ASYNCHRONOUS VERSIONS OF JACOBI, MULTIGRID, AND CHEBYSHEV SOLVERS

A Dissertation
Presented to
The Academic Faculty

By

Jordi Wolfson-Pou

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Computational Science and Engineering

Georgia Institute of Technology

August 2020

ASYNCHRONOUS VERSIONS OF JACOBI, MULTIGRID, AND CHEBYSHEV SOLVERS

Approved by:

Dr. Edmond Chow, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Tobin Isaac
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Felix Herrmann
School of Earth and Atmospheric Sciences
Georgia Institute of Technology

Dr. Ulrike Meier Yang
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

Date Approved: June 2, 2020

ACKNOWLEDGEMENTS

I would first like to express my deepest gratitude to my advisor Edmond Chow for his wisdom and support through my PhD journey. The completion of my PhD would be impossible without his expertise, advice, and guidance. I would also like to thank my committee members Ulrike Meier Yang, Felix Herrmann, Richard Vuduc, and Tobin Isaac for their insightful feedback on my dissertation.

My sincere thanks goes to all my collaborators and mentors. On the topic of multigrid methods, the mentoring I received from Ulrike Meier Yang and Stephen McCormick was invaluable. On the topic of asynchronous methods, Daniel Szyld, Christian Glusa, Faycal Chaouqui, Erik Boman, Ichitaro Yamazaki, and Sivasankaran Rajamanickam all helped me in important ways.

I would like to thank all the professors who were instructors for the courses I took at Georgia Tech. All these professors were excellent instructors and went out of their way to help me. I am also thankful for the mentorship from Richard Vuduc who helped me navigate through the first year of my PhD.

I would like to thank Lawrence Livermore National Laboratory for allowing me to use their state-of-the-art HPC resources and hiring me as an intern for three summers. As an intern, I was able to work with leading experts in my field on compelling research projects.

I am grateful for my friends and fellow graduate students Fred Hohman, Robert Pienta, Yacin Nadji, Caleb Robinson, Muyuan Li, Aradhya Biswas Paritosh Ramanan, Amrita Gupta, Elias Khalil, Patrick Flick, and Casey Battaglino who have all helped me in different ways. I will always be grateful for their friendship.

Lastly, and most importantly, I will always be grateful for my father David, mother Jocelyn, brother Jesse, and partner Jamie for their unwavering love, support, and patience.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics pro-

gram under Award Number DE-SC-0012538. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Iterative methods are commonly used for solving large, sparse systems of linear equations on parallel computers. Implementations of parallel iterative solvers contain kernels (e.g., parallel sparse matrix-vector products) in which parallel processes alternate between phases of computation and communication. Standard software packages use synchronous implementations where there are one or more synchronization points per iteration. These synchronization points occur during communication phases where each process sends data to other processes and idles until all data needed for the next iteration is received. Synchronization points scale poorly on massively parallel machines and may become the primary bottleneck for future exascale computers. This calls for research and development of asynchronous iterative methods, which is the subject of this dissertation.

In asynchronous iterative methods there are no synchronization points. This means that, after a phase of computation, processes immediately proceed to the next phase of computation using whatever data is currently available. Since the late 1960s, research on asynchronous methods has primarily considered basic fixed-point methods, e.g., Jacobi, where proving asymptotic convergence bounds has been the focus. However, the practical behavior of asynchronous methods is not well understood, and asynchronous versions of certain fast-converging solvers have not been developed. This dissertation focuses on studying the practical behavior of asynchronous Jacobi, developing new communication-avoiding asynchronous iterative solvers, and introducing the first asynchronous versions of multigrid and Chebyshev.

To better understand the practical behavior of asynchronous Jacobi, we examine a model of asynchronous Jacobi where communication delays are neglected. We call this model simplified asynchronous Jacobi. Simplified asynchronous Jacobi can be used to model asynchronous Jacobi implemented in shared memory or distributed memory with fast communication networks. We analyze simplified asynchronous Jacobi for linear sys-

tems where the coefficient matrix is symmetric positive-definite and compare our analysis to experimental results from shared and distributed memory implementations. We present three important results for asynchronous Jacobi: it can converge when synchronous Jacobi does not, it can reduce the residual norm when some processes are delayed, and its convergence rate can increase with increasing parallelism.

We develop new asynchronous communication-avoiding methods using the idea of the sequential Southwell method. In the sequential Southwell method, which converges faster than Gauss-Seidel, the component of the residual with the largest residual in absolute value is relaxed during each iteration. We use the idea of choosing large residual values to create communication-avoiding parallel methods, where residual values of communication neighbors are compared rather than computing a global maximum. We present three methods: the Parallel Southwell, Distributed Southwell, and Stochastic Parallel Southwell methods. All our methods converge faster than Jacobi and use less communication.

We introduce the first asynchronous multigrid methods. We use the idea of additive multigrid where smoothing on all grids is carried out concurrently. We present models of asynchronous additive multigrid and use these models to study the convergence properties of asynchronous multigrid. We also introduce algorithms for implementing asynchronous multigrid in shared and distributed memory. Our experimental results show that asynchronous multigrid can exhibit grid-size independent convergence and can be faster than classical multigrid in terms of wall-clock time.

Lastly, we present the first asynchronous Chebyshev methods. We present models of Jacobi-preconditioned asynchronous Chebyshev. We use a little-known version of the BPX multigrid preconditioner where BPX is written as Jacobi on an extended system, which makes BPX convenient for asynchronous execution within Chebsyhev. Our experimental results show that asynchronous Chebyshev is faster than its synchronous counterpart in terms of both wall-clock time and number of iterations.

CHAPTER 1
INTRODUCTION

1.1 Overview and Motivation

Sparse linear systems of the form $Ax = b$ appear in many scientific computing problems, e.g., numerical solutions to partial differential equations, where we are solving for the $n \times 1$ vector x, A is an $n \times n$ sparse matrix, and b is a $n \times 1$ vector. High-performance parallel computers are typically used to solve $Ax = b$ since n is typically large for most modern problems. Methods for solving $Ax = b$ fall into two categories: iterative and exact. In exact methods, a solution is achieved that is accurate up to machine precision. However, exact methods have high memory costs, and many applications do not require machine precision accuracy from the solver.

In iterative solvers, an initial guess is used to generate a sequence of improved approximations to x. Iterative solvers have significantly lower memory costs than exact solvers and can be stopped at any iteration once some desired convergence criterion is met. Iterative solvers can typically be written in terms of parallel sparse matrix-vector products and parallel dense inner products. A method such as Jacobi only requires one matrix-vector product per iteration where as a faster method such as preconditioned Conjugate Gradient requires one or more matrix-vector products and an inner product. Therefore, iterative methods alternate between phases of computation and communication, where there may be many computation and communication phases per iteration. In the communication phase, each parallel process sends data to other processes and then idles until data is received before proceeding to the next iteration. These communication phases are *synchronization points* where all processes must reach these points before moving to the next phase. While communication and computation can be overlapped to some degree, the overhead of syn-

chronization may still be high, especially on massively parallel heterogeneous machines, where load may not be perfectly balanced. There have been numerous U.S. Department of Energy reports that suggest that asynchronous methods will be important for exascale machines [1, 2, 3, 4, 5, 6].

In asynchronous iterative methods, synchronization points are removed. Specifically, processes simply proceed to the next computation phase using the most recently received information from other processes. Asynchronous methods can be defined through mathematical models. A typical synchronous iterative method is written such that all components of $x^{(t)}$ are updated at each iteration using information from the previous iteration. In classical mathematical models of asynchronous iterative methods, subsets of components of $x^{(t)}$ are updated at each iteration using information from a mixture of previous iterations. These mathematical models are closely related to randomized methods [7, 8] where subsets of components are randomly selected at each iteration. Mathematical models of asynchronous iterative methods have been studied since the late 1960s [9]. This research has primarily focused on asynchronous versions of basic iterative methods, such as the Jacobi method, where convergence theory has been studied [9, 10, 11, 12, 13, 14, 15] and implementations have been developed [16, 17].

While many basic iterative methods are highly parallel and easy to execute asynchronously when compared with other iterative methods, they typically converge more slowly than optimal methods such as Krylov or multigrid methods. It is still unknown (and may never be possible) how methods that involve orthogonalization, which require inner products, can be made asynchronous. To be clear, we are not including pipelined methods [18, 19] in our definition of asynchronous iterative methods since synchronization from inner products is "hidden" behind the computation of matrix-vector products rather than removed altogether. More recently, asynchronous optimized Schwarz methods have been developed in an effort to develop fast-converging linear solvers [20]. Asynchronous methods have also recently gained attention for solving optimization problems where asyn-

chronous versions of stochastic gradient descent have been developed [21, 22]. While we consider an $n \times n$ linear system in this dissertation, asynchronous Kaczmarz methods have also been studied for solving non-square linear systems [8]. In general, there are still open questions about the practical behavior of asynchronous methods, and asynchronous versions of various fast-converging iterative solvers must be developed.

There are two important topics studied in this dissertation: the practical behavior of asynchronous iterative methods and the development of new asynchronous versions of certain fast-converging iterative solvers. For studying the practical behavior of asynchronous methods, we focus on the asynchronous Jacobi method where we show that in practice, asynchronous execution of Jacobi actually improves the convergence properties of Jacobi. We also introduce Southwell methods, which can be thought of as communication-avoiding variants of the block Jacobi method. However, just like the Jacobi method, Southwell methods converge much more slowly than more commonly used optimal methods such as multigrid and Chebyshev methods. We therefore introduce asynchronous multigrid and Chebyshev methods. Multigrid methods are commonly used both as standalone solvers and as preconditioners. In this dissertation, we consider both cases. As a preconditioner, we use asynchronous multigrid within asynchronous Chebyshev, where our asynchronous multigrid preconditioned Chebyshev solver is able to outperform its synchronous counterpart.

1.2 Outline and Contributions

In Chapter 2, we study a model of the asynchronous Jacobi method where communication delays are neglected, which we call the *simplified asynchronous Jacobi* method. While not completely realistic, this simplified model can be used to approximate asynchronous Jacobi in shared memory or distributed memory with fast communication networks, where data transfers are fast compared with computation time. More importantly, simplifying the model to neglect communication delays allows us to write the update of rows in each

iteration of simplified asynchronous Jacobi using a *propagation matrix*, which is similar in concept to an iteration matrix. A propagation matrix has properties that give us insight into the transient behavior of simplified asynchronous Jacobi. The purpose of this chapter is to show that, in practice, asynchronous Jacobi often has better convergence properties than synchronous Jacobi, even though classical convergence theory for asynchronous methods suggests that asynchronous methods have poor convergence properties compared with their synchronous counterparts. We also outline shared and distributed memory implementations of Jacobi, where we use OpenMP in the shared memory case and one-sided MPI with passive target completion in the distributed memory case.

In Chapter 3, we develop Southwell-type methods which can be executed asynchronously and can be thought of as communication-avoiding versions of Jacobi. Our methods are based on the original Southwell method, which is sequential, where the row with the largest residual norm is relaxed each iteration. Southwell can converge faster than the Gauss-Seidel method in terms of the number of relaxations. Since each relaxation is associated with communication, Southwell also requires less communication. However, Southwell is sequential by definition and requires global communication after every relaxation step to determine the row with the largest residual. We present three new methods that are based on the idea of using large residuals to decide whether a row should be relaxed: the Parallel Southwell, Distributed Southwell, and Stochastic Parallel Southwell methods.

In Parallel Southwell, equation i is relaxed if it has the largest residual compared to the residuals of the equations coupled to variable i. This method allows equations to be relaxed simultaneously, and does not require global communication to determine the equation with the largest residual. While Parallel Southwell is suitable for shared memory, in distributed memory, deadlock can occur if processes use stale values of the residual norms of their communication neighbors. This deadlock issue motivates the Distributed Southwell and Stochastic Parallel Southwell methods. In Distributed Southwell, each process stores the estimates to its own residual norm that are held by its neighbors. These estimates

are used to avoid deadlock. Additionally, each process locally computes better estimates of residual norms that belong to its neighbors without any communication. Importantly, Distributed Southwell also uses new techniques to reduce communication compared to Parallel Southwell in distributed memory. However, Distributed Southwell still requires extra communication in order to avoid deadlock. In Stochastic Parallel Southwell, each process uses residual estimates to construct a probability of relaxing its rows. This does not require additional deadlock-avoiding communication, and Stochastic Parallel Southwell converges with a similar rate to Parallel and Distributed Southwell.

In Chapter 4, we present the first asynchronous multigrid methods. Since it is unclear how to execute classical multiplicative multigrid methods asynchronously, we use additive multigrid methods. We define new models of asynchronous multigrid which serve to define what asynchronous multigrid is. In classical asynchronous iterative methods subsets, of rows are relaxed at each time instant whereas in asynchronous multigrid, subsets of grids update the current approximation to the solution at each time instant. We outline shared and distributed memory implementations of asynchronous multigrid, where we use two-sided non-blocking MPI functions in our distributed memory implementation. We also introduce a new additive multigrid method that can be executed asynchronously, the AFACj method, which is a variation of the AFACx method.

In Chapter 5, we introduce the first descriptions and implementations of an asynchronous version of the Chebyshev iterative method [23], where we use an asynchronous multigrid method as a preconditioner within asynchronous Chebyshev. Instead of the asynchronous multigrid methods presented in Chapter 4, we are able to use a much simpler multigrid method called BPX [24]. The asynchronous multigrid methods proposed in Chapter 4 had to be based on a method that could converge on its own, i.e., as a standalone solver. When multigrid is used as a preconditioner, standalone convergence is not necessary, which allows us to use BPX, the simplest additive multigrid method. To create an asynchronous version of BPX, we use a formulation of the method based on an extended

matrix [25]. The standard formulation contains many types of operations (smoothing, restriction, prolongation, coarse grid solves) and it is very complicated to run these operations asynchronously with each other. In contrast, the extended matrix formulation is similar to using Jacobi iterations with a special "extended" matrix, which are easy to perform asynchronously.

CHAPTER 2
MODELING THE ASYNCHRONOUS JACOBI METHOD WITHOUT COMMUNICATION DELAYS

The practical and transient behavior of asynchronous Jacobi is not well understood in the literature. In this chapter, we study a model of the asynchronous Jacobi method with no communication delays, which we call the *simplified asynchronous Jacobi* method. While not completely realistic, this simplified model can be used to approximate asynchronous Jacobi in shared memory or distributed memory with fast communication networks, where data transfers are fast compared with computation time. More importantly, simplifying the model to neglect communication delays allows us to write the update of rows in each iteration of simplified asynchronous Jacobi using a *propagation matrix*, which is similar in concept to an iteration matrix. A propagation matrix has properties that give us insight into the transient behavior of simplified asynchronous Jacobi. By analyzing these matrices, and by experimenting with shared and distributed memory implementations, we show some important convergence results for asynchronous Jacobi.

2.1 Background

2.1.1 The Jacobi and Gauss-Seidel Methods

A general stationary iterative method for solving the sparse linear system $Ax = b$ can be written as

$$x^{(t+1)} = Gx^{(t)} + f, \tag{2.1}$$

where the recurrence is started with an initial approximation $x^{(0)}$. We define the update of the i^{th} component from $x_i^{(t)}$ to $x_i^{(t+1)}$ as the *relaxation* of row i.

If the exact solution is x^*, then we can write the *error* $e^{(t)} = x^* - x^{(t)}$ at iteration t as

$$e^{(t+1)} = Ge^{(t)}, \tag{2.2}$$

where G is the *error iteration matrix*. It is well known that a stationary iterative method will converge to the exact solution for any $x^{(0)}$ as $k \to \infty$ if the spectral radius $\rho(G) < 1$. Analyzing $\|G\|$ is also important since the spectral radius only tells us about the asymptotic behavior of the error. In the case of normal iteration matrices, the error decreases monotonically in the consistent norm if $\rho(G) < 1$ since $\rho(G) \leq \|G\|$. If G is not normal, $\|G\|$ can be ≥ 1 when $\rho(G) < 1$. This means that although convergence to the exact solution will be achieved, the reduction in the norm of the error may not be monotonic.

Stationary iterative methods are sometimes referred to as *splitting* methods where a splitting $A = M - N$ is chosen with nonsingular M. Equation 2.1 can be written as

$$x^{(t+1)} = (I - M^{-1}A)x^{(t)} + M^{-1}b, \tag{2.3}$$

where $G = I - M^{-1}A$. Just like in Equation 2.2, we can write

$$r^{(t+1)} = Hr^{(t)}, \tag{2.4}$$

where the *residual* is defined as $r^{(t)} = b - Ax^{(t)}$ and $H = I - AM^{-1}$ is the *residual iteration matrix*.

For the Gauss-Seidel method, $M = L$, where L is the lower triangular part of A, and for the Jacobi method, $M = D$, where D is the diagonal part of A. Gauss-Seidel is an example of a multiplicative relaxation method, and Jacobi is an example of an additive relaxation method. In practice, Jacobi is one of the few stationary iterative methods that can be efficiently implemented in parallel since the inversion of a diagonal matrix is a highly parallel operation. In particular, for some machine with n processors, all n rows can be

relaxed completely in parallel with processes $p_1, \ldots, p_n$ using only information from the previous iteration. However, Jacobi often does not converge, even for symmetric positive definite (SPD) matrices, a class of matrices for which Gauss-Seidel always converges. When Jacobi does converge, it can converge slowly, and usually converges more slowly than Gauss-Seidel.

2.1.2 Asynchronous Iterative Methods

We now consider a general model of an asynchronous stationary iterative method as presented in Chapter 5 of [26]. For simplicity, let us consider n processes, i.e., one process per row of A. The general form of a stationary iterative method as defined in Equation 2.1 can be thought of as *synchronous*. In particular, all elements of $x^{(t)}$ must be computed before iteration $t+1$ starts. Removing this requirement results in an asynchronous method, where each process relaxes its row using whatever information is available. An asynchronous stationary iterative method can be written element-wise as

$$x_i^{(t+1)} = \begin{cases} \sum_{j=1}^{n} G_{ij} x_j^{(s_{ij}(t))} + f_i, & \text{if } i \in \Psi(t), \\ x_i^{(t)}, & \text{otherwise.} \end{cases} \tag{2.5}$$

The set $\Psi(t)$ is the set of rows that are relaxed at time instant t. The mapping $s_{ij}(t)$ denotes the components of other rows that process i reads from memory. The following assumptions are standard for the convergence theory of Equation 2.5:

1. The mapping $s_{ij}(t) \leq k$. This means no future information is read from memory.

2. As $k \to +\infty$, $s_{ij}(t) \to +\infty$. This means rows will eventually read new information from other rows.

3. As $k \to +\infty$, the number of times i appears in $\Psi(t) \to +\infty$. This means that all rows eventually relax in a finite amount of time.

2.2 Related Work

An overview of asynchronous iterative methods can be found in Chapter 5 of [26]. Reviews of the convergence theory for asynchronous iterative methods can be found in [26, 27, 28, 29]. Asynchronous iterative methods were first introduced as *chaotic relaxation* methods by Chazan and Miranker [9]. This pioneering paper provided a definition for a general asynchronous iterative method with various conditions, and the main result of the paper is that for a given stationary iterative method with iteration matrix G, if $\rho(|G|) < 1$, then the asynchronous version of the method will converge. Here, $|G|$ is the element-wise absolute value of G. Other researchers have expanded on suitable conditions for asynchronous methods to converge using different asynchronous models [10, 13, 11, 12, 30, 31, 32]. One of these models introduces the idea of propagation matrices, which is what we analyze [31]. There are also papers that show that asynchronous methods can converge faster than their synchronous counterparts [33, 34]. In [33], it is shown that for monotone maps, asynchronous methods are at least as fast as their synchronous counterparts, assuming that all components eventually update. This was also shown in [34], and was extended to contraction maps. The speedup of asynchronous Jacobi was studied in [35] for random 2×2 matrices, where the main result is that, most of the time, asynchronous iterations do not improve the convergence compared with synchronous. This result is specific for 2×2 matrices, and we will discuss in Section 2.3.3 why speedup is not often suspected in this case. In [32], the authors use the idea of a directed acyclic graph (DAG) to show that the convergence rate of asynchronous Jacobi is exponential when the synchronous iteration matrix is non-negative. While the authors in [32] do not prove anything for the non-negative case, examples in which asynchronous Jacobi converges when synchronous Jacobi does not are shown, which we also explore in this chapter. The authors in [32] also provide numerical experiments that show that asynchronous Jacobi has a higher convergence rate for certain problems. As in [31], the idea of propagation matrices are used in [32].

Experiments using asynchronous methods have given mixed results, and it is not clear whether this is implementation or algorithm specific. It has been shown that in shared memory, asynchronous Jacobi can be significantly faster [10, 17]. Jager and Bradley reported results for several distributed implementations of asynchronous inexact block Jacobi (where blocks are solved using a single iteration of Gauss-Seidel) implemented using "the MPI-2 asynchronous communication framework" [36], which may refer to one-sided MPI. They showed that asynchronous "eager" Jacobi can converge in fewer relaxations and less wall-clock time. Their eager scheme can be thought of as semi-synchronous, where a process updates its rows only if it has received new information. Bethune et al. reported mixed results for several different implementations of asynchronous Jacobi [17]. The "racy" scheme presented in [17] is what we consider in this chapter, but we use one-sided MPI with passive target completion where as Bethune et al. used SHMEM and two-sided MPI. The results in [17] show that asynchronous Jacobi implemented with MPI was faster in terms of wall-clock time, but in some experiments with large core counts, synchronous Jacobi was significantly faster.

We also note that some research has been dedicated to supporting portable asynchronous communication for MPI, including the JACK and JACK2 APIs [37, 38], and Casper [39], which provides asynchronous progress control in certain cases. We are not using any of these tools in our implementations.

2.3 Asynchronous Iterative Methods Without Communication Delays

2.3.1 Mathematical Formulation

If there are no communication delays, and processes are only delayed in their computation (some processes take longer than others to relax their rows), we can write Equation 2.5 as

$$x_i^{(t+1)} = \begin{cases} \sum_{j=1}^{n} G_{ij} x_j^{(t)} + f_i, & \text{if } i \in \Psi(t), \\ x_i^{(t)}, & \text{otherwise.} \end{cases} \tag{2.6}$$

We define this as the *simplified* asynchronous iterative method model, and for asynchronous Jacobi, we define this as the *simplified asynchronous Jacobi* model [14, 15]. We can now write an asynchronous iterative method in matrix form as

$$x^{(t+1)} = (I - \hat{M}^{(t)} A) x^{(t)} + \hat{M}^{(t)} b \tag{2.7}$$

where

$$\hat{M}^{(t)}(i,:) = \begin{cases} M^{-1}(i,:), & \text{if } i \in \Psi(t), \\ 0, & \text{otherwise,} \end{cases} \tag{2.8}$$

where $\hat{M}^{(t)}(i,:)$ is row i of $\hat{M}^{(t)}$, and $M^{-1}(i,:)$ is row i of M^{-1}. Similar to the iteration matrix, we define the error and residual *propagation matrices* as

$$\hat{G}^{(t)} = I - \hat{M}^{(t)} A \quad \text{and} \quad \hat{H}^{(t)} = I - A\hat{M}^{(t)}, \tag{2.9}$$

respectively.

For $\hat{G}^{(t)}$ and $\hat{H}^{(t)}$ in simplified asynchronous Jacobi, $\hat{M}^{(t)}$ is the diagonal matrix $\hat{D}^{(t)}$

where

$$\hat{D}_{ii}^{(t)} = \begin{cases} 1/A_{ii}, & \text{if } i \in \Psi(t), \\ 0, & \text{otherwise.} \end{cases} \tag{2.10}$$

It is important to notice the structure of $\hat{G}^{(t)}$ and $\hat{H}^{(t)}$ matrices. For row i that is not relaxed at time instant t, row i of $\hat{G}^{(t)}$ is zero except for a one in the diagonal position of that row. Similarly, column i of $\hat{H}^{(t)}$ is zero except for a one in the diagonal position of that column. We can construct the error propagation matrix by starting with G and "replacing" rows of G with unit basis vectors if a row is not in $\Psi(t)$. Similarly, we replace columns of H to get the residual propagation matrix.

An example of a sequence of asynchronous relaxations is shown in Figure 2.1 for the simplified model. In this example, four processes, $p_1, \ldots, p_4$, are each responsible for a single row, and relax just once. The red dots (except at $k = 0$) denote relaxations and the blue arrows denote data transfer needed for relaxations. Asynchronous iteration count moves from left to right. There are three iterations in this example, which means we have three sets $\Phi(1) = \{4\}$, $\Phi(2) = \{1, 2\}$, and $\Phi(3) = \{3\}$. This gives us the three error propagation matrices for simplified asynchronous Jacobi,

$$\hat{G}^{(1)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \times & \times & 0 \end{bmatrix}, \; \hat{G}^{(2)} = \begin{bmatrix} 0 & \times & \times & 0 \\ \times & 0 & 0 & \times \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \; \hat{G}^{(3)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \times & 0 & 0 & \times \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where the × symbol denotes a non-zero value.

2.3.2 Connection of Simplified Asynchronous Jacobi to an Inexact Multiplicative Block Relaxation Method

Simplified asynchronous Jacobi can be viewed as an inexact multiplicative block relaxation method, where the number of blocks and block sizes change at every iteration. A block

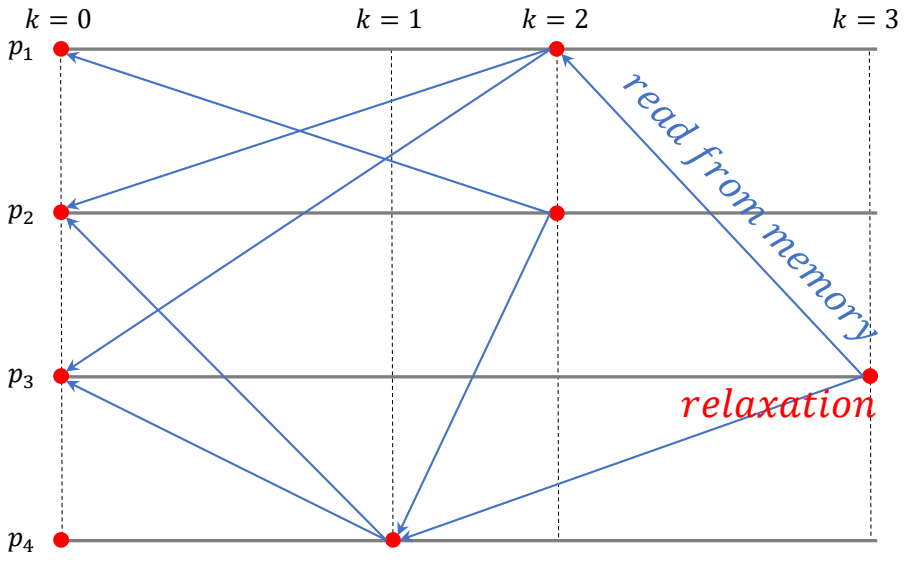


Figure 2.1: Example of four processes carrying out three iterations of an asynchronous iterative method without communication delays. Relaxations are denoted by red dots, and information used for relaxations is denoted by blue arrows.

corresponds to a coupled set of equations that are relaxed simultaneously. By "inexact" we mean that Jacobi relaxations are applied to the blocks of equations (rather than an exact solve, for example). By "multiplicative," we mean that not all blocks are relaxed at the same time, i.e., the updates build on each other multiplicatively like in the Gauss-Seidel method.

If a single row j is relaxed at time instant t, then

$$\hat{D}_{ii}^{(t)} = \begin{cases} 1/A_{ii}, & \text{if } i = j, \\ 0, & \text{otherwise.} \end{cases} \tag{2.11}$$

Relaxing all rows in ascending order of index is precisely Gauss-Seidel with natural ordering. For multicolor Gauss-Seidel, where rows belonging to an independent set (no rows in

the set are coupled) are relaxed in parallel, $\hat{D}^{(t)}$ can be expressed as

$$\hat{D}_{ii}^{(t)} = \begin{cases} 1/A_{ii}, & \text{if } i \in \Gamma, \\ 0, & \text{otherwise.} \end{cases} \tag{2.12}$$

where Γ is the set of indices belonging to the independent set. Similarly, Γ can represent a set of independent blocks, which gives the block multicolor Gauss-Seidel method.

2.3.3 Simplified Asynchronous Jacobi Can Reduce The Residual When Some Processes are Delayed in Their Computation

For this section only, we will assume A is weakly diagonally dominant (W.D.D.), i.e., $|A_{ii}| \geq \sum_{j \neq i} |A_{ij}|$ for all $i = 1, \ldots, n$ and thus $\rho(G) \leq 1$. Then the error and residual for simplified asynchronous Jacobi monotonically decreases in the infinity and L1 norms, respectively.

In general, the error and residual do not converge monotonically for asynchronous methods (assuming the error and residual at snapshots in time are available). However, monotonic convergence is possible in the infinity and L1 norms for the error and residual, respectively, if the propagation matrices are bounded by one in these norms. This is guaranteed when A is W.D.D. A norm of one means that the error or residual does not grow but may still decrease. Such a monotonic convergence result may be useful to help detect convergence of the asynchronous method in a distributed memory setting.

The following theorem supplies the norm of the propagation matrices.

Theorem 1. *Let A be W.D.D. and at least one process is delayed in its computation at time instant t. Then* $\rho(\hat{G}^{(t)}) = \|\hat{G}^{(t)}\|_\infty = 1$ *and* $\rho(\hat{H}^{(t)}) = \|\hat{H}^{(t)}\|_1 = 1$ *for simplified asynchronous Jacobi.*

Proof. Let the number of equations be n, and let $\xi_1, \ldots, \xi_n$ be the n unit (coordinate) basis vectors. Without loss of generality, consider a single equation i to be not relaxed at time

instant t. The proof of $\|\hat{G}^{(t)}\|_\infty = 1$ is straightforward. Since row i in $\hat{G}^{(t)}$ is ξ_i^T, and since A is W.D.D., $\|\hat{G}^{(t)}\|_\infty = 1$. Similarly, for $\|\hat{H}^{(t)}\|_1$, column i is ξ_i and so $\|\hat{H}^{(t)}\|_1 = 1$. To prove $\rho(\hat{G}^{(t)}) = 1$, consider the splitting $\hat{G}^{(t)} = I + Y$, where I is the identity matrix. The matrix Y has the same elements as $\hat{G}^{(t)}$ except the diagonal is $\text{diag}(\hat{G}^{(t)}) - I$ and the i^{th} row of Y is all zeros. Since Y has a row of zeros, it must have nullity ≥ 1. Therefore, an eigenvector of $\hat{G}^{(t)}$ is $v = \text{null}(Y)$ with eigenvalue of 1 since $(I+Y)v = v$. To prove $\rho(\hat{H}^{(t)}) = 1$, it is clear that ξ_i is an eigenvector of $\hat{H}^{(t)}$ since column i of $\hat{H}^{(t)} = \xi_i$. Therefore, $\hat{H}^{(t)}\xi_i = \xi_i$. □

We can say that, asymptotically, asynchronous Jacobi will be faster than synchronous Jacobi because inexact multiplicative block relaxation methods are generally faster than additive block relaxation methods. However, it is not clear if the error will continue to reduce if the same processes are delayed in their computation for a long period of time (equivalently, if some rows are not relaxed for a long period of time). An important consequence of Theorem 1 is that the error will not increase in the infinity norm no matter what the error propagation matrix is, which is also is true for the L1 norm of the residual. A more important consequence is that any residual propagation matrix will decrease the L1 norm of the residual with high probability (for a large enough matrix). This is due to the fact that the eigenvectors of $\hat{H}^{(t)}$ corresponding to eigenvalues of one are unit basis vectors. Upon multiplying $\hat{H}^{(t)}$ by the residual many times, the residual will converge to a linear combination of the unit basis vectors, where the number of these unit basis vectors is equal to the number of rows that are not relaxed. Since the eigenvalues corresponding to these unit basis vectors are all one, components in the direction of the unit basis vectors will not change, and all other components of the residual will go to zero. The case in which the residual will not change is when these components are already zero, which is unlikely given that the residual propagation matrix is constantly changing.

In the case of 2×2 random matrices, which was studied in [35], relaxing the same row after immediately relaxing that row will not change the current approximation since the

error and residual propagation matrices have the form

$$\hat{G}^{(t)} = \begin{bmatrix} 1 & 0 \\ \alpha & 0 \end{bmatrix}, \quad \hat{H}^{(t)} = \begin{bmatrix} 1 & \alpha \\ 0 & 0 \end{bmatrix}, \tag{2.13}$$

if the first row is not relaxed, where $\alpha = A_{12}/A_{11}$.

Since the only information needed by row two comes from row one, row two cannot continue to change without new information from row one. For larger matrices, iterating while having a small number of rows are not relaxed will reduce the error and residual.

For larger matrices, how quickly the residual converges depends on the eigenvalues that do not correspond to unit basis eigenvectors. If these eigenvalues are very small in absolute value (i.e., close to zero), convergence will be quick, and therefore the error/residual will not continue to reduce if some rows continue to not relax. To gain some insight into the reduction of the error and residual, we can use the fact that the components of the current approximation that are not relaxed do not change with successive applications of the same propagation matrix.

As an example, consider just the first row to not be relaxed starting at time instant t. We can write the iteration as

$$e^{(t+1)} = \hat{G}^{(t)} e^{(t)} = \begin{bmatrix} 1 & o^T \\ g_1^{(t)} & \tilde{G}^{(t)} \end{bmatrix} \begin{bmatrix} e_1^{(t)} \\ \tilde{e}^{(t)} \end{bmatrix} \tag{2.14}$$

where o is the $(n-1)\times 1$ zero vector, $g_1^{(t)}$ is an $(n-1)\times 1$ vector, and $\tilde{G}^{(t)}$ is a $(n-1)\times(n-1)$ symmetric principal submatrix of the synchronous iteration matrix G. Since $\tilde{G}^{(t)}$ is a principal submatrix of G, which is symmetric since A is symmetrically scaled to have unit diagonal values, we can use the interlacing theorem to bound the eigenvalues of $\tilde{G}^{(t)}$ with eigenvalues of G. Specifically, if $\lambda_1, \ldots, \lambda_n$ are the eigenvalues of G, the i^{th} eigenvalue μ_i of $\tilde{G}^{(t)}$ can be bounded as $\lambda_i \leq \mu_i \leq \lambda_{i+1}$.

For the general case in which m rows are being relaxed at time instant t, we can consider

the system $P^{(t)}AP^{(t)^T}P^{(t)}x = P^{(t)}b$, which has the iteration

$$P^{(t)}x^{(t+1)} = P^{(t)}\hat{G}^{(t)}P^{(t)^T}P^{(t)}x^{(t)} + P^{(t)}\hat{D}^{(t)}P^{(t)^T}P^{(t)}b. \tag{2.15}$$

The matrix $P^{(t)}$ is a permutation matrix that is chosen such that all rows that are not being relaxed are ordered first, resulting in the propagation matrix and error

$$e^{(t+1)} = \hat{G}^{(t)}e^{(t)} = \begin{bmatrix} I & O^T \\ G_I^{(t)} & \tilde{G}^{(t)} \end{bmatrix} \begin{bmatrix} e_I^{(t)} \\ \tilde{e}^{(t)} \end{bmatrix}, \tag{2.16}$$

where I is the $(n-m)\times(n-m)$ identity matrix, O is the $m\times(n-m)$ zero matrix, $G_I^{(t)}$ is $m\times(n-m)$, and $\tilde{G}^{(t)}$ is $m\times m$. For an eigenvalue μ_i of $\tilde{G}^{(t)}$, $\lambda_i \le \mu_i \le \lambda_{i+n-m}$ for $i = 1,\ldots,m$. This means that convergence for the propagation matrix will be slow if the convergence for synchronous Jacobi is slow. In other words, if eigenvalues of G are spaced somewhat evenly, or if many eigenvalues are clustered near one, we can expect a similar spacing of the eigenvalues of $\tilde{G}^{(t)}$.

2.3.4 Simplified Asynchronous Jacobi Can Converge When Synchronous Jacobi Does Not

A well known result, known as early as Chazan and Miranker [9], is that if G is the iteration matrix of a synchronous method then $\rho(|G|) < 1$ implies that the corresponding asynchronous method converges. From the fact that $\rho(G) < \rho(|G|)$ for all matrices G, it appears that convergence of the asynchronous method is harder than convergence of the synchronous method. However, this is an asymptotic result only, and does not capture any transient behavior. The following theorem provides a condition on $\tilde{G}^{(t)}$ that results in the decrease of the A-norm of the error at iteration t for the simplified asynchronous Jacobi method.

Theorem 2. *Let A be SPD and symmetrically scaled to have unit diagonal values. For*

simplified asynchronous Jacobi, if $\rho(\tilde{G}^{(t)}) < 1$*, and* $\tilde{e}^{(t)} \neq 0$*, then the* A*-norm of the error* $\|e^{(t)}\|_A$ *decreases at time instant* t*.*

Proof. We can write the squared A-norm of the error as

$$\begin{aligned} \|e^{(t+1)}\|_A^2 &= {e^{(t+1)}}^T A e^{(t+1)} \\ &= {e^{(t)}}^T {\hat{G}^{(t)}}^T A \hat{G}^{(t)} e^{(t)} \\ &= {e^{(t)}}^T (I - A\hat{D}^{(t)}) A (I - \hat{D}^{(t)} A) e^{(t)} \\ &= \|e^{(t)}\|_A^2 - {e^{(t)}}^T A (2\hat{D}^{(t)} - \hat{D}^{(t)} A \hat{D}^{(t)}) A e^{(t)}. \end{aligned} \tag{2.17}$$

Let m be the number of rows being relaxed at time instant t. Without loss of generality, we can consider the ordering from Equation 2.16, and write

$$\hat{D}^{(t)} = \begin{bmatrix} \mathbf{0} & O^T \\ O & I \end{bmatrix}, \tag{2.18}$$

where I is the $m \times m$ identity matrix, O is the $(n-m) \times m$ zero matrix, and $\mathbf{0}$ is the $(n-m) \times (n-m)$ zero matrix. Therefore,

$$2\hat{D}^{(t)} - \hat{D}^{(t)} A \hat{D}^{(t)} = \begin{bmatrix} \mathbf{0} & O^T \\ O & 2I - \tilde{A}^{(t)} \end{bmatrix}, \tag{2.19}$$

where $\tilde{A}^{(t)}$ is an $m \times m$ principal submatrix of A.

This means $\|e^{(t)}\|_A^2$ is reduced when $2I - \tilde{A}^{(t)}$ is SPD. The eigenvalues of $2I - \tilde{A}^{(t)}$ are $2 - \alpha$, where α is an eigenvalue of $\tilde{A}$. Since $\alpha > 0$, $2I - \tilde{A}^{(t)}$ is SPD when $2 > \alpha > 0$. The eigenvalues of $\tilde{G}^{(t)}$ are $\mu = 1 - \alpha$, so $1 > \mu > -1$, i.e., $|\mu| < 1$ or $\rho(\tilde{G}^{(t)}) < 1$. □

The proof of Theorem 2 can also be used to show that a single Gauss-Seidel relaxation

reduces $\|e^{(t)}\|_A^2$. This is because

$$2\hat{D}^{(t)} - \hat{D}^{(t)}A\hat{D}^{(t)} = \begin{bmatrix} \mathbf{0} & O^T \\ O & I \end{bmatrix}, \tag{2.20}$$

where I is the $m \times m$ identity. If we consider Gauss-Seidel with natural ordering, I is 1×1. When using red-black Gauss-Seidel on a 5-point or 7-point stencil, I is approximately or exactly of size $n/2 \times n/2$. In general, for a method in which an independent set of rows are relaxed in parallel, e.g., multicolor Gauss-Seidel, $\|e^{(t+1)}\|_A^2 < \|e^{(t)}\|_A^2$.

Returning to the discussion of simplified asynchronous Jacobi, it can happen that $\|e^{(t+1)}\|_A^2 < \|e^{(t)}\|_A^2$ since $\rho(\tilde{G}^{(t)}) \leq \rho(G)$ by the interlacing theorem. Matrix $\tilde{G}^{(t)}$ decreases in size when fewer rows are relaxed in parallel, which happens when the number of threads or processes is increased. Furthermore, $\tilde{G}^{(t)}$ can be block diagonal since removing rows can create blocks that are decoupled. The interlacing theorem can be further applied to these blocks, resulting in $\rho(\tilde{G}_i^{(t)}) \leq \rho(\tilde{G}^{(t)})$, where $\tilde{G}_i^{(t)}$ is block i of $\tilde{G}^{(t)}$ with the largest spectral radius. If many processes are used, it may happen that $\tilde{G}^{(t)}$ will have many blocks, resulting in $\rho(\tilde{G}_i^{(t)}) \ll \rho(\tilde{G}^{(t)})$. This can explain why increasing the concurrency can result in simplified asynchronous Jacobi converging faster than synchronous Jacobi, and converging when synchronous Jacobi does not. This is a result we will show experimentally.

An example of how $\rho(\tilde{G}^{(t)})$ changes as the fraction of rows that are being relaxed decreases is shown in Figure 2.2. The matrix used is the FE matrix (see Section 2.6.1). For each fraction of rows being relaxed, the max, min, and mean spectral radius of 200 different choices of $\tilde{G}^{(t)}$ is shown, where rows selected to be relaxed are chosen randomly. When the fraction of rows being relaxed is $\approx .7$ or less, the max of all $\rho(\tilde{G}^{(t)}) < 1$. Additionally, when the fraction of rows being relaxed is $\approx .9$ or less, the mean of all $\rho(\tilde{G}^{(t)}) < 1$. This suggests that we would likely see a consistent reduction in the error when the fraction of rows that are relaxed in parallel is $\approx .9$ or less.

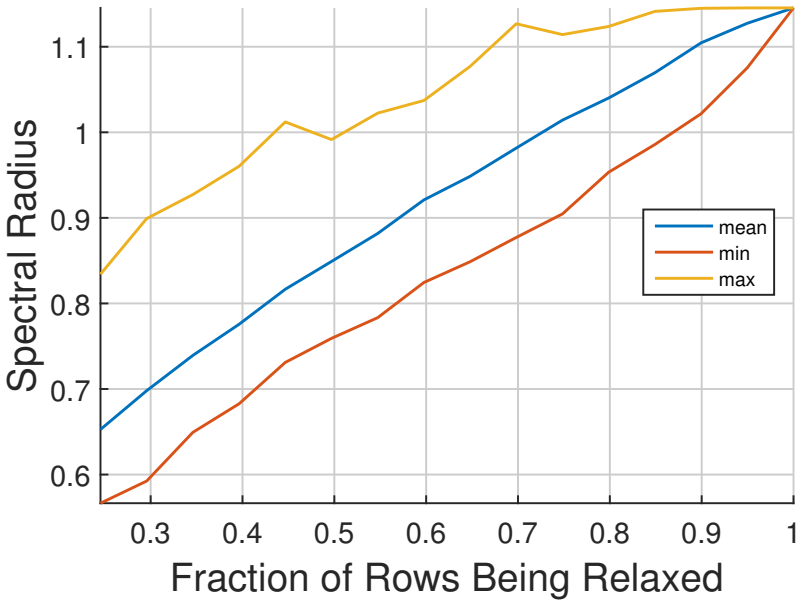


Figure 2.2: Spectral radius of $\tilde{G}^{(t)}$ versus the fraction of rows being relaxed. Max, min, and mean spectral radius is shown for 200 different choices of $\tilde{G}^{(t)}$, where the rows selected to be relaxed are chosen randomly. The test problem is the FE matrix.

2.4 Implementing Asynchronous Jacobi in Shared Memory

Our implementations use a sparse matrix-vector multiplication (SpMV) kernel to compute the residual, which is then used to correct the approximate solution. A single iteration of both synchronous and asynchronous Jacobi can be written in the following way:

1. For each row i, compute the residual $r_i^{(t)} = b_i - \sum_{j=1}^{n} A_{ij} x_j^{(t)}$ (this is the SpMV step).
2. For each row i, correct the approximation $x_i^{(t+1)} = x_i^{(t)} + r_i^{(t)}/A_{ii}$.
3. Check for convergence (detailed below and in Section 2.5 for distributed memory).

Since more than one row is assigned to each thread, each thread only computes $Ax^{(t)}$, $r^{(t)}$ and $x^{(t)}$ for the rows assigned to it. The contiguous set of rows assigned to a thread is defined as its *subdomain*, which is determined using a graph partitioner in general.

OpenMP was used for our shared memory implementation. The vectors $x^{(t)}$ and $r^{(t)}$ are stored in shared arrays. The only difference between the asynchronous and synchronous implementations is that the synchronous implementation uses a barrier after step 1 and

a reduction for step 3. Since each element in either $x^{(t)}$ or $r^{(t)}$ is updated by writing to memory (not by using a fetch-and-add operation), atomic operations can be avoided. Writing or reading a double precision word is atomic on modern Intel processors if the array containing the word is aligned to a 64-bit boundary.

For synchronous Jacobi, the iteration is stopped if the relative norm of the global residual falls below a specified tolerance (determined using a reduction), or if a specified number of iterations has been carried out. For asynchronous Jacobi, a shared array `is_done` is used to determine when the iteration should stop. `is_done` is of size equal to the number of threads, and is initialized to zero. If thread i satisfies the local stopping criterion, element $i - 1$ (zero based indexing) of `is_done` is set to one. Once a thread reads a one in all elements of `is_done`, that thread stops iterating. For the local stopping criterion, a thread has either carried out a specified number of iterations, or the residual norm for its subdomian has dropped below some tolerance.

