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.

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.

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

∗ 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-0012538, by the National Science Foundation under grant ACI-1306573 and by Intel Corporation.

† School of Computational Science and Engineering, College of Computing, Georgia Institute of Technology, 266 Ferst Drive, Atlanta, GA 30332-0765 (echow@cc.gatech.edu, aypatel@gatech.edu).
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

\[
\text{i} = 2 \text{ to } n \text{ do} \\
\quad \text{k} = 1 \text{ to } i - 1 \text{ and } (i, k) \in S \text{ do} \\
\quad \quad a_{ik} = a_{ik} / a_{kk} \\
\quad \quad \text{for } j = k + 1 \text{ to } n \text{ and } (i, j) \in S \text{ do} \\
\quad \quad \quad a_{ij} = a_{ij} - a_{ik} a_{kj} \\
\quad \text{end} \\
\text{end} \\
\]

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 to 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 1980’s.

The first parallel ILU algorithms were designed for problems on regular grids, where sets of grid points, organized in diagonal lines in 2-D or hyperplanes in 3-D, 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, 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, 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, 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, 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

\[(LU)_{ij} = a_{ij}, \quad (i, j) \in S\]

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].

Nowadays, 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

\[l_{ij}, \quad i > j, \quad (i, j) \in S\]
\[u_{ij}, \quad i \leq j, \quad (i, j) \in S.\]

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

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

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 approx-
imate 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 \(i \leq j\)),

\[
\begin{align}
    l_{ij} &= \frac{1}{u_{jj}} \left( a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) \\
    u_{ij} &= a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}.
\end{align}
\]

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

\[
x^{(p+1)} = G(x^{(p)}), \quad p = 0, 1, \ldots
\]

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 \((i, j)\) only depends
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 equations (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 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 gives 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:

$$
(L^{(0)})_{ij} = (A)_{ij}, \quad i > j, \quad (i, j) \in S
$$
$$
(U^{(0)})_{ij} = (A)_{ij}, \quad i \leq j, \quad (i, j) \in S
$$

(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. for \((i, j) \in S\) do
4. if \( i > j \) then
5. \( l_{ij} = \frac{(a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj})}{u_{jj}} \)
6. else
7. \( u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj} \)
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 is 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 format, CSR) and \( U \) should be stored in column-major order (using compressed sparse column format, CSC). 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
When the matrix $A$ is symmetric positive definite, the algorithm only needs to compute one of the triangular factors. Specifically, an incomplete Cholesky 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 $(i, j) \in S_U$ do
4. $s = a_{ij} - \sum_{k=1}^{j-1} u_{ki} u_{kj}$
5. if $i \neq j$ then
6. $u_{ij} = s / u_{ii}$
7. else
8. $u_{ii} = \sqrt{s}$
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

\begin{equation}
    x^{(p+1)} = G(x^{(p)}), \quad p = 0, 1, \ldots
\end{equation}

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

\begin{equation}
    x^{(i,j)} = \begin{cases} 
    l_{ij} & \text{if } i > j \\
    u_{ij} & \text{if } i \leq j 
    \end{cases}
\end{equation}

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

\begin{equation}
    G_{g(i,j)}(x) = \begin{cases} 
    \frac{1}{x^{g(j,i)}_{g(j,i)}} \left( a_{ij} - \sum_{1 \leq k < j-1}^{(i,k),(j,k) \in S} x_{g(i,k)} x_{g(k,j)} \right) & \text{if } i > j \\
    a_{ij} - \sum_{1 \leq k < i-1}^{(i,k),(j,k) \in S} x_{g(i,k)} x_{g(k,j)} & \text{if } i \leq j 
    \end{cases}
\end{equation}
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
\[
D = \{x \in \mathbb{R}^m \mid x_{g(j,j)} \neq 0, 1 \leq j \leq n\}.
\]

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

For a row where \(i \leq j\) (corresponding to the unknown \(u_{ij}\)), the partial derivatives are
\[
\begin{align*}
\frac{\partial G_{g(i,j)}}{\partial l_{ik}} &= -u_{kj} \quad k < i \\
\frac{\partial G_{g(i,j)}}{\partial u_{kj}} &= -l_{ik} \quad k < i.
\end{align*}
\]

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 \(\rho(G'(x)) = 0\) for all \(x\) 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:
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

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

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
\[ \|G(x) - G(y)\| \leq \alpha \|x - y\| \]
for all \( x, y \in D \); see e.g., [39]. Such a constant can be found if \( D \) is convex and there
is a constant \( \alpha < 1 \) satisfying \( \|G'(x)\| \leq \alpha \) for all \( x \in D \). 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 \( \|G'(x)\| < 1 \) for all \( x \in D \). 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
non-convergence for matrices that are far from diagonally dominant. This motivates
us to test non-diagonally 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 \( \infty \)-norm. In this section, let \( \hat{L} \) denote the strictly lower triangular sparse matrix
with nonzeros corresponding to the unknowns \( l_{ij} \). Let \( \hat{U} \) 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 \( G'((\hat{L}, \hat{U})) \).

