Implementing Asynchronous Linear Solvers Using Non-uniform Distributions

Evan Coleman[1] Erik Jensen[2] Masha Sosonkina[2]

[1] Naval Surface Warfare Center, Dahlgren Division
[2] Old Dominion University; Computational Modeling & Simulation Engineering Department

Categories: Computing methodologies~Linear algebra algorithms, Computing methodologies~Massively parallel algorithms, Applied computing~Mathematics and statistics
Keywords: Asynchronous iteration, linear solvers, randomized linear algebra

Abstract

Asynchronous iterative methods may improve the time-to-solution of their synchronous counterparts on highly parallel computational platforms. This paper considers asynchronous iterative linear system solvers that employ non-uniform randomization and develops a new implementation for such methods. Experiments with a two-dimensional finite-difference discrete Laplacian problem are presented. The new finer grain implementation is compared with an existing, block-based, one and shown to be superior in terms of the convergence speed and accuracy. In general, using non-uniform distributions in selecting components to update may lead to faster convergence. In particular, the new implementation convergences up to 10% faster when it uses a non-uniform distribution.

1 Introduction

Asynchronous iterative methods describe a class of parallel iterative algorithms where each computing element is allowed to perform its task without waiting for updates from any of the other processes. These methods are often applied to the parallel solution of fixed-point problems and have been used in a wide variety of applications including: the fault-tolerant solution of linear systems [1], the preconditioning of linear solvers [6], and optimization [16], among many others. These solvers tend not to converge to high precision as quickly as their Krylov subspace counterparts; however, they can converge very quickly to a low level of accuracy [3]. This loss of accuracy may cause the use of asynchronous linear solvers to be suboptimal for some applications, but the fact that they are able to reach an approximate solution quickly opens up several other application areas. Possible use cases include preconditioning to a Krylov method, solving systems that may not need a high level of accuracy (e.g., big data and machine learning), or smoothing a multigrid method.

Under study here are asynchronous iterative methods for solving linear systems of the form $Ax = b$, such as asynchronous Jacobi. One way to attempt to improve the performance of asynchronous linear solvers is to have each processor select randomly the (block of) components it updates next, as opposed to fixing an update order *a priori*. This approach has been studied previously by [3], for the case where the random selection is done uniformly. Our work continues to investigate the potential performance increase of *dynamically* weighting the random selection of the next component to update. In the synchronous case, weighting the selection using the norm of the row of A associated with the selected component has been done previously [21, 12, 10]. However, the idea employed here is to periodically sort and rank the residuals associated with each component and make the random selection using a non-uniform distribution that is more likely to select components with a larger contribution to the residual. This is motivated by the success of weighted stationary solvers, such as the Southwell iteration, which typically converge in fewer iterations than traditional Jacobi or Gauss-Seidel relaxation schemes do so (see e.g., [18] and[22]).

The preceding work, Enhancing Asynchronous Linear Solvers through Randomization, studied using a non-uniform distribution to select components to update. The present work extends that work in [7] by making the following new contributions:

- Proposes a new row-based randomized asynchronous linear solver with a significantly different approach to the selection of components to update;
- Develops an alternative component ordering criterion that uses component differences instead of residuals;
- Observes experimentally that new row-based solver exhibits convergence in fewer component relaxations than serial Gauss-Seidel;
- Compares the performance of the block- and row-based solvers and demonstrates that the proposed new solver improves upon the block-based one.

The structure of the rest of the paper is as follows: Section 2 provides information on related studies. Section 3 gives an overview of asynchronous iterative methods. Section 4 provides the design of randomized asynchronous iterative solvers that use non-uniform distributions. Section 5 presents experimental results, first, separately for each of the two implementations considered in this work and follows with their comparisons. Section 6 concludes.

2 Related Work

The Department of Energy has commissioned two very detailed reports about the progression towards exascale level computing; one from a general computing standpoint conducted by [2], and a report aimed specifically at applied mathematics for exascale computing by [8]; both of which emphasize the importance of developing scalable algorithms moving forward towards exascale platforms. Development of scalable applications on a large scale starts with modifying algorithms that form the basis for those applications, and the stationary iterative methods examined here (e.g., Jacobi, Gauss-Seidel, block variants) form an important aspect of many preconditioning techniques for Krylov subspace methods, as well as commonly acting as smoother in multigrid methods.

Several recent studies focus on improving scalability by attempting to remove the synchronization delay: a fine-grained algorithm for computing incomplete LU factors for the purposes of preconditioning of linear solvers was created by [6], an optimization technique based upon an asynchronous approach to stochastic gradient descent was created by [16], and the efficacy of asynchronous multigrid smoothers was explored for CFD applications in [11].

The use of randomization in linear algebra has found use in a variety of areas including transforming linear systems using Random Butterfly Transformations to eliminate (with probability 1) the need for pivoting. This has been used to aid in the performance of direct solvers for dense matrices by [15], and later adopted for sparse matrices by [4]. Other examples include the random component selection in stochastic gradient descent methods, including an early study in Srivastava and Nedic (2011) that incorporates asynchronous computation. More pertinent to the topic studied here, randomized linear relaxation based solvers have been studied in the past by [19] who extend the original asynchronous model presented by [5] to allow component choice and (theoretical) delay to be based upon probability distributions.

The present work follows a greedy approach, similar in spirit to the Southwell iteration. [22] extend a Southwell-oriented approach to the case of parallel asynchronous solvers, whereby an equation is relaxed if it has the largest residual among all coupled equations.

3 Overview of Asynchronous Iterative Methods

In asynchronous computation, each part of the problem is updated such that no information from other parts is needed while each individual computation is performed. This allows each processor to act independently. The model that is shown here to provide a basis for asynchronous computation comes mainly from [9]. To start, consider a fixed point iteration with the function, $G : D \to D$. 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 as shown in Algorithm 1.

If each processor (P_l) waits 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.

At the end of an update by processor P_l, the components associated with the block B_{P_l} 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, and k is a global iteration counter that is updated every time that any processing element makes an update. A set of indices I^k contains the components that were updated on the k^{th} iteration. Given these definitions, the three following conditions provide a framework for asynchronous computation:

Definition 1. *If the following three conditions hold:*

Algorithm 1 General Computational Model

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

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

Then given an initial $x^0 \in D$, the iterative update process defined by,

$$x_l^{(k)} = \begin{cases} x_l^{(k-1)} & l \notin I^k \\ G_l(x^{(k)}) & l \in I^k \end{cases},$$

where each $G_i(x)$ uses the latest updates available, is called an asynchronous iteration.

This basic computational model provided by the combination of Algorithm 1 and Definition 1 allows for many different results on fine-grained iterative methods. In particular, the authors' earlier work ([7]), introduced a block-based randomized asynchronous linear solver that used non-uniform distributions for dynamically prioritizing components to update.

Relaxation methods have been the focus of many studies related to asynchronous iterations starting with [5]. They are typically used to solve linear systems of the form $Ax = b$ and can be written as fixed point iterations that can be expressed as

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

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 a relaxation method that can be used in an asynchronous manner and the update for a given component x_i can be expressed as

$$x_i = \frac{-1}{a_{ii}} \left[\sum_{j \neq i} a_{ij} x_j - b_i \right]. \tag{2}$$

This iteration can give successive updates to the x_i component in 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 values for x_j in Eq. (2)). 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. (2) in a block form similar to Eq. (1) gives an iteration matrix of $C = -D^{-1}(L+U)$ where D is the diagonal portion of A, and L and U are the strictly lower and upper triangular portions of A respectively. Convergence of asynchronous fixed point methods of the form presented in Eq. (1) is determined by the spectral radius of the iteration matrix, C.

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

If x^* is the fixed point of the iteration defined by the matrix C, then convergence is given by ensuring that the error at a given iteration, $\|x^{(m)} - x^*\|$, is sufficiently small. In practice, this is accomplished by verifying that the residual, $r^{(k)} = b - Ax^{(k)}$, is beneath a given threshold. Asymptotic results such as this, i.e. that guarantee eventual convergence but offer no guarantee as to the rate of that convergence, exist for many variants of the iteration described above (see [9] for a summary).

3.1 Randomized Linear Solvers

The use of randomization in asynchronous linear solvers allows for the possibility of statements concerning the rate of convergence to be made. A randomized Gauss-Seidel method was introduced by [12] building off of the randomized Kaczmarz algorithm proposed by [21], whereby the decrease in the expected value of the error at each step is bounded. The analysis was generalized by Griebel and Oswald who also added a new parameter that allows for both over and under relaxation [10]. Both of these studies weight the random selection of row i by the size of the element $a_{ii} \in A$. In the case that A has unit diagonal this simplifies to a uniform distribution. More recently, [3] build upon the analysis by [12] and [10] and explicitly analyze the case of asynchronous computation with a uniform distribution.

All of the methods select the vector component to update (see Eq. (2)) from a random distribution instead of either sequentially looping through the available components or by tying the updates for a single component to a particular processor. In a traditional parallelization of either a synchronous or asynchronous linear solver, processor j is responsible for updating component j; the asynchronous variant allows processor j to continue to compute relaxations for the component assigned to it regardless of the state of the other processors. The use of randomization in the selection of which component to update allows for the possibility of any processor updating any component. In a randomized asynchronous linear solver, when a processor finishes computing an update to a component, it writes the update to the shared memory and then randomly draws the next component to update from the list of all available components. In the randomized asynchronous linear solvers proposed by others to date, this random selection is always done using either uniform random number generation, or with a probability proportional to a row norm of the matrix A. Leventhal and Lewis cite Fourier analysis [12] as an application area that can benefit from this type of weighting; however, there is no reason not to expect improved behavior for an arbitrary problem. The authors have proposed in [7] to use the non-uniform distributions in the asynchronous Jacobi iterative method. In this work, efficient implementations of such an iterative method are investigated.

