ASYNCHRONOUS SEMI-ITERATIVE METHODS AND THE ASYNCHRONOUS CHEBYSHEV METHOD WITH MULTIGRID PRECONDITIONING*

JORDI WOLFSON-POU† AND EDMOND CHOW‡

Abstract. This paper considers solving linear systems of equations by asynchronous versions of semi-iterative methods which involve one or more parameters. Semi-iterative methods can converge very rapidly when these parameters are chosen well. We first observe that the best parameters for an asynchronous semi-iterative method can be quite different from the best parameters for the corresponding standard, synchronous method. We then present an asynchronous version of the Chebyshev semi-iterative method. As a second-order method, the Chebyshev method is more sensitive to asynchronous execution than first-order semi-iterative methods. This sensitivity can be reduced by different choices of its parameters (e.g., when underestimating the spectrum of the matrix), and also by preconditioning. This motivates a major part of this paper, which is the development of an asynchronous additive multigrid preconditioner for the asynchronous Chebyshev method that is based on iteratively solving an extended, semidefinite system. We demonstrate a shared memory parallel implementation that uses this extended matrix efficiently in matrix-free fashion.

Key words. asynchronous iterative methods, Chebyshev iterative method, multigrid

MSC codes. 65F10, 65F08, 65Y05, 65N55

1. Introduction.

An *asynchronous* iterative method is a parallel method for solving the system of equations $x = G(x)$ via a fixed-point iteration such that processors, which are each assigned to update a subset of the components of x, do not synchronize at the end of each iteration. The processors, instead, proceed without waiting for other processors, using the latest values of the components of x available to them. This could be advantageous compared to standard synchronous iterations when communication is slow or when there is load imbalance.

Such iterative methods that neglect synchronization were introduced in the late 1960's, at a time when parallel computing for numerical methods was still nascent. Jack Rosenfeld at IBM first suggested and experimented with solving linear systems with the Gauss-Seidel method without synchronizations, which was called "chaotic relaxation" [44, 45]. As there were relatively few multiprocessing systems at the time, Rosenfeld *simulated* the execution of chaotic relaxation running on a simplified IBM System/360 architecture with 16 processors. His work contemplated that neglecting synchronization was important in order for multiprocessors "to achieve high speed in the solution of a single problem" and showed examples of this with his simulations. Soon after, Daniel Chazan and Willard Miranker, colleagues of Rosenfeld at IBM, published their landmark paper [15] that initiated the development of mathematical convergence analysis for chaotic relaxation. Early influential papers, which extended the study of chaotic relaxation to nonlinear fixed-point iterations, include those of Miellou [39] and of Robert, Charnay, and Musy [43]. Baudet [8] introduced a generalization of chaotic relaxation by removing the assumption of bounded delays,

*Submitted to the editors June 15, 2024. Accepted December 2, 2024, *SIAM Journal on Scientific Computing*.

Funding: 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.

†Intel Corporation, Santa Clara, CA. (jordi.wolfson-pou@intel.com).

‡School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA (echow@cc.gatech.edu).

calling it *asynchronous iteration*, which is the usual term that has been used since. The literature on chaotic relaxation and asynchronous iterative methods is very large; additional information can be found in the historical treatments [46, 13], the reference articles [20, 47], and the monographs [10, 4], among others.

The existing literature has mainly been focused on asymptotic convergence analysis, i.e., whether or not an asynchronous method will converge, but recent interest in asynchronous iterative methods stems from their potential practical use in very large-scale parallel computations, where synchronizing a very large number of processors can lead to high processor idling times, especially on workloads that are difficult to load balance and/or hardware that has heterogeneous characteristics. Thus, the existing literature says very little about the probability that an asynchronous method will converge or how fast convergence might be, as these questions depend on characteristics of the asynchrony arising from many factors, including computer hardware. These are challenging open questions; this paper can only hint at the challenges presented by these questions.

The existing literature has also been focused on Jacobi-like methods, which unfortunately converge slowly. See, however, asynchronous versions of additive and multiplicative domain decomposition methods [40, 23, 24, 25, 21, 14], including asynchronous optimized Schwarz methods [38, 58]. Nevertheless, a complaint exists that asynchronous iterative methods are versions of stationary iterative methods, rather than, for instance, Krylov subspace methods. Recently, however, it is proposed to asynchronously construct approximate conjugate search directions for the conjugate gradient method [17]. To avoid the inherent processor synchronization of parallel inner products, each processor runs its own version of the s-conjugate direction method with full-length vectors (and perform their own full-length inner products), and portions of search directions and other vectors are exchanged asynchronously [17]. Instead of asynchronous iterative methods, a different approach is pipelined techniques that can overlap the synchronization cost of inner products in Krylov subspace methods with other computations, e.g., [22, 33], and techniques that reduce the number of synchronizations [49].

It is in this context that this paper presents an asynchronous version of the Chebyshev iterative method. With a proper choice of parameters, the Chebyshev iterative method converges rapidly. Like the conjugate gradient (CG) method, the Chebyshev iterative method is based on finding a polynomial that is small in some sense (over an interval for Chebyshev and over a positive spectrum for CG). But unlike CG, the the Chebyshev method does not require inner products in its implementation, which makes the Chebyshev iterative method amenable to asynchronous implementation.

To gain insight into the asynchronous Chebyshev method, we use the connection that the Chebyshev method in the limit of a large number of iterations is equivalent to the second-order Richardson method with an optimal choice of parameter values. Earlier work on the asynchronous second-order Richardson method [16] can thus provide insight into the asynchronous Chebyshev method. In particular, the asynchronous second-order Richardson method seems more robust for well-conditioned systems. In this paper, we will similarly find that the new asynchronous Chebyshev iterative method is more robust for well-conditioned systems. This motivates us to use the Chebyshev iterative method with preconditioning, or equivalently, a matrix splitting. For this, we introduce a new implementation of an asynchronous additive multigrid preconditioner. The entire solver combining the Chebyshev method and multigrid preconditioner is asynchronous. Our implementation uses multiple threads working asynchronously on a shared memory computer.

Related to our work is the recent development of other asynchronous multigrid methods. In the segmental refinement method [1, 12], inter-process communication is not required when smoothing on and interpolating between certain grids, allowing for asynchronous processing of some grids. Some synchronization within an iteration is removed, but certain synchronizations still exist in certain steps of each iteration. In [30], the authors asynchronously execute a saw-tooth cycle (standard multiplicative multigrid with no pre-smoothing) by using unsmoothed aggregation. Unsmoothed aggregation allows the prolongation and restriction to be performed without communication, making the smoothing and prolongation in the up-cycle completely asynchronous. However, synchronization is still used on the finest grid and the method is limited to only using interpolation matrices that require no communication. In [55], the authors developed asynchronous additive multigrid methods based on the asynchronous fast adaptive composite grid method with smoothing (AFACx) [29, 34, 35, 42, 41] and additive variants of the classical multiplicative multigrid method [53, 51]. Here, different sets of processors are assigned to different grids in the multigrid hierarchy. Updates from each grid are then computed and added together on the finest grid asynchronously, i.e., at some time instant, unknowns on some subset of the grids are updated. Synchronization is required among processors assigned to a grid, making the method semi-asynchronous. Asynchronous multigrid smoothers have been studied by Anzt and co-authors [3].

The asynchronous additive multigrid preconditioner developed in this paper is based on Griebel's interpretation of multigrid as relaxation on an extended, semidefinite system of equations [27]. We will perform this relaxation asynchronously. Solving such an extended system using a randomized method has also been considered [28].

Section 2 presents background on asynchronous iterative methods and gives some new observations, such as the effect of the choice of parameter in the asynchronous first-order Richardson method. As a precursor to an asynchronous Chebyshev method, Section 3 reviews the asynchronous second-order Richardson method, and shows the importance of preconditioning and adjusting the method's parameters to achieve robust convergence. Section 4 then presents an asynchronous Chebyshev method, which can be analyzed as a nonstationary asynchronous iteration. An asynchronous multigrid preconditioner for the asynchronous Chebyshev method is presented in Section 5. The new methods are simulated and tested in Sections 6 and 7.

2. Observations on asynchronous first-order linear iterations.

2.1. Background. A fixed-point iteration for solving the nonsingular system of linear equations $Ax = b$ with splitting $A = M - N$ has the form

$$x^{(t+1)} = Tx^{(t)} + f, \quad t = 0, 1, \ldots \tag{2.1}$$

given an initial approximation $x^{(0)}$, where $T = I - M^{-1}A$ is the iteration matrix and $f = M^{-1}b$. In practice, the choice of M is limited when we wish to execute this iteration asynchronously in parallel. For example, if each processor is assigned exactly one of the components of x, then processor i executes

$$x_i \leftarrow \sum_{j \in S(T_i)} T_{ij} x_j + f_i \tag{2.2}$$

where $S(T_i)$ is the set of column indices corresponding to the non-zero values in row i of T. Thus, for practicality in this case, M should be chosen to be diagonal. If a processor is instead assigned a block of components of x to update, then M could be

chosen to be block diagonal, for instance. If components of x could be updated by more than one processor, then M could be chosen to be a preconditioner of domain decomposition type with overlap [19].

In a synchronous fixed-point iteration, all updates in an iteration are followed by a synchronization. The synchronization prevents a processor i from starting the next iteration before other processors have completed their updates. In the asynchronous case, there is no synchronization, and processor i simply moves on to its next iteration using components of x available to it. This means that processor i can use the same value of x_j for more than one iteration, and that values of x_j from far in the past can be used for the current iteration. Algorithm 2.1 shows this asynchronous iterative method in detail for a shared memory computer.

Algorithm 2.1 Asynchronous first-order linear iterative method (for processor p) for solving $Ax = b$ with preconditioner M.

Input: initial guess vector x in shared memory; matrix $T = I - M^{-1}A$, vector $f = M^{-1}b$.
Output: approximate solution x in shared memory.
while not converged **do**
  for row i assigned to processor p **do**
    $z \leftarrow f_i$
    for $j \in S(T_i)$ **do**
      Read from shared memory: $v \leftarrow x_j$
      $z \leftarrow z + T_{ij}v$
    end for
    Write to shared memory: $x_i \leftarrow z$
  end for
end while

A very general mathematical model of an asynchronous iterative method can be written as [15, 8, 10, 4]

$$x_i^{(t+1)} = \begin{cases} \sum_{j \in S(T_i)} T_{ij} x_j^{(z_{ij}(t))} + f_i, & \text{if } i \in \Psi(t) \\ x_i^{(t)} & \text{otherwise.} \end{cases} \tag{2.3}$$

Here, we refer to t as a *time instant*, since the concept of an iteration only holds from the point of view of one processor rather than all processors. The mapping $z_{ij}(t)$ maps the current time instant to some previous time instant, and $\Psi(t)$ is the set of rows that are being updated at time instant t. This model is very general, which allows it to account for different communication delays and also dynamic assignment of processors to the updates.

To prove convergence of the general mathematical model, there are three reasonable conditions on $z_{ij}(t)$ and $\Psi(t)$:

1. $z_{ij}(t) \leq t$
2. As $t \to +\infty$, $z_{ij}(t) \to +\infty$
3. As $t \to +\infty$, the number of times i appears in $\Psi(t) \to +\infty$

Given these conditions, the asynchronous method (2.3) will converge for any initial approximation $x^{(0)}$ if $\rho(|T|) < 1$, where $\rho(|T|)$ is the spectral radius of the element-wise absolute value of T. If $\rho(|T|) > 1$ then there exists the sequences $z_{ij}(t)$ and sets $\Psi(t)$ such that the asynchronous method does not converge [15, 8].

