Noname manuscript No.
(will be inserted by the editor)

Predictive Modeling of the Performance of Asynchronous Iterative Methods

Erik J. Jensen · Evan Coleman · Masha Sosonkina

the date of receipt and acceptance should be inserted later

Abstract Asynchronous algorithms may increase performance of parallel applications on large-scale HPC platforms due to decreased dependence among processing elements. This work investigates strategies for implementing asynchronous hybrid parallel MPI-OpenMP iterative solvers. Seven different implementations are considered, and results show that striking a balance between communication and computation that balances the number of iterations in each processing element benefits performance and solution quality. A predictive performance model that utilizes kernel density estimation to model the underlying probability density function to the collected data is then developed to optimize execution parameters for a given problem. For the majority of iteration executions, the performance model matches within about 6% of the empirical data. The different hybrid parallel implementations are examined further to find optimal parametric distributions whose parameters can be tuned to the problem at hand. The generalized extreme value distribution was selected based on a combination of quantitative and qualitative comparisons and, for the most of the iteration executions, the model matches the data within about 6.1%. Results from the parametric distribution model are examined along with results of the model on related problems, and possible further extensions to the predictive model are discussed.

Keywords Jacobi method, asynchronous iterative methods, predictive modeling, fixed-point iteration, hybrid parallel MPI OpenMP

1 Introduction

As future high-performance computing (HPC) platforms scale in processing elements, calculations will be increasingly parallelized. At this scale, asynchronous methods may offer performance superior to synchronous methods. Several U.S. Department of Energy reports (Ashby et al., 2010a,b; Dongarra et al., 2014) have cited the need for the development of asynchronous iterative methods to run efficiently on future exascale HPC environments. These methods are increasing in popularity recently due to their ability to be parallelized naturally on modern accelerators, such as GPUs.

These techniques are well suited for fine-grained computation, since fine-grained parallelism allows the computation to be executed either synchronously or asynchronously, and the use of asynchronous algorithms may tolerate latency better in HPC environments.A specific area of interest focuses on techniques that utilize fixed-point iteration, i.e., equations of the form

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

for some vector $x \in D$ and some map $G : D \to D$.

Asynchronous parallel methods avoid the performance cost of synchronizing MPI processes and OpenMP threads that are working together to complete a calculation. Instead of waiting for the slowest processing component to complete a computation, the other processing components may independently continue to compute. Asynchronous systems have some advantages over synchronous systems, in that they may mitigate the effect of typical performance variations between similar computing elements or between stages of computations; or they may harness the ability of a computing element to perform useful work while waiting for information from another computing element.

In (Bethune et al., 2011, 2014), several asynchronous Jacobi method implementations are presented to solve the Laplacian equation in three dimensions, which are parallelized with MPI, SHMEM, or OpenMP (Bethune et al., 2011, 2014). This

Erik J. Jensen · Evan Coleman · Masha Sosonkina
Modeling, Simulation and Visualization Engineering, Old Dominion University, Norfolk, VA

Evan Coleman
Naval Surface Warfare Center, Dahlgren Division, Dahlgren, VA

paper presents several asynchronous *hybrid* MPI–OpenMP implementations that solve a two-dimensional finite-difference discretization of the Laplacian equation using Jacobi method, similar to the work of Bethune *et al*. Because of the ubiquitous use of the Jacobi method in parallel solutions of large problems in many domains of science and engineering, the asynchronous Jacobi method was chosen this study. Note that many of the concepts and ideas expressed in this paper can be easily adapted to more complex algorithms. The data that is collected from each of the hybrid parallel implementations of the asynchronous Jacobi is used in the creation of a predictive model, which is then used to optimize some of the parameters that control the details of the hybrid parallel implementation. The predictive model may be generalized to optimize the performance of asynchronous iterative relaxation methods for any partial differential equation that is discretized over a grid-like domain, and the methodology used to create the model generalizes naturally to any asynchronous iterative method. An initial version of this work, which used a five-point stencil, was presented in (Jensen et al., 2018). Its extensions presented here include the following:

- fitting five-point stencil implementation histogram data to a parametric distribution function,
- a nine-point stencil implementation with the same partial differential equation (PDE),
- development and testing of a nine-point stencil predictive model,
- preliminary development and testing of a generalized predictive model, and
- extended validation testing and results for all predictive models developed in this work.

This paper is organized as follows: related work is provided in Section 2, and an overview of asynchronous iterative methods is presented in Section 3, with introductions to the asynchronous Jacobi method and the problem used in this study in Sections 3.1 and 3.2, respectively. The hybrid parallel implementations are described in Section 4. Performance results from testing five-point stencil implementations are examined in Section 5. Section 6 introduces the predictive model of a chosen five-point implementation, and Section 6.1 discusses the behavior observed from running the model. Section 6.2 validates the five-point model. Section 7 introduces the nine-point stencil, Section 7.1 discusses the use of parametric distributions for modeling implementation operations, and Section 7.2 shows how the model can be further extended. Finally, Section 8 concludes and projects future work.

2 Related Work

The efficacy of asynchronous methods, especially for grid systems, has been demonstrated, and a system for classifying parallel iterative algorithms, based on computational and communication strategies has been proposed (Bahi et al., 2006, 2003). Asynchronous methods have delivered superior performance in solving large sparse linear fixed-point problems (De Jager and Bradley, 2010). Voronin compares three parallel implementations using MPI and OpenMP, with asynchronous threads, and finds that utilizing a "postman" thread within each computational node to perform communications delivers superior performance, compared to the alternative hybrid MPI-OpenMP implementation (Voronin, 2014). Examples of work examining the performance of asynchronous iterative methods include an in-depth analysis from the perspective of utilizing a system with a co-processor (Anzt, 2012; Avron et al., 2015), as well as performance analysis of asynchronous methods (Bethune et al., 2011; Hook and Dingle, 2013; Bethune et al., 2014). In particular, both (Bethune et al., 2011, 2014) focus on low level analysis of the asynchronous Jacobi method, similar to the example problem presented here. Work exploring possibilities for reducing the communication costs in a distributed asynchronous solver has also been performed (Wolfson-Pou and Chow, 2016). While many recent research results for asynchronous iterative methods are focused on implementations that utilize a shared memory architecture, one area of asynchronous iterative methods that has seen significant development using a distributed memory architecture is optimization (e.g., Cheung and Cole (2016); Iutzeler et al. (2013); Hong (2017)). An initial exploration of fault tolerance for stationary iterative linear solvers (Jacobi) is given in (Anzt et al., 2015) and expanded in (Anzt et al., 2016). The general convergence of parallel fixed-point methods has been explored extensively (see, e.g., the survey (Frommer and Szyld, 2000) or (Bertsekas and Tsitsiklis, 1989)).

3 Asynchronous Iterative Methods

In asynchronous calculations, problem components, such as a matrix or vector (block) entry, may proceed without new information from other components during an iteration. This flexibility allows for each computing element (e.g., a single processor, CUDA core, or Xeon Phi core) to act independently from all other computing elements. A theoretical basis for asynchronous computation has been explored in (Frommer and Szyld, 2000), which in turn comes from (Chazan and Miranker, 1969; Baudet, 1978) and (Szyld, 1998), among many others. To keep the model of asynchronous computation as general as possible, consider a function, $G : D \to D$, where D is a domain that represents a product space $D = D_1 \times D_2 \times \cdots \times D_m$. The goal is to find a fixed point of the function G inside of the domain D. To this end, a fixed point iteration is performed such that

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

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

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

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

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

The assumption is made that there is some finite number of processing elements $P_1, P_2, \ldots, P_p$ each of which is assigned to a block $\mathcal{B}$ of components $B_1, B_2, \ldots, B_m$ to update. Note that the number p of processing elements will typically be significantly smaller than the number m of blocks to update. With these assumptions, the computational model can be stated in Algorithm 1 (Frommer and Szyld, 2000):

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 $j \in \mathcal{B}_l$
5 Update x_j in common memory with x_j^{i+1} for all $j \in \mathcal{B}_l$

This computational model has each processing element read all pertinent data from global memory that is accessible by each of the processors, update the pieces of data specific to the component functions that it has been assigned, and update those components in the global memory. Note that the computational model presented in Algorithm 1 allows for either synchronous or asynchronous computation; it only prescribes that an update has to be made as an "atomic" operation (in line 5), i.e., without interleaving of its result. If each processing element P_l is to wait for the other processors to finish each update, then the model describes a parallel synchronous form of computation. On the other hand, if no order is established for P_ls, then an asynchronous form of computation arises.

