PARILUT – A NEW PARALLEL THRESHOLD ILU FACTORIZATION

HARTWIG ANZT*, EDMOND CHOW†, AND JACK DONGARRA‡

Abstract. We propose a parallel algorithm for computing a threshold incomplete LU factorization. The main idea is to interleave a parallel fixed-point iteration that approximates an incomplete factorization for a given sparsity pattern with a procedure that adjusts the pattern. We describe and test a strategy for identifying nonzeros to be added and nonzeros to be removed from the sparsity pattern. The resulting pattern may be different and more effective than that of existing threshold ILU algorithms. Also in contrast to other parallel threshold ILU algorithms, much of the new algorithm has fine-grained parallelism.

Key words. incomplete factorization, ILU, parallel preconditioning

AMS subject classifications. 65F08, 65F50, 65Y05, 68W10

1. Introduction.

Preconditioners are important components in solving large, sparse linear systems via iterative methods. Among the most popular preconditioners for general problems are incomplete LU (ILU) factorizations [23]. These are based on the concept of truncating the fill-in that occurs in the Gaussian elimination process. How well an ILU factorization works as a preconditioner depends on the problem (the matrix and its ordering) and the factorization's sparsity pattern. This sparsity pattern can either be predetermined, as in the case of level-based ILU factorization, or it can be generated dynamically during the factorization process, as in the case of threshold-based ILU factorization. In the latter, the decision of whether or not an element is included in the sparsity pattern is traditionally based on whether its size is above a certain threshold [23]. In terms of preconditioner quality, incomplete factorizations based on thresholding [9, 14, 16, 19, 22] adapt the sparsity pattern to the matrix values and thus they can be superior to level-based strategies. On the other hand, thresholding makes the parallelization of the factorization process more challenging, compared to using a predetermined sparsity pattern. ILU factorizations based on thresholding cannot be parallelized using level scheduling or multicolor ordering techniques because the sparsity pattern is not known beforehand. The only existing strategy to parallelize threshold ILU factorizations is via graph partitioning or domain decomposition [3, 15]. Here, the interior portion of the subdomains can be factored in parallel, with one thread per subdomain. However, the factorization of the Schur complement corresponding to subdomain interfaces cannot be efficiently parallelized if thresholding will be performed dynamically. Further, this strategy only provides coarse-grained parallelism, and increasing the number of subdomains, which corresponds to changing the ordering of the matrix, generally degrades the preconditioner quality. There is currently no algorithm that computes a threshold ILU factorization with fine-grained parallelism.

In this paper, we propose a new algorithm for computing an ILU preconditioner that adapts its sparsity pattern to the values of the nonzeros in the matrix, and that can be considered a threshold ILU algorithm. The main idea is to combine a fixed-point iteration for approximating the incomplete factors for a given sparsity pattern [7] with a process that adaptively changes the sparsity pattern. Nonzeros

*Karlsruhe Institute of Technology, University of Tennessee (hanzt@icl.utk.edu)

†Georgia Institute of Technology (echow@cc.gatech.edu)

‡University of Tennessee, University of Manchester, Oak Ridge National Laboratory (dongarra@icl.utk.edu)

could both be added and removed from the sparsity pattern in each adaptive step. Thus the thresholding strategy is different from existing threshold ILU techniques and generates different sparsity patterns in general. While our initial goal was to match the preconditioner quality of existing threshold ILU factorizations, the new algorithm can generate factorizations that are better, for the same number of nonzeros in the factorizations. This is because, by *removing* nonzeros, we can exclude certain nonzeros that must be retained in existing techniques. In particular, a given fill-in element may be the result of an earlier fill-in element. In most existing threshold ILU techniques, the earlier fill-in element must be included in the pattern if the later fill-in element is included. This is not the case in our new strategy.

Section 2 provides background on using a fixed-point iteration to compute or update an ILU factorization. Section 3 proposes a new parallel threshold ILU factorization algorithm and discusses its implementation on a shared memory parallel computer. Section 4 shows numerical tests with the new preconditioner. Section 5 concludes the paper.

2. Fixed-point iteration for computing incomplete LU factorizations.

An incomplete factorization is the approximate factorization of a nonsingular sparse matrix A into the product of a sparse lower triangular matrix L and a sparse upper triangular matrix U, i.e., $A \approx LU$, where nonzero values are dropped in the factorization process. Conceptually, a sparsity pattern S is the set of matrix locations in L and U that are allowed to be nonzero. The choice of S can be made either before the factorization, or dynamically, during the generation of the incomplete factors.

The traditional approach of generating ILU preconditioners is based on a Gaussian elimination process. Gaussian elimination, however, is inherently sequential. Natural parallelism only exists if it is possible to find multiple rows that only depend on rows that already have been eliminated. There exist efforts to increase the parallelism with strategies like multicolor ordering or domain decomposition [3, 4, 11, 15, 13, 18, 21]. However, all these approaches have limited scalability, as they generally fail to leverage the fine-grained parallelism of current HPC architectures. Also, the parallelism increase often comes at the cost of reduced preconditioner quality [11, 18].

The recently proposed parallel *fixed-point ILU* algorithm [7] for a given sparsity pattern is fundamentally different from existing parallel ILU strategies, as it does not aim at parallelizing the Gaussian elimination process. Instead, a fixed-point iteration is used to approximate the incomplete factors. The goal is to iteratively generate incomplete factors L and U fulfilling the ILU property [23]

$$(LU)_{ij} = a_{ij}, \quad (i,j) \in S, \tag{1}$$

where $(LU)_{ij}$ denotes the (i,j) entry of the product of the computed factors L and U, and a_{ij} is the corresponding entry in the matrix A. This approach is appealing as the unknowns

$$\begin{aligned} l_{ij}, &\quad i \geq j, \quad (i,j) \in S, \\ u_{ij}, &\quad i < j, \quad (i,j) \in S \end{aligned}$$

of the incomplete factors can be computed iteratively in parallel using the constraints

$$\sum_{\substack{k=1 \\ (i,k)\in S \\ (k,j)\in S}}^{\min(i,j)} l_{ik} u_{kj} = a_{ij}, \quad (i,j) \in S \tag{2}$$

and the normalization $u_{ii} = 1$, i.e., the diagonal of U is all ones, assuming that the sparsity pattern S includes the diagonal, as it should to ensure nonsingularity of the resulting factors. From

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}, \quad i \geq j, \tag{3}$$

$$u_{ij} = \frac{1}{l_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right), \quad i < j, \tag{4}$$

with the sums over k implying $(i,k) \in S$ and $(k,j) \in S$, the strategy can be formulated as fixed-point iteration $x = G(x)$ where x is the vector containing the unknowns l_{ij} and u_{ij} for $(i,j) \in S$. Each fixed-point iteration is called a "sweep" in this paper. Algorithm 1 shows the pseudocode for one sweep of the fixed-point ILU algorithm. To promote convergence, we assume that the matrix A has been scaled such that it has a unit diagonal [7]. The initial values of L and U used to start the fixed-point iterations could be chosen to be the lower and upper triangular parts, respectively, of this scaled matrix.

