ENHANCING ASYNCHRONOUS LINEAR SOLVERS THROUGH RANDOMIZATION

Evan Coleman

Naval Surface Warfare Center
Dahlgren Division
17320 Dahlgren Rd
Dahlgren, VA, USA
ecole028@odu.edu

Erik J. Jensen
Masha Sosonkina

Department of Modeling, Simulation
and Visualization Engineering
Old Dominion University
5115 Hampton Blvd, Norfolk, VA, USA
{ejens003,msosonki}@odu.edu

ABSTRACT

Asynchronous iterative methods present a mechanism to improve the performance of parallel algorithms for highly parallel computational platforms by removing the overhead associated with synchronization among computing elements. This paper considers a class of asynchronous iterative linear system solvers that employ randomization to determine the component update orders, and specifically focusing on the effects of non-uniform distributions. Results show that using distributions favoring the selection of components with a larger residual may lead to a faster convergence than that when selecting uniformly. In particular, in the best case of parameter choices, average times for the normal and exponential distributions were, respectively, 13.3% and 17.3% better than the performance with a uniform distribution.

Keywords: asynchronous iteration, linear system solver, randomized linear algebra, Southwell

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. Asynchronous iteration is often applied to the parallel solution of fixed point problems, whereby a fixed point iteration, $x^{(k+1)} = G(x^{(k)})$, is updated in an asynchronous manner. This class of problems has been used in a wide variety of applications including: the solution of linear systems (Recht, Re, Wright, and Niu 2011), the preconditioning of linear solvers (Chow and Patel 2015), optimization (Srivastava and Nedic 2011), and techniques for solving partial differential equations (Magoules, Szyld, and Venet 2015), among many others.

Asynchronous linear 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 (Avron, Druinsky, and Gupta 2015). 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. For example, possible use cases include using the asynchronous linear solver as a preconditioner to a traditional Krylov subspace solver, to solve systems that only require lower accuracy solutions (e.g. big data, machine learning, etc), or else as a smoother to multigrid methods. One approach to potentially improving the performance of asynchronous linear solvers is to have each processor select randomly the (block of) components it updates next, as opposed fixing *a priori* an update order. This approach has been studied previously for the case where the random selection is done uniformly (Strikwerda

2002, Avron, Druinsky, and Gupta 2015). The main contribution of this work is to investigate the potential performance of randomly selecting the next component to update using a non-uniform distribution. This is motivated in part by weighted stationary solvers, such as the Southwell iteration, which are typically able to converge in fewer iterations than traditional Jacobi or Gauss-Seidel relaxation schemes.

1.1 Related Work

The more recent studies dedicated to this field tend to be related to the goal of removing the synchronization delay from high performance computing clusters that are comprised of heterogeneous components (e.g. systems that make use of co-processors/accelerators). Examples of this work include in-depth analysis from the perspective of utilizing a system with a co-processor (Avron, Druinsky, and Gupta 2015). Randomized linear algebra routines have found use in a variety of different area including transforming linear systems to aid in the performance of direct solvers (Parker 1995, Baboulin, Li, and Rouet 2014), improving the performance of hybrid CPU-GPU preconditioners (Jamal, Baboulin, Khabou, and Sosonkina 2016), and in improving the performance of asynchronous linear solvers (Avron, Druinsky, and Gupta 2015), among many others. Randomized linear relaxation based solvers have been studied in the past (Strikwerda 2002), and convergence bounds in the case of uniform selection of which component to update have been developed more recently (Avron, Druinsky, and Gupta 2015). These convergence bounds were based around work done for randomized Gauss-Seidel solvers (Leventhal and Lewis 2010, Griebel and Oswald 2012). Additionally, randomized optimization routines have been utilized as well, e.g., (Srivastava and Nedic 2011), and these methods utilize many of the same principles and analysis techniques as randomized linear solvers.

The structure of the rest of the paper is as follows: Section 2 overviews asynchronous iterative methods while Section 3 discusses their randomized counterparts and proposes variations of the randomization procedure, Section 4 provides numerical results, and finally Section 5 concludes.

2 OVERVIEW OF ASYNCHRONOUS COMPUTATION

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 (Frommer and Szyld 2000). 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 in Algorithm 1. If each processors (P_l) waits for the other processors to finish each

Algorithm 1: General Computational Model

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

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, the components associated with the block B_p will be updated. This results in a vector, $x = (x_1^{s_1(k)}, x_2^{s_2(k)}, \ldots, x_m^{s_m(k)})$ where $s_l(k)$ indicates how many times component l has been updated, and k is a global iteration counter. A set of indices I^k contains the components that were

updated on the kth iteration. Given these definitions, the three following conditions provide a framework for asynchronous computation:

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

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

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

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

where the individual functions $G_i(\vec{x})$ use the latest updates available, is called an asynchronous iteration.

