Convergence Models and Surprising Results for the Asynchronous Jacobi Method

Jordi Wolfson-Pou
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia, United States of America
jwp3@gatech.edu

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia, United States of America
echow@cc.gatech.edu

***Abstract*—Asynchronous iterative methods for solving linear systems have been gaining attention due to the high cost of synchronization points in massively parallel codes. Since future parallel computers will likely achieve exascale performance, synchronization may become the primary bottleneck. Historically, theory on asynchronous iterative methods has focused on proving that an asynchronous version of a fixed point method will converge. Additionally, some theory has shown that asynchronous methods can be faster, which has been supported by shared memory experiments. However, it is still not well understood how much faster asynchronous methods can be than their synchronous counterparts, and distributed memory experiments have shown mixed results. In this paper, we introduce a new way to model asynchronous Jacobi using propagation matrices, which are similar in concept to iteration matrices. With this model, we show how asynchronous Jacobi can converge faster than synchronous Jacobi. We also show that asynchronous Jacobi can converge when synchronous Jacobi does not. We compare model results to shared and distributed memory implementation results, and show that in practice, asynchronous Jacobi's convergence rate improves as we increase the number of processes.**

***Index Terms*—Jacobi, Gauss-Seidel, Asynchronous, Remote memory access**

I. Introduction

Modern supercomputers are continually increasing in core count and will soon achieve exascale performance. The U.S. Department of Energy has released several reports underlining the primary problems that will arise for exascale-capable machines, one of which is the negative impact synchronization will have on program execution time [1]–[4], [18]. 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. Each row 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, point-to-point communication is used, so processes are generally idle for some period of time while they wait to receive information.

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 synchronizing. 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 [17]. Since then, it has been shown both analytically and experimentally that asynchronous methods can be faster than synchronous methods. However, there are still open questions about how fast asynchronous methods can be compared to their synchronous counterparts, and if they can be efficiently implemented in distributed memory.

In this paper, we express asynchronous Jacobi as a sequence of propagation matrices, which are similar in concept to iteration matrices. With this model, we show that if some processes are delayed, which, for example, may be due to hardware malfunctions or imbalance, iterating asynchronously can result in considerable speedup, even if some delays are long. We also show that asynchronous Jacobi's convergence rate improves as the number of processes increases, and that it is possible for asynchronous Jacobi to converge when synchronous Jacobi does not. We demonstrate these results through shared and distributed memory experiments.

II. Background

A. The Jacobi and Gauss-Seidel Methods

The synchronous Jacobi method is an example of a *stationary iterative* method, for solving the linear system $Ax = b$ [25]. A general stationary iterative method can be written as

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

where $B \in \mathbb{R}^{n \times n}$ is the *iteration matrix* and the iterate $x^{(k)}$ is started with an initial approximation $x^{(0)}$. We define the update of the i^{th} component from $x_i^{(k)}$ to $x_i^{(k+1)}$ as the *relaxation* of row i. An *iteration* is the relaxation of all rows.

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

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

The matrix B is known as the *iteration matrix*. Let $\lambda_1, \ldots, \lambda_n$ be the n eigenvalues of B, and define the *spectral radius* $\rho(B)$ of B to be $\max_{1 \leq i \leq n}(|\lambda_i|)$. It is well known that a stationary iterative method will converge to the exact solution 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 norm if $\rho(B) < 1$ since $\rho(B) \leq \|B\|$. If B is not normal, $\|B\|$ can be ≥ 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. Therefore, Equation 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 Equation 2, we can write

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

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

For the Gauss-Seidel method, $M = L$, where L is the lower triangular part of A, and for the Jacobi method, $D = M$, where D is the diagonal part of A. The Gauss-Seidel is an example of a multiplicative relaxation method, and Jacobi is an example of an additive relaxation method. For the remainder of this paper, we will only consider symmetric A and assume A is scaled to have unit diagonal values. In this case $B = C$, and the Jacobi iteration matrix as G. 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 slower than Gauss-Seidel.

B. The Asynchronous Jacobi Method

We now consider a general model of asynchronous Jacobi as presented in Chapter 5 of [6]. For simplicity, let us consider n processes, i.e., one process per row of A. Jacobi defined by Equation 3 can be thought of as *synchronous*. In particular, all elements of $x^{(k)}$ must be relaxed before iteration $k+1$ starts. Removing this requirement results in asynchronous Jacobi, where each process relaxes its row using what ever information is available. Asynchronous Jacobi can be written element-wise as

$$x_i^{(t_{k+1})} = \begin{cases} \displaystyle\sum_{j=0}^{n} G_{ij} x_j^{(s_{ij}(k))} + b_i, & \text{if } i \in \Psi(k), \\ x_i^{(k)}, & \text{otherwise.} \end{cases} \tag{5}$$

The set $\Psi(k)$ is the set of rows that have written to $x^{(k)}$ at k. The mapping $s_{ij}(k)$ denotes the components of other rows that i has read from memory.

We make the following assumptions about asynchronous Jacobi:

1) As $k \to +\infty$, $s_{ij}(k) \to +\infty$. In other words, rows will eventually read new information from other rows.
2) As $k \to +\infty$, the number of times i appears in $\Psi(k) \to +\infty$. This means that all rows eventualy relax in a finite amount of time.
3) All rows are relaxed at least as fast as in the synchronous case.

III. Related Work

An overview of asynchronous iterative methods can be found in Chapter 5 of [6]. Reviews of the convergence theory for asynchronous iterative methods can be found in [6], [11], [12], [17]. Asynchronous iterative methods were first introduced as *chaotic relaxation* methods by Chazan and Miranker [14]. 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 [7]–[9], [23], [24]. Additionally, there are papers that show that asynchronous methods can converge faster than their synchronous counterparts [10], [19]. In [19], 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 [10], and was extended to contraction maps. The speedup of asynchronous Jacobi was studied in [22] for random 2×2 matrices. Several models are used to model asynchronous iterations in different situations. The main result is that most of the time, asynchronous iterations do not improve the convergence compared with synchronous, which we do not find surprising. This result is specific for 2×2 matrices, and we will discuss in Section IV-C why speedup is not often suspected in this case.

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 [7], [13]. 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" [16], which may refer to one-sided MPI. They showed that asynchronous "eager" Jacobi can converge in fewer relaxations and 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 distributed implementations of their "racy" scheme, where a process uses what ever information is available, regardless of whether it has already been used [13]. This is the scheme we consider in our paper, which was the original method defined by Baudet. The results in [13] show that asynchronous Jacobi implemented with MPI was faster in terms of wall-clock time except for the experiments with the largest core counts. A limitation of Bethune et al.'s implementations was that it used point-to-point communication, which means that processes have to dedicate time to receiving messages and completing sent messages. The authors found this to be a bottleneck. Additionally, the authors imposed a requirement that all information must be communicated atomically, which has additional overhead. We also note that some research has been dedicated to supporting portable asynchronous communication for MPI, including the JACK API [21], and Casper, which allows asynchronous progress control [26]. We are not using either of these tools in our implementations.

