SIAM J. SCI. COMPUT.
Vol. 37, No. 2, pp. C169–C193

FINE-GRAINED PARALLEL INCOMPLETE LU FACTORIZATION*

EDMOND CHOW† AND AFTAB PATEL†

Abstract. This paper presents a new fine-grained parallel algorithm for computing an incomplete LU factorization. All nonzeros in the incomplete factors can be computed in parallel and asynchronously, using one or more sweeps that iteratively improve the accuracy of the factorization. Unlike existing parallel algorithms, the amount of parallelism is large irrespective of the ordering of the matrix, and matrix ordering can be used to enhance the accuracy of the factorization rather than to increase parallelism. Numerical tests show that very few sweeps are needed to construct a factorization that is an effective preconditioner.

Key words. preconditioning, parallel computing, incomplete factorization

AMS subject classifications. 65Y05, 65F08, 65F50

DOI. 10.1137/140968896

1. Introduction.

The parallel computation of incomplete LU (ILU) factorizations has been a subject of much interest since the 1980s. Although ILU factorizations have been very useful as preconditioners in sequential environments, they have been less useful in parallel environments where more parallelizable algorithms are available. In this paper we propose a completely new algorithm for computing ILU factorizations in parallel. The algorithm is easy to parallelize and has much more parallelism than existing approaches. Each nonzero of the incomplete factors L and U can be computed in parallel with an asynchronous iterative method, starting with an initial guess. A feature of the algorithm is that, unlike existing approaches, it does not rely on reordering the matrix in order to enhance parallelism. Reordering can instead be used to enhance convergence of the solver.

The new algorithm addresses highly parallel environments, such as Intel Xeon Phi, where there may be more processing cores than the parallelism available in existing methods. In this paper we show the potential advantages of the new algorithm on current hardware, but the algorithm is expected to be even more advantageous than existing algorithms on future hardware with even more cores.

For very large sparse matrices, the best preconditioner is unlikely to be an ILU factorization of the entire matrix. Instead, a multilevel domain decomposition may be preferred, with ILU used in the subdomain solver on each node, especially if it is difficult to exploit the physics of a problem. Thus we focus on parallelism on a single node, which generally involves shared memory and is often called "fine-grained" parallelism.

When using ILU preconditioners in a parallel environment, the sparse triangular solves must also be parallelized. This has also been the subject of much research, and many options are available. In order not to neglect this important point, although it is not our focus, we briefly summarize some options at the end of this paper.

*Submitted to the journal's Software and High-Performance Computing section May 12, 2014; accepted for publication (in revised form) January 14, 2015; published electronically March 19, 2015. 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 DE-SC-0012538, by the National Science Foundation under grant ACI-1306573, and by Intel Corporation.
http://www.siam.org/journals/sisc/37-2/96889.html

†School of Computational Science and Engineering, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0765 (echow@cc.gatech.edu, aypatel@gatech.edu).

Background and related work. Given a sparse matrix A, an incomplete factorization $LU \approx A$ can be computed by a Gaussian elimination process where nonzeros or fill-in are only permitted in specified locations, (i, j) of L and U [34]. We define the sparsity pattern S to be the set of matrix locations where nonzeros are allowed, that is, $(i, j) \in S$ if l_{ij} in matrix L is permitted to be nonzero (in the case $i \geq j$) or if u_{ij} in matrix U is permitted to be nonzero (in the case $i \leq j$). The set S should always include the nonzero locations on the diagonal of the L and U factors so that these factors are nonsingular.

Algorithm 1, from [45], is an example of the conventional procedure for computing an incomplete factorization with sparsity pattern S, with S computed either beforehand or dynamically. The values a_{ij} are entries in A, where $(i, j) \in S$. The factorization is computed "in-place." At the end of the algorithm, the U factor is stored in the upper triangular part of A, and the L factor is unit lower triangular with its strictly lower triangular part stored in the strictly lower triangular part of A. Parallelization is challenging because of the sequential nature of Gaussian elimination and because the factors L and U remain very sparse and lack the large, dense blocks that arise in a *complete* LU factorization.

Algorithm 1: Conventional ILU Factorization.

```
1 for i = 2 to n do
2     for k = 1 to i − 1 and (i, k) ∈ S do
3         a_ik = a_ik / a_kk
4         for j = k + 1 to n and (i, j) ∈ S do
5             a_ij = a_ij − a_ik a_kj
6         end
7     end
8 end
```

Previous work on parallel incomplete factorizations depends on finding rows of the matrix that can be eliminated in parallel. Reordering the matrix can transform the problem into a related one that has more parallelism, although reordering can also affect the accuracy of the resulting factorization. The subject has a long history, beginning in the late 1980s.

The first parallel ILU algorithms were designed for problems on regular grids, where sets of grid points, organized in diagonal lines in two dimensions or hyperplanes in three dimensions, could be eliminated in parallel (e.g., [50, 27]). The generalization of this to irregular problems is an idea often called *level scheduling*, originally used for solving sparse triangular systems, to determine which rows or columns that, due to sparsity, can be eliminated in parallel (e.g., [21, 40, 18, 13, 17]).

To enhance parallelism, it is common to use *multicolor ordering* to reorder the rows and columns of the matrix. The nodes corresponding to the graph of the matrix are colored such that no two adjacent nodes share the same color. Then the matrix is reordered such that like colors are ordered together. Nodes corresponding to the same color can then be eliminated in parallel; see, e.g., [26]. The advantage of using multicolor ordering is that there is much more parallelism. A disadvantage, however, is that multicolor orderings lead to ILU factorizations that are not as good compared to when other orderings are used; see, e.g., [41, 14, 15, 11, 12, 5]. A more serious disadvantage of using reordering to enhance parallelism is that it takes away the ability to reorder a matrix to enhance solver convergence. Especially for difficult problems, such orderings promoting convergence are essential for efficiency and robustness; see,

e.g., [14, 10, 5]. On fine-grained parallel hardware, however, multicolor reordering is one of the few options and has recently been used for parallel ILU implementations on GPUs [30, 22].

Yet another approach for developing parallel ILU factorizations is to use *domain decomposition* which is suitable for distributed memory implementations. Here, the parallelism is *coarse-grained*. The graph corresponding to the matrix is first partitioned into subdomains; then interior nodes of each subdomain are ordered contiguously, subdomain after subdomain, followed by the interface nodes ordered at the end. The incomplete factorization is computed in parallel for each subdomain. Various techniques are then used to eliminate the interface nodes in parallel [31, 28, 53, 23, 24, 32]. There may be limited parallelism for the ILU factorization within each subdomain, but various approximations and other parallel techniques may be used; see, e.g., [35, 1, 43]. This approach also relies on reordering the matrix, but there are no negative effects of reordering when the subdomains are large enough.

The parallel ILU factorization presented in this paper is different from all of these previous approaches. First, parallelism is very fine-grained: individual entries, rather than rows, of the factorization can be computed in parallel, using an iterative algorithm. Second, reorderings are not used to create fine-grained parallelism, and thus any ordering can be used, including those that enhance convergence of the preconditioned iterative method.

2. New parallel ILU algorithm.

2.1. Reformulation of ILU.

The new parallel ILU algorithm is based on the sometimes overlooked property that

$$(LU)_{ij} = a_{ij}, \quad (i,j) \in S, \tag{2.1}$$

where $(LU)_{ij}$ denotes the (i,j) entry of the ILU factorization of the matrix with entries a_{ij}; see [45, Prop. 10.4]. In other words, the factorization is exact on the sparsity pattern S. The original ILU methods for finite-difference problems were interpreted this way [9, 52, 38] before they were recognized as a form of Gaussian elimination and long before they were called incomplete factorizations [34].

Today, an incomplete factorization is generally computed by a procedure analogous to Gaussian elimination. However, any procedure that produces a factorization with the above property is an incomplete factorization equivalent to one computed by incomplete Gaussian elimination. The new fine-grained parallel algorithm interprets an ILU factorization as, instead of a Gaussian elimination process, a problem of computing unknowns l_{ij} and u_{ij} which are the entries of the ILU factorization, using property (2.1) as constraints.

Formally, the unknowns to be computed are

$$\begin{aligned} &l_{ij}, \quad i > j, \quad (i,j) \in S, \\ &u_{ij}, \quad i \le j, \quad (i,j) \in S. \end{aligned}$$

We use the normalization that L has a unit diagonal, and thus the diagonal entries of L do not need to be computed. Therefore, the total number of unknowns is $|S|$, the number of elements in the sparsity pattern S. To determine these $|S|$ unknowns, we use the constraints (2.1), which can be written as

$$\sum_{k=1}^{\min(i,j)} l_{ik} u_{kj} = a_{ij}, \quad (i,j) \in S, \tag{2.2}$$

with the definition that l_{ij} and u_{ij} for (i,j) not in S are equal to zero. Each constraint can be associated with an element of S, and therefore there are $|S|$ constraints. Thus we have a problem of solving for $|S|$ unknowns with $|S|$ equations.

To be sure, these equations are nonlinear (more precisely, they are bilinear), and there are more equations than the number of rows in A. However, there are several potential advantages to computing an ILU factorization this way: (1) the equations can be solved in parallel with fine-grained parallelism; (2) the equations do not need to be solved very accurately to produce a good ILU preconditioner; and (3) we often have a good initial guess for the solution.

We note in passing that there exist other preconditioners that are constructed by minimizing an objective by solving a set of equations, e.g., factorized sparse approximate inverse (FSAI) preconditioners [29]. FSAI preconditioners differ in that the equations to be solved are linear and decouple naturally into independent problems, whereas the equations in our case are generally fully coupled.

2.2. Solution of constraint equations. We now discuss the parallel solution of the system of equations (2.2). Although these equations are nonlinear, we can write an explicit formula for each unknown in terms of the other unknowns. In particular, the equation corresponding to (i,j) can give an explicit formula for l_{ij} (if $i > j$) or u_{ij} (if $i \leq j$):

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right), \tag{2.3}$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}. \tag{2.4}$$

The second of these equations does not need a divide by l_{ii} because $l_{ii} = 1$.

