Fault Recovery Methods for Asynchronous Linear Solvers

Evan Coleman Erik J. Jensen Masha Sosonkina

August 24, 2020

Abstract

This study seeks to understand the soft error vulnerability of asynchronous iterative methods, with a focus on stationary iterative solvers such as Jacobi. A theoretical investigation into the performance of the asynchronous iterative methods is presented and used to motivate several fault recovery methods for asynchronous linear solvers. The numerical experiments utilize a hybrid-parallel implementation where the computational work is distributed over multiple nodes using MPI and parallelized on each node using OpenMP, and a series of runs are conducted to measure both the impact of soft faults and the effectiveness of the recovery methods. Trials are run to compare two models for simulating the occurrence of a fault as well as techniques for recovering from the effects of a fault. The results show that the proposed strategies can effectively recover from the impact of a fault and that the numerical model for simulating soft faults consistently produces fault effects that enable the investigation and tuning of recovery techniques in action.

1 Introduction

In 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 [12]. Because of this, developing algorithms that are resilient to faults is of paramount importance, especially on the road towards exascale.

Faults can broadly be divided into two categories: hard faults and soft faults [10]. 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, although its interruption may occur as a result of their impact. Transient soft faults are typically caused by solitary bit flips, occurring due to such events as, for example, radiation, hardware malfunction, or data-cache set incorrectly. The most important aspect to recovering from a soft fault is successful fault detection. However, this is often difficult in the case of a soft fault since, though it corrupts data, it does not cause direct interruption to the flow of the iterative process. Many detection techniques rely on choosing an appropriate tolerance to check a property of the algorithm that has predictable behavior (a residual that is monotonically decreasing, a known property concerning a vector or matrix norm, etc); a tolerance that is too loose will allow potentially harmful errors to go undetected, while a tolerance that is too strict may report a fault when none actually occurred ("false positive") and cause the program to do extra work to recover from a non-existent problem. The balance in choosing the correct fault tolerance method to recover from soft faults can be application dependent. The focus of this work is on developing recovery methods for transient soft faults that are specific to asynchronous linear solvers, specifically to stationary linear solvers such as Jacobi.

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

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

is updated in an asynchronous manner, with the ultimate goal of finding a fixed point, i.e., a location $\boldsymbol{x}^*$ in the domain such that $\boldsymbol{x}^* = G(\boldsymbol{x}^*)$. This class of problems has been used in a wide variety of applications including: the solution of linear systems [3], the preconditioning of linear solvers [15], optimization [28], and techniques for solving partial differential equations [25], among many others.

Asynchronous linear solvers tend not to converge to high precision as quickly as their Krylov subspace counterparts [4]. However, they can converge very quickly to a low level of accuracy. This loss of accuracy may cause the use of asynchronous linear solvers to be suboptimal for some applications. But the fact that they are able to reach an approximate solution quickly opens up several other application areas, such as as preconditioning to a Krylov subspace solver or big data and machine learning [4]. The amount of computational work done during an asynchronous iteration can be greater compared to a synchronous solver; however, the smaller cost of synchronization may enable the asynchronous variant to be faster overall.

The authors' prior work [16] focused mainly on comparisons of three different fault-injection models. A partial checkpointing strategy was picked in that work to show experimentally asynchronous-method recovery from faults of various severity. Building on that experimental study, only two fault-injection models are used here, and the present work makes the following novel contributions:

- developed theoretical underpinning to the resilient algorithms experimented with in prior work;
- proposed a new algorithm-based approach to fault tolerance that does not rely on any form of checkpointing;
- compared a standard checkpointing approach with the other two fault recovery techniques;
- performed a wide suite of experiments to compare all three recovery techniques considered here;
- investigated their scalability to larger problem sizes and increased computational resources.

The remainder of this paper is organized as follows: Section 2 gives an overview of related studies, Section 3 provides a review of asynchronous iterative method and discusses an approach to model fault tolerance for asynchronous fixed point iterations (in Section 3.1). Section 4 presents general theoretical and practical considerations for the development of resilient fine-grained algorithms. Sections 5 and 6 deal with fault-recovery techniques in asynchronous linear solvers and with their implementation followed by numerical experiments, respectively. Section 7 concludes with a summary and future work outline.

2 Related Work

Asynchronous methods themselves have a long and storied history. The development initially began in earnest in the late 1960's and continued into the 1970's. The paper that started investigation into asynchronous iterative methods was by [13] and investigated whether the updates when solving a linear system via a relaxation method (e.g., Jacobi) could occur in a random, uncoordinated manner that they termed "chaotic". Later, work continued towards further generalizing results in the seminal work by [6], where a framework for analyzing asynchronous iterative methods was proposed. The texts [8, 5] provide background on asynchronous iterative methods, while the survey presented in [21] provides a summary of results circa the year 2000.

Pertinent to the research conducted in this paper is work on termination criteria for asynchronous iterative methods. Historically, research along these lines arose much earlier (see [7, 30]), and has continued recently with work by [34, 27, 26] who, in addition, analyze the effect of round-off errors on asynchronous iterative methods. More recently, [24] investigated termination criteria for distributed asynchronous iterative methods similar to the asynchronous iterative methods tested in this study.

Fault tolerance for iterative methods, both stationary solvers and Krylov subspace solvers, has been studied extensively in recent years, including work specific to asynchronous iterative methods. For example, a fine-grained scheme for fault tolerance for (possibly asynchronous) stationary iterative solvers was recently proposed [3]; the method developed therein has many similarities with the algorithm-based fault tolerance check used here, but differs in some of the details. Analysis of fault tolerance for synchronous fixed point algorithms has been provided by [35]. The work presented here in Section 3.1 builds upon this analysis by extending it to the case where updates can occur asynchronously.

Characterization of the effects of the faults on solvers has been conducted by [11] for a variety of Krylov subspace methods, as well as by [31] for a more detailed analysis of the operations inside of the preconditioned Conjugate Gradient algorithm. Fault detection for iterative methods in linear algebra has been also studied. Both [14] and [18] develop detection mechanisms by verifying the orthogonality of the basis generated for the Krylov subspace, and [33] provide a set of algorithmic techniques for efficient fault detection. Fault tolerance for specific iterative methods has been investigated in, e.g., the work on FT-GMRES by [10], on S-ABFT-PCG by [32], or on analyzing the sensitivity of and developing fault detection criteria for the Conjugate Gradient method by [2].

Numerically modeling the occurrence of soft faults as some form of (possibly non-deterministic) data corruption as opposed to directly flipping bits inside the data structures of the executing program has gained traction recently. A general position paper on the efficacy of treating soft faults as numerical corruption has been provided by [19]. Several numerically based fault models have been utilized in recent studies. These include a fault model that is predicated on shuffling the components of an important data structure [20] as well as a perturbation based model [35].

3 Framework for Asynchronous Iterations

In fine-grained parallel computation involving linear systems, each component, such as a matrix element or vector entry, is updated in a manner that does not require specific information from the computations of other components while the update is being made. This allows for each computing element to act independently from all other computing elements. Depending on the size of the problem and the computing environment, each computing element may be responsible for updating a single entry or be assigned a block of multiple components.

There are several ways to define asynchronous iteration mathematically (see the survey by [21]); informally, the data for each component x_i may or may not be updated in the iteration that just occurred. Following standard assumptions about the amount of allowable delay on updates for the different components, convergence of many iterative algorithms is preserved. This will lead to different update patterns for individual component functions, each of which will be utilizing components that may be updated a different number of times. The convergence of parallel fixed point iterations is discussed in the literature for both synchronous [1] and asynchronous [21] cases. Note that there exist many possible combinations of synchronous and asynchronous updates. For example, blocks of components could be scheduled for updates asynchronously, but the individual component updates could be made in a synchronous manner inside blocks.

The general mathematical model used throughout this paper comes from [21]. In order to keep the mathematical model as general as possible, consider a function $G : D \to D$, where D is a domain that represents a product space $D = D_1 \times D_2 \times \cdots \times D_m$. The goal is to find a fixed point of the function G inside the domain D. To this end, a fixed point iteration is performed such that

$$\boldsymbol{x}^{(k+1)} = G(\boldsymbol{x}^{(k)}) \tag{2}$$

and a fixed point is declared if $\boldsymbol{x}^{(k+1)} \approx \boldsymbol{x}^{(k)}$. Note that the function G has internal component functions G_i, for each sub-domain D_i in the product space D. In particular, $G_i : D \to D_i$, which gives that for $\boldsymbol{x} = (x_1, x_2, \dots, x_m) \in D$,

$$\begin{aligned} \boldsymbol{x} = G(\boldsymbol{x}) &= G(x_1, x_2, \dots, x_m) \\ &= (G_1(\boldsymbol{x}), G_2(\boldsymbol{x}), \dots, G_m(\boldsymbol{x})) \in D. \end{aligned} \tag{3}$$

Further, the assumption is also made that there is some finite number of processing elements $P_1, P_2, \dots, P_r$ each of which is assigned to a block of components $B_1, B_2, \dots, B_m$ to update. Note that the number of processing elements r will typically be significantly smaller than the number of blocks m to update. With these assumptions, the computational model that defines asynchronous iterative methods can be stated as in Algorithm 1. The computational model presented

Algorithm 1: General Computational Model

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

in Algorithm 1 allows for either synchronous or asynchronous computation; it only prescribes that an update has to be made as an "atomic" operation (in line 5), to prevent possible race conditions. If each processing element P_l is to wait for the other processors to finish each update, then the model describes a parallel synchronous form of computation. On the other hand, if no order is established for P_ls, then an asynchronous form of computation arises.

To continue formalizing this computational model a few more definitions are necessary. First, set a global iteration counter k that increases every time any processor reads $\boldsymbol{x}$ from common memory. At the end of the work done by an individual processor p, the components associated with the block B_p will be updated. The results of a given iteration can be expressed in a vector, $\boldsymbol{x} = (x_1^{s_1(k)}, x_2^{s_2(k)}, \ldots, x_m^{s_m(k)})$ where the function $s_l(k)$ indicates how many times a specific component has been updated. Finally, a collection of individual components can be grouped into a set I^k that contains all the components that were updated on the kth iteration. Given these basic definitions, the three following conditions, along with the model presented in Algorithm 1, constitute a working mathematical framework for asynchronous computation.