3.1.1 Southwell Algorithm

The Southwell algorithm [18] works similarly to Jacobi by relaxing a single equation at a time, but chooses the equation with the largest local contribution to the residual. For a given row i, this local contribution is defined to be

$$r_i^{(k)} = b_i - Ax_i^{(k)} \tag{3}$$

at iteration k. This difference allows the Southwell algorithm to often converge in fewer iterations than Jacobi, but raises the expense of computing an update since the local residuals need to be stored and ranked at each iteration. After a given iteration, the Southwell algorithm chooses the component that contributes the most to the global residual; thus, the algorithm ranks the residuals from largest to smallest. Using the insight from the Southwell algorithm, the idea behind the randomized linear solvers developed here is for each processor to select the next component for updating randomly, using a distribution that more heavily weights selection of components that contribute more to the global residual. Pseudo-code for a randomized variant is provided in Algorithm 2. The key difference of the present work is that here non-uniform distributions in Line 3 of Algorithm 2 are investigated.

Algorithm 2 Generic Randomized Linear Solver

```
1: for each processing element P_l do
2:   for i = 1,2,..., until convergence do
3:     Pick j ∈ {1,2,...,n} using a given probability distribution
4:     Read the corresponding entries of A,x,b
5:     Perform the relaxation for equation x_j
6:     Update the data for x_j
7:   end for
8: end for
```

In an effort to simulate the effect of the Southwell algorithm using randomized asynchronous solvers, the *local residuals* associated with each equation (or block of equations) are ranked and sorted, and the selection of the next equation (i.e., component) to update is performed using a non-uniform distribution that forces the random selection to pick components with larger local residuals more frequently. The goal behind the proposed modification is that relaxing the components with a more significant contribution to the global residual may reduce the total number of

iterations required. Motivation for this comes from a myriad of different studies, see for instance the paper [14] that shows that for some cases (Gauss-)Southwell selection can converge faster than uniform random selection for coordinate descent. In general, the improvement in convergence will have to be shown to be significant enough to offset the extra computational and communication cost associated with storing and ranking all of the local residuals. To help offset the increased computational expense, the periodicity with which the sorting and ranking procedures are done is experimented with since it contributes directly to the overall efficiency of the algorithm.

4 Asynchronous Solver Design with Non-Uniform Distributions

The focus here is initially on the potential performance of different randomized asynchronous linear solvers through a series of tests in MATLAB® (Section 4.2), followed by the descriptions of two shared-memory algorithms, block-based and a novel row-based, in Sections 4.3 and 4.4, respectively.

4.1 Problem Description

This work examines the asynchronous Jacobi relaxation algorithm for solving finite-difference discretizations of PDEs on a regular grid. In science and engineering, 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 [17]. The specific problem under study here is Laplace equation in two dimensions:

$$\nabla^2 \phi = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = b \,, \tag{4}$$

where the two-dimensional finite-difference discretization uses Dirichlet boundary conditions. This PDE, which is a fundamental equation for modeling equilibrium and steady state problems, is also used in more complex problems based on PDEs. Equation (4) may be discretized such that a finite difference operator computes difference quotients over a discretized domain. For example, the two-dimensional discrete Laplace operator

$$\left(\nabla^2 f\right)(x,y) = f(x-1,y) + f(x+1,y) + f(x,y-1) + f(x,y+1) - 4f(x,y) \tag{5}$$

approximates the two-dimensional continuous Laplacian using a five-point stencil [13]. From this, a discretized version of the Jacobi algorithm

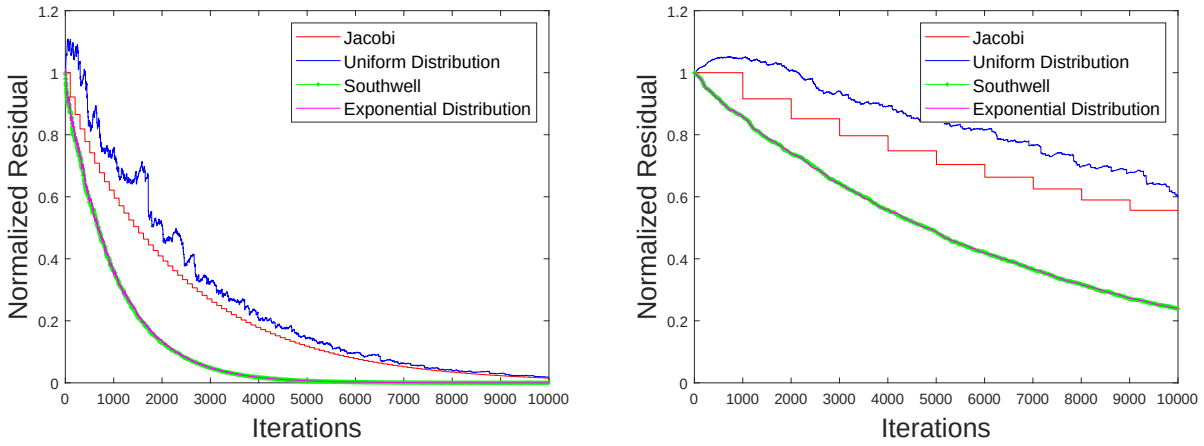
$$v_{l,m}^{k+1} = \frac{1}{4}(v_{l+1,m}^{k} + v_{l-1,m}^{k} + v_{l,m+1}^{k} + v_{l,m-1}^{k}) \tag{6}$$

may be applied to solve a two-dimensional sparse linear system of equations [20]. Indices l, m, and k define discrete grid nodes in two dimensions and the iteration number, respectively, for updating the discretized solution vector v.

In the particular instance of this 2D Laplacian problem, as solved with the Jacobi method here, the grid of 800×800 is used to obtain experimental results, the Dirichlet boundary conditions are 100, 0, 75, and 50 for the top, bottom, left, and right boundaries, respectively; the solution vector v is initilalized to 0 in each non-boundary grid point, and the right-hand side vector b is equal to the initial v.

4.2 Proof-of-Concept

Preliminary experiments are performed using MATLAB®, to demonstrate the improvement in convergence with Southwell and with non-uniform component selection, compared with Jacobi and with uniform component selection, for the problem tested in this work. As an example of potential convergence rates, Fig. 1 shows the progression of the residuals over the first 10,000 iterations when solving the two- and three-dimensional finite-difference discretizations of the Laplacian over a 10×10 and $10 \times 10 \times 10$ grids, respectively. Here, the four solution methods used are the traditional synchronous Jacobi algorithm, a traditional Southwell algorithm, and two randomized linear solvers: one choosing the component to update using a uniform random distribution, and another using an exponential random number distribution with the parameter $\lambda = 2$. Note that the convergence of the randomized linear solver using the uniform distribution is slightly inferior to traditional solvers and to the one with exponential distribution. The latter performs on par with the Southwell, both in the 2D and 3D cases.



(a) 2D problem (5-pt stencil, 10×10 grid)

(b) 3D problem (27-pt stencil, $10 \times 10 \times 10$ grid)

Figure 1: Residual (r/r_0) progression for the first 10,000 iterations of four stationary methods solving the 2D (a) and 3D (b) Laplacian.

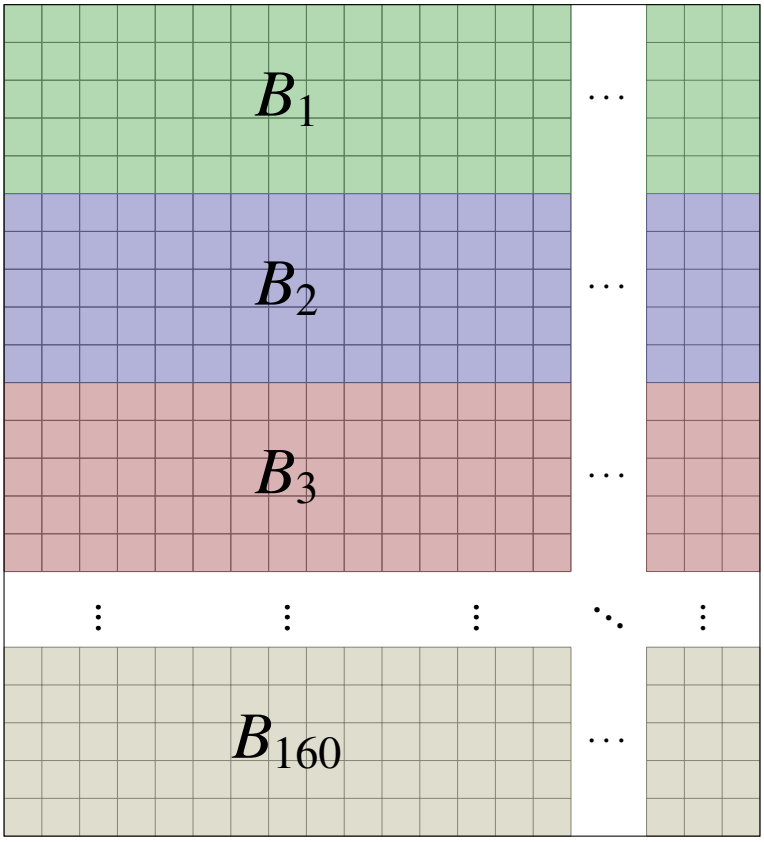


Figure 2: Block assignment used in the 800×800 grid of the example problem. The blocks consist of all components in a five-row section of the grid. This incorporates 4000 of the 640,000 grid points into each block resulting in $\hat{n} = 160$ blocks.