The above equations are in the form $x = G(x)$, where x is a vector containing the unknowns l_{ij} and u_{ij}. It is now natural to try to solve these equations via a fixed-point iteration,

$$x^{(p+1)} = G(x^{(p)}), \quad p = 0, 1, \ldots, \tag{2.5}$$

with an initial guess $x^{(0)}$. Each component of the new iterate $x^{(p+1)}$ can be computed in parallel.

There is a lot of structure in G. When the unknowns l_{ij} and u_{ij} are viewed as entries of matrices L and U, the formula (2.3) or (2.4) for unknown (i,j) depends only on other unknowns in row i of L to the left of j, and in column j of U above i. This is depicted in Figure 1, where the L and U factors are shown superimposed into one matrix. Thus, an explicit procedure for solving the nonlinear equations *exactly* is to solve for the unknowns using (2.3) and (2.4) in a specific order: unknowns in the first row of U are solved (which depend on no other unknowns), followed by those in the first column of L; this is followed by unknowns in the second row of U and the second column of L, etc.; see Figure 2. This ordering is just one of many topological orderings of the unknowns that could be used to solve the nonlinear equations via successive substitution. Another ordering is simply the ordering of the unknowns from left to right and top to bottom (or top to bottom and from left to right) if the unknowns were placed in a sparse matrix. These orderings could be called "Gaussian elimination orderings," since they are the orderings in which the l_{ij} and u_{ij} are produced in

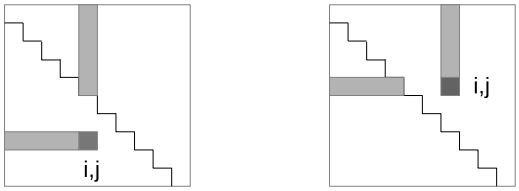

FIG. 1. *Formula for unknown at (i, j) (dark square) depends on other unknowns left of (i, j) in L and above (i, j) in U (shaded regions). The left figure shows dependence for a lower triangular unknown; the right figure shows dependence for an upper triangular unknown.*

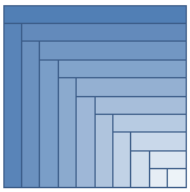

FIG. 2. *Illustration of a Gaussian elimination ordering. Entries $(i, j) \in S$ in darker rows and columns are ordered before those in lighter rows. Ordering the entries from left to right and top to bottom is another Gaussian elimination ordering.*

various forms of Gaussian elimination. To be clear, these orderings are not related to the reordering of the rows and columns of the matrix A, which we seek to avoid.

Different ways of performing the fixed-point iteration (2.5) in parallel give rise to slightly different methods. If the components of $x^{(k+1)}$ are computed in parallel with only "old" values x, then the method corresponds to the nonlinear Jacobi method [39]. At the other extreme, if the components of $x^{(k+1)}$ are computed in sequence with the latest values of x, then we have the nonlinear Gauss–Seidel method. If this latter method visits the equations in Gaussian elimination order, then nonlinear Gauss–Seidel solves the equations in a single sweep, and the solution process corresponds exactly to performing a conventional ILU factorization. In practice, a parallel implementation may perform something between these two extremes.

2.3. Matrix scaling and initial guess. Given a matrix A, we diagonally scale the matrix so that it has a unit diagonal before applying the new ILU algorithm. Assuming that the diagonal of A is positive, the scaled matrix is DAD, where D is the appropriate diagonal scaling matrix. The motivation for this scaling will be explained in section 3.3. Experimentally, we found that this scaling is essential for convergence for many problems, and we use this scaling for all the problems tested in this paper. In the remainder of this paper, we assume that A has been diagonally scaled.

The new ILU algorithm requires an initial guess for the unknowns to begin the fixed-point iterations. Given a matrix A, a simple initial guess is

$$\begin{aligned} (L^{(0)})_{ij} &= (A)_{ij}, \qquad i > j, \quad (i, j) \in S, \\ (U^{(0)})_{ij} &= (A)_{ij}, \qquad i \leq j, \quad (i, j) \in S; \end{aligned} \tag{2.6}$$

that is, we use elements from the strictly lower triangular part of A for the initial guess for L and elements from the upper triangular part of A for the initial guess for U. Outside the sparsity pattern S, the elements are zero. Since A has a unit

diagonal, $U^{(0)}$ also has a unit diagonal. We refer to this as the *standard* initial guess. Other options are also possible. For example, we can scale the rows of $L^{(0)}$ and the columns of $U^{(0)}$ such that the nonlinear constraint equations corresponding to diagonal elements of A are exactly satisfied. We refer to this as the *modified* initial guess.

In many applications, one needs to solve with a sequence of related matrices—for example, one at each time step of a dynamical simulation. In these cases, if the matrix sparsity pattern does not change, the ILU factorization of a matrix at one time step can be used as an initial guess for the factorization at the next time step. We can view this as updating the ILU factorization as the matrix changes. The availability of good initial guesses can make the new ILU algorithm very effective.

2.4. Algorithms and implementation. The new parallel algorithm for computing an ILU factorization is shown as pseudocode in Algorithm 2. Each fixed-point iteration updating all the unknowns is called a "sweep." Compared to the conventional algorithm shown in Algorithm 1, the new algorithm is very different and is actually simpler.

Algorithm 2: Fine-Grained Parallel Incomplete Factorization.

```
1  Set unknowns l_ij and u_ij to initial values
2  for sweep = 1, 2, ... until convergence do
3      parallel for (i, j) ∈ S do
4          if i > j then
5              l_ij = ( a_ij − Σ_{k=1}^{j−1} l_ik u_kj ) / u_jj
6          else
7              u_ij = a_ij − Σ_{k=1}^{i−1} l_ik u_kj
8          end
9      end
10 end
```

The algorithm is parallelized across the elements of S. Given p compute threads, the set S is partitioned into p parts, one for each thread. The threads run in parallel, updating the components of the vector of unknowns, x, asynchronously. Thus the latest values of x are used in the updates. The work associated with each unknown is unequal but is known in advance (generally more work for larger i and j), and the load for each thread can be balanced.

In our specific implementation, we use OpenMP to parallelize the loop across S. This leads to a very simple implementation for us to test the convergence of the parallel algorithm. We used dynamic scheduling of the loop iterations, with a chunk size of 4096. Dynamic scheduling is necessary to balance the load. The large chunk size is necessary for reducing loop overhead. The large chunk size also has the effect of reducing false sharing, i.e., writes to a cache line are generally performed by the same thread. The performance of the algorithm may be improved by statically assigning loop iterations to threads, especially if small problems are of interest.

To develop an efficient implementation of Algorithm 2, it is essential that sparsity be considered when computing (2.3) and (2.4). The inner products in these equations involve rows of L and columns of U. Thus L should be stored in row-major order (using compressed sparse row (CSR) format), and U should be stored in column-major order (using compressed sparse column (CSC) format). An inner product with a row of L and a column of U involves two sparse vectors, and this inner product should be

computed in "sparse-sparse" mode, i.e., using the fact that both operands are sparse.

We note that, as for any asynchronous method, there is no way to guarantee that Algorithm 2 always computes exactly the same factorization for the same inputs. The expectation is that the factorization is nearly the same for the same inputs, but differences in the convergence of the factorization may be much greater across different hardware.

When the matrix A is symmetric positive definite (SPD), the algorithm need only compute one of the triangular factors. Specifically, an incomplete Cholesky (IC) factorization computes $U^T U \approx A$, where U is upper triangular. Algorithm 3 shows the pseudocode for this case, where we have used u_{ij} to denote the entries of U and S_U to denote an upper triangular sparsity pattern.

Algorithm 3: Symmetric Fine-Grained Parallel Incomplete Factorization.

```
1  Set unknowns u_ij to initial values
2  for sweep = 1, 2, ... until convergence do
3      parallel for (i, j) ∈ S_U do
4          s = a_ij − Σ_{k=1}^{i−1} u_ki u_kj
5          if i ≠ j then
6              u_ij = s/u_ii
7          else
8              u_ii = √s
9          end
10     end
11 end
```

In the above algorithms, we assume that the sparsity patterns S are given. Patterns corresponding to level-based ILU factorizations can be computed in parallel [25]. Our approach does not preclude the possibility of using patterns that change dynamically as the nonlinear equations are being solved, i.e., threshold-based ILU factorizations, much like dynamic approaches for choosing the sparsity pattern for sparse approximate inverse preconditioners [19]. We leave this latter point for future work.

3. Convergence theory. Given the constraint equations $x = G(x)$, where $G : D \subseteq \mathbb{R}^m \to \mathbb{R}^m$, we consider the convergence of the synchronous fixed-point method

$$x^{(p+1)} = G(x^{(p)}), \quad p = 0, 1, \ldots, \tag{3.1}$$

and its asynchronous variant. Our analysis focuses on properties of the constraint function, $G(x)$, and its Jacobian, $G'(x)$.

Recall that each equation in $G(x)$ and each unknown in x is associated with one of the m elements in the sparsity pattern S. We begin here by specifying an ordering of the equations in $G(x)$ and the unknowns in x.