Definition 1. If the following three conditions hold:

1. $s_i(k) \leq k$, *i.e., only components that have finished computing are used in the current approximation.*
2. $\lim_{k\to\infty} s_i(k) = \infty$, *i.e., the newest updates for each component are used.*
3. $|k \in \mathbb{N} : i \in I^k| = \infty$, *i.e., all components will continue to be updated.*

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

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

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

This mathematical framework allows for obtaining results on fine-grained iterative methods that are either synchronous or asynchronous (although the three conditions given in Definition 1 are unnecessary in the synchronous case). Then, the iterative updates can be expressed in a possibly asynchronous format using the functions $s_i(k)$ that keep track of how many updates have occurred for each individual component as follows:

$$\begin{aligned} x_1^{(k+1)} &= G_1\left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_n^{(s_n(k))}\right), \\ x_2^{(k+1)} &= G_2\left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_n^{(s_n(k))}\right), \\ x_3^{(k+1)} &= G_3\left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_n^{(s_n(k))}\right), \\ &\vdots \\ x_n^{(k+1)} &= G_n\left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_n^{(s_n(k))}\right). \end{aligned} \tag{4}$$

The following two examples show how the framework detailed above can be used to express common fixed point iterations [1]:

Example 1. *Synchronous iterations are given by enforcing the additional condition that $s_i(k) = k$ for all blocks (subdomains) i and for each iteration k.*

Example 2. *The Jacobi method is given by letting $I^k = \{1, 2, \ldots, m\}$ for all k, i.e., all components are updated on every iteration. The synchronous block Gauss-Seidel method can be defined by letting $s_i(k) = k$ and $I^k = \{k \mod m+1\}$.*

Finally, note that the initial set of conditions required for convergence of a fixed-point iteration stems from the fundamental Banach Fixed-Point Theorem, stated here for the sake of completeness:

Theorem 1 (Banach Fixed Point Theorem)**.** *Assume that the space D is complete with respect to the norm $\|\cdot\|$. If the operator $G : D \to D$ satisfies,*

$$\|G(\boldsymbol{x}) - G(\boldsymbol{y})\| \leq \gamma \|\boldsymbol{x} - \boldsymbol{y}\| \tag{5}$$

with $\gamma \in [0, 1)$ for any $\boldsymbol{x}, \boldsymbol{y} \in D$, then G is said to be a contraction map with respect to the norm $\|\cdot\|$ and there is a unique fixed point $\boldsymbol{x}^* \in D$ such that the sequence defined by $\boldsymbol{x}^{(k+1)} = G\left(\boldsymbol{x}^{(k)}\right)$, converges to the fixed point $\boldsymbol{x}^*$.

This theorem will be used to help develop resilient algorithms in Section 4.

3.1 Probabilistic Analysis of the Impact of Soft Fault

The material in this subsection is drawn primarily from [35], and is included to motivate the view of the impact of a fault adopted throughout this paper. This, in turn, provides a foundation for the detection and recovery strategies adopted in Section 4 as well as the choice of soft fault model utilized in the numerical experiments detailed in Section 6.

Recall that a soft fault is an error that is undetected by the executing program or the operating system and introduces silent data corruption into the result of the operation where the fault is incurred. An undetected computing fault that occurs may end up causing either divergence or stagnation of the iterative algorithm. In order to determine the conditions for convergence in this case, the possible amount of data corruption must first be quantified. For any operation G on the vector $\boldsymbol{x} \in D$, where $G : D \to D$, some number of vector components of from the output can be affected by a fault. Even for a relatively trivial map G, it is often hard to pin down exactly where the error occurs if the corrupted output is the only location that the fault visibly manifests. Further, the corruption from one component can spread to other components and such a proliferation often occurs in a non-deterministic manner that depends on the timing and magnitude of the error in question. In the case of a fault occurring when computing a vector $\boldsymbol{x} \in D$, the resultant vector, denoted by $\hat{x} \in D$, can be expressed as

$$\hat{x} = \boldsymbol{x} + \tilde{x} \tag{6}$$

for some random vector $\tilde{x} \in D$. That is, the vector obtained differs from the vector that would arise in the fault-free case by some unknown amount that is linked to an unknown probability distribution, where for every operation there is some positive probability of having a fault occur.

In the work by [35], this probabilistic modeling of faults is done via a two-level probability combination. The first level indicates whether or not a fault has occurred on a particular iteration k and is represented by a sequence of discrete Bernoulli parameters $\boldsymbol{p} = \{p_1, p_2, p_3, \ldots\}$ signifying that a fault occurred at iteration k. Recall that in the asynchronous case, the iteration counter is incremented every time that a processor reads the vector

$$\boldsymbol{x}^{(k)} = \left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, \cdots, x_m^{(s_m(k))}\right) \tag{7}$$

from memory. Since the effect of the fault is represented as a random perturbation $\tilde{x}$ to the value of the iterate that would have been calculated in a fault-free calculation, it can be modeled as a sequence of unknown probability distributions $\boldsymbol{P} = \{\mathscr{P}_1, \mathscr{P}_2, \mathscr{P}_3, \ldots\}$, associated with each Bernoulli parameter. This creates a sequence of pairs $(\boldsymbol{p}, \boldsymbol{P}) = \big((p_i, \mathscr{P}_i),\ i = 1, \ldots, m\big)$ that are associated with each computational step. Convergence can then be defined by having an arbitrarily small expected value of the difference between a nominal fault-free run, and the result of a run where faults occur. This ensures that the final iterate of the execution subject to faults is within some tolerance of the final iterate in a fault-free case and does not require the same number of iterations till convergence.

The iterative nature of the algorithms being considered offers a natural ability to correct for many faults. For example, if a fault occurs on the iteration $(F-1)$, then the resultant Fth iteration can be written

$$\hat{x}^{(F)} = G\left(\boldsymbol{x}^{(F-1)}\right) + \tilde{x}^{(F)} \tag{8}$$

$$\hat{x}^{(F)} = \boldsymbol{x}^{(F)} + \tilde{x}^{(F)} \tag{9}$$

for some unknown perturbation $\tilde{x}^{(F)}$. However, as long as $\hat{x}^{(F)} \in D$ and $\hat{x}_i^{(F)} \in D_i$ for all $i \in \{1, 2, \ldots, m\}$, this can be seen to generate a new sequence that will still converge to the desired fixed point.

Further, the probability of a fault can be limited by assuming a so-called *selective reliability* [10], in which certain computations are executed in a faster, but less reliable setting while others are performed in a high reliability mode. In a selective reliability scenario, it can be assumed that operations executed in high reliability mode have a negligible chance of soft fault occurrence. This assumption has been utilized in earlier studies [10, 29] to create fault tolerant algorithms in which certain sections are assumed to complete without faults.

4 Developing Resilient Algorithms

The Banach Fixed-Point Theorem may be used to detect anomalous behavior by monitoring the progression of the difference between successive iterates. In particular, define a variable to hold the difference on the $(k+1)$st iterate

$$\delta^{(k+1)} = \|\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}\|, \tag{10}$$

and if it is known that the fault-free behavior of the algorithm guarantees

$$\delta^{(k+1)} < \gamma \delta^{(k)} , \tag{11}$$

a fault can be declared if this condition is violated. Alternatively, for some fixed point algorithms there are residuals that can be used to judge progress of an algorithm. However, the computation of residuals may be expensive.

In the case of asynchronous iterative methods, creating a global synchronization point every iteration in order to calculate $\delta^{(k+1)}$ is counter-productive, and therefore, a different approach is needed. When $G : D \to D$ is either a linear or nonlinear operator (see Theorems 4.1 and 4.4 of [21], respectively), convergence is guaranteed by ensuring that the spectral radius of the operator $|G|$ (if G is linear) or the spectral radius of the Jacobian $|G'|$ (if G is nonlinear) is less than 1. A sufficient but not necessary condition for convergence of fine-grained iterative methods is given by the proposed result below.

Theorem 2. *Assume that the space D is complete with respect to the norm $\|\cdot\|$. Let $D = D_1 \times D_2 \times \cdots \times D_m$. For $\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$, let $G : D \to D$ be expressed as*

$$\begin{aligned}
x_1^{(k+1)} &= G_1\left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_m^{(s_m(k))}\right) , \\
x_2^{(k+1)} &= G_2\left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_m^{(s_m(k))}\right) , \\
x_3^{(k+1)} &= G_3\left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_m^{(s_m(k))}\right) , \\
&\vdots \\
x_m^{(k+1)} &= G_n\left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_m^{(s_m(k))}\right)
\end{aligned} \tag{12}$$

using the framework defined by Algorithm 1 and Definition 1. Let $\boldsymbol{x}^{(s(k))} = \left(x_1^{(s_1(k))}, x_2^{(s_2(k))}, x_3^{(s_3(k))}, \cdots, x_m^{(s_m(k))}\right)$. If each component function $G_i : D \to D_i$ is a contraction mapping, then the asynchronous fixed-point iteration defined by

$$\begin{aligned}
\boldsymbol{x}^{(k+1)} &= G\left(\boldsymbol{x}^{(s(k))}\right) \\
&= \left(G_1\left(\boldsymbol{x}^{(s(k))}\right), G_2\left(\boldsymbol{x}^{(s(k))}\right), \ldots, G_m\left(\boldsymbol{x}^{(s(k))}\right)\right)
\end{aligned} \tag{13}$$

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

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

$$\|G_i(\boldsymbol{x}) - G_i(\boldsymbol{y})\| \le \gamma_i \|\boldsymbol{x} - \boldsymbol{y}\| \tag{14}$$

for all $\boldsymbol{x}, \boldsymbol{y} \in D_i$. Set $\gamma = \max\limits_{i \in \{1,\ldots,m\}} \gamma_i$. Then $\gamma < 1$ and

$$\|G_i(\boldsymbol{x}) - G_i(\boldsymbol{y})\| \le \gamma \|\boldsymbol{x} - \boldsymbol{y}\| , \tag{15}$$

which also gives that

$$\|G(\boldsymbol{x}) - G(\boldsymbol{y})\| \le \gamma \|\boldsymbol{x} - \boldsymbol{y}\| \tag{16}$$

for all $\boldsymbol{x}, \boldsymbol{y} \in D$, so that Theorem 1 can be applied to give convergence for the entire iterative procedure. □ □

An implication from Theorem 2 is that, if the separate component functions are individually contractive, then a check similar to Eq. (11) can be used on each component independently. This would allow the fine-grained asynchronous nature of the algorithm to be preserved without introducing any unnecessary synchronization points. In this case, define a component-wise difference

$$\delta_i^{(k+1)} = \|\boldsymbol{x}_i^{(k+1)} - \boldsymbol{x}_i^{(k)}\| , \tag{17}$$

which are specific to the individual subdomains D_i. Then, for fixed point iterations that satisfy the component-wise contractive property specified in Theorem 2, the following relation holds

$$\delta_i^{(k+1)} \le \gamma_i \delta_i^{(k)} , \tag{18}$$

and can then be used in the development of a fault-tolerant fine-grained fixed-point algorithm to be employed asynchronously. If $\delta_i^{(k)}$ is computed every iteration, then it can be monitored to see if it increases, may indicate of a fault. Note that this check will be effective for faults modeled as arbitrary data corruption that may affect any number of elements as shown in Eq. (6) in Section 3.1. Algorithm 2 presents a fine-grained approach towards fault tolerance that does not require any explicit recovery technique or global communication.

