Impacts of Three Soft-Fault Models on Hybrid Parallel Asynchronous Iterative Methods

Evan Coleman*†, Erik J. Jensen† and Masha Sosonkina†
*Naval Surface Warfare Center, Dahlgren Division, Dahlgren, VA
†Modeling, Simulation and Visualization Engineering Department, Old Dominion University, Norfolk, VA
Email: ecole028@odu.edu, ejens005@odu.edu, msosonki@odu.edu

***Abstract*—This study seeks to understand the soft error vulnerability of asynchronous iterative methods, with a focus on stationary iterative solvers such as Jacobi. The implementations make use of hybrid parallelism where the computational work is distributed over multiple nodes using MPI and parallelized on each node using OpenMP. A series of experiments is conducted to measure the impact of an undetected soft fault on an asynchronous iterative method, and to compare and contrast several techniques for simulating the occurrence of a fault and then recovering from the effects of the faults. The data shows that the two numerical soft-fault models tested here more consistently than a "bit-flip" model produce bad enough behavior to test a variety of recovery strategies, such as those based on partial checkpointing.**

***Index Terms*—Fault modeling, fault tolerance, hybrid parallelism, asynchronous iterative methods**

I. Introduction

The prevalence of faults is expected to increase as platforms continue to grow [1]–[5], which will cause the mean time between failures (MTBF) to continue to decrease. This calls for the design of fault-tolerant computational algorithms that are robust in the face of these failures. Development of such algorithms has become one of many priorities on the road towards exascale. At a high level, computing faults can be divided into two distinct categories: hard faults and soft faults [6]–[8]. The key characteristic of all hard faults is that they cause program interruption, which makes them difficult to deal with from an algorithmic standpoint. Conversely, soft faults do not immediately cause program interruption, although such an interruption may occur as a result of their impact. Transient soft faults are typically caused by solitary bit flips, which may be caused by different issues, such as radiation, hardware malfunction, or data-cache set incorrectly. Whether researchers choose to model faults using bit flips or adopt a more numerical approach, much of the previous work on the impact of silent data corruption (SDC) has to do with the modeling of transient errors.

The occurrence of soft faults has been commonly modeled by the injection of bit flips into the data structures of the algorithm [9], [10]. Recent research efforts (see, e.g., [8], [11]–[16]) have focused on modeling the impact of soft faults with a numerical approach that quantifies the potential impact by generating an appropriately sized fault using a more numerically-based scheme. This paper aims to establish the impact a soft fault can have on asynchronous iterative methods (specifically, fixed point methods capable of solving linear systems). It analyses a series of experiments featuring both transient bit flips and generalized numerical soft-fault models, which are based on corrupting data in certain data structures. Additionally, experiments with partial checkpointing to mitigate the impact of the soft faults are presented and analyzed.

The remainder of the paper is organized as follows: in Section II an overview of related studies is given, in Section III a review of asynchronous iterative methods—with an emphasis on the algorithm under study here—is provided. Section IV details on the hybrid parallel, asynchronous implementation used, Sections V and VI discusses various ways to model the impact of a soft fault and the technique used here for the recovery. Section VII provides numerical experiments and Section VIII concludes.

II. Related Work

The efficacy of asynchronous methods, especially for grid systems, is proposed based on computational and communication strategies [17], [18]. Work has also been performed to demonstrate the superior performance of asynchronous methods for solving large sparse linear fixed-point problems [19]. Comparisons of parallel implementations, using both MPI and OpenMP, have been conducted [20], and simulations have been developed [21].

Examples of work examining the performance of asynchronous iterative methods include an in-depth analysis from the perspective of utilizing a system with a co-processor [22], [23], as well as performance analysis of asynchronous methods [24]–[26]. In particular, both [24], [26] focus on low level analysis of the asynchronous Jacobi method, similar to the example problem presented here. Work exploring possibilities for reducing the communication costs inherent in a distributed asynchronous solver has also been performed [27].

Fault tolerance has been studied previously for specific asynchronous iterative methods. A fine-grained scheme for fault tolerance for stationary iterative solvers was proposed and refined [28], [29]. Similarly, analysis of the soft fault resilience of fine-grained incomplete factorizations has also been performed [13], [14], [30]. Numerical soft fault models have been previously compared [12], [31] for traditional Krylov subspace solvers. A general position paper on the efficacy of treating soft faults has numerical corruption has been provided [32]. The impact of bit flips specifically has been analyzed numerically [10] and examined in the case of synchronous iterative solvers (with a focus on Krylov subspace solvers) [9].