DEFINITION 3.1. *An* ordering *of the elements of a sparsity pattern S is a bijective function $g : S \to \{1, \ldots, m\}$, where m is the number of elements in S.*

Given an ordering, the vector of unknowns x is

$$x_{g(i,j)} = \begin{cases} l_{ij} & \text{if } i > j, \\ u_{ij} & \text{if } i \leq j, \end{cases} \tag{3.2}$$

where $(i,j) \in S$, and the function $G(x)$ is

$$G_{g(i,j)}(x) = \begin{cases} \dfrac{1}{x_{g(j,j)}} \left(a_{ij} - \displaystyle\sum_{\substack{1 \le k \le j-1 \\ (i,k),(k,j) \in S}} x_{g(i,k)} x_{g(k,j)} \right) & \text{if } i > j, \\ a_{ij} - \displaystyle\sum_{\substack{1 \le k \le i-1 \\ (i,k),(k,j) \in S}} x_{g(i,k)} x_{g(k,j)} & \text{if } i \le j, \end{cases} \tag{3.3}$$

where a_{ij} is a nonzero of matrix $A \in \mathbb{R}^{n \times n}$. Because of the $x_{g(j,j)}$ terms in the denominators of the $i > j$ case, the domain of definition of G is the set

$$D = \{ x \in \mathbb{R}^m \mid x_{g(j,j)} \neq 0,\ 1 \le j \le n \}. \tag{3.4}$$

The function $G(x)$ contains two types of equations, as shown in (3.3). Similarly, the Jacobian $G'(x)$ contains two types of matrix rows. For a row where $i > j$ (corresponding to the unknown l_{ij}), the partial derivatives are

$$\begin{aligned} \frac{\partial G_{g(i,j)}}{\partial u_{kj}} &= -\frac{l_{ik}}{u_{jj}}, \quad k < j, \\ \frac{\partial G_{g(i,j)}}{\partial l_{ik}} &= -\frac{u_{kj}}{u_{jj}}, \quad k < j, \\ \frac{\partial G_{g(i,j)}}{\partial u_{jj}} &= -\frac{1}{u_{jj}^2} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right). \end{aligned} \tag{3.5}$$

For a row where $i \le j$ (corresponding to the unknown u_{ij}), the partial derivatives are

$$\begin{aligned} \frac{\partial G_{g(i,j)}}{\partial l_{ik}} &= -u_{kj}, \quad k < i, \\ \frac{\partial G_{g(i,j)}}{\partial u_{kj}} &= -l_{ik}, \quad k < i. \end{aligned} \tag{3.6}$$

In the above, we have used l_{ik} instead of $x_{g(i,k)}$, etc., to simplify the notation. Note that the Jacobian is usually very sparse.

3.1. Local convergence. To prove local convergence of the synchronous iteration (3.1), we need to prove that $G(x)$ is F-differentiable at a fixed point x^* and that the spectral radius $\rho(G'(x^*)) < 1$ (Ostrowski's theorem [39]). All the partial derivatives are well defined and continuous on the domain of $G(x)$, and thus $G(x)$ is F-differentiable in particular at a fixed point x^*. To prove the result on the spectral radius, we will in fact prove $\rho(G'(x)) = 0\ \forall x$ in the domain of G.

DEFINITION 3.2. *Consider the partial ordering of the m elements of S,*

$$(1,1:n) \cap S \prec (2:n,1) \cap S \prec \cdots \prec (k,k:n) \cap S \prec (k+1:n,k) \cap S \prec \cdots \prec (n,n),$$

where MATLAB-like indexing notation is used. An ordering consistent with the above partial ordering is an example of a Gaussian elimination (GE) ordering. *(See section* 2.2.*)*

THEOREM 3.3. *The function $G(x)$, with a GE ordering of its equations and unknowns, has strictly lower triangular form; that is,*

$$G_k(x) = G_k(x_1, \ldots, x_{k-1})$$

for $1 \leq k \leq m$.

The proof is immediate from Figures 1 and 2 and from noticing that the component $G_k(x)$ does not contain the unknown x_k. Intuitively, the conventional ILU algorithm computes the unknowns in GE order.

A corollary of the above theorem is that the Jacobian $G'(x)$ has strictly lower triangular structure—in particular, it has all zeros on the diagonal. Thus the Jacobian has spectral radius zero for all x in the domain of $G(x)$. Thus we have proven the following theorem.

THEOREM 3.4. *If* $G : D \to \mathbb{R}^m$ *is given by* (3.3) *and has a fixed point* x^* *in the interior of* D*, then* x^* *is a point of attraction of the iteration* (3.1).

The local convergence of the asynchronous variant of the iteration can be proven similarly. In asynchronous iterations, each component of x is computed by one of the processors. These components are updated using whatever values of components of x are available at the time. We make the standard mild assumptions about asynchronous iterations [16, Def. 2.2]; see also [4]. To prove local convergence of the asynchronous iteration, we appeal to Theorem 4.4 in [16] (see also [8, 48]). To satisfy the conditions of this theorem, we need, as before, the F-differentiability of $G(x)$ at a fixed point x^*. This has already been shown. We also need to show that $\rho(|G'(x^*)|) < 1$. This is trivial since $|G'(x)|$ also has spectral radius zero for all x in the domain of $G(x)$. Thus we have the following result.

THEOREM 3.5. *If* $G : D \to \mathbb{R}^m$ *is given by* (3.3) *and has a fixed point* x^* *in the interior of* D*, then* x^* *is a point of attraction of the asynchronous iteration.*

3.2. Global convergence. The strictly lower triangular structure of $G(x)$ also leads to simple global convergence results (finite termination) of the synchronous and asynchronous iterations. Although they are not practically useful, we state them for completeness. We begin with the following result.

THEOREM 3.6. *Given* $G : D \to \mathbb{R}^m$ *defined by* (3.3)*, if a fixed point of* G *exists, then it is unique.*

Proof. Without loss of generality, assume that G and x are in GE ordering. Due to the strictly lower triangular form of G, the fixed point x^* can be computed by deterministically solving the equations $x = G(x)$ via successive substitution. The process completes if no $x_{g(j,j)}$, $1 \leq j \leq m$, is set to zero. If any $x_{g(j,j)}$ is set to zero, then there is no fixed point of G. ∎

We define a modified Jacobi-type method as identical to the iteration (3.1) except that when a component $x_{g(j,j)}$ equals zero, it is replaced with an arbitrary finite value. We have the following result.

THEOREM 3.7. *If* $G : D \to \mathbb{R}^m$ *has a fixed point* x^**, then the modified Jacobi-type method converges to* x^* *in at most* m *iterations from any initial guess* $x^{(0)}$.

Proof. Without loss of generality, we assume that G and x are in GE ordering. At the first iteration, $x_1^{(1)}$ has its exact value because it does not depend on any unknowns. Also, $x_1^{(p)} = x_1^* \ \forall p \geq 1$. Due to the strictly lower triangular structure of G, $x_2^{(p)} = x_2^* \ \forall p \geq 2$ because $x_2^{(p)}$ depends only on $x_1^{(p)}$ and the latter is exact for $p \geq 1$. Continuing this argument, we have $x_m^{(p)} = x_m^* \ \forall p \geq m$. Thus $x^{(m)} = x^*$. Note that we have made no assumptions about the initial guess $x^{(0)}$. ∎

This theorem and proof may be easily extended to the asynchronous iteration. The asynchronous iteration also terminates after a finite number of steps because, assuming GE ordering, once x_k achieves its exact value, x_{k+1} will achieve its exact value the next time it is updated. The base x_1 achieves its exact value the first time

it is updated.

We remark that although we have finite termination, in floating point arithmetic, the iterates may diverge and overflow before reaching the finite termination condition.

Given $G : D \to \mathbb{R}^m$ in GE ordering, we define a *Gauss–Seidel-type* method, given by

$$x_k^{(p+1)} = G(x_1^{(p+1)}, \ldots, x_{k-1}^{(p+1)}, x_k^{(p)}, \ldots, x_m^{(p)}), \quad p = 0, 1, \ldots. \tag{3.7}$$

The strictly lower triangular structure means that the Gauss–Seidel-type method has the form

$$x_k^{(p+1)} = G(x_1^{(p+1)}, \ldots, x_{k-1}^{(p+1)}), \quad p = 0, 1, \ldots.$$

Trivially, if G is in GE ordering and a fixed point exists, then the Gauss–Seidel-type method (3.7) converges to x^* in one iteration (one sweep). The initial guess $x^{(0)}$ is not utilized.

3.3. Contraction mapping property. The function $G : D \subseteq \mathbb{R}^m \to \mathbb{R}^m$ is a contraction on D if there is a constant $\alpha < 1$ such that

$$\|G(x) - G(y)\| \leq \alpha \|x - y\|$$

$\forall x, y \in D$; see, e.g., [39]. Such a constant can be found if D is convex and there is a constant $\alpha < 1$ satisfying $\|G'(x)\| \leq \alpha \ \forall x \in D$. Although for G defined by (3.3) the domain is not convex, the norm of the Jacobian is still suggestive of whether or not the corresponding fixed-point iteration will converge. We analyze this norm in this section. We note in advance that for a given matrix A, it is difficult to guarantee that $\|G'(x)\| < 1 \ \forall x \in D$. For notation, recall that the components of x are l_{ij} and u_{ij} as defined in (3.2).

Intuitively, for the norm of the Jacobian to be small, the partial derivatives (3.5) and (3.6) should be small (in magnitude), meaning u_{jj} should be large, and l_{ij} and u_{ij} for $i \neq j$ should be small. This suggests that the new ILU algorithm should be effective for matrices A that are diagonally dominant, and that there is a danger of nonconvergence for matrices that are far from diagonally dominant. This motivates us to test nondiagonally dominant problems later in this paper.

It is possible for certain rows and columns of the matrix A to be scaled so that some partial derivatives are very large compared to others. To try to balance the size of the partial derivatives, we symmetrically scale the matrix A to have a unit diagonal before we apply the new ILU algorithm. This explains the matrix scaling described in section 2.3.

We derive expressions for the 1-norm of the Jacobian, which has simpler form than the ∞-norm. In this section, let $\tilde{L}$ denote the strictly lower triangular sparse matrix with nonzeros corresponding to the unknowns l_{ij}. Let $\tilde{U}$ be defined analogously. For simplicity, we assume that the unknowns $u_{jj} = 1$ for $1 \leq j \leq n$, which is the case for the standard initial guess for the fixed-point iterations. Also in this section, instead of $G'(x)$, we will write $G'(\tilde{L}, \tilde{U})$.