2.2. Numerical examples. To illustrate the behavior of asynchronous iterative methods, we consider a baseline linear system that comes from the 5-point discretization of the Poisson equation on a square domain. The right-hand side vector was chosen randomly from a uniform distribution with mean zero. For each experiment, where many runs are performed, the same random right-hand side is used for each separate run. We ran the synchronous and asynchronous Jacobi methods using 20 threads on a dual processor Intel Xeon computer with a total of 20 cores. The threads were pinned to the cores using the "compact" thread affinity setting, which is the best setting for the synchronous method. Each thread was assigned an equal number of contiguous rows of the matrix equation, using the "natural" ordering of the sparse matrix.

Each dot in Figure 1a shows the result of one run of asynchronous or synchronous Jacobi (for the same matrix and right-hand side). Each run was terminated when the total number of updates reached the equivalent of m local iterations (with $m = 10, 11, \ldots, 1500$) by each thread, and then the time and the residual norm were measured. We did not use a procedure for terminating the iterations based on the residual norm as we wished to exclude and not conflate the effect of termination detection in our measurements. Each thread was assigned a fixed set of unknowns to update.

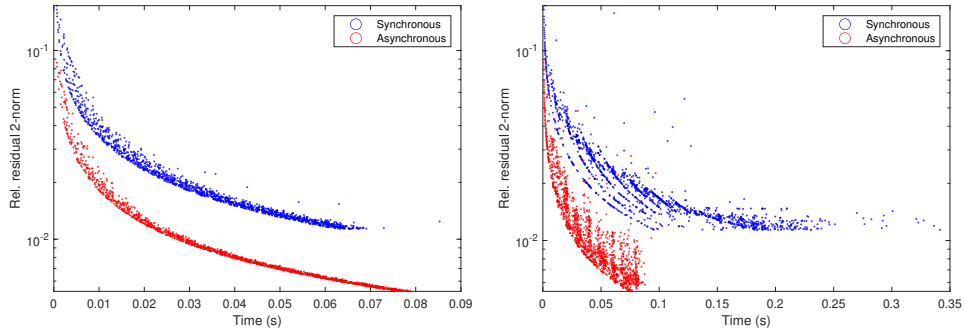
The results show that asynchronous iterations generally require less time to reach the same residual norm. However, the reason here is not because each asynchronous iteration requires on average less time than a synchronous iteration. In fact, for compact thread affinity, false sharing of cache lines seems to make asynchronous iterations slower than synchronous iterations (for the same number of updates). We do observe that individual asynchronous iterations are faster on average than synchronous iterations for 20 threads using "scatter" thread affinity.

The reason that asynchronous iterations require less time to reach the same residual norm is because the *convergence* of asynchronous Jacobi is improved: it appears that the updates at one time instant are soon utilized such that the iteration has a multiplicative nature (like Gauss-Seidel) rather than a purely additive nature (like Jacobi). Although the computations are performed in parallel, the updates to each x_i in shared memory appear mostly sequentially. The same multiplicative effect explains why it is possible for an asynchronous iteration to converge when the synchronous iteration diverges [54]. This perhaps surprising advantage of the asynchronous Jacobi method, however, will not carry over to all asynchronous methods, for example the second-order methods to be discussed in the next section.

The advantage of asynchronous Jacobi iterations over its synchronous counterpart is more pronounced if there are irregularities in the effective load on each processor core. For example, Figure 1b shows the same experiment but this time with another copy of asynchronous Jacobi running in the "background" on the same 20 cores. The result now shows that synchronous Jacobi often requires much longer to achieve the same residual norm.

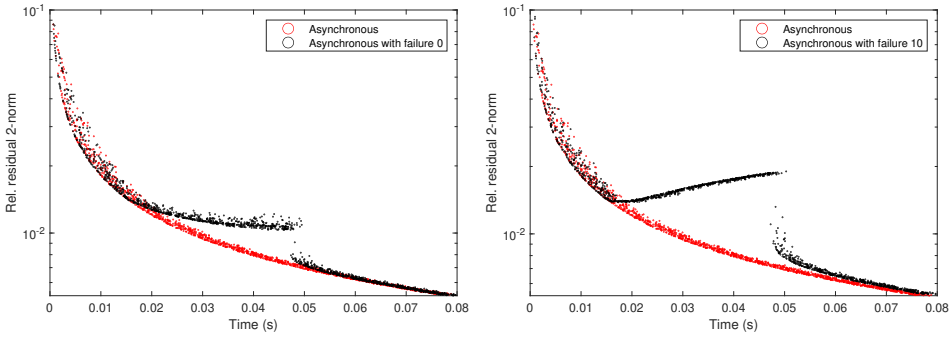
The above experiment illustrates the effect when applications compete for the same resources, a common situation in cloud computing environments [57]. Jobs submitted to the cloud run on *virtual CPUs* (VCPUs), where multiple VCPUs can be assigned to a single physical CPU if the user demands are high enough. Consequently, multiple applications run concurrently on a physical CPU and share one or more physical cores, which is the scenario emulated here.

Another advantage of asynchronous iterations can be observed when there is a temporary failure of one of the threads. Figure 2 shows the behavior when thread



(a) Without background processes. (b) With background processes.

FIG. 1. *Synchronous and asynchronous Jacobi using 20 threads for a linear system with 90,000 equations.*



(a) Temporary failure of thread 0. (b) Temporary failure of thread 10.

FIG. 2. *Asynchronous Jacobi using 20 threads, showing the effect of a temporary failure of one thread, for a linear system with 90,000 equations.*

0 or thread 10 temporarily stops updating the variables assigned to it. Although convergence slows down or even worsens, when computation on the failed thread resumes, convergence can rapidly improve as if the failure did not occur. This is because, during the failure, the equations are being made more consistent with each other, except for the unknowns associated with the failed thread. When computations resume on the failed thread, a small number of updates can rapidly make all the equations more consistent with each other.

2.3. First-order iterations with a parameter. We now consider using an acceleration parameter α in the iterative method. To solve $Ax = b$, the first-order Richardson method is

$$x^{(t+1)} = (I - \alpha A)x^{(t)} + \alpha b$$

corresponding to having a splitting matrix $M = \frac{1}{\alpha}I$. If the spectrum of A is real and lies in $[\lambda_a, \lambda_b]$, with $\lambda_a > 0$, then for synchronous iterations the optimal value of α is

$$\alpha_{\text{opt}} = 2/(\lambda_a + \lambda_b).$$

For the first-order Richardson method, is the optimal value of α for synchronous iterations also good for asynchronous iterations? As shown from the results above,

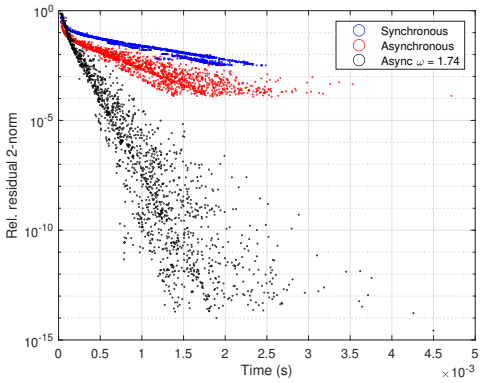


FIG. 3. *Synchronous and asynchronous first-order Richardson using 20 threads. The synchronous method uses the optimal parameter value $\alpha_{opt} = 1$. The asynchronous method is tested with both the Richardson optimal parameter value $\alpha_{opt} = 1$, and the SOR optimal parameter value $\omega_{opt} = 1.74$.*

asynchronous iterations operate in parallel but the updates of the unknowns can be close to being sequential, meaning the method is effectively multiplicative rather than additive. If updates to the x_i are performed exactly multiplicatively, i.e., one after another, we have what could be called the "multiplicative first-order Richardson" method,

$$x^{(t+1)} = (I + \alpha\tilde{L}_A)^{-1}\left[(I - \alpha U_A)x^{(t)} + \alpha b\right]$$

with splitting matrix

$$M = \frac{1}{\alpha}I + \tilde{L}_A$$

where $\tilde{L}_A$ is the strictly lower triangular part of A and U_A is upper triangular part of A. This is similar to the splitting matrix M in successive overrelaxation (SOR):

$$M = \frac{1}{\omega}D_A + \tilde{L}_A$$

where D_A is the diagonal of A and ω is the SOR parameter. This motivates us to use the SOR optimal parameter value instead of the Richardson optimal parameter value, assuming that the linear system has been scaled so that A has diagonal entries of all ones.

Figure 3 compares the convergence of the asynchronous first-order Richardson method for two different values of α. The matrix in the linear system used here is from the 5-point discretization of the Poisson equation on a 20×20 mesh, and is scaled so that its diagonal is all ones. For this matrix, the optimal Richardson parameter value is $\alpha_{\text{opt}} = 1$. On the other hand, the optimal SOR parameter value for this matrix is approximately $\omega_{\text{opt}} = 1.74$. The figure shows that, because the asynchronous iteration has a multiplicative nature, using the optimal SOR parameter value as α in the asynchronous Richardson method dramatically accelerates the convergence rate. The result for synchronous Richardson with $\alpha_{\text{opt}} = 1$ is also shown for comparison.

The main conclusion here is that the optimal parameter values used for asynchronous iterations may not be the same as those known for synchronous iterations.

3. Observations on asynchronous second-order linear iterations.

3.1. Background. To solve $Ax = b$, starting with $x^{(0)}$, the preconditioned first-order Richardson method with preconditioner $M \approx A$ is

$$x^{(t+1)} = (1-\alpha)x^{(t)} + \alpha(x^{(t)} + M^{-1}(b - Ax^{(t)})) \tag{3.1}$$

$$= x^{(t)} + \alpha M^{-1}(b - Ax^{(t)}) \tag{3.2}$$

for $t = 0, 1, \ldots$, where the first equation above shows that the new iterate is a linear combination with parameter α of the basic iterate $x^{(t)} + M^{-1}(b - Ax^{(t)})$ and the current iterate $x^{(t)}$.

The preconditioned second-order Richardson method repeats this process by combining the first-order Richardson iterate with the earlier iterate $x^{(t-1)}$,

$$x^{(t+1)} = (1+\beta)\left[x^{(t)} + \alpha M^{-1}(b - Ax^{(t)})\right] - \beta x^{(t-1)}$$

$$= x^{(t-1)} + (1+\beta)\left[\alpha M^{-1}(b - Ax^{(t)}) + x^{(t)} - x^{(t-1)}\right] \tag{3.3}$$

$$= (1+\beta)(I - \alpha M^{-1}A)x^{(t)} - \beta x^{(t-1)} + (1+\beta)\alpha M^{-1}b \tag{3.4}$$

for $t = 1, 2, \ldots$, with linear combination parameter β. The initial approximation is $x^{(0)}$, and $x^{(1)} = x^{(0)} + \alpha M^{-1}(b - Ax^{(0)})$, the first-order Richardson iterate. The form (3.4) is useful for the 2×2 block form that follows, while the form (3.3) is useful for comparison with a common way that the Chebyshev iterative method is written.

The first- and second-order Richardson methods are examples of *semi-iterative* methods since part of the new iterate is from a basic iteration, and the remaining part is from recombining previous iterates.