**Theorem 3.8.** Given a matrix \( A \) and \( G \) defined in (3.3), the 1-norm of the
Jacobian is bounded as
\[ \|G'((\hat{L}, \hat{U}))\|_1 \leq \max(\|\hat{U}\|_\infty, \|\hat{L}\|_1, \|\hat{R}_{L^*}\|_1) \]
where \( \hat{R}_{L^*} \) is the strictly lower triangular part of \( \hat{R} = A - \hat{T} \), where the matrix \( \hat{T} \) is
\[ (\hat{T})_{ij} = \begin{cases} (\hat{L}\hat{U})_{ij}, & (i, j) \in S \\ 0, & \text{otherwise}. \end{cases} \]

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 \( \hat{L} (\hat{U}) \)
has sparsity pattern equal to the strictly lower (upper) triangular part of \( A \), then for
\( G \) defined in (3.3),
\[ \|G'((\hat{L}, \hat{U}))\|_1 = \max(\|\hat{L}\|_{\max}, \|\hat{U}\|_{\max}, \|A_{L^*}\|_1) \]
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 (2-D or 3-D, 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 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 $10 \times 10$ 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 4-way simultaneous multithreading.

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

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

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

<table>
<thead>
<tr>
<th>Sweeps</th>
<th>Level 0</th>
<th>Level 1</th>
<th>Level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PCG nonlin ILU</td>
<td>PCG nonlin ILU</td>
<td>PCG nonlin ILU</td>
</tr>
<tr>
<td></td>
<td>iter resid resid</td>
<td>iter resid resid</td>
<td>iter resid resid</td>
</tr>
<tr>
<td>0</td>
<td>404 1.7e+04 41.1350</td>
<td>404 2.3e+04 41.1350</td>
<td>404 2.3e+04 41.1350</td>
</tr>
<tr>
<td>1</td>
<td>318 3.8e+04 32.7491</td>
<td>256 5.7e+03 18.7110</td>
<td>206 7.0e+03 17.8239</td>
</tr>
<tr>
<td>2</td>
<td>301 9.7e+02 32.1707</td>
<td>207 8.6e+02 12.4703</td>
<td>158 1.5e+03 6.7618</td>
</tr>
<tr>
<td>3</td>
<td>298 1.7e+02 32.1117</td>
<td>193 1.8e+02 12.3845</td>
<td>132 4.8e+02 5.8985</td>
</tr>
<tr>
<td>4</td>
<td>297 1.7e+02 32.1524</td>
<td>187 4.6e+01 12.4139</td>
<td>127 1.6e+02 5.8355</td>
</tr>
<tr>
<td>5</td>
<td>297 2.8e+01 32.1613</td>
<td>186 1.4e+01 12.4230</td>
<td>126 6.5e+01 5.8706</td>
</tr>
<tr>
<td>IC</td>
<td>297 0 32.1629</td>
<td>185 0 12.4272</td>
<td>126 0 5.8894</td>
</tr>
</tbody>
</table>

For levels 0, 1, and 2, the timings for one sweep using 1 thread were 0.189, 0.257, and 0.410 seconds, respectively. For 60 threads, the speedups over 1 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 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, 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 symmetric positive definite 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 3 significant digits), corresponding to little further improvement in PCG iteration count.

4.2. Nonsymmetric, non-diagonally 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 2-D 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 $\beta$ parameterizes the strength of the convection terms. As $\beta$ 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. The matrix is weakly diagonally dominant until $\beta > 2/h$, where $h$ is the mesh spacing, corresponding to $\beta > 902$ for our problem size. Values of $\beta$ larger than this threshold leads to matrices that are no longer diagonally dominant and that are challenging for iterative methods. Such
non-diagonally dominant problems have been used extensively as test problems for iterative methods and preconditioners, 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 et al. [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 non-diagonally 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 non-diagonally 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 the one 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 3 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.