Furthermore, set a global iteration counter k that increases *every* time any processor reads x from common memory. At the end of the work done by any individual processor P_l the components associated with the block $\mathcal{B}_l$ 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 the function $s(k)$ indicates how many times a specific component has been updated. Finally, a set I^k of individual components is defined such that it contains all the components that were updated on the kth iteration. Given these basic definitions, the three following conditions (along with the model presented in Algorithm 1) provide a working mathematical framework for fine-grained asynchronous computation.

Definition 1 If the following three conditions hold:

1. $s_i(k) \leq k,\ i = 1, \ldots, m$, *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

$$x_i^{k+1} = \begin{cases} x_i^k & i \notin I^{k+1} \\ G_i(x_1^{s_1(k)}, x_2^{s_2(k)}, \ldots, x_m^{s_m(k)}) & i \in I^{k+1} \end{cases}, \tag{5}$$

is called an asynchronous iteration.

This computational model (i.e., Algorithm 1 together with Definition 1) allows for many different implementations of fine-grained iterative methods that are either synchronous or asynchronous, although the three conditions in Definition 1 are unnecessary in the synchronous case.

3.1 Asynchronous Jacobi

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

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

where relaxation methods can be expressed as general fixed-point iterations of the form

$$x^{k+1} = Cx^k + d \ , \tag{7}$$

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 may be used to define the particular problem at hand. Following the methodology put forth in (Bertsekas and Tsitsiklis, 1989), this can be broken down to view a specific row—say, the row i—of the matrix A as

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \ , \tag{8}$$

and this equation can be solved for the component x_i of the solution to give

$$x_i = \frac{-1}{a_{ii}} \left[\sum_{j \neq i} a_{ij}x_j - b_i \right] . \tag{9}$$

This equation can then be computed in an iterative manner in order to give successive updates to the solution vector. In synchronous computing environments, each update to the solution vector component x_i is computed sequentially using the same data for the other components of the solution vector (i.e., the x_j in Eq. (9)). 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. (9) in a block matrix form is more similar to the original form of the iteration expressed in Eq. (7):

$$\begin{aligned} x &= -D^{-1}((L+U)x - b) \\ &= -D^{-1}(L+U)x + D^{-1}b \ , \end{aligned}$$

where D is the diagonal portion of A while L and U are the strictly lower and upper triangular portions of A, respectively. This gives an iteration matrix $C = -D^{-1}(L+U)$. Convergence of asynchronous fixed-point methods of the form in Eq. (7) is determined by the spectral radius of the iteration matrix, C (Chazan and Miranker, 1969) and (Baudet, 1978):

Theorem 1: *For a fixed point iteration of the form given in Eq.* (7) *that adheres to the asynchronous computational model provided by Algorithm 1 and the conditions in the associated definition, if the spectral radius $\rho(|C|)$ of C is less than one, then the iterative method will converge to the fixed-point solution.* The iteration matrix C that is used in the Jacobi relaxation method serves as a worst case for relaxation methods discussed in this work, as also noted in Wolfson-Pou and Chow (2016).

3.2 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 (Smith, 1985). The specific problem under study here is Laplace equation in two dimensions:

$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = b \ , \tag{10}$$

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 (10) may be discretized such that a finite difference operator computes difference quotients over a discretized domain. For example, the two-dimensional discrete Laplace operator

$$\left(\nabla^2 f\right)(x,y) = f(x-1,y) + f(x+1,y) + f(x,y-1) + f(x,y+1) - 4f(x,y) \tag{11}$$

approximates the two-dimensional continuous Laplacian using a five-point stencil (Lindeberg, 1990). From this, a discretized version of the Jacobi algorithm

$$v_{l,m}^{k+1} = \frac{1}{4}(v_{l+1,m}^{k} + v_{l-1,m}^{k} + v_{l,m+1}^{k} + v_{l,m-1}^{k}) \tag{12}$$

may be applied to solve a two-dimensional sparse linear system of equations (Strikwerda, 2004). Indices l, m and k define discrete grid nodes in two dimensions and the iteration number, respectively, for updating the discretized solution vector v. Note that, in the two dimensional discretization of the Laplacian, the spectral radius of the iteration matrix that would be formed by the grid-point equation (Eq. (12)) is less than one. Hence, according to Theorem 1, both the synchronous and asynchronous variants of the Jacobi algorithm will converge. Pseudocode for this algorithm is provided in Algorithm 2. Note that each processor P_l may not be available to compute updates at the same time. This lack of determinism in the

Algorithm 2: Asynchronous Jacobi

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

update order (i.e., the amount of time it will take a processor to perform the Jacobi relaxation for the components that are assigned to it) leads to the asynchronous nature of the algorithm.

4 Hybrid Implementations

All the hybrid parallel implementations discussed here solve a two-dimensional finite-difference discretization of the Laplacian using OpenMP for shared memory parallelization and MPI for distributed-memory parallelization. They all are based on a matrix-free version of the Jacobi method, whereby the values on the grid are computed directly, without their multiplication by the corresponding matrix entries. To complement matrix-free Jacobi algorithm, two arrays U_0 and U_1 store the grid-point values that each thread reads in adjacent iterations.

Generally, the Laplacian is discretized over a rectangular region, which is then divided into a number of approximately equal subregions to distribute to (MPI) processing elements. One more MPI process is dedicated solely to communication and determination of convergence. These subregions are then divided further to assign a block of grid rows to an OpenMP thread. Among the d threads assigned to each subregion, one is assigned the (possible additional) role of master/communicator.

All the implementations developed for this work are asynchronous, according to the computational model presented in Section 3, except for a synchronous variant used for comparisons. One implementation, denoted as SHRD, was developed exclusively for shared memory, which is also for comparisons, while all others are hybrid and tested in a distributed computing environment. The implementations that are provided belong to the following categories:

- Asynchronous Iterations with Asynchronous Communication (AIAC),
- Asynchronous Iterations with Synchronous Communication (AISC), and
- Synchronous Iterations with Synchronous Communication (SISC).

Descriptions of each of these different classifications follow.

Asynchronous Iterations – Asynchronous Communications (AIAC). Two matrices U_0 and U_1 store grid point values that each thread reads, such as, e.g., from U_1 to compute newer values to write to U_0. As the method is asynchronous, each thread independently determines which matrix stores its newer $u^{(t+1)}(i, j)$ values and older $u^{(t)}(i, j)$ values. When a thread copies grid-point values above or below its domain for the computation, OpenMP locks are employed to ensure that data is captured accurately, from a single iteration. Further, locks are used when updating values on boundary rows and subregion halos as well as when copying subregion boundaries. Each thread computes its local residual value every kth iteration, which it contributes to the set of residual values for the subregion. Using an OpenMP atomic operation, a single thread copies the set of subregion residuals, computes a sum, and sends the sum to the master MPI process. Within the AIAC category, there are five variants:

1. Work-Scaling-1 (WS1). The subregion is equally divided into blocks among all the OpenMP threads, and the first thread performs communication with the master MPI process. The first thread also computes the subregion residual and updates the subregion halo values.

2. Work-Scaling-X (WSX), where $X < 1$. The subregion is divided into blocks of rows, such that the communicating thread performs less computational work than the other threads. In particular, the row-block size for the communicating thread is that of the WS1 variant multiplied by X, while all the threads divide the subregion equally.
3. Rotating-Incrementing (RTINC). The subregion is equally divided among all OpenMP threads (as in WS1) but communication with the master MPI process is rotated in a round-robin fashion among the threads. The communicating thread also computes the subregion residual and updates the subregion halo values.
4. Rotating-Maximum-Iterations (RTMAXIT). Otherwise similar to RTINC, this variant chooses the OpenMP thread that has performed the most iterations to communicate.
5. Rotating-Minimum-Residual (RTMNRS). Otherwise similar to RTMAXIT or RTINC, this variant chooses the OpenMP thread that produces currently the smallest local residual to communicate.

Asynchronous Iterations – Synchronous Communications (AISC). In this category, one implementation, the Asynchronous-Direct (ASNCDIR) has been developed. ASNCDIR is similar to the AIAC implementations, but it has one OpenMP thread in each MPI process reserved exclusively for synchronous communication. In ASNCDIR, the communicating threads swap subregion halo values directly, without the master MPI process intermediary.

Synchronous Iterations – Synchronous Communications (SISC). In this category, the Synchronous-Direct (SNCDIR) is an implementation of a synchronous Jacobi algorithm. For SNCDIR, one thread per subregion communicates directly with other communicating threads regarding halo values, while the non-communicating threads are idle. OpenMP locks are unnecessary and removed.