Algorithm 1 One sweep of the fixed-point ILU algorithm.

```
Input sparse matrix $A$, desired sparsity pattern $S$, and current $L$ and $U$ factors
for $(i,j) \in S$ do
    if $i > j$ then
        $l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}\right)$
    else
        $u_{ij} = \left(a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}\right) / l_{ii}$
    end if
end for
```

The approach used in the fixed-point ILU algorithm has drawn attention due to its fine-grained parallelism, the possibility of updating all components in the incomplete factors simultaneously [7], its potential for architecture-specific optimization [6], and the ability for the fixed-point iterations to benefit from a good initial approximation [1].

The fixed point iteration can be performed asynchronously, usually resulting in faster convergence [7]. However, we use synchronous fixed-point iterations in our numerical tests in this paper. This allows our results to be reproduced.

3. New parallel threshold ILU algorithm.

To compute an ILU factorization where the sparsity patterns of the L and U factors are adapted to the values of A, we propose interleaving a method for computing an ILU factorization for a fixed sparsity pattern with a method for adjusting the sparsity pattern. However, in such an approach, we do not need to compute the ILU factorization for a given sparsity pattern exactly, since that sparsity pattern will be further adjusted. Therefore, it is natural to use one sweep of the fixed-point ILU algorithm to cheaply approximate an ILU factorization in between adjusting the sparsity pattern.

There could be many possibilities for how to adjust the sparsity pattern. In this section, we first describe one such procedure that we have found to work well. We then discuss how to implement this procedure in parallel on a shared memory computer.

3.1. Adjusting the ILU sparsity pattern.

To adjust the sparsity pattern for the ILU factorization, we use the heuristic of reducing the Frobenius norm of the ILU residual, $R = A - LU$. For symmetric positive definite A, the norm of R has been roughly correlated to the quality of the preconditioner [12]. However, there exists no simple quantity, including the norm of R, that can be minimized to produce an optimal incomplete factorization preconditioner, i.e., one that gives the least number of solver iterations. Thus the method we propose does not attempt to rigorously minimize the norm of R.

We separately consider the process of adding nonzeros and removing nonzeros from a given sparsity pattern. To add nonzeros to the sparsity pattern S of the current L and U approximations, consider an entry r_{ij} of $R = A - LU$. If L and U are exact ILU factors for the pattern S, then r_{ij} would be zero, from property (1). If r_{ij} is large in magnitude, then either L and U are very inaccurate incomplete factors, or (i, j) is not in the sparsity pattern of S. In the latter case, it is natural to consider (i, j) as a nonzero location to add to the sparsity pattern.

A nonzero location (i, j) in R is called a *candidate location*, or simply a *candidate*. To be clear, the candidate locations are the union of the nonzero locations in A and the matrix product LU that are not already in the current sparsity pattern S. A candidate can be added to S if its corresponding r_{ij} is large in magnitude according to a threshold. Alternatively, all candidates could be added to S. We suggest using the latter, simpler rule, that does not require selecting among the candidates. We show some tests to support this choice in Section 4.1.

We note that the sparsity pattern of LU corresponds to the pattern of the level-1 ILU factorization if the pattern of L and U corresponds to the level-0 ILU factorization. (Such simple relations do not hold for higher level ILU factorizations except for some regularly structured matrices.) This is additional justification to choose elements in the sparsity pattern of the product LU as candidates. However, it is also important to consider nonzero locations in A as candidates as well, since there can be large elements r_{ij} that are not in the sparsity pattern of LU. As an example, consider the following sparsity patterns for A, L, U, and LU,

$$A = \begin{bmatrix} x & x & 0 \\ x & x & x \\ 0 & x & x \end{bmatrix}, \quad L = \begin{bmatrix} x & 0 & 0 \\ 0 & x & 0 \\ 0 & x & x \end{bmatrix}, \quad U = \begin{bmatrix} x & x & 0 \\ 0 & x & x \\ 0 & 0 & x \end{bmatrix}, \quad LU = \begin{bmatrix} x & x & 0 \\ 0 & x & x \\ 0 & x & x \end{bmatrix}.$$

If location $(2, 1)$ is not in the sparsity pattern of L as shown above, then this location can never become part of the sparsity pattern of the incomplete factorization. Products of subsequent L and U factors will never contain a nonzero at $(2, 1)$ if the nonzero locations of A are not also considered as candidate locations.

Finally, we specify the procedure for removing nonzeros when adjusting the sparsity pattern S. Here, we simply remove nonzeros in L and U if they are small in magnitude. To select which nonzeros to remove, either a threshold on the size of the nonzeros and/or a threshold on the number of nonzeros can be used.

3.2. Algorithm outline. An outline of the parallel threshold ILU algorithm, called "ParILUT," is shown in Algorithm 2. Starting with an initial approximation for L and U, the algorithm performs several iterations, which we call "steps." We use this term to differentiate these types of steps from solver iterations; also recall that fixed-point iterations are called "sweeps" in this paper. In each step, (1) nonzeros are added to L and U, then (2) all nonzero values in L and U are adjusted by a fixed-point ILU sweep, then (3) selected nonzeros are removed from L and U, followed by (4) another adjustment of the nonzero values by a fixed-point ILU sweep. For simplicity, we do not consider performing more than one fixed-point ILU sweep at a time.

The ParILUT algorithm can be adapted for SPD matrices for a computational savings of up to 50%. In this version, called “ParICT,” only the incomplete Cholesky factor (also denoted by L) is computed.

Algorithm 2 ParILUT.

```
Input $A$, initial $L$ and $U$ factors
Output: incomplete factors $L$ and $U$
repeat
    Identify candidate locations
    Compute ILU residual at candidate locations
    Estimate ILU residual norm
    Add $m_L$ nonzeros to $L$ and $m_U$ nonzeros to $U$
    Do one sweep of the fixed-point ILU algorithm
    Remove the $m_L$ and $m_U$ smallest magnitude elements from $L$ and $U$, respectively
    Do one sweep of the fixed-point ILU algorithm
until (convergence)
```

Within one step, we have chosen to add nonzeros to the sparsity pattern before removing nonzeros. Compared to removing and then adding nonzeros, this gives somewhat more accurate L and U factors at the end of each step, although the cost is slightly higher because the fixed-point ILU sweeps then operate on more nonzeros. We have also chosen, at each step, to remove the same number of nonzeros as the number of nonzeros that were added earlier in the step (with nonzeros for L and U counted separately). This keeps the total number of nonzeros in L and U fixed from step to step. It is possible that slowly allowing the number of nonzeros in L and U to grow with each step (or some other strategy) would ultimately give more accurate L and U factors, but our scheme allows us to avoid devising the details of such a strategy.