While we do not implement any sophisticated termination schemes, research in terminating iterations has primarily been concerned with when to terminate without using global communication. Savari and Bertsekas [40, 34] have proposed several termination detection schemes, but in practice, these schemes require a variation of a broadcast operation, i.e., one process communicates with all others. The body of research is quite small for *decentralized* termination detection, i.e., termination detection where no global communication is used. Bahi [41, 42] proposed a method where a minimum spanning tree is constructed from the graph representing the connectivity of the parallel process topology, and each process sends a message up the tree to the root once it has converged. This scheme is implemented in JACK [37]. Dijkstra [43] proposed something similar, although, the paper was not meant to address asynchronous iterative methods but rather the more general case in which information moving through a graph comes to a stand still.

2.5 Implementing Asynchronous Jacobi in Distributed Memory

The program structure for our distributed implementation is the same as that of the shared memory implementation as described in the first paragraph of Section 2.4. However, there are no shared arrays. Instead, a process i stores a *ghost layer* of values received from its neighbors. A neighbor of i is determined by inspecting the non-zero pattern in the off-diagonal blocks that belong to i. Process j is a neighbor of i if an off-diagonal block belonging to row i contains a column index in the subdomain of j. The column indices in that block are the indices of the ghost layer values that j sends to i. Figure 2.3 shows an an example of a partitioned matrix when using four processes. The red diagonal blocks denote the connections among points in a subdomain. The off-diagonal blocks denote the connections of points in a subdomain to points in other subdomains. We note that multiple rows in a subdomain may require the same information from a different subdomain, i.e., off-diagonal column indices may be repeated across multiple rows in a subdomain. Therefore, a subdomain only requires data corresponding to the unique set of off-diagonal column indices.

We overlapped computation and communication in our SpMV. More specifically, we can write SpMV in the following steps, which are carried out in parallel on each process:

1. Send values of $\vec{x}_i^{(t)}$ to neighbors.
2. Compute $\vec{y}_i = \mathbf{A}_{ii}\vec{x}_i^{(t)}$.
3. Receive values of $\vec{x}_{q_{ij}}$ from neighbor q_{ij}, where $j = 1, \ldots, \mathcal{N}_i$.
4. Compute

$$\vec{x}_i^{(t+1)} = \vec{y}_i + \sum_{j=1}^{\mathcal{N}_i} \mathbf{A}_{iq_{ij}}\vec{x}_{q_{ij}}^{(t)}. \tag{2.21}$$

The matrix $\mathbf{A}_{ii}$ is the diagonal block that belongs to process i (red blocks in Figure 2.3), and $\vec{x}_i$ is the part of x that belongs to i. $\mathcal{N}_i$ is the number of neighbors of i, $\mathbf{A}_{iq_{ij}}$ is the off-diagonal block corresponding to neighbor q_{ij} of process i (blue blocks in Figure 2.3), and

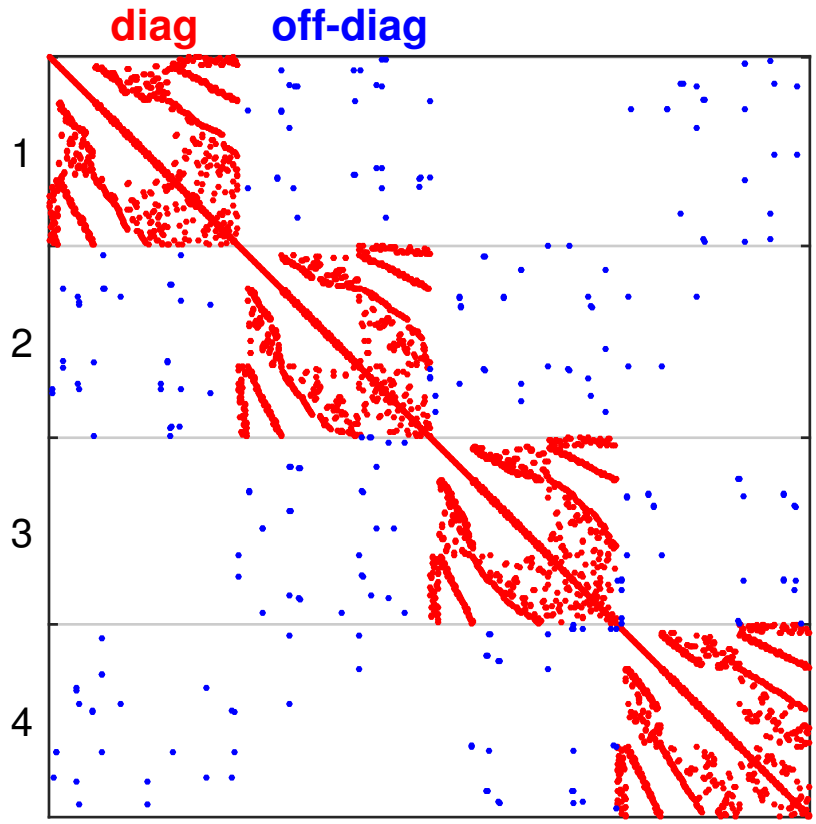


Figure 2.3: Sparsity pattern for an unstructured finite element matrix partitioned into four parts. The red points denote the non-zero values of the diagonal blocks of the matrix, and the blue points denote the non-zero values of the off-diagonal blocks.

$\vec{x}_{q_{ij}}^{(t)}$ is the vector of values in the subdomain of process q_{ij} (ghost layer values). Remember that process i only requires values of $\vec{x}_{q_{ij}}$ corresponding to the off-diagonal blocks, so the entirety of $\vec{x}_{q_{ij}}$ is not sent to process i.

A process terminates once it has carried out a specified number of iterations. For the synchronous case, all processes will terminate at the same iteration. For the asynchronous case, some processes can terminate even when other processes are still iterating. This naive scheme requires no communication. If it is desired that some global criterion is met, e.g., the global residual norm has dropped below some specified tolerance, a more sophisticated scheme must be employed. However, since we are only concerned with convergence rate rather than termination detection, we leave this topic for future research.

We used MPI for communication in our distributed implementations to communicate

ghost layer values. For our synchronous implementation, two-sided communication was used. Here, both the sending and receiving processes take part in the exchange of data. We implemented this using `MPI_Isend()`, which carries out a non-blocking send, and `MPI_Recv()`, which carries out a blocking receive. `MPI_Waitall()` is then used to complete all outstanding communication calls.

For our asynchronous implementation, remote memory access (RMA) communication was used [44], specifically, one-sided MPI with passive target completion (passive one-sided MPI). For RMA, each process must first allocate a region of memory that is accessible by remote processes. This is known as a memory window, and is allocated using the function `MPI_Win_allocate()`. For our implementation, we used a one dimensional array for the window, where each neighbor of a process writes to a subarray of the window. The size of each subarray is equal to the number of ghost layer values needed from that neighbor. The subarrays do not overlap so race conditions do not occur.

The *origin* process writes to the memory of the *target* process using `MPI_Put()`. For passive target completion, `MPI_Put()` is carried out without the involvement of the target. This is done by initializing an access epoch on a remote memory window using a lock operation. We used `MPI_Win_lock_all()`, which allows access to windows of all processes until `MPI_Win_unlock_all()` is called. Another option is to use `MPI_Win_lock()` and `MPI_Win_unlock()`, which locks and unlocks a specific target process. We found that `MPI_Put()` operations completed faster when using `lock_all()` functions instead of using `MPI_Win_lock()` and `MPI_Win_unlock()`. If using the `lock_all()` commands, completing messages must be done in the following way:

- At the origin, `MPI_Win_flush()` or `MPI_Win_flush_local()` must be called. The former blocks until the `MPI_Put()` is completed at the target, and the latter blocks until the send buffer can be reused. The functions `MPI_Win_flush_all()` and `MPI_Win_flush_local_all()` can also be used, which complete all outstanding messages. These `flush_all()` functions are faster, which we will show

later.

- In MPI implementations that use a "separate" memory model, `MPI_Win_sync()` must be used at the target in order for data from incoming messages to be transferred from the network card to the memory window.

It is important to note that `MPI_Put()` does not write an array of data from origin to target atomically, but is atomic for writing single elements of an array. We do not need to worry about writing entire messages atomically, which can be done using `MPI_Accumulate()` with `MPI_REPLACE` as the operation. This is because we are parallelizing the relaxation of rows, so blocks of rows do not need to be relaxed all at once, i.e., information needed for a row is independent of information needed by other rows.

To compare the different options for passive one-sided MPI, we created a benchmark. The goal of the benchmark is to measure how fast messages complete when using passive one-sided MPI. Another goal is to see if `MPI_Put()` will complete if we do not flush or use `MPI_Win_sync()`. Our benchmark simulates the communication pattern of a series of SpMV operations that require only nearest neighbor communication in a 2D mesh. Here, the 2D mesh is virtual, i.e., a process p is assigned an (x, y) coordinate on the virtual mesh using the MPI cartesian grid topology routines. Each process has two to four neighbors, one per cardinal direction.

Processes carry out a fixed number of iterations, and each process sends a single message to each neighbor in each iteration. In each iteration, starting with iteration zero, all processes execute the following steps in parallel:

1. Call `MPI_Put(sendbuff[i], sendcount,..., target_rank[i],...)`, where i ranges from 0 to the number of neighbors minus one. Here, `sendcount` is the size of the message being sent, and `sendbuff[i]` is the data being sent to neighbor i (array of double precision numbers). `sendbuff` is a 2D array, which is why we need to reference the i^{th} row. `sendbuff[i]` is initialized to zero at

iteration zero.

2. Poll the memory window until all information has been received. If a process does not receive a message after polling for s seconds, then the program exits, indicating a `MPI_Put()` did not complete. We set s to 60 for the experiments shown below.

3. Update the send buffers: `sendbuff[i][j]++`, where $j = 0, \ldots,$ (`sendcount` $- 1$).

From the steps above, in each iteration, we can see that each process expects to read the current iteration number at each element of its memory window. Step 2 means that if a process does not read the current iteration number after s seconds, the program terminates.

For our first experiment, we used nine Haswell nodes (3×3 mesh) of the Cori supercomputer at NERSC (see Section 2.6.1), with one MPI process per node. We found that some `MPI_Put()` operations did not complete when not using `MPI_Win_sync()` and `flush()` functions. Table 2.1 shows the total wall-clock time for 100 iterations by each process with a message size of 288 doubles. We chose 288 because this is the largest message size used in our distributed memory experiments with asynchronous Jacobi. The table shows results for using different passive one-sided MPI functions. The mean of 20 runs was taken for each entry in the table. With the exception of the *lock target* and *two-sided* entries in the table, `MPI_Win_lock_all()`, `MPI_Win_unlock_all()`, and `MPI_Put()` is used. When using any kind of `flush()` command, `MPI_Win_sync()` was also used. For lock target, `MPI_Win_lock()` was used before each `MPI_Put()` and `MPI_Win_unlock()` is used after. At the target, `MPI_Win_sync()` is used. The *none* entry denotes the absence of flushing and `MPI_Win_sync()`, which is what we used for our asynchronous Jacobi implementation. The time for two-sided MPI using `MPI_Isend()`, `MPI_Recv()` and `MPI_Wait_all()` is also shown. The results show that locking each target is over 10 times slower than *none*, which is the fastest, and two-sided is almost two times slower than *none*. Additionally, using either `flush_all()` function is faster than

flushing each target, even if there are at most four targets for the 2D mesh used in our benchmark. Therefore, if we wanted to write a code where all `MPI_Put()` operations are guaranteed to complete, we would use one of the `flush_all()` functions. However, we found that asynchronous Jacobi converged when not flushing and using `MPI_Win_sync()`. This is because although some information is overwritten before it is sent, information in subsequent iterations is still delivered.

Table 2.1: Benchmark wall-clock times in seconds for sending 100 messages of size 288 doubles. A 3×3 processor mesh is used.

	wall-clock time (seconds)
none	0.00077
flush	0.00134
flush local	0.00133
flush all	0.00084
flush local all	0.00086
lock target	0.00853
two-sided	0.00151

Figure 2.4 (a) shows how the overall wall-clock time of our benchmark is affected by different message sizes. In this figure, 32 nodes with 1,024 total MPI processes are used, and each process must send and successfully receive 100 messages. For each data point, the mean of 50 samples is taken. The legend entries refer to the same set of MPI functions as described for Table 2.1. We can see that for smaller messages, two-sided is the fastest. For larger messages, the flush target functions perform the worst due to the time that the unlock operation takes to complete, as shown in Figure 2.4 (b). Figure 2.4 (b) also shows that, when using `lock_all()` functions, the time for the lock operations dominates the overall time except for large message sizes. For the largest message size, the `lock_all()` functions take a similar amount of time as two-sided. The reason the `lock_all()` functions are more costly for the smaller message size than what is shown in Table 2.1 is because more MPI processes are used in Figure 2.4. The wall-clock time for `lock_all()` functions increases with increasing number of MPI processes. This suggests that some form of

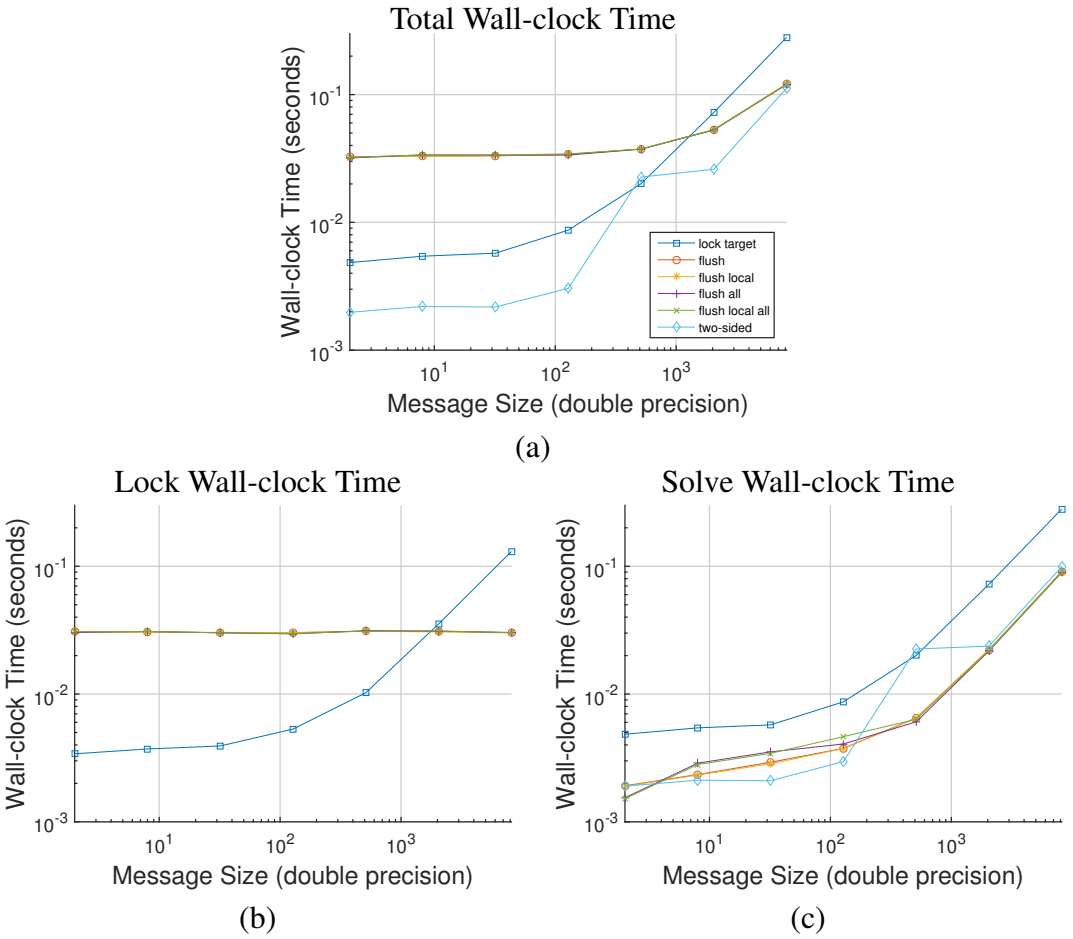


Figure 2.4: Wall-clock time as a function of message size for our benchmark. The benchmark measures how long it takes for communication operations to complete when using one-sided MPI with passive target completion. Figure (a) shows the overall wall-clock time, Figure (b) shows the wall-clock time for locking and unlocking windows, and Figure (c) shows the total wall-clock time again but without including the time it takes to execute the `lock_all()` functions. The purpose of (c) is to show only the time it takes to carry out some number of iterations, where we consider calls to `lock_all()` functions as part of the setup phase. 32 nodes with 1,024 total MPI processes are used, and each process must send and successfully receive 100 messages. For each data point, the mean of 50 samples is taken. Each curve denotes a different configuration of MPI functions used.

global communication may be happening when using the `lock_all()` functions. However, since we only need to call `MPI_Win_lock_all()` and `MPI_Win_unlock_all()` once, we consider this to be a cost that impacts only the setup phase of our code. Figure 2.4 (c) shows the timing for only the *solve* phase, where we do not include the time it takes to call `MPI_Win_lock_all()` and `MPI_Win_unlock_all()`. This figure shows that

`MPI_Put()` operations complete most quickly when using the `lock_all()` functions. As stated earlier, this is why we chose the `lock_all()` functions for our distributed memory asynchronous Jacobi method instead of locking the target. In our distributed memory asynchronous Jacobi method, we did not include the time it takes to execute `lock_all()` functions when recording the overall wall-clock time.

2.6 Results

2.6.1 Test Framework

All experiments were run on NERSC's Cori supercomputer. Shared memory experiments were run on an Intel Xeon Phi Knights Landing (KNL) processor with 68 cores and 272 threads (four hyperthreads per core), and distributed memory experiments were run on up to 128 nodes, each node consisting of two 16-core Intel Xeon E5-2698 "Haswell" processors. In all cases, we used all 32-cores of each Haswell node with one process per core. We used a random initial approximation $x^{(0)}$ and a random right-hand side b with elements in the range [-1,1], and the following test matrices:

1. Matrices arising from a five-point centered difference discretization of the Poisson equation with Dirichlet boundary conditions on a rectangular domain with a uniform grid. These matrices are irreducibly W.D.D., symmetric positive-definite, and $\rho(G) < 1$. We refer to these matrices as FD.

2. An unstructured finite element discretization of the Poisson equation with Dirichlet boundary conditions on a square domain. The matrix has 3,081 rows and 20,971 non-zero values. The matrix is not W.D.D. The matrix is symmetric positive-definite and $\rho(|G|) > \rho(G) > 1$. We refer to this matrix as FE.

3. Matrices listed in Table 2.2 from the SuiteSparse matrix collection [45].

Matrices are partitioned using METIS [46], including for FD, and are stored in compressed sparse row (CSR) format.

Table 2.2: Test problems from the SuiteSparse Matrix Collection. All matrices are symmetric positive definite.

Matrix	Non-zeros	Equations
thermal2	8,579,355	1,227,087
parabolic_fem	3,674,625	525,825
thermomech_dM	1,423,116	204,316
Dubcova2	1,030,225	65,025

2.6.2 Simplified Asynchronous Jacobi Compared to OpenMP Asynchronous Jacobi

The primary goal of this section is to validate the simplified asynchronous Jacobi model presented in Section 2.3 by comparing its behavior to *OpenMP asynchronous Jacobi*. OpenMP asynchronous Jacobi is our implementation of asynchronous Jacobi in shared memory using OpenMP. The model is simulated using a sequential implementation.

For our first experiment, we look at how simplified asynchronous Jacobi and OpenMP asynchronous Jacobi compare to the synchronous case. We consider the scenario where all threads run at the same speed, except one thread that runs at a slower speed. This could simulate a system in which one core is slower than others. We assign a computational delay, δ, to thread p_i corresponding to row i near the middle of a test matrix. For OpenMP asynchronous Jacobi, the delay corresponds to having p_i sleep for a certain number of microseconds. Since synchronous Jacobi must use a barrier when using OpenMP, all threads have to wait for p_i to finish sleeping and relaxing its rows before they can continue. For simplified asynchronous Jacobi, row i is delayed by δ iterations. This means row i only relaxes at multiples of δ iterations, while all other rows relax at every iteration. In the case of synchronous Jacobi, all rows relax at iteration numbers that are multiples of δ to simulate waiting for the slowest process.

We first look at how much faster simplified and OpenMP asynchronous Jacobi can be compared to synchronous Jacobi when we vary the delay in computation δ. The test matrix is an FD matrix with 68 rows (17×4 mesh) and 298 non-zero values, and we use 68 threads (available on the KNL platform), giving one row per thread. A relative residual

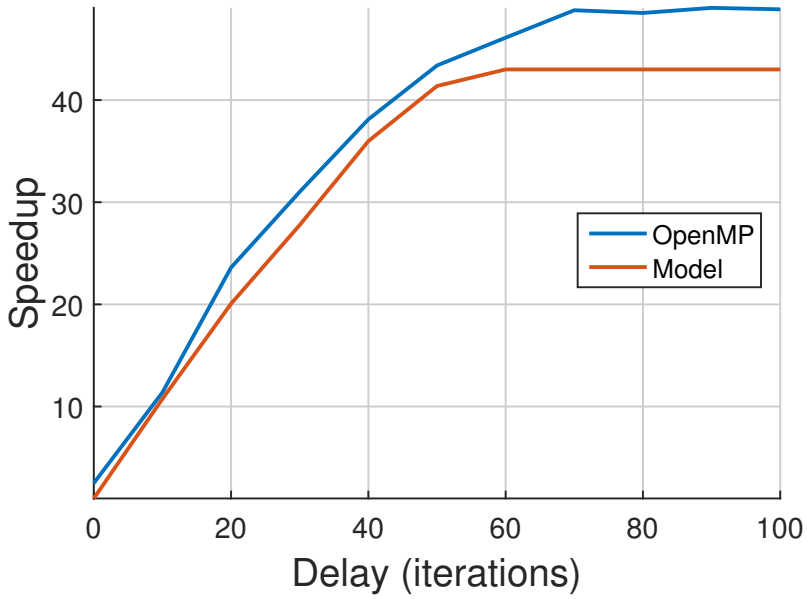


Figure 2.5: Speedup of simplified and OpenMP asynchronous Jacobi over synchronous Jacobi for 68 threads (on the KNL platform) as a function of the artificial delay in computation δ experienced by one thread. For OpenMP, the delay is varied from zero to 3000 microseconds. For the simplified asynchronous Jacobi, δ is varied from zero to 100, which is shown on the x-axis. A relative residual norm tolerance of $.001$ is used. The test problem is an FD matrix with 68 rows and 298 non-zeros. The mean of 100 samples is taken for each data point. We can see that a speedup of over 40 is achieved for larger delays.

1-norm tolerance of $.001$ is used. For OpenMP, we varied the delay from zero to 3000 microseconds, and recorded the mean wall-clock time for 100 samples for each delay. For the simplified asynchronous Jacobi, we varied δ from zero to 100. Figure 2.5 shows the speedup for simplified and OpenMP asynchronous Jacobi as a function of the delay parameter. The speedup for OpenMP is defined as the total wall-clock time for synchronous Jacobi divided by the total wall-clock time for asynchronous Jacobi. Similarly, for simplified asynchronous Jacobi, the speedup is defined as the total number of iterations for synchronous Jacobi divided by the total number of iterations for simplified asynchronous Jacobi.

Figure 2.5 shows a qualitative and quantitative agreement between simplified and OpenMP asynchronous Jacobi. As the delay is increased, both achieve a speedup above 40 before reaching a plateau. In general, this maximum speedup depends on the problem,

the number of threads, and which threads are delayed. In the case of the FD problem for Figure 2.5, for simplified asynchronous Jacobi, the speedup is based on how fast the components of the residual corresponding to non-delayed rows tend to zero. If one row is never relaxed, the residual propagation matrix $\hat{H}^{(t)}$ is fixed and thus, from Equation 2.14,

$$r^{(t+1)} = \hat{H}^{(t)} r^{(t)} = \begin{bmatrix} 1 & h_1^{(t)} \\ o & \tilde{H}^{(t)} \end{bmatrix} \begin{bmatrix} r_1^{(t)} \\ \tilde{r}^{(t)} \end{bmatrix}. \tag{2.22}$$

In this equation, $\hat{H}^{(t)}$ has just one eigenvalue equal to one corresponding to the eigenvector ξ_1 (first unit basis vector). The remaining eigenvalues are the eigenvalues of $\tilde{H}^{(t)}$. From the proof of Theorem 1, $r^{(t)}$ converges to $\begin{bmatrix} \gamma & o^T \end{bmatrix}^T$ when applying $\hat{H}^{(t)}$ to $r^{(t)}$ many times, where γ is some scalar. The remaining $n-1$ eigenvectors of $\hat{H}^{(t)}$ corresponding to the eigenvalues of $\tilde{H}^{(t)}$ are exactly $\begin{bmatrix} 0 & \mathbf{v}_j^T \end{bmatrix}$, where $\mathbf{v}_j$ is an eigenvector of $\tilde{H}^{(t)}$ with $j = 1, \ldots, n-1$. Therefore, for this FD matrix, with a starting residual of $\tilde{r}^{(t)}$, the speedup of simplified asynchronous Jacobi will always increase until the iteration governed by the residual iteration matrix $\tilde{H}^{(t)}$ converges, at which point the speedup will stay constant. Since any FD matrix is W.D.D., the reason the speedup will always increase is due to Theorem 1. Another way of thinking about this is that rows 2 to n eventually need the information of the first row in order for the iteration to not stall, but the iteration can still progress for many iterations using old information from the first row.

Note that without artificially slowing down a thread, OpenMP asynchronous Jacobi is still slightly faster than synchronous Jacobi, as shown by values corresponding to a delay of zero. This is due to the fact that natural delays in computation occur that make some threads faster than others. For example, some rows have fewer non-zeros (load imbalance), which means some threads finish relaxing their rows more quickly. Another example is operating system jitter, where some cores are also responsible for background events related to the operating system.

Figure 2.6 (a) and (b) show the relative residual 1-norm as a function of number of

iterations for simplified and OpenMP asynchronous Jacobi, respectively. The test matrix is again an FD matrix with 68 rows and 298 non-zero values, and we use 68 threads (available on the KNL platform), giving one row per thread. For each "Async" curve in (b) ("Async" in the legend refers to OpenMP asynchronous Jacobi), we recorded the mean wall-clock time of 100 runs for each number of iterations $1, 2, \ldots, 100$. To create a residual 1-norm history (residual norm versus wall-clock time), at each number of iterations, after the iteration stops, the global residual norm is calculated and the total wall-clock time is recorded. Since this is done 100 times, we take the mean of 100 relative residual norm values and wall-clock times. To be clear, when computing the residual norm and wall-clock time for some number iterations, e.g., 50 iterations, we restart from iteration zero instead of using the approximate solution from iteration 49.

Figure 2.6 shows that simplified asynchronous Jacobi approximates the behavior of OpenMP asynchronous Jacobi quite well. A major similarity is the convergence curves for the two largest delays. For both the simplified and OpenMP asynchronous Jacobi, we can see that even when a single row is delayed until convergence (this corresponds to the largest delay shown, which is 100 for the model, and 10000 microseconds for OpenMP), the residual norm can still be reduced by simplified and OpenMP asynchronous Jacobi. As explained in the analysis of the results shown in Figure 2.5, the stall in convergence for a delay of 100 is due to the convergence of the iteration corresponding to using $\tilde{H}^{(t)}$ as the residual iteration matrix. However, it takes ≈ 50 iterations to reach a stall, which is large compared to the size of the matrix. For other delays, we see a "saw tooth"-like pattern corresponding to the delayed row being relaxed. The existence of this pattern for both simplified and OpenMP asynchronous Jacobi further confirms the suitability of the model. Additionally, we see that with no delay, OpenMP asynchronous Jacobi converges faster than synchronous Jacobi.

Figure 2.7 shows how OpenMP asynchronous Jacobi scales when increasing the number of threads from one to 272, and without adding any artificial delays in computation. For

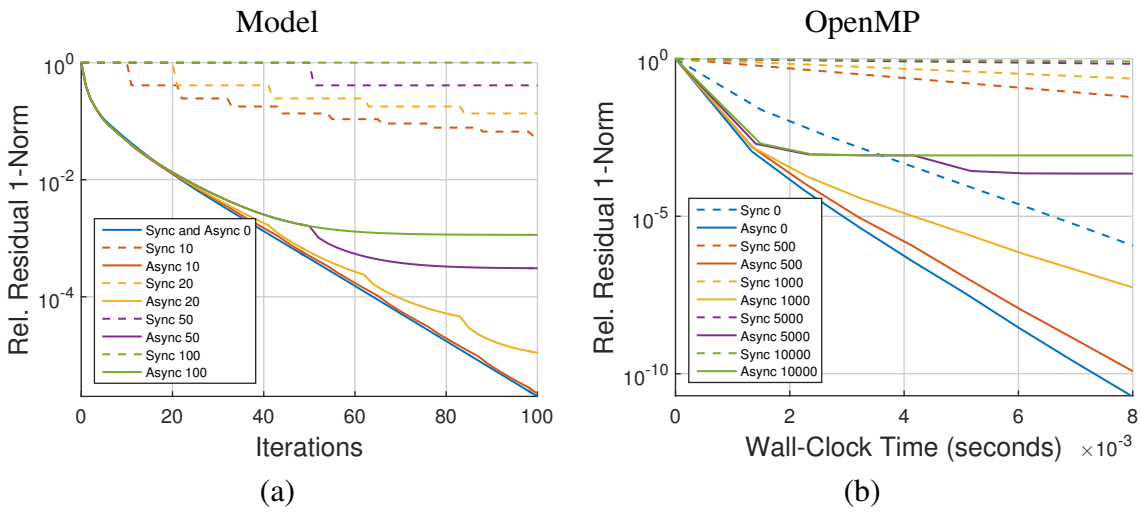


Figure 2.6: Relative residual 1-norm as a function of number of iterations for simplified asynchronous Jacobi, shown in (a), and relative residual norm as a function of OpenMP asynchronous Jacobi, shown in (b). Artificial delays in computation are added for both simplified and OpenMP asynchronous Jacobi, where "Async 10" denotes asynchronous with a delay of 10. The convergence for synchronous Jacobi with artificial delays is also shown. 68 threads on the KNL platform are used for OpenMP asynchronous Jacobi. The test problem is an FD matrix with 68 rows and 298 non-zeros.

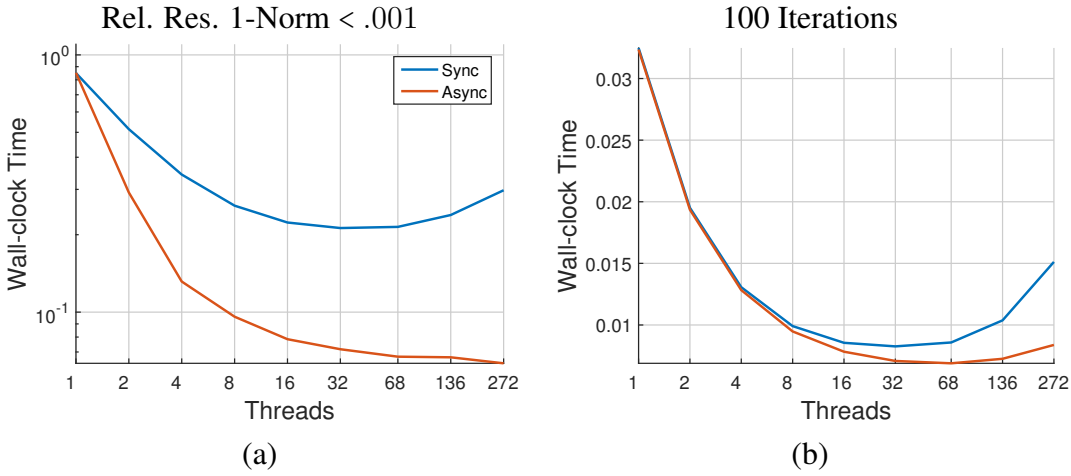
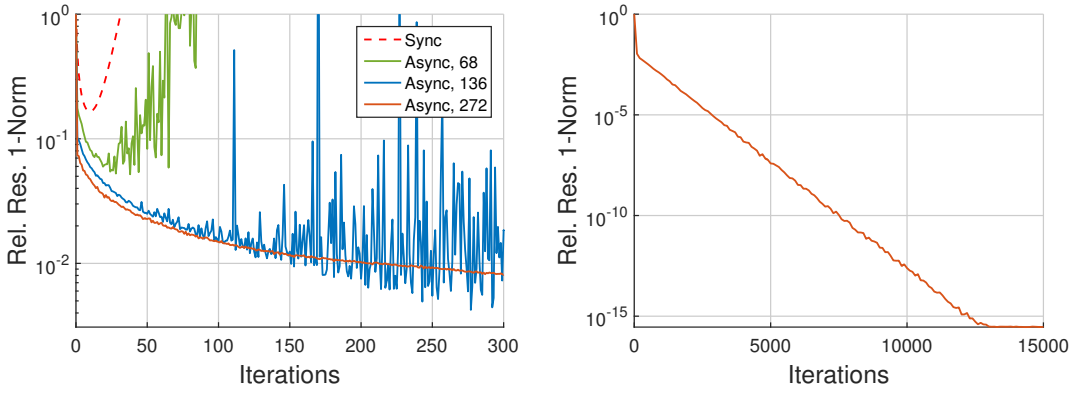


Figure 2.7: OpenMP asynchronous Jacobi compared with synchronous Jacobi as the number of threads increases. Figure (a) shows the wall-clock time when both methods reduce the relative residual 1-norm below .001. Figure (b) shows how much time is taken to carry out 100 iterations. The test problem is an FD matrix with 4,624 rows (17 rows per thread in the case of 272 threads) and 22,848 non-zero values. These results show that OpenMP asynchronous Jacobi is faster than synchronous Jacobi, especially when a specific reduction in the residual norm is desired.

these results, we used an FD matrix with 4,624 rows (17×16 mesh) and 22,848 non-zero values. When the number of threads does not divide 4,624 evenly, METIS is used. As in the previous set of results, we averaged the wall-clock time of 100 samples for each data point. Figure 2.7 (a) shows the wall-clock time for achieving a relative residual norm below .001. Figure 2.7 (b) shows the wall-clock time for carrying out 100 iterations regardless of what relative residual norm is achieved. Synchronous Jacobi is also shown.

Figure 2.7 (b) shows that, for OpenMP asynchronous Jacobi, using 136 threads is faster than using 272 threads when doing a fixed number of iterations. However, OpenMP asynchronous Jacobi is faster than synchronous Jacobi for 272 threads, even though OpenMP asynchronous Jacobi does more work since a thread only terminates once all threads have completed 100 iterations (see Section 2.6.1). This indicates that synchronization points have a higher cost than the extra computation done by OpenMP aynchronous Jacobi.

When comparing (a) and (b), we see another important result for OpenMP asynchronous Jacobi: the convergence rate increases as the concurrency increases. In particular, when reducing the residual norm to .001, using 272 threads for OpenMP asynchronous Jacobi gives the lowest wall-clock time compared to using a smaller number of threads (it takes 874.56 iterations on 272 threads and 937.79 iterations on 136 threads for OpenMP asynchronous Jacobi, and 2635 iterations for synchronous Jacobi). This can be explained by the fact that multiplicative relaxation methods often converge faster than additive methods, and increasing the number of threads results in OpenMP asynchronous Jacobi behaving more like a multiplicative relaxation scheme. When increasing the number of threads, the likelihood of coupled rows being relaxed in parallel is lower since the subdomains are smaller. This is because coupled rows within a subdomain will always be relaxed in parallel, but coupled rows that do not belong to the same subdomain may not be relaxed in parallel since threads are updating at different times. When the subdomains are smaller, a higher fraction of coupled rows do not belong to the same subdomain, so a higher fraction of the relaxations may be carried out in a multiplicative fashion.



(a) (b)

Figure 2.8: Relative residual 1-norm as a function of iterations for different numbers of threads (68, 136, and 272) on the KNL platform. Figure (a) shows that for a sufficient number of threads, asynchronous Jacobi can converge when synchronous Jacobi does not. Figure (b) shows that asynchronous Jacobi truly converges when using 272 threads. The test problem is the FE matrix.

We now look at a case in which OpenMP asynchronous Jacobi converges when synchronous Jacobi does not. Our test problem is the FE matrix. Figure 2.8 shows the residual norm as a function of the number of iterations. For OpenMP asynchronous Jacobi, the process of producing the residual norm history is the same as that of Figure 2.6 (b), but we only do one run per number of iterations (we are not taking the average of multiple runs), and we show the number of iterations instead of wall-clock time on the x-axis. Furthermore, the number of iterations shown on the x-axis is the average number of the local iterations carried out by all the threads (see Section 2.4 for details on how threads decide to stop iterating). In Figure 2.8 (a), we can see that as we increase the number of threads to 272, OpenMP asynchronous Jacobi starts to converge. This shows that the convergence rate of OpenMP asynchronous Jacobi can be dramatically improved by increasing the amount of concurrency, even to the point where OpenMP asynchronous Jacobi will converge when synchronous Jacobi does not. Figure 2.8 (b) shows that OpenMP asynchronous Jacobi truly converges, and does not diverge at some later time.

We can also show this result for simplified asynchronous Jacobi. Figures 2.9 and 2.10 show the convergence for simplified asynchronous Jacobi using the FE matrix. Figure 2.9

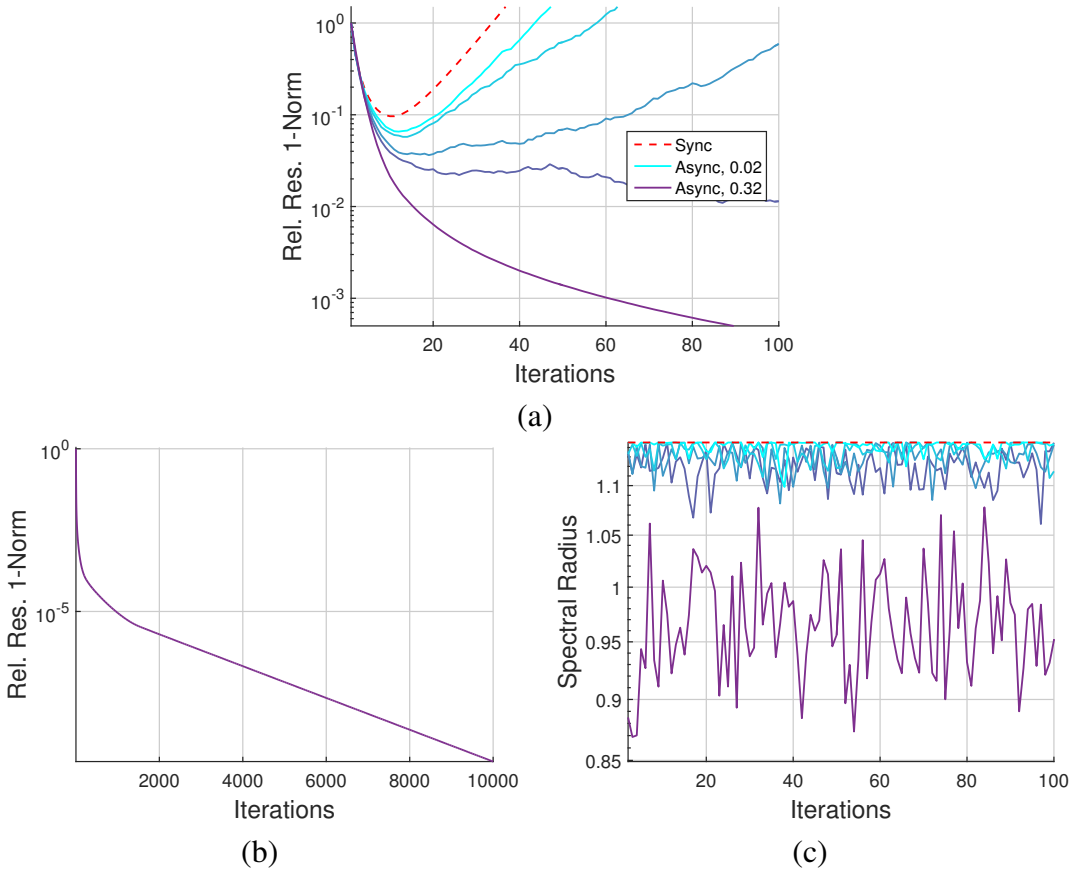


Figure 2.9: In this experiment, a random number of random rows is selected to not be relaxed (delayed) at each iteration for simplified asynchronous Jacobi. The fraction of delayed rows is varied from .02 to .32, where the cyan to purple gradient of the lines represents an increasing fraction of delayed rows. Figure (a) shows the relative residual norm as a function of number of iterations. Figure (b) shows the relative residual norm as a function of number of iterations only for a fraction of delayed rows of .32. Figure (c) shows $\rho(\tilde{G}^{(t)})$ as a function of the number of iterations. The test problem is the FE matrix. These results show that with a large enough fraction of delayed rows, e.g., .32, simplified asynchronous Jacobi will converge when synchronous Jacobi does not.

shows results for an experiment in which a random set of random rows are selected to be relaxed at each iteration. In Figure 2.9 (a), the relative residual 1-norm as a function of the number of iterations is shown. The fraction of rows selected to not be relaxed (delayed) is varied from .02 to .32. We can see that with a high enough fraction of delayed rows, simplified asynchronous Jacobi converges, as observed in Figure 2.8 for OpenMP asynchronous Jacobi. Just as in Figure 2.8 (b), Figure 2.9 (b) shows that the simplified asynchronous Jacobi truly converges. As discussed in Section 2.3.4, this convergence can be explained

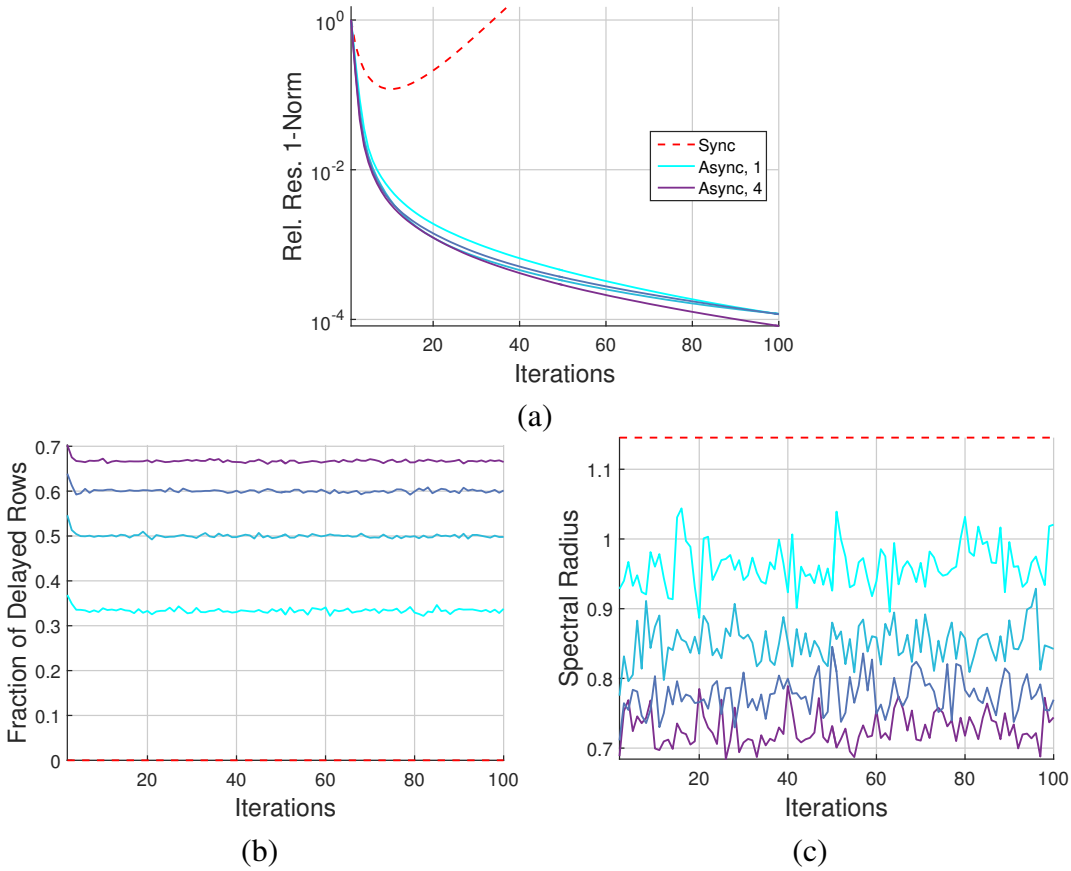


Figure 2.10: In this experiment, each row is assigned a delay in computation δ_i in the range $[0, 1, \ldots, \delta_{\max}]$ (the row is only relaxed when δ_i divides the iteration numbers evenly) for simplified asynchronous Jacobi. The maximum delay $\delta_{\max}$ is varied from one to four, where the cyan to purple gradient of the lines represents an increasing $\delta_{\max}$. Figures (a), (b) and (c) show the relative residual 1-norm, the fraction of delayed rows, and $\rho(\tilde{G}^{(t)})$, respectively. The x-axis for all three figures is the number of iterations. The test problem is the FE matrix. These results show that if each row is delayed by an average of just one iteration, simplified asynchronous Jacobi will converge when synchronous Jacobi does not.

by examining $\rho(\tilde{G}^{(t)})$. Figure 2.9 (c) shows $\rho(\tilde{G}^{(t)})$ as the number of iterations increases. For a fraction of delayed rows of .32, $\rho(\tilde{G}^{(t)})$ is often less than one.

