Journal of Parallel and Distributed Computing 128 (2019) 84–98

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Modeling the asynchronous Jacobi method without communication delays

Jordi Wolfson-Pou [*], Edmond Chow

School of Computational Science and Engineering, Georgia Institute of Technology, 266 Ferst Drive Northwest, Atlanta, GA 30332, USA

HIGHLIGHTS

- We study the asynchronous Jacobi (AJ) method for solving sparse linear systems.
- A model of AJ that assumes no communication delays is analyzed.
- We show that AJ can reduce the residual when a process is slower than others.
- We show that AJ can converge when synchronous Jacobi does not.
- Results for shared and distributed memory support our analysis of AJ.

ARTICLE INFO

Article history:
Received 6 July 2018
Received in revised form 10 January 2019
Accepted 15 February 2019
Available online 22 February 2019

Keywords:
Sparse linear systems
Iterative solvers
Jacobi
Gauss–Seidel
Asynchronous
Remote memory access

ABSTRACT

Asynchronous iterative methods for solving linear systems are gaining renewed interest due to the high cost of synchronization points in massively parallel codes. Historically, theory on asynchronous iterative methods has focused on asymptotic behavior, while the transient behavior remains poorly understood. In this paper, we study a model of the asynchronous Jacobi method without communication delays, which we call *simplified asynchronous Jacobi*. Simplified asynchronous Jacobi can be used to model asynchronous Jacobi implemented in shared memory or distributed memory with fast communication networks. Our analysis uses the idea of a *propagation matrix*, which is similar in concept to an iteration matrix. We show that simplified asynchronous Jacobi can continue to reduce the residual when some processes are slower than other processes. We also show that simplified asynchronous Jacobi can converge when synchronous Jacobi does not. We verify our analysis of simplified asynchronous Jacobi using results from asynchronous Jacobi implemented in shared and distributed memory.

1. Introduction

Modern supercomputers are continually increasing in core count and will soon achieve exascale performance. One problem that will arise for exascale-capable machines is the negative impact synchronization will have on program execution time. Implementations of current state-of-the-art iterative methods for solving the sparse linear system $Ax = b$ suffer from this problem.

In stationary iterative methods, the operation $M^{-1}(b-Ax^{(k)})$ is required, where $x^{(k)}$ is the iterate and M is usually far easier to invert than A. For the Jacobi method, M is a diagonal matrix, so the primary operation is $Ax^{(k)}$, a sparse matrix–vector product. If A is partitioned such that each parallel process is responsible for some number of non-overlapping rows (or subdomains), each process requires values of $x^{(k-1)}$, i.e., information from the previous iteration, so all processes need to be up-to-date. In typical distributed memory implementations, processes are generally idle for some period of time while they wait to receive information from other processes.

Asynchronous iterative methods remove the constraint of waiting for information from the previous iteration. When these methods were conceived, it was thought that continuing computation may be faster than spending time to synchronize. However, synchronization time was less of a problem when asynchronous methods were first proposed because the amount of parallelism was quite low, so asynchronous methods did not gain popularity in practice [13]. Since then, it has been shown experimentally that asynchronous methods can be faster than synchronous methods. However, most existing theory is concerned with asymptotic convergence, and does not study properties of the transient behavior of asynchronous iterative methods.

* Corresponding author.
E-mail addresses: jwp3@gatech.edu (J. Wolfson-Pou), echow@cc.gatech.edu (E. Chow).

https://doi.org/10.1016/j.jpdc.2019.02.002
0743-7315/© 2019 Elsevier Inc. All rights reserved.

In this paper, 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 of asynchronous Jacobi, we show two results:

1. Let A be weakly diagonally dominant. If some processes are slower than others, which, for example, may be due to slower cores in heterogeneous architectures or load imbalance in the problem being solved, iterating asynchronously can result in considerable speedup.
2. When A is symmetric positive definite, simplified asynchronous Jacobi can converge when synchronous Jacobi does not. This is a surprising result because the classical theory predicts that asynchronous Jacobi will not converge in general (precisely, there exists a sequence of asynchronous iterations that does not converge) if synchronous Jacobi does not converge.

2. Background

2.1. The Jacobi and Gauss–Seidel methods

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

$$x^{(k+1)} = Bx^{(k)} + f, \tag{1}$$

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

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

$$e^{(k+1)} = Be^{(k)}, \tag{2}$$

where B 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 $\rho(B) < 1$. Analyzing $\|B\|$ 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 $\rho(B) < 1$ since $\rho(B) \le \|B\|$. If B is not normal, $\|B\|$ can be ≥ 1 when $\rho(B) < 1$. 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. Eq. (1) can be written as

$$x^{(k+1)} = (I - M^{-1}A)x^{(k)} + M^{-1}b, \tag{3}$$

where $B = I - M^{-1}A$. Just like in Eq. (2), we can write

$$r^{(k+1)} = Cr^{(k)}, \tag{4}$$

where the *residual* is defined as $r^{(k)} = b - Ax^{(k)}$ and $C = I - AM^{-1}$ 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.

2.2. Asynchronous iterative methods

We now consider a general model of an asynchronous stationary iterative method as presented in Chapter 5 of [2]. 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 Eq. (1) can be thought of as *synchronous*. In particular, all elements of $x^{(k)}$ must be computed before iteration $k + 1$ 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

$$x_i^{(k+1)} = \begin{cases} \displaystyle\sum_{j=1}^{n} B_{ij} x_j^{(s_{ij}(k))} + f_i, & \text{if } i \in \Psi(k), \\ x_i^{(k)}, & \text{otherwise.} \end{cases} \tag{5}$$

The set $\Psi(k)$ is the set of rows that are relaxed at step k. The mapping $s_{ij}(k)$ denotes the components of other rows that process i reads from memory. The following assumptions are standard for the convergence theory of Eq. (5):

1. The mapping $s_{ij}(k) \le k$. This means no future information is read from memory.
2. As $k \to +\infty$, $s_{ij}(k) \to +\infty$. This means rows will eventually read new information from other rows.
3. As $k \to +\infty$, the number of times i appears in $\Psi(k) \to +\infty$. This means that all rows eventually relax in a finite amount of time.

3. Related work

An overview of asynchronous iterative methods can be found in Chapter 5 of [2]. Reviews of the convergence theory for asynchronous iterative methods can be found in [2,6,8,13]. Asynchronous iterative methods were first introduced as *chaotic relaxation* methods by Chazan and Miranker [10]. 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 B, if $\rho(|B|) < 1$, then the asynchronous version of the method will converge. Other researchers have expanded on suitable conditions for asynchronous methods to converge using different asynchronous models [3–5,14,20–22]. One of these models introduces the idea of propagation matrices, which is what we analyze in this paper [22]. There are also papers that show that asynchronous methods can converge faster than their synchronous counterparts [7,15]. In [15], 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 [7], and was extended to contraction maps. The speedup of asynchronous Jacobi was studied in [19] 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 4.3 why speedup is not often suspected in this case. In [14], 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 [14] do not prove anything for the non-negative case, examples in which asynchronous Jacobi converges when synchronous Jacobi does not are shown, which is also explored in this paper. The authors in [14] also provide numerical experiments that show that asynchronous Jacobi has a higher convergence rate for certain problems. As in [22], the idea of propagation matrices are used in [14].

