SIAM J. SCI. COMPUT.
Vol. 39, No. 1, pp. A303–A332

ENHANCING PERFORMANCE AND ROBUSTNESS OF ILU PRECONDITIONERS BY BLOCKING AND SELECTIVE TRANSPOSITION*

ANSHUL GUPTA†

Abstract. Incomplete factorization is one of the most effective general-purpose preconditioning methods for Krylov subspace solvers for large sparse systems of linear equations. Two techniques for enhancing the robustness and performance of incomplete LU factorization for sparse unsymmetric systems are described. A block incomplete factorization algorithm based on the Crout variation of LU factorization is presented. The algorithm is suitable for incorporating threshold-based dropping as well as unrestricted partial pivoting, and it overcomes several limitations of existing incomplete LU factorization algorithms with and without blocking. It is shown that blocking has a three-pronged impact: it speeds up the computation of incomplete factors and the solution of the associated triangular systems, it permits denser and more robust factors to be computed economically, and it permits a trade-off with the restart parameter of GMRES to further improve the overall speed and robustness. A highly effective heuristic for improving the quality of preconditioning and subsequent convergence of the associated iterative method is presented. The choice of the Crout variant as the underlying factorization algorithm enables efficient implementation of this heuristic, which has the potential to improve both incomplete and complete sparse LU factorization of matrices that require pivoting for numerical stability.

Key words. sparse solvers, iterative methods, preconditioning, incomplete factorization, GMRES

AMS subject classifications. 65F08, 65F10, 65F50

DOI. 10.1137/15M1053256

1. Introduction.

This paper presents two highly effective techniques for enhancing both the performance and robustness of threshold-based incomplete LU (ILU) factorization.

It is well known that the nature of computations in a typical iterative method for solving sparse linear systems results in poor CPU-utilization on cache-based microprocessors. In contrast, static symbolic factorization and the use of supernodal [20] and multifrontal [17, 41] techniques typically enable highly efficient implementations of direct methods. The problem of poor CPU-utilization in iterative methods relative to the CPU-utilization of a well-implemented direct solver is evident in varying degrees for almost all preconditioners [21]. In the context of incomplete factorization, which has long been used to precondition Krylov subspace methods [1, 49], the primary culprits are indirect addressing and lack of spatial and temporal locality during both the preconditioner generation and solution phases. As a result, incomplete factorization runs at a fraction of the speed of complete factorization [29], and for many small and medium-sized practical problems, a direct solution turns out to be faster than an iterative solution, even when complete factorization requires significantly more memory and floating-point operations than incomplete factorization [21, 22]. Conventional methods to compute an incomplete factor much smaller than its complete counterpart can cost as much or more in run time as complete factorization. Therefore, only very

*Submitted to the journal's Methods and Algorithms for Scientific Computing section December 17, 2015; accepted for publication (in revised form) November 28, 2016; published electronically February 23, 2017.

http://www.siam.org/journals/sisc/39-1/M105325.html

†IBM T.J. Watson Research Center, Yorktown Heights, NY 10598 (anshul@us.ibm.com).

sparse incomplete factors can be practically computed, and the resulting preconditioner is often not effective for hard problems. Performing computations on dense blocks instead of individual elements can potentially help close the performance gap between complete and incomplete factorization of sparse matrices.

We present a practical algorithm that uses either natural or induced blocking of rows and columns to significantly increase the speed of incomplete factorization. This algorithm, first presented at the 2012 SIAM Conference on Applied Linear Algebra [24], is based on the Crout variation of LU factorization and supports threshold-based dropping for fill reduction as well as unrestricted partial pivoting for numerical stability.

1.1. Benefits of blocking. Blocked incomplete factorization has several benefits other than the obvious time saving in the preconditioner setup phase. First, denser and more robust factors, which would have been impractical to compute with conventional methods, can now be computed economically. Second, blocking in the factors improves spatial and temporal locality, and therefore the computation speed, when the preconditioner is used to solve triangular systems during the iterative phase. Finally, the ability to practically compute faster and denser incomplete factors results in a somewhat less obvious way to further improve the speed and robustness of the solution process. Restarted GMRES [50] is often the algorithm of choice for iteratively solving large sparse unsymmetric linear system arising in many applications. The algorithm is typically referred to as GMRES(m), where m is the restart parameter. The algorithm restricts the size of the subspace to m. After m iterations, it restarts while treating the residual after m iterations as the initial residual. This requires a minimum of $(m + 2)n$ words of storage for solving an $n \times n$ system. Although the exact relationship between m and the overall convergence rate of GMRES(m) is not well-understood, and a reliable a priori estimator for the optimum value of m for a given system does not exist, it is generally observed than increasing m up to a point tends to improve convergence. If the density of the preconditioner can be increased without an excessive run time penalty, then a high-density ILU paired with a small m in GMRES(m) can replace a combination of low-density ILU and GMRES(m) with a large m, without changing the overall memory footprint. In our experiments on a diverse suite of test problems, a value around 40 appeared to be a good choice for m. Our experiments indicate that it is better to use the memory for retaining additional entries in the ILU preconditioner than for increasing m significantly beyond this point. On the other hand, if memory is scarce, then a sparse preconditioner is preferable to reducing m significantly below this point. Such a trade-off between the value of m and the density of the ILU preconditioner is possible only if the latter can be increased without undue increase in the time to compute the preconditioner. This is enabled by blocking and is not likely to be possible with conventional ILU factorization without blocking.