Figure 2.10 is a slightly different experiment with simplified asynchronous Jacobi. Again, the FE matrix is used. In this experiment, instead of selecting a specific number of rows to be delayed, each row i is assigned a random delay δ_i in the range $[0, 1, \ldots, \delta_{\max}]$ sampled from a uniform random distribution, and δ_i changes after row i is relaxed. In other words, row i is relaxed after waiting δ_i iterations from the last iteration in which it was re-

laxed, and then δ_i is reset by again sampling a random integer from a uniform distribution in the range $[0, 1, \ldots, \delta_{\max}]$. For this experiment, we vary the the maximum delay $\delta_{\max}$ from one to four. Figure 2.9 (a) shows the relative residual 1-norm as a function of iterations as the delay is varied. The figure shows that with just a delay of one, simplified asynchronous Jacobi converges. This can be explained by looking at the fraction of delayed rows at each iteration (Figure 2.9 (b)), and the resulting $\rho(\tilde{G}^{(t)})$ corresponding to that fraction (Figure 2.9 (c)). With a delay of one, the fraction of delayed rows is between .3 and .4, and $\rho(\tilde{G}^{(t)})$ is often less than one. For larger delays, $\rho(\tilde{G}^{(t)})$ is always less than one.

2.6.3 Asynchronous Jacobi in Distributed Memory

In this section, we show some similar results to that of the previous section, but for a distributed memory implementation. We ask if asynchronous Jacobi can be faster than synchronous Jacobi, and can it converge when synchronous Jacobi does not when a distributed memory implementation is used. We define *POS asynchronous Jacobi* as asynchronous Jacobi implemented using one-sided MPI with passive target completion (here, POS stands for "passive one-sided"). We look at how POS asynchronous Jacobi compares with synchronous Jacobi for the problems in Table 2.2.

For each matrix and number of MPI processes, we recorded the mean wall-clock time of 200 runs for each number of iterations $1, 2, \ldots, 100$. To create a residual 1-norm history (residual norm versus number of iterations), at each number of iterations, after the iteration stops, the global residual norm is calculated. Since this is done 200 times, we take the mean of 200 relative residual norm values. To be clear, when computing the residual norm for some number iterations, e.g., 50 iterations, we restart from iteration zero instead of using the approximate solution from iteration 49. We used linear interpolation on the $\log_{10}$ of the residual norm history in order to extract the wall-clock time for a specific residual norm value.

The first row of figures in Figure 2.11 show the relative residual 1-norm as a function

of number of relaxations for three problems (Dubcova2 is not included). The plots are organized such that the problem size increases from left to right. Since the amount of concurrency affects the convergence of POS asynchronous Jacobi, several curves are shown for different numbers of nodes ranging from one to 128 nodes (32 to 4,096 MPI processes). This is expressed in a green-to-blue color gradient, where green is one node and blue is 128 nodes. We can see that in general, POS asynchronous Jacobi tends to converge in fewer relaxations. More importantly, as the number of nodes increases, the convergence of POS asynchronous Jacobi is improved.

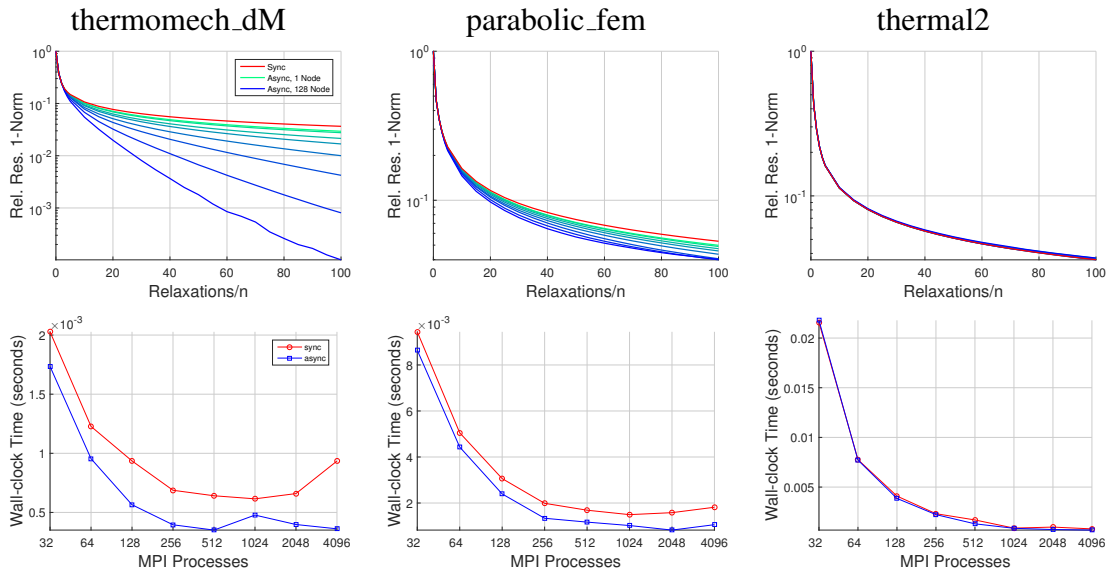


Figure 2.11: The first row shows the relative residual 1-norm as a function of relaxations/n for synchronous Jacobi and POS asynchronous Jacobi. For POS asynchronous Jacobi, one to 128 nodes are shown (32 to 4,096 MPI processes), where the green to blue color gradient of the lines represents an increasing number of nodes. The second row shows wall-clock time in seconds as a function of number of MPI processes for reducing the relative residual norm to 0.1. Results for three different problem sizes are given, where the size increases from left to right. These results show that POS asynchronous Jacobi is generally faster than synchronous Jacobi when the number of rows per process is relatively small.

The second row of figures in Figure 2.11 shows the wall-clock time in seconds for reducing the residual norm by a factor of 10 as the number of MPI processes increases. For POS asynchronous Jacobi, in the case of thermomech_dM, we can see that at 512 MPI processes, the time starts to increase, which is likely due to communication time outweigh-

ing computation time. However, since increasing the number of MPI processes improves convergence, wall-clock times for 2,048 and 4,096 MPI processes are lower than for 1,024 processes. This result is similar to that of Figure 2.7. In particular, the communication cost eventually outweighs the computation cost as the number of processes increases, resulting in the wall-clock time increasing if we fix the number of iterations. However, if we wish to reduce the residual norm by a fixed amount, the increase in convergence rate results in a lower total wall-clock time. We suspect that we would see the same effect in the cases of parabolic_fem if more processes were used. In general, we can see that POS asynchronous Jacobi is faster than synchronous Jacobi.

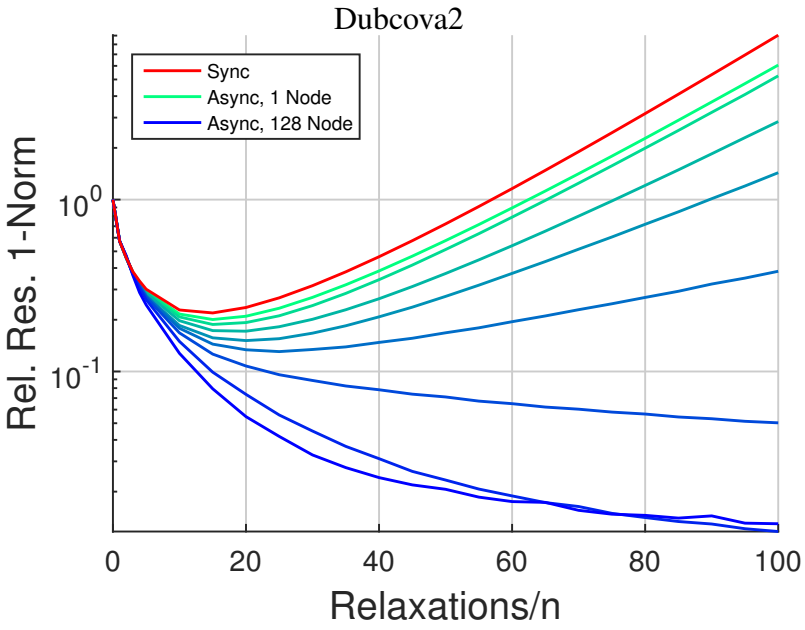


Figure 2.12: Relative residual 1-norm as a function of relaxations/n for synchronous Jacobi using two-sided communication and for POS asynchronous Jacobi. The Dubcova2 matrix is used as the test problem. For POS asynchronous Jacobi, results for one to 128 nodes are shown (32 to 4,096 MPI processes), where the green to blue color gradient of the lines represents an increasing number of nodes. As in Figure 2.8, increasing the number of processes improves the convergence rate of asynchronous Jacobi.

Improving the convergence with added concurrency is more dramatic in Figure 2.12, where the relative residual 1-norm as a function of number of relaxations is shown for Dubcova2. This behavior is similar to the behavior shown in Figure 2.8, where increasing the number of threads resulted OpenMP asynchronous Jacobi converging when synchronous

Jacobi did not.

Lastly, while we do not show results from weak scaling experiments, we would like to comment on the weak scaling case. Since the problem size increases as the number of processes increases, the convergence rate of Jacobi degrades. Therefore, the wall-clock time will increase as the number of processes increases. In the case of asynchronous Jacobi, since we have seen that increasing concurrency results in a higher convergence rate, the degradation in convergence rate will be smaller.

2.7 Conclusion

The transient convergence behavior of asynchronous iterative methods has not been well-understood. In this chapter, we study the transient behavior by analyzing the *simplified asynchronous Jacobi* method, where *simplified* refers to assuming no communication delays. For simplified asynchronous Jacobi, we are able to write an asynchronous iteration using *propagation matrices*, which are similar in concept to iteration matrices. By analyzing these propagation matrices, we showed that when the system matrix is weakly diagonally dominant, simplified asynchronous Jacobi can continue to reduce the residual even when some processes are slower than others (delayed in their computation). We also showed this result for asynchronous Jacobi implemented in OpenMP.

When the system matrix is symmetric positive definite, we showed the following properties: (a) simplified asynchronous Jacobi can converge when synchronous Jacobi does not, and (b) simplified asynchronous Jacobi will always converge if synchronous Jacobi converges. We observed property (a) in our shared and distributed memory experiments as well. This contrasts with the classical convergence theory for asynchronous iterative methods, which gives an overly negative picture. The classical theory predicts that in the worst case, asynchronous iterative methods diverge even if their synchronous counterparts converge. We note that although our explanations in this chapter used a simplified model for asynchronous iterations assuming no communication delays, we have also observed prop-

erty (a) when experimenting with nonsymmetric matrices and the general model (Equation 2.5) of asynchronous iterative methods.

CHAPTER 3
SOUTHWELL METHODS

For distributed computing, one of the most commonly used multigrid smoothers is Block Jacobi (as well as standard Jacobi), where the blocks come from an appropriate partitioning of the problem. This method is highly parallel, but has two main disadvantages: it does not converge for all symmetric positive definite matrices, and convergence degrades when parallelism is increased by using more blocks and thus smaller blocks. Additionally, all rows must be updated at every iteration, which can result in high communication costs for certain problems. On the other hand, Gauss-Seidel converges more rapidly than Block Jacobi and converges for all symmetric positive definite matrices. The disadvantage of Gauss-Seidel is that it is inherently a sequential method. Gauss-Seidel can be parallelized by using block multicoloring [47, 48], but a large number of colors may be needed for irregular problems.

In this chapter, our starting point is a related but little-known algorithm called the Southwell method [49, 50]. While Gauss-Seidel can be interpreted as relaxing a set of equations in a specific order, Southwell can be interpreted as relaxing equations one at a time in a dynamic and greedy fashion depending on which equation has the largest residual. In this way, Southwell can converge faster than Gauss-Seidel. However, Southwell is sequential by definition, since the choice made for which equation to relax depends on the previous relaxation. We call this method the Sequential Southwell method.

We first introduce the Parallel Southwell method, which can naturally be executed asynchronously. At each parallel step, Parallel Southwell simultaneously relaxes an "indepedent set" of equations, corresponding to residuals that are larger in magnitude than those of its neighboring equations (a neighboring equation will be defined precisely in this chapter). It has similarities to multicolor Gauss-Seidel, where each parallel step relaxes a set of equa-

tions of the same "color" which also constitute an independent set. The key advantage of Parallel Southwell appears to be utilizing residual information, focusing on computation where needed, on some equations and not others (equations can be relaxed multiple times before others are relaxed even once). The independent sets used by Parallel Southwell are not maximal, nor are they desired to be maximal. Thus the total work performed by each compute thread may be reduced, as some of the equations assigned to that thread may not need to be relaxed in a given parallel step. Associated with each relaxation is the update of residual information, which implies data movement. Therefore, Southwell not only reduces computation, but communication to shared memory regions, and concomitant energy costs [51].

In the distributed memory setting, Parallel Southwell can deadlock unless additional messages are sent to communication neighbors. We present the Distributed Southwell method in this chapter, which addresses this deadlock issue. In Distributed Southwell, each process stores the estimates to its own residual norm that are held by its neighbors. These estimates are used to avoid deadlock. Additionally, each process locally computes better estimates of residual norms that belong to its neighbors without any communication. Importantly, Distributed Southwell also uses new techniques to reduce communication compared to Parallel Southwell in distributed memory. However, Distributed Southwell still requires communication in order to avoid deadlock.

The last method introduced in this chapter is the Stochastic Parallel Southwell method, where the residual estimates stored by some process i are used to determine the probability that i relaxes its rows. This scheme requires no additional communication in order to avoid deadlock since even with a small probability, a process will eventually relax its rows and therefore communicate with other processes.

3.1 Background

3.1.1 The Sequential Southwell Method

We first introduce some notation necessary for explaining the Sequential Southwell method and its parallel variants. For row i, row indices $\eta_j \neq i$ are *neighbors* of i if $a_{\eta_j i} \neq 0$. We define the *neighborhood* of row i as the set $N_i = \{\eta_1, \eta_2, \ldots, \eta_{q_i}\}$ of cardinality q_i, where each index in the set is a neighbor, i.e., the neighborhood of row i is the set of rows coupled with row i. We also define $\Gamma_i = \{|r_{\eta_1}|, |r_{\eta_2}|, \ldots, |r_{\eta_{q_i}}|\}$ where r_{η_j} is the residual of equation η_j.

Instead of relaxing rows in some prescribed order as in the Gauss-Seidel method, each step of Sequential Southwell relaxes the row i with the largest component of the residual vector. Then the residual vector is updated, but notice that only components corresponding to neighbors of row i need to be updated. Formally,

$$x_i^{(t+1)} = \begin{cases} x_i^{(t)} + \dfrac{r_i^{(t)}}{a_{ii}}, & \text{if } |r_i| \text{ is the maximum for all } i, \\ x_i^{(t)}, & \text{otherwise,} \end{cases} \tag{3.1}$$

$$r_{\eta_j}^{(t+1)} = r_{\eta_j}^{(t)} - r_i^{(t)} \frac{a_{\eta_j i}}{a_{ii}}, \text{ for all } \eta_j \in N_i,\ j = 1, \ldots, q_i. \tag{3.2}$$

We also have that the updated residual $r_i^{(t+1)} = 0$ for the row i that was chosen to be relaxed. Note that we are technically using the Gauss-Southwell method, which is more natural to analyze, and which was actually first proposed by Gauss. In this method, we relax the row i with the largest $|r_i/a_{ii}|$. The method is identical to what we are calling the Sequential Southwell method when we scale the systems, as we do in this chapter, such that the matrices have unit diagonals.

Sequential Southwell can converge faster than Gauss-Seidel in terms of the number of relaxations. However, the method never caught on for automatic computers due to the relatively high cost of determining the row with the largest residual, compared to simply

cycling through all equations as in the Gauss-Seidel method. Nevertheless, it has recently found application as an *adaptive* multigrid smoother [52, 53], as a *greedy* multiplicative Schwarz method (where the subdomain with the largest residual norm is chosen to be solved next) [25], as a way of accelerating coordinate descent optimization methods for big data problems [54], and as a scheme for choosing basis vectors when finding sparse solutions to underdetermined inverse problems, e.g., [55, 56].

Our motivation to study and develop Southwell-like methods is due to today's high cost of interprocessor communication compared to computation. Assume for the moment that each parallel process is responsible for a single row of the matrix equation $Ax = b$. When a row is relaxed, that process must send data to the processes corresponding to neighboring rows in order for these processes to update their residuals (see formula (3.2)). Therefore, each relaxation is associated with communication. If Southwell-like methods reduce the number of relaxations required to solve a problem compared to that of stationary iterative methods, then they also can reduce the amount of communication.

3.2 Related Work

Several variants of Southwell's original method have been reported in the literature that are designed to reduce the cost of choosing the next row to relax and/or allow more than one equation to be relaxed at the same time. In general, these methods require parameters for calculating thresholds, but it is not clear how to tune such parameters in the case of sparse linear systems.

In the *sequential adaptive relaxation* method [52, 53], a small active set of rows is initially chosen. A row from this active set is chosen based on its residual and a preliminary relaxation is performed. If the updated value is not a significant change from the previous value, then the update is discarded and the row is removed from the active set. Otherwise, the updated value is kept, and the neighbors of the row are added to the active set. The number of rows to consider in each step is thus kept small in this strategy.

Alternatively, in the *simultaneous adaptive relaxation* method [53], a threshold θ is chosen. Rows with residual components larger than θ in magnitude are relaxed simultaneously. We note that such methods, like Jacobi, are not guaranteed to converge for all symmetric positive definite matrices, whereas such convergence is guaranteed for Multicolor Gauss-Seidel and Parallel Southwell, where an independent set of equations is relaxed simultaneously. Stagewise orthogonal matching pursuit methods also are accelerated by using the idea of a threshold to select multiple basis vectors simultaneously [55, 56].

In the context of large-scale optimization, greedy coordinate descent has been parallelized by partitioning the problem into subdomains. Each subdomain is solved using the greedy method corresponding to Sequential Southwell [57].

To reduce the number of messages sent and to improve the efficiency of an asynchronous iterative method, an *asynchronous variable threshold* method has been designed [36]. Here, thresholds are applied to the change in the solution after a block of equations corresponding to a subdomain or process has been relaxed (like in the sequential adaptive relaxation method mentioned above). If the change is too small, the update is not performed, and thus no messages need to be sent in this case. This method is not related to Distributed Southwell, but presents a possibility for further reducing communication cost.

Southwell-based techniques have been used by Rüde [52, 53] as adaptive smoothers for problems with irregular geometries or jumps in coefficients where there may be locally large residuals in the multigrid method. Sequential adaptive relaxation and simultaneous adaptive relaxation, mentioned above, were applied to augment a standard smoothing step.

Some parallelizable variants of the sequential Southwell can be found in the context of signal processing, specifically finding sparse solutions to underdetermined inverse problems. These methods are known as greedy pursuit methods[58, 59]. In [60], rows are grouped based on absolute value residual components, and the rows that reside in the bin with the largest collective magnitude is selected.

Randomized Kaczmarz and Gauss-Seidel [7] methods have similarities to our Stochas-

tic Parallel Southwell method, described in Section 3.5. In these methods, rows are relaxed based on probabilities that are different for each row. These probabilities are based on row or column norms, which is different than our method where probabilities are based on residual norms.

3.3 The Parallel Southwell Method

3.3.1 Mathematical Formulation

Before we introduce Parallel Southwell [61], we introduce some useful notation. For row i, row indices $\eta_j \neq i$ are *neighbors* of i if $a_{\eta_j i} \neq 0$. We define the *neighborhood* of row i as the set $N_i = \{\eta_1, \eta_2, \ldots, \eta_{q_i}\}$ of cardinality q_i, where each index in the set satisfies the neighbor requirement. We also define $\Gamma_i = \{|r_1|, |r_2|, \ldots, |r_{q_i}|\}$ as the set of corresponding absolute value residual components. An example of a neighborhood is shown in Figure 3.1, where the green points are neighbors of one of the red mesh points.

In Parallel Southwell, instead of computing a global maximum and relaxing a single row, at each parallel step, a row only relaxes if it calculates itself as having the maximum absolute value residual component in its neighborhood, i.e., row i relaxes if $|r_i|$ is maximum in $\{\Gamma, |r_i|\}$. This addresses the parallel shortcomings of the sequential Southwell method by (1) determining locally maximum absolute value residual components, and, as a result, (2) relaxing multiple rows per parallel step. More importantly, Parallel Southwell only requires rows to use neighbor information, which makes it a good candidate for an asynchronous implementation. This is because communication is limited to between neighbors, which means no global synchronization steps are required.

In element-wise form, Parallel Southwell is expressed as

$$x_i^{(t+1)} = \begin{cases} x_i^{(t)} + \dfrac{r_i^{(t)}}{a_{ii}}, & \text{if } |r_i| \text{ is maximum in } \{\Gamma, |r_i|\} \\ x_i^{(t)}, & \text{otherwise,} \end{cases} \tag{3.3}$$

$$r_{\eta_j}^{(t+1)} = r_{\eta_j}^{(t)} - r_i^{(t)} \frac{a_{ji}}{a_{ii}}, \text{ for all } \eta_j \in N_i, \ j = 1, \ldots, q_i. \tag{3.4}$$

Expressed in vector form,

$$x^{(t+1)} = (I - \tilde{D}A)x^{(t)} + \tilde{D}b, \tag{3.5}$$

$$\text{where } \tilde{D} = \begin{cases} \dfrac{1}{a_{ii}}, & \text{if } |r_i| \text{ is maximum in } \{\Gamma, |r_i|\}, \\ 0, & \text{otherwise.} \end{cases} \tag{3.6}$$

An important aspect of this method is that rows only determine if they, themselves, should relax, and do not instruct other rows to relax. Therefore, at each parallel step, an independent set of rows relax. Just like sequential Southwell, Parallel Southwell is a special case of what we will define as an *independent set iterative method*, which means that at any given parallel step, only a subset of rows are relaxed that form an independent set of points in the underlying geometry. Multicolor Gauss-Seidel is also an example of such a method. An example of selecting an independent set of rows to relax is shown in Figure 3.1.

Choosing an independent set of rows to relax on a given parallel step results in the following theorem.

Theorem 3. *If the iteration matrix G for a given parallel step is from an independent set iterative method, then G is a projector. Therefore, the spectral radius $\rho(G) = 1$.*

Proof. If we choose to not relax a particular row i, then the i-th diagonal of $\tilde{D}$ is zero, and the i-th diagonal of $I - \tilde{D}A$ is one, i.e., row i of $I - \tilde{D}A$ equals e_i^T. Therefore, G has zeros and ones on the diagonal. More importantly, because only an independent set of rows are to be relaxed, if row i relaxes, then all rows N_i of G are equal to $e_{\eta_j}^T$ for $j = 1, \ldots, q_i$. We

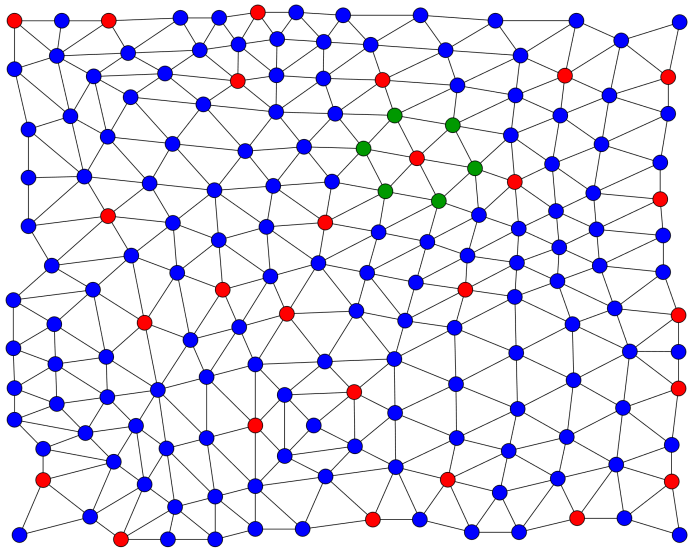


Figure 3.1: An example showing rows selected to relax during a parallel step of Parallel Southwell. The red mesh points denote the rows selected to be relaxed and represent an independent set. The green points denote the neighborhood of one of the red points.

show we can always reorder G to be either upper or lower triangular, which will maintain zeros and ones on the diagonal, resulting in zeros and ones as eigenvalues.

Since it does not matter if we reorder to upper or lower triangular, let us consider reordering G to lower triangular. To do this reordering, consider permuting row and column i of G. First, we find the right-most non-zero in row i at column j. We then swap row and column i and j, giving us the desired result. Because row j will always be equal to e_j^T, when permuting row and column i, the only other rows and columns that will be affected will be unit basis vectors, i.e., no other rows and columns that have non-zeros on the off-diagonal will be affected. With this process, we can always reorder G to upper or lower triangular, revealing eigenvalues of zero and one. □

The unit eigenvalues are benign in the sense that they do not alter the error in the components that are not relaxed. What this theorem shows is that there is a possibility that the Parallel Southwell method may converge even if the corresponding Jacobi method has spectral radius $\rho(G) > 1$ and diverges. We note that the multicolor Gauss-Seidel method also has $\rho(G) = 1$ for the iteration matrix G for each parallel step. However, since the

sequence of iteration matrices corresponding to a sweep of multicolor Gauss-Seidel is fixed, convergence in this case is more readily proved, in contrast to the Parallel Southwell case.

3.3.2 Implementation

It is natural to express Parallel Southwell as an asynchronous algorithm that can be implemented asynchronously on a shared memory machine with ℓ threads, shown in Algorithm 1, with variables defined in the following way:

- Each thread with ID τ (ranging from 0 to $\ell - 1$) is responsible for m_τ rows of the n total rows, where the partitioning $\{m_0, m_1, \ldots, m_{\ell-1}\}$ is determined beforehand (this is discussed in later in section).
- The values $\{\delta_0, \delta_1, \ldots, \delta_\ell\}$ are the $\ell + 1$ row index offsets, i.e., the prefix sum of $\{m_0, m_1, \ldots, m_{\ell-1}\}$.
- Each thread stores γ_τ, corresponding to the $\delta_\tau : (\delta_{\tau+1} - 1)$ (Matlab array notation) portion of the global residual array r.
- the value κ is the global index of a row, i.e., for i running from 0 to $m_{\tau-1}$ on thread τ, $\kappa = \delta_\tau + i$.

The Jacobi method can be expressed similarly, as shown in Algorithm 2.

To decrease the amount of accesses to shared memory, the only global array used was a single global residual array. Although it is not necessary for Jacobi and multicolor Gauss-Seidel to compute the residual, the residual must still be computed to check the convergence criteria. For Jacobi and Parallel Southwell, each thread stores a local copy of its portion of the matrix, and its portion of the residual and solution vectors. We define a subdomain as the set of unknowns that a thread is responsible for updating, and we define boundary points as points in a threads subdomain that have neighbor's on other threads. Given these definitions, a heuristic is that writing to the global array need only occur at the boundary

Algorithm 1: Parallel Southwell Method

Set $r = b - A \cdot x$
while *not converged on thread* τ **do**
  for $i = 0, 1, \ldots, m_\tau$ **do**
    if $|r_\kappa|$ *is maximum in* $\{\Gamma, |r_\kappa|\}$ **then**
      $x_i = x_i + \gamma_{\tau,i}/a_{ii}$
      for $j = \kappa, \eta_1, \eta_2, \ldots, \eta_{q_i}$ **do**
        $r_j = r_j - r_i \cdot a_{ji}/a_{ii}$
      end
    end
  end
  Set $\gamma_\tau = r_{\delta_\tau:(\delta_{\tau+1}-1)}$
end

Algorithm 2: Jacobi Method

Set $r = b - A \cdot x$
while *not converged on thread* τ **do**
  for $i = 0, 1, \ldots, m_\tau$ **do**
    $x_i = x_i + \gamma_{\tau,i}/a_{ii}$
    for $j = \kappa, \eta_1, \eta_2, \ldots, \eta_{q_i}$ **do**
      $r_j = r_j - r_i \cdot a_{ji}/a_{ii}$
    end
  end
  Set $\gamma_\tau = r_{\delta_\tau:(\delta_{\tau+1}-1)}$
end

points on a particular thread, so if a non-boundary point is encountered, a thread writes to its local residual array. When writing to the global array, atomic operations must be used to address race conditions. Race conditions can occur at boundary points, or when multiple threads contain points that share a neighbor.

We note that multiple boundary points on a thread may share neighbors on an adjacent thread. This means that some rows in the global array may be visited more than once during a parallel step, which is unnecessary. Eliminating this redundancy both reduces the amount of writes to the global array, and allows for vectorization, which is important when running code on the Intel Xeon Phi. To address this, each thread stores a local array that accumulates updates as rows are relaxed. The size of this array is the number of distinct boundary points. We also store a map that maps the global array elements to the this local array. We will refer to this as the residual map. The vectorization works as follows. When a row relaxes, the neighboring residuals are updated as shown in Algorithm 1, line 7. This involves a column of A multiplied with two scalars, and requires atomic operations if the global array is updated right away. Instead of eagerly updating the global array, we can accumulate updates in the local array, which can be vectorized using a gather operation. Doing this for each row accumulates all updates into the local array, and when the global array needs to be updated, the global array gathers values in the local array using the residual map. This second gather operation is not vectorizeable because it requires atomic operations, but the number of write operations to the global array is reduced.

Having control over this level of the implementation means that OpenMP parallel for loops cannot be used, because each thread needs to know beforehand which rows it is responsible for in order to construct the maps. Therefore, we used METIS[46] to reorder our problems into approximately balanced partitions. For multicolor Gauss-Seidel, the colors and partitions were determined by visiting the unknowns in breadth-first order. When implementing multicolor Gauss-Seidel, we used a simple OpenMP parallel for with guided scheduling, which we found to be the fastest, since it is difficult to construct balanced

multicolor sets.

3.3.3 Experimental Results

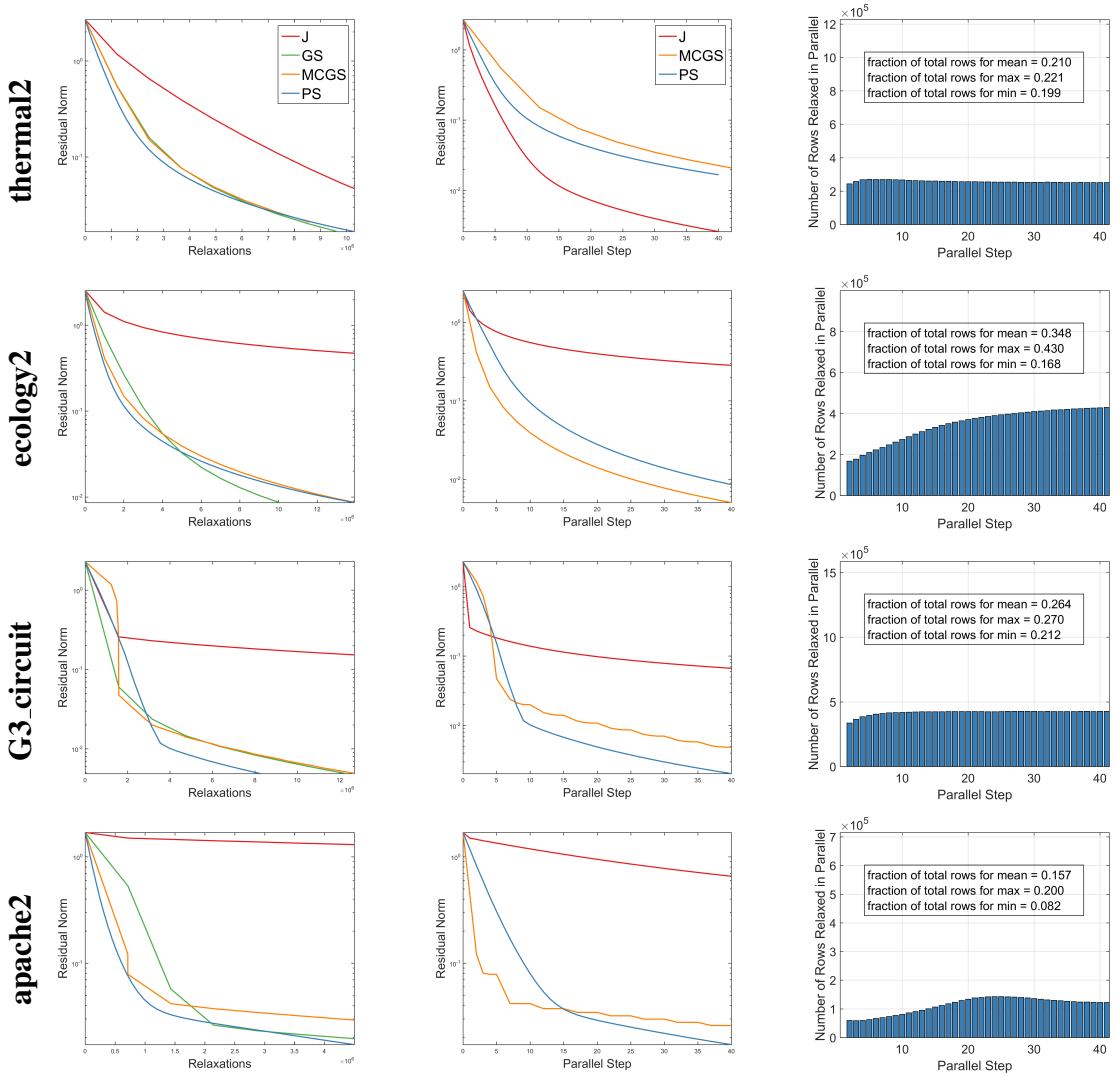


Figure 3.2: Comparison of Parallel Southwell (PS) with Jacobi (J), multicolor Gauss-Seidel (MGS), and Gauss-Seidel (GS). The rows denote four different test problems. The first column shows the residual norm as a function of the number of relaxations. The second column shows the residual norm as a function of the parallel step number. The last column shows the number of rows relaxed in parallel by Parallel Southwell for a given step number.

We first compare the convergence of Parallel Southwell (PS) to Jacobi (J), multicolor Gauss-Seidel (MCGS), and Gauss-Seidel (GS) implemented sequentially. The problems were taken from the University of Florida sparse matrix collection (Table 3.1). For these

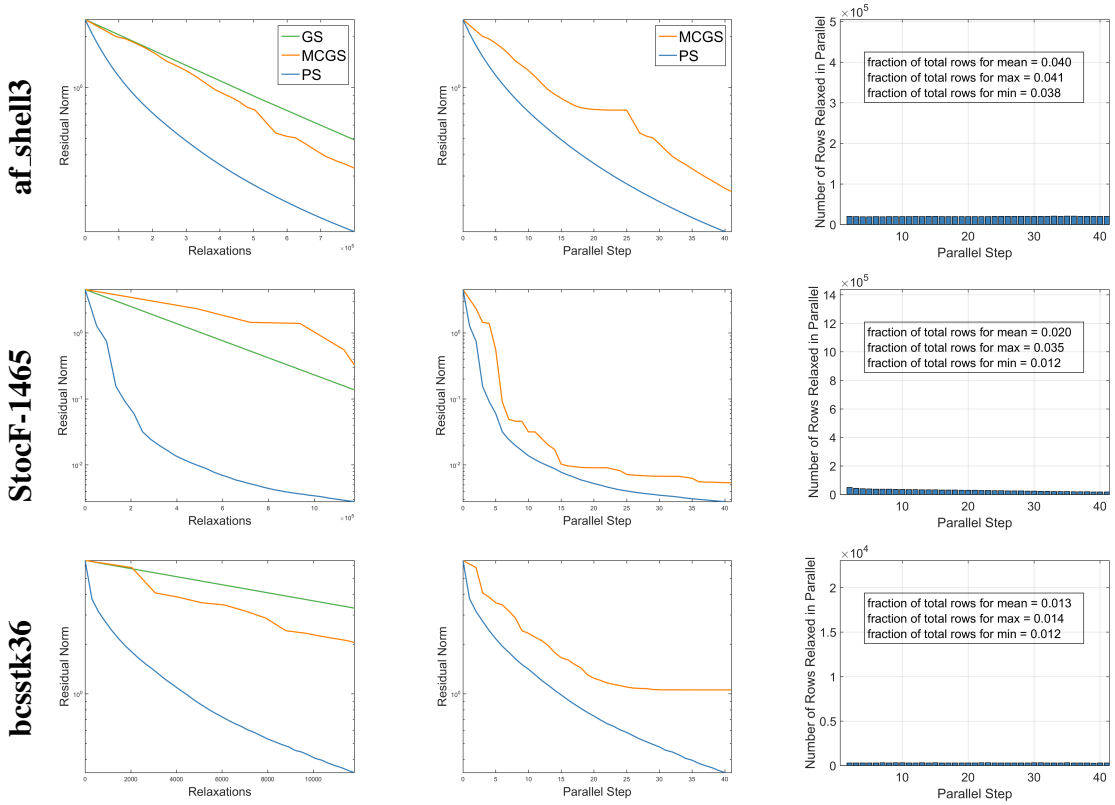


Figure 3.3: Comparison of Parallel Southwell (PS), multicolor Gauss-Seidel (MGS), and Gauss-Seidel (GS) for some matrices in which Jacobi does not converge. The rows denote three different test problems. The first column shows the residual norm as a function of the number of relaxations. The second column shows the residual norm as a function of the parallel step number. The last column shows the number of rows relaxed in parallel by Parallel Southwell for a given step number.

preliminary tests, we used about half of the problems from Table 3.1. For all the problems, we used a uniformly random initial guess, scaled such that the residual was of order unity, and a right-hand side of zeros. Because we are interested in early convergence behavior, i.e., the potential for Parallel Southwell as an efficient multigrid smoother and preconditioner, we ran each method for just 40 parallel steps.

The results are shown in Figures 3.2 and 3.3. Figure 3.3 shows results for some matrices in which Jacobi does not converge. For both figures, the first column shows the residual norm as a function of the number of relaxations, the second column shows the residual norm as a function of the number of parallel steps, and the third column shows the number

of rows relaxed by Parallel Southwell at each parallel step.

A general result when looking at the number of relaxations is that Parallel Southwell usually outperforms all other methods for short term convergence. In the case of red-black Gauss-Seidel, i.e., ecology2, Parallel Southwell still outperforms red-black Gauss-Seidel in the short term. If we consider multigrid smoothing, this shows that Parallel Southwell can be a better smoother than red-black Gauss-Seidel. However, in G3_circuit, we can see that multicolor Gauss-Seidel can also outperform Parallel Southwell in the short term.

A more interesting result can be seen when looking at the number of parallel steps Parallel Southwell takes to reach convergence. In ecology2 we can see that, although Jacobi initially performs better than Parallel Southwell in the first parallel step, Parallel Southwell ends up converging much faster than Jacobi even when doing less work per parallel step (Parallel Southwell relaxes only about .35 of the total rows per parallel step, shown in the last column of Figure 3.2). Parallel Southwell can also outperform multicolor Gauss-Seidel For example, in G3_circuit, we can see that although in the first few steps Jacobi and multicolor Gauss-Seidel outperform Parallel Southwell, Parallel Southwell eventually converges quicker. In this case, on average, Parallel Southwell does more work per parallel step than multicolor Gauss-Seidel. Notice the divot pattern in the parallel step behavior of multicolor Gauss-Seidel. This is due to the difficulty in load balancing the coloring, i.e., some parallel steps relax fewer rows than others. For af_shell3, we can see that Parallel Southwell outperforms multicolor Gauss-Seidel, but in this case, the two methods do approximately the same work per parallel step. This demonstrates the effectiveness of greedily choosing rows to relax, i.e., choosing rows with high absolute value residual norms.

For the remaining test problems, Parallel Southwell performs at its best. Specifically, Parallel Southwell can do less work than multicolor Gauss-Seidel and still converge quicker per parallel step. For example, in StocF-1465, Parallel Southwell, on average, relaxes about .04 of the total rows, where as multicolor Gauss-Seidel relaxes about .09 of its rows. For apache2, we can see that multicolor Gauss-Seidel outperforms Parallel Southwell initially,

but eventually Parallel Southwell overtakes multicolor Gauss-Seidel. Additionally, when Parallel Southwell overtakes multicolor Gauss-Seidel, the convergence rate is faster. This is surpising because Parallel Southwell only relaxes about .15 the number of equations in parallel and multicolor Gauss-Seidel relaxes approximately .2 of the total equations.

All these properties can be seen for bcsstk36. We see that Parallel Southwell converges in fewer relaxations. We see that Parallel Southwell converges faster per parallel step, and with a higher convergence rate. Even with this parallel step behavior, we see that it does less work per parallel step, i.e., it relaxes about .013 rows on average, where as multicolor Gauss-Seidel relaxes about .24.

Table 3.1: Test problems for parallel experiments in shared memory with OpenMP. FD and FE are 5-point centered difference and unstructured finite element discretizations of the Laplace equation, respectively, and the remaining matrices are from the University of Florida sparse matrix collection.

matrix name	num nonzeros	num equations	num colors
bcsstk36	1143140	23052	41
FE	1407811	203841	5
2cubes_sphere	1647264	101492	12
cfd2	3085406	123440	15
parabolic_fem	3674625	525825	4
offshore	4242673	259789	13
apache2	4817870	715176	5
ecology2	4995991	999999	2
FD	4996000	1000000	2
G3_circuit	7660826	1585478	5
thermal2	8579355	1227087	6
af_shell3	17562051	504855	25
StocF-1465	20976285	1436033	11

We now show timings on two different shared memory platforms. We used two different processing platforms:

1. 2x Intel Xeon E5-2650 CPUs v3 with a 2.3GHz clock speed and 10 cores.

2. Intel Xeon Phi co-processor with a 1.05GHz clock speed and 60 physical cores and 4 threads per core.

For these experiments, we ran on all the matrices shown in Table 3.1, making sure to reduce the residual norm below 10^{-1}. The convergence criteria for the synchronous methods is to stop when the global residual norm falls below some threshold. For the asynchronous methods, a convergence criteria similar to that in [62] is used, where each thread only checks its local part of the residual. Once the local criteria is met, a shared counter is incremented atomically. Once the shared counter is equal to the number of threads, the iteration terminates. Unlike in [62], we do not continue if, upon termination, the master thread calculates a residual norm that is above the tolerance. We found that the residual norm was usually very close with the tolerance. We chose a uniformly random initial guess and a zero right-hand side, with the initial guess scaled such that the residual was of order unity. Because iterative solves generally do not stop with the exact same residual norm, and, in the case of the asynchronous methods, a number of parallel steps cannot be reached exactly, we used interpolation on the residual norm to extract the number of parallel steps and relaxations required for each method to reach convergence.

The results are shown in Tables 3.2 and 3.3 for the Intel Xeon CPUs and Intel Xeon Phi, respectively. We used the maximum number of threads that gave us the best times, which was 20 for the Intel Xeon CPUs, and 180 for the Intel Xeon Phi. For about half the matrices, Jacobi did not converge, which is why some timings are missing. We can see that multicolor Gauss-Seidel is much faster than Parallel Southwell, even when Parallel Southwell converges quicker. This is due to Parallel Southwell having to compute a max at each parallel step. However, for thermal2, i.e., the one case in which Jacobi can compete with multicolor Gauss-Seidel, both Jacobi and asynchronous Jacobi converge with a shorter wall clock time on the Intel Xeon Phi, despite doing more relaxations. This demonstrates the two issues discussed in the implementation (lack of vectorization and abundance of write operations to shared memory), issues of which are amplified on the the Intel Xeon Phi. When looking at the Intel Xeon CPUs results, we can see that this is less of an issue, where asynchronous Jacobi only outperforms multicolor Gauss-Seidel.

When comparing synchronous and asynchronous Parallel Southwell, we can see that the asynchronous methods often take additional relaxations to converge. This could be happening for a couple reasons. First, as discussed in the section on asynchronous iterative methods, asynchronous methods may continue even when information is not up to date. This means that some rows may perform additional relaxations with stale information, i.e., information that was used in an earlier relaxation. Second, the convergence criteria requires that all threads reach the criteria in order for all threads to terminate. Therefore, some threads may do additional relaxations while waiting for other threads to catch up. However, given these reasons, even when doing some more relaxations to converge, the asynchronous methods often take less wall clock time.

In figure 3.4, we look at how a slight load imbalance can effect the wall clock time when using a high number of threads. In this figure, we did a weak scaling experiment on both the Intel Xeon CPUs and Intel Xeon Phi, ranging from 1 to 20 and 1 to 180 threads, respectively. As in the previous experiments, we used interpolation to extract an exact residual norm value of 10^{-1}. In the first column, we look at the difference in wall clock time when we have the desired load balancing resulting from METIS. For both platforms, as we increase the number of threads, asynchronous requires less wall clock time to converge. Additionally, as the number of cores increases, the difference is more noticeable. This can be scene when comparing the timings at the maximum thread count for each platform, and noticing that the difference in timings between asynchronous and synchronous is larger for 180 threads on the Intel Xeon Phi than at 20 threads on the Intel Xeon CPUs.