Algorithm 2: Resilient Parallel Fixed Point Iteration

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

In general, asynchronous updates to either component-wise differences or residuals remove the guarantee of monotonicity, even when the progression towards solution is monotonic for all-synchronous updates. Note that the non-monotonic behavior will depend on the algorithm, problem, and number of processors. As an example of this non-monotonic behavior, consider the solution of the Laplacian with the Dirichlet boundary conditions, discretized over a 10 by 10 grid with centered finite differences by the asynchronous Jacobi method, shown in Fig. 1. Here, a component residual is defined by $\|\boldsymbol{r}_j\| = \|\boldsymbol{b}_j - \boldsymbol{A}\boldsymbol{x}_j\|$, where the vector $\boldsymbol{x}_j \in D_j$, and the total of 100 vector components are split evenly between two processors.

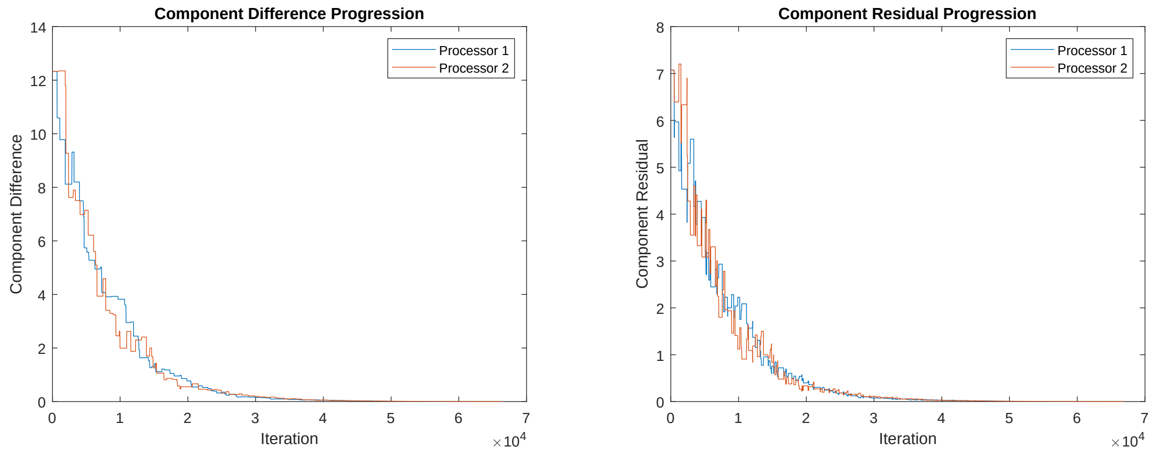


Figure 1: Convergence of the asynchronous Jacobi method for two simulated processors using both component-wise differences (left) and component-wise residuals (right) for the Laplacian problem.

Despite the general trend for both solution component differences and component residuals to decrease, it is possible that Algorithm 2 accepts false positives. Using a threshold parameter $\tilde{\gamma}_j > \gamma_j$ may help prevent false positives. For example, if $\tilde{\gamma}_j \gg \gamma_j$ then the check for anomalous behavior

$$\delta_j^{(k+1)} \leq \tilde{\gamma}_j \delta_j^{(k)} \tag{19}$$

can be used to filter excessively bad while small errors, such as bit flips in less significant bits in the mantissa, are allowed to remain since the iterative nature of the fixed-point algorithm may still lead to convergence. In the case of $\tilde{\gamma}_j \gg \gamma_j$, a single value of $\tilde{\gamma}_j$ may be employed for all subdomains $D_j, \; j = 1, \ldots, m$.

There are two additional concerns that merit consideration here. First, if computation of a residual is avoided entirely, there is a danger of terminating the fixed point iteration too early if only the differences in the solution vector components are used to determine convergence. Second, a stagnation may take place if too many consecutive iterates are rejected. One approach to alleviate these difficulties is to introduce a new parameter, α, that controls the frequency with which the check on progression (see Line 5 of Algorithm 2) is made. In particular, the differences may be compared every α iterations (see Lines 5 and 7 in Algorithm 3). For each of the differences, the initial value $\delta_j^{(k_0)}$ is set to a sufficiently large value $\Delta_j^{(0)}$ in Line 2. Computing the first update in a highly reliable computational mode allows this value to be set appropriately, but the algorithm should proceed as expected for any large value since $\delta_j^{(k_0)}$ will be updated every α iterations (Line 11).

Algorithm 3: Enhanced Resilient Parallel Fixed Point Iteration

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

For the type of execution shown in Fig. 1, observe that the behavior of the residual components exhibits far less monotonic behavior than that of the norm of the differences between two successive iterates, which suggests that a larger value of α may be necessary. In addition, since the progression of solution component differences appears to be smoother, using them—with an appropriate threshold for γ—rather than component residuals may be more beneficial to decrease the detection of false positives. Acceptable values for α as well as γ will be explored experimentally in Section 6.

5 Techniques for Recovery from Soft Faults

Throughout this and subsequent sections, a focus will be on stationary linear solvers, such as Jacobi and Gauss-Seidel, in order to consider concrete implementations of the theoretical developments discussed in earlier sections. However, the ideas developed in Section 4 are applicable and may be adapted to any fixed point iteration method of interest in which a difference (residual) between iterations may be computed and assessed. Generally, the same questions that arise in striking an efficient balance between computation and resilience when examining the asynchronous Jacobi algorithm should be expected to arise for any fixed point iteration. As with any fault tolerance techniques, care needs to be taken to not increase the amount of required computation to a point that the resulting algorithm becomes unbearably computationally expensive.

The asynchronous Jacobi method is a relaxation method for solving linear systems of the form $\boldsymbol{Ax} = \boldsymbol{b}$, in which each solution component my be computed separately. Consider the ith row of the matrix A. Then, solving for the ith component of the solution, x_i, gives

$$x_i = \frac{-1}{a_{ii}} \left[\sum_{j \neq i} a_{ij} x_j - b_i \right] . \tag{20}$$

When computed in an iterative manner, successive updates to x_i in Eq. (20) may be used as component differences to monitor convergence as discussed in Sections 3 and 3.1. Expressing the Jacobi method in a block matrix form gives

$$\boldsymbol{x} = -\boldsymbol{D}^{-1}((\boldsymbol{L}+\boldsymbol{U})\boldsymbol{x}-\boldsymbol{b}) \tag{21}$$

$$= -\boldsymbol{D}^{-1}(\boldsymbol{L}+\boldsymbol{U})\boldsymbol{x}+\boldsymbol{D}^{-1}\boldsymbol{b}\,, \tag{22}$$

where $\boldsymbol{D}$ is the diagonal portion of $\boldsymbol{A}$, and $\boldsymbol{L}$ and $\boldsymbol{U}$ are the strictly lower and upper triangular portions of $\boldsymbol{A}$ respectively. This gives an iteration matrix $\boldsymbol{C} = -\boldsymbol{D}^{-1}(\boldsymbol{L}+\boldsymbol{U})$, which may be used to define the global function $G : D \to D$ discussed in Sections 3 and 3.1. If the fixed point iteration is asynchronous, an update to every element x_i is made with the most recently updated information available at the update time. Pseudocode for the asynchronous Jacobi algorithm is provided in Algorithm 4, where it may be observed that a processor P_l (Line 4) is not necessarily available to compute updates at the given time. This lack of determinism in the update order leads to the asynchronous nature of the algorithm shown in Algorithm 4.

Algorithm 4: Asynchronous Jacobi

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

Next, three techniques for the recovery from faults are discussed followed by an overview of the fault-injection models that will be used in numerical experiments as described in Section 6. Each technique will be discussed in relation to the asynchronous Jacobi algorithm provided in Algorithm 4.

5.1 Global Checkpointing

If the global checkpointing (denoted CP) recovery method is used, then during the course of the fixed point iteration, at periodic intervals, all elements of the current iterate $\boldsymbol{x}^{(k)}$ are saved to memory. If a fault is detected, all elements of the corresponding iterate $\boldsymbol{x}^{(F)}$ are reset to the last known good state $\boldsymbol{x}^{(k)}$. CP tends to be robust to the occurrence of a soft fault, but may be very slow computationally. Furthermore, creating reliable techniques for detecting whether or not a soft fault has occurred often requires regular global communication.

At regular (or semi-regular) intervals during the asynchronous fixed point iteration, the latest information available can be used to calculate the current residual

$$\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}\,. \tag{23}$$

By keeping track of the residual over time, a fault can be declared if

$$\|\boldsymbol{r}^{(k+\alpha)}\| > \gamma^g \|\boldsymbol{r}^{(k)}\|\,, \tag{24}$$

where α is the delay in calculating the residual and γ^g is a constant selected by the user. Values of γ^g close to 1 assume monotonic decrease in the residual and may therefore declare false positives by detecting a non-existent fault, while values significantly larger than 1 may not detect anomalous behavior if the effect is not sufficiently large. If a fault is detected all elements of the current iterate $\boldsymbol{x}^{(F)}$ are reset to the last known good state.

5.2 Partial Checkpointing

In partial checkpointing (denoted as PCP), all elements of the current iterate $\boldsymbol{x}^{(k)}$ are saved to memory with some regularity, as in the case of global checkpointing. The difference, however, is that if a fault is detected, only some components of $\boldsymbol{x}^{(F)}$ are reset to their last known good state in PCP. Specifically, only the components that are determined to be affected by the fault are rolled back. PCP requires a finer-grained check on whether or not a fault has occurred, so that the location of the fault can be specified more precisely. Because of this, the partial checkpointing definition has natural affinity with fine-grained iterative methods: It is possible to detect faults for individual components and act on them independently (asynchronously); similarly, the computational model of fine-grained (asynchronous) iteration allows for the components to become out-of-sync with one another. Note that the convergence of a fine-grained iterative method is not typically affected negatively if the components that are assigned to a particular processor are reset to a state several iterations behind the other components.

As opposed to Eq. (23), where the global residual is calculated, in PCP, only the portion of the residual local to a processor p

$$\boldsymbol{r}_p = (\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x})_p. \tag{25}$$

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

$$\|\boldsymbol{r}_p^{(k+\alpha)}\| > \gamma^p \|\boldsymbol{r}_p^{(k)}\|\,, \tag{26}$$

where α is the delay in calculating the residual and γ^p is a constant selected by the user. The value of γ^p may be significantly higher that that in CP since the decrease of the local residual may not be monotonic. As in CP, values of γ^p close to 1 assume monotonic decrease in the residual and may therefore declare false positives by detecting a non-existent fault, while values significantly larger than 1 may not detect anomalous behavior if the effect is not sufficiently large. If a fault is detected, then all of the elements of the current iterate $\boldsymbol{x}^{(F)}$ that are assigned to the processor that detects a fault, i.e., $\boldsymbol{x}_p^{(F)}$, are reset to the last known good iterate.