1.2. Related work. Li, Saad, and Chow [38] proposed a Crout version of ILU factorization that they refer to as ILUC and highlighted several advantages of the approach over traditional row- or column-based ILU. Our block variant, which we will henceforth refer to as BILUC (block ILU in Crout formulation), follows a similar approach but incorporates several enhancements described in section 3.

Dense blocks have been used successfully in the past [6, 7, 18, 29, 31, 34, 35, 36, 39, 45, 48, 55] to enhance the performance of incomplete factorization preconditioners. Many of these block algorithms are designed for symmetric positive definite (SPD) systems, which we covered in an earlier publication [29]. Both complete and incomplete

factorization algorithms for unsymmetric sparse systems are inherently more complex than their SPD counterparts. In addition to possible structural unsymmetry, dynamic pivoting for numerical stability is the main source of this complexity. Pivoting can change the order of rows and columns and therefore the positions of potential nonzeros in the factors on the fly. In order to effectively search for a suitable pivot along both the rows and columns, at least a part of the unfactored portion of the matrix must be fully updated, which precludes a pure left-looking implementation. Therefore, in the remainder of this section, we restrict our attention to blocked implementations of ILU for unsymmetric systems only. Of these, only some are designed to handle dynamic pivoting. Furthermore, with a few exceptions like VBARMS [6] and SuperLU [39], these block algorithms have been applied to relatively simpler level-based incomplete factorization of matrices that have a naturally occurring block structure [42]. Dynamic dropping of factor entries based on a user-defined relative magnitude threshold is known to be significantly more effective than symbolic level-based dropping for general problems [2, 49]. Among the few threshold-based blocked ILU algorithms, VBARMS relies on a statically computed block structure [48] of the original matrix and, optionally, of one or more Schur complements. As a result, it is unable to perform partial pivoting, which can disrupt the static block structure. SuperLU can search for pivots within a supernode but relies on perturbing diagonal entries to avoid instability if a suitable pivot cannot be found within the confines of a supernode. The BILUC algorithm is able to use the classical complete factorization techniques of partial pivoting and delayed pivoting for numerical stability and does not require perturbations that can introduce extra error in factorization. In addition, unlike SuperLU's single-tier supernodes, BILUC uses a two-tier blocking scheme (section 3.1) that enables it to compute sparser factors with more precise dropping while maintaining the computational advantage of blocking.

To summarize, at the time of writing, to the best of our knowledge, the BILUC algorithm is the only blocked ILU factorization algorithm that employs threshold-based dropping, performs unrestricted pivoting, and uses dense blocks both in matrices in which such blocks occur naturally and in matrices that have poor or no block structure to begin with. It uses graph-partitioning for parallelism, nested-dissection ordering for fill reduction, and a combination of matching and selective transposition for enhancing numerical stability and reducing pivoting.

1.3. Selective transposition. In addition to using blocking to improve the speed of ILU factorization, we also introduce an effective and reliable heuristic to reduce extra fill-in due to pivoting and to improve convergence. Note that the solution x to a system of linear equations $Ax = b$ can be obtained either by factoring A as $L.U$, where L is unit lower triangular and U is upper triangular, or by factoring A^T into $U^T.L^T$, where U is unit upper triangular and L is lower triangular. Although there is some numerical advantage to using $A = LU$ factorization (because the first of the two solution phases uses a unit triangular matrix), it is often the case that the quality of $A^T = U^T L^T$ factorization is superior, making it more attractive. When A is sparse, there are two merit criteria for pivoting quality—pivot growth and extra fill-in due to pivoting. Both pivot growth and extra fill-in are affected by whether A or A^T is factored. In fact, when factorization is incomplete and is used for preconditioning, the impact is magnified because an inferior preconditioner can both increase the cost of each iteration and result in an increase in the number of iterations. We show that it is possible to make a reliable and inexpensive a priori determination of whether $A = LU$ or $A^T = U^T L^T$ factorization is likely to be superior. The Crout formulation,

in addition to its other benefits, permits an implementation that can seamlessly switch between factoring A or A^T, partly because U and L are treated almost identically.

