DAHLGREN DIVISION
NAVAL SURFACE WARFARE CENTER
Dahlgren, Virginia 22448-5100

NSWCDD/TR-18/176

SOFT FAULT RESILIENCE FOR FINE-GRAINED PARALLEL INCOMPLETE FACTORIZATIONS

BY EVAN COLEMAN AND MASHA SOSONKINA

STRATEGIC AND COMPUTING SYSTEMS DEPARTMENT

JUNE 2018

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 07-09-2017	**2. REPORT TYPE** Technical	**3. DATES COVERED** *(From - To)* 1 Feb 2016 – 1 Feb 2018
4. TITLE AND SUBTITLE SOFT FAULT RESILIENCE FOR ASYNCHRONOUS INCOMPLETE FACTORIZATIONS		**5a. CONTRACT NUMBER**
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Evan Coleman, Masha Sosonkina		**5d. PROJECT NUMBER**
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) AND ADDRESS(ES) Naval Surface Warfare Center Dahlgren Division (Code A13) Dahlgren, VA 22448-5100		**8. PERFORMING ORGANIZATION REPORT NUMBER** NSWCDD/TR-18/176
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)		**10. SPONSOR/MONITOR'S ACRONYM(S)**
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release – distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This document describes methodologies and algorithmic modifications to the fine-grained incomplete factorization that provide the algorithm with resilience to the occurrence of soft faults. Several variants of the original algorithm are proposed and both symmetric and non-symmetric problems are considered.

15. SUBJECT TERMS
Resilience, fault tolerance, algorithm development, incomplete factorizations, preconditioning

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Evan Coleman
a. REPORT UNCLASSIFIED	**b. ABSTRACT** UNCLASSIFIED	**c. THIS PAGE** UNCLASSIFIED	SAR	XX	**19b. TELEPHONE NUMBER** *(include area code)* 540-653-5949

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

FOREWORD

Large scale simulations are used in a wide variety of application areas inside of the science and engineering domain to help forward the progress of innovation. Many large scale simulations spend the clear majority of their computational time attempting to solve large systems of linear equations; typically arising from discretizations of (systems 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 clusters involves constantly improving these iterative algorithms.

Being able to improve the iterative algorithms in use requires analyzing the combination of the algorithm itself, the problem domain, and future high-performance computing platforms. Many iterative algorithms that are used in large scale modeling and simulation efforts require the use of a preconditioner. Incomplete factorizations represent one of the most popular classes of preconditioners; due to both their popularity as standalone preconditioning routines, and for their use inside of more complex preconditioners where an incomplete factorization is used to approximately solve a linear system on a sub-domain.

This paper presents an investigation into the convergence and resilience properties for improving the resilience of the fine-grained parallel algorithm for computing incomplete factorizations. These techniques include various approaches to checkpointing as well as a study into the feasibility of using a self-stabilizing periodic correction step. Results concerning convergence with respect to the occurrence of soft faults, and the impact of any sub-optimality in the produced incomplete factors in Krylov subspace solvers are given. Numerical tests show that the simple algorithmic changes suggested here can ensure convergence of the fine-grained parallel incomplete factorizations and improve the performance of the resulting factors as preconditioners in Krylov subspace solvers in the presence of transient soft faults.

Approved by:

S. KYLE JONES, Head
Strategic and Computing Systems Department

CONTENTS

1.0 INTRODUCTION

Fine-grained methods are increasing in popularity recently due to their ability to be parallelized naturally on modern co-processors such as GPUs and MICs. Many examples of recent work using fine-grained parallel methods are available [CAD15, CP15, ACD15, ACHD16, LLH+17, AHBD18]. A specific area of interest is on techniques that utilize fixed point iteration, i.e.,

$$x = G(x) \qquad 1$$

for some vector *x* and map *G*. These methods can be computed in either a synchronous or asynchronous manner which helps tolerate latency in high-performance computing (HPC) environments.

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

$$A \approx LU \qquad 2$$

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

$$A \approx LL^T \qquad 3$$

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 [Saa03, Ben02]. This study examines a fine-grained parallel incomplete LU (FGPILU) algorithm originally proposed by [CP15] that can be used to generate either incomplete LU or incomplete Cholesky factorizations in a highly parallel fashion.

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 (e.g. level based incomplete factorizations [Saa03, Ben02]), of which the incomplete factorization generated by the FGPILU is representative of, to more complex routines. These include threshold-based incomplete factorizations such as ILUT [Saa03] where the non-zero pattern of the incomplete factors is chosen adaptively, or multilevel incomplete factorizations such as the Algebraic Recursive Multilevel Solver (ARMS) [SS02]. Several more complex variants of fine-grained factorization routines that attempt to improve the performance of the FGPILU algorithm studied here are under development [ACJ17, Inn15].

Looking forward to the future of high performance computing (HPC) environments, it is important to keep in mind the need for developing algorithms that are resilient to faults. On future platforms, the rate at which faults occur is expected to increase dramatically [CGG+09, CGG+14, ABC+06, GL09]. The expected increase in faults for future HPC systems is detailed in several studies including, e.g., [ABC+06, CGG+09, CGG+14, ABC+10b]. Because of this, developing algorithms that are resilient to faults is of paramount importance and fine-grained parallel methods are no exception. Faults can broadly be divided into two categories: hard faults and soft faults. [BFHH12].

- **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 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. The focus of this study is on the effect that soft faults might have on the FGPILU algorithm, 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.

In this study, 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. An important aspect of this work is that it analyzes the resilience of the algorithm with respect to soft faults and also proposes several variants of the factorization that are fault tolerant. These variants make use of traditional fault tolerance mechanisms such as checkpointing, more modern ideas such as algorithmic self-stabilization, as well as fine-grained checks that preserve the fine-grained nature of the original algorithm. This last avenue of research is particularly important since it preserves the high level of parallelism offered by the fine-grained nature of the FGPILU algorithm and allows for computationally efficient versions of the fault resilient variants of the original algorithm.

While results about the convergence of the FGPILU algorithm have been generated previously [CP15], the ability of the algorithm to converge for general problems (i.e. including problems that are non-symmetric and indefinite) has not been studied extensively. Another aspect of this work is that the convergence of the FGPILU algorithm is analyzed, building upon the initial convergence analysis presented in [CP15, CS18b], 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 (in this instance, this can be taken to mean symmetric and positive-definite) 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 algorithm so far has focused on matrices that are symmetric and positive-definite (SPD) [CP15, CAD15, CSC17], 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.

The main contributions of this work are analyzing the ability of the fixed point iteration at the heart of the FGPILU algorithm to complete successfully when attempting to solve problems under a myriad of different configurations, investigating how the convergence is affected by the occurrence of a soft fault, and demonstrating that the effects of a fault can be mitigated by the variants proposed herein. This work presents an extension of the work initiated in [CP15] with the proposal of easy to implement, fine-grained incomplete factorizations that are well suited for computation in an asynchronous environment. This paper presents an overview of several fine-grained incomplete factorizations, discusses the mathematical theory behind the convergence of such algorithms, proposes several variants of the original algorithms that are capable of converging despite the occurrence of soft computing faults, and provides extensive numerical results concerning the performance of these fine-grained incomplete factorizations with respect to faults. Portions of the results that are provided here have been previously published in [CSC17, CS18a, CS18b], and this technical report aims to collect all of these results in a cohesive manner with additional material provided as necessary to present as complete of a picture as possible.

The structure of this paper is organized as follows: in section 2, a brief summary of some related studies is provided. In sections 3 and 4, brief background information on both incomplete factorizations and fixed point iterations (respectively) that is relevant to the work presented here is provided. In section 5, an overview of the fine-grained parallel incomplete factorization algorithm itself is given. In section 6, a theoretical underpinning of the fine-grained parallel incomplete LU algorithm with respect to its convergence is explored. Section 7 provides an overview of the variants of the FGPILU algorithm that have been proposed for their resilience to soft faults, in section 8, a series of numerical results are provided, while section 9 concludes.

2.0 RELATED WORK

An initial look into fault tolerance for the FGPILU algorithm was provided in [CSC17] which was expanded on in both [CS18b] and [CS18a]. As pointed out in section 1, this report aims to collect all of the results from these three reports along with additional clarifying material.

Research into the convergence of iterative methods when solving non-symmetric problems has been studied previously [CS97, BHT00], and these studies were used as a starting point for the work presented here on non-symmetric problems. The effect of matrix reordering on convergence was studied there and has been focused on exclusively in papers such as [BSVD99].

The self-stabilizing variants of the FGPILU algorithm using a periodic correction step that are introduced here are inspired by the self-stabilizing iterative solvers presented in [SV13], which in turn are built upon the ideas of selective reliability [HH11, BFHH12]. The use of a periodic correction step is one alternative class of methods for fault tolerance that offers several advantages [SV13]. First, these methods provide a way to avoid the cost of checkpointing itself which has been suggested to be prohibitively high on future exascale platforms [CGG+09, CGG+14, GL09]. 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.

The work done in this study to show the effectiveness of iterative methods when using a possibly faulty FGPILU preconditioner on a Krylov subspace solver is done using the Conjugate Gradient (CG) algorithm [Saa03] for the case that the test problem is symmetric, and the GMRES algorithm [SS86, Saa93] in the case that the test problem is non-symmetric. The analysis of the potential performance of a Krylov subspace method using a potentially suboptimal FGPILU algorithm is related to the analysis in [SV13]. The results for the experiments conducted for this effort are presented similarly to the results in [CP15, CAD15], but with more of a focus on the impact that a soft fault can have on the execution of both the FGPILU algorithm itself and the performance of the generated factors in Krylov subspace methods.