The initial approximations for L and U that we use in this paper are the lower and upper triangular parts of A, respectively. This corresponds to using the level-0 ILU pattern as the initial sparsity pattern S, which has a reasonable number of nonzeros for an incomplete factorization. It is, of course, also possible to use other sparsity patterns for the initial approximations for L and U, e.g., the level-1 ILU pattern. In this case, the initial approximations for L and U are still, respectively, the lower and upper triangular parts of A, but in an implementation, explicit zeros are added to the data structures for L and U to represent the additional nonzeros contained in the level-1 pattern.

In the procedure for adjusting the sparsity pattern, we must specify the initial values of the selected candidate locations, i.e., the matrix locations added to L and U. One natural choice for these initial values is zero. However, since we use a single fixed-point sweep to adjust the nonzero values, the zeros added do not contribute to adjusting existing nonzero values of L and U until they themselves have been updated to a nonzero value.

Another choice of initial value for a newly added l_{ij} or u_{ij} is

$$l_{ij} = r_{ij}/u_{jj},\ \text{if } i > j \quad \text{or} \quad u_{ij} = r_{ij}/l_{ii},\ \text{if } i < j.$$

If only a single nonzero is added to the sparsity pattern, these are the values of l_{ij} or u_{ij} that would reduce r_{ij} to zero. However, adding such a nonzero will also generally change the residual at other locations besides at (i, j). In the expressions above, the case $i = j$ is excluded because the diagonals of L and U must always be nonzero and are never added or removed from the sparsity pattern. Recall from Section 2 that the matrix A has been scaled such that it has a unit diagonal, and that the initial approximations for L and U are the lower and upper triangular parts of this scaled A. Since the diagonal elements of L and U will typically be close to 1 as the algorithm progresses, the simpler formulas for the initial values,

$$l_{ij} = r_{ij}, \text{ if } i > j \quad \text{or} \quad u_{ij} = r_{ij}, \text{ if } i < j \tag{5}$$

could be used as an expedient. We use these formulas for our numerical tests in this paper.

In summary, there are no parameters to choose in the way we have defined the ParILUT algorithm. The number of nonzeros in the factors is controlled by the number of nonzeros in the initial L and U factors used as the initial guess to the algorithm. Such a strategy makes it easy to predict the number of nonzeros in the preconditioner, in contrast to most forms of threshold-based incomplete factorizations.

3.3. Implementation details. The ParILUT algorithm for adjusting the sparsity pattern can divided into the following building blocks.

Convert U from CSC to CSR format. For efficiency of the fixed-point ILU algorithm (Algorithm 1), L is stored by rows in compressed sparse row (CSR) format and U is stored by columns in compressed sparse column (CSC) format. Unfortunately, this is not an efficient layout of the matrices for forming the sparse product LU that is needed for identifying candidate nonzero locations. To form the sparse product efficiently, both L and U should be stored by rows (or by columns). Thus, we create a copy of U in CSR format, which is a sparse matrix transpose operation, before forming the LU product.

To implement this transpose operation in parallel, each thread is assigned a subset of rows of the transpose of U to be constructed. All threads traverse the original CSC data structure of U in parallel and when a thread encounters nonzero elements in its rows, these elements are added to the new CSR data structure.

Identify candidates. Now that L and a copy of U are in CSR format, the set of candidate locations is generated by performing a specialized symbolic multiplication of L and U that includes the nonzero locations in A and ignores nonzero locations already in the current sparsity pattern S. Since L and U are in CSR format, the result is in CSR format. This procedure is parallelized by assigning each thread a subset of the rows of the result.

Compute residuals for the candidate locations. Separately, we compute the residuals r_{ij} corresponding to the candidate locations, denoted by the set S_c. The residual r_{ij} for $(i,j) \in S_c$ can be computed with fine-grained parallelism using an algorithm similar in structure to Algorithm 1. Each thread computes r_{ij} for a subset of the locations in S_c.

These residuals can be used for selecting which candidate locations to add to L and U if not all candidates will be added (see Section 3.1). Also, these residuals can be used in the formulas for providing initial values for the candidate nonzeros, given in (5). Further, these residuals can be used to estimate the ILU residual norm, to be discussed next.

Estimate ILU residual norm. The square of the ILU residual norm is

$$\|R\|_F^2 = \|(R)_S\|_F^2 + \|(R)_{S_c}\|_F^2$$

where S is the current sparsity pattern (before adding nonzeros) and S_c is the sparsity pattern of the set of candidates. The notation $(R)_S$ indicates the matrix R masked by the sparsity pattern S, i.e., $r_{ij} = 0$ if $(i,j) \notin S$. The term $\|(R)_S\|_F^2$ is small or zero if L and U are good approximations to the incomplete factorization with sparsity pattern S. Assuming that this is the case, we estimate the square of the ILU residual norm as $\|R\|_F^2 \approx \|(R)_{S_c}\|_F^2$. If the residuals for the candidate locations have been computed, then $\|(R)_{S_c}\|_F^2$ is easily computed via a parallel reduction across the threads. This approximation to the ILU residual norm could be used to help detect convergence of ParILUT, i.e., when the ILU residual norm stagnates, and also could be used to detect when the ParILUT steps are diverging.

Select nonzeros to add. Instead of adding all the candidates to the current sparsity pattern, it is possible to add a subset of the candidates. This is discussed further in Section 4.1.

Convert to CSC format. The nonzeros to be added to U were generated and stored in CSR format, but U is in CSC format. Thus a second transposition is required, which is typically smaller than the first, as it only involves the nonzeros to be added to U.

In the ParICT version of the algorithm, this conversion is not needed because there is no U factor. The nonzeros to be added to L, which is in CSR format, are already stored in a row-based format because they were generated in that format.

Add nonzeros to L and U. Adding nonzeros to L (or to U) involves merging two CSR (or CSC) data structures into one. This is implemented in two phases. In the first phase, the two data structures are traversed and the number of nonzeros in each row (or column) of the result is counted. Then data structures for the result are allocated and the second phase fills these data structures. Both phases are parallelized by partitioning the rows (or columns) of the result among the threads.

