Simulation Framework for Asynchronous Iterative Methods

ACM Classifications: *B.2.3 (Fault-Tolerance, Reliability, Testing, and [Arithmetic and Logic Structures] || SD G.1.3 Linear systems (direct and iterative methods) || SD J.2 Mathematics and statistics [Computer Applications]*
Keywords: *Asynchronous iterative methods || fault tolerance || asynchronous simulation*

Abstract

As high-performance computing (HPC) platforms progress towards exascale, computational methods must be revamped to successfully leverage them. In particular, (1) asynchronous methods become of great importance because synchronization becomes prohibitively expensive and (2) resilience of computations must be achieved, e.g., using checkpointing selectively which may otherwise become prohibitively expensive due to the sheer scale of the computing environment. In this work, a simulation framework is proposed and tested to examine the potential benefit of asynchronous iteration for various HPC accelerator architectures (which typically admit different granularities of computations). Additionally, an example of a case study using the simulation framework is presented to examine the efficacy of different checkpointing schemes for asynchronous relaxation methods.

1 Introduction

Asynchronous iterative methods are increasing in popularity recently due to their ability to be parallelized naturally on modern co-processors such as GPUs and Intel Xeon Phis. Many examples of recent work using fine-grained parallel methods are available (see [ADQO16, Anz12, CAD15, CP15, ACD15] and many others in Section 2). A specific area of interest is on techniques that utilize fixed point iteration, i.e. those equations of the form,

$$x = G(x) \tag{1}$$

for some vector $x \in D$ and some map $G : D \to D$. These techniques are well suited for fine-grained computation can be executed in either a synchronous or asynchronous manner which helps tolerate latency in high-performance computing (HPC) environments. Looking forward to the future of HPC, it is important to prioritize the develop of algorithms that are resilient to faults since on future platforms, the rate at which faults occur is expected to increase dramatically [CGG$^+$09, CGG$^+$14, ABC$^+$06, GL09].

Developing algorithms that are resilient to faults is of paramount importance, and fine-grained parallel fixed point methods are no exception. This study works towards creating a simulation framework for asynchronous iterative methods that can be used to help develop algorithms that are resilient to faults. Creating simulation frameworks such as this allows for experimentation that is not specific to any singular platform or hardware architectures and allows experiments to simulate performance on both current computing environments and look at how those results may continue to evolve along with the computer hardware. These include the possibility to: (1) test and validate different fault-models (which are still emerging), (2) experiment with different check-pointing libraries/mechanisms, and (3) help in efficiently implementing asynchronous iterative methods. Additionally, it can be difficult to implement asynchronous iterative methods on a variety of architectures to observe performance behavior in different computing environments, and having a working simulation framework allows users to conduct extensive experiments without any major programming investment.

While many asynchronous methods are designed for shared memory architectures, such as those found in GPUs, and asynchronous iterative methods have gained popularity for their efficient use of resources on shared memory accelerators in modern HPC environments such as GPUs [VV+09], there has been some work done at improving the performance of asynchronous iterative methods using a distributed memory environment. This work includes attempts to implement asynchronous iterative methods in MPI-3 using one sided remote memory access [GBH14] as well as efforts to reduce the cost of communication in these environments [WPC16].

The structure of this paper is organized as follows: in Section 2, a brief summary of some related studies is provided. Section 3 provides an overview of asynchronous iterative methods is given, while in Section 4, details concerning the simulation framework used to model the performance of these methods are given. Section 5 gives the background information related to the creation of efficient checkpointing routines and provides a series of numerical results, while Section 6 concludes.

Note that the experiments in this study were conducted on Rulfo, a system powered by an Intel Xeon Phi Knight's Landing 7210 model processor with 256 cores at 1.30 GHz each. Rulfo belongs to the High-Performance Computing group of the Department of Modeling, Visualization and Simulation Engineering at Old Dominion University. The simulation experiments (i.e. both the algorithms and the fault simulation routines) were implemented in `MATLAB`, while the programs used to validate the simulation were written in C/C++ using OpenMP.

2 Related work

The expected increase in faults for future HPC systems is detailed in a variety of different sources. A high level article detailing the expected increase in failure rate from a reasonably non-technical point of view is available in the various versions of the "Monster in the Closet" talk and paper [Gei11, Gei12, Gei16]. More technical and detailed reports are given in a variety of sources composed of groups of different researchers from both academia and industry [ABC+06,

CGG+09, CGG+14, SWA+14, GL09]. Additionally, the Department of Energy has commissioned two very detailed reports about the progression towards exascale level computing; one from a general computing standpoint [ABC+10a] (summarized in [ABC+10b]), and a report aimed specifically at applied mathematics for exascale computing [DHB+14].

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 [Anz12, ADG15], as well as performance analysis of asynchronous methods [BBDH11, HD13, BBDH14]. In particular, both [BBDH11, BBDH14] focus on low level analysis of the asynchronous Jacobi method, similar to the example problem presented here.

Several numerically based fault models similar to the one that is used in this study have been developed recently, and are used as a basis for the generalized fault simulation that is developed here. These include a "numerical" fault model that is predicated on shuffling the components of an important data structure [EHM15], and a perturbation based model put forth in [SW15] and [CS16b]. Other models that are not based upon directly injecting a bit flip, such as inducing a small shift to a single component of a vector have been considered as well [HH11, BFHH12]. Comparisons between various numerical soft fault models have been made in [CS16a, CJB+17].

Fine-grained parallel methods, specifically parallel fixed point methods, are an area of increased research activity due to the practical use of these methods on HPC resources. An initial exploration of fault tolerance for stationary iterative linear solvers (i.e. Jacobi) is given in [ADQO15] and expanded on in [ADQO16]. Fault tolerance for synchronous fixed point algorithms from a numerical analysis has been investigated in [SW15]. Error correction for GPU based oriented asynchronous methods were investigated in [ALDH12], while the general convergence of parallel fixed point methods has been explored extensively [AB05, FS00, BT89, OR00, Bau78, Ben07]. Fault tolerance for fine-grained asynchronous iterative methods has been studied at a fundamental level [Gär99, CS17], as well as made specific to certain algorithms [CS17, CS18, ADQO15, ADQO16].