THEOREM 3.8. *Given a matrix A and G defined in* (3.3)*, the* 1*-norm of the Jacobian is bounded as*

$$\|G'(\tilde{L}, \tilde{U})\|_1 \leq \max(\|\tilde{U}\|_\infty, \ \|\tilde{L}\|_1, \ \|\tilde{R_{L^*}}\|_1),$$

where $\tilde{R_{L^*}}$ *is the strictly lower triangular part of* $\tilde{R} = A - \tilde{T}$*, where the matrix* $\tilde{T}$ *is*

$$(\tilde{T})_{ij} = \begin{cases} (\tilde{L}\tilde{U})_{ij}, & (i,j) \in S, \\ 0 & \textit{otherwise}. \end{cases}$$

See Appendix A for the proof, which is straightforward. For certain structured matrices, this theorem simplifies, and we can obtain an exact expression for the norm.

THEOREM 3.9. *If* A *is a* 5*-point or* 7*-point finite difference matrix, and if* $\tilde{L}$ *(*$\tilde{U}$*) has sparsity pattern equal to the strictly lower (upper) triangular part of* A*, then for* G *defined in* (3.3)*,*

$$\|G'(\tilde{L}, \tilde{U})\|_1 = \max(\|\tilde{L}\|_{\max},\ \|\tilde{U}\|_{\max},\ \|A_{L^*}\|_1),$$

where A_{L^*} *is the strictly lower triangular part of* A *and* $\|\cdot\|_{\max}$ *denotes the maximum absolute value among the entries in the matrix.*

See Appendix B for the proof. As an example of the application of Theorem 3.9, any diagonally dominant 5-point or 7-point finite-difference matrix has Jacobian 1-norm with value less than 1 when the Jacobian is evaluated at the standard initial guess. Further, for the 5-point or 7-point centered-difference discretization of the Laplacian (in two dimensions or three dimensions, respectively), the Jacobian 1-norm has value 0.5 at the standard initial guess. This result is independent of the number of mesh points used in the discretization. (We assume that these matrices have been diagonally scaled.) Experimentally, using synchronous iterations (3.1), the 1-norm of the Jacobian for a 5-point Laplacian on a 10×10 mesh remains less than 1, increasing monotonically from 0.5 at the initial guess to about 0.686 at convergence.

4. Experimental results. Tests with the new parallel incomplete factorization algorithm are presented in this section. The main question is whether or not the L and U factors converge, and how many sweeps are required for convergence to an effective preconditioner as a function of the number of threads used in the computation.

The test platform is an Intel Xeon Phi coprocessor with 61 cores running at 1.09 GHz. Each core supports four-way simultaneous multithreading.

Convergence of the nonlinear equations can be measured by the 1-norm of the residual,

$$\sum_{(i,j)\in S} \left| a_{ij} - \sum_{k=1}^{\min(i,j)} l_{ik} u_{kj} \right|, \tag{4.1}$$

which we call the "nonlinear residual norm." Recall that all l_{ij} and u_{ij} for (i,j) not in S are equal to zero. For an SPD problem where only one triangular factor is computed, the sum above is taken over the nonzeros of the triangular factor.

We evaluate the factorizations produced by the new incomplete factorization algorithm by using them as preconditioners in the iterative solution of linear systems. Thus our key measure of the quality of the factorization is the *solver iteration count*. The convergence of these linear iterations should not be confused with the convergence of the factorization in the new algorithm. We continue to use the term "sweep" to mean a fixed-point iteration of the new algorithm. Each variable is updated a number of times equal to the number of sweeps.

4.1. Convergence of the algorithm. For the first test of the convergence of the new incomplete factorization algorithm, we used a matrix from the finite element discretization of the Laplacian on a square (two-dimensional (2D)) domain with Dirichlet boundary conditions using linear triangular elements. The matrix contains 203,841 rows and 1,407,811 nonzeros. The rows and columns of the matrix were ordered using reverse Cuthill–McKee (RCM) ordering, which is known to be the preferred ordering to use for incomplete factorizations with this type of matrix [14].

Incomplete factorizations for this matrix were constructed with the new symmetric incomplete factorization algorithm (Algorithm 3), also called incomplete Cholesky (IC), using the modified initial guess. Three different sparsity patterns S were used, corresponding to level 0, level 1, and level 2 IC factorizations. Higher level factorizations contain more nonzeros and are more accurate approximations to the original matrix. The numbers of nonzeros in the upper triangular incomplete factors for these three patterns are 805,826, 1,008,929, and 1,402,741, respectively. The factorizations were used as preconditioners for the PCG algorithm for solving linear systems with the test matrix and a random right-hand side with components uniformly distributed in $[-0.5, 0.5]$. The stopping criterion threshold was 10^{-6}.

Figure 3 plots the solver iteration counts for the three cases (levels 0, 1, and 2) as a function of the number of threads used to construct the factorizations. We test the factorization after 1, 2, and 3 fixed-point iteration sweeps. When a single thread is used, the solver iteration count agrees with the iteration count when a conventional IC factorization is used. (This is because the factorization is computed exactly since GE ordering of the unknowns is used; see the end of section 3.2.) Then, as the number of threads for computing the factorization increases, the solver iteration count also increases. However, the solver iteration count plateaus very quickly; after about 20 or more threads, no additional degradation is observed, and this was verified up to 240 threads. This suggests that it is possible to use a very large number of threads with the new algorithm.

As more sweeps are applied, the solver iteration counts decrease, showing that the L and U factors are indeed converging. A very positive observation is that *even with one or two sweeps, an effective preconditioner is constructed.* For a single sweep, we cannot expect that the L and U factors have converged numerically, but they are accurate enough to be good preconditioners.

We now compare the results for different incomplete factorization levels. Naturally, higher level factorizations lead to lower solver iteration counts. However, the degradation in solver iteration counts as the number of threads is increased is worse for higher level factorizations. In IC(0) after one sweep, the solver iteration count ranges from 297 for one thread to 318 for 60 threads. In IC(2), the iteration count ranges from 126 to 222. These effects, however, are mild and do not preclude the use of higher level factorizations.

For levels 0, 1, and 2, the timings for one sweep using one thread were 0.189, 0.257, and 0.410 seconds, respectively. For 60 threads, the speed-ups over one thread were 42.4, 44.7, and 48.8, respectively. Since the convergence of the new incomplete factorization algorithm is impacted by parallelism, the importance of these timing results is that they confirm that the factorizations were computed in a highly parallel fashion.

Results using 240 threads for the above problem are shown in Table 1, showing the nonlinear residual norm as a function of the number of sweeps. As expected, these norms decrease as the number of sweeps is increased. A more important observation,

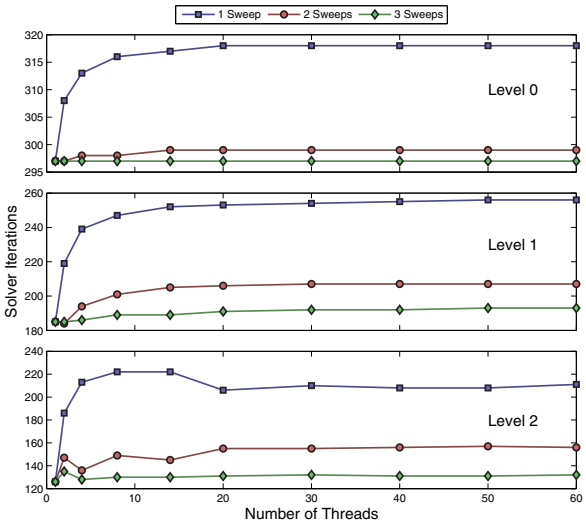

FIG. 3. *PCG solver iteration counts for a 2D finite element Laplacian problem for (top) level 0, (center) level 1, and (bottom) level 2 factorizations. In the unpreconditioned case, 1223 iterations are required for convergence. For the initial guess L and U factors (0 sweeps), 404 iterations are required for convergence.*

TABLE 1

Nonlinear and ILU residual norms for a 2D finite element Laplacian problem, using 240 threads. Zero sweeps denotes using the initial guess, and IC denotes the exact incomplete Cholesky factorization.

	Level 0			Level 1			Level 2		
Sweeps	PCG iter	nonlin resid	ILU resid	PCG iter	nonlin resid	ILU resid	PCG iter	nonlin resid	ILU resid
0	404	1.7e+04	41.1350	404	2.3e+04	41.1350	404	2.3e+04	41.1350
1	318	3.8e+03	32.7491	256	5.7e+03	18.7110	206	7.0e+03	17.3239
2	301	9.7e+02	32.1707	207	8.6e+02	12.4703	158	1.5e+03	6.7618
3	298	1.7e+02	32.1117	193	1.8e+02	12.3845	132	4.8e+02	5.8985
4	297	2.8e+01	32.1524	187	4.6e+01	12.4139	127	1.6e+02	5.8555
5	297	4.4e+00	32.1613	186	1.4e+01	12.4230	126	6.5e+01	5.8706
IC	297	0	32.1629	185	0	12.4272	126	0	5.8894

however, is that the nonlinear residuals are large, meaning that the L and U factors have not converged numerically although the solver iteration counts are essentially the same as those of the exact factorization. *This shows that a fully converged factorization is not necessary for preconditioning.* The table also shows the "ILU residual norm," defined as $\|A - LU\|_F$, for the values of L and U after each sweep. For SPD problems, the ILU residual norm is well correlated to the convergence of the preconditioned iterative solver [14]. (For simplicity, we retain the name "ILU residual norm" although the factorization is symmetric.) This measure of convergence, however, is costly to compute, and we use it only for diagnostic purposes. We observe that the ILU residual norm appears to converge very quickly (to about three significant digits), corresponding to little further improvement in PCG iteration count.