Due to the sparse dot products between rows of L and columns of U used in the fixed-point ILU algorithm, it is also desirable to have the nonzeros in rows of L sorted by column index and nonzeros in columns of U sorted by row index. This property is enforced by the second phase of the merging routine.

Select nonzeros to remove. If m_L nonzeros were added to L and m_U nonzeros were added to U, then to control the total number of nonzeros in the incomplete factors, we now remove m_L locations from L and m_U locations from U. The nonzeros we remove are the ones with the smallest absolute value. To accomplish this, we need to select a threshold for L such that there are exactly m_L nonzeros in L smaller than the threshold, and similarly for U. Choosing these thresholds, however, requires an expensive and hard-to-parallelize selection process [5].

For our purpose, the thresholds do not need to be exact, which would simply give slightly different numbers of nonzeros in L and U from step to step. Then, to efficiently parallelize the threshold selection process, we split the nonzeros in a factor (L or U) into subsets and select multiple thresholds in parallel. More precisely, we partition a set of nonzero values in a factor into p subsets and generate p thresholds, each separating the m_L/p (or m_U/p) smallest magnitude values in its subset. We take the median of the p thresholds as the threshold used for marking nonzeros to remove from the factor. We choose p to be 4 times the number of threads. Smaller subsets reduce the overall runtime for selecting the threshold, at a cost of less accuracy of the selected threshold. For factors containing fewer than 100,000 nonzeros, however, we use the standard sequential algorithm for selecting the threshold, which has worst-case linear complexity [5].

Remove nonzeros in L and U. Removing nonzeros is realized by contracting the current L (or U) factor into new CSR (or CSC) sparse matrix data structures, skipping the nonzeros previously marked for removal. This operation is implemented in two phases. First, the number of nonzeros in each row (or column) of the result is counted in parallel, and a global reduction is used to obtain the total nonzero count. After allocating the memory for the new structure, the second phase copies the nonzeros not marked for removal in row-parallel (or column-parallel) fashion.

4. Numerical Tests.

Comparisons are made with the threshold incomplete factorizations implemented in Matlab version 2017b. We call these the "classical" ILUT and ICT factorizations, the latter being for SPD matrices. For comparison purposes, the threshold is always chosen such that the number of nonzeros in the resulting factorization is almost equal to the number of nonzeros in the zero-fill incomplete factorization. We also make comparisons with the zero-fill incomplete factorizations, denoted by ILU(0) and IC(0).

The ParILUT algorithm is implemented using double precision arithmetic in C and parallelized using OpenMP. As already mentioned, we always take the upper and lower triangular parts of the problem matrix as the initial L and U factors for the ParILUT algorithm. Thus the factorizations that are generated have approximately the same number of nonzeros as their corresponding zero-fill incomplete factorizations.

The implementations of the PCG and GMRES solvers are taken from the MAGMA-sparse software package [2]. Full (not restarted) GMRES was used in order to focus the results on the effect of the preconditioners. For each matrix, the right-hand side vector contains random values uniformly chosen from $[-0.5, 0.5]$. The iterative solvers are started with a zero vector initial guess. Convergence is declared once the relative residual norm decreases by a factor of 10^{-10}.

4.1. Initial tests and observations.

In this section, we illustrate some ideas with a small SPD matrix, ANI3, from an anisotropic diffusion problem on a square discretized using irregular linear triangles by the finite element method (FEM). The ratio of the diffusion coefficients in the x and y directions is 1000. The matrix has 741 equations and 4951 nonzeros, and reverse Cuthill-McKee (RCM) reordering was applied. Refinements of the mesh will give larger matrices but classical incomplete Cholesky breaks down with negative pivots for these larger matrices. We first show the convergence of PCG with IC(0), classical ICT, and ParICT preconditioning. We then show the effect of varying the number of candidate locations that are added at each step of ParICT. Lastly, for this small matrix, we compare the sparsity patterns resulting from classical ICT and ParICT.

PCG convergence. ParICT preconditioners were generated using up to 15 steps of the algorithm and each preconditioner was used for the PCG method. Figure 1(left) shows the PCG iteration counts (along with results with the parameter δ to be explained later). The number of solver iterations using IC(0) and classical ICT are also shown. Classical ICT preconditioning gives about half the number of iterations as IC(0) preconditioning. Compared to classical ICT, ParICT preconditioning gives about the same number of iterations after 4 or 5 steps. It is also observed that ParICT can be slightly better than classical ICT when many steps are taken. This is possible because the sparsity patterns for ICT and for ParICT are not guaranteed to be the same.

Figure 1(right) shows the incomplete factorization residual norm $\|A - LL^T\|_F$ where L is the computed incomplete Cholesky factor. The norm is larger for IC(0)

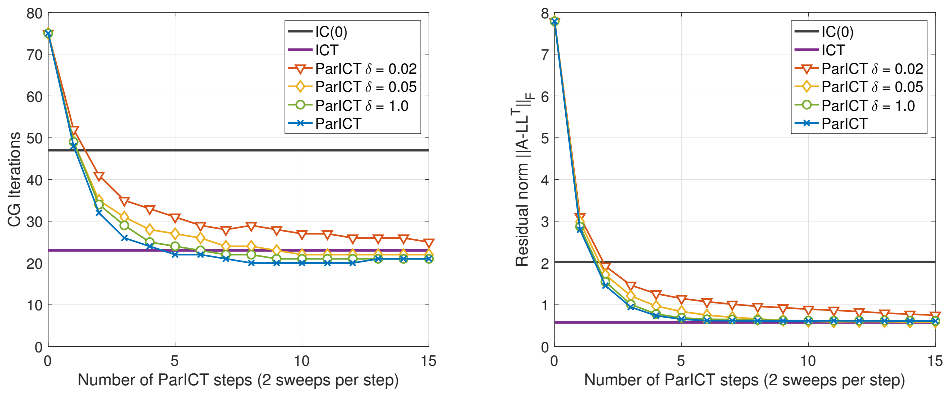

Fig. 1. *PCG iteration count and ILU residual norm for the ParICT preconditioner computed with different number of steps for the* ANI3 *matrix. The results for IC(0) and classical ICT are also shown.*

than for classical ICT. The norm for ParICT after many steps is very close to that for classical ICT. The ParICT norm does not appear to smaller than the classical ICT norm when the ParICT preconditioner gives fewer PCG iterations than classical ICT. The incomplete factorization residual norm is not sensitive enough to identify differences between preconditioners that lead to small but noticeable differences in PCG iteration count.