3.4 The Distributed Southwell Method

3.4.1 Block Methods on Distributed Memory Computers

For the Jacobi and Parallel Southwell methods on a distributed memory machine, it is natural to partition a problem into non-overlapping subdomains, with one subdomain for each process. To approximately solve the local subdomain problems, Gauss-Seidel may

Table 3.2: Results for Intel Xeon E5-2650 CPUs using all 20 cores. The subcolumns denote the methods, specifically, asynchronous and synchronous Parallel Southwell (APS and SPS, respectively), multicolor Gauss-Seidel (MCGS), and asynchronous and synchronous Jacobi (AJ and SJ, respectively).

	Time					Relaxations/(num equations)					Parallel Step		
matrix	APS	SPS	MCGS	AJ	SJ	APS	SPS	MCGS	AJ	SJ	SPS	MCGS	SJ
FD	0.053	0.062	0.010	0.536	0.979	2.295	2.240	2.767	284.951	318.694	10	6	319
parabolic_fem	0.106	0.149	0.017	2.779	8.184	5.073	4.956	5.560	2762.495	5175.324	37	22	5175
apache2	0.144	0.186	0.047	4.948	10.893	5.276	5.203	11.957	3316.831	4404.425	34	60	4404
ecology2	0.052	0.062	0.010	0.593	1.044	2.237	2.202	2.740	307.517	342.704	10	5	343
G3_circuit	0.086	0.092	0.013	0.991	1.276	2.025	1.851	1.991	279.696	269.793	9	10	270
thermal2	0.112	0.139	0.023	0.020	0.026	1.773	1.743	2.341	6.087	5.702	13	14	6
bcsstk36	0.048	0.068	0.019			1.375	1.355	5.376			114	214	
FE	0.036	0.043	0.009			6.416	5.849	6.423			40	32	
2cubes_sphere	0.026	0.034	0.006			0.414	0.411	1.914			34	20	
cfd2	0.081	0.118	0.026			2.051	1.995	4.878			61	68	
offshore	0.123	0.162	0.025			0.398	0.375	3.646			50	44	
af_shell3	0.614	0.886	0.063			1.887	1.839	3.117			77	77	
StocF-1465	0.180	0.230	0.041			0.365	0.343	1.876			13	17	

Table 3.3: Results for Intel Xeon Phi using 180 threads. The subcolumns denote the methods, specifically, asynchronous and synchronous Parallel Southwell (APS and SPS, respectively), multicolor Gauss-Seidel (MCGS), and asynchronous and synchronous Jacobi (AJ and SJ, respectively).

	Time					Relaxations/(num equations)					Parallel Step		
matrix	APS	SPS	MCGS	AJ	SJ	APS	SPS	MCGS	AJ	SJ	SPS	MCGS	SJ
FD	0.129	0.072	0.023	0.793	1.663	6.107	2.243	2.766	207.117	335.327	10	6	335
parabolic_fem	0.049	0.144	0.018	0.264	0.619	1.612	1.586	1.991	113.566	208.427	12	8	208
apache2	0.177	0.225	0.100	5.179	29.464	5.188	5.181	11.917	1696.096	7447.935	34	60	7448
ecology2	0.088	0.071	0.023	0.824	1.824	3.786	2.205	2.739	215.168	364.546	10	5	365
G3_circuit	0.101	0.119	0.032	1.368	2.020	2.274	2.029	1.994	223.710	257.619	10	10	258
thermal2	0.158	0.202	0.060	0.033	0.042	2.037	2.020	2.680	6.378	6.258	15	16	6
bcsstk36	0.107	0.132	0.047			1.367	1.324	5.397			111	214	
FE	0.061	0.074	0.034			5.952	5.858	6.454			40	32	
2cubes_sphere	0.062	0.080	0.016			0.395	0.390	1.917			39	20	
cfd2	0.181	0.224	0.053			2.031	2.007	4.984			62	69	
offshore	0.236	0.274	0.061			0.387	0.377	3.667			51	44	
af_shell3	1.247	1.453	0.124			1.837	1.833	3.121			76	77	
StocF-1465	0.280	0.357	0.067			0.367	0.350	1.645			13	15	

be used. In the case of Jacobi, this is often referred to as Hybrid Gauss-Seidel[63, 64], or Processor Block Gauss-Seidel[47, 48].

We use the following notation:

- Each parallel process with rank p (ranging from 0 to $\mathcal{P}-1$, where $\mathcal{P}$ is the total number of processes) is responsible for m_p rows of the n total rows, where the partitioning is determined, e.g., with METIS [46].

- The values $\{\delta_0, \delta_1, \ldots, \delta_{\mathcal{P}}\}$ are the $\mathcal{P}+1$ row index offsets, i.e., the prefix sum of

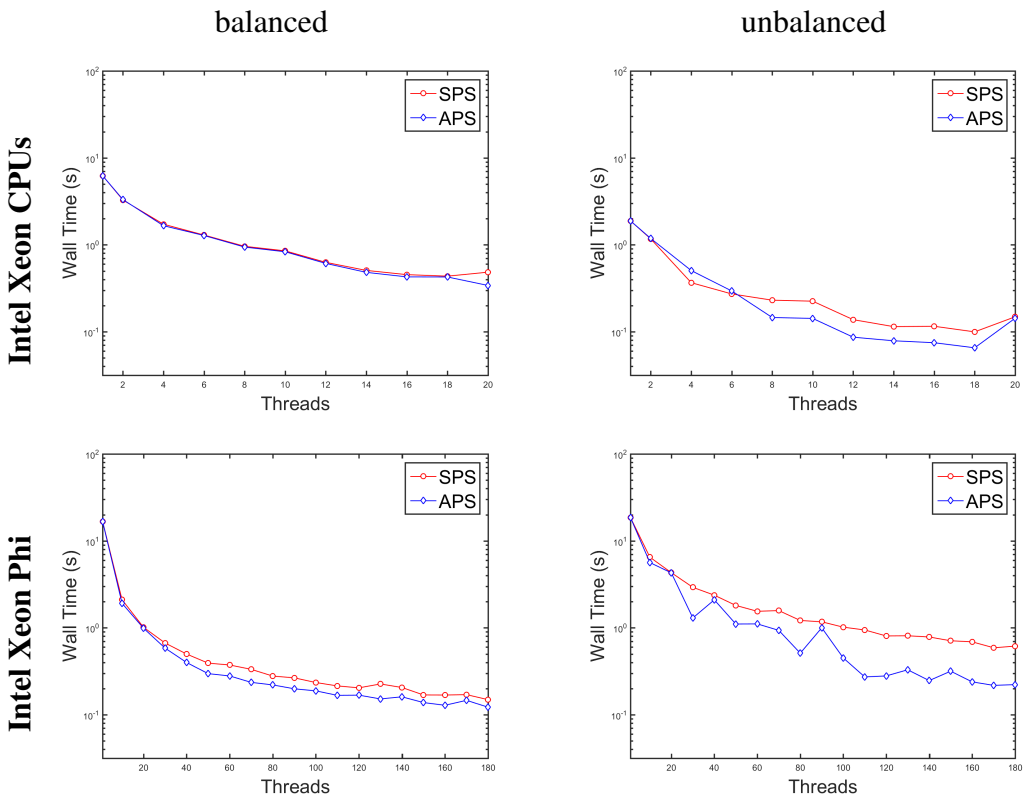


Figure 3.4: Comparison between synchronous and asynchronous Parallel Southwell as the number of threads increases. The test problem is bcsstk36, and both the Intel Xeon CPUs and Intel Xeon Phi are used. The first column denotes a balanced partitioning produced by METIS, while the second denotes a slightly unbalanced partitioning.

$\{0, m_0, m_1, \ldots, m_{\mathcal{P}-1}\}$.

- Each process stores r_p and x_p corresponding to the $\vec{\delta}_p = \delta_p : (\delta_{p+1} - 1)$ (Matlab array notation) portion of the global residual and solution arrays, respectively.

- We use the one-sided memory model, where p has a region of memory that remote processes can directly write to without the involvement of p. We define this region of memory as the *memory window* $\mathcal{W}_p$ of p.

In this notation, the Block Jacobi algorithm is shown in Algorithm 3.

For the block form of Parallel Southwell, instead of comparing the magnitude of individual residual vector components, we now compare the residual norm for the rows of process p to the residual norms for the rows that belong to the neighbors of p, i.e., we

Algorithm 3: Block Jacobi

Set $r = b - Ax$
for *each process with rank* p **do**
    Set $r_p = r(\vec{\delta}_p)$
    Set $x_p = x(\vec{\delta}_p)$
end
for $k = 1, \ldots, k_{max}$ *on process with rank* p **do**
    Update x_p and r_p by relaxing the equations belonging to p
    Write updates to $\{\mathcal{W}_1, \ldots, \mathcal{W}_{q_p}\}$
    Wait for neighbors to finish writing to $\mathcal{W}_p$
    Read from $\mathcal{W}_p$ to update r_p
end

redefine $\Gamma_p = \{\|r_1\|_2, \|r_2\|_2, \ldots, \|r_{q_p}\|_2\}$ where we assume the neighboring processes have indices $1, 2, \ldots, q_p$. If process p satisfies the Parallel Southwell criterion, it relaxes the equations in its subdomain and sends updates to its neighbors. Upon receiving these updates, the neighbors of p use this new information to update their boundary points. Additionally, at each parallel step, an extra communication step is needed in order for p to know the residual norms that belong to its neighbors. We call this an *explicit residual update*, where a process sends its updated residual norm to its neighbors.

A geometric interpretation of the block version of Parallel Southwell is shown in Figure 3.5, where the top mesh shows the partitioning, and the bottom mesh shows the subdomains that are selected to be updated via the Parallel Southwell criterion.

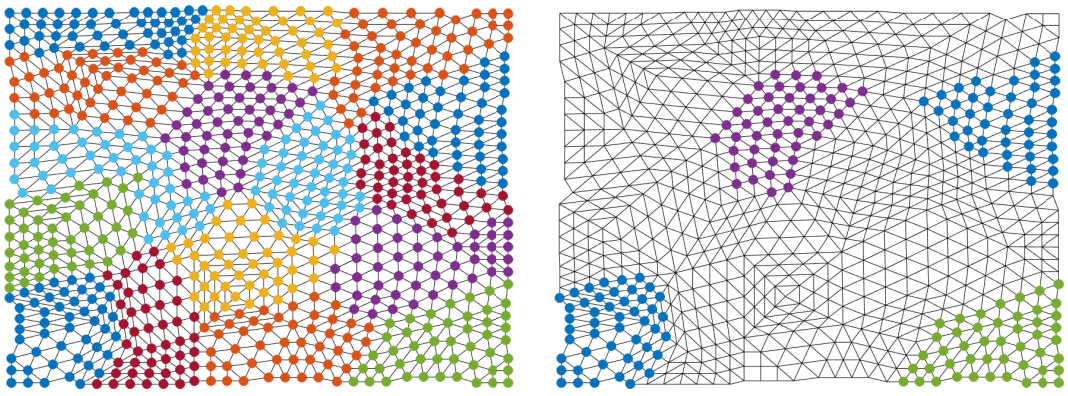
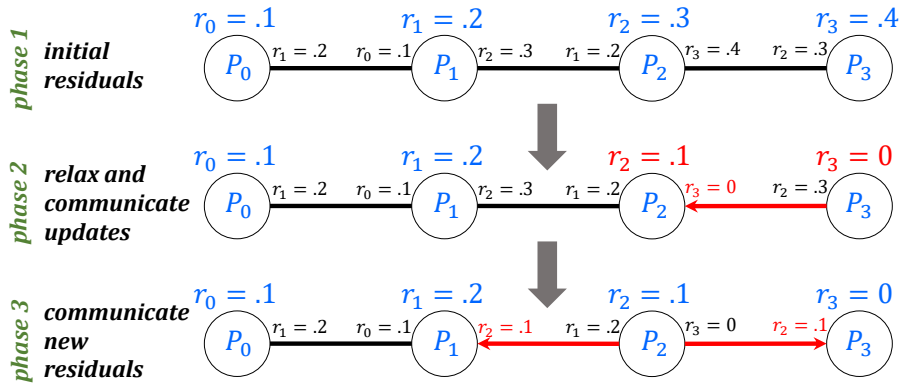


Figure 3.5: Parallel Southwell with multiple equations per process. Top: the subdomains assigned to each process. Bottom: four subdomains selected via the Parallel Southwell criterion.

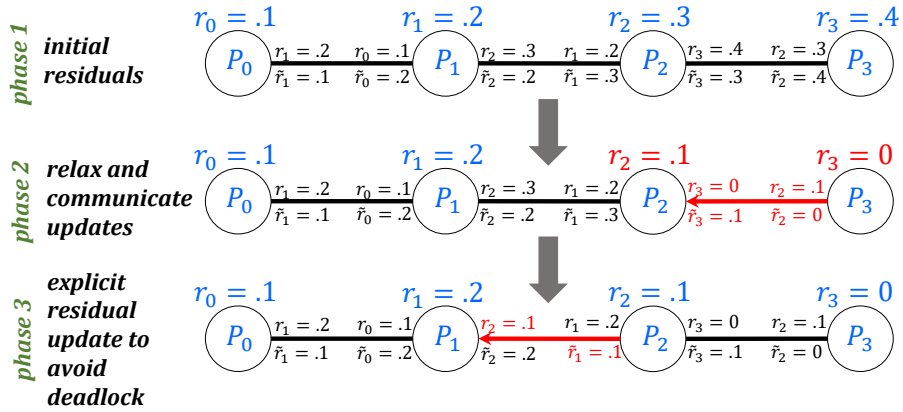
An illustration of the key phases of a parallel step of Parallel Southwell is shown in Figure 3.6(a). The illustration shows four processes, where the edges connecting them indicate a neighbor relationship. In phase 1 of the figure, p_3 is the only process that determines that it must update. In phase 2, p_3 updates, and writes to the memory of p_2, which counts as a single message. This changes the residual of p_2, and updates the copy of the residual of p_3 held by p_2. In phase 3, p_2 detects that its own residual has changed, so it updates the copies of its residual that p_1 and p_3 hold, which requires two additional messages to be sent.

Algorithm 4 shows the block form of Parallel Southwell implemented in distributed memory. To be clear, this algorithm is mathematically identical to Parallel Southwell implemented in shared memory. Note that this algorithm is different than that introduced in [61], which can possibly deadlock. To observe how that algorithm might deadlock, consider again Figure 3.6(a) but now assume that p_2 does not communicate its updates, i.e., if we remove the explicit residual norm update from the last phase of the diagram. Deadlock will now occur. This can be seen at phase 2, where the true residual norm of each process (r_i in blue shown above each node) is less than its copies of the residual norm of its neighbors (r_i in black on the right and left of each node, above the connection edge), resulting in all processes failing to satisfy the Parallel Southwell criterion.

There are a few communication-reducing optimizations shown in Algorithm 4. First, if process p does not relax its rows, and none of its neighbors relax their rows, there is no need for p to send its residual to its neighbors because its residual has not changed. This is shown in the `If` statement on line 19. Second, if p does satisfy the Parallel Southwell criterion, it can append its new residual norm to all outgoing messages, which eliminates the need to communicate its new residual to its neighbors in a separate message. This is shown in line 10.



(a) **Parallel Southwell parallel step**



(b) **Distributed Southwell parallel step**

Figure 3.6: Illustration of a parallel step of (a) Parallel Southwell and (b) Distributed Southwell. In the illustration, a line of four processes $P_0, \ldots, P_3$, with an array communication topology, start the parallel step with exact residuals $r_0, \ldots, r_3$ (shown in blue above the corresponding P), and their estimates of the residual norms of their neighbors (shown in black above the inter-process connections). Additionally, in (b), each P_i also stores the estimate of the residual norm of P_i stored by the neighbors of P_i, denoted by $\tilde{r}_0, \ldots, \tilde{r}_3$. Each line of four processes, three lines in total, denotes a phase of the parallel step. Red residuals denote an updated residual, and red arrow connections denote communication. Note that the illustration is not based on any data taken from any real experiments.

3.4.2 The Distributed Southwell Method

The premise of the Distributed Southwell method is that the residuals for the equations on neighboring processes do not need to be known exactly [65]. These residuals are only needed for processes to determine if they should relax their own equations. That this step

Algorithm 4: Parallel Southwell (block version)

Set $r = b - Ax$
for *each process with rank* p **do**
    Set $r_p = r(\vec{\delta}_p)$
    Set $x_p = x(\vec{\delta}_p)$
    Set $\Gamma_p = \{\|r_1\|_2, \ldots, \|r_{q_p}\|_2\}$
end
for $k = 1, \ldots, k_{max}$ *on process with rank* p **do**
    if $\|r_p\|_2$ *is maximum in* $\{\Gamma, \|r_p\|_2\}$ **then**
        Update x_p and r_p by relaxing the equations belonging to p
        Write updates and $\|r_p\|_2$ to $\{\mathcal{W}_1, \ldots, \mathcal{W}_{q_p}\}$
    else
        Wait for neighbors to finish writing to $\mathcal{W}_p$
        for $j = 1, \ldots, q_p$ **do**
            if *Neighbor* q_j *has written new information to* $\mathcal{W}_p$ **then**
                Read from $\mathcal{W}_p$ to update r_p
                Update $\|r_{q_j}\|_2$ in Γ
            end
        end
        if $\|r_p\|_2$ *has changed* **then**
            Write $\|r_p\|_2$ to $\{\mathcal{W}_1, \ldots, \mathcal{W}_{q_p}\}$
        end
    end
    Wait for neighbors to finish writing to $\mathcal{W}_p$
    for $j = 1, \ldots, q_p$ **do**
        if *Neighbor* q_j *has written new information to* $\mathcal{W}_p$ **then**
            Update $\|r_{q_j}\|_2$ in Γ
        end
    end
end

is done precisely following the Parallel Southwell criterion is not essential.

This premise allows many possibilities for reducing communication. In particular, processes do not need to carry out explicit residual updates every time their residual norm changes. Instead, a process p can maintain estimates of the residuals for the equations on neighboring processes. When a process p relaxes its own equations, it knows how these relaxations affect the residual on its neighbor q, without any communication. Referring to Equation 3.2, the update

$$-r_i^{(t)} \frac{a_{\eta_j i}}{a_{ii}}$$

to the neighbor residual $r_{\eta_j}^{(t)}$ only depends on local information, in particular $r_i^{(t)}$, while $a_{\eta_j i}$ and a_{ii} are matrix data that can be stored locally (i.e., the process responsible for row i stores column i of A).

Consider now a neighbor s of q, so that the matrix dependencies are $p \iff q \iff s$. If a neighbor s of process q relaxes its equations, then the effect on the residual of q will not be known to process p. This is how the estimate that process p has of the residual norm of process q loses accuracy. Again referring to Equation 3.2, the size of the discrepancy is related to the size of the residual component, i.e., it decreases in size as the iterations progress.

As explained in Section 3.4.1, there is a major drawback of using inaccurate residuals. If the residual norm estimates on all processes is such that no process thinks it has the largest residual norm, then deadlock occurs. This means that there is a risk of deadlock if the residual norm estimates are larger than the actual residual norms. Fortunately, this situation can be detected by a process q maintaining a copy of the estimate that p has of the residual norm of q. This copy can be maintained, like above, without communication. Thus q can detect if the estimates of its residual norm are larger than its actual residual norm. In this case, q sends p an explicit message to update its estimate. Deadlock is thus avoided.

Figure 3.7 shows the convergence of Distributed Southwell compared with other methods. The matrix is from a finite element discretization of the Poisson equation on a square domain. Irregularly structured linear triangular elements are used. The discrete right-hand side has elements sampled from a uniform random distribution with mean zero and is scaled such that its 2-norm is 1. The example problem has 3081 rows and the convergence for three sweeps of each method is shown. The convergence curves for Sequential Southwell, Parallel Southwell, and Multicolor Gauss-Seidel are shown. All methods in this figure use their scalar forms (i.e., subdomain size of 1). We observe that the behavior of Distributed Southwell closely matches that of Parallel Southwell (which uses the exact Parallel Southwell criterion for choosing which equations to relax) for low levels of accuracy (e.g., residual

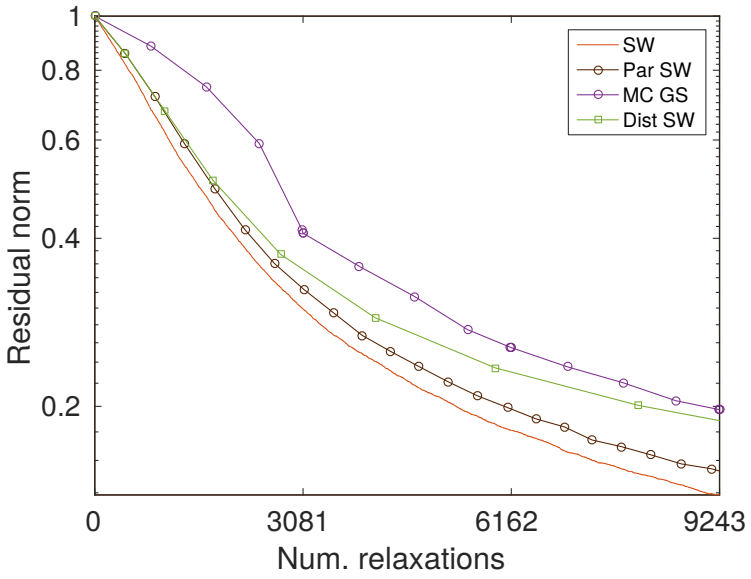


Figure 3.7: Convergence for a small finite element problem. Distributed Southwell is compared to other methods (all in scalar form). The markers along the curves for the parallel methods delineate the parallel steps.

norm 0.6), which is the "sweet spot" for using Southwell-like methods compared to using Gauss-Seidel. We also observe that with inexact residual estimates, Distributed Southwell relaxes more equations per parallel step, as shown by the markers along the curves in the figure. This may account for the degraded convergence of Distributed Southwell compared to Parallel Southwell as more parallel steps are taken.

The Distributed Southwell idea can be easily extended to use subdomains in a practical distributed code. Here, process p stores a ghost layer of residuals corresponding to all off-processor connections to the boundary points of p, where z_{q_j} denotes the residual ghost layer for the points β_{q_j} of neighbor q_j. When process p relaxes its equations, it also updates all points in the ghost layer, and uses this to update all the residual norms in Γ. These updates denote the contribution of p to the residual norm of its neighbors. This allows p to store more accurate copies of the residual norms of its neighbors, in the case that p updates often without receiving updates from neighbors. When p receives updates from neighbors, the values in the ghost layer and Γ are corrected.

In addition to p storing Γ, p also stores $\tilde{\Gamma}$, which are the residual norms of p stored

by the neighbors of p. When neighbor q_1 of p updates and writes to the memory of p, it includes its new estimate of the residual of p in the message. In the memory of p, this is the value $\|\tilde{r}_{q_1}\|_2$. This value is always exactly known by p, since only p and q_1 alter the estimate of the residual norm of p stored by q_1. If p determines that $\|\tilde{r}_{q_1}\|_2 > \|r_p\|_2$, then there is a possibility of deadlock, and p communicates its residual norm and boundary points to q_1, which brings the estimate of the residual norm of p stored by q_1 up to date.

The algorithm for Distributed Southwell (in block or subdomain form) is shown in Algorithm 5. An illustration of the key phases of a parallel step of Distributed Southwell is shown in Figure 3.6(b). As in (a), p_3 updates, but also updates its estimate of the residual norm of p_2, obtaining the new residual norm of p_2 exactly. If p_1 were to also update the residual norm of p_2 in this phase, p_3 would not have an exact estimate of the residual norm of p_2, but it would be a better estimate than if p_3 did nothing at all. In phase 3, p_2 detects possible deadlock on p_1, and sends a single message that updates the estimate of the residual norm of p_2 stored by p_1.

Our distributed implementations (for all algorithms including Distributed Southwell) use the one-sided semantics provided in MPI-3, also known as remote memory access (RMA)[44]. For one-sided, an origin process writes directly to the memory of a target process without the target being involved in the transfer of data, avoiding the communication required by the receiving side. Another reason we use one-sided functions is that in Parallel and Distributed Southwell, it is not always clear when, or if, a process should expect a message from a neighbor. Each process initially makes an MPI group consisting of its neighbors, and calls `MPI_Win_allocate()` to create a memory window, i.e., allocates a region of memory that is accessible by remote processes. During communication phases, assuming all processes have information to send, processes enter access epochs by calling `MPI_Win_post()` followed by `MPI_Win_start()`. Messages are then sent using `MPI_Put()`, and the epochs are ended using `MPI_Win_complete()` followed by `MPI_Win_wait()`. A process and all its neighbors must collectively call all commands

Algorithm 5: Distributed Southwell (block version)

Set $r = b - Ax$
for *each process with rank* p **do**
    Set $r_p = r(\vec{\delta}_p)$
    Set $x_p = x(\vec{\delta}_p)$
    Set $\Gamma_p = \{\|r_1\|_2, \ldots, \|r_{q_p}\|_2\}$
    Set $\tilde{\Gamma}_p = \{\|\tilde{r}_1\|_2, \ldots, \|\tilde{r}_{q_p}\|_2\}$
    for $j = 1, \ldots, q_p$ **do**
        Set $z_{q_j} = r(\beta_{q_j})$
    end
end
for $k = 1, \ldots, k_{max}$ *on process with rank* p **do**
    if $\|r_p\|_2$ *is maximum in* $\{\Gamma_p, \|r_p\|_2\}$ **then**
        Update x_p and r_p by relaxing the equations belonging to p
        for $j = 1, \ldots, q_p$ **do**
            Update z_{q_j} and compute $\|r_{q_j}\|_2$
            Set $\|\tilde{r}_{q_j}\|_2 = \|r_p\|_2$
            Write updates, z_p, $\|r_p\|_2$, and $\|r_{q_j}\|_2$ to $\mathcal{W}_{q_j}$
        end
    end
    Wait for neighbors to finish writing to $\mathcal{W}_p$
    for $j = 1, \ldots, q_p$ **do**
        if *Neighbor* q_j *has written new information to* $\mathcal{W}_p$ **then**
            Update r_p
            Overwrite z_{q_j}
            Overwrite $\|r_{q_j}\|_2$ in Γ and $\|\tilde{r}_{q_j}\|_2$ in $\tilde{\Gamma}$
        end
        if $\|r_p\|_2 < \|\tilde{r}_{q_j}\|_2$ **then**
            Set $\|\tilde{r}_{q_j}\|_2 = \|r_p\|_2$
            Write z_{q_j}, $\|r_p\|_2$ and $\|r_{q_j}\|_2$ to $\mathcal{W}_{q_j}$
        end
    end
    Wait for neighbors to finish writing to $\mathcal{W}_p$
    for $j = 1, \ldots, q_p$ **do**
        if *Neighbor* q_j *has written new information to* $\mathcal{W}_p$ **then**
            Overwrite z_{q_j}
            Overwrite $\|r_{q_j}\|_2$ in Γ and $\|\tilde{r}_{q_j}\|_2$ in $\tilde{\Gamma}$
        end
    end
end

for starting and ending the access epochs.

3.4.3 Experimental Results

Multigrid Smoothing

We first test the use of Distributed Southwell as a smoother for the multigrid method. Here, we use a scalar rather than block version of Distributed Southwell. The test problem is the 2D Poisson equation on a square discretized on a regular mesh by centered finite differences. The discrete right-hand side is chosen to be a vector with random entries uniformly distributed between -1 and 1. To test multigrid convergence, the grid dimensions are increased from 15×15 to 255×255. Each V-cycle uses multiple levels such that the coarsest level corresponds to a 3×3 grid, at which an exact solve is used.

Each V-cycle uses one step of pre-smoothing and one step of post-smoothing. As a baseline for comparison, we use Gauss-Seidel as a smoother. For Distributed Southwell as a smoother, we use a number of relaxations corresponding to exactly the number of relaxations as Gauss-Seidel (i.e., the number of unknowns in the grid at a given level, called "1 sweep"). We also test Distributed Southwell using half of the number of relaxations of Gauss-Seidel (called "1/2 sweep"). Distributed Southwell selects many rows to be relaxed simultaneously in a single parallel step. In order to achieve an exact total number of relaxations (for our comparison purposes), in the final parallel step of Distributed Southwell, a random subset of the rows selected to be relaxed are actually relaxed.

Figure 3.8 shows the residual norm relative to the initial residual norm after 9 V-cycles. The most important result is that Distributed Southwell as a smoother shows grid-size independent convergence, even though some rows may never have been relaxed in the smoother, which is particularly true in the case of "1/2 sweep." We also observe that Distributed Southwell is a more efficient smoother than Gauss-Seidel, resulting in better multigrid convergence even when Distributed Southwell uses the same number of relaxations as Gauss-Seidel.

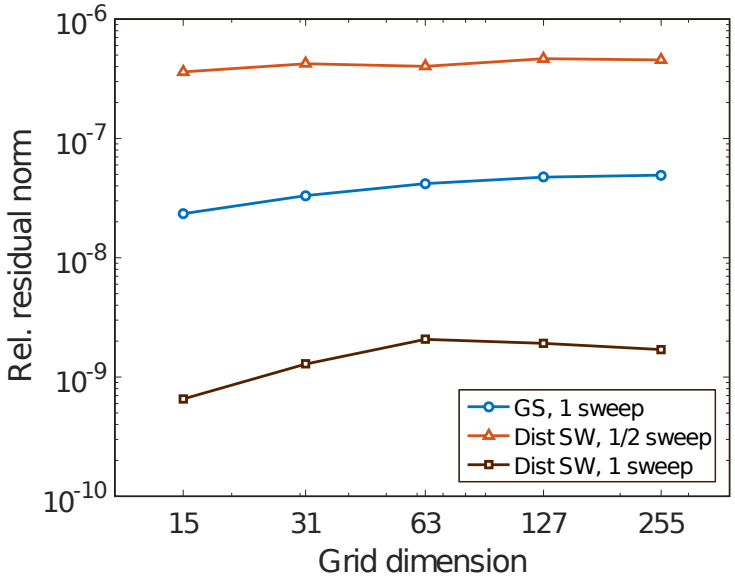


Figure 3.8: Relative residual norm after 9 V-cycles of multigrid applied to solving the 2D Poisson equation for increasing grid dimensions. Distributed Southwell as a smoother is compared to Gauss-Seidel (GS) as a smoother. The results show that convergence is independent of grid size in all cases. In addition, Distributed Southwell is more efficient as a smoother, per relaxation, than Gauss-Seidel.

Test Framework

In the following experiments, we compare Distributed Southwell, Parallel Southwell, and Block Jacobi implemented in distributed memory. Here, we used a random initial guess and a right-hand side $b = 0$. We scaled all initial guesses such that $\|r^{(0)}\|_2 = 1$. All test matrices are shown in Table 3.4, which were taken from the SuiteSparse Matrix Collection[45], and symmetrically scaled to have unit diagonal values. We used up to 256 32-core nodes on the NERSC Cori (Phase I) supercomputer. We varied the number of parallel steps from zero to 50, and took 50 samples at each parallel step. Out of 50 samples, we used the run that gave us the lowest wall-clock time, i.e., we considered the best time a method could obtain at a given parallel step.

For all methods, when a process updates, a single Gauss-Seidel sweep is carried out on the subdomain that the process is responsible for. We note that a single process per node could be used, with a multi-threaded local solver, e.g., Multicolor Gauss-Seidel. Another

Table 3.4: Test problems from the SuiteSparse Matrix Collection. All matrices are symmetric positive definite.

Matrix	Number of Non-zeros	Number of Equations
Flan_1565	114,165,372	1,564,794
audikw_1	77,651,847	943,695
Serena	64,122,743	1,382,121
Geo_1438	60,169,842	1,371,480
Hook_1498	59,344,451	1,468,023
bone010	47,851,783	986,703
ldoor	42,451,151	909,537
boneS10	40,878,708	914,898
Emilia_923	40,359,114	908,712
inline_1	36,816,170	503,712
Fault_639	27,224,065	616,923
StocF-1465	20,976,285	1,436,033
msdoor	19,162,085	404,785
af_5_k101	17,550,675	503,625

important note is that we are using the Parallel Southwell method as defined in Algorithm 4, and not as defined in Section 3.3. This is because Parallel Southwell as defined in [61] deadlocks for all our test problems.

Reducing $\|r\|_2$ to 0.1 Using 8192 Processes

Table 3.5 shows results for reducing $\|r\|_2$ to 0.1 with 8192 MPI processes (256 nodes). The table shows the wall-clock time, communication cost, number of parallel steps, number of relaxations, and the number of active processes. "Communication cost" is defined as the total number of messages sent by all processes, divided by the total number of processes. "Active processes" is defined as the average fraction of processes carrying out block relaxations of local subdomains at each parallel step.

The table shows that Block Jacobi can achieve $\|r\|_2 = 0.1$ for only three of the test matrices. Figure 3.9 plots the convergence with respect to different axes for four problems. In the case of bone010, Block Jacobi initially reduces the residual norm, but eventually

Table 3.5: Comparison of Distributed Southwell (DS) with Parallel Southwell (PS) and Block Jacobi (BJ) for reducing the residual to $\|r\|_2 = 0.1$. Linear interpolation on $\log_{10}(\|r\|_2)$ was used to extract this data. The † symbol indicates that a method could not achieve $\|r\|_2 \leq 0.1$ in 50 parallel steps. The wall-clock time was determined by taking the minimum of 50 samples, i.e., showing each method performing at its best. "Communication cost" is defined as the total number of messages sent by all processes divided by the number of processes. "Active processes" is defined as the average fraction of processes carrying out block relaxations of local subdomains at each parallel step.

	Wall-clock time			Communication cost			Parallel steps			Relaxations/n			Active processes		
Matrix	BJ	PS	DS	BJ	PS	DS	BJ	PS	DS	BJ	PS	DS	BJ	PS	DS
Flan_1565	†	0.547	**0.234**	†	336.185	**114.797**	†	46.073	**35.000**	†	**2.249**	2.330	†	**0.049**	0.066
audikw_1	†	1.100	**0.434**	†	483.379	**157.566**	†	44.907	**33.699**	†	**1.613**	1.737	†	**0.036**	0.049
Serena	†	0.731	**0.301**	†	394.546	**125.139**	†	44.193	**31.764**	†	**1.818**	1.839	†	**0.041**	0.057
Geo_1438	**0.068**	0.577	0.224	**53.835**	352.331	113.986	**3.805**	44.381	30.896	3.805	**1.872**	1.916	1.000	**0.042**	0.061
Hook_1498	**0.064**	0.523	0.234	**41.335**	308.938	103.964	**3.040**	37.368	29.495	3.040	**1.809**	1.939	1.000	**0.048**	0.064
bone010	†	0.700	**0.266**	†	383.214	**123.943**	†	41.750	**31.119**	†	**1.956**	2.000	†	**0.047**	0.064
ldoor	†	0.106	**0.055**	†	81.043	**32.788**	†	18.467	**15.515**	†	**1.889**	2.012	†	**0.101**	0.126
boneS10	†	0.363	**0.180**	†	175.872	**65.295**	†	27.220	**22.737**	†	**2.138**	2.257	†	**0.078**	0.099
Emilia_923	†	†	**0.309**	†	†	**134.476**	†	†	**38.669**	†	†	**2.085**	†	†	**0.054**
inline_1	†	0.673	**0.263**	†	322.954	**104.816**	†	34.164	**26.351**	†	**1.804**	2.045	†	**0.052**	0.077
Fault_639	†	†	**0.315**	†	†	**125.997**	†	†	**37.617**	†	†	**1.773**	†	†	**0.045**
StocF-1465	†	0.607	**0.227**	†	332.244	**110.737**	†	41.615	**28.841**	†	**1.661**	1.731	†	**0.039**	0.059
msdoor	†	0.128	**0.066**	†	88.255	**36.339**	†	18.708	**14.662**	†	**1.618**	1.776	†	**0.086**	0.121
af_5_k101	**0.021**	0.082	0.040	**16.148**	68.694	27.598	**2.624**	13.885	12.210	2.624	**1.733**	1.788	1.000	**0.123**	0.146

diverges. This can also be seen for Geo_1438 and Hook_1498, which are two cases where Block Jacobi can reach the target residual norm. This divergence underscores the unreliability of Block Jacobi, especially when a large number of processes is used. Matrix af_5_k101 is the only case in which Block Jacobi never diverged.

Table 3.5 also shows the superiority of Distributed Southwell over Parallel Southwell. Distributed Southwell is approximately twice as fast, requires close to a third of the communication, and converges in fewer parallel steps. Parallel Southwell requires fewer relaxations, but needs to communicate more per relaxation.

The fact that Parallel Southwell requires fewer relaxations but requires almost three times the communication shows how costly the explicit residual updates of Parallel Southwell are. This cost is shown in Table 3.6, where the explicit residual updates by Parallel Southwell dominate the overall communication cost.

We also observe that in Distributed Southwell, more processes are active compared to Parallel Southwell. This is a result of using inexact residual norms. We note that if adjacent

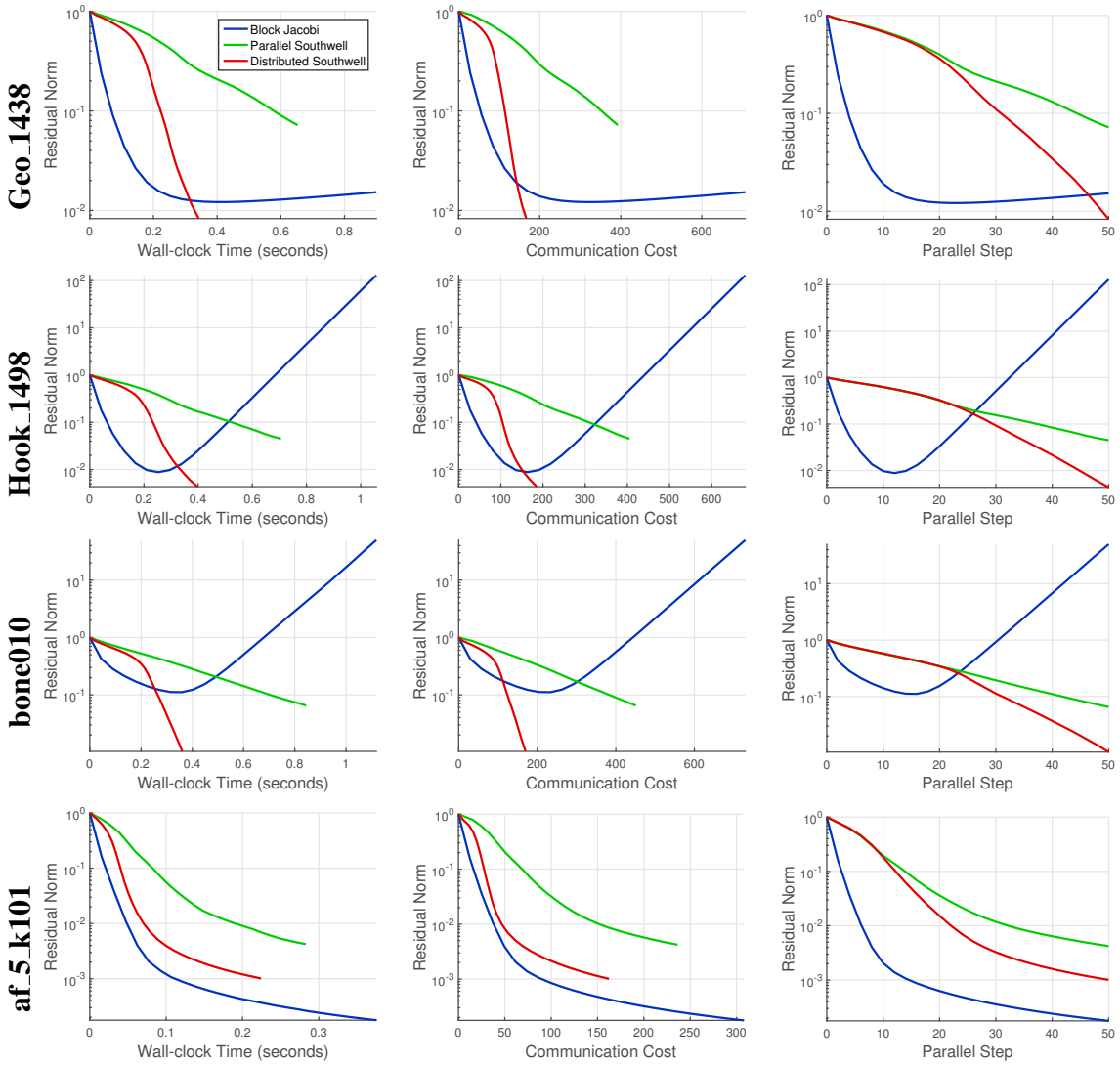


Figure 3.9: Comparison of Block Jacobi and Distributed and Parallel Southwell for four test problems that show different behavior of Block Jacobi. For Geo_1438 and Hook_1498, Block Jacobi is able to reach the target residual norm of 0.1, and is the best method for these problems for this level of accuracy. However, Block Jacobi diverges for these problems if more steps are taken. For bone010, Block Jacobi is not able to reach the target residual norm of 0.1. Distributed Southwell is the best method for this problem for this level of accuracy. Of the 14 test problems shown in Table 3.4, af_5_k101 is the only case in which Block Jacobi never diverged.

subdomains relax at the same time (rather than an independent set of subdomains), then convergence is at risk.

Since multigrid smoothing and preconditioning only requires a small number of sweeps, it is useful to look at the costs per parallel step. This is shown in Table 3.7, where

Distributed Southwell is faster than the other methods.

Table 3.6: Communication cost breakdown for Parallel Southwell (PS) and Distributed Southwell (DS), where "Solve comm" denotes the communication cost of sending updates to neighbors after a local subdomain is solved, and "Res comm" denotes the communication cost of explicit residual updates.

	Solve comm		Res comm	
Matrix	PS	DS	PS	DS
Flan_1565	**27.945**	28.961	308.240	**85.836**
audikw_1	**28.630**	30.634	454.749	**126.932**
Serena	**27.399**	27.748	367.147	**97.391**
Geo_1438	**26.485**	27.076	325.846	**86.910**
Hook_1498	**24.076**	25.744	284.861	**78.220**
bone010	**27.965**	28.617	355.249	**95.326**
ldoor	**11.980**	12.745	69.063	**20.043**
boneS10	**18.462**	19.433	157.410	**45.862**
Emilia_923	†	**32.101**	†	**102.375**
inline_1	**24.352**	27.311	298.601	**77.505**
Fault_639	†	**27.785**	†	**98.213**
StocF-1465	**24.188**	25.208	308.056	**85.529**
msdoor	**11.560**	12.669	76.695	**23.670**
af_5_k101	**10.603**	10.947	58.091	**16.651**

Strong Scaling

We first look at a target residual norm of $\|r\|_2 = 0.1$ and varying the number of MPI processes. Six examples are shown in Figure 3.10, where wall-clock time is shown as a function of the number of MPI processes. In most cases, and for all methods, the wall-clock time initially decreases as we increase the number of MPI processes, and then starts to increase. This is due to the local subdomain solves, where a single Gauss-Seidel sweep is used. This operation has the complexity of a sparse matrix-vector product, and as the number of MPI processes increases (i.e., local subdomain sizes decrease), the time spent on communication increasingly outweighs the time spent on computation. It can be observed that the poor scalability is worse for the smaller problems.

We can see that Distributed Southwell is always faster than Parallel Southwell, except

Table 3.7: Per parallel step results of Distributed Southwell (DS) compared with Parallel Southwell (PS) and Block Jacobi (BJ) for taking 50 parallel steps using 8192 MPI processes. Mean wall-clock time and communication cost over the 50 parallel steps are shown.

	Wall-clock time			Communication cost		
Matrix	BJ	PS	DS	BJ	PS	DS
Flan_1565	0.017	0.012	**0.006**	12.537	7.307	**3.056**
audikw_1	0.031	0.025	**0.013**	18.092	10.634	**5.204**
Serena	0.023	0.017	**0.009**	15.234	8.899	**3.607**
Geo_1438	0.018	0.013	**0.007**	14.149	7.854	**3.337**
Hook_1498	0.021	0.014	**0.008**	13.599	8.102	**3.705**
bone010	0.022	0.017	**0.007**	14.596	9.024	**3.406**
ldoor	0.008	0.006	**0.004**	6.319	4.243	**2.688**
boneS10	0.018	0.014	**0.006**	9.226	6.026	**2.562**
Emilia_923	0.023	0.014	**0.008**	15.370	7.574	**3.473**
inline_1	0.025	0.019	**0.010**	13.877	9.191	**5.075**
Fault_639	0.021	0.015	**0.008**	15.735	7.317	**3.411**
StocF-1465	0.021	0.014	**0.009**	14.616	7.798	**4.455**
msdoor	0.009	0.007	**0.005**	7.101	4.467	**2.955**
af_5_k101	0.008	0.006	**0.004**	6.155	4.728	**3.248**

for Flan_1565 on 64 processes, where the two wall-clock times are quite close. Additionally, when Block Jacobi achieves $\|r\|_2 = 0.1$, it is faster than Parallel and Distributed Southwell, e.g., for Hook_1498. However, it is often the case that Block Jacobi cannot achieve $\|r\|_2 = 0.1$, even for a small number of processes. For example, for Flan_1565, ldoor, and StocF-1465, Block Jacobi cannot achieve $\|r\|_2 = 0.1$ for more than 128 processes. This demonstrates that Block Jacobi can be an unreliable method even for a small number of processes.