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 [3,9]. 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" [12], 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 [9]. The "racy" scheme presented in [9] is what we consider in our paper, but we use one-sided MPI with passive target completion where as Bethune et al. used SHMEM and two-sided MPI. The results in [9] 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 [17,18], and Casper [23], which provides asynchronous progress control in certain cases. We are not using any of these tools in our implementations.

4. Asynchronous iterative methods without communication delays

4.1. 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 Eq. (5) as

$$x_i^{(k+1)} = \begin{cases} \sum_{j=1}^{n} B_{ij} x_j^{(k)} + f_i, & \text{if } i \in \Psi(k), \\ x_i^{(k)}, & \text{otherwise.} \end{cases} \tag{6}$$

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

$$x^{(k+1)} = (I - \hat{M}^{(k)} A) x^{(k)} + \hat{M}^{(k)} b \tag{7}$$

where

$$\hat{M}^{(k)}(i,:) = \begin{cases} M^{-1}(i,:), & \text{if } i \in \Psi(k), \\ 0, & \text{otherwise,} \end{cases} \tag{8}$$

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

$$\hat{B}^{(k)} = I - \hat{M}^{(k)} A \quad \text{and} \quad \hat{C}^{(k)} = I - A\hat{M}^{(k)}, \tag{9}$$

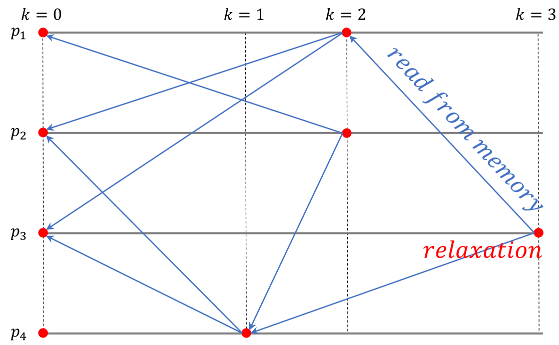


Fig. 1. Example of four processes carrying out three iterations of an asynchronous iterative method without communication delays. Relaxations are denoted by red dots, and information used for relaxations is denoted by blue arrows.

respectively.

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

$$\hat{D}_{ii}^{(k)} = \begin{cases} 1/A_{ii}, & \text{if } i \in \Psi(k), \\ 0, & \text{otherwise.} \end{cases} \tag{10}$$

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

An example of a sequence of asynchronous relaxations is shown in Fig. 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,

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

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

where the $\times$ symbol denotes a non-zero value.

4.2. 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 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 k, then

$$\hat{D}_{ii}^{(k)} = \begin{cases} 1/A_{ii}, & \text{if } i = j, \\ 0, & \text{otherwise.} \end{cases} \tag{11}$$

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, $\hat{D}^{(k)}$ can be expressed as

$$\hat{D}_{ii}^{(k)} = \begin{cases} 1/A_{ii}, & \text{if } i \in \Gamma, \\ 0, & \text{otherwise.} \end{cases} \tag{12}$$

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.

4.3. 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 $i = 1, \dots, n$ 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, as we do in our model discrete time points k). 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 step k. Then $\rho(\hat{G}^{(k)}) = \|\hat{G}^{(k)}\|_\infty = 1$ and $\rho(\hat{H}^{(k)}) = \|\hat{H}^{(k)}\|_1 = 1$ for simplified asynchronous Jacobi.*

Proof. Let the number of equations be n, and let $\xi_1, \dots, \xi_n$ be the n unit (coordinate) basis vectors. Without loss of generality, consider a single equation i to be not relaxed at step k. The proof of $\|\hat{G}^{(k)}\|_\infty = 1$ is straightforward. Since row i in $\hat{G}^{(k)}$ is ξ_i^T, and since A is W.D.D., $\|\hat{G}^{(k)}\|_\infty = 1$. Similarly, for $\|\hat{H}^{(k)}\|_1$, column i is ξ_i and so $\|\hat{H}^{(k)}\|_1 = 1$. To prove $\rho(\hat{G}^{(k)}) = 1$, consider the splitting $\hat{G}^{(k)} = I + Y$, where I is the identity matrix. The matrix Y has the same elements as $\hat{G}^{(k)}$ except the diagonal is $\text{diag}(\hat{G}^{(k)}) - I$ and the ith row of Y is all zeros. Since Y has a row of zeros, it must have nullity ≥ 1. Therefore, an eigenvector of $\hat{G}^{(k)}$ is $v = \text{null}(Y)$ with eigenvalue of 1 since $(I + Y)v = v$. To prove $\rho(\hat{H}^{(k)}) = 1$, it is clear that ξ_i is an eigenvector of $\hat{H}^{(k)}$ since column i of $\hat{H}^{(k)} = \xi_i$. Therefore, $\hat{H}^{(k)}\xi_i = \xi_i$. □

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 $\hat{H}^{(k)}$ corresponding to eigenvalues of one are unit basis vectors. Upon multiplying $\hat{H}^{(k)}$ 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 [19], 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

$$\hat{G}^{(k)} = \begin{bmatrix} 1 & 0 \\ \alpha & 0 \end{bmatrix}, \quad \hat{H}^{(k)} = \begin{bmatrix} 1 & \alpha \\ 0 & 0 \end{bmatrix}, \tag{13}$$

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 step k. We can write the iteration as

$$e^{(k+1)} = \hat{G}^{(k)} e^{(k)} = \begin{bmatrix} 1 & o^T \\ g_1^{(k)} & \tilde{G}^{(k)} \end{bmatrix} \begin{bmatrix} e_1^{(k)} \\ \tilde{e}^{(k)} \end{bmatrix} \tag{14}$$

where o is the $(n-1) \times 1$ zero vector, $g_1^{(k)}$ is an $(n-1) \times 1$ vector, and $\tilde{G}^{(k)}$ is a $(n-1) \times (n-1)$ symmetric principal submatrix of the synchronous iteration matrix G. Since $\tilde{G}^{(k)}$ 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 $\tilde{G}^{(k)}$ with eigenvalues of G. Specifically, if $\lambda_1, \dots, \lambda_n$ are the eigenvalues of G, the ith eigenvalue μ_i of $\tilde{G}^{(k)}$ 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 step k, we can consider the system $P^{(k)} A P^{(k)^T} P^{(k)} x = P^{(k)} b$, which has the iteration

$$P^{(k)} x^{(k+1)} = P^{(k)} \hat{G}^{(k)} P^{(k)^T} P^{(k)} x^{(k)} + P^{(k)} \hat{D}^{(k)} P^{(k)^T} P^{(k)} b. \tag{15}$$

The matrix $P^{(k)}$ 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

$$e^{(k+1)} = \hat{G}^{(k)} e^{(k)} = \begin{bmatrix} I & O^T \\ G_I^{(k)} & \tilde{G}^{(k)} \end{bmatrix} \begin{bmatrix} e_I^{(k)} \\ \tilde{e}^{(k)} \end{bmatrix}, \tag{16}$$

where I is the $(n-m) \times (n-m)$ identity matrix, O is the $m \times (n-m)$ zero matrix, $G_I^{(k)}$ is $m \times (n-m)$, and $\tilde{G}^{(k)}$ is $m \times m$. For an eigenvalue μ_i of $\tilde{G}^{(k)}$, $\lambda_i \leq \mu_i \leq \lambda_{i+n-m}$ for $i = 1, \dots, m$. 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 $\tilde{G}^{(k)}$.