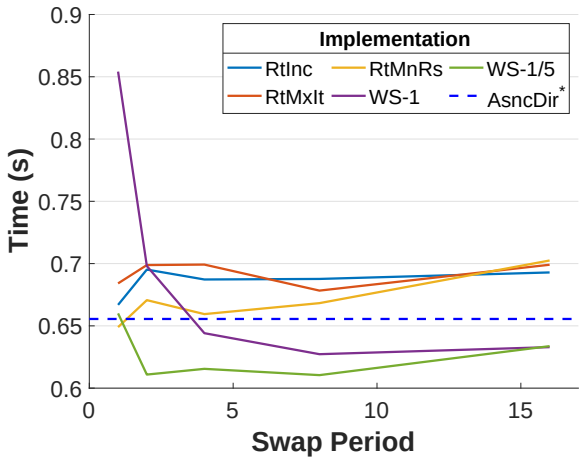
5 Performance of Five-Point Stencil Implementation

Experiments were conducted on the Turing High Performance Computing cluster at Old Dominion University, which contains 190 standard compute nodes, 10 GPU nodes, 10 Intel Xeon Phi Knight's Corner nodes, and 4 high memory nodes, connected with a Fourteen Data Rate (FDR) InfiniBand network. Compute nodes contain between 16 and 32 cores and 128 GB of RAM. Data was collected on nodes with two different hardware configurations, each with two sockets, consisting of either 10 Intel Xeon E5-2670 v2 2.50Ghz cores, or 10 Intel Xeon E5-2660 v2 2.20Ghz cores.

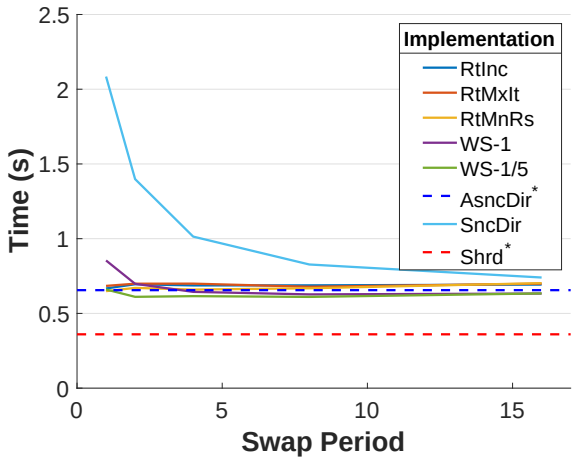
The data collected demonstrates the efficacy of different solving strategies, as a function of problem size, subdomain size, and swap period. All the tests were conducted with the user-defined convergence tolerance of 10^{-4}. The swap period P_{swap} is equal to one, an integer $k \geq 1$, meaning that halo values are exchanged every k iterations only. For a sufficiently small problem size, Fig. 1b shows that the exclusively shared-memory implementation SHRD outperforms the hybrid implementations. However, using larger subdomain sizes, e.g., Figs. 1d and 1f, demonstrates that the hybrid implementations benefit from the additional complexity of distributed parallelization. Figures 1a, 1c and 1e provide a more focused view on the asynchronous implementations. When P_{swap} is sufficiently large, the five different implementations that are of type AIAC perform similarly, with minor variations. This is especially true for the larger problem sizes shown in Figs. 1c and 1e. For these tested problem sizes, the AISC implementation ASNCDIR falls short of the other asynchronous implementations, possibly because the advantage of quick communication and constant halo swaps does not outweigh the loss of a computational thread. The SISC implementation, SNCDIR, cannot match the performance of the asynchronous implementations under these conditions, likely due to the expense of computation lost during the communication phase.

Figure 2 shows the scaling abilities of the implementations. In particular, Figs. 2c and 2d show that iteration rates (i.e., iterations per second) remain fairly constant as a function of problem size, meaning that a significant communication bottleneck is not observed. Increasing the swap period improves the iteration rate, especially for SNCDIR, which suffers from a costly communication phase and fails to match the iteration rates of the asynchronous implementations.

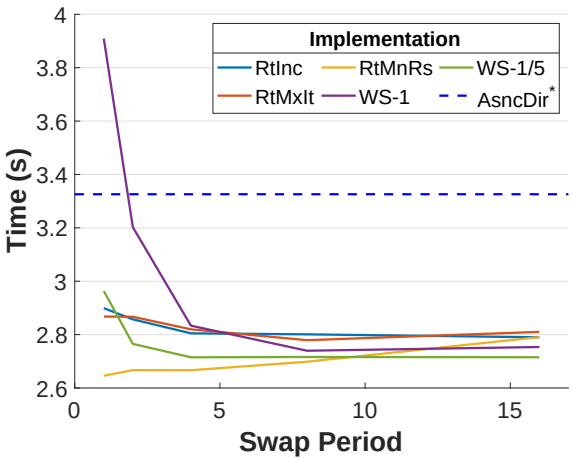
Figures 3 and 4 demonstrate that the shortcomings of the work-scaling implementation WS may be remedies with increasing the swap period P_{swap}. Because a dedicated thread performs communications as well as computations in WS, its iteration rate may be less than that for the non-communicating threads (see, e.g., Fig. 3a depicting results for $P_{swap} = 1$). However, to increase the iteration rate in the communicating thread, its time lost during communication may be balanced with a smaller computational load and larger swap period as shown in Fig. 3b, for example. In addition, increasing the iteration rate of the communicating thread lets it make more progress toward solution by producing a more accurate residual. Figure 4 shows the total residual (square) contribution from each thread for several WS factors and two P_{swap} values. Specifically, Fig. 4a shows that when the work is not scaled down appropriately for the communicating thread, a relatively large portion of the residual is contributed from this thread. However, Fig. 4b suggests that this problem can be mitigated by increasing the swap period.