III. Review of Asynchronous Iterative Methods

When performing computation asynchronously, each component of the problem should be updated such that no information from the other computations is needed while the update is being made, allowing for each processor to act independently from all others. Depending on the size of both the problem, each processor may update a single element, or a block of components. The model that is shown here to provide a basis for asynchronous computation comes mainly from [33].

The model adopted searches for fixed points of functions. To start, consider a function, $G : D \to D$. A fixed point iteration is performed such that,

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

and a fixed point is declared if $x^{k+1} \approx x^k$. Given a finite number of processors $P_1, P_2, \ldots, P_p$ each assigned to a block B of components $B_1, B_2, \ldots, B_m$, the computational model can be stated in Algorithm 1. If each processors (P_l) waits

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

Algorithm 1: General Computational Model

for the other processors to finish each update, then the model describes a parallel synchronous form of computation. If no order is established for the processors, then the computation is asynchronous.

Set a global iteration counter k that increases each time a processor reads x from memory. At the end of an update by processor p, the components associated with the block B_p will be updated. This results in a vector, $x = (x_1^{s_1(k)}, x_2^{s_2(k)}, \ldots, x_m^{s_m(k)})$ where $s_l(k)$ indicates how many times component l has been updated. Lastly, define a set of indices, I^k, that contains the components that were updated on the k^{th} iteration. Given these definitions, the three following conditions (along with the model presented in Algorithm 1) provide a framework for asynchronous computation.

Definition 1. If the following three conditions hold:

1) $s_i(k) \leq k - 1$, *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 $x^0 \in D$, the iterative update process defined by,

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

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

This basic computational model (i.e. the combination of Algorithm 1 and Definition 1 together) allows for many different results on fine-grained iterative methods.

A. Asynchronous Jacobi

Partial differential equations mathematically model systems in which continuous variables, such as temperature or pressure, change with respect to two or more independent variables, such as time, length, or angle [34]. Laplace's equation in two dimensions,

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

is fundamental for modeling equilibrium and steady state problems, such as incompressible fluid flow or heat transfer, and maintains that the rate at which a fluid enters a domain is equal to the rate at which a fluid leaves the domain. In practice, the partial differential equation is not used directly, but is discretized such that a finite difference operator computes difference quotients over a discretized domain. For example, the two-dimensional discrete Laplace operator,

$$\begin{aligned}\left(\nabla^2 f\right)(x, y) = f(x-1, y) + f(x+1, y) + f(x, y-1) \\ + f(x, y+1) - 4f(x, y),\end{aligned} \tag{3}$$

approximates the two-dimensional continuous Laplacian using a five-point stencil [35]. A special case of the Jacobi algorithm,

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

may be applied to solve a two-dimensional sparse linear system of equations [36]. This work uses the Jacobi algorithm to solve a two-dimensional finite-difference discretization of the Laplacian with Dirichlet boundary conditions. This can be viewed as a heat diffusion problem, in which a plate is held to specific temperatures along the boundary [37]. Pseudocode for this algorithm is provided below in Algorithm 2. Note that each processor, P_l, may not be available to compute updates at the same time. This lack of determinism in the update order (i.e. the amount of time it will take a processor to perform the Jacobi relaxation for the components that are assigned to it) leads to the asynchronous nature of the algorithm.

```
Input: $a_{ij} \in A$, initial guess for $x^{(0)}$, a number of
       processing elements $p$, an input random
       number distribution
Output: Solution vector $x$
Assign elements $x_i \in x$ to each processing element
for $t = 1, 2, \ldots$ until convergence do
    for each processor $P_l$ do
        if $P_l$ is ready to compute updates then
            for each element $x_i \in x$ assigned to $P_l$ do
                $x_i = \frac{-1}{a_{ii}} \left[ \sum_{j \neq i} a_{ij} x_j - b_i \right]$
            end
        end
    end
    Calculate the residual, $b - Ax^{(t)}$
    Check termination conditions
end
```

Algorithm 2: Asynchronous Jacobi

IV. Parallel Implementation

The asynchronous Jacobi implementation used in this study 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.

