Self-Stabilizing Fine-Grained Parallel Incomplete LU Factorization

Evan Coleman[a,b,*], Masha Sosonkina[b]

[a] *Naval Surface Warfare Center - Dahlgren Division, 17320 Dahlgren Rd, Dahlgren, VA*
[b] *Old Dominion University, 5115 Hampton Blvd, Norfolk, VA*

Abstract

This paper presents an investigation into the use of various mechanisms for improving the resilience of the fine-grained parallel algorithm for computing an incomplete LU factorization. These include various approaches to checkpointing as well as a study into the feasibility of using a self-stabilizing periodic correction step. Results concerning convergence of all of the self-stabilizing variants of the algorithm with respect to the occurrence of faults, and the impact of any sub-optimality in the produced incomplete L and U 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 factorization, and improve the performance of the resulting factors as preconditioners in Krylov subspace solvers in the presence of transient soft faults.

Keywords: Fault tolerance, parallel preconditioning, incomplete factorization, asynchronous iterative methods, self-stabilizing iterative algorithms

1. Introduction

Self-stabilizing methods are generally defined as those that return a system to a valid state within some finite number of steps [1]. As pointed out in [1], the self-stabilizing property provides a means for fault tolerance; if a non-persistent fault occurs, the self-stabilizing method should be able to correct any impact of the fault in such a way that the algorithm will still converge. This work provides an extension of the study presented in [2] following the spirit of [1].

On future exascale platforms, the expected increase in the number of faults is expressed in [3, 4, 5, 6]. Self-stabilizing methods form a superset of traditional checkpoint based fault-tolerance schemes that also include other mechanisms for resilience that may offer several advantages. The use of a periodic correction step [1] is one such alternative class of method that offers several advantages. 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 [4, 6]. 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.

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 typically used. One of the most commonly used classes of preconditioners is incomplete LU factorization. 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 fine-grained parallel incomplete LU (FGPILU) algorithm proposed in [7] is such an algorithm.

In order to devise routines resilient to faults, it is important to classify the potential impacts that can be caused by a fault. Faults can be divided into those that cause immediate program interruption (hard faults) and those that do not (soft faults) (e.g., [8]). Soft faults typically refer to some form of data corruption; often manifesting as bit-flips. This paper examines the potential impact of soft faults on the FGPILU factorization, and also investigates the use of preconditioners generated by the FGPILU algorithm (and the self-stabilizing variants presented here) on Krylov subspace solvers.

This work presents an extension of the work performed in [7]. The extensions presented in this work are: an expansion of the discussion of related work throughout the paper to incorporate recent research efforts more thoroughly, an extended discussion on the convergence of both the nominal FGPILU algorithm as well as the variants presented here (c.f. Section 3 and Section 4), discussion of additional variants to the FGPILU algorithm (in particular: Algorithm 2, and Algorithm 6), and an extension to

*Corresponding author
URL: ecole028@odu.edu (Evan Coleman), msosonki@odu.edu (Masha Sosonkina)

the fault model that is used to incorporate bit flips directly (vice solely using a numerical simulation of soft faults) in order to ensure that the worst case of a soft fault (i.e. a bit flip of sign or exponent bit) is properly captured (see Section 5.1). The numerical results (Section 5) have been updated and expanded to address the new changes throughout the body of the paper, and the outlook towards future directions has been updated as well.

The structure of this paper is organized as follows: in Section 2, a brief summary of some related studies is provided, in Section 3, background information is provided for the FGPILU algorithm, in Section 4, the titular self-stabilized variant(s) of the FGPILU algorithm are developed, in Section 5, a series of numerical results are provided, while Section 6 provides a summary and future directions.

2. Related Work

The expected increase in faults for future HPC systems is detailed in [3, 4, 5, 6]. An initial look into fault tolerance for the FGPILU algorithm is provided in [2]. 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 [1], which in turn are built upon the ideas of selective reliability [8, 9]. 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 [10]. The analysis of the potential performance of a Krylov subspace method using a potentially sub-optimal FGPILU algorithm is related to the analysis in [1]. The results for the experiments conducted for this effort are presented similarly to the results in [7, 11], 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 [12], and a numerical fault model that is predicated on shuffling the components of an important data structure [13]. Other numerical models, such as inducing a small shift to a single component of a vector have been considered as well [8]. Comparisons between various numerical soft fault models have been made in [14, 15]. The fault model used in this paper is a combination of a modified version of the one initially developed in [12] (related to the fault model developed in [13]) that was used in [2] and a simple model that flips bits directly. Details on the fault model used here are provided in Section 5.1.

Fault tolerance for fine-grained parallel methods is an area seeing increased research activity. An initial exploration of fault tolerance for the FGPILU factorization studied here is provided in [2], and an exploration of resilience for stationary iterative linear solvers (i.e. Jacobi) is given in [16] and expanded on in [17]. A more general exploration of fault tolerance for fine-grained methods is provided in [18].

3. 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, $A \approx LU$. Normally, the individual components of both L and U are computed in a manner that does not allow easy use of parallelization. The recent FGPILU algorithm proposed in [7] allows each element of both the L and U factors to be computed independently. 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 $(LU)_{ij} = a_{ij}$ 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 1) is defined by the following two non-linear equations:

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

Following the analysis presented in [7], 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,

$$x^{(p+1)} = G\left(x^{(p)}\right) \quad (2)$$

where the function G implements the two non-linear equations described above. The FGPILU algorithm is given in Algorithm 1. Keeping with the terminology used in [7, 11]

Algorithm 1: FGPILU algorithm as given in [7]

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

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

each pass the algorithm makes in updating all of the l_{ij} and u_{ij} elements (alternatively: each element of the vector x) is referred to as a "sweep". After each sweep of

the algorithm, the L and U factors progress towards convergence. At the beginning of the algorithm, the factors L and U are set with an initial guess. In this 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 [7, 2, 19]. Adopting the technique used in [7, 11, 2], 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 [7], this diagonal scaling is imperative to maintain reasonable convergence rates for the algorithm, and the working assumption in this paper is that all matrices have been scaled appropriately.