While many recent research results for asynchronous iterative methods are focused on implementations that utilize a shared memory architecture, one area of asynchronous iterative methods that has seen significant development using a distributed memory architecture is optimization [CC16, IBCH13, Hon17, ZC10, SN11, TBA86, BPC+11]

3 Asynchronous iterative methods

In fine-grained parallel computation, each component of the problem – i.e. a matrix or vector entry – is updated in a manner that does not require information from the computations involving other components while the update is being made. This allows for each computing element (i.e. a single processor, CUDA core or Xeon Phi core) to act independently from all other computing elements. Depending on the size of both the problem and the computing environment, each computing element may be responsible for updating a single entry to update, or may be assigned a block that contains multiple components. The generalized mathematical model that is used throughout this paper comes

from [FS00], which in turn comes from [CM69, Bau78] and [Szy98] (among many others).

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

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

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

$$x = (x_1, x_2, \ldots, x_m) \in D \longrightarrow G(x) = G(x_1, x_2, \ldots x_m) \tag{3}$$

$$= (G_1(x), G_2(x), \ldots, G_m(x)) \in D. \tag{4}$$

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

The assumption is also made that there is some finite number of processing elements $P_1, P_2, \ldots, P_p$ each of which is assigned to a block B of components $B_1, B_2, \ldots, B_m$ to update. Note that the number p of processing elements will typically be significantly smaller than the number m of blocks to update. With these assumptions, the computational model can be stated in Algorithm 1.

Algorithm 1: General Computational Model

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

This computational model has each processing element read all pertinent data from global memory that is accessible by each of the processors, update the pieces of data specific to the component functions that it has been assigned, and update those components in the global memory. Note that the computational model presented 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), i.e., without interleaving of its result. If each processing element P_l is to wait for the other processors to finish each update, then the model describes a parallel synchronous form of computation. On the other hand, if no order is established for P_ls, then an asynchronous form of computation arises.

To continue formalizing this computational model a few more definitions are necessary. First, set a global iteration counter k that increases *every* time

any processor reads $\vec{x}$ from common memory. At the end of the work done by any individual processor, p, the components associated with the block B_p will be updated. This results in a vector, $\vec{x} = (x_1^{s_1(k)}, x_2^{s_2(k)}, \ldots, x_m^{s_m(k)})$ where the function $s_l(k)$ indicates how many times an specific component has been updated. Finally, a set of individual components can be grouped into a set, I^k, that contains all of the components that were updated on the k^{th} iteration. Given these basic definitions, the three following conditions (along with the model presented in Algorithm 1) provide a working mathematical framework for fine-grained 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 $\vec{x}^0 \in D$, 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 that are both synchronous and asynchronous, though the three conditions given in Definition 1 are unnecessary in the synchronous case.

3.1 Asynchronous relaxation methods

Relaxation methods have been the focus of many of the works mentioned in Section 2 such as [CM69] and [Bau78]; a much more detailed description can be found in [BT89] among many other sources. This section provides an introduction that will serve as a reference for the later work in this study.

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

$$x^{k+1} = Cx^k + d \tag{5}$$

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

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

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

and following the methodology put forth in [BT89], this can be broken down to view a specific row – say the i^{th} – of the matrix A,

$$\sum_{j=1}^{n} a_{ij}x_j = b_i, \tag{7}$$

and this equation can be solved for the i^{th} component of the solution, x_i, to give,

$$x_i = \frac{-1}{a_{ii}} \left[\sum_{j \neq i} a_{ij} x_j - b_i \right]. \tag{8}$$

This equation can then be computed in an iterative manner in order to give successive updates to the solution vector. In synchronous computing environments, each update to an element of the solution vector, x_i, is computed sequentially using the same data for the other components of the solution vector (i.e. the x_j in Eq. (8)). Conversely, in an asynchronous computing environment, each update to an element of the solution vector occurs when the computing element responsible for updating that component is ready to write the update to memory and the other components used are simply the latest ones available to the computing element.

Expressing Eq. (8) in a block matrix form more similar to the original form of the iteration expressed in Eq. (5),

$$x = -D^{-1}((L+U)x - b) \tag{9}$$

$$= -D^{-1}(L+U)x + D^{-1}b \tag{10}$$

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

Convergence of asynchronous fixed point methods of the form presented in Eq. (5) is determined by the spectral radius of the iteration matrix, C, and dates back to the pioneering work done by both [CM69] and [Bau78]:

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

As noted in [WPC16], the iteration matrix C that is used in the Jacobi relaxation method serves as a worst case for relaxation methods of the form discussed here. However, because of the ubiquitous use of the Jacobi method in parallel solutions of large problems in many different domains in science and engineering we use the Asynchronous (Block) Jacobi method predominantly throughout the remainder of this study. Note that many of the concepts and ideas expressed in this paper can be easily adapted to more complex algorithms.

4 Overview of simulation framework

The simulation framework is designed to simulate the performance of an asynchronous iterative method operating on multiple computing elements using a single processing element. In this simulation framework, the emphasis is on fixed-point iterations[1]

$$\vec{x} = G(\vec{x}), \tag{11}$$

[1]Throughout the text, vector notation is occasionally adopted to emphasize when functions take all components of x as opposed to a single component, such as x_1.

for some $\vec{x} \in \mathbb{R}^n$. In the framework, certain components are assigned (possibly distinct) times for performing an update to their components, and the effects of various delay structures can be examined.

As a simple example, take $n = 2$. Then $\vec{x} = (x_1, x_2) \in \mathbb{R}^2$ and, using the terminology of Section 3,

$$x_1 = G_1(\vec{x}) = G_1(x_1, x_2), \tag{12}$$
$$x_2 = G_2(\vec{x}) = G_2(x_1, x_2). \tag{13}$$

In a traditional fully synchronous environment, both functions, G_1 and G_2, would be called simultaneously and no subsequent calls would be executed until both functions had returned and *synchronized* all results. In a fully asynchronous environment, both functions would be allowed to execute again immediately upon their own return, leading to a case where one of x_1 or x_2 may be updated more frequently than the other. Per Definition 1, both functions use the latest values of $\vec{x}$ that are available to them when the function call is initiated. For instance, if the processing element that was assigned to update the component x_1 was ten times as fast as the processing element assigned to update x_2, then in the amount of time needed to update x_2 once, the component x_1 will have been updated ten times, and when G_2 is called for the second time it will be called using the latest component of x_1 (which has been updated 10 times), and the latest component of x_2 (which has only been updated once).

