Using Modeling to Improve the Performance of Asynchronous Jacobi

Erik J. Jensen[1], **Evan Coleman**[1,2], **and Masha Sosonkina**[1]
[1]Modeling, Simulation and Visualization Engineering, Old Dominion University, Norfolk, VA, USA
[2]Naval Surface Warfare Center, Dahlgren Division, Dahlgren, VA, USA

Abstract—*Asynchronous algorithms may increase performance of parallel applications. This work investigates strategies for implementing asynchronous hybrid parallel MPI-OpenMP Jacobi solvers, and generates a predictive performance model that suggests solver parameters that are well-suited to a specific problem. Results show that making efforts to equalize the number of iterations for all processing elements benefits performance and solution quality.*

Keywords: Jacobi method, Asynchronous implementation, Fixed-point iteration, parallel hybrid MPI OpenMP, Laplace equation

1. Introduction

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 to proceed to the next step or iteration, 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, for example, from iteration to iteration; or they may harness the ability of a computing element to perform useful work while waiting for information from another computing element.

As future high-performance computing (HPC) platforms continue to scale in the number of processing elements and calculations are increasingly parallelized, asynchronous methods may offer performance superior to synchronous methods. Several U.S. Department of Energy reports [2], [3], [15] have cited the need for the development of asynchronous computational methods to run efficiently on future exascale HPC environments.

Bethune, et al. present several asynchronous Jacobi method implementations that solve the Laplace equation in three dimensions, which are parallelized with MPI, SHMEM, or OpenMP [8], [9]. Their finding that asynchronous Jacobi implemented with OpenMP is 33% faster than the synchronous OpenMP application leads them to suggest the potential for OpenMP in a hybrid MPI–OpenMP implementation. This work implements some asynchronous hybrid MPI–OpenMP models that solve a two-dimensional finite-difference discretization of the Laplacian using Jacobi's method, similar to the work of Bethune, et al. Related work and the Laplacian equation are outlined in Sections 2 and 3, respectively. The parallel solver models and implementations are described in Section 5. Implementation testing results are examined in Section 6. The predictive model is explained in section Section 7, model results are discussed in Section 7.1, and Section 8 concludes and projects future work.

2. Related Work

Bahi et al. demonstrate the efficacy of asynchronous methods, especially for grid systems, and propose a system for classifying parallel iterative algorithms, based on computational and communication strategies [6], [5]. Iterations and communications may be either synchronous or asynchronous. This work develops and tests implementations that are Asynchronous Iterations - Asynchronous Communications (AIAC), Synchronous Iterations - Synchronous Communications (SISC), and also an unmentioned class, Asynchronous Iterations - Synchronous Communications (AISC). Jager and Bradley also demonstrate superior performance of asynchronous methods for solving large sparse linear fixed-point problems [14]. 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 [28]. 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 [1], [4], as well as performance analysis of asynchronous methods [8], [19], [9]. In particular, both [8], [9] 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 [29]. 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 [13], [20], [18], [30], [24], [27], [10]

3. Problem Description

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 [23]. The Laplace equation in two dimensions,

$$\nabla^2 \phi = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = b, \tag{1}$$

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

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

approximates the two-dimensional continuous Laplacian using a five-point stencil [22]. From this, a discretized version of the Jacobi algorithm,

$$v_{l,m}^{k+1} = \frac{1}{4}(v_{l+1,m}^{k} + v_{l-1,m}^{k} + v_{l,m+1}^{k} + v_{l,m-1}^{k}, \tag{3}$$

may be applied to solve a two-dimensional sparse linear system of equations [25]. Subscript indices l, m and superscript index k define discrete grid nodes and iteration number, such that $v_{l,m}^{k+1}$ is the solution at node l, m at the $(k+1)^{th}$ iteration. This work uses the Jacobi algorithm to solve a two-dimensional finite-difference discretization of the Laplacian with Dirichlet boundary conditions, which may be viewed as a heat diffusion problem with a plate held to specific temperatures along the boundary [11].

4. Asynchronous Iterative Methods

In parallel computing, when performing the computation asynchronously, each component of the problem, e.g., a matrix or vector (block) entry, is updated in a manner that does not require information from the computations involving other components while the update is being made. This allows for each computing element (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 [16], which in turn comes from [12], [7] and [26] (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,

$$x^{k+1} = G(x^k), \tag{4}$$

and a fixed point is declared if $x^{k+1} \approx x^k$.

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 B of components $B_1, B_2, \ldots, B_m$ to update. Note that the number p of processing elements will typically be significantly smaller than the number m of blocks to update. With these assumptions, the computational model can be stated in Algorithm 1.