Several numerically based fault models have been utilized in recent studies. These include a perturbation-based fault model that injects a random perturbation into every element of a key data structure [CS16b], and a numerical fault model that is predicated on shuffling the components of an important data structure [EHM15]. Other numerical models, such as inducing a small shift to a single component of a vector have been considered as well [BFHH12, HH11]. Comparisons between various numerical soft fault models have been made in [CS16a] and [CJB+17]. The fault model used in this paper is a combination of a modified version of the one initially developed in [CS16b] (related to the fault model developed in [EHM15] and [SW15]) that was used in [CSC17] and a simple model that flips bits directly; this combination has been used previously to study the resilience of fine-grained incomplete factorizations [CS18b]. A perturbation-based model similar to the one used in this study was used to develop fault tolerant variants of a fixed point iterative method in [SW15]. Details on the fault model used here are provided in section 8.2.

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 in both [BdS08] and [SSR11] while fault detection for iterative methods in linear algebra has been studied as well in [Che13, SKB12]. Fault tolerance for specific iterative methods has also been studied; see for example [HH11, BFHH12, SSR12, EHM14a, EHM14b]. Additionally, changes to the underlying parallel framework (e.g. MPI) have been considered as an alternative to direct modification of the algorithm under analysis; e.g., [BBH$^+$12, Bla12, FD00, FBD01, ZSK04].

Research into fault tolerance mechanisms and techniques specifically designed for fine-grained parallel methods is an area seeing increased research activity. An initial exploration of resilience for stationary iterative linear solvers (i.e. Jacobi) is given in [ADQO15] and expanded on in [ADQO16]. A more general exploration of fault tolerance for fine-grained methods is provided in [CS17].

3.0 INCOMPLETE FACTORIZATIONS

One major domain area for High Performance Computing (HPC) is sparse linear solvers, specifically Krylov subspace solvers. To help improve the performance of these solvers, a preconditioner is often used to help accelerate convergence [Ben02, Saa03]. One of the most commonly used classes of preconditioners is incomplete factorizations; the FGPILU algorithm studied here presents a novel way of calculating incomplete factorizations that may be more beneficial in some scenarios.

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

$$A = LU \tag{4}$$

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 [Ben02]) 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. 1. In this example, the initial matrix *A* is taken to be a three dimensional finite-difference approximation of the Laplacian,

$$\Delta u = f, \tag{5}$$

over a $50 \times 50 \times 50$ grid. Note that the number of non-zero terms increases from 3.3 million non-zero elements in *A* to 312.9 million non-zero elements in *both* L and U respectively after the factorization is performed. More drastic evidence of fill-in is 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,

$$A \approx LU. \tag{6}$$

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,

$$Ax = b, \tag{7}$$

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

$$LUx = b, \tag{8}$$

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. (7), into one that is easier to solve.

In particular, the linear system 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., $M = LU$.

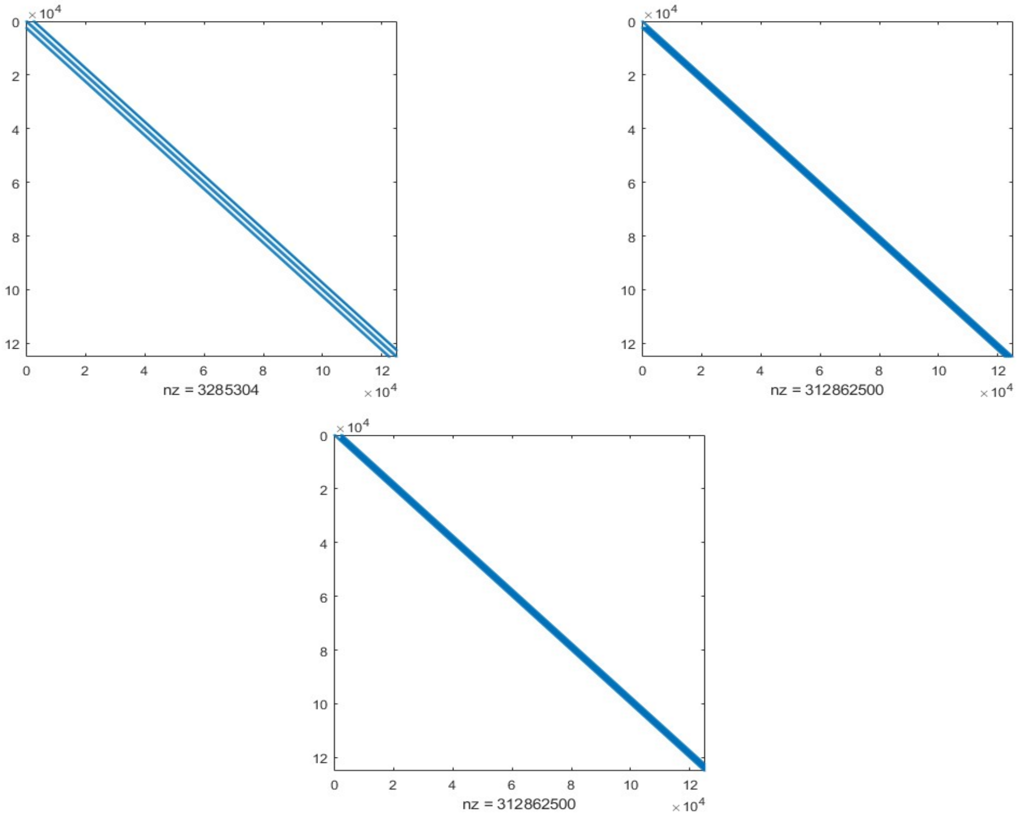


Figure 1 Example of the effects of fill-in during the Gaussian Elimination process. Note the relative sparsity of the input matrix *A* (left) compared to the factors *L* and *U* (middle and right respectively), especially in the total number of non-zero elements in each matrix.

Iterative methods for solving linear systems have become popular due to their efficacy at solving large sparse systems, such as those that arise from the discretization of partial differential equations. Popular iterative methods for large sparse systems include the Conjugate Gradient method if the input matrix A is symmetric, and GMRES for the case that A is non-symmetric. Additional details about the iterative solution of large sparse systems can be found in [Saa03].

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 1. 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.

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 [Saa94], ILUM [Saa96], BILUTM [SZ99], among many others)

Algorithm 1: Potential ILU algorithm

```
  Input: Input matrix $A$
1 for $i = 1, 2, \ldots, n$ do
2     for $k = 1, 2, \ldots, i-1$ and $(i, k) \in S$ do
3         $a_{ik} = a_{ik}/a_{kk}$
4         for $j = k+1, k+2, \ldots, n$ and $(i, j) \in S$ do
5             $a_{ij} = a_{ij} - a_{ik}a_{kj}$
6         end
7     end
8 end
```

The fine-grained parallel incomplete LU (FGPILU) factorization considered in this study can be thought of as a reformulation of the incomplete factorization process that offers a much higher degree of parallelism. Future HPC environments are likely to include a heterogeneous mixture of computing resources containing different types of accelerators (e.g., GPUs and MICs), and therefore algorithms that can take advantage of the computing structure of accelerators naturally will be advantageous. The FGPILU algorithm is an example of this kind of algorithm.

4.0 FIXED POINT ITERATION

Fixed point iterations are concerned with finding solutions to the iteration

$$x^{(k+1)} = G\left(\left(x^k\right)\right), \tag{9}$$

where $G : \mathrm{R}^n \rightarrow \mathrm{R}^n$ is composed of component-wise functionals g_i such that

$$x_1 = g_1(\boldsymbol{x}) \tag{10}$$
$$x_2 = g_2(\boldsymbol{x}) \tag{11}$$
$$\vdots$$
$$x_n = g_n(\boldsymbol{x}) \tag{12}$$

where the subscript represents the *component*, the iteration superscripts have been removed, and the (bold) vector notation is added to emphasize that each individual functional 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 functionals all utilize the same components of $\boldsymbol{x}$. In particular,

$$x_i^{(k+1)} = g_i\left(\boldsymbol{x}^{(k)}\right) \tag{13}$$

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

$$x_1^{(k+1)} = g_1\left(x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)}\right) \tag{14}$$
$$x_2^{(k+1)} = g_2\left(x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)}\right) \tag{15}$$
$$\vdots$$
$$x_n^{(k+1)} = g_n\left(x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)}\right) \tag{16}$$

On the other hand, in the asynchronous case processors will use the latest information available to them. There are several ways to define this mathematically (see, for example [FS00] or [MSV15]); informally, the data for each component, x_i, may not be from the iteration that just occurred. Under fairly standard assumptions about the amount of allowable delay [FS00, MSV15, BT89] for updates for the different components, convergence of many algorithms is preserved.

This will lead to different update patterns for each of the individual functionals, each of which will be utilizing components that are updated a different number of times. The convergence of parallel fixed point iterations is discussed in the literature for both the synchronous [AB05] and asynchronous [FS00] cases among many other sources [BT89, OR00, Bau78, Ben07]. 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. For the purposes of this study, this will be termed a *block asynchronous* update pattern.

5.0 FINE-GRAINED PARALLEL INCOMPLETE LU FACTORIZATION

The fine-grained parallel incomplete LU (FGPILU) factorization approximates the true LU factorization and writes a matrix *A* as the product of two factors *L* and *U* where,

$$A \approx LU \tag{17}$$

Normally, the individual components of both L and U are computed in a manner that does not allow easy use of parallelization (see section 3 for more details). The recent FGPILU algorithm proposed in [CP15] 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

$$(LU)_{ij} = a_{ij} \tag{18}$$

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 2) is defined by the following two nonlinear equations:

$$l_{ij} = \frac{1}{u_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}\right), \tag{19}$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}. \tag{20}$$

Following the analysis presented in [CP15], 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,

$$x^{(p+1)} = G\left(x^{(p)}\right), \tag{21}$$

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 2.