5.3 Algorithm Based Fault Tolerance

Algorithm based fault tolerance (ABFT) represents a wide class of algorithms that contain a variety of techniques for making algorithmic modifications to withstand faults without relying on checkpointing techniques necessarily. Here, a method for restoring the convergence of asynchronous Jacobi without requiring computation of the residual is provided. Instead, this ABFT variant is based on Algorithm 3 developed in Section 4. At iteration k, the difference in successive iterates of each component x_i is defined as:

$$\delta_i^{(k)} = |x_i^{(k)} - x_i^{(k-1)}|. \tag{27}$$

Due to the non-monotonicity of the progression, each $\delta_i^{(k)}$ is compared to the maximum $\hat{\delta}^{(k)}$ over all components, and a fault can be declared if

$$\delta_i^{(k+\alpha)} > \gamma^a \hat{\delta}^{(k)}\,, \tag{28}$$

where α is the iteration delay in comparing the differences, and γ^a is a constant selected by the user. If the check is failed, the new update to the ith component is rejected. Calculating the maximum value over all components incurs additional communication overhead, but helps to avoid false-positive fault identification when subdomains exchange information.

5.4 Fault Injection Techniques

In a majority of studies, the occurrence of an undetected soft fault is treated as a bit flip (see, e.g., [11]). Traditionally, bit flips are injected randomly according to a given distribution (often, a Poisson or Weibull distribution), or else in a more frequent manner designed to showcase worst case behavior. However, as the effect of a bit-flip (i.e., the amount of data corruption introduced) can vary wildly depending on which bit is affected, this necessitates a large number of runs to reveal statistically average behavior [19]. In this study, a series of experiments utilizing the direct injection of bit flips into memory is presented; additionally, more generic fault injection techniques are included.

Following the methodology outlined in [19], the numerical fault models used here are inspired by the idea of modeling an undetected soft fault as data corruption (c.f. Eq. (6) in Section 3.1). That is, instead of trying to model the exact impact of a fault on future large scale HPC machines, faults are treated as corrupted data where the size of the corruption can be controlled in an effort to produce consistent worst case behavior and help with the development of fault tolerant algorithms. The goal of considering a variety of soft-fault models is to produce fault-tolerant algorithms that are not too dependent on the precise mechanism of a fault, such as a bit-flip, in future computing platforms. Note that, in this work, faults are injected only into the data used by the algorithm as opposed to the metadata that includes also pointers, indices, and other data-structure descriptions, because the metadata, while necessary to be fault-free also, is tied to a specific implementation of the given algorithm on a given architecture, which is beyond the scope of this paper.

5.4.1 Bit-Flip Soft Fault Model (BFSFM)

The first method of simulating a fault adopted in this study is via the direct injection of a bit-flip into a data structure. Soft faults typically, at least for current HPC hardware, manifest as bit-flips, which makes it important to include analysis that responds to the effects of having a bit-flip occur during the run.

5.4.2 Perturbation-Based Soft Fault Model (PBSFM)

This model treats faults as random perturbations to the faulty subdomain, an approach that has been used in other recent studies [35, 17]. In the version of the PBSFM used here, a small random perturbation τ that is sampled from a uniform distribution of a given size is injected transiently into each component representing a value of the targeted data structure. For example, if the targeted data structure is a vector $\boldsymbol{x}$ and the maximum size of the perturbation-based fault is ε, then proceed as follows:

1. generate a random number $\tau_i \in (-\varepsilon, \varepsilon)$, and
2. set $\hat{x}_i = x_i + \tau_i$ for all applicable values of i.

The resultant vector $\hat{x}$ is, thus, perturbed away from the original vector $\boldsymbol{x}$. Note the direct comparison with the arbitrary data corruption modeled by Eq. (6) in Section 3.1.

5.4.3 Analysis of the Two Fault Injection Models

To compare the potential effects of the fault injection models used here, with an emphasis on the total amount of data corruption induced, a short analysis is presented for the two dimensional discretization of the Laplacian on the same domain that will be studied further in Section 6. Letting the initial $\boldsymbol{x}$ be a vector of all zeros and the initial $\boldsymbol{b}$ be a vector of all ones, the 50th iterate of $\boldsymbol{x}$ examined here, denoted $\boldsymbol{x}^{(50)}$, and the the effect of fault models is examined for this subdomain localized as follows. Consider the first quarter of the components of $\boldsymbol{x}$ of which the first tenth of the components is affected by faults. Note that this localization of sub-domain is motivated primarily by the implementation details (described in Section 6.2) and does not influence the outcomes of the fault-model behavior. The PBSFM and BFSM parameters compared are as follows:

- PBSFM– large (L): $\tau_i \in (-10^{16}, 10^{16})$, medium (M): $\tau_i \in (-10^{8}, 10^{8})$, and small (S): $\tau_i \in (-10^{2}, 10^{2})$.
- BFSFM– one vector component randomly selected, in which one bit is randomly selected.

A total of 10,000 trials were run, and aggregate data is presented in Table 1. The total amount c of data corruption is measured as $c = ||\boldsymbol{x} - \hat{x}||$, where $\hat{x}$ represents the iterate under study $\left(\boldsymbol{x}^{(50)}\right)$ with the specified fault injected. Mean and median information is provided over all the 10,000 trials as well as the average and standard deviation of the logarithm of c, which provides some insight into the average order of magnitude of corruption and how wide the spread of potential outcomes is. Note that the range of impacts is wider for the BFSFM, but that the average impact is worse for the PBSFM.

Table 1: Comparison of fault injection models.

	Mean(c)	Median (c)	Mean(log(c))	Std(log(c))
PBSFM (L)	3.65E+17	3.65E+17	40.44	0.007
PBSFM (M)	3.65E+09	3.65E+09	22.02	0.007
PBSFM (S)	3.65E+03	3.65E+03	8.20	0.007
BFSM	1.13E+304	3.05E-05	-0.81	52.8

6 Numerical Experiments

Experiments were conducted on the Turing High Performance Computing cluster at Old Dominion University, which contains 190 standard compute nodes, 10 GPU nodes, 10 Intel Xeon Phi Knight's Corner nodes, and 4 high memory nodes, connected with a Fourteen Data Rate (FDR) InfiniBand network. Compute nodes contain 16–32 cores and 128 GB of RAM. Data was collected using 3 regular memory nodes: 2 for computation and 1 for the master process. Each node contains 2 sockets, each with 10 Intel Xeon E5-2670 v2 2.50 GHz cores. The experiments conducted here make use of nodes that have 20 cores each. The experiments assign two MPI processes to each node (1 per socket). The experiments shown in Section 6.4, Section 6.5, and Section 6.6 use 3 nodes (up to 60 cores), while the larger experiments in Section 6.7 scale from the original 3 nodes up to 33 nodes (660 cores).

6.1 Asynchronous Jacobi Solver for 2D Laplacian

Partial differential equations model systems in which continuous variables, such as temperature or pressure, change with respect to two or more independent variables, such as time, length, or angle. Poisson's equation in two dimensions,

$$\nabla^2 \boldsymbol{\phi} = \frac{\partial^2 \boldsymbol{\phi}}{\partial x^2} + \frac{\partial^2 \boldsymbol{\phi}}{\partial y^2} = \boldsymbol{b}, \tag{29}$$

is fundamental for modeling equilibrium and steady state problems, such as incompressible fluid flow or heat transfer. In this work, the domain is discretized uniformly with each grid point a distance of 1 apart. This allows the partial differential equation to be discretized via finite-differences, for example, as follows:

$$\begin{aligned}\left(\nabla^2 f\right)(x,y) = f(x-1,y) + f(x+1,y) + f(x,y-1)\\ + f(x,y+1) - 4f(x,y)\,,\end{aligned} \tag{30}$$

which represents a two-dimensional discrete Laplacian operator using a five-point stencil. A special case of the Jacobi algorithm

$$v_{l,m}^{k+1} = \frac{1}{4}\left(v_{l+1,m}^{k} + v_{l-1,m}^{k} + v_{l,m+1}^{k} + v_{l,m-1}^{k}\right) \tag{31}$$

may be applied to solve a two-dimensional sparse linear system of equations on this regular grid. The numerical examples in this work use the asynchronous Jacobi algorithm (see Algorithm 4) to solve a two-dimensional finite-difference discretization of the Laplacian with Dirichlet boundary conditions.

6.2 Implementation Details of Hybrid Parallel Jacobi Solver

This asynchronous Jacobi implementation makes use of hybrid MPI-OpenMP parallelism. This implementation focuses on solving a two dimensional finite-difference discretization of the Laplacian on a 400×400 grid; including the boundary values the total problem size is 402×402. The problem is solved by a matrix-free implementation of the Jacobi algorithm whereby the approximate solution to the Laplacian is stored in place at the appropriate grid values. The work is divided among five MPI processes, but only four perform computations. One MPI process acts as a master process, which communicates with workers for memory transfer and global residual calculations. Each of the four worker processes is assigned an equal amount of the entire domain, which leads to each subdomain consisting of 200×200 grid points. Note that the working size of each subdomain grid will be 202×202 due to keeping track of the necessary halo values (i.e., a mixture of values from the boundary and neighboring subdomains). The work is parallelized inside each subdomain using OpenMP.

For an n by n grid that is equally divided among the n_p threads, each thread solves for n^2/n_p grid points, such that the grid is evenly partitioned along the y-axis. Ten OpenMP threads were used for each MPI process, which gives each thread $200 \times 20 = 4000$ vector components to compute updates for. Internally, two matrices U_0 and U_1 store the grid point values that each thread reads. As the method is asynchronous, each thread independently determines which matrix stores its newly updated vector components. Further, locks are used when updating values on boundary rows and subdomain halos, and when copying subdomain boundaries. Each thread p_n computes its local residual value every kth iteration, which it contributes to the set of residual values for the subdomain. Using an OpenMP atomic operation, a single thread copies the set of subdomain residuals, computes a sum, and sends the sum to the master MPI process. The subdomain is equally divided among all OpenMP threads, but in order to avoid a negative effect on the performance of a single OpenMP thread, communication with the master MPI process is rotated among the threads as was investigated in [22].

6.3 Implementation Details of Recovery Methods

Global checkpointing provides a *de facto* standard with which to compare the other checkpointing methods. In these experiments, the values for γ^g that were used ranged from 2.5 to 100. The value of α, representing the delay in how frequently that progress of residual was checked, was set to 4. The value of 4 was chosen after initial experiments suggested it offered a good compromise between performance and resilience. Further optimizing this parameter for both the global checkpoining and partial checkpointing methods is left as future work.