In the simulation framework, time is abstracted away to "units of time". In this manner, the relative timing between different components is the only metric that matters. This allows the framework to be adapted to heterogeneous HPC environments, as well as examining the potential impact of the standard variance of single core performance on multi-core hardware elements if the method that is used is tuned to be completely asynchronous. Similarly, by adding or removing appropriate communication penalties, different memory architectures (i.e. distributed or shared) can be simulated.

As a more concrete example, let the matrix A result from a simple two dimensional finite-difference discretization of the Laplacian over a 10×10 grid, resulting in a 100×100 matrix with an average of 4.6 non-zero entries per row. The Laplacian

$$\Delta u = g, \tag{14}$$
$$\tag{15}$$

is a partial differential equation (PDE) commonly found in both science and engineering. The example problems taken in this study can be thought of as simulating the diffusion of heat across a two dimensional surface given some heat source along the boundary of the problem.

Once the PDE is discretized over the desired grid using finite differences, typically central finite differences, the linear system

$$Ax = b \tag{16}$$

is set up to be solved for a random right-hand side b that represents the desired boundary conditions. All problems considered in this study use Dirichlet boundary conditions. For the examples in this particular subsection, the right-hand side is generated by taking each component sampled as a uniform random

number between -0.5 and 0.5, and then normalizing the resultant vector. The iterative Jacobi method proceeds until the residual

$$r = b - Ax \tag{17}$$

is reduced past some desired threshold. Pseudocode for this simulated asynchronous Jacobi is given in Algorithm 2.

Algorithm 2: Asynchronous Jacobi simulation

Input: $a_{ij} \in A$, initial guess for x_0, a number of processing elements p, an input random number distribution
Output: Solution vector x

```
Assign processor update times. τ_1, τ_2, ..., τ_p by sampling from an
  appropriate random number distribution
Assign elements x_i ∈ x to each simulated processing element
for t = 1, 2, ... until convergence do
    for each processing element P_l do
        if τ_l = t then
            for each element x_i ∈ x assigned to P_l do
                x_i = (-1/a_ii) [ Σ_{j≠i} a_ij x_j - b_i ]
            Retrieve a new update time τ_l by sampling from the input
              distribution
    Calculate the residual as in Eq. (17) and check termination
      conditions
```

Note that a given update time, τ_k, will not be sampled as an integer. The simulation adjusts for this by scaling the number that is sampled by the appropriate order of magnitude, adjusting the maximum value allowed for t accordingly, and then scaling back the final time calculated by the simulation. For example, if the desired time precision is hundredths of a second, and the time resulting for the first sampling of τ_k was $1.234s$, then the simulation would perform the following steps:

1. $\tau_k^{\texttt{new}} = s * \tau_k^{\texttt{old}}$
2. $t_{\texttt{max}}^{\texttt{new}} = s * t_{\texttt{max}}^{\texttt{old}}$
3. $t_{\texttt{final}}^{\texttt{new}} = (1/s) * t_{\texttt{final}}^{\texttt{old}}$

where if the desired precision was hundredths of a second, $s = 10^2$, and the sampled value would become,

$$\begin{aligned}
\tau_k &= 1.234 - \textit{initial sample} \\
\tau_k &= 123.4 - \textit{apply scale factor} \\
\tau_k &= 123 - \textit{round to nearest integer}
\end{aligned}$$

An example of nominal performance in a synchronous shared memory environment (i.e. all processors update in the same amount of time and there is no added communication penalty) is provided by Fig. 1.

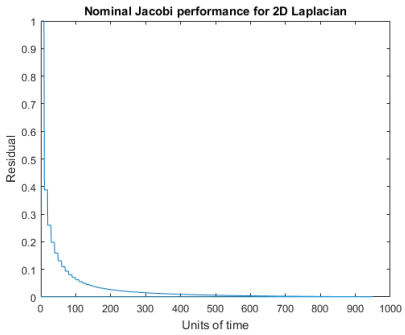

Figure 1: Example of nominal performance of the synchronous Jacobi iteration

As an example of the types of experiments that can be conducted, consider the same problem from above, but in two slightly more complicated scenarios. In the first—captured Fig. 2(left)—one of the ten processors involved in updating blocks of components of x is provided updates more slowly than the other processors. Each curve shows the progression of the (global) residual subject to having a single slower processor with different degrees of slowdown (from zero to 11x).

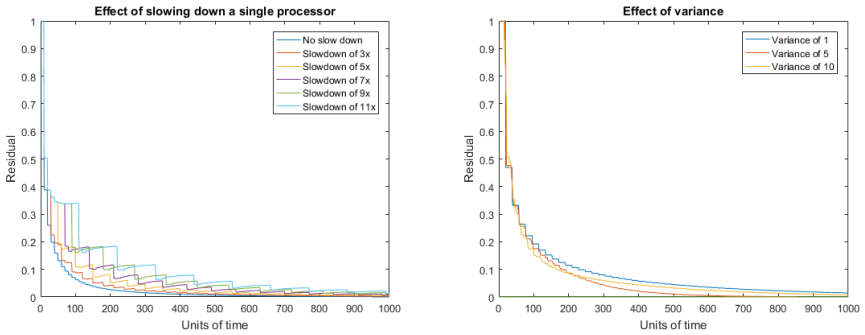

Figure 2: Examples of experiments within the simulation framework

In the second scenario—see Fig. 2(right)—the processor updates are not restricted to occur synchronously. Instead, the processors are assumed to have similar performance and perform their updates in time $t_i \sim N(\mu, \sigma^2)$ where the mean is set to 10 units of time and the variance is different for each curve depicted in the plot.

4.1 Tuning the simulation framework

The simple experiments highlighted above showcase a particular use of this simulation framework; namely, that it is easy to quickly generate data representative of *relative* differences in the performance of the individual processing elements. In order to observe the effect of some subset of the total processors performing updates in an asynchronous manner, possibly due to an active hardware issue on a node used, or to tie the simulation framework more closely with

true hardware performance and allow for extrapolation from a known point, a framework tuning process must be employed.

To tune and validate the simulation to current representative HPC environments, tools were created that use OpenMP to implement the asynchronous Jacobi method. These tools were executed on an Intel Xeon Phi Knight's Landing 7210 model processor.