### 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.

<table>
<thead>
<tr>
<th>No. of Threads</th>
<th>Number of Sweeps</th>
<th>Number of Sweeps</th>
<th>Number of Sweeps</th>
<th>Number of Sweeps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>1.3e-14</td>
</tr>
<tr>
<td>2</td>
<td>23.3</td>
<td>30.0</td>
<td>30.0</td>
<td>5.70</td>
</tr>
<tr>
<td>4</td>
<td>22.3</td>
<td>30.0</td>
<td>30.0</td>
<td>10.48</td>
</tr>
<tr>
<td>8</td>
<td>23.7</td>
<td>30.7</td>
<td>30.0</td>
<td>12.43</td>
</tr>
<tr>
<td>14</td>
<td>24.0</td>
<td>31.0</td>
<td>30.0</td>
<td>13.03</td>
</tr>
<tr>
<td>20</td>
<td>24.0</td>
<td>31.0</td>
<td>30.0</td>
<td>13.31</td>
</tr>
<tr>
<td>30</td>
<td>24.0</td>
<td>31.0</td>
<td>30.0</td>
<td>13.32</td>
</tr>
<tr>
<td>40</td>
<td>24.0</td>
<td>31.0</td>
<td>30.0</td>
<td>13.34</td>
</tr>
<tr>
<td>50</td>
<td>24.0</td>
<td>31.0</td>
<td>30.0</td>
<td>13.45</td>
</tr>
<tr>
<td>60</td>
<td>24.0</td>
<td>31.0</td>
<td>30.0</td>
<td>13.41</td>
</tr>
</tbody>
</table>
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 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 non-diagonally 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 0 sweeps) for these two non-symmetric 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.

<table>
<thead>
<tr>
<th>No. of Threads</th>
<th>Number of Sweeps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>66.0 67.0 67.0 67.0 67.0</td>
</tr>
<tr>
<td>2</td>
<td>87.7 73.7 67.3 66.7 67.0</td>
</tr>
<tr>
<td>4</td>
<td>156.7 61.0 70.0 66.3 67.7</td>
</tr>
<tr>
<td>8</td>
<td>190.0 54.3 77.0 62.3 67.0</td>
</tr>
<tr>
<td>14</td>
<td>194.3 49.7 80.7 58.7 66.7</td>
</tr>
<tr>
<td>20</td>
<td>195.0 57.0 81.7 57.7 67.0</td>
</tr>
<tr>
<td>30</td>
<td>195.3 50.0 82.0 60.3 67.0</td>
</tr>
<tr>
<td>40</td>
<td>196.0 54.0 82.3 56.3 67.7</td>
</tr>
<tr>
<td>50</td>
<td>195.0 50.0 82.3 56.3 66.7</td>
</tr>
<tr>
<td>60</td>
<td>195.0 50.0 82.0 56.3 66.7</td>
</tr>
</tbody>
</table>

### 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.

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

### 4.3. Results for general SPD problems.

In this section, we test 7 symmetric positive definite (SPD) matrices from the University of Florida Sparse Matrix collection (Table 5). These are the same 7 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 incomplete Cholesky decompositions. All results were averaged over 3 runs and 240 threads were used. The results show that in all cases, the solver iteration counts after 3 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 needing to fully converge the nonlinear residual.

Table 5

General SPD test matrices.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>No. equations</th>
<th>No. nonzeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>alfshell3</td>
<td>504855</td>
<td>17562551</td>
</tr>
<tr>
<td>thermal2</td>
<td>1228045</td>
<td>8580313</td>
</tr>
<tr>
<td>ecology2</td>
<td>999999</td>
<td>4995991</td>
</tr>
<tr>
<td>apache2</td>
<td>715176</td>
<td>4817870</td>
</tr>
<tr>
<td>G3circuit</td>
<td>1585478</td>
<td>7666826</td>
</tr>
<tr>
<td>offshore</td>
<td>259789</td>
<td>4242673</td>
</tr>
<tr>
<td>parabolic_fem</td>
<td>525825</td>
<td>3674625</td>
</tr>
</tbody>
</table>