Algorithm 3 Block Variant of Randomized Linear Solver

1: **Input:** ranking period τ, number of block-rows $\hat{n}$, number of block-relaxations m, probability distribution function f
2: Set $c = 0$ counter for all block relaxations
3: **for** each thread **do**
4: **for** $i = 1, 2, \ldots,$ until convergence **do**
5: **if** thread is *master* **and** $(c \mod \tau)$ is 0 **then**
6: Rank and sort block residuals
7: **end if**
8: Pick $j \in \{1, 2, \ldots, \hat{n}\}$ using f
9: Perform m relaxations on block B_j
10: Update the data for B_j
11: $c = c + m$
12: **end for**
13: **end for**

4.3 Block-based Algorithm

The following block-based algorithm design has been introduced in [7] and is provided here as the reference for a wider performance analysis and comparison with the novel, row-based, algorithm. In the task-based asynchronous solver, a thread chooses a block of grid rows to update by sampling from a distribution. The number it draws corresponds to an index in a list of blocks, ranked in order of descending component residuals. For example, if a thread draws the number zero from the distribution, it will update the block-row of components with the largest residual, assuming that block is not being updated by another thread. In the case that a thread selects a block that is already being worked on by another thread, the selecting thread searches sequentially either up or down in the rankings until it finds an available block.

Initially, block residual rankings are assigned via a natural ascending ordering (see Fig. 2). A single thread, denoted the residual ranking thread, is tasked with computing the component residuals, sorting the residual rankings, and updating the global ranking list that all the threads use to selects blocks for updating. Note that using a single thread leads to a more accurate global ranking list and does not result in a bottleneck for a moderate number of threads. For large-scale distributed implementations, a different ranking procedure has to be developed.

In this work, the residual ranking thread performs ranking and list-updating after every five iterations of the linear system solver. Essentially, Algorithm 2 may be modified to include ranking periodicity τ as shown in Algorithm 3. This ranking period needs to be chosen judiciously, depending on several factors, such as the number m of relaxations performed, the number of threads used, and the number $\hat{n}$ of block-rows to rank. Here, $\tau = 5$ was found experimentally to mitigate the ranking overhead for the obtained number of iterations to convergence, while the number of relaxations was varied. A more detailed investigation of the ranking periodicity is warranted and left as future work.

4.4 Row-based Algorithm

Algorithm 4 illustrates a novel row-based method . Similarly to Algorithm 3, the master thread periodically, every τ relaxations, ranks and sorts the rows (line 20). However, there are several important distinctions between the two algorithms, due to which Algorithm 4 exhibits better performance. In line 10, a thread uses a probability distribution function f to select a *single* target row to relax instead of a block of rows shown in Algorithm 3, and then transitions from the current (start) row $\tilde{r}$ to the target row r_v by relaxing all the rows between $\tilde{r}$ and r_v in their natural ordering, instead of jumping to the target row to relax next as done in the block-based implementation (Algorithm 3). Furthermore, while making this transition, a thread may move inward the domain or toward its top or bottom boundary rows, depending on the direction of the shortest distance d_v Eq. (7) from the current start row to the target.

$$d_v = \min\left(n - |\tilde{r} - r_v|, |\tilde{r} - r_v|\right), \tag{7}$$

where n is the total number of rows in the subdomain, and the direction of progression to the target is toward and across the boundary if the first term in Eq. (7) is taken as d_v; otherwise, the boundary is not crossed. The former is also

chosen when the terms are equal. Then, in line 13, the `nextr` function assigns the next row number to consider by decrementing or incrementing the row number $\tilde{r}$ for the boundary or non-boundary progression direction, respectively; and performing circular shift of the row numbers if they reach the boundary. Note that fewer than d_v rows may be relaxed if certain rows in the path towards the target row are not *free*, i.e., they are already being relaxed by another thread at the time of their consideration, as specified by the conditional statement in line 14. A shared array of size n maintains row availability, in which a threads "locks" the row number while it relaxes that row and releases the lock upon finishing the operations in lines 15–20.

Algorithm 4 Row-Based Variant of Randomized Linear Solver

1: **Input:** probability distribution function f, ranking period τ, number of rows n
2: Set row-sum differences $\Delta = \{\delta_j = N_{\max} \mid j = 1,\ldots,n\}$, where δ_j is row-sum difference between adjacent relaxations of row j and $N_{\max}$ is the largest double-precision number
3: Set row ranking R as ascending natural ordering
4: Set sorted rows $S = (1,2,3,\ldots,n)$
5: Set $c = 0$ counter for all row relaxations
6: **for** each thread **do**
7: Set $r_v \in \{1,2,\ldots,n\}$ for initial thread target row
8: **for** $i = 1,2,\ldots,$ until convergence **do**
9: Set previous target as new start row $\tilde{r} = r_v$
10: Set target row r_v from sorted rows S using f
11: Compute shortest distance d_v as in Eq. (7)
12: **for** $j = 1,\ldots,d_v$ **do**
13: Assign next row $\tilde{r} = \texttt{nextr}(\tilde{r}, j)$ as described in Section 4.4
14: **if** $\tilde{r}$ is *free* **then**
15: Perform a relaxation of $\tilde{r}$
16: Update the data for $\tilde{r}$
17: Compute row-sum difference $\delta_{\tilde{r}}$ as in Eq. (8)
18: Set $c = c + 1$
19: **if** thread is *master* **and** $(c \mod \tau)$ is 0 **then**
20: Update ranking R and sorted rows S based on Δ
21: **end if**
22: **end if**
23: **end for**
24: **end for**
25: **end for**

The use of the shortest distance is motivated by an attempt to adhere to the ranking order of rows while also relaxing in the neighborhood of the target row; thereby, making the transition to the target smoother. Additionally, in a distributed-memory environment, the ability to more frequently relax boundary rows may facilitate a better data movement among subdomains possibly leading to a faster convergence. Another distinction between the block-based implementation and the row-based one is that the row-based performs the ranking of rows using row-sum differences instead of residuals. In particular (see line 17), after every row $\tilde{r}$ relaxation, a thread performs a summation $\sigma_{\tilde{r}}$ of the absolute values of all the components in $\tilde{r}$ and updates the row-sum difference $\delta_{\tilde{r}}$ Eq. (8):

$$\delta_{\tilde{r}} = \left|\sigma_{\tilde{r}}^{-} - \sigma_{\tilde{r}}\right|, \tag{8}$$

where $\sigma_{\tilde{r}}^{-}$ is the sum taken after the previous relaxation of $\tilde{r}$. This difference $\delta_{\tilde{r}}$ is assumed to be decreasing between the two adjacent relaxations and arbitrarily small when the algorithm has converged. A strong linear relationship has been observed between the row difference method rank and the row residual rank during the entire convergence process. Table 1 presents a small sample of representative correlation coefficients R at regular intervals throughout a sample calculation, which quantify the magnitude and direction of this relationship. Of the hundreds of thousands of computed correlation coefficients, the minimum and mean coefficients are 0.77 and 0.96, respectively, with a standard deviation of 0.02. Using this difference instead of residuals decreases the computational overhead of ranking the rows. In particular, finding the row difference requires about 7 times fewer floating point operations per iteration than when

Table 1: Comparison of the row difference method rank with the row residual rank, for all rows, at various row ranking iterations during the calculation. Correlation coefficient R quantifies magnitude and direction of relationship.

ROW RANKING ITERATION	0	$20e3$	$40e3$	$60e3$	$80e3$	$100e3$	$120e3$	$140e3$
R	0.99	0.99	0.96	0.95	0.97	0.96	0.97	0.98

using the row residual for this problem. Note, that, while it is shown that the difference-ranking method works for this sample problem, it has not been tested with other types of problems.

5 Implementation Results

The block-based and row-based algorithms are implemented and tested on two shared-memory computing platforms. For both platforms and both implementations, results show that the calculation time decreases using non-uniform distributions, compared with a uniform distribution. Additionally, the row-based implementation shows a decrease in iterations, compared with Gauss-Seidel.

5.1 Experimental Design

The experiments using OpenMP® are conducted on two computing platforms at Old Dominion University[1] The Rulfo system has an Intel®Xeon Phi™Knight's Landing 7210 model processor with 64 cores running at 1.30 GHz and 112 GB of DDR4 physical memory used as DRAM in these experiments. One thread per core is utilized, with one core reserved for interfacing with the operating system, resulting in 63 computational threads for the experiments in Section 5.2. On the Wahab system, a single node of the cluster is utilized, containing two Intel Xeon Gold 6148 CPUs each with 20 physical cores and 376 GB of DDR4 memory. The code uses standard C++ routines for sorting residuals and generating random numbers, with the default parameters and the built-in distributions. Experimental parameters are presented in Table 2.

5.2 Block-based Implementation on Rulfo

For block selection, three different distributions are tested. The *uniform* distribution is used as a control; a thread may select any block with equal probability. The *normal* distribution is used to examine the effects of targeting different segments of blocks in the rankings, i.e., blocks with lower ranks and higher residuals versus blocks with higher ranks and lower residuals. This is achieved by varying the mean parameter μ while keeping the standard deviation σ fixed in the normal distribution. The *exponential* distribution, with the mode λ close to zero, will tend to sample lower-ranked blocks.