Partial checkpointing, which is based on the progression of the residual after it is recovered from all components of $\boldsymbol{x}^{(k)}$, necessitates communication among all the OpenMP threads as well as all the MPI processes. It checkpoints only based on the local portion of the residual, denoted $\boldsymbol{r}_l$. As the asynchronous computation progresses, each thread writes *periodically* the current value of the components $\boldsymbol{x}_l$ for checkpointing which it is responsible. The periodicity, α, can treated as a parameter, and in these experiments – similar to those with global checkpointing, it is set to 4. After the thread updates its components in the kth iteration $\boldsymbol{x}_l^{(k)}$, it checks the current local residual to see if a fault has occurred:

$$\|\boldsymbol{r}_l^{(k)}\| > \gamma^p \|\boldsymbol{r}_l^{(k+\alpha)}\|, \tag{32}$$

where $\gamma^p > 1$ is the checkpoint threshold, $\boldsymbol{r}_l^{(k)}$ and $\boldsymbol{r}_l^{(k+\alpha)}$ are local residuals for the iterations k and $k+\alpha$, respectively. The values of γ^p used in this study ranged from 1.01 to 1.25. At the end of an iteration, a thread compares the current component residual value to a previous value. If a fault is detected as an increase of the residual by more than the specified γ^p, the threads that detected the increase roll all of the components present in their subdomain back to the last checkpoint and continue calculating updates as before. A thread checkpoints after completing four iterations that do not require a rollback.

The algorithm-based fault tolerance (ABFT) approach attempts to define an efficient solution for fault tolerance that may scale better due to avoiding regular computation of residuals. In this implementation, the residual is still calculated periodically in order to determine convergence. Developing convergence criteria based upon other means is left as future work. Here, the periodicity with which the residual is calculated is changed dynamically; starting once every 10,000 iterations and increasing to every five iterations as the computation nears completion. In particular, the frequency α is determined as:

$$\alpha = 10^{\lfloor \log_{10}(rp(1/t^2)) \rfloor}, \tag{33}$$

where r is the current residual, p is the number of MPI processes, and t is the desired tolerance to solve for. The implementation used here puts an absolute max of $10,000$ as well as a minimum value of 5. The component differences are checked at every iteration, creating an accept/reject criteria for each new update. The values of γ^a ranges from 1.05 to 5. Initially, the maximum allowable difference is set globally to a problem dependent value. Here, the value is set to 100, which was the largest value on the grid. As the calculation progresses, this maximum allowable difference is updated by the communicating thread that is responsible for sharing the residual value at the dynamic period described above.

For a grid point in the ABFT approach, if the new difference, i.e., the difference between the current value and the update as computed according to the stencil, is less than or equal to γ times the global maximum difference, the point updates. Otherwise, if the grid point has failed to update within the last four iterations, the grid point rolls back to the checkpoint value. However, if the point fails to update after five iterations, the point is forced to update, as much as γ times the global maximum difference permits. That is, if the computed difference for a grid point is greater than the global maximum difference times γ, the point will update according to the current value plus the global maximum difference times γ.

6.4 Baseline Results

Before delving into the results regarding the impact and recovery of soft faults on the hybrid parallel iterative solver used here, a set of baseline runs is presented. The problem described in Sections 6.1 and 6.2 is solved 500 times and run times are captured via calls to `MPI_Wtime()`. A histogram showing the distribution of total run times, mean, and

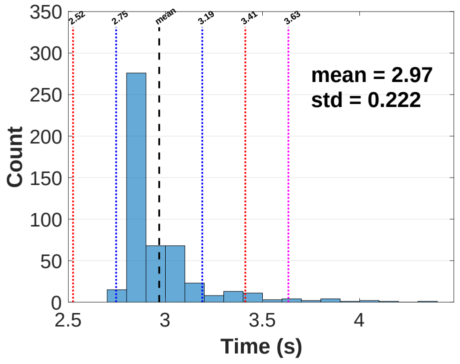


Figure 2: Distribution of run times in a fault-free environment.

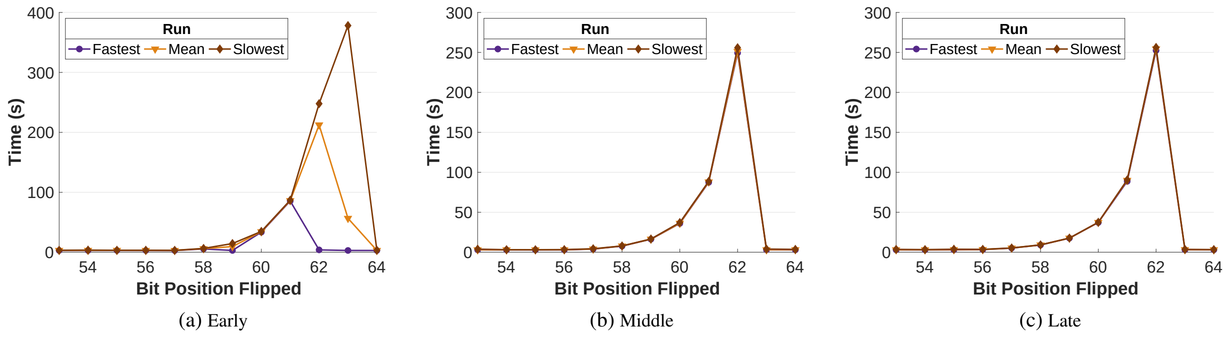


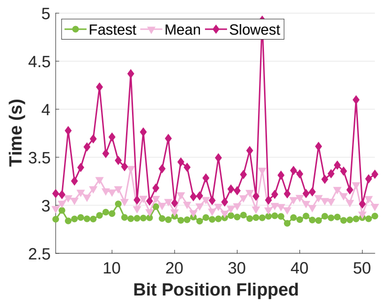
Figure 3: Effect of bit-flip faults in the exponent and sign bits for the three injection points (Early, Middle, and Late).

standard deviation is presented in Fig. 2. Some variation in run time is observed, but this is not unexpected for an asynchronous solver. A wide variation in iterations until convergence is also seen in [9], and [23] shows increased run time variation for asynchronous solvers. Here, the run time for +3 standard deviations is 1.31x the minimum run time. Almost 98% of runs are less than +3 standard deviations.

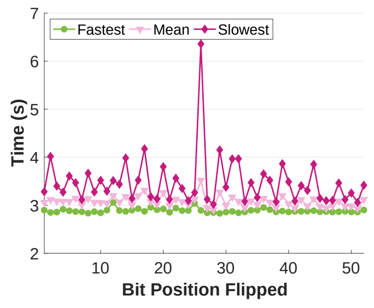
6.5 Impact of Soft Faults

The following perturbation τ values were taken from a set of intervals $(-10^{2j}, 10^{2j})$ for $j = 1, \ldots, 8$. Based on the mean runtime of $2.97s$, shown in Fig. 2, three different fault injection points were used as follows: *early*, which is equal to $0.1s$; *middle*, which is equal to $1.2s$, and *late*, which is at $2.5s$. Figures 3 to 5 show the effects from faults injected by each model at *early*, *middle*, and *late* time points. In addition, the effects of bit flips restricted to sign or exponent are distinguished from those restricted to mantissa in separate plots Fig. 3 and Fig. 4, respectively. Each experiment was replicated seven times on Turing. The plots show the results from the fastest, slowest, and average of these seven runs. In all experiments, the solver converged to a solution, with a relative residual reduction of $t = 10^{-7}$.

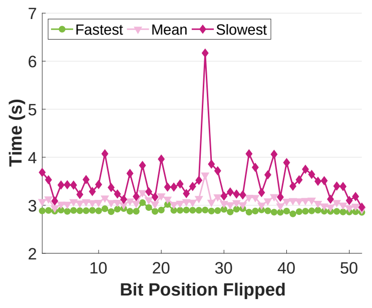
Figure 3 shows that flipping an exponent bit early in the run, when grid point values may still be small, might not be as deleterious as a later bit flip. In general, across all the fault models, faults injected early tend to have more of an effect on the total time for the solve to complete. Note also that, while bit-flip faults in the exponent and sign bits can have a catastrophic effect, bit flips occurring in the mantissa have very little impact on the performance of the solver (cf. Fig. 2). On the other hand, the PBSFM more consistently forces bad behavior. This reinforces the analysis presented Section 5.4.3 (see Table 1): A numerical soft-fault model can more effectively force an algorithm to run through bad behavior, while a stochastic bit-flip injection may either force an extreme behavior or have little effect. Numerical soft fault models afford users a higher level of control when investigating fault tolerance.



(a) Early

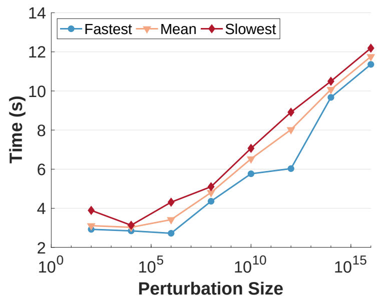


(b) Middle

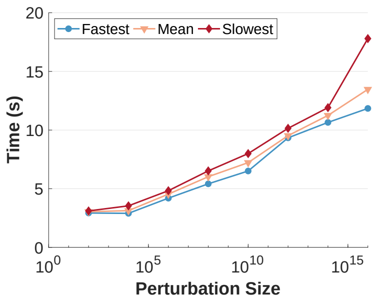


(c) Late

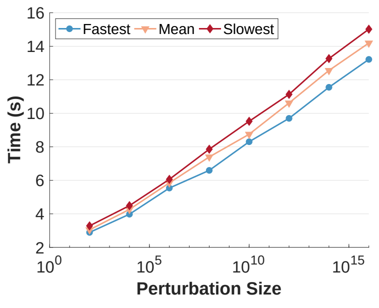
Figure 4: Effect of bit-flip faults in the mantissa bits for the three injection points (Early, Middle, and Late).



(a) Early



(b) Middle



(c) Late

Figure 5: Effect of faults injected using the PBSFM for the three injection points (Early, Middle, and Late).

6.6 Recovery from Soft Faults

For each test, the same desired tolerance was achieved at completion, and the same magnitude of error was inserted. The small, medium, and large values of γ for each respective fault tolerance method arise from trying to strike a balance between computational overhead and ability to provide resilience. Because each of the three methods operates via different mechanisms, the threshold values also differ for each. However, these values were chosen to offer fair comparisons across all three methods for the same level of resilience. Specifically, for the global checkpointing method, the values of γ^g are selected as 5, 10, and 100. The values of γ^p for the partial checkpointing method are 1.01, 1.05, and 1.25 to test for very, moderate, and least sensitive fault detection, respectively. Lastly, the values of γ^a used for the ABFT are 1.05, 1.25, and 2.5. All three sets of values are representative of the spectrum of values that were tested for the corresponding methods. Larger values of γ^g are used for the global checkpointing scheme are needed since a rollback affects far more components and takes a correspondingly larger amount of time. For ABFT, the goal is only to exclude large faults and to let the algorithm progress naturally through smaller errors.