3.1. Convergence of the Fine-Grained Parallel Incomplete LU Factorization

In a fault-free environment, it can be proven that the FGPILU algorithm is locally convergent in both the synchronous and asynchronous cases (see Section 3 of [7]). A few details of this analysis are recreated here since they will be pertinent to the effort of showing that the self-stabilizing variants are convergent.

The analysis to show convergence of the FGPILU algorithm relies on properties of the Jacobian associated with the non-linear mapping defined by $G : \mathbb{R}^m \to \mathbb{R}^m$ 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 the original input matrix, A. The following definition formalizes this concept.

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 $1, 2, \ldots, N$. Formally, this is a map $T : S \to 1, 2, \ldots, N$.

Specifically, 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. This can be described as follows:

- The first row of M
- The remainder of the first column of M
- The remainder of the second row of M
- The remainder of the second column of M
- $\cdots$

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):

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

In particular, an ordering g will map a pair of (i, j) coordinates specifying the location of a non-zero term in A to an index of the vector x. That is,

$$x_{g(i,j)} = \begin{cases} l_{ij} & i > j \\ u_{ij} & i \leq j \end{cases}$$

Given this, the two non-linear equations that define the FGPILU algorithm (Eq. (1)) can be rewritten so that,

$$G_{g(i,j)} = \begin{cases} \frac{1}{x_{g(j,j)}} \left(a_{ij} - \sum\limits_{1 \leq k \leq j-1} x_{g(i,k)} x_{g(k,j)} \right) & i > j \\ a_{ij} - \sum\limits_{1 \leq k \leq i-1} x_{g(i,k)} x_{g(k,j)} & i \leq j \end{cases} \tag{3}$$

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

The Jacobian itself is written $G'(x) = J(G(x))$ where $J : \mathbb{R}^{|S|} \to \mathbb{R}^{|S| \times |S|}$ and is defined by the following equations [7]:

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

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,

$$\begin{aligned} \frac{\partial G_{g(i,j)}}{\partial x_{g(i,k)}} &= -x_{g(i,k)}, k < i \\ \frac{\partial G_{g(i,j)}}{\partial x_{g(k,j)}} &= -x_{g(i,k)}, k < i \end{aligned}$$

Under the assumption that there is a single fixed point solution x^* of the non-linear iteration defined by $G(x)$, the following Theorem from [20] provides convergence for the nominal FGPILU algorithm:

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

The partial derivatives are continuous and well-defined anywhere on the domain of G as defined above so G is F-differentiable on its domain. What remains to be shown is that the spectral radius $\rho(G'(x^*)) < 1$.

The Gaussian Elimination partial ordering proposed in Definition 2 leads to the following result from [7] details the structure of mapping, G, defined by Eq. (3):

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

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

This leads to the following related result that also comes from Chow and Patel in [7]:

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.

However, in order to determine if the mapping will converge from its current location in the domain of the mapping G defined by Eq. (3) 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:

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

for some $x, y \in D$.

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

Definition 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. (3), the domain is not necessarily convex, but the norm of the associated Jacobian is still indicative of whether or not the corresponding fixed-point iteration will converge [7].

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 3
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 then or equal to 1 than the mapping is not a contraction and further iteration will not bring the current iterate, x^*, closer to the fixed point.

4. Self-Stabilizing Fault Tolerance for the FGPILU 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 encompasses both traditional approaches towards resilience such as checkpointing, as well as different algorithmically based variants.

In [1] a periodic correction step was used to ensure that the algorithm being studied returned to a valid state and would proceed to convergence successfully. The work performed here covers both checkpointing and the use of a periodic correction step. The goal of the 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. Note that a selective reliability mode [8, 9] must be assumed since the computations performed during the correcting step need to be executed successfully.

As discussed in [7], convergence of the FGPILU algorithm is strongly related to the Jacobian of the functional iteration, G (i.e. Eq. (2)). In order to determine what steps to take in the correcting block, 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. A naive correction step that constitutes a hybrid self-stabilizing/checkpointing method is given by Algorithm 2.

Algorithm 2: Naive 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 $A \approx LU$

for $sweep = 1, 2, \ldots, m$ **do**
  if $sweep \equiv 0 \mod F$ **then**
    Form the Jacobian of the current iterate, J
    Evaluate $\tau = ||J||$
    if $\tau < 1$ **then** Continue
    else
      Set l_{ij} and u_{ij} to the last known good state
  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 $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$

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 2 are:

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 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 k^{th} sweep, and all subsequent iterations are fault-free then Algorithm 2 will converge.*

Proof. Since the Jacobian at the fixed point of the algorithm has spectral radius less than 1 (see [20]) and the correcting step of Algorithm 2 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 2 will converge. □

While the method proposed by Algorithm 2 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 [7] 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,*

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

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

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

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

$$\alpha = \max(||U_i||_\infty, ||L_i||_1, ||R_i^L||_1) < 1 \tag{4}$$

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. Development of traditional checkpointing variants will be examined in the next two subsections Section 4.1 and Section 4.2, and the use of a periodic correction step will be examined in the following two subsections Section 4.3 and Section 4.4.

4.1. 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 5.1, 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

$$\hat{x}^{(k+1)} = G\left(x^{(k)}\right) + \delta \,, \tag{5}$$

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 [7] and [11] to monitor the non-linear residual norm. This is a value

$$\tau = \sum_{(i,j) \in S} \left| a_{ij} - \sum_{k=1}^{\min(i,j)} l_{ik} u_{kj} \right| \tag{6}$$

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 non-linear residual and the ILU residual for the first few iterations / sweeps of the FGPILU algorithm on the Apache problem (see Table 2 for descriptions of the example problems) are given in Table 1. Note that the non-linear residual norm will continue decreasing, but that the ILU residual quickly settles to a non-zero value.

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

Table 1: Typical progression of both the non-linear residual norm and ILU residual norm for the Apache2 test problem.

If a fault occurs on a given sweep then one or both non-linear equations from the FGPILU algorithm Algorithm 1 will have some amount of error. In particular, the update equations for l_{ij} and u_{ij} will become

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) + \delta_{ij} \tag{7}$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} + \delta_{ij} \tag{8}$$

where r_{ij} represents the component of the vector r that maps to the (i, j) location of the matrix. Comparing Eq. (7) and Eq. (6) shows that if a fault occurs during the computation of the incomplete LU factors that the non-linear residual norm τ will be affected.