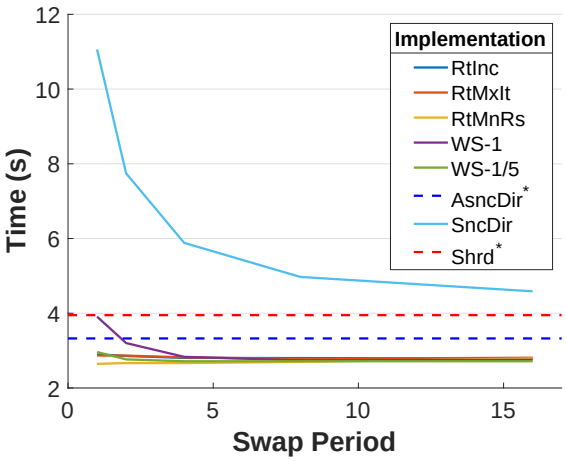
(a) Async. implementations, N=200



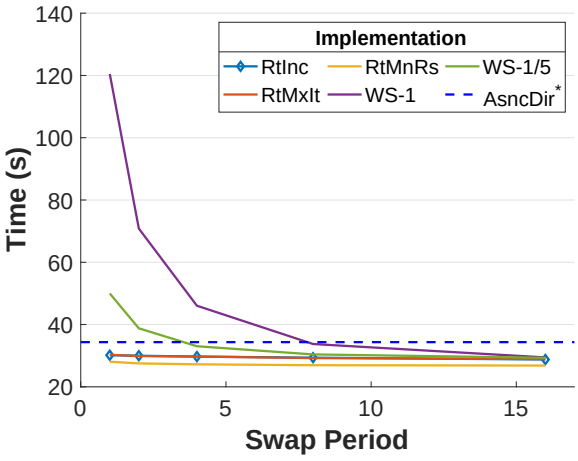
(b) All implementations, N=200



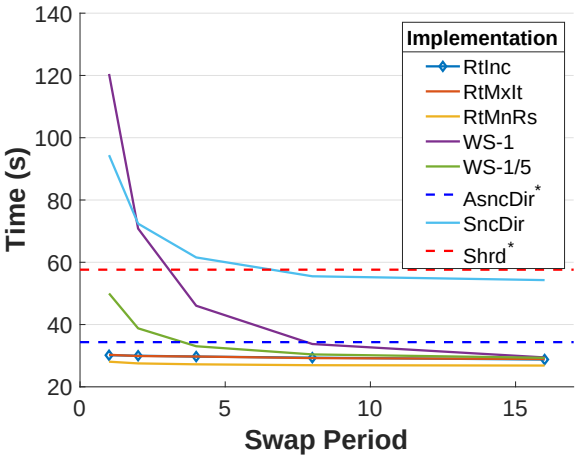
(c) Async. implementations, N=400



(d) All implementations, N=400

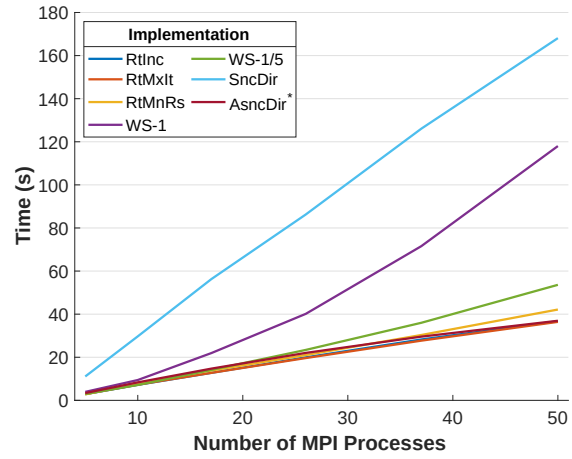


(e) Async. implementations, N=800

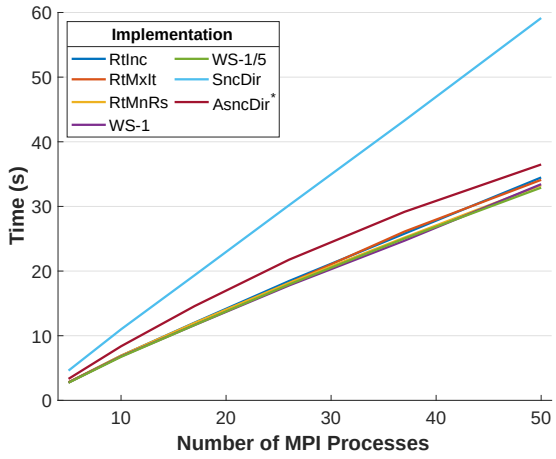


(f) All implementations, N=800

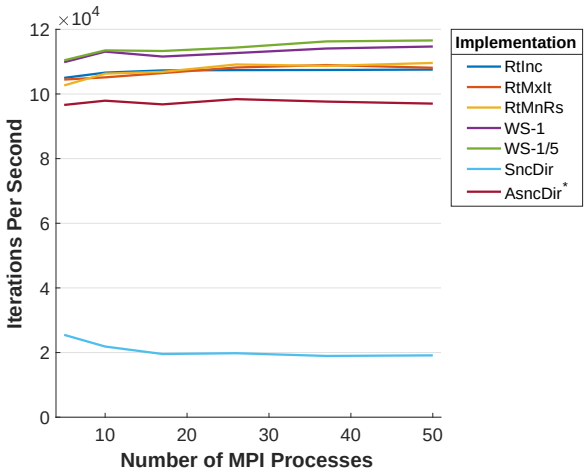
Figure 1: Convergence with respect to swap period. Synchronous, asynchronous, and shared-memory implementations compared, for three subdomain sizes. In the legends, an asterisk after AsncDir and Shrd indicates that they do not have a swap period.



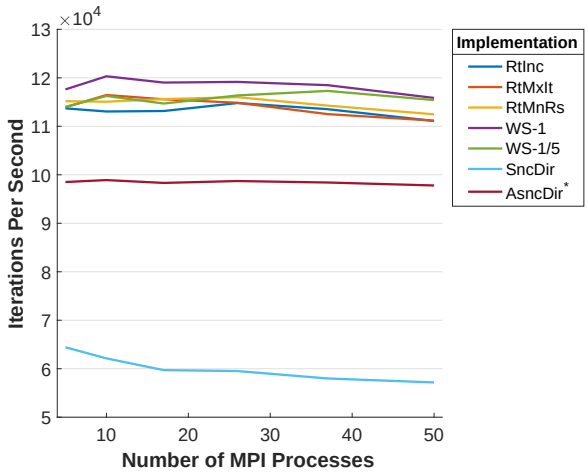
(a) Convergence, weak scaling, P_{swap}=1



(b) Convergence, weak scaling, P_{swap}=16

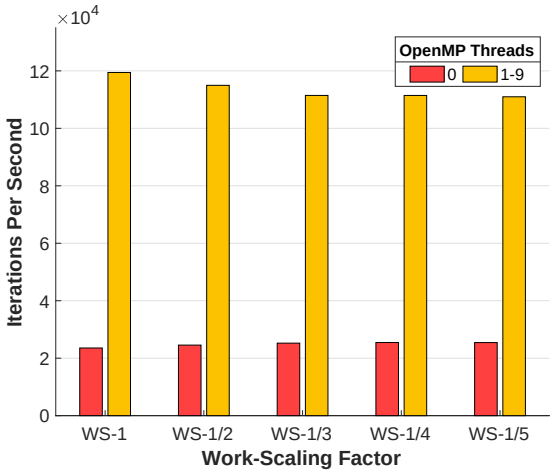


(c) Iteration rate, P_{swap}=1

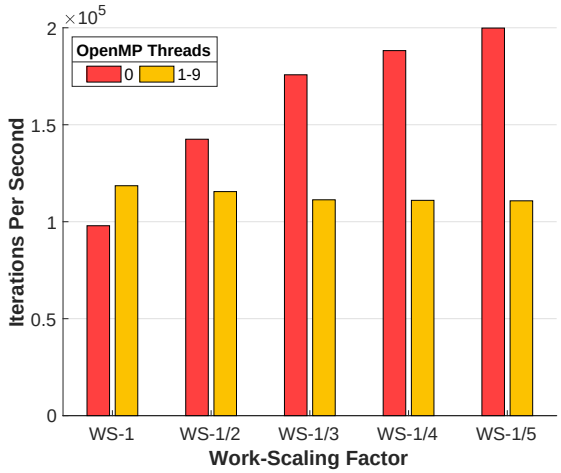


(d) Iteration rate, P_{swap}=16

Figure 2: Weak scaling and iteration rates for swap periods 1 and 16 with the subdomain size of 200. In the legends, an asterisk after AsncDir and Shrd indicates that they do not have a swap period.

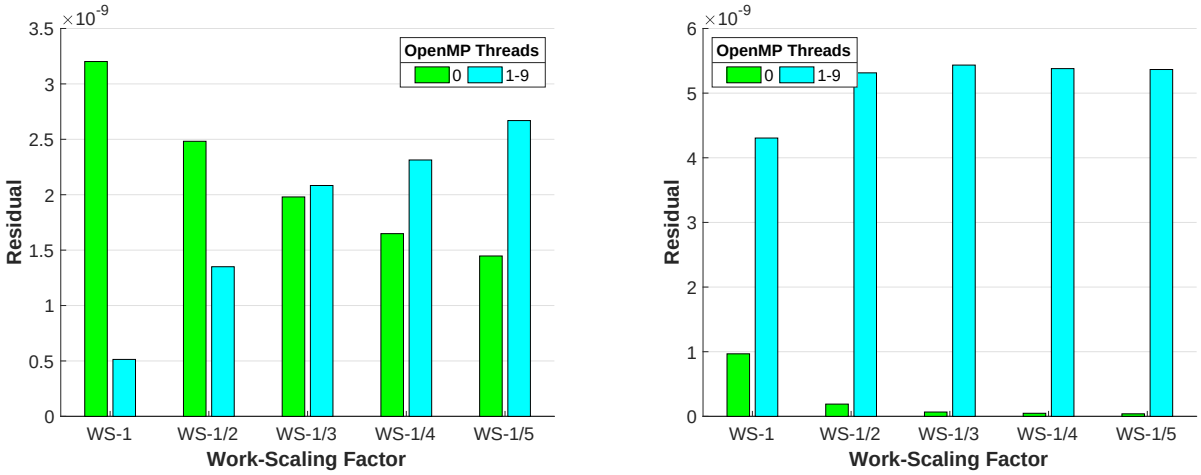


(a) P_{swap}=1



(b) P_{swap}=16

Figure 3: Iteration rates by thread number for swap periods 1 and 16, work-scaling implementation, and the subdomain size of 400.



(a) P_{swap}=1 (b) P_{swap}=16

Figure 4: Final residual (squared) contribution by thread number for swap periods 1 and 16, work-scaling implementation, and the subdomain size of 400.