Algorithm 1: General Computational Model

```
1 for each processing element $P_l$ do
2     for $i = 1, 2, \ldots$ 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 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 the function $s_l(k)$ indicates how many times an specific component has been updated. Finally, a set of individual components can be grouped into a set I^k, which contains all of the components that were updated on the k^{th} iteration. Given these basic definitions, the three following conditions (along with the model presented in Algorithm 1) provide a working mathematical framework for fine-grained asynchronous computation.

Definition 1: If the following three conditions hold:

1) $s_i(k) \le k - 1$, *i.e. only components that have finished computing are used in the current approximationr,*
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

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

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

This basic 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.

5. Hybrid Implementations

The implementations solve a two-dimensional finite-difference discretization of the Laplacian, using OpenMP for shared memory parallelization, and MPI for distributed-memory parallelization. Generally, the Laplacian is discretized over a rectangular region, which is then divided into a number of evenly distributed regions equal to the number of MPI processes. These regions are then divided further to assign a number of rows to an OpenMP thread. For the n_p threads in each subdomain, one is assigned the role of master/communicator[1]. Several types of implementations have been developed and used in this study. These implementations are predominantly asynchronous in nature, according to the computational model presented in the previous subsection, but a synchronous implementation and an exclusively shared-memory-parallel implementation are included as well as points of comparison. All of the implementations except for the shared-memory implementation are fundamentally suited for computation on a distributed computing platform. The implementations that are provided fall into the following categories: Asynchronous Iterations with Asynchronous Communication (AIAC), Asynchronous Iterations with Synchronous Communication (AISC), and Synchronous Iterations with Synchronous Communication (SISC). Descriptions of these different classifications follow in the next few subsections. In addition to these implementations, an exclusively shared-memory-parallel implementation, SHRD, has been developed as a benchmark, to measure the effectiveness of the hybrid-parallel implementations.

a) Asynchronous Iterations–Asynchronous Communications (AIAC): Two matrices U_0 and U_1 store grid point values that each thread reads, e.g. from U_1, to compute newer values to write, e.g. to U_0. As the method is asynchronous, each thread independently determines which matrix stores its newer $u^{(t+1)}(i,j)$ values and older $u^{(t)}(i,j)$ values. For an $n+2$ by $n+2$ subdomain grid that is equally divided among all n_p threads, each thread solves for n^2/n_p grid points, such that the grid is evenly partitioned along the y-axis. When a thread copies grid point values above or below its domain for the computation, OpenMP locks are employed to ensure that data is captured accurately, from a single iteration. Further, locks