{(p)} - x^*||_\infty\}$ is convergent, which implies that the sequence $\{x^{(p)}\}$ is bounded, which in turn completes Step 1 of the proof.

Step 2 The second step is to show that the subsequence of iterates that is defined by the (sub)sequence provided by the hypothesis $h0$, i.e. the sequence $\{x^{(p_k)}\}$, converges to a fixed

point of G.

First, since the sequence $\{x^{(p)}\}$ is bounded by Step 1, the subsequence defined by $\{x^{(p_k)}\}$ is also bounded. Note that even if a fault were to occur at one of the times p_k that by the way a fault has been defined, this will just slow the progression of the sequence $\{x^{(p_k)}\}$ and will not affect the boundedness. In order to show that the subsequence $\{x^{(p_k)}\}$ converges to a fixed point x^* of the operator G, first define another sequence,

$$\{y^{(p)} = x^{(p)} - G(x^{(p)})\} \tag{62}$$

and the desired result follows from showing that the subsequence of $\{y^{(p)}\}$ generated by the indices p_k, i.e. $\{y^{(p_k)}\}$ converges to 0.

To do this, start by establishing a bound on the norm of $\{y_{(p_k)}\}$. Note that,

$$\begin{aligned} ||x^{(p_k)} - x^*||^2 &= ||y^{(p_k)} + G(x^{(p_k)}) - x^*||^2 \\ &= ||y^{(p_k)}||^2 + ||G(x^{(p_k)}) - x^*||^2 + 2\langle G(x^{(p_k)}) - x^*, y^{(p_k)}\rangle, \end{aligned} \tag{63}$$

which gives an estimates of $||y^{(p_k)}||^2$ as,

$$||y^{(p_k)}||^2 = ||x^{(p_k)} - x^*||^2 - ||G(x^{(p_k)}) - x^*||^2 - 2\langle G(x^{(p_k)}) - x^*, y^{(p_k)}\rangle \tag{64}$$

This can be expanded further using the hypothesis $h4$ since,

$$\begin{aligned} \langle G(x^{(p_k)}) - x^*, y^{(p_k)}\rangle &= \langle G(x^{(p_k)}) - G(x^*), x^{(p_k)} - G(x^{(p_k)})\rangle \\ &= \langle G(x^{(p_k)}) - G(x^*), [x^{(p_k)} - G(x^*)] - [G(x^{(p_k)}) - G(x^*)]\rangle \\ &= \langle G(x^{(p_k)}) - G(x^*), x^{(p_k)} - x^*\rangle - ||G(x^{(p_k)}) - G(x^*)||^2 \\ &\geq 0. \end{aligned} \tag{65}$$

Which provides a bound on $||y^{(p_k)}||^2$ as,

$$\begin{aligned} ||y^{(p_k)}||^2 &\leq ||x^{(p_k)} - x^*||^2 - ||G(x^{(p_k)}) - x^*||^2 \\ &= ||x^{(p_k)} - x^*||^2 - ||(x^{(p_k+1)}) - x^*||^2, \end{aligned} \tag{66}$$

where the latter equality follows since $G(x^{(p_k)}) = (x^{(p_k+1)})$ by following the iterates defined in the subsequence created by $h0$. Here, it is possible that a fault could cause a delay in the update of the iterate $x^{(p_k)}$. In particular, if a fault were to occur,

$$G(x^{(p_k)}) = G(x^{(p_k-d_f)}). \tag{67}$$

However, the bound on $||y^{(p_k)}||^2$ is still valid since, due to the assumptions made about the nature of a fault in Definition 2, the sequence $\{x^{(p_k)}\}$ will eventually successfully reach the value $x^{(p_k+1)}$.

Next, from Step 1, the sequence generated by $\{||x^{(p)} - x^*||_\infty\}$ is convergent, even if a soft fault is to occur. This further implies that the sequence $\{||x^{(p)} - x^*||\}$ is convergent with or without the occurrence of a soft fault. The limit of the sequence $\{||x^{(p)} - x^*||\}$ is then independent of the occurrence of a soft fault and is given by,

$$\begin{aligned} \lim_{p\to\infty} ||x^{(p)} - x^*|| &= \lim_{p_k\to\infty} ||x^{(p_k)} - x^*|| \\ &= \lim_{p_k\to\infty} ||x^{(p_k+1)} - x^*|| \\ &= \lim_{p_k\to\infty} ||x^* - x^*||. \end{aligned} \tag{68}$$

This shows that,

$$\lim_{p_k\to\infty} ||y^{(p_k)}|| = 0, \tag{69}$$

which additionally gives that,

$$\lim_{p_k\to\infty} y^{(p_k)} = 0. \tag{70}$$

Because of how the sequence $\{y^{(p_k)}\}$ is defined, this establishes that the subsequence $\{x^{(p_k)}\}$ converges to x^*, which is a fixed point of the operator G.

Step 3 The last step is to combine the results of the first two steps to show that the sequence of iterates, $\{x^{(p)}\}$ converges to x^*. This follows naturally because $\{x^{(p)}\}$ is bounded (as shown in Step 1), as the iterates generated by $\{x^{(p_k)}\}$ progress, the sequence $\{x^{(p)}\}$ also moves towards x^* since the progression of $\{x^{(p_k)}\}$ towards x^* will be the same, i.e.,

$$\lim_{p\to\infty} ||x^{(p)} - x^*||_\infty = \lim_{p_k\to\infty} ||x^{(p_k)} - x^*||_\infty = 0 \tag{71}$$

which shows that the sequence $\{x^{(p)}\}$ converges to x^* with respect to the norm $||\cdot||_\infty$. Note again that the results that are being combined from both Step 1 and Step 2 have taken into account the possible occurrence of a soft fault, and therefore the sequence of asynchronously generated iterates will converge for the given nonexpansive operators (i.e. operators that satisfy the hypotheses $h0, h1, h2, h3$, and $h4$) even if a soft fault were to occur. □

This result on nonexpansive operators represents a result for a more specific class of operators than is given in Section 3.1. Extending the result in Theorem 7 to account for multiple faults follows in the same manner as the extension of the proofs from Theorem 3 to Theorem 4 and the formal statement and proof are omitted.

3.1.2 BOUNDS ON INDUCED ERROR

With a fault as defined by Definition 2, the effects of a fault do not propagate between the various components in the vector $x \in D$. However, this is not always possible to enforce in practice. The purpose of this subsection is to explore the possible negative effect of an undetected soft fault. In order to keep the analysis as general as possible it is necessary to take a probabilistic viewpoint. The analysis presented here follows closely the work shown in [54] where the authors presented a probabilistic analysis of the possible error introduced for synchronous fixed point iterations. The work shown here restates many

results for completeness and takes special care to ensure that the parallel, asynchronous nature of the algorithms that are the focus of this work is accounted for.

An undetected computing fault that occurs may end up causing either divergence or stagnation of the iterative algorithm. In order to determine the conditions for convergence if some bounded amount of data corruption is injected into the algorithm (i.e. by an undetected soft fault), the amount of data corruption must first be quantified.

For any operation G on the vector $x \in D$ where $G : D \to D$, some number of components, x_i, from the output can be affected by a fault that occurs. Even for a relatively trivial map G it is often hard to pin down exactly where the error occurs if the corrupted output is the only location that the fault visibly manifests. Further, the corruption from one component can spread to other components and this proliferation of data corruption often occurs in a non-deterministic manner that depends on the timing and magnitude of the error in question.

With this in mind, in order to attempt to provide some bound on the amount of acceptable error for a parallel fixed point algorithm, a probabilistic approach similar to the one taken in [54] is adopted. The work presented in [54] provides a strong foundational framework for how to analyze soft faults for generic computational tasks and provides an example that shows how this framework can be adapted to fit with synchronous parallel fixed point problems. The extension provided here provides a further adaptation to the asynchronous case.

In the case of a fault occurring when computing a vector $x \in D$, the resultant vector, denoted by $\bar{x} \in D$, can be expressed as

$$\hat{x} = x + \tilde{x} \tag{72}$$

for some random vector $\tilde{x} \in D$. That is, the vector obtained differs from the vector that would arise in the fault-free case by some unknown amount that is linked to an unknown probability distribution. This is captured more precisely by the following assumptions.

Assumption 2. *Soft faults are independent events.*

Assumption 3. *For every operation not conducted in a high reliability mode there is a positive probability of having a soft fault occur.*

Many studies make Assumption 2 above; namely that soft faults are independent events (e.g., [54], [126], [129]); however, data collected on DOE supercomputers has suggested that failure rates may not follow an exponential distribution exactly [13], especially near the beginning and end of the life of the HPC platform. Some recent research efforts have used mathematical strategies with respect to the timing and occurrence of faults [130] in an effort to quantify the possible numerical error. Following the majority of studies, soft faults are assumed to be independent in this work; however, a pessimistic view on the effect of a fault, where arbitrarily large perturbations are allowed to occur on every iteration, and stringent convergence criteria are imposed in order to ensure that the effects of a fault will not hinder an algorithm variant proposed here when it is used in application.

Similar to [54], these assumptions allow the modeling of faults to be done via a two level probability mixture. The first level indicates whether or not a fault has occurred as a given step and is represented by a sequence of discrete Bernoulli parameters, $\mathbf{p} = \{p_1, p_2, p_3, \ldots\}$, signifying that a fault has or has not occurred at iteration k. Note that in the parallel asynchronous case, this iteration counter is incremented every time that a processor reads the vector

$$x^k = \left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, \cdots, x_m^{(s_m(k))}\right) \tag{73}$$

from memory. As such the relative value of k may be significantly higher in the asynchronous case than it is in the synchronous case.

Next, since the effect of the fault is realized as a random perturbation, $\tilde{x}$, to the value of the iterate that would have been calculated in a fault-free calculation, this can be modeled as a sequence of unknown probability distributions, $\mathbf{P} = \{P_1, P_2, P_3, \ldots\}$, associated with

each Bernoulli parameter. This creates a list of pairs, $(\mathbf{p}, \mathbf{P}) = (p_i, P_i)$, that are associated with each computational step that is not executed in high reliability mode.

Following still the notation from [54], let e represent the difference between one series of iterations, possibly subject to the effects of soft faults and the result that would occur in a fault-free environment. To be precise, assume that in the nominal, fault-free environment the iterative fixed point algorithm terminates after M_1 iterations to an initial vector x, while when subject to faults, M_2 iterations to an initial vector y are required. This error term, e, can be expressed formally as

$$e = \left\| x^{(M_1)} - y^{(M_2)} \right\| \tag{74}$$

in the desired norm. Note that e itself is a random variable that depends on the sequence of random variable pairs (p_i, P_i). The strict approach is to assume that the effect of the fault has no impact on the outcome of the fixed point iteration, i.e., that the random variable e always returns 0. This is captured in the following definition.

Definition 6. (Def. 2 from [54]) An iterative algorithm is convergent when subject to soft faults if the mean and variance of e can be made arbitrarily small with a finite amount of computational effort. Denoting mean with E and variance with V this can be expressed by requiring that

$$\sup_{\mathbf{P}} E_{(\mathbf{p},\mathbf{P})}(e) < \epsilon \tag{75}$$

$$\sup_{\mathbf{P}} V_{(\mathbf{p},\mathbf{P})}(e) < \epsilon \tag{76}$$

for every $\epsilon > 0$, every series of (unknown) probability distributions $\mathbf{P}$ can be achieved in a finite amount of computation.

Note that this ensures that the final iterate when subject to faults will be the same as the final iterate in a fault-free case, which does not require either of the following to be true:

- the algorithm that allows the occurrence of hardware faults finishes in the same number of iterations, or
- every iterate of the algorithm continues to satisfy the desired properties concerning e, i.e., a fault may have an impact so long as it is corrected.

Returning more specifically to the case of fixed point iteration, each iterative update can be expressed as shown in Algorithm 7.

Algorithm 7: Parallel Fixed Point Iteration

1 **for** *each processing element* P_l **do**
2   **for** $k = 1, 2, \ldots$ *until convergence* **do**
3   Compute $x_j^{(k+1)} = G_j(x^{(k)})$ for all $j \in \mathcal{B}_l$

The iterative nature of the algorithms being considered offers a natural ability to correct for many faults. As such, if more can be known about the probability distributions pairs (p_i, P_i) it may be possible to guarantee convergence of the algorithm without requiring any algorithmic modifications.

For example, if a fault occurs on the $(F-1)^{th}$ iteration, then the resultant F^{th} iteration can be written

$$\hat{x}^{(F)} = G\left(x^{(F-1)}\right) + \tilde{x}^{(F)} \tag{77}$$

$$\hat{x}^{(F)} = x^{(F)} + \tilde{x}^{(F)} \tag{78}$$

for some unknown perturbation $\tilde{x}^{(F)}$. However, following Section 3.1 of [54], so long as $\hat{x}^{(F)} \in D$ and $\hat{x}_i^{(F)} \in D_i$ for all $i \in \{1, 2, \ldots, n\}$, this can be seen to generate a new sequence that will still converge to the desired fix point.

The next step is to introduce new criteria for convergence that can more easily be exploited in the building of resilient fixed point algorithms. While Theorem 2 and Theorem 5 provide sufficient conditions for convergence, the next result, often referred to as the Banach Fixed Point Theorem, provides an alternative set of conditions required for the convergence of a fixed point algorithm.

Theorem 8 (Banach Fixed Point Theorem). *Assume that the space D is complete with respect to the norm $\|\cdot\|$. If the operator $G : D \to D$ is a contraction map with respect to the same norm $\|\cdot\|$ then there is a unique fixed point $x^* \in D$ such that the sequence defined by $x^{(k+1)} = G\left(x^{(k)}\right)$, converges to the fixed point x^*.*

This has been restated slightly for asynchronous fixed point iterations in the case where $G : D \to D$ is a linear operator (see Theorem 4.1 of [58]) and the case where G is a nonlinear operator (see Theorem 4.4 of [58]). In both cases, convergence is reframed in terms of ensuring that the spectral radius of the operator $|G|$ (if the operator is linear) or the spectral radius of the Jacobian of G, $|G'|$ (when the operator is nonlinear) is less than 1.

For the case of asynchronous fixed point iterations, a further condition is assumed. This is detailed in the following result.

Theorem 9. *Assume that the space D is complete with respect to the norm $\|\cdot\|$. Let $D = D_1 \times D_2 \times \cdots \times D_n$. Let $G : D \to D$ be expressed as,*

$$\begin{aligned}
x_1^{(k+1)} &= G_1\left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_n^{(s_n(k))}\right) \\
x_2^{(k+1)} &= G_2\left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_n^{(s_n(k))}\right) \\
x_3^{(k+1)} &= G_3\left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_n^{(s_n(k))}\right) \\
&\vdots \\
x_n^{(k+1)} &= G_n\left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_n^{(s_n(k))}\right).
\end{aligned} \tag{79}$$

using the framework defined by Algorithm 5 and Definition 1. Let

$$x^{(s(k))} = \left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_n^{(s_n(k))}\right). \tag{80}$$

If each component function, $G_i : D \to D_i$, is a contraction mapping then the asynchronous fixed point iteration defined by

$$x^{(k+1)} = G\left(x^{(s(k))}\right) = \left(G_1\left(x^{(s(k))}\right), G_2\left(x^{(s(k))}\right), \ldots, G_n\left(x^{(s(k))}\right)\right) \tag{81}$$

converges to x^*, the unique fixed point of $G : D \to D$.

Proof. Since each G_i is individually contractive, there exists a sequence of constants $\{\gamma_i\}_{i=1}^n$ that are all less than 1, such that

$$\|G_i(x) - G_i(y)\| \leq \gamma_i \|x - y\| \tag{82}$$

for all $x, y \in D_i$. Set $\gamma = \max_{i \in \{1,\ldots,n\}} \gamma_i$. Then $\gamma < 1$ and

$$\|G_i(x) - G_i(y)\| \leq \gamma \|x - y\| \tag{83}$$

which also gives that

$$\|G(x) - G(y)\| \leq \gamma \|x - y\| \tag{84}$$

for all $x, y \in D$. □

For some fixed point algorithms there are residuals that can be checked to judge progress of an algorithm, but the computation of residuals tends to be expensive. Generally, since the goal of fixed point iteration is to find a point in the domain such that

$$x^{(k)} \approx G\left(x^{(k)}\right), \tag{85}$$

one termination criterion that can be used is to monitor the progression of the successive iterates. In particular, this involves defining a tolerance ϵ with respect to a desired norm and declaring convergence if

$$\|x^{(k+1)} - x^{(k)}\| < \epsilon. \tag{86}$$

Note that this will detect convergence, but runs a risk of declaring convergence prematurely if progression of the fixed point iteration reaches a region of the domain D where progression of the successive iterates falls below the specified threshold. Next, set

$$\delta^{(k+1)} = \|x^{(k+1)} - x^{(k)}\| \tag{87}$$

and additionally, define

$$\delta_i^{(k+1)} = \|x_i^{(k+1)} - x_i^{(k)}\| \tag{88}$$

specific to the individual subdomains D_i. Then, for fixed point iterations that satisfy the component-wise contractive property specified in Theorem 9, the following relation holds:

$$\delta_i^{(k+1)} \leq \gamma_i \delta_i^{(k)}. \tag{89}$$

This can then be used to help develop a fine-grained fixed point algorithm that is capable of being employed asynchronously. The idea being that if $\delta_i^{(k)}$ is computed every iteration then it can be monitored to see if it ever increases. Any increase can be seen as indicative of a fault. This idea is detailed further in Algorithm 8.

This algorithm presents a fine-grained approach towards fault tolerance that does not require any recovery technique or global communication. However, it is possible that Algorithm 8 accepts false positives and using a number $\tilde{\gamma}_j > \gamma_j$ may help prevent this. Additionally, there is still danger of accepting convergence and terminating the fixed point iteration

Algorithm 8: Resilient Parallel Fixed Point Iteration

1 **for** *each processing element* P_l **do**
2   **for** $k = 1, 2, \ldots$ *until convergence* **do**
3   Compute $x_j^{(k+1)} = G_j(x^{(k)})$ for all $j \in \mathcal{B}_l$
4   Compute $\delta_j^{(k+1)} = \|x_j^{(k+1)} - x_j^{(k)}\|$
5   **if** $\delta_j^{(k+1)} \leq \gamma_j \delta_j^{(k)}$ **then**
6   Accept the update $x_j^{(k+1)}$
7   **else**
8   Reject the update $x_j^{(k+1)}$

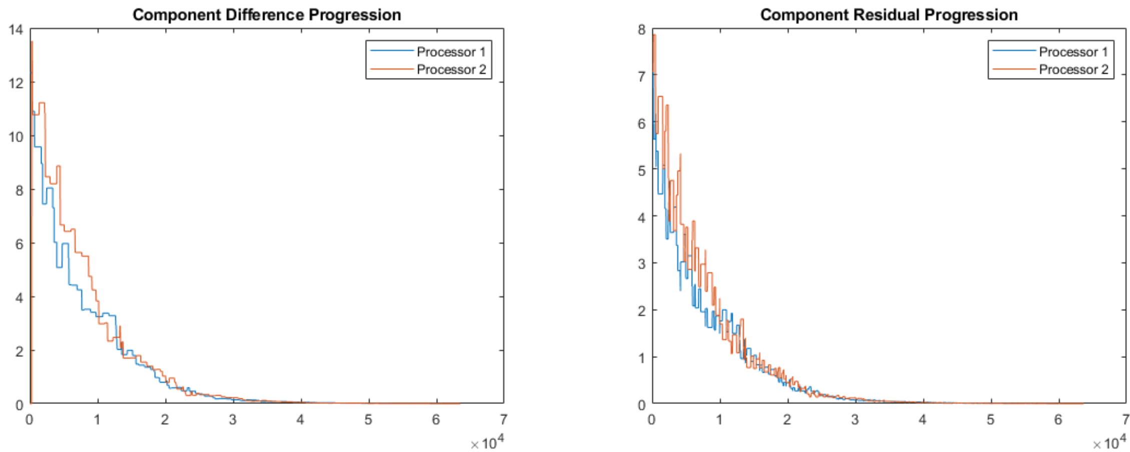


Fig. 9: Component-wise progression of common termination conditions for the asynchronous Jacobi algorithm.

too early if the criterion $\delta^{(k+1)} < \epsilon$ is used. Lastly, there is a danger of stagnation if too many consecutive iterates are rejected. For many algorithms, asynchronous updates to both component-wise differences and residuals remove the guarantee of monotonicity. An example of this non-monotonic behavior for the solution of the Laplacian discretized over a 10 by 10 grid with centered finite differences by the asynchronous Jacobi behavior is shown in Fig. 9. The performance of two processors each assigned to perform updates for half of the 100 total components whose updates patterns are randomized using the simulation framework described in [168] is captured.

One potential solution to these difficulties is to introduce a new parameter, α, that delays the frequency with which the check on progression (c.f. Line 5 of Algorithm 8) is made. In particular, the check on δ is between the $(k+\alpha)^{th}$ iterate and the $(k)^{th}$ iterate. This is shown in Algorithm 9 (see in particular, Line 7).

Example:

For the asynchronous Jacobi algorithm as shown in Fig. 9, the behavior of the local part of the residual exhibits far less monotonic behavior than the norm of the local difference between two successive iterates, suggesting that a larger value of α may be necessary.

The initial value for each of the values $\delta_j^{(k_0)}$ is set to some user defined values N_0 that can be set to any sufficiently large value. Computing the first update in a highly reliable computational mode allows this value to be set appropriately, but the algorithm should proceed as expected for other values since the values of $\delta_j^{(k_0)}$ will be updated every α iterations.

Algorithm 9: Modified Resilient Parallel Fixed Point Iteration

1 **for** *each processing element* P_l **do**
2   Set $\delta_j^{(k_0)} = N_0$
3   **for** $k = 1, 2, \ldots$ *until convergence* **do**
4   Compute $x_j^{(k+1)} = G_j(x^{(k)})$ for all $j \in \mathcal{B}_l$
5   **if** $\mod(k, \alpha) \equiv 0$ **then**
6   Compute $\delta_j^{(k+\alpha)} = \|x_j^{(k)} - x_j^{(k)}\|$
7   **if** $\delta_j^{(k+\alpha)} \leq \gamma_j \delta_j^{(k_0)}$ **then**
8   Accept the update $x_j^{(k+1)}$
9   **else**
10   Reject the update $x_j^{(k+1)}$
11   Set $\delta_j^{(k_0)} = \delta_j^{(k+\alpha)}$

There is discussion in [54] regarding convergence rate, stagnation, and guarding against false positives with respect to the synchronous parallel fixed point iteration, with a focus on the Jacobi algorithm. The final version of their resilient fixed point iteration was built

around a series of synchronous updates to the fixed point equation; however, key points include putting in safeguards to avoid rejecting two consecutive iterates, checking termination conditions against both the current iterate and (a slightly lessened criterion) against the previous iterate. Using properties related to the monotonic progression of the component-wise differences, they are also able to establish practically useful theoretical bounds (c.f. α and β from Algorithm 3 in [54]) that help detect faults with more precision.

A modified version of the check presented here in Algorithm 8 that is specific to the asynchronous Jacobi algorithm with extended discussion of the other points is developed in [21], [26]. The solution presented therein is discussed here in Section 3.2.1 as an example of algorithm based fault tolerance for parallel asynchronous fixed point iterations.

3.2 RESILIENCE STRATEGIES

This section proposes a variety of recovery techniques that can be used to help ensure that convergence of a given asynchronous iterative method occurs without suffering from either an incorrect or significantly delayed result. The first subsection discusses the techniques from a more general, theoretical viewpoint, and the next subsection provides examples of how to use these techniques for specific algorithms.

3.2.1 RECOVERY TECHNIQUES

This subsection will discuss a few methods for ensuring recovery in the d_f iterations specified in Definition 2, or otherwise correcting the behavior of the algorithm. In some cases, the d_f extra iterations can be viewed as corresponding to the self correcting behavior via checkpointing or another type of self corrective behavior.

The several fault mitigation strategies that will be discussed in more detail below are: migration, checkpointing, partial checkpointing, and algorithm based fault tolerance. Each of these methods is capable of restoring nominal performance of an iterative algorithm, and as such, each is capable of bringing the iterative method back to a location in the domain

where the algorithm can converge to the intended result. Throughout the subsequent series of subsections, the examples presented are intentionally kept similar in order to clearly delineate the differences between the various techniques for recovery.

3.2.1.1 Migration

If the processing element, P_k, assigned to compute the updates for block B_k fails, the elements in B_k can be *migrated* to another block/processing element pair, B_l/P_l. Successful migration requires either a flexible component assignment structure, or holding extra processing elements in reserve.

Example:

Consider the centered finite-difference discretization of the two-dimensional Laplacian,

$$\Delta u = f, \tag{90}$$

conducted over an $n \times n$ grid. This results in a matrix, $A \in \mathbb{R}^{n^2 \times n^2}$, that has N non-zero elements. The discrete version of this partial differential equation can then be written as,

$$Ax = b, \tag{91}$$

and the solution to this discretization of the Laplacian can be generated by a stationary fixed point iteration, such as the Jacobi method. If Jacobi is used, the solution for each non-zero element can be written as,

$$x_i = \frac{-1}{a_{ii}} \left[\sum_{j \neq i} a_{ij} x_j - b_i \right]. \tag{92}$$

If the fixed point iteration is asynchronous, each update to every element x_i is made with the latest information available. In this particular example, assume that the computation is being made on a system that has $P > N$ available processors. Further, assume that the work is initially assigned to processors $1, 2, \ldots, p$. If during the course of

the iteration, faulty or delayed performance is detected in processor 1, then the system can assign the work that was originally assigned to processor 1 to processor $p+1$ and let the computation continue as before. If processor $p+1$ starts with the initial guess at the value of x_1 then this can be viewed as a delay in the convergence of element x_1.

Note that the migration could happen with no input or plan a priori by the user, e.g., a user requests the use of 10 nodes, each consisting of 20 processors on an HPC platform, if one of the nodes crashes the system could potentially restart the processes that were assigned to the faulty node on another node of the HPC system.

Recently, several research efforts have proposed the use of what effectively amounts to a migration technique by way of using randomization in the selection of which component to update (see [105]). Following this methodology, each processor selects randomly which component to update before beginning its computation. While this technique is more suitably viewed as a possible mitigation to hard faults (e.g. if a particular processor fails to post an update or has a significant delay in updating due to a hardware malfunction), it may be able to help in the case of soft faults as well.

3.2.1.2 Checkpointing

During the course of the fixed point iteration, at periodic intervals, all elements of the current iterate, $x^{(k)}$, are saved to memory. If a fault is detected, all elements of the corresponding iterate, $x^{(F)}$, are reset to the last known good state, $x^{(k)}$. This method tends to be robust to the occurrence of a soft fault, but may be very slow computationally. Additionally, creating reliable techniques for detecting whether or not a soft fault has occurred often require global communication.

Example:

Consider again the centered finite-difference discretization of the two-dimensional Laplacian over an $n \times n$ grid, where the matrix equation is solved by asynchronous Jacobi. At regular (or semi-regular) intervals during the asynchronous fixed point it-

eration, the latest information available can be used to calculate the current residual,

$$r = b - Ax, \tag{93}$$

and valid values of the iterate $x^{(k)}$ can be written to memory. By keeping track of the residual over time, a fault can be declared if,

$$\|r^{(k+d_r)}\| > \gamma \|r^{(k)}\|, \tag{94}$$

where d_r is the delay in calculating the residual and γ is a constant selected by the user. Values of γ close to 1 assume monotonic decrease in the residual and may therefore declare false positives by detecting a non-existent fault, while values significantly larger than 1 may not detect anomalous behavior if the effect is not sufficiently large.

If a fault is detected all elements of the current iterate, $x^{(F)}$ are reset to the last known good iterate. This reset can be viewed as a delay in the convergence of *all* components.

3.2.1.3 Partial Checkpointing

As in the case of checkpointing above, all elements of the current iterate, $x^{(k)}$, are saved to memory with some regularity. The difference with this method is that if a fault is detected, only some subset of the current iterate, $x^{(F)}$, is reset to the last known good state. In particular, only the components that are determined to be affected by the fault need to be rolled back.

This method requires a finer-grained check on whether or not a fault has occurred, so that the location of the fault can be specified more precisely. Because of this, the partial checkpointing methodology has more natural synergy with fine-grained iterative methods; i.e., it is possible for individual components to detect faults and act accordingly. Further, the computational model of fine-grained (asynchronous) iteration allows for the components

to become out-of-sync with one another. Convergence of the fine-grained iterative is not typically affected negatively if the components that are assigned to a particular processor are reset to a state multiple iterations behind the other components.

Example:

Consider again the centered finite-difference discretization of the two-dimensional Laplacian over an $n \times n$ grid, where the matrix equation is solved by asynchronous Jacobi. Similar to the checkpointing example given in Section 3.2.1.2, at regular intervals during the asynchronous fixed point iteration each processor can independently write the latest valid version of the portion of the iterate $x^{(k)}$, denoted $x_i^{(k)}$, to memory. The latest information available can then be used to calculate the current residual. As opposed to Section 3.2.1.2 where the global residual is calculated, in this method only the local portion of the residual

$$r_i = b_i - Ax_i \tag{95}$$

is monitored. By monitoring the progression of the local portion of the residual over time, a fault can be declared if,

$$\|r_i^{(k+d_r)}\| > \gamma \|r_i^{(k)}\|, \tag{96}$$

where d_r is the delay in calculating the residual and γ is a constant selected by the user. Compared with Section 3.2.1.2, the value of γ may be significantly higher since the decrease of the local residual may not be monotonic. As before in Section 3.2.1.2, values of γ close to 1 assume monotonic decrease in the residual and may therefore declare false positives by detecting a non-existent fault, while values significantly larger than 1 may not detect anomalous behavior if the effect is not sufficiently large.

If a fault is detected, then all of the elements of the current iterate $x^{(F)}$ that are assigned to the processor that detects a fault (i.e., $x_i^{(F)}$) are reset to the last known

good iterate. This reset can be viewed as a delay in the convergence of the components inside of the block x_i.

3.2.1.4 Algorithm Based Fault Tolerance

Algorithm Based Fault Tolerance (ABFT) is a wide class of algorithms that contains many different techniques for making algorithmic modifications that do not necessary rely on more traditional checkpointing style techniques. This includes a sub-class called self-stabilizing techniques as introduced in [125] for the Conjugate Gradient method. Self-stabilizing methods are a type of Algorithm-Based Fault Tolerance (ABFT) that are generally defined as methods that return a system to a valid state within some finite number of steps. This property provides a means for fault tolerance; if a non-persistent fault occurs, the self-stabilizing method should correct any impact such that the algorithm will converge. This technique tends to be application specific as it relies on correcting the values in the current iterate *without* saving a state to memory, and (if possible) avoiding an explicit fault detection mechanism. ABFT techniques include a wide variety of methodologies that are all capable restoring the performance of an algorithm. The example given below provides a method for restoring the convergence of asynchronous Jacobi without requiring computation of the residual.

Example:

Consider again the centered finite-difference discretization of the two-dimensional Laplacian over an $n \times n$ grid, where the matrix equation is solved by asynchronous Jacobi. In this method, due to computational expense, the residual is not calculated. Instead, this method comes from [21], [26] and functions on an individual component level. For each individual component, x_i, there is a constant, ϕ_i $(0 < \phi_i < 1)$, such that,

$$|x_i^k - x_i^{k-1}| \leq \phi_i |x_i^{k-1} - x_i^{k-2}| \leq \phi_i^2 |x_i^{k-2} - x_i^{k-3}| \leq \cdots . \tag{97}$$

Each difference can then be grouped together for convenience – i.e. let $z_i^k = |x_i^k - x_i^{k-1}|$.

If the problem in question defined by the matrix A has a linear convergence rate, than the component specific convergence ratio, $c_i = \frac{z_i^{k-1}}{z_i^k}$, can be used in place of a traditional fault detector since the value for c_i should remain constant. In particular, one should compute component wise convergence ratio values for every element and use them to detect faults throughout the algorithm. In particular, given a valid estimate of c_i, the following bound can be used,

$$\left| \frac{z_i^{k-1}}{z_i^k} - c_i \right| \leq c_i \cdot \delta, \tag{98}$$

where δ is a user-defined threshold parameter. This leads to an algorithm very similar to the one defined by Section 3.2.1.2 where the fault detection is replaced by Eq. (98), and instead of rolling back all the components of the current iterate, $x^{(k)}$, to a previous good state. In this variant, the updates to individual components can be either accepted or rejected on a case-by-case basis.

3.2.1.5 Discussion of Techniques

Generally, the methods for recovering from *any* fault in the case of a fine-grained (asynchronous) iterative method are concerned with making the program in question robust to any negative numerical or fault-induced effects. Given the computational model presented here, it is important to note that typical convergence results will still hold naturally – assuming that the recovery process is executed in a reliable manner – for the following methods: migration, checkpointing, partial checkpointing. This is due to the asynchronous nature which allows for certain elements to be updated significantly after others. If the components associated with a failed block B_F are not updated for some finite amount of time, the asynchronous fine-grained iterative algorithm will still converge so long as the components are *eventually* updated. This eventual update is guaranteed by each of the three methods listed. Convergence for both the general ABFT method and the specific self-stabilizing methodol-

ogy tend to be both application and problem specific. Examples of such results are presented in [26] (ABFT) and [169] (self-stabilizing) for fine-grained iterative methods.

3.3 SUMMARY

This chapter has presented a survey of basic results concerning asynchronous, fine-grained iterative methods for solving fixed point problems, attempted to develop some general analytical statements about how fault tolerance methods can be used for this class of iterative method, and provided examples of how these techniques can be applied to practical problems in High Performance Computing environments. The new theoretical results establish conditions that ensure convergence can still be achieved despite the occurrence of faults. All of the techniques for recovery that are presented in Section 3.2 provide examples based upon the asynchronous Jacobi algorithm, which is a popular asynchronous algorithm and as such provides a meaningful, practical connection between the theoretical results presented in Section 3.1 and the real usage that a user could expect to get. The foundation provided by the theoretical results here can serve as a basis for developing fault tolerant fixed point algorithms which will be shown in detail in the use case presented in Chapter 5.

CHAPTER 4

NUMERICAL MODELING OF FAULTS

The focus of the studies and experiments presented in this chapter is to develop methods for simulating the occurrence of a soft fault, and to to understand the soft error vulnerability of iterative methods with a focus on asynchronous iterative methods. By modeling a soft fault using a more numerical approach, it is possible to drive algorithm development for large scale HPC platforms. When developing a fault tolerant algorithm for future HPC platforms, it is important to ensure that the algorithm is capable of making progress through the worst case behavior that can be induced by faults. Due to the fact that the exact manner that a fault will manifest on future HPC platforms remains unknown, protecting an algorithm against a more generic, numerical form of data corruption may potentially protect an algorithm against ill-effects going forward. Moreover, large-scale sampling of typical fault injection techniques such as inducing bit-flips tends to showcase average behavior rather than worst case behavior [133]. By using numerical approaches, it is much easier to control the size of the effect of a fault and therefore to study the progression of the algorithm when subject to potentially catastrophic impacts due to faults.

One key area is the use of iterative methods for solving sparse linear systems of algebraic equations. Iterative linear solvers constitute a very large use case of computational science and find use throughout different areas in both science and engineering. Much of the previous work on the impact of soft faults (including work on iterative linear methods) has to do with modeling the impact of transient errors. Recent research efforts (see [54], [130], [131], [149], [170]) have focused on modeling the impact of soft faults with a numerical approach that quantifies the potential impact by generating an appropriately sized fault using a more numerically-based scheme; similar to the efforts presented here.

In this chapter, an effort has been made to present data and subsequent analysis on areas that have not been studied as fully. The first area to be studied is the impact of transient soft faults on asynchronous iterative methods. The impact of transient soft faults on traditional, synchronous iterative algorithms has been explored previously (e.g., [126], [130], [131]); however, asynchronous algorithms present a viable means for the creation of scalable algorithms for next generation HPC platforms that are not bottlenecked by the increasing cost of synchronization [1], [2]. The second focus area presented here is on the impact of recurring soft faults (e.g., sticky and persistent faults from Fig. 3) on traditional Krylov subspace methods, where the effect of error magnitude and timing is evaluated for the FGMRES convergence in the solving of an elliptical PDE problem on a regular grid. The implementations make use of both hybrid parallelism, where the computational work is distributed over multiple nodes using MPI and parallelized on each node using OpenMP®, as well as more traditional parallelizations where the distribution of work is done entirely using MPI.

In all of the analysis presented here, the emphasis is on examining the impact that a soft fault may have on an iterative algorithm, and presenting that data in such a way that it can be usefully exploited in the development of fault tolerant algorithms for future HPC platforms. The data shows that the numerical soft-fault models tested here more consistently than a "bit-flip" model produce bad enough behavior to accurately judge the impact that a soft fault may have. This allows for development of successful recovery strategies that will allow for successful algorithm completion despite the occurrence of soft faults. Specifically, two numerical fault simulation schemes are detailed and analyzed in depth alongside the direct injection of bit-flips. Recovery techniques are discussed, and some numerical results related to recovering from the impact of a soft fault are given, however the focus is on observing the impact a soft fault may have. The current expectation is that soft faults will still continue to overwhelmingly manifest as bit-flips, which provides a means of comparison between the numerical models and the direct injection of a bit-flip that can be used as a starting point

for the use of the numerical soft fault models in the development of algorithmic variants. Some results that are provided here have been previously published in [149], [171]–[174].

The structure of this chapter is organized as follows: in Section 4.1, an extended discussion of the numerical modeling of soft faults is provided, along with a discussion of different techniques for recovering from soft faults. Next, in Section 4.2, a series of numerical results are provided (Section 4.2.1 focuses on asynchronous iterative methods and Section 4.2.2 focuses on Krylov subspace methods), while Section 4.3 concludes.

4.1 SIMULATING SOFT FAULTS

In a majority of studies conducted on the fault tolerance of iterative methods, the occurrence of an undetected soft fault is overwhelmingly treated as a bit flip (see [126]). Traditionally, bit flips are injected randomly according to a given distribution (e.g., a Poisson or Weibull distribution) or in a more frequent manner designed to showcase worst case behavior. However, as the effect of a bit-flip (i.e., the amount of data corruption introduced) can vary wildly depending on which bit is affected, this necessitates a large number of runs to reveal statistically average behavior [133]. In this dissertation, a series of experiments utilizing the direct injection of bit flips into memory is presented; additionally, more generic fault injection techniques are included. Generally, the following outcomes are most likely to occur when a fault occurs during the execution of an iterative solver [126], [134]:

- The solver will converge in the approximately the same number of iterations, with an error in the final solution.
- The solver will converge in the approximately the same number of iterations, with no error in the final solution.
- The solver will converge in more iterations than in a fault free run; with or without an error in the final solution.
- The progress of the solver towards the solution will stagnate, and it will fail to converge.

Following the methodology outlined in [133], the numerical fault models used here are inspired by the idea of modeling an undetected soft fault as data corruption. That is, instead of trying to model the exact impact of a fault on future large scale HPC machines, faults are treated as corrupted data where the size of the corruption can be controlled in an effort to produce consistent worst case behavior and help with the development of fault tolerant algorithms. To help illustrate the effect of each soft fault model that is discussed, a simple visualization is presented for each model. In these examples, there is a single data structure, a vector x that consists of 8 elements, and two processors that are each assigned 4 consecutive elements of the vector x. This set-up is depicted in Fig. 10.

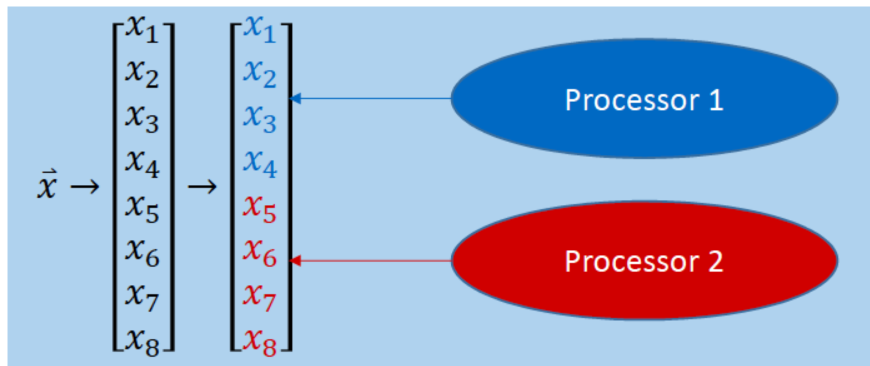


Fig. 10: Simple fault injection baseline example.

The goal of considering a variety of soft-fault models is to produce fault-tolerant algorithms that are not too dependent on the precise mechanism of a fault injection, such as a bit-flip, in future computing platforms. Note that, in this work, faults are injected only into the data used by the algorithm as opposed to the metadata that includes pointers, indices, and other data-structure descriptions, because the metadata, while necessary to be fault-free also, is tied to a specific implementation of the given algorithm on a given architecture, which is beyond the scope of this dissertation.

4.1.1 BIT-FLIP SOFT FAULT MODEL (BFSFM)

The first method of simulating a fault adopted in this dissertation is via the direct injection of a bit-flip into a data structure. While it is important for future computing platforms not to become too dependent on the precise mechanism that is used to model the instantiation of a fault, since bit-flips are currently the most likely form of a soft fault to affect HPC hardware, it is important to include analysis that responds to the effects of having a bit-flip occur during the run.

The visualization of this fault injection is very simple. A domain assigned to a particular processing element is designated to suffer a fault and an element is selected at random to encounter a bit-flip in a randomly selected bit. This corruption is illustrated in Fig. 11. Note that the element x_2 differs from $\tilde{x_2}$ by a single (randomly selected) bit.

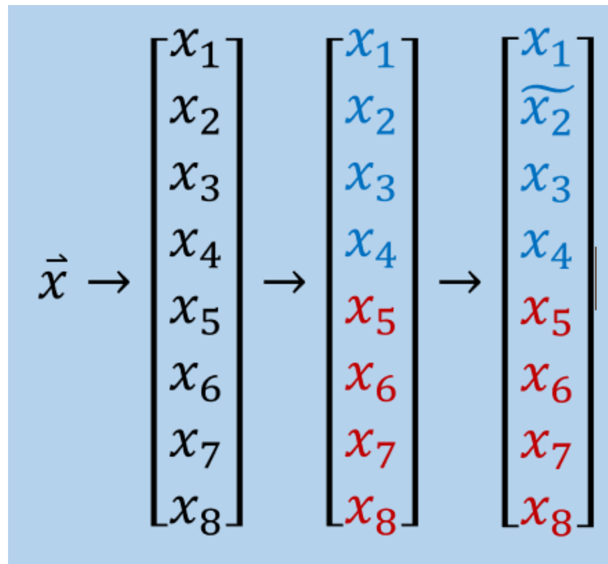


Fig. 11: BFSFM injection example.