If the spectrum of $M^{-1}A$ is real and lies in $[\lambda_a, \lambda_b]$, with $\lambda_a > 0$, then the optimal second-order Richardson parameter values are:

$$\alpha_{\text{opt}} = \frac{2}{\lambda_a + \lambda_b}$$

$$\beta_{\text{opt}} = \left(\frac{\sqrt{\lambda_b} - \sqrt{\lambda_a}}{\sqrt{\lambda_b} + \sqrt{\lambda_a}}\right)^2$$

which can be found by optimizing the spectral radius of the block 2×2 iteration matrix, $T_{\alpha,\beta}$ in

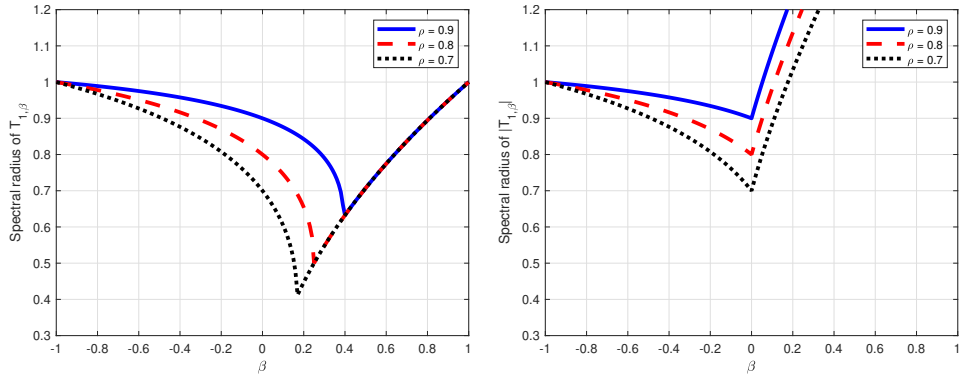
$$\begin{bmatrix} x^{(t+1)} \\ x^{(t)} \end{bmatrix} = \underbrace{\begin{bmatrix} (1+\beta)(I - \alpha M^{-1}A) & -\beta I \\ I & 0 \end{bmatrix}}_{T_{\alpha,\beta}} \begin{bmatrix} x^{(t)} \\ x^{(t-1)} \end{bmatrix} + \begin{bmatrix} (1+\beta)\alpha M^{-1}b \\ 0 \end{bmatrix} \tag{3.5}$$

which corresponds to the iteration (3.4).

In [16], the asynchronous second-order Richardson method was analyzed by considering the spectral radius of $|T_{\alpha,\beta}|$. The eigenvalues λ of $|T_{\alpha,\beta}|$ satisfy

$$\lambda^2 - |1+\beta|\mu\lambda - |\beta| = 0$$

where μ is an eigenvalue of $|I - \alpha M^{-1}A|$. From this result, the spectral radius of $|T_{\alpha,\beta}|$ can be easily computed. As the spectral radius depends on the eigenvalues of $M^{-1}A$, let ρ, with $0 \leq \rho < 1$, denote the value such that the eigenvalues of $M^{-1}A$ lie in $[1-\rho, 1+\rho]$. With this information, the optimal value of α is 1, so it only remains to investigate the spectral radius as a function of β.



(a) Spectral radius of $T_{1,\beta}$ as a function of β. (b) Spectral radius of $|T_{1,\beta}|$ as a function of β.

FIG. 4. *Spectral radius of $T_{1,\beta}$ and $|T_{1,\beta}|$ corresponding to the synchronous and asynchronous cases, respectively. The matrix $M^{-1}A$ has a real spectrum in $[1-\rho, 1+\rho]$.*

Figure 4 plots the spectral radius of $T_{\alpha,\beta}$ for the synchronous method and of $|T_{\alpha,\beta}|$ for the asynchronous method as a function of β, assuming $\alpha = 1$, for three cases of $M^{-1}A$, corresponding to $\rho = 0.7$, 0.8, and 0.9. The matrix $M^{-1}A$ is more ill-conditioned for values of ρ closer to 1.

In the synchronous case, as ρ increases, the optimal value of β increases. The second-order Richardson iteration converges for all $-1 < \beta < 1$. In the asynchronous case, for a given value of ρ, the spectral radius increases rapidly as β increases from 0. The spectral radii for the synchronous and asynchronous cases are the same for $\beta \leq 0$. For $\rho = 0.7$, the optimal β for the synchronous iteration is near 0.18. For this value of β the spectral radius of $|T(\alpha, \beta)|$ is very near 1. For problems with larger ρ, the corresponding optimal value of β would give an asynchronous iteration that is not guaranteed to converge.

What can be concluded from this result is that whether or not an asynchronous second-order Richardson iteration is guaranteed to converge will depend on ρ, or equivalently, the condition number of the matrix. We will apply this result to understanding the asynchronous Chebyshev iterative method in the next section.

Note that the size of the spectral radius of $|T_{\alpha,\beta}|$ is not directly related to the convergence of the asynchronous method; the asynchronous method can converge as rapidly as the synchronous method if its execution is mostly synchronous.

3.2. Implementation and numerical examples. Algorithm 3.1 shows an implementation of the asynchronous second-order Richardson method. Each thread is assigned a fixed set of unknowns to update. An implementation choice that must be decided is when to make the updated values available to other threads. A natural choice is for an updated value to be written to shared memory immediately after it is computed. A second choice is for a thread to save the updated values and only write them to shared memory after all the updates assigned to that thread have been performed in its local iteration. Algorithm 3.1 implements this second choice which gave somewhat better results. It appears that for the second-order algorithm, compared to the first-order algorithm, it is important to have some synchronization (local to a thread) to improve convergence.

To analyze these two choices, one can consider a multiplicative version of the iteration (3.5), just like we considered a multiplicative version of the first-order Richardson

Algorithm 3.1 Asynchronous second-order Richardson method (for processor p) for solving $Ax = b$ with preconditioner M.

Input: parameters α, β; matrix $B = M^{-1}A$; vector $f = \alpha M^{-1}b$; initial approximation u^0 in a temporary vector; current approximation $x = u^0 + (f - \alpha B u^0)$ in shared memory.
Output: approximate solution x in shared memory.
$u_i \leftarrow x_i$ (read local values of x_i)
while not converged **do**
  for row i assigned to processor p **do**
    $z \leftarrow 0$
    for $j \in S(B_i)$ **do**
      $z \leftarrow z + B_{ij}x_j$ (read x_j from shared memory)
    end for
    $u_i \leftarrow u_i^0 + (1+\beta)(f_i - \alpha z + u_i - u_i^0)$
  end for
  for row i assigned to processor p **do**
    $u_i^0 \leftarrow x_i$ (read local values of x_i)
    $x_i \leftarrow u_i$ (write to shared memory)
  end for
end while

method. This multiplicative version has the iteration matrix

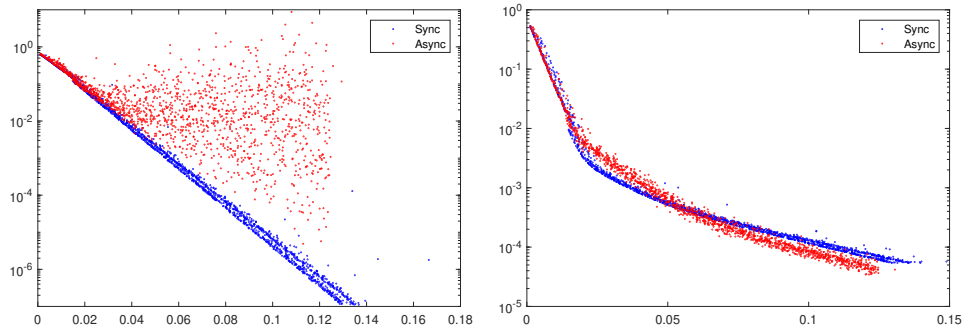
$$\begin{bmatrix} (I + (1+\beta)\alpha\tilde{L}_A)^{-1}(1+\beta)(I - \alpha U_A) & -\beta(I + (1+\beta)\alpha\tilde{L}_A)^{-1} \\ I & 0 \end{bmatrix}.$$

Whereas in the first-order method, the spectral radius of the multiplicative iteration matrix can be made smaller (with a proper choice of parameter value) than the additive one, this is not the case in the second-order Richardson method, i.e., running the second-order iteration in multiplicative fashion would not be expected to improve convergence.

Figure 5 compares the convergence of synchronous and asynchronous second-order Richardson methods using 10 threads. The same baseline linear system is used as in the previous section, with the matrix scaled so that its diagonal is all ones. In Figure 5a, the methods use the optimal (synchronous) parameter values, $\alpha = 1$ and $\beta = 0.979$. According to Figure 4b, the asynchronous method is not guaranteed to converge for these parameter values ($\rho \approx 0.99995$ for this matrix). The results in Figure 5a show that, although we do observe progress of the asynchronous method toward the solution in most of the cases, convergence is very poor compared to that of the synchronous method.

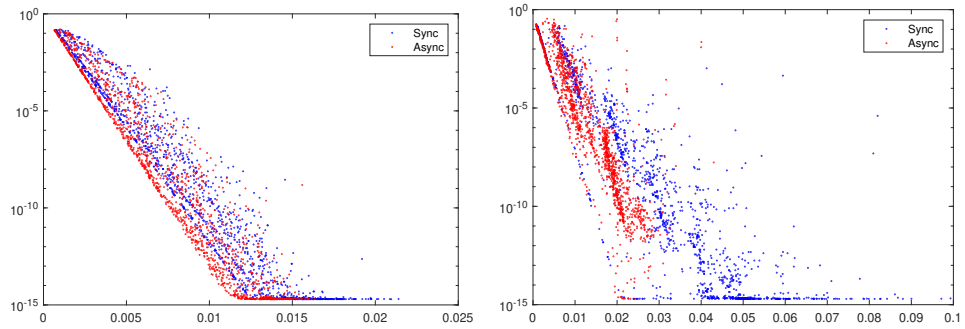
In Figure 5b, both methods used $\alpha = 1$ and $\beta = 0.95$, i.e., a slightly smaller value of β than optimal. To guarantee convergence of the asynchronous method, a value of β very nearly zero would be needed for this matrix. However, we observe in Figure 5b that just a small decrease in β from optimal in this case (which corresponds to underestimating the spectrum of $M^{-1}A$), results in very consistent asynchronous convergence, comparable to the convergence of the synchronous method. As we also saw for first-order Richardson, a good choice of parameter value for an asynchronous method may be different from that of the corresponding synchronous method.

As observed in Figure 4b, systems with larger ρ, which are more ill-conditioned, have a smaller range of β for which convergence is guaranteed. Thus, the asyn-



(a) Optimal $\beta = 0.979$ is used. (b) Non-optimal $\beta = 0.95$ is used.

FIG. 5. *Synchronous (blue) and asynchronous (red) second-order Richardson using 10 threads for a linear system with 90,000 equations. The y-axis is the relative residual 2-norm, and the x-axis is time.*



(a) Transformed problem with $\rho = 0.9899$ using the optimal $\beta = 0.752$. (b) Same as (a) but with background processes.