are used when updating values on boundary rows and subdomain halos, and when copying subdomain boundaries. The swap period $P_{\text{swap}}= k,\ k = 1, 2, \ldots, N$ denotes that halo values are exchanged every kth iteration only. Each thread p_n computes the its local residual 2-norm squared[2] every kth iteration, which it contributes to the subdomain residual. Using an OpenMP atomic operation, a single thread computes the subdomain residual-norm value, as a sum of thread-local residual norms, and sends it to the master MPI process.

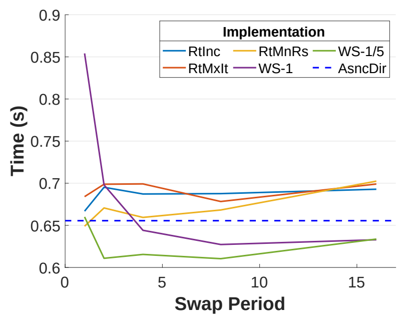
Within the AIAC category, there are five variants:

1) Work-Scaling-1 (WS1): The subdomain is equally divided among all OpenMP threads, and thread zero, p_0, performs communication with the master MPI process. In addition to the halo swap, p_0 computes the residual norm for the subdomain and updates the master with this value.
2) Work-Scaling-X (WSX): The subdomain is divided, such that the communicating thread p_0 performs less computational work—specifed by the fraction $0 < X < 1$—than the other threads. The size of the p_0 compute region is $n/n_p \times X$ rows, and the other threads equally divide the remainder of the subdomain. As in WS1, thread p_0 computes and updates the subdomain residual norm.
3) Rotating-Incrementing (RTINC): The subdomain is equally divided among all the OpenMP threads, but communication with the master MPI process is rotated in a round-robin fashion among the threads. Communication passes to the next thread after one halo swap operation. The communicating thread also computes and updates the subdomain residual norm.
4) Rotating-Maximum-Iterations (RTMAXIT) is similar to RTINC, but chooses the OpenMP thread that has performed the most iterations for the next communication and update operations.
5) Rotating-Minimum-Residual (RTMNRS) is similar to RTMAXIT, but chooses the OpenMP thread that has the smallest contribution to the residual norm for the next communication and update operations.

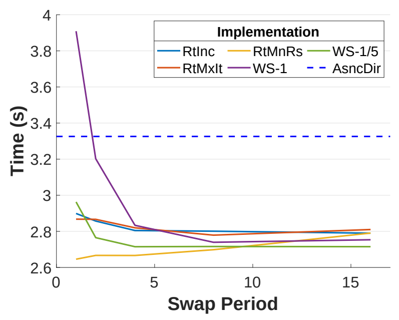
b) Asynchronous Iterations–Synchronous Communications (AISC): The AISC implementation Asynchronous-Direct (ASNCDIR) is similar to the AIAC implementations, but one OpenMP thread in each MPI process is reserved exclusively for synchronous communication with other subdomains. In ASNCDIR, the communicating threads swap subdomain halo values directly, without the master MPI pro-

[1]In this paper, *master thread* and *communicating thread* are equivalent and used interchangeably.

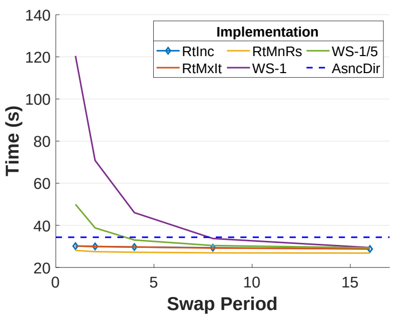
[2]In the remainder of the paper, *norm* will be used instead of *2-norm squared* for brevity.



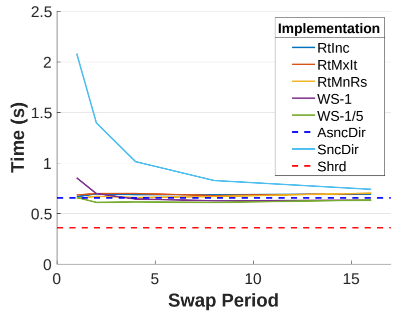
(a) Async. implementations, N=200



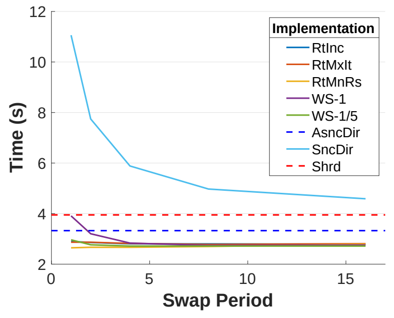
(b) Async. implementations, N=400



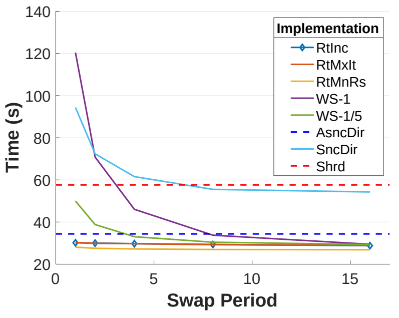
(c) Async. implementations, N=800



(d) All implementations, N=200



(e) All implementations, N=400



(f) All implementations, N=800

Fig. 1: Convergence of hybrid-parallel and shared-memory implementations with respect to swap period P_{swap}, for three subdomain sizes.

cess intermediary. The compute threads still require OpenMP locks for copying and updating shared variables values, and the communication thread also retains locks for copying the subdomain boundary and updating the subdomain halo.