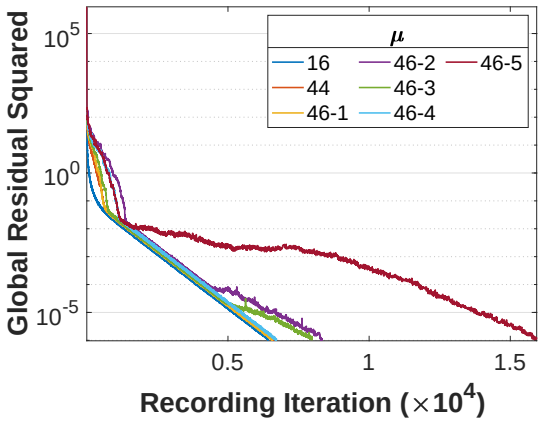
For both normal and exponential distributions, the algorithm convergence may be observed in Figs. 3 and 4, respectively. Here and in figures throughout Section 5, the term *Recording Iteration* points out that the data is recorded by a thread every 1,000 iterations. For the normal distribution, it may be observed in Fig. 3a that the convergence rate depends strongly on μ: Its smaller values (up to $\mu = 46$) lead to rapid convergence whereas, at $\mu = 46$, the convergence sharply deteriorates. This may be also observed when considering the time-to-solution in Fig. 3b. Due to very slow convergence, at large μ values, the normal distribution becomes extremely non-competitive with the uniform distribution, which timing is shown as red dashed line in Fig. 3b. Figure 4a shows that the parameter λ for the exponential distributions does not have as much an impact on performance as the parameter μ does so for the normal distribution runs. As λ moves farther away from zero, however, it hinders convergence and the exponential distribution results in slower timings than those obtained with the uniform distribution as seen in Fig. 4b. Once the best parameter choices are found for normal and exponential distributions, their performances compare favorably to the uniform distribution (Fig. 5), and up to 10% and 13% fewer iterations are observed, respectively.

Figure 6 provides a more detailed explanation for performance differences based on the selection of μ. In particular, Figs. 6a and 6b depict that the ordered component residual values for μ equal to 16 and 44 and are nearly indistinguishable. However, when μ increases to 48 (Fig. 6c) and then again to 52 (Fig. 6d) residuals of the lowest-ranked blocks decrease slowly while the residuals of all other blocks decrease more quickly. Note that all the block-based

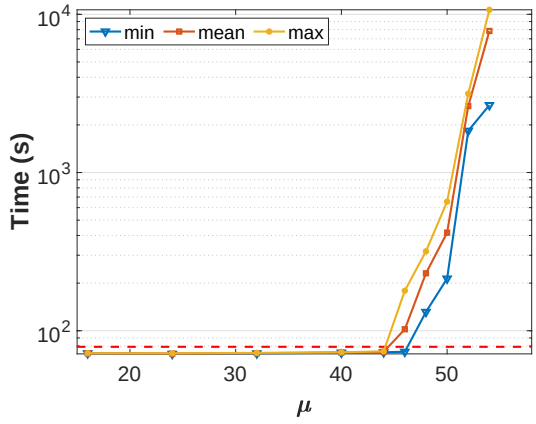
[1]The code is available from the corresponding author by request.

Table 2: Experiment parameters for BLOCK-BASED and ROW-BASED implementations run on Rulfo and Wahab platforms (column `Hardw`). The number of OpenMP® threads is shown in column `Thrds`. The problem (grid) size is shown in column `Grid`. The number of rows considered by a thread at a time is given in column `Block`. Input tolerance for the algorithm convergence is provided in column `Tol`, while the ranges of the normal (μ, σ) and exponential λ distribution parameters are provided in columns `Norm` and `Expo`, respectively.

	Hardw	Thrds	Grid	Block	Tol	Norm	Expo
BLOCK-BASED	Rulfo	63	800×800	5	1e-3	(16–54,8)	0.01–0.8
BLOCK-BASED	Wahab	40	800×800	5	1e-3	(16–40,8)	0.01–0.8
ROW-BASED	Wahab	40	800×800	1	1e-3	(80–400,40)	0.002–0.16

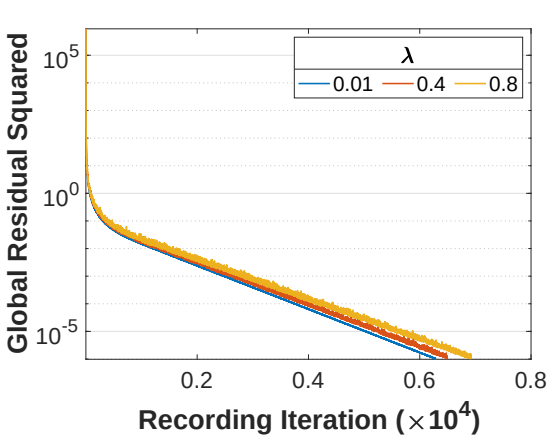


(a) Convergence history;
For a given μ, '-1',...,'-5' enumerate the runs

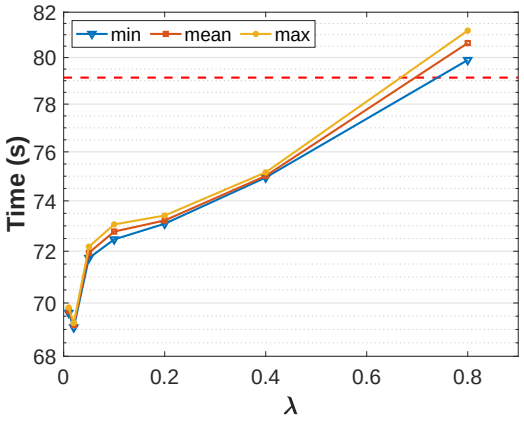


(b) Time-to-solution: minimum, average, and maximum timings over 5 runs

Figure 3: BBI convergence for normal distribution.



(a) Convergence history



(b) Time-to-solution: minimum, average, and maximum timings over 5 runs

Figure 4: BBI convergence for exponential distribution.

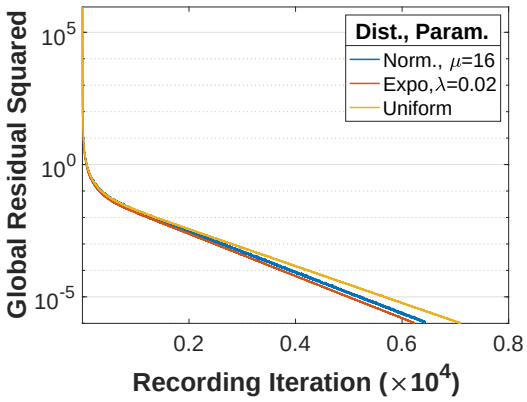
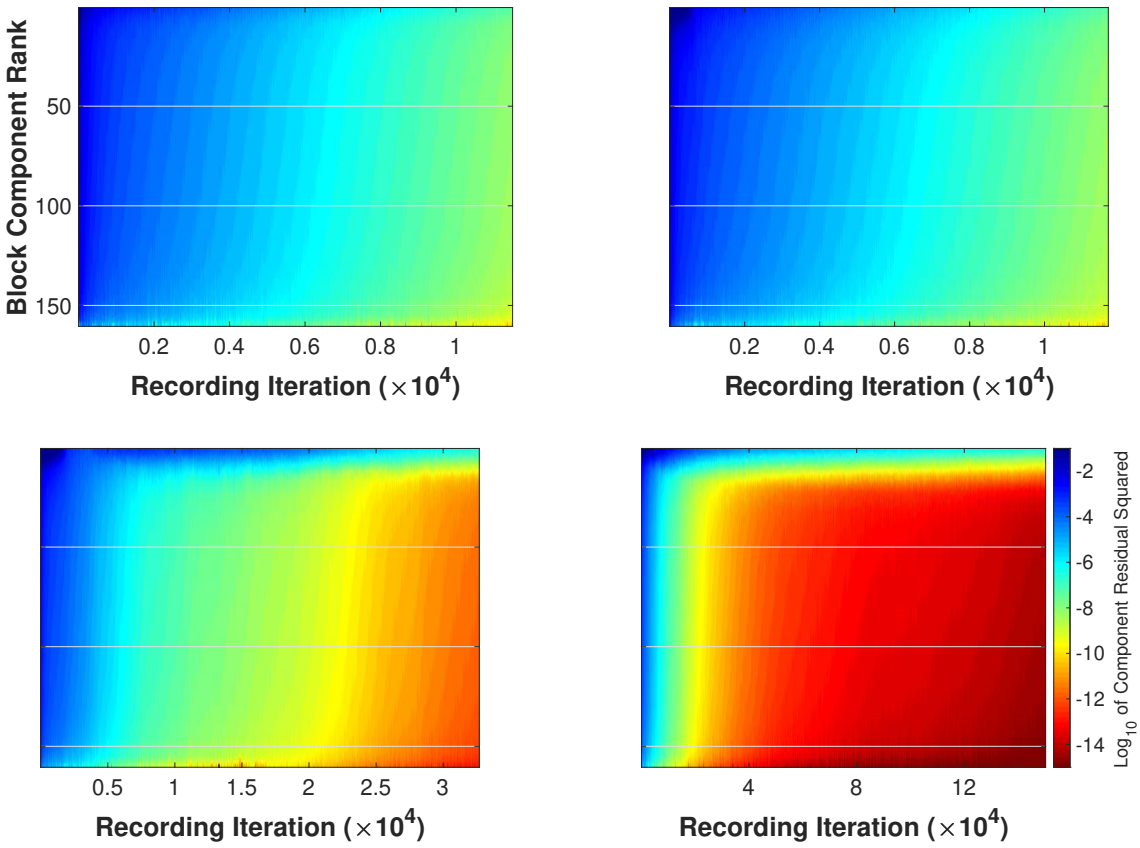