In order to ensure that a fault does not negatively effect the outcome of the algorithm, the first checkpointing variant that is proposed involves a simple monitoring of the non-linear residual norm τ. In principle, since $S \subset A$, when the FGPILU algorithm converges, the non-linear 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 3.

In this case, a fault is declared if the currently computed non-linear residual norm $\tau^{(sweep)}$ is some factor α greater than the previously computed non-linear 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, the non-linear residual norm will not be strictly monotonically decreasing, especially as the algorithm proceeds closer to convergence.

Algorithm 3: 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 $\tau^{(sweep)} > \alpha \cdot \tau^{(sweep+r)}$ **then**
        $Fault$ = TRUE

Therefore using the factor $\alpha = 1$, 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 non-linear residual).

4.2. Partial Checkpointing

Next, note that since there is a contribution from every $(i, j) \in S$, the individual non-linear residual norms for each $(i, j) \in S$, denoted here by τ_{ij}, can be defined as

$$\tau_{ij} = \left| a_{ij} - \sum_{k=1}^{\min(i,j)} l_{ik} u_{kj} \right| \tag{9}$$

where the total non-linear residual norm can always be recovered by taking the sum of all the individual non-linear residual norms over all $(i, j) \in S$. To establish a baseline for fault tolerance, define individual non-linear 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 non-linear 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 non-linear residual norms it is possible to determine if a large fault occurred. Specifically, it is of interest to determine if a fault occurred that was large enough to cause a potential divergence of the algorithm. To do this, first a tolerance t is set and then a fault is signaled if

$$\tau_{ij} > t \tag{10}$$

since the individual non-linear residual norms are generally decreasing as the FGPILU algorithm progresses. Set the

value t as $t = \max(\tau^*_{ij})$ initially (Line 3 of Algorithm 4), and then update t during the course of the algorithm if desired. It is also possible to use the previous individual non-linear residual norms as opposed to a maximum that is taken across all current non-linear individual norms. In particular, similarly to the global checkpointing variants advocated in Section 4.1, a fault can be declared if,

$$\tau_{ij}^{sweep} > \alpha \cdot \tau_{ij}^{sweep-r} \tag{11}$$

for similar parameters α and r.

Note that if a fault is signaled by any of the individual non-linear residual norms, it is only known that a fault occurred somewhere in the current row of the factor L or the current column of the factor U. As such, the conservative approach would require the rollback of both the current row of L and the current column of U to their values at the previous checkpoint (e.g., Lines 5 to 9 of Algorithm 4).

It is possible for the individual non-linear 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 non-linear residual, $\frac{\Delta\tau}{\Delta t}$, can be checked to ensure that it is also increasing before switching the current row and/or column (see Line 15 of Algorithm 4). This algorithm is detailed in Algorithm 4.

Algorithm 4: Checkpoint-Based Fault Tolerant FGPILU (CP-FGPILU)

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

1 **for** $(i,j) \in S$ **do in parallel**
2 $\quad \tau_{ij} = \left| a_{ij} - \sum_{k=1}^{\min(i,j)} l_{ik}u_{kj} \right|$
3 $t = \max(\tau_{ij})$
4 **for** $sweep = 1, 2, \ldots, m$ **do**
5 $\quad$ **if** *Fault* **then**
6 $\quad\quad$ Set $i = \max_{i,j}(k^1_{ij})$ and $j = \max_{i,j}(k^2_{ij})$
7 $\quad\quad$ Rollback $\{l_{ik}\}_{k=1}^{i-1}$ and $\{u_{kj}\}_{k=1}^{j-1}$
8 $\quad\quad$ $Fault$ = FALSE
9 $\quad\quad$ $sweep = sweep - 1$
10 $\quad$ **else**
11 $\quad\quad$ **for** $(i,j) \in S$ **do in parallel**
12 $\quad\quad\quad$ **if** $i > j$ **then**
$\quad\quad\quad\quad l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj})/u_{jj}$
13 $\quad\quad\quad$ **else** $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$
14 $\quad\quad\quad$ Compute τ
15 $\quad\quad\quad$ **if** $\tau_{ij} > t$ *and* $\tau' > 0$ **then**
16 $\quad\quad\quad\quad$ Set $k^1_{ij} = i$ and $k^2_{ij} = j$
17 $\quad\quad\quad\quad$ $Fault$ = TRUE

Note that if a fault is detected, the algorithm only restores (i.e., "rolls back") the affected row of L and column of U. Additionally, since in practice it has been proposed [7, 11] to use a limited number of sweeps of the FGPILU algorithm as opposed to converging the algorithm according to the global non-linear 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.

While no global communication is required to check for the presence of a fault via the individual non-linear residual norms, τ_{ij}, there is global communication required to compute the derivative of the global non-linear residual norm. A simple (forward) finite difference scheme is used to approximate this derivative to minimize the global communication required by Algorithm 4. The frequency with which the global non-linear residual norm is computed can be determined independently of the rest of the algorithm. Specifically, it may be possible to compute these updates less frequently in order to minimize the communication that takes place between the different components.

Additionally, if a fault is detected there will be some communication required between processes in order to fix the effects of the fault. Since the component detecting a fault will have to roll back elements that it is not directly responsible for updating, further computation on all affected elements will have to cease momentarily. Note also that when using the CP-FGPILU algorithm, the size of the faults that are not caught by the algorithm are determined by the tolerance that is set. In particular,

$$||r|| \leq t \tag{12}$$

where r represents a fault that was not caught by the proposed checkpointing scheme, since if $||r|| > t$ then the fault would be caught by the check on Line 15 of Algorithm 4. This, in turn, affects the update equations Eqs. (5) and (7).