IV. A New Model For Asynchronous Jacobi

A. Mathematical Formulation

For simplicity, assume $s_{1j}(k) = s_{2j}(k) = \cdots = s_{(n-1)j}(k) = s_{nj}(k)$ for $j = 1, \ldots, n$ in Equation 5, i.e., all rows always have the same information. Also, assume that $s_{ij}(k)$ always maps to the iteration number corresponding to the most up-to-date information, i.e., processes always have exact information. Our model takes Equation 5 and writes it in matrix form as,

$$x^{(t_{k+1})} = (I - \hat{D}^{(k)}A)x^{(k)} + \hat{D}^{(k)}b \tag{6}$$

where

$$\hat{D}^{(k)} = \begin{cases} 1, & \text{if } i \in \Psi(k), \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

Similar to the iteration matrix, we define the error and residual *propagation matrices* as

$$\hat{G}^{(k)} = I - \hat{D}^{(k)}A, \quad \hat{H}^{(k)} = I - A\hat{D}^{(k)}, \tag{8}$$

since a fixed iteration matrix cannot be defined for asynchronous methods.

It is important to notice the structure of these matrices. For a row i that is not relaxed at time k, row i of $\hat{G}^{(k)}$ is zero except for a 1 in the diagonal position of that row. Similarly, column i of $\hat{H}^{(k)}$ is zero except for a 1 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 $\Psi(k)$ (similarly, we replace columns of G to get the residual propagation matrix).

If we remove the simplification mentioned in the first paragraph of this section, i.e., assume $s_{1j}(k) \neq s_{2j}(k) \neq \cdots \neq s_{(n-1)j}(k) \neq s_{nj}(k)$, it is still possible to write Equation 5 in matrix form. This can be done by ordering the relaxations of rows in the following way:

- Define a new set of rows $\Phi(\ell)$ that denotes the set of rows to be relaxed in parallel at parallel step ℓ.
- Let κ_i be the number number of relaxations that row i has carried out at parallel step ℓ, where $\kappa_i = 0$ when $\ell = 0$. If i is added to $\Phi(\ell)$ at any point, κ_i is increased by one.
- At parallel step ℓ, row i is added to $\Phi(\ell)$ if
  1) all j rows have relaxed $s_{ij}(\kappa_i)$ times, i.e., all necessary information is available.
  2) $\kappa_i \geq s_{ji}(\kappa_j)$, i.e., the relaxation of row i would not result in any row j reading an old version of row i.

If these two conditions cannot be satisfied at a given parallel step, some relaxations cannot be expressed as sequence propagation matrices.

Two examples of sequences of asynchronous relaxations are shown in Figure 1 (a) and (b). In these examples, four processes, $p_1, \ldots, p_4$, are responsible for a single row and relax once in parallel. The red dots denote the time at which process p_i writes x_i to memory. The blue arrows denote the information used by other processes, e.g., in (a), p_2 uses relaxation zero (initial value) from p_1 and relaxation one from p_4. Time, or relaxation count, moves from left to right. In (a), $\kappa_i = 1$, we know the following information:

- For p_1, $s_{12}(1) = 0$ and $s_{13}(1) = 0$.
- For p_2, $s_{21}(1) = 0$ and $s_{24}(1) = 1$.
- For p_3, $s_{31}(1) = 1$ and $s_{34}(1) = 1$.
- For p_4, $s_{42}(1) = 0$ and $s_{43}(1) = 0$.

We can see that even though $s_{21}(1) \neq s_{31}(1)$, we can set $\Phi(1) = \{4\}$, $\Phi(2) = \{1, 2\}$, and $\Phi(3) = \{3\}$ to get three propagation matrices that create a correct sequence of

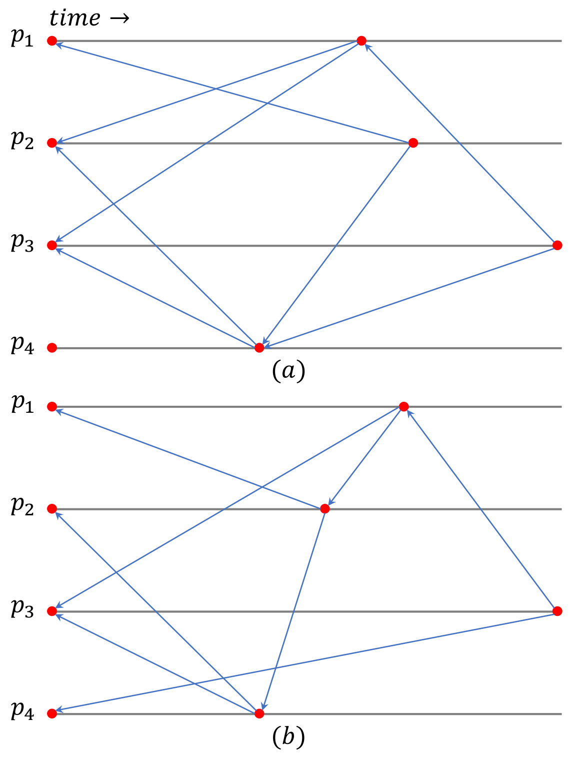

Fig. 1: Two examples of four processes carrying out one relaxation asynchronously. Relaxations are denoted by red dots, and information for a relaxation is denoted by blue arrows. In (a), it is possible to express the relaxation as a sequence of propagation matrices. In (b), this is not possible.

relaxations. Example (b) is a modification of (a), where now $s_{12}(1) = 1$ and $s_{34} = 0$. In this example, $\Phi(1) = \{4\}$ since this is the only row in which all necessary information is available (this is from the first condition above). However, this violates the second condition since $\kappa_3 > s_{43}(\kappa_4)$ after p_4 relaxes row four.

Example (b) only shows that all four relaxations cannot be expressed as a sequence of propagation matrices. However, we can ignore the second condition when we form $\Phi(1)$. We then have $\Phi(1) = 4$, $\Phi(2) = 2$, $\Phi(3) = 1$, and treat the relaxation by p_3 separately. This shows that we can express the majority of the relaxations in terms of sequences of propagation matrices. In Section VII, we will show that this is true in practice as well.

B. Connection to Inexact Multiplicative Block Relaxation

Equation 6 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 contiguous set of equations that are relaxed. 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

$$\hat{D}^{(k)} = \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{otherwise.} \end{cases} \tag{9}$$

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

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

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.