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

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

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

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

V. Modeling the Impact of a Soft Fault

In the majority of studies, the occurrence of an undetected soft fault is overwhelmingly treated as a bit flip (see, e.g., [9]). 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 [32]. 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 [32], the numerical fault models used here are inspired by the idea of modeling an undetected soft fault as data corruption. That is, instead of trying to model the exact impact of a fault on future large scale HPC machines, faults are treated as corrupted data where the size of the corruption can be controlled in an effort to produce consistent worst case behavior and help with the development of fault tolerant algorithms. 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.

a) 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. While it is important for future computing platforms not to become too dependent on the precise mechanism that is used to model the instantiation of a fault, since bit-flips are currently the most likely form of a soft fault to affect HPC hardware it is important to include analysis that responds to the effects of having a bit-flip occur during the run.

b) Perturbation-Based Soft Fault Model (PBSFM): This approach models faults as perturbations to the faulty subdomain, and have been already used in several other recent studies (see, e.g., [11], [13]–[15], [30]) along with similar approaches. In PBSFM, a small random perturbation τ that is sampled from a uniform distribution of a given size is injected transiently into each component representing a value of the targeted data structure. For example, if the targeted data structure is a vector x and the maximum size of the perturbation-based fault is ϵ, then proceed as follows: (1) generate a random number $\tau_i \in (-\epsilon, \epsilon)$, (2) set $\hat{x}_i = x_i + \tau_i$ for all values of i. The resultant vector $\hat{x}$ is, thus, perturbed away from the original vector x.

c) Shuffle-Based Soft Fault Model (SBSFM): This approach models faults primarily as a shuffling of the elements inside of the faulty subdomain. This approach was originally detailed in [16] and simulates the occurrence of a soft fault by a permutation of the components inside of the subdomain in which a fault was injected, a scaling of the data inside of the subdomain in which a fault was injected, or a combination of these two effects. The analysis that was performed in [16], [38] details the impact of the SBSFM model in the case where it is modeling transient soft faults with various scaling values for traditional synchronous iterative methods; specifically Krylov subspace methods such as GMRES and CG. The focus in this study is on the effect of various fault injection models on *asynchronous* iterative methods. The impact of this fault model relative to the impact of a single bit flip is studied in [16]. bit flip occurs, the SBSFM will perform in a similar way to the worst case scenario induced by BFSFM.

A. Analysis of Fault Injection Utilities

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

For this iterate of x from the matrix representation of the problem, the first quarter of the elements are isolated (i.e., to correspond to one of the four subdomains that is assigned to a single MPI process as described in Section IV), and then the first tenth of these elements are isolated (corresponding to the components assigned to the first OpenMP thread) and the effect of fault-models is examined for this localized subdomain. The methods that are compared are as follows:

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

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

TABLE I
COMPARISON OF DIFFERENT FAULT INJECTION TECHNIQUES

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

VI. Techniques for Recovery from Soft Faults

Formulating efficient techniques that allow an algorithm to recover from soft faults is an important area of research as HPC platforms progress towards exascale. The prevailing wisdom is that globally checkpointing all processors will not be feasible computationally for large-scale problems due to the immense costs of reading/writing data and globally communicating [2], [3]. While this is true for any iterative method, for the fine-grained asynchronous iterative algorithms the use of global (or even large-subgroup) communications is prohibitive because synchronization used in such communications goes against the very nature of these algorithms relying on a great number of light-weight thread or process computations. In this paper, a partial checkpointing method is used that avoids many of the communication related pitfalls inherent in the simple global checkpointing algorithm. This method is similar to the partial checkpointing method used for the FGPILU algorithm in [13], [14].

Progress of the Jacobi algorithm is often judged by the progression of the residual $r = b - Ax^{(k)}$. However, checkpointing based on the progression of the residual after it is recovered from all components of $x^{(k)}$ necessitates communication among all the OpenMP threads as well as all the MPI processes. The partial checkpointing checkpoints only based on the local portion of the residual, denoted r_l. As the asynchronous computation progresses, each thread writes *periodically* the current value of the components x_l for which it is responsible to checkpoint. The periodicity is treated as a parameter, which is studied in Section VII. After the thread updates its components in the k^{th} iteration, $x_l^{(k)}$, it checks the current local residual to see if a fault has occurred:

$$r_l^{(k)} > \gamma \cdot r_l^{(k+1)} \,, \tag{5}$$

where $\gamma > 1$ is the checkpoint threshold, explored in Section VII, $r_l^{(k)}$ and $r_l^{(k+1)}$ are local residuals for the iteration k and $k+1$, respectively. γ in this study ranged from 1.01 to 1.25.

VII. 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

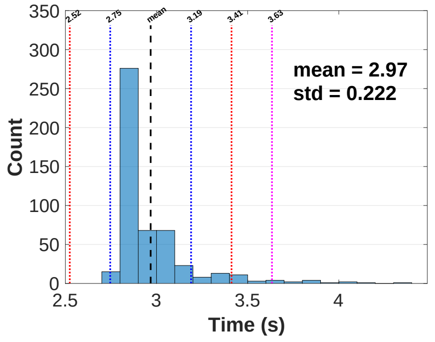

Fig. 1. Distribution of run times in a fault-free environment.