Similar to Section 4, the test problem is a two dimensional discretization of the Laplacian

$$\Delta u = b \,, \tag{18}$$

where the right-hand side is initialized with Dirichlet boundary counditions.

Once data from representative performance code is obtained, metrics can be taken on the pertinent performance data, and these metrics can be used to help tune the simulation framework towards the desired model in order to allow the simulation that is generated to produce meaningful results. This tuning process has several benefits: In addition to providing empirical data with which to validate the simulation, it also allows a meaningful start to extrapolating the performance. Two implementations were set up to solve Eq. (18), and they will be discussed separately in the following two subsections.

4.1.1 Implementation 1

The Laplacian was this time generated over a 100×100 grid resulting in a matrix of size $10,000 \times 10,000$ with $49,600$ non-zeros with an average of 4.96 non-zeros per row. The thread count used was varied and the following thread numbers were utilized: 11, 21, 41, 51, 81, and 101. The vector b from the resulting linear system,

$$Ax = b, \tag{19}$$

is initialized such that the final solution vector has $x_i = 1$ for all i, the initial guess x^0 is all zeros.

In this implementation, each thread retrieves the data it needs from shared memory, performs the necessary computations, and writes the result back to shared memory. A dedicated thread computes the global residual value $b - Ax^{(t)}$ that determines satisfactory convergence. All threads use OpenMP locks to safely copy $x^{(t)}$ from shared memory, whether calculating the residual or computing $x_j^{(t+1)}$. Pseudocode for this process is given in Algorithm 3.

Algorithm 3: OpenMP Implementation 1

Input: $a_{ij} \in A$, initial guess for x_0, a number of processing elements p
Output: Solution vector x

1 Assign elements $x_i \in x$ to each processing element
2 **for** $t = 1, 2, \ldots$ *until convergence* **do**
3 **for** *each processing element* **do**
4 Copy the necessary components of $x^{(t)}$ from common memory
5 Compute $x_j^{(t+1)} = \frac{-1}{a_{ii}} \left[\sum_{j \neq i} a_{ij} x_j^{(t)} - b_i \right]$
6 Update $x_j^{(t)}$ in common memory with $x_j^{(t+1)}$ for j assigned to the current processing element

For each trial, the times for each thread to access the memory it needs (Line 4 of Algorithm 3), compute the relaxation for the rows assigned to it (Line 5), and to write the updated answer back to common memory (Line 6) were captured. Data was collected over multiple experiments, and this data was used to match a random number distribution that can be used to draw random numbers indicating the amount of time needed for a simulated processing element in the framework described above. This distribution can then be used as an input parameter to Algorithm 2 to generate random numbers that can be used to accurately predict update times for the various computing elements within the desired HPC architecture. For this particular problem, a lognormal distribution was found to match the data most closely. The pertinent parameters, μ and σ for this distribution, varied slightly for the various thread counts. Histograms containing data from the experiments conducted for a thread count of 51 are provided in Fig. 3 for the copy, compute action, and update action. Additionally, the best fitting lognormal distribution is plotted as a continuous curve on top of the data.

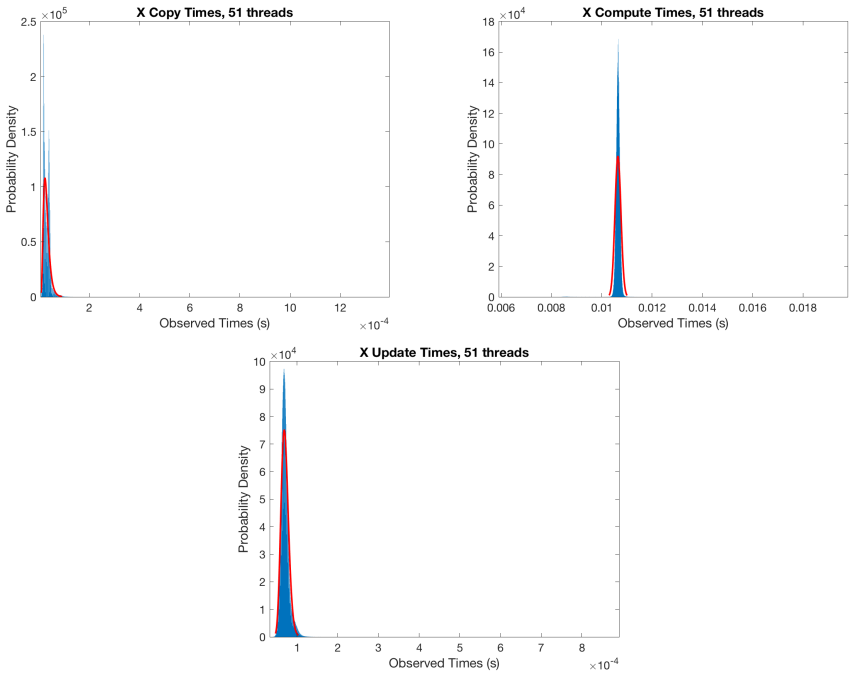

Figure 3: Histogram of copy (upper left), compute (upper right), and update (lower middle) times for the case of 51 threads

Aggregate statistics for these experiments—across all the thread and across the multiple runs of each experiment—-are provided in Table 1. These statistics can be used to tune the performance of the simulation framework, so that it provides results indicative of the current state of HPC hardware. In particular, the input distribution specified in Algorithm 2 can be set to the distribution that best matches the empirical data discovered.

The effect of increasing the number of threads on the shape of the distribution is shown in Fig. 4. The table captures the average time required for each

of the three main activities that were benchmarked in this implementation.

Table 1: Summary statistics for asynchronous performance tuning showing the average time to copy data to a thread, compute a relaxation, write the result back to memory, and the total update time.

Threads	Ave Copy (s)	Ave Comp. (s)	Ave Update (s)	Total (s)
11	1.86E-05	1.86E-02	3.01E-04	1.89E-02
21	2.09E-05	1.23E-02	1.53E-04	1.25E-02
41	2.89E-05	1.07E-02	8.40E-05	1.08E-02
51	2.96E-05	1.05E-02	7.00E-05	1.06E-02
81	7.00E-05	1.10E-02	7.40E-05	1.12E-02
101	2.51E-04	1.21E-02	1.02E-04	1.25E-02

Similarly, Fig. 4 shows the effects of increasing the thread count by plotting the probability density function of the distribution that best fit the empirical data that was captured. In order to help make the plot more readable, histograms of the actual data found are not included (in contrast to what Fig. 3 shows for a fixed thread count), but the probability density functions for all thread counts are overlayed.