4.2. Nonsymmetric, nondiagonally dominant problems. We now test the convergence of the new incomplete factorization algorithm on more challenging matrices: matrices that are far from being diagonally dominant. Consider the finite

difference discretization of the 2D convection-diffusion problem

$$-\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+\beta\left(\frac{\partial e^{xy}u}{\partial x}+\frac{\partial e^{-xy}u}{\partial y}\right)=g$$

on a square domain $[0,1]\times[0,1]$ with Dirichlet boundary conditions. The derivatives are discretized with centered differences. The parameter β parameterizes the strength of the convection terms. As β is increased, the matrix becomes more nonsymmetric.

A regular mesh of size 450×450 (on the interior of the domain) was used, leading to matrices with 202,500 rows and 1,010,700 nonzeros. Large values of β lead to matrices that are not diagonally dominant and that are challenging for iterative methods. Such nondiagonally dominant problems have been used extensively as test problems for iterative methods and preconditioners; see, e.g., [6].

We generated two test matrices—for $\beta = 1500$ and 3000. After symmetrically scaling the matrices such that the diagonal entries are all ones, the average sum of the absolute values of the entries of each row are 2.76 and 4.50, for $\beta = 1500$ and 3000, respectively. Thus the two matrices are far from being diagonally dominant.

These problems can be challenging to solve. Without preconditioning, the two problems required 1211 and 1301 solver iterations for convergence, using GMRES(50), with a relative residual norm stopping tolerance of 10^{-6}. In the natural or RCM ordering, the level 0 ILU factorization produces factors that are unstable; i.e., triangular solves with U are unstable, or, equivalently, $\|U^{-1}\|$ is large. Benzi, Szyld, and Van Duin [6] recommend using minimum degree ordering for these problems. In our tests, the SYMAMD ordering (as implemented in MATLAB) combined with a level 0 ILU preconditioner for the $\beta = 1500$ problem required 690 solver iterations.

The level 1 ILU factorization with RCM ordering for these problems, however, produces stable factors, and solver convergence required fewer than 70 iterations for the $\beta = 3000$ problem. Thus we tested level 1 ILU factorizations and used RCM orderings for these nondiagonally dominant matrices. The level 1 sparsity patterns have 808,201 nonzeros for each of the L and U factors.

Table 2 shows results for the convection-diffusion problem with $\beta = 1500$. The standard initial guess was used for generating the factorizations. From the solver iteration counts, we observe that even for this nondiagonally dominant problem, the number of sweeps required to produce a useful preconditioner is very small. After a single sweep, a factorization is produced that happens to be a better preconditioner than that produced by the conventional ILU algorithm. (This is due to the fact that the incomplete factorization for a given sparsity pattern is not guaranteed to be the optimal preconditioner with that pattern.) After three sweeps, for all numbers of threads, the factorization is equivalent to the conventional factorization in terms of solver iteration counts.

Table 2 also shows the corresponding nonlinear residual norms (expression (4.1) computed after each sweep). We observe that the nonlinear residuals decrease with increasing numbers of sweeps, and they tend to be larger for higher thread counts. Again, the nonlinear residuals are large, although the solver iteration counts are comparable to those of the exact factorization.

To further understand the convergence of the new ILU algorithm, we consider the effect of increasing the average size of the off-diagonal entries in the matrix. We use the convection-diffusion matrix with $\beta = 3000$ for this purpose, with results shown in Table 3. The new ILU algorithm now appears to converge more slowly; a single sweep no longer produces as good a preconditioner. For a single sweep, the degradation with

TABLE 2
Solver iteration counts and nonlinear residuals (in units of 10^3) for the convection-diffusion problem with $\beta = 1500$. Results shown are averaged over three trials.

	Solver iterations			Nonlinear residual norm		
No. of	Number of sweeps			Number of sweeps		
threads	1	2	3	1	2	3
1	30.0	30.0	30.0	1.3e-14	9.6e-15	9.6e-15
2	23.3	30.0	30.0	5.70	0.184	0.0035
4	22.3	30.0	30.0	10.48	0.746	0.0388
8	23.7	30.7	30.0	12.43	0.966	0.0821
14	24.0	31.0	30.0	13.03	1.114	0.1047
20	24.0	31.0	30.0	13.31	1.159	0.1110
30	24.0	31.0	30.0	13.32	1.161	0.1094
40	24.0	31.0	30.0	13.34	1.166	0.1137
50	24.0	31.0	30.0	13.45	1.178	0.1147
60	24.0	31.0	30.0	13.41	1.172	0.1145

number of threads is also very pronounced. Thus the convergence of the new ILU factorization is affected by the degree of diagonal dominance in the matrix. Overall, however, the new ILU algorithm is still able to compute a good preconditioner after a very small number of sweeps, even for this very nondiagonally dominant problem. Table 4 shows the nonlinear residual for the $\beta = 3000$ problem. We observe that the nonlinear residual decreases monotonically with additional sweeps, although the solver iteration counts for this problem do not.

We note that the initial guesses (corresponding to zero sweeps) for these two nonsymmetric problems are unstable and cause GMRES(50) to diverge.

TABLE 3
Solver iteration counts for the convection-diffusion problem with $\beta = 3000$. Results shown are averaged over three trials.

No. of	Number of sweeps				
threads	1	2	3	4	5
1	66.0	67.0	67.0	67.0	67.0
2	87.7	73.7	67.3	66.7	67.0
4	156.7	61.0	70.0	66.3	67.7
8	190.0	54.3	77.0	62.3	67.0
14	194.3	49.7	80.7	58.7	66.7
20	195.0	57.0	81.7	57.7	67.0
30	195.3	50.0	82.0	60.3	67.0
40	196.0	54.0	82.3	56.3	67.7
50	195.0	50.0	82.3	56.3	66.7
60	195.0	50.0	82.0	56.3	66.7

4.3. Results for general SPD problems. In this section, we test seven SPD matrices from the University of Florida Sparse Matrix collection (Table 5). These are the same seven SPD problems tested in [37] to evaluate the NVIDIA GPU implementation of level scheduled IC factorization. We used the level 0 sparsity pattern and ran the new IC algorithm for up to three sweeps. The nonlinear residual norms and solver iteration counts are presented in Table 6. Also presented are solver iteration counts with the exact IC decompositions. All results were averaged over three runs, and 240 threads were used. The results show that in all cases, the solver iteration counts after three sweeps of the new algorithm are very nearly the same as those for the exact IC factorization. As before, a good preconditioner was computed without

TABLE 4
Nonlinear residual norms (in units of 10^5) for the convection-diffusion problem with $\beta = 3000$. Results shown are averaged over three trials.

No. of threads	Number of sweeps 1	2	3	4	5
1	3.8e-16	1.8e-16	1.8e-16	1.8e-16	1.8e-16
2	0.6674	0.0613	0.0075	0.0020	0.0003
4	1.1526	0.2083	0.0470	0.0127	0.0033
8	1.3768	0.2838	0.0831	0.0264	0.0088
14	1.4594	0.3108	0.1007	0.0367	0.0115
20	1.4580	0.3191	0.1083	0.0386	0.0142
30	1.4978	0.3351	0.1137	0.0388	0.0139
40	1.4906	0.3349	0.1109	0.0392	0.0136
50	1.5014	0.3358	0.1128	0.0399	0.0138
60	1.5014	0.3382	0.1148	0.0401	0.0140

needing to fully converge the nonlinear residual.

TABLE 5
General SPD test matrices.

Matrix	No. equations	No. nonzeros
af_shell3	504855	17562051
thermal2	1228045	8580313
ecology2	999999	4995991
apache2	715176	4817870
G3_circuit	1585478	7660826
offshore	259789	4242673
parabolic_fem	525825	3674625

4.4. Variation of convergence with problem size. To study the convergence of the nonlinear iterations with problem size, we use a set of 7-point finite difference Laplacian matrices with different mesh sizes. Figure 4 plots the nonlinear residual norms (relative to the initial norm) for 1–7 sweeps of the asynchronous fixed point iteration with 240 threads. We observe that, anti-intuitively, convergence is *better* for larger problem sizes. This is likely due to a higher fraction of unknowns being updated simultaneously for small problems, resulting in the asynchronous method being closer in character to a Jacobi-type fixed-point method, and thus evincing poorer convergence behavior for small problems. For large problems, there is little variation with problem size.

4.5. Timing comparison with level scheduled ILU. Figure 5 compares the timings for the new ILU algorithm with those for the standard level scheduled ILU algorithm. The test case is a 5-point finite-difference matrix on a 100×100 grid in the natural ordering. The ILU(2) factorization is computed. Both algorithms assume that the matrix is nonsymmetric. For the level scheduled ILU algorithm, the timings do not include the time for constructing the levels.

The level scheduled ILU algorithm scales well when there are large amounts of parallelism. This example is chosen to emphasize how well the new ILU algorithm may perform when parallelism is limited, which is equivalent to having a very large number of cores available on future machines. For large problems with more parallelism, the level scheduled ILU algorithm may be faster than the new incomplete factorization algorithm, including for large thread counts.

TABLE 6

PCG iterations and nonlinear residuals for three sweeps with 240 *threads of the general SPD test matrices. Results shown are averaged over three trials.*

	Sweeps	Nonlin. resid.	PCG iter
af_shell3	0	1.58+05	852.0
	1	1.66+04	798.3
	2	2.17+03	701.0
	3	4.67+02	687.3
	IC	0	685.0
thermal2	0	1.13+05	1876.0
	1	2.75+04	1422.3
	2	1.74+03	1314.7
	3	8.03+01	1308.0
	IC	0	1308.0
ecology2	0	5.55+04	2000+
	1	1.55+04	1776.3
	2	9.46+02	1711.0
	3	5.55+01	1707.0
	IC	0	1706.0
apache2	0	5.13+04	1409.0
	1	3.66+04	1281.3
	2	1.08+04	923.3
	3	1.47+03	873.0
	IC	0	869.0
G3_circuit	0	1.06+05	1048.0
	1	4.39+04	981.0
	2	2.17+03	869.3
	3	1.43+02	871.7
	IC	0	871.0
offshore	0	3.23+04	401.0
	1	4.37+03	349.0
	2	2.48+02	299.0
	3	1.46+01	297.0
	IC	0	297.0
parabolic_fem	0	5.84+04	790.0
	1	1.61+04	495.3
	2	2.46+03	426.3
	3	2.28+02	405.7
	IC	0	405.0

For our example, the level scheduled algorithm is faster for small numbers of threads, partly because the new ILU algorithm performs more work (three sweeps) than the level scheduled algorithm. However, as the number of threads increases, the level scheduled algorithm stops improving, while the new ILU algorithm continues to scale. Performance is bound by the available instruction-level parallelism in the new ILU algorithm.