Table 6

PCG iterations and nonlinear residuals for 3 sweeps with 240 threads of the general SPD test matrices. Results shown are averaged over three trials.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>sweeps</th>
<th>nonlin resid</th>
<th>PCG iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>alfshell3</td>
<td>0</td>
<td>1.58e+05</td>
<td>852.0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.66e+04</td>
<td>798.3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.17e+03</td>
<td>701.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4.67e+02</td>
<td>687.3</td>
</tr>
<tr>
<td></td>
<td>IC</td>
<td>0</td>
<td>685.0</td>
</tr>
<tr>
<td>thermal2</td>
<td>0</td>
<td>1.13e+05</td>
<td>1876.0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2.75e+04</td>
<td>1422.3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.74e+03</td>
<td>1314.7</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8.03e+01</td>
<td>1308.0</td>
</tr>
<tr>
<td></td>
<td>IC</td>
<td>0</td>
<td>1308.0</td>
</tr>
<tr>
<td>ecology2</td>
<td>0</td>
<td>5.55e+04</td>
<td>2000+</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.55e+04</td>
<td>1776.3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>9.46e+02</td>
<td>1711.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5.55e+01</td>
<td>1707.0</td>
</tr>
<tr>
<td></td>
<td>IC</td>
<td>0</td>
<td>1706.9</td>
</tr>
<tr>
<td>apache2</td>
<td>0</td>
<td>5.13e+04</td>
<td>1409.0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3.66e+04</td>
<td>1281.3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.08e+04</td>
<td>923.3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.47e+03</td>
<td>873.0</td>
</tr>
<tr>
<td></td>
<td>IC</td>
<td>0</td>
<td>869.0</td>
</tr>
<tr>
<td>G3circuit</td>
<td>0</td>
<td>1.06e+05</td>
<td>1048.0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4.39e+04</td>
<td>981.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.17e+03</td>
<td>869.3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.43e+02</td>
<td>871.7</td>
</tr>
<tr>
<td></td>
<td>IC</td>
<td>0</td>
<td>871.0</td>
</tr>
<tr>
<td>offshore</td>
<td>0</td>
<td>3.23e+04</td>
<td>401.0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4.37e+03</td>
<td>349.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.48e+02</td>
<td>299.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.46e+01</td>
<td>297.0</td>
</tr>
<tr>
<td></td>
<td>IC</td>
<td>0</td>
<td>297.0</td>
</tr>
<tr>
<td>parabolic_fem</td>
<td>0</td>
<td>5.84e+04</td>
<td>790.0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.61e+04</td>
<td>495.3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.46e+03</td>
<td>426.3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2.28e+02</td>
<td>405.7</td>
</tr>
<tr>
<td></td>
<td>IC</td>
<td>0</td>
<td>405.0</td>
</tr>
</tbody>
</table>
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 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, 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.

For our example, the level scheduled algorithm is faster for small numbers of threads, partly because the new ILU algorithm performs more work (3 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 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], as well as other approximate techniques for inverting ILU factors, 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

![Fig. 5. Execution time (seconds) comparison of the new ILU algorithm (3 sweeps) with the level scheduled ILU algorithm. The test case is a 5-point finite difference matrix on a 100 x 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.](image)
algorithm. Convergence within a small number of iterations is desired, however, and this depends on the norm or non-normality of $G$ and its pseudo-spectra [47]. Clearly, the non-normality 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 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 1 incomplete Cholesky 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, e.g., [44].

Our examples in this section have all been symmetric positive definite. 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 non-optimal (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 \(l_{kl}\),

\[
\frac{\partial G_{g(i,j)}}{\partial l_{kl}} = -\frac{u_{lj}}{u_{jj}}, \quad \text{if } k = i, \ l < j, \ j < i, \ (i, j) \in S, \ (l, j) \in S
\]

A column with \(k < l\) contains elements corresponding to partial derivatives with respect to \(u_{kl}\),

\[
\frac{\partial G_{g(i,j)}}{\partial u_{kl}} = -l_{ik}, \quad \text{if } l = j, \ k < j, \ j < i, \ (i, j) \in S, \ (i, k) \in S
\]

A column with \(k = l\) contains elements corresponding to partial derivatives with respect to \(u_{kk}\),

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

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 \(u_{jj} = 1\) for all \(1 \leq j \leq n\).