4.1.2 PERTURBATION-BASED SOFT FAULT MODEL (PBSFM)

This approach models faults as perturbations inside of a single subdomain, and has been already used in several other recent studies (see [54], [149], [169], [170], [175]) along with similar approaches. In the PBSFM, a small random perturbation τ that is sampled from a uniform distribution of a given size is injected transiently into each component representing a value of the targeted data structure. For example, if the targeted data structure is a vector x and the maximum size of the perturbation-based fault is ϵ, then proceed as follows: (1) generate a random number $\epsilon_i \in (-\epsilon, \epsilon)$, (2) set $\hat{x}_i = x_i + \epsilon_i$ for all values of i. The resultant vector $\hat{x}$ is, thus, perturbed away from the original vector x.

A visualization of this process is provided in Fig. 12. In this simple example, all 4 of the elements in the affected subdomain have been adjusted by unique random perturbations, $\epsilon_i, i = 1, 2, 3, 4$.

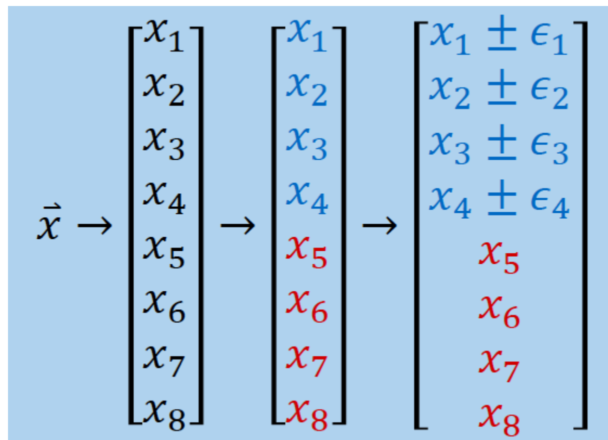


Fig. 12: PBSFM injection example.

4.1.3 SHUFFLE-BASED SOFT FAULT MODEL (SBSFM)

This approach models faults primarily as a shuffling of the elements inside of a subdomain. This approach was originally detailed in [134] and simulates the occurrence of a soft fault by

a permutation of the components inside of the subdomain in which a fault was injected, a scaling of the data inside of the subdomain in which a fault was injected, or a combination of these two effects.

Since progression of the iterative methods studied here relies on the progression of the $L-2$ norm of the residual, the effect of the SBSFM can be divided into three scenarios:

1. $\alpha = 1$: $\|x\|_2 = \|\hat{x}\|_2$
2. $0 \leq \alpha < 1$: $\|x\|_2 > \|\hat{x}\|_2$
3. $\alpha > 1$: $\|x\|_2 < \|\hat{x}\|_2$,

where α is the scaling factor, x the original vector, and $\hat{x}$ is the vector with an injected fault.

The analysis that was performed in [131], [134] details the impact of the SBSFM model in the case where it is modeling transient soft faults with various scaling values for traditional synchronous iterative methods; specifically Krylov subspace methods such as GMRES and CG. The focus in this chapter is on the effect of the SBSFM for transient faults with respect to asynchronous iterative methods, and for sticky faults for Krylov subspace methods. The impact of this fault model relative to the impact of a single bit flip is studied in [134] and shows that regardless of where the bit flip occurs, the SBSFM will perform in a similar way to the worst case scenario induced by BFSFM. Analysis showing the impact of a bit flip based on where in the storage of a floating point number it occurs is given in [132].

A visualization of the effects of the SBSFM is provided in Fig. 13.

4.1.4 ADAPTATIONS FOR NON-TRANSIENT FAULTS

As shown in Fig. 14, soft faults can be divided into three distinct categories: transient faults, sticky faults, and persistent faults. Throughout this work, transient faults are modeled with an instantaneous error, injected at the specified point of execution, and the remaining two types of faults are modeled through the use of recurring injection of data corruption.

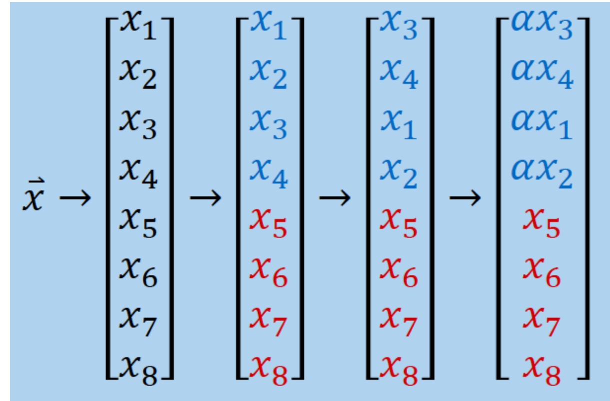


Fig. 13: SBSFM injection example.

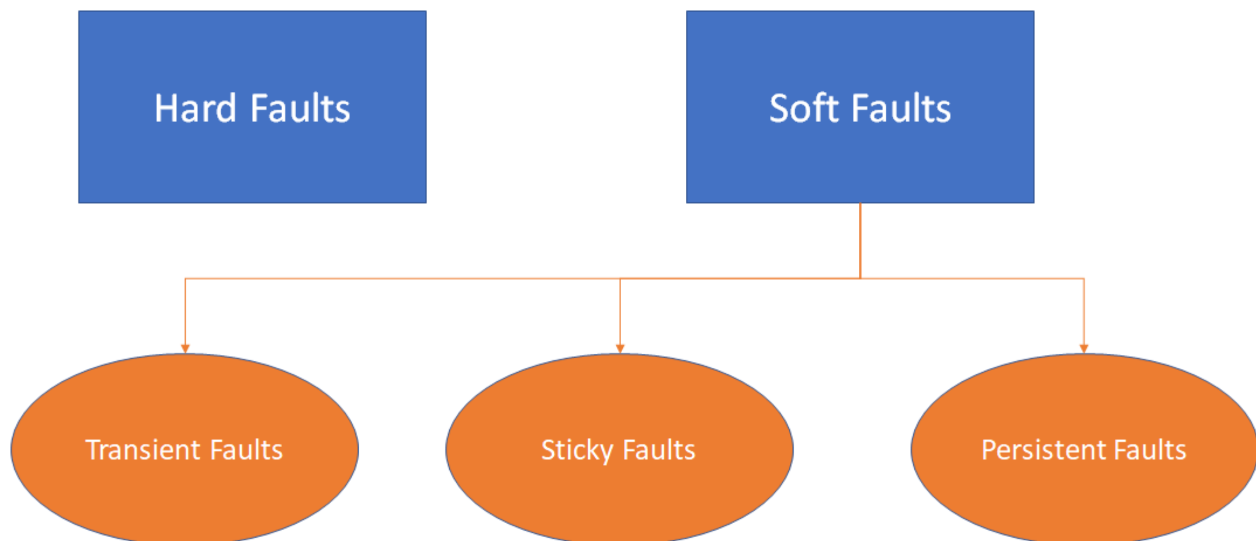


Fig. 14: Breakdown of Types of Soft Faults.

Sticky faults are meant to represent all soft errors whose effect lingers for a long enough period of time to force the fault to be treated as something other than instantaneous, but where the source of the data corruption related to the fault eventually (naturally) stops corrupting data. Note that this does not remediate the effect of the fault that was caused while the fault was occurring, but simply means that the fault causing mechanism ceases to continue to occur. Here, sticky faults were simulated through calling the routine that implements the specified numerical soft fault model (for sticky faults, either the PBSFM or

the SBSFM) on consecutive iterations. In this manner, for a period of time – controlled by the specified number of iterations – the iterative solver is being executed by hardware in a temporarily faulty state.

Persistent faults are representative of hardware malfunctions or errors that induces undetectable data corruption into the data elements that they are processing. Note that if the hardware malfunction were detectable by the HPC platform, it is likely that execution would be terminated. The focus throughout this chapter is on *soft* faults which represent those faults linked to undetected data corruption.

Persistent faults were simulated through calling the routine that implements the specified numerical soft fault model (for persistent faults, the PBSFM) for *every* iteration from when the fault is designated to begin until the solver execution terminates. For example, if the persistent fault was scheduled to begin occurring on the 20th iteration then the numerical soft fault model would be called upon to inject data corruption into the desired component for every iteration from the 20th until the routine finished.

4.1.5 ANALYSIS OF FAULT INJECTION UTILITIES

4.1.5.1 Comparison of PBSFM, SBSFM, and BFSFM

To compare the potential effects of the various fault injection techniques used here—with an emphasis on the total amount of data corruption induced—a short analysis is presented for the problem studied in Section 4.2.1. The discretization of the Laplacian described in Section 4.2.1.1 results in a matrix of size 160000×160000 if the problem is converted to the matrix form $Ax = b$. Letting the initial x be a vector of all zeros and the initial b be a vector of all ones, the iterate of x examined here is the 50th iterate, denoted $x^{(50)}$.

For this iterate of x from the matrix representation of the problem, the first quarter of the elements are isolated (i.e., to correspond to one of the four subdomains that is assigned to a single MPI process as described in Section 4.2.1.1), and then the first tenth of these elements

are isolated (corresponding to the components assigned to the first OpenMP®thread) and the effect of fault-models is examined for this localized subdomain in terms of corruption of the vector x. This reflects the mechanism with which faults are expected to appear, i.e. as isolated incidents, and the experiments conducted here limit them to the most local subdomain resident to them. The methods that are compared are as follows:

- PBSFM large: $\tau_i \in (10^{-16}, 10^{16})$, medium: $\tau_i \in (10^{-8}, 10^{8})$, and small: $\tau_i \in (10^{-2}, 10^{2})$.
- SBSFM large: $\alpha = 10^{14}$, medium: $\alpha = 10^{6}$, and small: $\alpha = 10^{2}$.
- BFSFM: one vector component randomly selected, in which one bit is randomly selected.

A total of 10,000 trials were run, and aggregate data is presented in Table 1. The total amount c of data corruption is measured as $c = ||x - \hat{x}||$, where $\hat{x}$ represents the iterate under study with the specified fault injected. Mean and median information is provided over the 10,000 trials as well as the average and standard deviation of the logarithm of c, which provides some insight into the average order of magnitude of corruption and how wide the spread of potential outcomes is. Note that the range of impacts is wider for the BFSFM, but the average impact is the worst for the numerical fault models.

Table 1: Comparison of different fault injection techniques.

	Mean(c)	Median (c)	Mean(log(c))	Std(log(c))
PBSFM (L)	3.65E+17	3.65E+17	40.44	0.007
PBSFM (M)	3.65E+09	3.65E+09	22.02	0.007
PBSFM (S)	3.65E+03	3.65E+03	8.20	0.007
SBSFM (L)	5.77E+17	5.77E+17	40.90	7.30E-12
SBSFM (M)	5.77E+09	5.77E+09	22.48	7.41E-09
SBSFM (S)	5.74E+05	5.74E+05	13.26	7.51E-05
BF	1.13E+304	3.05E-05	-0.81	52.8

4.1.5.2 Comparison of PBSFM and SBSFM

Examining this for the test problem that is focused on in Section 4.2.2, the result of the outer matvec operation in the FGMRES algorithm is a zero vector initially and as the FGMRES algorithm progresses closer to the solution, this vector will begin to approach the original right hand side of the equation, b. In the iteration associated with this problem, the entries in the final iterates of the product Ax_i before convergence will have entries, b_i, where $-0.01 \leq b_i \leq 0.01$ forms a loose bound on all entries of $b = Ax_i$. To show the potential difference in magnitude between a given vector b and a vector $\hat{b}$ representing the vector b with a fault injected, 1000 random vectors were generated in MATLAB®[1] for vectors b of varying sizes (to represent varying problem sizes) and the l^2 norm of $\|b - \hat{b}\|$ was calculated for each of the two fault models. These results are shown in Table 2.

Table 2: Difference in the effect of each of the fault models on random vectors with entries in (-0.01, 0.01). Note: The scaling factor in the SBSFM was set to 1.0 and the fault size in the PBSFM was set to 5×10^{-4}. Columns 2 and 3 represent average differences over 1,000 runs.

Vector Size	$\|b - \hat{b}\|^2$ - SBSFM	$\|b - \hat{b}\|^2$ - PBSFM
10	0.0238	0.0009
100	0.0814	0.0029
1,000	0.2581	0.0091
10,000	0.8166	0.0289
100,000	2.5824	0.0913
1,000,000	8.1650	0.2887

The SBSFM allows slightly more exact statements to be made concerning the effect of the injected fault on the l^2-norm, as the l^2-norm will be the exact same for all but the affected subdomains since shuffling the elements of a vector does not change the l^2-norm. However, the size of the fault, measured as a difference from a fault free run, is dependent on both the problem size and the size of the entries in the data structure in the case of the SBSFM.

[1] MathWorks, Inc., Natick, MA

For example, if the size of the entries in b is allowed to increase to the range $(-1, 1)$, the size of the fault for a subdomain with 1,000 entries for the SBSFM (c.f., Row 3, Column 2 of Table 2) increases from an average of 0.2581 to 538.1. On the other hand, statements concerning the l^2-norm are inherently less exact when the PBSFM is used, as the l^2-norm of the faulty subdomain is not precisely controlled, but the difference from a fault free run, i.e. the "size" of the fault, is easier to control by way of simply adjusting the bounds on the perturbation that is used.

In looking to see which of the two fault models induces a "larger" fault, in general it will be the case that the SBSFM will create a larger difference between a given data structure with a fault injected and the same data structure in a fault free environment.

4.1.5.3 Comparison of BFSFM and PBSFM

Next, a further comparison between the bit-flip model and the PBSFM is presented. As stated above, each fault model works on an input vector, x, and corrupts in some way the specified component(s). In order to illustrate the potential impact of each fault model, x is taken to be the initial set of non-zero components for the 2D finite difference discretization of the Laplacian. The matrix was symmetrically scaled to have unit diagonal, so that the entries in the vector x are bounded inside of $[-1, 1]$.

Due to the non-deterministic nature of both of these fault models, the comparison between them was made over 1000 trials. In each trial, a fault is injected according to one of the methodologies in order to create a vector with a fault $\hat{x}$, and the norm of the difference in these two quantities,

$$d = ||x - \hat{x}|| \tag{99}$$

was computed. In this comparison, the magnitude of $\hat{x}$ is bounded for the perturbation-based fault model, but it is possible for the bit-flip fault model to produce a result of either `NaN` or `INF` for certain combinations of component and bit selected. For the purposes of this

Table 3: Comparison of the effects between the various fault models used for the matrix LAPLACE2D.

	Bit-flip Model	Bit-flip Model (bounded)	PBSFM (s)	PBSFM (m)	PBSFM (l)
mean(d)	—	8.2388e-02	6.4500e+00	6.4499e+02	6.4499e+04
max(d)	4.4942e+307	1.0000e+00	6.4593e+00	6.4575e+02	6.4584e+04
mean(log(d))	-3.2281e+00	-7.0040e+00	8.0956e-01	2.8096e+00	4.8096e+04
std(log(d))	3.4646e+01	5.0639e+00	1.7075e-04	1.7287e-04	1.7194e-04

quick look analysis, these results were discarded since scanning for either of these incorrect values is not a difficult problem. Summary results are provided in Table 9.

In Table 3, the 'Bit-flip Model' column corresponds to randomly selecting a single component of the vector x, randomly selecting a bit to flip, and injecting a single bit-flip. The column 'Bit-flip Model (bounded)' corresponds to the same bit-flip model, but where bit-flips that caused large magnitude changes due to bit-flips in exponent bits were removed. In particular, any instance where $d > 10000$ was removed from the data. The three columns corresponding to the perturbation-based soft fault model (PBSFM) are separated by the bounds on the range that the perturbations were sampled from. The (s) column corresponds to faults in $r_i \in (-0.01, 0.01)$, the (m) column to faults in $r_i \in (-1, 1)$, and the (l) column relates to faults in $r_i \in (-100, 100)$. The vector d corresponds to the size of the fault introduced by the given fault model. In the table, the mean of the 1000 entries of d is provided, along with the maximum value, and the mean and standard deviation of the log of the entries in d.

The data presented in Table 3 shows the potential impact of a fault introduced by each of the fault models. Note that the 'Bit-flip Model' contained 12 cases (1.2%) that exceeded the threshold of $||x - \hat{x}|| > 10000$, indicating that while a severely large impact is possible, it is not probable. The statistics on the log values of the entries in d gives some indication as to the relative order of magnitude of the various fault models, and the spread of the level of impact. Generally, the size of the faults induced by the bit-flip model are much more varied

than those created by the perturbation-based soft fault model.

4.1.6 TECHNIQUES FOR RECOVERY FROM SOFT FAULTS

Formulating efficient techniques that allow an algorithm to recover from soft faults is an important area of research as HPC platforms progress towards exascale. The prevailing wisdom is that globally checkpointing all processors will not be feasible computationally for large-scale problems due to the immense costs of reading/writing data and globally communicating [4], [5].

While this is true for any iterative method, for the fine-grained asynchronous iterative algorithms the use of global (or even large-subgroup) communication is prohibitive because synchronization used in such communications goes against the very nature of these algorithms, which rely on a great number of light-weight thread or process computations. In the investigation into the impact of the numerical soft fault models on asynchronous iterative methods, a partial checkpointing method is used that avoids many of the communication related pitfalls inherent in the simple global checkpointing algorithm. This method is similar to the partial checkpointing method used for the fine-grained parallel incomplete LU factorization in [169], [170], [176].

In order to devise a mechanism that can be used to indicate the presence of a fault, the algorithm under study needs to be examined to find suitable metrics that can be used to monitor the progression of the algorithm itself. Ideally, obtaining these values will present a very minimal computational effort. Progress of the Jacobi algorithm – either in the synchronous or asynchronous case – is often judged by the progression of the norm of the residual vector,

$$\|r\| = \|b - Ax^{(k)}\|. \tag{100}$$

However, checkpointing based on the progression of the residual after it is recovered from all components of $x^{(k)}$ necessitates communication among all the OpenMP®threads as well as all the MPI processes. Additionally, the regular computation of the residual can present

a relatively significant computational burden; i.e., the amount of effort required to compute the residual is not trivial compared with the amount of effort required to compute an update to the individual components, x_i.

The partial checkpointing method studied here checkpoints only based on the norm of the local portion of the residual vector, denoted r_l for the lth component. As the asynchronous computation progresses, each thread writes *periodically* the current value of the components x_l that reside in the block for which it is responsible to compute updates to a checkpoint. The periodicity is treated as a parameter, denoted m in Eq. (101), which is studied in Section 4.2.1.

After the thread updates its components in the kth iteration, $x_l^{(k)}$, it checks the current local residual to see if a fault has occurred:

$$\|r_l^{(k)}\| > \gamma \cdot \|r_l^{(k+m)}\|, \tag{101}$$

where γ is the checkpoint threshold explored in Section 4.2.1, and $r_l^{(k)}$ and $r_l^{(k+1)}$ are local residuals for the iteration k and $k+1$, respectively.

4.2 NUMERICAL RESULTS

4.2.1 ASYNCHRONOUS ITERATIVE METHODS

4.2.1.1 Parallel Implementation

The asynchronous Jacobi implementation used in this portion of the dissertation makes use of hybrid MPI-OpenMP®parallelism. This implementation focuses on solving a two dimensional finite-difference discretization of the Laplacian on a 400×400 grid; including the boundary values the total problem size is 402×402. The problem is solved by a matrix-free implementation of the Jacobi algorithm whereby the approximate solution to the Laplacian

is stored in place at the appropriate grid values. Matrix-free implementations provide an efficient means to solve PDEs on regular structured grids.

The work is divided among five MPI processes, but only four perform computations. One MPI process acts as a master process, which communicates with workers for memory transfer and global residual calculations. Each of the four worker processes is assigned an equal amount of the entire domain, which leads to each subdomain consisting of 200×200 grid points. Note that the working size of each subdomain grid will be 202×202 due to keeping track of the necessary halo values (i.e. a mixture of values from the boundary and neighboring subdomains). The work is parallelized inside of each subdomain using OpenMP®.

For an n by n grid that is equally divided among the n_p threads, each thread solves for n^2/n_p grid points, such that the grid is evenly partitioned along the y-axis. Ten OpenMP®threads were used for each MPI process, which gives each thread $200 \times 20 = 4000$ vector components to compute updates for.

Internally, two matrices U_0 and U_1 store the grid point values that each thread reads, e.g. from U_1, to compute newer values to write, e.g. to U_0. As the method is asynchronous, each thread independently determines which matrix stores its newer $u^{(t+1)}(i, j)$ values and older $u^{(t)}(i, j)$ values. When a thread copies grid-point values located above or below its domain, OpenMP®locks are employed to ensure that data is captured accurately, from a single iteration.

Further, locks are used when updating values on boundary rows and subdomain halos, and when copying subdomain boundaries. Each thread p_n computes its local residual value every k^{th} iteration, which it contributes to the set of residual values for the subdomain. Using an OpenMP®atomic operation, a single thread copies the set of subdomain residuals, computes a sum, and sends the sum to the master MPI process. The subdomain is equally divided among all OpenMP®threads, but in order to avoid a negative effect on the performance of a single OpenMP®thread, communication with the master MPI process is rotated among the

threads.

4.2.1.2 Experiment Set-up

Experiments were conducted on the Turing High Performance Computing cluster at Old Dominion University, which contains 190 standard compute nodes, 10 GPU nodes, 10 Intel Xeon Phi®Knight's Corner nodes, and 4 high memory nodes, connected with a Fourteen Data Rate (FDR) InfiniBand®[2] network. Compute nodes contain 16–32 cores and 128 GB of RAM. Data were collected on sockets consisting of 10 Intel Xeon®E5-2670 v2 2.50 Ghz cores.

4.2.1.3 Baseline Case

Before delving into the results regarding the impact and recovery of soft faults on the hybrid parallel iterative solver used here, a set of baseline runs is presented. The problem described in Section 4.2.1.1 is solved 500 times, and a histogram showing the distribution of total run times, and mean and standard deviation, is presented in Fig. 15. Some variation in run time is observed, but this is not unexpected for an asynchronous solver. A wide variation in iterations until convergence is seen in [108], and [177] shows increased run time variation for asynchronous solvers. Here, the run time for +3 standard deviations is 1.31 times the minimum run time. Almost 98% of runs are less than +3 standard deviations.

4.2.1.4 Impact of Soft Faults

The following model parameter values were used

- For PBSFM, the pertubation τ values were taken from a set of intervals $(10^{-2j}, 10^{2j})$ for $j = 1, \ldots, 8$.
- For SBSFM, the α values were $10^2, 10^6, 10^{10}$, and 10^{16}.

[2] InfiniBand Trade Association, Beaverton, OR

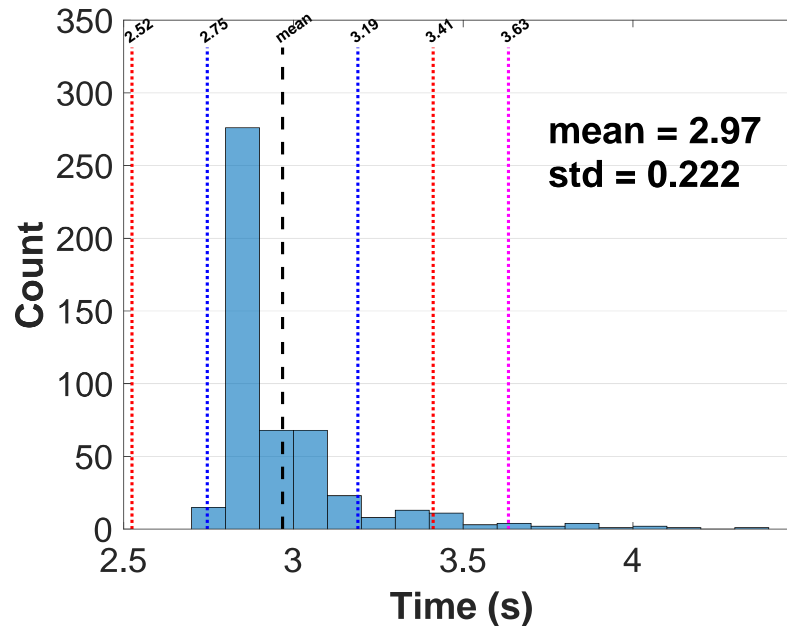
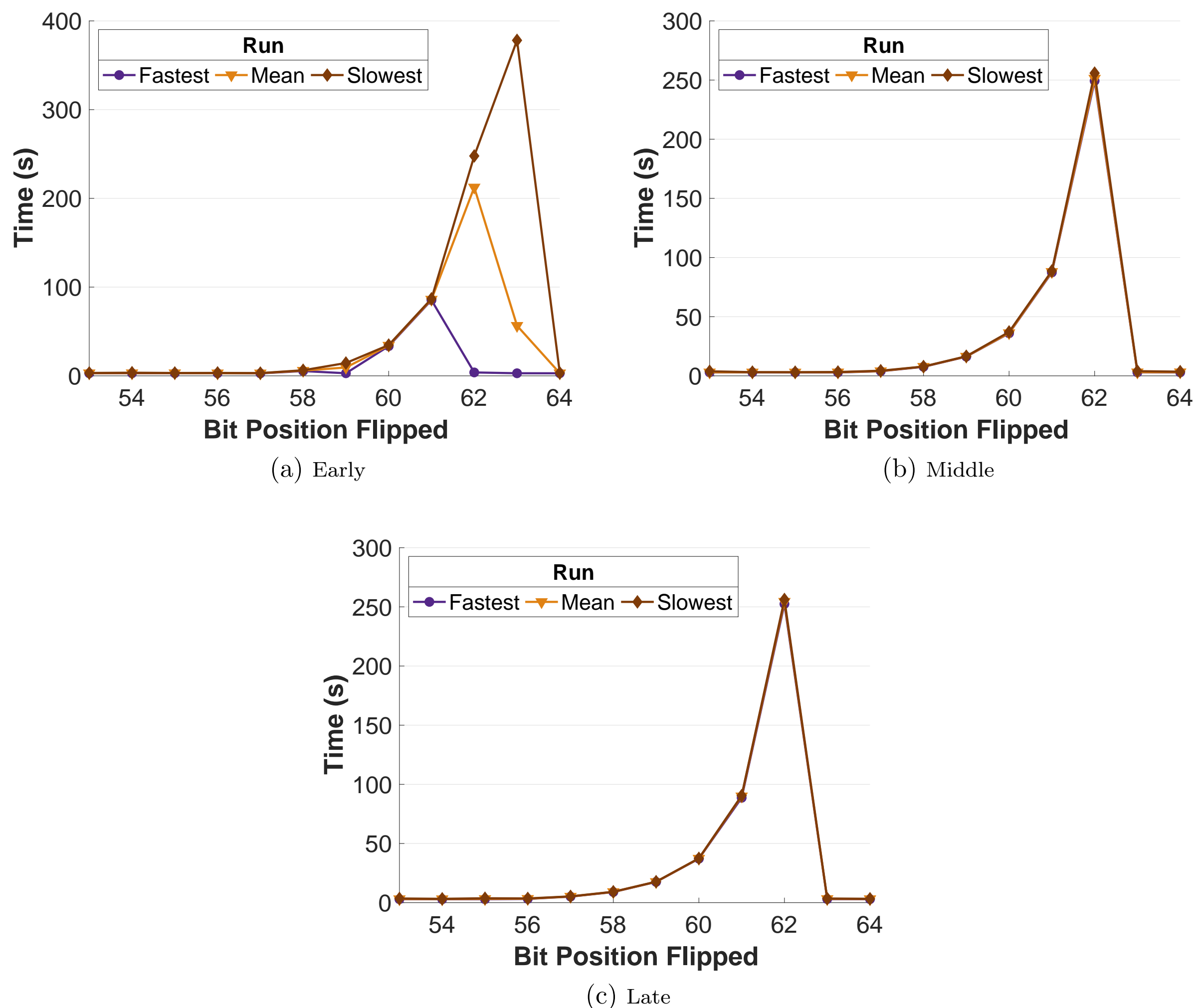


Fig. 15: Distribution of run times in a fault-free environment.

Based on the mean runtime of 2.97s shown in Fig. 15, three different fault injection times were used as follows: *early*, equal to $0.1s$, *middle*, equal to $1.2s$, and *late* equal to $2.5s$.

Figures 16 to 19 show the effects from faults injected by each model at *early*, *middle*, and *late* time points. In addition, the effects of bit flips restricted to sign or exponent are distinguished from those restricted to mantissa in separate plots, Fig. 16 and Fig. 17, respectively. Each experiment was replicated seven times on Turing. The plots show the results from the fastest, slowest, and average of these seven runs. In all experiments, the solver converged to a correct solution, within a tolerance of $1e^{-4}$.

Figure 16 shows that flipping an exponent bit early in the run, when grid point values may still be small, might not be as harmful as a later bit flip. Across all the faults models, faults injected early tend to have more of an effect on the total time for the solve to complete. Note also that, while bit-flip faults in the exponent and sign bits can have a catastrophic effect, bit-flips occurring in the mantissa have very little impact on the performance of the solver (cf. Fig. 15). PBSFM and SBSFM force more consistently bad behavior. This



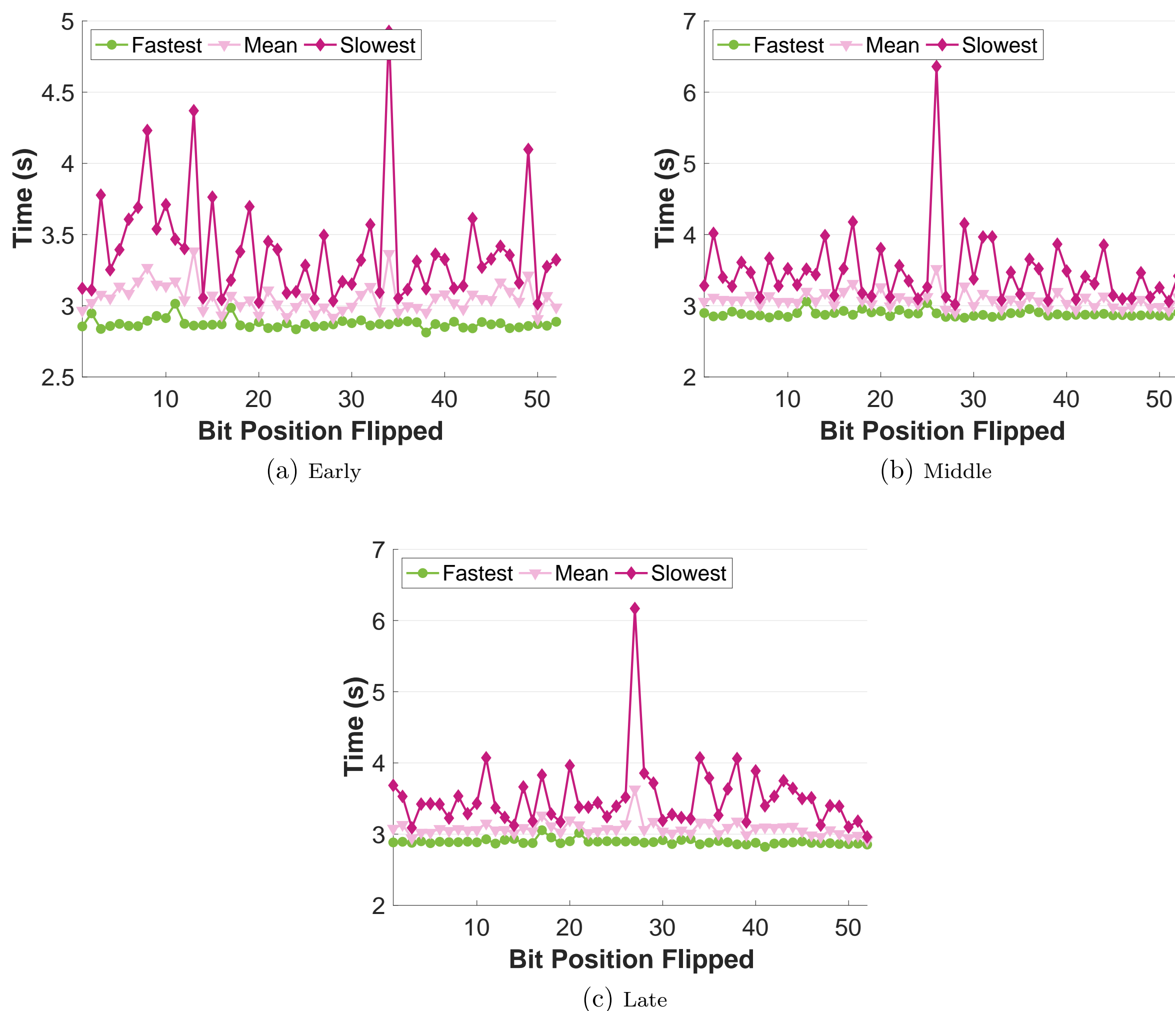
(a) Early (b) Middle (c) Late

Fig. 16: Effect of bit-flip faults in the exponent and sign bits.

reinforces experimental outcome (see Table 1 in Section 4.1): a numerical soft-fault model can more effectively force an algorithm to run through bad behavior, while a stochastic bit-flip injection may force an extreme behavior, but may have little effect. Numerical soft fault models afford users a higher level of control.

4.2.1.5 Recovery from Soft Faults

Consider the partial checkpointing scheme detailed in Section 4.1.6. The fault is injected near the *middle* time, at 1.4s, in each run. The values of γ in Eq. (101) are 1.01, 1.05 and



(a) Early

(b) Middle

(c) Late

Fig. 17: Effect of bit-flip faults in the mantissa bits.

1.25 to test for very, moderate, and least sensitive fault detection, respectively.

At the end of an iteration, a thread compares the current component residual value to a previous value. If a fault is detected as an increase of the residual by more than the specified γ, the thread(s) that detected the increase roll(s) all of the components present in their subdomain back to the last checkpoint and continue(s) calculating updates as before. A thread checkpoints after completing four iterations that do not require a rollback.

Figure 20, compared with Fig. 16, shows that the checkpointing and rollback technique employed for this work effectively managed the exponent bit-flip fault injections. Comparing

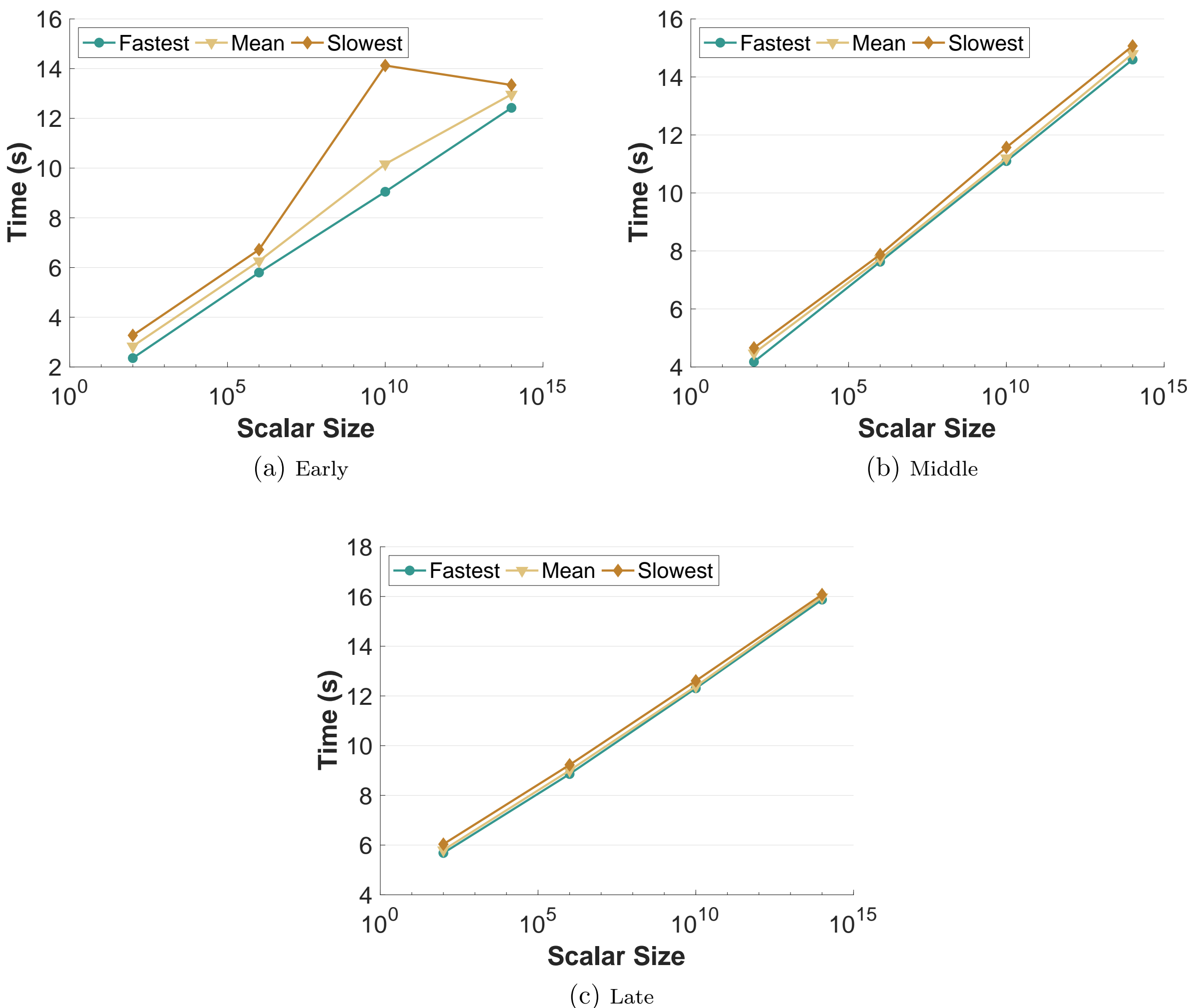


Fig. 18: Effect of faults injected using the SBSFM.

Fig. 21 with Fig. 17 yields little difference—as expected—while attesting to only a moderate overhead of checkpointing. In particular, the largest mean value in Fig. 17(b) was $\sim 3.5s$ while the largest mean value in Fig. 21(c), i.e., for the least sensitive fault detection, at $\gamma = 1.25$, was $\sim 3.8s$

If corrupted values are on the edge of a thread compute region, they may spread to the compute region of a neighboring thread and compromise resiliency. This behavior is more readily observed when using numerical soft fault models, such as PBSFM and SBSFM, since they impact all of the components assigned to the thread, including boundary values.

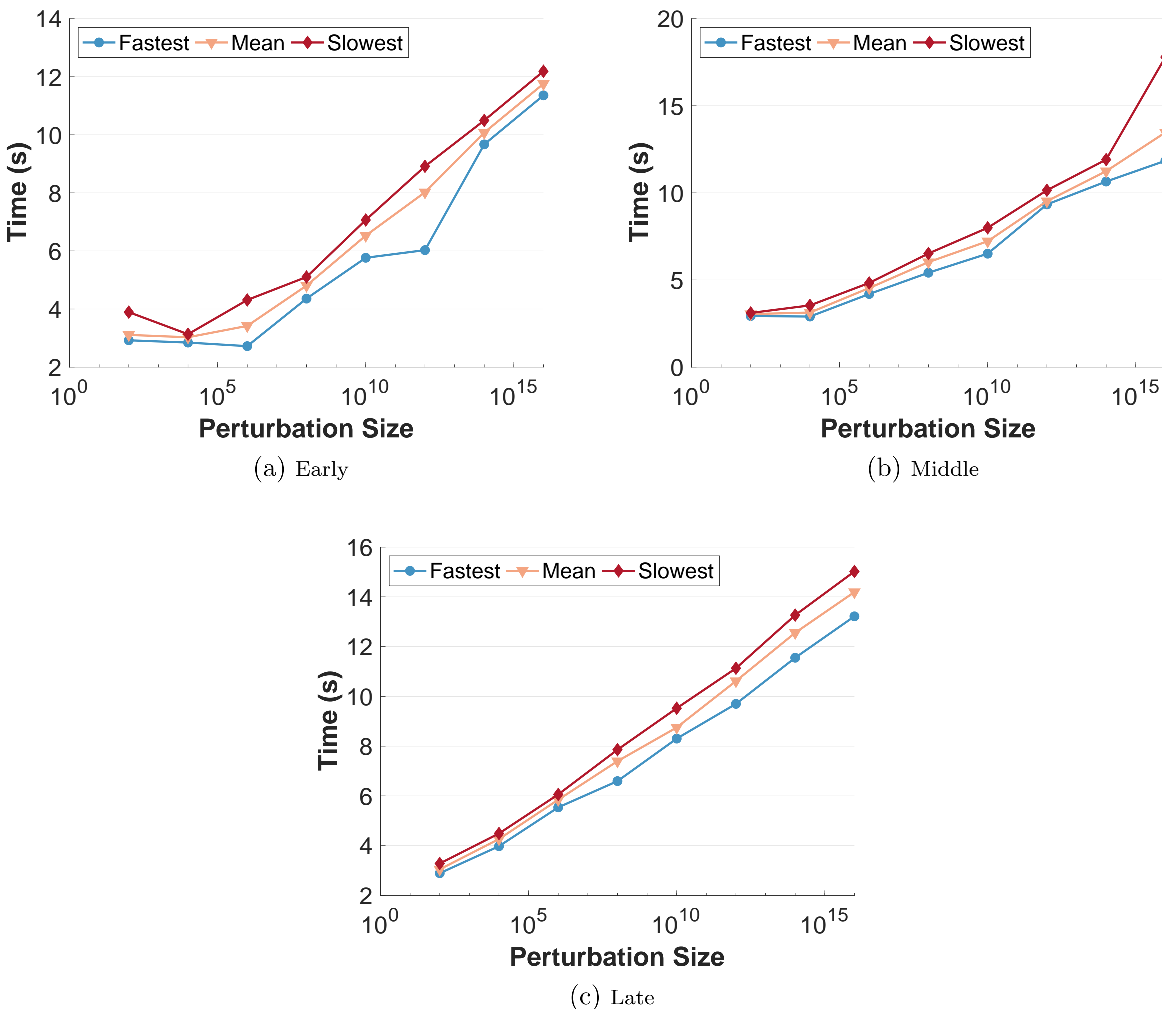
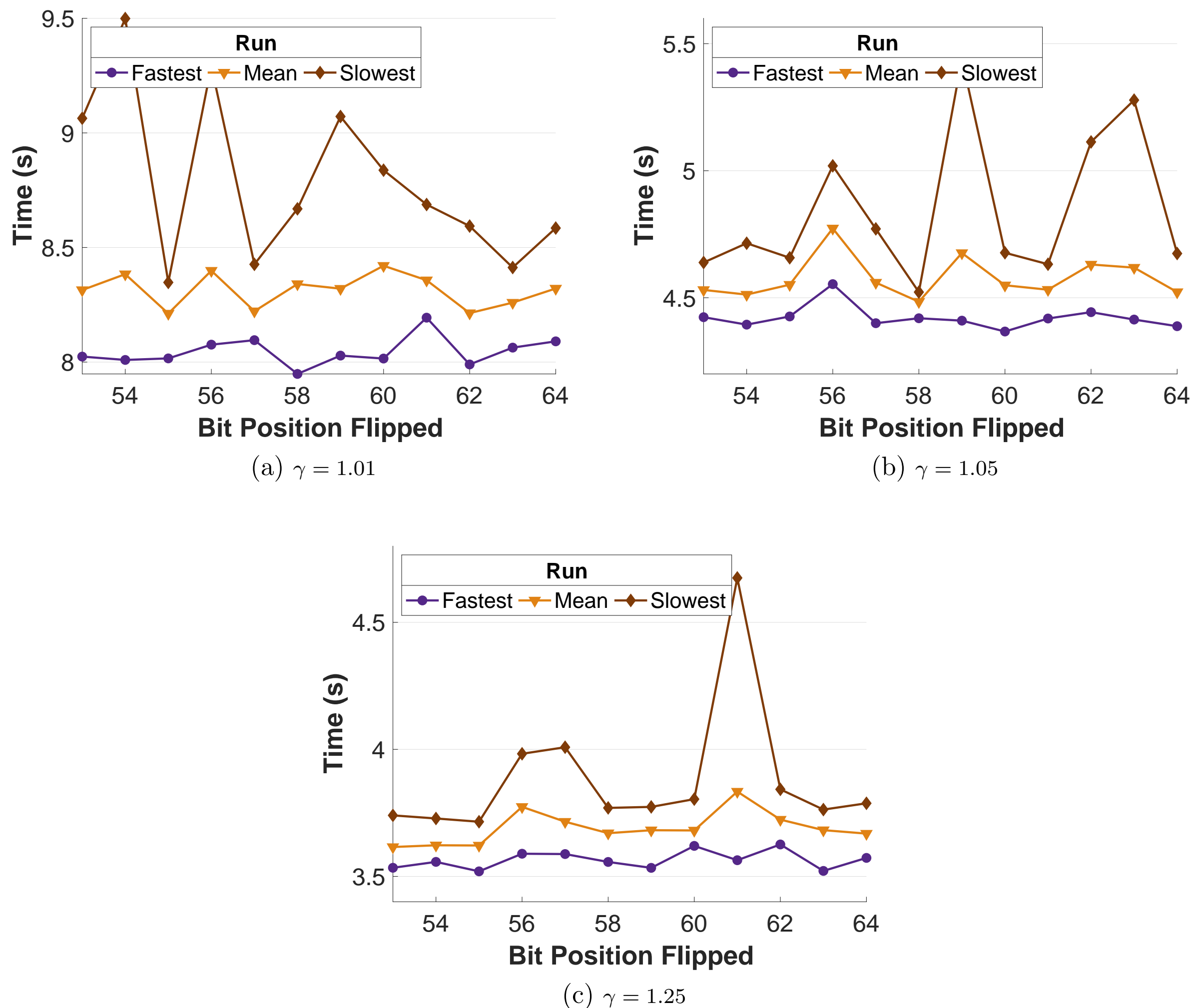


Fig. 19: Effect of faults injected using the PBSFM.