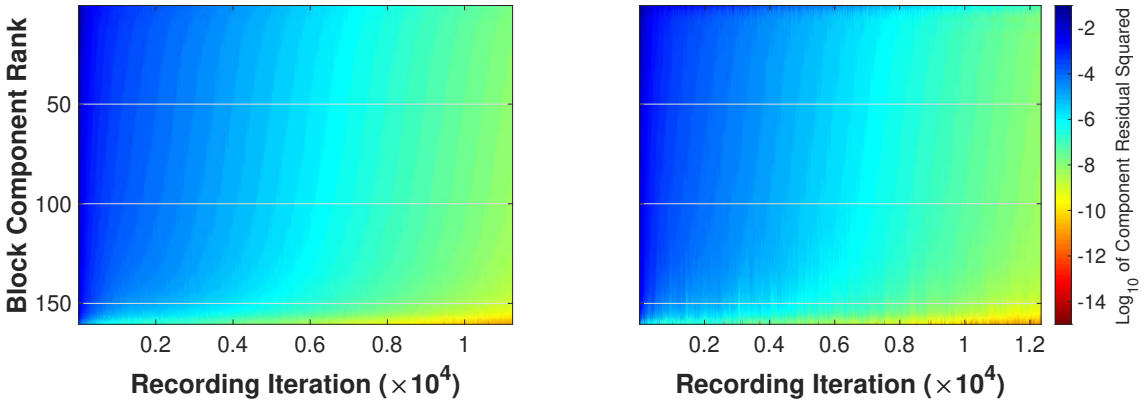
Figure 5: BBI convergence history comparisons among distributions in the best case.



(c) μ=48

(d) μ=52

Figure 6: Block-row residuals for calculations using normal distributions.



(a) λ=0.01 (b) λ=0.8

Figure 7: Block-row residuals for calculations using exponential distributions.

implementation (BBI) experiments use 8 for σ, which is appropriate for all the chosen μ ranges of 16 to 54 on Rulfo and 16 to 40 on Wahab.

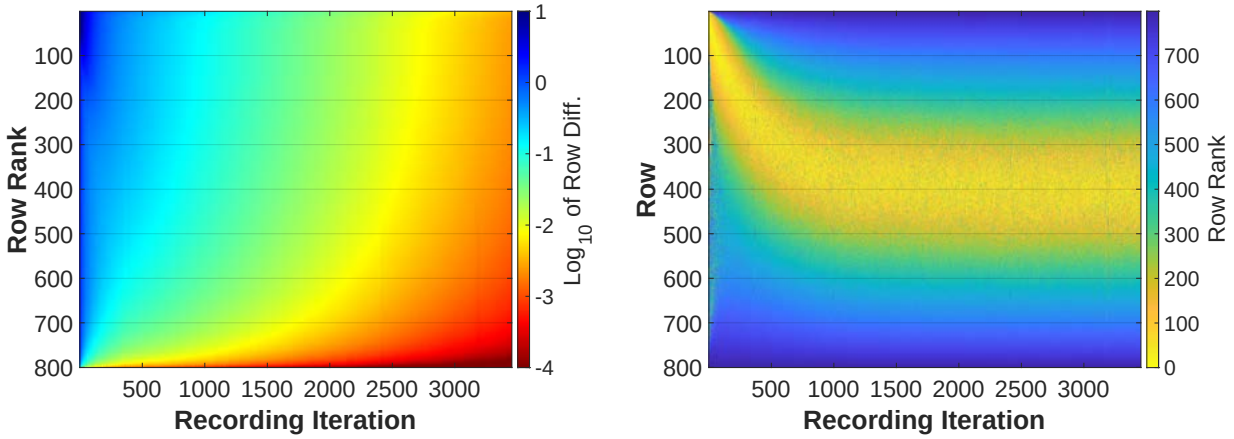
Figures 7a and 7b show that, for the minimum and maximum values of λ, respectively, the component residual decrease is balanced among the component ranks as iterations progress.

5.3 Row-based Implementation on Wahab

The results of the BBI show that non-uniform probability distribution functions may be used to efficiently select components to update, leading to convergence for the sample problem used in this work. However, relaxing blocks of rows asynchronously tends to cluster errors on block boundaries, and thereby hindering convergence. A row-based implementation (RBI) has been introduced to mitigate this problem. Here, the RBI solves the same sample problem in shared memory as BBI (see Section 5.2). Recall that RBI does not consider blocks of rows to be relaxed by a single thread. Instead, a thread selects only a single row to relax at a time.

For row selection, as with the BBI, the same three distributions are tested. Again, the uniform distribution is used as a baseline for comparison with the normal and exponential distributions. Similar to the BBI experiments, the normal and exponential distributions are geared to consider different ranges of row numbers by, respectively, keeping the standard deviation σ parameter fixed and the parameter λ close to zero. Figure 8a shows the diminishing row differences as the system converges, and the disparity between the rows with the least and greatest differences decreases. In Fig. 8b, initially the lowest-index rows have the greatest differences since these are the boundary rows, and in effect, the greatest discontinuity initially is between the top boundary and the first row of grid points (see Section 4.1). Conversely, the least discontinuity initially is between the bottom boundary and the last row of grid points. These respective discontinuities are reflected in the row component differences of consecutive iterations, i.e., the top row initially changes quickly, while the bottom row changes slowly. However, as the calculation progresses, the change in the first rows decreases. For most of the calculation, the middle rows experience the most change.

For the normal distribution, Fig. 9 shows the effects of choosing appropriate and excessively large values of the normal distribution mean parameter μ, values of 80 and 400, respectively. Note that the normal distribution standard deviation parameter σ is kept at 40, which is appropriate for the range of μ values considered for RBI here. Compared with Fig. 9a, Fig. 9b shows increased iterations, greater row-difference disparity between bottom and top-ranked rows, and increasing row differences for ranks 300–400 during the first 1000 iterations. Similarly to Fig. 8b, Fig. 10 shows the rank changes during the convergence processes albeit here for the normal distribution for the same parameters as in Fig. 9. In Fig. 10a with $\mu = 80$, the middle rows are targeted so frequently that the ranks of the rows with the greatest differences are pushed outward, toward the first and last ranks, much more than what is observed for the uniform and normal with $\mu = 400$ distributions (cf. Figs. 8b and 10b, respectively). For $\mu = 400$, because the lower-difference rows are targeted more often, the group of high-ranked rows (shown as the middle yellow band) does not shift ranks to the extent seen with the uniform distribution in Fig. 8b, and hence, is updated fewer times, which leads to inferior convergence. This pattern is expected to continue for $\mu > 400$. For the exponential distribution, Figs. 11 and 12 show the progression of row differences and rankings, respectively. Here, both small and large values of λ provide similar



(a) Row Differences

(b) Row Rankings

Figure 8: Progression of row differences and rankings using a uniform distribution.

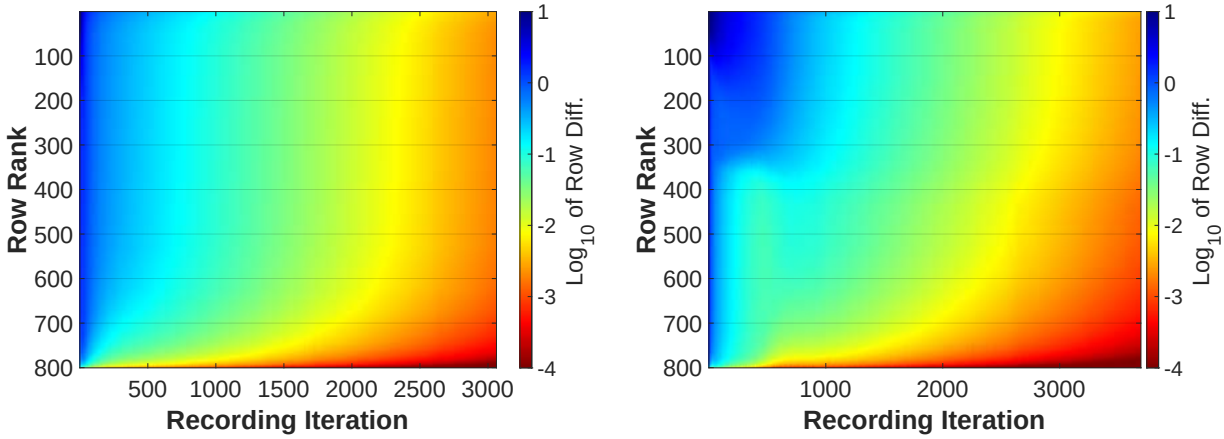
results and are equally effective, on par with good values of μ when using the normal distribution. The convergence history is presented in Fig. 13 for the three distributions and their respective parameter choices considered for RBI. As expected, for the exponential distribution and the normal distribution with smaller μ of 80, the residual decreases more quickly than with the uniform distribution, whereas with the normal distribution parameter $\mu = 400$, the residual decreases the mostly slowly in Fig. 13.

5.4 Performance Comparison of Block- and Row-based Implementations

Here, block- and row-based implementations are mutually compared on the same platform, Wahab, as to their number of relaxations and time to converge for a range of non-uniform distribution parameters μ and λ, as shown in Table 2.