Keeping with the terminology used in [CP15, CAD15] 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 FGPILU 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 study, 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 [CP15, CSC17, CS18b, ACSD16]). Adopting a technique used in [CP15, CAD15, CSC17, CS18b], 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 [CP15], this diagonal scaling is imperative to maintain reasonable convergence rates for the algorithm, and the working assumption throughout this paper is that all matrices have been scaled appropriately.

Algorithm 2: FGPILU algorithm as given in [CP15]

```
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     for $(i, j) \in S$ do in parallel
3         if $i > j$ then
4             $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj})/u_{jj}$
5         else
6             $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$
7         end
8     end
9 end
```

6.0 CONVERGENCE OF THE FINE-GRAINED PARALLEL INCOMPLETE FACTORIZATION 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 7 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. (16) and eq. (17)) which when collected together as suggested by eq. (18) define a map,

$$\boldsymbol{G}: \mathbb{R}^{\boldsymbol{m}} \rightarrow \mathbb{R}^{\boldsymbol{m}}, \tag{22}$$

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 this corresponds to non-zero elements in the original input matrix, *A*. The following definition formalizes the concept of an ordering.

Definition 1. 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,...,*N*. Formally, this is a map $T : S \rightarrow 1,2,...,N$.

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 orders 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 2. 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®[*] 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 the vector $\boldsymbol{x}$ from section 4) needs to be defined. Call this ordering h. The ordering h will map a given pair of (i,j) coordinates 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 $\{1,2,3,...,m\}$ where $m = nnz(L) + nnz(U)$. That is,

$$x_{h(i,j)} = \begin{cases} l_{ij} & i > j \\ u_{ij} & i \le j \end{cases} \qquad 23$$

Given this, the two nonlinear equations that define the FGPILU factorization, i.e., eq. (19) and eq. (20), can be rewritten to account for this ordering. Doing this gives,

$$G_{h(i,j)} = \begin{cases} \dfrac{1}{x_{h(j,j)}}\left(a_{ij} - \displaystyle\sum_{1 \le k \le j-1} x_{h(i,k)} x_{h(k,j)}\right) & i > j \\ a_{ij} - \displaystyle\sum_{1 \le k \le i-1} x_{h(i,k)} x_{h(k,j)} & i \le j \end{cases} \qquad 24$$

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, $G'(x) = J\big(G(x)\big)$, where

$$J: \mathbb{R}^{|S|} \to \mathbb{R}^{|S| \times |S|} \qquad 25$$

and is defined by the partial derivatives of the map given by eq. (24). These partial derivatives are given by the following equations [CP15]:

$$\frac{\partial G_{h(i,j)}}{\partial x_{h(k,j)}} = -\frac{x_{h(i,k)}}{x_{h(j,j)}}, k < j \qquad 26$$

$$\frac{\partial G_{h(i,j)}}{\partial x_{h(i,k)}} = -\frac{x_{h(i,k)}}{x_{h(j,j)}}, k < j \qquad 27$$

[*] MathWorks, Inc., Natick MA

$$\frac{\partial G_{h(i,j)}}{\partial x_{h(j,j)}} = -\frac{1}{x_{h(j,j)}^2}\left(a_{ij} - \sum_{k=1}^{j-1} x_{h(i,k)} x_{h(k,j)}\right) \tag{28}$$

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:

$$\frac{\partial G_{h(i,j)}}{\partial x_{h(i,k)}} = -x_{h(i,k)}, k < i \tag{29}$$

$$\frac{\partial G_{h(i,j)}}{\partial x_{h(k,j)}} = -x_{h(i,k)}, k < i \tag{30}$$

Under the assumption that there is a single fixed point solution x^* of the nonlinear iteration defined by $G(x)$ in eq. (21), the following result given in Theorem 1 provides convergence for the nominal, fault-free version of the FGPILU algorithm:

Theorem 1 ([FS00]). *Assume that x^* lies in the interior of the domain of G and that G is F-differentiable at x^*. If $\rho\big(G'(x^*)\big) < 1$, 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 $\rho(G^0(x^*)) < 1$. The Gaussian Elimination partial ordering proposed in definition 2 leads to the following result from [CP15] that details the structure of mapping, G, defined by eq. (24):

Theorem 2 (Chow and Patel). *The function $G(x)$ with a Gaussian Elimination partial ordering has a strictly lower triangular form. Formally,*

$$G_k(x) = G_k(x_1, x_2, \dots x_{k-1}) \tag{31}$$

This leads to the following related result that also comes from Chow and Patel in [CP15]:

Theorem 3. *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.*

where this result can be combined with results from theorem 1 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 [CP15].

However, in order to determine if the mapping will converge from its current location in the domain of the mapping G defined by eq. (24) it is necessary to define what it means for a mapping to be a contraction:

Definition 3. The function G: $D \subseteq R^m \rightarrow R^n$ is a *contraction* on D if there exists a constant $\alpha < 1$ such that,

$$\left|\left|G(x) - G(y)\right|\right| \leq \alpha||x - y|| \qquad 32$$

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 4. The function G: $D \subseteq R^m \rightarrow R^n$ 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 $D \subseteq R^m$ is convex.

For the mapping G defined by eq. (24), the domain is not necessarily convex [CP15], but the norm of the associated Jacobian is still indicative of whether or not the corresponding fixed point iteration will converge [CP15].

With respect to the occurrence of a fault, the fault model proposed in this study 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
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 1 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

[CAD15, CSC17]. One notable exception is the 2D convection-diffusion problem that is presented in [CP15]. The problem,

$$-\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+\beta\left(\frac{\partial \mathrm{e}^{\mathrm{xy}} u}{\partial x}+\frac{\partial \mathrm{e}^{-\mathrm{xy}} u}{\partial y}\right)=g \qquad 33$$

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 [CP15], 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®[*]. Because of this, in the testing on non-symmetric problems that is presented here (previously captured in [CS18a]), the same SYMAMD reordering is included in the experiments.

6.1 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 8 these techniques are only applied to the more difficult problems featured in section 8.4.

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 resulting L and U factors to actually slow convergence or prevent convergence entirely (see both Table 7 and [Man80]).

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

[*] MathWorks, Inc., Natick MA

conventional incomplete LU factorizations, or else 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 [DK01]. 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®[*]. As stated before, this reordering has previously been observed to help convergence of the FGPILU algorithm on non-symmetric problems [CP15] and has also seen success with conventional incomplete LU factorizations for non-symmetric and indefinite problems [BHT00]. 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 [CAD15, CP15, CSC17, CS18b].

After the ordering is applied, the second technique consists of an α-shift that is performed in the manner originally suggested in [Man80]. Specifically, the original input matrix A can be written

$$A = D - B \qquad 34$$

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

$$\hat{A} = (1 + \alpha)D - B \qquad 35$$

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 6 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 3.

[*] MathWorks, Inc., Natick MA

Algorithm 3: Modified FGPILU algorithm for non-symmetric and indefinite matrices

```
   Input: Input matrix $A$, shift factor $\alpha$
   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: $A = (1 + \alpha)D - B$
 4 Generate initial guesses for $L$ and $U$ from the reordered and scaled
    input matrix $A$
 5 for $sweep = 1, 2, \ldots, m$ do
 6     for $(i, j) \in S$ do in parallel
 7         if $i > j$ then
 8             $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj})/u_{jj}$
 9         else
10             $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$
11         end
12     end
13 end
```

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 8.4.

7.0 SOFT FAULT RESILIENCE FOR FINE-GRAINED INCOMPLETE FACTORIZATIONS

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 8.

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 [ABC+10a, ABC+10b, CGG+09, CGG+14, ABC+06]. 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 [SV13], 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 4.

```
Algorithm 4: 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 $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$
8          end
9      end
10 end
```

In the prototypical self-stabilizing algorithm provided by algorithm 4, 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 [HH11, BFHH12] 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 6, convergence of the FGPILU algorithm is strongly related to the Jacobian of the functional iteration, G (i.e. eq. (24)). 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 f^{th} 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 naive 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 a 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 5.