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 were collected on sockets consisting of 10 Intel Xeon E5-2670 v2 2.50 Ghz cores.

Before delving into the results regarding the impact and recovery of soft faults on the hybrid parallel iterative solver used here, a set of baseline runs is presented. The problem described in Section IV is solved 500 times and a histogram showing the distribution of total run times, and mean and standard deviation, is presented in Fig. 1. Some variation in run time is observed, but this is not unexpected for an asynchronous solver. A wide variation in iterations until convergence is seen in [26], and [39] shows increased run time variation for asynchronous solvers. Here, the run time for +3 standard deviations is 1.31 times the minimum run time. Almost 98% of runs are less than +3 standard deviations.

A. *Impact of Soft Faults*

The following model parameter values were used

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

Based on the mean runtime of 2.97s shown in Fig. 1, three different fault injection times were used as follows: *early*, equal to $0.1s$, *middle*, equal to $1.2s$, and *late* of $2.5s$.

Figures 2 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. 2 and Fig. 3, respectively. Each experiment was replicated seven times on Turing. The plots show the results from the fastest, slowest, and average of these seven runs. In all experiments, the solver converged to a correct solution, within a tolerance of $1e^{-4}$.

Figure 2 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. Across all the faults models, faults injected early tend to have more of an effect on the total time for the solve to complete. Note also that, while bit-flip faults in the exponent and sign bits can have a catastrophic effect, bit-flips occurring in the mantissa have very little impact on the performance of the solver (cf. Fig. 1). PBSFM and SBSFM force more consistently bad behavior. This reinforces experimental outcome (see Table I in Section V): a numerical soft-fault model can more effectively force an algorithm to run through bad behavior, while a stochastic bit-flip injection may force an extreme behavior, but may little effect. Numerical soft fault models afford users a higher level of control.

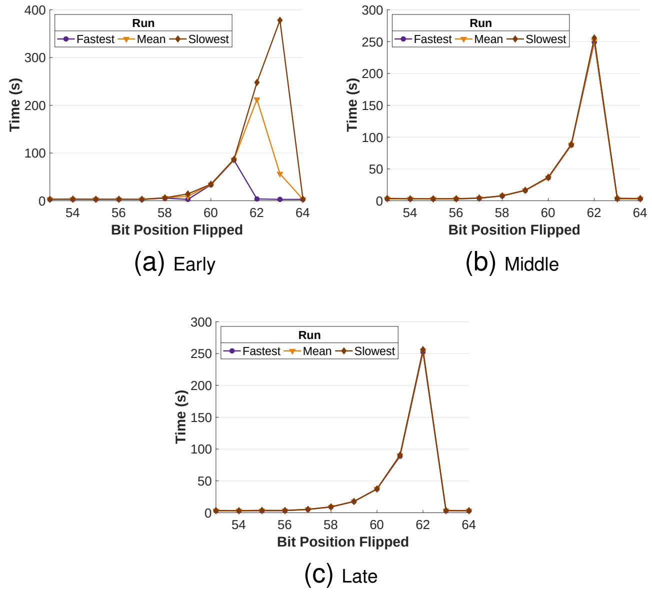

Fig. 2. Effect of bit-flip faults in the exponent and sign bits.

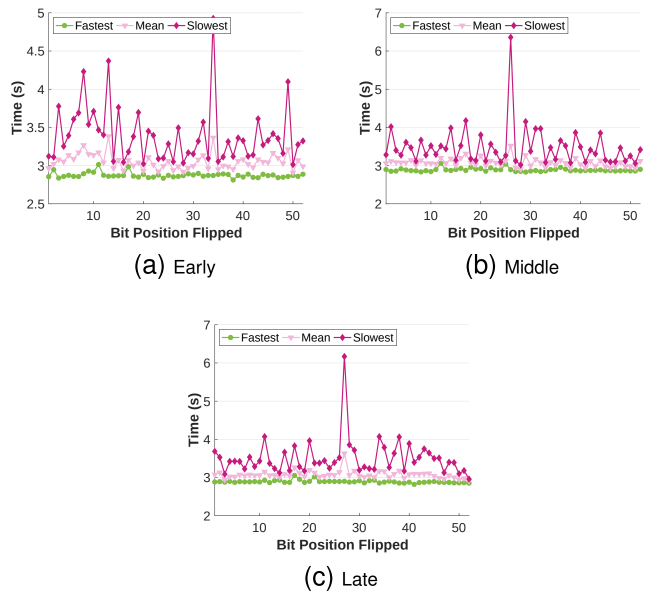

Fig. 3. Effect of bit-flip faults in the mantissa bits.