We now look at the residual norm after 50 parallel steps of each method as we vary the number of MPI processes from 32 to 8192 (from 1 to 256 nodes). Six examples are shown in Figure 3.11. It is clear that for larger numbers of MPI processes, the convergence of Block Jacobi severely degrades or Block Jacobi may even diverge after 50 parallel steps. The degradation is much more mild for Parallel Southwell and Distributed Southwell. The fact that the residual norm of Distributed Southwell does not significantly degrade is why it

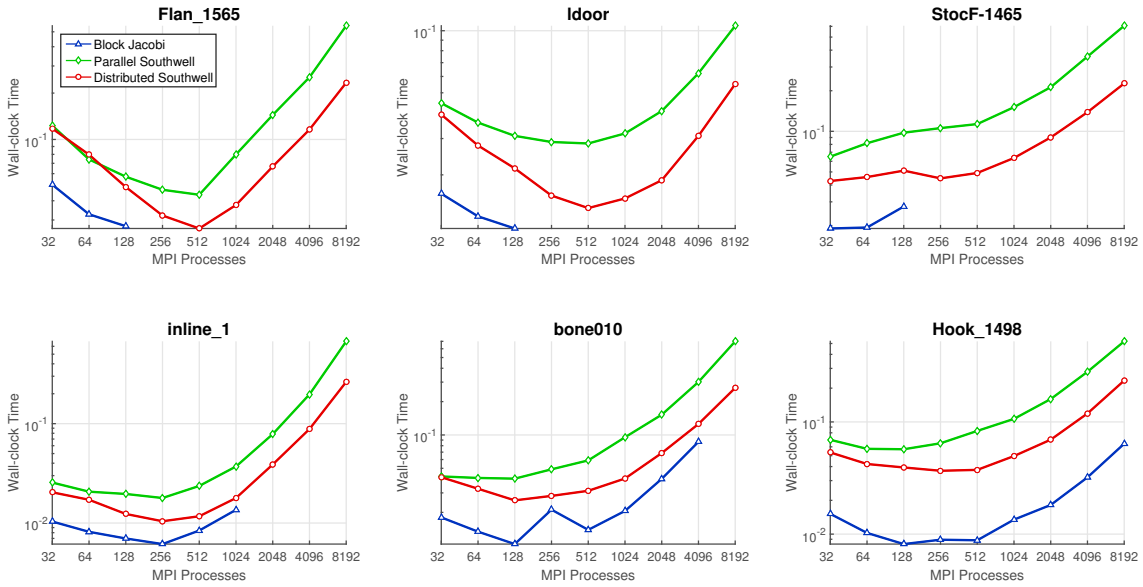


Figure 3.10: Wall-clock time as a function of the number of MPI processes for reducing $\|r\|_2$ to 0.1. Missing data for Block Jacobi indicates that Block Jacobi could not achieve $\|r\|_2 \leq .1$ in 50 parallel steps, usually due to divergence of the Block Jacobi method. The Block Jacobi method is fastest when it does converge.

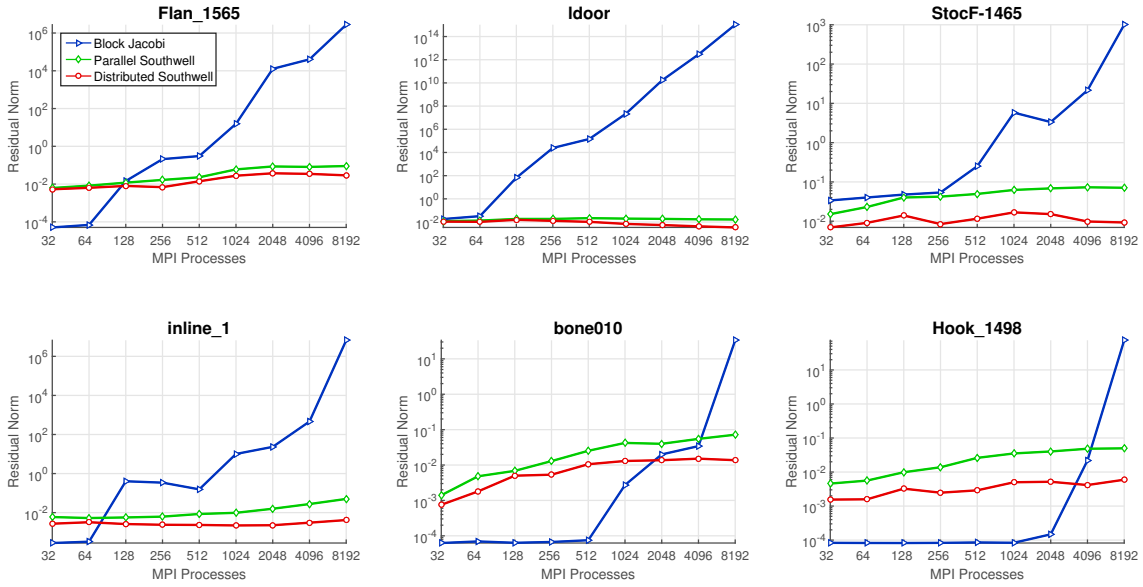


Figure 3.11: Residual norm after 50 parallel steps as a function of the number of MPI processes for different test problems. When the residual norm is above 1, this indicates that the method has diverged after 50 parallel steps. For larger numbers of processes, Block Jacobi is more likely to diverge after many steps.

can be considered a competitor to Block Jacobi for massively parallel multigrid smoothing and preconditioning.

3.5 The Stochastic Parallel Southwell Method

In the Stochastic Parallel Southwell Method, the magnitude of a residual determines the likelihood that that row will be relaxed. If a residual is large compared with the neighboring residual, that row will have a high probably of being relaxed, and vice versa. The probability that a row is relaxed is determined by the *score* z_i of the residual. The score of a row i is the number of neighboring residuals that are larger than $|r_i|$. We then use this score to construct an update probability, e.g., $\rho_i = \exp(-\pi z_i)$ or $\rho_i = 1/(\pi z_i + 1)$, where π is a parameter that controls how the probability changes as the score increases. For example, we may want all processes that do not hold the maximum residual in there neighborhood to have a very low probability, so we could set $\pi = 2$. Alternatively, we could set $\pi = .5$ if we want more processes to have a high probability. Similar to Parallel Southwell, we can write Stochastic Parallel Southwell as

$$\hat{D}_{ii}^{(t)} = \begin{cases} \omega/A_{ii}, & \text{if } \xi \leq \rho_i, \\ 0, & \text{otherwise,} \end{cases} \tag{3.7}$$

where ξ is a uniform random number in the range $[0, 1]$. The scalar ω is an optional weight, similar to weighted Jacobi. This weight can be used to guarantee convergence for the SPD case, since by Theorem 2 in [15], SPS will converge when weighted Jacobi converges. Therefore, by taking an ω value such that weighted Jacobi converges, e.g., the optimal value $\omega = 2/(\lambda_{\max}(D^{-1}A) + \lambda_{\min}(D^{-1}A))$ where D is the Jacobi smoothing matrix, SPS will converge.

Figure 3.12 compares several smoothers with SPS for the 5-point centered difference discretization of the Poisson equation. We used $\exp(-z_i)$ for the SPS update probability.

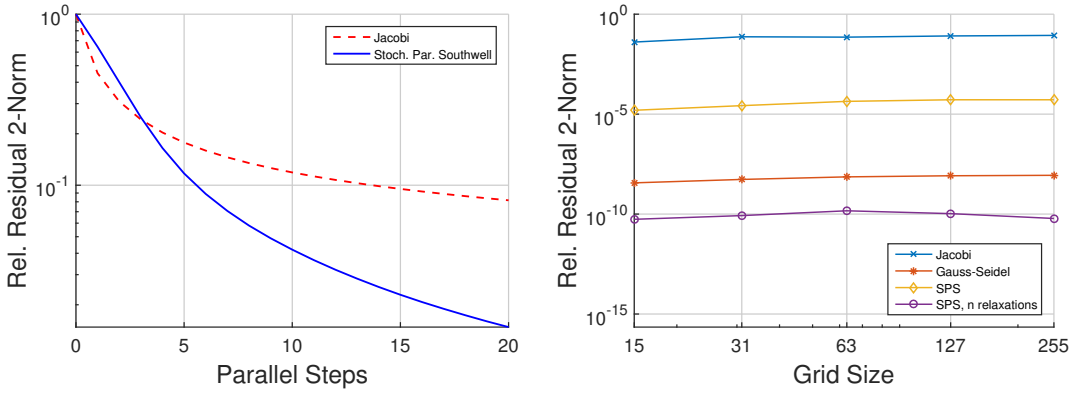


Figure 3.12: Convergence comparison of Stochastic Parallel Southwell (SPS) with other smoothers. The test problem is the 5-point centered difference discretization of the Poisson equation. Sub-figure (a) shows the relative residual 2-norm versus number of parallel steps (iterations) of Jacobi and SPS. Sub-figure (b) shows the convergence of classical multiplicative multigrid for several smoothers as the grid size increases.

Sub-figure (a) shows the relative residual 2-norm versus the number of parallel steps (iterations) for SPS and Jacobi. A 255×255 grid is used for this experiment. For SPS, less than half the rows update per parallel step, showing that even with less work per parallel step, SPS still converges faster than Jacobi. This indicates that not only can communication be avoided, but the convergence rate can also be accelerated. Sub-figure (b) shows SPS, Jacobi, Gauss-Seidel as smoothers for classical multiplicative multigrid using one pre- and and one post-smoothing sweep. For SPS, we show the convergence when only one parallel step is used, and for when n relaxations are used (this takes 2-3 parallel steps on average). The n relaxations is included to show convergence for when SPS does the same amount of work as Jacobi and Gauss-Seidel. We can see that grid-size independent convergence is achieved, even when only one parallel step is taken, indicating that not all rows need to be relaxed for a smoother to be effective. Additionally, we can see that when SPS with n relaxations is the fastest method.

Algorithm 6 shows asynchronous SPS for the general case in which a process owns a sub-domain. The p superscript denotes a sub-domain vector, where x^p and r^p are sub-vectors of x and r, respectively. Let A_p be the diagonal block of A denoting connections

in the underlying discretization stencil between points in the sub-domain belonging to p. Let A_{offp_i} be the off-diagonal block of A denoting connections between points in the sub-domain belonging to p to points belonging to i.

If a process determines that it must update, it solves the equations $A_p e^p = r^p$ (lines 5). This could be an exact solve or single smoothing sweep. For our implementation, we use a single weighted Jacobi sweep. The residual is then updated (line 7), and the data is sent to intra grid neighbors (lines 8-14). Lastly, the receive phase occurs where if there is new data, r_p and Φ_p are updated (lines 16-24). Note that we can also add in inter grid communication as well where e^p is sent to inter grid neighbors. This is exactly what we do, allowing asynchronous SPS to be used as a smoother for grid zero.

Algorithm 6: Block Asynchronous Stochastic Parallel Southwell for Distributed Memory

while *not converged on process* p **do**
  z = Score($\|r^p\|, \Phi_p$) ▷ compute the score for p
  ξ = RandNum$(0, 1)$ ▷ generate random number in the range $[0, 1]$
  if $\xi \leq$ *UpdateProbability*(z) **then** ▷ compute update probability and update if necessary
    e^p = Solve(A_p, r^p)
    $x^p = x^p + e^p$
    $r^p = r^p - A_d e^p$
    for *each intra grid neighbor* i *of process* p **do**
      `[inflight_flag, j] = CheckInflight(send_requests[i])`
      if *`inflight_flag`* **then**
        `send_buff[i][j][:] =` $[e^p_{\text{offd}_i}, \|r^p\|]$ ▷ copy boundary updates and new residual norm into send buffer
        `MPI_Isend(send_buff[i][j],..., send_requests[i])` ▷ non-blocking send
      end
    end
  end
  for *each intra grid neighbor* i *of process* p **do**
    `MPI_Test(recv_requests[i], &flag,...)` ▷ check completion of receive
    if *`flag`* **then**
      $e^p = 0$
      $[e^p_{\text{offd}_i}, \Phi_p$`[i]] = recv_buff[i][:]` ▷ copy received boundary updates into e^p and residual norm into Φ_p
      `MPI_Irecv(recv_buff[i],..., recv_requests[i])` ▷ non-blocking receive
      $r^p = r^p - A_{\text{offd}_i} e^p$ ▷ update r^p
    end
  end
end

3.6 Conclusion

In Section 3.3, we presented the Parallel Southwell method. The method is based on the sequential Southwell method, where the row with the maximum absolute value residual is relaxed on each parallel step. We parallelized this method by, per parallel step, relaxing all rows that calculate themselves as maximum in their neighborhood of mesh points in the underlying geometry. This eliminates the need to calculating a global maximum, i.e., global synchronization, and allowed for multiple rows to be updated in parallel.

This idea of greedily selecting rows can actually speed up convergence. We showed this experimentally using matrices from the University of Florida sparse matrix collection. The first result from these experiments is that Parallel Southwell consistently requires fewer relaxations to converge than Jacobi, Gauss-Seidel, and multicolor Gauss-Seidel. A more important result is that Parallel Southwell can converge quicker per parallel step than both Jacobi and multicolor Gauss-Seidel, even though it does less work. Additionally, parallel Southwell can also have a higher convergence rate.

By definition, Parallel Southwell does not require global synchronization, which makes it a good candidate for an asynchronous implementation. We showed this using a shared memory implementation, and ran experiments on Intel Xeon CPUs, and an Intel Xeon Phi. We showed that although our asynchronous implementation required more relaxations to converge, the overall wall clock time was lower than its synchronous counterpart. Additionally, we showed that a minor load imbalance can greatly slow a synchronous algorithm, while an asynchronous method handles the imbalance quite well.

The Parallel Southwell method demonstrates an important concept in high performance computing, namely that doing less work can result in a faster algorithm. This has applications to distributed computing, where reducing communication and execution wall time may become equally important on future exascale machines. Similarly, for many-core shared memory architectures, as in the Intel Xeon Phi, it is desired to reduce expen-

sive shared memory write operations. We can see that parallel Southwell competes quite well with elementary stationary iterative methods, which are commonly used as multigrid smoothers and preconditioners. Future work would involve experimenting with Parallel Southwell as a multigrid smoother or preconditioner.

In Section 3.4, we presented the Distributed Southwell method. Parallel Southwell is a natural way to parallelize the Sequential Southwell method. However, in a distributed setting, Parallel Southwell has a high communication cost that stems from the requirement for neighboring processes to exchange the residual norms of their local subproblems. In Distributed Southwell, the main idea is that these residual norms do not need to be known exactly. Instead, estimates of the residual norms of neighbors can be computed locally without communication. However, deadlock may occur if the estimates are such that no process thinks it has the largest residual norm. We presented a novel scheme to avoid deadlock by sending explicit residual norm update messages *only when necessary*. The result is that Distributed Southwell uses much less communication than Parallel Southwell.

Distributed Southwell is also a potential improvement over Block Jacobi when a large number of processes is used. This is an important issue when considering future exascale machines, where the number of cores will be massive. In this case, block sizes will be small, possibly leading to slow or no convergence for Block Jacobi.

In Section 3.5, we introduced the Stochastic Parallel Southwell method. In this method, each process uses its residual norm estimates to construct a probability. This is the probability that the process will relax its rows. Unlike Distributed Southwell, Stochastic Parallel Southwell method does not require explicit residual updates since even if a process computes a low probability, the rows belonging to that process will eventually be relaxed. We showed that Stochastic Parallel Southwell can converge with a similar rate to Parallel Southwell, and multigrid exhibits grid-size independent convergence when using Stochastic Parallel Southwell as a smoother.

CHAPTER 4
ASYNCHRONOUS MULTIGRID METHODS

Multigrid methods are a way of accelerating a basic iterative method (referred to as a smoother in the context of multigrid). The simplest multigrid method is the two-grid method, where a smoother is combined with a coarse grid correction. The smoother and the coarse grid correction synergize excellently; the smoother eliminates components of the error corresponding to rough eigenmodes of A, and the coarse grid correction eliminates smooth components. For large problems, additional coarser grids must be used since the first coarse grid might be too large to make an exact solve feasible, where a smoother is used on all grids except the coarsest grid. Multigrid methods are widely used due to their scalability. They are optimal in the sense that their convergence rate is independent of the problem size, and modern implementations are efficient on parallel computers since the core kernel is a matrix-vector product [66]. However, per iteration, classical multigrid methods have many points where communication neighbors must synchronize, which can be costly especially on the coarser grids.

In this chapter, we introduce the first asynchronous multigrid methods as standalone solvers, not as preconditioners. These methods are asynchronous versions of additive multigrid methods. In additive multigrid methods, smoothing and exact solves on different grids are computed concurrently and added to the current approximation to the solution on the finest grid. Additive multigrid methods are more parallel than classical multiplicative multigrid methods, which allow us to execute additive multigrid methods asynchronously. We introduce models and implementations of asynchronous multigrid and provide experimental results that show that asynchronous mutligrid can exhibit grid-size independent convergence and can converge faster than classical multigrid. We also introduce a new additive multigrid method, the AFACj method, that can be executed asynchronously.

4.1 Background

4.1.1 Classical Multiplicative Multigrid Methods

To define the classical multigrid V$(1,1)$-cycle, we first start with some definitions. For grid numbers $k = 0, \ldots, \ell - 1$, where ℓ is the coarsest grid, we define:

- the two-level interpolant P_{k+1}^k that transfers a vector from grid $k+1$ to k. For simplicity, we will choose $(P_{k+1}^k)^T$ as the restriction matrix that transfers a vector from grid k to $k+1$.
- the coarse grid operator $A_{k+1} = (P_{k+1}^k)^T A_k P_{k+1}^k$ at grid $k+1$.
- the smoothing iteration matrix $G_k = I - M_k^{-1} A_k$, where the smoothing matrix M_k is typically easy to invert.

We can now define the classical V(1,1)-cycle, as shown in Algorithm 7.

Algorithm 7: Multiplicative V(1,1)-Multigrid

```
Initialize r_0 = b
for t = 1, 2, ..., t_max do
    Sequential for k = 0, ..., ℓ − 1 do
        e_k = M_k^{-1} r_k                          ▷ pre-smoothing
        r_{k+1} = (P_{k+1}^k)^T (r_k − A_k e_k)     ▷ restriction
    end
    e_ℓ = A_ℓ^{-1} r_ℓ                              ▷ exact solve on coarsest grid
    Sequential for k = ℓ − 1 ..., 0 do
        e_k = e_k + P_{k+1}^k e_{k+1}               ▷ coarse grid correction
        e_k = e_k + M_k^{-1} (r_k − A_k e_k)        ▷ post-smoothing
    end
    x = x + e_0                                     ▷ correct solution on finest grid
    r_0 = b − Ax                                    ▷ compute new residual
end
```

4.1.2 Additive Multigrid Methods

We define the multi-level interpolant P_k^0 for $k = 0, 1, \ldots \ell - 1$ that transfers a vector from grid k to the finest grid. Additionally, we define $(P_k^0)^T$ as the corresponding restriction

matrix. For $k = 0$, $P_0^0 = I$. We consider $P_k^0 = P_1^0 P_2^1 \cdots P_k^{k-1}$ and is not explicitly formed, i.e., each P_j^{j-1} for $j = 1, \ldots, k$ is applied to the vector that is being interpolated.

In additive multigrid methods, smoothing on each grid can be done concurrently. This allows the *updates* from each grid to be added together on the fine grid. We can write an additive multigrid method as

$$x^{(t+1)} = x^{(t)} + \sum_{k=0}^{\ell-1} e_k^{(t)}. \tag{4.1}$$

where $e_k^{(t)}$ is the update for grid k. The classical additive multigrid method is known as the BPX method [24]. One V-cycle of BPX can be written as

$$x = x + \sum_{k=0}^{\ell-1} P_k^0 \Lambda_k (P_k^0)^T r, \tag{4.2}$$

where Λ_k is the inverse of the smoothing matrix for $k = 0, \ldots, \ell - 1$ and $\Lambda_\ell = A_\ell^{-1}$.

BPX is typically used as a preconditioner because adding the updates "over-corrects" x, resulting in a divergent solver. This over-correction occurs because the right-hand sides on the coarse grids are approximately equal, resulting in redundant updates. In [67], BPX is modified using multicoloring to create a convergent solver. The authors suggest that this new solver could be asynchronous, but do not precisely define what asynchronous multigrid means in this context.

Additive Variants of Multiplicative Multigrid (Multadd)

Multadd [68, 69] is derived by re-writing the multiplicative method as

$$x = x + \sum_{k=0}^{\ell-1} \overline{P}_k^0 \Lambda_k (\overline{P}_k^0)^T r. \tag{4.3}$$

This method looks like BPX, but with the multi-level smoothed interpolants $\overline{P}_k^0 = \overline{P}_1^0 \cdots \overline{P}_k^{k-1}$, where the two-level smoothed interpolants are $\overline{P}_{k+1}^k = G_k P_{k+1}^k$ for $k = 0, \ldots, \ell -$

1.

If Λ_k is chosen to be the *symmetrized smoothing matrix* $\overline{M}_k^{-1} = M_k^{-T}(M_k + M_k^T - A_k)M_k^{-1}$, then Multadd is mathematically equivalent to a symmetric multiplicative V(1,1)-cycle (where G_k^T is chosen as the post-smoothing iteration matrix). If it is not a requirement for Multadd to be mathematically equivalent to the classical multiplicative multigrid method, an approximation $\overline{\Lambda}_k$ to Λ_k can also be used as the symmetrized smoother, e.g., $\overline{\Lambda}_k = D_k$. We consider this case for a hybrid smoother (see Section 4.3.2).

Expressing the multiplicative method in this additive form may seem too good to be true since now each grid can be processed concurrently without sacrificing multiplicative convergence properties. However, the additive form introduces redundant computation, since grid $k + 1$ must carry out the same set of prolongation and restriction steps as grid k. This suggests that Multadd would likely be slower than the multiplicative method, but if we were to make the method asynchronous, the increased computational cost might be outweighed by the gain in speed from not having to synchronize.

The Asynchronous Fast Adaptive Composite Grid Method (AFACx)

The asynchronous fast adaptive composite grid method (AFACx) [70, 71, 72, 73, 74] is an additive multigrid method for solving PDEs on composite grids. These composite grids typically arise from adaptive mesh refinement processes. A composite grid can be decomposed into a hierarchy of grids with different resolutions and different domain sizes. We can use AFACx as a multigrid method by thinking of the multigrid hierarchy as a hierarchy from a fully refined composite grid. There are three key steps in AFACx when computing the update for grid k:

1. The quantity e_{k+1} is computed by smoothing on the equations $A_{k+1}e_{k+1} = r_{k+1}$, where an initial guess of zero is used and r_{k+1} is the fine grid residual restricted to grid $k+1$.

2. The quantity e_k is then computed by smoothing on the equations $A_k e_k = r_k$ using an initial guess of $P_{k+1}^k e_{k+1}$.

3. x is corrected: $x = x + P_k^0(e_k - P_{k+1}^k e_{k+1})$.

The subtraction of $P_k^0 P_{k+1}^k e_{k+1}$ from x in the third step is what prevents an over-correction of x. This is because grids k and $k+1$ may produce approximately the same updates, so subtracting $P_{k+1}^k e_{k+1}$ from $P_k^k e_k$ serves to remove the portion of $P_k^k e_k$ that is close in value to the update from grid $k+1$.

Algorithm 8 shows AFACx, where V(1,1)-cycles are used in the inner loop. Here, the V(1,1) notation refers to using one smoothing sweep to compute e_{k+1} and one smoothing sweep to compute e_k, (not to be confused with a V(1,1)-cycle of classical multiplicative multigrid). A V(s_1, s_2)-cycle can also be defined, where s_1 smoothing sweeps are used to compute e_{k+1} and s_2 smoothing sweeps are used to compute e_k. The redundant computation of $P_k^0 e_k$ and $P_{k+1}^0 e_{k+1}$ can be avoided by modifying how e_k is computed: we use an initial guess of zero and a modified right-hand side of $r_k - A_k P_{k+1}^k e_{k+1}$, as shown in lines 8 and 9 of Algorithm 8.

Algorithm 8: V(1,1)-AFACx

```
1  Initialize r = b
2  for t = 1, 2, ..., t_max do
3      for k = 0, ..., ℓ − 1 do
4          r_k = (P_k^0)^T r                          ▷ restriction
5          if k == ℓ − 1 then
6              e_k = A_k^{-1} r_k                     ▷ exact solve on coarsest grid
7          else
8              e_{k+1} = M_{k+1}^{-1} (P_{k+1}^k)^T r_k
9              e_k = M_k^{-1} (r_k − A_k P_{k+1}^k e_{k+1})   ▷ smooth
10         end
11         x = x + P_k^0 e_k                          ▷ correct solution on finest grid
12     end
13     r = b − Ax                                     ▷ compute new residual
14 end
```

Other Additive Multigrid Methods

The first additive multigrid method was proposed in [75]. To address the over-correction issue, the updates are done sequentially, where each update is made to be orthogonal to the

new residual (the residual after the approximation is corrected) on the next finer grid. This sequential aspect, however, is not ideal for devising an asynchronous method.

Residual splitting methods [76] split the residual into a rough and smooth part using an appropriate filter. The smoother then uses the rough part of the residual, and the coarse grid correction uses the smooth part. These two updates can then be added together without over-correcting. In [77], all grids carry out this process simultaneously. These methods converge slower than multiplicative methods, and the added cost of filtering increases the solve time.

4.2 Models of Asynchronous Multigrid Methods

In this section, we present new models of asynchronous additive multigrid methods [78]. We emphasize that our definitions of asynchronous multigrid are different than that of asynchronous task-based processing of grids, as in [79]. The purpose of this section is not to analyze these models, but to define what asynchronous multigrid actually is, which has not been done before. In other words, the models presented in this section give us a clear picture of what is meant by asynchronous multigrid: at some time instant t, some set of grids update without any of the grids synchronizing, i.e., each grid has no information about the progress made by other grids. In the case that some update is delayed, this means that multiple updates from other grids have been performed before the delayed grid has corrected once.

There has been previous work on making multigrid methods asynchronous. In [80], the authors asynchronously execute a saw-tooth cycle (classical multiplicative multigrid with no pre-smoothing) by using un-smoothed aggregation. Un-smoothed aggregation allows the prolongation and restriction to be carried out without communication, making the up-cycle completely asynchronous. However, synchronization is still used on the finest grid, and the method is limited to only using interpolation matrices that require no communication.

The first model is the *semi-asynchronous* model (semi-async),

$$x^{(t+1)} = x^{(t)} + \sum_{k \in \Psi(t)} B_k(x^{(z_k(t))}), \tag{4.4}$$

and the second is the *fully asynchronous* model (full-async),

$$x^{(t+1)} = x^{(t)} + \sum_{k \in \Psi(t)} B_k(x_1^{(z_{k1}(t))}, \ldots, x_n^{(z_{kn}(t))}). \tag{4.5}$$

We refer to these two models as the *solution-based* versions of semi-async and full-async since $x^{(t)}$ is written to and read from by all grids. If $k > 0$, B_k is a function that outputs the update for grid k. If $k = 0$, the output of B_k corresponds to one smoothing sweep applied to the error equations. For example, for grid k in the semi-async model of Multadd, $B_k(x) = \overline{P}_k^0 \Lambda_k (\overline{P}_k^0)^T (b - Ax^{(z_k(t))})$. In these models, the computation of $B_k(x)$ is carried out synchronously by the threads belonging to grid k. However, an asynchronous smoother could also be used, i.e., we could apply Λ_k asynchronously.

There are similarities between Equations 4.4 and 4.5, and Equation 2.5. First, the set $\Psi(t)$ is now the set of grids correcting the solution at time instant t. Second, we now have the mappings $z_k(t)$ and $z_{ki}(t)$ for $i = 1, \ldots, n$. These two mappings are what make semi-async and full-async different from each other. For full-async, x can be corrected by one grid while a different grid is simultaneously reading x from memory. The result is that the copy of x read from memory contains elements from different time instants. For semi-async, all components of x read from memory come from the same time instant.

Alternatively, with the observation that any fixed-point iteration can be expressed as

$$\begin{aligned} x &= x + M^{-1} r \\ r &= r - AM^{-1} r \end{aligned} \tag{4.6}$$

we can also express the semi-async and full-async models in terms of the residual:

$$r^{(t+1)} = r^{(t)} - A \sum_{k \in \Psi(t)} C_k(r^{(z_k(t))}), \tag{4.7}$$

and,

$$r^{(t+1)} = r^{(t)} - A \sum_{k \in \Psi(t)} C_k(r_1^{(z_{k1}(t))}, \ldots, r_n^{(z_{kn}(t))}), \tag{4.8}$$

where C_k is defined similarly to B_k. We refer to these two models as the *residual-based* versions of semi-async and full-async, respectively. In the case of semi-async, there is no difference between the residual-based and solution-based versions, given that, for all k and t, $z_k(t)$ is the same in both cases. However, for full-async, the solution-based and residual-based versions are different since the vectors $(r_1^{(z_{k1}(t))}, \ldots, r_n^{(z_{kn}(t))})^T$ and $b - A(x_1^{(z_{k1}(t))}, \ldots, x_n^{(z_{kn}(t))})^T$ can be different, even when $z_{k1}(t), \ldots, z_{kn}(t)$ are the same for all k and t.

To demonstrate the difference in convergence among our four models (solution-based and residual-based versions of semi-async and full-async), we simulated asynchronous multigrid by implementing Equations 4.4, 4.5, and 4.8 as solvers to be executed sequentially. In the simulation, grid k has an *update probability* p_k, i.e., grid k has the probability p_k of being in $\Psi(t)$ at time instant t. In our experiments, p_k is determined in advance (before we start solving $Ax = b$) by sampling from a uniform random integer distribution in the range $[\alpha, 1]$, where α is the *minimum update probability* and $1 > \alpha > 0$. As α decreases, the grids will become more "out of sync", i.e., the values of p_k will have a higher variation resulting in some grids updating more often than others.

If $k \in \Psi(t)$, the value of $z_k(t)$ ($z_{ki}(t)$ in the case of full-async) is chosen randomly by sampling from a uniform random integer distribution in the range $(\min(z_k(\tau_k), t - \delta), t]$. The time instant τ_k denotes the last time instant that grid k read from. The *maximum read delay* δ is defined as the maximum value of $t - z_k(t)$ and denotes the minimum past time instant that grid k can read from. In other words, we are assuming two things: 1) a grid

cannot read older information than what has already been read ($z_k(\tau_k)$ term), and 2) even if the updates for a grid are computed very slowly compared to other grids, there is still some bound on how old the information can be that is read from memory ($t - \delta$ term).

A minimum of 20 updates are computed for each grid, and the iteration is terminated after 20 updates have been computed for all grids. We compare this to 20 V(1,1)-cycles of synchronous multigrid. For our test framework, we used the 27pt test set (see Section 4.3.2 for matrix descriptions) with mesh sizes ranging from $40 \times 40 \times 40$ to $80 \times 80 \times 80$. Weighted Jacobi was used as a smoother with a weight of $.9$. We used the BoomerAMG package [81] to generate the interpolation and coarse grid matrices. For our BoomerAMG options, we chose HMIS coarsening with one aggressive level, and classical modified interpolation.

Figure 4.1 demonstrates the effect of α on the convergence of semi-async when $\delta = 0$. The figure shows the relative residual 2-norm versus the grid length for Multadd and AFACx. Each data point is the mean relative residual 2-norm of 20 runs. Each figure shows synchronous multigrid and simulations of semi-async with different values of α. The figures show that with small values of α, convergence is slower, but the convergence is still independent of the grid length.

Figure 4.2 demonstrates the effect of δ on the convergence of full-async with $\alpha = .1$. Each figure shows synchronous multigrid and simulations of either the solution-based or residual-based versions of full-async. These results show that with larger values of δ, convergence is slower, but the convergence is still independent of the grid length. Additionally, the residual-based versions converge faster than the solution-based versions for large values of δ.

4.3 Asynchronous Multigrid for Shared Memory

4.3.1 Algorithms for Asynchronous Implementations

This section presents asynchronous additive multigrid methods for shared memory parallel computers. The main issue to address is the computation of the residual on the fine grid.

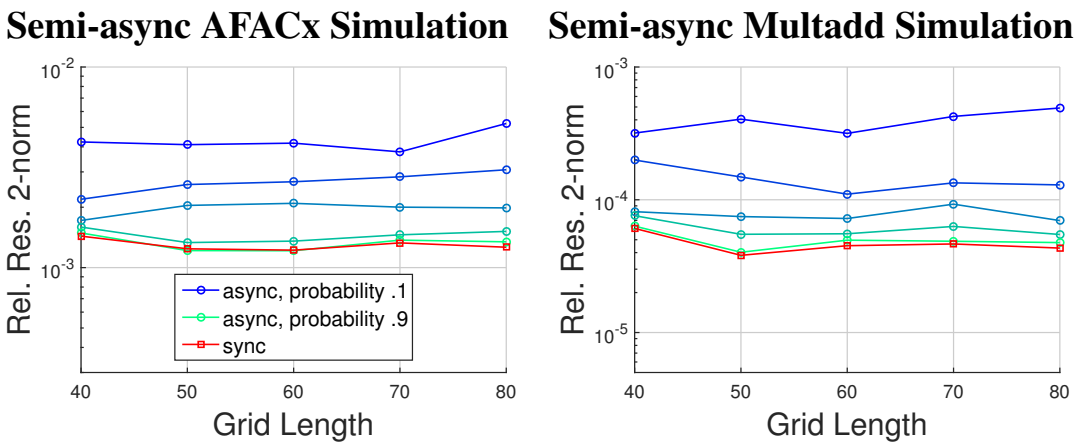


Figure 4.1: Final relative residual 2-norm after 20 V-cycles versus grid length for the semi-asynchronous multigrid model (Equation 4.4) for AFACx and Multadd. A maximum delay of zero is used. Results are shown for five minimum update probabilities, where blue-to-green corresponds to increasing minimum update probability. The 27pt test set is used (see Section 4.3.2). These results show that even with a small minimum update probability, asynchronous multigrid still exhibits grid-size independent convergence.

We first describe two implementations for synchronous two-grid Multadd, and then extend these to the asynchronous case. The implementations are mathematically the same when executed synchronously. We proceed with an example. We have five threads, t_0, t_1, t_2, t_3 and t_4. The fine grid has seven points, and the coarse grid has three points. Recall that one V(1,1)-cycle of Multadd is

$$\begin{aligned} r &= b - Ax \\ x &= x + \Lambda_0 r + \overline{P}_1^0 A_1^{-1} (\overline{P}_1^0)^T r. \end{aligned} \tag{4.9}$$

Threads t_0 and t_1 are responsible for computing $\Lambda_0 r$, and threads t_2, t_3 and t_4 are responsible for computing $\overline{P}_1^0 A_1^{-1} (\overline{P}_1^0)^T r$. We say that t_0 and t_1 are assigned to grid 0, and t_2, t_3 and t_4 are assigned to grid 1. In the general case, threads are distributed among the grids to balance the amount of "work", where the work for a grid is approximately the number of flops required to compute the update for that grid.

We present two algorithms for parallel synchronous Multadd which differ only in how r is computed:

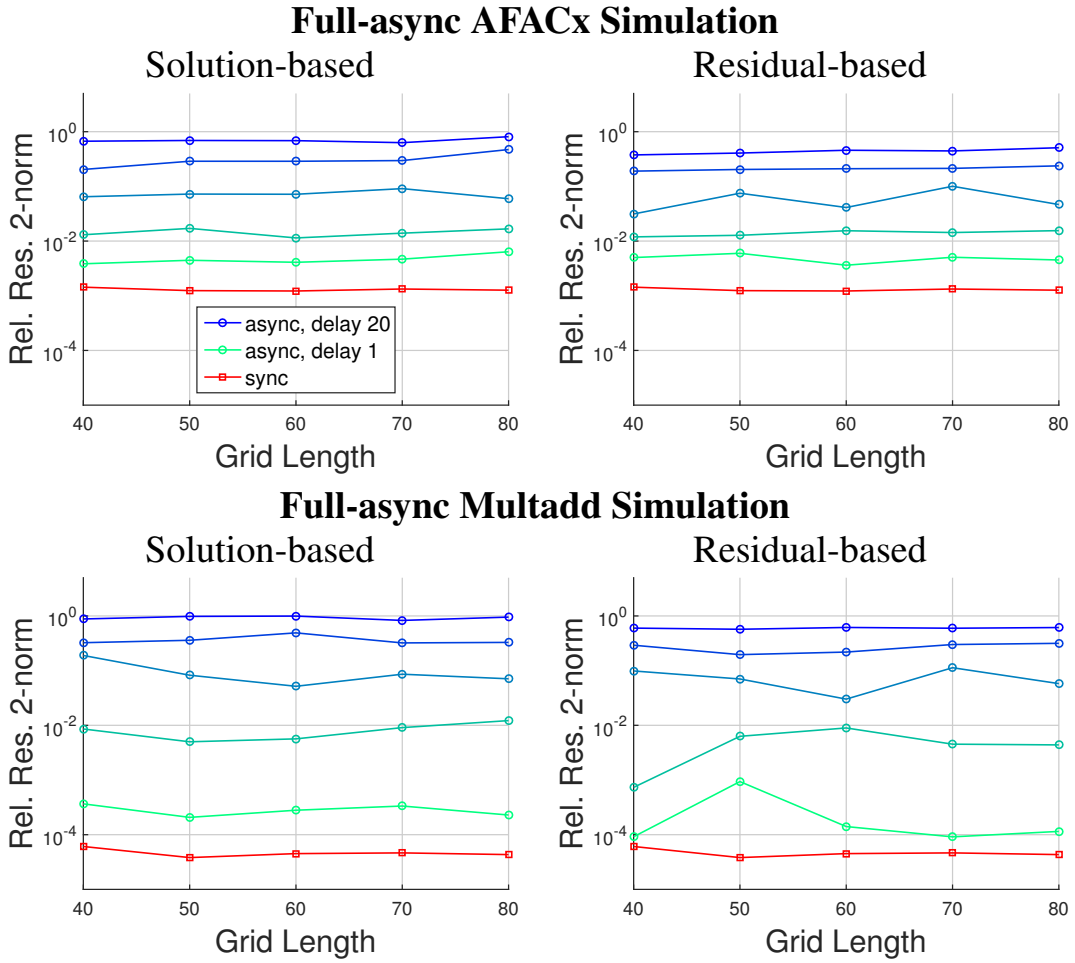


Figure 4.2: Final relative residual 2-norm after 20 V-cycles versus grid length for the full-asynchronous multigrid model. The solution-based (Equation 4.5) and residual-based versions (Equation 4.8) of AFACx and Multadd are shown. A minimum update probability of .1 is used and results for five maximum delay values are shown, where blue-to-green gradient corresponds to decreasing maximum delay. The 27pt test set is used (see Section 4.3.2). These results show that even with large delays, asynchronous multigrid still exhibits grid-size independent convergence.

1. **global-res:** Just like in classical multigrid, each thread would be responsible for computing some number of elements of the fine grid residual r, and r would be computed using a parallel SpMV operation using all five threads. We call this the *global-res algorithm* since, in addition to x, r is a "global" variable. Here, "global" refers to memory that can be read by all threads, while memory that is "local" to a grid refers to memory that can be read only by threads assigned to that grid. Algorithm 9 and

Figure 4.3 show global-res for this example, where Sync() denotes the synchronization of the threads listed. In line 1 of Algorithm 9, all threads take part in computing r using a parallel SpMV operation. If we are using OpenMP, the computation of r would be parallelized using a parallel for loop with a static scheduling.

In the if statements, only the threads assigned to a grid take part in each operation, which are also carried out with parallel loops. For example, in the case of grid 0, if we are using OpenMP, $\Lambda_0 r$ would be computed using a parallel for loop but only with the threads t_0 and t_1. In the case of grid 1, the application of $\overline{P}_1^0$, A_1^{-1}, and $(\overline{P}_1^0)^T$ to a vector are carried out by threads t_2, t_3 and t_4 (SpMV and triangular solve operations), where the three threads synchronize after each application.

Note that the updates for both grid are added to x concurrently in lines 6 and 11, which creates a race condition. We will discuss later in this section how these race conditions are handled.

2. **local-res:** Only x is a global variable. Threads assigned to a grid would read x from memory and then compute a local residual, e.g., threads t_0 and t_1 would compute the local residual r^0 using a parallel for loop. We call this the *local-res algorithm*, which is shown in Algorithm 10 and in Figure 4.3. The two threads first read x in line 1, and then in lines 4 and 9, threads t_0 and t_1 compute r^0 and r^1, respectively, which are the local residuals. The rest of the algorithm is the same as that of global-res.

In Algorithms 9 and 10, to make these algorithms asynchronous, we simply replace all Sync(t_0, t_1, t_2, t_3, t_4) operations with Sync(t_0, t_1) and Sync(t_2, t_3, t_4), i.e., we replace all global synchronizations with synchronizations of subsets of threads, where each subset is the set of threads assigned to a grid, and the union of all the subsets is the set of all threads. This means that there is some synchronization, but only among threads assigned to the same grid.

Algorithm 9: **global-res** for two-grid synchronous Multadd with five threads

$r = b - Ax$
Sync$(t_0, t_1, t_2, t_3, t_4)$
if *threads* t_0, t_1 **then**
    Sync(t_0, t_1)
    $e^0 = \Lambda_0 r$
    Sync(t_0, t_1)
    $x = x + e^0$
end
if *threads* t_2, t_3, t_4 **then**
    $c = (\overline{P}_1^0)^T r$
    Sync(t_2, t_3, t_4)
    $d = A_1^{-1} c$
    Sync(t_2, t_3, t_4)
    $e^1 = \overline{P}_1^0 d$
    Sync(t_2, t_3, t_4)
    $x = x + e^1$
end
Sync$(t_0, t_1, t_2, t_3, t_4)$

Algorithm 10: **local-res** for two-grid synchronous Multadd with five threads

$x^0 = x^1 = x$
Sync$(t_0, t_1, t_2, t_3, t_4)$
if *threads* t_0, t_1 **then**
    $r^0 = b - Ax^0$
    Sync(t_0, t_1)
    $e^0 = \Lambda_0 r^0$
    Sync(t_0, t_1)
    $x = x + e^0$
end
if *threads* t_2, t_3, t_4 **then**
    $r^1 = b - Ax^1$
    Sync(t_2, t_3, t_4)
    $c = (\overline{P}_1^0)^T r^1$
    Sync(t_2, t_3, t_4)
    $d = A_1^{-1} c$
    Sync(t_2, t_3, t_4)
    $e^1 = \overline{P}_1^0 d$
    Sync(t_2, t_3, t_4)
    $x = x + e^1$
end
Sync$(t_0, t_1, t_2, t_3, t_4)$

A problem with the global-res algorithm comes from how the residual is computed. As stated earlier, if the computation of an update for some grid is severely delayed, the faster grids may do many updates with a residual that has some components that are up-to-date, and other components that are very out-of-date. We will see in Section 4.3.2 that this can result in asynchronous multigrid diverging or converging more slowly than synchronous multigrid. The local-res algorithm does not suffer from this problem, but requires more

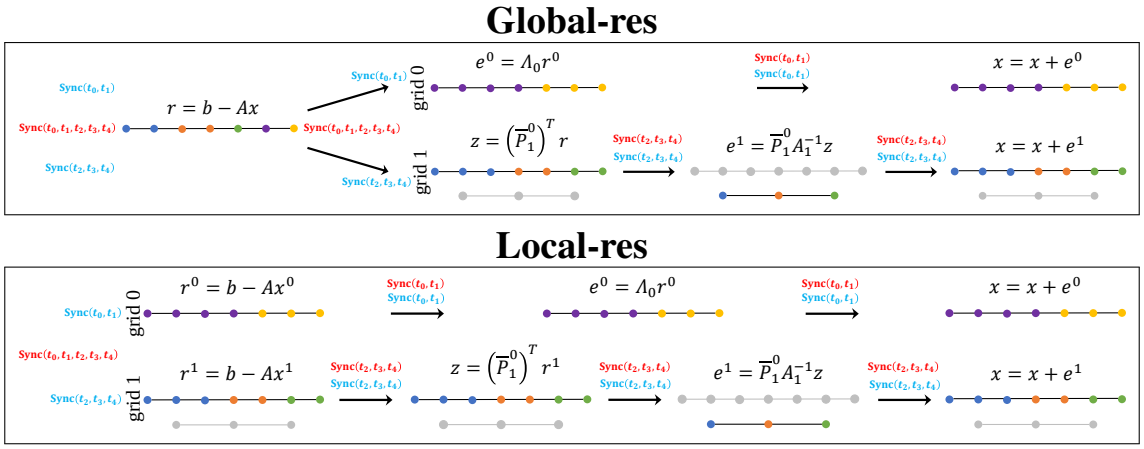


Figure 4.3: Global-res and local-res partitionings for the Multadd example presented in Section 4.3 for each step of the computation of the updates e^0 and e^1. Arrows denote moving to the next step of the computation. Sync() denotes a synchronization point, where the list of threads passed to Sync() denotes the threads that synchronize. Blue **Sync()** denotes a synchronization for asynchronous multigrid, and red **Sync()** denotes a synchronization point for synchronous multigrid. Colored points denote points used in a calculation, where t_0 is assigned the purple points, t_1 is assigned the yellow points, t_2 is assigned the blue points, t_3 is assigned the orange points, and t_4 is assigned the green points. Gray points denote points not used in a calculation.

computation per thread.