Relaxation methods are typically used to solve linear systems of the form $Ax = b$ and can be expressed as fixed point iterations of the form

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

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.

3 RANDOMIZED LINEAR SOLVERS

Randomized asynchronous linear solvers (Avron, Druinsky, and Gupta 2015, Leventhal and Lewis 2010, Griebel and Oswald 2012) 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 shared memory and then randomly draws the next component to update from the list

of all available components. To the best of our knowledge, in the randomized asynchronous linear solvers proposed to date this random selection is always done using uniform random number generation. A major point of exploration here is to investigate the feasibility of using non-uniform distributions in this selection.

3.1 Improved Randomized Linear Solvers

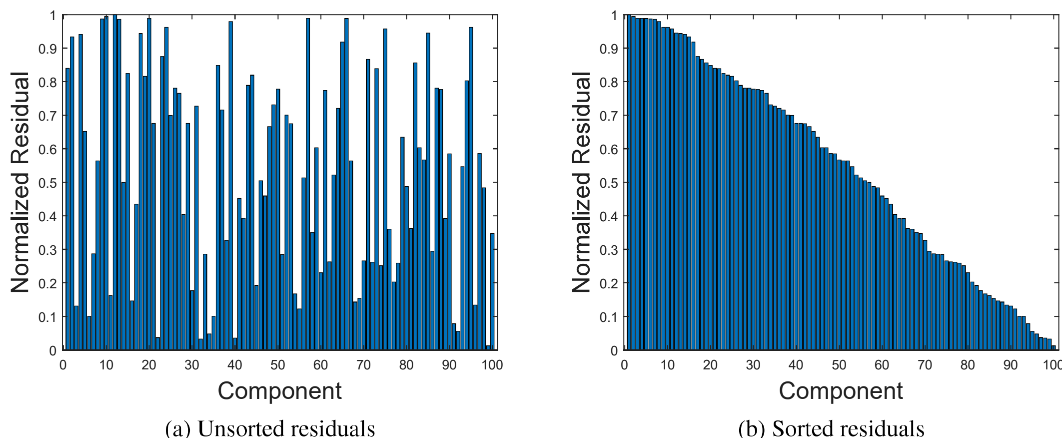
The Southwell algorithm (Southwell 1946) works similar to Jacobi method described in Section 2 by relaxing a single equation at a time, but chooses the equation with the largest local residual. This difference allows the Southwell algorithm to often converge in fewer iterations than Jacobi, but raises the expense of computing an update since the local residuals need to be stored and ranked at each iteration. For example, after a given iteration, the Southwell algorithm will choose the component that contributes the most to the global residual to update; in order to do this, the algorithm must rank the residuals from largest to smallest.

Using the insight from the Southwell algorithm and the success found in the parallel Southwell implementations (Wolfson-Pou and Chow 2016, Wolfson-Pou and Chow 2017), the idea behind the randomized linear solvers considered here is for each processor to select the next component it is responsible 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 by Algorithm 2, which is the same as the one for the randomized asynchronous Jacobi presented in (Avron, Druinsky, and Gupta 2015), where a uniform distribution over $\{1,2,\ldots,n\}$ is studied. The key difference of the present work is that here non-uniform distributions in Line 3 of Algorithm 2 are investigated. As motivating example, consider the two dimensional finite-difference discretization of the Laplacian $-\Delta u = f$ with Dirichlet boundary conditions taken over a 10×10 grid. This results in 100 different components to update. The initial residuals for each component are shown in Fig. 1 both unsorted (left) and sorted from largest to smallest (right). From Fig. 1b, it may be clearly seen that using a non-uniform distribution that favors specific parts of the "slope" is effectively possible. And prioritizing the updates of the components contributing the most to the global residual may be beneficial to convergence as was shown for the Southwell method.

Algorithm 2: Generic Randomized Linear Solver

1 **for** *each processing element* P_l **do**
2 **for** $i = 1,2,\ldots$ *until convergence* **do**
3   Pick $j \in \{1,2,\ldots,n\}$ using 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

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. This reduction will have to be shown to be significant enough to offset the extra computational and communication base cost associated with storing and ranking the local residuals. 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. Since ranking and sorting local residuals can be expensive, the periodicity with which this is done contributes to the overall efficiency of the algorithm. Previously in (Jensen, Coleman, and Sosonkina 2018), the authors have studied a balance between computational effort spent performing relaxations compared with other algorithmic operations and communications in asynchronous methods.