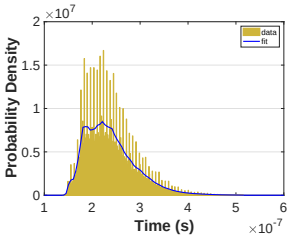
6 Model to Predict Appropriate P_{swap} and WSX Pairings

This predictive model is built around the 5-point stencil of the Laplacian as described in Section 3.2, and is specific to this stencil, Although performance models for other stencils may be inferred from a several given (parametric) models, as will be shown in Section 7.2, for instance. The model built here utilizes the time distributions generated from empirical data. Similar to (Jensen and Sosonkina, 2018), the collected empirical data was fit to several distributions, each of type KERNEL, a non-parametric distribution function that uses multiple superimposed normal distribution functions to represent data with a non-normal probability density. This distribution function works well for multimodal data as shown in Fig. 5 for key operations of the implementation, including the time required for a thread to (a) copy the left and right halo values of the subdomain, (b) copy the top and bottom halo values of the subdomain, (c) copy the interior boundary values from a neighbor, (d) compute updates for a boundary or interior row, (e) compute and update an interior row, (f) update the left and right subdomain boundary values, (g) update the top and bottom subdomain boundary values, (h) update interior boundary row values, and (i) compute its residual component. The communicating OpenMP thread additionally must (j) update the subdomain residual, (k) copy the subdomain boundary values, and (l) update the subdomain halo values, and the MPI master rank (m) communicates and updates the global residual, and (n) communicates and updates the halo values.

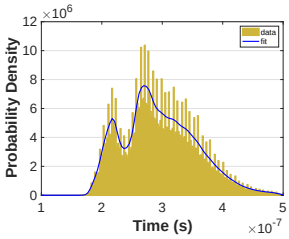
The flow of the simulation using modeled times is shown in Fig. 6, which relates the key operations (as in Fig. 5) to their execution per variants of hybrid implementation (as in Section 4). The upper half of Fig. 6 shows how the master MPI process is simulated. Its simulation times come from the distributions shown in Figs. 5m and 5n. The lower half of Fig. 6 represents a communicating thread in a non-master MPI process. This thread operations primarily correspond to Figs. 5j to 5l, while the remaining subfigures in Fig. 5 describe the times for the individual operations in the relaxation `Compute Phase` of Fig. 6. The two types of MPI processes exchange messages when a non-master process reaches the `Wait in Queue` stages. In the figure, green blocks signify a dependency between the master and non-master processes, orange blocks are actions specific to the simulation, dark blue blocks represent conditional branches, and all other actions are captured in blue blocks.

6.1 Results of Modeling

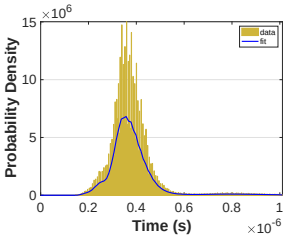
Figure 7 and Fig. 8 show all thread behavior is adequately modeled and comparable to actual performance on the Old Dominion University Turing cluster. This is evident in Figs. 7 and 8 and is supported by the implementation features described in Section 5. Figure 8 shows the average iteration rate over all threads (i.e. both communicating and non-communicating) as a function of work-scaling factor. The impact of different swap periods tends to have less effect for larger work-scaling values. Therefore, the model may be useful for the selection of good implementation parameters. In particular, to find the amount by which to scale the work given to the communicating thread in each subdomain, it is useful to compare the iteration rate I_c of the communicating thread with that (I_{nc}) of the non-communicating ones, as averaged over all the non-communicating threads in a subdomain. In Fig. 9, such a comparison is presented as a ratio of I_c to I_nc—with its optimal value equating to one—for the runs in which the communicating thread computes an area of the subdomain ranging from



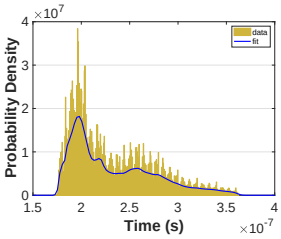
(a) OpenMP thread copy time, subdomain left and right halo values



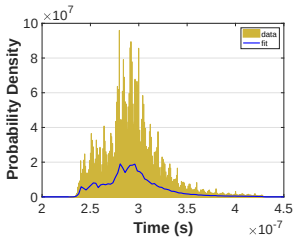
(b) OpenMP thread copy time, subdomain top and bottom halo values



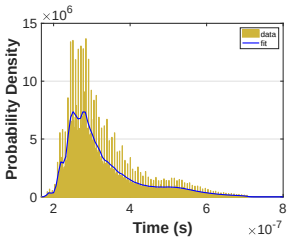
(c) OpenMP thread copy time, interior boundary row neighbor values



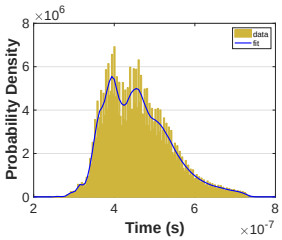
(d) OpenMP thread compute time, boundary row values



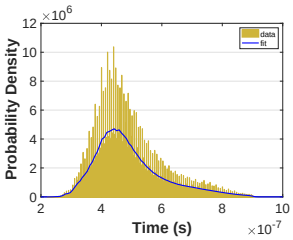
(e) OpenMP thread compute and update time, interior row values



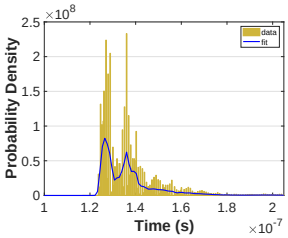
(f) OpenMP thread update time, subdomain left and right boundary values



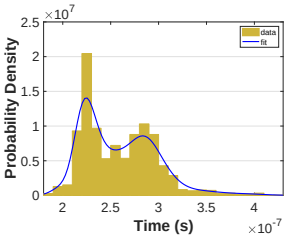
(g) OpenMP thread update time, subdomain top and bottom boundary values



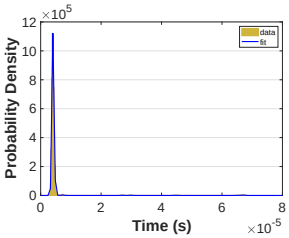
(h) OpenMP thread update time, interior boundary row values



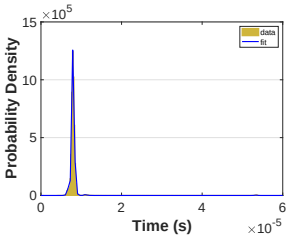
(i) OpenMP thread compute time, thread residual component



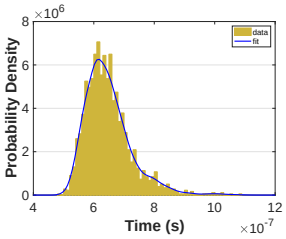
(j) OpenMP communicating thread compute time, subdomain residual



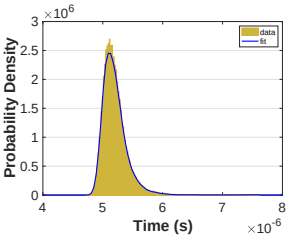
(k) OpenMP communicating thread copy time, subdomain boundary values



(l) OpenMP communicating thread update time, subdomain halo values



(m) MPI master rank communication and update time, global residual



(n) MPI master rank communication and update time, halo swap

Figure 5: Empirically-measured histogram data and time distributions from the five-point stencil implementation used in predictive model.

6% to 100% of n/p rows (i.e., WS1 factor) for swap periods P_{swap} ranging from 1 to 16. Note that only with $P_{swap} > 2$, the optimal ratio of one may be achieved.

6.2 Model Validation

Model validation is performed for the Work-Scaling WS1 implementation, in which the subdomain is equally divided among all OpenMP threads by comparing iteration times generated by the model to empirical iteration times as measured on the Turing cluster. Note that, although the other implementations as described in Section 4 may be validated in the same