1.4. Experimental setup. In the paper, wherever practical and useful, we present experimental results on matrices derived from real applications to demonstrate the benefits of the newly introduced techniques. All experiments were performed on a 4.2 GHz Power 6 system running AIX with 96 GB of memory. We used three Krylov subspace solvers in our experiments: restarted GMRES [50], BiCGStab [54], and TFQMR [19]. A maximum total of 1200 inner iterations of restarted GMRES [50] were permitted in any experiment. Iterations were terminated when both of the following conditions were satisfied: (a) the relative residual norm dropped below 10^{-8} and (b) a simple a posteriori relative error norm estimate, based on the change in the norm of the computed solution between iterations, fell below 10^{-4}. The second condition permits GMRES iterations to continue even if the relative residual norm has fallen below the specified threshold until a certain level of stability has been achieved in the computed solution. Our GMRES implementation adds approximate eigenvectors corresponding to a few smallest eigenvalues of the matrix to the subspace in order to mitigate the impact of restarting on convergence [43]. We found that this augmented version of restarted GMRES converges faster in general than conventional restarted GMRES for the same amount of memory. The software implementation of the BILUC preconditioner and the Krylov subspace solvers is a part of the Watson Sparse Matrix Package (WSMP) library [28], whose object code and documentation is available for testing and benchmarking at http://www.research.ibm.com/projects/wsmp.

1.5. Organization. The remainder of the paper is organized as follows. Section 2 contains an overview of the entire BILUC-based preconditioning scheme, including the preprocessing steps that precede the incomplete factorization. In section 3, we describe the BILUC algorithm in detail and present some experimental results to demonstrate the benefits of the blocking scheme. In section 4, we discuss a heuristic for boosting the robustness of BILUC (or any ILU) preconditioner and present experimental results to demonstrate its effectiveness. Section 5 includes a brief experimental comparison with some contemporary ILU-based preconditioners. Section 6 contains concluding remarks and possible directions of future investigations.

2. Overview of preconditioning scheme. Figure 1 gives an overview of the shared-memory parallel preconditioning scheme based on BILUC factorization. Note that the parallelization strategy is aimed at exploiting only a moderate degree of parallelism, suitable for up to 8–16 cores, depending on the problem size. A scalable distributed-address-space parallel ILU-based preconditioning would require additional algorithmic techniques [33, 37] and is being pursued as a continuation of the work reported in this paper. Therefore, parallelism is not the primary concern in this paper, and we will discuss parallelism minimally and only when necessary for a clear description of the overall algorithm. Nevertheless, BILUC's mild parallelism is important for its use in a practical industrial-strength solver like WSMP [28]. A shared-memory parallel implementation will still be relevant in a highly-parallel distributed scenario because we expect this algorithm to be used in each of the multithreaded MPI processes.

The overall solution process consists of three major phases, namely, *preprocessing*, *numerical factorization*, and *solution*. In this paper, we primarily focus on preprocessing and preconditioner generation. Standard preconditioned Krylov subspace solvers are used in the solution phase; however, the routines for the solution of triangular