Hence, a trade-off between the sensitivity of the fault-detection and checkpointing overhead is desirable. For example, compare plots for $\gamma = 1.05$ in Figs. 22 and 23 with the ones for smaller and larger γ values for SBSFM and PBSFM respectively.

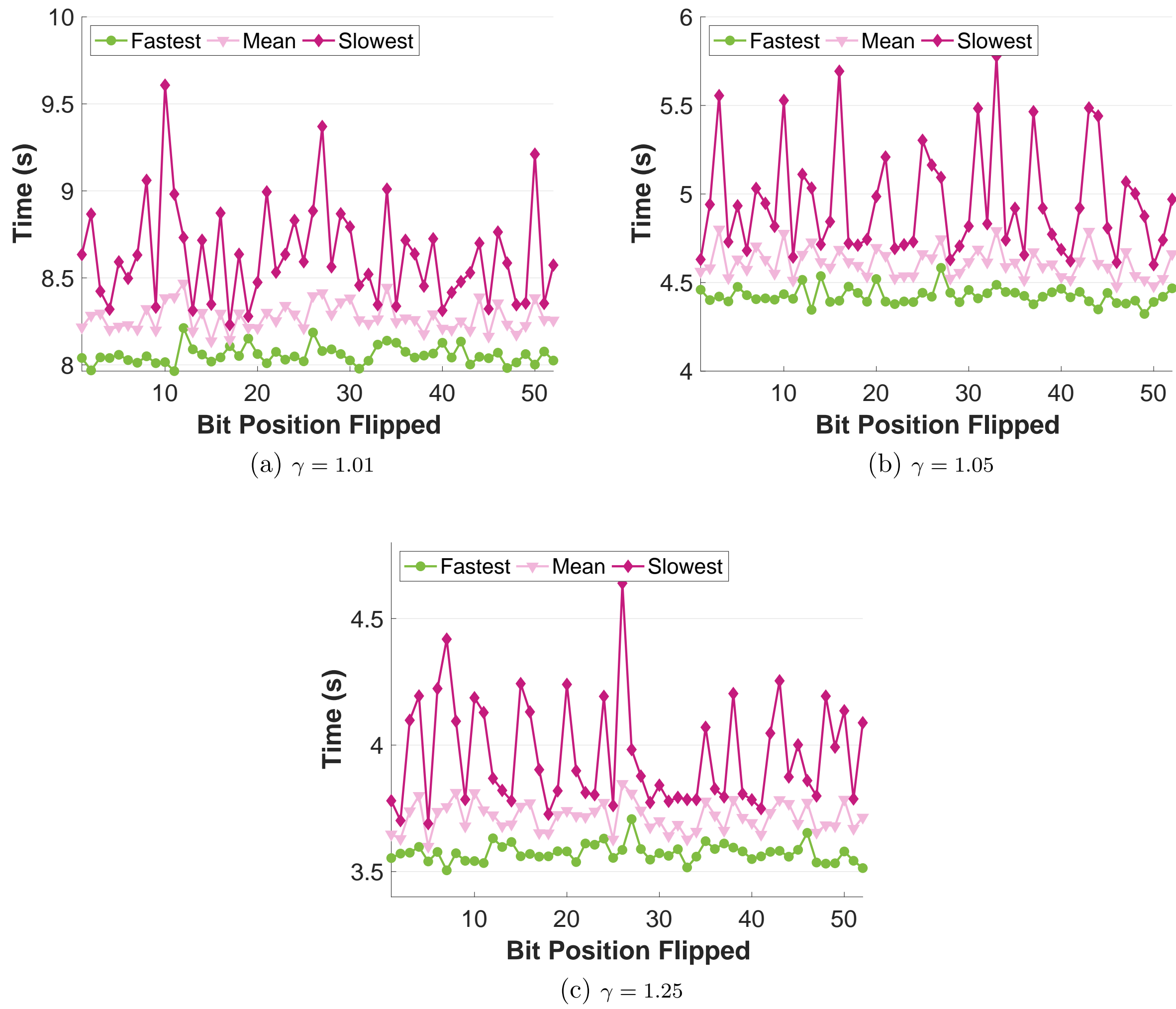
Note also that the recovery with SBSFM in Fig. 22 exhibits consistently increasing difference between the slowest and fastest runs with the increase in pertubation size. Fault recovery mechanisms in some cases are able to correct PBSFM and SBSFM faults, depending on the circumstances of the run, i.e. if the fault thread is able to detect the fault and roll back before adjacent threads copy bad values to their compute regions. Successful and

failing recovery outcomes are shown in Fig. 22, where the fastest runs indicate successful recovery and the slowest runs correspond to recovery failure. The implementation tested in this work corrected SBSFM faults at a higher rate than it did so for the PBSFM faults.



(a) $\gamma = 1.01$ (b) $\gamma = 1.05$ (c) $\gamma = 1.25$

Fig. 20: Effect of recovery with bit-flip faults in the exponent and sign bits.



(a) $\gamma = 1.01$

(b) $\gamma = 1.05$

(c) $\gamma = 1.25$

Fig. 21: Effect of recovery with bit-flip faults in the mantissa bits.

4.2.2 KRYLOV SUBSPACE METHODS

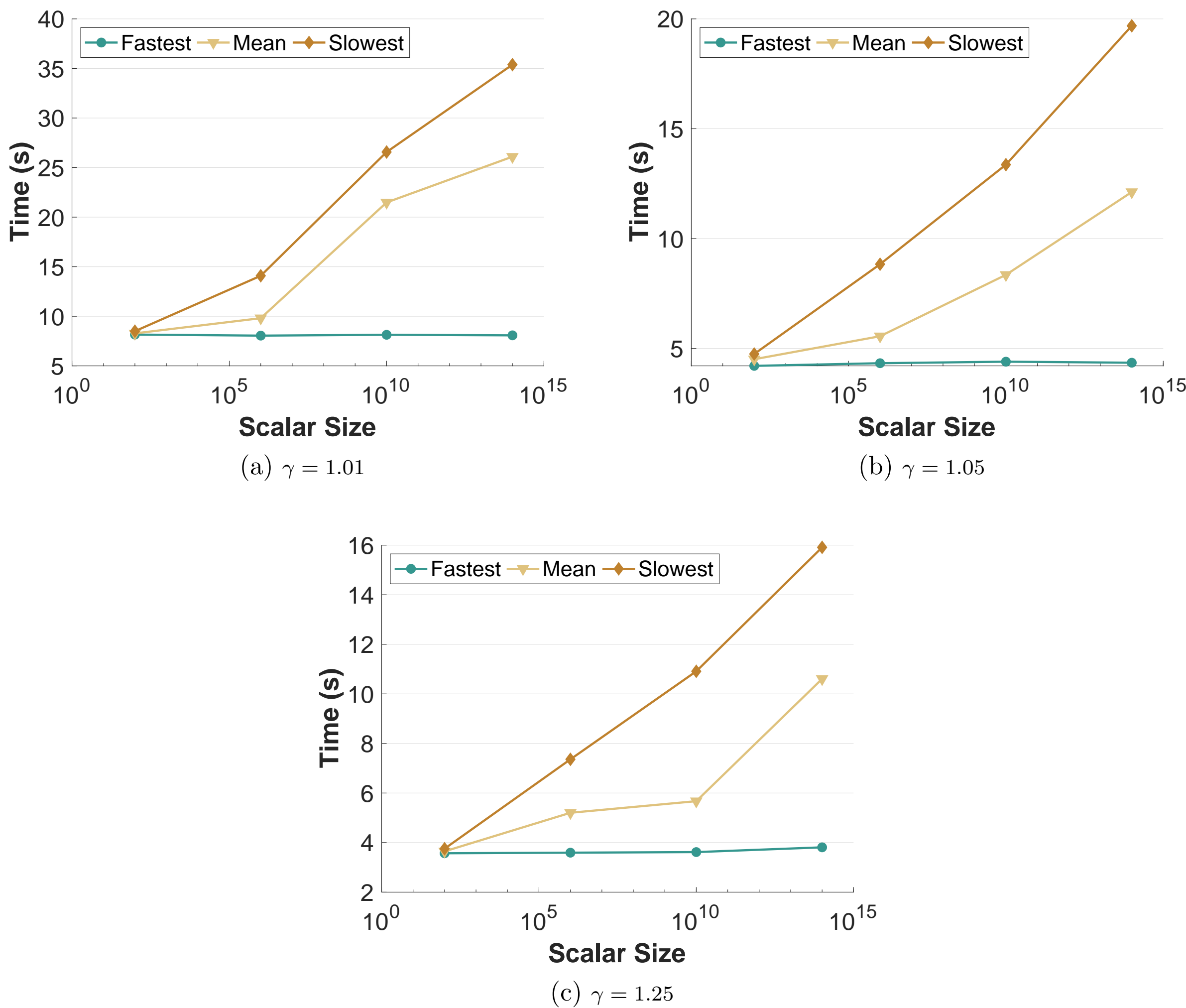
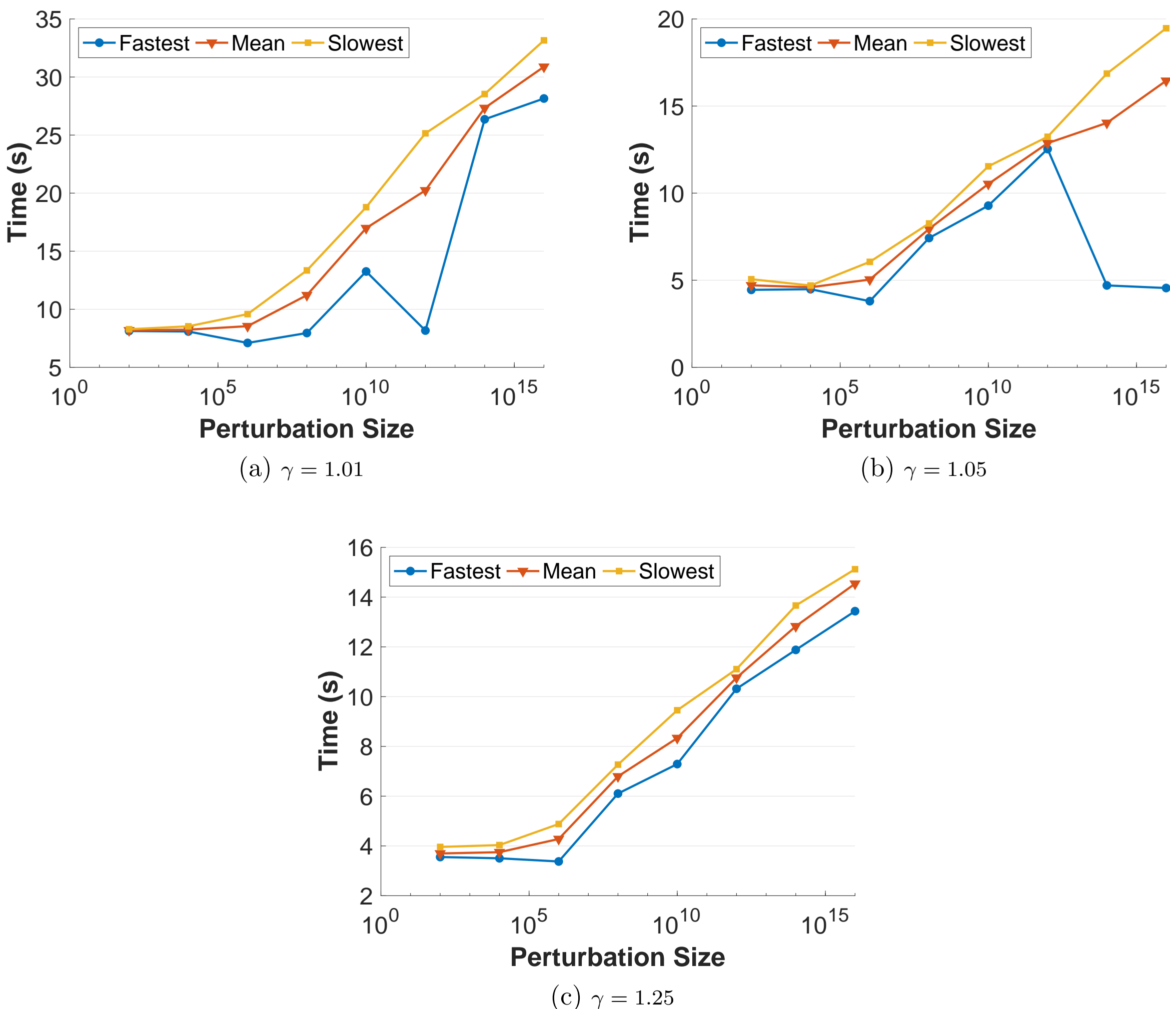


Fig. 22: Effect of fault recovery with the SBSFM.

4.2.2.1 Experiment Set-up

The test problem for both the series of experiments on sticky faults and the series of experiments concerning persistent faults comes from the pARMS library [50], and represents the discretization of the following elliptic 2D partial differential equation,

$$-\Delta u + 100\frac{\partial}{\partial x}(e^{xy}u) + 100\frac{\partial}{\partial y}(e^{-xy}u) - 10u = f \tag{102}$$



(a) $\gamma = 1.01$

(b) $\gamma = 1.05$

(c) $\gamma = 1.25$

Fig. 23: Effect of fault recovery with the PBSFM.

on a square region with Dirichlet boundary conditions, using a five-point centered finite-difference scheme on an $n_x \times n_y$ grid, excluding boundary points. The mesh is mapped to a virtual $p_x \times p_y$ grid of processors, such that a subrectangle of $r_x = n_x/p_x$ points in the x direction and $r_y = n_y/p_y$ points in the y direction is mapped to a processor.

The size of the problem was varied and controlled by changing the size of the mesh that was used in the creation of the domain. In the first series of experiments, related to sticky faults, the mesh sizes that were considered corresponded to a "small" problem with $n_x = n_y = 200$ and a "large" problem variant with $n_x = n_y = 400$. Both of these two

problem sizes were run on a $p_x = p_y = 20$ grid of 400 processors in total. This leads to problem sizes of $16,000,000$ and $64,000,000$, respectively ($n = p_x \times p_y \times n_x \times n_y$). For the results on sticky faults, only the SBSFM and PBSFM were considered since the emphasis was on creating a comparison between the generalized numerical methods for simulating the occurrence of a fault.

For the experiments related to the effect of persistent faults, the two problem sizes that were used correspond to $n_x = n_y = 200$ and a "large" problem variant with $n_x = n_y = 500$. Only the PBSFM was considered for these experiments since the algorithm was not able to converge for a meaningful number of parameter combinations for the other fault models. Examining the convergence of FGMRES with respect to the effects of the PBSFM gives some indication of how much data corruption the FGMRES algorithm is able to tolerate before diverging. Combining these observations with the comparison between the PBSFM, SBSFM, and BSFM provided in Section 4.1.5) can provide insight into the performance that would be achieved with the other numerical soft fault models.

Two set of fault injection locations were used. Both were selected due to the high computational cost associated with them, making the algorithm more likely to spend more time executing them and therefore more likely to be executing those instructions if a fault were to occur. The first location is the "outer matvec" operation in the FGMRES algorithm in Line 1 of Algorithm 10), and the second is the application of the preconditioner shown in Line 4 of Algorithm 3) in the FGMRES algorithm.

One point of focus is on comparing the resiliency of the two preconditioners introduced earlier, ILUT and ARMS, to evaluate their respective performance when faults are introduced. In all of the experiments that were conducted, multiple sets of runs were executed for each set of parameters (i.e. perturbation size, iteration number at which the fault was first injected) and their effect on the convergence of FGMRES was averaged across all other runs with the same parameters for analysis.

Algorithm 10: Flexible GMRES algorithm

Input: A linear system $Ax = b$ and an initial guess at the solution, x_0
Output: An approximate solution x_m for some $m \geq 0$

1 $r_0 := b - Ax_0$,
2 $\beta := ||r_0||_2, v_1 := r_0/\beta$
3 **for** $j = 1, 2, \ldots, m$ **do**
4   $z_j = M_j^{-1} v_j$
5   $w = Az_j$
6   **for** i = *1, 2, ...,* j **do**
7     $h_{i,j} := w \cdot v_i$
8     $w := w - h_{i,j} v_i$
9   $h_{j+1,j} := ||w||_2$
10   $v_{j+1} := w/h_{j+1,j}$
11   $Z_m := [z_1, \ldots, z_m]$
12   $\bar{H}_m := h_{i,j_{1 \leq i \leq j+1; 1 \leq j \leq m}}$
13 $y_m := argmin_y ||\bar{H}_m y - \beta e_1||_2$
14 $x_m := x_0 + Z_m y_m$
15 **if** *Convergence was reached* **then**
16   **return** x_m
17 **else**
18   go to line 1

4.2.2.2 Sticky Faults

The experiments that attempt to model the impact of sticky faults have been carried out on the computing platform Edison which was located at the National Energy Research Scientific Computing Center (NERSC). It was a Cray XC30 with 134,064 cores and 357 TB memory across a total of 5586 nodes. Each node had two sockets, with a 12-core Intel® "Ivy Bridge" processor at 2.4 GHz per socket. Edison is scheduled to be decommissioned in March of 2019. All the experiments in this subsection were conducted on a subset of 400 cores.

For the experiments related to persistent faults, the mesh sizes that were considered ranged from $n_x = n_y = 100$ to $n_x = n_y = 500$, and these mesh sizes were run on numbers of processors that varied from 4 ($p_x = p_y = 2$) to 100 ($p_x = p_y = 10$). For the experiments related to persistent faults, only the PBSFM was used. Use of the SBSFM in a recurring manner would have precluded convergence of the FGMRES algorithm and not provided a

meaningful point of comparison. The tests performed for both the large and small problem include a fault-free run, a series of runs using the SBSFM model and a series of runs using the PBSFM model. For the SBSFM, the variable that will have the largest impact upon the fault injected is the scaling factor α while for the PBSFM the largest contributor to the impact of the fault is the size of the perturbation ϵ. For these experiments, three values of both α and ϵ were used:

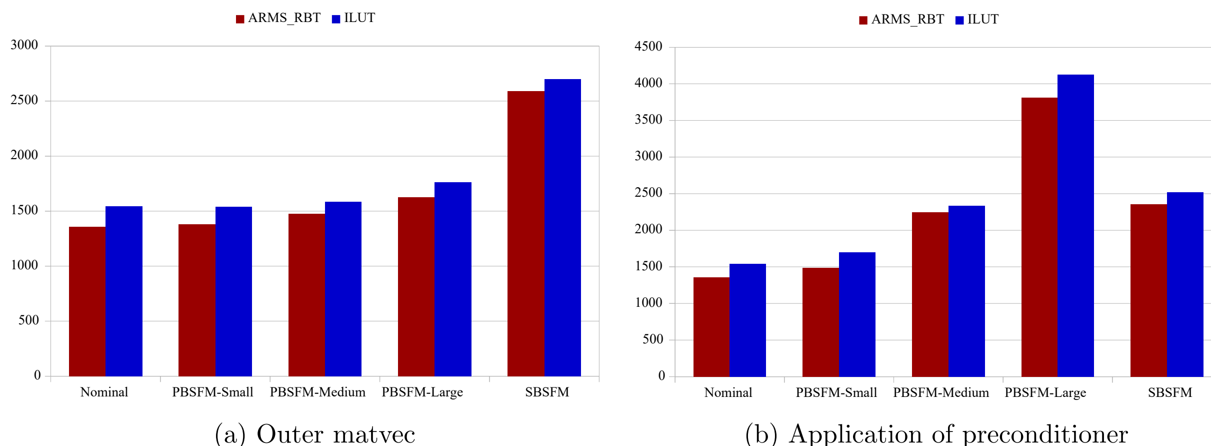
- $\alpha = 1/2, 1, 2$, and
- $\epsilon = 1e^{-3}, 5e^{-4}, 1e^{-4}$.

All three variants of the PBSFM were utilized. To compare with the runs of the SBSFM runs using $\alpha = 1/2$, $\alpha = 1$, and $\alpha = 2$, were compared to the three variants of PBSFM that decreases the norm, that leaves the norm approximately the same (referred to as "neutral" in the remainder), and that increases the norm, respectively (see Section 4.2.1.4). Sticky faults were conservatively defined to be present during the first 1000 iterations of the iterative solver execution. While the number of iterations required for convergence in a fault-free environment is a factor of many variables (problem size, preconditioner, error tolerance, inner iterations of FGMRES, etc), for the fault-free test, the small problem converged in roughly 1500 iterations, and the large problem in approximately 3500 iterations.

The plots are only presented for the neutral norm variants of the fault models in Figs. 24 and 25. To be specific, this involves the variants of the PBSFM where the norm remains approximately the same, and the version of the SBSFM where the scaling factor α is set to 1. Each figure shows five different fault methods: a nominal (fault-free) run, a PBSFM run with a "small" fault ($1e^{-4}$), a PBSFM run with a "medium" fault ($5e^{-4}$), a PBSFM with a "large" fault ($1e^{-3}$), and a SBSFM run with $\alpha = 1$. Complete results, including PBSFM variants that decrease or increase the norm, are provided in Table 4 for all the experiments.

The first plots that are shown in Fig. 24a depict the effects of the various soft faults injected into the outer matrix vector operation of the FGMRES algorithm when solving the

small problem. In this figure, it is apparent that for the neutral variants, for both the ARMS and ILUT preconditioners (see the background information in Section 2.3), the SBSFM has a more negative effect on the convergence of the FGMRES algorithm than the PBSFM does. For instance, compared to the fault-free runs, the SBSFM runs needed more than 1000 additional iterations to converge for both preconditioners while the additional number of iterations is at most around 150 for the different PBSFM variants. Fig. 24b shows the results when the faults are injected into the vector resulting from the preconditioner application.



(a) Outer matvec (b) Application of preconditioner

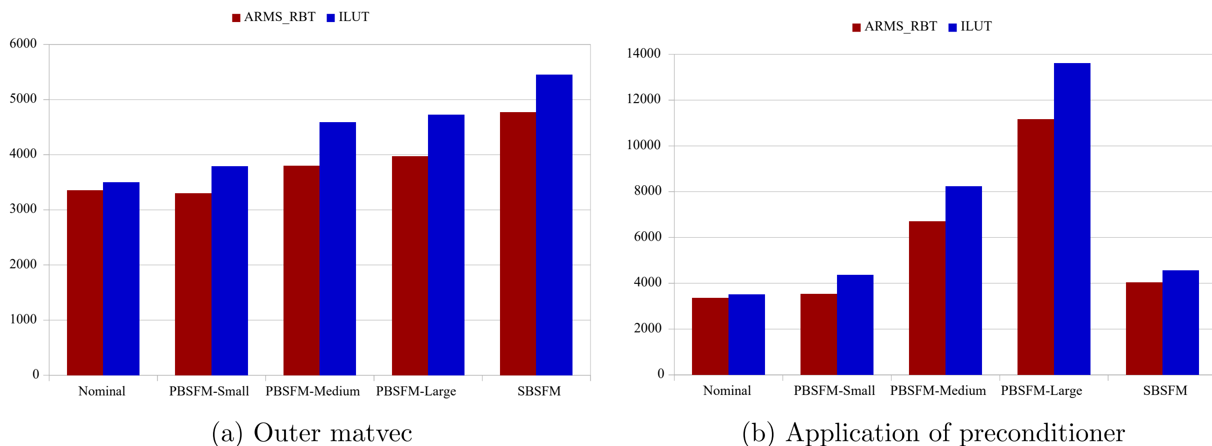
Fig. 24: Soft fault comparison on total number of iterations for the small problem for faults injected at the indicated operation using ARMS and ILUT preconditioners. Fault methods are displayed along the x-axis and total iterations required for convergence are represented by the y-axis.

Figure 25a displays the number of iterations to convergence when injecting faults into the outer matrix vector operation for the large problem. As in Fig. 24a, the results in Fig. 25a show a steady increase in the delay in the convergence of FGMRES from the nominal case to the PBSFM cases (ordered by the increasingly sized faults), then to the faults simulated by the SBSFM case.

The plots in Fig. 25b depict the injection of faults into the result of the preconditioning operation for the large problem. Note the one instance where the “large” fault size associated

Table 4: Full results for the small (`SP`) and large (`LP`) problems with the neutral, decrease, and increase norm variants in rows represented by signs =, −, and +, respectively. The † symbol indicates that the corresponding solver does not converge. We recall that in SBSFM, the cases =, −, and + correspond to $\alpha = 1$, $1/2$, and 2, respectively.

		$\|\| \|\|_2$	Nominal		PBSFM-S		PBSFM-M		PBSFM-L		SBSFM	
			SP	LP	SP	LP	SP	LP	SP	LP	SP	LP
ILUT	matvec	=	1542	3496	1380	3300	1477	3797	1624	3969	2590	4768
		−	1542	3496	2236	3807	2318	4170	2352	4380	2565	4660
		+	1542	3496	2241	3603	2326	4140	2358	4386	2637	4788
	precond	=	1542	3496	1487	3523	2243	6703	3811	11156	2355	4022
		−	1542	3496	1499	3280	2155	5163	2782	7639	2324	4093
		+	1542	3496	1499	3518	2168	5162	2780	7735	†	†
ARMS	matvec	=	1359	3357	1538	3790	1585	4594	1764	4727	2698	5456
		−	1359	3357	2323	4199	2426	4810	2459	7639	2697	5375
		+	1359	3357	2339	3825	2423	4655	2459	5059	2646	5426
	precond	=	1359	3357	1700	4349	2336	8221	4125	13607	2518	4550
		−	1359	3357	1706	4010	2201	6063	2925	9492	2570	4493
		+	1359	3357	1657	3989	2205	6061	2927	9005	†	†



(a) Outer matvec

(b) Application of preconditioner

Fig. 25: Soft fault comparison on total number of iterations for the large problem for faults injected at the indicated operation. Fault methods are displayed along the x-axis and total iterations required for convergence are represented by the y-axis.

with the PBSFM ($1e^{-3}$) causes a larger delay in the convergence of the FGMRES solver than the corresponding run of the SBSFM does. For this specific case, the same observation holds for all the three norm variants (cf. Table 4).

For a fault-free case, the FGMRES algorithm converged in fewer iterations when using the ARMS preconditioner compared to the use of the ILUT preconditioner. This remained true when faults were injected into the application of the preconditioner; however, the injection of faults into the outer matrix vector operation caused FGMRES to converge in roughly the same number of iterations whether it was preconditioned with ILUT or with ARMS. This suggests that for faults occurring at the outer matrix vector operation, the advantage of the ARMS preconditioner is not as present as it is elsewhere. Note that, for both ILUT and ARMS preconditioners, faults injected into the outer matrix vector operation had a larger impact than did identical faults injected into the resulting vector from the preconditioner application. Similar results were obtained in [149]. In addition, the impact of the faults injected using each of the two soft fault models with effects on the norm seems to be more pronounced in the PBSFM case; although, this is clearly adjustable through the use of the parameters available to both soft fault models. For instance, using larger values for α in the SBSFM may provide a better comparison.

When comparing the two fault models presented here directly, it is evident that the SBSFM has a larger negative impact on the convergence of the iterative FGMRES than the PBSFM in most scenarios. In every instance tested except for preconditioner faults on the larger problem size, the comparable version of the SBSFM delayed convergence longer than the PBSFM did. This is in part due to the fact that the SBSFM moves the vector where a fault is injected much further from its original location than the PBSFM does (see Table 2). In summary, for recurring faults specifically, the PBSFM offers a greater level of fine-tuned control over the fault impacts. However, the size of the fault in the PBSFM does not seem to have as large of an impact on the convergence of FGMRES in the runs that attempted to manipulate the norm.

4.2.2.3 Persistent Faults

The experiments related to the impact of persistent faults were executed on two different hardware platforms. The first test environment was a workstation with an Intel Core i7®processor having four physical cores at 2.50 GHz each and 16 GB of main memory. The second was the Hopper supercomputer, which was a compute resource of the National Energy Research Scientific Center (NERSC) that was decommissioned in December 2015. Hopper had a total of 153,216 compute cores, 212 Terabytes of memory and nodes are connected with a custom high-bandwidth, low-latency network provided by Cray. Up to five compute nodes of Hopper were utilized. The problem size was scaled appropriately for each environment, by adjusting the size of the square mesh per subdomain; namely, 200 and 500 points for the Intel Core i7®and Hopper, respectively. The parameters that were varied in these experiments are detailed in Table 5.

Table 5: Input parameters the value of which varied in the experiments.

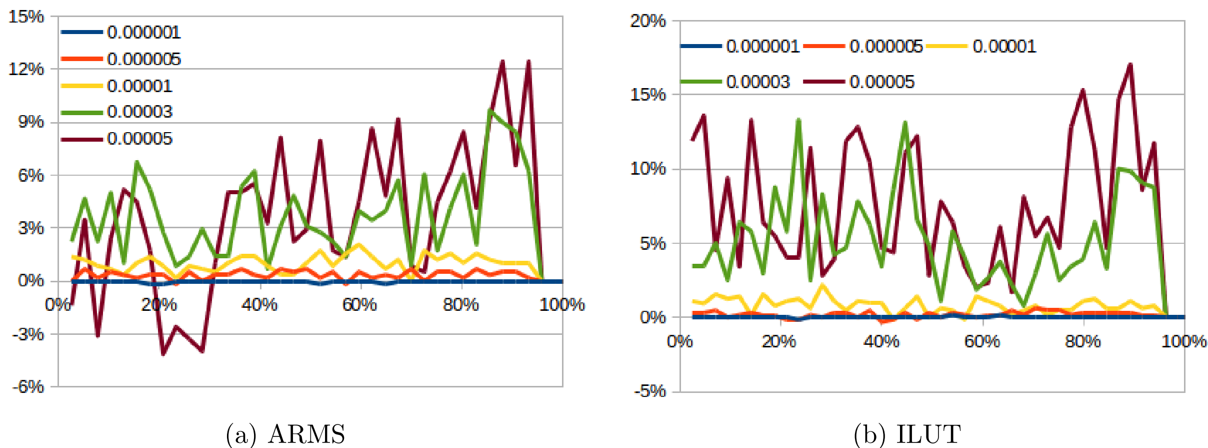
Parameter	Acceptable Values
Global Preconditioner	Block Jacobi
Local Preconditioner	ILUT, ARMS
Tolerance Required for Convergence	10^{-6}
Starting Iteration at which Fault Appears	≥ 5
Order of Perturbation	$10^{-6}, \ldots, 10^{-4}$
Effect on l^2-norm	Any, Decrease, Increase

Since the fault model presented here is based on a series of random perturbations, multiple runs/solves were conducted for each set of parameters, and the results were averaged and depicted in the plots of this section. In all of the experiments, a maximum number of iterations was instituted and, if a run did not converge within this preset number of iterations, then it was terminated, and determined to have failed.

The results shown in all the figures of Section 4.2.2.3 come from runs on the Intel Core

i7®platform using four MPI ranks, one per core. The results from the runs with larger problem sizes performed on Hopper showed similar convergence tendencies under perturbation-based faults considered here. Note that, in all the plots, the x-axis represents the fraction (as %) of the execution when a fault begins and the y-axis shows the increase (or decrease) in the number of iterations with respect to non-perturbed case. For example, a data point with x coordinate of 50% shows an effect from the fault injected halfway through the number of iterations that would be required by a fault-free run. This effect is quantified by the y coordinate of the point, such that, if $y = +100\%$, e.g., then the run corresponding to this data point required twice as many iterations to converge than that did in a fault free case.

The following results are provided the instance where the sign of the perturbation (and hence, the magnitude of the l^2-norm) was not controlled by the fault model.



(a) ARMS

(b) ILUT

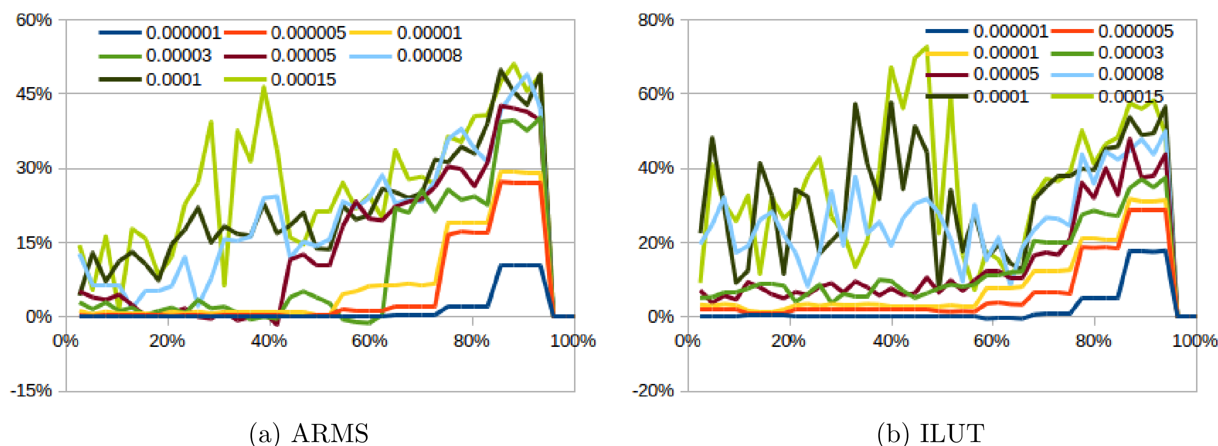
Fig. 26: Outer matvec perturbation faults with varied l^2-norm. On y- and x-axis, % of extra iterations and of fault-commencing iteration, respectively, compared to the number of iterations in the fault-free run.

Figure 26 shows the effects of faults with various perturbation sizes in outer matvec when the ARMS (Fig. 26a) or ILUT (Fig. 26b) local preconditioner is used. Only perturbation sizes no larger than 5×10^{-5} are shown since for larger values the solver failed to converge.

Comparing the results in Fig. 26a and Fig. 26b, a similar convergence behavior may be observed. However, the faults corresponding to smaller perturbations ($10^{-6}, \ldots, 5 \times 10^{-5}$) have a slightly greater negative effect on the runs with the ILUT preconditioner than on those with ARMS.

When examining effects of very small perturbations (on the order of convergence tolerance, which is 10^{-6} here), it was found that they had either no effect at all on the convergence rate or slightly decreased the total number of iterations. This beneficial effect was noted regardless of when the fault started during the run, and it appears more often with the ARMS preconditioner than with ILUT.

Next, results for the case where the sign of the perturbation was matched with the sign of the existing vector component in order to ensure that the l^2-norm of the perturbed operation result decreased (Fig. 27). In order to match the sign appropriately, the fault model checks the sign of the original vector component before applying the fault.



(a) ARMS

(b) ILUT

Fig. 27: Outer matvec perturbation faults with decreasing l^2-norm. On y- and x-axis, % of extra iterations and of fault-commencing iteration, respectively, compared to the number of iterations in the fault-free run.

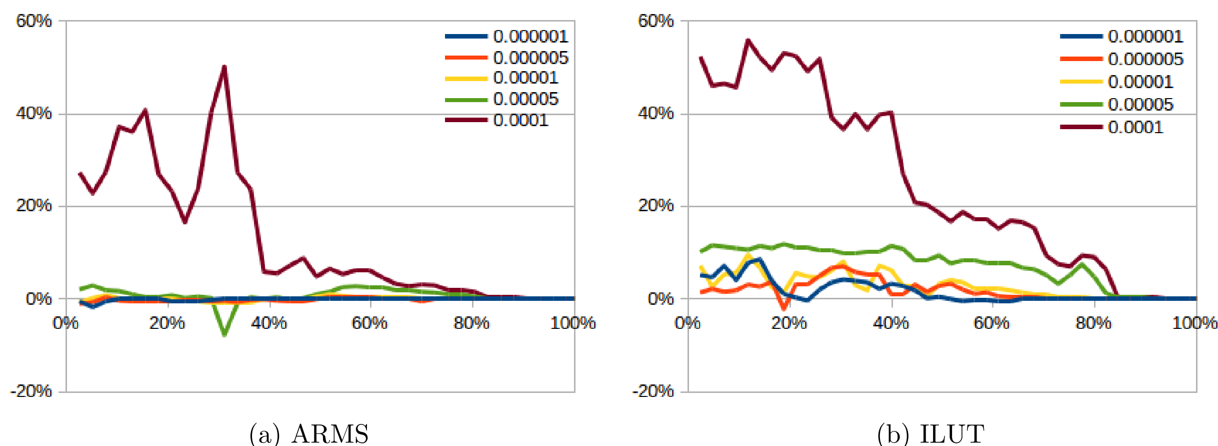
It is interesting to observe in Fig. 27 the increased rate of successful convergence for a

much larger range of fault magnitudes. A larger spectrum of perturbation sizes resulted in successful convergence and, hence, is represented in Fig. 27. Comparing the effects of injecting faults that vary the l^2-norm (Fig. 26) to those that shrink the l^2-norm (Fig. 27), there is also a decrease in the negative effect that a fault of the same magnitude has upon the FGMRES algorithm.

In general, the performance of the two preconditioners is fairly similar in the case when faults are incurred in the outer matvec operation. For instance, as expected, there is a tendency for the fault to have more of an impact on the convergence if the fault commences later in the execution; smaller perturbations show little effect while larger perturbations produce a much higher variations of convergence results. Perturbed executions resulting in fewer iterations than non-perturbed ones appear to arise with about equal frequency between scenarios using either the ARMS or the ILUT preconditioner. These results are seen for all the fault sizes—although much more commonly for faults of size $\leq 10^{-4}$—and are observed most when faults occur before the run reaches approximately 60% of completion of a fault-free run.

Here, the results of applying the perturbation-based soft fault model to the result of the preconditioner application in the FGMRES algorithm. Results (Fig. 28) are presented for each of the two preconditioners, ARMS and ILUT, and exclusively for the version of the perturbation-based soft fault model that decreases the l^2-norm of the vector that it is applied to.

Comparing the results with the ILUT preconditioner to those with ARMS, it again appears that the runs with the latter suffer less of a negative effect than those with the former for the faults of an equivalent size. Next, when examining results with the ARMS preconditioner (in Fig. 28a), it is clear that injecting a perturbation-based fault into the result of the application of the preconditioner (from Line 4 in Algorithm 3) has less of an effect on a FGMRES solve using the ARMS preconditioner than that from injecting a similar fault into the result of the outer matvec iteration (Fig. 27a).



(a) ARMS

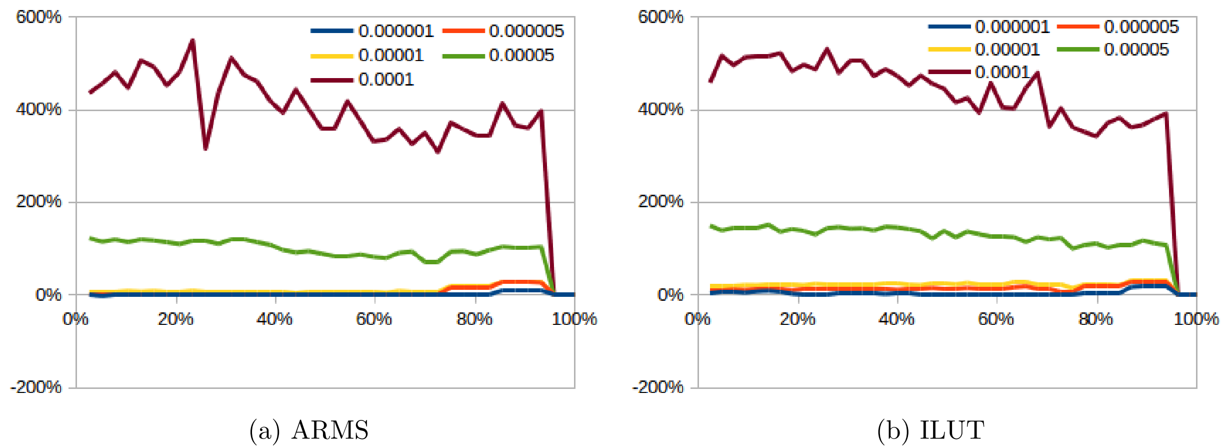
(b) ILUT

Fig. 28: Preconditioner perturbation faults with decreasing l^2-norm. On y- and x-axis, % of extra iterations and of fault-commencing iteration, respectively, compared to the number of iterations in the fault-free run.