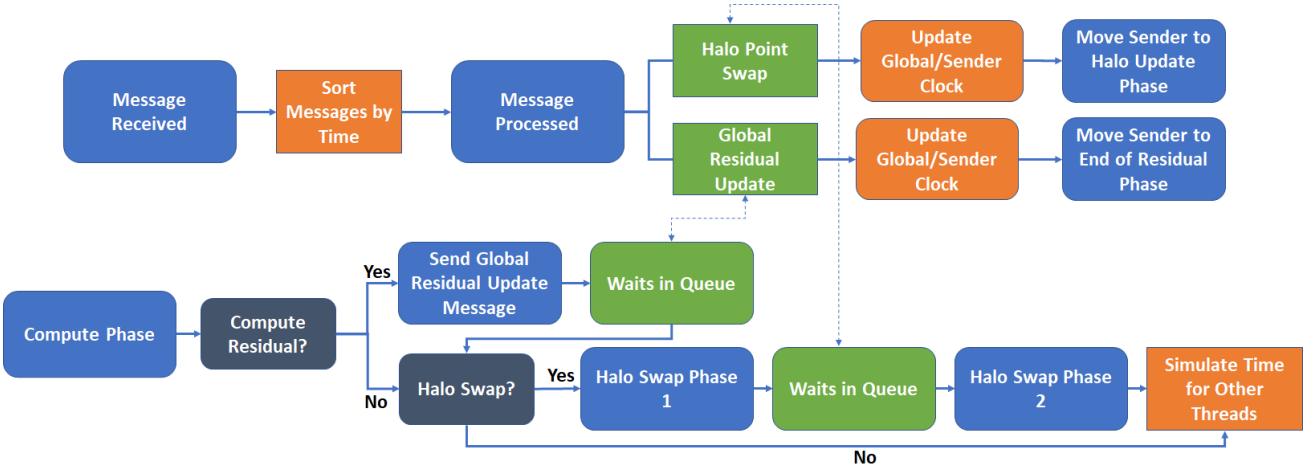
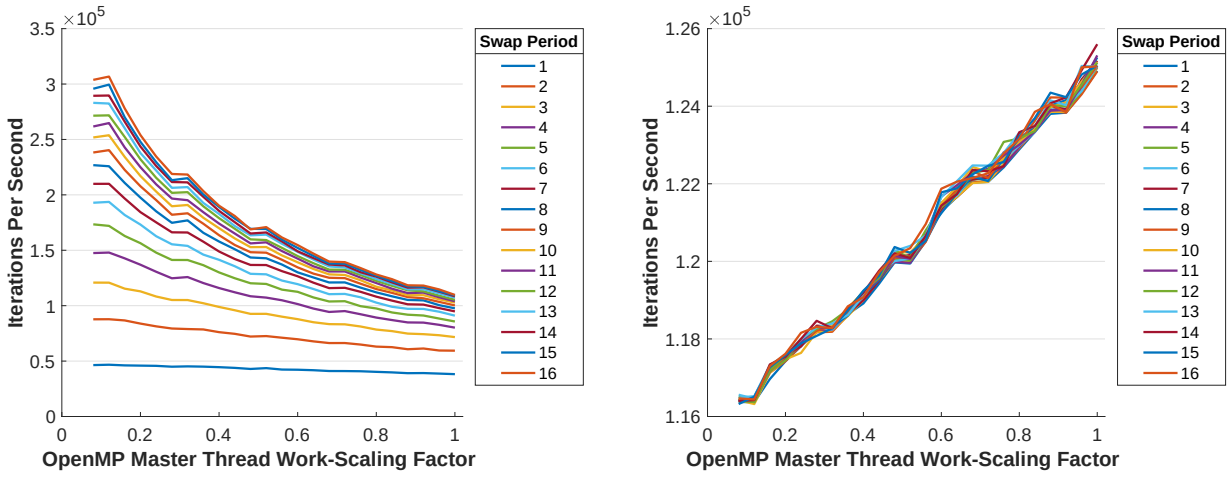


Figure 6: High level diagram of the flow of the predictive model. The upper half shows the flow of the master process as it processes messages in its queue, and the lower half provides an overview of the simulation of the communication threads in a non-master process. Dashed lines show communication points between the two types of processes.



(a) Rates, communicating thread

(b) Rates, non-communicating threads

Figure 7: Predictive model results for work-scaling implementation. Iteration rates of communicating and non-communicating threads, as a function of swap period and work-scaling factor.

manner, only the validation for WS1 is presented here. Four types of iteration execution are considered as follows (see Table 1):

1. For the communicating thread, denoted as `C`,
   (a) either just perform vector component update, denoted as `U`, or
   (b) update components (`U`), perform the halo swap (`S`), compute its residual (`R`), and communicate with the master MPI process (`r`).
2. For the non-communicating threads, denoted as `N`, in Table 1
   (a) either just update its vector components, denoted as `U`, or
   (b) update components (`U`) and compute its residual (`R`).

Table 1 shows the average relative error and its standard deviation between the modeled and measured iteration times. The average was taken over all the iterations that were performed within a three-second runtime period. Large error value (-26.35%) for the communicating thread was observed because of the specific node topology needed for the model construction. In particular, the model was developed based on a single-node (two-MPI-process) run, so that the homogeneity of the node type allocated on the cluster yields a better-fitting uni modal distributions. When this model—in which the 200 by 200 grid is mapped to one subdomain on one socket and communication is happening intranode—is extended to larger

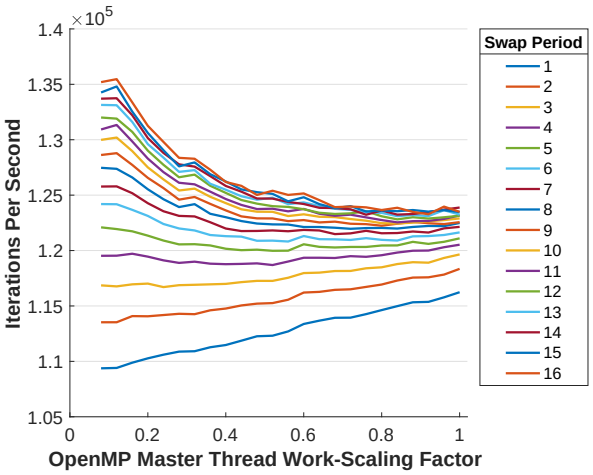


Figure 8: Iteration rates of all threads as a function of swap period and work-scaling factor.

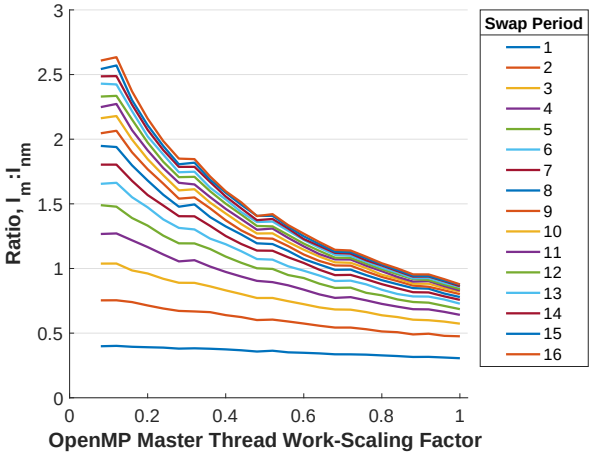


Figure 9: Iteration rate ratios as a function of swap period and work-scaling factor.

multi-subdomain problems, its modeled communication results in a faster time than that measured from multi-node runs, which incur internode communication.

Table 1: Average error between modeled and measured times of the 5-point stencil 'WS1' implementation for communicating 'C' and non-communicating 'N' threads and operations as follows: update 'U', halo swap 'S', local residual computation 'R', and communication with the master MPI process 'r'.

Execution Type	Mean, %	Std., %
C_U	9.46	-44.49
C_USRr	-26.35	-32.13
N_U	6.36	-12.39
N_UR	12.93	-19.07

7 Extending the Predictive Model

The model described in Section 6 is extended here to a nine-point stencil PDE as solved by the asynchronous as implemented in Section 4. Similar to Eq. (11) for the five-point stencil, Eq. (13) is a discrete Laplace operator that approximates the continuous Laplacian with a nine-point stencil.

$$\begin{aligned}\left(\nabla^2 f\right)(x,y) = {} & f(x-1,y) + f(x+1,y) + f(x,y-1) + f(x,y+1) \\ & + f(x-1,y-1) + f(x+1,y-1) + f(x-1,y+1) + f(x+1,y+1) - 8f(x,y)\end{aligned} \tag{13}$$

From this operator, the nine-point Jacobi algorithm in Eq. (14) follows, which is used in the nine-point implementation. The nine-point stencil requires eight add and one multiply operations, i.e., twice the add operations of the five-point stencil.

$$v_{l,m}^{k+1} = \frac{1}{8}\left(v_{l+1,m+1}^{k} + v_{l+1,m}^{k} + v_{l+1,m-1}^{k} + v_{l,m+1}^{k} + v_{l,m-1}^{k} + v_{l-1,m+1}^{k} + v_{l-1,m}^{k} + v_{l-1,m-1}^{k}\right) \tag{14}$$

7.1 Modeling with Parametric Distributions

The KERNEL distribution effectively models operation times, including irregular or multimodal data that may not be as clearly modeled by parametric distributions. In particular, parametric distributions visually cannot capture the nuance of more difficult data sets. However, one significant advantageous use of parametric distributions is their potential for the further predictive model extensions. For example (see Section 7.2), the two proposed distribution models may be extended to describe a 7-point stencil PDE solution by judicious selecting the pertinent distribution parameters based on relative operation counts for the three stencil types.