Algorithm 5: 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 $sweep = 1, 2, \dots, m$ 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          end
9      else
10         for $(i, j) \in S$ do in parallel
11             if $i > j$ then $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj})/u_{jj}$
12             else $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$
13         end
14     end
15 end
```

Note that, while the algorithm presented in Algorithm 5 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 naive check presented in Algorithm 5.

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 5 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 [CAD15, CP15] mitigates this issue somewhat since the original guess can always be reused, but if a higher level of fidelity is desired than the runtime could be prohibitively long.

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

Theorem 4. *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 5 will converge.*

Proof. Since the Jacobian at the fixed point of the algorithm has spectral radius less than 1 (see [FS00]) and the correcting step of Algorithm 5 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 5 will converge. □

While the method proposed by Algorithm 5 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 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 [CP15] can be used:

Theorem 5. (Chow and Patel) *Given a matrix A and G as defined above, the 1-norm of the current iterate G_i' can be bounded by,*

$$||G_i'||_1 \leq \max\left(||U_i||_\infty, ||L_i||_1, \left|\left|R_i^L\right|\right|_1\right) , \qquad 36$$

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

$$T_{ij} = \begin{cases} (LU)_{ij} & (i,j) \in S \\ 0 & o/w \end{cases} . \qquad 37$$

However there is still a larger than desirable computational burden in forming the matrix

$$R = A - T, \tag{38}$$

and the bound itself may not be sharp enough for practical use since the result is only useful if,

$$\alpha = \max\left(||U_i||_\infty, ||L_i||_1, \left||R_i^L\right||_1\right) < 1. \tag{39}$$

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 5 simplifies further to the result provided below in Theorem 6.

Theorem 6. (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,*

$$||G_i'||_1 = \max\left(||U_i||_{max}, ||L_i||_{max}, ||A_L||_1\right), \tag{40}$$

where A_L *is the strictly lower triangular part of A.*

Development of a periodic correction step based upon explicit calculation of the Jacobian (or that utilizes properties of the Jacobian as discussed above) is left as future work. The following subsections develop a spectrum of resilient variants of the FGPILU algorithm based upon other ideas. Development of a traditional checkpointing variants will be examined in the next subsections section 7.1, while development of a checkpointing variant that attempts to leverage the fine-grained nature of the FGPILU algorithm is provided in section 7.2. The use of a periodic correction step will be examined in the following two subsections: section 7.3 provides a computationally light variant designed around the performance of the algorithm on finite-difference discretization of partial differential equations, and section 7.4 provides a checkpoint free variant based upon the progression of a residual.

7.1 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. Using the fault model described in section 8.2, if a fault occurs at the computation of the *k*th iterate (affecting the outcome of the (*k* + 1)st vector), it is possible to write the corrupted (*k* + 1)st iteration of *x* as

$$\hat{x}^{(k+1)} = G\left(x^{(k)}\right) + r \tag{41}$$

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 [CP15] and [CAD15] to monitor the nonlinear residual norm. This is a value

$$\tau = \sum_{(i,j)\in S} \left| a_{ij} - \sum_{k=1}^{\min(i,j)} l_{ik}u_{kj} \right| \tag{42}$$

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. 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 Apache problem (see Table 4 for descriptions of the example problems) are given in Table 1. Note that the nonlinear residual norm will continue decreasing, but that the ILU residual quickly settles to a non-zero value.

Table 1: 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 1 is a three dimensional finite-difference discretization of partial differential equations that is one of the best conditioned problems from the “easier” problem set Table 5. Alternatively, Table 2 shows the nonlinear residual progression only for the Apache2 problem featured above, the offshore problem (which is the most ill-conditioned problem from the first problem set), and the two non-symmetric problems that are studied more extensively in the “difficult” problem set. 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.

If a fault occurs on a given sweep, then one or both nonlinear equations from the FGPILU algorithm (c.f. Algorithm 2) will have some amount of error. In particular, the update equations for l_{ij} and u_{ij} will become

$$l_{ij} = \frac{1}{u_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}\right) + r_{ij} \qquad 43$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj} + r_{ij} \qquad 44$$

where r_{ij} represents the component of the vector r that maps to the (i,j) location of the matrix. Comparing eq. (32) and eq. (31) shows that, if a fault occurs during the computation of the incomplete LU factors, then the nonlinear residual norm τ will be affected.

Table 2: 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

In order to ensure that a fault does not negatively affect 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 6.

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).

Algorithm 6: Checkpoint-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 $A \approx LU$
for $sweep = 1, 2, \ldots, m$ do
    if $Fault$ then
        Rollback $L$ and $U$
        $Fault$ = FALSE
        $sweep = sweep - 1$
    else
        for $(i, j) \in S$ do in parallel
            if $i > j$ then $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj})/u_{jj}$
            else $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$
            $\tau_{ij} = \left|a_{ij} - \sum_{k=1}^{\min(i,j)} l_{ik}u_{kj}\right|$
            if $\tau^{(sweep)} > \gamma \cdot \tau^{(sweep+r)}$ then
                $Fault$ = TRUE
            end
        end
    end
end
```

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.

7.2 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

$$\tau_{ij} = \left|a_{ij} - \sum_{k=1}^{\min(i,j)} l_{ik}u_{kj}\right| \quad 45$$

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

$$\tau_{ij} > t \qquad 46$$

since the individual nonlinear residual norms are generally decreasing as the FGPILU algorithm progresses. Set the value *t* as *t* = max(τ_{ij^*}) initially (line 4 of Algorithm 7), 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 7.1, a fault can be declared if,

$$\tau_{ij}^{sweep} > \gamma \tau_{ij}^{sweep-r} \qquad 47$$

for similar parameters γ and *r*.

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 6 to 10 of Algorithm 7).

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 16 of Algorithm 7). This algorithm is detailed in Algorithm 7.

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 [CP15, CAD15] 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 12 of Algorithm 7 extends over all elements (*i*,*j*) ∈ *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

7. 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.

Algorithm 7: 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 $A \approx LU$
for $(i,j) \in S$ do in parallel
    $\tau_{ij} = \left| a_{ij} - \sum_{k=1}^{\min(i,j)} l_{ik}u_{kj} \right|$
end
$t = \max(\tau_{ij})$
for $sweep = 1, 2, \ldots, m$ do
    if $Fault$ then
        Set $i = \max_{i,j}(k_{ij}^1)$ and $j = \max_{i,j}(k_{ij}^2)$
        Rollback $\{l_{ik}\}_{k=1}^{i-1}$ and $\{u_{kj}\}_{k=1}^{j-1}$
        $Fault$ = FALSE
        $sweep = sweep - 1$
    else
        for $(i,j) \in S$ do in parallel
            if $i > j$ then $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj})/u_{jj}$
            else $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$
            Compute $\tau$
            if $\tau_{ij} > t$ and $\tau' > 0$ then
                Set $k_{ij}^1 = i$ and $k_{ij}^2 = j$
                $Fault$ = TRUE
            end
        end
    end
end
```

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,

$$||\boldsymbol{r}|| \leq \boldsymbol{t} \qquad 48$$

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 16 of Algorithm 7. This, in turn, affects the update equations eqs. (30) and (32).

7.3 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 5), 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 8. 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 8 was designed to correct problems arising from simple finite difference discretizations of partial differential equations (i.e. L2D and APA from Table 4). The technique of observing the magnitude of the elements used in the fixed point iteration and their relative changed 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 7.5 and section 8 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).

Algorithm 8: 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
       $\beta$ to determine the strictness of the component level check
Output: Factors $L$ and $U$ such that $A \approx LU$
for $sweep = 1, 2, \dots, m$ do
    if $sweep \equiv 0 \mod F$ then
        for $(i,j) \in S$ do in parallel
            if $\{\|l_{ij}\|, \|u_{ij}\|\} \gg \|a_{ij}\|$ or $|\{l_{ij}, u_{ij}\} - a_{ij}|/|a_{ij}| > \beta$ or
              $\{l_{ij}, u_{ij}\} = \{0, \mathtt{NaN}\}$ then $\{l_{ij}, u_{ij}\} = a_{ij}$
            if $i > j$ then $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj})/u_{jj}$
            else $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$
        end
    else
        for $(i,j) \in S$ do in parallel
            if $i > j$ then $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj})/u_{jj}$
            else $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$
        end
    end
end
```

The following result establishes a convergence property for the variant of the FGPILU algorithm proposed in Algorithm 8.

Theorem 7. *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 8 will converge.*

Proof. This follows from noticing that the correcting (or "stabilizing") step (lines 2 to 7 of Algorithm 8) 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 [CP15]) which rely upon the assumptions and base arguments from [FS00]. □

7.4 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. (34)), in a manner similar in spirit to Algorithm 7. Recall from section 7.1 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 9.

Algorithm 9: Component-Wise Residual Check for FGPILU (CW-FGPILU)

```
Input: Initial guesses for $l_{ij} \in L$ and $u_{ij} \in U$, parameters $F$ and $\alpha$ that
       define the frequency and strictness of the periodic correction
       step respectively
Output: Factors $L$ and $U$ such that $A \approx LU$
1  for $sweep = 1, 2, \ldots, m$ do
2      if $sweep \equiv 0 \mod F$ then
3          for $(i, j) \in S$ do in parallel
4              if $\tau_{ij}^{sweep} > \gamma \cdot \tau_{ij}^{sweep-F}$ 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          end
9      else
10         for $(i, j) \in S$ do in parallel
11             if $i > j$ then $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj})/u_{jj}$
12             else $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$
13         end
14     end
15 end
```

The CW-FGPILU algorithm variant (Algorithm 9) can be seen as a modified version of the partial checkpointing method that is utilized Algorithm 7, 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 τ 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 number of false positives that the algorithm reports. The periodicity of the check is controlled by the parameter F. The smaller that 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, allowing the algorithm to apply 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 [CP15]). 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.

7.5 Notes on the Convergence of the FGPILU Variants

The main result concerning the convergence of the FGPILU algorithm comes from [FS00], 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. (24). 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 [CP15, CAD15] 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. It is therefore possible that any adverse effects 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, etc) that the computed factors are used as preconditioner for. These potential impacts will be explored numerically in section 8.

8.0 NUMERICAL RESULTS

8.1 Experimental Set-up

The experiments for this study were all conducted on the Turing High Performance computing cluster at Old Dominion University. For the experiments with symmetric matrices, a NVIDIA Tesla®[*] K40m GPU was used, while for the experiments featuring the non-symmetric problem set a NVIDIA Tesla®[4] K80 GPU was used. The nominal, fault-free iterative incomplete factorization algorithms and iterative solvers were taken from the MAGMA open-source software library [Inn15], 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. All of the results provided in this study reflect double precision, real arithmetic.

8.2 Fault Model

Soft faults typically manifest as bit-flips, and much of the current research (e.g., [BdS08]) treat faults exclusively as bit flips. This reflects the current method in which faults occur, and covers the *current* projected behavior for future HPC platforms. However, it is important when looking forward towards producing fault tolerant algorithms for future computing platforms not to become too dependent on the precise mechanism that is used to model the instantiation of a fault. Regardless of how a fault manifests in future hardware, the result of a fault will be a corruption of the data that is used by the algorithm. To this end, two different fault injection methodologies were used. The first fault injection technique relies on flipping bits directly, as is typically done in studies concerning fault tolerance of algorithms. Additionally, a generalized numerical scheme for simulating the occurrence of a fault is adopted in an attempt to provide a more complete look at how the FGPILU algorithm and the variants proposed here respond to data corruption.