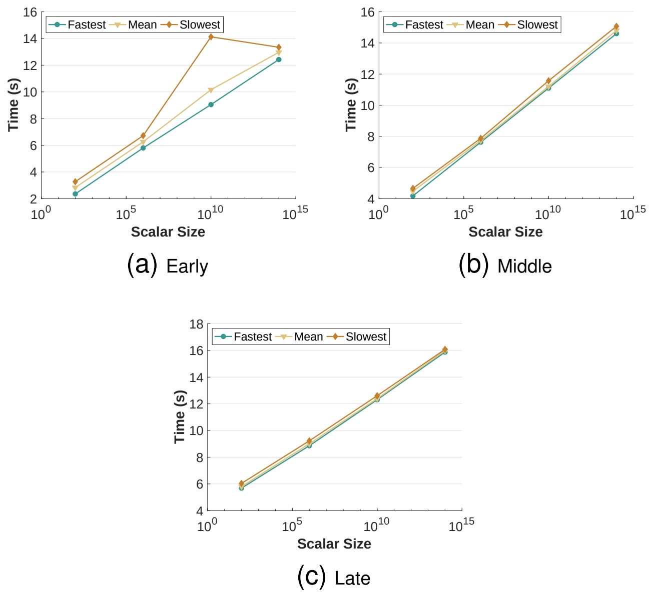

Fig. 4. Effect of faults injected using the SBSFM.

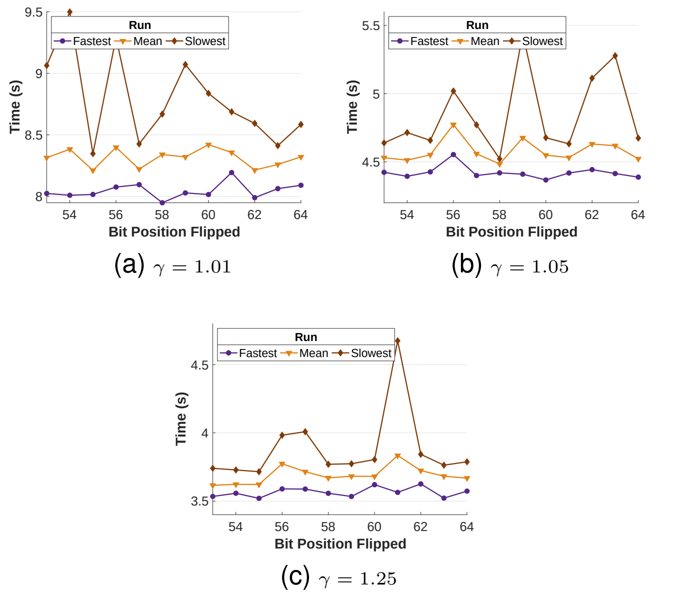

Fig. 6. Effect of recovery with bit-flip faults in the exponent and sign bits.

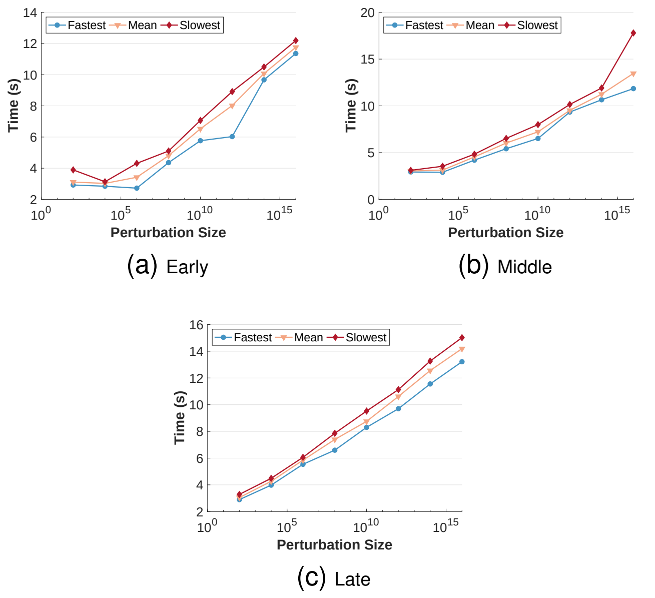

Fig. 5. Effect of faults injected using the PBSFM.