C. Asynchronous Jacobi Can Reduce The Error When Processes are Delayed

Let A be weakly diagonally dominant (W.D.D.), i.e., $|a_{ii}| \geq \sum |a_{ij}|$ for all $1 \leq i \leq n$ and thus $\rho(G) \leq 1$. Then the error and residual for 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 L1/infinity norms if the propagation matrices are bounded by 1 in these norms. A norm of 1 means that the error or residual does not grow but may still decrease. Such a 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 (not active) at time k. Then $\rho(\hat{G}^{(k)}) = \|\hat{G}^{(k)}\|_\infty = 1$ and $\rho(\hat{H}^{(k)}) = \|\hat{H}^{(k)}\|_1 = 1$.*

Proof. Let the number of processes $= n$, and let $\xi_1, \ldots, \xi_n$ be the n unit (coordinate) basis vectors. Without loss of generality, consider a single process p_i to be delayed. The proof of $\|\hat{G}^{(k)}\|_\infty = 1$ is straightforward. Since p_i is delayed, 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 elements are the diagonal elements of $\hat{G}^{(k)} - 1$ and the i^{th} row is all zeros. Since Y has a row of zeros, it must have a 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 some rows are delayed for a long time. An important consequence of Theorem 1 is that the error will not increase in the infinity norm no matter what error propagation matrix is chosen, 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 1 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 unit basis vectors is equal to the number of delayed processes. 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 [22], applying propagation matrices more than once will not change the solution since the error and residual propagation matrices have the form

$$\hat{G}^{(k)} = \begin{bmatrix} 1 & 0 \\ \alpha & 0 \end{bmatrix}, \quad \hat{H}^{(k)} = \begin{bmatrix} 1 & \beta \\ 0 & 0 \end{bmatrix}, \tag{11}$$

if the first process is delayed, where $\alpha = A_{21}/A_{11}$ and $\beta = A_{12}/A_{11}$. Both these matrices have a nullspace of dimension one, so the error will converge in one iteration to the eigenvector with eigenvalue equal to one. In other words, 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 delayed rows will reduce the error and residual.

For larger matrices, how quickly the residual converges depends on the eigenvalues that do no 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 for long delays. To gain some insight into the reduction of the error/residual, we can use the fact that the delayed components of the solution do not change with successive applications of the same propagation matrix.

As an example, consider just the first row to be delayed starting at time k. We can write the iteration as

$$x^{(k+1)} = \hat{G}^{(k)} x^{(k)} + \hat{D}^{(k)} b \tag{12}$$

$$\begin{bmatrix} x_1^{(k)} \\ y^{(k+1)} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ g & \tilde{G} \end{bmatrix} \begin{bmatrix} x_1^{(k)} \\ y^{(k)} \end{bmatrix} + \begin{bmatrix} 0 \\ c \end{bmatrix}, \tag{13}$$

where g and $y^{(k)}$ are $(n-1)\times 1$ vectors, and $\tilde{G}$ is a $(n-1)\times(n-1)$ symmetric principal submatrix of G. Since only $y^{(k)}$ changes, we can write the iteration as

$$y^{(k+1)} = \tilde{G} y^{(k)} + c + x_1 g = \tilde{G} y^{(k)} + f. \tag{14}$$

From this expression, and because f is constant, we can reduce our analysis to how quickly the error/residual corresponding to $y^{(k)}$ reduces to 0. Since $\tilde{G}$ is a principal submatrix of G, we can use the interlacing theorem to bound the eigenvalues of $\tilde{G}$ with eigenvalues of G. Specifically, the i^{th} eigenvalue μ_i of $\tilde{G}$ can be bounded as $\lambda_i \leq \mu_i \leq \lambda_{i+1}$.

For the general case in which m rows are active (not delayed), we can consider the system $PAP^T Px = Pb$, which has the iteration

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

The matrix P is a permutation matrix that is chosen such that all delayed rows are ordered first, resulting in the propagation matrix

$$\begin{bmatrix} I & O \\ g & \tilde{G} \end{bmatrix}. \tag{16}$$

where I is the $(n-m)\times(n-m)$ identity matrix, O is the $(n-m)\times m$ zero matrix, g is $m\times(n-m)$, and $\tilde{G}$ is $m\times m$. For an eigenvalue μ_i of $\tilde{G}$, $\lambda_i \leq \mu_i \leq \lambda_{i+n-m}$ 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 in the interval $(0,1)$, or if they are clustered near one, we can expect a similar spacing between the eigenvalues of $\tilde{G}$.

In summary, for W.D.D. A, we can say that asynchronous Jacobi can continually reduce the error and residual norms even with very large delays, and will never increase the norms. Additionally, since all rows will eventually relax at least as fast as synchronous Jacobi, the additional relaxations that were carried out while a process was delayed will only help reduce the error and residual norms.

D. Asynchronous Jacobi Can Converge When Synchronous Jacobi Does Not

A well known result, known as early as Chazan and Miranker [14], 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. The transient convergence behavior depends on the *norm* of the propagation matrices. Suppose $\|G\| \geq \rho(G) > 1$. In this case, synchronous Jacobi may reduce the error initially, but the error will eventually increase unbounded as $k \to \infty$.

On the other hand, the changing propagation matrices can result in asynchronous Jacobi converging when synchronous Jacobi does not. Since $\|G\|_\infty \geq \|\hat{G}^{(k)}\|_\infty \geq 1$ for any $\Psi(k)$, and since the $\|\hat{G}^{(k)}\|_\infty$ can only get smaller (or stay the same) as more rows are delayed, the error/residual can continually be reduced if enough rows are delayed at each iteration. If many rows have the W.D.D. property, then it may be that $\|G^{(k)}\|_\infty = \|H^{(k)}\|_1 = 1$ happens quite often when enough rows are delayed, which means that the error and residual are reduced with high probability.

This is also apparent by looking at principal submatrices of G (Equation 16), as we did previously. Since we know that $\rho(\tilde{G}) \leq \rho(G)$ by the interlacing theorem, we can say that $\|\tilde{G}\|_2 \leq \|G\|_2$ since $\tilde{G}$ is symmetric, and $\|\tilde{G}\|_1 \leq \|G\|_1$ since we are removing rows and columns from G (equivalently, $\|\tilde{G}\|_\infty \leq \|G\|_\infty$). If enough rows are delayed, these submatrices can be very small, resulting in a significantly smaller $\rho(\tilde{G})$. Returning to the discussion from Section IV-B, it is important to note that if our matrix is sparse, $\tilde{G}$ 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 $\rho(\tilde{D}) \leq \rho(\tilde{G})$, where $\tilde{D}$ is a diagonal block of $\tilde{G}$. If many processes are used, it may happen that $\tilde{G}$ will have many blocks, resulting in $\rho(\tilde{D}) << \rho(\tilde{G})$. This can explain why increasing the concurrency can result in asynchronous Jacobi converging faster than synchronous Jacobi, and converging when synchronous Jacobi does not. This is a result we will show experimentally.

In summary, asynchronous Jacobi can converge when synchronous Jacobi does not, given appropriate sequences of error and residual propagation matrices are chosen. Additionally, increasing the concurrency can improve the convergence rate of asynchronous Jacobi.

V. 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 solution. In particular, a single step of both synchronous and asynchronous Jacobi can be written as

1) compute the residual $r = b - Ax$.
2) correct the solution $x = x + D^{-1}r$.

3) check for convergence (detailed below and in Section VI for distributed memory).