Number of nonzero locations to add. Figure 1 also shows results for variant ParICT preconditioners where different numbers of nonzero locations are added at each step. Recall that ParICT generates candidate locations to add to the sparsity pattern of the incomplete factors, and for simplicity, *all* these candidate locations are added. It is possible, however, to add only a subset of these locations. This reduces the cost of the subsequent operations, such as the fixed-point ILU sweeps. The nonzeros are added more gradually, but it is not clear how this typically affects the quality of the resulting factorization after many steps.

Procedures for selecting the subset of nonzero locations to add, however, can be very costly. In some sense, it is ideal to choose the nonzero locations associated with the largest magnitude locations in the residual $R = A - LU$, however, selection algorithms to accomplish this are difficult to parallelize unless severe approximations to this process are permitted.

Here, we propose a cheaper strategy for selecting the nonzero locations to add, which consists of two phases. In the first phase, the nonzero locations with the largest magnitude residual in each row of the current L factor (and each column of the current U factor, in the nonsymmetric case) are identified. This can be performed in parallel without interaction between threads. In the second phase, a fraction of these nonzeros are selected, based on the magnitude of their residuals. Cooperation between threads is required in this phase. The first phase reduces the number of nonzero locations that need to be considered. Some of these nonzero locations, however, can be very small, and these are eliminated in the second phase. The second phase also reduces the total number of nonzero locations to be added, in case this number is still larger than desired after the first phase. The fraction of nonzeros to keep in the second phase is the δ parameter shown in the legend in Figure 1.

The figure shows that utilizing only a subset of the possible nonzero locations to be added is never better, at least for this test problem. Smaller values of δ lead to worse results in terms of PCG convergence.

Resulting sparsity pattern. The new parallel threshold incomplete factorization algorithm in general does not produce the same sparsity patterns for the incomplete factors as the classical incomplete factorization. It is thus interesting to check how similar these patterns are. To quantify the similarity between the sparsity pattern of two matrices, A and B, we define the "pattern discrepancy" as the number of nonzero locations that are included in one sparsity pattern but not the other, i.e., the number of elements in

$$\{(i,j) \mid a_{ij} \neq 0 \wedge b_{ij} = 0\} \cup \{(i,j) \mid b_{ij} \neq 0 \wedge a_{ij} = 0\}.$$

Figure 2 shows the pattern discrepancy for the ANI3 matrix. Starting with an IC(0) pattern for the ParICT algorithm, the pattern discrepancy decreases as the ParICT sparsity pattern is updated. Note, however, that the pattern discrepancy never reaches zero, i.e., the ultimate ParICT sparsity pattern and the classical ICT sparsity pattern are different, although the preconditioners can be comparable in effectiveness.

Adding all possible nonzero locations at each step results in the fastest decrease of the pattern discrepancy. In the plot, dotted lines show the maximum possible decrease of the pattern discrepancy for factorizations using the δ parameter. For smaller values of δ, the sparsity pattern changes more slowly, and the change in the sparsity pattern comes close to the maximum possible rate of decrease of the pattern discrepancy. This means that the sparsity pattern is adapted efficiently in the sense that the nonzeros that are added at each step are typically not removed at future steps. In the remaining numerical tests in this paper, however, we consider the case where all possible nonzero locations are added, as this usually results in the fastest convergence without degrading the quality of the preconditioner constructed in the end.

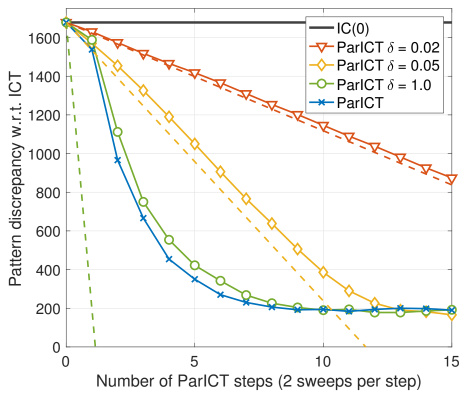

FIG. 2. *Pattern discrepancy between the classical ICT factor and the ParICT factor, the latter computed using different numbers of steps. The discrepancy between the classical ICT factor and the IC(0) factor is also shown. The test matrix is* ANI3. *Dotted lines show the maximum possible decrease of the pattern discrepancy for ParICT factorizations using the δ parameter.*

4.2. Scaling up the problem size. We now illustrate how the ParICT algorithm scales as the problem size is increased. The mesh for the ANI3 problem was refined uniformly to obtain FEM discretizations with 3081, 12561, 50721, and 203841 equations. (The latter three matrices, called ANI5, ANI6, and ANI7, will also be used in later tests.) For the timings in this section, the ParICT algorithm was run on an Intel Haswell system using 20 threads on 20 cores. The timings are the average of several separate runs.

For these matrices, as the ParICT steps progress, the number of candidates found at each step stays approximately constant. To see how the number of candidates varies for problems of different sizes, Figure 3(left) plots this relation for step 3 of the ParICT algorithm. The number of candidates is proportional to the matrix dimension, and this result could be expected to apply to other FEM and regular meshes.

Figure 3(right) shows how the total time for step 3 of the ParICT algorithm increases for increasing problem size. The scaling is expected to be at best linear in the matrix dimension, like for sparse matrix multiplication, but we observe sublinear scaling for small problem sizes, and behavior closer to linear scaling for large problem sizes. The reason for this is likely because the ParICT building blocks are less efficient for smaller problem sizes. The scaling of two of the building blocks is also shown in the figure.

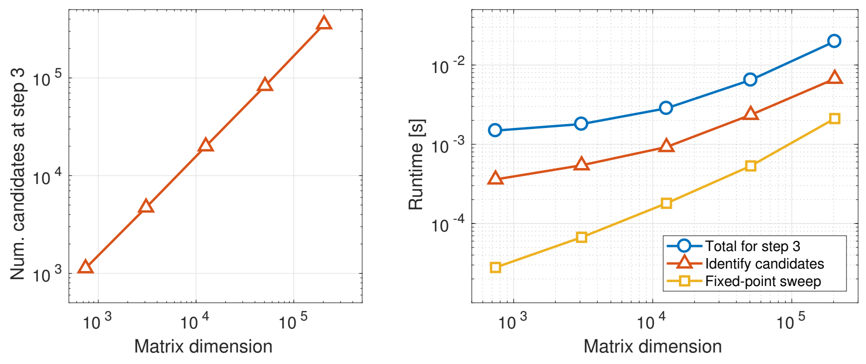

FIG. 3. *(Left) Number of candidates at step 3 of the ParICT algorithm vs. matrix dimension. (Right) Time (in seconds) as a function of matrix dimension for step 3 of the ParICT algorithm, along with the portion of time for identifying candidates and for the first fixed-point sweep of the step. These results are for a sequence of finite element matrices of different sizes.*

4.3. Benchmark problems. Benchmark test matrices are listed in Table 1 and include both nonsymmetric and SPD matrices. The matrices have different origins, as shown in the table. All matrices are reordered using RCM. For the nonsymmetric matrices, the RCM permutation is found using the structure of $A + A^T$.