5. Approximate triangular solves. This paper has focused on the parallel computation of the incomplete factorization. For completeness, we briefly discuss the parallel solution of sparse triangular systems, which are needed if the factorization is used as a preconditioner. On highly parallel systems, the time for sparse triangular solves dominates the overall solve time, including for matrix-vector multiplies and for constructing the preconditioner. This points to the importance of further research on parallelizing sparse triangular solves, as well as on constructing more accurate but sparser incomplete factors (even if the factorization time must increase), thereby reducing the number of sparse triangular solves and the cost of individual triangular

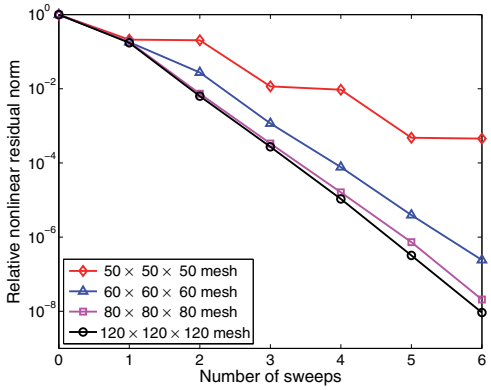

FIG. 4. *Nonlinear residual norm for three-dimensional Laplacian problems of different sizes. Convergence is better for larger problems.*

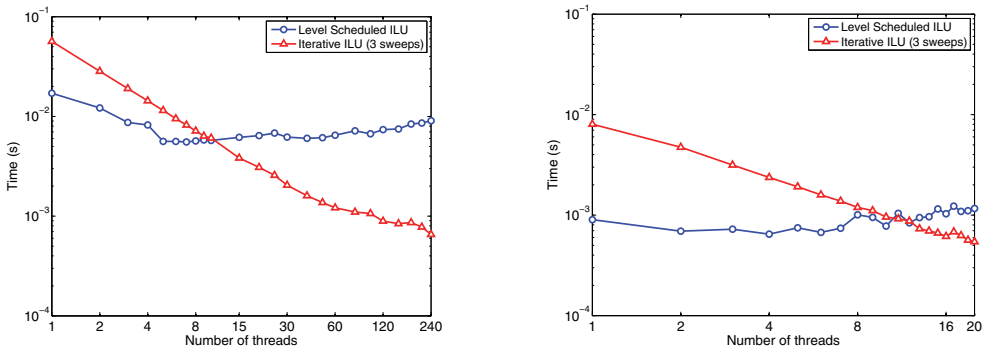

FIG. 5. *Execution time (seconds) comparison of the new ILU algorithm (three sweeps) with the level scheduled ILU algorithm. The test case is a five-point finite difference matrix on a* 100×100 *grid, and the ILU*(2) *factorization is computed. Left: Timings for Intel Xeon Phi. Right: Timings on a two-socket Intel Xeon E5-2680 v2 system (2.8 GHz) with a total of* 20 *cores.*

solves.

A large literature already exists on parallel sparse triangular solves. Among the many options, the most common methods are those based on level scheduling (already mentioned above), where sets of variables are found that can be solved simultaneously. This technique dates from at least 1989 (e.g., [3, 46, 20]), and recent research focuses on efficient implementations (e.g., [33, 54, 36]).

Another parallel method for solving with a sparse triangular matrix R is to represent the inverse of R as the *product* of sparse triangular factors, called a partitioned inverse [42, 2]. Solves with R now only involve sparse matrix-vector products with the factors.

This method is related to a large number of other techniques based on *approximating* the inverse of R. An approximation is acceptable because it is used for preconditioning, which is already approximate. Sparse approximate inverse techniques have been applied to ILU factors [51], and other approximate techniques have also been applied for inverting ILU factors; see, e.g., [7]. Also, the inverse ILU factors may be approximated via a truncated Neumann series [49, 7].

The idea of using iterative approximate solves, which we applied to computing the

ILU factorization, can be extended to solving sparse triangular systems for preconditioning. Here, we briefly illustrate solving a triangular system $Rx = b$ approximately by a fixed small number of steps of the Jacobi iteration (related to using a truncated Neumann series; other polynomial approximations to the inverse of R may also be used).

$$x_{k+1} = (I - D^{-1}R)x_k + D^{-1}b,$$

where D is the matrix consisting of the diagonal of R. Thus the iteration matrix $G = I - D^{-1}R$ is strictly triangular (the diagonal is all zeros) and has spectral radius 0, providing trivial asymptotic convergence, just like in the new parallel ILU algorithm. Convergence within a small number of iterations is desired, however, and this depends on the norm or nonnormality of G and its pseudospectra [47]. Clearly, the nonnormality of G may be arbitrarily bad, but for triangular matrices R arising from *stable* incomplete factorizations of physically based problems, R is often close to being diagonally dominant, making the norm of G small.

TABLE 7

Solver iteration counts and timings; comparison of level scheduled triangular solves and iterative triangular solves. The last column shows the number of Jacobi steps used to approximately solve with the triangular factors in the iterative triangular solves.

	IC	PCG iterations		Timing (seconds)		Num.
	level	LevSch	Iterative	LevSch	Iterative	steps
af_shell3	1	375	592	79.59	23.05	6
thermal2	0	1860	2540	120.06	48.13	1
ecology2	1	1042	1395	114.58	34.20	4
apache2	0	653	742	24.68	12.98	3
G3_circuit	1	329	627	52.30	32.98	5
offshore	0	341	401	42.70	9.62	5
parabolic_fem	0	984	1201	15.74	16.46	1

Table 7 shows results from this approach for the seven SPD problems used earlier. PCG iteration counts and solver iteration time are reported for parallel triangular solves using level scheduling (denoted "LevSch") and for iterative triangular solves (denoted "Iterative"). Either a level 0 or a level 1 IC factorization was used, depending on which gave better performance, and 60 threads were used. We observe that although the number of solver iterations is higher for the iterative approach, the time can be significantly lower. The advantage of using iterative triangular solves is greater when more threads are used. In these results, a fixed number of synchronous Jacobi iterations were used, from 1 to 6. Since a fixed number is used, the preconditioning operation is a fixed linear operator, and solvers that can accommodate a changing preconditioner are not required. A flexible solver is required if asynchronous triangular solves are used; see, e.g., [44].

Our examples in this section have all been SPD. We point out that diagonally scaled PCG can converge after a very large number of iterations, and this is because an approximate solution must eventually be found in the space searched by the PCG method. Depending on the number of iterations, diagonally scaled PCG may be faster than IC preconditioning in highly parallel environments. For nonsymmetric problems, however, short-recurrence iterative methods do not exist, meaning that all methods must either store a vector for every iteration and then restart when storage is exhausted (e.g., restarted GMRES) or be nonoptimal (e.g., BiCGStab). Thus diagonally scaled restarted GMRES or BiCGStab may never converge. In this case, a better preconditioner such as ILU is absolutely necessary for convergence.

6. Conclusions and future work. This paper presented a new parallel algorithm for computing an incomplete factorization. Individual matrix entries in the factorization can be computed in parallel. The algorithm is based on a reformulation of ILU as the solution of a set of bilinear equations which can be solved using fine-grained asynchronous parallelism. The nonlinear equations are solved using fixed-point iteration sweeps that are performed in parallel. Very few sweeps are needed to construct an incomplete factorization that is an effective preconditioner. The new parallel ILU algorithm is very different from existing approaches for parallelizing an ILU factorization. Many existing approaches require reordering the matrix to enhance the available parallelism. In the new approach, the amount of parallelism is large for any ordering of the matrix, including for orderings that enhance convergence of the solver.

Future work includes (1) developing variants that dynamically update the sparsity pattern during the nonlinear iterations, to obtain a method akin to threshold-based ILU, (2) applying the new algorithm to time-dependent and other problems involving a sequence of linear systems, where the ILU factorization from a previous time step is a natural initial guess for the factorization at the current time step, (3) examining the effect on convergence of how variables and equations are assigned to threads, and (4) extending the approach presented here to other sparse approximate factorizations, such as factorized sparse approximate inverses.

Appendix A. Proof of Theorem 3.8. Each column of the Jacobian is associated with a nonzero $(k,l) \in S$. Let s_{kl} denote the absolute sum of the column associated with nonzero (k,l). The values of the variables at which the Jacobian is evaluated are denoted by subscripted l and u.