FIG. 6. *Synchronous (blue) and asynchronous (red) second-order Richardson using 10 threads for a* transformed *linear system with 90,000 equations. The y-axis is the relative residual 2-norm, and the x-axis is time.*

chronous second-order Richardson method may have more of an advantage over its synchronous counterpart for matrices that are better conditioned, or systems that are preconditioned. We test this experimentally by transforming the system used above such that the spectrum of $M^{-1}A$ still lies in $[1-\rho, 1+\rho]$, but ρ has now been reduced to 0.9899. For this value of ρ, the optimal β is 0.752. Figure 6a shows the convergence of synchronous and asynchronous second-order Richardson in this case. We observe that convergence is very rapid for both methods and that the asynchronous method is often faster than the synchronous method, primarily due to the fact that each asynchronous update is faster than a synchronous update. Figure 6b shows the results when background processes are also running on the 10 threads. Both methods are slowed down by the background processes, but the synchronous method is slowed down more significantly.

4. An asynchronous Chebyshev iterative method. To solve the preconditioned system $M^{-1}Ax = M^{-1}b$, where we assume the spectrum of $M^{-1}A$ is real and

lies in $[\lambda_a, \lambda_b]$, with $\lambda_a > 0$, the (synchronous) Chebyshev iterative method is

$$\begin{aligned} \omega^{(t+1)} &= 1/(1 - \omega^{(t)}/4\mu^2) \\ x^{(t+1)} &= x^{(t-1)} + \omega^{(t+1)}(\alpha M^{-1}(b - Ax^{(t)}) + x^{(t)} - x^{(t-1)}) \end{aligned} \tag{4.1}$$

for $t = 1, 2, \ldots$, with initial approximation $x^{(0)}$, first approximation $x^{(1)} = x^{(0)} + \alpha M^{-1}(b - Ax^{(0)})$, and $\omega^{(1)} = 2$, $\alpha = 2/(\lambda_b + \lambda_a)$, $\mu = (\lambda_b + \lambda_a)/(\lambda_b - \lambda_a)$. We have used a version of the recurrence for $\omega^{(t+1)}$ that only depends on one previous value [50]. While not considered in this paper, computing λ_a and λ_b can be combined with a small number of initial iterations of preconditioned CG for solving the preconditioned system before switching to the Chebyshev method.

Like the first- and second-order Richardson methods, the Chebyshev iterative method is semi-iterative. The iterate at the t-th step is a combination of a basic iteration and the iterates at all previous steps. The form of the iteration is the same as (3.3) for the second-order Richardson method; with the Chebyshev method, $(1+\beta)$ in the Richardson method is replaced by the dynamic parameter $\omega^{(t+1)}$.

Equation (4.1) can be written as a single step method via the block form

$$\begin{bmatrix} x^{(t+1)} \\ y^{(t+1)} \end{bmatrix} = \begin{bmatrix} \omega^{(t+1)}(I - \alpha M^{-1}A) & (1 - \omega^{(t+1)}) \\ I & 0 \end{bmatrix} \begin{bmatrix} x^{(t)} \\ y^{(t)} \end{bmatrix} + \begin{bmatrix} \omega^{(t+1)}\alpha M^{-1}b \\ 0 \end{bmatrix}. \tag{4.2}$$

If we assume for simplicity that each row i of A is assigned to one processor, then the mathematical model of the asynchronous Chebyshev method based on the block iteration is

$$\begin{aligned} &\text{If } i \in \Psi(t)\text{:} \\ &\quad \omega_i^{(t+1)} = 1/(1 - \omega_i^{(t)}/4\mu^2) \\ &\quad x_i^{(t+1)} = y_i^{(t)} + \omega_i^{(t+1)}\Big(f_i - \alpha \sum_{j \in S(B_i)} B_{ij} x_j^{(z_{ij}(t))} + x_i^{(t)} - y_i^{(t)}\Big) \\ &\quad y_i^{(t+1)} = x_i^{(t)} \\ &\text{Else:} \\ &\quad x_i^{(t+1)} = x_i^{(t)} \\ &\quad y_i^{(t+1)} = y_i^{(t)} \\ &\quad \omega_i^{(t+1)} = \omega_i^{(t)} \end{aligned} \tag{4.3}$$

where $B = M^{-1}A$ and $f = \alpha M^{-1}b$. As in (2.3), we have the mappings $z_{ij}(t)$ and the sets $\Psi(t)$. The iteration is initialized with initial approximation $x^{(0)}$ and first approximation $y^{(0)}$. We now have three variables that are propagated asynchronously: x, y and ω. In the synchronous Chebyshev method, all processors share the same value of the scalar ω, whereas in the asynchronous method, ω is local to each processor, and thus we have used the subscript i on ω_i. The initial value of $\omega_i = 2$ for all processors.

To analyze the asynchronous Chebyshev method, we consider its iteration matrix in (4.2). Unlike in the Richardson case, the iteration matrix here is not fixed. Theory exists for analyzing nonstationary asynchronous iterations [20, 16] which can be applied to the asynchronous Chebyshev method. First note that (4.2) has the same fixed point for any nonzero value of $\omega^{(t+1)}$. Let $T(\omega^{(t+1)})$ denote the iteration matrix

in (4.2). Then, if the spectral radius of $|T(\omega^{(t+1)})| < 1$ for all time instants t, then the asynchronous iteration converges for any initial approximation. In practice, the asynchronous Chebyshev method may converge even if this condition is not satisfied.

Algorithm 4.1 shows a computational model for implementing the asynchronous Chebyshev method on a shared memory computer. We have kept this version of the algorithm simple: updates of the unknowns are written immediately to shared memory.

Algorithm 4.1 Asynchronous Chebyshev iterative method (for processor p) for solving $Ax = b$ with preconditioner M.

Input: parameters α, μ; matrix $B = M^{-1}A$; vector $f = \alpha M^{-1}b$; initial approximation u^{prev} in a temporary vector; current approximation $x = u^{\text{prev}} + (f - \alpha B u^{\text{prev}})$ in shared memory.
Output: approximate solution x in shared memory.
$u_i \leftarrow x_i$ (read local values of x_i)
$\omega \leftarrow 2$
while not converged on processor p **do**
  Local update: $\omega \leftarrow 1/(1 - \omega/4\mu^2)$
  for row i assigned to processor p **do**
    $z \leftarrow 0$
    for $j \in S(B_i)$ **do**
      $z \leftarrow z + B_{ij}x_j$ (read x_j from shared memory)
    end for
    $u_i^{\text{temp}} \leftarrow u_i$
    $u_i \leftarrow u_i^{\text{prev}} + \omega\,(f_i - \alpha z + u_i - u_i^{\text{prev}})$
    Write to shared memory: $x_i \leftarrow u_i$
    $u_i^{\text{prev}} \leftarrow u_i^{\text{temp}}$
  end for
end while

Adjusting the Chebyshev weight. In Figure 5 in the previous section, we observed that decreasing β from the optimal one in the asynchronous second-order Richardson method could lead to more consistent convergence behavior. In the same spirit, we will adjust the Chebyshev weight ω in the asynchronous Chebyshev method. Indeed, $\beta = 0$ corresponds to a first-order method, which we saw to be less sensitive to asynchronous computations than second-order methods; compare Figures 1a and 5a, showing that convergence of the asynchronous second-order method is much more variable than that of the asynchronous first-order method. Noting that substituting $\omega^{(t+1)}$ for $1 + \beta$ in the second-order Richardson method (3.3) gives the Chebyshev method (4.1), we adjust ω at each iteration by *decreasing* it by a fixed value. However, the adjusted value of ω is not used in the recurrence to generate the next value. Using a smaller value of ω can also be interpreted as underestimating the spectrum of the preconditioned matrix. This is consistent with the result of Golub and Overton [26] that underestimating the spectrum is advantageous in the Chebyshev method in the symmetric case (as opposed to the skew-symmetric case) when the iteration is performed *inexactly*.

5. Asynchronous multigrid based on solving an extended system.

5.1. Griebel's extended system. Section 3 motivated the need to combine preconditioning with the asynchronous Chebyshev method. In this section, we present an asynchronous multigrid preconditioner based on solving an extended semidefinite system of equations [27]; see also [52] for an alternative presentation. Our main contribution in the second part of this paper is a parallel implementation of this method that is matrix-free (the extended system is not constructed explicitly), which is necessary for computational efficiency.

To implement an ℓ-level multigrid method for solving $Ax = b$, Griebel [27] proposed solving the "extended" block system of equations with ℓ block rows,

$$\begin{bmatrix} A_0 & A_0P_1^0 & \cdots & A_0P_{\ell-1}^0 \\ R_0^1A_0 & A_1 & \cdots & A_1P_{\ell-1}^1 \\ \vdots & \vdots & \ddots & \vdots \\ R_0^{\ell-1}A_0 & R_1^{\ell-1}A_1 & \cdots & A_{\ell-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{\ell-1} \end{bmatrix} = \begin{bmatrix} b \\ R_0^1b \\ \vdots \\ R_0^{\ell-1}b \end{bmatrix} \tag{5.1}$$

which we denote as $A^Ex^E = b^E$. The matrix P_1^0 interpolates a vector from grid 1 (a coarse grid) to grid 0 (original fine grid) and the restriction matrix R_0^1 transfers a vector from grid 0 to grid 1. The original fine grid matrix is $A = A_0$. The coarse grid matrix A_1 is computed as $R_0^1A_0P_1^0$ (known as the Galerkin coarse grid) in this paper. This notation is extended in the obvious way, e.g., A_i is the matrix for the ith grid.

Griebel [27] showed that Gauss-Seidel iteration on this extended system $A^Ex^E = b^E$ is equivalent to the standard multiplicative multigrid on the original system, $Ax = b$. More precisely, one iteration of

$$x^E \leftarrow x^E + (L^E)^{-1}(b^E - A^Ex^E),$$

where L^E is the lower triangular part of A_E, is the same as one V(1,0) cycle of multiplicative multigrid applied to $Ax = b$, where the presmoother is Gauss-Seidel and the coarsest grid solve is a solve with the lower triangular part of the coarse grid operator, $A_{\ell-1}$. If the $(\ell-1, \ell-1)$ block of L^E is replaced with the full $A_{\ell-1}$, then the coarsest grid solve is an "exact" solve with the coarsest grid operator. The approximate solution of the original system at iteration t is recovered as $x_0^{(t)} + P_1^0x_1^{(t)} + \cdots + P_{\ell-1}^0x_{\ell-1}^{(t)}$.

Similarly, one Jacobi-like iteration,

$$x^E \leftarrow x^E + (D^E)^{-1}(b^E - A^Ex^E), \tag{5.2}$$

where D_E is an appropriately chosen diagonal matrix, is equivalent to one step of the additive BPX preconditioner of Bramble, Pasciak, and Xu [11] applied to the original system, assuming a diagonal solve is used for the coarsest grid problem. Further, accelerating this iteration with the conjugate gradient method, we have that the diagonally preconditioned CG method for solving the extended system, with appropriately-chosen D_E as the preconditioner, is equivalent to BPX-preconditioned CG for solving the original system.

In this paper, we propose implementing an asynchronous BPX-preconditioned Chebyshev method by analogously applying a diagonally preconditioned asynchronous Chebyshev method to solve the extended system. We call this method the asynchronous EBPX-Chebyshev method, where E signifies that an extended system is solved. Matrix-vector multiplies with the extended matrix in the asynchronous Chebyshev method are carried out asynchronously. However, explicitly constructing

the extended system is expensive. In the next section, we describe how to asynchronously apply matrix-vector products with the extended matrix in an economical, matrix-free fashion.