Each thread or process is responsible for some number of rows of r and x, so it only computes Ax, r and x for said rows. The contiguous rows that a process or thread is responsible for is defined as its *subdomain*.

OpenMP was used for our shared memory implementation. The vectors x and r are stored in shared arrays. The only difference between the asynchronous and synchronous implementations is that synchronous requires a barrier after steps 1) and 3). Since each element in either x or r is updated by writing to memory (not incrementing), 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.

Convergence is achieved if the relative norm of the global residual falls below a specified tolerance, or if all threads have carried out a specified number of iterations. Each thread computes a local norm of the shared residual array. A thread terminates only if all other threads have also converged. To ensure this, a shared array of flags is used, which has a length equal to the number of threads and is initialized to all zeros. When a thread converges, it writes a one to its place in the array. All threads take the sum of the array after relaxing their rows to determine if all threads have converged. To test for convergence in asynchronous Jacobi, each thread computes the norm of r. For synchronous Jacobi, all threads always have the same iteration count, and can compute a global residual norm with a parallel reduction.

VI. Implementing Asynchronous Jacobi in Distributed Memory

The program structure is the same as that of the shared memory implementation as described in the first paragraph of Section V. However, there are no shared arrays. Instead, each process sends messages containing values of x to its neighbors. A neighbor of process p_i is determined by inspecting the non-zero values of the matrix rows of p_i. If the index of a value is in the subdomain of a different process p_j, then p_j is a neighbor of p_i. During a SpMV, p_i requires these points in x from p_j, so p_j sends these points to p_i, i.e., p_i always locally stores a *ghost layer* of points that p_j sent to p_i previously.

We used MPI for communication in our distributed implementations. The communication of ghost layer points was done with point-to-point communication for the synchronous implementation. In point-to-point, both the sending process, or *origin*, and the receiving process, or *target*, 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.

For our asynchronous implementation, remote memory access (RMA) communication was used [5]. 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 subarrays do not overlap so that race conditions do not occur. To initialize an access epoch on a remote memory window, `MPI_Win_lock_all()` was used, which allows access to windows of all processes until `MPI_Win_unlock_all()` is called. We found that this was faster than locking and unlocking individual windows using `MPI_Win_lock()` and `MPI_Win_unlock()`. Writing to the memory of a remote window was done using `MPI_Put()`. 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. 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 need by other rows.

For our implementation, 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. This is not true in general for the asynchronous case, where some processes can terminate even when other processes are still iterating. This naive scheme requires no communication. If it is desired that some global criteria 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 latter topic for future research.

VII. Results

A. Test Framework

All experiments were run on either NERSC's Cori supercomputer or a single node with two 10-core Intel Xeon E5-2650 CPUs (2 hyperthreads per core) housed at Georgia Institute of Technology. On Cori, shared memory experiments were run on an Intel Xeon Phi Knights Landing (KNL) processor with 68 cores and 272 threads (4 hyperthreads per core), and distributed 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 Haswell node. We used a random initial approximation $x^{(0)}$ and right-hand side b in the range [-1,1], and the following test matrices:

1) Matrices arising from a five-point centered difference discretization of the Laplace equation on a rectangular domain with uniform spacing between points. These matrices are irreducibly W.D.D., symmetric positive definite, and $\rho(G) < 1$. We will refer to these matrices as FD.
2) An unstructured finite element discretization of the Laplace equation on a square domain. The matrix is not W.D.D., but approximately half the rows have the W.D.D. property. The matrix is symmetric positive definite, and $\rho(G) > 1$. We will refer to this matrix as FE.

3) Matrices taken from the SuiteSparse matrix collection as shown in Table I [15].

Matrices were partitioned using METIS [20], and are stored in compressed sparse row (CSR) format.

TABLE I: Test problems from the SuiteSparse Matrix Collection. All matrices are symmetric positive definite.

Matrix	Non-zeros	Equations
thermal2	8,579,355	1,227,087
G3_circuit	7,660,826	1,585,478
ecology2	4,995,991	999,999
apache2	4,817,870	715,176
parabolic_fem	3,674,625	525,825
thermomech_dM	1,423,116	204,316
Dubcova2	1,030,225	65,025

B. Asynchronous Jacobi Model and Shared Memory

The primary goal of this section is to validate the *model* of asynchronous Jacobi presented in Section IV by comparing its behavior to actual asynchronous Jacobi computations carried out using an OpenMP implementation. The model is a mathematical simplification of actual asynchronous computations, ignoring many factors that are hopefully not salient to the convergence behavior of the asynchronous method. We execute the model using a sequential computer implementation.

For our first experiment, we look out how likely asynchronous relaxations can be expressed as a sequence of propagation matrices. We took a history of asynchronous relaxations from an OpenMP experiment and looked out how many of the relaxations could be expressed in terms of propagation matrices. For each row i, we printed the solution components that i read from other rows for each relaxation of i, and used this information to construct a sequence of propagation matrices based on the two conditions from Section IV-A. This is done using the same process described in the two examples in Section IV-A. We say that if a propagation matrix can be constructed that satisfies the two conditions, then all relaxations carried out via the application of that matrix are defined as “propagated” relaxations. If the second condition is not satisfied, then any subsequent relaxation that uses old information is not counted as a propagated relaxation.

Figure 2 shows the fraction of propagated rows as the number of threads increases for the Phi and CPU. For the Phi, the test matrix is an FD matrix with 272 rows and 1294 non-zero values, and the number of threads used are 17, 34, 68, 136, and 272. For the CPU, the test matrix is an FD matrix with 40 rows and 174 non-zero values, and the number of threads used are 5, 10, 20, and 40. The figure shows that the majority of relaxations can be correctly expressed via the application of propagation matrices. In particular, in the worst case (Phi with 34 threads), $\approx .8$ of the relaxation are propagated, and in the best case (CPU with 40 threads), $\approx .99$ of the relaxations are propagated. We can see that as the number of threads increases, the fraction also increases. This indicates that analyzing real asynchronous experiments using the idea of propagation matrices is more appropriate when the number of rows per thread is small. Therefore, looking towards solving large problems on exascale machines, propagation matrix analysis may be useful since the enormous concurrency may result in a small number of rows per process.

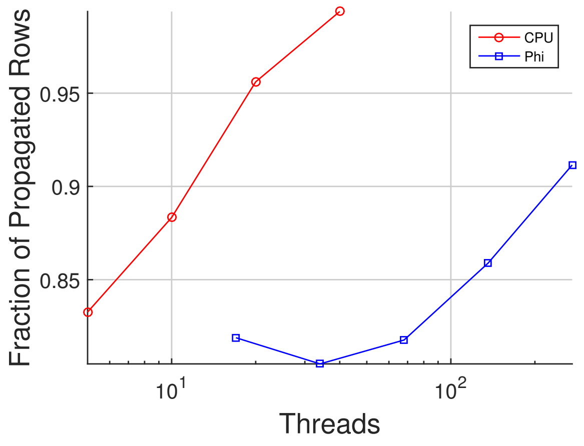

Fig. 2: Fraction of propagated rows as a function of number of threads for the CPU and Phi. For the Phi, the test matrix is an FD matrix with 272 rows and 1294 non-zero values. For the CPU, the test matrix is an FD matrix with 40 rows and 174 non-zero values.