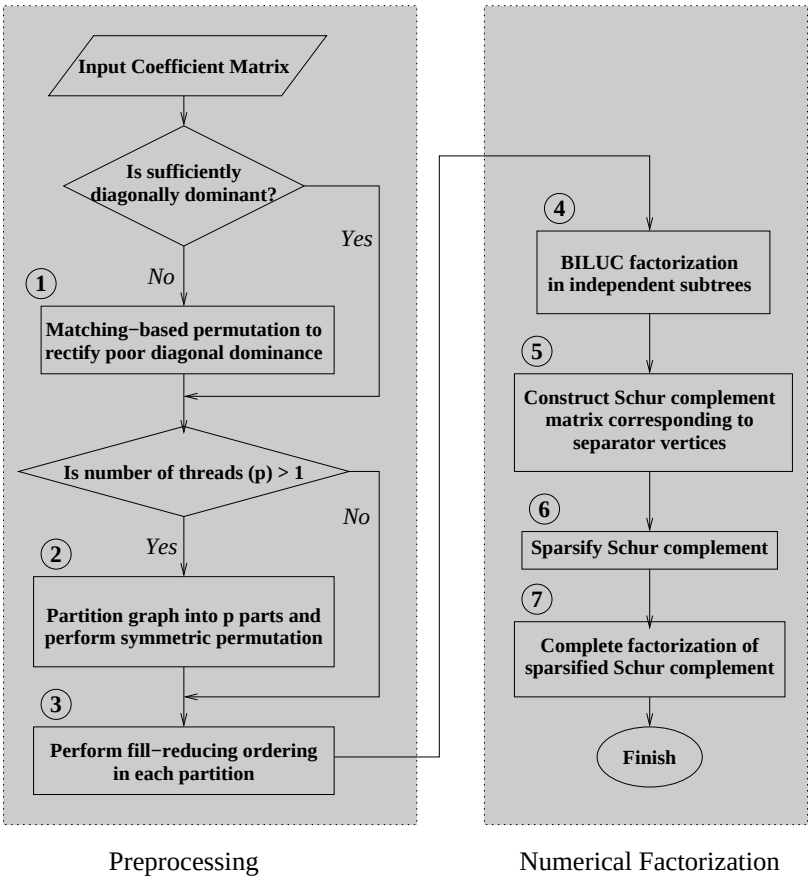

FIG. 1. *An overview of BILUC-based preconditioning.*

systems involving the incomplete factors for preconditioning work with blocks instead of individual nonzeros.

2.1. Preprocessing. The preprocessing phase consists of multiple steps, as shown in Figure 1.

First, if the matrix has poor diagonal dominance, then it is subject to permutation to either improve its diagonal dominance or to mitigate the impact of poor diagonal dominance. For matrices with less than 90% structural symmetry, we use a maximum weight bipartite matching (MWBP) [15, 30] to compute an unsymmetric permutation of rows or columns that maximizes the product of the diagonal entries. Thus, the permuted matrix that is factored is more diagonally dominant than the original one. If the original matrix is nearly structurally symmetric, then often the destruction of structural symmetry washes out the advantages of improved diagonal dominance from applying an MWBP-based unsymmetric permutation. Therefore, for matrices with a symmetric or nearly symmetric structure, we apply a symmetry-preserving heuristic to minimize pivoting and pivot-growth during factorization. This greedy heuristic matches each index corresponding to a small or zero diagonal with another index that could supply a suitable pivot during factorization. We permute the matrix such that the matched indices become consecutive in the permuted matrix. We also ensure

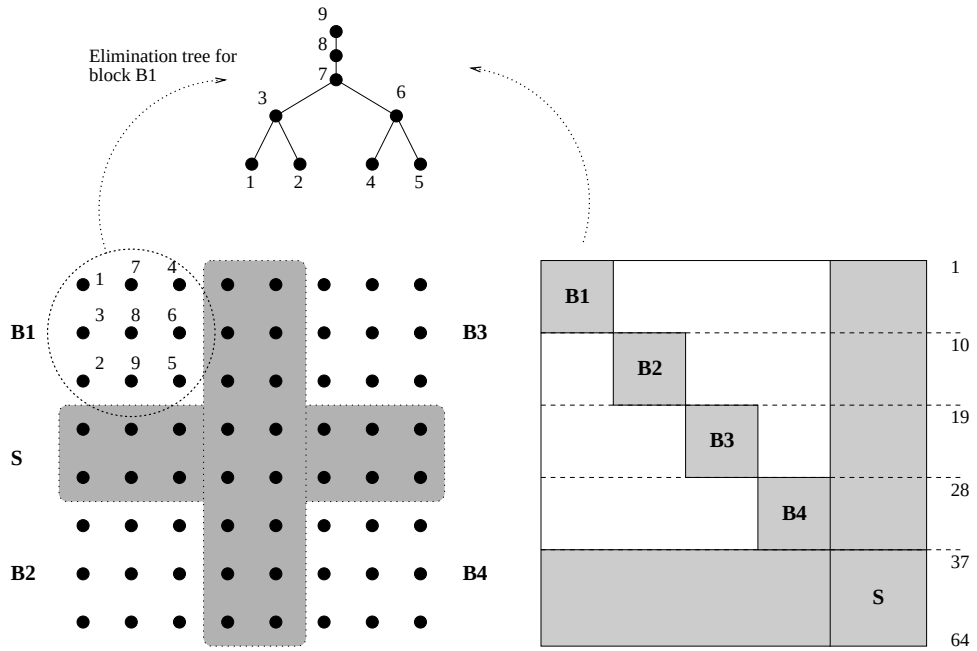

FIG. 2. *Correspondence between the partitioned (symmetrized) graph, the elimination tree, and the reordered matrix.*