5.2. Matrix-free implementation. We propose a version of EBPX where the extended system matrix A^E is multiplied with a vector without explicitly constructing and storing A^E. Using an explicitly constructed A^E has two major drawbacks:

1. Constructing the off-diagonal blocks of A^E requires many matrix-matrix multiplication operations, leading to high multigrid preconditioner setup costs.
2. There is redundant computation involved when multiplying a block-row of A^E with a vector.

To see the redundant computation that occurs when multiplying A^E by a vector $x = (x_0^T, x_1^T, \ldots, x_{\ell-1}^T)^T$ to obtain vector $y = (y_0^T, y_1^T, \ldots, y_{\ell-1}^T)^T$, consider an upper-triangular off-diagonal block

$$A_{ij}^E = A_i P_j^i = A_i P_{i+1}^i \cdots P_j^{j-1}.$$

To compute y_0, we need to perform the computations $P_1^0 x_1$, $P_1^0 P_2^1 x_2$, $P_1^0 P_2^1 P_3^2 x_3$, etc., and calculate their sum. It would be more efficient to only perform the computation $P_1^0(x_1 + P_2^1(x_2 + P_3^2(x_3 + \ldots)))$, similar to a standard multigrid prolongation process. Redundant calculation also occurs for the restriction matrices in the block lower-triangular part of A^E. Further, there are also redundancies when multiplying by the grid matrices A_i. For example, in the above matrix-vector product, A_0 is multiplied by x_0 when calculating each of the y_i.

In our implementation, the available processors are assigned to different grids. Each grid corresponds to a block row of the extended system. The processors assigned to grid i synchronize with each other to compute the update for grid i, but do not synchronize with processors assigned to other grids. The reason synchronization is needed within a grid is that many intermediate vectors are computed when a block-row is multiplied with a vector since there can be many off-diagonal blocks if there are many grids. We could use intermediate vectors from previous time instants, but we found experimentally that this often caused asynchronous EBPX-Chebyshev to diverge, especially when there are many grids.

Let $\mathcal{P}_i$, $i = 0, \ldots, \ell - 1$ be the set of processors assigned to grid i, or equivalently, block row i. Each processor is only assigned to one block row. Note that the grids at each level have different sizes, so generally each grid is assigned a different number of processors. These assignments could be determined by balancing the computation and/or the communication costs.

Algorithm 5.1 shows the matrix-free asynchronous EBPX-Chebyshev method, implementing the Chebyshev method to accelerate one step of (5.2). The algorithm is written from the point-of-view of the processors in $\mathcal{P}_i$ assigned to block-row i. The function `ExtendSysMatFreeMV_RowBlock()` is used to perform matrix-vector products with a block row of the extended matrix A^E asynchronously and in matrix-free fashion. This is followed by a diagonal smoothing for grid i with D_i^{-1}. We use the ℓ_1-Jacobi smoother [7] here, which gave better results than regular Jacobi preconditioning.

Algorithm 5.2 shows the pseudocode for `ExtendSysMatFreeMV_RowBlock()`. This function multiplies block-row i of the extended matrix by a vector to compute sub-vector y_i. In addition to x, this function also uses an additional shared memory vector $v = (v_0^T, v_1^T, \ldots, v_{\ell-1}^T)^T$ where each v_i is computed as $v_i = A_i x_i$. The purpose of v_i is to further reduce the number of matrix-vector products with A_i since the operation $A_i x_i$ appears $\ell - i$ times in the block-columns of A^E. There are two important phases

Algorithm 5.1 Matrix-free asynchronous EBPX-Chebyshev method.

Input: Chebyshev parameters α, μ; matrix A_i; diagonal smoother matrix D, vector $f = \alpha D^{-1} b$; initial approximation u^{prev} in a temporary vector; current approximation $x = u^{\text{prev}} + (f - \alpha B u^{\text{prev}})$ in shared memory.
Output: approximate solution x in shared memory.
$u_i \leftarrow x_i$ (read local values of x_i)
$\omega \leftarrow 2$
while processors $\mathcal{P}_i$ assigned to block-row i have not converged **do**
  $y_i \leftarrow$ `ExtendSysMatFreeMV_RowBlock`$(A_i,\ i,\ x)$
  $z_i = D_i^{-1} y_i$
  $\omega \leftarrow 1/(1 - \omega/4\mu^2)$
  $u_i^{\text{temp}} \leftarrow u_i$
  $u_i \leftarrow u_i^{\text{prev}} + \omega\,(f_i - \alpha z_i + u_i - u_i^{\text{prev}})$
  Write to shared memory: $x_i \leftarrow u_i$
  $u_i^{\text{prev}} \leftarrow u_i^{\text{temp}}$
end while

Algorithm 5.2 $y_i =$ `ExtendSysMatFreeMV_RowBlock`$(A_i,\ i,\ x)$ which implements the asynchronous matrix-free matrix-vector multiplication $y = A^E x^E$ for block row i, computed by processors in $\mathcal{P}_i$. In the algorithm, v is a shared workspace vector.

Input: matrix A_i; index i; x in shared memory.
Output: y_i.

{Compute quantities to be used by other processors}
if $i < \ell - 1$ **then**
  Write to shared memory: $v_i \leftarrow A_i x_i$
end if

{Multiply x with blocks from the upper triangular part of the matrix}
Read from shared memory: $y_{\ell-1}^{\text{upper}} \leftarrow x_{\ell-1}$
for $j = \ell - 2$ **down to** i **do**
  Read from shared memory: $h \leftarrow x_j$
  $y_j^{\text{upper}} \leftarrow h + P_{j+1}^{j} y_{j+1}^{\text{upper}}$
end for
$y_i^{\text{upper}} \leftarrow A_i y_i^{\text{upper}}$

{Multiply v with blocks from the strict lower triangular part of the matrix}
$y_0^{\text{lower}} \leftarrow 0$
for $j = 0$ **up to** $i - 1$ **do**
  Read from shared memory: $h \leftarrow v_j$
  $y_{j+1}^{\text{lower}} \leftarrow R_j^{j+1}(h + y_j^{\text{lower}})$
end for

{Output is the sum of the upper and lower partial sums}
$y_i \leftarrow y_i^{\text{upper}} + y_i^{\text{lower}}$

of Algorithm 5.2: the multiplication of x with the upper-triangular part of block-row i and the multiplication of v with the strict lower-triangular part of block-row i. The multiplication of the upper-triangular part with x can be thought of as interpolating the updates from coarser grids to grid i, and the multiplication of the strict lower-triangular part with v can be thought of as restricting the updates from the finer grids to grid i. In our asynchronous Chebyshev method using shared memory, the blocks of x and v are updated asynchronously. This means that while y_i is computed synchronously by $\mathcal{P}_i$, the sub-vectors (and even individual elements) of x and v used by $\mathcal{P}_i$ from other block-rows will in general come from different time instants.

Our matrix-free algorithm reduces the computational cost when compared to explicitly constructing and using the extended matrix. For each A_i, we have reduced the number of multiplications with a vector from $\ell - i - 1$ to 2 which is significant for small i, i.e., for the matrices that correspond to the finer grids. The redundant calculations in the prolongation and restriction steps have also been significantly reduced. The cost of the prolongation and restriction steps is still higher than that in standard multigrid since each block-row must prolong and restrict vectors from all other block-rows. However, the benefits from asynchronous execution can outweigh the cost of performing additional prolongation and restriction steps.

While Algorithm 5.2 is written for shared memory systems, it can be extended to distributed memory environments by using point-to-point communication to transfer data between processors instead of modifying data in shared memory. *Intra-grid* communication is used to transfer data between processors that belong to the same $\mathcal{P}_i$ and asynchronous *inter-grid* communication is used otherwise. Intra-grid communication occurs during sparse matrix-vector product (SpMV) operations that involve A, R, and P. In the implementation of these SpMV operations, the conventional approach is used, where the processors in $\mathcal{P}_i$ are assigned a partition of A, R, and P and synchronously compute each SpMV operation for grid i.

Inter-grid communication replaces the steps in Algorithm 5.2 where data in shared memory is modified. Asynchronous point-to-point communication is used to transfer data, e.g., passive one-sided or non-blocking MPI functions. Using the inter-grid SpMV partitions, a processor $p \in \mathcal{P}_i$ asynchronously communicates its partitions of v and x with all $q \in \mathcal{P}_j$ (where $j \neq i$) that have overlapping partitions with those of p. During the steps where v is sent asynchronously, a processor sends its current value of v to all inter-grid neighbors (those with overlapping intra-grid partitions) and immediately continues to the next step of Algorithm 5.2 without waiting for the message to be received at the target processor. During the steps where x is asynchronously received, a processor checks for new values of x. If no new values have arrived, the processor continues using its most recently received version of x.

At scale, communication latencies are significantly higher in distributed memory than in shared memory. Therefore, in order to scale to massively parallel distributed memory systems, asynchronous EBPX-Chebyshev must converge as communication delays increase. In Section 7.5 we will show that not only does asynchronous EBPX-Chebyshev converge, but, given sufficiently large delays, it converges faster than synchronous BPX-Chebshev, where the synchronous BPX baseline is implemented in the widely-used Hypre package [18].

6. Simulations of the asynchronous preconditioned Chebyshev method.

The asynchronous preconditioned Chebyshev method will be tested on a shared memory parallel computer in Section 7. Such results can be difficult to interpret, however, since they depend on characteristics of the computer and software, such as communi-

cation delays. Thus, in this section, we first investigate the behavior of the asynchronous preconditioned Chebyshev method via simulation. Simulation parameters can be altered to investigate the behavior of asynchronous methods in different situations.

To simulate asynchronous iterations, at each time instant, t, we must choose $\Psi(t)$ and $z_{ij}(t)$. We do this randomly using an *update probability* and a *communication delay* (comm-delay) bound. Strikwerda [48] analyzed a very similar model. For simplicity, we assume that the number of processors is equal to the number of unknowns. The update probability is the probability that $i \in \Psi(t)$ at time instant t for any processor i. For example, if the update probability is 0.1, an average of 10% of the processors will update their unknowns at each time instant. For a smaller update probability, the simulation is more asynchronous in the sense that fewer processors are updating at any time instant.

The comm-delay bound controls how far back in time we can look when choosing the mapping $z_{ij}(t)$. This simulates the speed at which data is communicated between processors, e.g., large comm-delay bounds simulate a system with slow communication. More precisely, for a comm-delay bound of d, $z_{ij}(t)$ must satisfy $t-d \leq z_{ij}(t) \leq t$, and $z_{ij}(t)$ is chosen uniformly randomly in the range $[t, t-\min(d, s_{ij})]$ where s_{ij} is the previous time instant from which processor i used information from processor j. The variable s_{ij} is needed to ensure processor i will not use information from processor j that is older than what has already been used, as one might expect on a realistic computer system. For example, if d is 10 and t is 200, processor i cannot read farther back than 190. If s_{ij} is 195, i.e., the last version of x_j used to update x_i is from time instant 195, then processor i cannot read farther than 195 when using information from processor j. Note that since $z_{ij}(t)$ is chosen randomly, processor i will in general not read all values of x_j from the sequence of time instants that are simulated. This can simulate a scenario where a processor can write to memory multiple times before another processor reads from the same memory location once.

For our simulation experiments in this section, the test problem is the five-point centered-difference discretization of the two-dimensional (2-D) Poisson equation on a 32×32 grid. For BPX, we used Galerkin coarse grid matrices, full weighting to construct the interpolation and restriction matrices, and Jacobi with weight 4/5 as a smoother. To estimate the extremal eigenvalues needed for BPX-Chebyshev, we used the sparse eigensolver implemented in Matlab. Given a preconditioner, the same estimates are used in both the synchronous and asynchronous versions of Chebyshev.