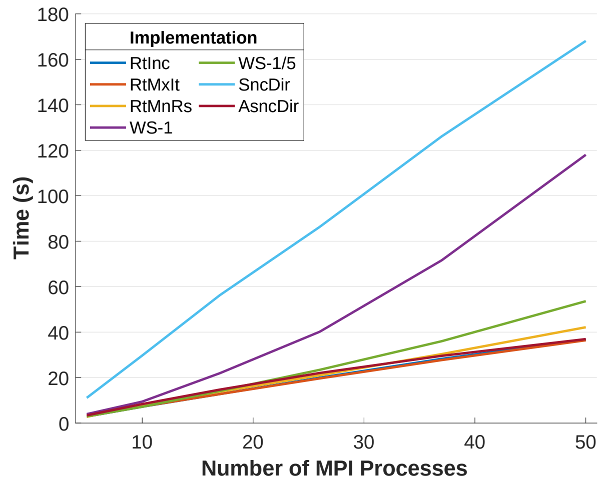
c) Synchronous Iterations–Synchronous Communications (SISC): The SISC implementation Synchronous-Direct (SNCDIR) is a modification of an AIAC implementation, in which iterations and communications have been synchronized across threads and subdomains. For SNCDIR, while one subdomain thread communicates directly with other subdomains, all the non-communicating threads are idle in that subdomain. OpenMP locks are unnecessary and removed.

6. Performance Results

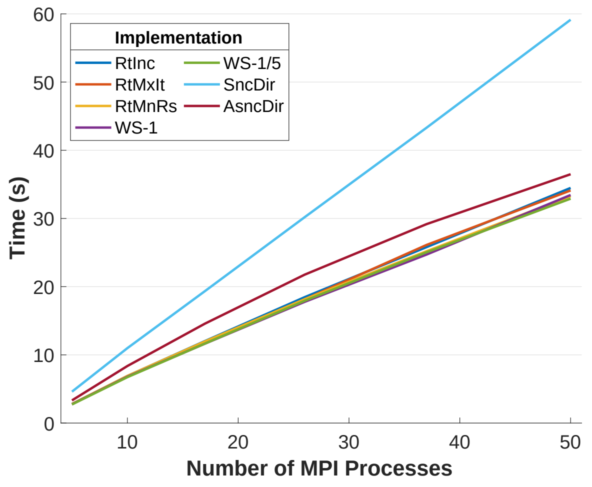
Experiments were conducted on the Turing High Performance Computing cluster at Old Dominion University, which contains 190 standard compute nodes, 10 GPU nodes, 10 Intel Xeon Phi Knight's Corner nodes, and 4 high memory nodes, connected with a Fourteen Data Rate (FDR) InfiniBand network. Compute nodes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

1) 10 Intel Xeon E5-2670 v2 2.50Ghz cores, or
2) 10 Intel Xeon E5-2660 v2 2.20Ghz cores.

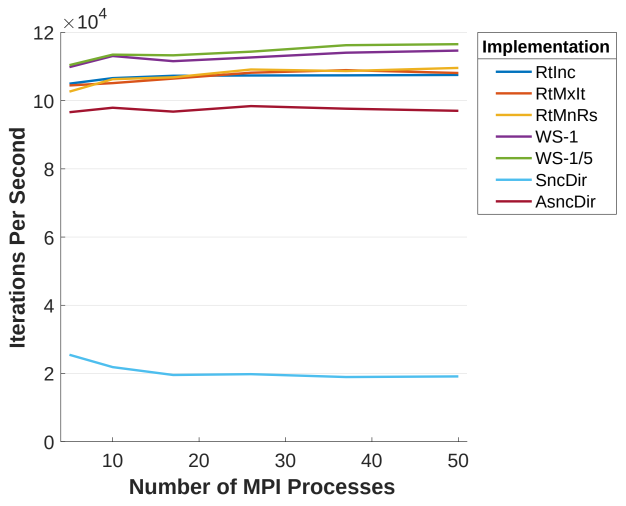
Data collected from empirical testing of the implementations demonstrates the efficacy of different solving strategies, as a function of problem size, subdomain size, and swap period P_{swap}, which ranges from 1 to 16. Figure 1 presents time to convergence as a function of swap period for all implementations that were developed and tested for this work. Error tolerance for all tests is $1e^{-4}$. Grids were divided into four subdomains, with two divisions along the midpoints of each axis. Figure 1(a), (b), and (c) depict data only for asynchronous hybrid-parallel implementions, while Fig. 1(d), (e), and (f) also includes data for the synchronous hybrid-parallel and shared-memory implementations SNCDIR and SHRD. ASNCDIR, SNCDIR, and SHRD are included as dashed reference lines because they lack swap-period functionality. Figure 1(a) and (d) show that for domain size $N = 200$, the shared-memory implementation outperforms the hybrid-parallel implementations; the added complexity of hybrid parallelization cannot be justified for such a small domain. However, using larger subdomain sizes demonstrates that the hybrid implementations benefit from the additional complexity of distributed parallelization. Figure 1(b) and (e) show that for domain size $N = 400$, the asynchronous hybrid-parallel implementations outperform SHRD, in terms



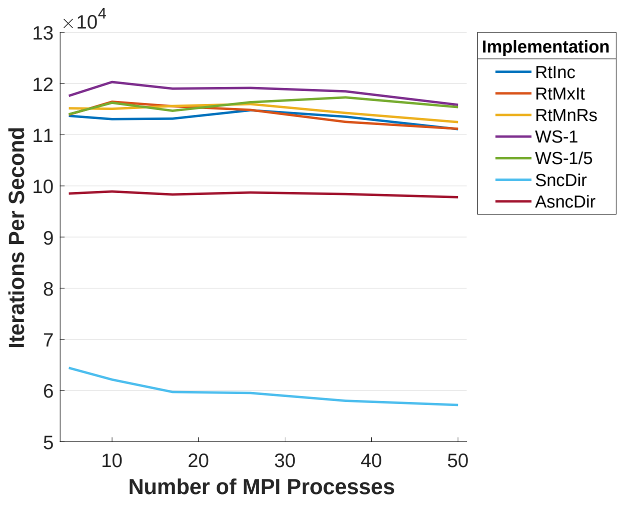
(a) Convergence, weak scaling, P_{swap}=1



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



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



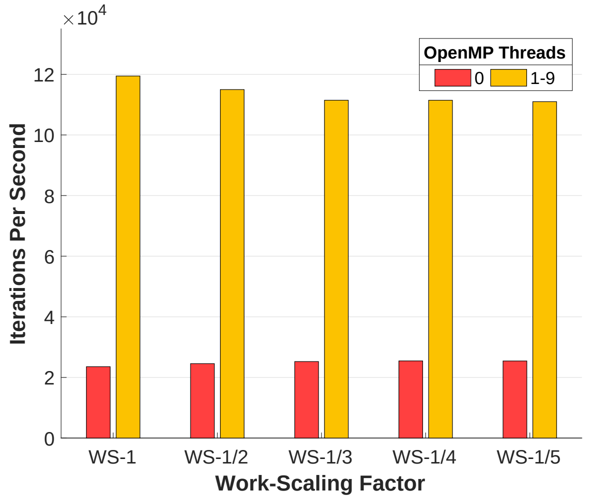
(d) Iteration rate, P_{swap}=16

Fig. 2: Weak scaling and iteration rates for swap periods 1 and 16. Individual weak-scaling trajectories may be interpreted only relative to other implementations.

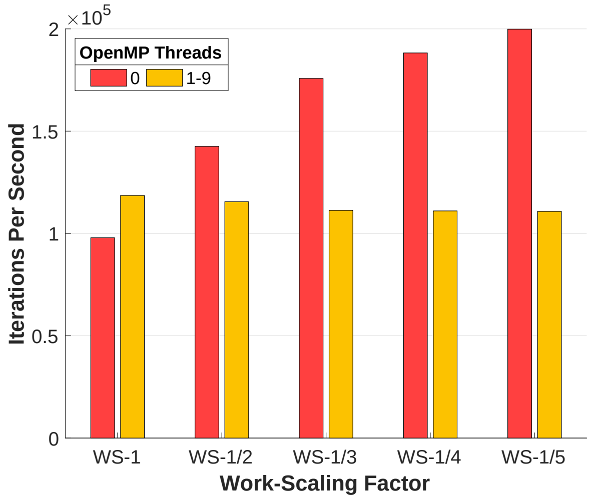
of calculation time. In Fig. 1(c) and (f), for domain size $N = 400$, the gap between the asynchronous implementations and SHRD widens substantially. As shown in Fig. 1(e) and (f), the asynchronous implementations run more quickly than the synchronous hybrid-parallel implementation SNCDIR, which loses opportunity to compute on non-communicating threads during the communication phase. When P_{swap} is sufficiently large, the five different implementations that are of type AIAC perform similarly, with minor variations. 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. However, the arc in the ASNCDIR weak-scaling trajectory in Figure 2(b) suggests ASNCDIR may perform better than AIAC implementations at a larger scale.