4.4. Simplified asynchronous Jacobi can converge when synchronous Jacobi does not

A well known result, known as early as Chazan and Miranker [10], 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 $\tilde{G}^{(k)}$ that results in the decrease of the A-norm of the error at iteration k 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 $\rho(\tilde{G}^{(k)}) < 1$, and $\tilde{e}^{(k)} \neq 0$, then the A-norm of the error $\|e^{(k)}\|_A$ decreases at step k.*

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

$$\begin{aligned}\|e^{(k+1)}\|_A^2 &= e^{(k+1)^T} A e^{(k+1)} \\ &= e^{(k)^T} \hat{G}^{(k)^T} A \hat{G}^{(k)} e^{(k)} \\ &= e^{(k)^T} (I - A\hat{D}^{(k)}) A (I - \hat{D}^{(k)} A) e^{(k)} \\ &= \|e^{(k)}\|_A^2 - e^{(k)^T} A (2\hat{D}^{(k)} - \hat{D}^{(k)} A \hat{D}^{(k)}) A e^{(k)}.\end{aligned} \tag{17}$$

Let m be the number of rows being relaxed at step k. Without loss of generality, we can consider the ordering from Eq. (16), and write

$$\hat{D}^{(k)} = \begin{bmatrix} \mathbf{0} & O^T \\ O & I \end{bmatrix}, \tag{18}$$

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,

$$2\hat{D}^{(k)} - \hat{D}^{(k)} A \hat{D}^{(k)} = \begin{bmatrix} \mathbf{0} & O^T \\ O & 2I - \tilde{A}^{(k)} \end{bmatrix}, \tag{19}$$

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

The proof of Theorem 2 can also be used to show that a single Gauss–Seidel relaxation reduces $\|e^{(k)}\|_A^2$. This is because

$$2\hat{D}^{(k)} - \hat{D}^{(k)} A \hat{D}^{(k)} = \begin{bmatrix} \mathbf{0} & O^T \\ O & I \end{bmatrix}, \tag{20}$$

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, $\|e^{(k+1)}\|_A^2 < \|e^{(k)}\|_A^2$.

Returning to the discussion of simplified asynchronous Jacobi, it can happen that $\|e^{(k+1)}\|_A^2 < \|e^{(k)}\|_A^2$ since $\rho(\tilde{G}^{(k)}) \leq \rho(G)$ by the interlacing theorem. Matrix $\tilde{G}^{(k)}$ decreases in size when fewer rows are relaxed in parallel, which happens when the number of threads or processes is increased. Furthermore, $\tilde{G}^{(k)}$ 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 $\rho(\tilde{G}_i^{(k)}) \leq \rho(\tilde{G}^{(k)})$, where $\tilde{G}_i^{(k)}$ is block i of $\tilde{G}^{(k)}$ with the largest spectral radius. If many processes are used, it may happen that $\tilde{G}^{(k)}$ will have many blocks, resulting in $\rho(\tilde{G}_i^{(k)}) \ll \rho(\tilde{G}^{(k)})$. 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.

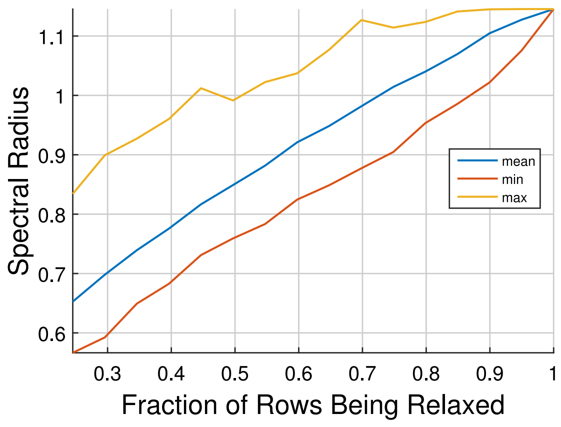


Fig. 2. Spectral radius of $\tilde{G}^{(k)}$ versus the fraction of rows being relaxed. Max, min, and mean spectral radius is shown for 200 different choices of $\tilde{G}^{(k)}$, where the rows selected to be relaxed are chosen randomly. The test problem is the FE matrix.

An example of how $\rho(\tilde{G}^{(k)})$ changes as the fraction of rows that are being relaxed decreases is shown in Fig. 2. The matrix used is the FE matrix (see Section 7.1). For each fraction of rows being relaxed, the max, min, and mean spectral radius of 200 different choices of $\tilde{G}^{(k)}$ 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 $\rho(\tilde{G}^{(k)}) < 1$. Additionally, when the fraction of rows being relaxed is $\approx .9$ or less, the mean of all $\rho(\tilde{G}^{(k)}) < 1$. 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.

5. 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 $r_i^{(k)} = b_i - \sum_{j=1}^{n} A_{ij} x_j^{(k)}$ (this is the SpMV step).
2. For each row i, correct the approximation $x_i^{(k+1)} = x_i^{(k)} + r_i^{(k)} / A_{ii}$.
3. Check for convergence (detailed below and in Section 6 for distributed memory).

Since more than one row is assigned to each thread, each thread only computes $Ax^{(k)}$, $r^{(k)}$ and $x^{(k)}$ 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 $x^{(k)}$ and $r^{(k)}$ 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 $x^{(k)}$ or $r^{(k)}$ 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.

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

$$\vec{x}_i^{(k+1)} = \vec{y}_i + \sum_{j=1}^{\mathcal{N}_i} \mathbf{A}_{iq_{ij}}\vec{x}_{q_{ij}}^{(k)}. \tag{21}$$

The matrix $\mathbf{A}_{ii}$ is the diagonal block that belongs to process i (red blocks in Fig. 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 Fig. 3), and $\vec{x}_{q_{ij}}^{(k)}$ 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.