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

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

\[
c_{ijkl} = \begin{cases} |u_{lj}| & \text{for } k = i, \ l < j, \ j < i, \ (i, j) \in S \\ |u_{ij}| & \text{for } k = i, \ l < i, \ i < j, \ (i, j) \in S \\ 0 & \text{otherwise.} \end{cases}
\]
These expressions are from (A.1). Using indicator functions we can express $c_{ijkl}$ as

$$c_{ijkl} = I[k = i] I[l < j] I[j < i] I[(i,j) \in S] |u_{ij}| + I[k = i] I[l < i] I[i < j] I[(i,j) \in S] |u_{ij}|.$$ 

Define

$$s^1_{kl} = \sum_{j=1}^{n} \sum_{i=1}^{n} I[k = i] I[l < j] I[j < i] I[(i,j) \in S] |u_{ij}|$$

$$= \sum_{j=1}^{n} I[l < j] I[j < k] I[(k,j) \in S] |u_{ij}|$$

(A.4)

$$= \sum_{j=k+1}^{k-1} |u_{ij}|,$$

$$s^2_{kl} = \sum_{j=1}^{n} \sum_{i=1}^{n} I[k = i] I[l < i] I[i < j] I[(i,j) \in S] |u_{ij}|$$

$$= \sum_{j=1}^{n} I[l < k] I[k < j] I[(k,j) \in S] |u_{ij}|$$

(A.5)

$$= \sum_{j=k}^{n} |u_{ij}|.$$

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

$$s_{kl} = s^1_{kl} + s^2_{kl} = \sum_{j=l+1}^{k-1} |u_{ij}| + \sum_{j=k}^{n} |u_{ij}| = \sum_{j=l+1}^{k-1} |u_{ij}|.$$  

\[\Box\]

**Lemma A.2.** If $k < l$, then $s_{kl} = \sum_{i=k+1}^{n} |l_{ik}|$.

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 $t_{ik} = \sum_{p=1}^{k-1} t_{ip} u_{pk}$.

**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,

\[
s_{kk} = \sum_{i=1}^{n} \sum_{j=1}^{n} I[k = j] I[j < i] I[(i, j) \in S] |a_{ij} - t_{ij}|
\]

\[
= \sum_{i=1}^{n} I[k < i] I[(i, k) \in S] |a_{ik} - t_{ik}|
\]

\[
= \sum_{i=k+1}^{n} |a_{ik} - t_{ik}|, \quad \Box
\]

**Proof.** Proof of Theorem 3.8.

\[
\|G'(\tilde{L}, \tilde{U})\|_1 = \max_{(k,l) \in S} (s_{kl})
\]

\[
= \max \left( \max_{k>l} (s_{kl}), \max_{k<l} (s_{kl}), \max_{k} (s_{kk}) \right).
\]

For \(k > l\), applying Lemma A.1,

\[
s_{kl} = \sum_{j=l+1}^{n} |u_{lj}| \leq \sum_{j=l+1}^{n} |u_{ij}|
\]

\[
\max_{k>l} (s_{kl}) \leq \|\tilde{U}\|_{\infty}.
\]

For \(k < l\), applying Lemma A.2,

\[
s_{kl} = \sum_{i=k+1}^{n} |l_{ik}| \leq \sum_{j=k+1}^{n} |l_{ik}|
\]

\[
\max_{k<l} (s_{kl}) \leq \|\tilde{L}\|_{1}.
\]

For \(k = l\), applying Lemma A.3,

\[
s_{kk} = \sum_{i=k+1}^{n} |a_{ik} - t_{ik}|
\]

\[
\max_{k} (s_{kk}) = \|\tilde{R}_{L^*}\|_{1}
\]

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

\[
\|G'(\tilde{L}, \tilde{U})\|_1 = \max_{(k,l) \in S} (s_{kl}) \leq \max(\|\tilde{U}\|_{\infty}, \|\tilde{L}\|_{1}, \|\tilde{R}_{L^*}\|_{1}). \quad \Box
\]

22
Appendix B. Proof of Theorem 3.9.

Proof. We will first show that for the 5-point matrix on a 2-D 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,

\[
s_{kl} = \sum_{j=l+1}^{n} |u_{lj}|.
\]

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 2-D grid.) The 5-point stencils at vertex \( k \) and at 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,

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

in particular 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 hold true also for the 7-point stencil on a 3-D grid.

The 1-norm of the Jacobian is the maximum among

\[
\max_{k > l}(s_{kl}) = \max_{k > l}(|u_{lk}|),
\]

\[
\max_{k < l}(s_{kl}) = \max_{k < l}(|l_{lk}|),
\]

\[
\max_{k}(s_{kk}) = \max_{k} \sum_{i=k+1}^{n} |a_{ik}| = \|A_{L^*}\|_1
\]

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

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

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

REFERENCES