For our next set of experiments, we look at how the model and our OpenMP implementation compares with the synchronous case. For our first experiment in this set, we consider the scenario where all threads run at the same speed, except one thread which runs at a slower speed. This could simulate a hardware problem associated with one thread. We assign a delay δ to the thread p_i corresponding to row i near the middle of a test matrix. For the OpenMP implementation, the delay corresponds to having p_i sleep for a certain number of microseconds. Since synchronous Jacobi uses a barrier, all threads have to wait for p_i to finish sleeping and relaxing its rows before they can continue. For the model, time is in unit steps, and δ is the number of those steps that row i is delayed by. In the asynchronous case, row i only relaxes at multiples of δ, while all other rows relax at every time step. In the synchronous case, all rows relax at multiples of δ to simulate waiting for the slowest process.

We first look at how much faster asynchronous Jacobi can be compared to synchronous Jacobi when we vary the delay parameter δ. The test matrix is 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. A relative residual norm tolerance of .001 is used. For OpenMP, we varied δ from zero to 3000 microseconds, and recorded the mean wall-clock time for 100 samples for each delay. For the model, we varied δ from zero to 100. Figure 3 shows the speedup for the model and for actual asynchronous OpenMP computations as a function of the delay parameter. The speedup for OpenMP is defined as the total wall-clock time of synchronous Jacobi divided by the total wall-clock time of asynchronous Jacobi.

Similarly, for the model, the speedup is defined as the total model time of synchronous Jacobi divided by the model time of asynchronous Jacobi.

Figure 3 shows a qualitative and quantitative agreement between the model and actual computations. Both achieve a speedup above 40 before plateauing. In general, this speedup depends on the problem, the number of threads, and which threads are delayed at each iteration, which all affect convergence. Note that without artificially slowing down a thread, actual asynchronous Jacobi computations are still slightly faster, as shown by values corresponding to 0 delay. This is due to the fact that natural delays occur that make some threads faster than others. For example, maintaining cache coherency while many threads are writing to and reading from a shared array can cause natural delays.

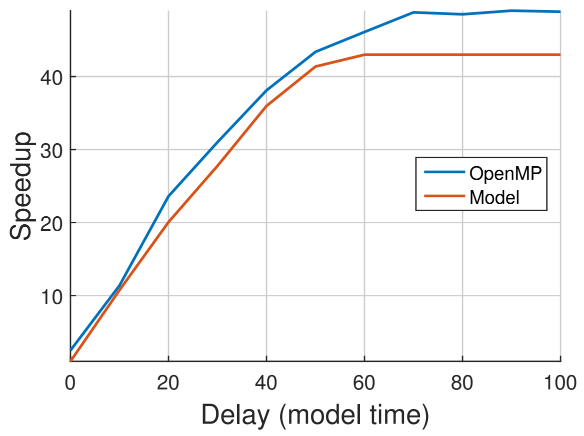

Fig. 3: Speedup of asynchronous over synchronous computations for 68 threads as a function of the delay δ experienced by one thread. The result of asynchronous OpenMP computations is compared to the result predicted by the model. The test problem is an FD matrix with 68 rows and 298 nonzeros.

Figure 4 shows the relative residual 1-norm as a function of the model and wall-clock times. Results for synchronous and asynchronous Jacobi are plotted with different delays. The figure shows that the model approximates the behavior of the OpenMP results quite well. A major similarity is the convergence curves for the two largest delays. For both the model and OpenMP, 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 asynchronous Jacobi. For the second largest delay, we see a "saw tooth"-like pattern corresponding to asynchronous Jacobi no longer reducing the residual. The existence of this pattern for both the model and OpenMP further confirms the accuracy of our model. Most importantly, we again see that with no delay, asynchronous Jacobi converges faster.

Figure 5 shows how asynchronous Jacobi scales when increasing the number of threads from one to 272, and without adding any delay. For these results, we used an FD matrix with 4624 rows (17 rows per thread in the case of 272 threads) and 22,848 non-zero values. This small matrix was chosen such that most of the time was spent writing/reading from memory rather than computing, i.e., communication time outweighs computation time. As in the previous set of results, we averaged the wall-clock time of 100 samples for each data point.

Figure 5 (a) shows the wall-clock time for achieving a relative residual norm below .001. First, asynchronous Jacobi can be over 10 times faster when many threads are used. More importantly, we can see that asynchronous Jacobi is the fastest when using 272 threads, while synchronous Jacobi is fastest when using fewer than 272 threads. Figure 5 (b) shows the wall-clock time for carrying out 100 iterations regardless of what relative residual norm is achieved. As explained in Section VII-A, a thread only terminates once all threads have completed 100 iterations. This means that asynchronous Jacobi will carry out more iterations per thread in this case. The figure shows that although using 272 threads does minimize the wall-clock time for asynchronous Jacobi, it is still faster than synchronous Jacobi. This indicates that synchronization points have a higher cost than reading from and writing to memory, which is more abundant in asynchronous Jacobi.

For 272 threads, both Figures 5 (a) and (b) demonstrate an important property of asynchronous Jacobi: even if asynchronous Jacobi is slower per iteration as the number of threads increases (in this case, the combined computation and communication time of 272 threads is higher than 136 threads, per iteration), the convergence rate of asynchronous Jacobi can accelerate when increasing the number of threads (achieving a relative residual norm below .001 is faster at 272 threads than at 136 threads). This can be explained by the fact that multiplicative relaxation methods are often faster than Jacobi, and increasing the number of threads results in asynchronous Jacobi behaving more like a multiplicative relaxation scheme.

For our final shared memory experiment, we look at a case in which asynchronous Jacobi converges when synchronous Jacobi does not. We use an FE matrix with 3,081 rows and 20,971 non-zero values. Figure 6 (a) shows the residual norm as a function of the number of iterations. For asynchronous Jacobi, the number of iterations is the average number of local iterations carried out by all the threads. We can see that as we increase the number of threads to 272, asynchronous Jacobi starts to converge. This shows that the convergence rate of asynchronous Jacobi can be dramatically improved by increasing the amount of concurrency, even to the point where asynchronous Jacobi will converge when synchronous Jacobi does not. Figure 6 (b) shows that asynchronous Jacobi truly converges, and does not diverge at some later time.