(a) Unsorted residuals (b) Sorted residuals

Figure 1: Initial component residuals ($r_i/\max(r_i)$).

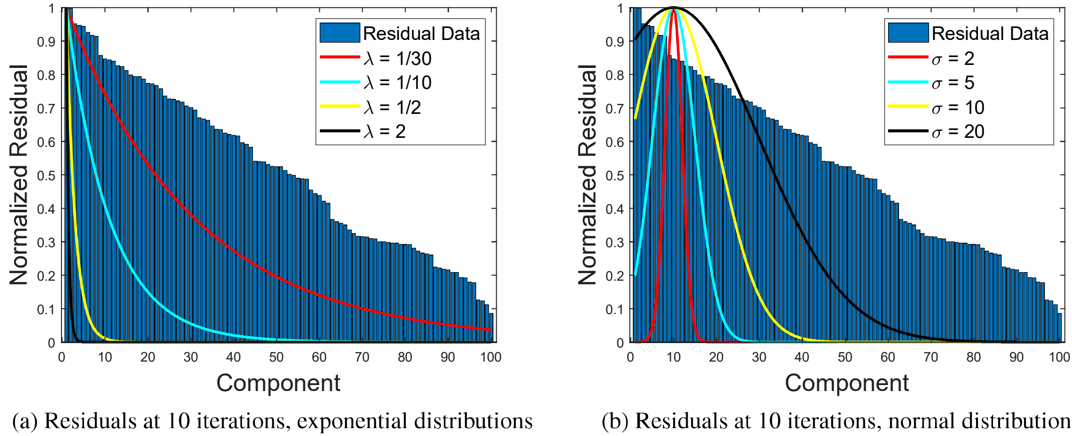
4 EXPERIMENTS WITH DIFFERENT RANDOMIZED SOLVERS AND DISTRIBUTIONS

Two series of numerical experiments are presented in this section. The first focuses on investigating the potential performance of different randomized asynchronous linear solvers through a series of MATLAB® experiments, and the second, presented in Section 4.1, provides a more detailed analysis of shared memory implementations of several different randomized asynchronous linear solvers.

In Fig. 2, several different distributions are shown normalized against the sorted list of residuals from the solution of Laplacian. Figure 2a and Fig. 2b show the same distributions against the list of residuals after each component has been updated 10 times. Both plots are normalized against the largest single residual. A variety of other iteration numbers were investigated, however behavior of the normalized residuals is very similar and therefore only a single instance is shown.

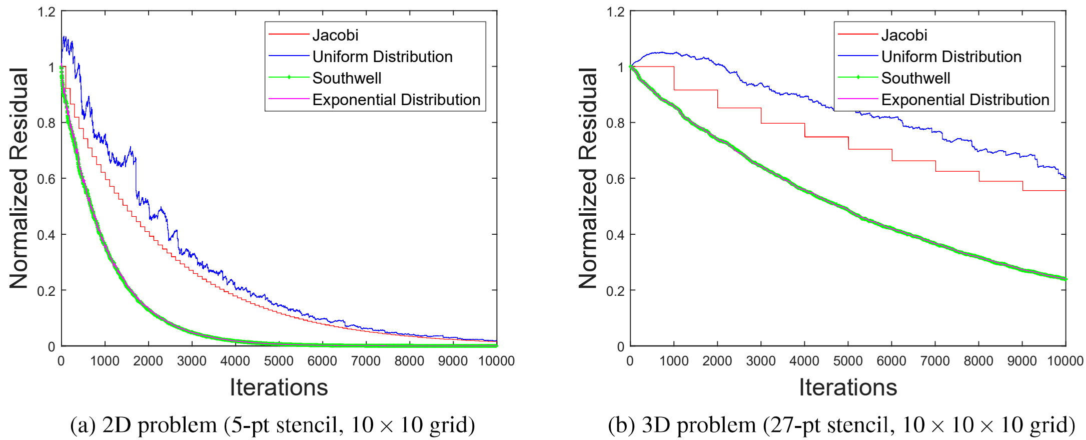
Throughout this section the experiments are performed using MATLAB®. The probability density function (PDF) for the exponential distribution uses the parameter λ which ranges from $\frac{1}{30}$ to 2. For the normal distributions, the PDF is defined by a mean, μ, and a standard deviation, σ. In each of the distributions shown, μ is set to 10, and σ ranges from 2 to 20. Regardless of the distribution chosen to weight the asynchronous linear solver, tuning the parameters that control the distribution to the specific problem at hand can be used to improve performance. For instance, in a problem where most components have a very similar contribution to the residual, using a moderately flat distribution may be a way to accelerate convergence. For example, if the gap between the largest and smallest contributors shown in Fig. 2a or Fig. 2b were very small, there would be less benefit to weighting the distribution used to randomly select the next component in a non-uniform manner.