4.3. 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 previous section, the emphasis in this section 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 5. 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 5 was designed to correct problems arising from simple finite difference discretizations of partial differential equations. The technique of observing the magnitude of the elements used in the fixed point iteration

Algorithm 5: Self-Stabilizing Fault Tolerant FGPILU (SS-FGPILU)

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

```
1  for sweep = 1, 2, ..., m do
2      if sweep ≡ 0 mod F then
3          for (i, j) ∈ S do in parallel
4              if {‖l_ij‖, ‖u_ij‖} ≫ ‖a_ij‖ or
                 |{l_ij, u_ij} − a_ij|/|a_ij| > β or
                 {l_ij, u_ij} = {0, NaN} then
                 {l_ij, u_ij} = a_ij
5              if i > j then
                 l_ij = (a_ij − Σ_{k=1}^{j−1} l_ik u_kj)/u_jj
6              else u_ij = a_ij − Σ_{k=1}^{i−1} l_ik u_kj
7      else
8          for (i, j) ∈ S do in parallel
9              if i > j then
                 l_ij = (a_ij − Σ_{k=1}^{j−1} l_ik u_kj)/u_jj
10             else u_ij = a_ij − Σ_{k=1}^{i−1} l_ik u_kj
```

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 4.5 and Section 5 this technique will not generalize to all other problems but may extend to other similar matrices (i.e. symmetric positive definite, strongly diagonally dominant, small bandwidth, etc).

The next result establishes a convergence property for the variant of the FGPILU algorithm proposed in Algorithm 5.

Theorem 6. *For any state of $l_{ij} \in L$ and $u_{ij} \in U$, if a correction is performed in the k^{th} 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 5 will converge.*

Proof. This follows from noticing that the correcting (or "stabilizing") step (Lines 2 to 6 of Algorithm 5) 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 [7]) which rely upon the assumptions and base arguments from [20]. □

4.4. Component-Wise Periodic Correction Step

The next resilient variant of the FGPILU algorithm to be discussed relies on tracking the component-wise progression of the individual non-linear norms (Eq. (9)) in a manner similar in spirit to Algorithm 4. Recall from Section 4.1 that the individual non-linear 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 non-linear residual norm which requires communication between all of the components. This scheme is detailed in Algorithm 6.

Algorithm 6: Component-Wise Self-Stabilizing Fault Tolerant FGPILU (CW-FGPILU)

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

```
1  for sweep = 1, 2, ..., m do
2      if sweep ≡ 0 mod F then
3          for (i, j) ∈ S do in parallel
4              if τ_ij^sweep > α · τ_ij^(sweep−F) then
5                  Set i = max_{i,j}(k_ij^1) and
                     j = max_{i,j}(k_ij^2)
6                  Rollback {l_ik}_{k=1}^{i−1} and {u_kj}_{k=1}^{j−1}
7      else
8          for (i, j) ∈ S do in parallel
9              if i > j then
                 l_ij = (a_ij − Σ_{k=1}^{j−1} l_ik u_kj)/u_jj
10             else u_ij = a_ij − Σ_{k=1}^{i−1} l_ik u_kj
```

The CW-FGPILU algorithm variant (Algorithm 6) can be seen as a modified version of the partial checkpointing method that is utilized in Algorithm 4 where the check on the global non-linear residual norm, τ, is omitted but the frequency of the check on the progression of the individual non-linear residual norms τ, is decreased to compensate. This method can be viewed as a simple modification of the partial checkpointing scheme that can limit the amount of communication that takes place between the individual components in the factors L and U.

4.5. General Notes on the Convergence of the FGPILU Variants

The main result concerning the convergence of the FGPILU algorithm comes from [20], 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 non-linear update equations Eq. (1). 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, 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 [7, 11] 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 non-linear residual norm, τ below some threshold tolerance, ϵ. It is therefore possible that any adverse affects that a fault may have on the convergence of the FGPILU algorithm itself will not have sufficient time to propagate throughout the entirety of the computed L and U factors to have a meaningful impact on the performance of the overarching iterative method (e.g. CG, GMRES, etc) that the computed factors are used as preconditioner for. These potential impacts will be explored numerically in Section 5.

5. Numerical Results

The experimental setup for this study is an NVIDIA Tesla K40m GPU on the Turing High Performance Cluster at Old Dominion University. The nominal, fault-free iterative incomplete factorization algorithms and iterative solvers were taken from the MAGMA open-source software library [21]. All of the results provided in this study reflect double precision, real arithmetic.

The test matrices that were used predominantly come from the University of Florida sparse matrix collection maintained by Tim Davis [22], and the matrices selected for this study are the same as the ones that were selected for the study [11] that detailed the performance of the FGPILU algorithm on GPUs without the presence of faults. 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 $50 \times 50 \times 50$ mesh.

Mimicking the study conducted in [11], all six of the matrices from the University of Florida sparse matrix collection were reordered using the Reverse Cuthill-McKee (RCM) ordering in an effort to decrease the bandwidth and help to improve convergence. An example of the effect that this can have is provided by Fig. 1.

All of the matrices considered in this 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 3 that each of the eight matrices used in this 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 2.

The experiments are divided into two sets. This first set of experiments focuses on the convergence of the FGPILU algorithm despite the occurrence of faults and features comparisons of the L and U factors produced by the preconditioning algorithms. The second set of experiments shows the impact of using in a Krylov subspace solver the preconditioners obtained from the first set of experiments. Note that in all of the experiments conducted, the condition $u_{jj} = 0$ was never encountered. Since all the test matrices are SPD, the preconditioning algorithms are Incomplete Cholesky variants, and the the solver is the preconditioned conjugate gradient (PCG), as implemented in the MAGMA library [21].

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. Optimal checkpointing libraries for GPU based applications have been explored in [23] and [24], and extended performance analysis would be needed to produce performance-oriented prototypes of each of the variants.

Matrix Name	Abbreviation	Dimension	Non-zeros	Description
APACHE2	APA	715,176	4,817,870	SPD 3D finite difference
ECOLOGY2	ECO	999,999	4,995,991	circuit theory applied to animal/gene flow
G3_CIRCUIT	G3	1,585,478	7,660,826	circuit simulation problem
OFFSHORE	OFF	259,789	4,242,673	3D FEM, transient electric field diffusion
PARABOLIC_FEM	PAR	525,825	3,674,625	parabolic FEM, diffusion-convection reaction
THERMAL2	THE	1,228,045	8,580,313	unstructured FEM, steady state thermal problem
LAPLACE2D	L2D	250,000	1,248,000	Laplacian 2D finite difference, 5-point stencil
LAPLACE3D	L3D	125,000	3,329,698	Laplacian 3D finite difference, 27-point stencil

Table 2: Summary of the 8 symmetric positive-definite matrices used in this study. Descriptions come from [22].