For the SPD problems, Table 2 shows PCG iteration counts using various preconditioners. Zero to five steps were used for ParICT. The first observation is that classical ICT often breaks down, i.e., encounters a negative pivot. Recall that we try to select a threshold to generate factors with the same number of nonzeros as IC(0). For the matrices tested, either the classical ICT factorization succeeds for wide ranges of the threshold, or it fails for wide ranges of the threshold. In the latter case, thresholds that lead to completion of the factorization are generally those that give either very accurate factorizations (small thresholds) or very inaccurate ones (large thresholds and factorizations with sparsity patterns that are nearly diagonal).

It is also observed that ParICT can successfully compute an incomplete factorization for cases where classical ICT fails. This is the case for the ANI matrices. For TOPOPT060 and TOPOPT120 where the classical ICT factorization also fails, ParICT

Table 1
Test matrices.

Matrix	Origin	SPD	Num. Rows	Nz	Nz/Row
ANI5	2D anisotropic diffusion	yes	12,561	86,227	6.86
ANI6	2D anisotropic diffusion	yes	50,721	349,603	6.89
ANI7	2D anisotropic diffusion	yes	203,841	1,407,811	6.91
APACHE1	Suite Sparse [10]	yes	80,800	542,184	6.71
APACHE2	Suite Sparse	yes	715,176	4,817,870	6.74
CAGE10	Suite Sparse	no	11,397	150,645	13.22
CAGE11	Suite Sparse	no	39,082	559,722	14.32
JACOBIANMAT0	Fun3D fluid flow [20]	no	90,708	5,047,017	55.64
JACOBIANMAT9	Fun3D fluid flow	no	90,708	5,047,042	55.64
MAJORBASIS	Suite Sparse	no	160,000	1,750,416	10.94
TOPOPT010	Geometry optimization [24]	yes	132,300	8,802,544	66.53
TOPOPT060	Geometry optimization	yes	132,300	7,824,817	59.14
TOPOPT120	Geometry optimization	yes	132,300	7,834,644	59.22
THERMAL1	Suite Sparse	yes	82,654	574,458	6.95
THERMAL2	Suite Sparse	yes	1,228,045	8,580,313	6.99
THERMOMECH_TC	Suite Sparse	yes	102,158	711,558	6.97
THERMOMECH_DM	Suite Sparse	yes	204,316	1,423,116	6.97
TMT_SYM	Suite Sparse	yes	726,713	5,080,961	6.99
TORSO2	Suite Sparse	no	115,967	1,033,473	8.91
VENKAT01	Suite Sparse	no	62,424	1,717,792	27.52

is able to compute a useful factorization with up to 2 steps, but then starts to produce worse factors. Since square roots of diagonal elements of L are used in the fixed-point incomplete Cholesky factorization algorithm, ParICT can also break down. These matrices are challenging for both classical ICT and ParICT.

In all other cases, three steps of the ParICT algorithm are sufficient to generate a preconditioner comparable or superior to that of classical ICT.

For the nonsymmetric problems, Table 3 shows GMRES iteration counts using various preconditioners. We also include, however, the SPD problems for which the classical ICT factorization broke down, i.e., these matrices are treated as nonsymmetric matrices. The main observation in Table 3 is that ParILUT is able to compute a preconditioner comparable in quality to that of classical ILUT with a small number of adaptive steps.

4.4. Execution time breakdown and scalability. Timing tests here and in the rest of this paper are conducted with an Intel Xeon Phi 7250 (KNL) processor with 68 cores running at 1.40 GHz. The on-package MCDRAM is configured in “hybrid mode,” which splits the 16 GB memory into 8 GB addressable main memory and 8 GB cache. The `scatter` thread affinity setting is used in the tests. The timing results we report are averaged over several runs for each thread count.

We focus on the timing and scalability of the individual building blocks of the ParILUT algorithm for two matrices, TOPOPT120 and THERMAL2. These two matrices have a similar number of nonzeros, but TOPOPT120 has much fewer rows and thus much more nonzeros per row. Figure 4 shows the execution time and parallel scalability of the ParILUT building blocks for one step of the ParILUT algorithm (the SPD matrices are treated as being nonsymmetric for timing purposes).

The results show that candidate search, i.e., identifying candidate locations to add to the sparsity pattern, is the most expensive building block of ParILUT. This is especially true for the TOPOPT120 matrix, where the higher nonzero-per-row ratio results in a much larger number of nonzero elements in the ILU residual $A - LU$.

Table 2

PCG iteration counts using various preconditioners. ParICT preconditioners were generated with various numbers of steps. A dash (–) indicates a breakdown of the classical ICT factorization or lack of PCG convergence when using the ParICT preconditioner.

				ParICT					
Matrix	no prec.	IC(0)	ICT	0	1	2	3	4	5
ANI5	951	226	–	297	184	136	108	93	86
ANI6	1,926	621	–	595	374	275	219	181	172
ANI7	3,895	1,469	–	1,199	753	559	455	405	377
APACHE1	3,727	368	331	1,480	933	517	321	323	323
APACHE2	4,574	1,150	785	1,890	1,197	799	766	760	754
THERMAL1	1,640	453	412	626	447	409	389	385	383
THERMAL2	6,253	1,729	1,604	2,372	1,674	1,503	1,457	1,472	1,433
THERMOMECH_DM	21	8	8	8	7	7	7	7	7
THERMOMECH_TC	21	8	7	8	7	7	7	7	7
TMT_SYM	5,481	1,453	1,185	1,963	1,234	1,071	1,012	992	1,004
TOPOPT010	2,613	692	331	845	551	402	342	316	313
TOPOPT060	3,123	871	–	988	749	693	1,116	–	–
TOPOPT120	3,062	886	–	991	837	784	2,185	–	–

Table 3

GMRES iteration counts using various preconditioners. ParILUT preconditioners were generated with various numbers of steps.