Even a magnitude of fault (e.g., 10^{-4}) that may cause stagnation when injected into the outer matvec operation, causes only a 40–50% increase in the total number of iterations here and only has a large impact if injected throughout the majority of the run. Similar observations may be made for the ILUT preconditioning in Fig. 28b: less of a negative effect is evident when perturbation-based faults appear in this preconditioning operation than in the outer matvec (c.f. Fig. 27b). In general, FGMRES, being able to converge with a preconditioner that changes at each iteration, does not negatively react to preconditioner changes due to faults in the course of linear system solution.

The graphs in Fig. 29 show the effect of injecting a fault into the two fault sites considered simultaneously (i.e., at the same FGMRES iteration), the outer matvec *and* preconditioner application, such that the l^2-norm decreases.

In Fig. 29, notice that, for large faults (starting at 10^{-4}), the increase in the number of iterations required to converge was very high—between 400-600% at times. This increase is also much higher than that for either matvec- or preconditioner-only incurred faults producing the highest increases of ∼60% and ∼55%, respectively, for the same perturbation value of 10^{-4}.



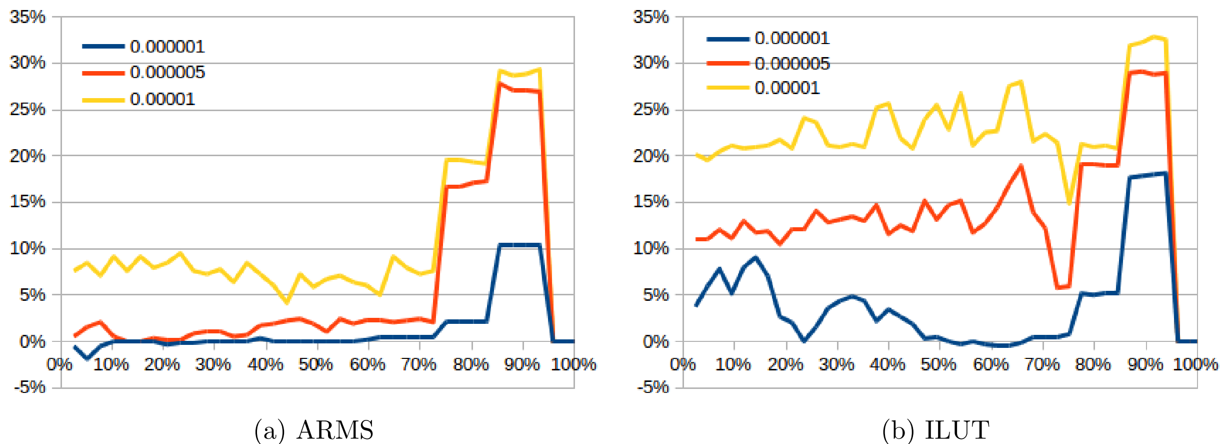
(a) ARMS (b) ILUT

Fig. 29: Outer matvec and preconditioner application faults with decreasing l^2-norm. On y- and x-axis, % of extra iterations and of fault-commencing iteration, respectively, compared to the number of iterations in the fault-free run.

Conversely, for perturbation sizes of 10^{-5} and smaller (shown in Fig. 30), the effect on convergence appears similar to that of matvec faults. This suggests that the ability of FGMRES to accept faulty preconditioners is inhibited by the coexistence of a matvec fault. Note also, there were fewer cases where the number of iterations to converge decreased due to faults in the scenarios with small faults.

All of the experiments were also performed with a variant of this perturbation-based fault model that increased the l^2-norm of the operation result. In all instances, smaller fault sizes caused FGMRES to fail to converge compared with the other l^2-norm variants of the fault model, and for the cases in which the iterative solver converged, many more iterations were required. Despite these differences, the overall trends are similar and the graphs featuring this data are not displayed here.

The series of runs using persistent fault simulation showcased experiments designed to exhibit a persistent fault model with faults affecting bounds within an iterative solver, which may be monitored and plays a role in the solvers reaction to faults. Specifically, similar to the conclusions in [131], effects on the l^2-norm of the fault-perturbed vector were explored and it was found that persistent faults may be treated similarly to episodic faults in quantifying



(a) ARMS

(b) ILUT

Fig. 30: Outer matvec and preconditioner application faults with decreasing l^2-norm. On y- and x-axis, % of extra iterations and of fault-commencing iteration, respectively, compared to the number of iterations in the fault-free run. Small fault sizes only.

Table 6: Summary of Beneficial Results- Note: Its (Iteration), PC (Preconditioner)

Fault Size	Fault Location	Starting Its	l^2-norm Effect	PC	Improvement
5×10^{-5}	matvec	0% - 30%	Varied	ARMS	2% - 4%
10^{-6}	matvec	(anywhere)	Varied	ARMS	0% - 1%
$10^{-6} - 10^{-5}$	matvec	(anywhere)	Varied	ILUT	0% - 1%
(any size)	matvec	0% - 60%	Decreasing	ARMS, ILUT	0% - 1%
$\leq 5 \times 10^{-5}$	PC	(anywhere)	Decreasing	ARMS	0% - 5%
$\leq 5 \times 10^{-6}$	PC	(anywhere)	Decreasing	ILUT	0% - 2%

their effects except that the application possibly needs to adjust to continuing operation "under failure". In particular, persistent faults that shrink the l^2-norm have less of a negative effect upon the convergence of the iterative solver. It was also found that injecting faults into the outer matvec operation, in general, had a greater impact upon the FGMRES convergence than doing so for the preconditioner application—including causing more cases in which the iterative solver failed—which was observed for both the ARMS and ILUT preconditioners. It appears that runs using the ARMS preconditioner are more naturally resilient to the injection of persistent perturbation-based faults than runs using the ILUT preconditioner, regardless of which of the two fault sites is chosen. In addition, a small fault injection

resulted in several runs that converged in up to 5% fewer iterations than would be typically required. Table 6 summarizes beneficial outcomes from the results presented in here.

4.3 SUMMARY

This chapter presented analysis of numerical soft fault models for the development of fault tolerant algorithms, including the development of a novel fault model that can be used for either synchronous or asynchronous iterative methods. Results were presented for asynchronous iterative methods; specifically for a hybrid parallel implementation of the asynchronous Jacobi algorithm, as well as for a Krylov subspace solver implemented in a typical distributed parallel computing environment using MPI. The results indicate that the use of numerical soft fault models can be useful for the development of fault tolerant algorithms for future High Performance Computing platforms since the average impact induced by the numerical soft fault models is large enough to cause detrimental effect to the execution of the iterative algorithm.

The testing conducted here was designed to be exhaustive with respect to the number of numerical soft fault models, including representing a variety of different methods for simulating the occurrence of a fault in addition to presenting results for a variety of parameters that control the fault models. The elliptic partial differential equations (e.g., the two dimensional discretization of the Laplacian for the asynchronous iterative methods, and the test problem given by Eq. (102) for the Krylov subspace solver experiments) that were used serve as common test problems, due to their close connection with many important problems throughout science and engineering. The numerical simulation of soft faults provides a consistent, reliable way to force sufficient data corruption to examine the behavior of iterative algorithms. This makes numerical soft fault models a valuable means of developing novel fault tolerant algorithms; an activity that will become increasingly important as HPC environments edge closer to exascale levels of performance. The comparison and analysis of multiple numerical soft fault models provided in this dissertation offers shows the potential benefit offered by

this line of research.

CHAPTER 5

USE CASE: FINE-GRAINED INCOMPLETE FACTORIZATIONS

The fine-grained parallel incomplete LU factorization (FGPILU) algorithm is a nonlinear fixed point iteration that can be used for finding an approximate factorization of an input matrix A, such that

$$A \approx LU \tag{103}$$

in the case that A is non-symmetric, referred to as incomplete LU factorization, or such that,

$$A \approx LL^T \tag{104}$$

in the case that A is symmetric, referred to as incomplete Cholesky factorization. This factorization is suitable for use as a preconditioner in a linear solver routine, or when rough approximations to the solution of a linear system are acceptable. In practice, these incomplete factorizations are commonly used in conjunction with a Krylov subspace solver such as Conjugate Gradient of FGMRES [34], [147].

The FGPILU algorithm can be used as a building block for iterative linear-system solvers geared towards novel computing platforms, including accelerators and co-processors. Typically when working with difficult problems, preconditioning techniques move beyond simple incomplete LU factorizations, such as those generated by the FGPILU algorithm, to more complex routines. These include routines such as ILUT and ARMS (see Section 2.3). A more complex variant of fine-grained factorization that attempts to improve upon the performance of the FGPILU algorithm studied here has been recently proposed [178]. Alternatively, work on using conventional incomplete LU factorizations for solving difficult problems from var-

ious disciplines has been conducted previously, including the more general studies found in [51] and [52].

This chapter examines the FGPILU algorithm proposed by Chow and Patel [24] that can be used to generate incomplete factorizations in a highly parallel fashion. Several variants of the FGPILU algorithm are presented, each capable of converging despite the occurrence of soft computing faults. A discussion of the mathematical theory behind the convergence of such techniques is presented, extensive numerical results concerning the performance of these fine-grained incomplete factorizations with respect to faults are provided. In the numerical experiments, the potential impact of soft faults on the fine-grained parallel incomplete LU factorization is studied from several different perspectives. Specifically, the ability of the algorithm to converge successfully despite the occurrence of a fault is evaluated, as well as the performance of the incomplete factor(s) that are generated when they are used as preconditioners for Krylov subspace solvers. This theory and the variants of the algorithm that are developed build on the more general theory presented in Chapter 3. Each of the techniques proposed for recovery in Chapter 3 is adapted to the FGPILU algorithm and included in the numerical experiments.

Another aspect of the work presented in this chapter is that the convergence of the FGPILU algorithm is analyzed, building upon the initial convergence analysis presented in [24], and this convergence is then explored numerically with several test problems from varying domains in science and engineering. These test problems include a set of problems that are relatively well behaved (i.e. SPD) as well as a set of problems that are more difficult for the algorithm to solve; i.e., non-symmetric, indefinite and ill-conditioned problems. The majority of the work on the FGPILU algorithm so far has focused on matrices that are SPD [23], [24], and the performance of the algorithm on non-symmetric and indefinite matrices has not been firmly established. Moreover, if the convergence of the algorithm for these classes of problems is less than desirable, they may be more prone to suffer divergence when faced with a fault. These more difficult problems are also included in the numerical

experiments concerning the impact that a fault may have. Some results provided here have been previously published [169], [170], [175], [176].

The structure of this chapter is organized as follows: in Section 5.1, an overview of the fine-grained parallel incomplete factorization algorithm itself is given. In Section 5.2, a theoretical underpinning of the fine-grained parallel incomplete LU algorithm with respect to its convergence is explored. Section 5.3 provides an overview of the variants of the FGPILU algorithm that have been proposed for their resilience to soft faults, in Section 5.4, a series of numerical results are provided, while Section 5.5 provides a summary of the work discussed in this chapter.

5.1 FINE-GRAINED PARALLEL ALGORITHM

The FGPILU algorithm approximates the true LU factorization and writes a matrix A as the product of two factors L and U where,

$$A \approx LU. \tag{105}$$

Normally, the individual components of both L and U are computed in a manner that does not allow easy use of parallelization (see Section 2.3.1 for more details). The recent FGPILU algorithm proposed in [24] allows each element of both the L and U factors to be computed independently. Because of this, the level of parallelism in the algorithm scales as the number of non-zero terms in the factorization increases.

The algorithm progresses towards the incomplete LU factors that would be found by a traditional algorithm in an iterative manner. To do this, the FGPILU algorithm uses the property

$$(LU)_{ij} = a_{ij} \tag{106}$$

for all (i, j) in the sparsity pattern S of the matrix A, where $(LU)_{ij}$ represents the (i, j) entry of the product of the current iterate of the factors L and U. This leads to the observation

that the FGPILU algorithm (given in Algorithm 11) is defined by the following two nonlinear equations:

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) \tag{107}$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}. \tag{108}$$

Following the analysis presented in [24], it is possible to collect all of the unknowns l_{ij} and u_{ij} into a single vector x, then express these equations as a fixed point iteration,

$$x^{(p+1)} = G\left(x^{(p)}\right), \tag{109}$$

where the function G implements the two nonlinear equations described above and the current iteration ($(p+1)$ or (p) respectively) is given by the superscript. The FGPILU algorithm is given in Algorithm 11.

Algorithm 11: FGPILU algorithm as given in [24]

Input: Initial guesses for $l_{ij} \in L$ and $u_{ij} \in U$
Output: Factors L and U such that $A \approx LU$

```
1 for sweep = 1, 2, ..., m do
2     for (i, j) ∈ S do in parallel
3         if i > j then
4             l_ij = (a_ij − Σ_{k=1}^{j−1} l_ik u_kj)/u_jj
5         else
6             u_ij = a_ij − Σ_{k=1}^{i−1} l_ik u_kj
```

Keeping with the terminology used in [23], [24] each pass the algorithm makes in updating all of the l_{ij} and u_{ij} elements (alternatively: each element of the vector x) is referred to as a "sweep". After each sweep of the algorithm, the L and U factors progress towards

convergence.

At the beginning of the algorithm, the factors L and U are set with an initial guess. In this dissertation, the initial L factor will be taken to be the lower triangular part of A and the initial U will be taken to be the upper triangular portion of A (as in [24], [65], [169], [170]). Adopting a technique used in [23], [24], [169], [170], a scaling of the input matrix A is first performed such that the diagonal elements of A are equal to one. As pointed out in [24], this diagonal scaling is imperative to maintain reasonable convergence rates for the algorithm, and the working assumption throughout this chapter is that all matrices have been scaled appropriately.

5.2 CONVERGENCE OF THE ALGORITHM

This section serves to provide a discussion of the convergence of the FGPILU algorithm. The work here to examine the properties related to the convergence of the algorithm (i.e. the rate at which it converges, from which initial conditions, what can cause divergence, etc) serves to provide a foundation for the algorithmic variants that are proposed in Section 5.3 which attempt to provide soft fault resilience for the FGPILU algorithm.

The analysis to show convergence of the FGPILU algorithm relies on properties of the Jacobian associated with the nonlinear mapping that defines the FGPILU factorization (Eq. (107) and Eq. (108)) which when collected together as suggested by Eq. (109) define a map

$$G : \mathbb{R}^m \to \mathbb{R}^m \tag{110}$$

where m represents the number of non-zero terms in the matrix A. In order to discuss the properties of this function and it's Jacobian, it is necessary to define an order on the elements that make up the vector x upon which G operates. Every element in x is one of the non-zero elements in either the matrix L or the matrix U; with the initial guess taken as defined in Section 5.1 this corresponds to non-zero elements in the original input matrix,

A. The following definition formalizes the concept of an ordering.

Definition 7. An *ordering* of the elements $m_{ij} \in M$ is a bijective function from the sparsity pattern S of M to the set $1, 2, \ldots, N$. Formally, this is a map $T : S \to 1, 2, \ldots, N$.

Less formally, every non-zero element that will be updated needs to be given an order to make the algorithm well defined. In the case of this specific algorithm, it is of interest to have an ordering that arranges the elements in the order they would be updated following a traditional Gaussian Elimination style process; similar to what would be used in a conventional incomplete LU factorization. This style of ordering can be described as follows:

1. The first row of M
2. The remainder of the first column of M
3. The remainder of the second row of M
4. The remainder of the second column of M
5. $\cdots$

The following definition captures this more precisely:

Definition 8. A *Gaussian Elimination partial ordering* of the elements $m_{ij} \in M$ is a partial ordering of the elements in the sparsity pattern, S, of M (using MATLAB®[1] style notation):

$$(1, 1 : n) \cap S < (2 : n, 1) \cap S < \cdots < (k + 1 : n, k) \cap S < (n, n)$$

As stated above, in order to define the Jacobian of the nonlinear map G that defines the FGPILU factorization, an order of the elements in both the L and U factors which together constitute all of the elements in a single vector $\vec{x}$ (e.g., as discussed in Section 2.4) needs to be defined. Call this ordering h. The ordering h will map a given pair of (i, j) coordinates

[1] MathWorks, Inc., Natick MA

specifying the location of a non-zero term in either L or U to an index of the vector x. The indices of the vector x will be the set $\{1, 2, 3, \ldots, m\}$ where $m = nnz(L) + nnz(U)$. That is,

$$x_{h(i,j)} = \begin{cases} l_{ij} & i > j \\ u_{ij} & i \leq j \; . \end{cases} \tag{111}$$

Given this, the two nonlinear equations that define the FGPILU factorization, i.e., Eq. (107) and Eq. (108), can be rewritten to account for this ordering. Doing this gives,

$$G_{h(i,j)} = \begin{cases} \frac{1}{x_{h(j,j)}} \left(a_{ij} - \sum\limits_{1 \leq k \leq j-1} x_{h(i,k)} x_{h(k,j)} \right) & i > j \\ a_{ij} - \sum\limits_{1 \leq k \leq i-1} x_{h(i,k)} x_{h(k,j)} & i \leq j \; , \end{cases} \tag{112}$$

where both sums are taken over all pairs, (i, k) and $(k, j) \in S(A)$.

The Jacobian itself can then be written as a function, $G'(x) = J(G(x))$, where

$$J : \mathbb{R}^{|S|} \rightarrow \mathbb{R}^{|S| \times |S|} \tag{113}$$

and is defined by the partial derivatives of the map given by Eq. (112). These partial derivatives are given by the following equations [24]:

$$\begin{aligned} \frac{\partial G_{h(i,j)}}{\partial x_{h(k,j)}} &= -\frac{x_{h(i,k)}}{x_{h(j,j)}}, k < j \\ \frac{\partial G_{h(i,j)}}{\partial x_{h(i,k)}} &= -\frac{x_{h(k,j)}}{x_{h(j,j)}}, k < j \\ \frac{\partial G_{h(i,j)}}{\partial x_{h(j,j)}} &= -\frac{1}{x_{h(j,j)}^2} \left(a_{ij} - \sum_{k=1}^{j-1} x_{h(i,k)} x_{h(k,j)} \right) \end{aligned} \tag{114}$$

for a row in the Jacobian where $i > j$ (i.e., corresponding to an unknown $l_{ij} \in L$). Conversely, for a row $i \leq j$ (i.e., corresponding to an unknown $u_{ij} \in U$), the partial derivatives are given by:

$$\frac{\partial G_{h(i,j)}}{\partial x_{h(i,k)}} = -x_{h(i,k)}, k < i, \tag{115}$$

$$\frac{\partial G_{h(i,j)}}{\partial x_{h(k,j)}} = -x_{h(i,k)}, k < i. \tag{116}$$

Under the assumption that there is a single fixed point solution x^* of the nonlinear iteration defined by $G(x)$ in Eq. (112), the following result given in Theorem 10 provides convergence for the nominal, fault-free version of the FGPILU algorithm:

Theorem 10 ([58]). *Assume that x^* lies in the interior of the domain of G and that G is F-differentiable at x^*. If $\rho(G'(x^*)) < 1$, then there exists some local neighborhood of x^* such that the asynchronous iteration defined by G converges to x^* given that the initial guess is inside of this neighborhood.*

The partial derivatives are continuous and well-defined anywhere on the domain of G as defined above so G is F-differentiable on its domain. What remains to be shown is that the spectral radius $\rho(G'(x^*)) < 1$. The Gaussian Elimination partial ordering proposed in Definition 8 leads to the following result from [24] that details the structure of mapping, G, defined by Eq. (112):

Theorem 11 (Chow and Patel). *The function $G(x)$ with a Gaussian Elimination partial ordering has a strictly lower triangular form. Formally,*

$$G_k(x) = G_k(x_1, \ldots, x_{k-1}).$$

This leads to the following related result that also comes from Chow and Patel in [24]:

Theorem 12. *Given a Gaussian Elimination partial ordering for the mapping $G(x)$, the associated Jacobian, $J(G(x))$, has a strictly lower triangular structure. In particular, Jacobian has zeros along the diagonal and a spectral radius of 0.*

This result can be combined with results from Theorem 10 to show that there is some neighborhood of the fixed point of the mapping where the FGPILU algorithm will converge. Extended details of this analysis are provided in [24].

However, in order to determine if the mapping will converge from its current location in the domain of the mapping G defined by Eq. (112) it is necessary to define what it means for a mapping to be a contraction:

Definition 9. The function $G : D \subseteq \mathbb{R}^m \to \mathbb{R}^n$ is a *contraction* on D if there exists a constant $\alpha < 1$ such that,

$$||G(x) - G(y)|| \leq \alpha ||x - y||,$$

for some $x, y \in D$.

Note that an iterate of the function G, written $x \in D$, is a collection of all the non-zero values in both L and U. The form of the Jacobian is determined by the ordering of the elements inside of x, but the norm of the Jacobian (for any matrix norm) is associated with the value of the elements in the current iterate, x. In particular, the spectral radius of the Jacobian is determined by the (partial) ordering imposed upon the mapping G, but the norm of the Jacobian changes as the FGPILU algorithm progresses. The following helps identify when the fixed point iteration associated with the FGPILU algorithm is a contraction:

Definition 10. The function $G : D \subseteq \mathbb{R}^m \to \mathbb{R}^n$ is a contraction at the location of the current iterate $x^* \in D$ if $||J(G(x^*))|| < 1$ for some matrix norm $|| \cdot ||$ and the domain $D \subseteq \mathbb{R}^m$ is convex.

For the mapping G defined by Eq. (112), the domain is not necessarily convex [24], but the norm of the associated Jacobian is still indicative of whether or not the corresponding fixed point iteration will converge [24].

With respect to the occurrence of a fault, the fault model proposed in this dissertation limits the effects of a fault to the *values* stored in L and U and *not* the coordinates of the values. As such, it is not possible for a fault (as defined here) to change the spectral radius of

the mapping associated with the FGPILU algorithm; however, a fault can (and often does) change the norm of the corresponding Jacobian since it changes the values of the entries $x_i \in x$.

This leads to the following sequence of computational steps to identify if the mapping G is still a contraction:

1. Define a Gaussian Elimination partial ordering of the elements in L and U
2. Form the Jacobian, J, according to the partial derivatives defined in Section 5.1
3. Calculate the norm of J as found in step 2

To be clear, if the norm of the Jacobian is less than 1 and the current iterate is located in a convex portion of the domain then the mapping is still a contraction and it will eventually converge; however, if the norm of the Jacobian is greater than or equal to 1 then the mapping is not a contraction and further iteration will not bring the current iterate, x^*, closer to the fixed point.

One consequence of Theorem 10 is that the algorithm will be successful when the norm of the Jacobian is small. Examining the equations that define the partial derivatives inside of the Jacobian, this implies that the FGPILU algorithm will be effective when the terms on the diagonal are large and the off diagonal terms are small; indicating that the FGPILU algorithm will perform well for matrices that are diagonally dominant.

In previous work on the FGPILU algorithm, much of the emphasis has been placed on symmetric, positive definite (SPD) matrices that are symmetrically scaled to have unit diagonal [23], [169]. One notable exception is the 2D convection-diffusion problem that is presented in [24]. The problem,

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + \beta\left(\frac{\partial e^{xy} u}{\partial x} + \frac{\partial e^{-xy} u}{\partial y}\right) = g \tag{117}$$

is examined for two different values of β; the resultant finite-difference matrix being increasingly non-diagonally dominant and non-symmetric for larger values of β. In [24], the

authors recommend using a minimum degree ordering (as opposed to the Reverse Cuthill-Mckee (RCM) ordering used in the rest of the work) and find success producing stable preconditioning factors using the `SYMAMD` (e.g. symmetric approximate minimum degree permutation) ordering implemented in MATLAB®[2]. Because of this, in the testing on non-symmetric problems that is presented here (previously captured in [175]), the same `SYMAMD` reordering is included in the experiments.

5.2.1 IMPROVING THE CONVERGENCE OF THE FGPILU ALGORITHM

Here, an investigation is made into the performance of the FGPILU algorithm, and various attempts are made at improving both the rate of convergence and the effect of the generated FGPILU preconditioning factors. Generally speaking, the FGPILU algorithm works well on symmetric, positive-definite problems and the techniques detailed in this section are designed to be used with more difficult problems; i.e., problems that are non-symmetric, indefinite, or poorly conditioned. As such, in Section 5.4 these techniques are only applied to the more difficult problems featured in Section 5.4.4.

For a given problem, the FGPILU algorithm may fail to converge; i.e., a desired residual fails to decrease below a given threshold or else the iterates of the factorization diverge entirely. Additionally, the structure of the input matrix may preclude unmodified use of the FGPILU algorithm; e.g., due to zeros on the diagonal. If the progression of the algorithm reaches a point where the norm of the Jacobian is greater than one, the fixed point iteration no longer represents a (local) contraction and further sweeps will not help the algorithm make progress towards the desired preconditioning factors.

Even if the FGPILU algorithm converges to a set of preconditioning factors, it is possible that, if the system was changed too much – either intentionally in order to ensure convergence or by the occurrence of an undetected computing fault – the preconditioning factors will not aid in the convergence of the associated Krylov subspace solver. In fact, it is possible for the

[2]MathWorks, Inc., Natick MA

resulting L and U factors to actually slow convergence or prevent convergence entirely (see both Table 13 and [179]).

In an effort to improve the convergence of the FGPILU algorithm, this study focuses on employing two techniques that have been previously associated with either the preparation of more conventional incomplete LU factorizations or with the solution of a linear system using a Krylov subspace solver. Both of these techniques aim to increase the diagonal dominance of the original matrix, which should in turn reduce the norm of the Jacobian and help ensure that the fixed point iteration continues to make progress. Note that while these techniques may improve the convergence of the algorithm, care must be taken to ensure that they truly improve the overall time to solution.

The first technique involves reordering the matrix in order to aid the convergence of the algorithm. Three reorderings are considered here. The first is the `MC64` reordering that attempts to permute the largest entries of the matrix to the diagonal [180]. The `MC64` algorithm has been successful at improving the performance of algorithms requiring diagonal dominance, but is one of the more expensive reordering algorithms computationally. The second is the approximate minimum degree (`AMD`) as implemented in `MATLAB`®[3]. As stated before, this reordering has previously been observed to help convergence of the FGPILU algorithm on non-symmetric problems [24] and has also seen success with conventional incomplete LU factorizations for non-symmetric and indefinite problems [52]. The third and final ordering algorithm to be considered is the Reverse Cuthill-Mckee (`RCM`), which attempts to reduce the bandwidth of the matrix. This can potentially aid in the convergence of the FGPILU algorithm and has shown to be effective in the case of symmetric, positive-definite (SPD) matrices [23], [24], [169], [170]. It is important to note that some of these reorderings may not have a positive effect on other algorithms. For example, [181] shows that the RCM algorithm does not work as well as several other reorderings when applied to problems using sparse approximate inverses.

[3]MathWorks, Inc., Natick MA

After the ordering is applied, the second technique consists of an α-shift that is performed in the manner originally suggested in [179]. Specifically, the original input matrix A can be written,

$$A = D - B, \tag{118}$$

where D holds only the diagonal elements of A, and B contains all other elements. Instead of performing the incomplete LU factorization on the original matrix A, the factorization is instead applied to a matrix that is close to A but has an increased level of diagonal dominance. In particular, the incomplete LU factorization can be applied to

$$\hat{A} = (1 + \alpha)D - B, \tag{119}$$

where $\hat{A} \approx A$ but the size of the diagonal has been increased. This α-shift technique has been used historically for improving the stability of the preconditioning factors generated by conventional incomplete LU factorizations, but given the discussion above in Section 5.2 concerning the fine-grained incomplete LU factorization that is the subject of this work, it is reasonable to expect this shift to improve the convergence of the FGPILU algorithm. Note that it is possible for the incomplete factorization to be applied to a matrix that has been shifted too far from the original matrix where even if the FGPILU algorithm converges successfully, the associated Krylov subspace solver may not be able to make use of the generated preconditioning factors. A brief summary of this algorithm is presented in Algorithm 12.

Since incomplete LU factorizations are by nature, approximate, using the preconditioning factors obtained from applying the FGPILU algorithm to $\hat{A}$ before a Krylov solve of the original matrix A can be expected to accelerate the overall convergence for reasonable values of α. These claims will be explored numerically in Section 5.4.4.

Algorithm 12: Modified FGPILU algorithm for non-symmetric and indefinite matrices

Input: Input matrix A, shift factor α
Output: Factors L and U such that $A \approx LU$

```
1  Perform matrix reordering.
2  Factor the reordered matrix: A = D − B
3  Perform diagonal scaling: A = (1 + α)D − B
4  Generate initial guesses for L and U from the reordered and scaled input matrix A
5  for sweep = 1, 2, . . . , m do
6      for (i, j) ∈ S do in parallel
7          if i > j then
8              l_ij = (a_ij − Σ_{k=1}^{j−1} l_ik u_kj)/u_jj
9          else
10             u_ij = a_ij − Σ_{k=1}^{i−1} l_ik u_kj
```

5.3 SOFT FAULT RESILIENCE

In this section, several variants of the FGPILU factorization are proposed in an effort to provide soft fault resilience to the algorithm. First, general comments concerning the convergence of the algorithm with respect to soft faults and generalized and idealized notions about how to create fault tolerant variants are discussed, and then specific variants of the algorithm are proposed in the following subsections. The efficacy of these algorithms is tested numerically in Section 5.4.

The idea of creating fault tolerant algorithms has taken a renewed place of prominence in the research community due to the expected increase in the rate that faults will occur for future HPC platforms [1], [2], [4]–[6]. In this study, the focus is on creating so called *self-stabilizing* variants of the algorithm.

Self-stabilizing iterative methods stem from the idea of creating an algorithm that is capable of starting from any state and returning to a valid state within a finite number of steps. This can be viewed to encompass both traditional approaches towards resilience such as checkpointing, as well as different algorithmically based variants. It is also important to design self-stabilizing algorithms such that the computational cost of ensuring resilience is

minimal, especially in the case that no faults happen to occur.

In [125], a self-stabilizing variant of the Conjugate Gradient solver was proposed that made use of a periodic correction step to ensure that the algorithm returned to a valid state and proceed to convergence successfully. The work performed here proposes variants that take advantage of both checkpointing and the use of a periodic correction step. A notional, prototypical variant of the FGPILU algorithm that utilizes a periodic correction step is given by Algorithm 13.

Algorithm 13: Prototype algorithm for a Self-stabilizing FGPILU

Input: Initial guesses for $l_{ij} \in L$ and $u_{ij} \in U$
Output: Factors L and U such that $A \approx LU$
1 **for** $sweep = 1, 2, \ldots, m$ **do**
2   **if** $sweep \equiv 0 \mod F$ **then**
3   (Perform self-stabilizing computation)
4   **else**
5   **for** $(i, j) \in S$ **do in parallel**
6   **if** $i > j$ **then** $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj})/u_{jj}$
7   **else** $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$

In the prototypical self-stabilizing algorithm provided by Algorithm 13, every F^{th} iteration a yet-to-be-defined series of calculations is executed in order to ensure that the algorithm continues to progress towards convergence. The goal of this periodic correction step is that the computation done every F iterations in the periodic correction step will sufficiently correct the course of the algorithm to where it will converge. These calculations also need to ensure that they do not harm the convergence of the algorithm in the case that no fault occurred where no corrective action needed to be taken. Note that a selective reliability mode [31], [32] where some calculations occur in a high reliability mode that is assumed to be safe from the occurrence of faults, must be assumed since the computations performed during the correcting step need to be executed successfully.

As discussed in Section 5.2, convergence of the FGPILU algorithm is strongly related to the Jacobian of the functional iteration, G (i.e. Eq. (112)). In order to determine what steps need to be taken during the periodic correction step, it is important to make note of what needs to be accomplished. The mapping defined by G is a contraction if $||G'(x)|| < 1$ for some matrix norm $||\cdot||$. Therefore, if the initial guess x_0 has the property that $||G'(x_0)|| < 1$ then the algorithm should converge so long as the domain is locally convex. However, if a fault occurs on the fth iteration that causes the Jacobian to move into a region of the domain where G is no longer a contraction, or the domain is no longer convex, then subsequent iterations will not aid in convergence. Following this reasoning, a naïve correction step that constitutes a hybrid use of checkpointing and a periodic correction step would: (1) form the Jacobian explicitly, (2) calculate a matrix norm of the Jacobian, and (3) reset all non-zeros in both L and U (i.e. all elements of x) to the last known good state. By occasionally saving off the vector x when no fault has been detected to have occurred, the algorithm can avoid reverting back to the initial guess. Pseudocode for this algorithm is given by Algorithm 14.

Algorithm 14: Naïve algorithm for a hybrid self-stabilizing/checkpointing FGPILU

Input: Initial guesses for $l_{ij} \in L$ and $u_{ij} \in U$
Output: Factors L and U such that $A \approx LU$

1 **for** $sweep = 1, 2, \ldots, m$ **do**
2   **if** $sweep \equiv 0 \mod F$ **then**
3   Form the Jacobian of the current iterate, J
4   Evaluate $\tau = ||J||$
5   **if** $\tau < 1$ **then** Continue
6   **else**
7   Set l_{ij} and u_{ij} to the last known good state
8   **else**
9   **for** $(i, j) \in S$ **do in parallel**
10   **if** $i > j$ **then** $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}) / u_{jj}$
11   **else** $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$

Note that, while the algorithm presented in Algorithm 14 is most likely not viable due

to the high cost in both computation and memory associated with forming the Jacobian and calculating a matrix norm for such a large matrix, it illustrates the goal of all the fault resilient variants of any fixed point iterative method, including the FGPILU algorithm: ensure that the algorithm is still making progress towards the eventual solution.

In the case of nonlinear fixed point methods, this can be ensured by calculating the local Jacobian, and ensuring that the associated spectral radius still indicates that the mapping is locally a contraction. While the methods proposed in the subsequent subsections do not form and evaluate the Jacobian explicitly, the goal of each of them is to ensure progress of the FGPILU algorithm. Therefore the variants should have the same effect on the FGPILU algorithm as the naïve check presented in Algorithm 14.

If checkpointing is desired to be excluded entirely from the process of creating factors L and U with the FGPILU algorithm, then a failed check will result in a restart using the initial guess. Two large problems with Algorithm 14 are as follows:

1. The expense of the correction step. The cost of forming the Jacobian and evaluating its norm may be restrictive for many problems.

2. The reliance on knowing a previous good state. The quick convergence of the algorithm to usable L and U factors [23], [24] mitigates this issue somewhat since the original guess can always be reused, but if a higher level of fidelity is desired, then the runtime could be prohibitively long.

Convergence of this prototypical algorithm is captured in the following result.

Theorem 13. *For any state of $l_{ij} \in L$ and $u_{ij} \in U$, if a correction is performed in the kth sweep, and all subsequent iterations are fault-free then Algorithm 14 will converge.*

Proof. Since the Jacobian at the fixed point of the algorithm has spectral radius less than 1 (see [58]) and the correcting step of Algorithm 14 ensures that the 1-norm of the Jacobian associated with the current iterate is less than 1 – which forces the algorithm to stay in a

region of the problem domain where the asynchronous mapping defined by the algorithm is a contraction − Algorithm 14 will converge. □

While the method proposed by Algorithm 14 is not computationally viable, it does suggest a mechanism for creating a successful self-stabilizing variant of the FGPILU algorithm. First, a bound on the norm of the Jacobian that can be computed efficiently needs to be determined, and then a correcting mechanism that does not require (pseudo) checkpointing will need to be created. For the first issue, the following result from [24] can be used:

Theorem 14. (Chow and Patel) *Given a matrix A and G as defined above, the 1-norm of the current iterate G_i' can be bounded,*

$$||G_i'||_1 \leq \max(||U_i||_\infty, ||L_i||_1, ||R_i^L||_1)$$

where R^L is the strictly lower triangular part of $R = A - T$ and the matrix T is defined by,

$$T_{ij} = \begin{cases} (LU)_{ij} & (i,j) \in S \\ 0 & o/w \end{cases}$$

However there is still a larger than desirable computational burden in forming the matrix $R = A - T$ and the bound itself may not be sharp enough for practical use since the result is only useful if,

$$\alpha = \max(||U_i||_\infty, ||L_i||_1, ||R_i^L||_1) < 1 \tag{120}$$

In the case that the input matrix comes from a 5-point or 7-point finite difference discretization of a partial differential equation, the Theorem 14 simplifies further to the result provided below in Theorem 15.

Theorem 15. (Chow and Patel) *If A is a 5-point or 7-point finite different matrix, and if L and U have sparsity patterns equal to the strictly lower and upper triangular portions of*

A respectively, then for G as defined above, the 1-norm of the current iterate G_i' is given by,

$$||G_i'||_1 = \max(||U_i||_{max}, ||L_i||_{max}, ||A_L||_1)$$

where A_L is the strictly lower triangular part of A.

Development of a traditional checkpointing variants will be examined in the next subsections Section 5.3.1, while development of a checkpointing variant that attempts to leverage the fine-grained nature of the FGPILU algorithm is provided in Section 5.3.2. The use of a periodic correction step will be examined in the following two subsections: Section 5.3.3 provides a computationally light variant designed around the performance of the algorithm on finite-difference discretization of partial differential equations, and Section 5.3.4 provides a checkpoint free variant based upon the progression of a residual.

5.3.1 CHECKPOINTING

In this section, some theoretical bounds on the impact of a fault on the FGPILU algorithm are developed, and these projected impacts are used to develop checkpointing based fault tolerant adaptations to the original FGPILU algorithm. If a fault occurs at the computation of the kth iterate (affecting the outcome of the $(k+1)^{st}$ vector), it is possible to write the corrupted $(k+1)^{st}$ iteration of x as

$$\hat{x}^{(k+1)} = G\left(x^{(k)}\right) + r \tag{121}$$

where the vector r accounts for the occurrence of a fault. Note that the magnitude of r corresponds only to the soft fault that was injected and is not a part of the FGPILU algorithm itself: for a sweep of the algorithm that does not contain a fault, $r = 0$. To track the progression of the FGPILU algorithm, it was proposed in [24] and [23] to monitor the

nonlinear residual norm. This is a value

$$\tau = \sum_{(i,j)\in S} \left| a_{ij} - \sum_{k=1}^{\min(i,j)} l_{ik} u_{kj} \right| \tag{122}$$

which decreases as the number of sweeps progresses the factors produced by the algorithm closer to the conventional L and U factors that would be computed by a traditional ILU factorization. Note that the $\min(i, j)$ notation is used to emphasize that computation will be minimized and superfluous pairs of components will not be considered. Alternatively, the ILU residual can be considered which evaluates the same difference (i.e. the Frobenius norm of A) but over all entries as opposed to restricting the calculation to the sparsity pattern of S. Sample values for both the nonlinear residual and the ILU residual for the first few iterations / sweeps of the FGPILU algorithm on the Apache2 problem are given in Table 7. Apache2 is a finite-difference problem featured in the numerical experiment on symmetric problems given in Section 5.4.3. See Table 10 for descriptions of the symmetric example problems. Note that the nonlinear residual norm will continue decreasing, but that the ILU residual quickly settles to a non-zero value.

Table 7: Typical progression of both the nonlinear residual norm and ILU residual norm for the Apache2 test problem.

Sweep	Non-linear residual (τ)	ILU residual
1	1.05e+02	379.88
2	8.81e+01	376.74
3	2.38e+01	367.10
4	1.36e+01	366.45
5	2.39e+00	366.45
6	1.21e+00	366.45
7	5.24e-01	366.45
8	2.24e-02	366.45
9	1.05e-03	366.45

The Apache2 test problem in Table 7 is a three dimensional finite-difference discretization of partial differential equations that is one of the best conditioned matrices from the symmetric problem set shown in Table 11. Alternatively, Table 8 shows the nonlinear residual progression for the Apache2 problem featured above, the offshore problem (which is the most ill-conditioned problem from the symmetric problem set), and the two non-symmetric problems that are studied more extensively in the non-symmetric problem set. The two non-symmetric problems are studied more extensively in Section 5.4.4, with details provided in Table 12. The large difference in initial nonlinear residual norm between the different problems shows how far the standard initial guess for each problem is from the standard incomplete factorization using the same sparsity pattern as the input matrix.

Table 8: Typical progression of the nonlinear residual norm for a variety of test problems.

Sweep	Apache2	offshore	ecl32	fs_760_3
0	202.4	103.31	2128	13986
1	96.176	38.501	771.55	62.755
2	106.01	26.026	37.636	165.74
3	53.639	12.65	117.59	217.54
4	87.454	9.2839	5.1749	20.338
5	2.6809	4.9959	57.625	8.6786
6	0.87554	29.425	1.1898	8.2413
7	0.16503	79.832	1.879	11.663
8	0.055735	70.867	0.1794	6.3104
9	0.017221	5.6606	0.13366	0.64612
10	0.006134	0.9699	0.04506	0.19334

If a fault occurs on a given sweep, then one or both nonlinear equations from the FGPILU algorithm (c.f. Algorithm 11) will have some amount of error. In particular, the update

equations for l_{ij} and u_{ij} will become

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) + r_{ij} \tag{123}$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} + r_{ij} \ , \tag{124}$$

where r_{ij} represents the component of the vector r that maps to the (i, j) location of the matrix. Comparing Eqs. (123) and (124) with Eq. (122) shows that, if a fault occurs during the computation of the incomplete LU factors, then the nonlinear residual norm τ will be affected.

In order to ensure that a fault does not negatively effect the outcome of the algorithm, the first checkpointing variant that is proposed involves a simple monitoring of the nonlinear residual norm τ. In principle, since $S \subset A$, when the FGPILU algorithm converges, the nonlinear residual norm will be at a minimum, $\tau \approx 0$. Call this variant the Checkpoint All variant (CPA-FGPILU). The pseudo-code for this algorithm is provided in Algorithm 15.

Algorithm 15: Checkpoint-All-Based Fault Tolerant FGPILU (CPA-FGPILU)

Input: Initial guesses for $l_{ij} \in L$ and $u_{ij} \in U$
Output: Factors L and U such that $A \approx LU$

```
1  for sweep = 1, 2, ..., m do
2      if Fault then
3          Rollback L and U
4          Fault = FALSE
5          sweep = sweep − 1
6      else
7          for (i, j) ∈ S do in parallel
8              if i > j then  l_ij = (a_ij − Σ_{k=1}^{j−1} l_ik u_kj)/u_jj
9              else  u_ij = a_ij − Σ_{k=1}^{i−1} l_ik u_kj
10             τ^(sweep) = Σ_{(i,j)∈S} | a_ij − Σ_{k=1}^{min(i,j)} l_ik u_kj |
11             if τ^(sweep) > γ · τ^(sweep−r) then
12                 Fault = TRUE
```

In this case, a fault is declared if the currently computed nonlinear residual norm $\tau^{(sweep)}$ is some factor γ greater than the previously computed nonlinear residual norm $\tau^{(sweep-r)}$, where r provides a delay that determines how frequently the factors L and U are stored to memory.