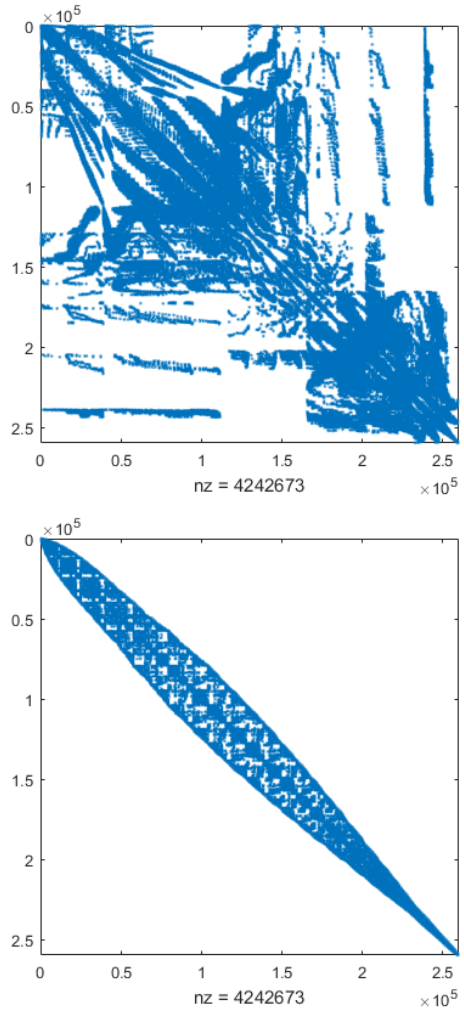

Figure 1: The sparsity pattern for the 'OFFSHORE' matrix with the natural ordering (left), and the RCM ordering (right)

5.1. Fault Model

Soft faults typically manifest as bit-flips. In this study, two different fault injection methodologies were used. 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. Much of the current research (e.g., [25]) treats faults exclusively as bit flips; which reflects the current method in which faults occur. 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, a generalized numerical scheme for simulating the occurrence of a fault is adopted in addition to injecting bit flips directly.

Using the perturbation-based model used in [2], the modified model targets a single data structure and injects a small random perturbation into each component transiently, as opposed to doing so persistently as was done in [12]. For example, if the targeted data structure is a vector x and the maximum size of the perturbation-based fault is ϵ, then proceed as follows:

- Generate a random number $r_i \in (-\epsilon, \epsilon)$ for every component x_i, where i ranges over entire length of x
- Set $\hat{x}_i = x_i + r_i$ for all i's

The resultant vector $\hat{x}$ is, thus, perturbed away from the original vector x. 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:

- The perturbation-based model shows resilience of the proposed algorithmic variants to small-moderate errors and 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 [26])
- 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

To be precise, for the perturbation-based model, if the vector to be perturbed is x and the size of the perturbation based fault is ϵ than to inject a fault, then generate a random number in the range $r_\epsilon \in (-\epsilon, \epsilon)$ and set $x_i = x_i + r_\epsilon$ for all i. 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, $||x||_2 \approx ||\hat{x}||_2$.

It was shown in [13] that the numerical soft-fault model proposed there corresponds to a "sufficiently bad" impact of a soft fault rather than attempting to determine the "damage" *exactly* of a soft fault. Explorations of the similarities and differences between the numerical soft fault model presented in [13] and the perturbation based model are presented in [14, 15]. Simulating these numerical soft fault models for iterative algorithms may force them to run consistently through bad errors only; however the bit-flip model was added to ensure that the "worst case" scenario was also captured fully. Furthermore, by varying the size of the perturbation, it is possible to produce errors with an increasingly large 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 number before convergence. Three fault-size ranges (corresponding to differing orders of magnitude) for the faults injected by the perturbation-based model were considered:

$$r_i \in (-0.01, 0.01) \quad (13)$$
$$r_i \in (-1, 1) \quad (14)$$
$$r_i \in (-100, 100) \quad (15)$$

The bit-flip model was included to appropriately gage the worst case scenario, but no effort was made to force the bit selected to be in a particular position. Results for both the perturbation-based model and the bit-flip model are presented separately, but as averages over all trials run 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.

5.2. Convergence of FGPILU algorithm

For the purposes of this study, the FGPILU algorithm is said to have converged successfully if the non-linear 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 non-linear 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 non-linear residual norm for a single fault-free run of each problem is depicted in Fig. 2 (top), which is a as an example of the typical progression of the non-linear residual norm as the algorithm progresses towards convergence.