				ParILUT					
Matrix	no prec.	ILU(0)	ILUT	0	1	2	3	4	5
ANI5	882	172	78	278	161	105	84	74	66
ANI6	1,751	391	127	547	315	211	168	143	131
ANI7	3,499	828	290	1,083	641	459	370	318	289
CAGE10	20	8	8	9	7	8	8	8	8
CAGE11	21	9	8	9	7	7	7	7	7
JACOBIANMAT0	315	40	34	63	36	33	33	33	33
JACOBIANMAT9	539	66	65	110	60	55	54	53	53
MAJORBASIS	95	15	9	26	12	11	11	11	11
TOPOPT010	2,399	565	303	835	492	375	348	340	339
TOPOPT060	2,852	666	397	963	584	445	417	412	410
TOPOPT120	2,765	668	396	959	584	445	416	408	408
TORSO2	46	10	7	18	8	6	7	7	7
VENKAT01	195	22	17	42	18	17	17	17	17

Luckily, candidate search also scales fairly well, giving a speedup between 40 to 50 for 68 threads.

The fixed-point ILU sweeps and the element-wise parallel computation of the ILU residuals also account for a large portion of the total computation time. These routines are fine-grained parallel and scale almost perfectly for both problems. Note that the first fixed-point sweep is more expensive than the second, since the first one utilizes a sparsity pattern that includes candidate nonzero locations just added.

The components of ParILUT with the worst scaling are the conversions between CSR and CSC formats. For these components, speedup is limited to about 10 on 68 threads. These components are likely memory bandwidth limited. The component with next worst scaling is the estimation of the ILU residual norm. This component is a reduction operation across all threads and is thus communication bound. The execution time for this component, however, is very small.

Overall, the ParILUT algorithm can use the 68 cores of the KNL system efficiently, achieving 37× speedup for the THERMAL2 problem, and 52× speedup for the

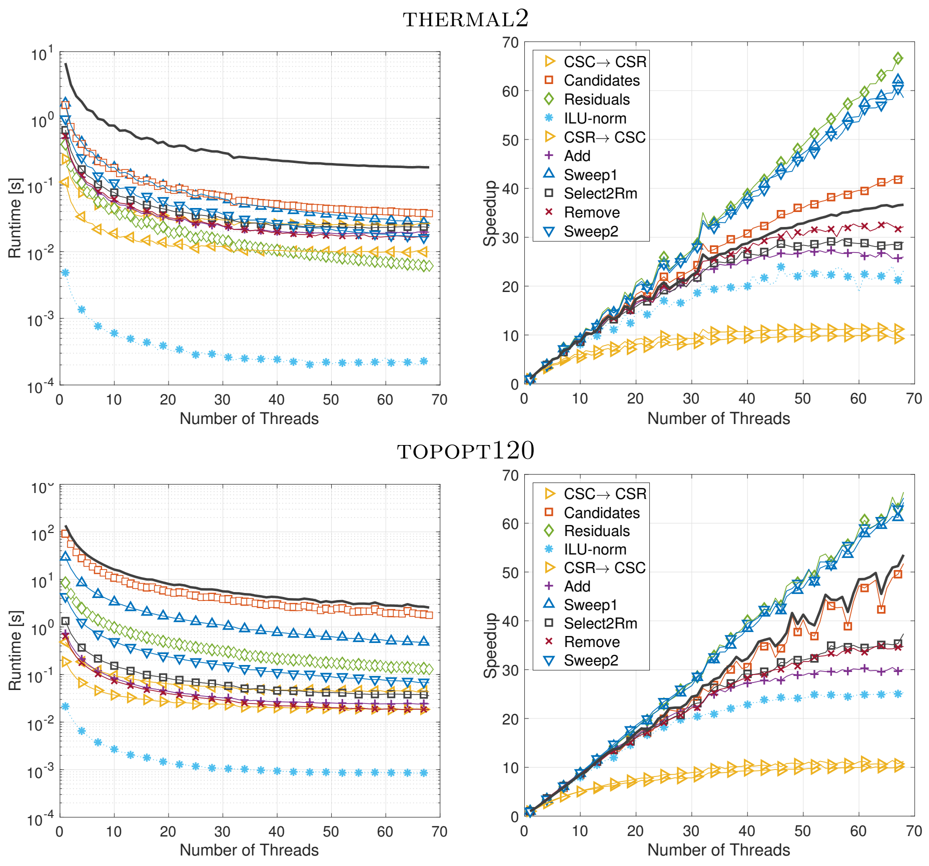

FIG. 4. *For the* THERMAL2 *matrix (top row of plots) and* TOPOPT120 *matrix (bottom row of plots), the execution time (left) and multithreaded speedup (right) for the various building blocks of ParILUT. The black solid line (without markers) is the overall execution time (left) and the total speedup (right).*

TOPOPT120 problem.

Comparing the raw timings, the factorization for TOPOPT120 is an order of magnitude more expensive than for THERMAL2. The main reason is the high cost of candidate search for TOPOPT120, due to its large average number of nonzeros per row. To reduce the cost, and to reduce the number of candidates generated, an option is to perform the multiplication LU for generating candidates with only large magnitude values in L and U. However, we did not test this option in our numerical experiments.

4.5. Timing comparison. Table 4 shows factorization timings for ParICT (on the SPD matrices), ParILUT (for all matrices) and, for comparison, for the threshold ILU code implemented in the SuperLU library [17]. SuperLU is designed for nonsymmetric matrices and a Cholesky version of the incomplete factorization is not available. Thus, we also treat the SPD matrices as being nonsymmetric when using the SuperLU code.

The incomplete factorization in SuperLU uses supernodes to gain efficiency, but is a single-threaded code. To control the number of nonzeros in the incomplete fac-

TABLE 4

Execution time for ParICT, ParILUT, and SuperLU incomplete LU factorization (all in seconds), and speedup of ParILUT over SuperLU incomplete factorization. For ParILUT and ParICT, five steps were used. For ParICT, dash (–) indicates that the matrix is nonsymmetric and no ParICT factorization was computed.

Matrix	ParICT	ParILUT	SuperLU	Speedup
ANI5	0.16	0.18	0.65	3.54
ANI6	0.19	0.23	2.68	11.45
ANI7	0.30	0.43	10.48	24.11
APACHE1	0.24	0.33	6.20	18.89
APACHE2	0.65	1.17	62.27	53.21
CAGE10	–	0.29	7.82	26.53
CAGE11	–	0.51	60.89	119.29
JACOBIANMAT0	–	6.99	153.17	21.90
JACOBIANMAT9	–	7.08	153.84	21.72
MAJORBASIS	–	0.46	9.23	20.05
THERMAL1	0.19	0.24	6.09	24.90
THERMAL2	0.68	1.17	91.83	78.59
THERMOMECH_DM	0.22	0.27	15.20	56.29
THERMOMECH_TC	0.20	0.22	7.65	34.03
TMT_SYM	0.41	0.67	53.42	79.81
TOPOPT060	8.01	13.96	43.12	3.09
TOPOPT120	8.24	14.15	44.22	3.13
TORSO2	–	0.26	10.78	41.34
VENKAT01	–	0.72	8.53	11.85

tors, SuperLU uses a "fill factor" parameter. This allows us to generate factors with numbers of nonzeros comparable to those in ParILUT, which in turn are comparable to those in ILU(0).