The numerical scheme for simulating a fault that is used here attempts to model faults as perturbations to the data being used by the algorithm. This style of fault model has been used previously (see, for example, [SW15, CS16a, CJB$^+$17, CS17, CS18a, CS18b]) to model the occurrence of a fault and a slightly modified version of these models is used here. The results presented here utilize the perturbation-based model used in [CSC17, CS18b] where the modified model targets a single data structure and injects a small, uniformly sized random perturbation into each component transiently. This is contrast to the perturbation based model used in [CS16b], where the perturbation was injected continuously, and is different from [SW15] since the perturbations that are injected are uniformly sized for a single run.

To better define the perturbation-based soft fault model used here, consider the following. If the targeted data structure is a vector *x* and the maximum size of the perturbation-based fault is, then the simulation of a fault using this methodology proceeds as follows:

[*] NVIDIA Corp., Santa Clara CA

1. Generate a random number $r_i \in (-\epsilon, \epsilon)$ for every component x_i, where i ranges over entire length of x.

2. Set $\widehat{x_i} = x_i + r_i$ for all i's.

The resultant vector $\hat{x}$ is, thus, perturbed away from the original vector x. Note that the sign of x_i is not taken into account, and therefore, $||x||_2 \approx ||\hat{x}||_2$. After a fault occurs, it is possible for an algorithm to detect the error caused by the perturbation and correct it.

In addition to the numerical scheme discussed above, additional runs were conducted where a bit was flipped directly in the data structure in question. The advantages of combining these two distinct methods are as follows:

- The perturbation-based model shows resilience of the proposed algorithmic variants to consistently sized errors as well as evaluating the FGPILU variants against any numerical instability.
- The bit-flip mode shows how robust the algorithms are with respect to potentially large changes (e.g. flips in sign or large exponent bits [EMSW13]).
- The perturbation-based model injects a fault into all of components of the FGPILU update, whereas the bit-flip model only corrupts a single entry
    - This duality stresses two opposing features of the fine-grained nature of the algorithm.

As pointed out in [EHM15], one of the advantages of numerical soft-fault models are that they are able to consistently correspond to a "sufficiently bad" impact of a soft fault. This stresses the fault tolerant variants of the algorithm being studied. Explorations of the similarities and differences between the numerical soft fault model presented in [EHM15] and the perturbation based model are presented in [CS16a, CJB+17]. Simulating the occurrence of a soft fault using a numerical soft fault model may force iterative algorithms to run consistently through bad errors only. However, the direct injection of bit-flips ensures that the "worst case" scenario was also captured fully. Furthermore, by varying the size of the perturbation, it is possible to produce errors a desired level of impact.

In this study, faults are injected into the FGPILU algorithm following the combined methodology described above. Due to the relatively short execution time of the FGPILU algorithm on the given test problems, a fault is induced only once during each run, at a random sweep before convergence. Three fault-size ranges (corresponding to differing orders of magnitude) for the faults injected by the perturbation-based model were considered:

$$r_i \in (-0.01, 0.01) \tag{49}$$
$$r_i \in (-1.00, 1.00) \tag{50}$$
$$r_i \in (-100, 100) \tag{51}$$

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 bitflip model (BF) are presented separately, but as averages over all trials executed for each methodology.

Note: The working assumption in this study 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.

8.2.1 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).

To illustrate the potential impact of each fault model for the FGPILU algorithm under study in this paper, 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 4). Recall that all of the matrices in this study 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, $x\hat{}$, and the norm of the difference in these two quantities,

$$d = \left|\left|x - \hat{x}\right|\right| \qquad 52$$

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 3.

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.

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

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. 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 the paper are capable of handling large errors.

8.3 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 [Dav94], and the matrices selected for this study are the same as the ones that were selected for the studies [CAD15, CSC17, CS18b] 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®[*] for testing the incomplete LU factorization that is part of the CUDA®[†] library [Nau11].

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 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 × 50 × 50 mesh.

All of the matrices considered in this portion of the study 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 that each of the eight matrices used in this

[*] Nvidia Corporation, Sunnyvale CA
[†] NVIDIA Corp., Santa Clara CA

study 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 4.

Plots of where the non-zeros are located in the matrix are provided for all eight matrices in fig. 2 for the case where the matrices are unordered, and in fig. 3 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. 3 relative to the dispersal of non-zero elements shown in fig. 2. This reordering was shown to be effective for similar matrices with respect to the convergence of the FGPILU algorithm in [CAD15, CP15, CSC17, CS18b]

Table 4: Summary of the 8 symmetric positive-definite matrices used in this study. Descriptions come from [Dav94].

Matrix Name	Abbreviation	Dimension	Nonzeros	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

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.

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} =

* MathWorks, Inc., Natick MA

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 [Inn15].

Table 5: 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

Finally, note that the implementation of the variants that was examined in this paper is not 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 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. 7 (right) and fig. 8 (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. [NTM11], etc) and extended performance analysis would be needed to produce performance-oriented prototypes of each of the variants.

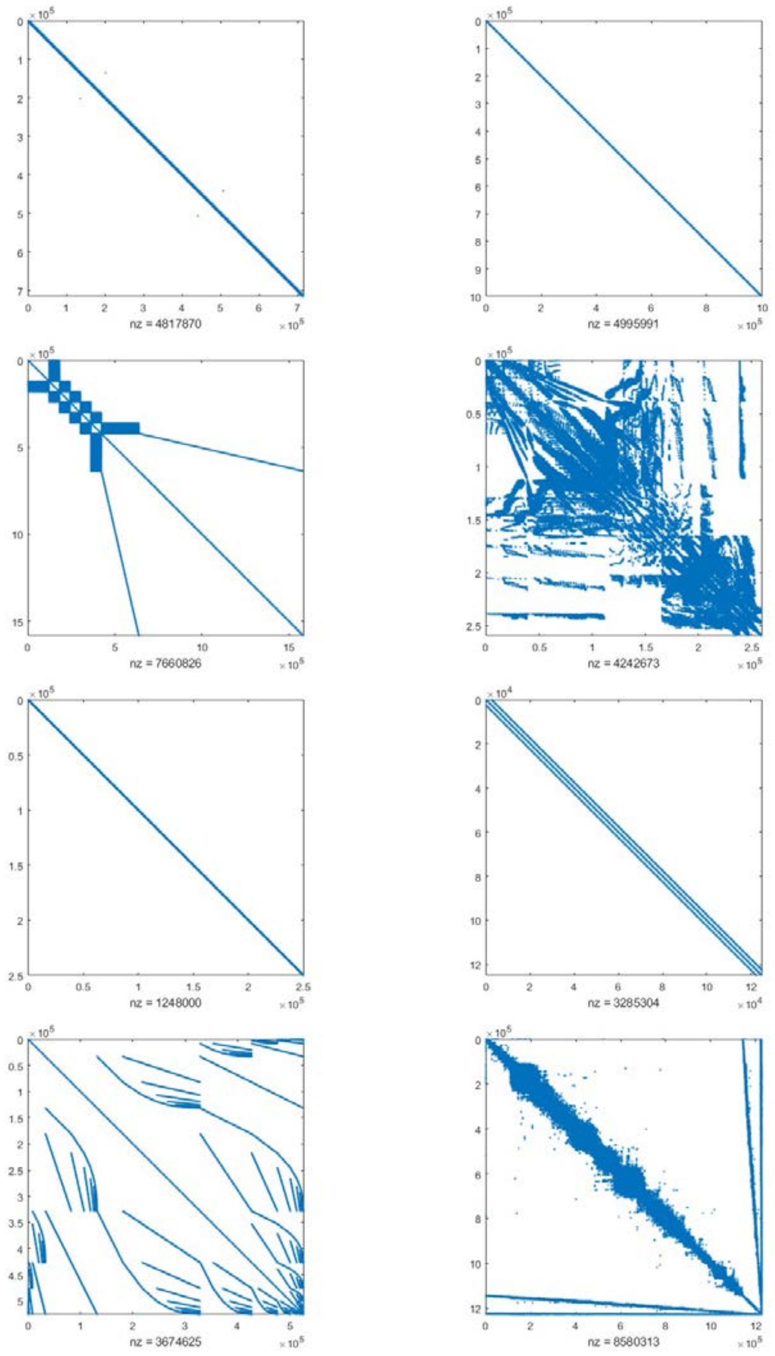


Figure 2: Sparsity plots showing the location of all non-zeros for each of the 8 matrices with no reordering applied that were considered in the first set of experiments.