As mentioned earlier, when multiple updates are added to the global variable x (x and r in the case of global-res), we must handle race conditions since threads assigned to different grids update x concurrently. One option is to use a mutex lock. For this option, all grids have a master thread. All threads assigned to grid k block until the master thread for grid k acquires a mutex lock. Once the lock is acquired, the variable is updated by all threads assigned to grid k using a parallel for loop. We call this option the *lock-write* option. The second option is to use an atomic fetch-and-add operation inside the parfor loop. We call this option the *atomic-write* option.

We can now write asynchronous multigrid, as shown in Algorithm 11. In the algorithm:

- The k superscript denotes a variable stored in the local memory of grid k.
- For grid k, the operations Smooth(), Prolong(), Restrict(), Read(), Ax^k, and $x + e^k$ are carried using blocking parallel for loops (threads synchronize after completing

the loop), where only the threads assigned to k carry out the loops.

- x and r are global, i.e., can be accessed by any grid.
- The flag *rescomp_type* (local or global) specifies whether global-res or local-res is used.
- The Write() operation handles race conditions (explained above) when writing to a global variable.
- The **GlobalParFor** loop is executed by all threads, which is the global computation of the residual for global-res. The **No Wait** denotes a non-blocking parallel for loop, which is conceptually the same as adding a "no wait" clause to an OpenMP parfor loop.
- In lines 19-26, the residual can instead be computed as $r = r - Ae$ instead of $r = b - Ax$ as outlined in Section 4.2.

Algorithm 11: Asynchronous multigrid for grid k

```
1  Initialize r_0^k = r = b                          ▷ initialize local residuals
2  while grid k has not converged do                 ▷ procedure for grid k
3  |   r_k^k = Restrict(r_0^k)
4  |   if k == ℓ then
5  |   |   e_k^k = Smooth(A_k, r_k^k)
6  |   else
7  |   |   e_k^k = ExactSolve(A_k, r_k^k)
8  |   end
9  |   e_0^k = Prolong(e_k^k)
10 |   x = Write(x + e_0^k)                          ▷ add to x
11 |   x^k = Read(x)                                 ▷ store x to local memory
12 |   if rescomp_type == local then
13 |   |   r_0^k = b − Ax^k                          ▷ recompute local residual
14 |   else
15 |   |   No Wait GlobalParfor i = 1, ..., n do
16 |   |   |   r_i = Write(b_i − Σ_{j=1}^n a_ij x_j)   ▷ compute global residual
17 |   |   end
18 |   |   r_0^k = Read(r)                           ▷ store r to local memory
19 |   end
20 end
```

In terms of the models presented in Section 4.2, only local-res with the lock-write option can be modeled by semi-async (Equation 4.4). All other variations of Algorithm 11 can be modeled by full-async (Equations 4.5 and 4.8).

4.3.2 Experimental Results

Test Framework

For our numerical results, we used an Intel Xeon Phi Knights Landing (KNL) processor with 68 cores and 272 threads. We implemented synchronous multigrid (Mult) using OpenMP parallel for loops with static scheduling. For synchronous Multadd and AFACx, each grid was assigned threads in the same way as the asynchronous local-res implementation. This thread partitioning is used only to compute updates concurrently. At the end of a single cycle, all threads synchronize and carry out an SpMV to compute the residual using an OpenMP parallel for loop. This is the same way the residual is computed in Mult.

We experimented with four different smoothers: weighted Jacobi (ω-Jacobi), ℓ_1-Jacobi [63], hybrid Jacobi Gauss-Seidel (hybrid JGS) [63], and asynchronous Gauss-Seidel (async GS). As in ω-Jacobi, ℓ_1-Jacobi uses a diagonal smoothing matrix, where the diagonal entries are the L1 norms of the rows of A, i.e., $M_{ii} = \sum_{j=1}^{n} |a_{ij}|$. It can be shown that if A is symmetric and positive-definite, the error monotonically decreases in the A-norm when ℓ_1-Jacobi is used as a smoother.

The hybrid Jacobi Gauss-Seidel smoother can be thought of as an inexact block Jacobi method where the blocks are solved inexactly using a small number of Gauss-Seidel sweeps. In our experiments, we only consider using one sweep. For parallel smoothers, the number of subdomains is equal to the number of processes or threads, making the method highly parallel. However, without proper weighting [82] or using an ℓ_1 variation of the method [63], the method can diverge if many subdomains are used. Asynchronous Gauss-Seidel is an asynchronous version of hybrid JGS. For a shared memory implementation, a thread relaxes a subset of rows (approximately n/p rows), and immediately writes the new

information to memory after each relaxation. This means that information that is read from memory could be a mix of new and old information, which is modeled in Equation 2.5.

We used BoomerAMG [81] to generate the prolongation and coarse grid matrices for all our multigrid methods. For Multadd, if ℓ_1-Jacobi was used as a smoother, we used the ℓ_1-Jacobi iteration matrix to construct the smoothed interpolants. For all other smoothers, we used the ω-Jacobi iteration matrix. We did this for performance reasons, i.e., we wanted to keep the smoothed interpolants sparse, even though the convergence may be slower than when using a hybrid or asynchronous smoother. For example, for a V(1,1)-cycle of Multadd with hybrid JGS, $\overline{P}_{k+1}^k = (I - \omega D_k^{-1} P_{k+1}^k)$ and Λ_k is the block diagonal matrix with blocks $L_{k1}^{-1}, \ldots, L_{kp}^{-1}$, where L_{ki} is the lower triangular part of block i of A_k, for $i = 1, \ldots, p$. In our results, we only consider using one smoothing sweep since it is not clear how to do multiple sweeps with Multadd while keeping the smoothed interpolants fixed.

We used four sets of test matrices with different problem sizes within each set. Two of these sets were generated using the MFEM software package [83]:

- The three-dimensional Laplace matrices in a cube discretized using the 7-point and 27-point centered difference methods. We will refer to these two sets as 7pt and 27pt.
- The three-dimensional Laplace matrices in a sphere discretized using a NURBS mesh [84] and H^1 nodal finite elements. These matrices were generated using the MFEM package [83]. We will refer to these matrices as MFEM Laplace.
- Three-dimensional linear elasticity matrices modeling a multi-material cantilever beam using a tetrahedral mesh and H^1 nodal finite elements. These matrices were generated using the MFEM package [83]. We will refer to these matrices as MFEM Elasticity.

We used random right-hand sides with values in $[-1, 1]$.

Convergence detection for asynchronous methods is not a trivial task (see Chapter 2). In our implementations, we do not try to detect when the global relative residual norm

$\|r\|_2/\|b\|_2$ falls below some specified tolerance τ. This would require a subset of grids to compute a norm, which is an extra delay on that grid, and the relative residual norm generally does not monotonically decrease. There are two convergence criteria we use to detect when t_{max} V-cycles have been carried out:

- **Criterion 1**: A grid immediately breaks from the main loop when it has computed $t_{\max}$ updates. This means that grids can finish iterating before other grids have finished. This is the same criterion used in the simulations in Section 4.2.

- **Criterion 2**: A single master thread is in charge of making sure all grids have computed at least $t_{\max}$ updates. This thread then sets a flag indicating that the iteration must terminate. For a thread that is not the master, it reads this flag after finishing computing an update. If the flag is not set, the thread computes another update. Otherwise, it exits the main solve loop.

To find the wall-clock time required to reduce $\|r\|_2/\|b\|_2$ below some tolerance, we plot $\|r\|_2/\|b\|_2$ versus wall-clock time, saving time stamps of $\|r\|_2/\|b\|_2$ for doing a small to large number of cycles, e.g., we do $5, 10, \ldots, 100$ V-cycles, saving $\|r\|_2/\|b\|_2$ and the wall-clock time for each number of V(1,1)-cycles. When saving the wall-clock times, we do multiple runs for each number of cycles and take the mean of the wall-clock times for those runs (for asynchronous methods, we also take the mean of the relative residual 2-norms). We then find the wall-clock time corresponding to the first occurrence of $\|r\|_2/\|b\|_2 < \tau$. For all our experiments, we took the average of 20 runs and set $\tau = 10^{-9}$.

Grid-size Independent Convergence

We first show that asynchronous multigrid methods can exhibit grid-size independent convergence. Figure 4.4 shows $\|r\|_2/\|b\|_2$ after 20 V(1,1)-cycles (see Section 4.3.2 for how a V-cycle is defined in the asynchronous case) versus the grid length for the 7pt and 27pt test sets (a grid length of 40 denotes a $40 \times 40 \times 40$ mesh) using 68 threads. Each data point

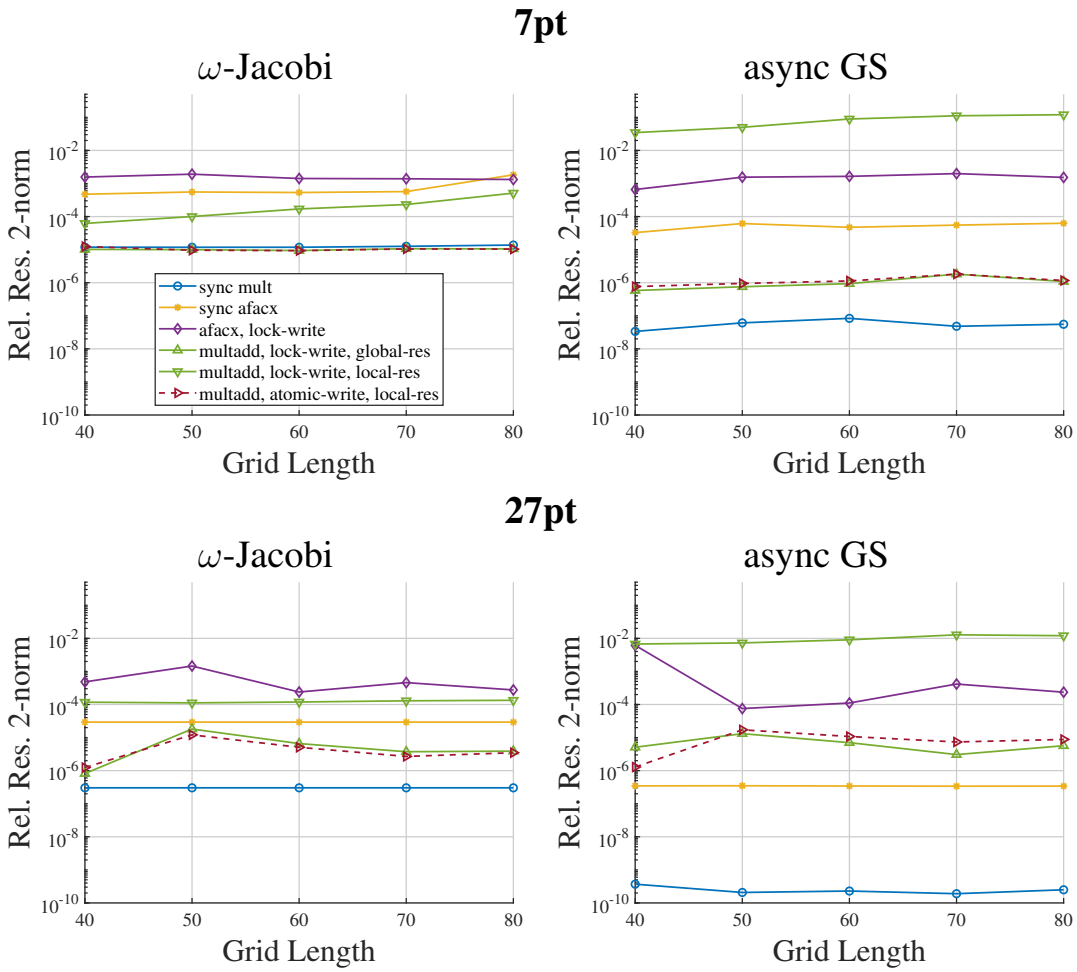


Figure 4.4: Relative residual 2-norm versus grid length for 20 V(1,1)-cycles and 68 threads. Results for the 7pt and 27pt test sets are shown. For each test set, results for two smoothers are shown. For the asynchronous methods, we used Criterion 1 as our stopping criterion (see Section 4.3.2), and each data point is the mean relative residual 2-norm of 20 runs. The figures show that asynchronous multigrid methods can exhibit grid-size independent convergence.

is the mean $\|r\|_2/\|b\|_2$ of 20 runs. For the asynchronous methods, we used Criterion 1 for convergence detection (see Section 4.3). Results for ω-Jacobi and async GS smoothing are shown. For our BoomerAMG options, we chose HMIS coarsening with one aggressive level, and classical modified interpolation. If "sync" is written next to a legend entry, the method is synchronous. Otherwise, the method is asynchronous.

Figure 4.4 shows that all the asynchronous methods approximately achieve grid-size independent convergence, even when using async GS as the smoother. We can also see that

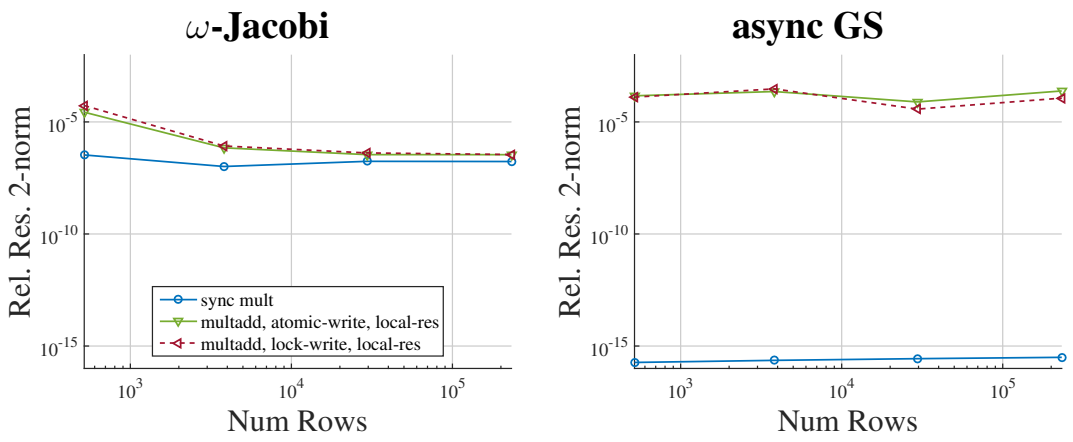


Figure 4.5: Relative residual 2-norm versus number of rows for 20 V(1,1)-cycles and 68 threads. The MFEM Laplace matrix is used and results for the ω-Jacobi and async GS smoothers are shown. The figures show that asynchronous multigrid can exhibit grid-size independent convergence.

in most cases, global-res results in a solver that converges more slowly than when using local-res. This is due to grids using fine grid residual values that are delayed. In other words, since the threads assigned to grid k compute an update using values of r_0 that are computed exclusively by other grids, grid k could be using very old values if another update is delayed. Figure 4.5 shows the same experiment but with the MFEM Laplace test set and no aggressive coarsening. Multadd local-res lock-write exhibits grid-size independent convergence. AFACx (synchronous and asynchronous) and Multadd global-res did not exhibit grid-size independent convergence for this test set.

Wall-clock Time for Reducing $\|r\|_2/\|b\|_2 < 10^{-9}$

Table 4.1 shows results for four test matrices, one from each test set, and 272 threads. For the asynchronous methods, we used Criterion 2 for convergence detection (see Section 4.3). *Updates* is the average number of updates over all grids. For each test matrix, results for four smoothers are shown. For our BoomerAMG options, we chose HMIS coarsening with two aggressive levels, and classical modified interpolation. The r- prefix in r-Multadd denotes that Multadd was implemented using the residual-based implementation (in Sec-

tion 4.3, see the last bullet of the explanation of Algorithm 11). These results show that, with the exception of async GS for the MFEM Elasticity matrix, asynchronous Multadd requires the lowest wall-clock time, even if it requires more computation than Mult (higher number in the updates column). Additionally, using atomic operations is slower than using locks, with the exception of r-Multadd for the MFEM Laplace matrix with the async GS smoother. In some cases (MFEM Laplace with ω-Jacobi smoothing, and 7pt with hybrid JGS smoothing), global-res is the best solver. In most cases, local-res is the best solver since it requires significantly fewer V-cycles to converge. Finally, using async GS smoothing always requires the lowest number of V(1,1)-cycles and least wall-clock time compared to the other smoothers.

Strong Scaling

Figure 4.6 shows the wall-clock time versus the number of threads for the same four matrices from Table 4.1. The BoomerAMG options are the same options used in Table 4.1 and w-Jacobi smoothing is used. Each subfigure shows three methods: sync Mult, sync Multadd lock-write, and Multadd lock-write local-res. In all subfigures, we can see that with a low number of threads, Mult is typically the fastest since synchronization is not a large cost compared to the cost of computation. However, asynchronous Multadd is the fastest for a sufficiently large number of threads, and scales better, i.e., as the number of threads increases, the wall-clock time of asynchronous Multadd does not increase as much as that of Mult. This provides a good outlook for distributed memory, where the number of parallel processes is orders of magnitude higher, and in the case of exascale machines, the problem size per process may be quite small. We also see that synchronous Multadd scales better than Mult, demonstrating that computing updates concurrently can be beneficial. This is because there is only global communication on the fine grid for synchronous Multadd, whereas for Mult, there is global synchronization on every grid.

Table 4.1: Timing results for four test matrices, and for each matrix, four smoothers. 272 threads are used. For each smoother, results for all multigrid methods are shown (see Section 4.3 for explanations of lock-write, atomic-write, local-res, and global-res). The † marker indicates that a method diverged. For each smoother, the **bolded** number indicates the lowest wall-clock time among all the methods. These results show that asynchronous Multadd is generally faster than the classical multiplicative multigrid method (Mult) in terms of wall-clock time, and async GS is the best smoother for all matrices.

7pt: 27,000 rows and 183,600 non-zero values

	ω-Jacobi, ω = .9			ℓ_1-Jacobi			hybrid JGS			async GS		
method	time	updates	V-cycles	time	updates	V-cycles	time	updates	V-cycles	time	updates	V-cycles
sync Mult	0.1164	75	75	0.1927	120	120	0.1009	65	65	0.0828	55	55
sync Multadd, lock-write	0.0305	75	75	0.0490	120	120	0.0405	100	100	0.0323	80	80
sync Multadd, atomic-write	0.0299	75	75	0.0465	120	120	0.0393	100	100	0.0322	80	80
sync AFACx, lock-write	0.0489	135	135	†	†	†	0.0420	115	115	0.0339	95	95
sync AFACx, atomic-write	0.0481	135	135	†	†	†	0.0418	115	115	0.0337	95	95
AFACx, lock-write	0.0429	154	110	†	†	†	0.0430	142	110	0.0349	115	90
AFACx, atomic-write	0.0575	160	120	†	†	†	0.0533	138	110	0.0466	121	95
Multadd, lock-write, global-res	0.0249	89	70	†	†	†	**0.0267**	97	75	0.0591	192	155
Multadd, lock-write, local-res	**0.0200**	73	45	**0.0326**	123	75	0.0269	97	60	**0.0203**	74	45
Multadd, atomic-write, global-res	0.0286	78	70	†	†	†	0.0351	97	85	0.0293	80	70
Multadd, atomic-write, local-res	0.0259	71	50	0.0441	123	85	0.0360	98	70	0.0310	86	60
r-Multadd, atomic-write, local-res	0.0257	69	50	0.0452	122	90	0.0359	94	70	0.0281	76	55

27pt: 27,000 rows and 681,472 non-zero values

	ω-Jacobi, ω = .9			ℓ_1-Jacobi			hybrid JGS			async GS		
method	time	updates	V-cycles	time	updates	V-cycles	time	updates	V-cycles	time	updates	V-cycles
sync Mult	0.0939	65	65	0.1553	105	105	0.0795	55	55	0.0581	40	40
sync Multadd, lock-write	0.0259	65	65	0.0414	105	105	0.0349	90	90	0.0281	70	70
sync Multadd, atomic-write	0.0250	65	65	0.0400	105	105	0.0355	90	90	0.0254	65	65
sync AFACx, lock-write	0.0451	120	120	†	†	†	0.0383	100	100	0.0282	75	75
sync AFACx, atomic-write	0.0429	120	120	†	†	†	0.0380	100	100	0.0274	75	75
AFACx, lock-write	0.0420	120	85	†	†	†	0.0418	110	85	0.0324	85	65
AFACx, atomic-write	0.0465	112	85	†	†	†	0.0464	108	85	0.0385	90	70
Multadd, lock-write, global-res	0.0254	79	65	†	†	†	0.0321	119	95	0.0481	150	125
Multadd, lock-write, local-res	**0.0206**	58	40	**0.0304**	93	60	**0.0280**	85	55	**0.0231**	65	45
Multadd, atomic-write, global-res	0.0339	98	95	†	†	†	0.0357	105	100	0.0342	99	95
Multadd, atomic-write, local-res	0.0223	56	40	0.0336	89	60	0.0308	81	55	0.0253	65	45
r-Multadd, atomic-write, local-res	0.0254	62	45	0.0362	89	65	0.0391	92	70	0.0282	69	50

MFEM Laplace: 29,521 rows and 781,297 non-zero values

	ω-Jacobi, ω = .5			ℓ_1-Jacobi			hybrid JGS			async GS		
method	time	updates	V-cycles	time	updates	V-cycles	time	updates	V-cycles	time	updates	V-cycles
sync Mult	0.2404	150	150	0.2473	155	155	†	†	†	0.0924	60	60
sync Multadd, lock-write	0.0924	150	150	0.0964	155	155	0.0847	140	140	0.0588	95	95
sync Multadd, atomic-write	0.0909	150	150	0.0949	155	155	0.0845	140	140	0.0586	95	95
sync AFACx, lock-write	0.1316	295	295	†	†	†	†	†	†	0.0572	100	100
sync AFACx, atomic-write	0.1314	295	295	†	†	†	†	†	†	0.0563	100	100
AFACx, lock-write	0.1442	300	235	†	†	†	†	†	†	0.0730	135	120
AFACx, atomic-write	0.1532	296	230	†	†	†	†	†	†	0.0751	127	115
Multadd, lock-write, global-res	**0.0737**	189	160	†	†	†	0.0677	177	145	0.0652	166	140
Multadd, lock-write, local-res	0.0782	148	110	**0.0818**	154	115	**0.0636**	127	90	0.0513	94	70
Multadd, atomic-write, global-res	0.0788	172	160	†	†	†	0.0721	159	145	0.0691	147	140
Multadd, atomic-write, local-res	0.0836	149	115	0.0899	158	120	0.0732	135	100	0.0564	97	75
r-Multadd, atomic-write, local-res	0.0790	145	110	0.0845	153	115	0.0644	122	90	**0.0512**	93	70

MFEM Elasticity: 37,281 rows and 251,617 non-zero values

	ω-Jacobi, ω = .5			ℓ_1-Jacobi			hybrid JGS			async GS		
method	time	updates	V-cycles	time	updates	V-cycles	time	updates	V-cycles	time	updates	V-cycles
sync Mult	0.3425	190	190	0.3352	190	190	0.1736	100	100	0.1465	85	85
sync Multadd, lock-write	0.1367	190	190	0.1361	190	190	0.1134	165	165	0.0902	125	125
sync Multadd, atomic-write	0.1337	190	190	0.1346	190	190	0.1119	165	165	**0.0888**	125	125
sync AFACx, lock-write	0.2301	385	385	†	†	†	0.1150	195	195	0.1109	170	170
sync AFACx, atomic-write	0.2269	385	385	†	†	†	0.1134	195	195	0.1107	170	170
AFACx, lock-write	0.2103	404	310	†	†	†	†	†	†	0.1603	268	235
AFACx, atomic-write	0.2378	405	315	†	†	†	†	†	†	0.2006	301	260
Multadd, lock-write, global-res	†	†	†	†	†	†	†	†	†	†	†	†
Multadd, lock-write, local-res	**0.1098**	192	145	**0.1099**	195	145	**0.0934**	171	125	0.0904	152	115
Multadd, atomic-write, global-res	†	†	†	†	†	†	†	†	†	†	†	†
Multadd, atomic-write, local-res	0.1266	201	160	0.1268	202	160	0.1174	192	150	0.1008	156	125
r-Multadd, atomic-write, local-res	0.1177	193	155	0.1185	195	155	0.1014	169	135	0.0927	150	120

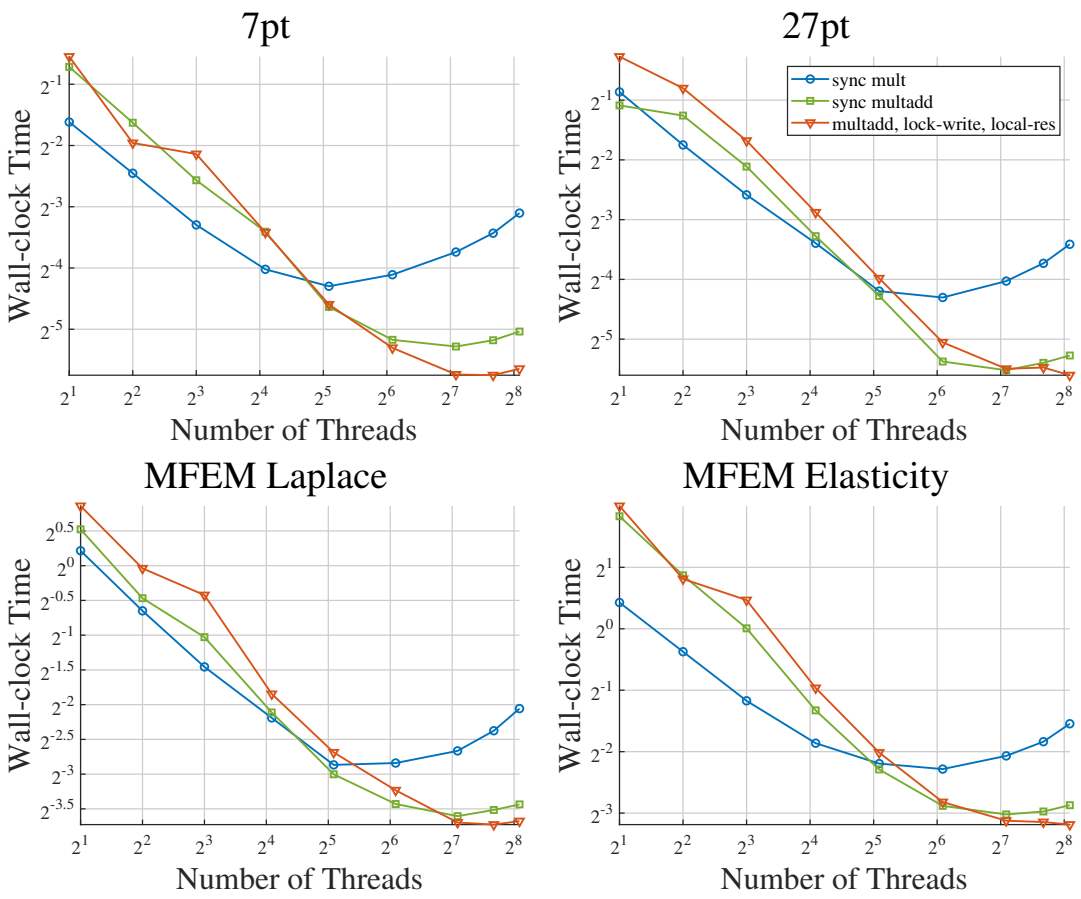


Figure 4.6: Wall-clock time versus number of threads for the 7pt, 27pt, MFEM Laplace, and MFEM Elasticity matrices (see Table 4.1) are shown with ω-Jacobi smoothing. The BoomerAMG options are the same as that of Table 4.1. The figures show that asynchronous multigrid is faster than synchronous multigrid for a sufficiently large number of threads, and typically scales better.

4.4 The AFACj Multigrid Method

We start by returning to the AFACx method, where the steps are carried out by the processes assigned to grid k after the residual is restricted down to grid k:

1. Compute e_{k+1} by smoothing on the equations $A_{k+1}e_{k+1} = r_{k+1}$ with an initial guess of zero.

2. Compute e_k by smoothing on the equations $A_k e_k = r_k$ with an initial guess of $P_{k+1}^k e_{k+1}$.

3. On the finest grid, add e_k to and subtract e_{k+1} from x to prevent an over-correction: $x = x + P_k^0(e_k - P_{k+1}^k e_{k+1})$.

Note that we can avoid redundant prolongation operations by modifying the last two steps in the following way:

2. Compute e_k by smoothing on the equations $A_k e_k = r_k - A_k P_{k+1}^k e_{k+1}$ with an initial guess of zero.

3. On the finest grid, add e_k to x: $x = x + P_k^0 e_k$.

We can now write AFACx(1,1) in a similar way to BPX (Equation 4.2). Unlike the classical multiplicative V(1,1)-cycle, the (1,1) notation for AFACx refers to using one smoothing sweep to compute e_{k+1} and one smoothing sweep to compute e_k. This gives us

$$x^{(t+1)} = x^{(t)} + \sum_{k=0}^{\ell-1} P_k^0 \Lambda_k (P_k^0)^T r^{(t)}, \tag{4.10}$$

where $\Lambda_k = M_k^{-1}(I_k - A_k P_{k+1}^k M_{k+1}^{-1}(P_{k+1}^k)^T)$ if $k < \ell - 1$ and $\Lambda_k = A_k^{-1}$ otherwise. We could also write the more general AFACx(s_1, s_2) method, where we use s_1 sweeps for computing e_{k+1} and s_2 sweeps for e_k, but for the derivation of AFACj we only consider the (1,1) case.

AFACx(1,1) can be written in terms of smoothed interpolants, which we will call AFACy, where AFACx(1,1) and AFACy are mathematically equivalent. This done by having the computation of e_{k+1} be done by grid $k+1$ instead of grid k. We can write AFACy as

$$x^{(t+1)} = x^{(t)} + \sum_{k=0}^{\ell-1} P_1^0 \cdots P_{k-1}^{k-2} \overline{P}_k^{k-1} \Lambda_k (P_k^0)^T r^{(t)}, \tag{4.11}$$

where $\Lambda_k = M_k^{-1}$ if $k < \ell - 1$ and $\Lambda_k = A_k^{-1}$ otherwise. This is equivalent to BPX but with a single smoothed interpolant for each grid.

Modifying AFACy to introduce additional smoothed interpolants results in the AFACj(s_1, s_2) method. As in the case of AFACy, to derive this method we start with AFACx. When

computing e_k for grid k, the important variation on AFACx that leads to AFACj is to include pre-smoothing, i.e., smoothing on $Ae_k = r_k$ before smoothing $Ae_{k+1} = r_k$. Here, pre-smoothing is different than in the case of classical multiplicative multigrid. With this variation, we can again write AFACx in terms of smoothed interpolants, as in the case of AFACy. We will write AFACx as AFACx(s_1, s_2, s_3), where for grid k, s_1 is the number of pre-smoothing sweeps on $A_k e_k = r_k$, s_2 is the number of post-smoothing sweeps on $A_k e_k = r_k$, and s_3 is the number smoothing sweeps on $A_{k+1} e_{k+1} = r_{k+1}$. In the original AFACx method as presented in [73], $s_1 = 0$ since pre-smoothing is not considered.

To incorporate pre-smoothing, first consider the following steps for AFACx$(1, 1, 2)$, where $b_k = (P_k^0)^T r_0$:

1. Pre-smooth on $A_k x_k = b_k$ with zero initial guess: $x_k = M_k^{-T} b_k$.
2. Compute residual: $r_k = b_k - A x_k$.
3. Smooth on $A_{k+1} e_{k+1} = (P_{k+1}^k)^T r_k$ with zero initial guess: $e_{k+1} = \Lambda_{k+1} (P_{k+1}^k)^T r_k$.
4. correct: $x_k = x_k + P_{k+1}^k e_{k+1}$.
5. post-smooth on $A_k x_k = b_k$: $x_k = x_k + M_k^{-1}(b_k - A_k x_k)$.
6. subtract to prevent over-correction: $x_k = x_k - P_{k+1}^k M_{k+1}^{-1} (P_{k+1}^k)^T b_k$.
7. Interpolate x_k to the finest grid and add to the current approximation to the solution.

Note that steps 3-7 are the same as in AFACx$(0, 1, 1)$, i.e., the classical AFACx that we have been considering up until now, but $(P_{k+1}^0)^T r_0$ is used as the right-hand side when computing e_{k+1} and $\Lambda_{k+1} = M_{k+1}^{-1}$. Let $C = P_{k+1}^k \Lambda_{k+1} (P_{k+1}^k)^T$. Writing out x_k in terms of b_k

and collecting terms:

$$\begin{aligned} x_k &= \left[(M_k^{-1} + M_k^{-T} - M_k^{-1} A_k M_k^{-T}) - \right. \\ &\quad \left. C A_k M_k^{-T} - M_k^{-1} A_k C + M_k^{-1} A_k C A_k M_k^{-T} \right] b_k \\ x_k &= \left[\Lambda_k - C A_k M_k^{-T} - M_k^{-1} A_k C + M_k^{-1} A_k C A_k M_k^{-T} \right] b_k, \end{aligned} \tag{4.12}$$

where $\Lambda_k = M_k^{-1} + M_k^{-T} - M_k^{-1} A_k M_k^{-T}$. This shows that the symmetrized smoothing matrix plus a perturbation is multiplied by b_k. If we set Λ_{k+1} to also be the symmetrized smoothing matrix, two smoothing sweeps are used to compute e_{k+1}, which is why $s_3 = 2$ in this case. To derive AFACy(1,1,2), we move the perturbation down a grid. For grid $k+1$ this gives us

$$\begin{aligned} x_{k+1} &= \left[P_{k+1}^k \Lambda_{k+1} (P_{k+1}^k)^T - C A_k M_k^{-T} - M_k^{-1} A_k C + M_k^{-1} A_k C A_k M_k^{-T} \right] b_k \\ &= (I_k - M_k^{-1} A_k) C (I_k - A_k M_k^{-T}) b_k \\ &= \overline{P}_{k+1}^k \Lambda_{k+1} (\overline{P}_{k+1}^k)^T b_k, \end{aligned} \tag{4.13}$$

This gives us the AFACy$(1,1,2)$ method.

If we generalize this idea to include more smoothed interpolants, we have the AFACj method. Specifically, s_1 smoothed prolongation matrices and s_2 smoothed restriction matrices., which gives us the AFACj(s_1, s_2) method. For example, AFACj(2,2) can be written as

$$x^{(t+1)} = x^{(t)} + \sum_{k=0}^{\ell-1} P_1^0 \cdots P_{k-2}^{k-3} \overline{P}_{k-1}^{k-2} \overline{P}_k^{k-1} \Lambda_k (\overline{P}_k^{k-1})^T (\overline{P}_{k-1}^{k-2})^T (P_{k-2}^{k-3})^T \cdots (P_1^0)^T r^{(t)}, \tag{4.14}$$

where $\Lambda_k = M_k^{-T} + M_k^{-1} - M_k^{-T} A_k M_k^{-1}$ if $k < \ell - 1$ and $\Lambda_k = A_k^{-1}$ otherwise.

While we do not present any analytical convergence results for AFACj, we have found that in practice AFACj converges faster than AFACx in terms of number of iterations. Figure 4.7 shows the relative residual 2-norm versus number of cycles for

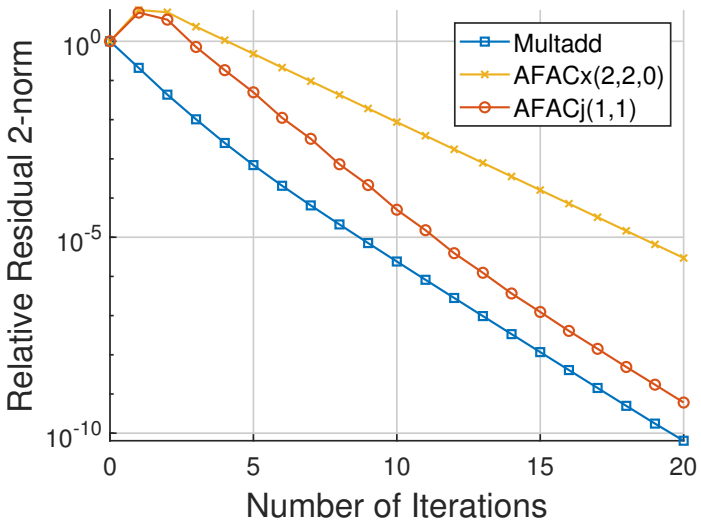


Figure 4.7: Relative residual 2-norm versus number of iterations for Multadd, AFACx$(2,2,0)$, and AFACj$(1,1)$. All methods are synchronous. The test problem is the five-point centered-difference discretization of the 2D Laplace equation on a 1024×1024 grid.

Multadd, AFACx$(2,2,0)$, and AFACj$(1,1)$. For this experiment, all methods are synchronous. We include AFACx$(2,2,0)$ since the same amount of computation is carried out in AFACx$(2,2,0)$ and AFACj$(1,1)$. The test problem is the five-point centered-difference discretization of the 2D Laplace equation on a 1024×1024 grid. Weighted Jacobi smoothing is used with a weight of 4/5, which is optimal for Multadd. The results show that AFACj$(1,1)$ converges significantly faster than AFACx$(2,2,0)$. Additionally, while AFACj$(1,1)$ converges more slowly than Multadd, AFACj$(1,1)$ and Multadd have approximately the same convergence rate.

4.5 Asynchronous Multigrid for Distributed Memory

4.5.1 Implementation

Our implementation of asynchronous multigrid is a distributed memory implementation of the local-res algorithm presented in [65]. In [65], the local-res algorithm was implemented in shared memory. In the local-res algorithm, each grid k is assigned a set of threads and those threads synchronously compute the update for grid k. Each thread is assigned to

one grid so that threads from different grids do not synchronize. This assignment is static, i.e., this assignment does not change during the solve phase. Threads are assigned to grids such that the computational cost per process is approximately balanced. The current approximation to the solution x is a shared variable, where the threads assigned to grid k atomically read x from shared memory in order to compute the update for grid k. After an update is computed, it is atomically added to x. For our distributed memory implementation, two important things are considered that make our implementation different from the shared memory implementation in [65]: how information is communicated such that the our implementation is asynchronous, and where x is stored.

An example of a process assignment to grids is as follows, where we consider a total of seven processes and three grids (this is an artificial example and does not come from any real problem):

$$x^{(t+1)} = x^{(t)} + \underbrace{\Lambda_0 r^{(t)}}_{p_0,p_1} + \underbrace{\overline{P}_1^0 \Lambda_1 (\overline{P}_1^0)^T r^{(t)}}_{p_2,p_3} + \underbrace{\overline{P}_2^0 A_2^{-1} (\overline{P}_2^0)^T r^{(t)}}_{p_4,p_5,p_6}. \tag{4.15}$$

In this example, processes p_0 and p_1 compute the update for grid 0 (this is just a smoothing step), processes p_2 and p_3 compute the update for grid 1, and processes p_4, p_5, and p_6 compute the update for grid 2. All of these updates are computed concurrently. Additionally, with the exception of grid 0, each update is computed synchronously, e.g., processes p_2 and p_3 must synchronize after each step. These steps are the multiplication of $\overline{P}_1^0$, Λ_1, and $(\overline{P}_1^0)^T r^{(t)}$ with a vector, the computation of the residual, and the computation of the residual norm for checking convergence.

Unlike the shared memory implementation in [65] where x is shared with all threads, in our implementation the processes assigned to grid k store a version of x that is distributed among these processes. This means that there are ℓ versions of x, each version distributed among the processes assigned to each grid. After a grid k computes an update, it sends the update to all processes not assigned to grid k (inter grid communication, explained below).

A process assigned to grid $j \neq k$ then adds the update from k to its version of x. If we instead only had one version stored by all the processes (or a subset of all the processes), processes would need to wait to receive new values of x, making the implementation synchronous. If the processes assigned to grid k did not wait and simply used the same values of x to compute the next update, an over-correction could occur and the method could diverge. We will refer to the version of x on grid k as the *local* vector $x^{(t_k,k)}$. Here, the superscript (t_k, k) refers to a vector local to grid k with local iteration number t_k. The processes assigned to grid k also store a local residual vector $r^{(t_k,k)}$. In the synchronous case, all t_k values are the same for all k.

Our implementation consists of two types of communications: intra grid communication and inter grid communication. Before explaining these two types of communication, the following steps are the most important in our implementation. These steps occur on some grid k:

1. Compute the update $B_k(x^{(t_k,k)})$ (intra grid communication).

2. Send the update to all other grids. Receive updates from other grids and add them to $x^{(t_k,k)}$ (inter grid communication).

3. Compute the residual $r^{(t_k,k)} = b - Ax^{(t_k,k)}$ (intra grid communication).

Intra grid communication is the communication between processes assigned to the same grid. It is used in matrix-vector products (communication of elements in the vector corresponding to off-diagonal values in the matrix) and inner products (reduction operations). Matrix-vector products are used for computing the update and residual, and inner products are used for computing the residual norm which is needed for checking the stopping criterion. This is similar to communication in a typical synchronous multiplicative multigrid implementation. Intra grid communication is also used for asynchronous smoothing. For the Jacobi smoother, our implementation only allows for asynchronous smoothing on grid zero since there is no benefit to using additional smoothing sweeps on coarser grids unless

there are also additional applications of the smoothing iteration matrix to the smoothed interpolants, which is due to how Multadd is derived.

The primary part of our implementation is the *inter grid* communication, which is the communication of updates. Inter grid communication occurs between processes assigned to different grids. Since each grid stores a copy of $x^{(t)}$, an update on one grid needs to be sent to all other grids in order for all copies of $x^{(t)}$ to be equal when the iteration has converged. We say that process q is an *inter grid neighbor* of process p if p needs to send data to q. We define $\Omega_{p,q}$ as the set of indices from the partition belonging to p that p needs to send to q. Similarly, we define $\Gamma_{p,q}$ as the set of indices from the partition of p that p needs to receive from q.

Returning to our example, Figure 4.8 shows the intra and inter grid communication of p_0 (p_0 is rank 0 in the figure). The hierarchy of grids needed for the updates of grids 0 to 2 are shown from left to right, where the processes assigned to grid 0 only smooth on the finest grid and the processes assigned to grid 2 restrict the residual down two grids where an exact solve is carried out. The colored blocks denote the partitionings of $x^{(t_k,k)}$ (and $r^{(t_k,k)}$) for the processes assigned to each grid. These partitionings are used in the matrix-vector products when computing updates. The blue arrows denote inter grid communication and the brown arrow denotes intra grid communication. For the inter grid communication, we can see that p_0 must send its sub-vector to p_2 and p_3 on grid one and p_5 and p_6 on grid two. For the intra grid communication, p_0 just needs to send boundary points to p_1 which is needed for matrix-vector products. Note that for grids one and two, there is also intra grid communication on coarser grids.

We used non-blocking two-sided MPI functions for the inter grid communication. Using one-sided MPI is also an option, but, as shown in [61], messages sometimes do not complete, and the performance of one-sided MPI can vary widely from system to system. Future work would be to integrate one-sided communication into implementation for systems where one-sided performs well. We used the BoomerAMG package to construct the

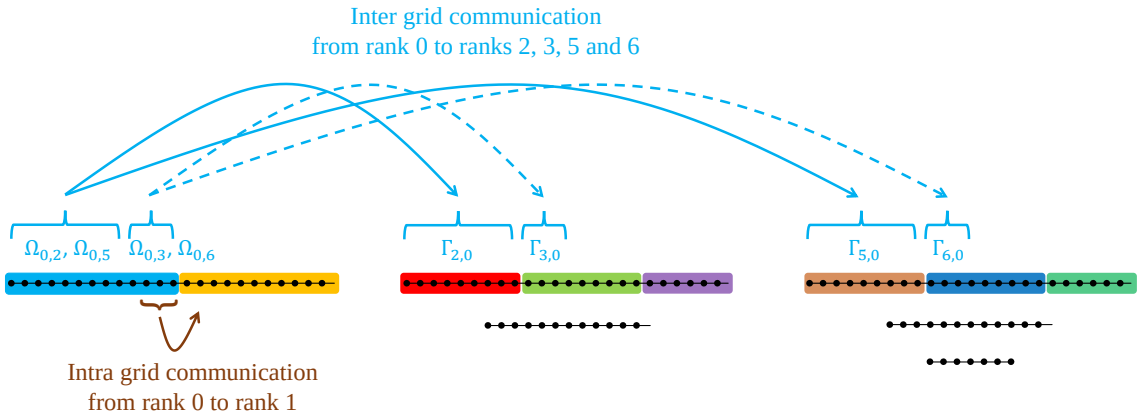


Figure 4.8: Inter and intra grid communication of process p_0 for the example in Section 4.5.1.

restriction, prolongation, and coarse grid matrices. Since intra grid communication is built into BoomerAMG, we did not need to implement our own intra grid communication except in the case of the fine grid asynchronous smoother.