As an example of potential convergence rates, Fig. 3 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) Residuals at 10 iterations, exponential distributions (b) Residuals at 10 iterations, normal distributions

Figure 2: Sorted residuals ($r_i/\max(r_i)$) with exponential and normal distributions for reference.



(a) 2D problem (5-pt stencil, 10×10 grid) (b) 3D problem (27-pt stencil, $10 \times 10 \times 10$ grid)

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

4.1 Larger Scale Experiments Using Shared-Memory Platforms

This series of experiments details an implementation of the randomized asynchronous linear solver described earlier written in C++ using OpenMP® conducted on the Rulfo system at Old Dominion University. The code used standard C++ routines for sorting residuals and generating random numbers, with the default parameters and the built-in distributions. 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 was utilized, with one core reserved for interfacing with the operating system, resulting in 63 computational threads.

The same test problem, the finite-difference discretization of the Laplacian, used here but over a larger, 800×800, grid. Instead of selecting a single component in the grid, a block of components is selected and Gauss-Seidel sweeps are performed on the components in the block. 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 as sketched in Fig. 4.

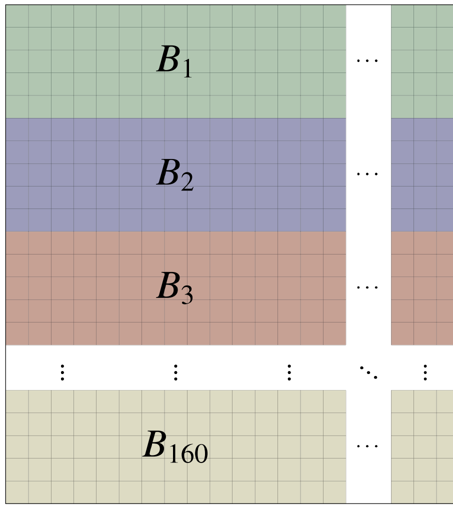
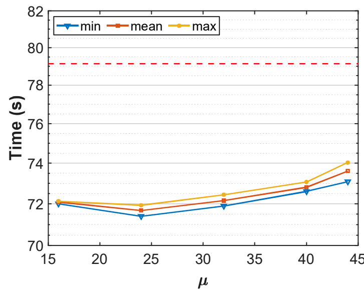


Figure 4: Block assignment used in the grid for the example problem.

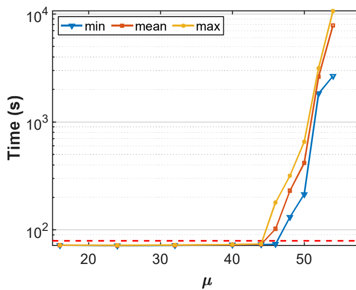
Algorithm 3: Block Variant of Randomized Linear Solver

input: ranking period τ, number of block-rows $\hat{n}$, number of block relaxations m

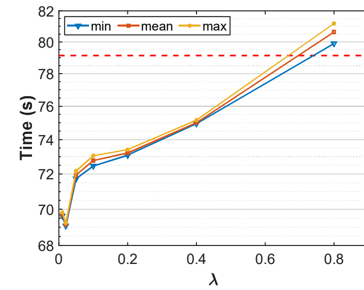
1. $c_1 = 0, c_2 = 0$
2. **for** *each processing element* P_l **do**
3.   **for** $i = 1, 2, \ldots$ *until convergence* **do**
4.   **if** $(c_1 \mod \tau)$ is 0 **then**
5.   Master: rank and sort residuals
6.   **if** $(c_2 \mod m)$ is 0 **then**
7.   Pick $j \in \{1, 2, \ldots, \hat{n}\}$ using probability distribution
8.   Read corresponding entries of A, x, b
9.   Perform 1 relaxation for block B_j
10.   Update the data for B_j
11.   Master: $c_1 = c_1 + 1$
12. $c_2 = c_2 + 1$