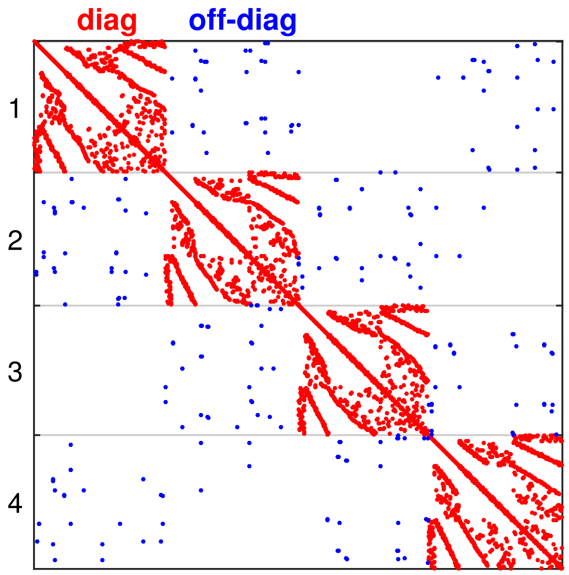


Fig. 3. Sparsity pattern for an unstructured finite element matrix partitioned into four parts. The red points denote the non-zero values of the diagonal blocks of the matrix, and the blue points denote the non-zero values of the off-diagonal blocks. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

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 [1], 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 ith 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 $j = 0, \dots, (\texttt{sendcount} - 1)$.

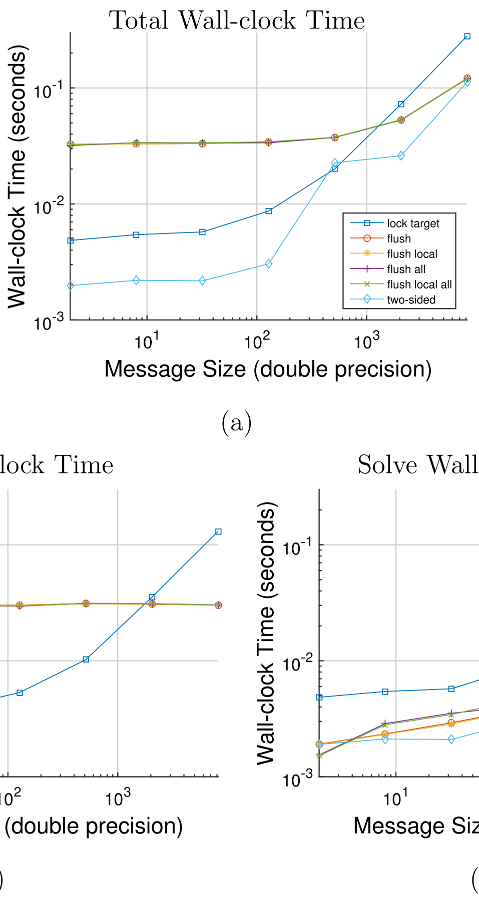
From the steps above, in each iteration, we can see that each process expects to read the current iteration number at each element of its memory window. Step 2 means that if a process does not read the current iteration number after s seconds, the program terminates.

For our first experiment, we used nine Haswell nodes (3×3 mesh) of the Cori supercomputer at NERSC (see Section 7.1), 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 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 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 (s)
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

Fig. 4(a) shows how the overall wall-clock time of our benchmark is affected by different message sizes. In this figure, 32 nodes with 1024 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 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 Fig. 4 (b). Fig. 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 1 is because more MPI processes are used in Fig. 4. The wall-clock time for `lock_all()` functions increases with increasing number of MPI processes. This suggests that some form of 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. Fig. 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.



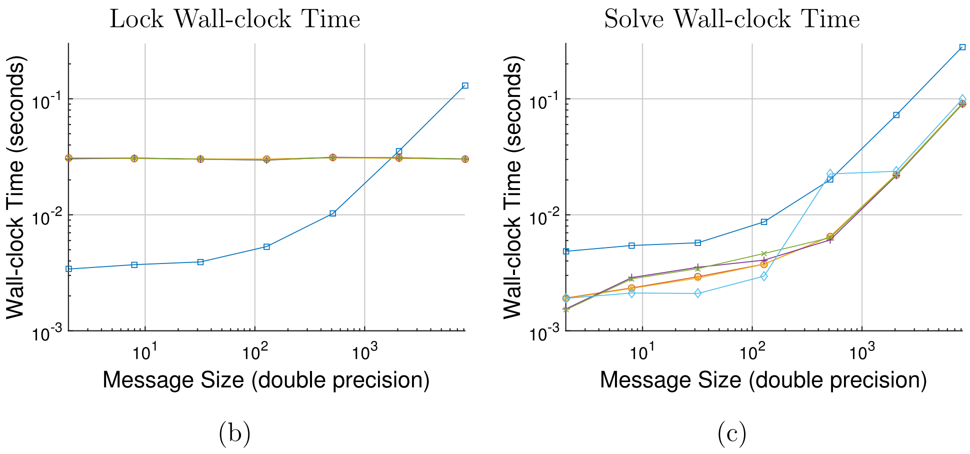


Fig. 4. Wall-clock time as a function of message size for our benchmark. The benchmark measures how long it takes for communication operations to complete when using one-sided MPI with passive target completion. Figure (a) shows the overall wall-clock time, Figure (b) shows the wall-clock time for locking 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 1024 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.

7. Results

7.1. 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 3081 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.

Table 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

3. Matrices listed in Table 2 from the SuiteSparse matrix collection [11].

Matrices are partitioned using METIS [16], including for FD, and are stored in compressed sparse row (CSR) format.

7.2. 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 4 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

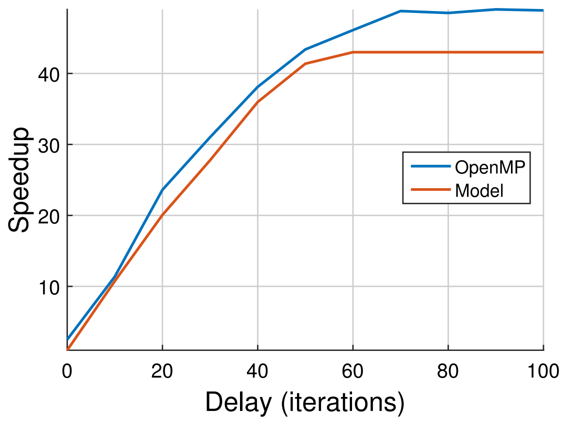


Fig. 5. Speedup of simplified and OpenMP asynchronous Jacobi over synchronous Jacobi for 68 threads (on the KNL platform) as a function of the artificial delay in computation δ experienced by one thread. For OpenMP, the delay is varied from zero to 3000 microseconds. For the simplified asynchronous Jacobi, 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.

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 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. Fig. 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.

Fig. 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 Fig. 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 $\hat{H}^{(k)}$ is fixed and thus, from Eq. (14),

$$r^{(k+1)} = \hat{H}^{(k)} r^{(k)} = \begin{bmatrix} 1 & h_1^{(k)} \\ o & \tilde{H}^{(k)} \end{bmatrix} \begin{bmatrix} r_1^{(k)} \\ \tilde{r}^{(k)} \end{bmatrix}. \tag{22}$$

In this equation, $\hat{H}^{(k)}$ has just one eigenvalue equal to one corresponding to the eigenvector ξ_1 (first unit basis vector). The remaining eigenvalues are the eigenvalues of $\tilde{H}^{(k)}$. From the proof of Theorem 1, $r^{(k)}$ converges to $\begin{bmatrix}\gamma & o^T\end{bmatrix}^T$ when applying $\hat{H}^{(k)}$ to $r^{(k)}$ many times, where γ is some scalar. The remaining $n-1$ eigenvectors of $\hat{H}^{(k)}$ corresponding to the eigenvalues of $\tilde{H}^{(k)}$ are exactly $\begin{bmatrix}0 & \mathbf{v}_j^T\end{bmatrix}$, where $\mathbf{v}_j$ is an eigenvector of $\tilde{H}^{(k)}$ with $j = 1, \ldots, n-1$. Therefore, for this FD matrix, with a starting residual of $\tilde{r}^{(k)}$, the speedup of simplified asynchronous Jacobi will always increase until the iteration governed by the residual iteration matrix $\tilde{H}^{(k)}$ 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.