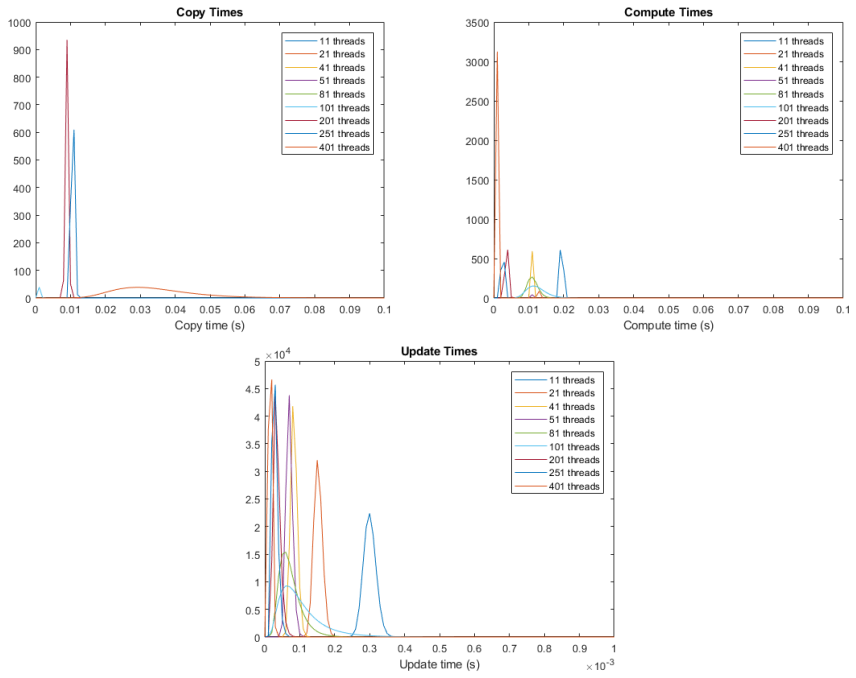

Figure 4: Effect of increasing the number of threads on copy (upper left), compute (upper right), and update (lower middle) times

The total time each core spends on calculating and storing the updated values is dominated by the "Compute" time; viewing the data in both Table 1 and in Fig. 4 shows that increasing the number of threads helps decrease the amount of time taken up to a point. At a saturation level of more than 1

thread per physical core, both the compute and the total update time begin to increase again. Before exhausting the cores that are physically available, increasing the number of threads has the effect of assigning each physical core fewer components to update which allows the computations to be done more quickly. The memory read and write activities (e.g. "Copy" and "Update") are less predictable with respect to the number of threads utilized, however, given the implementation used here, they both take orders of magnitude less time than actually performing the calculation.

4.1.2 Implementation 2

This second implementation performs the Jacobi relaxation on the grid directly using the neighboring points required by the 5-point stencil as opposed to explicitly forming the matrix A, and in a sense a implements *matrix-free* solution. For this implementation, the Laplacian was discretized over a 500×500 grid with boundary conditions set according to Table 2

Table 2: Boundary conditions for the second implementation of the Laplacian

0	100	...	...	100	0
75	XXX	...	...	XXX	50
⋮	XXX	...	...	XXX	⋮
⋮	XXX	...	...	XXX	⋮
75	XXX	...	...	XXX	50
0	0	...	...	0	0

The implementation used here stems from code provided by [HW10]; similar code solves a three dimensional discretization of the Laplacian in the study featured in [BBDH11] and [BBDH14]. The routine solves a heat diffusion problem, in which a two-dimensional heated plate has Dirichlet boundary-condition temperatures. Two matrices, u_0 and u_1, store grid point values that each thread reads, e.g. from u_1, to compute newer values to write, e.g. to u_0. As the method is asynchronous, each thread independently determines which matrix stores its newer $u^{(t+1)}(i,j)$ values and older $u^{(t)}(i,j)$ values. For an $n+2$ by $n+2$ grid, each thread solves for n^2/n grid points, such that the grid is evenly divided along the y-axis. When a thread copies grid point values above or below its domain for the computation, OpenMP locks are employed to ensure that data is captured from a single iteration. Further, locks are used when updating values on domain boundaries. Each thread p_n computes its local residual value every k^{th} iteration, which it contributes to the global residual value using an OpenMP atomic operation, such that it adds the local residual from the current iteration and subtracts the local residual from the previous iteration. A single thread checks for convergence with an atomic capture operation, and updates a shared flag variable if the criterion is satisfied. Pseudocode for this implementation is provided in Algorithm 4.

In this implementation, data was only collected for the total iteration time for an individual thread. Thread counts of 10, 25, 50, and 100 were used in this series of experiments. The average total iteration time for the varying

Algorithm 4: OpenMP Implementation 2

Input: Initial guess for $u^{(0)}(i,j)$, a number of processing elements p
Output: Solution vector $u(i,j)$

Assign elements $u(i,j)$ to each processing element
for $t = 1, 2, \ldots$ *until convergence* **do**
  for *each point in the grid,* $u^{(t)}(i,j)$ **do**
    Copy necessary boundary values for $u^{(t+1)}(i,j)$
    Compute $u^{(t+1)}(i,j) =$
     $1/4 * (u^{(t)}(i+1,j) + u^{(t)}(i-1,j) + u^{(t)}(i,j+1) + u^{(t)}(i,j-1)$
    Update $u_j^{(t)}(i,j)$ in common memory with $u_j^{(t+1)}(i,j)$

Table 3: Average iteration time with standard deviation (by thread count).