Table 2: Goodness-of-fit metrics, χ_2 test statistic for the given distributions and operations

Distribution	compute.5	compute.9	computeUpdate.5	computeUpdate.9
kernel	1.72E+04	2.20E+04	2.39E+05	5.61E+04
lognormal	1.83E+06	7.26E+05	8.89E+06	1.61E+07
normal	3.01E+06	1.11E+06	3.13E+07	3.75E+07
weibull	4.13E+06	2.99E+06	1.26E+08	1.54E+08
gamma	2.14E+06	8.37E+05	1.34E+07	2.11E+07
loglogistic	1.96E+06	6.18E+05	3.22E+06	6.56E+06
gev	9.93E+05	7.19E+04	7.58E+05	8.76E+05

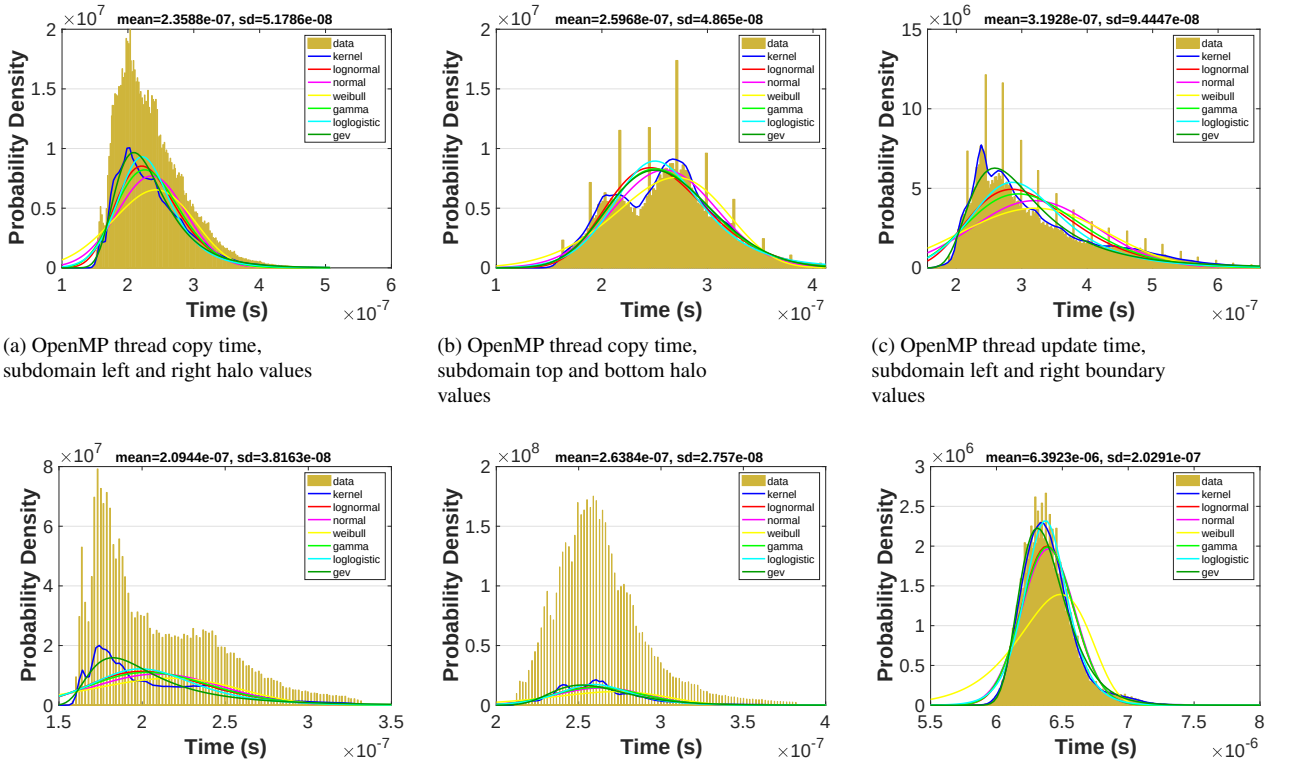
Table 3: Percent Error, Model Iteration Times CW Implementation, Five- and Nine-Point Stencils, KERNEL and GEV. For iteration types, 'C' refers to the communicating thread, 'N' refers to a non-communicating thread, 'U' refers to updating components, 'S' refers to the halo swap, 'R' refers to computing the thread-level residual, and 'r' refers to updating the global residual with the master MPI process.

Stencil	Distribution	Iteration Type	Mean	Std.
5-pt	KERNEL	C_U	9.21	-47.08
		C_USRr	-26.61	-36.68
		N_U	6.01	-14.44
		N_UR	12.62	-20.88
	GEV	C_U	9.34	-44.77
		C_USRr	-26.39	-69.81
		N_U	6.13	-12.94
		N_UR	12.71	-19.57
9-pt	KERNEL	C_U	6.75	-35.35
		C_USRr	-18.79	66.66
		N_U	5.55	-3.16
		N_UR	10.45	-9.28
	GEV	C_U	6.77	-37.76
		C_USRr	-21.72	-52.17
		N_U	5.61	-6.02
		N_UR	10.51	-11.72

Figure 10 plots the KERNEL distribution along with several parametric distributions, for a representative subset of iteration operations for 5-point stencil. Note that some of the histograms in Fig. 10 contain empty bins, which may affects the quality of the fitted distributions and which may be due to either the timing function granularity used or small time width of the bins. Most of the distributions, however, appear to fit normally distributed data well. Figures 10a, 10c and 10d all show that the generalized extreme value distribution GEV, most closely matches KERNEL in skewed datasets. In other words, the peak of the GEV curve more closely aligns with the mode of the histogram. Further, GEV lacks the long left tails of other parametric distributions, which model unobserved and impossibly quick operation times as a result of their poor fit. The GEV distribution has also been previously used to model asynchronous update times for an iterative process (Jensen and Sosonkina, 2018).

In addition to the visual distribution assessment given above, for each of the distributions a goodness-of-fit metric can be calculated to provide a more quantitative argument for or against specific distributions. For this, the χ_2 test was used, and the value of the test statistic as calculated by MATLAB is provided in Table 2 for the compute and update operations (columns compute and computeUpdate, respectively) for the 5-point and 9-point stencils (column names suffixed with .5 and .9, respectively). For the 5-point implementation, data for these operations can be seen in Figs. 5d and 5e. Note that the compute and update operations were selected for the goodness-of-fit analysis since the distributions vary the most based on the size of the stencil used. In Table 2, it is apparent that the GEV distribution provides a better fit according to the χ_2 test by an order of magnitude relative to the other parametric distributions. Hence, overall, the GEV distribution is the best parametric distribution to replace KERNEL in the model presented here.

Table 3 shows that for 5-point and 9-point models, each of the iteration execution types, as defined in Section 6.2 modeled by the GEV distribution generates mean times similar to those modeled by KERNEL distribution. As seen in Table 3, the predictive model with GEV-modeled operations generates iteration times that are similar to observed iteration times on the Turing cluster. Recall that large mean errors in the C_USRr cases are due to the differences in node topologies of the model and measured hardware setup (see Section 6.2 for more details.)



(a) OpenMP thread copy time, subdomain left and right halo values

(b) OpenMP thread copy time, subdomain top and bottom halo values

(c) OpenMP thread update time, subdomain left and right boundary values

(d) OpenMP thread compute time, boundary row values

(e) OpenMP thread compute and update time, interior row values

(f) MPI master rank communication and update time, halo swap

Figure 10: Empirically-measured histogram data from the five-point stencil implementation with multiple fitted distributions, for some iteration operations.