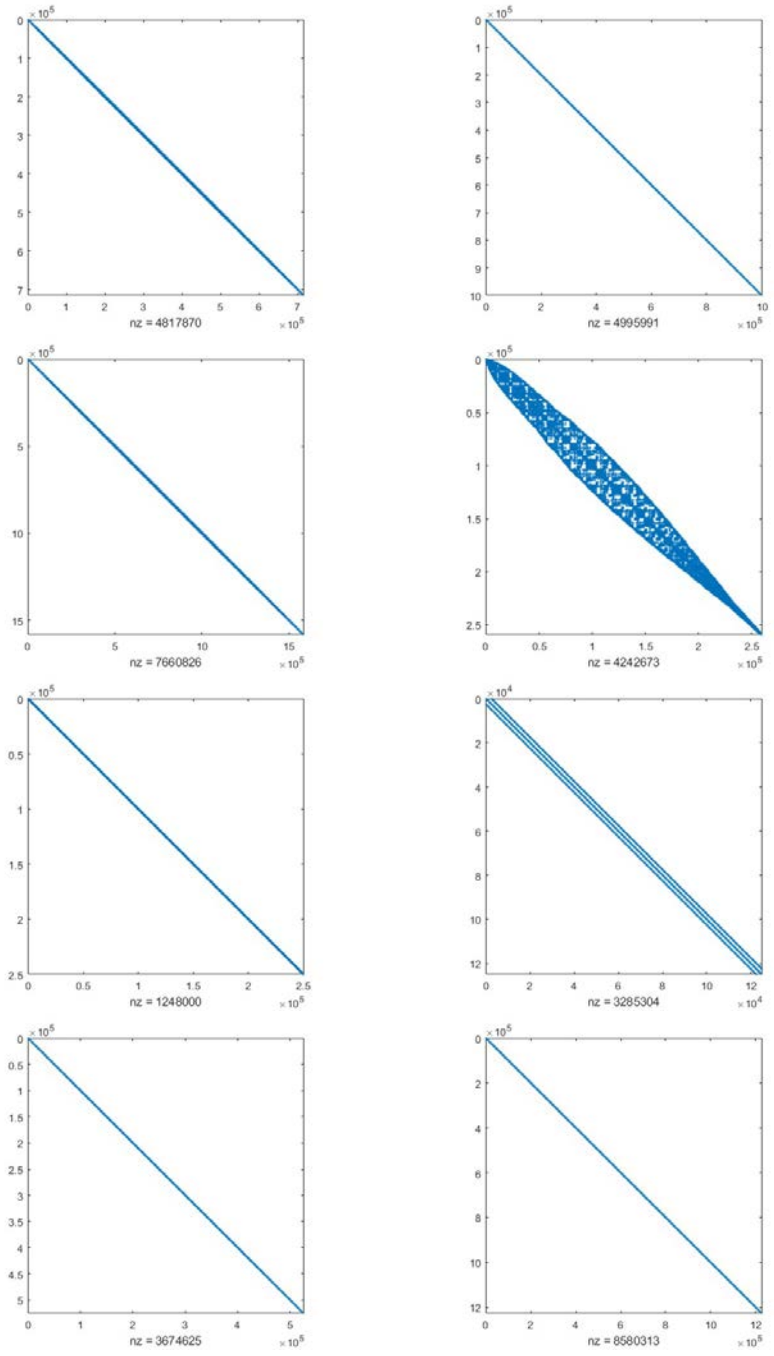


Figure 3: Sparsity plots showing the location of all non-zeros for each of the 8 matrices with the Reverse Cuthill-Mckee (RCM) reordering applied that were considered in the first set of experiments

8.3.1 Convergence of FGPILU algorithm

For the purposes of this study, 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. 4 (left), which is a as an example of the typical progression of the nonlinear residual norm as the algorithm progresses towards convergence.

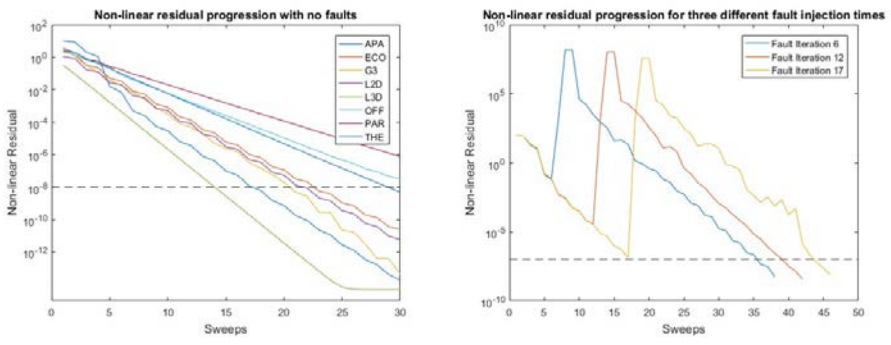


Figure 4: The progression of the nonlinear residual for 30 sweeps of a typical fault-free run for each of the 8 test problems (left). The progression of the nonlinear residual for the Apache test problem for three different fault injection times and fault sizes in the (-1,1) range (right). The horizontal dashed line indicates the FGPILU convergence tolerance of 10^{-8}.

To illustrate the potential impact of a fault, fig. 4 (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 Apache test problem converges to the desired level of nonlinear residual in 20 iterations when faults are not present.

From fig. 4 (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. 4 (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. 4 (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 6 (CPA),
- the CP-FGPILU which rolls back a single row and column of the L and U factors and is described in Algorithm 7 (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 8 (SS), • the periodic correction step based on checking component-wise progression of the individual nonlinear residuals, τ_{ij} which is given in Algorithm 9 (CW).

8.3.1.1 Perturbation-Based Faults

This section examines the effects of a soft fault (modeled as a perturbation as described in section 8.2) on the FGPILU algorithm and the variants discussed throughout the paper. 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. 5.

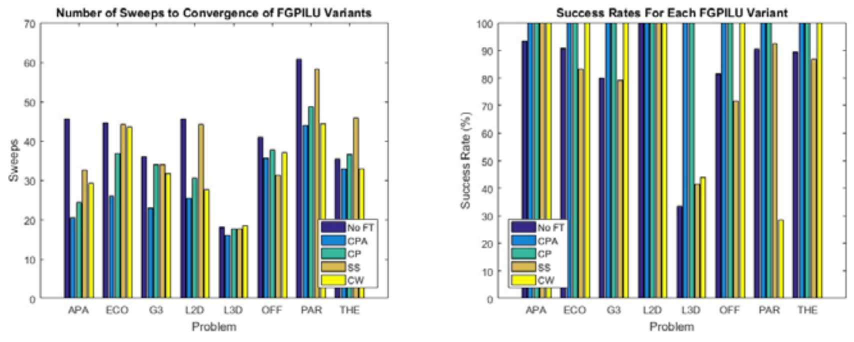


Figure 5: For perturbation-based faults (PBSFM): the number of sweeps required for convergence for each of the 8 test problems (left). The percentage of runs that produced a preconditioner that corresponded to a successful PCG solve (right).

Figure 5 (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 study, they are more effective at mitigating faults.

The small degradation in the number of sweeps to convergence depicted in fig. 5 (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. 5 (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 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. 5 or fig. 6, but that this run would cause the success rates captured in the right of fig. 5 and fig. 6 to decrease.

For the FGPILU variants tested, the success rates captured in fig. 5 (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. Finding a way to obtain optimal parameters for the FGPILU algorithm variant utilizing the periodic correction step featured in algorithm 8 efficiently from intrinsic properties of the linear system in question is left as future work. 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.

8.3.1.2 Bit-Flip Faults

This section provides 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. 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.

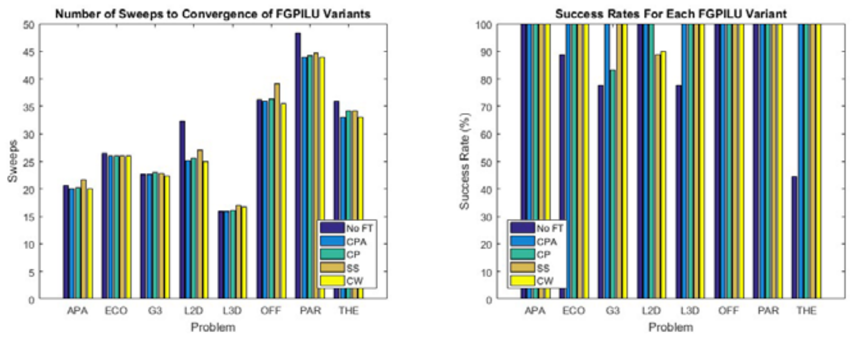


Figure 6: For bit-flip (BF) faults: the number of sweeps required for convergence for each of the 8 test problems (left). The percentage of runs that produced a preconditioner that corresponded to a successful PCG solve (right).

Figure 6 (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. 5. The success rates for the trials run with bit flips (see fig. 6 (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 very little or catastrophic impact. This is seen when comparing fig. 5 and fig. 6 in that in the number of sweeps taken until convergence on the successful runs (i.e. the top 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.

8.3.2 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 8.3.1. 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.

8.3.2.1 Perturbation-Based Faults

Figure 7 captures only the cases in which a preconditioner was successfully prepared (c.f. fig. 5 (right)). Figure 7 (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. 7 (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.

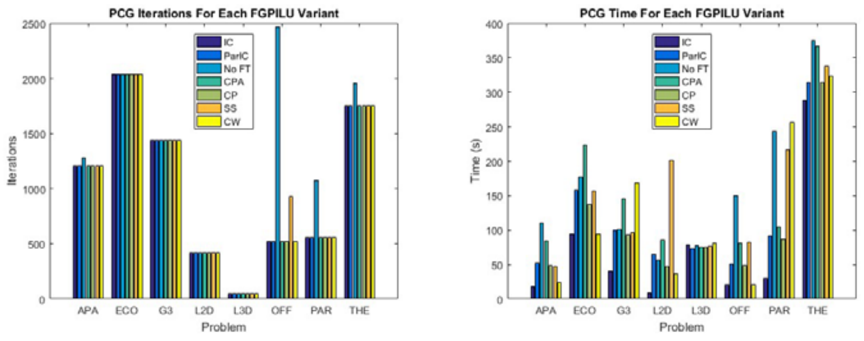
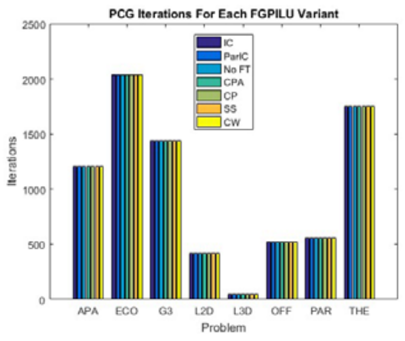


Figure 7: For perturbation-based faults (PBSFM): the number of iterations required for successful PCG solves for each of the 8 test problems (left). The time required for successful PCG solves for each of the 8 test problems (right).