(a) Normal distribution, small μ



(b) Normal distribution, all μ



(c) Exponential distribution

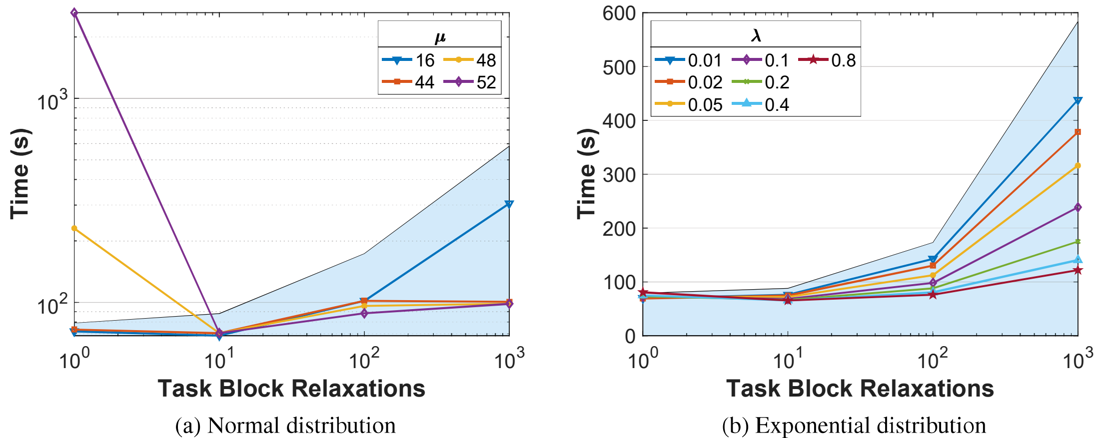
Figure 5: Comparisons of normal, exponential, and uniform distributions for runs with one block-relaxation per thread task. The dashed line references calculation time with uniform distribution. In (b), μ ranges from 16 to 54. In (c), λ ranges from 0.01 to 0.8.

In the task-based asynchronous solver, a thread chooses a block 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 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. 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 to update. 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 will have to be developed in the future.

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.



(a) Normal distribution

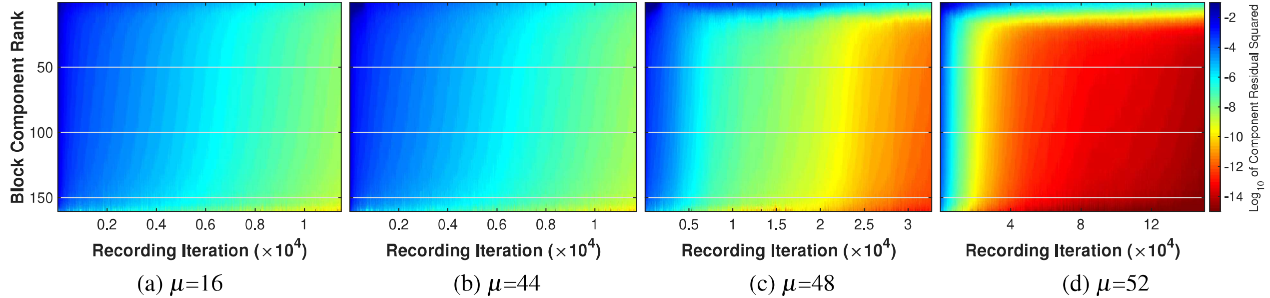
(b) Exponential distribution

Figure 6: Comparisons of normal, exponential, and uniform distributions for runs using one to thousand block-relaxations per thread task. The shaded blue area indicates faster performance than that with the uniform distribution.

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, that is blocks with lower ranks and higher residuals versus blocks with higher ranks and lower residuals. Finally, the *exponential* distribution, as a best-case scenario, is used to define the maximum performance gain for this problem, compared to uniform sampling. When a thread selects a block, it may perform one or more relaxations to update the block, depending on the application configuration. Figure 5 shows results for the three distributions, for one relaxation per block-task. For μ less than or equal to 44, Fig. 5a shows calculation times are relatively small, but when μ increases to 46 and beyond, calculation times vary and tend to increase, as seen in Fig. 5b. Note that, for normal distribution calculations in this work, σ is always eight which helps keep an emphasis on selecting components near the mean. For exponential distribution (Fig. 5c), performance degradation is gradual as λ increases. Due to the nature of the exponential distribution the mode is always zero regardless of λ. With either distribution, appropriate parameters give performance faster than the selection with the uniform distribution does.

For the normal and exponential distributions, the effects of varying the number of relaxations per block-task are shown in Fig. 6. Figure 6a demonstrates that increasing the number of relaxations has some capacity to compensate for inappropriately large values of μ. When a large μ is used to select ranked blocks, selecting a low-ranked block with relatively high residual is uncommon. In the case that a low-ranked block is selected, only one relaxation may not be adequate to reduce the component residual such that the rank for that block is increased enough to increase the probability of future selection. Hence, performing more relaxations on