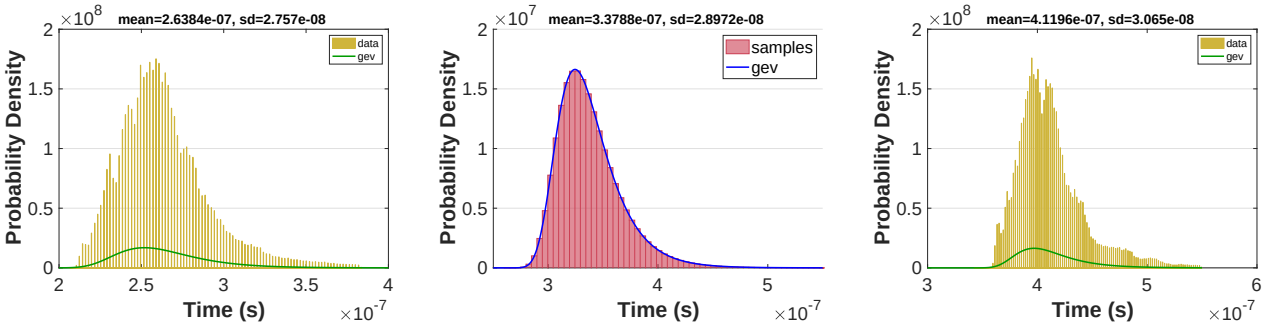
7.2 Example of parametric model extension

Given that GEV is defined by the location μ, scale σ, and shape ξ parameters, it may be possible to modify them, based on problem and system characteristics, to model iteration operations for untested problems. Specifically, instead of taking measurements from a calculation and generating a distribution from that empirical data, a new set of distribution parameters may be deduced from the problem and system characteristics if their relation to some problem(s) with already known distribution parameters. Figure 11b shows the results of *interpolating* GEV parameters from Fig. 11a and Fig. 11c to generate a new set of GEV distribution parameters that may model well the a 7-point stencil problem. This new set of distribution parameters generates samples with mean and standard deviation of 3.3788e-07 and 2.8972e-08 seconds, respectively, which are similar to the average of the 5- and 9-point means and standard deviations of 3.379e-7 and 2.911e-8 seconds, respectively. Indeed, compared to 5- and 9-point stencils, the 7-point stencil requires 50% more and 25% fewer additions, respectively, and hence its computation be approximated by an average of the two stencil sizes. Compute and update operation times from a seven-point stencil implementation shown in Fig. 12 compare favorably to the predicted times in Fig. 11b, with mean time relative error of -2.32%.

8 Conclusions and Future Work

This work has presented several distinct hybrid parallel implementations of the asynchronous Jacobi algorithm and analyzed the performance of each. Generally speaking, the asynchronous implementations outperform the synchronous implementation for the problem studied, however, further optimization is possible in both cases. Predictive distribution-based models were developed and validated for two common stencils used to solve the Laplace PDE. In the majority of cases the performance model matches within 6% of the empirical data. The models were used to tune the implementation parameters. In particular, it was found that decreasing the halo swap frequency in the base hybrid parallel implementation WS1 provides significant performance improvements and equalizes progress towards solution among subdomains.

An example of the manipulation of the GEV distribution parameters was presented to show possibilities of extending the proposed models and modeling methodology.In the future, by changing multiple system parameters and generating



(a) 5-point compute and update time $(\mu, \sigma, \xi) = (2.52e{-}7, 2.18e{-}8, -0.0133)$

(b) Interpolated 7-point compute and update time $(\mu, \sigma, \xi) = (3.25e{-}7, 2.21e{-}8, 0.0155)$

(c) 9-point compute and update time $(\mu, \sigma, \xi) = (3.98e{-}7, 2.25e{-}8, 0.0443)$

Figure 11: Compute and update operation times for five- and nine-point stencil implementations and modeled seven-point stencil.

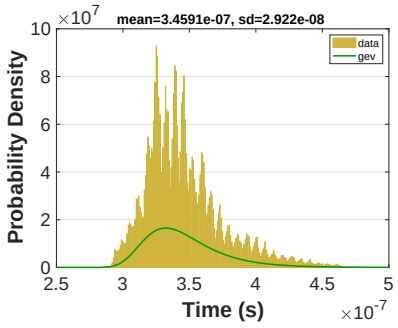


Figure 12: Seven-point stencil implementation compute and update operation times.

sufficient data points for analysis may lead to more general model extensions. Another avenue of future work lies in the direction of modeling different asynchronous solvers and in combining the results from various asynchronous solvers.

Acknowledgements This work was supported in part by the Air Force Office of Scientific Research under the AFOSR award FA9550-12-1-0476, by the U.S. Department of Energy (DOE) Office of Advanced Scientific Computing Research under the grant DE-SC-0016564 and the Exascale Computing Project (ECP) through the Ames Laboratory, operated by Iowa State University under contract No. DE-AC00-07CH11358, by the U.S. Department of Defense High Performance Computing Modernization Program, through a HASI grant, the Turing High Performance Computing cluster at Old Dominion University, and through the ILIR/IAR program at the Naval Surface Warfare Center - Dahlgren Division.

References

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

Anzt H, Dongarra J, Quintana-Ortí ES (2015) Tuning stationary iterative solvers for fault resilience. In: Proceedings of the 6th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ACM, p 1

Anzt H, Dongarra J, Quintana-Ortí ES (2016) Fine-grained bit-flip protection for relaxation methods. Journal of Computational Science

Ashby S, Beckman P, Chen J, Colella P, Collins B, Crawford D, Dongarra J, Kothe D, Lusk R, Messina P, et al. (2010a) Ascac subcommittee report: The opportunities and challenges of exascale computing. Tech. rep., Technical report, United States Department of Energy, Fall

Ashby S, Beckman P, Chen J, Colella P, Collins B, Crawford D, Dongarra J, Kothe D, Lusk R, Messina P, et al. (2010b) The opportunities and challenges of exascale computing–summary report of the advanced scientific computing advisory committee (ascac) subcommittee. US Department of Energy Office of Science

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

Bahi JM, Contassot-Vivier S, Couturier R (2003) Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational grid. In: Parallel and Distributed Processing Symposium, 2003. Proceedings. International, IEEE, pp 9–pp

Bahi JM, Contassot-Vivier S, Couturier R (2006) Performance comparison of parallel programming environments for implementing aiac algorithms. The Journal of Supercomputing 35(3):227–244

Baudet GM (1978) Asynchronous iterative methods for multiprocessors. Journal of the ACM (JACM) 25(2):226–244

Bertsekas DP, Tsitsiklis JN (1989) Parallel and distributed computation: numerical methods, vol 23. Prentice hall Englewood Cliffs, NJ

Bethune I, Bull JM, Dingle NJ, Higham NJ (2011) Investigating the Performance of Asynchronous Jacobi's Method for Solving Systems of Linear Equations. To appear in International Journal of High Performance Computing Applications

Bethune I, Bull JM, Dingle NJ, Higham NJ (2014) Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP. The International Journal of High Performance Computing Applications 28(1):97–111

Chazan D, Miranker W (1969) Chaotic relaxation. Linear algebra and its applications 2(2):199–222

Cheung YK, Cole R (2016) A unified approach to analyzing asynchronous coordinate descent and tatonnement. arXiv preprint arXiv:161209171

De Jager DV, Bradley JT (2010) Extracting state-based performance metrics using asynchronous iterative techniques. Performance Evaluation 67(12):1353–1372

Dongarra J, Hittinger J, Bell J, Chacon L, Falgout R, Heroux M, Hovland P, Ng E, Webster C, Wild S (2014) Applied mathematics research for exascale computing. Tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA

Frommer A, Szyld DB (2000) On asynchronous iterations. Journal of computational and applied mathematics 123(1):201–216

Hong M (2017) A distributed, asynchronous and incremental algorithm for nonconvex optimization: An admm approach. IEEE Transactions on Control of Network Systems

Hook J, Dingle N (2013) Performance analysis of asynchronous parallel jacobi. Numerical Algorithms pp 1–36

Iutzeler F, Bianchi P, Ciblat P, Hachem W (2013) Asynchronous distributed optimization using a randomized alternating direction method of multipliers. In: Decision and Control (CDC), 2013 IEEE 52nd Annual Conference on, IEEE, pp 3671–3676

Jensen E, Sosonkina M (2018) Modeling a task-based matrix-matrix multiplication application for resilience decision making. In: Proceedings of the 26th High Performance Computing Symposium, HPC '18

Jensen E, Coleman E, Sosonkina M (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

Lindeberg T (1990) Scale-space for discrete signals. IEEE transactions on pattern analysis and machine intelligence 12(3):234–254

Smith GD (1985) Numerical solution of partial differential equations: finite difference methods. Oxford university press

Strikwerda JC (2004) Finite difference schemes and partial differential equations, vol 88. Siam

Szyld DB (1998) Different models of parallel asynchronous iterations with overlapping blocks. Computational and applied mathematics 17:101–115

Voronin K (2014) A numerical study of an mpi/openmp implementation based on asynchronous threads for a three-dimensional splitting scheme in heat transfer problems. Journal of Applied and Industrial Mathematics 8(3):436–443

Wolfson-Pou J, Chow E (2016) Reducing communication in distributed asynchronous iterative methods. Procedia Computer Science 80:1906–1916