The timing results presented in fig. 7 (right) are for the total time required for the preconditioner preparation and PCG solve. While the former may vary much 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 9 and represented by CW may be one of the more efficient variants.

8.3.2.2 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 other fault injection methodology described in section 8.2 - are explored at the level of timing and accuracy results in the corresponding PCG solve.



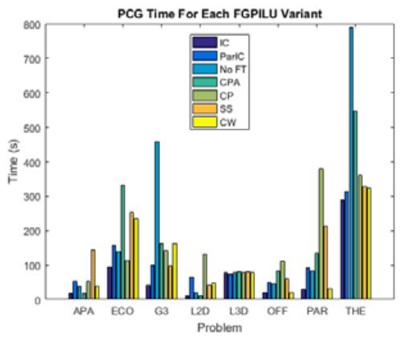


Figure 8: For bit-flip (BF) faults: the number of iterations required for successful PCG solves for each of the 8 test problems (left). The time required for successful PCG solves for each of the 8 test problems (right).

Figure 8 (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. 7 (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. 8 (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. 7 (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.

8.3.3 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. 5 and fig. 6 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.

8.3.3.1 Highlights of FGPILU variant differences

The component-wise check put forth in Algorithm 9 (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 7) 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. 5 and fig. 6), 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 this paper. 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.

8.3.3.2 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 6), 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 9 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 7.4 for their discussion).

8.4 Results for non-symmetric matrices

This section attempts to provide a set of results complementary to what was presented in section 8.3, by examining problems that are more difficult to solve. The test problems that were used in this portion of the study 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 [CSC17, CAD15, CP15], however, solving fixed point iterations that feature nonlinear functionals (i.e., in Algorithm 2) is often difficult. Developing the associated convergence theory, especially results that carry practical meaning, is also typically hard to accomplish (see for example: [BT89, OR00]).

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 [CS97], fs_760_3. The next matrix comes from the domain of circuit simulation, ecl32, and has been studied previously [LD99, Gup02]. The last matrix comes from the set of 8 SPD matrices that were studied in section 8.3, 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 6.

Table 6: 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 engineerin g
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 with respect to matrices that are

[*] MathWorks, Inc., Natick MA

more challenging computationally than the problems that are featured in the majority of the previous work on the algorithm (i.e. [CP15, CAD15, CSC17, CS18a, CS18b]).

Lastly, it is important note that many other problems from both [CS97], [BHT00], [Dav94], 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.

8.4.1 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 [Ben02] or [Saa03] for a clear description of levels of incomplete LU factorizations), and three different values of α in the α-shift described in section 6.1. 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 study 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 7. 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 that both the number of iterations and the time required for convergence of the Krylov subspace solver are greatly increased.

Table 7: 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 that were tested: 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 8.

Table 8: 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 [BHT00, CS97], 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 9.

Table 9: 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

8.4.2 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 6. 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 7.5 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 6, $\tau^{(sweep)} > \gamma \cdot \tau^{(sweep+r)}$, 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 [CSC17, CS18b], 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 study.

A summary of the data found in these experiments is provided in Table 10, 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 CP) were employed, respectively. Three ratios of the results with CP and NoFT are shown in Table 10 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. 9.

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.

Table 10: Solver performance using FGPILU with no fault tolerance (NoFT) and checkpointing (CP).

Scenario	Success Rate (NoFT)	Success Rate (CP)	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

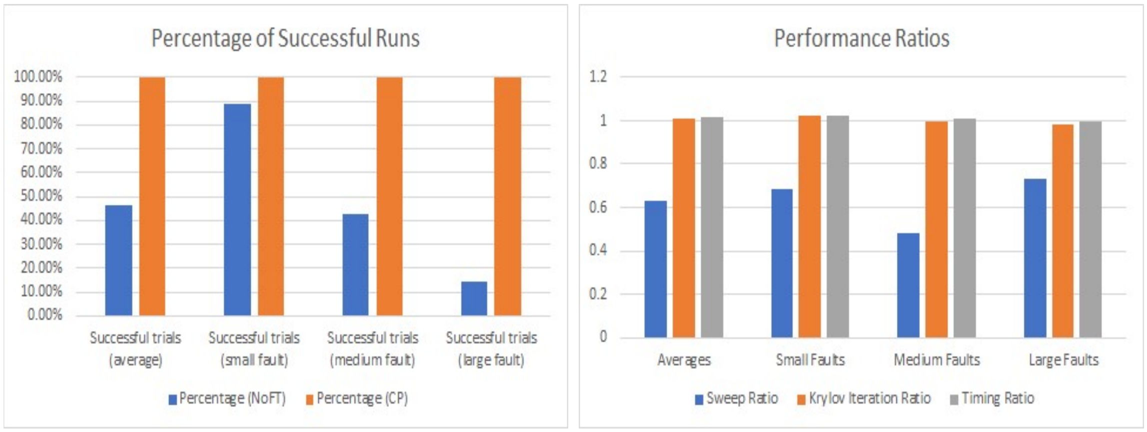


Figure 9: Percentage of successful runs for no fault tolerance (NoFT) and checkpointing (CP) (left), ratios showing relative performance of the checkpointing variant to the nominal FGPILU algorithm (right).

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 study, 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.

9.0 SUMMARY & FUTURE WORK

This study has examined the potential impact of soft faults on the FGPILU algorithm, and proposed several variants to remedy the impact that these faults may have. 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 discussed in this paper 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.

In the future, 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 between them. It would also be advantageous to further explore the convergence of the FGPILU algorithm – both subject to faults, and in a fault-free environment – in more diverse problem domains. The vast majority of the problems that have been studied in the majority of the work on the FGPILU algorithm (i.e. [CP15, CAD15, CSC17, CS18b], and the majority of the effort showcased in this report) have all explored problems that tend to be similar in nature. This study has presented some experiments and analysis concerning the convergence and resilience of the FGPILU factorization with respect to more difficult problems, but this avenue of research could be further explored to give a more complete picture of the performance of the algorithm over a wide range of potential use cases.

Moving forward, further adaptations to the FGPILU algorithm are possible. This series of experiments has worked with level-based incomplete LU factorization, but for many problems throughout science and engineering there are more effective strategies for selecting which non-zero elements to allow in the incomplete factors. A variant of threshold based incomplete LU factorization (in the conventional case, ILUT [Saa94]) that takes advantage of fine-grained parallelism called ParILUT [ACJ17] has been proposed recently that may prove more effective for non-symmetric and indefinite matrices such as those studied in section 8.4. It would be also helpful to compare the performance of various solvers, such as Bi-CGSTAB and TFQMR, with the preconditioning factors that are generated by the FGPILU algorithm.

Another avenue for future exploration includes expanding the experimentation conducted with respect to the parameters used in the various fault tolerance techniques. In particular, the parameters could be further analyzed in an attempt to optimize the balance of the increased computational cost with the increased level of resilience against the impact of a soft fault. Most of the experiments shown here used a very limited range of parameters for the different algorithms that have been experimented with, and further optimization for many of these parameters is certainly worth pursuing. If possible, it may be helpful to tie the parameters of the algorithm to intrinsic properties of the matrix itself.

While it has been shown in previous works [CP15, CAD15] that it is possible to generate a suitable ILU preconditioner using a small number of sweeps of the FGPILU algorithm, 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.

10.0 ACKNOWLEDGMENTS

This work was supported in part by the Air Force Office of Scientific Research under the AFOSR award FA9550-12-1-0476, by the U.S. Department of Energy (DOE) Office of Advanced Scientific Computing Research under the grant DESC-0016564 through the Ames Laboratory, operated by Iowa State University under contract No. DE-AC00-07CH11358, by the U.S. Department of Defense High Performance Computing Modernization Program, through a HASI grant, and through the ILIR/IAR program at the Naval Surface Warfare Center Dahlgren Division. This work was also supported by the Turing High Performance Computing cluster at Old Dominion University.

11.0 REFERENCES

[AB05] Ahmed Addou and Abdenasser Benahmed. Parallel synchronous algorithm for nonlinear fixed point problems. International Journal of Mathematics and Mathematical Sciences, 2005(19):3175-3183, 2005.

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from Berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[ABC+10a] Steve Ashby, PETE Beckman, Jackie Chen, Phil Colella, Bill Collins, Dona Crawford, Jack Dongarra, Doug Kothe, Rusty Lusk, Paul Messina, et al. Ascac subcommittee report: The opportunities and challenges of exascale computing. Technical report, Technical report, United States Department of Energy, Fall, 2010.

[ABC+10b] Steve Ashby, PETE Beckman, Jackie Chen, Phil Colella, Bill Collins, Dona Crawford, Jack Dongarra, Doug Kothe, Rusty Lusk, Paul 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.

[ACD15] Hartwig Anzt, Edmond Chow, and Jack Dongarra. Iterative sparse triangular solves for preconditioning. In European Conference on Parallel Processing, pages 650-661. Springer, 2015.

[ACHD16] Hartwig Anzt, Edmond Chow, Thomas Huckle, and Jack Dongarra. Batched generation of incomplete sparse approximate inverses on GPUs. In Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), 2016 7th Workshop on, pages 49-56. IEEE, 2016.

[ACJ17] Hartwig Anzt, Edmond Chow, and Dongarra Jack. Private communication, 2017.

[ACSD16] Hartwig Anzt, Edmond Chow, Jens Saak, and Jack Dongarra. Updating incomplete factorization preconditioners for model order reduction. Numerical Algorithms, 73(3):611-630, 2016.

[ADQO15] Hartwig Anzt, Jack Dongarra, and Enrique S Quintana-Orti. Tuning stationary iterative solvers for fault resilience. In Proceedings of the 6th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, page 1. ACM, 2015.

[ADQO16] Hartwig Anzt, Jack Dongarra, and Enrique S Quintana-Orti. Fine-grained bit-flip protection for relaxation methods. Journal of Computational Science, 2016.