Fig. 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.

Fig. 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 Fig. 5, the stall in convergence for a delay of 100 is due to the convergence of the iteration corresponding to using $\tilde{H}^{(k)}$ as the residual iteration matrix. However, it takes $\approx$ 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.

Fig. 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 these results, we used an FD matrix with 4624 rows (17×16 mesh) and 22,848

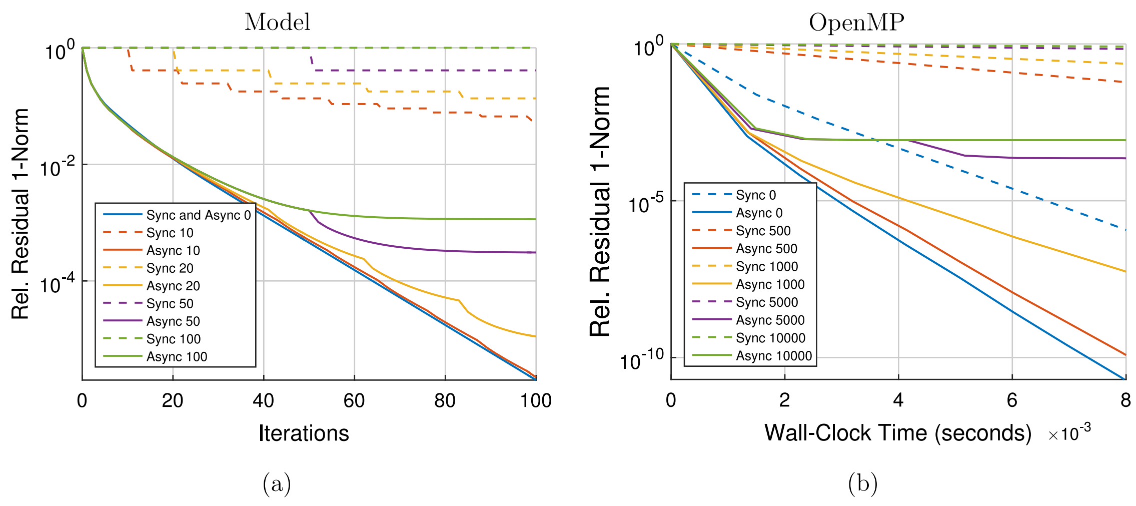


Fig. 6. Relative residual 1-norm as a function of number of iterations for simplified asynchronous Jacobi, shown in (a), and relative residual norm as a function of OpenMP asynchronous Jacobi, shown in (b). Artificial delays in computation are added for both simplified and OpenMP asynchronous Jacobi, 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.

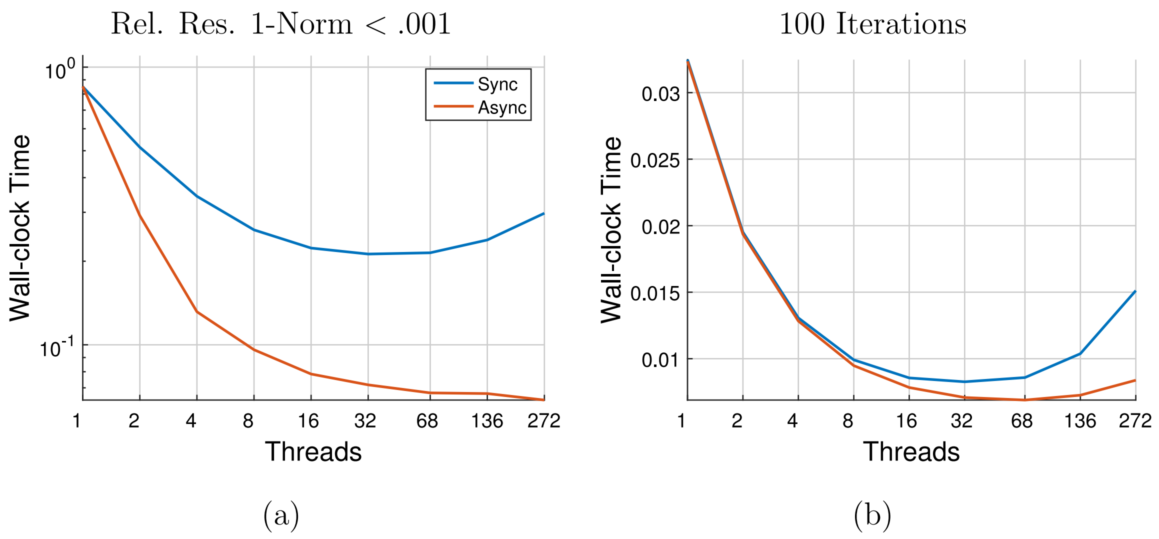


Fig. 7. OpenMP asynchronous Jacobi compared with synchronous Jacobi as the number of threads increases. Figure (a) shows the wall-clock time when both methods reduce the relative residual 1-norm below .001. Figure (b) shows how much time is taken to carry out 100 iterations. The test problem is an FD matrix with 4624 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.

non-zero values. When the number of threads does not divide 4624 evenly, METIS is used. As in the previous set of results, we averaged the wall-clock time of 100 samples for each data point. Fig. 7(a) shows the wall-clock time for achieving a relative residual norm below .001. Fig. 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.

Fig. 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 7.1). This indicates that synchronization points have a higher cost than the extra computation done by OpenMP asynchronous 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