Threads	Ave Iter. Time (s)	Std. Iter. Time
10	0.6478E-04	0.3154E-05
25	0.3047E-04	0.2277E-05
50	0.2095E-04	0.2848E-05
100	2.2834e-05	5.5648e-06

An example of the iteration times for the case of 10 threads and 50 threads is given by Fig. 5

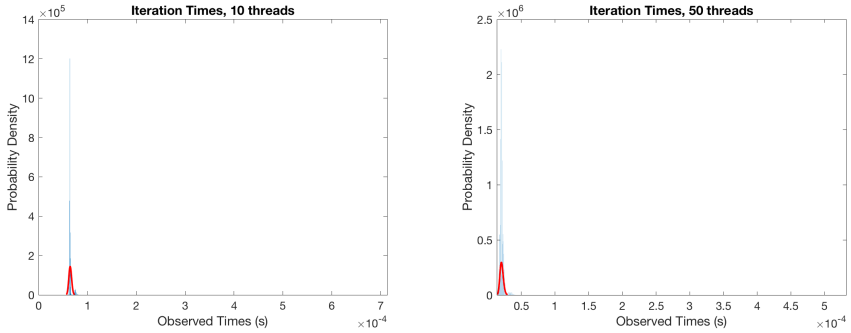

Figure 5: Histogram of iteration times for 10 threads (left) and 50 threads (right)

Since this implementation is even more compute bound then the first one, comparing Fig. 6 and Table 3 shows a general decrease in the time for each iteration as the thread count is increased. While there is no inflection point evident in the data presented in Table 3 (compared to Table 1), Fig. 6 still suggests that once the number of threads outnumbers the number of available physical cores that performance gains begin to drop off. For denser matrices, or for different applications, these trends could change as the memory based activities become relatively more expensive. The finite difference discretization of the Laplacian is a very sparse matrix that does not require much data movement.

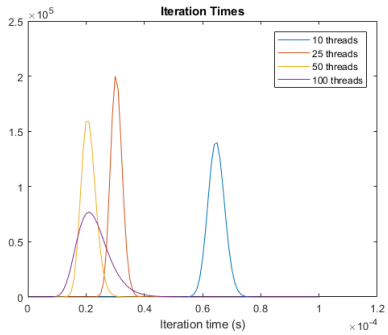

Figure 6: Effect of increasing thread count on total iteration time

Implementation Comparison. The first implementation has the relative disadvantage of requiring an explicit copy operation for each row a thread updates, but generalizes further to any sparse matrix, A, with which the Jacobi method can be used. From the result given in Theorem 1 this will occur if the spectral radius of the iteration matrix, C, is less than 1. In the case of the Jacobi method, the iteration matrix is given by

$$C = -D^{-1}(L + U). \tag{20}$$

Note that in the two dimensional discretization of the Laplacian, the spectral radius of the Jacobian is less then l, which says that both the synchronous and asynchronous variants of the Jacobi algorithm will converge.

The second implementation studied here minimizes the copy portion of the routine, assuming an appropriate number of processing elements for a given grid, and thus minimizes a portion of the routine more susceptible to natural fluctuations in HPC hardware performance. However, this routine only generalizes to finite difference discretizations of partial differential equations over rectangular grids.

The point of showing two distinct implementations is to emphasize that the simulation framework proposed here is capable of being easily tuned to reflect the performance of varying implementation styles. This simulation framework can be further adapted to any asynchronous iterative method through the process of collecting data representative of individual update times and using the resultant data to tune the model towards a particular use case.

5 Example of use: developing efficient checkpointing routines

An example of the efficacy of the simulation framework described in Section 4 is demonstrated here as to how the simulation may be used to determine efficient times to checkpoint data while executing an asynchronous iterative method.

5.1 Fault model

For this part of the study, faults are modeled as perturbations similar to several recent studies [CS16b, CSC17, SW15]; the goal being producing fault tolerant

algorithms for future computing platforms that are not too dependent on the precise mechanism of a fault (e.g. bit flip). Modifying the perturbation-based fault model described in [CSC17], a single data structure is targeted and a small random perturbation is injected into each component transiently. 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: sample a random number $r_i \in (-\epsilon, \epsilon)$ using a uniform distribution, and then set

$$\hat{x}_i = x_i + r_i \tag{21}$$

for all values of i. The resultant vector $\hat{x}$ is then perturbed away from the original vector x. Other similar perturbation-based fault models have sampled the components r_i from different ranges. This can allow the creation of scenarios where some components are perturbed by large amounts, and some are only changed incrementally.

In this study, faults are injected into the asynchronous Jacobi algorithm following the perturbation based methodology described above. Due to the relatively short execution time of the asynchronous Jacobi algorithm on the given test problems, a fault is induced only once during each run, and the fault is designated to occur at a random iteration number before convergence. To be precise – since "iteration" loses some meaning in an asynchronous iterative algorithm – the fault is injected on a single simulated time before the algorithm terminates. It is not necessary for the program to have an update scheduled on the same simulated time for the fault to be injected.

5.2 Experiments with the Simulation Framework

Similar to the earlier results in the paper, this study covers the solution of the linear system resulting from a two-dimensional finite difference discretization of the Laplacian. Before presenting simulation results, it is important to note that faults, as modeled here, will not prevent the *eventual* solution of the linear system using the (asynchronous) Jacobi method. Since the spectral radius of the associated iteration matrix is strictly less than 1, it will converge for any initial guess $x^{(0)}$. Since faults are assumed to only affect the memory storing the vector x and are assumed to occur in a transient manner, if a fault occurs on iteration F then the subsequent iterate, $x^{(F+1)}$ can be taken to be the new starting iterate and eventual convergence is guaranteed due to the iteration matrix which has remained the same throughout the occurrence of the fault. This model can reflect the scenario where certain parts of the routine are designated to run on hardware with a higher reliability threshold, and other parts of the algorithm are allowed to run on hardware that may be more susceptible to the occurrence of a fault. This sandbox type design has been suggested as a possible means for providing energy efficient fault tolerance on future HPC environments [BFHH12, HH11, SV13].

While eventual convergence may be guaranteed, greatly accelerated convergence is possible through a simple checkpointing scheme. An example of such a scheme (as an extension of the asynchronous Jacobi simulation provided by Algorithm 2) is provided in Algorithm 5.