Note that, due to a combination of the asynchronous nature of the the FGPILU algorithm and the nature of the input matrix itself, the nonlinear residual norm may not be strictly monotonically decreasing, especially as the algorithm proceeds closer to convergence. Therefore using the factor $\gamma = 1$, i.e., expecting a strict monotonic decrease, may cause the algorithm to report false positives, especially when nearing convergence (as judged by the progression of the nonlinear residual).

Additionally, while this method can be very effective for both detecting and recovering from faults, the computation of the global nonlinear residual is a relatively expensive computationally. This variant of the algorithm may induce more overhead then desired if the frequency of the check is not severely limited, which would in turn lower the effectiveness of the algorithm.

5.3.2 PARTIAL CHECKPOINTING

Next, note that since there is a contribution from every $(i,j) \in S$, the individual nonlinear residual norms for each $(i,j) \in S$, denoted here by τ_{ij}, can be defined as

$$\tau_{ij} = \left| a_{ij} - \sum_{k=1}^{\min(i,j)} l_{ik} u_{kj} \right| \tag{125}$$

where the total nonlinear residual norm can always be recovered by taking the sum of all the individual nonlinear residual norms over all $(i,j) \in S$. To establish a baseline for fault tolerance, define individual nonlinear residual norms τ_{ij} for each $(i,j) \in S$ based on the initial guess that is used to seed the iterative FGPILU algorithm. In particular, if L^* and U^* are the initial guesses for the incomplete L and U factors, then take $l^*_{ij} \in L$ and $u^*_{ij} \in U$

and define baseline individual nonlinear residual norms τ_{ij}^* using the original values τ_{ij} and the values $l_{ij}^* \in L$ and $u_{ij}^* \in U$.

Since for each sweep of the FGPILU algorithm, the components $l_{ij} \in L$ and $u_{ij} \in U$ can be computed, by testing the individual nonlinear residual norms it is possible to determine if a large fault occurred. Specifically, it is of interest to determine if a fault occurred that was large enough to cause a potential divergence of the algorithm. To do this, first a tolerance t is set and then a fault is signaled if

$$\tau_{ij} > t \tag{126}$$

since the individual nonlinear residual norms are generally decreasing as the FGPILU algorithm progresses. Set the value t as $t = \max(\tau_{ij}^*)$ initially (Line 3 of Algorithm 16), and then update t during the course of the algorithm if desired. It is also possible to use the previous individual nonlinear residual norms as opposed to a maximum that is taken across all current nonlinear individual norms. In particular, similarly to the global checkpointing variants advocated in Section 5.3.1, a fault can be declared if,

$$\tau_{ij}^{sweep} > \gamma \cdot \tau_{ij}^{sweep-r} \tag{127}$$

for parameters γ and r similar to those in the CPA-FGPILU variant.

Note that if a fault is signaled by any of the individual nonlinear residual norms, it is only known that a fault occurred somewhere in the current row of the factor L or the current column of the factor U. As such, the conservative approach would require the rollback of both the current row of L and the current column of U to their values at the previous checkpoint (e.g., Lines 5 to 9 of Algorithm 16).

It is possible for the individual nonlinear residuals as defined to increase by a small amount, especially at very early or very late iterations in the progression of the algorithm. To counteract the potential for reporting false positives on fault detection, the derivative of

the global nonlinear residual, $\frac{\Delta\tau}{\Delta t}$, can be checked to ensure that it is also increasing before switching the current row and/or column (see Line 15 of Algorithm 16).

Algorithm 16: Partial Checkpoint-Based Fault Tolerant FGPILU (CP-FGPILU)

Input: Initial guesses for $l_{ij} \in L$ and $u_{ij} \in U$
Output: Factors L and U such that $A \approx LU$

```
1  for (i, j) ∈ S do in parallel
2      τ_ij = | a_ij − Σ_{k=1}^{min(i,j)} l_ik u_kj |
3  t = max(τ_ij)
4  for sweep = 1, 2, ..., m do
5      if Fault then
6          Set i = max_{i,j}(k^1_ij) and j = max_{i,j}(k^2_ij)
7          Rollback {l_ik}_{k=1}^{i−1} and {u_kj}_{k=1}^{j−1}
8          Fault = FALSE
9          sweep = sweep − 1
10     else
11         for (i, j) ∈ S do in parallel
12             if i > j then  l_ij = (a_ij − Σ_{k=1}^{j−1} l_ik u_kj)/u_jj
13             else  u_ij = a_ij − Σ_{k=1}^{i−1} l_ik u_kj
14             Compute τ
15             if τ_ij > t and τ' > 0 then
16                 Set k^1_ij = i and k^2_ij = j
17                 Fault = TRUE
```

Note that if a fault is detected, the algorithm only restores (i.e., "rolls back") the affected row of L and column of U. Additionally, since in practice it has been proposed [23], [24] to use a limited number of sweeps of the FGPILU algorithm as opposed to converging the algorithm according to the global nonlinear residual norm, the number of sweeps conducted is decremented so that all elements of L and U are updated *at least* the desired number of times. Also note that the `for` loop on Line 11 of Algorithm 16 extends over all elements $(i, j) \in S$ so that every individual nonlinear residual norm is checked. Because of this, if there are multiple faults that cause the individual nonlinear residual norms to exceed the threshold τ_{ij}, they should all be detected.

While no global communication is required to check for the presence of a fault via the individual nonlinear residual norms, τ_{ij}, there is global communication required to compute the derivative of the global nonlinear residual norm. A simple (forward) finite difference scheme is used to approximate this derivative to minimize the global communication required by Algorithm 16. The frequency with which the global nonlinear residual norm is computed can be determined independently of the rest of the algorithm. Specifically, it may be possible to compute these updates less frequently in order to minimize the communication that takes place between the different components.

Additionally, if a fault is detected there will be some communication required between processes in order to fix the effects of the fault. Since the component detecting a fault will have to roll back elements that it is not directly responsible for updating, further computation on all affected elements will have to cease momentarily. Note also that when using the CP-FGPILU algorithm, the size of the faults that are not caught by the algorithm are determined by the tolerance that is set. In particular,

$$||r|| \leq t \tag{128}$$

where r represents a fault that was not caught by the proposed checkpointing scheme, since if $||r|| > t$ then the fault would be caught by the check on Line 15 of Algorithm 16. This, in turn, affects the update equations: Eq. (121) as well as Eqs. (123) and (124).

5.3.3 PERIODIC CORRECTION STEP

The periodic correction step must be computed reliably regardless of what actions are undertaken during the periodic correction in order to ensure that the algorithm will continue to progress towards convergence. In particular, it cannot be negatively affected by the occurrence of a fault. Despite the robustness of an explicit check on the norm of the Jacobian as proposed in the beginning of this section (see Algorithm 14), the emphasis here will be upon developing variants of the FGPILU algorithm that are able to mitigate the impact of a soft fault without requiring the explicit formation of the Jacobian for the current iterate.

The first variant of the FGPILU algorithm that makes use of a periodic correction step is shown in Algorithm 17. An update sweep is expected every F iterations. The implicit expectation is that the steps that are undertaken during this periodic correction step will be able to mitigate any potential consequences of a soft fault that occurs during the prior $F-1$ iterations.

Algorithm 17: Self-Stabilizing Fault Tolerant FGPILU (SS-FGPILU)

Input: Initial guesses for $l_{ij} \in L$ and $u_{ij} \in U$, parameter F that defines the frequency of the periodic correction step, and a parameter β to determine the strictness of the component level check

Output: Factors L and U such that $A \approx LU$

1 **for** $sweep = 1, 2, \ldots, m$ **do**
2   **if** $sweep \equiv 0 \mod F$ **then**
3   **for** $(i,j) \in S$ **do in parallel**
4   **if** $\{\|l_{ij}\|, \|u_{ij}\|\} \gg \|a_{ij}\|$ *or* $|\{l_{ij}, u_{ij}\} - a_{ij}|/|a_{ij}| > \beta$ *or* $\{l_{ij}, u_{ij}\} = \{0, \texttt{NaN}\}$ **then** $\{l_{ij}, u_{ij}\} = a_{ij}$
5   **if** $i > j$ **then** $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}) / u_{jj}$
6   **else** $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$
7   **else**
8   **for** $(i,j) \in S$ **do in parallel**
9   **if** $i > j$ **then** $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}) / u_{jj}$
10   **else** $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$

Algorithm 17 was designed to correct problems arising from simple finite difference discretizations of partial differential equations (i.e. L2D and APA from Table 10). The technique of observing the magnitude of the elements used in the fixed point iteration and their relative change was created after observing the component-wise progression of all of the elements in the preconditioning factors that are generated for the discretization of the two dimensional Laplacian with a 5-point stencil. As will be discussed further in Section 5.4 (see specifically, Section 5.4.3.1) this technique will not generalize to all other problems but may extend to other similar matrices (i.e. symmetric positive definite, strongly diagonally dominant, small bandwidth, etc).

The following result establishes a convergence property for the variant of the FGPILU algorithm proposed in Algorithm 17.

Theorem 16. *For any state of $l_{ij} \in L$ and $u_{ij} \in U$, if a correction is performed in the kth sweep, all subsequent iterations are fault-free, no elements in the final L and U factors differ by more than β percent from the original factors in the matrix A, and β is chosen such that if a fault occurs a fault is signaled, then the algorithm using a periodic correction step that is featured in Algorithm 17 will converge.*

Proof. This follows from noticing that the correcting (or "stabilizing") step (Lines 2 to 6 of Algorithm 17) ensures that the state $l_{ij} \in L$ and $u_{ij} \in U$ of the incomplete L and U factors will be in the original domain of the problem and then invoking the convergence arguments for the original FGPILU algorithm (see [24]) which rely upon the assumptions and base arguments from [58]. □

5.3.4 COMPONENT-WISE RESIDUAL CHECK

The last resilient variant of the FGPILU algorithm to be discussed relies on tracking the component-wise progression of the individual nonlinear norms (Eq. (125)), in a manner similar in spirit to Algorithm 16. Recall from Section 5.4.3.1 that the individual nonlinear

residual norms are not strictly monotonic in their decrease; however, by periodically checking the progression of the individual τ_{ij}'s it is possible to use them to detect faults without relying on computation of the global nonlinear residual norm which requires communication between all of the components. This scheme is detailed in Algorithm 18.

Algorithm 18: Component-Wise Residual Check for FGPILU (CW-FGPILU)

Input: Initial guesses for $l_{ij} \in L$ and $u_{ij} \in U$, parameters F and α that define the frequency and strictness of the periodic correction step respectively

Output: Factors L and U such that $A \approx LU$

```
1  for sweep = 1, 2, ..., m do
2      if sweep ≡ 0 mod F then
3          for (i, j) ∈ S do in parallel
4              if τ_ij^{sweep} > γ · τ_ij^{sweep−F} then
5                  Set k_ij^1 = i and k_ij^2 = j
6                  Set i = max_{i,j}(k_ij^1) and j = max_{i,j}(k_ij^2)
7                  Rollback {l_ik}_{k=1}^{i−1} and {u_kj}_{k=1}^{j−1}
8      else
9          for (i, j) ∈ S do in parallel
10             if i > j then  l_ij = (a_ij − Σ_{k=1}^{j−1} l_ik u_kj)/u_jj
11             else  u_ij = a_ij − Σ_{k=1}^{i−1} l_ik u_kj
```

The CW-FGPILU algorithm variant (Algorithm 18) can be seen as a modified version of the partial checkpointing method from Algorithm 16, where the check on the global nonlinear residual norm τ is omitted but the frequency of the check on the progression of the individual nonlinear residual norms is increased to compensate. This method can limit the amount of communication that takes place between the individual components in the factors L and U.

The convergence of the CW-FGPILU is related primarily to two key factors: (1) the detection rate and (2) the periodicity of the check. In a practical sense, the rate of detection of the CW-FGPILU algorithm is determined by a combination of the size of the fault, measured by the impact of the fault on the nonlinear residual norm, τ, and the size of the factor γ which helps control the amount of false positives that the algorithm reports. The periodicity

of the check is controlled by the parameter F. The smaller F is, the more frequently the checks occur; raising both the computational burden on the program and the likelihood of detecting a fault before it is able to propagate to other elements in the preconditioning factors L and U. Reducing F to 1 allows the check on the individual nonlinear residual norms to be applied each time an update is computed, and, hence, provides a fine-grained fault detector to each new value and accept or reject it based upon the tolerance defined by γ.

Since the convergence of the algorithm is determined by a combination of these two factors, the algorithm will converge if the periodicity is small enough, such that faults are detected before they have a chance to propagate much their effects into too many elements of L and U, and γ is selected such that faults that have a negative impact on the convergence of algorithm are detected. Even if certain component updates are rejected due to an increase in the corresponding individual nonlinear residual norm τ_{ij}, the FGPILU algorithm is designed to converge in an asynchronous computing environment under the standard mild assumptions about the nature of the asynchronous computing set-up (see Theorem 3.5 of [24]). As such, even though the updates may become out-of-sync due to the rejection of certain updates, the algorithm will still converge to the intended result.

5.3.5 NOTES ON THE CONVERGENCE OF THE FGPILU VARIANTS

The main result concerning the convergence of the FGPILU algorithm comes from [58], but this result only guarantees a neighborhood of the fixed point (i.e. the final incomplete L and U factors) in which the algorithm is convergent. For certain problems, this neighborhood may be quite large (in a practical sense), where many different initial guesses will exhibit good convergence properties. In such a scenario, a fault may delay convergence by moving the current iterate farther away from the fixed point, but not cause divergence by moving the current iterate outside of the neighborhood of the fixed point guaranteed by the main convergence result.

For other problems (specifically with matrices that are far from symmetric or highly indefinite) this neighborhood may not encapsulate a large portion of the problem domain. In this case, care must be taken to use a good initial guess to get the FGPILU algorithm to converge at all. Additionally, if a fault does occur it is quite possible for the fault to move the current iterate to a location in the domain where further iterations will not help the algorithm progress towards convergence.

Convergence of the FGPILU algorithm is closely related to the Jacobian associated with the nonlinear update equations Eq. (112). If a fault occurs that is not caught by the fault detection (either the periodic correction step, or by the fault detection mechanisms in the checkpointing variants) of the FGPILU algorithm, then it is possible for the Jacobian to move to a regime of the domain where the fixed point mapping that represents the FGPILU algorithm is no longer a contraction (i.e. $||J|| > 1$). In this case, the fault tolerance mechanisms of the FPGILU variants will not help, and subsequent iterations of the algorithm will not aid in convergence.

The convergence of the checkpoint-based variants of the FGPILU variants follows directly from the convergence of the original FGPILU algorithm. Assuming that faults do not occur after a certain number of sweeps, the algorithm will converge under the assumption that it was successfully returned to a state not affected by a fault. Note that if a fault is detected, the state is restored to the last known good state - how recent that state is depends on the frequency with which the checkpoint is stored. More frequent storage of a “good” state via checkpointing will slow down the overall progression of the algorithm, but will provide a more recent fail-safe state if a fault is detected.

Additionally, note that an application of the FGPILU preconditioner is effectively only an approximation of the conventional ILU preconditioner. The application of the generated preconditioners can be expressed as, $\widetilde{z}_j \approx P^{-1} v_j$. Both [23], [24] have shown that it is possible to successfully use the incomplete LU factorization resulting from the FGPILU algorithm before it has converged completely – when convergence is judged by the progression of the

nonlinear residual norm, τ, below some threshold tolerance, ϵ. It is therefore possible that any adverse affects that a fault may have on the convergence of the FGPILU algorithm itself will not have sufficient time to propagate throughout the entirety of the computed L and U factors to have a meaningful impact on the performance of the overarching iterative method (e.g. CG, GMRES) that the computed factors are used as preconditioner for. These potential impacts will be explored numerically in Section 5.4.

5.4 NUMERICAL RESULTS

5.4.1 EXPERIMENTAL SET-UP

The experiments were all conducted on the Turing High Performance computing cluster at Old Dominion University. For the experiments with symmetric matrices, a NVIDIA Tesla®[4] K40m GPU was used, while for the experiments featuring the non-symmetric problem set a NVIDIA Tesla® K80 GPU was used. The nominal, fault-free iterative incomplete factorization algorithms and iterative solvers were taken from the MAGMA open-source software library [182], and minimal modifications were made to the existing MAGMA source code in order to implement the modifications to the FGPILU algorithm, add the α-shift, and to inject faults into the algorithm. Note that this approach causes the preconditioning factors to be applied in a manner more similar to conventional incomplete factorizations whereby the application is not fine-grained or asynchronous. In both the symmetric problems presented in Section 5.4.3 and the non-symmetric problems presented in Section 5.4.4, a set of parameters were chosen for each of the algorithm variants that apply to all matrices reasonably well; however, parameter choices for a specific problem could be tuned more efficiently. All of the results provided in this dissertation reflect double precision, real arithmetic.

[4]NVIDIA Corp., Santa Clara CA

5.4.2 FAULT MODEL

Three fault-size ranges (corresponding to differing orders of magnitude) for the faults injected by the perturbation-based model (c.f., PBSFM from Chapter 4) were considered:

$$r_i \in (-0.01, 0.01) \tag{129}$$

$$r_i \in (-1, 1) \tag{130}$$

$$r_i \in (-100, 100) \tag{131}$$

The bit-flip model was included to appropriately gauge the worst case scenario, but no effort was made to force the bit selected to be in a particular position. Because of this, the impact of a a bit-flip ranges from almost none (bit-flip in less significant bit of mantissa) to catastrophic (bit-flip in exponent or sign). Results for both the perturbation-based soft fault model (PBSFM) and the bit-flip model (BFSFM) are presented separately, but as averages over all trials executed for each methodology. Note that the working assumption is that faults only effect the values of the entries l_{ij} and u_{ij}. If faults are also allowed to affect the indices used in the sparse storage scheme, then it is possible that the strictly lower triangular structure of the Jacobian could be altered which would have a large impact on the convergence of the FGPILU algorithm.

5.4.2.1 Comparison of fault models

In order to provide a better feel for the effect that each of these fault models can have, a quick investigation into the relative effects of each fault model is presented in this subsection. Each of these methods for simulating the occurrence of a fault works on an input vector, x, and corrupts in some way the specified component(s).

In order to illustrate the potential impact of each fault model for the FGPILU algorithm under study in this chapter, x is taken to be the initial set of non-zero components for the

2D finite difference discretization of the Laplacian that is used (i.e. the `LAPLACE2D` matrix detailed in Table 10). Recall that all of the matrices in this dissertation are symmetrically scaled to have unit diagonal, so that the entries in the vector x are bounded inside of $[-1, 1]$.

Due to the non-deterministic nature of both of these fault models, the comparison between them was made over 1000 trials. In each trial, a fault is injected according to one of the methodologies in order to create a vector with a fault, $\hat{x}$, and the norm of the difference in these two quantities,

$$d = ||x - \hat{x}|| \tag{132}$$

was computed. In this comparison, the magnitude of $\hat{x}$ is bounded for the perturbation-based fault model, but it is possible for the bit-flip fault model to produce a result of either `NaN` or `INF` for certain combinations of component and bit selected. For the purposes of this quick look analysis, these results were discarded since scanning for either of these incorrect values is not a difficult problem. Summary results are provided in Table 9.

In the table, the 'Bit-flip Model' column corresponds to randomly selecting a single component of the vector x, randomly selecting a bit to flip, and injecting a single bit-flip. The column 'Bit-flip Model (bounded)' corresponds to the same bit-flip model, but where bit-flips that caused large magnitude changes due to bit-flips in exponent bits were removed. In particular, any instance where $d > 10000$ was removed from the data. The three columns corresponding to the perturbation-based soft fault model (PBSFM) are separated by the bounds on the range that the perturbations were sampled from. The (s) column corresponds to faults in $r_i \in (-0.01, 0.01)$, the (m) column to faults in $r_i \in (-1, 1)$, and the (l) column relates to faults in $r_i \in (-100, 100)$.

The vector d corresponds to the size of the fault introduced by the given fault model. In the table, the mean of the 1000 entries of d is provided, along with the maximum value, and the mean and standard deviation of the log of the entries in d.

The data presented in Table 9 shows the potential impact of a fault introduced by each

Table 9: Comparison of the effects between the various fault models used for the matrix LAPLACE2D.

	Bit-flip Model	Bit-flip Model (bounded)	PBSFM (s)	PBSFM (m)	PBSFM (l)
mean(d)	—	8.2388e-02	6.4500e+00	6.4499e+02	6.4499e+04
max(d)	4.4942e+307	1.0000e+00	6.4593e+00	6.4575e+02	6.4584e+04
mean(log(d))	-3.2281e+00	-7.0040e+00	8.0956e-01	2.8096e+00	4.8096e+04
std(log(d))	3.4646e+01	5.0639e+00	1.7075e-04	1.7287e-04	1.7194e-04

of the fault models. Note that the 'Bit-flip Model' contained 12 cases (1.2%) that exceeded the threshold of $||x - \hat{x}|| > 10000$, indicating that while a severely large impact is possible, it is not probable. The statistics on the log values of the entries in d gives some indication as to the relative order of magnitude of the various fault models, and the spread of the level of impact. Generally, the size of the faults induced by the bit-flip model are much more varied than those created by the perturbation-based soft fault model. The perturbation-based model was selected in order to model the typical worst case effect on the FGPILU algorithm, and the inclusion of the bit-flip model was intended to provide completeness and show that the fault tolerant variants proposed throughout this chapter are capable of handling large errors.

5.4.3 RESULTS FOR SYMMETRIC MATRICES

The test matrices that were used in this set of experiments predominantly come from the University of Florida sparse matrix collection maintained by Tim Davis [183], and the matrices selected for this dissertation are the same as the ones that were selected for the studies [23], [169], [170] that looked into the performance of the FGPILU algorithm on GPUs both with and without the presence of faults. Note that these problems also include the problems selected by NVIDIA®[5] for testing the incomplete LU factorization that is part of the CUDA® library [184].

There are six matrices selected from the University of Florida sparse matrix collection, and the two other test matrices that were used come from the finite difference discretization

[5]Nvidia Corporation, Sunnyvale CA

of the Laplacian in both 2 and 3 dimensions with Dirichlet boundary conditions. For the 2D case, a 5-point stencil was used on a 500×500 mesh, while for the 3D case, a 27-point stencil was used on a $50 \times 50 \times 50$ mesh.

All of the matrices considered in this portion of the dissertation are symmetric positive-definite (SPD) and as such the symmetric version of the FGPILU algorithm (i.e. the incomplete Cholesky factorization) was used. Also, recall from Section 5.1 that each of the eight matrices used in this dissertation will be symmetrically scaled to have a unit diagonal in order to help improve the performance of the FGPILU algorithm. A summary of all of the matrices that were tested is provided in Table 10.

Table 10: Summary of the 8 symmetric positive-definite matrices used in this dissertation. Descriptions come from [183].

Matrix Name	Abbreviation	Dimension	Non-zeros	Description
APACHE2	APA	715,176	4,817,870	SPD 3D finite difference
ECOLOGY2	ECO	999,999	4,995,991	circuit theory applied to animal/gene flow
G3_CIRCUIT	G3	1,585,478	7,660,826	circuit simulation problem
OFFSHORE	OFF	259,789	4,242,673	3D FEM, transient electric field diffusion
PARABOLIC_FEM	PAR	525,825	3,674,625	parabolic FEM, diffusion-convection reaction
THERMAL2	THE	1,228,045	8,580,313	unstructured FEM, steady state thermal problem
LAPLACE2D	L2D	250,000	1,248,000	Laplacian 2D finite difference, 5-point stencil
LAPLACE3D	L3D	125,000	3,329,698	Laplacian 3D finite difference, 27-point stencil

Plots of where the non-zeros are located in the matrix are provided for all eight matrices in Fig. 31 for the case where the matrices are unordered, and in Fig. 32 for the case where all of the matrices have been reordered using a Reverse Cuthill-Mckee (RCM) algorithm. The RCM algorithm is designed to reduce the bandwidth of the input matrix, and this effect can be seen in the clustering of non-zero terms around the main diagonal in the images shown in Fig. 32 relative to the dispersal of non-zero elements shown in Fig. 31. This reordering was shown to be effective for similar matrices with respect to the convergence of the FGPILU

algorithm in [23], [24], [169], [170]

Additionally, the condition number of each of these matrices (as estimated by the `condest` function in `MATLAB`® gives some further indication of how easy the problem will be to solve. Matrices with a lower condition number tend to have better performance in iterative methods.

Table 11: Condition number for each of the symmetric positive-definite problems.

Matrix	Condition Number
APACHE2	5.3169E+06
ECOLOGY2	6.6645E+07
G3_CIRCUIT	2.2384E+07
LAPLACE2D	6.0107E+03
LAPLACE3D	1.1060E+03
OFFSHORE	2.2384E+13
PARABOLIC_FEM	2.1108E+05
THERMAL2	7.4806E+06

The experiments in this section are divided into two sets. This first set of experiments focuses on the convergence of the FGPILU algorithm despite the occurrence of faults and features comparisons of the L and U factors produced by the preconditioning algorithms. The second set of experiments shows the impact of using in a Krylov subspace solver the preconditioners obtained from the first set of experiments. Note that in all of the experiments conducted, the condition $u_{jj} = 0$ was never encountered. Since all the test matrices are SPD, the preconditioning algorithms are Incomplete Cholesky variants, and the solver is the preconditioned conjugate gradient (PCG), as implemented in the MAGMA library [182].

Finally, note that the implementation of the variants that were examined in this chapter is not necessarily optimal from a performance point of view. The goal of the experiments was to quantify the ability of each of the variants proposed to provide a measure of resilience to the FGPILU algorithm when it is forced to run through undetected (by the system) soft faults. This focus translates to the observing the efficacy of the various algorithms which is

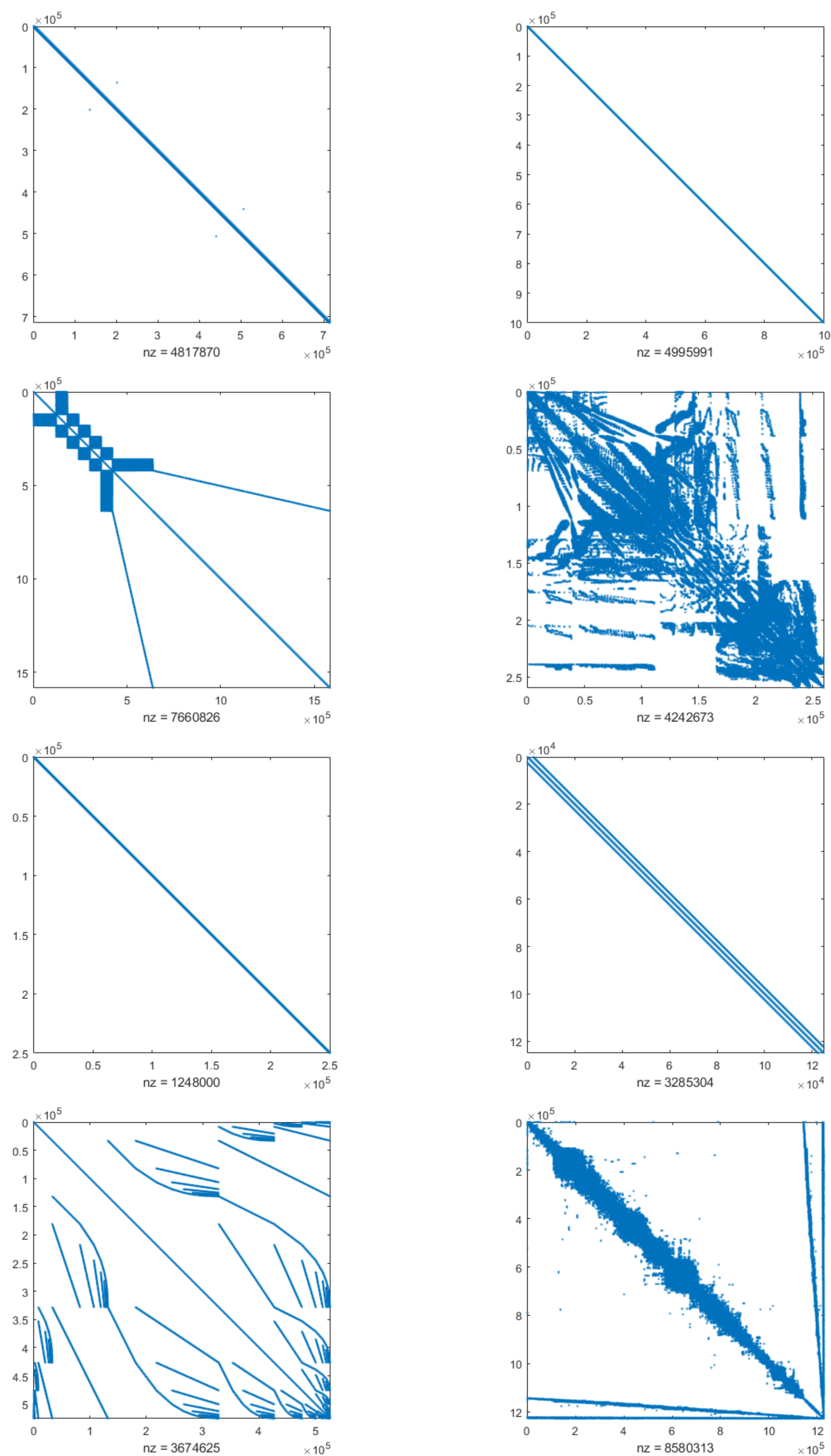


Fig. 31: Sparsity plots showing the location of all non-zeros for each of the 8 matrices with no reordering applied that were considered in the first set of experiments.

captured in the results that are presented throughout the remainder of this section. Because of this focus, the excessively small convergence chosen to declare the FGPILU algorithm converged (i.e. 10^{-8}), and some issues with resource contention, the time for all of the FGPILU variants (e.g. Fig. 36 (right) and Fig. 37 (right)) may be inflated relative to the performance of traditional incomplete factorization (`IC`). Further optimization, including the use of optimal checkpointing libraries for GPU based applications (i.e. [185], etc) and extended performance analysis would be needed to produce performance-oriented prototypes of each of the variants.

5.4.3.1 Convergence of FGPILU algorithm

Here, the FGPILU algorithm is said to have converged successfully if the nonlinear residual norm progresses below 10^{-8}. Although this threshold is unnecessarily small from a practical point of view—it is possible to achieve good performance from a preconditioner with a larger nonlinear residual norm—it was chosen so that more sweeps would have to be conducted before the algorithm converges to better judge the impact of faults. The progression of the nonlinear residual norm for a single fault-free run of each problem is depicted in Fig. 33 (left), which is a as an example of the typical progression of the nonlinear residual norm as the algorithm progresses towards convergence.

To illustrate the potential impact of a fault, Fig. 33 (right) shows the impact a fault can have on the FGPILU algorithm when it is injected (and ignored) at the beginning, the middle, or near the end of how long it would take the algorithm to converge with no faults present. Note that the Apache2 test problem converges to the desired level of nonlinear residual in 20 iterations when faults are not present.

From Fig. 33 (right), it may be observed that it took about twice as many sweeps for FGPILU to converge under a single occurrence of a fault; and the number of these extra sweeps is similar for each of the three injection locations. Although the example shown in Fig. 33 (right) is typical of what was observed experimentally with the test cases selected,

it is by no means general or conclusive. Faults may cause the FGPILU algorithm to diverge entirely or cause the resulting L and U factors to cause the Krylov subspace solver to stagnate or even diverge. A major point of the example in Fig. 33 (right) is to show the non-monotonous decrease of the FGPILU residual norm after a fault takes place.

Aggregate results for the performance of several variants of FGPILU algorithm are provided in the following figures as follows:

- when no attempt is made to mitigate the impact of the faults (`No FT`),
- the CPA-FGPILU variant wherein the L and U factors may be replaced in their entirety and is described in Algorithm 15 (`CPA`),
- the CP-FGPILU which rolls back a single row and column of the L and U factors and is described in Algorithm 16 (`CP`),
- the periodic correction step based on checking component-wise progression of the elements in the L and U factors and is given in Algorithm 17 (`SS`),
- the periodic correction step based on checking component-wise progression of the individual nonlinear residuals, τ_{ij} which is given in Algorithm 18 (`CW`).

Perturbation-Based Faults Now, the effects of a soft fault (modeled as a perturbation as described in Chapter 4) on the FGPILU algorithm and the variants discussed throughout this chapter are examined. The convergence of the FGPILU algorithm itself - as judged by the number of sweeps until the desired tolerance is met and the percent of trials that resulted in preconditioning factors that led to a successful solve of the associated linear system - is given in Fig. 34.

Figure 34 (left) shows the average number of sweeps to reach convergence for the cases that were successful. Note that this number is generally lower for the checkpoint-based schemes but that this is not the case for all of the problems that were tested. However, the higher success rate of the CPA-FGPILU and CP-FGPILU algorithms combined with the

generally faster convergence of those methods suggests that, with the parameters used in this dissertation, they are more effective at mitigating faults.

The small degradation in the number of sweeps to convergence depicted in Fig. 34 (left) for certain problems (i.e., L3D) for the `No FT` variant reflects the fact that only successful runs are included in the averages here. In Fig. 34 (right), a corresponding drop in the "success rate" can be seen for the problems where the increase in the number of sweeps required is not as large as expected for variants without fault mitigation. Here, a preconditioner is deemed as resulting in success if both the FGPILU converges, and the PCG solve using it terminates before the maximum number of iterations is reached. Practically, this means that if a fault caused the FGPILU algorithm to diverge and/or produce preconditioning factors that could not lead to convergence inside of the PCG solver, then the amount of sweeps required for the FGPILU algorithm would not be included in the left images of either Fig. 34 or Fig. 35, but that this run would cause the success rates captured in the right of Fig. 34 and Fig. 35 to decrease.

For the FGPILU variants tested, the success rates captured in Fig. 34 (right) show that both of the checkpoint-based variants are usually more successful than the self-stabilizing one at mitigating faults modeled as perturbations and producing acceptable preconditioners.

It is important to note that a large, unoptimized value of $\beta = 4$ was used for the percent difference check inside of the `SS` runs, and that this value may certainly be improved and tuned for the particular case at hand. The lower success rates associated with the SS-FGPILU algorithm are due to the fact that some of the smaller faults are not caught by this large value of β and the Jacobian moves to a portion of the domain where the mapping is not a contraction. It is possible that the method presented by this algorithm could be tuned to the specific problem at hand in a manner that efficiently made the FGPILU algorithm resilient to soft faults.

Bit-Flip Faults Next, results concerning the convergence of the FGPILU algorithm (and the variants presented in this work) when subjected to faults directly corresponding to a bit-flip are provided. The range of impacts possibly induced by a bit flip fault is wider than those caused by the perturbation-based fault model that was used above in the previous subsection. This gives the possibility of creating a fault that drastically impedes the ability of the FGPILU algorithm to converge as well as making it possible for a fault to have an almost negligible impact; detectable by only the strictest of fault detection mechanisms. As before, the results are averaged over multiple trials and aggregate results are presented.

Figure 35 (left) shows the number of sweeps until convergence for each of the FGPILU algorithm variants when subjected to a single bit-flip fault. The number of sweeps in this case (i.e. with a bit flip instead of a perturbation) is fairly consistent across the methods tested, especially when compared with Fig. 34. The success rates for the trials run with bit flips (see Fig. 35 (right)) are significantly higher relative to the success rates when the algorithm variants were subject to perturbation-based faults. This owes to the fact that only a single component is affected by the faults injected using a bit-flip based methodology.

Generally speaking, the higher variance with the amount of data corruption associated with a random bit flip causes the trials using a bit-flip fault methodology to have either very little or catastrophic impact. This is seen when comparing Fig. 34 and Fig. 35 in that in the number of sweeps taken until convergence on the successful runs (i.e. the left images of each figure) the number of sweeps until convergence is generally lower for faults modeled as bit flips and that the variance in performance (as judged by the number of sweeps until convergence) between the different variants of the FGPILU algorithm is lower.

5.4.3.2 Preconditioner Performance in Iterative Methods

In this set of experiments, a maximum number of 3000 PCG iterations was used; any run that had not converged by that point was declared to have diverged. While all of the preconditioners to be evaluated are forms of incomplete LU decomposition, they are

constructed by algorithms described in Section 5.4.3.1. For the purpose of an extended comparison, results are provided for the traditional Incomplete Cholesky (`IC`) and the Fine Grained Parallel Incomplete Cholesky (`ParIC`); neither of these two variants is subjected to faults.

Perturbation-Based Faults Figure 36 captures only the cases in which a preconditioner was successfully prepared (c.f. Fig. 34 (right)). Figure 36 (left) indicates that a successful FGPILU variant is typically capable of accelerating the PCG solve to the levels similar to those achieved by the no-fault constructions of a more traditional incomplete LU factorization. The few anomalous bars from Fig. 36 (left) correspond to runs of the FGPILU algorithm where no fault tolerance was attempted (`NoFT`) and enough of these runs were able to produce a PCG solve that converged in far more iterations than would typically be required to skew the averages. This seems to suggest that this behavior is not entirely anomalous and that the FGPILU algorithm has some nature level of resilience (else, the solves would not have been "successful" in the first place) to soft faults.

The timing results presented in Fig. 36 (right) are for the total time required for the preconditioner preparation and PCG solve. While the former may vary greatly depending on which variant is considered, the latter is rather uniform across the variants due to their similar numbers of iterations performed to convergence. More efficient implementations of the fault tolerance mechanisms and a more realistic tolerance for the nonlinear residual norm may improve the performance of the three fault-tolerant variants of the FGPILU algorithm, however the initial results show that the periodic correction step proposed in Algorithm 18 and represented by `CW` may be one of the more efficient variants.

Bit-Flip Faults Again, the differing impacts caused by a fault modeled as a bit-flip - as opposed to the perturbation-based data corruption that corresponds to the bit flip fault injection methodology described in Chapter 4 - are explored at the level of timing and accuracy results in the corresponding PCG solve.

Figure 37 (left) shows that the number of sweeps required for the PCG solver to convergence is even across all FGPILU algorithm variants. This shows that when the corresponding FGPILU algorithm variant *successfully* produces preconditioning factors the effect that the factors have on the PCG solver is similar. The fact that no runs without fault tolerance (`NoFT`) were able to converge in a large number of iterations similar to Fig. 36 (left) is also indicative of the dichotomy of possible effects caused by a bit-flip; either the effect is fairly negligible and the preconditioning factors that are produced accelerate the PCG solve as expected, or the effect is large enough that incomplete factorization does not lead to a successful solve of the associated linear system.

Conversely, Fig. 37 (right) shows that the time required for both preconditioner preparation and the PCG solve vary more from on method to another. There is more overhead associated for the two checkpointing schemes than the other variants and this could be (at least partially) mitigated by optimizing the number of times the required checkpoint data is stored to limit the data transfer and read/write overhead, or improving the implementation that is used for checkpointing. This is seen as well in Fig. 36 (right) but the discrepancy between the checkpointing based variants (`CP` and `CPA`) and the other variants is not as great. In the case of the periodic correction step variants (`SS` and `CW`) the overhead is possibly due to the extra work required on the component level since the perturbation-based faults tend to corrupt all of the components in the preconditioning factors L and U whereas in the bit-flip fault only a single component is corrupted. In general, the `CW` variant seems to exhibit the least amount of overhead from a time oriented perspective.

5.4.3.3 Discussion of FGPILU algorithm variants in the symmetric case

The experiments conducted here have shown that (1) the FGPILU algorithm is naturally resilient to smaller faults as modeled here—either by perturbations or bit-flips that affect less significant bits in the mantissa—and (2) larger faults can cause FGPILU to diverge and produce L and U factors that (if used) prohibit the corresponding Krylov subspace method

from solving the original linear system $Ax = b$ successfully. Examining the images on the right side of Fig. 34 and Fig. 35 a few conclusions can be drawn:

- The data indicates that the FGPILU algorithm and the variants discussed here tend to be more resilient to errors that only corrupt a single component.
- The rates of successful convergence within the desired tolerance are higher for all the proposed variants then for the original algorithm, regardless of the generated fault types.

Highlights of FGPILU variant differences The component-wise check put forth in Algorithm 18 (`CW`) has the ability to be implemented in a very efficient manner, but it may not detect faults as well as the `CP` algorithm (Algorithm 16) from which it stems. For particular problems that have a higher natural success rate (see the `NoFT` columns from the images on the right side of both Fig. 34 and Fig. 35), the `CW` variant could provide a low overhead approach to fault tolerance for the FGPILU algorithm.

The two checkpointing-based algorithms (`CP` and `CPA`) offer the highest likelihood of achieving the correct final answer, but also tend to rank quite highly with respect to the time required for convergence. One possibility to alleviate this additional computational burden is to adjust their input parameters to lessen the amount of checkpointing that occurs based on problem at hand, which is beyond the scope of the work presented in this dissertation. Hence, the results reported here focused only on a single set of parameters designed to compare the variants and show their potential efficacy.

The self-stabilizing variant (`SS`) may need the most work of any of the variants in terms of tuning parameters for success with a given problem, but is the only one of the four variants tested that avoids computing (global or individual) nonlinear residual norms entirely. As such, one may implement it very efficiently, and `SS-FGPILU` may be very effective if the problems of interest are similar enough to leverage the same values of the input parameters.

Lastly, note that while the variants presented here do perform differently and may be best suited to different use cases, when they are able to successfully converge they tend to produce very similar performance in the associated Krylov subspace solver.

Error detection capability The proposed fault-tolerant variants of the FGPILU algorithm are designed not to detect every fault that occurs but rather to make the end user unaware of the negative convergence effects of any faults that do occur. Such a design choice has been made, in part, because some faults may have a negligible effect and because comprehensive error detection additional modification to the original FGPILU routine.

For example, while in the `CPA` variant (Algorithm 15), it is straightforward to define detection as a positive check on the progression of the global nonlinear residual norm, for the other variants it is not as simple. The success of Algorithm 18 is very closely related to the ability of the algorithm to detect the presence of a fault on the fine-grained level. Large faults tend to be easy to detect looking solely at changes in the individual nonlinear residual norms τ_{ij}'s and the FGPILU algorithm tends to converge naturally through faults that have a sufficiently small impact. However, detection of the more moderately sized faults is key to ensuring a high success rate and is related to the parameters γ and F (see Section 5.3.4 for their discussion).