Figure 7 shows the convergence history of the synchronous Jacobi- and BPX-preconditioned Chebyshev methods and of the asynchronous Jacobi- and EBPX-preconditioned Chebyshev methods. The x-axis is the number of scalar updates to the unknowns divided by the number of matrix rows, n. Different values of the update probability and comm-delay bound are used, which affect the results of the simulated asynchronous methods. Since the result of a simulated asynchronous method is not deterministic, the asynchronous methods are run multiple times; the result for each of these runs is shown. We observe that asynchronous EBPX-Chebyshev converges in all tested cases whereas the convergence rate of asynchronous Jacobi-Chebyshev is sensitive to the update probability and comm-delay bound. Specifically, asynchronous Jacobi-Chebyshev converges only in the case with least asynchrony (update probability of 0.9 and comm-delay bound of 1).

We now experiment with adjusting the Chebyshev weight ω. Figure 8 shows the convergence of the methods with three different values of the adjustment. For larger adjustments, the asynchronous Jacobi-Chebyshev method now converges, albeit slowly. The effect of the adjustments will be further investigated for the asynchronous

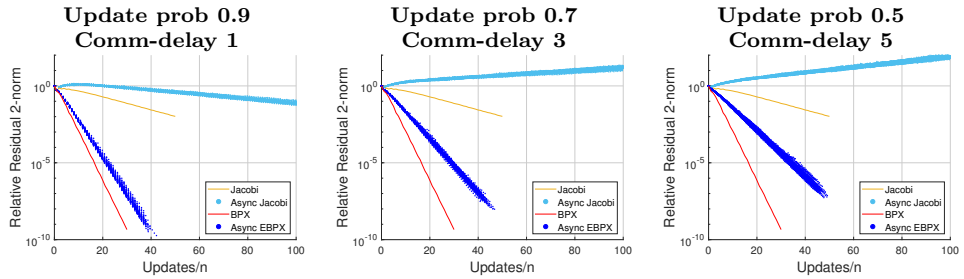


FIG. 7. *Simulation of the asynchronous Chebyshev method with asynchronous Jacobi and asynchronous EBPX preconditioners (light blue and dark blue, respectively), for different simulation update probabilities and comm-delay bounds. For comparison, the convergence of synchronous Chebyshev with Jacobi and BPX preconditioners (yellow and red, respectively) are shown. The x-axis is the number of solution component updates divided by n, the number of matrix rows. The results show that the asynchronous Chebyshev method converges consistently with the new asynchronous EBPX preconditioner, but not with the asynchronous Jacobi preconditioner.*

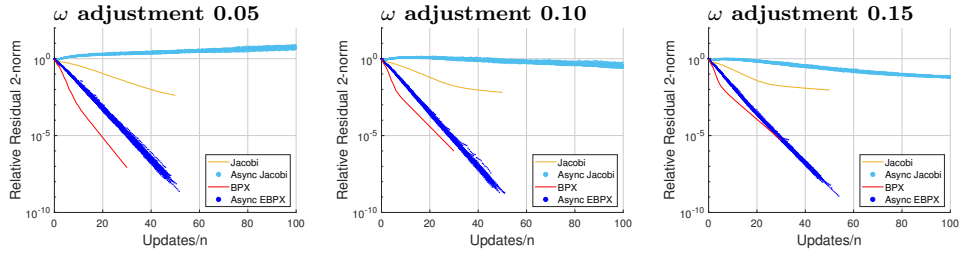


FIG. 8. *Simulation results when the value of ω is adjusted at each iteration of the synchronous and asynchronous Chebyshev methods. The simulations used an update probability of 0.7 and a comm-delay bound of 3 (compare to Figure 7, center plot). For adjustments of 0.10 and 0.15, the asynchronous Chebyshev method with asynchronous Jacobi preconditioning now converges.*

EBPX-Chebyshev method in Section 7.

From other simulations (results not shown), we further find that the asynchronous EBPX-Chebyshev method continues to converge well when simulated using a wide range of simulation parameters, including for the combination of small update probability 0.1 and large comm-delay bound 1000.

Finally, we simulate our asynchronous methods for linear systems of different sizes. Figure 9 shows the relative residual 2-norm after 20 time instants (or 20 iterations in the synchronous case) versus the number of grid points in one dimension (grid length) in the same finite-difference discretization of the 2-D Poisson problem. Results for synchronous and asynchronous versions of both Jacobi-Chebyshev and BPX-Chebyshev are shown. Again, the results of multiple runs (multiple dots) are shown for the asynchronous cases. These results show that for large problem sizes, asynchronous Jacobi-Chebyshev diverges (the relative residual norm after 20 time instants is greater than 1). In contrast, asynchronous EBPX-Chebyshev converges with a rate independent of the grid length.

7. Numerical tests on a parallel shared memory computer.

7.1. Testing methodology. Numerical tests were conducted on a computer with a 32-core Intel Xeon E5-2698 CPU. All solvers were implemented for multithreaded execution using OpenMP and run using 32 threads with "compact" thread

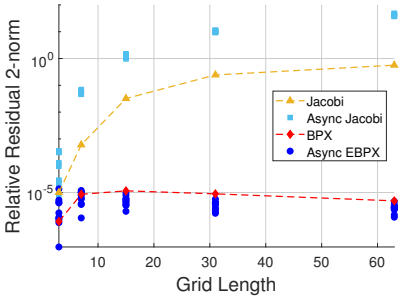


FIG. 9. *Relative residual 2-norm versus grid length after 20 time instants or 20 iterations. Results for synchronous and asynchronous versions of both Jacobi-Chebyshev and EBPX-Chebyshev are shown. Simulations of the asynchronous methods used update probability 0.7 and comm-delay bound 2. The asynchronous EBPX-Chebyshev method appears to converge independently of grid size for this problem.*

affinity.

Three test problems were used:

- **27pt**: A 27-point discretization of a three-dimensional (3-D) anisotropic diffusion equation in a cube,

$$-\nabla \cdot (c\nabla u) = f,$$

  where the fixed diffusion tensor c is diagonal with diagonal entries $c_x = 10$, $c_y = 5$, and $c_z = 1$.
- **conv-diff**: A 7-point discretization of a 3-D anisotropic convection-diffusion equation in a cube,

$$-\nabla \cdot (c\nabla u) + a \cdot \nabla u = f,$$

  with the same fixed diffusion tensor as above, and with fixed convection velocity a with components $a_x = 5$, $a_y = 2$, and $a_z = 1$.
- **square-disc**: An unstructured finite element discretization of the 2-D isotropic diffusion equation, using curvilinear triangular elements, generated by MFEM [2]. The domain is a square with a circle-shaped hole in the center.

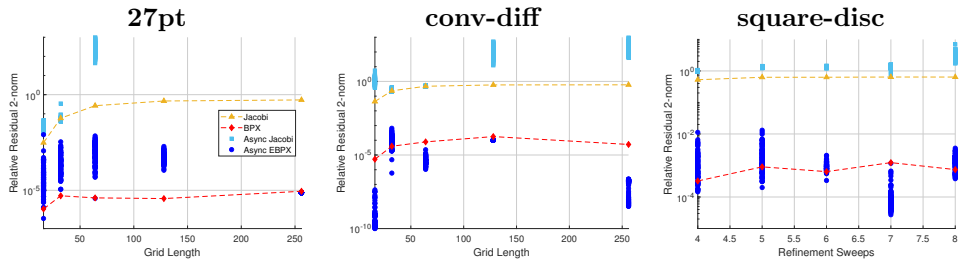
With the exception of the results in Section 7.2 in which different problem sizes are considered, we used a grid of size $32 \times 32 \times 32$ for the two finite difference problems, and a system with 79,616 unknowns for the finite element problem.

To generate the multigrid components needed for BPX, we used BoomerAMG [31] from the Hypre package [18]. For multigrid smoothing, the ℓ_1-Jacobi smoother was used in all cases [7], which provided faster convergence than standard Jacobi smoothing. To compute the extremal eigenvalues of the preconditioned matrix needed for the Chebyshev method, we used the SLEPc package [32]. We found that estimating these eigenvalues using the Arnoldi method to a relative convergence tolerance of 0.1 did not significantly change our results. For a given preconditioned matrix, the same estimates of the spectrum were used for both the synchronous and asynchronous runs of the preconditioned Chebyshev method.

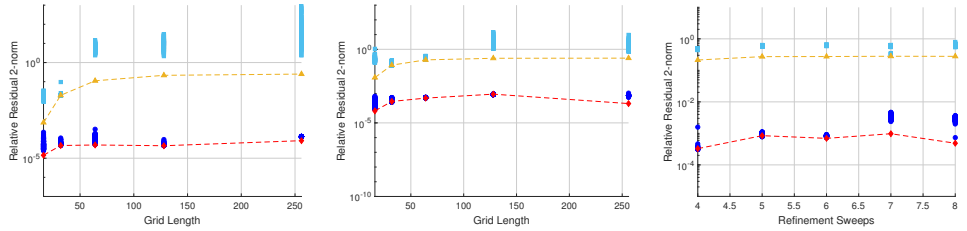
7.2. Results for different problem sizes. Figure 10 compares the relative residual 2-norm achieved after 20 iterations of the synchronous preconditioned Chebyshev method and 20 local iterations of the asynchronous preconditioned Chebyshev method for different problem sizes. The problem size is specified by the number of grid points along one dimension (grid length) for the finite difference problems (giving

4096 to 16,777,216 matrix rows for grid lengths 16 to 256) or the number of refinement sweeps for the finite element problem (giving 20,096 to 5,052,416 matrix rows for 4 to 8 refinement sweeps).

For the synchronous Chebyshev method, the Jacobi and BPX preconditioners were used. For the asynchronous Chebyshev method, the corresponding asynchronous Jacobi and asynchronous EBPX preconditioners were used. Since the result of the asynchronous methods is not deterministic, the asynchronous methods were run 50 times for each problem size, and each data point in the figure corresponds to one of these runs. The figure also shows the difference between no adjustment of the Chebyshev ω weight, and an adjustment of this weight by decreasing it by 0.075 (see Section 4 for a discussion on how ω is adjusted).



(a) No ω adjustment.



(b) ω adjustment 0.075.

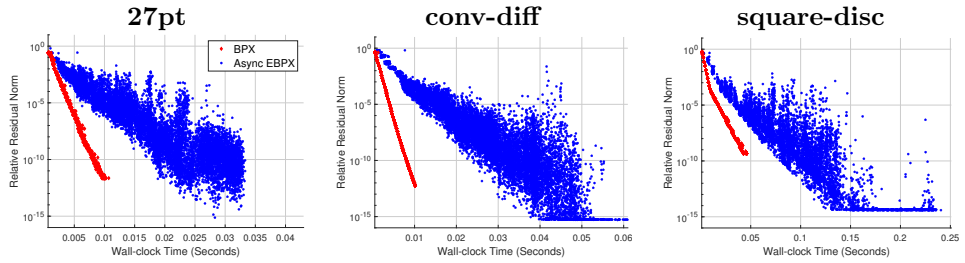
FIG. 10. *Relative residual 2-norm achieved after 20 iterations of the synchronous preconditioned Chebyshev method and 20 local iterations of the asynchronous preconditioned Chebyshev method for different problem sizes (grid length or refinement sweeps). For the asynchronous methods, each dot represents one run, and the spread of the dots indicate the variation in the results. When the Chebyshev ω weight is decreased by 0.075 (second row of plots), the variations in the results are much smaller.*

We observe that the convergence of synchronous Jacobi-Chebyshev is slow, and asynchronous Jacobi-Chebyshev often diverges for the three test problems. On the other hand, synchronous BPX-Chebyshev converges very well, and shows convergence rates that are essentially independent of the problem size. Asynchronous EBPX-Chebyshev has good convergence, but convergence is irregular for different problem sizes when ω is not adjusted. The irregular behavior may be due to the fact that different problem sizes may be better or worse partitioned when using 32 threads. Asynchronous EBPX-Chebyshev convergence also shows no strong degradation with increasing problem size. When ω is adjusted, the convergence is more regular and, in some cases, faster for smaller problem sizes (e.g., for the 27pt problem).