Note that the asynchronous nature of the iterative method means that a strict check on the decrease of the residual (i.e. expecting monotonic decrease)

Algorithm 5: Asynchronous Jacobi simulation with checkpointing

Input: $a_{ij} \in A$; initial guess x_0; number of processing elements p; input random number distribution; checkpointing tolerance α; checkpointing frequency ω

Output: Solution vector x

```
1  Assign processor update times τ_1, τ_2, ..., τ_p by sampling from an
     appropriate random number distribution
2  Assign a part of x to each processing element
3  Initialize r_old to a large value
4  for t = 1, 2, ..., until convergence do
5      for each processing element, P_l do
6          if τ_k = t then
7              for each element x_i ∈ x assigned to P_l do
8                  x_i = (-1/a_ii) [ Σ_{j≠i} a_ij x_j - b_i ]
9              Retrieve a new update time τ_k by sampling from the input
                 distribution
10     Inject a fault if appropriate
11     Calculate the residual r_new as in Eq. (17)
12     if r_new > α × r_old then
13         x ← x_cp
14     if  mod (t, ω) == 0 then
15         x_cp ← x
16     Check termination conditions
```

is not possible. In particular, the checkpointing tolerance α needs to be taken such that $\alpha > 1$. However, the expected manifestation of faults as rare, transient events allows α to be taken fairly large. Taking α too large results in a fault having a substantial impact on the convergence rate of algorithm since large faults will be allowed to impact the algorithm with no correction. Conversely, taking α too small causes the algorithm to checkpoint more frequently than needed. Examples of the effects of a fault with different values selected for α are given by Fig. 7.

Note in Fig. 7 that no checkpointing results in a delay to convergence relative to the use of checkpointing with either $\alpha = 1$ or $\alpha = 10$. The size of the fault selected in this study, $r_i \in (-100, 100)$, which may be reflective of an exponent or sign bit flip [CS18], results in the values $\alpha = 1$ and $\alpha = 10$ having the same performance since the error induced by the fault is sufficiently large that the new residual is more than $\alpha = 10$ times the prior residual. Faults that induce a smaller error may be detected by certain values of α and not by others which would lead to differing performance.

The residual progress in the plot showing the effects of using $\alpha = 1$ can be explained by the updates provided by certain simulated processing elements being rejected despite being necessary for the convergence of the algorithm. This can be seen in the small, momentary jumps in the progression of the residual visible in the other graphs. These rejections lead to stagnation in the progression of the algorithm and show why the value of $\alpha = 1$ should not be selected for a checkpointing scheme for an asynchronous iterative method.

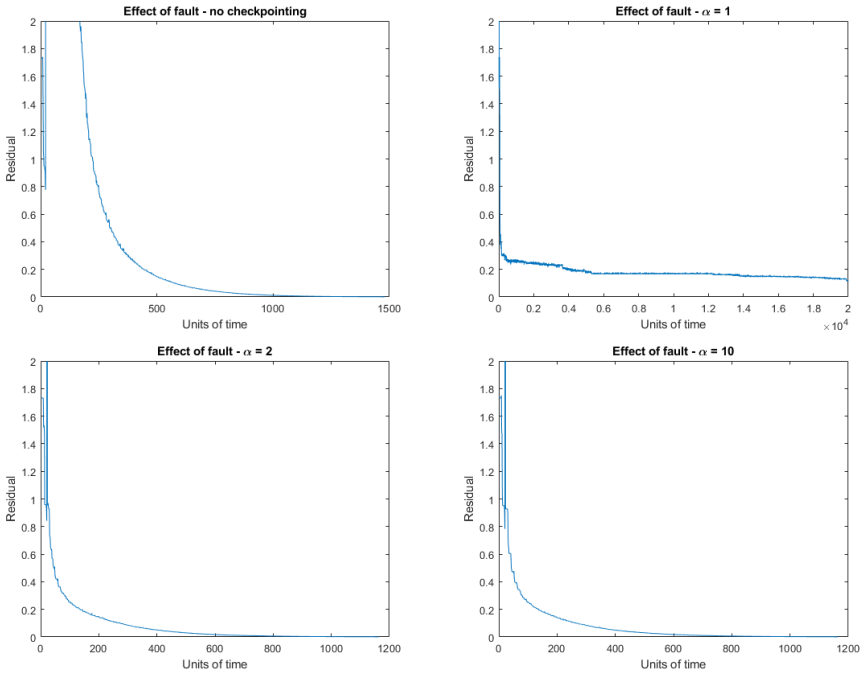

Figure 7: Effect of differing values of α on the progression of the residual

6 Conclusions and Future Work

This work has developed a framework that can be used to efficiently simulate the outcomes of asynchronous methods in heterogeneous computing environments and in the presence of transient soft faults. Given that asynchronous methods are notoriously difficult to study theoretically, their simulation is an invaluable tool for observing their behavior and making quantitative and qualitative assertions as was shown by example in Section 5.2, in which the checkpointing tolerance α was assessed. The framework is extensible and flexible and is able to: (1) admit a variety of asynchronous methods, beyond the Jacobi algorithm, (2) incorporate different fault models, and (3) vary hardware parameters such as thread and processor counts.

In the future, it is planned to automate the selection of the checkpointing tolerance as well as checkpointing frequency in the course of simulation. Furthermore, simulation framework will be augmented with runtime simulation measurements, such those provided by Intel Running Average Power Limit (RAPL) interface [Gui11], to obtain simulated application execution traces in order to model application performance and energy consumption.

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 would also like to acknowledge Edmond Chow for discussions regarding simulation of asynchronous iterative processes and the `MATLAB` script he provided that evolved into part of the proposed simulation framework.

7 References

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

[ABC+06] K Asanovic, R Bodik, BC Catanzaro, JJ Gebis, P Husbands, K Keutzer, DA Patterson, WL Plishker, J Shalf, SW Williams, et al. The landscape of parallel computing research: A view from Berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[ABC+10a] 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, Technical report, United States Department of Energy, Fall, 2010.

[ABC+10b] Steve Ashby, PETE Beckman, Jackie Chen, Phil Colella, BILL Collins, DONA Crawford, Jack Dongarra, DOUG Kothe, Rusty Lusk, PAUL Messina, et al. The opportunities and challenges of exascale computing–summary report of the advanced scientific computing advisory committee (ascac) subcommittee. *US Department of Energy Office of Science*, 2010.

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

[ADG15] Haim Avron, Alex Druinsky, and Anshul Gupta. Revisiting asynchronous linear solvers: Provable convergence rate through randomization. *Journal of the ACM (JACM)*, 62(6):51, 2015.

[ADQO15] Hartwig Anzt, Jack Dongarra, and Enrique 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*, page 1. ACM, 2015.