If corrupted values are on the edge of a thread compute region, they may spread to the compute region of a neighboring thread and compromise resiliency. This behavior is more readily observed when using numerical soft fault models, such as PBSFM, since they impact all of the components assigned to the thread, including boundary values. Hence, a trade-off between the sensitivity of the fault-detection and checkpointing overhead is desirable. For example, compare plots for the middle values of their respective γ's for all recovery methods in Fig. 8 with the ones for smaller and larger γ values, respectively. Fault recovery mechanisms in some cases are able to correct PBSFM faults, depending on the circumstances of the run, i.e., if a given thread is able to detect the fault and roll back before adjacent threads copy bad values to their compute regions. Successful and failing recovery outcomes are shown in Fig. 8, where the fastest runs indicate successful recovery and the slowest runs correspond to recovery failure.

Figure 6, compared with Fig. 3, shows that the different recovery techniques employed for this work effectively managed the exponent bit-flip fault injections. Comparing Fig. 7 with Fig. 4 yields little difference—as expected since bit flips in less significant bits are not expected to induce a large negative impact on the solver—while attesting to the overhead of the respective recovery methods. In particular, the largest mean time is $\sim 3.5s$ in Fig. 7 while the largest mean time for partial checkpointing, global checkpointing, and ABFT are $\sim 3.8s$, $\sim 3.2s$, and $\sim 3.4s$ at their corresponding least sensitive γ values, as seen in Figs. 7c, 7f and 7i, respectively.

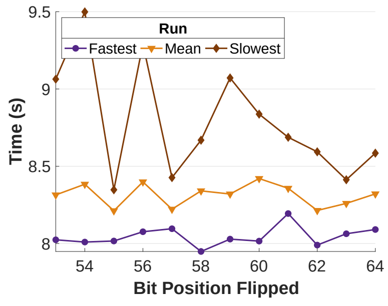
Comparing the different fault recovery methods across all the fault injection methods (Figs. 6 to 8) shows that they all are able to rectify the effects of a fault almost equally well. Overall, the partial checkpointing method tended to be the most robust. The ABFT method performs very well for large faults, such as those induced by the PBSFM. Specifically, Fig. 8 shows that the ABFT is able to keep the overhead of fault tolerance very low even when the size of the perturbation is large. Comparing Figs. 8c, 8f and 8i shows that even for the less strict values of γ, the ABFT technique is able to control the overhead associated with recovery.

In general, a trade-off between the sensitivity of the fault-detection and checkpointing overhead is desirable. For example, in Fig. 8 for each recovery method, compare the middle plots with those for smaller and larger γ values. Both successful and failing recovery outcomes may be observed in Fig. 8, such that the fastest runs indicate successful recovery and the slowest runs correspond to possible recovery failure. The solution timing results for the middle values of γ across all the recovery methods are significantly better than those for less or more sensitive γ's.

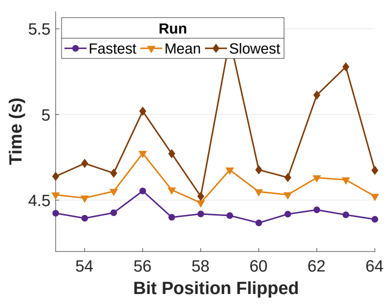
6.7 Scaling to Larger Problems

The experiments presented here focus on the scalability of all three methods to larger total problem sizes with the increase in the total node count while keeping the subdomain size fixed at 200×200 and the number of threads per subdomain at 10. Hence, to increase the problem size, the number of subdomains was increased from 2×2 up to 8×8, resulting in a grid size growing from 400×400 to $1,600 \times 1,600$. Each subdomain was assigned to an MPI process. Each problem was solved with the tolerance $t = 10^{-7}$ to match the relative residual reduction from the earlier experiments.

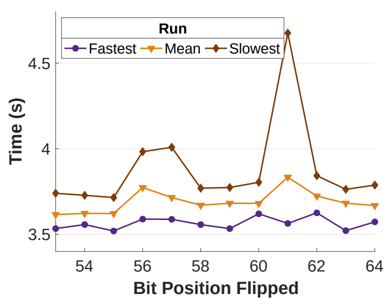
Note a few small differences between the recovery method variants tested here as compared to the ones tested before. Firstly, slightly different values of γ are used for all three methods, which were experimentally determined to be acceptable thresholds on these large-scale runs. The modified γ's remain, as before, representative of the early, middle, and late termination ranges. Secondly, in the ABFT method here, the dynamically scheduled residual computation (see Eq. (33)) is replaced with a regularly scheduled residual computation every 1,000 iterations since the dynamic update formula caused excessive overhead for the largest problem sizes. Investigating improvements to this dynamic update



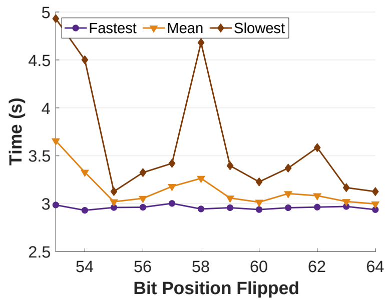
(a) $\gamma^p = 1.01$



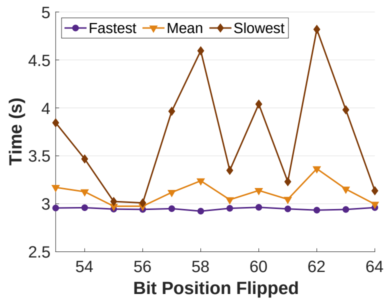
(b) $\gamma^p = 1.05$



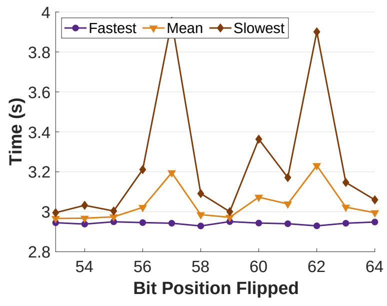
(c) $\gamma^p = 1.25$



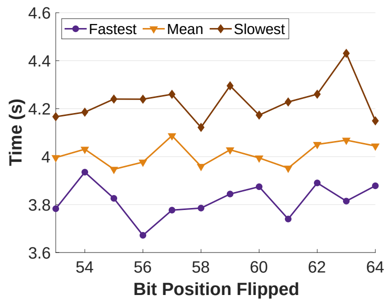
(d) $\gamma^g = 5$



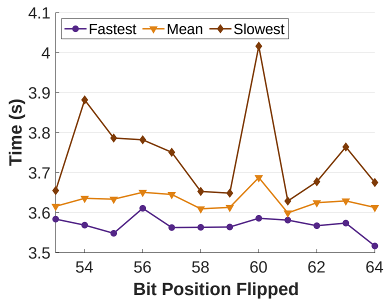
(e) $\gamma^g = 10$



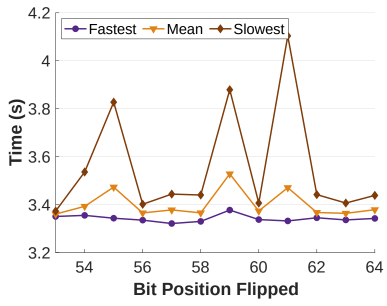
(f) $\gamma^g = 100$



(g) $\gamma^a = 1.05$

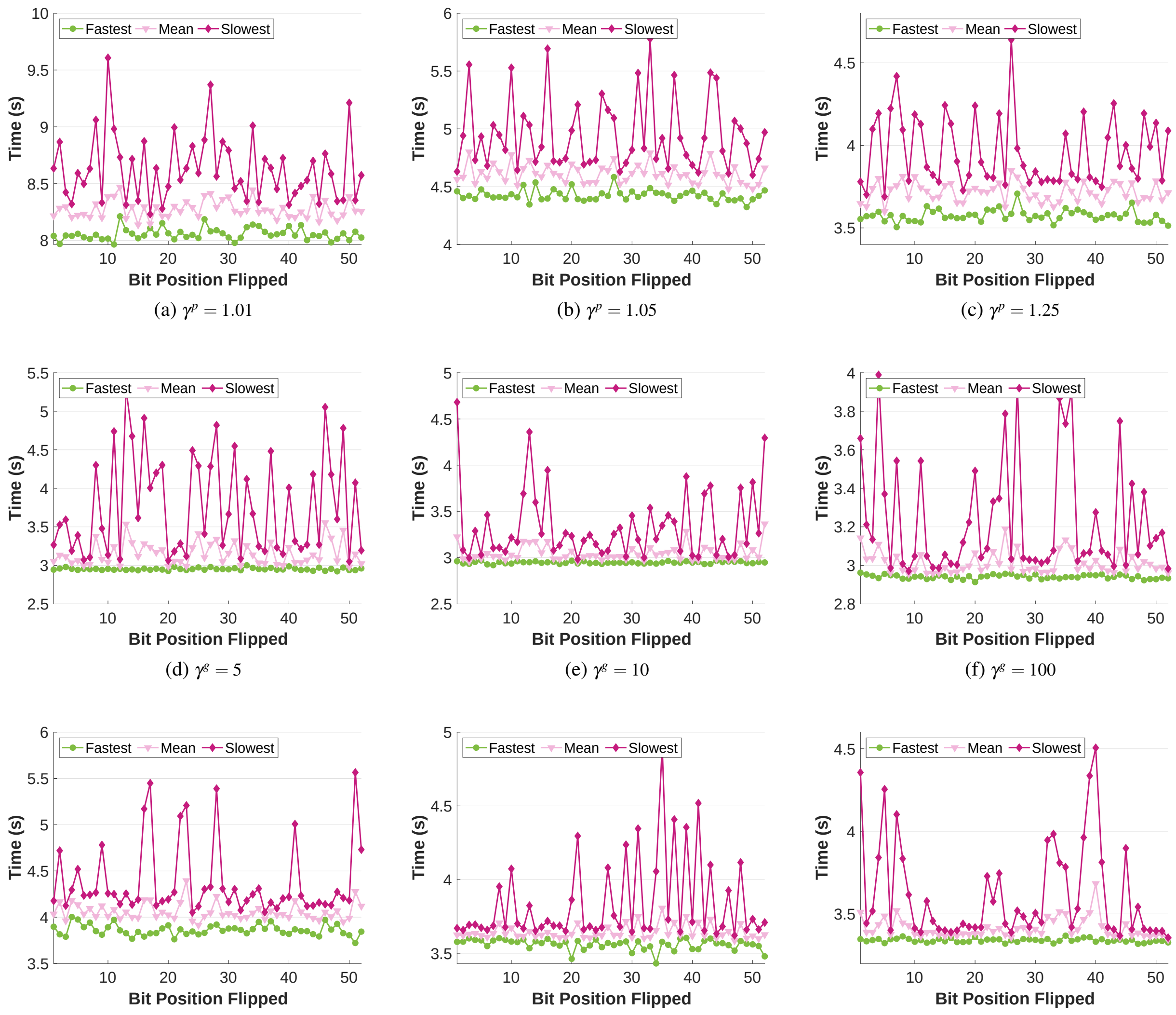


(h) $\gamma^a = 1.25$



(i) $\gamma^a = 2.5$

Figure 6: Effect of recovery methods with bit-flip faults in the exponent and sign bits for partial checkpointing Figs. 6a to 6c, global checkpointing Figs. 6d to 6f, and ABFT Figs. 6g to 6i.