5.4.4 RESULTS FOR NON-SYMMETRIC MATRICES

This section provides a set of results complementary to what was presented in Section 5.4.3, by examining problems that are more difficult to solve. The test problems that were used in this portion of the dissertation are intended to form a representative but not complete set of matrices that are harder to solve than the simpler SPD problems that have been utilized previously. The convergence of the fixed point iteration associated with the FGPILU algorithm displays good convergence with problems that are SPD [23], [24], [169]; however, solving fixed point iterations that feature nonlinear functionals (i.e., in Algorithm 11)

is often difficult. Developing the associated convergence theory, especially results that carry practical meaning, is also typically hard to accomplish (see for example: [59], [60]).

The test matrices used here come from a variety of sources. The first comes from the seminal work on the performance of incomplete LU factorization for indefinite matrices [51], `fs_760_3`. The next matrix comes from the domain of circuit simulation, `ecl32`, and has been studied previously [186], [187]. The last matrix comes from the set of 8 SPD matrices that were studied in Section 5.4.3, and is the matrix among those eight with the largest condition number (as estimated by `MATLAB`®'s `CONDEST` function); 'offshore'. Condition numbers for the 8 previously studied SPD problems range from 1.11e+03 to 2.24e+13. A brief summary of all three matrices is provided in Table 12.

Table 12: Characteristics of the matrices used: Column `Sym?` reflects the symmetry, `PD?` provides positive-definiteness, `Dim`—number of rows, and `Non-zeros`–number of non-zeros in each matrix.

Matrix Name	Abbr.	Sym?	PD?	`CONDEST`	Dim.	Non-zeros	Description
fs_760_3	FS	N	N	9.93E+19	760	5,816	chemical engineering
ecl32	ECL	N	N	9.41E+15	51,993	380,415	circuit simulation
OFFSHORE	OFF	Y	Y	2.24E+13	259,789	4,242,673	electric field diffusion

The matrices that are presented here attempt to give some indication as to the performance of the nonlinear fixed point iteration associated with the FGPILU algorithm with respect to matrices that are more challenging computationally than the problems that are featured in the majority of the previous work on the algorithm (i.e. [23], [24], [169], [170], [175]).

Lastly, it is important note that many other problems from both [51], [52], [183], and the domain of circuit simulation were considered; only about 7% of the problems studied were able to converge with the standard initial guess and no fundamental alterations to the matrix. While this percentage could be increased with a more careful analysis of each problem it is brought up here to emphasize the difficulty this fixed point algorithm can have

with non-symmetric and indefinite problems.

5.4.4.1 Convergence of the FGPILU algorithm

In these fault-free experiments, the convergence of the FGPILU algorithm is examined for three different levels (0,1, and 2) of the incomplete LU factorization (see [147] or [34] for a clear description of levels of incomplete LU factorizations), and three different values of α in the α-shift described in Section 5.2.1. Note that regardless of the ordering being utilized, all runs start with a symmetrically scaled matrix such that the entries on the diagonal are less than or equal to 1. As such, appropriate values for α range from 0 to 1 and in this dissertation three discrete values were selected from this range: 0, 0.5, 1.0.

More extreme values for α can help improve the convergence of the FGPILU algorithm by increasing the diagonal dominance of the matrix that the FGPILU algorithm is applied to, but this comes at the expense of preparing the preconditioner for a problem increasingly less related to the original problem. As an example, for the `OFFSHORE` problem with `AMD` ordering and symmetrical scaling, the FGPILU algorithm converges in a progressively smaller number of sweeps for increasing values of α. However, the overall performance of the Krylov subspace solver deteriorates. Details are provided in Table 13. Note that as α is increased, the number of sweeps required for the FGPILU algorithm to reduce the nonlinear residual norm below the desired tolerance is greatly decreased, but both the number of iterations and the time required for convergence of the Krylov subspace solver are greatly increased.

Table 13: Effects of increasing α for the `OFFSHORE` problem.

α	FGPILU Sweeps	Krylov solver iterations	Krylov solver time
0	24	30	24.8067
1	9	56	46.4995
10	5	144	130.0958

For each of the three matrices: four orderings were tested (`MC64, AMD, RCM`, and the

natural ordering), 3 level of ILU fill-in were tested (levels 0, 1, and 2), and 3 factors for α were used (0, 0.5, and 1.0). This leads to a total of 108 permutations to test. Of these 108 combinations, 84 (77.78%) led to a case were the FGPILU algorithm converged, but only 29 (26.85%) resulted in a successful GMRES solve of the entire linear system using a restart parameter of 50 and a tolerance of 1e-10. Details for those 29 cases are provided below in Table 14.

Table 14: Successful runs with their parameter combinations.

Matrix	**Ordering**	α	**ILU Level**	**Sweeps**	**Krylov Its.**	**Time (s)**
offshore	AMD	0	0	19	30	18
offshore	AMD	0.5	0,1,2	10,11,11	40,34,34	24,55,144
offshore	AMD	1	0,1,2	8,9,9	56,54,54	34,96,229
offshore	RCM	0	0	19	19	35
offshore	RCM	0.5	0,1,2	10,11,11	37,34,34	68,306,771
offshore	RCM	1	0,1,2	9,9,9	54,54,54	101,484,1226
offshore	Natural	0	0	22	22	84
offshore	Natural	0.5	0,1,2	11,12,12	38,34,34	146,312,695
offshore	Natural	1	0,1,2	9,10,10	54,54,54	210,491,1104
ecl32	AMD	0	2	15	127	104
ecl32	RCM	0	2	24	9	39
ecl32	Natural	0	2	18	11	16
fs_760_3	AMD	0	2	55	3	0.4
fs_760_3	RCM	0	1,2	52,63	2,2	0.4,0.4
fs_760_3	MC64	0	1	16	3	0.3
fs_760_3	Natural	0	1	16	3	0.3

In general, higher levels of fill are capable of producing better preconditioning factors [51], [52], but come at the cost of increased storage and computational costs. There is an inherent trade-off in using higher fill levels to produce incomplete factors that are closer to the full L and U factors that must be evaluated. A few other general observations:

- the two non-symmetric problems tend to perform better with smaller values of α and higher levels of fill-in allowed, and

- the level of ILU fill-in tends to not have as much of an impact on whether or not the problem can be solved when compared to the ordering or value for α, but affects the performance. In the results found here, the benefit of having more complete L and U factors from going to a higher fill-in level tends to be outweighed by the increased computational cost of the fixed point iteration associated with the FGPILU algorithm for a drastically larger number of elements.

As an example of the drastic increase in the number of non-zero elements for each of the matrices, consider the data in Table 15.

Table 15: Increase in non-zeros for different levels of ILU fill-in. The data in the first two rows is given in millions (m) of non-zero elements, and the last row specifies thousands (k) of non-zero elements.

Matrix	**nnz(ILU-0)**	**nnz(ILU-1)**	**nnz(ILU-2)**
offshore	4.5m	10.0m	21.7m
ecl32	0.4m	1.0m	2.0m
fs_760_3	6.5k	17.6k	32.3k

5.4.4.2 Resilience of the FGPILU algorithm

The experiments conducted in this section reflect the resilience of the FGPILU algorithm with respect to transient soft faults for this section set of problems. The only variant considered for this set of experiments is the `CPA-FGPILU` variant detailed in Algorithm 15. The reason for this selection is that the success of the FGPILU algorithm for these problems in a fault-free case was low enough that only the most successful variant of the FGPILU algorithm was considered for this problem set.

Further, in evaluating the resilience of the FGPILU algorithm, only combinations of ordering, ILU-level, and α from Section 5.3.5 that were successful in the fault-free scenario have been selected for experimentation. A single set of parameters for the fault detection

check in Algorithm 15, $\tau^{(sweep)} > \gamma \cdot \tau^{(sweep-r)}$, was used. In these experiments, γ and r were set to one so that a strict check on the monotonicity of the nonlinear residual norm is performed after every sweep. For SPD problems, this level of check may be unnecessary [169], [170], but this provides the maximum level of protection for the FGPILU algorithm and provides a measure of how effective this check can be for the more difficult problems under investigation in this dissertation.

A summary of the data found in these experiments is provided in Table 16, which depicts the percentage of runs that succeeded—resulted in a successful linear system solve—subject to faults (column `Scenario`), when no fault tolerance (column `NoFT`) and the checkpointing FGPILU variant (column `CPA`) were employed, respectively. Three ratios of the results with `CP` and `NoFT` are shown in Table 16 as `Timing`, `Sweeps`, and `Its`, defining the timing increase, reduction in the total number of sweeps needed, and the change in the GMRES iterations, respectively. As an alternative representation, a visual representation of portions of this data is provided in Fig. 38.

Table 16: Solver performance using FGPILU with no fault tolerance (`NoFT`) and checkpointing (`CPA`).

Scenario	Success Rate (NoFT)	Success Rate (CPA)	Timing Ratio	Sweeps Ratio	Its. Ratio
Total	46.65%	100.00%	1.02	0.63	1.01
Small fault	88.59%	100.00%	1.03	0.69	1.03
Medium fault	42.94%	100.00%	1.01	0.48	1.00
Large fault	14.71%	100.00%	1.00	0.73	0.99

The checkpointing algorithm mitigates well the potential impact of a fault. Note that the largest benefit comes from correcting the impact of a large fault. Smaller faults—which cause

effects similar to those produced by bit flips in a less significant bit of the mantissa—tend to be corrected naturally by the iterative nature of the fixed point iteration.

Another important factor in comparing any fault tolerance methods is quantifying how much overhead they introduce. Due to the non-deterministic block-asynchronous nature of the GPU implementation of the FGPILU algorithm in the absence of faults and the inherent randomness involved in the fault model utilized in this dissertation, it is difficult to compare individual cases. However, comparing runs utilizing the same parameters over all cases where both the fault-free variants and the checkpointing variant solved the linear system successfully, there is about a 2% increase in the time required to reach a solution in order to provide fault tolerance to the FGPILU algorithm using this methodology. There is more of an impact on cases with small faults since it is often possible for the iterative nature of the algorithm to correct the impact of a sufficiently small fault. Note that varying the parameters γ and r that determine the frequency and strictness of the check could change both the efficiency and efficacy of the checkpointing variant of the FGPILU.

5.5 SUMMARY

This chapter of the dissertation has examined the impact of soft faults on the FGPILU algorithm, and proposed several variants to remedy the impact. Soft faults which are undetected by the original FGPILU algorithm have the potential to cause severe disruption to the preconditioning routine; and, even if the FGPILU algorithm reports successful convergence, the solver that uses the incomplete factors generated by the FGPILU algorithm as a preconditioner may be affected. The ability of the FGPILU algorithm to tolerate and mitigate certain soft faults arising in the construction of L and U factors has been explored using several algorithm variants and two distinct ways of modeling the impact of a soft fault. The results shown here indicate that any undetected soft fault that affects multiple components will be significantly more compromising for the FGPILU algorithm. The variants of the FGPILU algorithm developed in this chapter have provided mechanizations that supply a

measure of resilience to the procedure and allow it to converge successfully. Additionally, the techniques discussed offer an abundance of methods that can be used to create further variants that may provide better performance and/or resilience for specific problem domains.

This chapter has presented some experiments and analysis concerning the convergence and resilience of the FGPILU factorization with respect to both symmetric and non-symmetric problems. The use of fine-grained preconditioning algorithms is increasing in general, and as new fine-grained preconditioning algorithms are developed, some may use the FGPILU algorithm as a building block and require the FGPILU algorithm to execute successfully inside of a more complex preconditioning scheme. In these cases, it may be critical to have the FGPILU algorithm converge more completely, and the work presented here could be used as a starting point towards ensuring that can happen successfully even when computing faults occur.

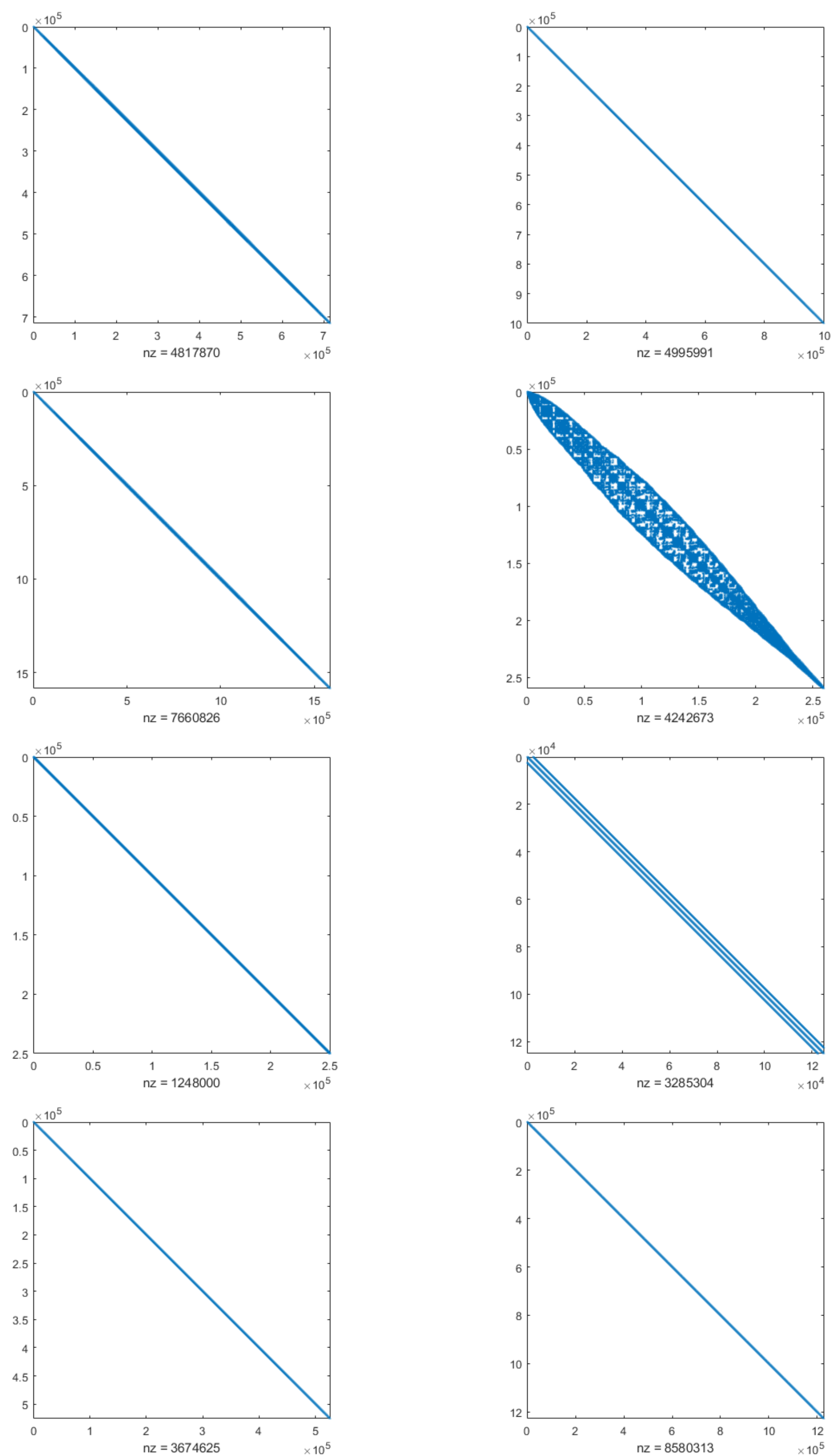


Fig. 32: Sparsity plots showing the location of all non-zeros for each of the 8 matrices with the Reverse Cuthill-Mckee (RCM) reordering applied that were considered in the first set of experiments.

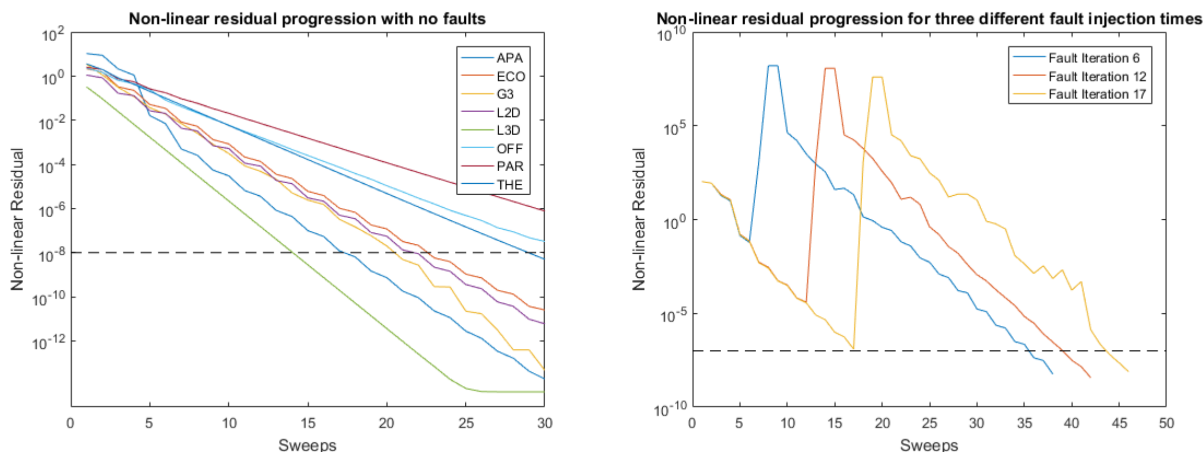


Fig. 33: The progression of the nonlinear residual for 30 sweeps of a typical fault-free run for each of the 8 test problems (left). The progression of the nonlinear residual for the Apache2 test problem for three different fault injection times and fault size in the $(-1, 1)$ range (right). The horizontal dashed line is indicated the FGPILU convergence tolerance of 10^{-8}.

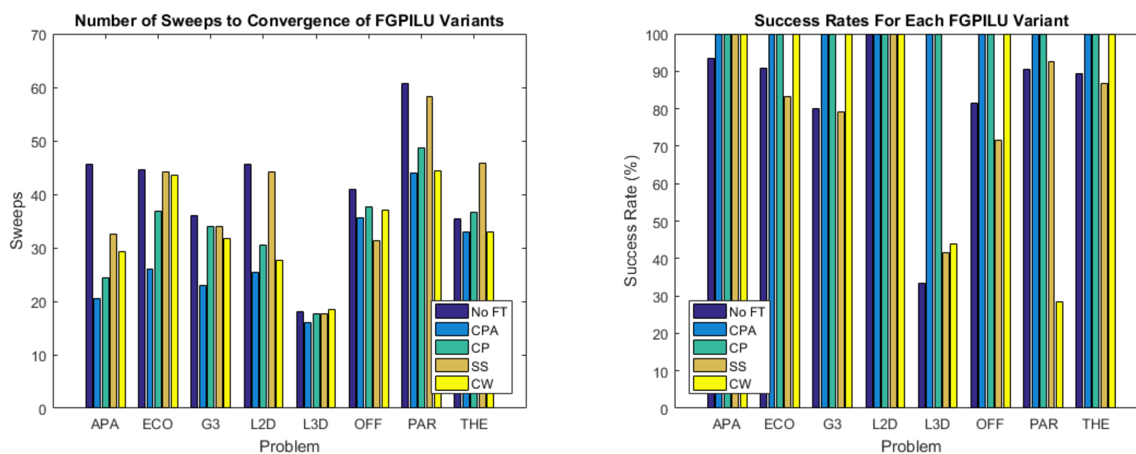


Fig. 34: For perturbation-based faults (PBSFM): the number of sweeps required for convergence for each of the 8 test problems (left). The percentage of runs that produced a preconditioner that corresponded to a successful PCG solve (right).

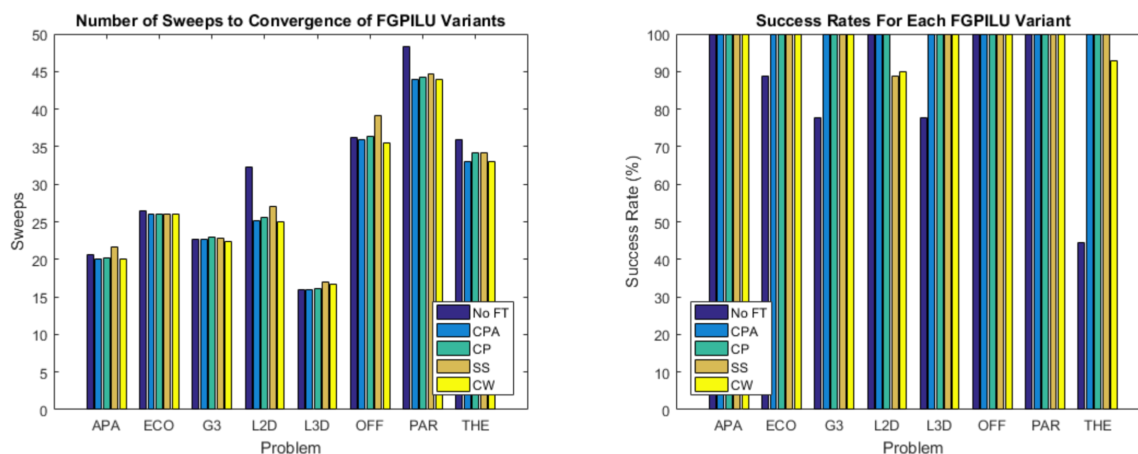


Fig. 35: For bit-flip (BF) faults: the number of sweeps required for convergence for each of the 8 test problems (left). The percentage of runs that produced a preconditioner that corresponded to a successful PCG solve (right).

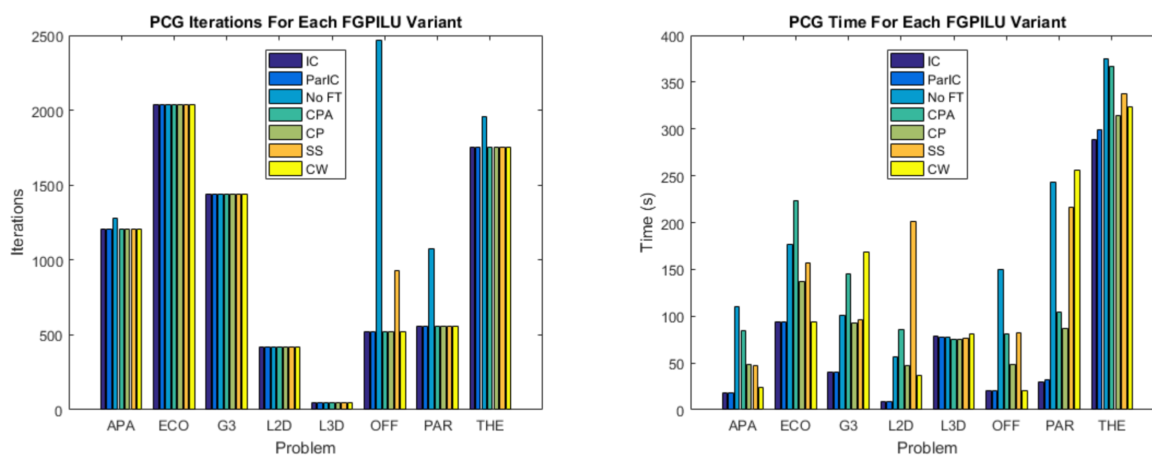


Fig. 36: For perturbation-based faults (PBSFM): the number of iterations required for successful PCG solves for each of the 8 test problems (left). The time required for successful PCG solves for each of the 8 test problems (right).

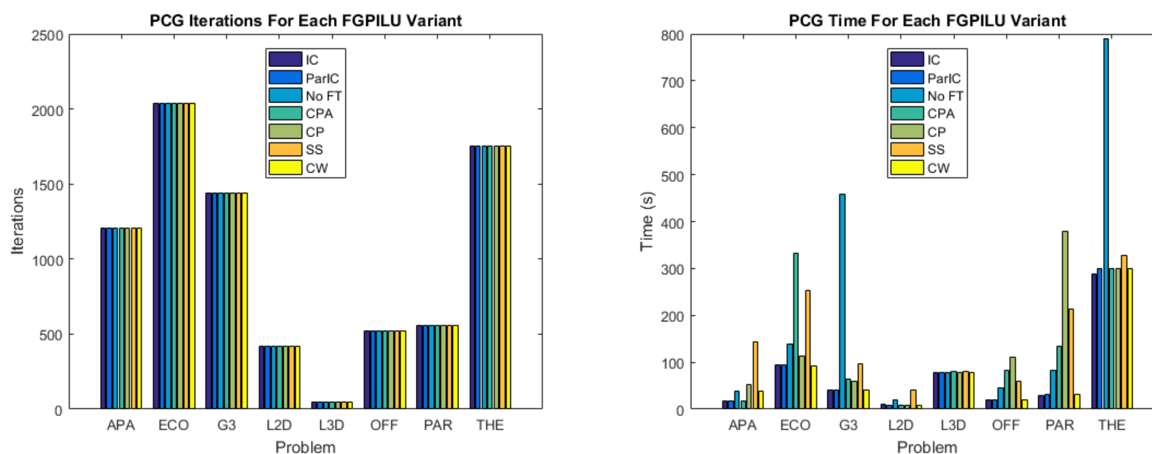


Fig. 37: For bit-flip (BF) faults: the number of iterations required for successful PCG solves for each of the 8 test problems (left). The time required for successful PCG solves for each of the 8 test problems (right).

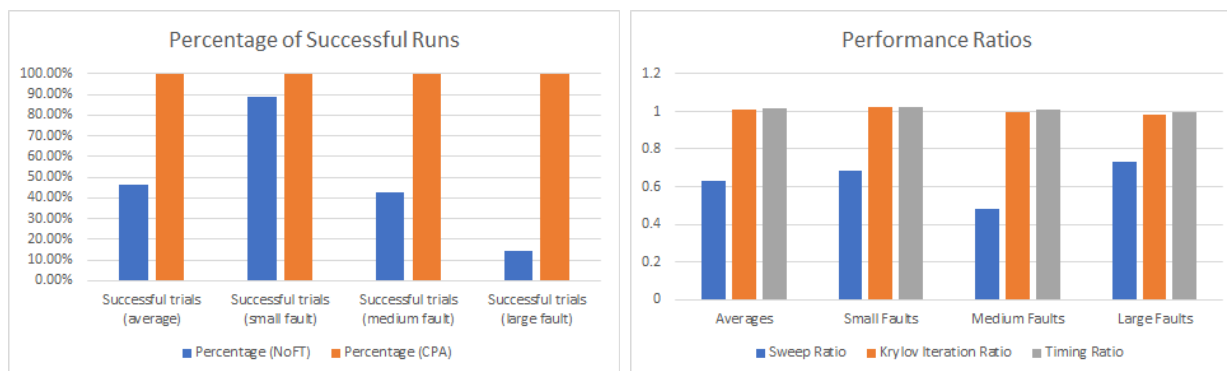


Fig. 38: Percentage of successful runs for no fault tolerance (NoFT) and checkpointing (CPA) (left), ratios showing relative performance of the checkpointing variant to the nominal FG-PILU algorithm (right).

CHAPTER 6

FRAMEWORK FOR MODELING AND ANALYSIS

When constructing any algorithm that is designed to be used on future hardware, it is important to take into account as many properties of the potential systems as possible. Since the variability of future hardware, both individual processor performance and the make-up of the HPC system itself, contribute directly to the amount of asynchronism that can be expected, it is necessary to keep in mind a range of potential performance characteristics. The amount of delay between the fastest and slowest processors (see Chapter 3) can have an impact on the convergence rate of different algorithms, and it is possible to develop an intuitive understanding of reasonable delays through simulation.

In this chapter, a simulation framework[1] is proposed and tested to examine the potential benefit of asynchronous iteration for various HPC accelerator architectures, which typically admit different granularities of computations. Additionally, an example of a case study using the simulation framework is presented to examine the efficacy of different checkpointing schemes for asynchronous relaxation methods. The simulation framework discussed is capable of simulating the potential performance of a variety of asynchronous iterative methods for a range of difference performance parameters. The construction of the framework is modular which allows the performance of new algorithms (and variants) to be examined. Some of the results presented here have been previously published [168]. Related work creating predictive models for the performance of iterative methods is captured in [188], [189] but is not included in this dissertation.

The structure of this chapter is organized as follows: Section 6.1 describes the shared memory experiments, and introduces the simulation framework. The subsections contained

[1] Send requests for source code to `evan.coleman1@navy.mil`

inside of Section 6.1 detail results and validation efforts for two different solver implementations. Next, Section 6.2 provides an example of a use case that shows how to extend the simulation framework to experiment with fault tolerant algorithms, Section 6.3 provides some tests using this extension, and Section 6.4 concludes.

6.1 DESIGN OF SIMULATION FRAMEWORK

The simulation framework proposed here is designed to simulate the performance of an asynchronous iterative method operating on multiple computing elements using a single processing element. In this simulation framework, the emphasis is on fixed-point iterations

$$x = G(x), \tag{133}$$

for some $x \in \mathbb{R}^n$. In the framework, certain components are assigned (possibly distinct) times for performing an update to their components, and the effects of various delay structures can be examined.

The development of the present computational framework may be described by the flow diagram given in Fig. 39, which is typical for computation frameworks, except for the third *Timing Distributions* stage, which is used for the proposed framework as described below along with other stages from Fig. 39. A mathematical formulation of a problem (e.g., as a set of equations) is presented first (*Mathematical Model* in Fig. 39). The mathematical model is then implemented in an HPC environment (*Parallel Implementation* stage). Timing and algorithm-performance data (e.g., iterations to convergence) are collected from parallel executions on a subset of configurations and problem sizes, such that, in the proposed framework, timing distributions may be constructed (*Timing Distributions* stage) and used to simulate the performance of the mathematical model for target configurations and requirements. Since such simulations are faster and less-cumbersome to set-up, they allow for easy experimenting with variations of the underlying mathematical model, parallel

implementation type and environment, or, eventually, in showing the expected performance.

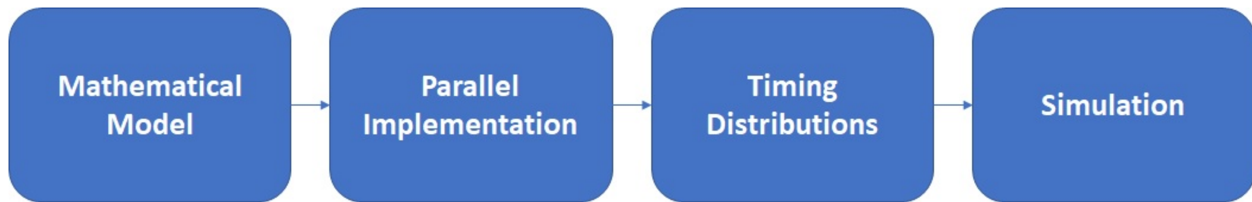


Fig. 39: Stages in the proposed framework development.

The simulation framework developed here works to simulate the performance of generic asynchronous relaxation methods in shared memory environments. The simulation framework can then be modified to reflect changes in the environment, or it can be utilized to demonstrate the effectiveness of algorithmic modifications.

As a simple example, take $n = 2$. Then $x = (x_1, x_2) \in \mathbb{R}^2$ and, using the terminology of Section 2.1.1.2,

$$x_1 = G_1(x) = G_1(x_1, x_2), \tag{134}$$

$$x_2 = G_2(x) = G_2(x_1, x_2). \tag{135}$$

In a traditional fully synchronous environment, both functions, G_1 and G_2, would be called simultaneously and no subsequent calls would be executed until both functions had returned and *synchronized* all results. In a fully asynchronous environment, both functions would be allowed to execute again immediately upon their own return, leading to a case where one of x_1 or x_2 may be updated more frequently than the other. Per Definition 1, both functions use the latest values of all components x that are available to them when the function call is initiated. For instance, if the processing element that was assigned to update the component x_1 was ten times as fast as the processing element assigned to update x_2, then in the amount of time needed to update x_2 once, the component x_1 will have been updated ten times, and

when G_2 is called for the second time it will be called using the latest component of x_1 (which has been updated 10 times), and the latest component of x_2 (which has only been updated once).

A block diagram showing the flow of the simulation framework is provided in Fig. 40. The framework models the performance of methods that solve the linear system

$$Ax = b \tag{136}$$

using relaxation methods in either a synchronous or asynchronous manner.

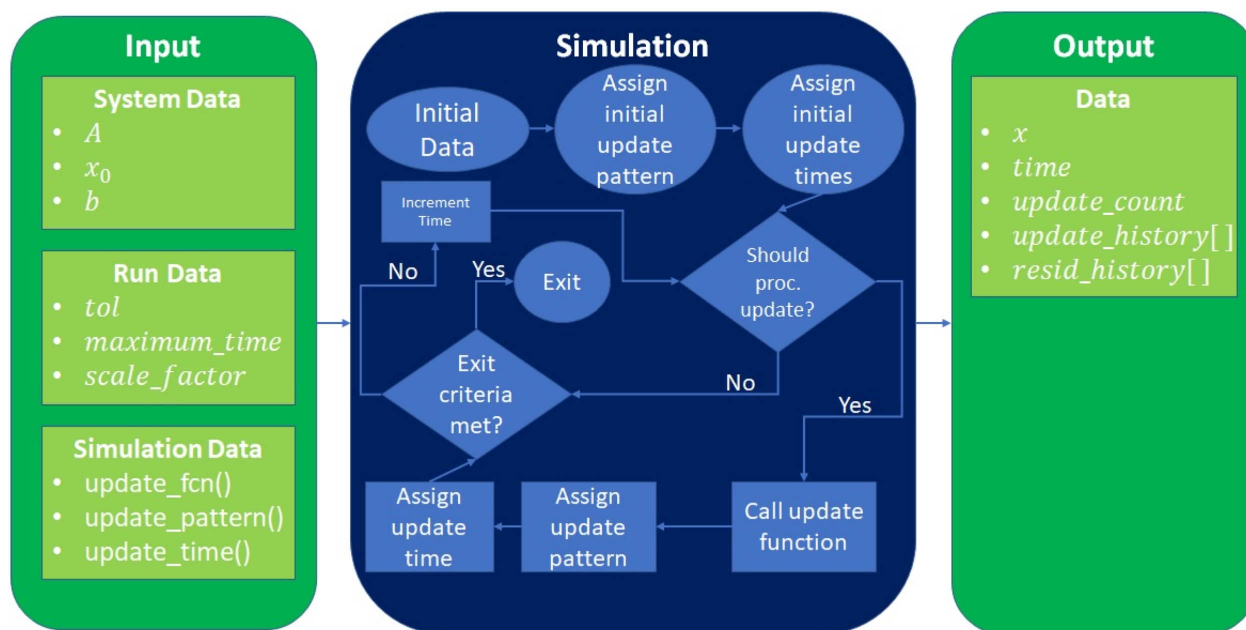


Fig. 40: Block diagram of the simulation framework.

The simulation requires as input the matrix A, the right hand side b and an initial guess at the solution, x_0. The important pieces of the simulation are all passed as functions to the tool. There are three functions required:

1. An update function that specifies how to perform the relaxation. A common technique

for this is given by Eq. (92). It is certainly possible to modify this equation to obtain different updates, as described, e.g., in [34].

2. An update pattern function that determines which elements of the matrix A are assigned to each simulated processor. A common approach for this assignment is to evenly divide the work among all of the available processors; however, other patterns are also possible. For example, the use of randomization in the solution of linear systems via relaxation methods has gained some popularity in the fields of optimization and machine learning (see [105] and references therein) and update patterns such as this are easy to implement inside of this framework.

3. An update time function that captures the empirical information that was captured from parallel performance runs on the HPC hardware. This function will typically be used to sample from the timing distribution that was generated beforehand. Note that, since each simulated processor makes calls to this function independently, the simulated performance will be asynchronous so long as the function returns different values upon different calls. Defining an update time function that has constant return (or constant return for every processor) provides a means to show synchronous performance.

By varying the three functions that are passed to the framework, not only can the HPC performance be predicted by making changes to the update time function, but various modifications to the basic algorithm can be quickly and easily compared in a manner that reflects real world asynchronous performance. With the renewed research interest in asynchronous iterative methods that perform relaxation updates, oftentimes performance between new variants and existing algorithms is *only* compared in simple synchronous experiments; the simulation framework proposed here allows for a more meaningful comparison between methods that does not require development of parallel implementations of all the methods or algorithm variations that are involved.

The simulation framework requires some data that specifies parameters concerning the

particular run of the simulation such as the desired tolerance, the number of processors to simulate, and a computational scale factor. The framework itself is developed in `MATLAB`®and the three required functions are passed as function handles.

The simulation itself (see *Simulation* block in Fig. 40) progresses by reading in the user provided input data, assigning an initial update pattern and time to each processor, and then beginning the main loop. Inside of the main loop, the time increments and a check is performed to see if the current time matches with the scheduled update time for any of the processors, if so, the update function is called and then a time for the next update is assigned to the processor that just updated and (if desired) the update pattern for the current processor is changed. After this, a check is performed on the size of the residual to determine if the exit criteria is met before the time is incremented again and the loop starts over. A pseudocode representation of the simulation framework for simulated asynchronous Jacobi is given in Algorithm 19.

Algorithm 19: Asynchronous Jacobi simulation

Input: $a_{ij} \in A$, initial guess for x_0, a number of processing elements p, an input random number distribution

Output: Solution vector x

1 Assign processor update times, $\tau_1, \tau_2, \ldots, \tau_p$, by sampling from an appropriate random number distribution
2 Assign elements $x_i \in x$ to each simulated processing element
3 **for** $t = 1, 2, \ldots$ *until convergence* **do**
4   **for** *each processing element* P_l **do**
5   **if** $\tau_l = t$ **then**
6   **for** *each element* $x_i \in x$ *assigned to* P_l **do**
7                 $x_i = \frac{-1}{a_{ii}} \left[\sum_{j \neq i} a_{ij} x_j - b_i \right]$
8   Retrieve a new update time τ_l by sampling from the input distribution
9   Calculate the residual as in Eq. (138) and check termination conditions

In Algorithm 19, a given update time τ_l will often not be sampled as an integer. The simulation adjusts for this by scaling the number that is sampled by the appropriate order

of magnitude, adjusting the maximum value allowed for t accordingly, and then scaling back the final time calculated by the simulation. For example, if the desired time precision is hundredths of a second, and the time resulting for the first sampling of τ_l was 1.234 seconds, then the simulation would perform the following steps:

1. $\tau_l^{\texttt{new}} = s \times \tau_l^{\texttt{old}}$
2. $t_{\texttt{max}}^{\texttt{new}} = s \times t_{\texttt{max}}^{\texttt{old}}$
3. $t_{\texttt{final}}^{\texttt{new}} = (1/s) \times t_{\texttt{final}}^{\texttt{old}}$.

where s is the "scale_factor" defined in the block diagram given by Fig. 40. For example, if the desired precision is hundredths of a second, $s = 10^2$, and the sampled value τ_l becomes

$$\tau_l = 1.234 - \text{initial sample}$$
$$\tau_l = 123.4 - \text{apply scale factor}$$
$$\tau_l = 123 - \text{round to nearest integer.}$$

Inside of the simulation framework, time is abstracted away to "units of time", and then the final time is scaled back into the appropriate units. This allows the framework to be adapted to future HPC environments, as well as examining the impact of the standard variance of single core performance on multi-core hardware elements if the method that is used is tuned to be completely asynchronous.

6.1.1 SAMPLE USE-CASES FOR THE FRAMEWORK

Let the matrix A result from a simple two dimensional finite-difference discretization of the Laplacian over a 10×10 grid, resulting in a 100×100 matrix with an average of 4.6 non-zero entries per row. Once the PDE is discretized over the desired grid, the linear system

$$Ax = b \tag{137}$$

is set up to be solved for a random right-hand side b that represents the desired boundary conditions. All problems considered in this chapter use Dirichlet boundary conditions. For the examples in this particular subsection, the right-hand side is generated by taking each component sampled as a uniform random number between -0.5 and 0.5, and then normalizing the resultant vector. The iterative Jacobi method proceeds until the residual

$$r = b - Ax \tag{138}$$

is reduced past some desired threshold.

To begin with, an example of nominal performance of the solution of the two dimensional Laplacian in a synchronous environment is provided by Fig. 41. Next, consider the same

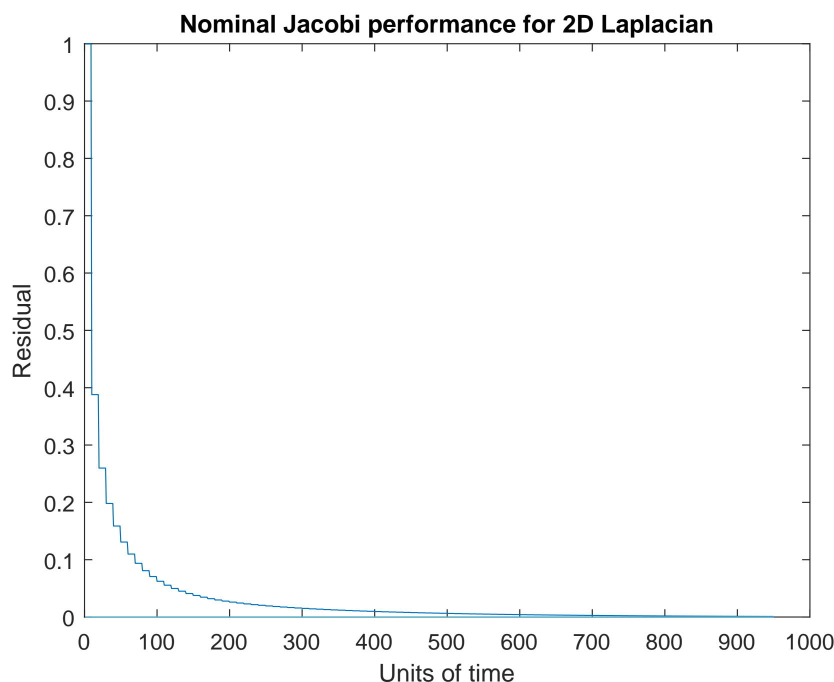


Fig. 41: Example of nominal performance of the synchronous Jacobi iteration.

problem from above, but in two slightly more complicated scenarios. In Fig. 42 one of the

ten processors involved in updating blocks of components of x is provided updates more slowly than the other processors. This could reflect the scenario where updates are either performed synchronously or asynchronously, where the effect of variance in performance is negligible, and a single processor has degraded performance. This can also be viewed as a look at the impact of asynchronous behavior on the Jacobi algorithm. Each curve shows the progression of the (global) residual subject to having a single slower processor with different degrees of slowdown (from zero to 11x).