We first observe in Table 4 that the timings for ParICT are less than those for ParILUT. For the largest problems, the ParICT time is about half of the ParILUT time, but the ParICT time is generally more than half when the problem size is small. The main observation is that ParILUT is much faster than the incomplete factorization in SuperLU. This is mostly explained by the fact that 68 KNL threads were used for ParILUT (and ParICT), but only one thread could be used for the SuperLU single-threaded code.

5. Conclusion.

Threshold-based ILU factorizations are typically more accurate than level-based ILU factorizations with similar numbers of nonzeros. However, up to now, there exists no fine-grained parallel algorithms for threshold-based factorizations. We have presented such an algorithm in this paper, basing it on a fixed-point iteration procedure.

The sparsity pattern of the resulting L and U factors of the new algorithm are generally different than those generated by existing threshold ILU factorizations. We observed that, for some SPD matrices, classical ICT factorizations may break down for large ranges of the threshold parameter. However, the new ParICT factorization may be successfully computed, likely because it is able to find a better sparsity pattern.

The robustness of ILU preconditioning is a persistent issue. The observation has been made in the past, at least for indefinite matrices, that threshold-based ILU factorizations can produce very poor factorizations when small pivots are produced, generating very large off-diagonal entries that are propagated through the factorization [8]. Factorizations where the sparsity pattern is fixed beforehand (such as zero-fill ILU) do not propagate these large, likely erroneous entries. Similarly, this phenomenon does not occur in the new threshold-based ILU factorization presented here, as the sparsity pattern is fixed when the matrix values in the factors are adjusted, and a small number of adaptive steps are taken, suggesting that the new algorithm at least is not prone to this type of failure.

Acknowledgments. This material is based upon work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Number DE-SC0016513. Since April 2017, H. Anzt was supported by the "Impuls und Vernetzungsfond" of the Helmholtz Association under grant VH-NG-1241. The authors would like to thank Sherry Li and Meiyue Shao for their support with the SuperLU software package.

REFERENCES

[1] H. Anzt, E. Chow, J. Saak, and J. Dongarra, *Updating incomplete factorization preconditioners for model order reduction*, Numerical Algorithms, 73 (2016), pp. 611–630.
[2] H. Anzt, M. Gates, J. Dongarra, M. Kreutzer, G. Wellein, and M. Köhler, *Preconditioned Krylov solvers on GPUs*, Parallel Computing, 68 (2017), pp. 32 – 44.
[3] A. Basermann, *Parallel block ILUT/ILDLT preconditioning for sparse eigenproblems and sparse linear systems*, Numerical Linear Algebra with Applications, 7 (2000), pp. 635–648.
[4] M. Benzi, W. Joubert, and G. Mateescu, *Numerical experiments with parallel orderings for ILU preconditioners*, Electronic Transactions on Numerical Analysis, 8 (1999), pp. 88–114.
[5] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, *Time bounds for selection*, J. Comput. Syst. Sci., 7 (1973), pp. 448–461.
[6] E. Chow, H. Anzt, and J. Dongarra, *Asynchronous iterative algorithm for computing incomplete factorizations on GPUs*, in Proceedings of 30th International Conference, ISC High Performance 2015, Lecture Notes in Computer Science, Vol. 9137, J. Kunkel and T. Ludwig, eds., 2015, pp. 1–16.
[7] E. Chow and A. Patel, *Fine-grained parallel incomplete LU factorization*, SIAM Journal on Scientific Computing, 37 (2015), pp. C169–C193.
[8] E. Chow and Y. Saad, *Experimental study of ILU preconditioners for indefinite matrices*, Journal of Computational and Applied Mathematics, 85 (1997), pp. 387–414.
[9] E. Chow and Y. Saad, *ILUS: an incomplete LU preconditioner in sparse skyline format*, Internatinal Journal for Numerical Methods in Fluids, 25 (1997), pp. 739–748.
[10] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Transactions on Mathematical Software, 38 (2011), pp. 1:1–1:25.
[11] S. Doi, *On parallelism and convergence of incomplete LU factorizations*, Applied Numerical Mathematics, 7 (1991), pp. 417–436.
[12] I. S. Duff and G. A. Meurant, *The effect of ordering on preconditioned conjugate gradients*, BIT, 29 (1989), pp. 635–657.
[13] D. Hysom and A. Pothen, *A scalable parallel algorithm for incomplete factor preconditioning*, SIAM Journal on Scientific Computing, 22 (2001), pp. 2194–2215.
[14] M. T. Jones and P. E. Plassmann, *An improved incomplete Cholesky factorization*, ACM Transactions on Mathematical Software, 21 (1995), pp. 5–17.
[15] G. Karypis and V. Kumar, *Parallel threshold-based ILU factorization*, in 1997 ACM/IEEE Conference on Supercomputing, Nov 1997, pp. 1–24.
[16] N. Li, Y. Saad, and E. Chow, *Crout versions of ILU for general sparse matrices*, SIAM Journal on Scientific Computing, 25 (2003), pp. 716–728.
[17] X. S. Li and M. Shao, *A supernodal approach to incomplete LU factorization with partial pivoting*, ACM Transactions on Mathematical Software, 37 (2011), pp. 43:1–43:20.
[18] D. Lukarski, *Parallel Sparse Linear Algebra for Multi-core and Many-core Platforms - Parallel Solvers and Preconditioners*, PhD thesis, Karlsruhe Institute of Technology (KIT), Germany, 2012.
[19] N. Munksgaard, *Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients*, ACM Transactions on Mathematical Software, 6 (1980), pp. 206–219.
[20] NASA, *https://fun3d.larc.nasa.gov/*.
[21] E. L. Poole and J. M. Ortega, *Multicolor ICCG methods for vector computers*, SIAM Journal on Numerical Analysis, 24 (1987), pp. 1394–1417.
[22] Y. Saad, *ILUT: a dual threshold incomplete LU factorization*, Numerical Linear Algebra with Applications, 1 (1994), pp. 387–402.
[23] Y. Saad, *Iterative Methods for Sparse Linear Systems, 2nd Edition*, SIAM, Philadelphia, PA,

USA, 2003.

[24] S. Wang, E. de Sturler, and G. H. Paulino, *Large-scale topology optimization using preconditioned Krylov subspace methods with recycling*, International Journal for Numerical Methods in Engineering, 69 (2007), pp. 2441–2468.