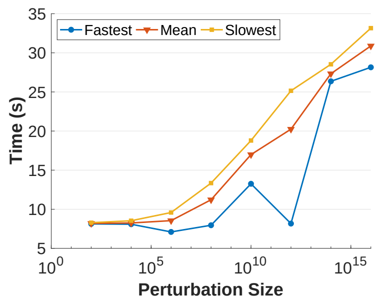


(a) $\gamma^p = 1.01$ (b) $\gamma^p = 1.05$ (c) $\gamma^p = 1.25$

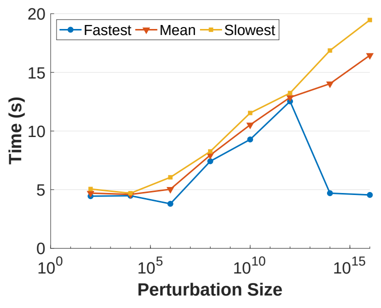
(d) $\gamma^g = 5$ (e) $\gamma^g = 10$ (f) $\gamma^g = 100$

(g) $\gamma^a = 1.05$ (h) $\gamma^a = 1.25$ (i) $\gamma^a = 2.5$

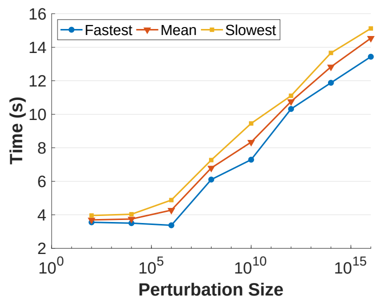
Figure 7: Effect of recovery methods with bit-flip faults in the mantissa bits for partial checkpointing Figs. 7a to 7c, global checkpointing Figs. 7d to 7f, and ABFT Figs. 7g to 7i.



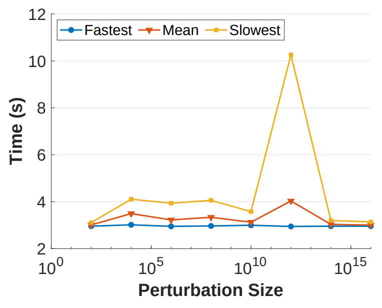
(a) $\gamma^p = 1.01$



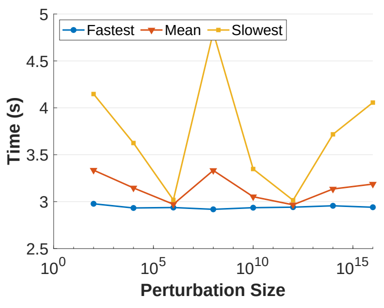
(b) $\gamma^p = 1.05$



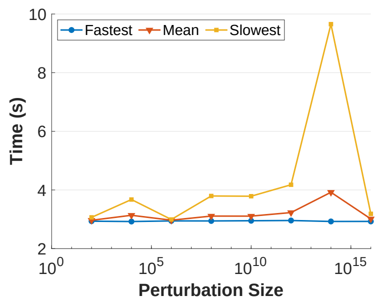
(c) $\gamma^p = 1.25$



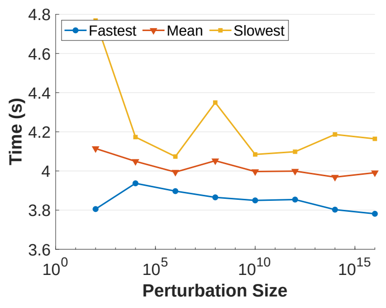
(d) $\gamma^g = 5$



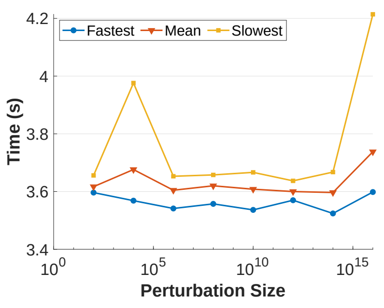
(e) $\gamma^g = 10$



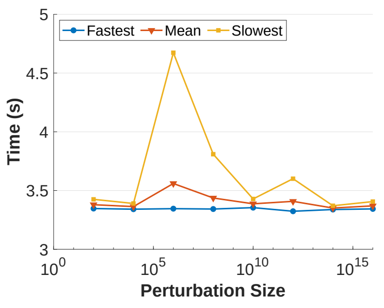
(f) $\gamma^g = 100$



(g) $\gamma^a = 1.05$



(h) $\gamma^a = 1.25$



(i) $\gamma^a = 2.5$

Figure 8: Effect of fault recovery with the PBSFM for partial checkpointing Figs. 8a to 8c, global checkpointing Figs. 8d to 8f, and ABFT Figs. 8g to 8i.

procedure to allow for efficient computation of residual information in the fine-grained ABFT method is left as future work. A series of experiments were conducted using only the PBSFM in order to examine scalability with the focus on faults having a severe impact that is difficult to correct (as noticed in Section 6.6). The range for $\tau_i \in (-10^{10}, 10^{10})$ was chosen to be between the medium and large values used earlier in order to ensure a repeatable severe effect.

Each recovery method was run seven times in each configuration: baseline, with faults plus recovery, and without faults but with inserted recovery mechanism, except for the ABFT method which lacks a distinct baseline configuration because the implementation used always performs an accept/reject check on each new update. Figure 9 shows the data from these experiments. Each point on the curve is an average of the seven trials, and in the `Fault+Recovery` (see the red markers) an `x` is placed to represent the time for each trial, which can differ significantly due to the PBSFM utilization. Note that, in the ABFT experiment, there is often a single slower outlier. Also, the average final computed residual (among all trials) is shown as blue labels next to the points on the curves. The infrequent computation of the termination condition in the ABFT method tends to lead to solutions with smaller final residuals.

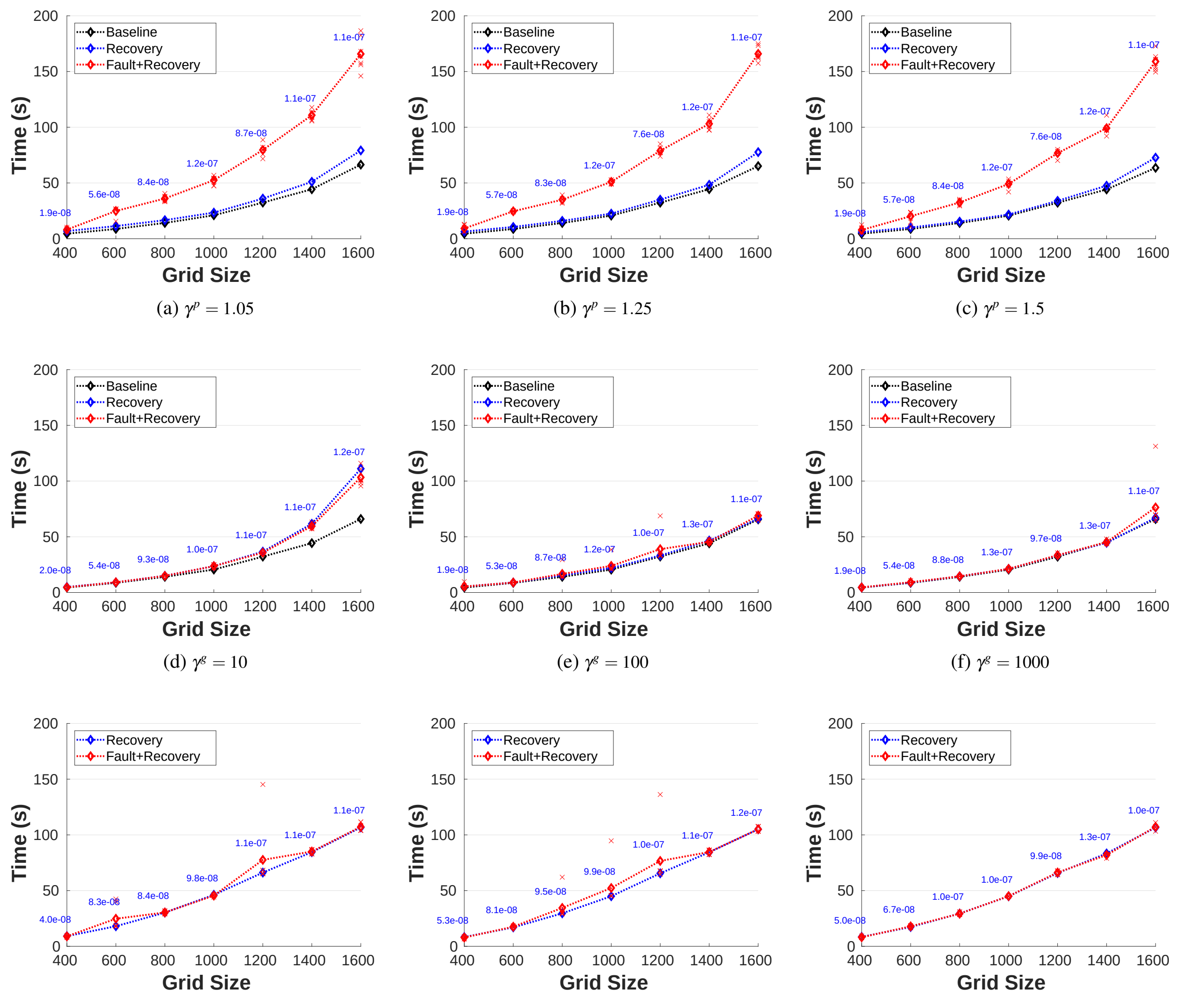
The subplots for `PCP` and `CP` in Fig. 9 (the top two rows of figures, respectively) show the overhead of adding the fault recovery mechanisms (curve `Recovery`) to the code in the case of a fault-free (curve `Baseline`) configuration as well as the overhead incurred when a fault is injected (curve `Fault+Recovery`). `PCP` runs consistently slower with `Fault+Recovery`, compared with `Baseline` and `Recovery`, likely due to the inability of `PCP` to contain faults affecting boundary grid points: as was seen with the smaller-scale experiments (Section 6.6), corrupted values proliferate between subdomains. It is worth noting that the PBSFM fault model (with large random perturbations) used in this subsection represents a pessimistic few of the potential data corruption that will be caused by a fault, and fault detection mechanisms will often detect spurious faults before they can impact all boundary grid points. Injecting solitary bit flips would likely reduce the difference between the `Recovery` and `Fault+Recovery` curves in Figs. 9a to 9c. `PCP`, otherwise, generates predictable, consistent results despite this present shortcoming and also appears to be robust for larger problems.