[AHBD18] Hartwig Anzt, Thomas K Huckle, Jurgen Brackle, and Jack Dongarra. Incomplete sparse approximate inverses for parallel preconditioning. Parallel Computing, 71:1{22, 2018.

[Bau78] Gerard M Baudet. Asynchronous iterative methods for multiprocessors. Journal of the ACM (JACM), 25(2):226{244, 1978.

[BBH+12] Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J Dongarra. An evaluation of user-level failure mitigation support in mpi. In European MPI Users' Group Meeting, pages 193-203. Springer, 2012.

[BdS08] Greg Bronevetsky and Boris de Supinski. Soft error vulnerability of iterative linear algebra methods. In Proceedings of the 22nd annual international conference on Supercomputing, pages 155-164. ACM,2008.

[Ben02] Michele Benzi. Preconditioning techniques for large linear systems: a survey. Journal of computational Physics, 182(2): 418-477, 2002.

[Ben07] Abdenasser Benahmed. A convergence result for asynchronous algorithms and applications. Proyecciones (Antofagasta), 26(2):219-236, 2007.

[BFHH12] Patrick G Bridges, Kurt B Ferreira, Michael A Heroux, and Mark Hoemmen. Fault-tolerant linear solvers via selective reliability. arXiv preprint arXiv:1206.1390, 2012.

[BHT00] Michele Benzi, John C Haws, and Miroslav Tuma. Preconditioning highly indefinite and nonsymmetric matrices. SIAM Journal on Scientific Computing, 22(4):1333-1353, 2000.

[Bla12] Wesley Bland. User level failure mitigation in mpi. In Euro-Par Workshops, pages 499-504. Springer, 2012.

[BSVD99] Michele Benzi, Daniel B Szyld, and Arno Van Duin. Orderings for incomplete factorization preconditioning of nonsymmetric problems. SIAM Journal on Scientific Computing, 20(5):1652-1670, 1999.

[BT89] Dimitri P Bertsekas and John N Tsitsiklis. Parallel and distributed computation: numerical methods, volume 23. Prentice hall Englewood Cliffs, NJ, 1989.

[CAD15] Edmond Chow, Hartwig Anzt, and Jack Dongarra. Asynchronous iterative algorithm for computing incomplete factorizations on GPUs. In International Conference on High Performance Computing, pages 1-16. Springer, 2015.

[CGG+09] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. The International Journal of High Performance Computing Applications, 23(4):374-388, 2009.

[CGG+14] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. Supercomputing frontiers and innovations, 1(1), 2014.

[Che13] Zizhong Chen. Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In ACM SIGPLAN Notices, volume 48, pages 167{176. ACM, 2013.

[CJB+17] Evan Coleman, Aygul Jamal, Marc Baboulin, Amal Khabou, and Masha Sosonkina. A Comparison and Analysis of Soft-Fault Error Models using FGMRES and ARMS RBT. In Proceedings of the 12th International Conference on Parallel Processing and Applied Mathematics. ACM, 2017.

[CP15] Edmond Chow and Aftab Patel. Fine-grained parallel incomplete LU factorization. SIAM journal on Scientific Computing, 37(2):C169-C193, 2015.

[CS97] Edmond Chow and Yousef Saad. Experimental study of ilu preconditioners for indefinite matrices. Journal of Computational and Applied Mathematics, 86(2):387-414, 1997.

[CS16a] Evan Coleman and Masha 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.

[CS16b] Evan Coleman and Masha 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.

[CS17] Evan Coleman and Masha 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.

[CS18a] Evan Coleman and Masha Sosonkina. Convergence and Resilience of the Fault Tolerant Variants 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.

[CS18b] Evan Coleman and Masha Sosonkina. Self-Stabilizing Fine-Grained Parallel Incomplete LU Factorization. Sustainable Computing: Informatics and Systems, 2018.

[CSC17] Evan Coleman, Masha Sosonkina, and Edmond 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.

[Dav94] Tim A Davis. The University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/, 1994.

[DK01] Iain S Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. SIAM Journal on Matrix Analysis and Applications, 22(4):973-996, 2001.

[EHM14a] James Elliott, Mark Hoemmen, and Frank Mueller. Evaluating the impact of SDC on the GMRES iterative solver. In Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, pages 1193-1202. IEEE, 2014.

[EHM14b] James Elliott, Mark Hoemmen, and Frank Mueller. Tolerating Silent Data Corruption in Opaque Preconditioners. arXiv preprint arXiv:1404.5552, 2014.

[EHM15] James Elliott, Mark Hoemmen, and Frank 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.

[EMSW13] James J Elliott, Frank Mueller, Miroslav K Stoyanov, and Clayton G Webster. Quantifying the impact of single bit flips on floating point arithmetic. Technical report, Oak Ridge National Laboratory (ORNL), 2013.

[FBD01] Graham E Fagg, Antonin Bukovsky, and Jack J Dongarra. Harness and fault tolerant mpi. Parallel Computing, 27(11):1479-1495, 2001.

[FD00] Graham Fagg and Jack Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. Recent advances in parallel virtual machine and message passing interface, pages 346-353, 2000.

[FS00] Andreas Frommer and Daniel B Szyld. On asynchronous iterations. Journal of computational and applied mathematics, 123(1):201-216, 2000.

[GL09] Al Geist and Robert Lucas. Major computer science challenges at exascale. International Journal of High Performance Computing Applications, 2009.

[Gup02] Anshul Gupta. Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices. SIAM Journal on Matrix Analysis and Applications, 24(2):529-552, 2002.

[HH11] Mark Hoemmen and Michael 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, volume 3, page 9. Citeseer, 2011.

[Inn15] Innovative Computing Lab. Software distribution of MAGMA. http://icl.cs.utk.edu/magma/, 2015.

[LD99] Xiaoye S Li and James Demmel. A scalable sparse direct solver using static pivoting. In PPSC, 1999.

[LLH+17] Weifeng Liu, Ang Li, Jonathan D Hogg, Iain S Duff, and Brian Vinter. Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides. Concurrency and Computation: Practice and Experience, 29(21), 2017.

[Man80] Thomas A Manteuffel. An incomplete factorization technique for positive definite linear systems. Mathematics of computation, 34(150):473-497, 1980.

[MSV15] Frederic Magoules, Daniel B Szyld, and Cedric Venet. Asynchronous optimized Schwarz methods with and without overlap. Numerische Mathematik, pages 1-29, 2015.

[Nau11] Maxim Naumov. Incomplete-lu and cholesky preconditioned iterative methods using cusparse and cublas. Nvidia white paper, 2011.

[NTM11] Akira Nukada, Hiroyuki Takizawa, and Satoshi 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, pages 104-113. IEEE, 2011.

[OR00] James M Ortega and Werner C Rheinboldt. Iterative solution of nonlinear equations in several variables. SIAM, 2000.

[Saa93] Yousef Saad. A flexible inner-outer preconditioned GMRES algorithm. SIAM Journal on Scientific Computing, 14(2):461-469, 1993.

[Saa94] Yousef Saad. Ilut: A dual threshold incomplete lu factorization. Numerical linear algebra with applications, 1(4):387-402, 1994.

[Saa96] Yousef Saad. Ilum: a multi-elimination ilu preconditioner for general sparse matrices. SIAM Journal on Scientific Computing, 17(4):830{847, 1996.

[Saa03] Yousef Saad. Iterative methods for sparse linear systems. SIAM, 2003.

[SKB12] Joseph Sloan, Rakesh Kumar, and Greg 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, pages 1-12. IEEE, 2012.

[SS86] Yousef Saad and Martin H Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM Journal on scientific and statistical computing, 7(3):856-869, 1986.

[SS02] Y Saad and B Suchomel. ARMS: An algebraic recursive multilevel solver for general sparse linear systems. Numerical linear algebra with applications, 9(5):359-378, 2002.

[SSR11] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. Characterizing the impact of soft errors on iterative methods in scientific computing. In Proceedings of the international conference on Supercomputing, pages 152-161. ACM, 2011.

[SSR12] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In Proceedings of the 26th ACM international conference on Supercomputing, pages 69-78. ACM, 2012.

[SV13] Piyush Sao and Richard Vuduc. Self-stabilizing iterative solvers. In Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, page 4. ACM, 2013.

[SW15] Miroslav Stoyanov and Clayton Webster. Numerical analysis of fixed point algorithms in the presence of hardware faults. SIAM Journal on Scientific Computing, 37(5):C532-C553, 2015.

[SZ99] Yousef Saad and Jun Zhang. Bilutm: a domain-based multilevel block ilut preconditioner for general sparse matrices. SIAM Journal on Matrix Analysis and Applications, 21(1):279-299, 1999.

[ZSK04] Gengbin Zheng, Lixia Shi, and Laxmikant V Kale. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In 2004 IEEE International Conference on Cluster Computing, pages 93-103. IEEE, 2004.

DISTRIBUTION

	Copies		Copies
DOD ACTIVITIES (CONUS) DIRECTOR STRATEGIC SYSTEMS PROGRAMS DIRECTOR STRATEGIC SYSTEMS PROGRAMS ATTN SP232 1 1250 10TH STREET SE SUITE 3600 WASHINGTON NAVY YARD DC 20374-5127	0/1	**NON-DOD ACTIVITIES (CONUS)** ATTN DOCUMENT CENTER THE CNA CORPORATION 303 WASHINGTON BOULEVARD ARLINGTON VA 22201	1/1
DEFENSE TECH INFORMATION CTR 8725 JOHN J. KINGMAN ROAD SUITE 0944 FORT BELVOIR VA 22060-6218	0/2	**INTERNAL** A13 COLEMAN	1/0

NAVSEA

WARFARE CENTERS

DAHLGREN