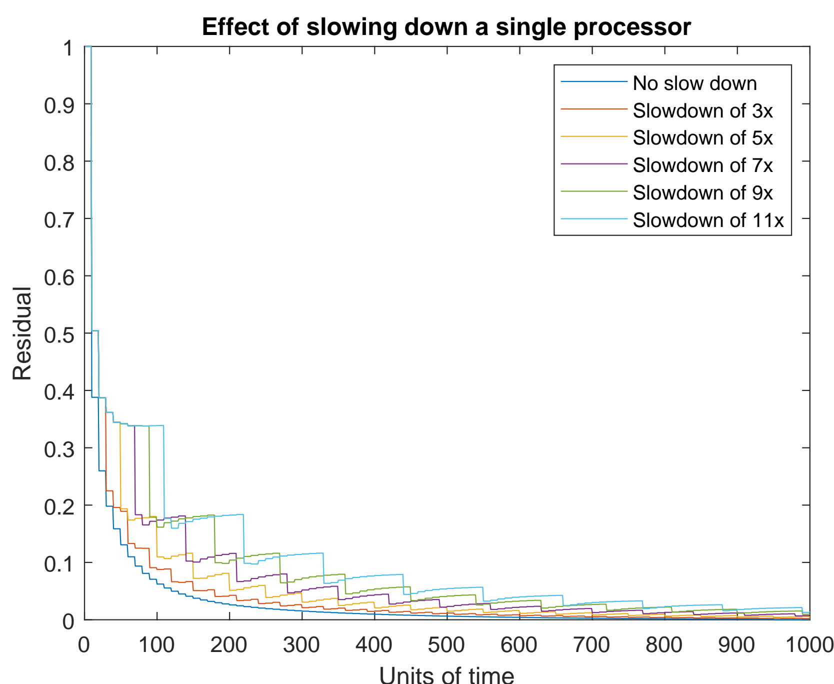


Fig. 42: Example of experiments within the simulation framework. Each line shows the effect of slowing down a single processor to some factor of the (synchronous) performance of the other processors.

In Fig. 43 the processor updates are not restricted to occur synchronously. Instead, the processors are assumed to have similar performance and perform their updates in time $t_i \sim N(\mu, \sigma^2)$, where the mean is set to 10 units of time and the variance is different for each curve depicted in the plot. An increase in the variance of processor performance, regardless

of the timing distribution, could come about for a variety of reasons; an example of a scenario in the future could be having chips with more cores and lower voltage that are designed to address the challenges in creating very large scale HPC environments.

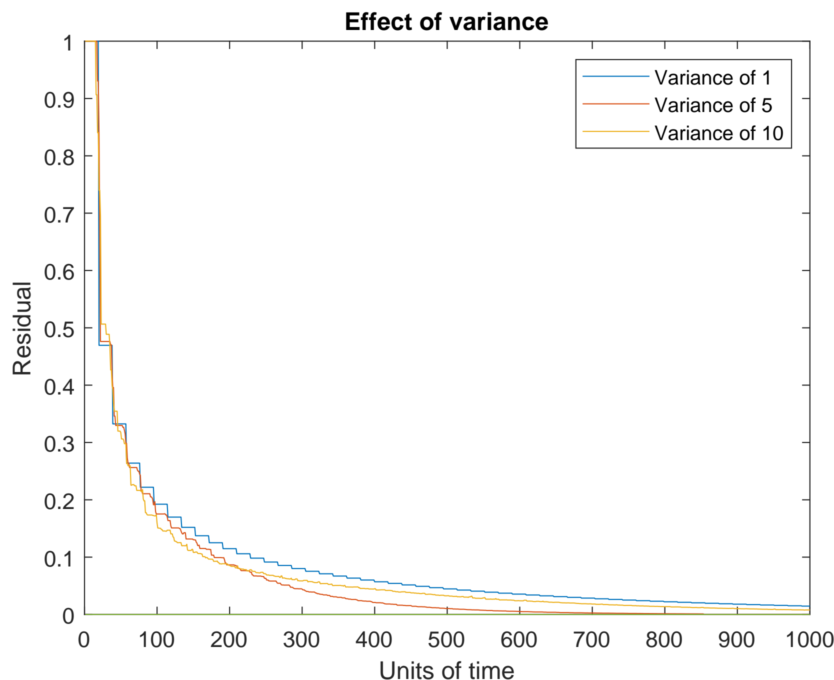


Fig. 43: Example of experiments within the simulation framework. Each line shows the effect of increasing the variance in processor performance from 1 to 5 to 10.

6.1.2 ASYNCHRONOUS JACOBI IMPLEMENTATIONS FOR THE FRAMEWORK

Figures 42 and 43 show relative differences in compute times among shared-memory computing elements for a specific problem and a specific asynchronous iterative method. A more general simulation framework, which can be used for modeling and testing any synchronous or asynchronous iterative relaxation method, is presented here. Baseline, non-resilient method behavior may be reproduced in the framework; further, the user may also

investigate fault injection and checkpointing.

The user decomposes the method according to the input parameters required by the simulation framework. The update function that performs the relaxation has an associated operational time, both of which are defined by the user. Functionality within the relaxation may be isolated into discrete operations with corresponding time information; the level of granularity is decided by the user. For example, time to complete an operation in the simulation framework may be modeled with a probability density function derived from empirical data. To model time to perform specific operations or calculations during method execution, data is collected from the application during execution. In the implementation code, operations are enclosed within calls to time functions, which measure time to perform the operations. In this work the OpenMP®library function `omp_get_wtime()` is used to measure wall time. For HPC implementations that use MPI, `MPI_Wtime()` may be used to measure wall time. Fine-grained operations in the code should not overlap such that measurements overlap, i.e. for one operation, do not measure time function calls of another operation. After taking sufficient measurements, an operation is modeled by fitting a probability density function to a normalized histogram of the time data. This function may be included as part of the input to the framework. Note that when comparing simulated run times with HPC run times, it may be preferable to use an unmodified version of the HPC implementation code that does not have time function falls and mechanisms for storing or printing times. These functions and activities may increase run time and provide an inaccurate metric for comparison.

This section describes two asynchronous relaxation method implementations and two corresponding use cases of the simulation framework. For both implementations, the test problem is a two dimensional discretization of the Laplacian where the right-hand side is initialized with Dirichlet boundary conditions. Both implementations use OpenMP®for shared-memory parallelism and are executed on the shared-memory computing platform nicknamed

Rulfo, which is an Intel Xeon Phi™Knight's Landing[2] having 7210 model processor with 64 cores, Each core may optimally execute 4 threads for 256 threads total, and runs at 1.30 GHz. The simulation framework and experiments were implemented in MATLAB®R2018a, while the Jacobi implementations were written in C/C++ using the Intel®C compiler version 17.04 and OpenMP®version 4.5.

6.1.3 IMPLEMENTATION 1: GENERAL JACOBI SOLVER

In this case, Laplacian is represented mathematically by a sparse matrix, which is solved by an asynchronous general Jacobi method. The Laplacian is generated over a 100×100 grid resulting in a matrix of size $10,000 \times 10,000$ with $49,600$ non-zeros with an average of 4.96 non-zeros per row. The vector b from the resulting linear system,

$$Ax = b, \tag{139}$$

is initialized such that the final solution vector has $x_i = 1$ for all i. The initial guess $x^{(0)}$ is all zeros.

In this implementation, all threads but one perform relaxations on assigned components, and a dedicated thread computes the global residual norm value $b - Ax^{(t)}$ that determines satisfactory convergence. Each thread retrieves the data it needs from shared memory, performs the necessary computations, and, in the case of the relaxation threads, writes the result back to shared memory. Synchronous shared-memory implementations of all classes of algorithms commonly use mutex locks to avoid race conditions with read and write operations. However, this type of asynchronous relaxation method may be less dependent on these safeguards for two reasons: (1) iterative methods can correct some errors with more iterations, if necessary, and (2) threads executing operations in asynchronous iterative methods are more likely to be at different stages of the iterative cycle, meaning fewer threads

[2]Rulfo is a part of computing resources of the Department of Modeling, Simulation and Visualization Engineering at Old Dominion University.

may be writing to and reading from the same memory location concurrently. This general Jacobi solver has two varieties: (a) SAFE which uses mutex locks to avoid race conditions, and (b) RACE which permits race conditions. SAFE uses OpenMP®locks to copy $x^{(t)}$ safely from shared memory and to update $x^{(t+1)}$. Pseudocode for this process is given in Algorithm 20, where bold upper-case text indicates that OpenMP®locks are employed. The algorithm for RACE is identical to Algorithm 20, with the exception that locks are omitted.

Figure 44 compares SAFE and RACE calculation times and number of iterations. Calculation times and average iteration counts are similar for thread counts up to 81, but behavior diverges beyond that. For thread counts 101 through 501, RACE requires more iterations, perhaps to compensate for threads reading and computing with inaccurate x vectors. Despite this, Fig. 44a shows that RACE is still quicker for the largest thread counts, perhaps because threads do not use locks to access data and eliminate that overhead cost. Figure 44a also shows that perhaps locks are not too costly for intermediate thread counts 101, 201, and 251, where SAFE outperforms RACE in terms of calculation time.

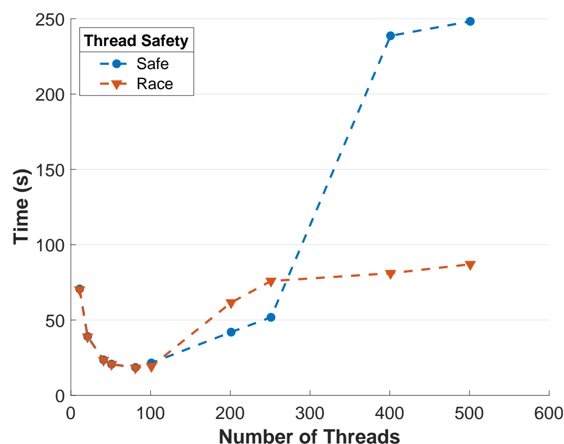
Algorithm 20: OpenMP®Implementation 1 (a) SAFE

Input: $a_{ij} \in A$, b, initial guess for X_0, n processing elements p
Output: Solution vector X

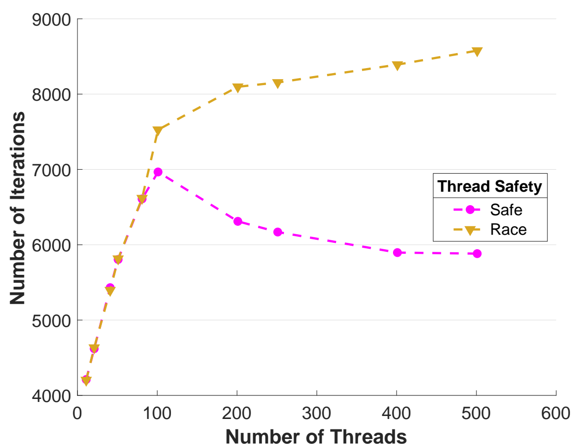
1 Assign elements $X_i \in X$ to $n-1$ processing elements, i $= [\alpha, \omega]$
2 **for** ***parallel*** *each processing element in* $p_1 \ldots p_n$ **do**
3   **while** *residual norm* $>$ *tolerance* **do**
4   **COPY** global $X^{(t)}$ from shared memory to local $x^{(t)}$
5   **if** p_1 **then**
6   Compute residual norm $||b - Ax||_2$
7   **else if** $p_2 \ldots p_n$ **then**
8   **for** x *index* $i = \alpha \ldots \omega$ **do**
9   Compute $x_i^{(t+1)} = \frac{-1}{a_{ii}} \left[\sum_{j \neq i} a_{ij} x_j^{(t)} - b_i \right]$
10   **UPDATE** $X_i^{(t+1)}$ in shared memory with $x_i^{(t+1)}$ for all i belonging to processing element

---

Both SAFE and RACE were executed over several trials and varying thread counts on the



(a) Calculation time



(b) Number of iterations, average per thread

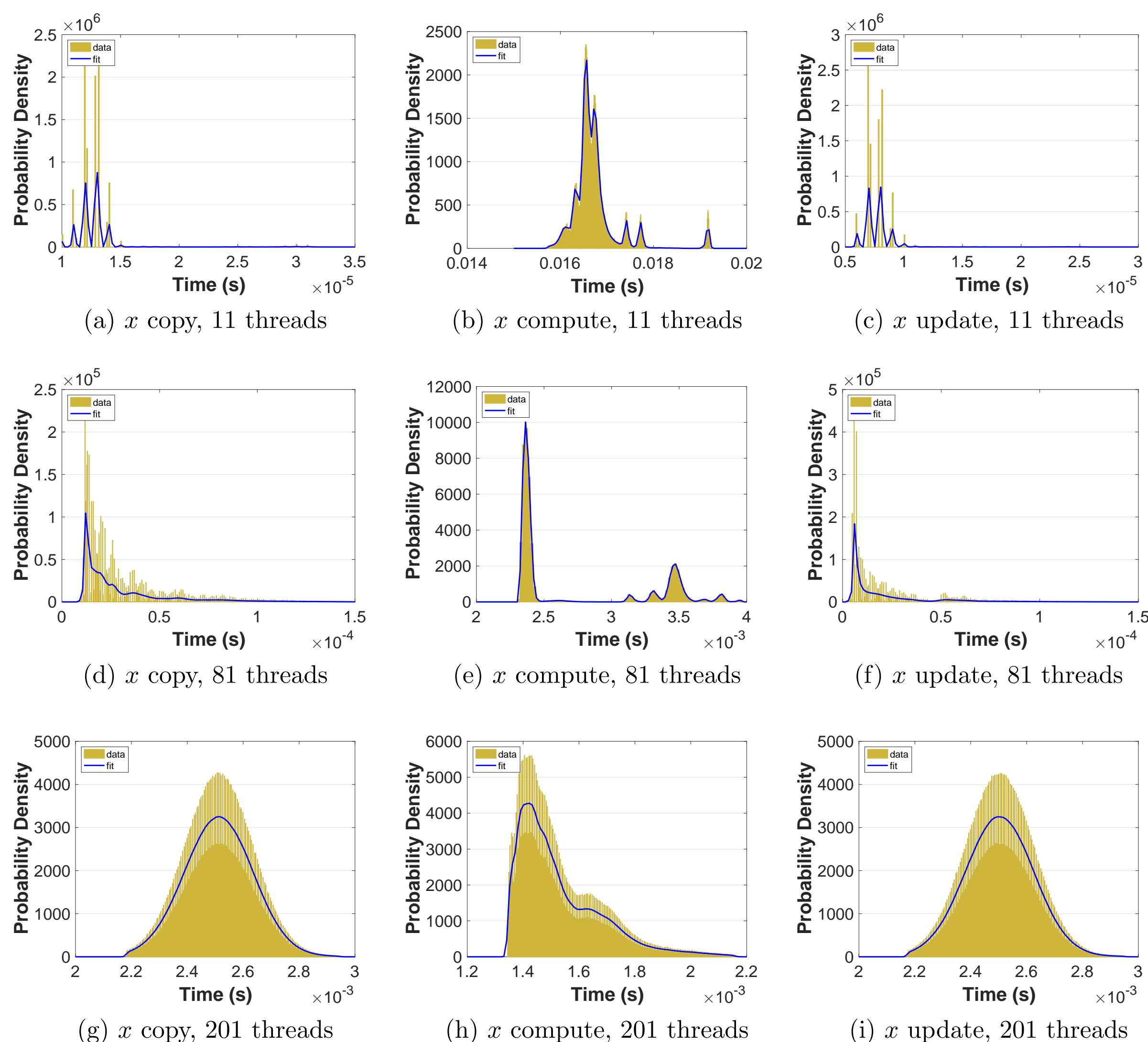
Fig. 44: Performance variations between SAFE and RACE as a function of thread count.

experimental HPC platform. For each trial, the times for a thread to access the solution in shared memory (Line 4 of Algorithm 20), compute the relaxation for the rows assigned to it (Line 9), and to update the solution in shared memory (Line 10) were captured. This data was used to generate `MATLAB`®kernel probability density functions for modeling the amount of time a thread takes to complete a copy, compute, or update operation. These distributions may be used in the simulation framework as an input parameter, for the generation of random variables corresponding to key operational times in the HPC architecture. Algorithm 19 demonstrates the use of a time distribution in the framework. Thread counts of 11, 21, 41,

81, 101, 201, 251, and 401 were used to collect data for the generation of distributions, some of which are in Fig. 45 and Fig. 46 For 201 threads, SAFE in Fig. 45d and Fig. 45f shows the tendency of locks to stratify copy and update times, compared with RACE in Fig. 46d and Fig. 46f, which are less uniform. These findings are mirrored in Table 17, which provides mean times for each of the three operations that were benchmarked in this implementation, for SAFE and RACE. RACE copy and update times are slightly or significantly quicker than comparable SAFE times. Compute times typically dominate total iteration time, except for SAFE copy and update times for threads 201, 251, and 401. Table 17 shows that increasing the number of threads decreases RACE copy, compute, and update times until cores are sufficiently over-subscribed: at 201 threads, these operations have become significantly more costly, as compared with 101 threads. This cost may be attributed to thread context switching. Compute times for SAFE do not increase with higher thread counts because thread behavior is controlled controlled explicitly using locks. These statistics can be used to validate the performance of the time distributions, so that the framework provides results comparable to the HPC hardware.

Table 17: Mean times for copy, compute, and update operations.

Threads	SAFE			RACE		
	Copy ($10^{-5}s$)	Compute ($10^{-4}s$)	Update ($10^{-6}s$)	Copy ($10^{-5}s$)	Compute ($10^{-4}s$)	Update ($10^{-6}s$)
11	1.28	167	7.76	1.15	167	2.79
21	1.31	84.3	6.98	1.17	83.6	1.96
41	1.38	43.0	7.09	1.23	43.1	1.63
81	2.98	27.3	20.6	1.43	27.2	1.79
101	36.7	23.4	357	1.64	25.2	1.79
201	251	15.3	2500	11.8	74.3	4.33
251	345	13.3	3440	16.6	90.9	4.55
401	1880	8.23	18 700	20.2	91.6	4.52



(a) x copy, 11 threads (b) x compute, 11 threads (c) x update, 11 threads

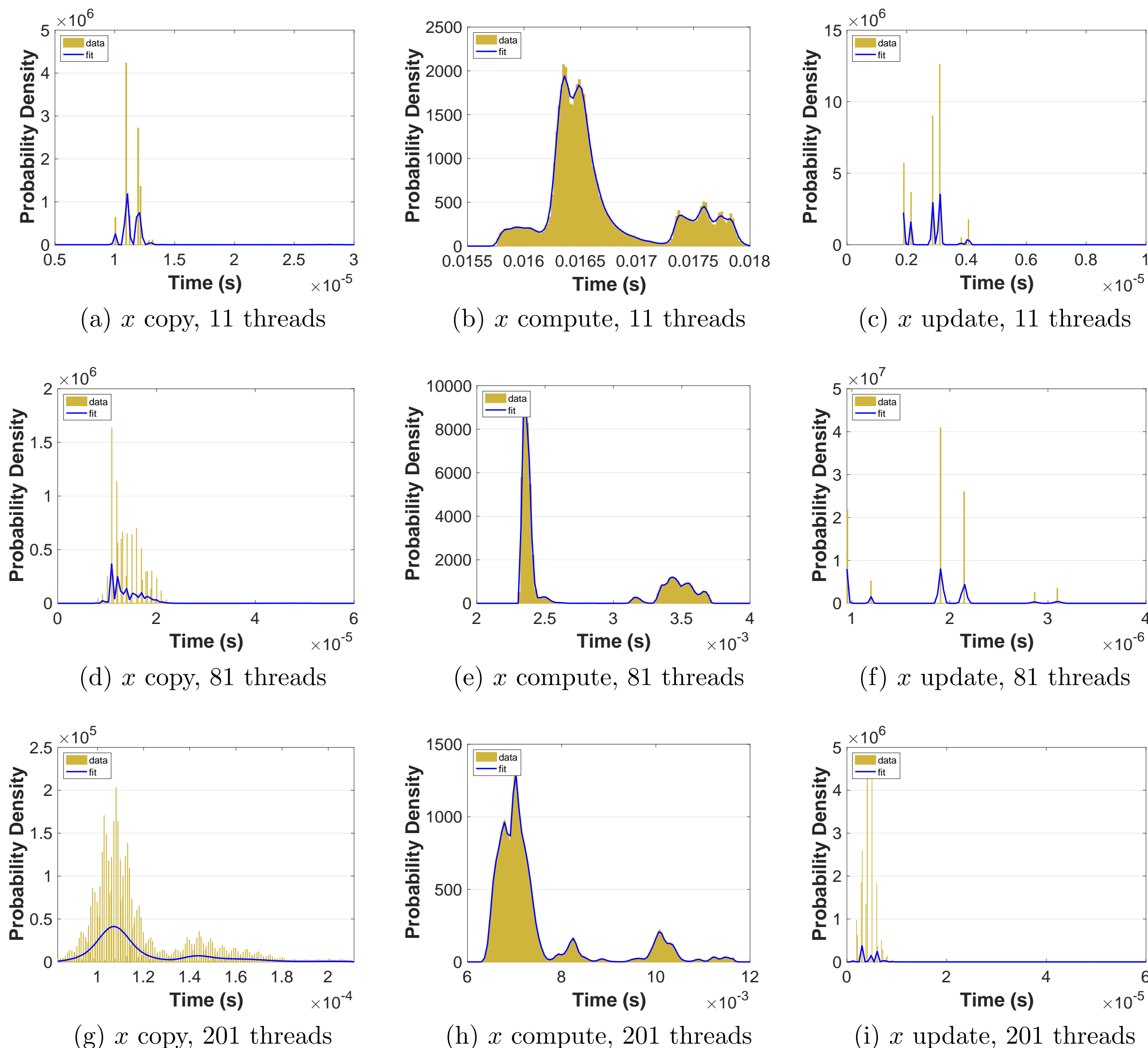
(d) x copy, 81 threads (e) x compute, 81 threads (f) x update, 81 threads

(g) x copy, 201 threads (h) x compute, 201 threads (i) x update, 201 threads

Fig. 45: Safe copy, compute, and update histograms with kernel fits.

6.1.4 IMPLEMENTATION 2: FINITE DIFFERENCE JACOBI SOLVER

This second implementation performs the Jacobi relaxation on the grid directly using the neighboring points required by the 5-point stencil as opposed to explicitly forming the matrix A, and in a sense implements a *matrix-free* solution. For this implementation, the Laplacian was discretized over a 600×600 grid with boundary conditions set according to Table 18.



(a) x copy, 11 threads (b) x compute, 11 threads (c) x update, 11 threads

(d) x copy, 81 threads (e) x compute, 81 threads (f) x update, 81 threads

(g) x copy, 201 threads (h) x compute, 201 threads (i) x update, 201 threads

Fig. 46: RACE copy, compute, and update histograms with kernel fits.

The implementation used here stems from code provided by [190]; similar code solves a three dimensional discretization of the Laplacian in the study featured in [107] and [108]. The routine solves a heat diffusion problem, in which a two-dimensional heated plate has Dirichlet boundary-condition temperatures. Two matrices, u_0 and u_1, store grid point values that each thread reads, e.g., from u_1, to compute newer values to write, e.g., to u_0. As the method is asynchronous, each thread independently determines which matrix stores its newer $u^{(t+1)}(i, j)$ values and older $u^{(t)}(i, j)$ values. For an $N + 2$ by $N + 2$ grid, each thread solves

Table 18: Boundary conditions for the second implementation of the Laplacian.

0	100	...	...	100	0
75	XXX	...	...	XXX	50
⋮	XXX	...	...	XXX	⋮
⋮	XXX	...	...	XXX	⋮
75	XXX	...	...	XXX	50
0	0	...	...	0	0

for N^2 grid points divided by n processing elements, such that the grid is evenly divided along the y-axis. When a thread copies grid point values above or below its domain for the computation, OpenMP®locks are employed to ensure that data is safely captured from a single iteration. Further, locks are used when updating values on domain boundaries. Each thread p_n computes its local residual value every k^{th} iteration, which it contributes to the global residual value using an OpenMP®atomic operation, such that it adds the local residual from the current iteration and subtracts the local residual from the previous iteration. A single thread checks for convergence with an atomic capture operation, and updates a shared flag variable if the criterion is satisfied. Pseudocode for this implementation is provided in Algorithm 21, where bold upper-case text indicates that OpenMP®locks are employed. Locks are used only with interior boundary rows, meaning they are unnecessary for the first and last rows in the domain.

In this implementation, data was collected only for the time to complete an iteration. Thread counts of 10, 25, 50, 75, 100, and 150 were used in this series of experiments. The average total iteration time for the varying

Figure 47 provides histograms and kernel fits for each of the thread counts. Table 19 and Fig. 47 show that with increasing thread count, mean iteration time decreases, but iteration times variance increases. This increase in iteration time variation may result from increased opportunities for lock collisions with greater thread counts.

Since this implementation is even more compute bound than the first one, Table 19 shows

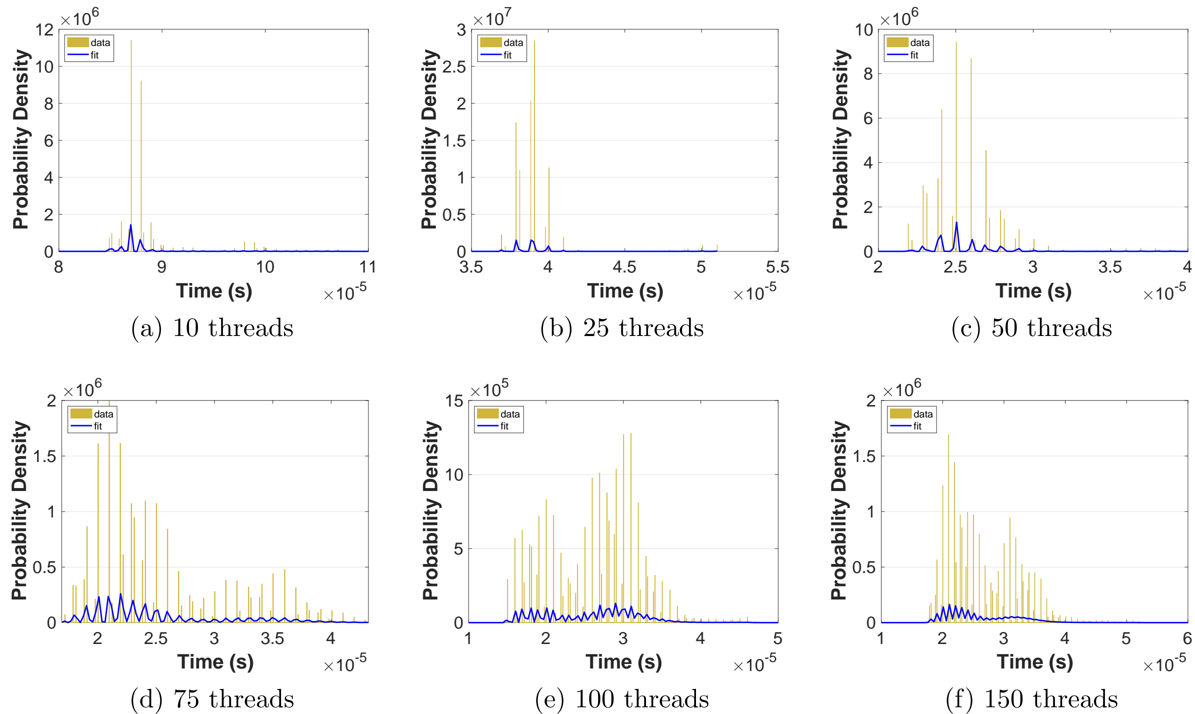
Algorithm 21: OpenMP®Implementation 2

Input: Initial guess for $u^{(0)}(i,j)$, n processing elements p
Output: Solution vector $u(i,j)$
1 Assign rows $u(i) \in u$ to each processing element, i = $[\alpha, \omega]$
2 **for *parallel*** *each processing element in* $p_1 \dots p_n$ **do**
3   **while** *residual norm* $>$ *tolerance* **do**
4   **for** *row index* $i = \alpha \dots \omega$ **do**
5   **if** $i \neq 1$ *AND* $i \neq N$ $i = \alpha$ *OR* $i = \omega$ **then**
6   **COPY** neighbor p_{n-1} or p_{n+1} boundary row values $u^{(t)}(i,j)$ for $u^{(t+1)}(i,j)$
7   Compute $u^{(t+1)}(i,j) = 1/4 * (u^{(t)}(i+1,j) + u^{(t)}(i-1,j) + u^{(t)}(i,j+1) + u^{(t)}(i,j-1)$
8   **if** $i \neq 1$ *AND* $i \neq N$ $i = \alpha$ *OR* $i = \omega$ **then**
9   **UPDATE** own p_n boundary row values $u_j^{(t)}(i,j)$ in shared memory with $u_j^{(t+1)}(i,j)$

Table 19: Mean iteration time and standard deviation by thread count.

Threads	Mean ($10^{-5}s$)	Std. ($10^{-6}s$)
10	8.86	3.87
25	3.92	2.08
50	2.55	2.34
75	2.53	5.80
100	2.61	5.95
150	2.64	5.76

a general decrease in the time for each iteration as the thread count is increased. While there is no inflection point evident in the data presented in Table 19, compared to Race in Table 17, Table 19 still suggests that once the number of threads outnumbers physical cores, performance gains diminish. For denser matrices, or for different applications on different systems, these trends could change as the memory-based activities become relatively more expensive. The finite difference discretization of the Laplacian is a very sparse matrix that does not require much data movement.



(a) 10 threads (b) 25 threads (c) 50 threads

(d) 75 threads (e) 100 threads (f) 150 threads

Fig. 47: Iteration time histograms with kernel fits.

6.1.5 IMPLEMENTATION COMPARISON

The SAFE variant of the first implementation incurs significant overhead costs for x copy and update operations, as thread count increases, because each thread must copy the entire x vector. In the second implementation, data shared between threads is differentiated and specific to domain location; therefore specific locks may be used when copying and updating segments of the subdomain. Assuming an appropriate number of processing elements for a given grid, i.e. a thread has significantly more middle rows than boundary rows, copy operations, and the associated variability and costs, are minimal compared with compute operations. The RACE implementation of the general solver eliminates much of the overhead cost from mutex locks, and convergence time is satisfactory for the given system. Implementation 2 is more constrained than Implementation 1, generalizing only to finite difference discretizations of partial differential equations over rectangular grids. Implementation 1

generalizes further to any sparse matrix, A, with which the Jacobi method can be used. According to Theorem 1, convergence will occur if the spectral radius of the iteration matrix, C, is less than 1. In the case of the Jacobi method, the iteration matrix is given by

$$C = -D^{-1}(L + U). \tag{140}$$

Note that in the two dimensional discretization of the Laplacian, the spectral radius of the Jacobian is less than l, which says that both the synchronous and asynchronous variants of the Jacobi algorithm will converge. Note RACE behavior is unknown for different problems and HPC systems.

The purpose of the two distinct implementations is to emphasize that the simulation framework proposed here can adapt to the behavior of different problems and platforms. The framework may be adapted to any asynchronous iterative method through the process of collecting data representative of individual update times and using the resultant data to model the system in the framework.

6.1.6 FRAMEWORK VALIDATION

To validate the performance of the simulation framework when initialized with appropriate distributions, a case study utilizing output from Implementation 1 (see Section 6.1.3 for details) was considered. Data was collected for a smaller problem size only in order to facilitate the collection of data over a large number of runs. Specifically, the Laplacian was discretized over a 20×20 grid resulting in a matrix of size 400×400. Similarly to the process in Section 6.1.3, distributions were fit to the output of the OpenMP®implementation, and these distributions were used in the simulation framework to provide update times to the simulated processors that are reflective of the HPC hardware that the data was collected on. Output from the average of these runs is provided in Table 20. The leftmost column provides the number of threads that were used (or simulated), the middle column shows the

average over multiple runs of the parallel implementation, and the rightmost column shows the average over multiple runs of the simulation generated by the simulation framework. In the case of this small problem, the similarity of actual and simulated run times helps to validate the model. Running multiple trials of larger problems in the framework is currently time-prohibitive, which is an issue that may be improved with framework implementation changes.

Table 20: Comparisons of run times between parallel executions and simulation.

Thread Count	Run Average (s)	Simulation Average (s)
11	0.01	0.01
21	0.02	0.02
41	0.04	0.04
51	0.04	0.05
81	0.09	0.09
101	0.12	0.12
201	0.34	0.35

6.2 FRAMEWORK EXTENSION FOR FAULT-TOLERANCE

The modular nature of this framework allows for extra functionality to be easily added to the framework itself that can be used to adapt the base algorithm to suit a specific set of requirements. With the projected increase of faults, development of fault tolerant algorithms is an important endeavor. A block diagram showing the additional functionality dealing with fault-tolerance is shown in Fig. 48. The new functionality is achieved by passing in another function handle that performs the fault tolerance check and recovery work. The contents of the newly added *Fault tolerance check* module may be organized as follows: Each processor makes a call to find the global residual and rolls the state back to the previous known good state if the behavior of the residual is not as expected. See Section 6.3 for more details. Note that this strategy is not being advocated for due to its optimality, but is being shown as

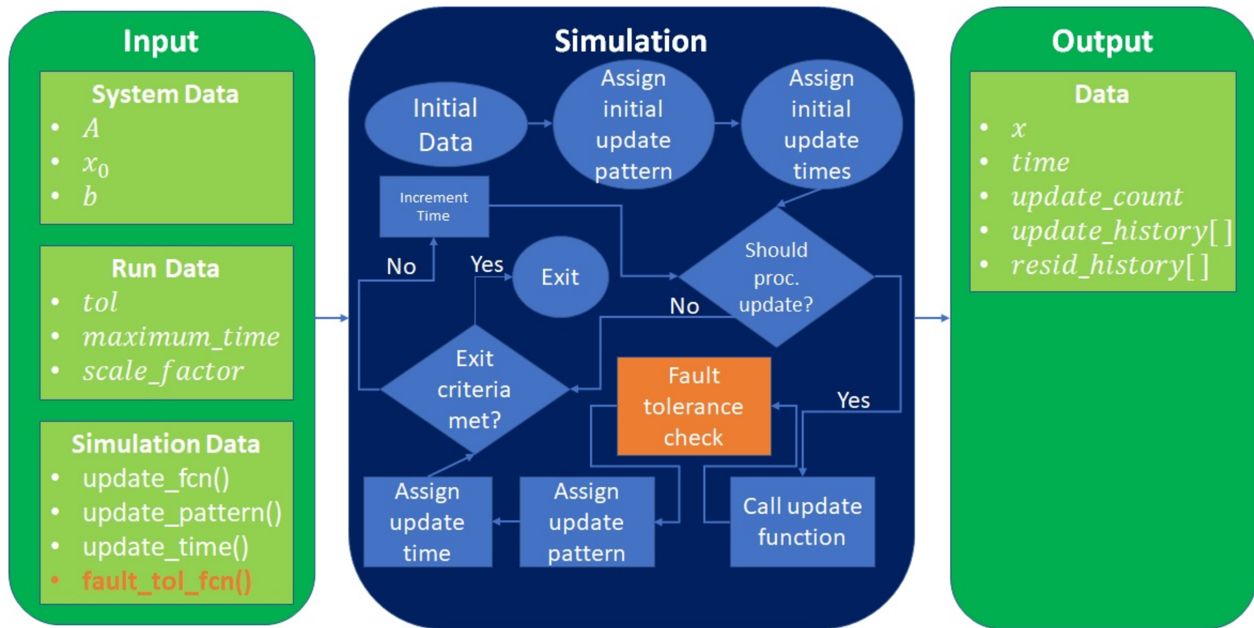


Fig. 48: Block diagram of the simulation framework with added support for fault tolerance mechanisms

an example of how to extend the framework for algorithm development. Techniques such as monitoring the progression of the component-wise residuals (e.g., [21], [26]) or only rolling back portions of the state vector (e.g., [165], [170]) would probably be more computationally efficient.

6.3 NUMERICAL EXPERIMENTS

For the numerical experiments shown in this chapter, faults are modeled using only the perturbation-based soft fault model (PBSFM) detailed in Chapter 4. Similar to the earlier results in the chapter using the nominal simulation framework, these experiments cover the solution of the linear system resulting from a two-dimensional finite difference discretization of the Laplacian. Before presenting simulation results, it is important to note that faults, as modeled here, will not prevent the *eventual* solution of the linear system using the (asynchronous) Jacobi method. Since the spectral radius of the associated iteration matrix is strictly less than 1, it will converge for any initial guess $x^{(0)}$.

Since faults are assumed to only affect the memory storing the vector x and are assumed to occur in a transient manner, if a fault occurs on iteration F then the subsequent iterate, $x^{(F+1)}$ can be taken to be the new starting iterate and eventual convergence is guaranteed due to the iteration matrix which has remained the same throughout the occurrence of the fault. This model can reflect the scenario where certain parts of the routine are designated to run on hardware with a higher reliability threshold, and other parts of the algorithm are allowed to run on hardware that may be more susceptible to the occurrence of a fault. This sandbox type design has been suggested as a possible means for providing energy efficient fault tolerance on future HPC environments [31], [32], [125].

While eventual convergence may be guaranteed, greatly accelerated convergence is possible through a simple checkpointing scheme. An example of such a scheme (as an extension of the asynchronous Jacobi simulation provided by Algorithm 19) is provided in Algorithm 22.

Algorithm 22: Asynchronous Jacobi simulation with checkpointing

Input: $a_{ij} \in A$; initial guess x_0; number of processing elements p; input random number distribution; checkpointing tolerance γ; checkpointing frequency ω

Output: Solution vector x

```
1  Assign processor update times τ_1, τ_2, ..., τ_p by sampling from an appropriate
     random number distribution
2  Assign a part of x to each processing element
3  Initialize r_old to a large value
4  for t = 1, 2, ..., until convergence do
5      for each processing element, P_l do
6          if τ_k = t then
7              for each element x_i ∈ x assigned to P_l do
8                  x_i = (−1/a_ii) [ Σ_{j≠i} a_ij x_j − b_i ]
9              Retrieve a new update time τ_k by sampling from the input distribution
10     Inject a fault if appropriate
11     Calculate the residual r_new as in Eq. (138)
12     if r_new > γ × r_old then
13         x ← x_cp
14     if  mod (t, ω) == 0 then
15         x_cp ← x
16     Check termination conditions
```

Note that the asynchronous nature of the iterative method means that a strict check on the decrease of the residual (i.e. expecting monotonic decrease) is not possible. In particular, the checkpointing tolerance γ needs to be taken such that $\gamma > 1$. However, the expected manifestation of faults as rare, transient events allows γ to be taken fairly large. Taking γ too large results in a fault having a substantial impact on the convergence rate of algorithm since large faults will be allowed to impact the algorithm with no correction. Conversely, taking γ too small causes the algorithm to checkpoint more frequently than needed. Examples of the effects of a fault with different values selected for γ are given by Fig. 49.

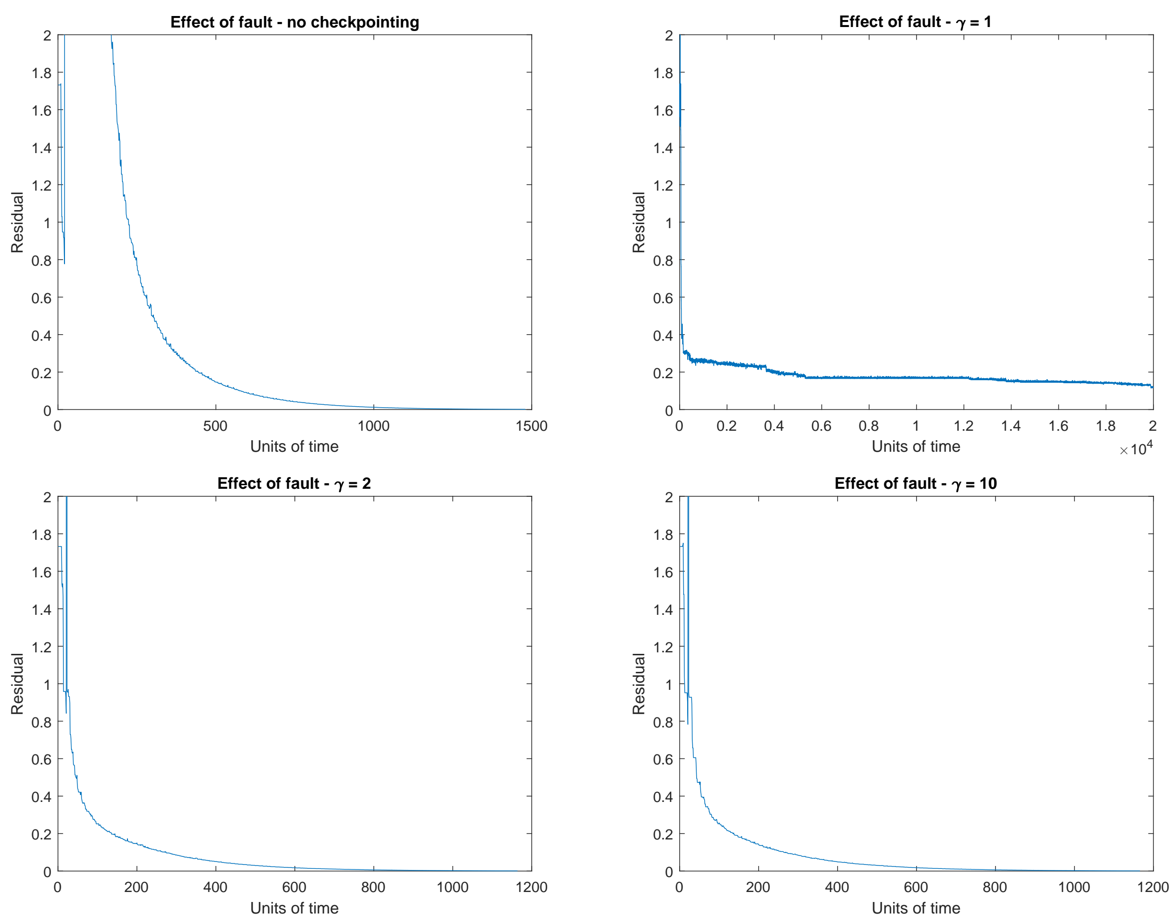


Fig. 49: Effect of differing values of γ on the progression of the residual

Note in Fig. 49 that no checkpointing results in a delay to convergence relative to the

use of checkpointing with either $\gamma = 1$ or $\gamma = 10$. The size of the fault selected in this study, $r_i \in (-100, 100)$, which may be reflective of an exponent or sign bit flip [170], results in the values $\gamma = 1$ and $\gamma = 10$ having the same performance since the error induced by the fault is sufficiently large that the new residual is more than $\gamma = 10$ times the prior residual. Faults that induce a smaller error may be detected by certain values of γ and not by others which would lead to differing performance.

The residual progress in the plot showing the effects of using $\gamma = 1$ can be explained by the updates provided by certain simulated processing elements being rejected despite being necessary for the convergence of the algorithm. This can be seen in the small, momentary jumps in the progression of the residual visible in the other graphs. These rejections lead to stagnation in the progression of the algorithm and show why the value of $\gamma = 1$ should not be selected for a checkpointing scheme for an asynchronous iterative method.