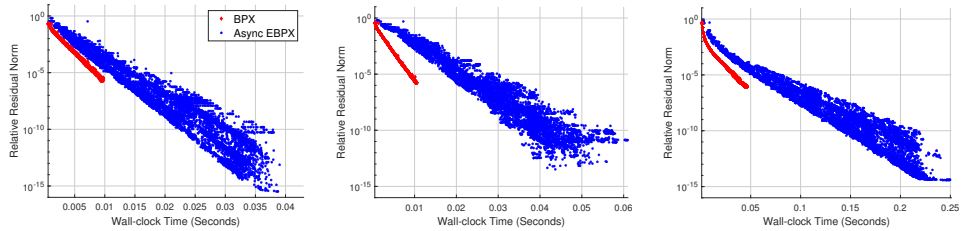
Overall, these results confirm the suggestion from our observations of asynchronous second-order Richardson and simulations in Section 6 that ill-conditioning is detrimental to asynchronous Chebyshev convergence, while asynchronous Chebyshev can be effective with good preconditioning.

7.3. Adjusting the Chebyshev weight. We now further examine the effect of adjusting the Chebyshev weight, ω. Figure 11 shows the relative residual 2-norm versus wall-clock time of synchronous BPX-Chebyshev and asynchronous EBPX-Chebyshev, with and without adjustment of ω. Each dot in the plot represents one run of different durations; the relative residual norm was measured after the end of the run to produce one dot in the plot.

We observe that the convergence of the asynchronous Chebyshev method has smaller variation with the adjustment of ω. However, the adjustment also leads to a decrease in the average convergence rate. These observations are consistent with those in Figure 5 for the second-order Richardson method where the β parameter is adjusted.



(a) No ω adjustment.



(b) ω adjustment 0.2.

FIG. 11. *Relative residual 2-norm versus wall-clock time for several runs of the synchronous and asynchronous preconditioned Chebyshev method (BPX and EBPX preconditioning, respectively). When ω is adjusted down by 0.2, the variation in convergence is smaller.*

7.4. Tests with a background process. We now consider a case in which heterogeneity is present in the computing environment. Figure 12 shows the relative residual 2-norm versus wall-clock time of synchronous BPX-Chebyshev and asynchronous EBPX-Chebyshev where an instance of the Hypre [18] code is running in the background on one core. As in the previous experiments, asynchronous Chebyshev runs in the foreground using 32 threads, with one thread on each of the 32 cores.

Compared to the results in Figure 11, we observe that asynchronous EBPX-Chebyshev is significantly faster than synchronous BPX-Chebyshev, with the exception of a few runs for the 27pt problem. For example, for relative residual norm 10^{-10}, asynchronous EBPX-Chebyshev is more than 10 times faster than synchronous BPX-Chebyshev. The cost of global synchronization in synchronous BPX-Chebyshev is significantly increased by the background process.

7.5. Asynchronous stopping criterion and artificial load imbalance. We now describe an asynchronous stopping criterion for shared memory asynchronous iterative methods and test it for the asynchronous preconditioned Chebyshev method.

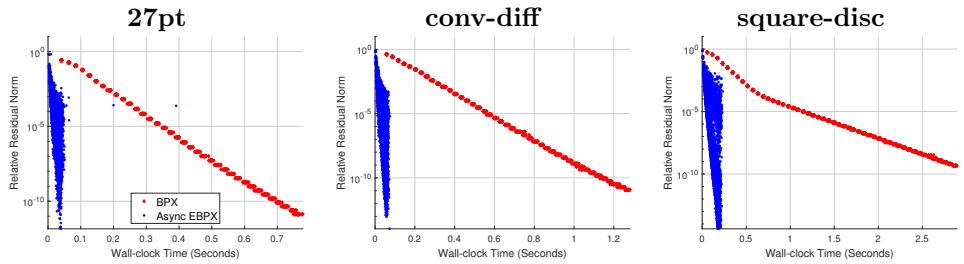


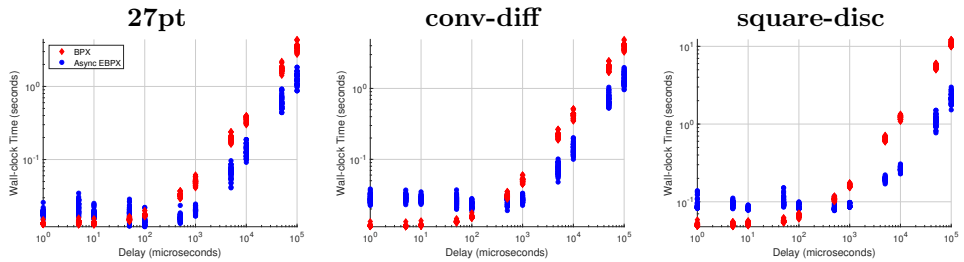
FIG. 12. *Relative residual 2-norm versus wall-clock time for synchronous BPX-Chebyshev and asynchronous EBPX-Chebyshev while a separate program runs on one core in the background.*

The stopping criterion is implemented with a *global convergence thread* Ξ and *local convergence threads* $\xi_0, \xi_1, \ldots, \xi_{\ell-1}$, where ξ_i checks if the threads in $\mathcal{P}_i$ have satisfied a local convergence criterion. In our implementation, $\xi_i \in \mathcal{P}_i$ and, therefore, ξ_i performs updates alongside the other threads in $\mathcal{P}_i$. As in standard synchronous iterative methods, we use a tolerance τ on the residual norm and an upper limit σ on the number of updates. Thread ξ_i uses τ and σ to check if *local* convergence has been satisfied by the threads in $\mathcal{P}_i$. Specifically, ξ_i checks if the residual norm for block-row i has dropped below τ or if the number of updates by the threads in $\mathcal{P}_i$ has exceeded σ (note that the threads in $\mathcal{P}_i$ will perform the same number of updates since they synchronize with each other). To compute the residual norm for block-row i, the threads in $\mathcal{P}_i$ perform a parallel reduction. If the residual norm falls below τ, ξ_i atomically increments the *residual counter* and atomically writes the residual norm to a portion of memory that is visible to Ξ. If the number of updates by the threads in $\mathcal{P}_i$ exceeds σ, ξ_i atomically increments the *update counter*. After the local convergence criterion has been satisfied, the threads in $\mathcal{P}_i$ begin checking for global convergence.

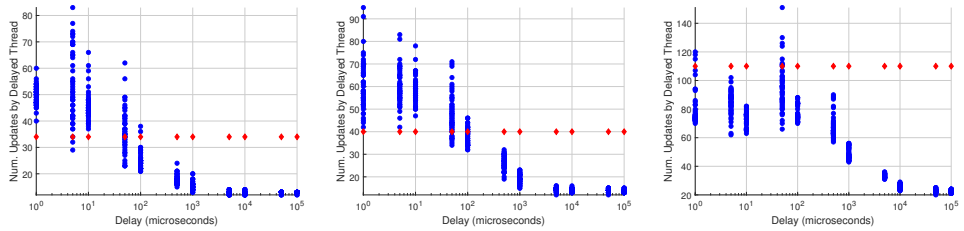
For global convergence, a thread terminates when either the update counter has reached the number of block-rows (this indicates that all threads have performed at least σ updates) or the *residual flag* is set to true. Thread Ξ is responsible for modifying the residual flag. Once the value of the residual counter is equal to the number of block-rows, which indicates that all threads have satisfied their local residual norm convergence criterion, Ξ sums the block-row residual norms to get the global residual norm. If the global residual norm has dropped below τ, Ξ atomically sets the residual flag to true. In our implementation, Ξ is also assigned to a block-row and carries out this summation after each of its own updates. Note that the concept of an iteration is well-defined for synchronous methods but not for asynchronous methods. In this paper we say that we have done σ iterations of an asynchronous method if all threads have performed at least σ updates.

Note that this convergence detection scheme is non-scalable since a single thread reads data from all other threads. For a distributed memory implementation, more sophisticated convergence detection must be used [5, 6, 36, 37, 9].

We now test this stopping criterion for the asynchronous preconditioned Chebyshev method using relative residual norm tolerance 10^{-10}. To stress the test, we artificially slow down one thread. Specifically, one thread sleeps and is thus *delayed* for some number of microseconds after every update of its unknowns. This can model, for example, a heterogeneous computing environment, i.e., a scenario in which some processors compute more slowly than others due to the underlying hardware.



(a) Timings for different values of the delay.



(b) Number of updates performed by the delayed thread.

FIG. 13. *Timings when one thread is artificially slowed down, for different values of the delay. Synchronous BPX-Chebyshev and asynchronous EBPX-Chebyshev are compared. An asynchronous stopping criterion is used with relative residual norm tolerance of* 10^{-10}. *The number of updates per unknown performed by the delayed thread is also shown.*

Figure 13 shows the results for different values of the delay, from 1 to 100,000 microseconds. For small delays (and for no delays, not shown), synchronous BPX-Chebyshev is faster than asynchronous EBPX-Chebyshev. In this case, the benefits of asynchronous execution do not outweigh the higher computational cost of EBPX-Chebyshev. For large delays, larger than approximately 0.001 seconds, the reverse is true and asynchronous EBPX-Chebyshev is faster. In this case, the number of updates performed by the delayed thread is much smaller than the number of updates performed by the other threads, yet convergence can still be attained. The updates performed by the other threads help make progress toward the global solution by making the equations being assigned to them consistent among themselves, while one thread is delayed. This has also been observed for the asynchronous Jacobi method [56]. (Also compare this to the effect in Figure 2.)

We can extend these results to the distributed memory case, as in [56], where the speedup of asynchronous over synchronous Jacobi increases with the communication delays. The artificial delays can be thought of as communication delays in a massively parallel distributed memory system, which are substantially higher than shared memory accesses. Our results show that significant progress to the solution is made by performing frequent local updates while data from remote processors are in-flight. Since scaling up a distributed system also scales up communication latencies, asynchronous execution of EBPX-Chebyshev can be beneficial for a sufficiently large system.

We also note in Figure 13b that, for small delays, the average number of updates per unknown performed by the delayed thread can be larger than in the synchronous case in order to satisfy the convergence criterion. This can be due to a combination of the imprecision of the convergence test or slower convergence of the asynchronous method.