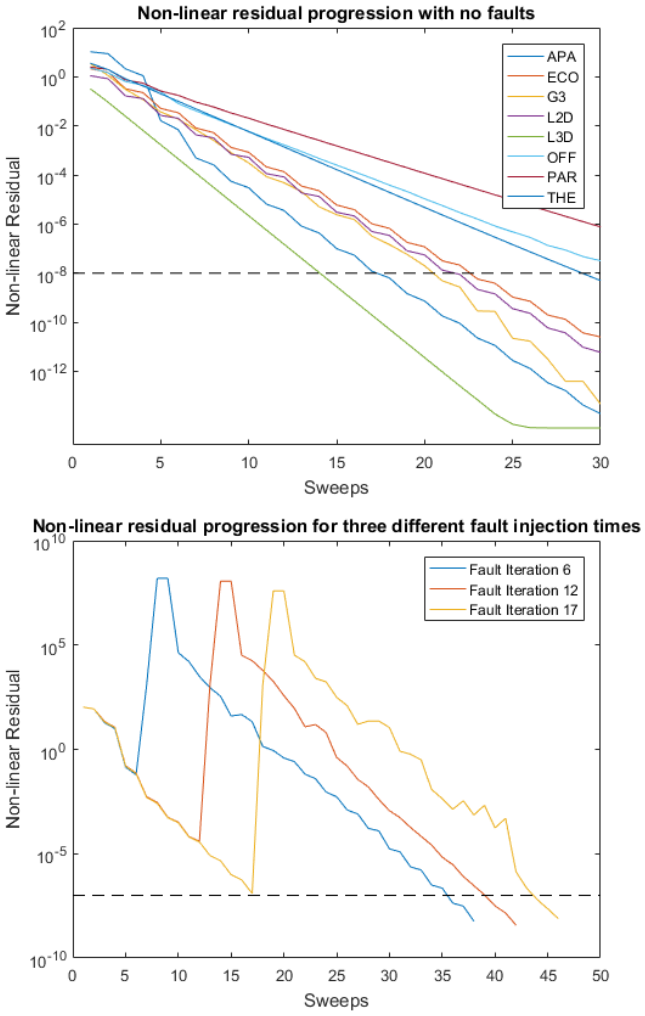

Figure 2: The progression of the non-linear residual for 30 sweeps of a typical fault-free run for each of the 8 test problems (top). The progression of the non-linear residual for the Apache test problem for three different fault injection times and fault size in the $(-1, 1)$ range (bottom). The horizontal dashed line is indicated the FGPILU convergence tolerance of 10^{-8}.

To illustrate the potential impact of a fault, Fig. 2 (bottom) 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 non-linear residual in 20 iterations when faults are not present.

From Fig. 3 (bottom), 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 the three injection places. Although the example shown in Fig. 3 (bottom) is typical of what 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 en-

tirely or the resulting L and U factors may cause the PCG solver to either stagnate or even diverge. A major point of the example in Fig. 3 (bottom) is to report the beneficial effects on FGPILU convergence of larger number of sweeps if faults are ignored in FGPILU and 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 Fig. 3 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 3 (`CPA`),
- the CP-FGPILU which rolls back a single row and column of the L and U factors and is described in Algorithm 4 (`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 5 (`SS`),
- the periodic correction step based on checking component wise progression of the individual non-linear residuals, τ_{ij} which is given in Algorithm 6 (`CW`).

5.2.1. Perturbation-Based Faults

This section examines the effects of a soft fault (modeled as a perturbation as described in Section 5.1) 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. 3.

Figure 3 (top) 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. 3 (top) 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. 3 (bottom), 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. For the FGPILU variants tested, the success rates captured in Fig. 3

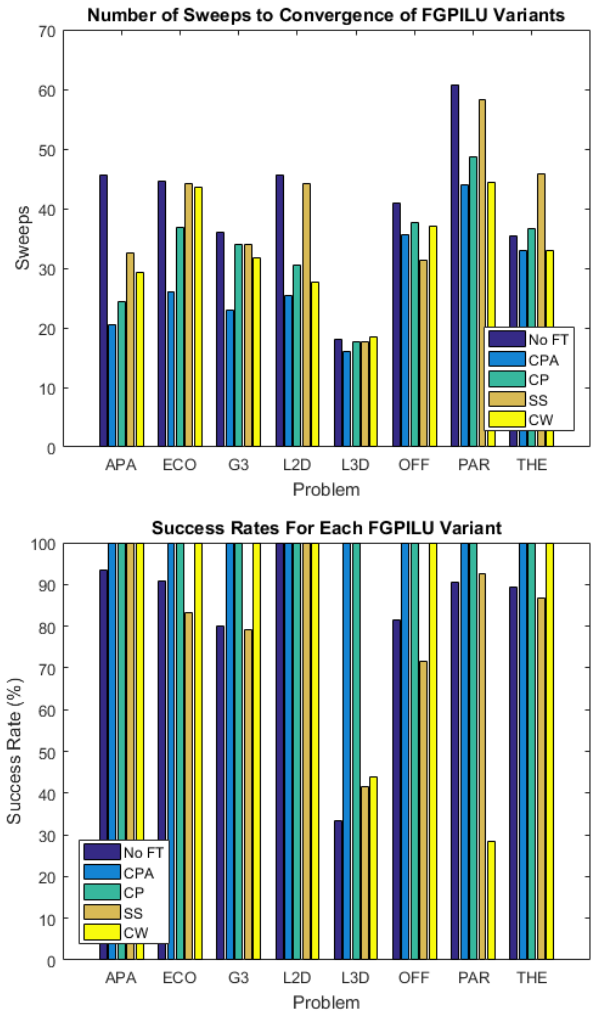

Figure 3: For perturbation-based faults: the numbers sweeps required for convergence for each of the 8 test problems (top). The percentage of runs that produced a preconditioner that corresponded to a successful PCG solve (bottom).

(bottom) 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 5 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.

5.2.2. 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 Section 5.2.1. 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.

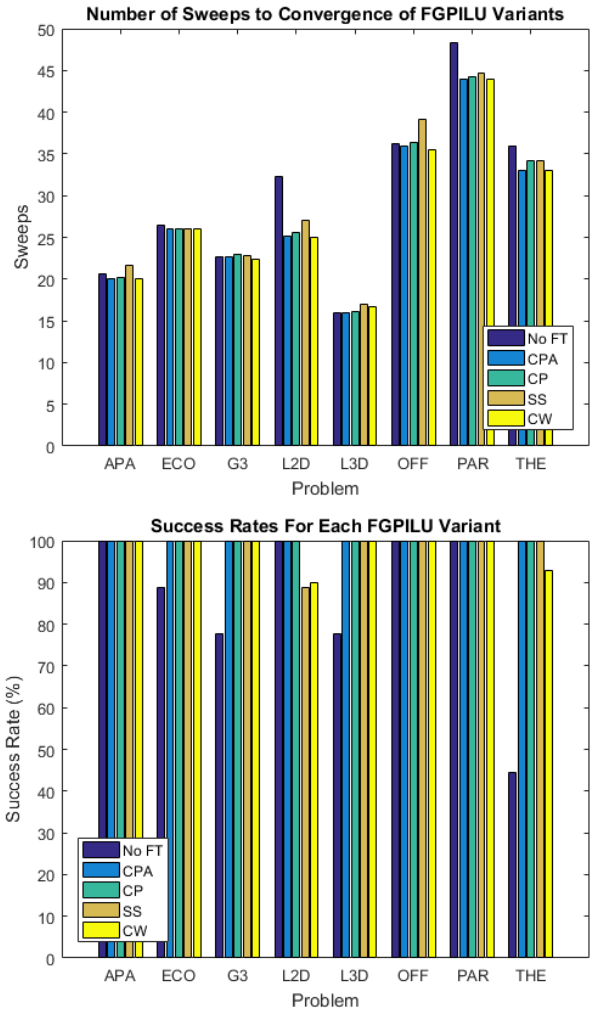

Figure 4: For bit-flip faults: the numbers sweeps required for convergence for each of the 8 test problems (top). The percentage of runs that produced a preconditioner that corresponded to a successful PCG solve (bottom).