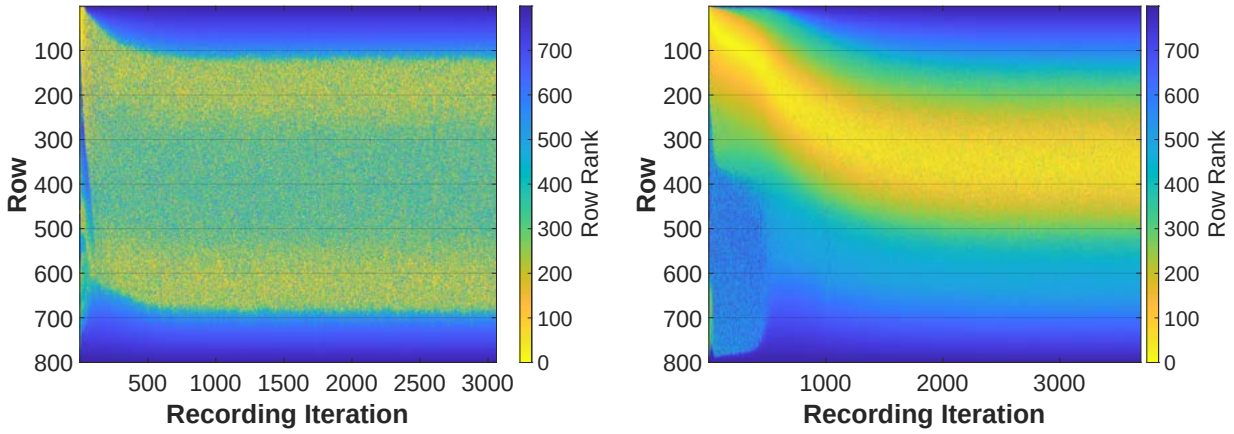
Note that the distribution parameters in the row-based implementation differ from those used by the block-based one, which reflects the sorted array sizes and different convergence behavior of the implementations. In particular, for the given test problem, the BBI has 160 entities (blocks) to sort, while there are 800 entities (rows) to sort in the case of RBI. The difference in convergence behavior is especially evident when comparing results from the two implementations when both use normal distributions to select components. In Fig. 14a, for BBI, there is a distinct difference in results for $\mu = 44$ and $\mu = 46$. For RBI, Fig. 14b shows a smoother transition between *good* and *poor* normal distribution parameters. Note that *good* and *poor*, respectively, are termed so because they yield the calculation times faster and slower than those for the uniform distribution test cases. In particular, the *poor* distribution parameters are those starting with the first μ that yields a significant jump in the calculation time; and this percentage increase for RBI is taken to be comparable with the one in BBI. By comparing the results for the μ values in Figs. 14a and 14b, it is seen that the RBI tolerates a much higher relative value for μ than the BBI does so before significantly degrading the performance. For example, while $\mu = 46$ is already a poor choice for the BBI, $\mu = 230$ (which is equal to 46×5 rows in a block) is still well within the range of good parameters for the RBI.

In addition, Figs. 14a and 14b compare the block- and row-based implementation iterations with the iterations of serial Gauss-Seidel (shown as red horizontal lines), respectively, to converge for the sample problem. The BBI cannot converge in fewer than serial Gauss-Seidel component relaxations even with the best distribution parameters. The RBI, however, converges in about 10% fewer component relaxations than serial Gauss-Seidel, using non-uniform distributions with appropriate parameters. This happens consistently, although it has been shown theoretically that more component relaxations may be required when threads update components asynchronously [3]. A better convergence in the RBI compared with that in BBI may be attributed to the (*fine-grained*) ranking of rows rather than blocks and to relaxing all the rows on the path from the current and the selected target one. Such a relaxation process leads to a smoother transition between rows and possibly to relaxations of more rows by a thread at a time than those contained in a block of the BBI. Although the row-based implementation ranks and sorts more entries than the BBI does so, the former has faster time-to-solution (see Fig. 18) and is not hindered at large scales—where distributed implementations



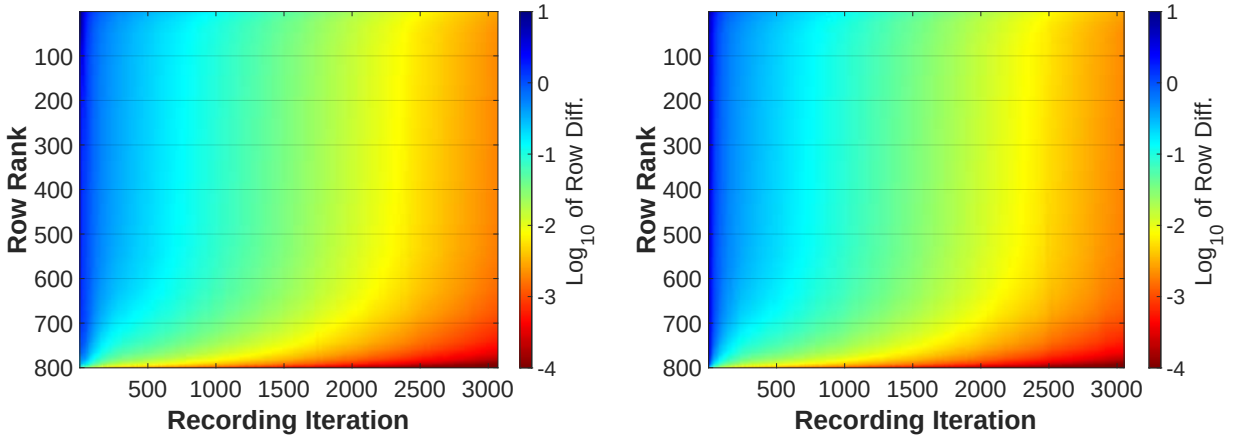
(a) $\mu = 80$ (b) $\mu = 400$

Figure 9: Progression of row differences using normal distributions.



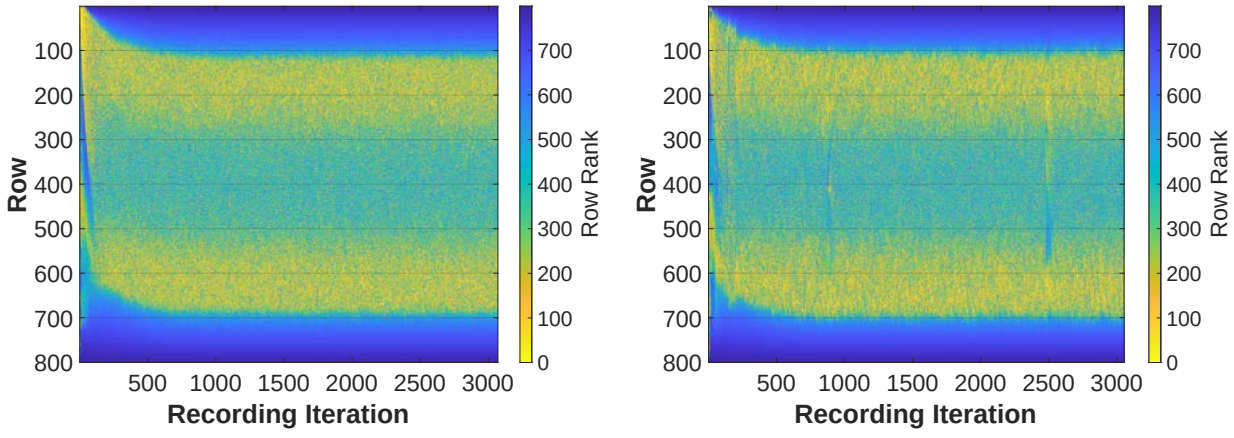
(a) $\mu = 80$ (b) $\mu = 400$

Figure 10: Progression of row rankings using normal distributions.



(a) $\lambda = 0.02$ (b) $\lambda = 0.16$

Figure 11: Progression of row differences using exponential distributions.



(a) $\lambda = 0.02$ (b) $\lambda = 0.16$

Figure 12: Progression of row rankings using exponential distributions.

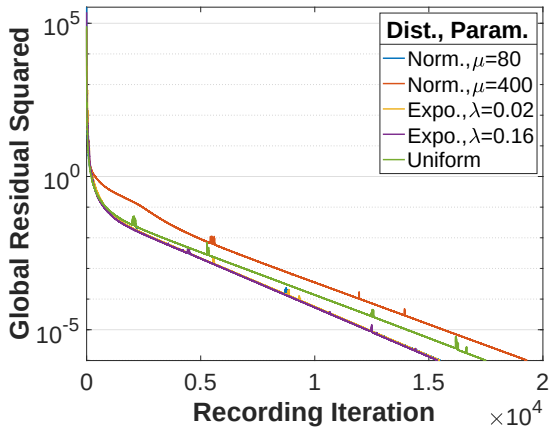
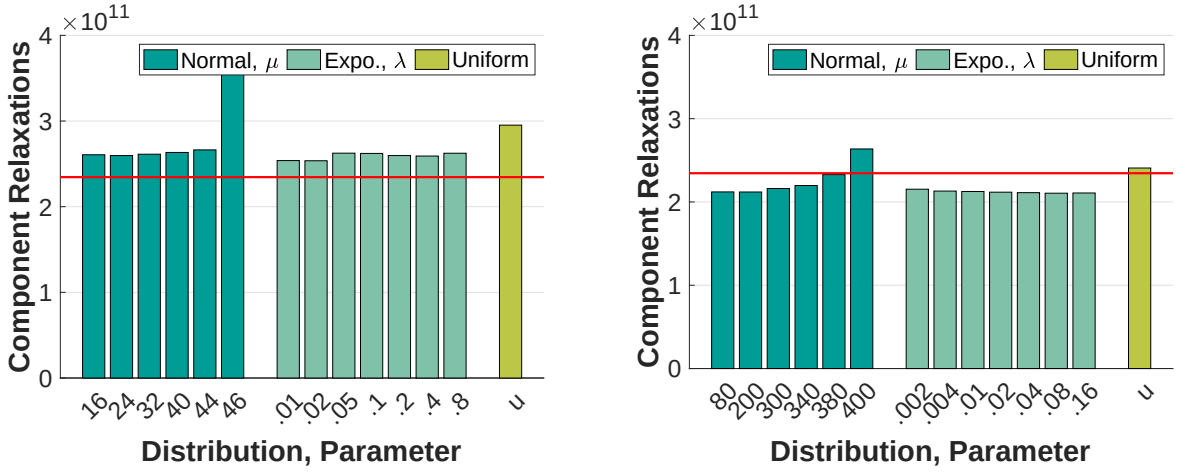