C. *Asynchronous Jacobi in Distributed Memory*

The purpose of this section is to see if we can produce results in distributed memory that have a similar behavior to that of the shared memory case. In particular, can asynchronous Jacobi before faster than synchronous Jacobi, and can it converge when synchronous Jacobi does not. We look at how asynchronous Jacobi compares with synchronous Jacobi

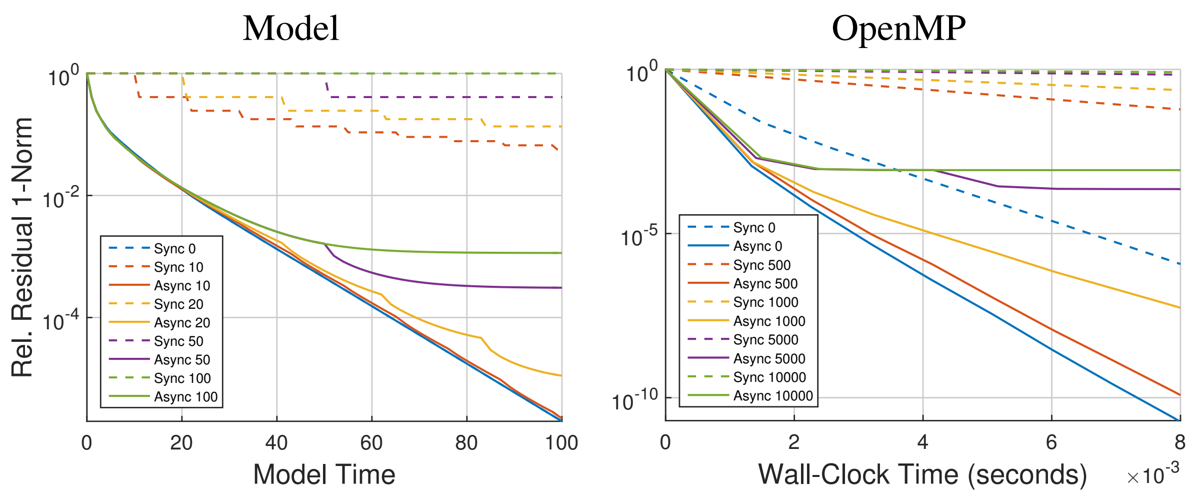

Fig. 4: Relative residual norm as a function of model time and wall-clock time (seconds) for the model and asynchronous OpenMP computations. Convergence from using different delays is shown. The test problem is an FD matrix with 68 rows and 298 nonzeros.

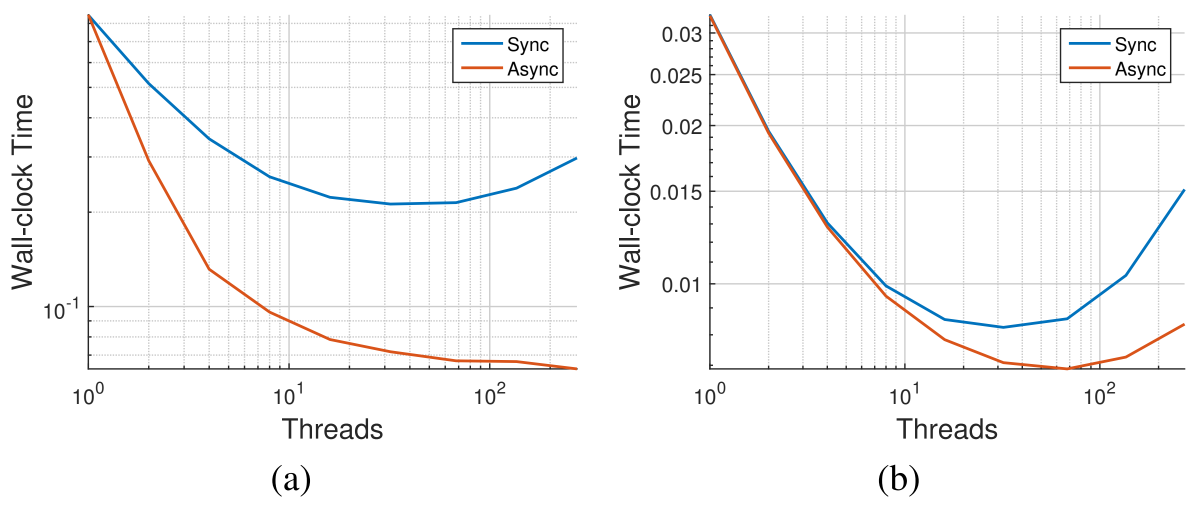

Fig. 5: Asynchronous compared with synchronous Jacobi as the number of threads increases. Plot (a) shows wall-clock time when both methods achieve a relative residual norm below .001 upon convergence. Plot (b) shows how much time is taken to carry out 100 iterations, regardless of the tolerance on the relative residual norm. 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.

for the problems in Table I. These problems were selected because they are the largest SPD matrices in the SuiteSparse collection for which Jacobi converges, with the exception of Dubcova2, which does not converge with Jacobi. We carried out 200 runs per number of MPI processes per matrix, and took the mean wall-clock time. Since our convergence termination does not allow us to terminate based on the residual norm, we used interpolation on the wall-clock time curves. In particular, to measure wall-clock times for a specific residual norm, linear interpolation on the $\log_{10}$ of the relative residual norm was used.

Figure 7 shows the relative residual norm as a function of relaxations for six problems (not including Dubcova2). The plots are organized such that the smallest problem is shown first (thermomech_dM), and the problem size increases along the first row and then the second row. Since the amount of concurrency affects the convergence of 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, asynchronous Jacobi tends to converge in fewer relaxations. More importantly, as the number of nodes increases, convergence is improved. This is more prevalent for smaller problems, especially thermomech_dM, where subdomains are smaller. As explained earlier, using smaller subdomains increases the likelihood that the convergence will behave similarly to a multiplicative relaxation method. This is because if a snapshot

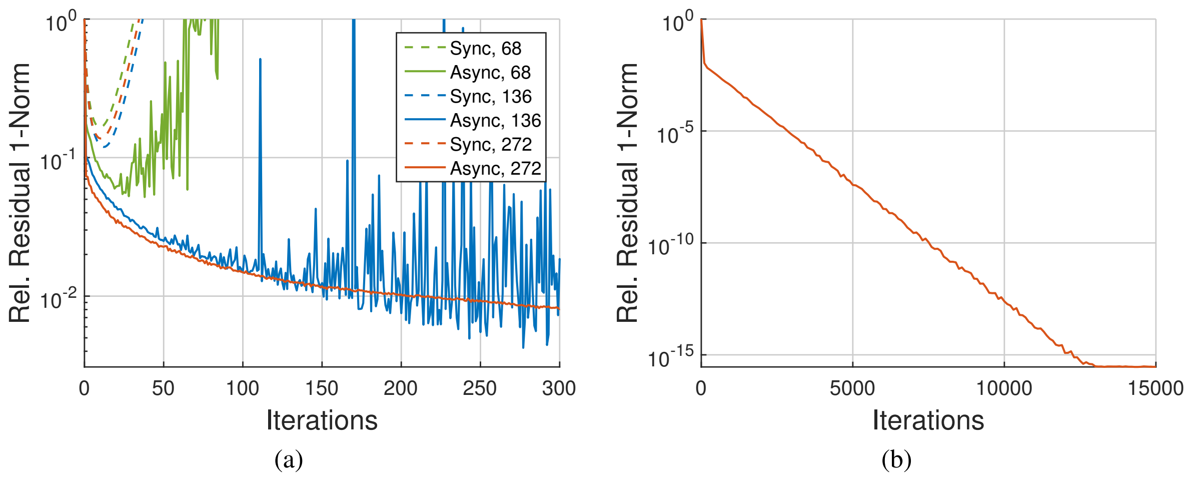

Fig. 6: In plot (a), relative residual norm as a function of iterations for different numbers of threads (68, 136, and 272). The plot shows that using more threads can improve the convergence rate of asynchronous Jacobi to the point where asynchronous Jacobi converges when synchronous Jacobi does not. Plot (b) shows that asynchronous Jacobi using 272 threads truly converges. The test problem is an FE matrix with 3,081 rows and 20,971 non-zero values.

is taken at some point in time, it is less likely that coupled points are being relaxed simultaneously.