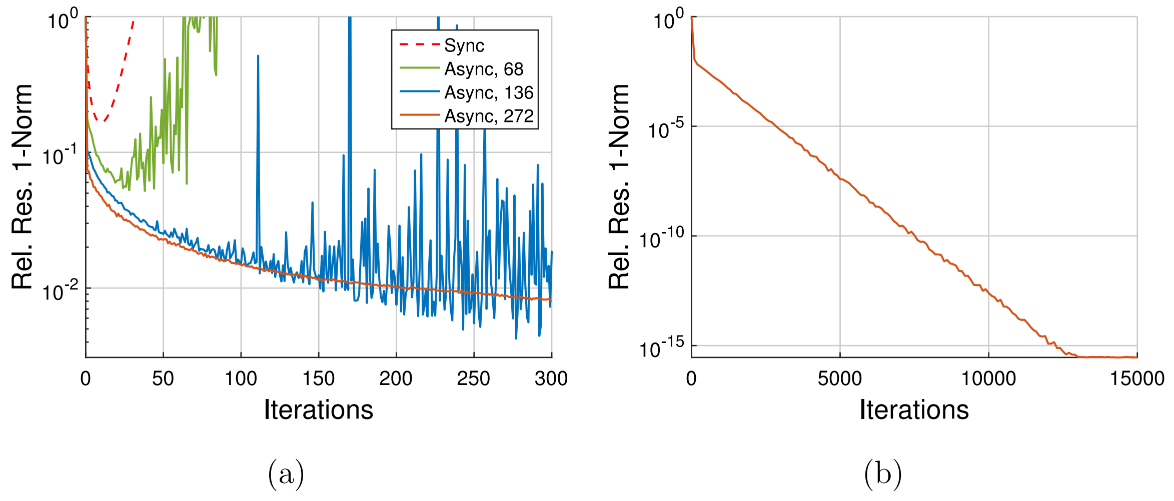


Fig. 8. Relative residual 1-norm as a function of iterations for different numbers of threads (68, 136, and 272) on the KNL platform. Figure (a) shows that for a sufficient number of threads, asynchronous Jacobi can converge when synchronous Jacobi does not. Figure (b) shows that asynchronous Jacobi truly converges when using 272 threads. The test problem is the FE matrix.

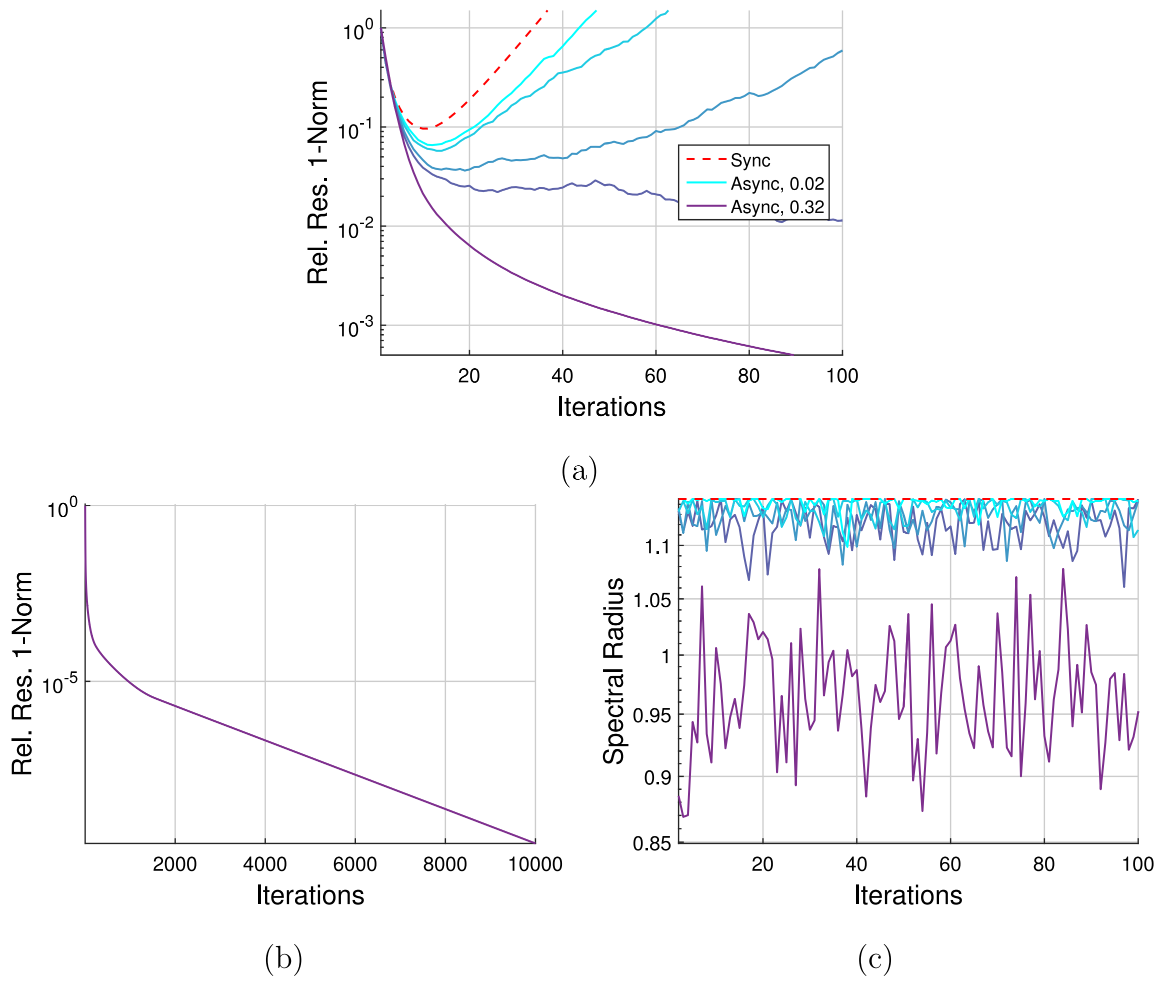


Fig. 9. In this experiment, a random number of random rows is selected to not be relaxed (delayed) at each iteration for simplified asynchronous Jacobi. The fraction of delayed rows is varied from .02 to .32, where the cyan to purple gradient of the lines represents an increasing fraction of delayed 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 $\rho(\tilde{G}^{(k)})$ 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.