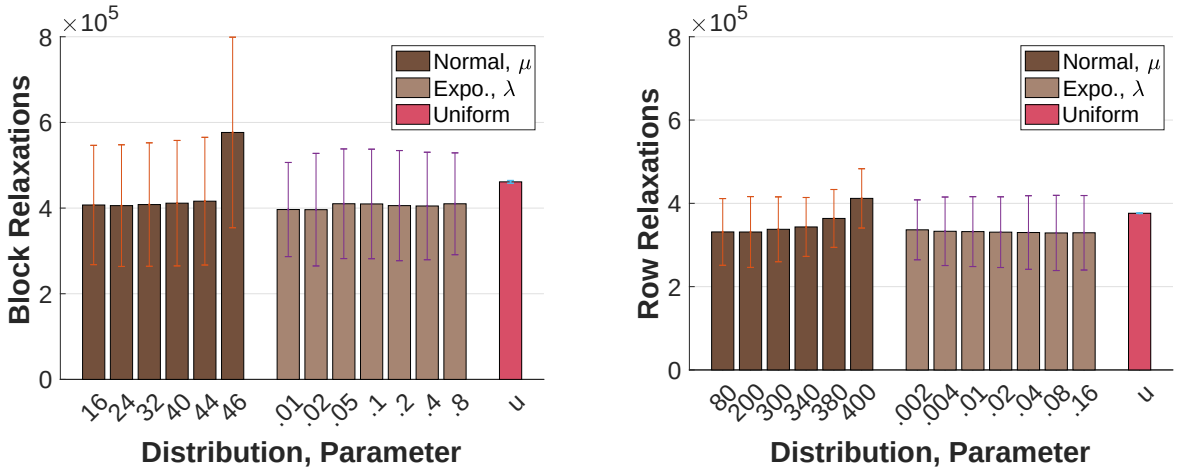
Figure 13: Change in residual throughout the calculation, for each distribution.



(a) Block-based implementation

(b) Row-based implementation

Figure 14: The total number of all the grid-component relaxations until convergence, for different probability distributions and parameters. The red lines refer to the number of component relaxations for serial Gauss-Seidel.



(a) Block-based implementation

(b) Row-based implementation

Figure 15: The average number of (a) block and (b) row relaxations required to converge for different probability distributions and parameters for the two implementations. The vertical lines in each bar show the standard deviation of the number of row relaxations among all rows.

are a must—because ranking and sorting will be performed within each node independently.

Complementing the convergence comparisons of BBI and RBI from Fig. 14, Fig. 15 demonstrates (as vertical lines in each bar) a greater variability in how often each block in BBI may be relaxed compared with each row relaxation in RBI. This metric bears significance for the non-uniform distributions since they may "neglect" certain components to relax often enough to hinder convergence, as has been shown earlier in Section 5, and thereby making a proof of convergence more difficult.

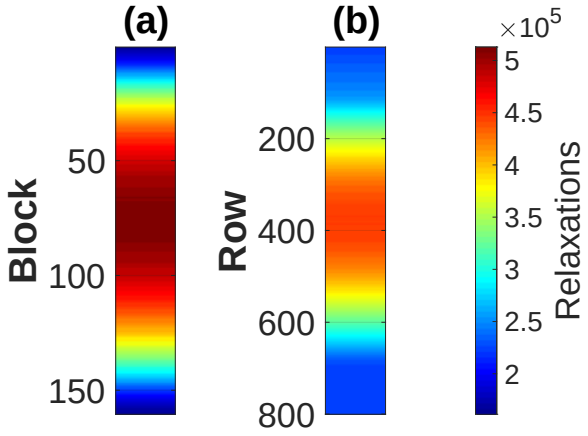


Figure 16: The number of block (a) or row (b) relaxations required to converge with *good* normal distribution parameters.

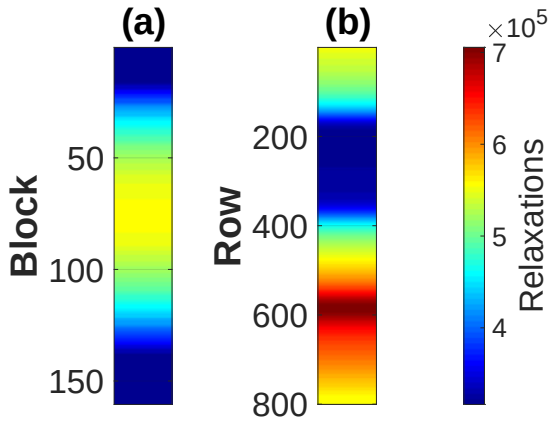
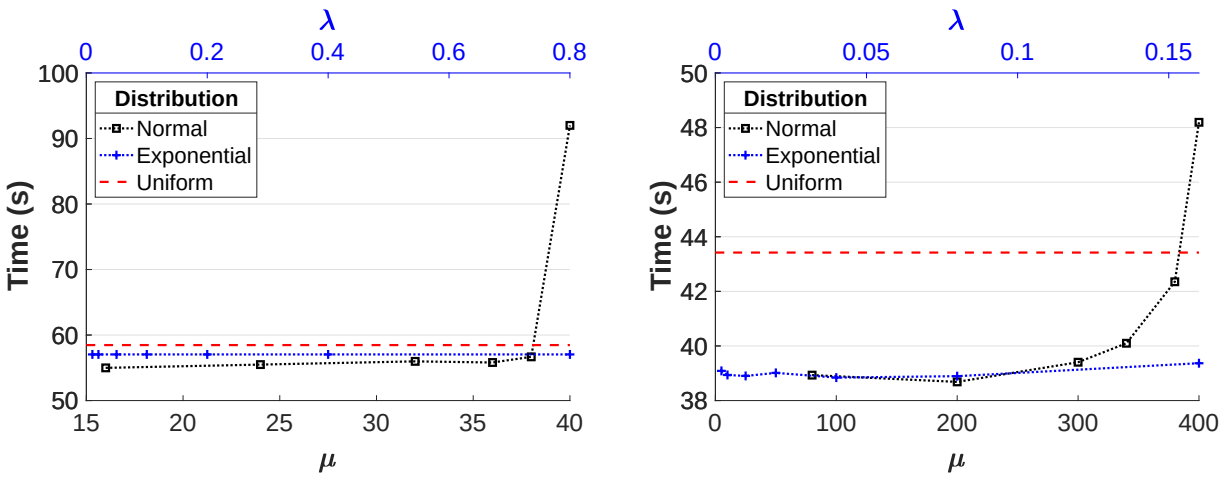


Figure 17: The number of block (a) or row (b) relaxations required to converge with *poor* normal distribution parameters.

Figures 16 and 17 compare BBI and RBI as to which parts of the problem grid are relaxed more times when *good* or *poor* μ is used, respectively. For the former, Fig. 16 shows not only that both implementations emphasize the relaxation of the middle rows, away from the fixed top and bottom boundaries, but also that the RBI places greater emphasis on the rows near the top and bottom boundaries, and less emphasis on the middle rows, compared to the BBI. In particular, about 15% of component selections result in a boundary-crossing event in the row based implementation, which provides for relaxing all the rows more uniformly. With poor distribution parameters, Fig. 17 shows a different behavior of the RBI from the one in Fig. 16. Now, the RBI relaxes boundary rows more frequently than it does so for the innermost rows. In particular, some of the inner rows are now relaxed about as many times as for good μ but the



(a) Block-based implementation

(b) Row-based implementation

Figure 18: Wahab calculation times for each implementation and all three distributions. Note the λ-labeled axis pertains to the exponential distribution trajectory while the μ-labeled axis refers to the normal distribution trajectory.

boundary rows are relaxed more frequently leading to an overall higher number of iterations to converge. Generally, the RBI permits more frequent relaxation of boundary rows, compared with the BBI. Note that the frequency of boundary-row relaxation stays low for BBI given either value of μ (cf. Fig. 16a and Fig. 17a). Such a beneficial behavior of RBI is expressed in line 13 of Algorithm 4, in which the `nextr` function directs a thread to or from a boundary row according to the shortest distance (line 11) as determined by Eq. (7).