Figure 4 (top) 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. 3. The success rates for the trials run with bit flips (see Fig. 4 (bottom)) 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. 3 and Fig. 4 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.

5.3. Preconditioner Performance in Iterative Methods

In this set of experiments, a maximum number of 3000 PCG iterations was used; any run that had not converged by that point was declared to have diverged. While all of the preconditioners to be evaluated are forms of incomplete LU decomposition, they are constructed by algorithms described in Section 5.2. 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.

5.3.1. Perturbation-Based Faults

Figure 5 captures only the cases in which a preconditioner was successfully prepared (c.f. Fig. 3 (bottom)). Figure 5 (top) 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. 5 (top) correspond to runs of the FGPILU algorithm where no fault tolerance was attempted (`NoFT`) and enough of these runs were able to produce a PCG solve that converged in far more iterations than would typically be required to skew the averages. This seems to suggest that this behavior is not entirely anomalous and that the FGPILU algorithm has some nature level of resilience (else, the solves would not have been "successful" in the first place) to soft faults.

The timing results presented in Fig. 5 (bottom) 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 non-linear residual norm may

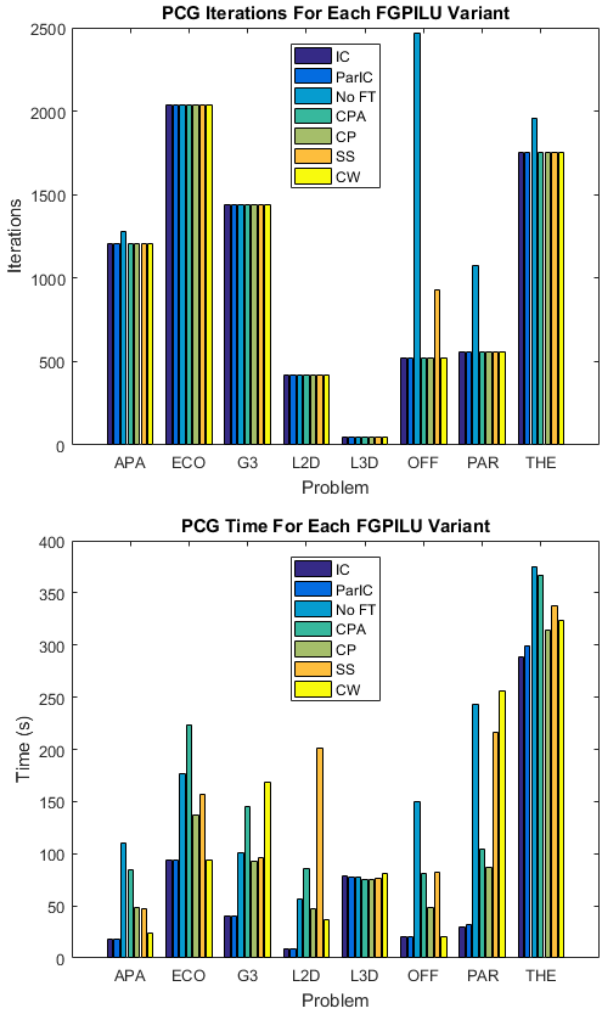

Figure 5: For perturbation-based faults: the numbers of iterations required for the successful PCG solves for each of the 8 test problems (top). The time required for the successful PCG solves for each of the 8 test problems (bottom).

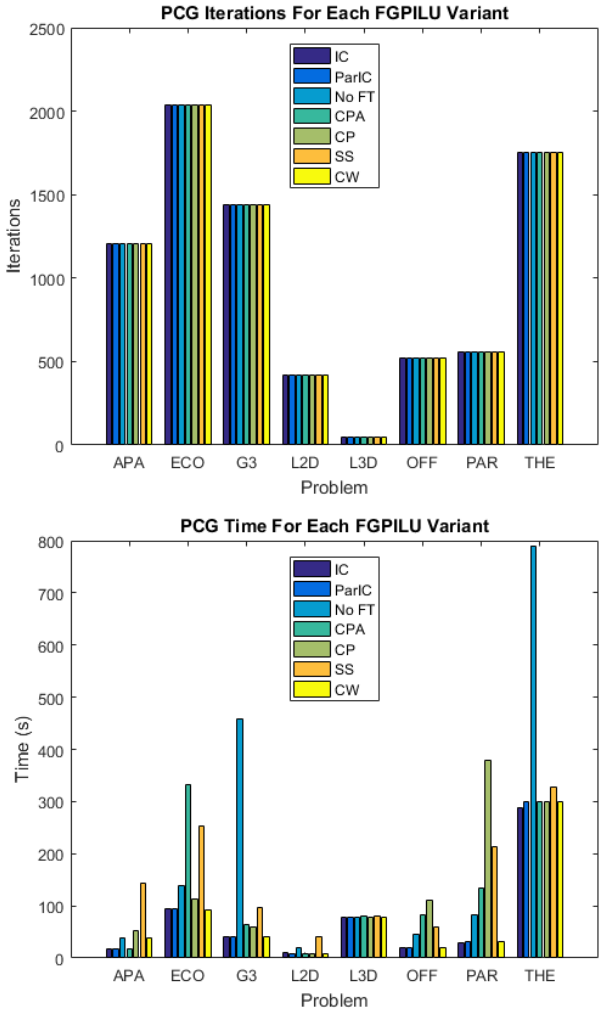

Figure 6: For bit-flip faults: the numbers of iterations required for the successful PCG solves for each of the 8 test problems (top). The time required for the successful PCG solves for each of the 8 test problems (bottom).