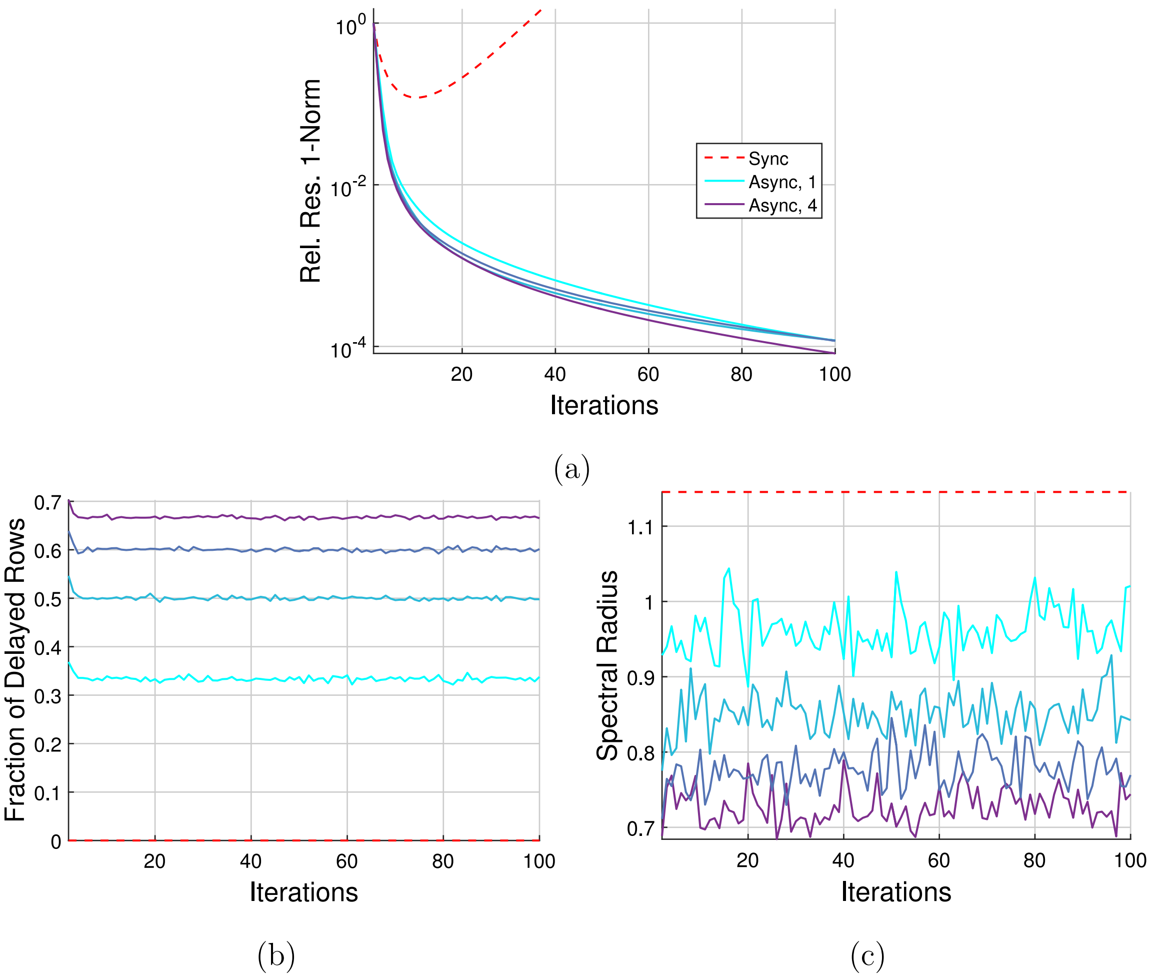


Fig. 10. In this experiment, each row is assigned a delay in computation δ_i in the range $[0, 1, \ldots, \delta_{\max}]$ (the row is only relaxed when δ_i divides the iteration numbers evenly) for simplified asynchronous Jacobi. The maximum delay $\delta_{\max}$ is varied from one to four, where the 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 $\rho(\tilde{G}^{(k)})$, 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.

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.

We now look at a case in which OpenMP asynchronous Jacobi converges when synchronous Jacobi does not. Our test problem is the FE matrix. Fig. 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 Fig. 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 5 for details on how threads decide to stop iterating). In Fig. 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. Fig. 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. Figs. 9 and 10 show the convergence for simplified asynchronous Jacobi using the FE matrix. Fig. 9 shows results for an experiment in which a random set of random rows are selected to be relaxed at each iteration. In Fig. 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 Fig. 8 for OpenMP asynchronous Jacobi. Just as in Figs. 8(b), 9(b) shows that the simplified asynchronous Jacobi truly converges. As discussed in Section 4.4, this convergence can be explained by examining $\rho(\tilde{G}^{(k)})$. Fig. 9(c) shows $\rho(\tilde{G}^{(k)})$ as the number of iterations increases. For a fraction of delayed rows of .32, $\rho(\tilde{G}^{(k)})$ is often less than one.

Fig. 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 relaxed, 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

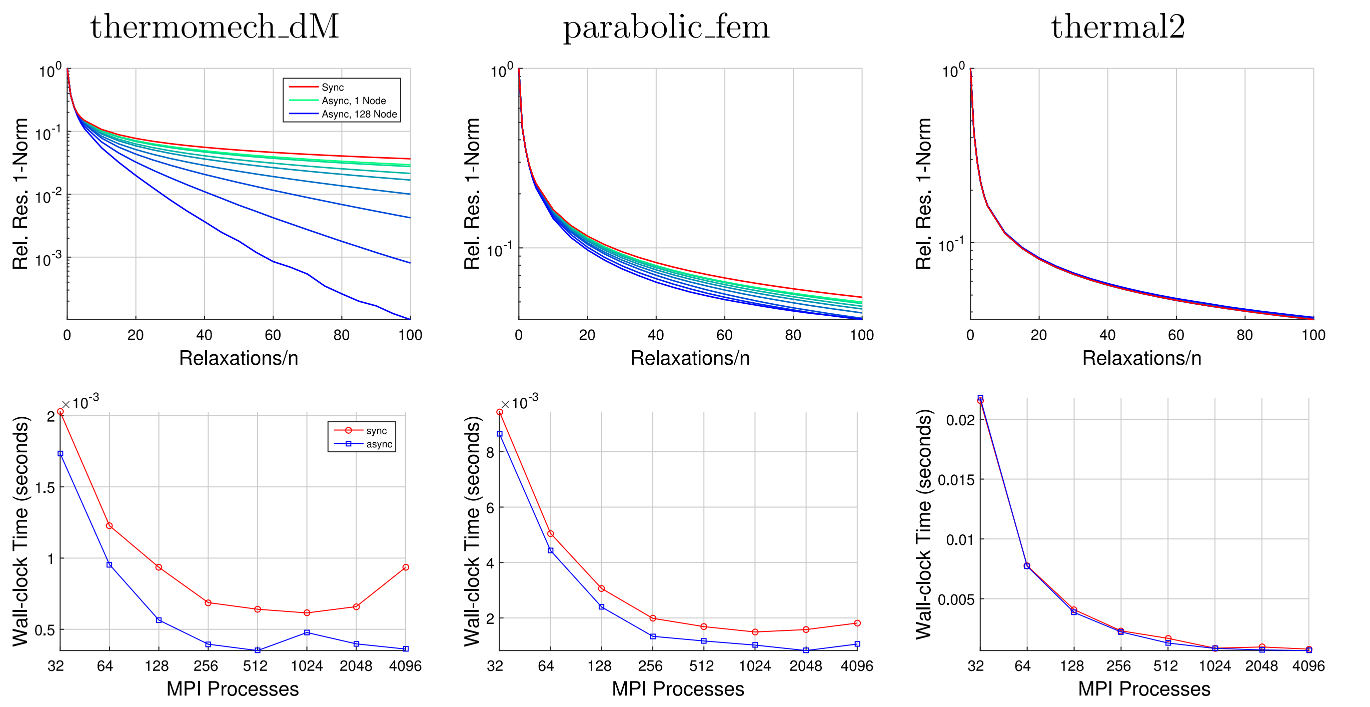


Fig. 11. The first row shows the relative residual 1-norm as a function of relaxations/n for synchronous Jacobi and POS asynchronous Jacobi. For POS asynchronous Jacobi, one to 128 nodes are shown (32 to 4096 MPI processes), where the green to blue color gradient of the lines represents an increasing 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. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

vary the maximum delay $\delta_{\max}$ from one to four. Fig. 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 (Fig. 9(b)), and the resulting $\rho(\tilde{G}^{(k)})$ corresponding to that fraction (Fig. 9(c)). With a delay of one, the fraction of delayed rows is between .3 and .4, and $\rho(\tilde{G}^{(k)})$ is often less than one. For larger delays, $\rho(\tilde{G}^{(k)})$ is always less than one.