B. Recovery from Soft Faults

Consider the partial checkpointing scheme detailed in Section VI. The fault is injected near the *middle* time, at 1.4s, in each run. The values of γ in Eq. (5) are 1.01, 1.05 and 1.25 to test for very, moderate, and least sensitive fault detection, respectively.

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

Figure 6, compared with Fig. 2, shows that the checkpointing and rollback technique employed for this work effectively managed the exponent bit-flip fault injections. Comparing Fig. 7 with Fig. 3 yields little difference—as expected—while attesting to only a moderate overhead of checkpointing. In particular, the largest mean value in Fig. 3(b) was $\sim$3.5s while the largest mean value in Fig. 7(c), i.e., for the least sensitive fault detection, at $\gamma = 1.25$, was $\sim$3.8s

If corrupted values are on the edge of a thread compute region, they may spread to the compute region of a neighboring thread and compromise resiliency. This behavior is more readily observed when using numerical soft fault models, such as PBSFM and SBSFM, since they impact all of the components assigned to the thread, including boundary values. Hence, a trade-off between the sensitivity of the fault-detection and checkpointing overhead is desirable. For example, compare plots for $\gamma = 1.05$ in Figs. 8 and 9 with the ones for smaller and larger γ values, respectively. Note also that the recovery with SBSFM in Fig. 8 exhibits consistently increasing difference between the slowest and the fastest runs with the increase in the pertubation size. Fault recovery mechanisms in some cases are able to correct PBSFM and SBSFM faults, depending on the circumstances of the run, i.e. if the fault thread is able to detect the fault and roll back before adjacent threads copy bad values to their compute regions. Successful and failing recovery outcomes are shown in Fig. 8, where the fastest runs indicate successful recovery and the slowest runs correspond to recovery failure. The implementation tested in this work corrected SBSFM faults at a higher rate than PBSFM faults.

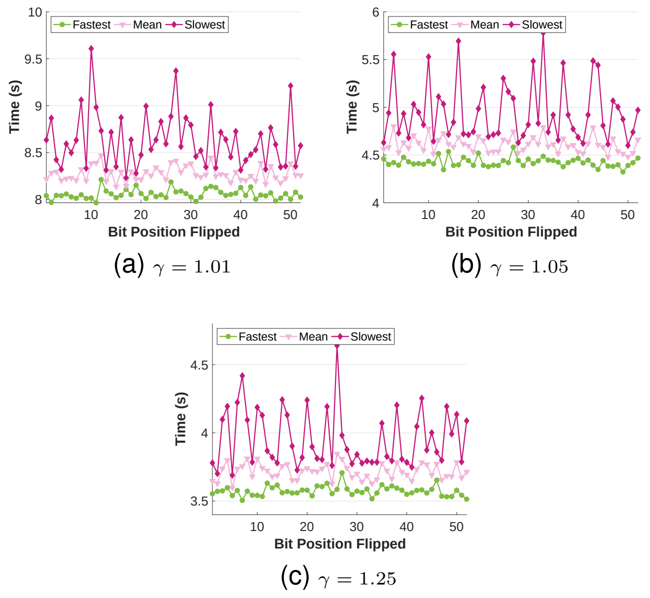

Fig. 7. Effect of recovery with bit-flip faults in the mantissa bits.

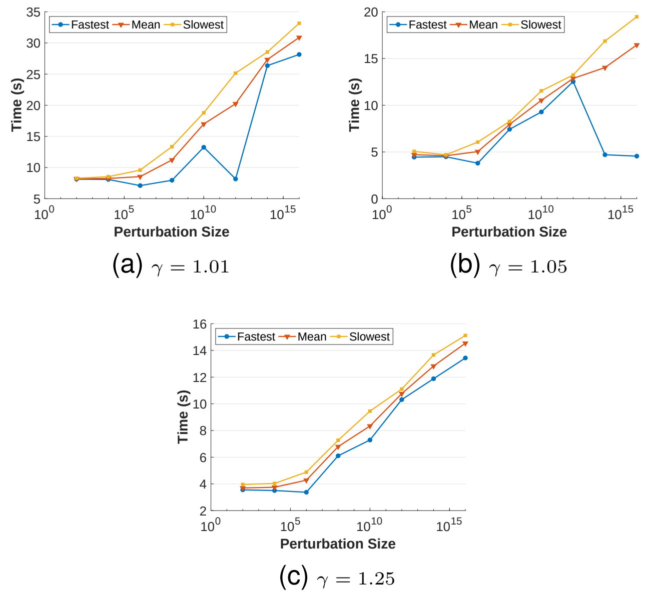

Fig. 9. Effect of fault recovery with the PBSFM.