Figure 2 shows the scaling abilities of the implementations, for subdomain size $n = 200$. Figure 2(c) and (d) 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 the shortcomings of the work-scaling implementation WS, for subdomain size $n = 200$ and 4 subdomains. While WS is the easiest AIAC

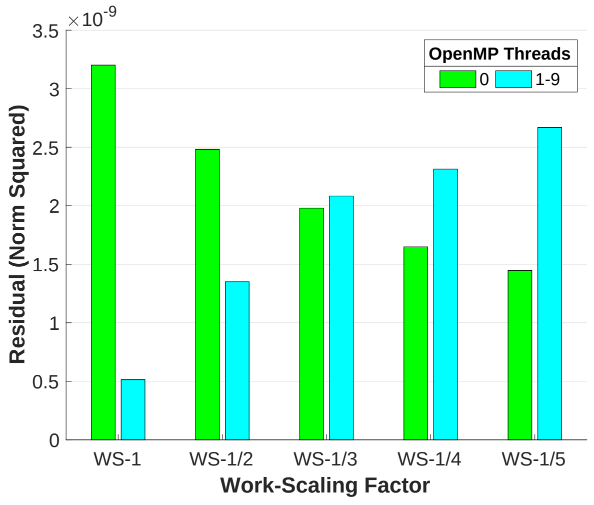


(a) P_{swap}=1

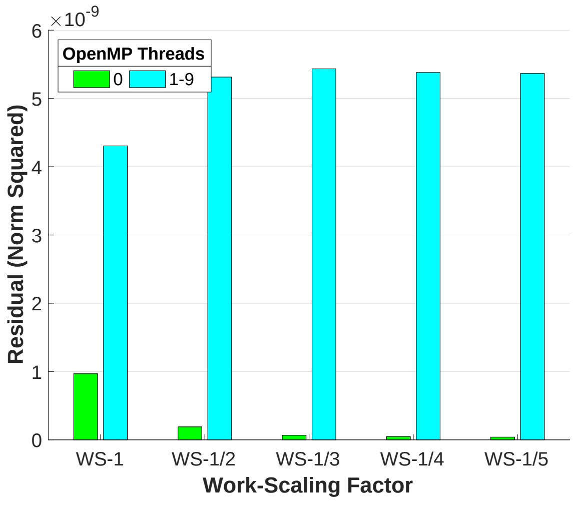


(b) P_{swap}=16

Fig. 3: Iteration rates by thread number for swap periods 1 and 16, work-scaling implementation.



(a) P_{swap}=1



(b) P_{swap}=16

Fig. 4: Final residual norm contribution by thread number for swap periods 1 and 16, work-scaling implementation.

variant to implement, convergence and solution quality may suffer if care is not taken to balance communication with computation. As a dedicated thread performs communications as well as computations, the iteration rate may be less than that for the non-communicating threads, if communication and computation phases are imbalanced. Figure 3 implies balance can be achieved by pairing the work-scaling factor X with an appropriate swap period, such that time, and therefore iterations, lost during the communication phase, may be budgeted with a sufficiently small compute load.

Equalizing iteration rates of master and non-master threads also may help mitigate the error imbalance seen in Fig. 4; error imbalance in this context refers to the contribution to the residual norm from the elements assigned to specific threads. Explicitly, the residual at the k^{th} iteration $r^k = b - Ax^k$ can be examined at the component level to see the contribution to the total residual norm from each element. Figure 4(a) shows that when the work is not scaled appropriately, a relatively large quantity of the residual norm is contributed from the thread that is responsible for communication, indicating that it may not be able to iterate enough to sufficiently minimize the error associated with the components that it is responsible for updating. In other words, Fig. 4(a) shows that when a communicating thread does not match the iteration rate of non-communicating threads, a disproportionately large amount of the solution error originates from the communicating thread. However, Fig. 4(b) suggests that this problem can be mitigated by increasing the swap period. The predictive model discussed in Section 7 aims to investigate appropriate swap-period-work-scaling-factor pairings for a given problem.

7. Predictive Model

The predictive model utilizes the time distributions generated from empirical data for subdomain size $n = 200$ (shown in Fig. 5) that correspond to key operations of the implementations, 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 norm component.

The communicating OpenMP thread additionally must

(j) update the subdomain residual norm,
(k) copy the subdomain boundary values, and
(l) update the subdomain halo values,

and the MPI master rank

(m) communicates and updates the global error, the sum of the subdomain residual norms, and
(n) communicates and updates the halo values.

Using MATLAB, the collected empirical data was fit to several distributions, each of type KERNEL, a nonparametric 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.