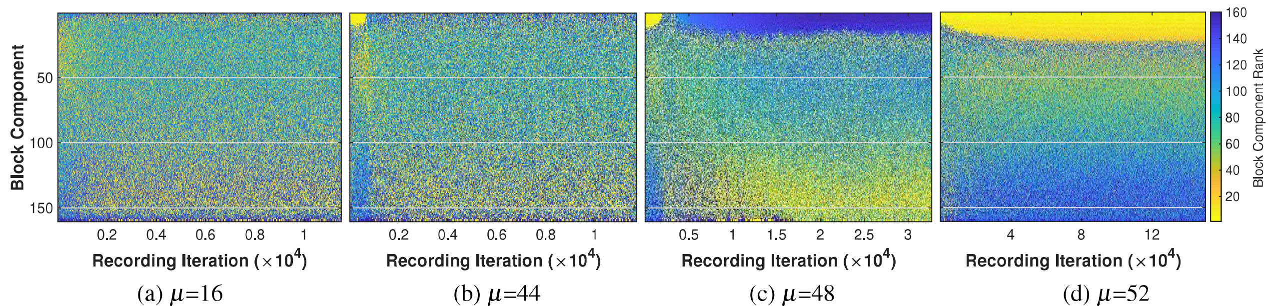
a low-ranked block increases the shuffling of the block rankings. However, if too many relaxations are performed performance can degrade; note the case of $\mu = 16$ and 1000 iterations per selection in Fig. 6a. Average best times for normal distribution with μ=16, one block-relaxation, exponential λ=0.8 and ten relaxations, and uniform with one relaxation were 68.60, 65.42, and 79.13 seconds, respectively.



(a) μ=16 (b) μ=44 (c) μ=48 (d) μ=52

Figure 7: Progression of block component residuals throughout the calculation when normal distribution is used with one block relaxation. Residuals are recorded every 50 iterations.

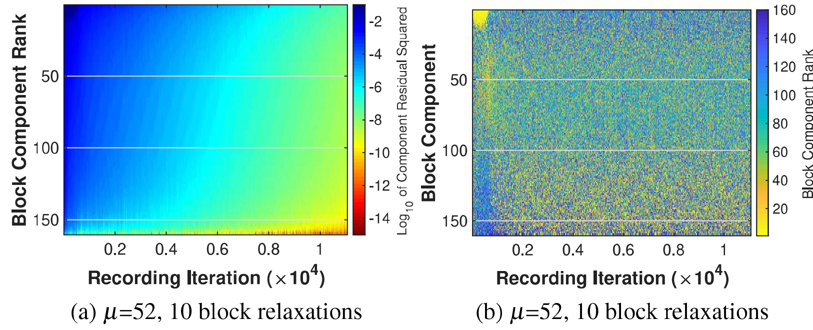
Figures 7 and 8 provide a more detailed explanation for performance differences based on μ selection. In particular, Figs. 7a and 7b depict that the ordered component residual values for μ equal to 16 and 44 and are nearly indistinguishable. However, when μ increases to 48 (Fig. 7c) and then again to 52 (Fig. 7d) residuals of the lowest-ranked blocks decrease slowly while the residuals of all other blocks are much smaller in comparison. Figures 8a and 8b show that block rankings are well-shuffled for μ equal 16 and 44, but in Figs. 8c and 8d block rankings become stratified for μ greater than 48. This stratification indicates that, throughout the calculation, some blocks are updated less frequently than others, which eventually leads to performance degradation. Figure 9b shows that when μ is 52, this stratification can be corrected by increasing the number of block relaxations to 10, leading to the improved performance seen in Fig. 9a. Figure 10 shows good block rank shuffling and balanced component residuals for the minimum and maximum values of λ used in this work and 1 block relaxation per selection.



(a) μ=16 (b) μ=44 (c) μ=48 (d) μ=52

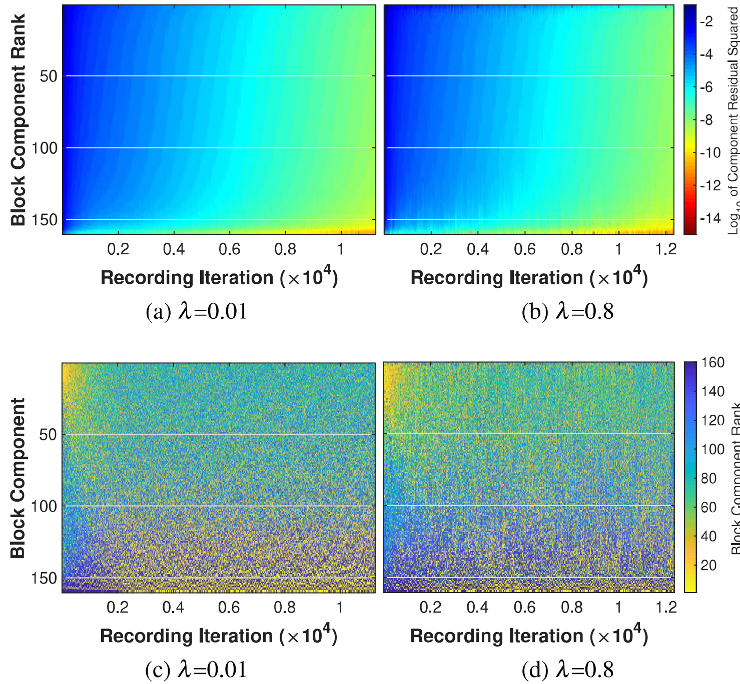
Figure 8: Progression of block component rankings throughout the calculation when normal distribution is used with one block relaxation. Rankings are shown every 50 iterations.