8. Conclusion. This paper introduced an asynchronous Chebyshev iterative method and an asynchronous multigrid preconditioner, called asynchronous EBPX. It was shown experimentally that an adjustment of the parameter values of the asynchronous preconditioned Chebyshev method can make convergence less variable, although slower. On a dedicated shared memory computer system, the asynchronous EBPX-Chebyshev method is somewhat slower than the synchronous BPX-Chebyshev method. However, the asynchronous method may be much faster than the synchronous method when processor cores are shared with other jobs. Asynchronous versions of iterative methods may be of most benefit when otherwise tightly-coupled solvers must be run in non-dedicated computing environments.

Acknowledgments. The authors are grateful for invaluable discussions over long periods of time with Daniel B. Szyld, Andreas Frommer, Erik Boman, Christian Glusa, Ulrike M. Yang, and Stephen F. McCormick.

REFERENCES

[1] M. F. Adams, J. Brown, M. Knepley, and R. Samtaney, *Segmental refinement: A multigrid technique for data locality*, SIAM Journal on Scientific Computing, 38 (2016), pp. C426–C440.

[2] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. C. V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, and S. Zampini, *MFEM: A modular finite element methods library*, Computers & Mathematics with Applications, 81 (2021), pp. 42–74.

[3] H. Anzt, S. Tomov, M. Gates, J. Dongarra, and V. Heuveline, *Block-asynchronous multigrid smoothers for GPU-accelerated systems*, Proceedings of the International Conference on Computational Science (ICCS), 9 (2012), pp. 7–16.

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

[5] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, *An Efficient and Robust Decentralized Algorithm for Detecting the Global Convergence in Asynchronous Iterative Algorithms*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 240–254.

[6] J. M. Bahi, S. Contassot-Vivier, R. Couturier, and F. Vernier, *A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms*, IEEE Transactions on Parallel and Distributed Systems, 16 (2005), pp. 4–13.

[7] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, *Multigrid smoothers for ultraparallel computing*, SIAM Journal on Scientific Computing, 33 (2011), pp. 2864–2887.

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

[9] G. G. Benissan and F. Magoulès, *Protocol-free asynchronous iterations termination*, Adv. Eng. Softw., 146 (2020), p. 102827.

[10] D. Bertsekas and J. Tsitsiklis, *Parallel and Distributed Computation, Numerical Methods*, Prentice Hall, Englewood Cliffs N.J., 1989.

[11] J. H. Bramble, J. E. Pasciak, and J. Xu, *Parallel multilevel preconditioners*, Mathematics of Computation, 55 (1990), pp. 131–144.

[12] A. Brandt, *Multi-level adaptive solutions to boundary-value problems*, Mathematics of Computation, 31 (1977), pp. 333–390.

[13] C. Brezinski, G. Meurant, and M. Redivo-Zaglia, *A Journey through the History of Numerical Linear Algebra*, Society for Industrial and Applied Mathematics, 2022.

[14] F. Chaouqui, E. Chow, and D. B. Szyld, *Asynchronous domain decomposition methods for nonlinear PDEs*, Electronic Transactions on Numerical Analysis, 58 (2023), pp. 22–42.

[15] D. Chazan and W. Miranker, *Chaotic relaxation*, Linear Algebra and its Applications, 2 (1969), pp. 199 – 222.

[16] E. Chow, A. Frommer, and D. B. Szyld, *Asynchronous Richardson iterations: Theory and practice*, Numerical Algorithms, 87 (2021), p. 1635–1651.

[17] L. Erlandson, Z. Atkins, A. Fox, C. J. Vogl, A. Miedlar, and C. Ponce, *Resilient s-ACD for asynchronous collaborative solutions of systems of linear equations*, 2023 18th Conference on Computer Science and Intelligence Systems (FedCSIS), (2023), pp. 441–450.

[18] R. D. Falgout and U. M. Yang, *hypre: A library of high performance preconditioners*, in Computational Science — ICCS 2002, P. M. A. Sloot, A. G. Hoekstra, C. J. K. Tan, and J. J. Dongarra, eds., Berlin, Heidelberg, 2002, Springer Berlin Heidelberg, pp. 632–641.

[19] A. Frommer, H. Schwandt, and D. B. Szyld, *Asynchronous weighted additive Schwarz methods*, Electronic Transactions on Numerical Analysis, 5 (1997), pp. 48–61.

[20] A. Frommer and D. B. Szyld, *On asynchronous iterations*, J. Comput. Appl. Math., 123 (2000), pp. 201–216.

[21] G. Gbikpi-Benissan and F. Magoulès, *Asynchronous multiplicative coarse-space correction*, SIAM Journal on Scientific Computing, 44 (2022), pp. C237–C259.

[22] P. Ghysels, T. J. Ashby, K. Meerbergen, and W. Vanroose, *Hiding global communication latency in the GMRES algorithm on massively parallel machines*, SIAM Journal on Scientific Computing, 35 (2013), pp. C48–C71.

[23] L. Giraud and P. Spiteri, *Implementation of parallel solutions for nonlinear boundary value problems*, in Parallel Computing'91, D. J. Evans, G. R. Joubert, and H. Liddel, eds., Elsevier Science Publishers, 1992, pp. 203–211.

[24] C. Glusa, E. G. Boman, E. Chow, S. Rajamanickam, and P. Ramanan, *Asynchronous one-level and two-level domain decomposition solvers*, in Domain Decomposition Methods in Science and Engineering XXV, Lecture Notes in Computational Science and Engineering, vol. 138, Springer, 2020, pp. 134–142.

[25] C. Glusa, E. G. Boman, E. Chow, S. Rajamanickam, and D. B. Szyld, *Scalable asynchronous domain decomposition solvers*, SIAM Journal on Scientific Computing, 42 (2020), pp. C384–C409.

[26] G. H. Golub and M. L. Overton, *The convergence of inexact Chebyshev and Richardson iterative methods for solving linear systems*, Numerische Mathematik, 53 (1988), pp. 571–593.

[27] M. Griebel, *Multilevel algorithms considered as iterative methods on semidefinite systems*, SIAM Journal on Scientific Computing, 15 (1994), pp. 547–565.

[28] M. Griebel and P. Oswald, *Greedy and randomized versions of the multiplicative Schwarz method*, Linear Algebra and its Applications, 437 (2012), pp. 1596 – 1610.

[29] L. Hart and S. McCormick, *Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors: Basic ideas*, Parallel Computing, 12 (1989), pp. 131–144.

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

[31] V. E. Henson and U. M. Yang, *BoomerAMG: A parallel algebraic multigrid solver and preconditioner*, Applied Numerical Mathematics, 41 (2002), pp. 155–177.

[32] V. Hernandez, J. E. Roman, and V. Vidal, *SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems*, ACM Trans. Math. Software, 31 (2005), pp. 351–362.

[33] M. Hoemmen, *Communication-avoiding Krylov subspace methods*, PhD thesis, University of California, Berkeley, 2010.

[34] B. Lee, S. McCormick, B. Philip, and D. Quinlan, *Asynchronous fast adaptive composite-grid methods: Numerical results*, SIAM Journal on Scientific Computing, 25 (2003), pp. 682–700.

[35] B. Lee, S. McCormick, B. Philip, and D. Quinlan, *Asynchronous fast adaptive composite-grid methods for elliptic problems: Theoretical foundations*, SIAM Journal on Numerical Analysis, 42 (2004), pp. 130–152.

[36] F. Magoulès and G. Gbikpi-Benissan, *JACK: an asynchronous communication kernel library for iterative algorithms*, The Journal of Supercomputing, 73 (2017), pp. 3468–3487.

[37] F. Magoulès and G. Gbikpi-Benissan, *JACK2: An MPI-based communication library with non-blocking synchronization for asynchronous iterations*, Advances in Engineering Software, 119 (2018), pp. 116 – 133.

[38] F. Magoulès, D. Szyld, and C. Venet, *Asynchronous optimized Schwarz methods with and without overlap*, Numerische Mathematik, 137 (2017), pp. 199 – 227.

[39] J. Miellou, *Algorithmes de relaxation chaotique à retards*, RAIRO Analyse Numérique, 9 (1975), pp. 55–82.

[40] J. C. Miellou, L. Giraud, A. Laouar, and P. Spiteri, *Subdomain decomposition methods with overlapping and asynchronous iterations*, in Progress in partial differential equations: the Metz surveys, M. Chipot and J. S. J. Paulin, eds., vol. 249, Longman Scientific and Technical, Pitman Research Notes in Mathematics, 1991, pp. 166–183.

[41] C. D. Murray and T. Weinzierl, *Stabilized asynchronous fast adaptive composite multigrid using additive damping*, Numerical Linear Algebra with Applications, 28 (2021), p. e2328.

[42] D. Quinlan, *Adaptive Mesh Refinement for Distributed Parallel Architectures*, PhD thesis, University of Coloardo Denver, 1993.
[43] F. Robert, M. Charnay, and F. Musy, *Itérations chaotiques série-parallèle pour des équations non-linéaires de point fixe*, Aplikace Matematiky, 20 (1975), pp. 1–38.
[44] J. L. Rosenfeld, *A case study on programming for parallel processors*, Tech. Report RC-1864, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1967.
[45] J. L. Rosenfeld, *A case study in programming for parallel-processors*, Commun. ACM, 12 (1969).
[46] Y. Saad, *Iterative methods for linear systems of equations: A brief historical journey*, Contemporary Mathematics, edited by: Brenner, S., Shparlinski, I., Shu, C.-W., and Szyld, D., American Mathematical Society, Providence, Rhode Island, 754 (2020), pp. 197–215.
[47] P. Spiteri, *Parallel asynchronous algorithms: A survey*, Advances in Engineering Software, 149 (2020), p. 102896.
[48] J. C. Strikwerda, *A probabilistic analysis of asynchronous iteration*, Linear Algebra Appl., 349 (2002), pp. 125–154.
[49] K. Świrydowicz, J. Langou, S. Ananthan, U. Yang, and S. Thomas, *Low synchronization Gram–Schmidt and generalized minimal residual algorithms*, Numerical Linear Algebra with Applications, 28 (2021), p. e2343.
[50] R. S. Varga, *Matrix Iterative Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1962. A revised and expanded edition was published by Springer in 2000.
[51] P. S. Vassilevski, *Multilevel Block Factorization Preconditioners*, Springer-Verlag New York, 2008.
[52] P. S. Vassilevski, *Lecture notes on multigrid methods*, Tech. Report LLNL-TR-439511, Lawrence Livermore National Laboratory, 2010.
[53] P. S. Vassilevski and U. M. Yang, *Reducing communication in algebraic multigrid using additive variants*, Numerical Linear Algebra with Applications, 21 (2014), pp. 275–296.
[54] J. Wolfson-Pou and E. Chow, *Convergence models and surprising results for the asynchronous Jacobi method*, IEEE International Parallel and Distributed Processing Symposium (IPDPS), (2018), pp. 940–949.
[55] J. Wolfson-Pou and E. Chow, *Asynchronous multigrid methods*, 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), (2019), pp. 101–110.
[56] J. Wolfson-Pou and E. Chow, *Modeling the asynchronous Jacobi method without communication delays*, Journal of Parallel and Distributed Computing, 128 (2019), pp. 84 – 98.
[57] S. Wu, Z. Xie, H. Chen, S. Di, X. Zhao, and H. Jin, *Dynamic acceleration of parallel applications in cloud platforms by adaptive time-slice control*, 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), (2016), pp. 343–352.
[58] I. Yamazaki, E. Chow, A. Bouteiller, and J. Dongarra, *Performance of asynchronous optimized Schwarz with one-sided communication*, Parallel Computing, 86 (2019), pp. 66–81.