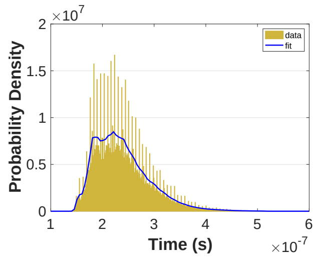
Model communication data is useful for predicting behavior of a calculation with any number of subdomains of size $n = 200$. Model communication times are from the perspective of the master MPI process and have no knowledge of communication delay due to queueing. The predictive model effectively simulates communication delays due to multiple MPI worker processes attempting to communicate with the master process.

Similar to the method described in [21], a MATLAB script models thread computation and master-worker communication for WS. The predictive model does not simulate or estimate computations, but simulates times to complete operations in the WS implementation. Similar to the WS implementation, the model is assigned a number of MPI processes and a number of OpenMP threads for each process; each process has a communicating thread. Unlike the WS implementation, the simulation runs serially; it uses loops and logic statements to simulate the parallel behavior of WS. In the outermost loop, for a given application parameter configuration, code that models the behavior of the MPI master process sorts through messages from workers, works on the chronologically first message, and assigns the thread associated with the earliest message to the next appropriate phase in its iteration, e.g. initiating a subdomain halo swap, or updating subdomain halo values after swap communication. After the master-side communication has completed, any worker communicating threads that are not waiting for a response from the master process begin, continue, or complete an iteration. After a communicating thread has completed an iteration, each non-communicating thread belonging to the same MPI process iterates until its simulation time is equal to or greater than that of the communicating thread. The simulation terminates when the master process clock meets or exceeds the set run time. Model outputs are simulation times and number of iterations completed for each thread.

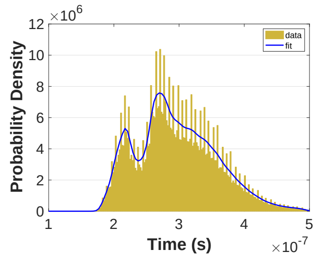
Based on the results and discussion given in Section 5, it is reasonable to expect the performance of distributed asynchronous Jacobi to be improved when the iteration rate of the non-communicating (i.e. non-master) thread is roughly equal to the iteration rate of the communicating thread inside each subdomain. Due to the extra time required for the communicating thread to update the master process, it is unlikely that this balance will be achieved when the communicating thread is responsible for updating as large a portion of the subdomain as the other threads. In order to find this point of balance, the simulation of the predictive model developed here was used.

7.1 Model Results

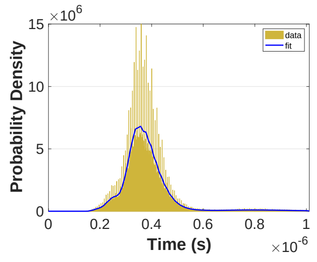
Figure 6 and Fig. 7 depict reasonable model output, considering the relationship between work-scaling factor and swap period. Further, results are comparable to actual performance on the Old Dominion University Turing cluster, specifically iteration rates seen in Fig. 3. Differences between Fig. 6 and Fig. 3 may be attributable to different hardware used for data collection. Therefore the model may be useful for the selection of good application parameters. In particular, when trying to find the amount by which to scale the work given to the communicating thread in each subdomain, it is