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

[ALDH12] Hartwig Anzt, Piotr Luszczek, Jack Dongarra, and Vincent Heuveline. GPU-accelerated asynchronous error correction for mixed precision iterative refinement. *Euro-Par 2012 Parallel Processing*, pages 908–919, 2012.

[Anz12] Hartwig Anzt. *Asynchronous and multiprecision linear solvers-scalable and fault-tolerant numerics for energy efficient high performance computing*. PhD thesis, Karlsruhe, Karlsruher Institut für Technologie (KIT), Diss., 2012, 2012.

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

[BBDH11] Iain Bethune, J Mark Bull, Nicholas J Dingle, and Nicholas 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.

[BBDH14] Iain Bethune, J Mark Bull, Nicholas J Dingle, and Nicholas J Higham. Performance analysis of asynchronous Jacobis method implemented in MPI, SHMEM and OpenMP. *The International Journal of High Performance Computing Applications*, 28(1):97–111, 2014.

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

[BFHH12] PG Bridges, KB Ferreira, MA Heroux, and M Hoemmen. Fault-tolerant linear solvers via selective reliability. *arXiv preprint arXiv:1206.1390*, 2012.

[BPC+11] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

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

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

[CC16] Yun Kuen Cheung and Richard Cole. A unified approach to analyzing asynchronous coordinate descent and tatonnement. *arXiv preprint arXiv:1612.09171*, 2016.

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

[CGG+14] F Cappello, A Geist, W Gropp, S Kale, B Kramer, and M Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014.

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

[CM69] Daniel Chazan and Willard Miranker. Chaotic relaxation. *Linear algebra and its applications*, 2(2):199–222, 1969.

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

[CS16a] Evan Coleman and Masha Sosonkina. A Comparison and Analysis of Soft-Fault Error Models using FGMRES. In *Proceedings of the 6th annual Virginia Modeling, Simulation, and Analysis Center Capstone Conference*. Virginia Modeling, Simulation, and Analysis Center, 2016.

[CS16b] Evan Coleman and Masha Sosonkina. Evaluating a Persistent Soft Fault Model on Preconditioned Iterative Methods. In *Proceedings of the 22nd annual International Conference on Parallel and Distributed Processing Techniques and Applications*, 2016.

[CS17] Evan Coleman and Masha Sosonkina. Fault Tolerance for Fine-Grained Iterative Methods. In *Proceedings of the 7th annual Virginia Modeling, Simulation, and Analysis Center Capstone Conference*. Virginia Modeling, Simulation, and Analysis Center, 2017.

[CS18] Evan Coleman and Masha Sosonkina. Self-Stabilizing Fine-Grained Parallel Incomplete LU Factorization. *Sustainable Computing: Informatics and Systems*, 2018.

[CSC17] Evan Coleman, Masha Sosonkina, and Edmond Chow. Fault Tolerant Variants of the Fine-Grained Parallel Incomplete LU Factorization. In *Proceedings of the 2017 Spring Simulation Multiconference*. Society for Computer Simulation International, 2017.

[DHB+14] Jack Dongarra, Jeffrey Hittinger, John Bell, Luis Chacon, Robert Falgout, Michael Heroux, Paul Hovland, Esmond Ng, Clayton Webster, and Stefan Wild. Applied mathematics research for exascale computing. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2014.

[EHM15] 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.

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

[Gär99] Felix C Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys (CSUR)*, 31(1):1–26, 1999.

[GBH14] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. Enabling highly-scalable remote memory access programming with MPI-3 one sided. *Scientific Programming*, 22(2):75–91, 2014.

[Gei11] A Geist. What is the monster in the closet? In *Invited Talk at Workshop on Architectures I: Exascale and Beyond: Gaps in Research, Gaps in our Thinking*, volume 2, 2011.

[Gei12] A Geist. Exascale monster in the closet. In *2012 IEEE Workshop on Silicon Errors in Logic–System Effects, Champaign-Urbana, IL, March*, pages 27–28, 2012.

[Gei16] Al Geist. Supercomputing's monster in the closet. *IEEE Spectrum*, 53(3):30–35, 2016.

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

[Gui11] Part Guide. Intel® 64 and ia-32 architectures software developers manual. *Volume 3B: System programming Guide, Part*, 2, 2011.

[HD13] James Hook and Nicholas Dingle. Performance analysis of asynchronous parallel jacobi. *Numerical Algorithms*, pages 1–36, 2013.

[HH11] M Hoemmen and MA Heroux. Fault-tolerant iterative methods via selective reliability. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE Computer Society*, volume 3, page 9. Citeseer, 2011.

[Hon17] Mingyi Hong. A distributed, asynchronous and incremental algorithm for nonconvex optimization: An admm approach. *IEEE Transactions on Control of Network Systems*, 2017.

[HW10] Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[IBCH13] Franck Iutzeler, Pascal Bianchi, Philippe Ciblat, and Walid Hachem. Asynchronous distributed optimization using a randomized alternating direction method of multipliers. In *Decision and Control (CDC), 2013 IEEE 52nd Annual Conference on*, pages 3671–3676. IEEE, 2013.

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

[SN11] Kunal Srivastava and Angelia Nedic. Distributed asynchronous constrained stochastic optimization. *IEEE Journal of Selected Topics in Signal Processing*, 5(4):772–790, 2011.

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

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

[SWA+14] M Snir, RW Wisniewski, JA Abraham, SV Adve, S Bagchi, P Balaji, J Belak, P Bose, Franck Cappello, B Carlson, et al. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, 2014.

[Szy98] Daniel B Szyld. Different models of parallel asynchronous iterations with overlapping blocks. *Computational and applied mathematics*, 17:101–115, 1998.

[TBA86] John Tsitsiklis, Dimitri Bertsekas, and Michael Athans. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE transactions on automatic control*, 31(9):803–812, 1986.

[VV+09] Sundaresan Venkatasubramanian, Richard W Vuduc, et al. Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems. In *Proceedings of the 23rd international conference on Supercomputing*, pages 244–255. ACM, 2009.

[WPC16] Jordi Wolfson-Pou and Edmond Chow. Reducing communication in distributed asynchronous iterative methods. *Procedia Computer Science*, 80:1906–1916, 2016.

[ZC10] Minyi Zhong and Christos G Cassandras. Asynchronous distributed optimization with event-driven communication. *IEEE Transactions on Automatic Control*, 55(12):2735–2750, 2010.