`CP` shows a strong similarity between `Recovery` and `Fault+Recovery` runs, indicating that it is effective in containing faults affecting border grid points. The overhead from fault detection and correction is seen when comparing with `Baseline` runs. `ABFT` (see Figs. 9g to 9i) scales more linearly with the increase in problem sizes and computational resources compared with either `PCP` or `CP` in the `Fault+Recovery` cases, even though `ABFT` sometimes fails to recover from the perturbation fault quickly as indicated by the `x` outliers in Figs. 9g to 9i. In general, although all three recovery techniques were able to successfully mitigate the impact of a fault in the scalability testing, the increasing overhead for `PCP` and `CP` made them less scalable than `ABFT`. Table 2 shows average iteration numbers for various grid sizes, which correspond to the points on the curves in Fig. 9. Analyzing the residual progression for all runs using `ABFT` show remarkably consistent behavior across all trials, whereas the experiments using `CP` tended to progress based on the value of γ, and the experiments with `PCP` are grouped by run type.

7 Summary and Future Directions

In this paper, fault tolerance strategies for asynchronous linear solvers are investigated both from theoretical and practical viewpoints. In particular, the former provided rationale and background for the proposed fault-tolerance techniques and models. The latter included an experimental study on how the bit-flip and perturbation-based soft fault models affect asynchronous linear solver implementations and on the ability of the proposed techniques to recover. Results were presented for a hybrid MPI-OpenMP parallel implementation of the Jacobi method. They demonstrated that the three recovery techniques considered here may all successfully deal with faults, albeit with varying degrees of overhead, across increasing problem sizes. This overhead may be controlled to some extent by specifying the proposed parameters for each recovery method, as discussed in this paper. Furthermore, the proposed algorithm-based fault tolerance technique appears to be the most scalable of the three by exhibiting a linear increase in the solution time with the growth in the problem size and computational resources used.

In the future, several research directions may prove beneficial: fine-tuning fault-recovery implementations to account better for problem partitioning into subdomains, considering the impact of different parallel implementations, conducting similar experiments on a larger problem set containing a broader suite of problems, and investigating improved methods for terminating asynchronous iterations – specifically fine-grained approaches to fault tolerance for asynchronous methods. It would also be helpful to extend the testing to a larger suite of algorithms, such as randomized Gauss-Seidel [4] and parallel Southwell [36].



(a) $\gamma^p = 1.05$ (b) $\gamma^p = 1.25$ (c) $\gamma^p = 1.5$

(d) $\gamma^g = 10$ (e) $\gamma^g = 100$ (f) $\gamma^g = 1000$

(g) $\gamma^a = 1.25$ (h) $\gamma^a = 2.5$ (i) $\gamma^a = 5.0$

Figure 9: Scalability of recovery methods as the problem size is increased for partial checkpointing Figs. 9a to 9c, global checkpointing Figs. 9d to 9f, and ABFT Figs. 9g to 9i. Individual times are marked with a red x for each `fault+recovery` run. Blue text shows the average final residual for all trials of that size.

Table 2: Average Iteration Numbers to Convergence for Each Recovery Method and Increasing Grid Sizes.

		Grid Size *n*	400	600	800	1000	1200	1400	160
Method	**Gamma**	**Run Type**							
PCP	1.05	baseline	2.312E+05	4.513E+05	7.312E+05	1.088E+06	1.678E+06	2.304E+06	3.457E+
		recovery	3.504E+05	5.816E+05	8.579E+05	1.206E+06	1.857E+06	2.649E+06	4.111E+
		fault+recovery	4.159E+05	1.279E+06	1.858E+06	2.708E+06	4.104E+06	5.746E+06	8.604E+
	1.25	baseline	2.312E+05	4.510E+05	7.304E+05	1.073E+06	1.673E+06	2.309E+06	3.395E+
		recovery	3.263E+05	5.410E+05	8.218E+05	1.160E+06	1.807E+06	2.511E+06	4.040E+
		fault+recovery	4.700E+05	1.268E+06	1.808E+06	2.636E+06	4.078E+06	5.343E+06	8.622E+
	1.5	baseline	2.316E+05	4.515E+05	7.310E+05	1.071E+06	1.673E+06	2.298E+06	3.322E+
		recovery	3.017E+05	5.127E+05	7.879E+05	1.119E+06	1.753E+06	2.468E+06	3.786E+
		fault+recovery	4.011E+05	1.038E+06	1.676E+06	2.539E+06	3.973E+06	5.149E+06	8.266E+
CP	10	baseline	2.316E+05	4.516E+05	7.319E+05	1.073E+06	1.674E+06	2.312E+06	3.439E+
		recovery	2.383E+05	4.504E+05	7.322E+05	1.149E+06	1.702E+06	2.552E+06	4.036E+
		fault+recovery	2.309E+05	4.511E+05	7.313E+05	1.149E+06	1.674E+06	2.505E+06	3.844E+
	100	baseline	2.317E+05	4.506E+05	7.302E+05	1.070E+06	1.677E+06	2.300E+06	3.421E+
		recovery	2.579E+05	4.492E+05	7.752E+05	1.102E+06	1.660E+06	2.336E+06	3.321E+
		fault+recovery	2.580E+05	4.489E+05	8.362E+05	1.199E+06	1.941E+06	2.273E+06	3.432E+
	1000	baseline	2.311E+05	4.506E+05	7.304E+05	1.071E+06	1.672E+06	2.358E+06	3.436E+
		recovery	2.302E+05	4.487E+05	7.282E+05	1.067E+06	1.675E+06	2.250E+06	3.383E+
		fault+recovery	2.306E+05	4.588E+05	7.278E+05	1.066E+06	1.673E+06	2.275E+06	3.853E+
ABFT	1.25	recovery	3.048E+05	5.987E+05	1.008E+06	1.537E+06	2.217E+06	2.882E+06	3.688E+
		fault+recovery	2.967E+05	8.242E+05	1.008E+06	1.514E+06	2.595E+06	2.897E+06	3.701E+
	2.5	recovery	2.760E+05	5.656E+05	9.869E+05	1.497E+06	2.195E+06	2.880E+06	3.631E+
		fault+recovery	2.622E+05	5.817E+05	1.146E+06	1.739E+06	2.562E+06	2.881E+06	3.627E+
	5	recovery	2.815E+05	5.759E+05	9.790E+05	1.495E+06	2.203E+06	2.844E+06	3.686E+
		fault+recovery	2.702E+05	5.930E+05	9.741E+05	1.494E+06	2.215E+06	2.789E+06	3.701E+

Acknowledgments

This work was supported in part by the U.S. Department of Energy (DOE) Office of Advanced Scientific Computing Research under the grant DE-SC-0016564 and the Exascale Computing Project (ECP) through the Ames Laboratory, operated by Iowa State University under contract No. DE-AC00-07CH11358, the Turing High Performance Computing cluster at Old Dominion University, by the National Science Foundation under grant CNS-1828593, and through the In-House Laboratory Independent Research (ILIR) program at the Naval Surface Warfare Center, Dahlgren Division. The authors would also like to thank the reviewers for their thoughtful comments that helped to improve this paper.

References

[1] 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.

[2] Emmanuel Agullo, Siegfried Cools, Emrullah Fatih-Yetkin, Luc Giraud, and Wim Vanroose. On soft errors in the conjugate gradient method: sensitivity and robust numerical detection. *Research Report 9226, Inria Bordeaux Sud-Ouest*, 2018.

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

[4] Haim Avron, Alex Druinsky, and Anshul Gupta. Revisiting asynchronous linear solvers: Provable convergence rate through randomization. *Journal of the ACM (JACM)*, 62(6):1–27, 2015.

[5] Jacques Mohcine Bahi, Sylvain Contassot-Vivier, and Raphael Couturier. *Parallel iterative algorithms: from sequential to grid computing*. Chapman and Hall/CRC, 2007.

[6] Gérard M Baudet. Asynchronous iterative methods for multiprocessors. *Journal of the ACM (JACM)*, 25(2):226–244, 1978.

[7] Dimitri P Bertsekas and John N Tsitsiklis. Convergence rate and termination of asynchronous iterative algorithms. In *Proceedings of the 3rd International Conference on Supercomputing*, pages 461–470. ACM, 1989.

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

[9] Iain Bethune, J Mark Bull, Nicholas J Dingle, and Nicholas J Higham. Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP. *The International Journal of High Performance Computing Applications*, 28(1):97–111, 2014.

[10] 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.

[11] 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.

[12] 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.

[13] Daniel Chazan and Willard Miranker. Chaotic relaxation. *Linear algebra and its applications*, 2(2):199–222, 1969.

[14] 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.

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

[16] Evan Coleman, Erik J Jensen, and Masha Sosonkina. Impacts of three soft-fault models on hybrid parallel asynchronous iterative methods. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 458–465. IEEE, 2018.

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

[18] 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.

[19] James Elliott, Mark Hoemmen, and Frank Mueller. Resilience in numerical methods: a position on fault models and methodologies. *arXiv preprint arXiv:1401.3013*, 2014.

[20] 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.

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

[22] Erik J Jensen, Evan Coleman, and Masha Sosonkina. Predictive modeling of the performance of asynchronous iterative methods. *The Journal of Supercomputing*, 75(8):5084–5105, 2019.

[23] Fabienne Jezequel, Raphaël Couturier, and Christophe Denis. Solving large sparse linear systems in a grid environment: the gremlins code versus the petsc library. *The Journal of Supercomputing*, 59(3):1517–1532, 2012.

[24] Frédéric Magoulès and Guillaume Gbikpi-Benissan. Distributed convergence detection based on global residual error under asynchronous iterations. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):819–829, 2017.

[25] Frédéric Magoules, Daniel B Szyld, and Cédric Venet. Asynchronous optimized Schwarz methods with and without overlap. *Numerische Mathematik*, pages 1–29, 2015.

[26] JC Miellou, P Spiteri, and D El Baz. A new stopping criterion for linear perturbed asynchronous iterations. *Journal of computational and applied mathematics*, 219(2):471–483, 2008.

[27] Jean-Claude Miellou, Pierre Spiteri, and Didier El Baz. Stopping criteria, forward and backward errors for perturbed asynchronous linear fixed point methods in finite precision. *IMA journal of numerical analysis*, 25(3):429–442, 2005.

[28] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[29] Piyush Sao and Richard Vuduc. Self-stabilizing iterative solvers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 1–8, 2013.

[30] Serap Ayșe Savarí and Dimitri P Bertsekas. Finite termination of asynchronous iterative algorithms. *Parallel Computing*, 22(1):39–56, 1996.

[31] 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.

[32] 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.

[33] 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.

[34] Pierre Spiteri, Jean-Claude Miellou, and Didier El Baz. Perturbation of parallel asynchronous linear iterations by floating point errors. *Electronic Transactions on Numerical Analysis*, 13:38–55, 2002.

[35] 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.

[36] Jordi Wolfson-Pou and Edmond Chow. Distributed southwell: an iterative method with low communication costs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2017.