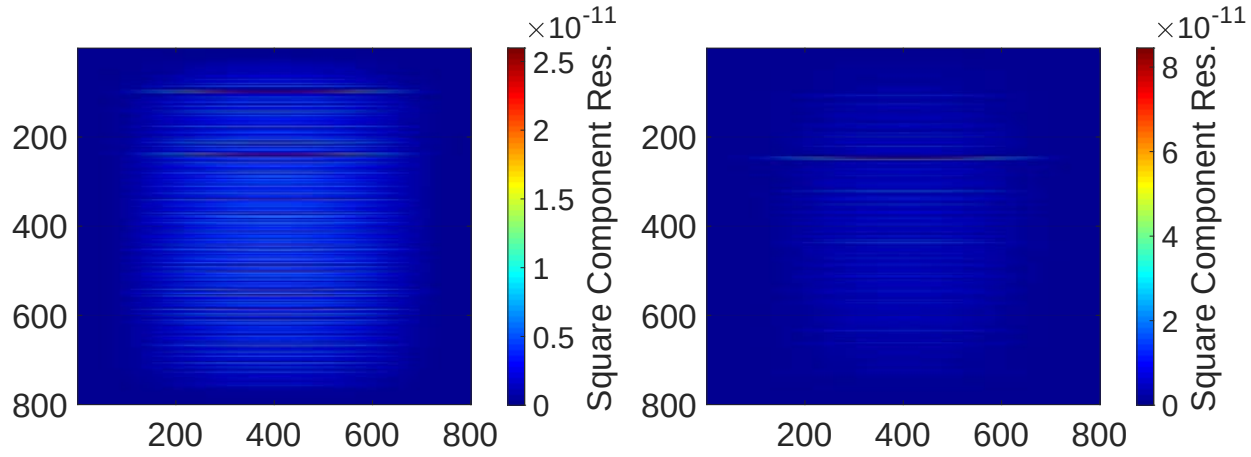
Figure 18 shows that RBI decreases calculation time (Fig. 18b), compared with BBI(Fig. 18a) for all the distributions on the Wahab cluster. Furthermore, a 10% convergence-time reduction is observed for the row-based implementation using normal and exponential distributions with good parameter choices, as compared to a uniform distribution. Figure 18b shows a gradual increase in calculation time for increasing values of μ beyond 200, similar to the gradual increase in numbers of relaxations seen in Figs. 14b and 15b. For BBI on the Wahab platform (Fig. 18a), the results show a jump in calculation time when the normal distribution is used, which is also observed on Rulfo (cf. Fig. 3b) albeit at a larger μ value of 46. On Wahab, the BBI threshold μ is 40, which suggests that, for the normal distribution shared-memory implementation, good parameter selection is platform-dependent, as expected. In particular, having more threads results in smaller size blocks, which may mitigate poor μ selection in the BBI.

In addition to the performance benefit seen with the row-based implementation, Figs. 19 and 20 illustrate that the RBI produces a solution with the residual values more uniformly dispersed among all components. For each implementation, the plots display the two runs with the smallest and largest maximum component residuals, out of a set of ten runs that use the exponential distribution with $\lambda = 0.05$ for BBI and $\lambda = 0.01$ for RBI. The BBI gives a mean maximum component residual of 4.3e-11, with a standard deviation of 2.2e-11, while the row-based implementation gives a mean of 1.0e-11 and a standard deviation of 1.6e-12. Note that the largest maximum component residual produced by the RBI, as seen in Fig. 20b, is about half the size of the smallest component residual produced by the BBI, as seen in Fig. 19a. Observe also that the variations between runs are less for RBI than they are for BBI.

6 Summary and Future Work

This paper develops and tests a novel implementation of a randomized asynchronous iterative solver that uses non-uniform distributions. Complementing a traditional approach of block-row updates, this implementation blends aspects of different solvers and relies on a finer granularity (row-based) of grid component updates. As a result, the row-based implementation (RBI) improves on the block-based one in multiple aspects: solution quality, the number of iterations required for convergence, and the calculation time. The RBI also supports a wider range of parameters that yield fast convergence for the normal distribution.

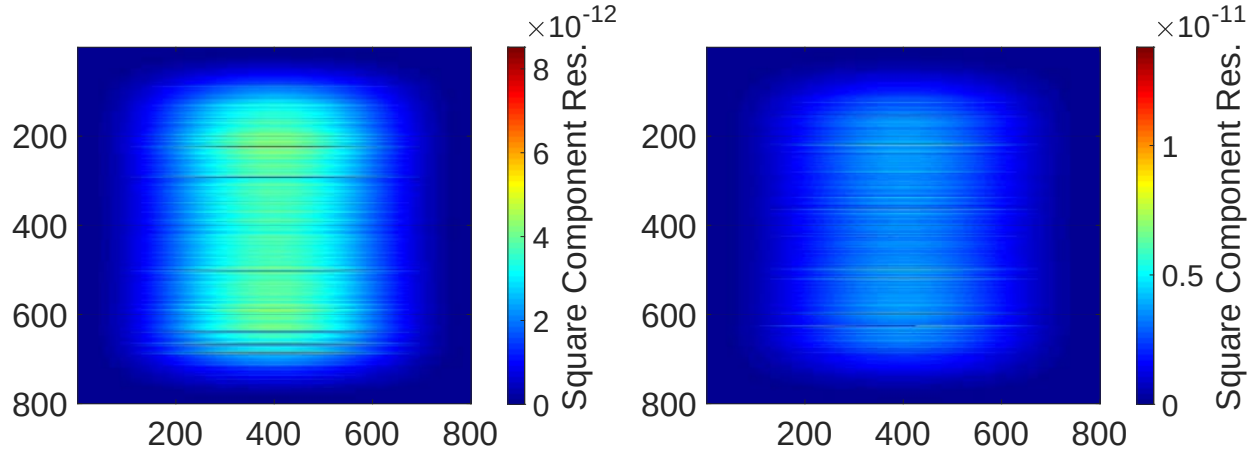
For the two asynchronous randomized solver implementations, block-based and novel row-based, this paper demonstrates a benefit of using a non-uniform distribution in prioritizing component updates. Both BBI and RBI



(a) Smallest

(b) Largest

Figure 19: BBI solution component residual values from the runs with the smallest and largest maximum component residuals, for exponential distribution, $\lambda = 0.05$



(a) Smallest

(b) Largest

Figure 20: RBI solution component residual values from the runs with the smallest and largest maximum components residuals, for exponential distribution, $\lambda = 0.01$

with non-uniform distributions converge 10% faster than their counterparts with the uniform distribution do so. The row-based implementation may also converge with 10% fewer iterations than serial Gauss-Seidel, which is not observed for the block-based implementation.

A further investigation into the ranking periodicity and technique for sorting the residuals is warranted in the scope of studying the overall efficiency of future randomized asynchronous linear solver variants. Continuing to optimize the implementations will improve their ability to be used either in a standalone capacity or as part of another solution scheme, such as preconditioners for Krylov subspace methods or as smoothers in multigrid methods. Additionally, testing on a more diverse problem set may reveal further benefits to the solver by dynamically focusing on the components that are furthest from convergence.

Acknowledgments

This work was supported in part by the U.S. Department of Energy (DOE) Office of Science, Office of Basic Energy Sciences, Computational Chemical Sciences (CCS) Research Program under work proposal number AL-18-380-057 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 and through the ILIR/IAR program at the Naval Surface Warfare Center, Dahlgren Division and by the National Science Foundation under grant CNS-1828593.

References

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

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

[3] 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.

[4] Marc Baboulin, Xiaoye S Li, and François-Henry Rouet. Using random butterfly transformations to avoid pivoting in sparse direct methods. In *International Conference on High Performance Computing for Computational Science*, pages 135–144. Springer, 2014.

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

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

[7] Evan Coleman, Erik Jensen, and Masha Sosonkina. Enhancing Asynchronous Linear Solvers through Randomization. In *Proceedings of the 2019 Spring Simulation Multiconference (Submitted)*. Society for Computer Simulation International, 2019.

[8] 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.

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

[10] Michael Griebel and Peter Oswald. Greedy and randomized versions of the multiplicative schwarz method. *Linear Algebra and its Applications*, 437(7):1596–1610, 2012.

[11] Aditya Kashi, Syam Vangara, and Sivakumaran Nadarajah. Asynchronous fine-grain parallel smoothers for computational fluid dynamics. In *2018 Fluid Dynamics Conference*, page 3558, 2018.

[12] Dennis Leventhal and Adrian S Lewis. Randomized methods for linear constraints: convergence rates and conditioning. *Mathematics of Operations Research*, 35(3):641–654, 2010.

[13] Tony Lindeberg. Scale-space for discrete signals. *IEEE transactions on pattern analysis and machine intelligence*, 12(3):234–254, 1990.

[14] Julie Nutini, Mark Schmidt, Issam Laradji, Michael Friedlander, and Hoyt Koepke. Coordinate descent converges faster with the gauss-southwell rule than random selection. In *International Conference on Machine Learning*, pages 1632–1641, 2015.

[15] D Stott Parker. Random butterfly transformations with applications in computational linear algebra. *Technical Report CSD-950023*, 1995.

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

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

[18] Richard Vynne Southwell. *Relaxation Methods in Theoretical Physics: A Continuation of the Treatise, Relaxation Methods in Engineering Science*, volume 2. The Clarendon Press, 1946.

[19] John C Strikwerda. A probabilistic analysis of asynchronous iteration. *Linear algebra and its applications*, 349(1-3):125–154, 2002.

[20] John C Strikwerda. *Finite difference schemes and partial differential equations*, volume 88. Siam, 2004.

[21] Thomas Strohmer and Roman Vershynin. A randomized kaczmarz algorithm with exponential convergence. *Journal of Fourier Analysis and Applications*, 15(2):262, 2009.

[22] Jordi Wolfson-Pou and Edmond Chow. Distributed Southwell: an iterative method with low communication costs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 48. ACM, 2017.