Figure 11a shows global residual behavior at the threshold of μ equal 46, at which runs may differ significantly from the behavior of $\mu = 16$ runs. Figures 11a to 11c show that the performance continues to degrade as μ increases. Figure 11d shows that the parameter λ for the exponential distributions does not have as much an impact on performance as the parameter μ does for the normal distribution runs. Figure 11e, shows the best performance for the parameters tested in each of the distributions with one block relaxation executed. The same outcome as before seen here as well: Calculations with either the normal and exponential distributions outperform the uniform one.



(a) μ=52, 10 block relaxations (b) μ=52, 10 block relaxations

Figure 9: Change in block component residuals for large $\mu = 52$ in normal distribution when 10 block relaxations are performed.



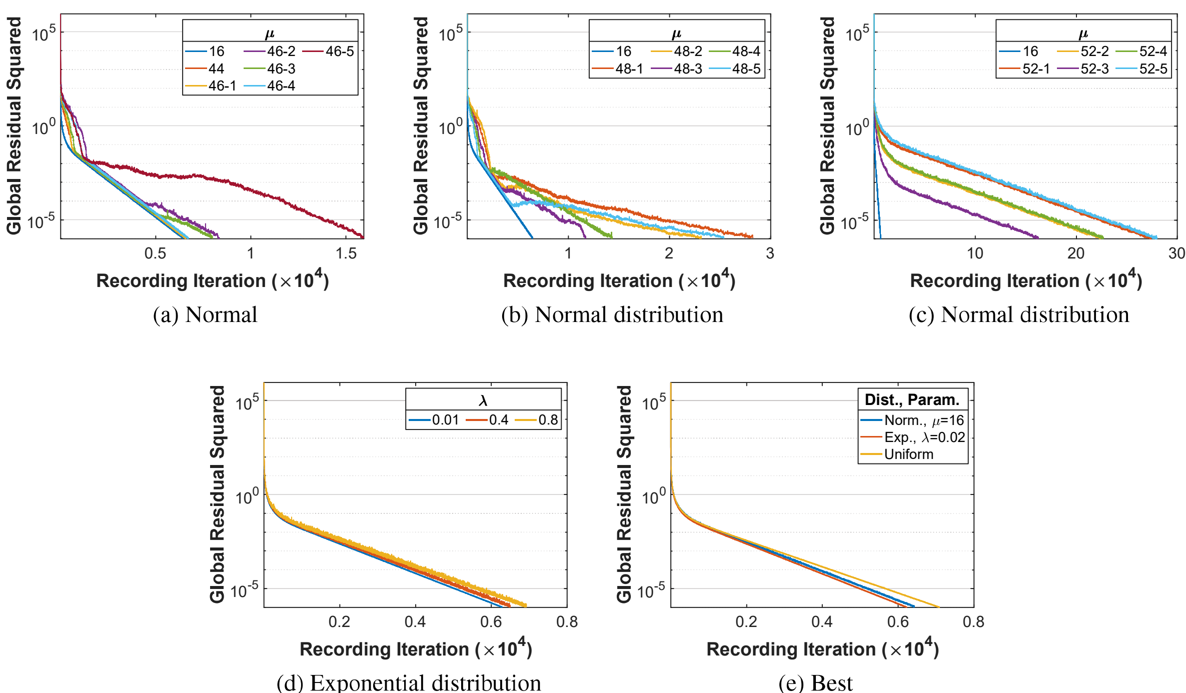
(a) λ=0.01 (b) λ=0.8

(c) λ=0.01 (d) λ=0.8

Figure 10: Block component residual and corresponding ranking progression for the calculations using exponential distribution and 1 block relaxations per selection.