For each inter grid neighbor an `MPI_Isend()` and `MPI_Irecv()` is always posted. `MPI_Test()` is used to check for the completion of `MPI_Isend()` and `MPI_Irecv()`. Since the send buffer used in `MPI_Isend()` cannot be overwritten until the `MPI_Isend()` completes, the user can specify the maximum number of in-flight messages, a concept introduced in [17]. This specifies how many `MPI_Isend()` calls can be simultaneously outstanding, where each `MPI_Isend()` has a separate send buffer that is allocated.

Algorithm 12 shows asynchronous multigrid implemented with non-blocking two-sided MPI. First, the update for grid k is computed (line 2). For grid zero, asynchronous smoothing can be used to compute the update. After the update is computed, inter grid communication is carried out (lines 4-19). For each inter grid neighbor of each process assigned to grid ($\mathcal{P}_k$ is the set of processes assigned to grid k), the number of in-flight messages is checked inside `CheckInflight()` based on the rules outlined in the previous paragraph. Note that `MPI_Test()` is used inside `CheckInflight()`. If the check fails, i.e., it is determined that no new message should be sent, $e_{\Omega_i}^{(t_k,k)}$ is saved in `accum_buff[i]`. The notation `[:]` denotes an array or vector operation where

all elements of an array are acted upon. For the operation `accum_buff[i][:] +=` $e_{\Omega_i}^{(t_k,k)}$`[:]`, each element of `accum_buff[i]` is accumulated by the corresponding element of $e_{\Omega_i}^{(t_k,k)}$, where `accum_buff[i]` and $e_{\Omega_i}^{(t_k,k)}$ are the same length. If the in-flight check passes, `accum_buff[i]` is copied into `send_buff[i][j]`, an `MPI_Isend()` is posted, and `accum_buff[i]` is reset to zero. After the send phase is complete, outstanding `MPI_Irecv()` calls are checked for completion. If an `MPI_Irecv()` has completed, $x^{(t_k,k)}$ is updated and a new `MPI_Irecv()` is posted. Finally, the residual $r^{(t_k,k)}$

is computed.

Algorithm 12: Asynchronous multigrid using non-blocking two-sided MPI functions

```
1  while not converged on grid k do
2      e^(t_k,k) = B_k(x^(t_k,k))                ▷ compute update (intra grid communication)
3      x^(t_k,k) = x^(t_k,k) + e^(t_k,k)         ▷ correction of local x with local update
4      for all p ∈ P_k do                        ▷ do inter grid communication for each process assigned to grid k
5          for each inter grid neighbor i of process p do
6              accum_buff[i][:] += e_Ωi^(t_k,k)[:]          ▷ save send data
7              [inflight_flag, j] = CheckInflight(send_requests[i])
8              if inflight_flag then
9                  accum_buff[i][:] = send_buff[i][j][:]    ▷ copy saved data to send buffer
10                 MPI_Isend(send_buff[i][j],..., send_requests[i])   ▷ non-blocking send
11                 accum_buff[i][:] = 0                     ▷ reset buffer to save future send data
12             end
13             MPI_Test(recv_requests[i], &flag,...)        ▷ check completion of receive
14             if flag then
15                 x_Γi^(t_k,k)[:] += recv_buff[i][:]        ▷ update local x
16                 MPI_Irecv(recv_buff[i],..., recv_requests[i])   ▷ non-blocking receive
17             end
18         end
19     end
20     r^(t_k,k) = b − Ax^(t_k,k)                ▷ compute new local residual (intra grid communication)
21     t_k = t_k + 1
22 end
```

For determining convergence, we have two criteria that we consider. For convergence criterion 1, a process p does not stop until it has met some local convergence criterion. For convergence criterion 2, a process p does not stop until it and all its inter grid neighbors have met some local convergence criterion. Unlike convergence criterion 1, this prevents process

p from prematurely stopping when the neighbors of process p still have many updates to perform. Stopping too early when other grids have many more updates to perform can result in slow convergence, which we will see in Section 4.5.2. For convergence criterion 2, the following steps are completed by process p before stopping, where process p is assigned to grid k:

1. Check the local convergence criterion. This occurs when either $\|r^{(t_k,k)}\| < \tau$ or $t_k < t_{\text{max}}$, where τ is some prescribed threshold and t_{max} is the maximum number of updates. This is done using `MPI_Allreduce()` with all processes in $\mathcal{P}_k$.

2. A flag is then appended to all outgoing inter grid messages indicating that process p has met the local convergence criterion.

3. Once process p has received the same flags from all inter grid neighbors, process p communicates this to all processes in $\mathcal{P}_k$ using `MPI_Allreduce()`. In the case that the inter grid neighbors of process p have all met their local convergence criterion before process p, the `MPI_Allreduce()` call can be combined with the residual norm computation. If all processes in $\mathcal{P}_k$ have reached this step, process p has completed and exits the outer solve loop.

If process p is assigned to grid zero and asynchronous smoothing is being used, process p does not check any local convergence criterion and computes updates until all other grids have stopped, i.e., process p only needs to complete step two. For convergence criterion 1, process p simply exits the solve loop after step 2 above. Note that a termination message still needs to be sent to inter grid neighbors in order for all messages to be completed. This is because inter grid neighbors need to know when to stop checking for messages.

4.5.2 Experimental Results

We used the Lassen computer housed at Lawrence Livermore National Laboratory. Each node contains two 22-core IBM Power9 processors and four NVIDIA Tesla V100 GPUs.

The nodes are connected using an EDR InfiniBand interconnect. For our test problems, we used matrices coming from a 27-point centered difference discretization of the Poisson equation, and four matrices from the SuiteSparse matrix collection, as shown in Table 4.2. All of our test problems are SPD. We compared our asynchronous multigrid solvers with the classical multiplicative V(1,1)-cycle implemented in the BoomerAMG package. We also used BoomerAMG to construct the interpolants and coarse grid operators and used PMIS as the coarsening scheme. We used Jacobi and asynchronous Jacobi using the weight $\omega = 1/\lambda_{\max}(D^{-1}A)$, which is often a good choice for smoothing in algebraic multigrid [82]. For each experiment, ten separate runs are carried out.

In all our experiments, we use four MPI processes per node with one thread per MPI process where each MPI processes uses a single GPU for all its computation. This computation includes sparse matrix-vector products and SAXPY operations. We used the single GPU kernels included in Hypre [85], which call cuSPARSE kernels for sparse matrix-vector products and Thrust kernels for SAXPY operations.

Table 4.2: Statistics for the SuiteSparse Matrix Collection test matrices. All matrices are symmetric positive definite.

Matrix	Equations	Non-zeros	Number of Grids
Queen_4147	4,147,110	316,548,962	11
Flan_1565	1,564,794	114,165,372	9
Serena	1,391,349	64,131,971	9
Geo_1438	1,437,960	60,236,322	10

We used a maximum of one in-flight message for our asynchronous solvers in all our experiments, which gave us the best performance. The performance degradation when using more than one maximum number of in-flight messages is due to the cost of packing and unpacking buffers. Table 4.3 shows the wall-clock time and mean number of updates over all grids when increasing the maximum number of in-flight messages for Queen_4147 when using asynchronous Multadd with asynchronous Jacobi. A relative residual norm tolerance of 10^{-6} is used with convergence criterion 2. We can see that as the maximum

number of in-flight messages increases, the number of iterations initially decreases and then increases, indicating that throughput of the communication network is fully utilized when a maximum number of two in-flight messages is used. However, we can also see that as the maximum number of in-flight messages increases, the wall-clock time also increases, where the increase in wall-clock time from packing and unpacking buffers is greater than the performance gain from converging in overall fewer updates. In other words, the time per update increases dramatically as we increase the maximum number of in-flight messages. This result has also been documented in [17] for asynchronous Jacobi.

Table 4.3: Wall-clock time and mean number of updates when varying the maximum number of in-flight messages. The Multadd solver with asynchronous Jacobi smoothing is used. The Queen_4147 problem is being solved. A relative residual norm tolerance of 10^{-6} is used with convergence criterion 2.

Max In-flight	Mean Wall-clock Time	Mean Number of Updates
1	6.48385	193.90
2	6.66103	143.83
3	7.63114	148.47
5	8.29311	167.83
10	11.3593	217.69

Figure 4.9 shows the relative residual 2-norm versus grid length for the 27pt test using 16 nodes (64 MPI processes). Results for both asynchronous convergence criteria are shown. For the asynchronous methods, $t_{\max} = 10$, i.e., the local convergence criterion is met when 10 updates are completed. For Mult, 10 iterations are carried out. Grid length denotes the number of grid points of each side of the three dimensional grid, so the matrix has a number of rows equal to the cubed grid length. The first column of plots shows Mult, asynchronous Multadd (aMultadd), and aMultadd with asynchronous Jacobi (async J) as the smoother on the finest grid. We can see that in the case of convergence criterion 1, where aMultadd does the same amount of computation as in the synchronous Multadd case, we see approximate grid-size independent convergence when synchronous Jacobi is used as the smoother. This means that even if some grids stop updating before others have finished,

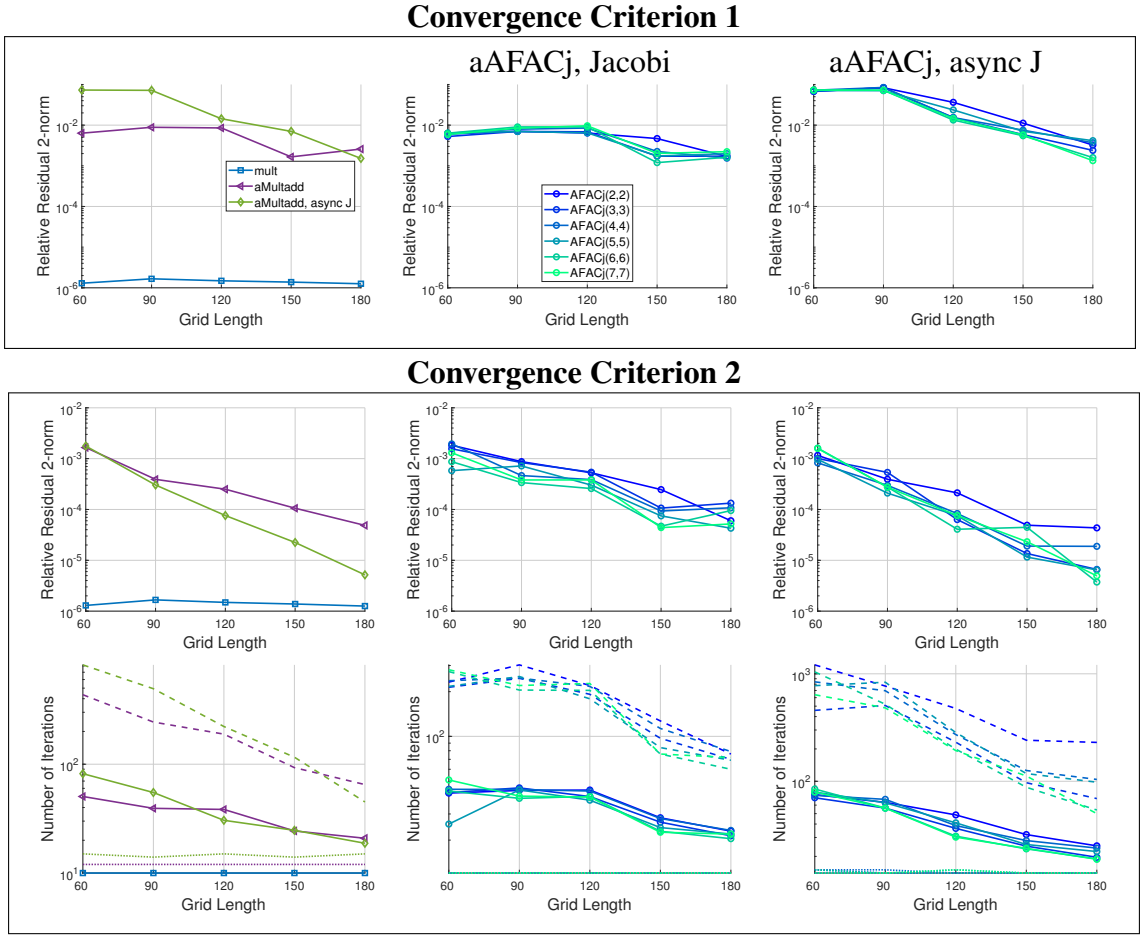


Figure 4.9: Relative residual 2-norm versus grid length for the 27pt test set using 16 nodes (64 MPI processes). The local convergence criterion is $t_{\max} = 10$ for the asynchronous methods. For Mult, 10 iterations are carried out. The first column of plots shows Mult, asynchronous Multadd (aMultadd), and aMultadd with asynchronous Jacobi as the smoother on the finest grid. The second and third columns of plots show asynchronous AFACj (aAFACj) and aAFACj with asynchronous Jacobi (async J) as the smoother on the finest grid, respectively. The blue-to-green gradient denotes an increasing number of interpolants used in AFACj. Results for both asynchronous convergence criteria are shown. The last row of plots shows the mean number of updates over all grids when using convergence criterion 2, where the dashed line denotes the maximum number of updates (i.e., the number of updates carried out by the fastest grid), and the dotted line denotes the minimum number of updates (i.e., the number of updates carried out by the slowest grid).

we can still observe grid-size independent convergence. However, using convergence criterion 1 results in a method that converges more slowly than in the case of convergence criterion 2 since the faster grids do not carry out additional updates.

In the case of asynchronous Jacobi, the convergence rate actually improves as the grid

length increases. This increase in convergence rate is due to the fact that 16 nodes are used for all grid lengths. Since the number of grids increases as the grid length increases, the number of processes assigned to grid 0 (the only grid in which asynchronous smoothing is carried out) decreases, which changes the behavior of the asynchronous smoother. In other words, the asynchronous smoother becomes more synchronous as the grid length increases since fewer processes are smoothing asynchronously. This indicates that using a synchronous smoother within asynchronous multigrid is the best choice, i.e., synchronous Jacobi has better smoothing properties than asynchronous Jacobi in this case.

In the case of convergence criterion 2, we can see that the convergence rate of asynchronous multigrid increases with increasing grid length. The reason for this is shown in the last row of plots, where the mean (solid line), maximum (dashed line) and minimum (dotted line) number of updates for the asynchronous methods is shown. In other words, the mean is the average number of updates over all grids, the maximum is the number of updates of the fastest grid and the minimum is the number of updates of the slowest grid. We can see that as we increase the grid length, the mean and maximum number of updates decreases. This is because the number of grids is increasing allowing for more updates to be overlapped with the updates of the slowest grid. For example, consider just two grids, where grid 0 updates faster than grid 1. Even if grid 0 updates far more than grid 1, grid 0 eventually will need information from grid 1 in order for the overall iteration to converge quickly. This is because the smoothing sweeps from grid 0 will eventually have little effect on reducing the error. If we now consider 10 grids where grid 9 is the slowest, grid 0 may update faster than other grids, but will receive information from grid 1-8 before receiving information from grid 9. This results in a faster solver than in the two-grid case since neglecting grid 9 still results in a multilevel solver with 9 grids. This indicates that asynchronous multigrid is useful when many grids are used.

The second and third columns of plots show asynchronous AFACj (aAFACj) and aAFACj async J, respectively. The blue-to-green gradient denotes an increasing number

of interpolants used in AFACj. We can see that in all cases, aAFACj converges with a rate independent of the grid size. Additionally, AFACj$(1,1)$ converges with a rate similar to that of AFACj$(6,6)$ which indicates that using fewer smoothed interpolants does not significantly degrade the convergence rate. Lastly, we can see that asynchronous smoothing is beneficial in this case. Specifically, all asynchronous multigrid methods require a fewer number of updates on average when using asynchronous Jacobi as a smoother.

Figure 4.10 shows strong scaling for the matrices from Table 4.2. A relative residual 2-norm tolerance of 10^{-6} is used with convergence criterion 2. For the asynchronous methods, we again we show the mean, maximum and minimum number of updates. Unlike in the case of our 27pt problem, we can see that Multadd with asynchronous smoothing is the slowest method in terms of wall-clock time in all cases since it generally requires more updates on average to converge. In terms of wall-clock time, we can see that aAFACj is the best asynchronous multigrid method . We can also see that in general, the asynchronous multigrid methods scale better than Mult. More precisely, the majority of the time, we see that the wall-clock time for asynchronous multigrid monotonically decreases with increasing numbers of nodes, where as in the case of Mult, the wall-clock time generally increases. However, since asynchronous multigrid has a higher computational cost, Mult is significantly faster than all the asynchronous multigrid methods when the number of nodes is low. As the number of nodes increases, communication costs tend to outweigh computational costs, which is why asynchronous multigrid scales so well. This is also why we see the maximum and minimum number of updates increase and decrease, respectively, with increasing concurrency. In other words, as concurrency increases, the problem size per process decreases, allowing for the faster grids to update more frequently.

In the case of Serena and Geo_1438, while aAFACj is more scalable, aAFACj does not outperform Mult when comparing the best timings (Mult at 8 nodes compared with aFACj at 128 nodes). More precisely, aAFACj is $\approx .99$ as fast as Mult for Geo_1438 and $\approx .98$ as fast as as Mult for Serena. However, for Queen_4147 and Flan_1565, aAFACj is ≈ 1.18

faster than Mult for Flan_1565 and ≈ 2.75 faster than Mult for Queen_4147. One reason we see a higher speedup for Queen_4147 compared with other problems is because of the convergence rate of aAFACj. We can see that unlike the other problems, aAFACj requires fewer iterations on average than Mult. In other words, the average number of updates by aAFACj is less than the number of iterations by Mult. This indicates that many updates by faster grids can actually accelerate the convergence rate of asynchronous multigrid for certain problems. This is especially important for a higher level of parallelism than what is considered here, where some grids may be much faster than others.

Note that we only show aAFACj(1,1) since we found it to be the best method for all four matrices from Table 4.2. Table 4.4 shows the wall-clock time and mean number of updates for Queen_4147 using 128 nodes. The table shows results for aAFACj(1,1) up to aAFACj(6,6) and Multadd. Jacobi smoothing and a relative residual 2-norm tolerance of 10^{-6} are used. We can see that even though aAFACj(1,1) requires more updates on average to converge than aAFACj(6,6), aAFACj(1,1) still converges faster in terms of wall-clock time. This indicates that the reduction in computation of aAFACj(1,1) over aAFACj(6,6) and Multadd more than makes up for the higher number of updates required for aAFACj(1,1) to converge.

Table 4.4: Effect on wall-clock time and number of iterations for aAFACj when varying the number of smoothed interpolants. The Queen_4147 problem is being solved. A relative residual norm tolerance of 10^{-6} is used with convergence criterion 2.

	Wall-clock Time	Mean Number of Updates
aAFACj(1,1)	4.37	224.55
aAFACj(2,2)	4.52	215.46
aAFACj(3,3)	4.82	191.64
aAFACj(4,4)	5.02	181.00
aAFACj(5,5)	5.31	183.36
aAFACj(6,6)	5.41	180.82
aMultadd	5.95	185.00

4.6 Conclusion

In this chapter, we introduced asynchronous multigrid methods. These methods are asynchronous versions of additive multigrid methods. Although we have used the familiar term "V-cycle" in these methods to mean one set of corrections from every grid in the multigrid hierarchy, there is no concept of a cycle in asynchronous additive multigrid methods: updates from all grids are computedsimultaneously and do not wait for each other. Our models and experiments show that grid-independent convergence can be retained in this asynchronous setting. However, in our simulations and experimental tests, the number of corrections from each grid is approximately balanced. It is possible to show that if the number of corrections is not balanced (e.g., far more corrections from some grids compared to others), then grid-independent convergence is lost.

We presented the AFACj additive multigrid method which is an improvement over additive variants of classical multiplicative multigrid methods (Multadd). In Multadd, one of the primary computational costs comes from the fact that smoothed interpolants (smoothing iteration matrix applied to a standard interpolant) are used. In AFACj, we use fewer smoothed interpolants to reduce computation costs without significantly the degrading the convergence rate, giving us an overall faster method in terms of wall-clock time.

In the shared memory case, we showed that asynchronous Multadd can be faster (in terms of wall-clock time) than the classical synchronous multiplicative method when the problem size per thread is sufficiently small. Additionally, we showed that an asynchronous smoother is the best choice in smoother, even when using just one smoothing sweep. In the distributed memory case, non-blocking two-sided MPI functions are used to communicate updates between processes assigned to different grids. We test our asynchronous multigrid solvers on up to 128 nodes of a GPU cluster (512 total GPUs). We show that asynchronous multigrid exhibits convergence independent of the grid size. We also provide strong scaling results that show that asynchronous multigrid often scales better than synchronous multi-

grid. For the Queen_4147 test problem, when comparing asynchronous AFACj to classical multigrid, we observed a peak speedup of ≈ 2.75.

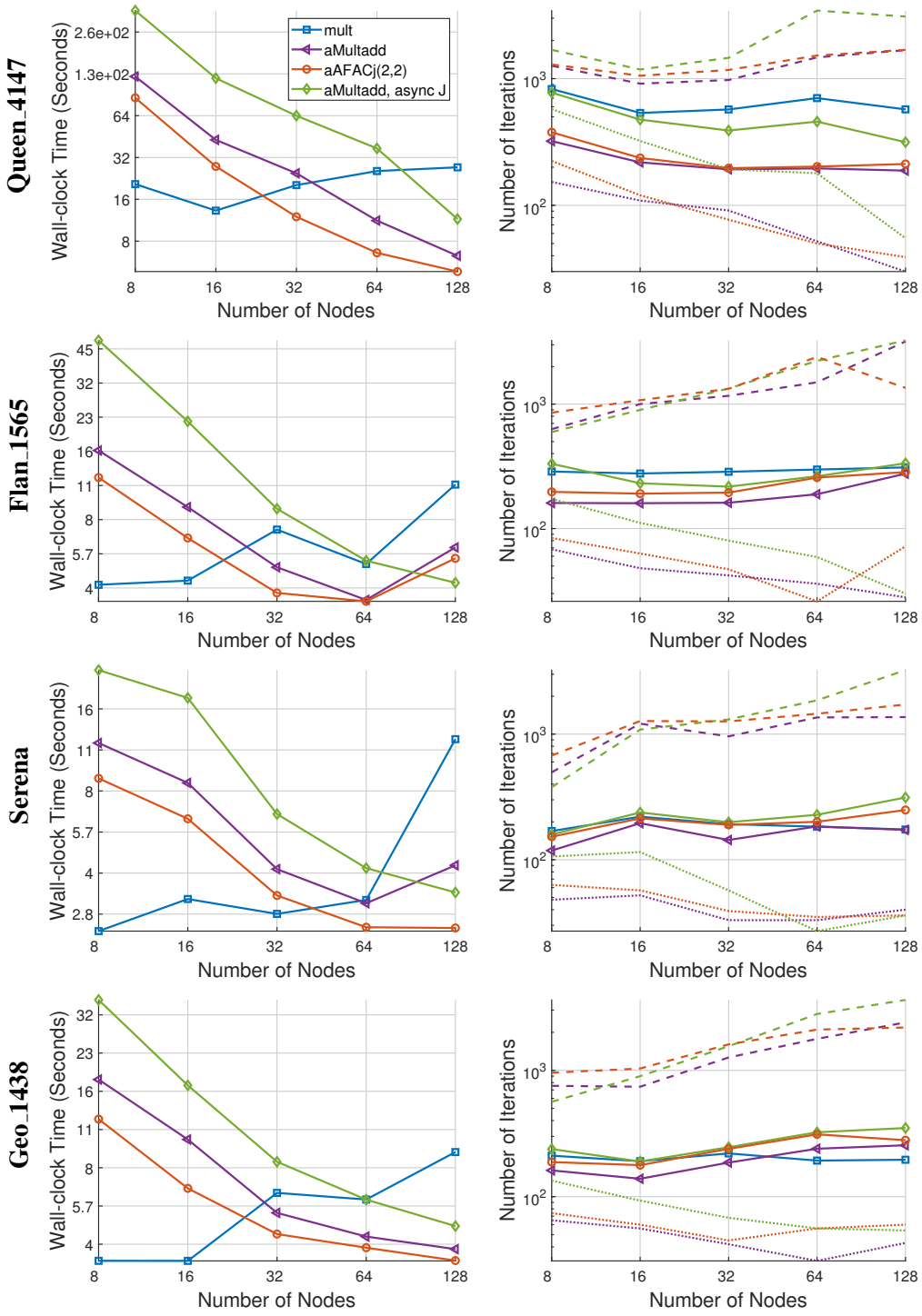


Figure 4.10: Strong scaling results for the test problems from Table 4.2. Wall-clock time and iterations versus number of nodes is shown. A relative residual 2-norm tolerance of 10^{-6} is used with convergence criterion 2. For the asynchronous methods, iterations denotes the mean number of updates over all grids is shown when using convergence criterion 2, where the dashed line denotes the maximum number of updates (i.e., the number of updates carried out by the fastest grid), and the dotted line denotes the minimum number of updates (i.e., the number of updates carried out by the slowest grid).

CHAPTER 5
ASYNCHRONOUS CHEBYSHEV METHODS

In this chapter, we introduce the first descriptions and implementations of an asynchronous version of the Chebyshev iterative method [23]. The Chebyshev iterative method for symmetric systems of equations has a convergence rate similar to that of the celebrated conjugate gradient (CG) method. Indeed, the nonsymmetric Chebyshev iterative method was actively promoted into the 1980s [86] and was only displaced by the development of GMRES [87] and other iterative solvers for nonsymmetric systems (we will only consider the Chebyshev method for symmetric positive definite (SPD) systems in this paper). The main disadvantage of the Chebyshev iterative method is that estimates of the extremal eigenvalues of the system matrix are required. This issue can be easily alleviated but does make the Chebyshev method more complicated to use than CG and GMRES.

The Chebyshev iterative method, however, has a key advantage that makes it relevant to high-performance computing: unlike CG and GMRES, the Chebyshev iterative method does not require any inner products. Inner products, which depend on global communication, scale poorly on parallel machines. Scaling is poor not only because all processes are involved in communication, but any system noise or load imbalance can dramatically increase the synchronization cost [88].

The inner products in CG and GMRES cannot be performed asynchronously, otherwise the orthogonality and other mathematical properties on which these methods depend are no longer satisfied. Thus it is unlikely that there will ever be asynchronous versions of CG and GMRES (in the sense of processes proceeding with the latest available data, as used in this paper). The Chebyshev iterative method, like the Jacobi iterative method, performs communication but does not require any global communication, and thus is a candidate for the development of an asynchronous version to relieve synchronization pressure in the

communication phases.

The mathematical theory for asynchronous iterative methods only applies to fixed-point iterations like the Jacobi method. The Chebyshev iterative method, like CG, uses different parameters at each iteration and thus is not a fixed-point iteration. Although there is presently no mathematical theory for an asynchronous version of the Chebyshev iterative method, one may be optimistic because theory does exist for the "inexact" Chebyshev iterative method [23] where certain operations such as residual computations, are performed inexactly. This inexact method has some analogy to the asynchronous Chebyshev method proposed here in the sense that residual vectors computed using stale values could be considered inexact computations.

The Chebyshev iterative method, like CG and GMRES, can be preconditioned using an approximation of the system matrix called a preconditioner. This is significant because an asynchronous Chebyshev method allows other asynchronous iterative methods, like asynchronous Jacobi, to be used as a preconditioner for the first time. Previously, it would be pointless to use asynchronous Jacobi as a preconditioner for CG or GMRES, since these methods have multiple synchronization points at each iteration. As an example, multigrid and domain decomposition methods are often used as preconditioners inside a CG or GMRES method. The recently developed asynchronous versions of domain decomposition solvers [20] can now be used as preconditioners inside our proposed asynchronous Chebyshev method.

In this chapter, we demonstrate asynchronous Chebyshev with a new asynchronous multigrid preconditioner. Instead of the asynchronous multigrid method proposed in Chapter 4, we are able to use a much simpler multigrid method called BPX [24] (named after the initials of its three authors). The asynchronous multigrid method proposed earlier had to be based on a method that could converge on its own, i.e., as a standalone solver. When multigrid is used as a preconditioner, standalone convergence is not necessary, which allows us to use BPX, the simplest additive multigrid method. To create an asynchronous

version of BPX, we use a formulation of the method based on an extended matrix [89], to be described later in this paper. The standard formulation contains many types of operations (smoothing, restriction, prolongation, coarse grid solves) and it is very complicated to run these operations asynchronously with each other. In contrast, the extended matrix formulation is similar to using Jacobi iterations with a special "extended" matrix, which are easy to perform asynchronously.

It should be noted that we have found the performance of the asynchronous Chebyshev method to be sensitive to the preconditioner. If the preconditioner is poor, the asynchronous computations can cause the overall iterative method to diverge. We do not observe this when the preconditioner is good, i.e., a good approximation to the system matrix, and this concept is supported by the theory of the inexact Chebyshev method. In particular, the Jacobi preconditioner for discretizations of partial differential equations becomes poorer as the discretization becomes finer and the problem size increases. We have observed that asynchronous Chebyshev with this preconditioner fails to converge for large problem sizes. On the other hand, the BPX preconditioner, by design, is a good approximation to the system matrix for all problem sizes. In this case, we observe that asynchronous Chebyshev with this preconditioner converges for all problem sizes in our test examples. We therefore propose asynchronous Chebyshev combined with asynchronous BPX as a solver in contexts where asynchronous computations are important for performance.

5.1 Background

5.1.1 The Chebyshev Method

Consider again an iterative method with an iteration matrix of the form $I - M^{-1}A$, where M is some preconditioner, e.g., the Jacobi or BPX preconditioner. The Chebyshev method [23, 87, 90] is derived by finding the degree t polynomial of $I - M^{-1}A$ with minimum spectral radius over the interval $[\alpha, \beta]$, where α and β are the smallest and largest eigenvalues of

$M^{-1}A$, respectively. More precisely, we seek the degree t polynomial p such that

$$\min_{p\in\mathbb{P}_t, p(\gamma)=1} \max_{\lambda\in[\alpha,\beta]} |p(\lambda)|, \tag{5.1}$$

where $\mathbb{P}_t$ is the space of polynomials of degree $\leq t$. The degree t Chebyshev polynomial of the first kind satisfies this requirement. Using the recursive definition of the Chebyshev polynomials, we can write the Chebyshev method as

$$\begin{aligned} x^{(1)} &= x^{(0)} + \delta M^{-1} r^{(t)}, \\ x^{(t+1)} &= x^{(t-1)} + \omega^{(t)}\left(\delta M^{-1} r^{(t)} + x^{(t)} - x^{(t-1)}\right), \\ \omega^{(t+1)} &= 1/\left(1 - \omega^{(t)}/4\mu^2\right), \end{aligned} \tag{5.2}$$

where $\omega^{(1)} = 2$, $\delta = \frac{2}{\beta+\alpha}$, and $\mu = \frac{\beta+\alpha}{\beta-\alpha}$.

It can be shown that the classical upper bound on the A-norm of the error for Chebyshev is the same as that of the Conjugate Gradient method [90]. Additionally, Chebyshev can be used on non-symmetric problems [86], and unlike GMRES, no restarting is needed. The advantage of Chebyshev over Krylov subspace methods is that no inner products are required. Inner products are expensive at scale, but more importantly, we need an inner product-free method in order to develop an asynchronous method. This is because it is unclear how to compute inner products asynchronously without ruining the orthogonality properties in Krylov subspace methods. However, Chebyshev does require the estimates of the largest and smallest eigenvalues of $M^{-1}A$, which can be done while also carrying out an iterative solver. In other words, we can start by carrying out a few iterations of preconditioned Conjugate Gradient and then switch to asynchronous Chebyshev once we have good eigenvalue estimates. In [86], an adaptive Chebyshev method is introduced where eigenvalue estimates are continually improved while Chebyshev iterates.

5.2 Asynchronous Chebyshev Methods

5.2.1 Models of the Jacobi-preconditioned Asynchronous Chebyshev Method

We start by defining Jacobi-preconditioned Asynchronous Chebyshev through a mathematical model similar to Equation 2.5. For simplicity, let the number of processes equal the number of rows. First, note that Equation 5.2 can be written as the iteration

$$\begin{bmatrix} x^{(t+1)} \\ y^{(t+1)} \end{bmatrix} = \begin{bmatrix} \omega^{(t)}(I - \delta M^{-1}A) & (1-\omega^{(t)})I \\ O & I \end{bmatrix} \begin{bmatrix} x^{(t)} \\ y^{(t)} \end{bmatrix} + \begin{bmatrix} \omega^{(t)}\delta M^{-1}b \\ 0 \end{bmatrix}. \tag{5.3}$$

We can model a special case of running this iteration asynchronously using the Jacobi preconditioner:

$$\begin{aligned}
&\text{If } i \in \Psi(t)\text{:} \\
&\quad \omega_i^{(t+1)} = 1/(1 - \omega_i^{(t)}/4\mu^2), \\
&\quad x_i^{(t+1)} = y_i^{(t)} + \omega_i^{(t+1)} \left(f_i - \delta \sum_{j \in S(A_i)} A_{ij} x_j^{(z_{ij}(t))}/A_{ii} + x_i^{(t)} - y_i^{(t)} \right), \\
&\quad y_i^{(t+1)} = x_i^{(t)}, \\
&\text{Else:} \\
&\quad x_i^{(t+1)} = x_i^{(t)}, \\
&\quad y_i^{(t+1)} = y_i^{(t)}, \\
&\quad \omega_i^{(t+1)} = \omega_i^{(t)},
\end{aligned} \tag{5.4}$$

where $f_i = \delta b_i/A_{ii}$. As in Equation 2.5, we have the mappings $z_{ij}(t)$ and the set $\Psi(t)$. We say that this is a special case since Equation 5.3 tells us that we have two vectors that are asynchronously propagated: $x^{(t)}$ and $y^{(t)}$. In other words, if we had, for example, n additional processes, these processes could be used to update $y^{(t)}$. However, when simulating this, we found that using n processes and having process i update both $x_i^{(t)}$ and $y_i^{(t)}$ resulted

in a convergent method. We also see that each $\omega_i^{(t)}$ is different in general on each process which means there are actually three variables that are propagated asynchronously. While we could have a single thread that updates all $\omega_i^{(t)}$ values, we found that the best convergence behavior is achieved when process i updates $\omega_i^{(t)}$ (as in the case of $y_i^{(t)}$), which is what is modeled in Equation 5.4.

We can also think of asynchronous preconditioned Chebyshev as inexactly applying the preconditioner each iteration [23]. Let the equation for applying the preconditioner be $Mz = r^{(t)}$, where we are solving for z. From [23], if the relative residual norm in the inexact solve is reduced to one or less, then Chebyshev converges. In other words, if $\|Mz - r^{(t)}\|/\|r^{(t)}\| \leq \gamma$, where $\gamma \in (0, 1)$, then Chebyshev converges. While we do not prove convergence for our models of asynchronous Chebyshev, the analysis from [23] gives us some intuition as to why asynchronous Chebyshev could converge in the first place.

5.2.2 Asynchronous EBPX-Chebyshev Method

Our goal is to develop a fast-converging asynchronous Chebyshev method. When we experimented with Jacobi-preconditioned asynchronous Chebyshev, we observed divergence as the problem size increased (see Section 5.3). Therefore, we looked to multigrid methods, which converge independent of the problem size, but are less straightforward than Jacobi to execute asynchronously.

Algorithm 13 shows the algorithm for Jacobi-preconditioned asynchronous Chebyshev on a shared memory machine. Again, for simplicity, let the number of threads equal the number of rows. In the general case, each thread owns multiple rows, so $\sum_{j \in S(A_i)} A_{ij} x_j$ in line 5 would be replaced with a matrix-vector product. In the algorithm, thread i reads data from memory (x_j values that are needed for the relaxation of x_i), updates all variables needed for the relaxation of x_j, and writes x_i to shared memory. Note that x_j could also be read inside the summation $\sum_{j \in S(A_i)}$ instead of reading all x_j values beforehand, which may increase the convergence rate since newer information will be used. However, if process

i owns multiple rows that have the same column indices, the amount of data re-use will decrease.

Algorithm 13: Jacobi-preconditioned Asynchronous Chebyshev

while *not converged on thread* i **do**
  Read x_j for $j \in S(A_i)$ from shared memory
  $\omega = 1/(1 - \omega/4\mu^2)$
  $z = x_i$
  $x_i = y + \omega\left(f_i - \delta \sum_{j \in S(A_i)} A_{ij}x_j/A_{ii} + x_i^{(t)} - y\right)$
  $y = z$
  Write x_i to shared memory
end

If we can express multigrid in relaxation form without a substantial increase in setup costs, we can then naturally precondition Chebyshev using multigrid. First consider the extended block system $\mathcal{A}\mathbf{u} = \mathbf{g}$, as discussed in [89], where $\mathcal{A}$ is symmetric positive semi-definite. The diagonal blocks of $\mathcal{A}$ are the A_k matrices, and the off-diagonal blocks denote the connections between grids via the prolongation and restriction matrices. For example, if we consider just two grids, the block system has the form

$$\begin{bmatrix} A_0 & A_0P_1^0 \\ (P_1^0)^T A_0 & A_1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \end{bmatrix} = \begin{bmatrix} b \\ (P_1^0)^T b \end{bmatrix}. \tag{5.5}$$

In the general case where we have ℓ grids, we have

$$\begin{bmatrix} A_0 & A_0P_1^0 & \cdots & A_0P_{\ell-1}^0 \\ \vdots & A_1 & \cdots & A_1P_{\ell-1}^1 \\ \vdots & \ddots & \ddots & \vdots \\ \cdots & \cdots & \cdots & A_{\ell-1} \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{\ell-1} \end{bmatrix} = \begin{bmatrix} b \\ (P_1^0)^T b \\ \vdots \\ (P_{\ell-1}^0)^T b \end{bmatrix} \tag{5.6}$$

In [89], the authors show that carrying out Gauss-Seidel on this block system is equivalent to classical multiplicative multigrid on the original system $Ax = b$, where the solution

is recovered as $u_0 + P_1^0 u_1 + \cdots + P_{\ell-1}^0 u_{\ell-1}$. Similarly, carrying out Jacobi on this block system is equivalent to BPX. We will call Jacobi on this extended system EBPX. For the experimental results in this chapter, the extended system is explicitly constructed. We then simply run Algorithm 13 on the system. The down side of this method is that the extended system needs to be explicitly formed, which requires additional products of interpolants during the setup phase. If the interpolants are not sparse enough, this cost could be high, especially at scale. In a distributed memory environment, we could consider an implicit method where the off-diagonal row blocks are not explicit constructed. These blocks would be applied to vectors via multiple matrix-vector products. However, this would require synchronization between subsets of processes, so the scheme would not be fully asynchronous.

5.2.3 Simulating Models of Asynchronous EBPX-Chebyshev

While there are not yet any results on convergence for our models of asynchronous Chebyshev, we have simulated asynchronous EBPX-Chebyshev to understand its general behavior. In these simulations, we consider the highly-concurrent case where we have a single process per row. There are two parameters that must be chosen for each row i: the *update probability* p_i and the *read delay bound* d_i. The update probability denotes the probability that $i \in \Psi(t)$, i.e., p_i is a random number in the range $[p_{\text{min}}, 1)$, where $1 > p_{\text{min}} > 0$ is the minimum update probability. The purpose of this parameter selection is to simulate the case where some processes update more often than others. As we decrease p_{min}, the simulation becomes more asynchronous in the sense that some processes update a lot more often than others. The read delay bound denotes the number of previous time instants that i is allowed to read from, i.e., d_i limits how far back in time i can read. As in the case of the update probability, d_i is chosen randomly, but from the range of integers $[0, d_{\text{max}}]$. For example, if d_i is 10, i cannot read farther back than $t - 10$, where t is the current time instant. Additionally, if t_i is that last time instant from which i read data from, i cannot later read from a time instant that is $< t_i$. The read delay parameter is meant to simulate

communication delays.

While we do not show this in our results, we also simulated the scenario where y is also asynchronously propagated, as discussed in Section 5.2.1. This could correspond to adding up to n additional processes that update y. In these simulations, the method diverged in general. This scenario is also not practical since it is inexpensive for the threads that update x to also update y and ω, as in Equation 5.4..

Figure 5.1 shows some results for our simulations, where we simulated the iteration in Equation 5.4. Each figure denotes a different minimum update probability. Our convergence detection is outlined in Section 5.3 and is not different here. In short, the simulation does not stop until all rows have been relaxed at least 50 times. This means that some rows may be relaxed many more times than others The problem being solved is the five-point centered-difference discretization of the 2D Laplace equation on a 64×64 grid.

The important result in Figure 5.1 is that asynchronous EBPX-Chebyshev converges, even in the most asynchronous case of $p_{\text{min}} = .1$ and $d_{\text{min}} = 1000$. We also see that in the best case scenario where $p_{\text{min}} = .8$ and $d_{\text{min}} = 0$, which could simulate a shared memory environment with fast communication, asynchronous EBPX-Chebyshev converges in fewer relaxations. In general, we see that asynchronous EBPX-Chebyshev converges in fewer relaxations. We also see that in some cases asynchronous EBPX-Chebyshev achieves the lowest relative residual 2-norm, indicating that asynchronous EBPX-Chebyshev would converge faster in terms of wall-clock time in these situations. This is because the slowest process carries out only 50 relaxations, so the additional relaxations carried out by other processes are overlapped with the time it takes for the slow process to finish.

5.3 Experimental Results

For our experimental results, we used a node with two Intel Xeon E5-2695 v4 18-core CPUs. We implemented both our synchronous and asynchronous methods using parallel

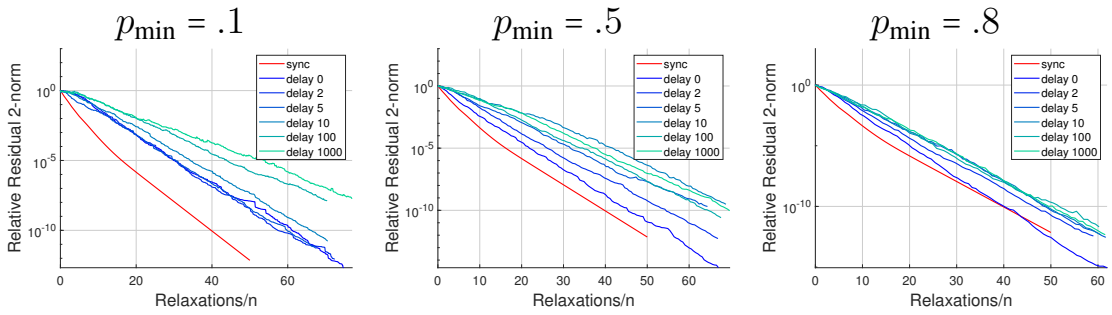


Figure 5.1: Relative residual 2-norm versus number of relaxations/n for a simulation of asynchronous EBPX-Chebyshev. The problem being solved is the five-point centered-difference discretization of the 2D Laplace equation on a 64×64 grid. From left to right, the minimum update probability of a row being relaxed is increased. The blue to green gradient denotes an increasing bound on the read delay. Note that these plots demonstrate convergence over a wide range of asynchronous conditions and do not imply that one case is more rapid than another.

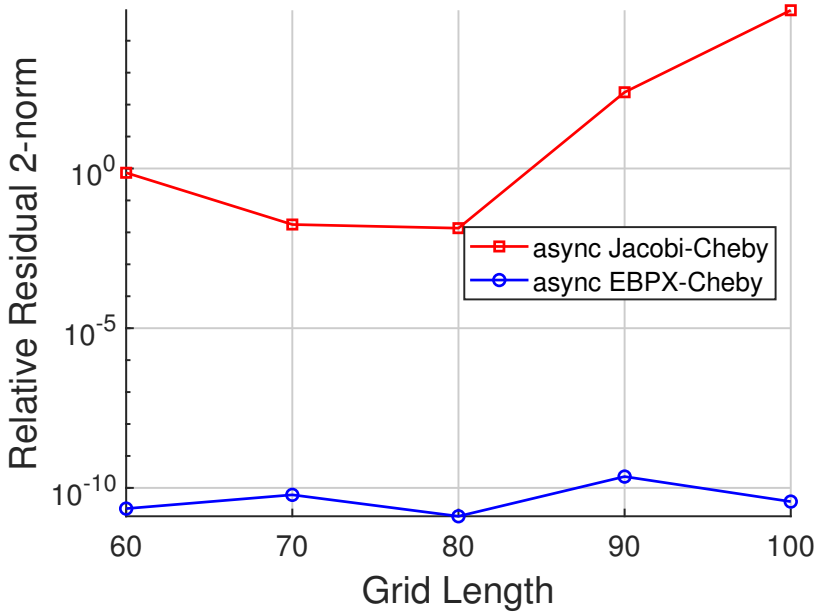


Figure 5.2: Relative residual 2-norm after 50 asynchronous iterations versus grid length for the 27pt problem using 36 threads. Jacobi-preconditioned asynchronous Chebyshev is compared with EBPX-preconditioned asynchronous Chebyshev. Jacobi-preconditioned asynchronous Chebyshev diverges for larger problems where as EBPX-preconditioned asynchronous Chebyshev converges with a rate independent of the grid size.

for loops, where a *nowait* clause is used in the asynchronous case. We used a compact thread affinity, which gave us the best performance. We generated the Galerkin coarse grids and interpolants using the BoomerAMG package [85].