Figure 8 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 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 2,048 and 4,096 MPI processes are lower than for 1,024. We suspect that we would see the same effect in the cases of parabolic_fem and apache2 if more processes were used. In general, we can see that asynchronous Jacobi is faster than synchronous Jacobi.

Improving the convergence with added concurrency is most dramatic in Figure 9, where the relative residual norm as a function of number of relaxations is shown for Dubcova2. This behavior is similar to that in Figure 6, where increasing the number of threads allowed asynchronous Jacobi to converge when synchronous Jacobi did not. In general, asynchronous Jacobi is faster in terms of convergence rate and wall-clock time.

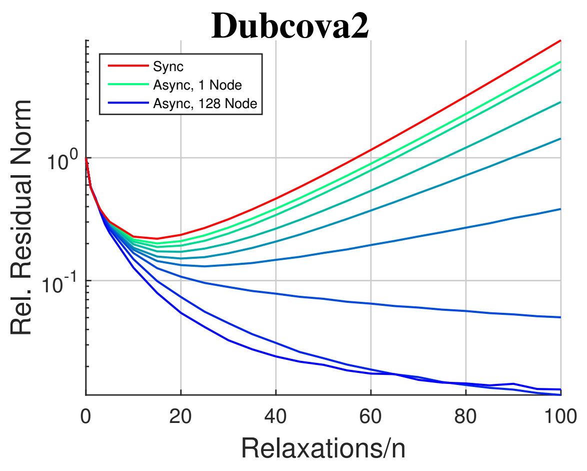

Fig. 9: For Dubcova2, relative residual norm as a function of relaxations/n for synchronous and asynchronous Jacobi. As in Figure 6, increasing the number of processes improves the convergence rate of asynchronous Jacobi.

VIII. Conclusion

The transient convergence behavior of asynchronous iterative methods has been well-understood. In this paper, we presented a new model where we expressed convergence in terms of propagation matrices. With this model, we showed that, under certain conditions, even when a process is severely delayed, asynchronous Jacobi can still reduce the error and residual, and, even if the delay is not severe, asynchronous Jacobi is still faster than synchronous Jacobi. Additionally, we showed that if the right sequence of propagation matrices is chosen, asynchronous Jacobi can converge when synchronous does not. We verified our model with shared and distributed memory experiments. These experiments indicate that increasing the number of processes or threads can actually accelerate the convergence of asynchronous Jacobi.

Efficiently implementing asynchronous methods in distributed memory has been a challenge because support for high performance one-sided communication (with passive target completion) has not always been available. This paper has also briefly described an example of how to implement asynchronous methods efficiently using MPI on the Cori supercomputer.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced

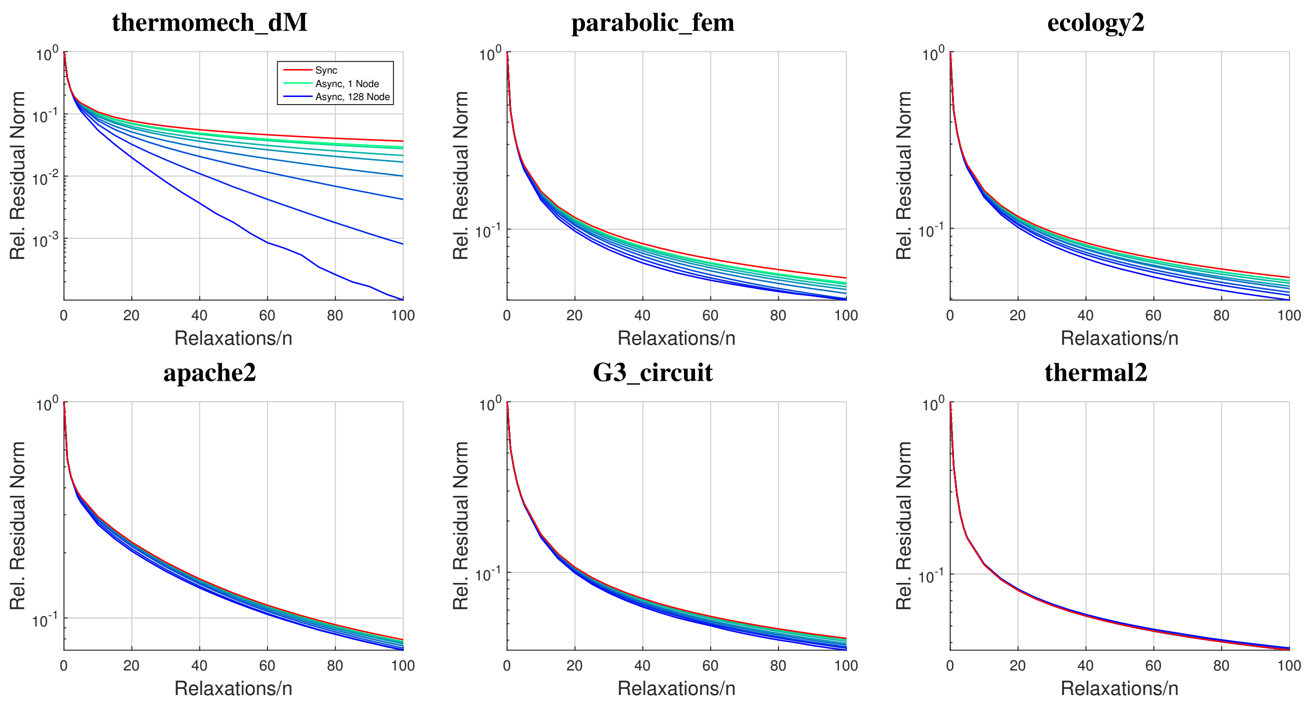

Fig. 7: Relative residual norm as a function of relaxations/n for synchronous and asynchronous Jacobi. For asynchronous Jacobi, 1 to 128 nodes are shown, where the green to blue gradient represents increasing numbers of nodes. Results for six test problems are shown.