useful to look at the ratio of the iteration rate I_m of the communicating thread compared with the iteration rate I_{nm} of the non-master threads, which is an average over all the other threads in the subdomain. In the simulation, the communicating thread computes an area of the subdomain ranging from 8% to 100% of n/n_p rows for the swap period P_{swap} ranging from 1 to 16. Note that not all swap periods are able to achieve this optimal balance, and that the swap periods that achieve this balance have a corresponding work-scaling factor where its Fig. 8 swap-period trajectory intersects $y = 1$.



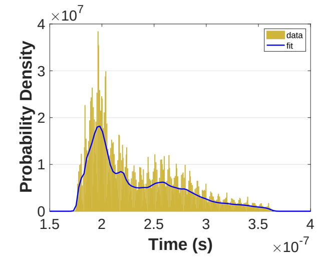
(a) OpenMP thread copy time, subdomain left and right halo values



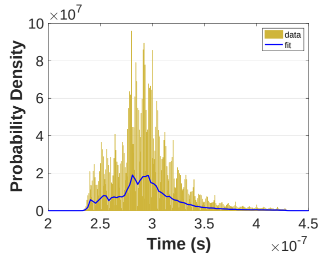
(b) OpenMP thread copy time, subdomain top and bottom halo values



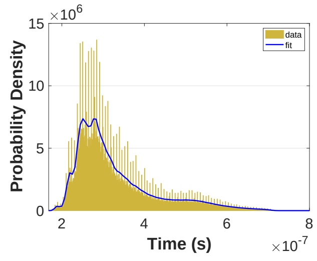
(c) OpenMP thread copy time, interior boundary row neighbor values



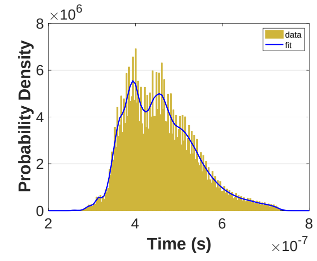
(d) OpenMP thread compute time, boundary row values



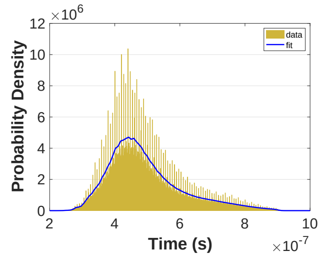
(e) OpenMP thread compute and update time, interior row values



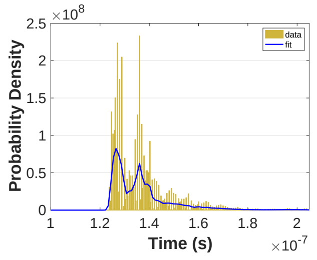
(f) OpenMP thread update time, subdomain left and right boundary values



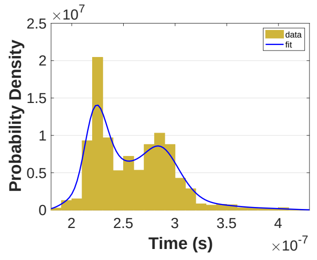
(g) OpenMP thread update time, subdomain top and bottom boundary values



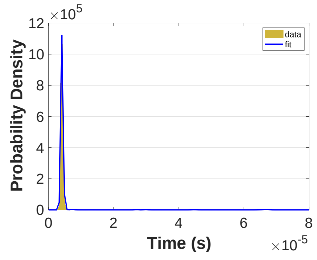
(h) OpenMP thread update time, interior boundary row values



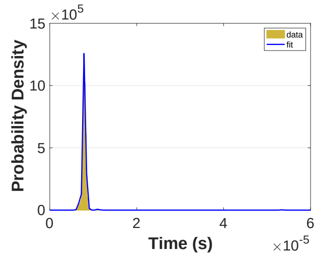
(i) OpenMP thread compute time, thread residual norm component



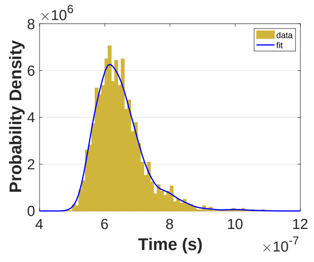
(j) OpenMP master thread compute time, subdomain residual norm



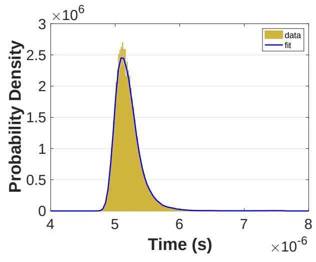
(k) OpenMP master thread copy time, subdomain boundary values



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

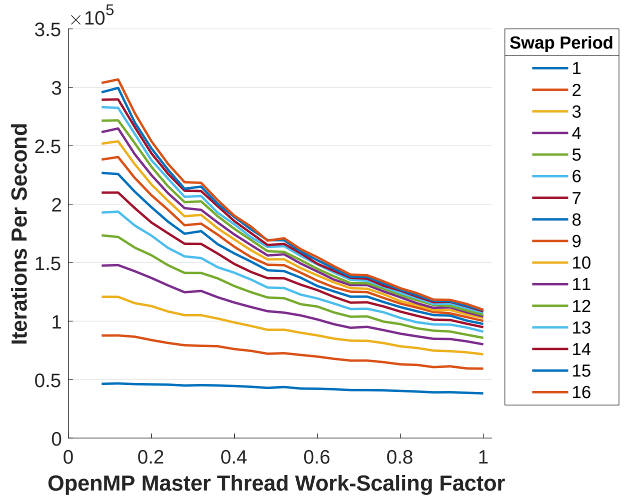


(m) MPI master rank communication and update time, global residual norm

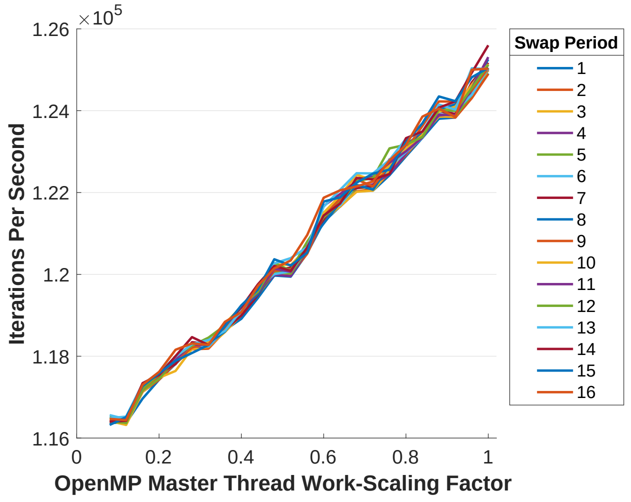


(n) MPI master rank communication and update time, halo swap

Fig. 5: Empirically-measured histogram data and operation-time distributions used in predictive model.



(a) Rates, master thread



(b) Rates, non-master threads

Fig. 6: Predictive model results for work-scaling implementation. Iteration rates of master and non-master threads, as a function of swap period and work-scaling factor.

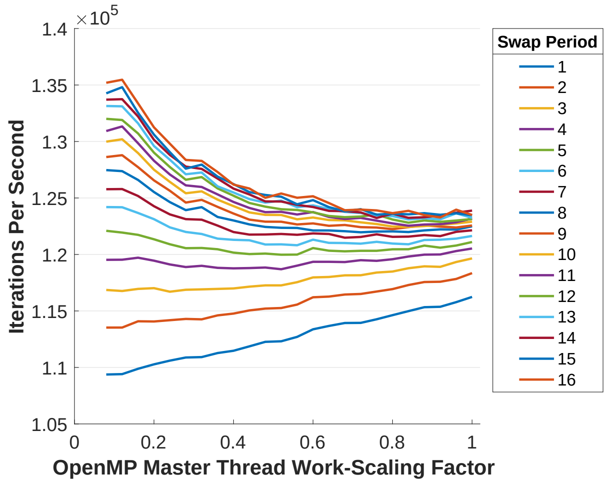


Fig. 7: Iteration rates of all threads, as a function of swap period and work-scaling factor.

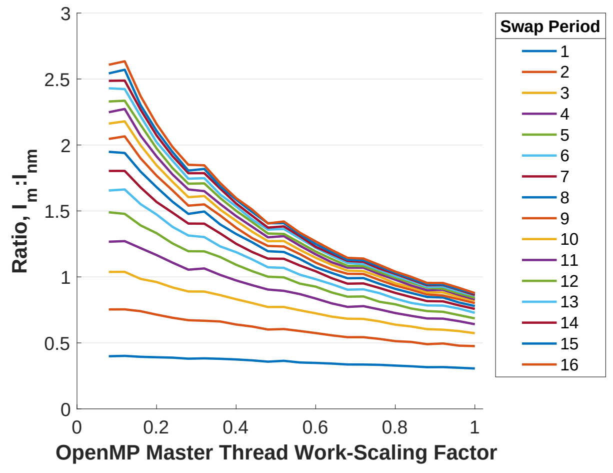


Fig. 8: Ratio of iteration rates of master and non-master threads, as a function of swap period P_{swap} and work-scaling factor.

8. Conclusions and Future Work

Modifying the halo swap strategy of the base hybrid-parallel model WS1 provides significant performance improvements and equalizes error quantity in subdomain regions. Generally speaking, the asynchronous implementations outperform the synchronous implementation for the problem studied, however further optimization is possible in both cases. While focusing on a single problem allows for efficient implementations, in the future the Jacobi solver could be generalized to solve an arbitrary problem instead of being tuned specifically to the Laplacian. This would allow data to be collected over a range of different input parameters that could be used in a generalized predictive model, such that problem-specific parameters may be considered or queried.