A column with $k > l$ contains elements corresponding to partial derivatives with respect to l_{kl}:

$$\begin{aligned} &\frac{\partial G_{g(i,j)}}{\partial l_{kl}} = -\frac{u_{lj}}{u_{jj}} \quad \text{if } k = i, \quad l < j, \quad j < i, \quad (i,j) \in S, \quad (l,j) \in S, \\ &\frac{\partial G_{g(i,j)}}{\partial l_{kl}} = -u_{lj} \quad \text{if } k = i, \quad l < i, \quad i \le j, \quad (i,j) \in S, \quad (l,j) \in S. \end{aligned} \tag{A.1}$$

A column with $k < l$ contains elements corresponding to partial derivatives with respect to u_{kl}:

$$\begin{aligned} &\frac{\partial G_{g(i,j)}}{\partial u_{kl}} = -\frac{l_{ik}}{u_{jj}} \quad \text{if } l = j, \quad k < j, \quad j < i, \quad (i,j) \in S, \quad (i,k) \in S, \\ &\frac{\partial G_{g(i,j)}}{\partial u_{kl}} = -l_{ik} \quad \text{if } l = j, \quad k < i, \quad i \le j, \quad (i,j) \in S, \quad (i,k) \in S. \end{aligned} \tag{A.2}$$

A column with $k = l$ contains elements corresponding to partial derivatives with respect to u_{kk}:

$$\frac{\partial G_{g(i,j)}}{\partial u_{kk}} = -\frac{1}{u_{kk}^2} \left(a_{ij} - \sum_{p=1}^{j-1} l_{ip} u_{pj} \right) \quad \text{if } k = j, \quad j < i. \tag{A.3}$$

We now state and prove three lemmas, one for each of the above three cases, $k > l$, $k < l$, and $k = l$. Below we assume that $u_{jj} = 1 \; \forall 1 \le j \le n$.

LEMMA A.1. *If $k > l$, then $s_{kl} = \sum_{\substack{j=l+1 \\ (k,j) \in S}}^{n} |u_{lj}|$.*

Proof. Let $s_{kl} = \sum_{j=1}^{n} \sum_{i=1}^{n} c_{ijkl}$, where

$$c_{ijkl} = \begin{cases} |u_{lj}| & \text{for } k = i, \quad l < j, \quad j < i, \quad (i,j) \in S, \\ |u_{lj}| & \text{for } k = i, \quad l < i, \quad i \le j, \quad (i,j) \in S, \\ 0 & \text{otherwise.} \end{cases}$$

These expressions are from (A.1). Using indicator functions, we can express c_{ijkl} as

$$\begin{aligned} c_{ijkl} \;=\; & I[k = i]\, I[l < j]\, I[j < i]\, I[(i,j) \in S]\, |u_{lj}| \\ & + I[k = i]\, I[l < i]\, I[i \le j]\, I[(i,j) \in S]\, |u_{lj}|. \end{aligned}$$

Define

$$\begin{aligned} s_{kl}^1 &= \sum_{j=1}^{n} \sum_{i=1}^{n} I[k = i]\, I[l < j]\, I[j < i] I[(i,j) \in S]\, |u_{lj}| \\ &= \sum_{j=1}^{n} I[l < j]\, I[j < k]\, I[(k,j) \in S]\, |u_{lj}| \\ &= \sum_{\substack{j=l+1 \\ (k,j) \in S}}^{k-1} |u_{lj}|, \qquad \text{(A.4)} \\ s_{kl}^2 &= \sum_{j=1}^{n} \sum_{i=1}^{n} I[k = i]\, I[l < i]\, I[i \le j]\, I[(i,j) \in S]\, |u_{lj}| \\ &= \sum_{j=1}^{n} I[l < k]\, I[k \le j]\, I[(k,j) \in S]\, |u_{lj}| \\ &= \sum_{\substack{j=k \\ (k,j) \in S}}^{n} |u_{lj}|. \qquad \text{(A.5)} \end{aligned}$$

From (A.4) and (A.5) we have

$$s_{kl} = s_{kl}^1 + s_{kl}^2 = \sum_{\substack{j=l+1 \\ (k,j) \in S}}^{k-1} |u_{lj}| + \sum_{\substack{j=k \\ (k,j) \in S}}^{n} |u_{lj}| = \sum_{\substack{j=l+1 \\ (k,j) \in S}}^{n} |u_{lj}|. \qquad \square$$

LEMMA A.2. *If $k < l$, then $s_{kl} = \sum_{\substack{i=k+1 \\ (i,l) \in S}}^{n} |l_{ik}|$.*

The proof of this lemma is analogous to that of Lemma A.1.

LEMMA A.3. *If $k = l$, then $s_{kk} = \sum_{i=k+1}^{n} |a_{ik} - t_{ik}|$, where $t_{ik} = \sum_{p=1}^{k-1} l_{ip} u_{pk}$.*

Proof. Let $s_{kk} = \sum_{j=1}^{n} \sum_{i=1}^{n} c_{ijk}$, where

$$c_{ijk} = \begin{cases} |a_{ij} - t_{ij}| & \text{for } k = j, \quad j < i, \quad (i,j) \in S, \\ 0 & \text{otherwise,} \end{cases}$$

which is from (A.3). Now, using indicator functions,

$$\begin{aligned} s_{kk} &= \sum_{i=1}^{n} \sum_{j=1}^{n} I[k=j]\; I[j<i]\; I[(i,j) \in S]\; |a_{ij} - t_{ij}| \\ &= \sum_{i=1}^{n} I[k<i]\; I[(i,k) \in S]\; |a_{ik} - t_{ik}| \\ &= \sum_{i=k+1}^{n} |a_{ik} - t_{ik}|. \end{aligned}$$ ∎

Proof of Theorem 3.8.

$$\begin{aligned} \|G'(\tilde{L},\tilde{U})\|_1 &= \max_{(k,l)\in S} (s_{kl}) \\ &= \max\left(\max_{k>l}(s_{kl}),\; \max_{k<l}(s_{kl}),\; \max_{k}(s_{kk}) \right). \end{aligned}$$

For $k > l$, applying Lemma A.1,

$$\begin{aligned} s_{kl} &= \sum_{\substack{j=l+1 \\ (k,j)\in S}}^{n} |u_{lj}| \le \sum_{j=l+1}^{n} |u_{lj}|, \\ \max_{k>l}(s_{kl}) &\le \|\tilde{U}\|_\infty. \end{aligned}$$

For $k < l$, applying Lemma A.2,

$$\begin{aligned} s_{kl} &= \sum_{\substack{i=k+1 \\ (i,l)\in S}}^{n} |l_{ik}| \le \sum_{j=k+1}^{n} |l_{ik}|, \\ \max_{k<l}(s_{kl}) &\le \|\tilde{L}\|_1. \end{aligned}$$

For $k = l$, applying Lemma A.3,

$$\begin{aligned} s_{kk} &= \sum_{i=k+1}^{n} |a_{ik} - t_{ik}|, \\ \max_{k}(s_{kk}) &= \|\tilde{R}_{L^*}\|_1, \end{aligned}$$

where $\tilde{R}_{L^*}$ is the strictly lower triangular part of $A - \tilde{T}$. Combining the above,

$$\|G'(\tilde{L},\tilde{U})\|_1 = \max_{(k,l)\in S}(s_{kl}) \le \max(\|\tilde{U}\|_\infty,\; \|\tilde{L}\|_1,\; \|\tilde{R}_{L^*}\|_1).$$ ∎

Appendix B. Proof of Theorem 3.9.

Proof. We will first show that for the 5-point matrix on a 2D grid, the expression for s_{kl} is very simple:

$$s_{kl} = \begin{cases} |u_{lk}| & \text{for } k > l, \\ |l_{lk}| & \text{for } k < l, \\ \sum_{i=k+1}^{n} |a_{ik}| & \text{for } k = l. \end{cases}$$

Consider the first case, $k > l$, and Lemma A.1,

$$s_{kl} = \sum_{\substack{j=l+1 \\ (k,j)\in S}}^{n} |u_{lj}|.$$

The terms in the sum are in row l of $\tilde{U}$. The terms in the sum that survive the condition $(k, j) \in S$ correspond to nonzeros in column k of $\tilde{U}$. In the directed graph of $\tilde{U}$, vertex l and vertex k share an edge because $(k, l) \in S$. (This graph is the obvious subgraph of the 2D grid.) The 5-point stencils at vertex k and vertex l do not intersect, except for the edge (k, l). Therefore, the only term in the sum is $|u_{lk}|$, as promised. Similarly, using Lemma A.2, $s_{kl} = |l_{lk}|$ for $k < l$.

To prove the case $k = l$, consider Lemma A.3,

$$s_{kk} = \sum_{i=k+1}^{n} |a_{ik} - t_{ik}|, \quad \text{where} \quad t_{ik} = \sum_{p=1}^{k-1} l_{ip} u_{pk},$$

particularly the summation for t_{ik}. The sparsity patterns of row i of $\tilde{L}$ and column k of $\tilde{U}$ do not intersect except for edge (i, k) when vertex i is a neighbor of vertex k in the graph of the matrix A. Ignoring the limits of the summation, the only nonzero term in the sum is $l_{ik}u_{kk}$. However, this term is outside the upper limit of the summation. Therefore, $t_{ik} = 0$, and the case $k = l$ is proven.

The above results obviously also hold true for the 7-point stencil on a three-dimensional grid.

The 1-norm of the Jacobian is the maximum among

$$\begin{aligned}
\max_{k>l}(s_{kl}) &= \max_{k>l}(|u_{lk}|), \\
\max_{k<l}(s_{kl}) &= \max_{k<l}(|l_{lk}|), \\
\max_{k}(s_{kk}) &= \max_{k} \sum_{i=k+1}^{n} |a_{ik}| = \|A_{L^*}\|_1,
\end{aligned}$$

where A_{L^*} denotes the strictly lower triangular part of A. Therefore,

$$\|G'(\tilde{L}, \tilde{U})\|_1 = \max(\|\tilde{L}\|_{\max},\ \|\tilde{U}\|_{\max},\ \|A_{L^*}\|_1). \qquad \square$$

Acknowledgment. The authors thank the two anonymous referees for comments and corrections that helped improve the presentation of this paper.

REFERENCES

[1] J. I. Aliaga, M. Bollhöfer, A. F. Martín, and E. S. Quintana-Ortí, *Exploiting thread-level parallelism in the iterative solution of sparse linear systems*, Parallel Comput., 37 (2011), pp. 183–202.