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 6 and represented by `CW` may be one of the more efficient variants. Note that because of the excessively small convergence chosen to declare the FGPILU algorithm converged (i.e. 10^{-8}) the time for all of the FGPILU variants (including `ParIC`) are inflated relative to the performance of traditional incomplete factorization (`IC`).

5.3.2. 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 5.1 - are explored at the level of timing and accuracy results in the corresponding PCG solve.

Figure 6 (top) 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. 5 (top) 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. 6 (bottom) 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. 5 (bottom) 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.

6. Conclusions

This paper has examined the potential impact of soft faults on the FGPILU algorithm. These faults, undetected by the unmodified program, have the potential to cause severe disruption to the preconditioning routine as well as the solver that uses the incomplete factors generated by the preconditioning algorithm. 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 experiments conducted here have shown that the FGPILU algorithm is somewhat naturally resilient to smaller faults as modeled here – i.e. either perturbations or bit-flips that affect less significant bits in the mantissa – but that larger faults can cause the algorithm 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. Comparing the bottom images of Fig. 3 and Fig. 4 indicates that the FGPILU algorithm and the variants discussed here tend to be more resilient to errors that only corrupt a single component. The results given 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 gage of the relative performance between them. Additionally, it would be advantageous to 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 (i.e. [7, 11, 2]) on the FGPILU algorithm have all explored problems that tend to be similar in nature. In particular, exploring the convergence of the algorithm for highly non-symmetric or indefinite matrices could help provide means for improving the convergence of the algorithm in a more general, global setting.

While it has been shown in previous works [7, 11] 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 important 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 happen to occur.

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 DE-SC-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.

References

[1] P. Sao, R. Vuduc, Self-stabilizing iterative solvers, in: Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ACM, 2013, p. 4.

[2] E. Coleman, M. Sosonkina, E. Chow, Fault Tolerant Variants of the Fine-Grained Parallel Incomplete LU Factorization, in: Proceedings of the 2017 Spring Simulation Multiconference, Society for Computer Simulation International, 2017.

[3] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, et al., The landscape of parallel computing research: A view from Berkeley, Tech. rep., Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (2006).

[4] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, M. Snir, Toward exascale resilience: 2014 update, Supercomputing frontiers and innovations 1 (1).

[5] M. Snir, R. Wisniewski, J. Abraham, S. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, et al., Addressing failures in exascale computing, International Journal of High Performance Computing Applications.

[6] A. Geist, R. Lucas, Major computer science challenges at exascale, International Journal of High Performance Computing Applications.

[7] E. Chow, A. Patel, Fine-grained parallel incomplete LU factorization, SIAM Journal on Scientific Computing 37 (2) (2015) C169–C193.

[8] M. Hoemmen, M. Heroux, Fault-tolerant iterative methods via selective reliability, in: Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE Computer Society, Vol. 3, Citeseer, 2011, p. 9.

[9] P. Bridges, K. Ferreira, M. Heroux, M. Hoemmen, Fault-tolerant linear solvers via selective reliability, arXiv preprint arXiv:1206.1390.

[10] Y. Saad, Iterative methods for sparse linear systems, Siam, 2003.

[11] E. Chow, H. Anzt, J. Dongarra, Asynchronous iterative algorithm for computing incomplete factorizations on GPUs, in: International Conference on High Performance Computing, Springer, 2015, pp. 1–16.

[12] E. Coleman, M. Sosonkina, Evaluating a Persistent Soft Fault Model on Preconditioned Iterative Methods, in: Proceedings of the 22nd annual International Conference on Parallel and Distributed Processing Techniques and Applications, 2016.

[13] J. Elliott, M. Hoemmen, F. Mueller, A Numerical Soft Fault Model for Iterative Linear Solvers, in: Proceedings of the 24nd International Symposium on High-Performance Parallel and Distributed Computing, 2015.

[14] E. Coleman, M. Sosonkina, A Comparison and Analysis of Soft-Fault Error Models using FGMRES, in: Proceedings of the 6th annual Virginia Modeling, Simulation, and Analysis Center Capstone Conference, Virginia Modeling, Simulation, and Analysis Center, 2016.

[15] M. Baboulin, E. Coleman, A. Jamal, A. Khabou, M. 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.

[16] H. Anzt, J. Dongarra, E. S. Quintana-Ortí, Tuning stationary iterative solvers for fault resilience, in: Proceedings of the 6th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ACM, 2015, p. 1.

[17] H. Anzt, J. Dongarra, E. S. Quintana-Ortí, Fine-grained bit-flip protection for relaxation methods, Journal of Computational Science.

[18] E. Coleman, M. Sosonkina, Fault Tolerance for Fine-Grained Iterative Methods, in: Proceedings of the 7th annual Virginia Modeling, Simulation, and Analysis Center Capstone Conference, Virginia Modeling, Simulation, and Analysis Center, 2017.

[19] H. Anzt, E. Chow, J. Saak, J. Dongarra, Updating incomplete factorization preconditioners for model order reduction, Numerical Algorithms (2016) 1–20.

[20] A. Frommer, D. Szyld, On asynchronous iterations, Journal of computational and applied mathematics 123 (1) (2000) 201–216.

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

[22] T. Davis, The University of Florida Sparse Matrix Collection, `http://www.cise.ufl.edu/research/sparse/matrices/` (1994).

[23] A. Nukada, H. Takizawa, S. Matsuoka, Nvcr: A transparent checkpoint-restart library for nvidia cuda, in: Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, IEEE, 2011, pp. 104–113.

[24] K. B. Ferreira, R. Riesen, R. Brighwell, P. Bridges, D. Arnold, libhashckpt: hash-based incremental checkpointing using gpus, in: European MPI Users' Group Meeting, Springer, 2011, pp. 272–281.

[25] G. Bronevetsky, B. de Supinski, Soft error vulnerability of iterative linear algebra methods, in: Proceedings of the 22nd annual international conference on Supercomputing, ACM, 2008, pp. 155–164.

[26] J. Elliott, F. Mueller, M. Stoyanov, C. Webster, Quantifying the impact of single bit flips on floating point arithmetic, preprint.