Another avenue of future work lies in the direction of experimenting with different asynchronous solvers. For example, [29] proposed a parallel asynchronous Southwell relaxation method that shows small communication cost. Additionally, the use of one-sided remote memory access [17] may help improve the implementations shown here.

Acknowledgments

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

References

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

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

[3] Steve Ashby, PETE Beckman, Jackie Chen, Phil Colella, Bill Collins, Dona Crawford, Jack Dongarra, Doug Kothe, Rusty Lusk, Paul Messina, et al. The opportunities and challenges of exascale computing–summary report of the advanced scientific computing advisory committee (ascac) subcommittee. *US Department of Energy Office of Science*, 2010.

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

[5] Jacques M Bahi, Sylvain Contassot-Vivier, and Raphaël Couturier. Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational grid. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 9–pp. IEEE, 2003.

[6] Jacques M Bahi, Sylvain Contassot-Vivier, and Raphaël Couturier. Performance comparison of parallel programming environments for implementing aiac algorithms. *The Journal of Supercomputing*, 35(3):227–244, 2006.

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

[8] Iain Bethune, J Mark Bull, Nicholas J Dingle, and Nicholas J Higham. Investigating the Performance of Asynchronous Jacobi's Method for Solving Systems of Linear Equations. *To appear in International Journal of High Performance Computing Applications*, 2011.

[9] Iain Bethune, J Mark Bull, Nicholas J Dingle, and Nicholas J Higham. 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, 2014.

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

[11] Steven C Chapra and Raymond P Canale. *Numerical methods for engineers*, volume 2. McGraw-Hill New York, 1998.

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

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

[14] Douglas V De Jager and Jeremy T Bradley. Extracting state-based performance metrics using asynchronous iterative techniques. *Performance Evaluation*, 67(12):1353–1372, 2010.

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

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

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

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

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

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

[21] Erik Jensen and M. Sosonkina. Modeling a task-based matrix-matrix multiplication application for resilience decision making. In *Proceedings of the 26th High Performance Computing Symposium*, HPC '18, 2018.

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

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

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

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

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

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

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

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

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