[2] F. L. Alvarado and R. Schreiber, *Optimal parallel solution of sparse triangular systems*, SIAM J. Sci. Comput., 14 (1993), pp. 446–460.

[3] E. C. Anderson and Y. Saad, *Solving sparse triangular systems on parallel computers*, Intl. J. High Speed Comput., 1 (1989), pp. 73–96.

[4] G. M. Baudet, *Asynchronous iterative methods for multiprocessors*, J. ACM, 25 (1978), pp. 226–244.

[5] M. Benzi, W. D. Joubert, and G. Mateescu, *Numerical experiments with parallel orderings for ILU preconditioners*, Electron. Trans. Numer. Anal., 8 (1999), pp. 88–114.
[6] M. Benzi, D. B. Szyld, and A. van Duin, *Orderings for incomplete factorization preconditioning of nonsymmetric problems*, SIAM J. Sci. Comput., 20 (1999), pp. 1652–1670.
[7] M. Benzi and M. Tůma, *A comparative study of sparse approximate inverse preconditioners*, Appl. Numer. Math., 30 (1999), pp. 305–340.
[8] D. P. Bertsekas, *Distributed asynchronous computation of fixed points*, Math. Programming, 27 (1983), pp. 107–120.
[9] N. I. Buleev, *A numerical method for the solution of two-dimensional and three-dimensional equations of diffusion*, Mat. Sb., 51 (1960), pp. 227–238; English transl.: Rep. BNL-TR-551, Brookhaven National Laboratory, Upton, New York, 1973.
[10] E. Chow and Y. Saad, *Experimental study of ILU preconditioners for indefinite matrices*, J. Comput. Appl. Math., 86 (1997), pp. 387–414.
[11] S. Doi, *On parallelism and convergence of incomplete LU factorizations*, Appl. Numer. Math., 7 (1991), pp. 417–436.
[12] S. Doi and T. Washio, *Ordering strategies and related techniques to overcome the trade-off between parallelism and convergence in incomplete factorizations*, Parallel Comput., 25 (1999), pp. 1995–2014.
[13] X. Dong and G. Cooperman, *A bit-compatible parallelization for ILU(k) preconditioning*, in Proceedings of Euro-Par 2011, Parallel Processing (Part 2), Springer-Verlag, Berlin, 2011, pp. 66–77.
[14] I. S. Duff and G. A. Meurant, *The effect of ordering on preconditioned conjugate gradients*, BIT, 29 (1989), pp. 635–657.
[15] H. C. Elman and E. Agrón, *Ordering techniques for the preconditioned conjugate-gradient method on parallel computers*, Comput. Phys. Comm., 53 (1989), pp. 253–269.
[16] A. Frommer and D. B. Szyld, *On asynchronous iterations*, J. Comput. Appl. Math., 123 (2000), pp. 201–216.
[17] F. Gibou and C. Min, *On the performance of a simple parallel implementation of the ILU-PCG for the Poisson equation on irregular domains*, J. Comput. Phys., 231 (2012), pp. 4531–4536.
[18] P. González, J. C. Cabaleiro, and T. F. Pena, *Parallel incomplete LU factorization as a preconditioner for Krylov subspace methods*, Parallel Process. Lett., 9 (1999), pp. 467–474.
[19] M. J. Grote and T. Huckle, *Parallel preconditioning with sparse approximate inverses*, SIAM J. Sci. Comput., 18 (1997), pp. 838–853.
[20] S. W. Hammond and R. Schreiber, *Efficient ICCG on a shared memory multiprocessor*, Intl. J. High Speed Comput., 4 (1992), pp. 1–21.
[21] M. A. Heroux, P. Vu, and C. Yang, *A parallel preconditioned conjugate-gradient package for solving sparse linear-systems on a Cray Y-MP*, Appl. Numer. Math., 8 (1991), pp. 93–115.
[22] V. Heuveline, D. Lukarski, and J.-P. Weiss, *Enhanced Parallel ILU(p)-Based Preconditioners for Multi-Core CpuS and GpuS—the Power(Q)-Pattern Method*, Tech. Report EMCL-2011-08, Karlsruhe Institute of Technology, Karlsruhe, Germany, 2011.
[23] D. Hysom and A. Pothen, *Efficient parallel computation of ILU(k) preconditioners*, in Proceedings of the 1999 ACM/IEEE Conference on Supercomputing, ACM, New York, IEEE, Washington, DC, 1999, 29.
[24] D. Hysom and A. Pothen, *A scalable parallel algorithm for incomplete factor preconditioning*, SIAM J. Sci. Comput., 22 (2001), pp. 2194–2215.
[25] D. Hysom and A. Pothen, *Level-Based Incomplete LU Factorization: Graph Model and Algorithms*, Tech. Report UCRL-JC-159789, Lawrence Livermore National Laboratory, Livermore, CA, 2002.
[26] M. T. Jones and P. E. Plassmann, *Scalable iterative solution of sparse linear-systems*, Parallel Comput., 20 (1994), pp. 753–773.
[27] W. Joubert and T. Oppe, *Improved SSOR and incomplete Cholesky solution of linear equations on shared memory and distributed memory parallel computers*, Numer. Linear Algebra Appl., 1 (1994), pp. 287–311.
[28] G. Karypis and V. Kumar, *Parallel threshold-based ILU factorization*, in Proceedings of the 1997 ACM/IEEE Conference on Supercomputing, ACM, New York, IEEE, Washington, DC, 1997.
[29] L. Yu. Kolotilina and A. Yu. Yeremin, *Factorized sparse approximate inverse preconditionings* I. *Theory*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 45–58.
[30] R. Li and Y. Saad, *GPU-Accelerated Preconditioned Iterative Linear Solvers*, Tech. Report UMSI-2010-112, University of Minnesota Supercomputing Institute, Minneapolis, MN, 2010.

[31] S. Ma and Y. Saad, *Distributed ILU(0) and SOR Preconditioners for Unstructured Sparse Linear Systems*, Tech. Report 94-027, Army High Performance Computing Research Center, Minneapolis, MN, 1994.

[32] M. Magolu Monga Made and H. A. van der Vorst, *Spectral analysis of parallel incomplete factorizations with implicit pseudo-overlap*, Numer. Linear Algebra Appl., 9 (2002), pp. 45–64.

[33] J. Mayer, *Parallel algorithms for solving linear systems with sparse triangular matrices*, Computing, 86 (2009), pp. 291–312.

[34] J. A. Meijerink and H. A. Van der Vorst, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Math. Comp., 31 (1977), pp. 148–162.

[35] K. Nakajima, *Parallel iterative solvers of GeoFEM with selective blocking preconditioning for nonlinear contact problems on the Earth Simulator*, in Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, ACM, New York, IEEE, Washington, DC, 2003.

[36] M. Naumov, *Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU*, Tech. Report NVR-2011-001, NVIDIA, Santa Clara, CA, 2011.

[37] M. Naumov, *Parallel Incomplete-LU and Cholesky Factorization in the Preconditioned Iterative Methods on the GPU*, Tech. Report NVR-2012-003, NVIDIA, Santa Clara, CA, 2012.

[38] T. A. Oliphant, *An implicit numerical method for solving two-dimensional time-dependent diffusion problems*, Quart. Appl. Math., 19 (1961), pp. 221–229.

[39] J. Ortega and W. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press, New York, 1970.

[40] M. Pakzad, J. L. Lloyd, and C. Phillips, *Independent columns: A new parallel ILU preconditioner for the PCG method*, Parallel Comput., 23 (1997), pp. 637–647.

[41] E. L. Poole and J. M. Ortega, *Multicolor ICCG methods for vector computers*, SIAM J. Numer. Anal., 24 (1987), pp. 1394–1417.

[42] A. Pothen and F. L. Alvarado, *A fast reordering algorithm for parallel sparse triangular solution*, SIAM J. Sci. Statist. Comput., 13 (1992), pp. 645–653.

[43] P. Raghavan and K. Teranishi, *Parallel hybrid preconditioning: Incomplete factorization with selective sparse approximate inversion*, SIAM J. Sci. Comput., 32 (2010), pp. 1323–1345.

[44] Y. Saad, *A flexible inner-outer preconditioned GMRES algorithm*, SIAM J. Sci. Comput., 14 (1993), pp. 461–469.

[45] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed., SIAM, Philadelphia, 2003.

[46] J. H. Saltz, *Aggregation methods for solving sparse triangular systems on multiprocessors*, SIAM J. Sci. Statist. Comput., 11 (1990), pp. 123–144.

[47] L. N. Trefethen and M. Embree, *Spectra and Pseudospectra: The Behavior of Nonnormal Matrices and Operators*, Princeton University Press, Princeton, NJ, 2005.

[48] A. Üresin and M. Dubois, *Sufficient conditions for the convergence of asynchronous iterations*, Parallel Comput., 10 (1989), pp. 83–92.

[49] H. A. van der Vorst, *A vectorizable variant of some ICCG methods*, SIAM J. Sci. Statist. Comput., 3 (1982), pp. 350–356.

[50] H. A. van der Vorst, *High performance preconditioning*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 1174–1185.

[51] A. C. N. van Duin, *Scalable parallel preconditioning with the sparse approximate inverse of triangular matrices*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 987–1006.

[52] R. S. Varga, *Factorization and normalized iterative methods*, in Boundary Problems in Differential Equations (Santa Barbara, CA, 1960), R. E. Langer, ed., University of Wisconsin Press, Madison, WI, 1960, pp. 121–142.

[53] C. Vuik, R. R. P. van Nooyen, and P. Wesseling, *Parallelism in ILU-preconditioned GMRES*, Parallel Comput., 24 (1998), pp. 1927–1946.

[54] M. M. Wolf, M. A. Heroux, and E. G. Boman, *Factors impacting performance of multithreaded sparse triangular solve*, in High Performance Computing for Computational Science—VECPAR 2010, Lecture Notes in Comput. Sci. 6449, Springer, Berlin, 2011, pp. 32–44.