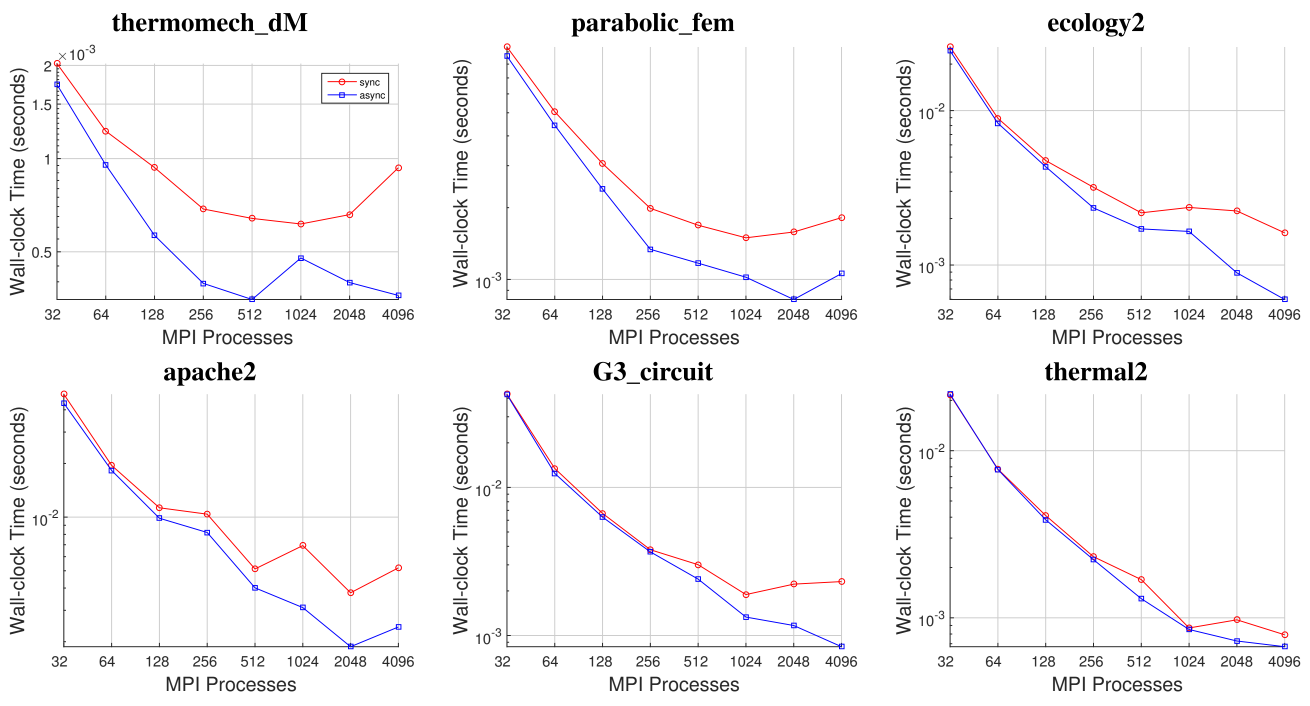

Fig. 8: Wall-clock time in seconds as a function of MPI processes for synchronous and asynchronous Jacobi. Results for six test problems are shown.

Scientific Computing Research, Applied Mathematics program 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.

REFERENCES

[1] *Scientific grand challenges: Architectures and technology for extreme scale computing*, tech. rep., U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, December 2009.

[2] *Scientific grand challenges: Crosscutting technologies for computing at the exascale*, tech. rep., U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, February 2010.

[3] *Exascale and beyond: Configuring, reasoning, scaling*, tech. rep., U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, August 2011.

[4] *Exascale programming challenges*, tech. rep., U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, July 2011.

[5] *MPI: Message Passing Interface Standard, Version 3.0*, High-Performance Computing Center Stuttgart, September 2012.

[6] J. M. BAHI, S. CONTASSOT-VIVIER, AND R. COUTURIER, *Parallel Iterative Algorithms: From Sequential to Grid Computing*, Chapman & Hall/CRC, 2007.

[7] G. BAUDET, *Asynchronous iterative methods for multiprocessors*, Journal of the ACM, 25 (1978), pp. 226–244.

[8] B. F. BEIDAS AND G. P. PAPAVASSILOPOULOS, *Convergence analysis of asynchronous linear iterations with stochastic delays*, Parallel Computing, 19 (1993), pp. 281 – 302.

[9] D. BERTSEKAS, *Distributed asynchronous computation of fixed points*, Mathematical Programming, 27 (1983), pp. 107–120.

[10] D. BERTSEKAS AND J. N. TSITSIKLIS, *Convergence rate and termination of asynchronous iterative algorithms*, in Proceedings of the 3rd International Conference on Supercomputing, ICS '89, New York, NY, USA, 1989, ACM, pp. 461–470.

[11] ———, *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[12] D. BERTSEKAS AND J. N. TSITSIKLIS, *Some aspects of parallel and distributed iterative algorithms: A survey*, Automatica, 27 (1991), pp. 3 – 21.

[13] 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, 28 (2014), pp. 97–111.

[14] D. CHAZAN AND W. MIRANKER, *Chaotic relaxation*, Linear Algebra and its Applications, 2 (1969), pp. 199 – 222.

[15] T. DAVIS AND Y. HU, *The University of Florida sparse matrix collection*, ACM Transactions on Mathematical Software, 38 (2011), pp. 1:1–1:25.

[16] D. DE JAGER AND J. BRADLEY, *Extracting State-Based Performance Metrics using Asynchronous Iterative Techniques*, Performance Evaluation, 67 (2010), pp. 1353–1372.

[17] A. FROMMER AND D. SZYLD, *On asynchronous iterations*, Journal of Computational and Applied Mathematics, 123 (2000), pp. 201 – 216.

[18] E. M. W. GROUP, *Applied mathematics research for exascale computing*, tech. rep., U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, March 2014.

[19] 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, 11 (1997), pp. 1–11.

[20] G. KARYPIS AND V. KUMAR, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392.

[21] F. MAGOULÈS AND G. GBIKPI-BENISSAN, *JACK: an asynchronous communication kernel library for iterative algorithms*, The Journal of Supercomputing, 73 (2017), pp. 3468–3487.

[22] A. C. MOGA AND M. DUBOIS, *Performance of asynchronous linear iterations with random delays*, in Proceedings of International Conference on Parallel Processing, April 1996, pp. 625–629.

[23] Z. PENG, Y. XU, M. YAN, AND W. YIN, *Arock: An algorithmic framework for asynchronous parallel coordinate updates*, SIAM Journal on Scientific Computing, 38 (2016), pp. A2851–A2879.

[24] Z. PENG, Y. XU, M. YAN, AND W. YIN, *On the convergence of asynchronous parallel iteration with arbitrary delays*, tech. rep., U.C. Los Angeles, 2016.

[25] Y. SAAD, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, PA, USA, 2nd ed., 2003.

[26] M. SI, A. J. PEA, 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, IPDPS '15, Washington, DC, USA, 2015, IEEE Computer Society, pp. 665–676.