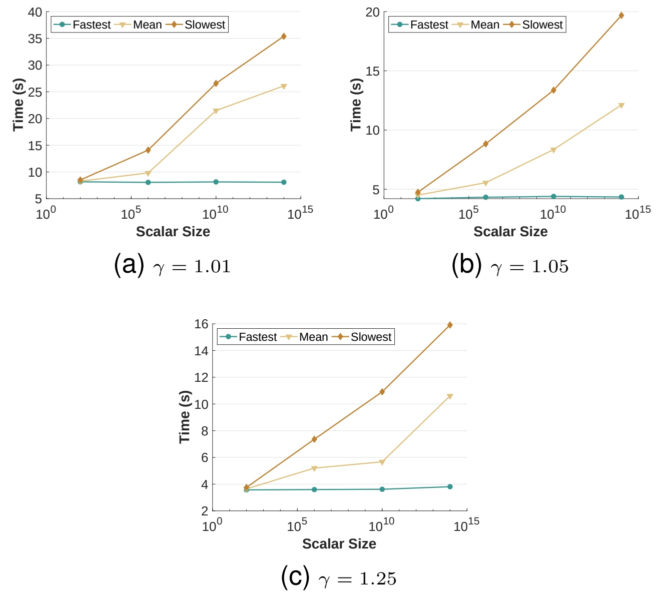

Fig. 8. Effect of fault recovery with the SBSFM.

VIII. Summary and Future Directions

This study presented analysis of two numerical soft fault models for the development of fault tolerant algorithms. Results were presented for asynchronous iterative methods; specifically those implemented in a hybrid parallel fashion. The results show that the use of numerical soft fault models may be useful for the development of fault tolerant algorithms since the average impact induced by the numerical soft fault models is large enough to cause detrimental effect to the execution of the iterative algorithm.

In the future, it would be helpful to conduct similar experiments on a larger problem set. The finite difference discretization of the Laplacian is a common test problem since it is found inside of many complex problems, and can be indicative of performance for sparse symmetric positive-definite problems. However, examining the performance of the asynchronous iterative methods on a broader suite of problems may allow the analysis to be extended. It would also be helpful to extend the testing to a larger suite of algorithms. Examples include, randomized Gauss-Seidel [23] and parallel Southwell [27], [40]. While a global checkpointing scheme (i.e., across all processes) will most likely be prohibitively slow, it may prove useful as a comparison. Checkpointing across an entire process would provide another point of comparison, as would including other modern techniques that avoid the calculation of the residual altogether [28], [29]. Given this larger set of data, more extensive analysis across the methods may be possible. For example, examining the recovery rate and induced overhead allows one to make informed resilience decisions when developing resilient asynchronous iterative algorithms for next-generation extreme-scale computing.

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 and the Exascale Computing Project (ECP) 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, the Turing High Performance Computing cluster at Old Dominion University, and through the ILIR/IAR program at the Naval Surface Warfare Center, Dahlgren Division. The authors also wish to acknowledge helpful comments from the reviewers that increased the quality of this paper.

References

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

[2] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *The International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

[3] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing frontiers and innovations*, vol. 1, no. 1, 2014.

[4] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina *et al.*, "Ascac subcommittee report: The opportunities and challenges of exascale computing," Technical report, United States Department of Energy, Fall, Tech. Rep., 2010.

[5] ——, "The opportunities and challenges of exascale computing–summary report of the advanced scientific computing advisory committee (ascac) subcommittee," *US Department of Energy Office of Science*, 2010.

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

[7] P. G. Bridges, K. B. Ferreira, M. A. Heroux, and M. Hoemmen, "Fault-tolerant linear solvers via selective reliability," *arXiv preprint arXiv:1206.1390*, 2012.

[8] J. Elliott, M. Hoemmen, and F. Mueller, "Evaluating the impact of sdc on the gmres iterative solver," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 1193–1202.

[9] G. Bronevetsky and 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.

[10] J. J. Elliott, F. Mueller, M. K. Stoyanov, and C. G. Webster, "Quantifying the impact of single bit flips on floating point arithmetic," Oak Ridge National Laboratory (ORNL), Tech. Rep., 2013.

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

[12] E. Coleman, A. Jamal, M. Baboulin, A. Khabou, and M. Sosonkina, "A Comparison of Soft-Fault Error Models in the Parallel Preconditioned Flexible GMRES," in *Proceedings of the 12th International Conference on Parallel Processing and Applied Mathematics*. ACM, 2017.