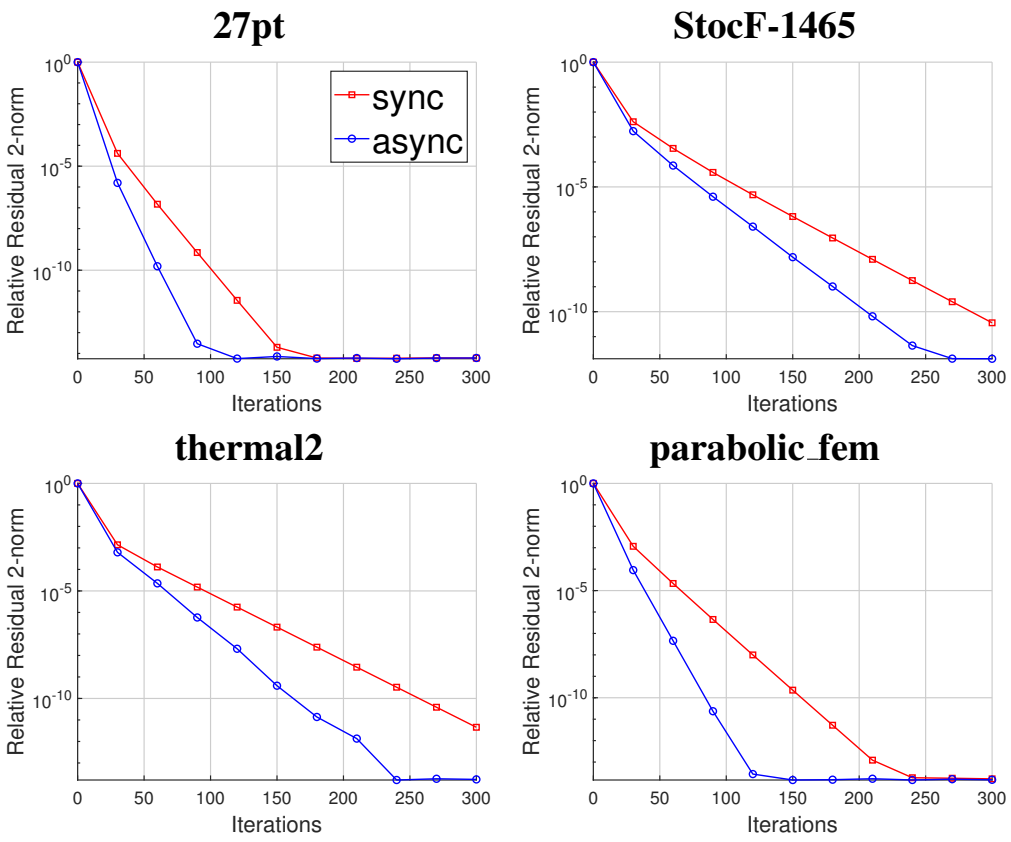


Figure 5.3: Relative residual 2-norm versus number of iterations using 36 threads for four test problems. Synchronous and asynchronous EBPX-Chebyshev are compared, where we can see asynchronous EBPX-Chebyshev has a faster convergence rate.

In the asynchronous case, for convergence testing, a thread i computes the residual norm of the rows that i relaxes. This partial residual norm is computed every iteration. That partial residual norm is then written to shared memory along with the number of relaxations that i has carried out. A master thread, in this case thread 0, checks the following two things:

1. Periodically sum up all the partial residual norms to see if the global residual norm has dropped below some prescribed tolerance τ. Note that this global residual norm is an approximation since there is no global iteration counter.

2. Check to see if the minimum number of relaxations over all rows is above some threshold σ. In other words, all threads update until the slowest thread has completed

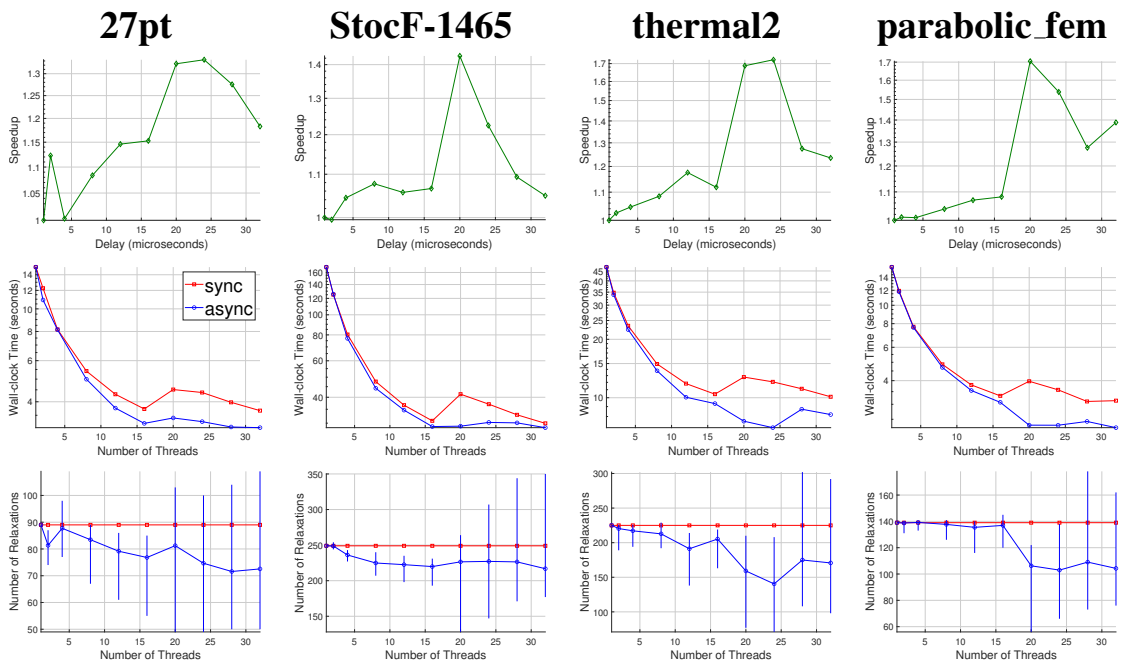


Figure 5.4: Strong scaling results for four test problems (columns). The residual norm is computed at each iteration with $\tau = 10^{-9}$. The first row shows the speedup, defined as the wall-clock time of synchronous Chebyshev divided by the wall-clock time of asynchronous Chebyshev. The second row shows the solve wall-clock time. The third row shows the number of relaxations. In the asynchronous case, each thread finishes having carried out a different number of relaxations. Therefore, each data point is the mean number of relaxations and the error bar denotes the minimum and maximum number of relaxations.

σ relaxations.

If either of these conditions are met, the master thread sets a flag. All threads will periodically read this flag and terminate once it has been set. We say that we have done σ *asynchronous iterations* if the second criterion is met. It is important to note that σ is not the same in general as a *time instant* (Equations 2.5 and 5.4). In the synchronous case, the residual norm is also computed after each iteration. Just like in the asynchronous case, each thread computes a partial residual norm. A parallel **for** loop with a reduction clause is then used to sum all the partial residual norms. For a fair comparison between synchronous and asynchronous, if the asynchronous iteration stops due to condition 1) above, we made sure that the final relative residual norm for asynchronous was always $\leq$ the final relative residual norm for synchronous.

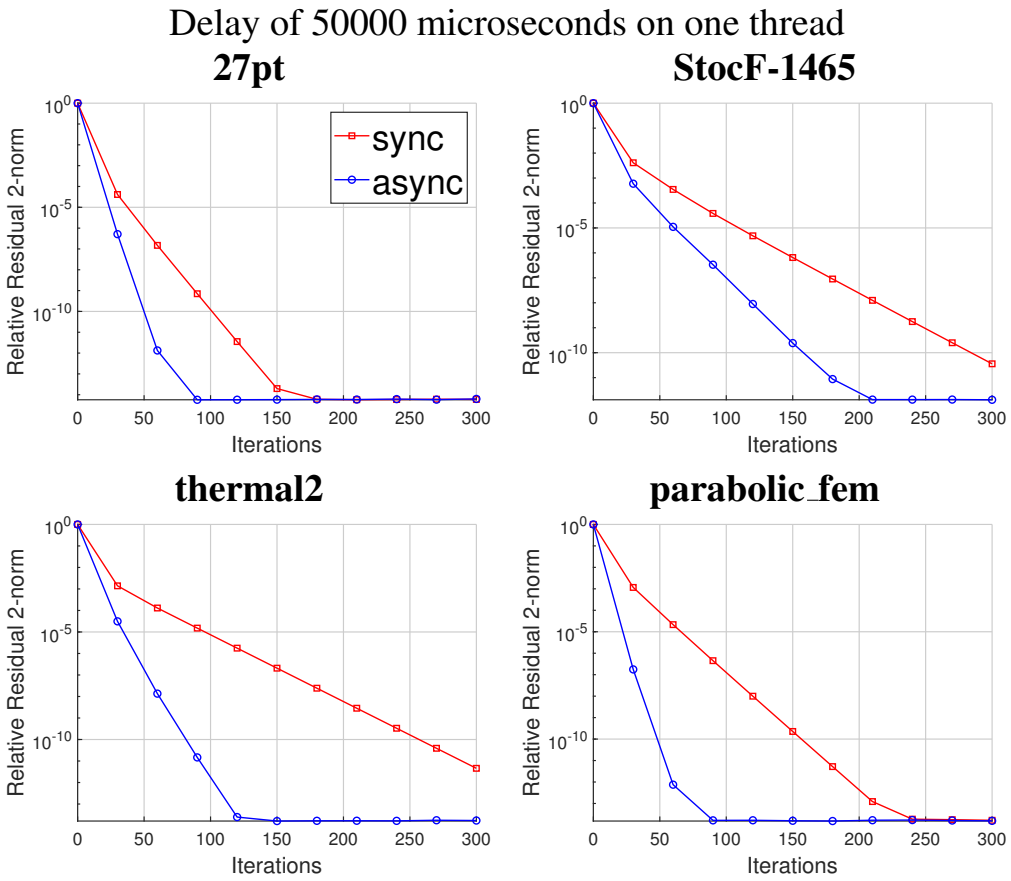


Figure 5.5: Relative residual 2-norm versus number of iterations using 36 threads for four test problems. Thread 18 has an artificial delay where it sleeps for 50000 microseconds every iteration. Synchronous and asynchronous EBPX-Chebyshev are compared, where we can see that asynchronous EBPX-Chebyshev has a faster convergence rate. Additionally, the convergence rate of asynchronous EBPX-Chebyshev is significantly faster than in the un-delayed case.

Jacobi converges poorly when compared to multigrid, which converges with a rate independent of the problem size. When integrating this into asynchronous Chebyshev, we have observed asynchronous Jacobi-Chebyshev to diverge as we increase the problem size. Figure 5.2 demonstrates this behavior, where we compare asynchronous Jacobi-Chebyshev (Chebyshev with the Jacobi preconditioner) to asynchronous EBPX-Chebyshev. The problem being solved is the 27-point discretization of the 3D Laplace equation (we will refer to this as 27pt). The figure shows the relative residual 2-norm versus grid length (the global problem size is the grid length cubed) after 50 asynchronous iterations using 36 threads.

The figure shows that asynchronous Jacobi-Chebyshev starts to diverge for larger problem sizes where as asynchronous EBPX-Chebyshev exhibits grid-size independent convergence. We can see this for grid lengths 90 and 100 where asynchronous Jacobi-Chebyshev diverges.

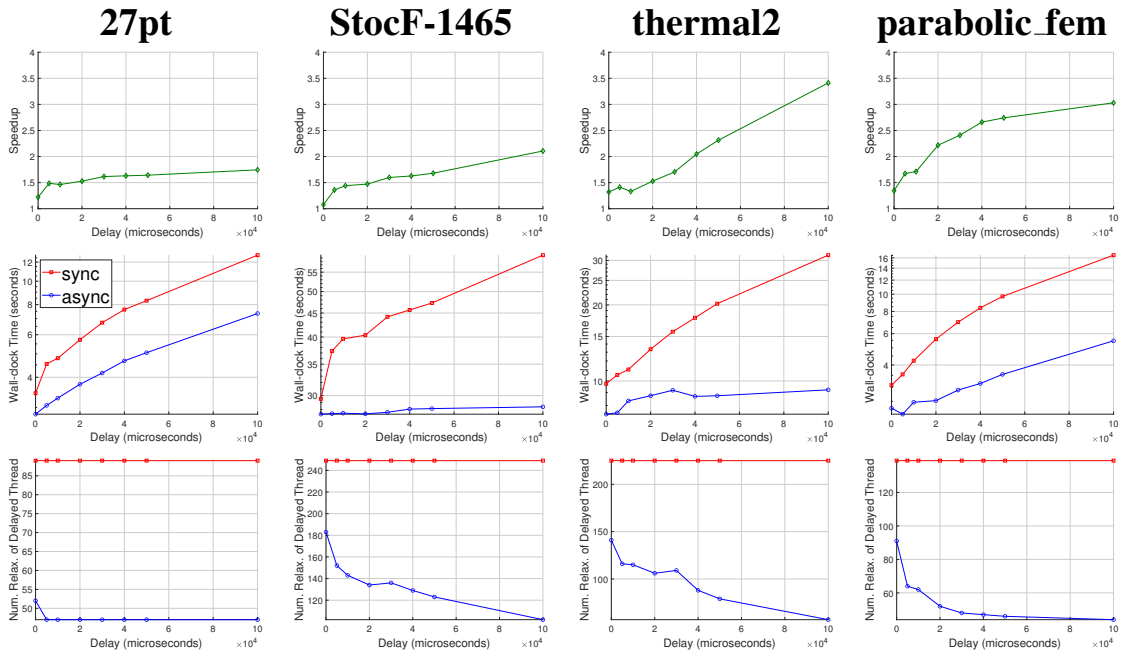


Figure 5.6: Results for varying an artificial delay on a single thread using 36 total threads. The columns denote the test problems. The residual norm is computed at each iteration with $\tau = 10^{-9}$. The first row shows the speedup, defined as the wall-clock time of synchronous Chebyshev divided by the wall-clock time of asynchronous Chebyshev. The second row shows the solve wall-clock time. The third row shows the number of relaxations of the delayed threads.

For the remaining experiments, we will be solving systems with the SPD matrices shown in Table 5.1, three of which are taken from the SuiteSparse matrix collection. Figure 5.3 shows the relative residual 2-norm versus number of iterations. In the asynchronous case, we are showing asynchronous iterations on the x-axis (explained earlier in this section). Since we cannot precisely track the residual norm versus number of asynchronous iterations while the asynchronous method is iterating, each data point represents a separate run. In other words, we first run the code for 30 asynchronous iterations, then reset and run for 60 asynchronous iterations, and repeat this process all the way up to 300 asyn-

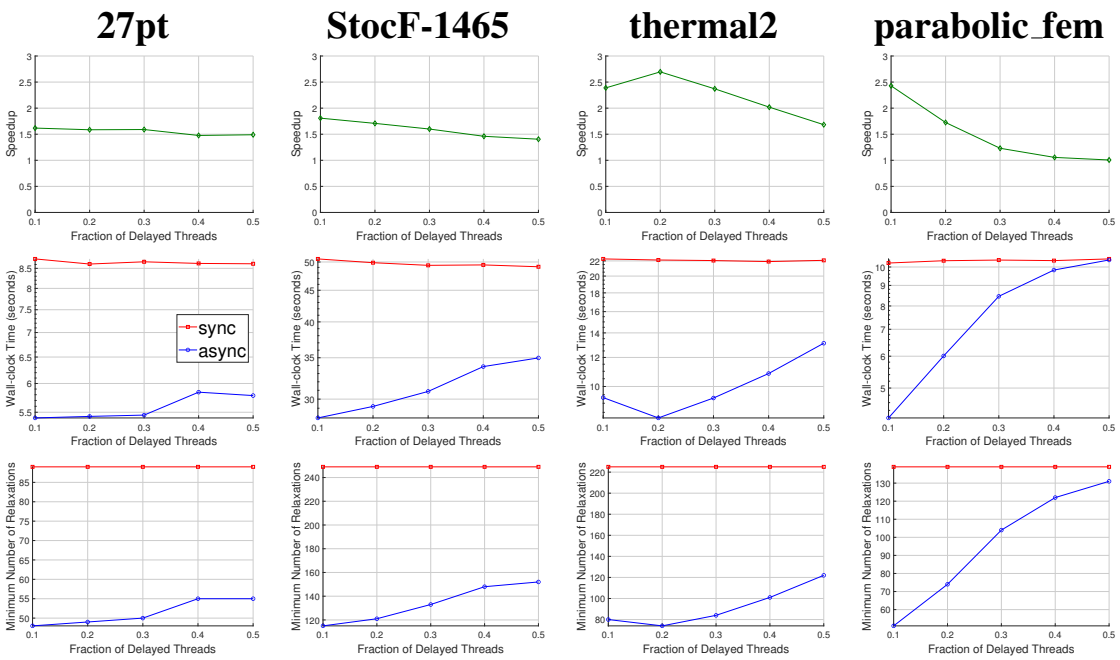


Figure 5.7: Results for varying the fraction of delayed threads using 36 total threads. The residual norm is computed at each iteration with $\tau = 10^{-9}$. The delayed threads use a random delay in microseconds taken from the range [10000, 50000]. The columns denote the test problems. The first row shows the speedup, defined as the wall-clock time of synchronous Chebyshev divided by the wall-clock time of asynchronous Chebyshev. The second row shows the solve wall-clock time. The third row shows the minimum number of relaxations, which is the number of relaxations of the slowest thread.

chronous iterations. In this experiment, we do not compute the relative residual norm until the iteration has completed since it is not needed to generate this plot. We used 36 threads in this experiment. We can see that in all cases, asynchronous EBPX-Chebyshev has a higher convergence rate than synchronous EBPX-Chebyshev. This agrees with the conclusions from our simulations in Section 5.2.3, where we showed that in a scenario with fast communication, asynchronous EBPX-Chebyshev could converge faster than synchronous EBPX-Chebyshev. This means that the difference in wall-clock time between asynchronous EBPX-Chebyshev and synchronous EBPX-Chebyshev (discussed in the next paragraph) is not only due to the removal of synchronization points, but also due to asynchronous EBPX-Chebyshev having a faster convergence rate.

Figure 5.4 shows strong scaling experiments where the residual norm is computed at

Table 5.1: Statistics for our test matrices. The last three matrices are taken from the SuiteSparse Matrix Collection. All matrices are symmetric positive definite.

Matrix	Equations	Non-zeros
27pt	1,000,000	26,463,592
StocF-1465	1,465,137	21,005,389
thermal2	1,228,045	8,580,313
parabolic_fem	525,825	3,674,625

each iteration with $\tau = 10^{-9}$. The first row of plots shows the speedup, defined as the solve wall-clock time of synchronous EBPX-Chebyshev divided by the solve wall-clock time of asynchronous EBPX-Chebyshev. In general, asynchronous EBPX-Chebyshev scales better than synchronous EBPX-Chebyshev with a peak speedup of 1.7. This speedup spikes when the CPU in the second socket is introduced. Once more cores from the second CPU are used, the speedup falls back down, but a speedup above one is still observed. The number of iterations is shown in the third row of plots, where the error bar denotes the maximum and minimum numbers of relaxations. We can see that as the number of threads increases, the convergence rate of asynchronous EBPX-Chebyshev improves since the mean number of relaxations decreases. This indicates that when the faster threads do many more relaxations, progress is made towards the solution, even if information used from the slower threads comes from time instants that are far in the past.

Figures 5.5 and 5.6 show results for an experiment in which we add a delay to a single thread (the 18th thread in this case). The residual norm is computed at each iteration with $\tau = 10^{-9}$. These experiments simulate a scenario in which there is hardware failure and the threads corresponding to the working cores continue to iterate. Figure 5.5 shows the relative residual 2-norm versus number of iterations for a delay of 50000. When comparing Figure 5.5 to Figure 5.3, we can see that when a thread is delayed, the extra relaxations carried out by the un-delayed threads actually increases the convergence rate. Figure 5.6 shows the delay in seconds on the x-axis and speedup, solve wall-clock time, and number of relaxations of the delayed thread are shown on the y-axes. We can see that even for very

large delays, speedup is still achieved, with a speedup of over three for StocF-1465. We can also see that in the case of 27pt and parabolic_fem, the minimum number of relaxations plateaus, which correlates with a plateau in speedup. This is because the delay approaches the total time it takes for the method to converge in the un-delayed case resulting in the un-delayed rows carrying out extra relaxations that do not make progress. In the case of StocF-1465 and thermal2, a higher speedup is achieved. This indicates that for problems that are slower to converge, the additional relaxations from the un-delayed threads make more progress towards convergence. In general, these results show that our method is resilient, i.e., even if some processes stall for a long period of time, progress towards convergence can still be made by other threads.

In our last experiment, we add delays to a fraction of the 36 total threads. This simulates a heterogeneous environment in which some processes are slower in computation than others. Figure 5.7 shows our results where the fraction of delayed threads is shown on the x-axis and speedup, solve wall-clock time, and minimum number of relaxations are shown on the y-axes. The residual norm is computed at each iteration with $\tau = 10^{-9}$. The delayed threads use a random delay in microseconds taken from the range [10000, 50000]. In general, as the fraction of delayed threads increases, the speedup decreases. This is expected since the amount of old data that the fast threads use will increase as the fraction increases, resulting in a slower convergence rate in terms of number of relaxations. However, we still see a positive speedup for the largest fraction of .5. This underscores the importance of our method when running on a heterogenous system. Although a large number of processes might be slower than others, the fast processes can still make progress, even with large portions of out-of-date data.

5.4 Conclusion

Asynchronous versions of fast-converging state-of-the-art solvers have yet to be developed. This chapter introduces the first asynchronous Chebyshev method. We use a variant of the

BPX multigrid method as a preconditioner since methods using multigrid preconditioners converge with a rate independent of the problem size. This variant, which we call EBPX-Chebyshev, is based on expressing multigrid as a basic iterative method on an extended system.

We define models of asynchronous Chebyshev. While we do not provide any analytical results for these models, we show through simulations that asynchronous EBPX-Chebyshev converges even in extreme cases. For example, when data read from memory is very old or when some processes are slower than others. We also show that in the case of fast communication, e.g., in a shared memory environment, asynchronous EBPX-Chebyshev actually converged faster then synchronous EBPX-Chebyshev.

We present numerical results using a shared memory implementation on up to 36 cores. We present strong scaling results which show that asynchronous EBPX-Chebyshev scales better than synchronous EBPX-Chebyshev. We simulate the resilience of asynchronous EBPX-Chebyshev by adding an artificial delay to a thread. We show that as we increase the delay, the speedup of asynchronous EBPX-Chebyshev over synchronous EBPX-Chebyshev also increases. Lastly, we simulate a heterogeneous environment, where some fraction of the total threads are delayed. While speedup decreases as the fraction of delayed threads increases, which is expected, we are still able to observe speedup even when half the threads are delayed.

CHAPTER 6
CONCLUSION

One of the primary challenges of scaling iterative solvers to massively parallel computers is reducing synchronization costs. Asynchronous execution of iterative solvers addresses this challenge. While research on asynchronous methods has been on going since the late 1960s and has recently grown in popularity, there is still important research that must be done. This includes gaining a better understanding of the current set of asynchronous methods and developing new fast-converging asynchronous iterative solvers that can outperform state-of-the-art synchronous solvers. In this dissertation, we make important steps towards this goal. Here, we summarize the main contributions of this dissertation and future work.

6.1 Contributions and Future Work

6.1.1 Asynchronous Jacobi

- **Contributions:** Since asymptotic convergence bounds for asynchronous fixed-point methods have been studied in the literature, our goal was to better understand the practical behavior of asynchronous Jacobi. We introduced a simplified model of asynchronous Jacobi which was easier to analyze than more general models and compared our analytical results to experimental results from our shared and distributed memory implementations. In the simplified model, asynchronous Jacobi can be written in matrix form, which allowed us to analyze the transient convergence behavior of asynchronous Jacobi. We observed that, in practice, asynchronous Jacobi has "better" convergence properties than synchronous Jacobi, which is the main contribution of this research. More precisely, we observed three important results in our experimental tests:

1. Asynchronous Jacobi can converge when synchronous Jacobi does not. This aligned with our analysis of the simplified model where we showed that even if the A-norm of the error increases at each iteration in the synchronous case, the A-norm of the error could still be reduced in the asynchronous case.
2. The convergence rate of asynchronous Jacobi increases as the number of processes increases. This is because asynchronous Jacobi becomes more *multiplicative* as the concurrency increases. In other words, asynchronous Jacobi becomes more like block Gauss-Seidel (which converges faster than Jacobi) in terms of the order in which rows are relaxed.
3. If a process is delayed in its computation, the residual and error norm can still be reduced at each time instant. This is because, while the delayed row is idle, the rows that are being relaxed make significant progress towards convergence.

- **Future work:**
  - We only considered a simplified model in which communication delays are neglected. A future direction would be to extend our analysis to a more general model.
  - Only asynchronous Jacobi was considered in this research. Other parallel fixed-point methods, such as weighted and block Jacobi, could also be studied.
  - This research only considered linear systems with SPD matrices which allowed us to examine the A-norm of the error. An analysis of non-symmetric matrices could also be carried out where a different error or residual norm is considered.

6.1.2 Asynchronous Southwell Methods

- **Contributions:** We developed communication-avoiding methods based on an idea from the original Southwell method, which is sequential. In the original Southwell method, the row with the maximum absolute value residual component is relaxed

at each iteration. We used this idea to develop three new methods, which is the main contribution: Parallel Southwell, Distributed Southwell, and Stochastic Parallel Southwell. In all three methods, the general idea is that a process p compares its local residual norm (residual norm of the rows in the partition assigned to p) to the residual norms of the communication neighbors of p. Based on this comparison, p relaxes the rows in its partition using some criterion that is different for each method. Parallel Southwell was the first method we developed, which was suitable for shared memory. For distributed memory, Parallel Southwell was prone to deadlock, which is why we developed Distributed Southwell, which is deadlock-free. However, additional communication is still required within Distributed Southwell in order to avoid deadlock. In Stochastic Parallel Southwell, no additional communication is required, making it the method with the lowest communication cost. When used as standalone solvers, our experimental tests showed that all three methods converged faster than Jacobi both in number of relaxations and number of iterations, and required less communication. Additionally, all three methods worked as multigrid smoothers, i.e., we observed grid-size independent convergence of multigrid when using our methods as smoothers.

- **Future work:**
  - Our methods can be used as multigrid smoothers and we observed that for the five-point discretization of the Laplace equation (which is considered an easy problem to solve), multigrid converged faster in terms of number of relaxations when using Distributed or Stochastic Parallel Southwell versus Gauss-Seidel as smoothers. However, for other problems such as those in Chapter 4.5.2 multigrid converges more slowly (in terms of number of iterations and wall-clock time) when using our methods than when using other more standard multigrid smoothers such as weighted or block Jacobi. Future work would be to develop

a row selection scheme that results in fast convergence for multigrid.

- We do not provide analytical results as to why our methods work as multigrid smoothers. Future work would be to perform Fourier analysis (or other classical analytical methods) on our current methods.

6.1.3 Asynchronous Multigrid Methods

- **Contributions:** Our main contribution is that we introduced the first asynchronous multigrid methods, which are asynchronous versions of additive multigrid methods. We introduced models that served to define asynchronous multigrid, where subsets of grids update the current approximation to the solution at each time instant, unlike classical models in which subsets of rows are relaxed at each time instant. Through simulations of these models, and through experimental tests with shared and distributed memory implementations, we showed that synchronous multigrid can exhibit grid-size independent convergence. We also introduced a new additive multigrid method that can be executed asynchronously, which is based on the AFACx method [73]. We also showed that our shared and distributed memory implementations can be faster than the classical synchronous multiplicative multigrid solver implemented in the Hypre package. In the distributed memory case, asynchronous AFACj was the fastest method in terms of wall-clock time.

- **Future work:** Synchronous multigrid methods converge with a rate independent of the problem size. While we have shown this fact for asynchronous multigrid through experiments, we do not provide analytical results which we regard as future work. This is a challenge since it is unclear how to use classical analytical techniques, such as Fourier analysis, for asynchronous additive multigrid. This is because, in general, these classical techniques rely on the assumption that the smoother and the coarse grid correction (in the simple case of two grids) are both applied at every iteration. In the asynchronous case this is not true in general, e.g., many updates could be

computed for the coarse grid before the first grid computes its first update.

6.1.4 Asynchronous Chebyshev Methods

- **Contributions:** Our main contribution is that we introduced the first known asynchronous Chebyshev method. We introduced a model of Jacobi-preconditioned asynchronous Chebyshev and showed experimentally that for the standard Jacobi preconditioner, asynchronous Chebyshev diverges for sufficiently large problem sizes. Therefore, we used a little-known variant of the BPX multigrid method as a preconditioner within asynchronous Chebyshev, where BPX can be written as Jacobi on an extended system. We provided experimental results that showed that our asynchronous Chebyshev method can converge faster than its synchronous counter-part, both in wall-clock time and number of relaxations.

- **Future work:**

  - As in the case of asynchronous multigrid, future work would be to provide analytical convergence results. This would involve proving under what conditions on the Chebyshev parameters asynchronous Chebyshev would converge and under what conditions asynchronous BPX can exhibit grid-size independent convergence within Chebyshev.

  - In our current shared memory implementation, the extended system is explicitly constructed and stored, which results in redundant computation. Future work would be to develop an efficient distributed memory implementation where the extended system is implicitly stored.

REFERENCES

[1] *U.S. D.O.E. Workshop Report: Applied Mathematics Research for Exascale Computing*, 2014.

[2] *U.S. D.O.E. Workshop Report: Scientific Grand Challenges: Architectures and Technology for Extreme Scale Computing*, 2009.

[3] *U.S. D.O.E. Workshop Report: Scientific Grand Challenges: Crosscutting Technologies for Computing at the Exascale*, 2010.

[4] *U.S. D.O.E. Workshop Report: Exascale and Beyond: Configuring, Reasoning, Scaling*, 2011.

[5] *U.S. D.O.E. Workshop Report: Exascale Programming Challenges*, 2011.

[6] *Summary Report: The Opportunites and Challenges of Exascale Computing*, 2010.

[7] A. Hefny, D. Needell, and A. Ramdas, "Rows vs. columns: Randomized Kaczmarz or Gauss-Seidel for ridge regression," *SIAM Journal on Scientific Computing*, vol. 39, Jul. 2015.

[8] J. Liu, S. J. Wright, and S. Sridhar, *An asynchronous parallel randomized Kaczmarz algorithm*, 2014. arXiv: `1401.4780 [math.NA]`.

[9] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear Algebra and its Applications*, vol. 2, no. 2, pp. 199 –222, 1969.

[10] G. Baudet, "Asynchronous iterative methods for multiprocessors," *J. ACM*, vol. 25, no. 2, pp. 226–244, Apr. 1978.

[11] Z. Peng, Y. Xu, M. Yan, and W. Yin, "On the convergence of asynchronous parallel iteration with arbitrary delays," U.C. Los Angeles, Tech. Rep., 2016.

[12] Z. Peng, Y. Xu, M. Yan, and W. Yin, "ARock: An algorithmic framework for asynchronous parallel coordinate updates," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, A2851–A2879, 2016.

[13] D. Bertsekas, "Distributed asynchronous computation of fixed points," *Mathematical Programming*, vol. 27, no. 1, pp. 107–120, 1983.

[14] J. Wolfson-Pou and E. Chow, "Convergence models and surprising results for the asynchronous Jacobi method," *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 940–949, 2018.

[15] ——, "Modeling the asynchronous Jacobi method without communication delays," *Journal of Parallel and Distributed Computing*, vol. 128, pp. 84 –98, 2019.

[16] J. Bull and T. Freeman, "Numerical performance of an asynchronous Jacobi iteration," in *Parallel Processing: CONPAR 92—VAPP V: Second Joint International Conference on Vector and Parallel Processing Lyon, France, September 1–4, 1992 Proceedings*, L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 361–366.

[17] I. Bethune, J. M. Bull, N. J. Dingle, and N. J. Higham, "Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP," *International Journal on High Performance Computing Applications*, vol. 28, no. 1, pp. 97–111, 2014.

[18] P. Sanan, S. Schnepp, and D. May, "Pipelined, flexible Krylov subspace methods," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. C441–C470, 2016.

[19] M. Hoemmen, "Communication-avoiding Krylov subspace methods," AAI3413388, Ph.D. dissertation, USA, 2010, ISBN: 9781124140834.

[20] I. Yamazaki, E. Chow, A. Bouteiller, and J. Dongarra, "Performance of asynchronous optimized schwarz with one-sided communication," *Parallel Computing*, vol. 86, pp. 66 –81, 2019.

[21] X. Lian, W. Zhang, C. Zhang, and J. Liu, *Asynchronous decentralized parallel stochastic gradient descent*, 2017. arXiv: 1710.06952 [math.OC].

[22] F. Niu, B. Recht, C. Ré, and S. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *Advances in Neural Information Processing Systems*, vol. 24, Jun. 2011.

[23] G. Golub and M. Overton, "The convergence of inexact Chebyshev and Richardson iterative methods for solving linear systems," *Numerische Mathematik*, vol. 53, no. 5, pp. 571–594, 1988.

[24] J. H. Bramble, J. E. Pasciak, and J. Xu, "Parallel multilevel preconditioners," *Mathematics of Computation*, vol. 55, no. 191, pp. 131–144, 1990.

[25] M. Griebel and P. Oswald, "Greedy and randomized versions of the multiplicative Schwarz method," *Linear Algebra and its Applications*, vol. 437, no. 7, pp. 1596 –1610, 2012.

[26] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, *Parallel Iterative Algorithms: From Sequential to Grid Computing*. Chapman & Hall/CRC, 2007.

[27] A. Frommer and D. Szyld, "On asynchronous iterations," *Journal of Computational and Applied Mathematics*, vol. 123, no. 1–2, pp. 201 –216, 2000.

[28] D. Bertsekas and J. N. Tsitsiklis, "Some aspects of parallel and distributed iterative algorithms: A survey," *Automatica*, vol. 27, no. 1, pp. 3 –21, 1991.

[29] ——, *Parallel and Distributed Computation: Numerical Methods*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989, ISBN: 0-13-648700-9.

[30] B. F. Beidas and G. P. Papavassilopoulos, "Convergence analysis of asynchronous linear iterations with stochastic delays," *Parallel Computing*, vol. 19, no. 3, pp. 281 –302, 1993.

[31] F. Robert, M. Charnay, and F. Musy, "Itérations chaotiques série-parallèle pour des équations non-linéaires de point fixe," *Aplikace Matematiky*, vol. 20, no. 1, pp. 1 – 38, 1975.

[32] J. Hook and N. Dingle, "Performance analysis of asynchronous parallel Jacobi," *Advances in Engineering Software*, vol. 77, no. 3, pp. 831–866, 2018.

[33] J. Hu, T. Nakamura, and L. Li, "Convergence, complexity and simulation of monotone asynchronous iterative method for computing fixed point on a distributed computer," *Parallel Algorithms and Applications*, vol. 11, no. 1-2, pp. 1–11, 1997.

[34] D. Bertsekas and J. N. Tsitsiklis, "Convergence rate and termination of asynchronous iterative algorithms," in *Proceedings of the 3rd International Conference on Supercomputing*, ser. ICS '89, Crete, Greece: ACM, 1989, pp. 461–470.

[35] A. C. Moga and M. Dubois, "Performance of asynchronous linear iterations with random delays," in *Proceedings of International Conference on Parallel Processing*, 1996, pp. 625–629.

[36] D. de Jager and J. Bradley, "Extracting State-Based Performance Metrics using Asynchronous Iterative Techniques," *Performance Evaluation*, vol. 67, no. 12, pp. 1353–1372, 2010.

[37] F. Magoulès and G. Gbikpi-Benissan, "JACK: An asynchronous communication kernel library for iterative algorithms," *The Journal of Supercomputing*, vol. 73, no. 8, pp. 3468–3487, 2017.

[38] ——, "JACK2: An MPI-based communication library with non-blocking synchronization for asynchronous iterations," *Advances in Engineering Software*, vol. 119, pp. 116 –133, 2018.

[39] M. Si, A. J. Peña, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa, "Casper: An asynchronous progress model for mpi rma on many-core architectures," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '15, Washington, DC, USA: IEEE Computer Society, 2015, pp. 665–676.

[40] S. Savari and D. Bertsekas, "Finite termination of asynchronous iterative algorithms," *Parallel Computing*, vol. 22, no. 1, pp. 39–56, 1996.

[41] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, "An efficient and robust decentralized algorithm for detecting the global convergence in asynchronous iterative algorithms," in *High Performance Computing for Computational Science - VECPAR 2008: 8th International Conference, Toulouse, France, June 24-27, 2008. Revised Selected Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 240–254.

[42] J. M. Bahi, S. Contassot-Vivier, R. Couturier, and F. Vernier, "A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 1, pp. 4–13, 2005.

[43] E. W. Dijkstra and C. Scholten, "Termination detection for diffusing computations," *Information Processing Letters*, vol. 11, no. 1, pp. 1 –4, 1980.

[44] *MPI: Message passing interface standard, version 3.0*, High-Performance Computing Center Stuttgart, 2012.

[45] T. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, 1:1–1:25, Dec. 2011.

[46] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998.

[47] M. F. Adams, "A distributed memory unstructured Gauss-Seidel algorithm for multigrid smoothers," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, 2001.

[48] M. Adams, M. Brezina, J. Hu, and R. S. Tuminaro, "Parallel multigrid smoothing: Polynomial versus Gauss-Seidel," *Journal of Computational Physics*, vol. 188, no. 2, pp. 593–610, 2003.

[49] R. Southwell, *Relaxation Methods in Engineering Science - A Treatise on Approximate Computation*. Oxford University Press, 1940.

[50] ——, *Relaxation Methods in Theoretical Physics, a continuation of the treatise, Relaxation methods in engineering science*. Oxford University Press, 1946.

[51] T. Moreshet, R. I. Bahar, and M. Herlihy, "Energy reduction in multiprocessor systems using transactional memory," in *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ser. ISLPED '05, 2005.

[52] U. Rüde, "Fully adaptive multigrid methods," *SIAM Journal on Numerical Analysis*, vol. 30, no. 1, pp. 230–248, 1993.

[53] U. Rüde, *Mathematical and Computational Techniques for Multilevel Adaptive Methods*. Philadelphia, PA, USA: SIAM, 1993.

[54] J. Nutini, M. Schmidt, I. Laradji, M. Friedlander, and H. Koepke, "Coordinate descent converges faster with the Gauss-Southwell rule than random selection," in *ICML-15 Proceedings of the 32nd International Conference on Machine Learning*, 2015, pp. 1632–1641.

[55] T. Blumensath and M. E. Davies, "Stagewise weak gradient pursuits," *IEEE Transactions on Signal Processing*, vol. 57, no. 11, pp. 4333–4346, 2009.

[56] D. L. Donoho, Y. Tsaig, I. Drori, and J.-L. Starck, "Sparse solution of underdetermined systems of linear equations by stagewise orthogonal matching pursuit," *IEEE Transactions on Information Theory*, vol. 58, no. 2, pp. 1094–1121, 2012.

[57] Y. You, X. Lian, J. Liu, H. Yu, I. S. Dhillon, J. Demmel, and C. Hsieh, "Asynchronous parallel greedy coordinate descent," in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds., Curran Associates, Inc., 2016, pp. 4682–4690.

[58] S. G. Mallat and Z. Zhang, "Matching pursuits with time-frequency dictionaries," *IEEE Transactions on Signal Processing*, vol. 41, no. 12, pp. 3397–3415, 1993.

[59] Y. C. Pati, R. Rezaiifar, and P. S. Krishnaprasad, "Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition," in *Proceedings of the 27 th Annual Asilomar Conference on Signals, Systems, and Computers*, 1993, pp. 40–44.

[60] D. Needell and R. Vershynin, "Uniform uncertainty principle and signal recovery via regularized orthogonal matching pursuit," *Foundations of Computational Mathematics*, vol. 9, no. 3, pp. 317–334, 2009.

[61] J. Wolfson-Pou and E. Chow, "Reducing communication in distributed asynchronous iterative methods," in *ICCS Workshop on Mathematical Methods and Al-*

gorithms for Extreme Scale (Procedia Computer Science), vol. 80, 2016, pp. 1906–1916.

[62] K. Blathras, D. Szyld, and Y. Shi, "Timing models and local stopping criteria for asynchronous iterative algorithms," *Journal of Parallel and Distributed Computing*, vol. 58, no. 3, pp. 446 –465, 1999.

[63] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Multigrid smoothers for ultraparallel computing," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2864–2887, 2011.

[64] E. Chow, R. D. Falgout, J. J. Hu, R. S. Tuminaro, and U. M. Yang, "A survey of parallelization techniques for multigrid solvers," *Frontiers of Parallel Processing for Scientific Computing*, 2005.

[65] J. Wolfson-Pou and E. Chow, "Distributed Southwell: An iterative method with low communication costs," in *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC17)*, Nov. 2017, pp. 1–13.

[66] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Scaling hypre's multigrid solvers to 100,000 cores," *High-Performance Scientific Computing: Algorithms and Applications*, pp. 261–279, 2012.

[67] X. C. Tai and P. Tseng, "Convergence rate analysis of an asynchronous space decomposition method for convex minimization," *Mathematics of Computation*, vol. 71, no. 239, pp. 1105–1135, 2002.

[68] P. S. Vassilevski and U. M. Yang, "Reducing communication in algebraic multigrid using additive variants," *Numerical Linear Algebra with Applications*, vol. 21, no. 2, pp. 275–296, 2014.

[69] P. S. Vassilevski, *Multilevel Block Factorization Preconditioners*. Springer-Verlag New York, 2008.

[70] L Hart and S McCormick, "Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors: Basic ideas," *Parallel Computing*, vol. 12, no. 2, pp. 131–144, 1989.

[71] B. Lee, S. McCormick, B. Philip, and D. Quinlan, "Asynchronous fast adaptive composite-grid methods for elliptic problems: Theoretical foundations," *SIAM Journal on Numerical Analysis*, vol. 42, no. 1, pp. 130–152, 2004.

[72] ——, "Asynchronous fast adaptive composite-grid methods: Numerical results," *SIAM Journal on Scientific Computing*, vol. 25, no. 2, pp. 682–700, 2003.

[73] D. Quinlan, "Adaptive mesh refinement for distributed parallel architectures," Ph.D. dissertation, University of Colorado Denver, 1993.

[74] S. McCormick, *Multilevel Adaptive Methods for Partial Differential Equations*. Society for Industrial and Applied Mathematics, 1989.

[75] A. Greenbaum, "A multigrid method for multiprocessors," *Applied Mathematics and Computation*, vol. 19, no. 1-4, pp. 75–88, 1986.

[76] T. F. Chan and R. S. Tuminaro, "Design and implementation of parallel multigrid algorithms," *Proceedings of the Fourth Copper Mountain Conference on Multigrid Methods*, pp. 101–115, 1987.

[77] D. Gannon and J. V. Rosendale, "On the structure of parallelism in a highly concurrent PDE solver," *Journal of Parallel and Distributed Computing*, vol. 3, no. 1, pp. 106–135, 1986.

[78] J. Wolfson-Pou and E. Chow, "Asynchronous multigrid methods," *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 101–110, 2019.

[79] A. AlOnazi, G. S. Markomanolis, and D. Keyes, "Asynchronous task-based parallelization of algebraic multigrid," *Proceedings of the Platform for Advanced Scientific Computing Conference*, no. 5, pp. 1–11, 2017.

[80] J. Hawkes, G. Vaz, A. Phillips, C. Klaij, S. Cox, and S. Turnock, "Chaotic multigrid methods for the solution of elliptic equations," *Computer Physics Communications*, vol. 237, pp. 26–36, 2019.

[81] V. E. Henson and U. M. Yang, "BoomerAMG: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155–177, 2002.

[82] U. M. Yang, "On the use of relaxation parameters in hybrid smoothers," *Numerical Linear Algebra with Applications*, vol. 11, no. 23, pp. 155–172, 2004.

[83] *MFEM: Modular finite element methods library*, `mfem.org`.

[84] R. Sevilla, "NURBS: Enhanced finite element method (NEFEM)," Ph.D. dissertation, Polytechnic University of Catalonia, 2009.

[85] R. Falgout and U. M. Yang, "Hypre: A library of high performance preconditioners," vol. 2331, Apr. 2002, pp. 632–641.

[86] T. A. Manteuffel, "An iterative method for solving nonsymmetric linear systems with dynamic estimation of parameters," *Department of Computer Science, University of Illinois Urbana-Champaign*, 1975.

[87] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd. Philadelphia, PA, USA: SIAM, 2003.

[88] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," *In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC10*, pp. 1–11, Nov. 2010.

[89] M. Griebel, "Multilevel algorithms considered as iterative methods on semidefinite systems," *SIAM Journal on Scientific Computing*, vol. 15, no. 3, pp. 547–565, 1994.

[90] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA: SIAM, 1994.