6.4 SUMMARY

This work has developed a framework that can be used to efficiently simulate the outcomes of asynchronous methods for future High Performance Computing environments. Given that asynchronous methods are notoriously difficult to study theoretically, their simulation is an invaluable tool for observing behavior and making quantitative and qualitative assertions. The modular and extensible nature of the framework proposed here allows for easy experimentation with modifications to a popular class of algorithms that finds uses in many areas of science and engineering.

The work presented was designed to show the ability of the framework to adapt to new algorithm variants, such as those capable of handling algorithm recovery in the presence of transient soft faults as was shown by example in Section 6.3. The simulation framework presented here is extensible and flexible and is able to:

1. admit a variety of asynchronous methods (i.e., beyond the simple Jacobi algorithm)

2. incorporate different fault models and recovery techniques for the development of fault

tolerant algorithms, and

3. vary hardware parameters such as thread and processor counts and the performance of those parameters as governed by the timing distributions that are supplied.

CHAPTER 7

CONCLUSIONS

This dissertation has examined the impact of soft faults on fine-grained parallel iterative algorithms. Distributed and shared memory HPC environments have been studied, and modeling and simulation tools to study such algorithms when implemented either synchronously or asynchronously have been developed.

7.1 THEORETICAL RESULTS AND TECHNIQUES

The theoretical results provide extensions to existing theorems that help define the conditions required for asynchronous iterative algorithms to converge despite the occurrence of a fault. Two approaches to modeling the impact of a fault were considered; one that assumes the effect of the fault is contained, and another that allows for arbitrary data corruption. The theoretical results developed led to the categorization of several techniques that can be used for recovery, and a series of specific examples showing how the techniques can be applied to popular linear solvers were included. The theory can be applied to other algorithms for the development of further fault tolerant algorithms.

7.2 NUMERICAL SOFT FAULT MODELING

A novel technique for simulating the occurrence of a soft fault on an HPC environment based upon injecting random perturbations to pertinent data structures was developed and tested. Experiments were conducted on both distributed and shared memory applications, with a focus on fine-grained parallel iterative techniques and popular projection based solvers. Further, a comparison and analysis of the proposed model with an existing fault model currently used in the research community was presented. These techniques were compared against each other as well as direct simulation of faults and analyzed for their

ability to aid in the development of fault tolerant algorithms. Results were presented for asynchronous iterative methods, including a hybrid parallel implementation of the asynchronous Jacobi algorithm. The results indicate that the use of numerical soft fault models may be useful for the development of fault tolerant algorithms for future High Performance Computing platforms. The testing show that numerical simulation of soft faults provides a consistent, reliable way to force sufficient data corruption to examine the behavior of iterative algorithms. This makes numerical soft fault models a valuable means of developing novel fault tolerant algorithms; an activity that will become increasingly important as HPC environments progresses towards exascale levels of performance.

7.3 FAULT TOLERANT FINE-GRAINED INCOMPLETE FACTORIZATIONS

This dissertation provided a use case of how the proposed techniques for resilience can be combined with the novel modeling and simulation tools to develop fault tolerant variants of algorithms used in HPC applications. These examples use the fine-grained parallel incomplete factorization (FGPILU) algorithm that can be used to generate preconditioners for the large sparse linear systems that often arise in large scale simulation of phenomena in science and engineering. The impact of soft faults on the FGPILU algorithm was studied, and several variants to remedy their negative effects were proposed. The variants offer a means to safely use the algorithm in HPC environments that may not be fault-free.

7.4 FRAMEWORK FOR MODELING AND ANALYSIS

Multiple performance models based upon the evaluation of asynchronous iterative methods performing tasks central to large scale simulation were developed, and were used in the development of a framework to efficiently simulate the outcomes of asynchronous methods for future HPC environments. The modular and extensible nature of the framework proposed here allows for easy experimentation with modifications to a popular class of algorithms that finds uses in many areas of science and engineering. The work presented in this dissertation

was designed to show the ability of the framework to adapt to new algorithm variants, such as those capable of handling algorithm recovery in the presence of transient soft faults.

7.5 FUTURE WORK

As the HPC environments that aid in large scale modeling and simulation efforts continue to progress towards exascale levels of performance, there are many areas to expand upon the line of research presented in this dissertation. Moving forward, it will be important to continue expanding on the general theory of fault tolerance for fine-grained iterative methods in order to further develop understanding for how this large class of methods will respond to any unforeseen errors that arise during computing. One manner that this could be done beneficially is in the development of results that have meaningful bounds on convergence rate. Many of the results developed are asymptotic in nature, guaranteeing eventual convergence, but being able to bound the maximum required time to convergence can be very beneficial. Further, it is always important to continue applying it to specific applications that make use of asynchronous fixed point methods.

Similarly, it would also be helpful to extend the testing to a larger suite of algorithms. Examining the performance of the techniques and models developed here on a suite of problems and solvers may help improve the quality of the analysis. Examples of other fine-grained parallel iterative algorithms include randomized linear solvers (e.g., [104], [105]), randomized optimization routines (e.g., [27], [64], [191]), weighted asynchronous linear solvers (e.g., [160], [192]), and more robust incomplete factorizations (e.g., [178]). The iterative methods used were limited to Conjugate Gradient for symmetric problems and GMRES for non-symmetric problems; while these are the most popular choices, extending to other routines may help increase the value of the analysis. The use of recent parallel computing constructs such as one-sided remote memory access [116] may help improve the performance of the implementations shown here.

One other area that the techniques developed could positively impact is in scenarios

where the given HPC architecture is heterogeneous in nature, e.g., where some compute resources are strictly CPU based, some are GPU based, and the size of the problem assigned to each component or node is not necessarily uniform. In this scenario, the asynchronous nature of the algorithms under study throughout this dissertation may be able to help with load balancing enough to make a large class of algorithms able to be viably deployed on this heterogeneous system. A series of studies dedicated to proving the efficacy of these ideas would help pave the way for new techniques.

While this dissertation has made a thorough study of the effect of soft faults on asynchronous iterative methods, the effect of hard faults is an area that has not been examined much in the literature. The use of randomization in the selection of components to update is one manner in which hard faults may be dealt with successfully, and it is worth exploring in the future. Additionally, it would be beneficial to create more streamlined performance prototypes of each of the variants in order to get a more accurate gauge of the relative performance among them. Lastly, it may be helpful to tie the parameters of the algorithm to intrinsic properties of the problem itself, in order to alleviate the burden of any performance tuning from the user.

REFERENCES

[1] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, *et al.*, "Ascac subcommittee report: The opportunities and challenges of exascale computing," tech. rep., Technical report, United States Department of Energy, Fall, 2010.

[2] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, *et al.*, "The opportunities and challenges of exascale computing–summary report of the advanced scientific computing advisory committee (ascac) subcommittee," *US Department of Energy Office of Science*, 2010.

[3] J. Dongarra, J. Hittinger, J. Bell, L. Chacon, R. Falgout, M. Heroux, P. Hovland, E. Ng, C. Webster, and S. Wild, "Applied mathematics research for exascale computing," tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2014.

[4] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *The International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

[5] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing frontiers and innovations*, vol. 1, no. 1, 2014.

[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, *et al.*, "The landscape of parallel computing research: A view from Berkeley," tech. rep., Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[7] A. Geist and R. Lucas, "Major computer science challenges at exascale," *International Journal of High Performance Computing Applications*, 2009.

[8] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, *et al.*, "Addressing failures in exascale computing," *International Journal of High Performance Computing Applications*, 2014.

[9] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, vol. 15, 2008.

[10] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, *et al.*, "The international exascale software project roadmap," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011.

[11] J. Tsitsiklis, D. Bertsekas, and M. Athans, "Distributed asynchronous deterministic and stochastic gradient optimization algorithms," *IEEE transactions on automatic control*, vol. 31, no. 9, pp. 803–812, 1986.

[12] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: a large-scale field study," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 37 (no. 1), pp. 193–204, ACM, 2009.

[13] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010.

[14] M. R. Varela, K. B. Ferreira, and R. Riesen, "Fault-tolerance for exascale systems," in *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pp. 1–4, IEEE, 2010.

[15] R. T. Biedron, J.-R. Carlson, J. M. Derlaga, P. A. Gnoffo, D. P. Hammond, W. T. Jones, B. Kleb, E. M. Lee-Rausch, E. J. Nielsen, M. A. Park, *et al.*, "Fun3d manual: 13.3," 2018.

[16] H. Jasak, A. Jemcov, Z. Tukovic, *et al.*, "Openfoam: A c++ library for complex physics simulations," in *International workshop on coupled methods in numerical dynamics*, vol. 1000, pp. 1–20, IUC Dubrovnik, Croatia, 2007.

[17] I. Karlin, J. Keasler, and J. Neely, "Lulesh 2.0 updates and changes," tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2013.

[18] M. S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells, "The fenics project version 1.5," *Archive of Numerical Software*, vol. 3, no. 100, pp. 9–23, 2015.

[19] W. Bangerth, R. Hartmann, and G. Kanschat, "deal. iia general-purpose object-oriented finite element library," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 4, p. 24, 2007.

[20] D. Gaston, C. Newman, G. Hansen, and D. Lebrun-Grandie, "Moose: A parallel computational framework for coupled systems of nonlinear equations," *Nuclear Engineering and Design*, vol. 239, no. 10, pp. 1768–1778, 2009.

[21] H. Anzt, J. Dongarra, and E. S. Quintana-Ortí, "Fine-grained bit-flip protection for relaxation methods," *Journal of Computational Science*, 2016.

[22] H. Anzt, *Asynchronous and multiprecision linear solvers-scalable and fault-tolerant numerics for energy efficient high performance computing.* PhD thesis, Karlsruhe, Karlsruher Institut für Technologie (KIT), Diss., 2012, 2012.

[23] E. Chow, H. Anzt, and J. Dongarra, "Asynchronous iterative algorithm for computing incomplete factorizations on GPUs," in *International Conference on High Performance Computing*, pp. 1–16, Springer, 2015.

[24] E. Chow and A. Patel, "Fine-grained parallel incomplete LU factorization," *SIAM journal on Scientific Computing*, vol. 37, no. 2, pp. C169–C193, 2015.

[25] H. Anzt, E. Chow, and J. Dongarra, "Iterative sparse triangular solves for preconditioning," in *European Conference on Parallel Processing*, pp. 650–661, Springer, 2015.

[26] H. Anzt, J. Dongarra, and E. S. Quintana-Ortí, "Tuning stationary iterative solvers for fault resilience," in *Proceedings of the 6th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, p. 1, ACM, 2015.

[27] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in neural information processing systems*, pp. 693–701, 2011.

[28] N. Aybat, Z. Wang, and G. Iyengar, "An asynchronous distributed proximal gradient method for composite convex optimization," in *International Conference on Machine Learning*, pp. 2454–2462, 2015.

[29] Y. K. Cheung and R. Cole, "A unified approach to analyzing asynchronous coordinate descent and tatonnement," *arXiv preprint arXiv:1612.09171*, 2016.

[30] F. Magoules, D. B. Szyld, and C. Venet, "Asynchronous optimized Schwarz methods with and without overlap," *Numerische Mathematik*, pp. 1–29, 2015.

[31] M. Hoemmen and M. A. Heroux, "Fault-tolerant iterative methods via selective reliability," in *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE Computer Society*, vol. 3, p. 9, Citeseer, 2011.

[32] P. G. Bridges, K. B. Ferreira, M. A. Heroux, and M. Hoemmen, "Fault-tolerant linear solvers via selective reliability," *arXiv preprint arXiv:1206.1390*, 2012.

[33] K.-H. Huang *et al.*, "Algorithm-based fault tolerance for matrix operations," *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.

[34] Y. Saad, *Iterative methods for sparse linear systems.* SIAM, 2003.

[35] Y. Saad and M. H. Schultz, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on scientific and statistical computing*, vol. 7, no. 3, pp. 856–869, 1986.

[36] Y. Saad, "A flexible inner-outer preconditioned GMRES algorithm," *SIAM Journal on Scientific Computing*, vol. 14, no. 2, pp. 461–469, 1993.

[37] H. A. Van der Vorst and C. Vuik, "Gmresr: a family of nested gmres methods," *Numerical linear algebra with applications*, vol. 1, no. 4, pp. 369–386, 1994.

[38] L. B. Wigton, N. Yu, and D. P. Young, "Gmres acceleration of computational fluid dynamics codes," *NASA Technical Report SEE A85-40926 19-34*, 1985.

[39] A. Chapman, Y. Saad, and L. Wigton, "High-order ilu preconditioners for cfd problems," *International Journal for numerical methods in fluids*, vol. 33, no. 6, pp. 767–788, 2000.

[40] J. Zhang, "Preconditioned krylov subspace methods for solving nonsymmetric matrices from cfd applications," *Computer methods in applied mechanics and engineering*, vol. 189, no. 3, pp. 825–840, 2000.

[41] Y. Saad, A. Soulaimani, and R. Touihri, "Variations on algebraic recursive multilevel solvers (arms) for the solution of cfd problems," *Applied numerical mathematics*, vol. 51, no. 2-3, pp. 305–327, 2004.

[42] M. Sosonkina, Y. Saad, and X. Cai, "Using the parallel algebraic recursive multilevel solver in modern physical applications," *Future Generation Computer Systems*, vol. 20, no. 3, pp. 489–500, 2004.

[43] J. van den Eshof and G. Sleijpen, "Inexact Krylov subspace methods for linear systems," *SIAM Journal on Matrix Analysis and Applications*, vol. 26, no. 1, pp. 125–153, 2004.

[44] V. Simoncini and D. B. Szyld, "Theory of inexact Krylov subspace methods and applications to scientific computing," *SIAM Journal on Scientific Computing*, vol. 25, no. 2, pp. 454–477, 2003.

[45] V. Simoncini and D. B. Szyld, "On the occurrence of superlinear convergence of exact and inexact krylov subspace methods," *SIAM review*, vol. 47, no. 2, pp. 247–272, 2005.

[46] R. Li and Y. Saad, "Gpu-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, 2013.

[47] A. Jamal, M. Baboulin, A. Khabou, and M. Sosonkina, "A hybrid cpu/gpu approach for the parallel algebraic recursive multilevel solver parms," in *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2016 18th International Symposium on*, pp. 411–416, IEEE, 2016.

[48] Y. Saad and M. Sosonkina, "parms: A package for solving general sparse linear systems on parallel computers," in *International Conference on Parallel Processing and Applied Mathematics*, pp. 446–457, Springer, 2001.

[49] Z. Li, Y. Saad, and M. Sosonkina, "parms: a parallel version of the algebraic recursive multilevel solver," *Numerical linear algebra with applications*, vol. 10, no. 5-6, pp. 485–509, 2003.

[50] Y. Saad and M. Sosonkina, "parms: A package for the parallel iterative solution of general large sparse linear systems users guide," *Minneapolis: MN*, 2004.

[51] E. Chow and Y. Saad, "Experimental study of ilu preconditioners for indefinite matrices," *Journal of Computational and Applied Mathematics*, vol. 86, no. 2, pp. 387–414, 1997.

[52] M. Benzi, J. C. Haws, and M. Tuma, "Preconditioning highly indefinite and nonsymmetric matrices," *SIAM Journal on Scientific Computing*, vol. 22, no. 4, pp. 1333–1353, 2000.

[53] M. Benzi, D. B. Szyld, and A. Van Duin, "Orderings for incomplete factorization preconditioning of nonsymmetric problems," *SIAM Journal on Scientific Computing*, vol. 20, no. 5, pp. 1652–1670, 1999.

[54] M. Stoyanov and C. Webster, "Numerical analysis of fixed point algorithms in the presence of hardware faults," *SIAM Journal on Scientific Computing*, vol. 37, no. 5, pp. C532–C553, 2015.

[55] H. Anzt, P. Luszczek, J. Dongarra, and V. Heuveline, "GPU-accelerated asynchronous error correction for mixed precision iterative refinement," *Euro-Par 2012 Parallel Processing*, pp. 908–919, 2012.

[56] A. Addou and A. Benahmed, "Parallel synchronous algorithm for nonlinear fixed point problems," *International Journal of Mathematics and Mathematical Sciences*, vol. 2005, no. 19, pp. 3175–3183, 2005.

[57] A. Benahmed, "A convergence result for asynchronous algorithms and applications," *Proyecciones (Antofagasta)*, vol. 26, no. 2, pp. 219–236, 2007.

[58] A. Frommer and D. B. Szyld, "On asynchronous iterations," *Journal of computational and applied mathematics*, vol. 123, no. 1, pp. 201–216, 2000.

[59] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and distributed computation: numerical methods*, vol. 23. Prentice hall Englewood Cliffs, NJ, 1989.

[60] J. M. Ortega and W. C. Rheinboldt, *Iterative solution of nonlinear equations in several variables.* SIAM, 2000.

[61] M. Hong, "A distributed, asynchronous and incremental algorithm for nonconvex optimization: An admm approach," *IEEE Transactions on Control of Network Systems*, 2017.

[62] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011.

[63] M. Zhong and C. G. Cassandras, "Asynchronous distributed optimization with event-driven communication," *IEEE Transactions on Automatic Control*, vol. 55, no. 12, pp. 2735–2750, 2010.

[64] K. Srivastava and A. Nedic, "Distributed asynchronous constrained stochastic optimization," *IEEE Journal of Selected Topics in Signal Processing*, vol. 5, no. 4, pp. 772–790, 2011.

[65] H. Anzt, E. Chow, J. Saak, and J. Dongarra, "Updating incomplete factorization preconditioners for model order reduction," *Numerical Algorithms*, vol. 73, no. 3, pp. 611–630, 2016.

[66] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear algebra and its applications*, vol. 2, no. 2, pp. 199–222, 1969.

[67] J. Donnelly, "Periodic chaotic relaxation," *Linear algebra and its Applications*, vol. 4, no. 2, pp. 117–128, 1971.

[68] J. C. Miellou, “Chaotic relaxation algorithms delays,” *Revue fran c c automatic, computer science, operational research. Numerical analysis*, vol. 9, no. R1, pp. 55–82, 1975.

[69] H. Kung, “Synchronized and asynchronous parallel algorithms for multiprocessors,” *New Directions and Recent Results in Algorithms and Complexity*, 1976.

[70] G. M. Baudet, “Asynchronous iterative methods for multiprocessors,” *Journal of the ACM (JACM)*, vol. 25, no. 2, pp. 226–244, 1978.

[71] G. M. Baudet, “The design and analysis of algorithms for asynchronous multiprocessors.,” tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1978.

[72] A. Üresin and M. Dubois, “Sufficient conditions for the convergence of asynchronous iterations,” *Parallel Computing*, vol. 10, no. 1, pp. 83–92, 1989.

[73] D. P. Bertsekas and J. N. Tsitsiklis, “Convergence rate and termination of asynchronous iterative algorithms,” in *Proceedings of the 3rd International Conference on Supercomputing*, pp. 461–470, ACM, 1989.

[74] E. W. Dijkstra and C. S. Scholten, “Termination detection for diffusing computations,” *Information Processing Letters*, vol. 11, no. 1, pp. 1–4, 1980.

[75] S. Li and T. Basar, “Asymptotic agreement and convergence of asynchronous stochastic algorithms,” *IEEE Transactions on Automatic Control*, vol. 32, no. 7, pp. 612–618, 1987.

[76] J. N. Tritsiklis, “A comparison of jacobi and gauss-seidel parallel iterations,” *Applied Mathematics Letters*, vol. 2, no. 2, pp. 167–170, 1989.

[77] A. Bojańczyk, “Optimal asynchronous Newton method for the solution of nonlinear equations,” *Journal of the ACM (JACM)*, vol. 31, no. 4, pp. 792–803, 1984.

[78] M. Anwar and M. N. El Tarazi, "Asynchronous algorithms for Poisson's equation with nonlinear boundary conditions," *Computing*, vol. 34, no. 2, pp. 155–168, 1985.

[79] P. Spiteri, "Parallel asynchronous algorithms for solving boundary value problems," in *Proceedings of the international workshop on Parallel algorithms & architectures*, pp. 73–84, North-Holland Publishing Co., 1986.

[80] D. Smart and J. White, "Reducing the parallel solution time of sparse circuit matrices using reordered Gaussian elimination and relaxation," tech. rep., MASSACHUSETTS INST OF TECH CAMBRIDGE MICROSYSTEMS RESEARCH CENTER, 1988.

[81] D. Mitra, "Asynchronous relaxations for the numerical solution of differential equations by parallel processors," *SIAM journal on scientific and statistical computing*, vol. 8, no. 1, pp. s43–s58, 1987.

[82] W. K. Tsai, "Convergence of gradient projection routing methods in an asynchronous stochastic quasi-static virtual circuit network," *IEEE transactions on automatic control*, vol. 34, no. 1, pp. 20–33, 1989.

[83] D. P. Bertsekas, "Distributed asynchronous computation of fixed points," *Mathematical Programming*, vol. 27, no. 1, pp. 107–120, 1983.

[84] M. N. El Tarazi, "Some convergence results for asynchronous algorithms," *Numerische Mathematik*, vol. 39, no. 3, pp. 325–340, 1982.

[85] D. Bertsekas, "Distributed dynamic programming," *IEEE transactions on Automatic Control*, vol. 27, no. 3, pp. 610–616, 1982.

[86] A. Uresin and M. Dubois, "Generalized asynchronous iterations," *CONPAR 86*, pp. 272–278, 1986.

[87] B. Lubachevsky and D. Mitra, "A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius," *Journal of the ACM (JACM)*, vol. 33, no. 1, pp. 130–150, 1986.

[88] J. N. Tsitsiklis, "On the stability of asynchronous iterative processes," *Mathematical systems theory*, vol. 20, no. 1, pp. 137–153, 1987.

[89] D. P. Bertsekas and J. N. Tsitsiklis, "Some aspects of parallel and distributed iterative algorithmsa survey," *Automatica*, vol. 27, no. 1, pp. 3–21, 1991.

[90] E. Kaszkurewicz, A. Bhaya, and D. Šiljak, "On the convergence of parallel asynchronous block-iterative computations," *Linear Algebra and its Applications*, vol. 131, pp. 139–160, 1990.

[91] A. "Uresin and M. Dubois, "Parallel asynchronous algorithms for discrete data," *Journal of the ACM (JACM)*, vol. 37, no. 3, pp. 588–606, 1990.

[92] A. Frommer, "Generalized nonlinear diagonal dominance and applications to asynchronous iterative methods," *Journal of Computational and Applied Mathematics*, vol. 38, no. 1-3, pp. 105–124, 1991.

[93] J. M. Bull and T. Freeman, "Numerical performance of an asynchronous jacobi iteration," in *Parallel Processing: CONPAR 92VAPP V*, pp. 361–366, Springer, 1992.

[94] A. Bhaya, E. Kaszurewicz, and F. Mota, "Asynchronous block-iterative methods for almost linear equations," *Linear algebra and its applications*, vol. 154, pp. 487–508, 1991.

[95] P. Tseng, "Distributed computation for linear programming problems satisfying a certain diagonal dominance condition," *Mathematics of Operations Research*, vol. 15, no. 1, pp. 33–48, 1990.

[96] A. Frommer and D. B. Szyld, “Asynchronous two-stage iterative methods,” *Numerische Mathematik*, vol. 69, no. 2, pp. 141–153, 1994.

[97] R. Bru, V. Migallón, J. Penadés, and D. B. Szyld, “Parallel, synchronous and asynchronous two-stage multisplitting methods,” *Electronic Transactions on Numerical Analysis*, vol. 3, pp. 24–38, 1995.

[98] S. A. Savarí and D. P. Bertsekas, “Finite termination of asynchronous iterative algorithms,” *Parallel Computing*, vol. 22, no. 1, pp. 39–56, 1996.

[99] D. B. Szyld, “Different models of parallel asynchronous iterations with overlapping blocks,” *Computational and applied mathematics*, vol. 17, pp. 101–115, 1998.

[100] J. M. Bahi, “Asynchronous iterative algorithms for nonexpansive linear systems,” *Journal of Parallel and Distributed Computing*, vol. 60, no. 1, pp. 92–112, 2000.

[101] A. Frommer, H. Schwandt, and D. B. Szyld, “Asynchronous weighted additive Schwarz methods,” *Electronic Transactions on Numerical Analysis*, vol. 5, no. 48-61, pp. 1–5, 1997.

[102] J. C. Strikwerda, *Finite difference schemes and partial differential equations*, vol. 88. Siam, 2004.

[103] A. Bhaya, E. Kaszkurewicz, and Y. Su, “Stability of asynchronous two-dimensional Fornasini–Marchesini dynamical systems,” *Linear Algebra and Its Applications*, vol. 332, pp. 257–263, 2001.

[104] J. C. Strikwerda, “A probabilistic analysis of asynchronous iteration,” *Linear algebra and its applications*, vol. 349, no. 1-3, pp. 125–154, 2002.

[105] H. Avron, A. Druinsky, and A. Gupta, “Revisiting asynchronous linear solvers: Provable convergence rate through randomization,” *Journal of the ACM (JACM)*, vol. 62, no. 6, p. 51, 2015.

[106] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, *Parallel iterative algorithms: from sequential to grid computing*. Chapman and Hall/CRC, 2007.

[107] I. Bethune, J. M. Bull, N. J. Dingle, and N. J. Higham, "Investigating the Performance of Asynchronous Jacobi's Method for Solving Systems of Linear Equations," *To appear in International Journal of High Performance Computing Applications*, 2011.

[108] I. Bethune, J. M. Bull, N. J. Dingle, and N. J. Higham, "Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP," *The International Journal of High Performance Computing Applications*, vol. 28, no. 1, pp. 97–111, 2014.

[109] J. Hook and N. Dingle, "Performance analysis of asynchronous parallel jacobi," *Numerical Algorithms*, pp. 1–36, 2013.

[110] H. Anzt, S. Tomov, J. Dongarra, and V. Heuveline, "A block-asynchronous relaxation method for graphics processing units," *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1613–1626, 2013.

[111] H. Anzt, J. Dongarra, and E. Chow, "On block-asynchronous execution on GPUs," *Preprint*, 2015.

[112] H. Anzt, S. Tomov, J. Dongarra, and V. Heuveline, "Weighted Block-Asynchronous Iteration on GPU-Accelerated Systems.," in *Euro-Par Workshops*, pp. 145–154, Springer, 2012.

[113] H. Anzt, E. Chow, D. B. Szyld, and J. Dongarra, "Domain Overlap for Iterative Sparse Triangular Solves on GPUs," in *Software for Exascale Computing-SPPEXA 2013-2015*, pp. 527–545, Springer, 2016.

[114] M. Si, A. J. Pena, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa, "Casper: An asynchronous progress model for MPI RMA on many-core architectures," in *Parallel*

and Distributed Processing Symposium (IPDPS), 2015 IEEE International, pp. 665–676, IEEE, 2015.

[115] F. Magoulès and G. Gbikpi-Benissan, "JACK: an asynchronous communication kernel library for iterative algorithms," *The Journal of Supercomputing*, pp. 1–20, 2016.

[116] R. Gerstenberger, M. Besta, and T. Hoefler, "Enabling highly-scalable remote memory access programming with MPI-3 one sided," *Scientific Programming*, vol. 22, no. 2, pp. 75–91, 2014.

[117] A. Geist, "What is the monster in the closet?," in *Invited Talk at Workshop on Architectures I: Exascale and Beyond: Gaps in Research, Gaps in our Thinking*, vol. 2, 2011.

[118] A. Geist, "Exascale monster in the closet," in *2012 IEEE Workshop on Silicon Errors in Logic–System Effects, Champaign-Urbana, IL, March*, pp. 27–28, 2012.

[119] A. Geist, "Supercomputing's monster in the closet," *IEEE Spectrum*, vol. 53, no. 3, pp. 30–35, 2016.

[120] G. Fagg and J. Dongarra, "Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world," *Recent advances in parallel virtual machine and message passing interface*, pp. 346–353, 2000.

[121] G. E. Fagg, A. Bukovsky, and J. J. Dongarra, "Harness and fault tolerant mpi," *Parallel Computing*, vol. 27, no. 11, pp. 1479–1495, 2001.

[122] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra, "An evaluation of user-level failure mitigation support in mpi," in *European MPI Users' Group Meeting*, pp. 193–203, Springer, 2012.

[123] W. Bland, "User level failure mitigation in mpi.," in *Euro-Par Workshops*, pp. 499–504, Springer, 2012.

[124] G. Zheng, L. Shi, and L. V. Kalé, "Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi," in *Cluster Computing, 2004 IEEE International Conference on*, pp. 93–103, IEEE, 2004.

[125] P. Sao and R. Vuduc, "Self-stabilizing iterative solvers," in *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, p. 4, ACM, 2013.

[126] G. Bronevetsky and B. de Supinski, "Soft error vulnerability of iterative linear algebra methods," in *Proceedings of the 22nd annual international conference on Supercomputing*, pp. 155–164, ACM, 2008.

[127] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Characterizing the impact of soft errors on iterative methods in scientific computing," in *Proceedings of the international conference on Supercomputing*, pp. 152–161, ACM, 2011.

[128] Z. Chen, "Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods," in *ACM SIGPLAN Notices*, vol. 48, pp. 167–176, ACM, 2013.

[129] J. Sloan, R. Kumar, and G. Bronevetsky, "Algorithmic approaches to low overhead fault detection for sparse linear algebra," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pp. 1–12, IEEE, 2012.

[130] J. Elliott, M. Hoemmen, and F. Mueller, "Evaluating the impact of sdc on the gmres iterative solver," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 1193–1202, IEEE, 2014.

[131] J. Elliott, M. Hoemmen, and F. Mueller, "Tolerating Silent Data Corruption in Opaque Preconditioners," *arXiv preprint arXiv:1404.5552*, 2014.

[132] J. J. Elliott, F. Mueller, M. K. Stoyanov, and C. G. Webster, “Quantifying the impact of single bit flips on floating point arithmetic,” tech. rep., Oak Ridge National Laboratory (ORNL), 2013.

[133] J. Elliott, M. Hoemmen, and F. Mueller, “Resilience in numerical methods: a position on fault models and methodologies,” *arXiv preprint arXiv:1401.3013*, 2014.

[134] J. Elliott, M. Hoemmen, and F. Mueller, “A Numerical Soft Fault Model for Iterative Linear Solvers,” in *Proceedings of the 24nd International Symposium on High-Performance Parallel and Distributed Computing*, 2015.

[135] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, “Trace-based microarchitecture-level diagnosis of permanent hardware faults,” in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pp. 22–31, IEEE, 2008.

[136] F. A. Bower, D. J. Sorin, and S. Ozev, “Online diagnosis of hard faults in microprocessors,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 4, no. 2, p. 8, 2007.

[137] H. Casanova, “Simgrid: A toolkit for the simulation of application scheduling,” in *Cluster computing and the grid, 2001. proceedings. first ieee/acm international symposium on*, pp. 430–437, IEEE, 2001.

[138] H. Casanova, A. Legrand, and M. Quinson, “Simgrid: A generic framework for large-scale distributed experiments,” in *Computer Modeling and Simulation, 2008. UKSIM 2008. Tenth International Conference on*, pp. 126–131, IEEE, 2008.

[139] C. L. Dumitrescu and I. Foster, “Gangsim: a simulator for grid scheduling studies,” in *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, vol. 2, pp. 1151–1158, IEEE, 2005.

[140] R. Buyya and M. Murshed, "Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing," *Concurrency and computation: practice and experience*, vol. 14, no. 13-15, pp. 1175–1220, 2002.

[141] R. N. Calheiros, R. Ranjan, C. A. De Rose, and R. Buyya, "Cloudsim: A novel framework for modeling and simulation of cloud computing infrastructures and services," *arXiv preprint arXiv:0903.2525*, 2009.

[142] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and experience*, vol. 41, no. 1, pp. 23–50, 2011.

[143] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, "Performance comparison of parallel programming environments for implementing aiac algorithms," *The Journal of Supercomputing*, vol. 35, no. 3, pp. 227–244, 2006.

[144] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, "Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational grid," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pp. 9–pp, IEEE, 2003.

[145] D. V. De Jager and J. T. Bradley, "Extracting state-based performance metrics using asynchronous iterative techniques," *Performance Evaluation*, vol. 67, no. 12, pp. 1353–1372, 2010.

[146] K. Voronin, "A numerical study of an mpi/openmp implementation based on asynchronous threads for a three-dimensional splitting scheme in heat transfer problems," *Journal of Applied and Industrial Mathematics*, vol. 8, no. 3, pp. 436–443, 2014.

[147] M. Benzi, "Preconditioning techniques for large linear systems: a survey," *Journal of computational Physics*, vol. 182, no. 2, pp. 418–477, 2002.

[148] E. Chow, *Robust preconditioning for sparse linear systems.* PhD thesis, University of Minnesota, 1997.

[149] E. Coleman and M. Sosonkina, "Evaluating a Persistent Soft Fault Model on Preconditioned Iterative Methods," in *Proceedings of the 22nd annual International Conference on Parallel and Distributed Processing Techniques and Applications*, 2016.

[150] Y. Saad, "Ilut: A dual threshold incomplete lu factorization," *Numerical linear algebra with applications*, vol. 1, no. 4, pp. 387–402, 1994.

[151] Y. Saad, "Ilum: a multi-elimination ilu preconditioner for general sparse matrices," *SIAM Journal on Scientific Computing*, vol. 17, no. 4, pp. 830–847, 1996.

[152] Y. Saad and J. Zhang, "Bilutm: a domain-based multilevel block ilut preconditioner for general sparse matrices," *SIAM Journal on Matrix Analysis and Applications*, vol. 21, no. 1, pp. 279–299, 1999.

[153] Y. Saad and B. Suchomel, "ARMS: An algebraic recursive multilevel solver for general sparse linear systems," *Numerical linear algebra with applications*, vol. 9, no. 5, pp. 359–378, 2002.

[154] Saad, Yousef, "Matlab suite." `http://www-users.cs.umn.edu/~saad/software/`, Accessed 2017.

[155] M. Baboulin, A. Jamal, and M. Sosonkina, "Using random butterfly transformations in parallel Schur complement-based preconditioning," in *2015 Federated Conference on Computer Science and Information Systems, FedCSIS 2015, Lódz, Poland, September 13-16, 2015*, pp. 649–654, 2015.

[156] D. S. Parker, "Random butterfly transformations with applications in computational linear algebra," *Technical Report CSD-950023*, 1995.

[157] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov, "Accelerating linear system solutions using randomization techniques," *ACM Transactions on Mathematical Software (TOMS)*, vol. 39, no. 2, p. 8, 2013.

[158] M. Baboulin, X. S. Li, and F.-H. Rouet, "Using random butterfly transformations to avoid pivoting in sparse direct methods," in *International Conference on High Performance Computing for Computational Science*, pp. 135–144, Springer, 2014.

[159] E. Andersen, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, *et al.*, *LAPACK Users' Guide*. SIAM, 1999.

[160] J. Wolfson-Pou and E. Chow, "Reducing communication in distributed asynchronous iterative methods," *Procedia Computer Science*, vol. 80, pp. 1906–1916, 2016.

[161] G. D. Smith, *Numerical solution of partial differential equations: finite difference methods*. Oxford university press, 1985.

[162] T. Lindeberg, "Scale-space for discrete signals," *IEEE transactions on pattern analysis and machine intelligence*, vol. 12, no. 3, pp. 234–254, 1990.

[163] S. C. Chapra and R. P. Canale, *Numerical methods for engineers*, vol. 2. McGraw-Hill New York, 1998.

[164] E. Coleman, E. Jensen, and M. Sosonkina, "Enhancing Asynchronous Linear Solvers through Randomization," in *Proceedings of the 2019 Spring Simulation Multiconference (Submitted)*, Society for Computer Simulation International, 2019.

[165] E. Coleman and M. Sosonkina, "Fault Tolerance for Fine-Grained Iterative Methods," in *Proceedings of the 7th annual Virginia Modeling, Simulation, and Analysis Center Capstone Conference*, Virginia Modeling, Simulation, and Analysis Center, 2017.

[166] E. Coleman, E. Jensen, and M. Sosonkina, "Fault Tolerance for Asynchronous Linear Solvers," *Journal TBD*, 2019.

[167] E. Coleman and M. Sosonkina, "Analysis of Soft Fault Resilience for Parallel Asynchronous Fixed Point Problems," *Naval Surface Warfare Center, Dahlgren Division Technical Report TR-19/41 (Under review)*, 2019.

[168] E. Coleman, E. Jensen, and M. Sosonkina, "Simulation Framework for Asynchronous Iterative Methods," *Journal of Simulation Engineering*, 2018.

[169] E. Coleman, M. Sosonkina, and E. Chow, "Fault Tolerant Variants of the Fine-Grained Parallel Incomplete LU Factorization," in *Proceedings of the 2017 Spring Simulation Multiconference*, Society for Computer Simulation International, 2017.

[170] E. Coleman and M. Sosonkina, "Self-Stabilizing Fine-Grained Parallel Incomplete LU Factorization," *Sustainable Computing: Informatics and Systems*, 2018.

[171] E. Coleman and M. Sosonkina, "A Comparison and Analysis of Soft-Fault Error Models using FGMRES," in *Proceedings of the 6th annual Virginia Modeling, Simulation, and Analysis Center Capstone Conference*, Virginia Modeling, Simulation, and Analysis Center, 2016.

[172] E. Coleman, A. Jamal, M. Baboulin, A. Khabou, and M. Sosonkina, "A Comparison of Soft-Fault Error Models in the Parallel Preconditioned Flexible GMRES," in *Proceedings of the 12th International Conference on Parallel Processing and Applied Mathematics*, ACM, 2017.

[173] E. Coleman, E. Jensen, and M. Sosonkina, "Impacts of Three Soft-Fault Models on Hybrid Parallel Asynchronous Iterative Methods," in *30th International Symposium on Computer Architecture and High Performance Computing*, 2018.

[174] E. Coleman, E. Jensen, and M. Sosonkina, "Numerical Soft Fault Simulation for Iterative Algorithm Development," *Naval Surface Warfare Center, Dahlgren Division Technical Report TR-18/280 (Under review)*, 2018.

[175] E. Coleman and M. Sosonkina, "Convergence and Resilience of the of the Fine-Grained Parallel Incomplete LU Factorization for Non-Symmetric Problems," in *Proceedings of the 2018 Spring Simulation Multiconference*, Society for Computer Simulation International, 2018.

[176] E. Coleman and M. Sosonkina, "Soft Fault Resilience for Fine-Grained Parallel Incomplete Factorizations," *Naval Surface Warfare Center, Dahlgren Division Technical Report TR-18/176 (Submitted)*, 2018.

[177] F. Jezequel, R. Couturier, and C. Denis, "Solving large sparse linear systems in a grid environment: the gremlins code versus the petsc library," *The Journal of Supercomputing*, vol. 59, no. 3, pp. 1517–1532, 2012.

[178] H. Anzt, E. Chow, and J. Dongarra, "Parilut—a new parallel threshold ilu factorization," *SIAM Journal on Scientific Computing*, vol. 40, no. 4, pp. C503–C519, 2018.

[179] T. A. Manteuffel, "An incomplete factorization technique for positive definite linear systems," *Mathematics of computation*, vol. 34, no. 150, pp. 473–497, 1980.

[180] I. S. Duff and J. Koster, "On algorithms for permuting large entries to the diagonal of a sparse matrix," *SIAM Journal on Matrix Analysis and Applications*, vol. 22, no. 4, pp. 973–996, 2001.

[181] M. Benzi and M. Tuma, "Orderings for factorized sparse approximate inverse preconditioners," *SIAM Journal on Scientific Computing*, vol. 21, no. 5, pp. 1851–1868, 2000.

[182] Innovative Computing Lab, "Software distribution of MAGMA." `http://icl.cs.utk.edu/magma/`, 2015.

[183] T. A. Davis, "The University of Florida Sparse Matrix Collection." `http://www.cise.ufl.edu/research/sparse/matrices/`, 1994.

[184] M. Naumov, "Incomplete-lu and cholesky preconditioned iterative methods using cusparse and cublas," *Nvidia white paper*, 2011.

[185] A. Nukada, H. Takizawa, and S. Matsuoka, "Nvcr: A transparent checkpoint-restart library for nvidia cuda," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 104–113, IEEE, 2011.

[186] X. S. Li and J. Demmel, "A scalable sparse direct solver using static pivoting.," in *PPSC*, 1999.

[187] A. Gupta, "Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices," *SIAM Journal on Matrix Analysis and Applications*, vol. 24, no. 2, pp. 529–552, 2002.

[188] E. Jensen, E. Coleman, and M. Sosonkina, "Using Modeling to Improve the Performance of Asynchronous Jacobi," in *Proceedings of the 24th annual International Conference on Parallel and Distributed Processing Techniques and Applications*, 2018.

[189] E. Jensen, E. Coleman, and M. Sosonkina, "Predictive Modeling of the Performance of Asynchronous Iterative Methods," *Journal of Supercomputing*, 2019.

[190] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[191] F. Iutzeler, P. Bianchi, P. Ciblat, and W. Hachem, "Asynchronous distributed optimization using a randomized alternating direction method of multipliers," in *Decision*

and Control (CDC), 2013 IEEE 52nd Annual Conference on, pp. 3671–3676, IEEE, 2013.

[192] J. Wolfson-Pou and E. Chow, "Distributed southwell: an iterative method with low communication costs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 48, ACM, 2017.

VITA

Evan Coleman
Department of Modeling & Simulation
Old Dominion University
Norfolk, VA 23529

Education

Doctor of Philosophy, Engineering with a concentration in Modeling & Simulation, Old Dominion University, 05/2019
Master of Science, Mathematics, Syracuse University, 05/2011
Bachelor of Science, Mathematics, Oregon State University, 03/2009

Employment

Scientist, Naval Surface Warfare Center, Dahlgren Division, Dahlgren, VA, 10/2011 - Present

Selected Publications

1. *Predictive Modeling of the Performance of Asynchronous Iterative Methods*, Erik Jensen, Evan Coleman and Masha Sosonkina, Journal of Supercomputing, 02/2019
2. *Simulation Framework for Asynchronous Iterative Methods*, Evan Coleman, Erik Jensen and Masha Sosonkina, Journal of Simulation Engineering, 06/2018
3. *Self-Stabilizing Fine-Grained Parallel Incomplete LU Factorization*, Evan Coleman and Masha Sosonkina, Sustainable Computing: Informatics and Systems, 02/2018

Awards & Grants

- Naval Surface Warfare Center Dahlgren Division In-house Laboratory Independent Research Grant (100k/yr), *Fault Tolerant Methods in Scientific Computing*, Fiscal Year 2017, 2018, 2019
- Naval Surface Warfare Center Dahlgren Division Academic Fellow, Fiscal Year 2016, 2019 (Additional funding in: Fiscal Year 2015, 2017, 2018)