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

[14] E. Coleman and M. Sosonkina, "Self-Stabilizing Fine-Grained Parallel Incomplete LU Factorization," *Sustainable Computing: Informatics and Systems*, 2018.

[15] M. Stoyanov and C. Webster, "Numerical analysis of fixed point algorithms in the presence of hardware faults," *SIAM Journal on Scientific Computing*, vol. 37, no. 5, pp. C532–C553, 2015.

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

[17] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, "Performance comparison of parallel programming environments for implementing aiac algorithms," *The Journal of Supercomputing*, vol. 35, no. 3, pp. 227–244, 2006.

[18] ——, "Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational grid," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE, 2003, pp. 9–pp.

[19] D. V. De Jager and J. T. Bradley, "Extracting state-based performance metrics using asynchronous iterative techniques," *Performance Evaluation*, vol. 67, no. 12, pp. 1353–1372, 2010.

[20] K. Voronin, "A numerical study of an mpi/openmp implementation based on asynchronous threads for a three-dimensional splitting scheme in heat transfer problems," *Journal of Applied and Industrial Mathematics*, vol. 8, no. 3, pp. 436–443, 2014.

[21] E. Coleman, E. Jensen, and M. Sosonkina, "Simulation Framework for Asynchronous Iterative Methods," *Journal of Simulation Engineering*, 2018.

[22] H. Anzt, "Asynchronous and multiprecision linear solvers-scalable and fault-tolerant numerics for energy efficient high performance computing," Ph.D. dissertation, Karlsruhe, Karlsruher Institut für Technologie (KIT), Diss., 2012, 2012.

[23] H. Avron, A. Druinsky, and A. Gupta, "Revisiting asynchronous linear solvers: Provable convergence rate through randomization," *Journal of the ACM (JACM)*, vol. 62, no. 6, p. 51, 2015.

[24] I. Bethune, J. M. Bull, N. J. Dingle, and N. J. Higham, "Investigating the Performance of Asynchronous Jacobi's Method for Solving Systems of Linear Equations," *To appear in International Journal of High Performance Computing Applications*, 2011.

[25] J. Hook and N. Dingle, "Performance analysis of asynchronous parallel jacobi," *Numerical Algorithms*, pp. 1–36, 2013.

[26] I. Bethune, J. M. Bull, N. J. Dingle, and N. J. Higham, "Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP," *The International Journal of High Performance Computing Applications*, vol. 28, no. 1, pp. 97–111, 2014.

[27] J. Wolfson-Pou and E. Chow, "Reducing communication in distributed asynchronous iterative methods," *Procedia Computer Science*, vol. 80, pp. 1906–1916, 2016.

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

[29] ——, "Fine-grained bit-flip protection for relaxation methods," *Journal of Computational Science*, 2016.

[30] E. Coleman and M. Sosonkina, "Convergence and Resilience of the of the Fine-Grained Parallel Incomplete LU Factorization for Non-Symmetric Problems," in *Proceedings of the 2018 Spring Simulation Multiconference*. Society for Computer Simulation International, 2018.

[31] ——, "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.

[32] J. Elliott, M. Hoemmen, and F. Mueller, "Resilience in numerical methods: a position on fault models and methodologies," *arXiv preprint arXiv:1401.3013*, 2014.

[33] A. Frommer and D. B. Szyld, "On asynchronous iterations," *Journal of computational and applied mathematics*, vol. 123, no. 1, pp. 201–216, 2000.

[34] G. D. Smith, *Numerical solution of partial differential equations: finite difference methods*. Oxford university press, 1985.

[35] T. Lindeberg, "Scale-space for discrete signals," *IEEE transactions on pattern analysis and machine intelligence*, vol. 12, no. 3, pp. 234–254, 1990.

[36] J. C. Strikwerda, *Finite difference schemes and partial differential equations*. Siam, 2004, vol. 88.

[37] S. C. Chapra and R. P. Canale, *Numerical methods for engineers*. McGraw-Hill New York, 1998, vol. 2.

[38] J. Elliott, M. Hoemmen, and F. Mueller, "Tolerating Silent Data Corruption in Opaque Preconditioners," *arXiv preprint arXiv:1404.5552*, 2014.

[39] F. Jezequel, R. Couturier, and C. Denis, "Solving large sparse linear systems in a grid environment: the gremlins code versus the petsc library," *The Journal of Supercomputing*, vol. 59, no. 3, pp. 1517–1532, 2012.

[40] J. Wolfson-Pou and E. Chow, "Distributed southwell: an iterative method with low communication costs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 48.