7.3. 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.

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, . . . , 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 Fig. 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 4096 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.

The second row of figures in Fig. 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 outweighing computation time. However, since increasing the number of MPI processes improves convergence, wall-clock times for 2048 and 4096 MPI processes are lower than for 1024 processes. This result is similar to that of Fig. 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.

Improving the convergence with added concurrency is more dramatic in Fig. 12, where the relative residual 1-norm as a function of number of relaxations is shown for Dubcova2. This

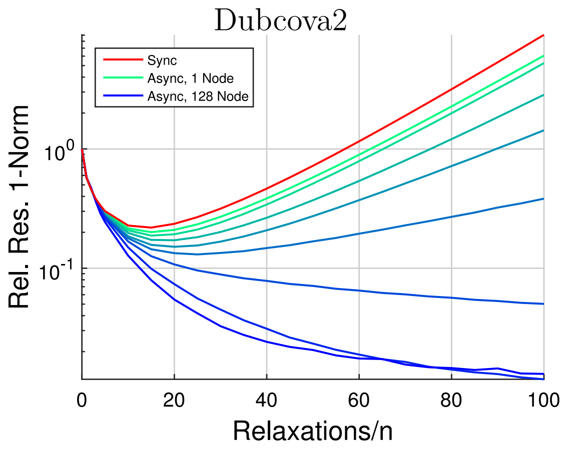


Fig. 12. Relative residual 1-norm as a function of relaxations/n for synchronous Jacobi using two-sided communication and for POS asynchronous Jacobi. The Dubcova2 matrix is used as the test problem. For POS asynchronous Jacobi, results for one to 128 nodes are shown (32 to 4096 MPI processes), where the green to blue color gradient of the lines represents an increasing number of nodes. As in Fig. 8, increasing the number of processes improves the convergence rate of asynchronous Jacobi. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

behavior is similar to the behavior shown in Fig. 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 in this paper, 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.

8. Conclusion

The transient convergence behavior of asynchronous iterative methods has not been well-understood. In this paper, 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 paper used a simplified model for asynchronous iterations assuming no communication delays, we have also observed property (a) when experimenting with nonsymmetric matrices and the general model (Eq. (5)) of asynchronous iterative methods.

Acknowledgments

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 program under Award Number DE-SC-0016564. 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.

References

[1] MPI: Message Passing Interface Standard, Version 3.0, High-Performance Computing Center Stuttgart, September 2012.
[2] J.M. Bahi, S. Contassot-Vivier, R. Couturier, Parallel Iterative Algorithms: From Sequential to Grid Computing, Chapman & Hall/CRC, 2007.
[3] G. Baudet, Asynchronous iterative methods for multiprocessors, J. ACM 25 (2) (1978) 226–244.
[4] B.F. Beidas, G.P. Papavassilopoulos, Convergence analysis of asynchronous linear iterations with stochastic delays, Parallel Comput. 19 (3) (1993) 281–302.
[5] D. Bertsekas, Distributed asynchronous computation of fixed points, Math. Program. 27 (1) (1983) 107–120.
[6] D. Bertsekas, J.N. Tsitsiklis, Parallel and Distributed Computation: Numerical Methods, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
[7] D. Bertsekas, J.N. Tsitsiklis, Convergence rate and termination of asynchronous iterative algorithms, in: Proceedings of the 3rd International Conference on Supercomputing, in: ICS '89, ACM, New York, NY, USA, 1989, pp. 461–470.
[8] D. Bertsekas, J.N. Tsitsiklis, Some aspects of parallel and distributed iterative algorithms: a survey, Automatica 27 (1) (1991) 3–21.
[9] I. Bethune, J.M. Bull, N.J. Dingle, N.J. Higham, Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP, Int. J. High Perform. Comput. Appl. 28 (1) (2014) 97–111.
[10] D. Chazan, W. Miranker, Chaotic relaxation, Linear Algebra Appl. 2 (2) (1969) 199–222.
[11] T. Davis, Y. Hu, The University of Florida sparse matrix collection, ACM Trans. Math. Software 38 (1) (2011) 1:1–1:25.
[12] D. De Jager, J. Bradley, Extracting state-based performance metrics using asynchronous iterative techniques, Perform. Eval. 67 (12) (2010) 1353–1372.
[13] A. Frommer, D. Szyld, On asynchronous iterations, J. Comput. Appl. Math. 123 (12) (2000) 201–216.
[14] J. Hook, N. Dingle, Performance analysis of asynchronous parallel Jacobi, Adv. Eng. Softw. 77 (3) (2018) 831–866.
[15] J. Hu, T. Nakamura, L. Li, Convergence, complexity and simulation of monotone asynchronous iterative method for computing fixed point on a distributed computer, Parallel Algorithms Appl. 11 (1–2) (1997) 1–11.
[16] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20 (1) (1998) 359–392.
[17] F. Magoulès, G. Gbikpi-Benissan, JACK: an asynchronous communication kernel library for iterative algorithms, J. Supercomput. 73 (8) (2017) 3468–3487.
[18] F. Magoulès, G. Gbikpi-Benissan, JACK2: an MPI-based communication library with non-blocking synchronization for asynchronous iterations, Adv. Eng. Softw. 119 (2018) 116–133.
[19] A.C. Moga, M. Dubois, Performance of asynchronous linear iterations with random delays, in: Proceedings of International Conference on Parallel Processing, 1996, pp. 625–629.
[20] Z. Peng, Y. Xu, M. Yan, W. Yin, ARock: an algorithmic framework for asynchronous parallel coordinate updates, SIAM J. Sci. Comput. 38 (5) (2016) A2851–A2879.
[21] Z. Peng, Y. Xu, M. Yan, W. Yin, On the convergence of asynchronous parallel iteration with arbitrary delays, in: Tech. report, U.C. Los Angeles, 2016.
[22] F. Robert, M. Charnay, F. Musy, Itérations chaotiques série-parallèle pour des équations non-linéaires de point fixe, Apl. Mat. 20 (1) (1975) 1–38.
[23] M. Si, A.J. Pea, J. Hammond, P. Balaji, M. Takagi, 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, in: IPDPS '15, IEEE Computer Society, Washington, DC, USA, 2015, pp. 665–676.

Jordi Wolfson-Pou is a Ph.D. student in the School of Computational Science and Engineering at Georgia Institute of Technology. He received a B.S. in physics and a B.A. in mathematics at the University of California, Santa Cruz. His research focuses on using iterative methods for solving large sparse linear systems on high-performance computers.

Edmond Chow is an Associate Professor in the School of Computational Science and Engineering at Georgia Institute of Technology. He previously held positions at D. E. Shaw Research and Lawrence Livermore National Laboratory. His research is in developing numerical methods specialized for high-performance computers, and applying these methods to solve large-scale scientific computing problems. Dr. Chow was awarded the 2009 ACM Gordon Bell prize and the 2002 U.S. Presidential Early Career Award for Scientists and Engineers (PECASE). He serves as Associate Editor for ACM Transactions on Mathematical Software and previously served as Associate Editor for SIAM Journal on Scientific Computing.