5 SUMMARY & FUTURE WORK

This paper has shown a benefit of using a non-uniform distribution in the selection of the component to update for an asynchronous linear solver. A further investigation into the ranking periodicity and technique for sorting the residuals is warranted in the scope of studying the overall efficiency of the proposed randomized linear solver variant. Future work also includes examining the performance of distributed memory implementations, testing on more diverse problems, considering weighted asynchronous linear solvers as multigrid smoothers, and developing theoretical results that provide bounds on the rate of convergence.



(a) Normal (b) Normal distribution (c) Normal distribution

(d) Exponential distribution (e) Best

Figure 11: Global residual progression for normal and exponential distributions with various distribution parameter values, possibly suffixed with '-1',...,'-5' to enumerate a run for a given value. Note that in Fig. 11e the parameters are optimized for a single relaxation per selection.

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. The authors thank reviewers for their comments that helped improve the paper.

REFERENCES

Avron, H., A. Druinsky, and A. Gupta. 2015. "Revisiting asynchronous linear solvers: Provable convergence rate through randomization". *Journal of the ACM (JACM)* vol. 62 (6), pp. 51.

Baboulin, M., X. S. Li, and F.-H. Rouet. 2014. "Using random butterfly transformations to avoid pivoting in sparse direct methods". In *International Conference on High Performance Computing for Computational Science*, pp. 135–144. Springer.

Chow, E., and A. Patel. 2015. "Fine-grained parallel incomplete LU factorization". *SIAM journal on Scientific Computing* vol. 37 (2), pp. C169–C193.

Frommer, A., and D. B. Szyld. 2000. "On asynchronous iterations". *Journal of computational and applied mathematics* vol. 123 (1), pp. 201–216.

Griebel, M., and P. Oswald. 2012. "Greedy and randomized versions of the multiplicative Schwarz method". *Linear Algebra and its Applications* vol. 437 (7), pp. 1596–1610.

Jamal, A., M. Baboulin, A. Khabou, and M. Sosonkina. 2016. "A hybrid CPU/GPU approach for the parallel algebraic recursive multilevel solver pARMS". In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2016 18th International Symposium on*, pp. 411–416. IEEE.

Jensen, E., E. Coleman, and M. Sosonkina. 2018. "Using Modeling to Improve the Performance of Asynchronous Jacobi". In *Proceedings of the 24th annual International Conference on Parallel and Distributed Processing Techniques and Applications*.

Leventhal, D., and A. S. Lewis. 2010. "Randomized methods for linear constraints: convergence rates and conditioning". *Mathematics of Operations Research* vol. 35 (3), pp. 641–654.

Magoules, F., D. B. Szyld, and C. Venet. 2015. "Asynchronous optimized Schwarz methods with and without overlap". *Numerische Mathematik*, pp. 1–29.

Parker, D. S. 1995. "Random butterfly transformations with applications in computational linear algebra". *Technical Report CSD-950023*.

Recht, B., C. Re, S. Wright, and F. Niu. 2011. "Hogwild: A lock-free approach to parallelizing stochastic gradient descent". In *Advances in neural information processing systems*, pp. 693–701.

Southwell, R. V. 1946. *Relaxation Methods in Theoretical Physics: A Continuation of the Treatise, Relaxation Methods in Engineering Science*, Volume 2. The Clarendon Press.

Srivastava, K., and A. Nedic. 2011. "Distributed asynchronous constrained stochastic optimization". *IEEE Journal of Selected Topics in Signal Processing* vol. 5 (4), pp. 772–790.

Strikwerda, J. C. 2002. "A probabilistic analysis of asynchronous iteration". *Linear algebra and its applications* vol. 349 (1-3), pp. 125–154.

Wolfson-Pou, J., and E. Chow. 2016. "Reducing Communication in Distributed Asynchronous Iterative Methods". *Procedia Computer Science* vol. 80, pp. 1906–1916.

Wolfson-Pou, J., and E. Chow. 2017. "Distributed Southwell: an iterative method with low communication costs". In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 48. ACM.

AUTHOR BIOGRAPHIES

EVAN COLEMAN is a scientist with the Naval Surface Warfare Center, Dahlgren Division. He holds an MS in Mathematics from Syracuse University and is working on a PhD in Modeling and Simulation from Old Dominion University. His email address is ecole028@odu.edu.

ERIK J. JENSEN is a PhD student in Modeling and Simulation at Old Dominion University. His research interests include parallel computing, numerical analysis, and predictive modeling for High Performance Computing. His email address is ejens003@odu.edu.

MASHA SOSONKINA is a Professor of Modeling, Simulation and Visualization Engineering at Old Dominion University. Her research interests include high-performance computing, large-scale simulations, parallel numerical algorithms, and performance analysis. Her email address is msosonki@odu.edu.