that these indices stay consecutive during subsequent partitioning and reordering steps. We scan the columns of the $n \times n$ coefficient matrix A for those with diagonal dominance below a threshold δ. A column i would qualify if $\frac{|A(i,i)|}{\|A(*,i)\|} < \delta$. Then we look for an unmatched index j with the highest matching score, which is a weighted geometric mean of two quantities: (1) the ratio $\frac{|A(j,i)|}{\|A(*,i)\|}$ and (2) the ratio of the number of overlapping nonzero row indices in columns i and j to the total number of nonzero indices in columns i and j. The idea is that columns i and j will likely end up in the same block during factorization, and swapping $A(i,i)$ and $A(j,i)$ will satisfy the pivoting criterion. The reason we look for a high degree of overlap between the nonzero locations in columns i and j is that membership in the same block will require any zero in one of these columns in a row that has a nonzero in the other column to be stored as a 0-valued nonzero entry. Clearly, we want to minimize the fill-in due to such padding.

Next, an undirected graph of the matrix is constructed and partitioned [26] into p parts, where p is the number of parallel threads being used. The partitioning (Figure 2) seeks to divide the graph into p subgraphs of nearly equal size while minimizing the total number of edges crossing the partition boundaries. This enables each thread to independently factor block-diagonal submatrices corresponding to each partition. Similarly, portions of forward and backward substitutions corresponding to the interior vertices of the partitions can be performed independently by the threads when applying the preconditioner withing the chosen Krylov subspace solver. The partitioning is also useful for minimizing the interaction among threads during sparse matrix-vector multiplication steps of the solver.

After partitioning the overall matrix, a fill reducing ordering is computed for each of the submatrices corresponding to the p partitions. This step can be performed

independently, and in parallel, for each submatrix. Reverse Cuthill–McKee (RCM) [9, 14] ordering is used if the incomplete factors are expected to be relatively sparse (based on the dropping criteria); otherwise, nested dissection [20, 26] ordering is used. The choice of ordering is based on our and others' [16] observation that RCM generally performs better with low fill-in and nested dissection performs better with relatively higher fill-in.

Note that the initial coefficient matrix may undergo up to three permutations before numerical factorization. The first of these is a possibly unsymmetric permutation to address poor diagonal dominance, if needed. Next, a symmetric permutation is induced by graph-partitioning for enhancing parallelism, which is performed if more than one thread is used. Finally, the portion of the graph to be factored by each thread is reordered for fill reduction via another symmetric permutation. During the preprocessing phase, we construct single composite row and column permutation vectors that are applied to the right-hand side (RHS) and solution vectors in the solution phase. While the second and third permutations depend solely on the sparsity pattern of the matrix, the first one depends on the numerical values of the nonzero entries in it and, if performed, affects the two symmetric permutations that follow.

After all the permutations have been applied to the input matrix, the final step of the preprocessing phase is to construct elimination trees [40] from the structures of $B_i + B_i^T$, where B_i is the ith diagonal block corresponding to the ith domain ($1 \leq i \leq p$) of the coefficient matrix (Figure 2). An elimination tree defines the task and data dependencies in the factorization process.

Figure 2 shows the correspondence between the partitioned symmetrized graph, the elimination tree, and the reordered matrix for the case of four threads. Note that in the case of a single thread, there would be only one elimination tree corresponding to the entire matrix.

2.2. Numerical factorization. The numerical factorization phase has two main steps. The first step employs the BILUC algorithm independently on each of the domains that the graph corresponding to the coefficient matrix has been partitioned into during the preprocessing phase. Each thread follows its own elimination tree to factor its diagonal block using the BILUC algorithm, which is described in detail in section 3.

After all the rows and columns corresponding to the interior vertices of the partitions are factored in the first step, a Schur complement matrix is constructed corresponding to the remaining graph vertices that have edges traversing partition boundaries. This Schur complement (matrix S in Figure 2) is then further sparsified through a round of dropping, and the sparsified matrix is factored using a parallel direct solver [27]. The sparsification of the Schur complement matrix is necessary because it is factored by a direct solver through complete LU factorization without any further dropping. This sparsification can use the same drop tolerance as the preceding BILUC phase; however, we have observed that often a smaller threshold results in better preconditioners with only a slight increase in memory use. WSMP allows the user to define this threshold. We used half of the drop tolerance of the BILUC phase in our experiments, which is the WSMP default.

Note that if a single thread is used, then the entire incomplete factorization is performed in the first step by the BILUC algorithm. In this case, there is no Schur complement computation or the second factorization step. The proportion of the matrix factored in the second step increases as the number of threads increases because relatively more vertices of the graph belong to separators than to interior portions

of the partitions. The second step, which consists of complete factorization of the sparsified Schur complement, does not introduce a serial component to preconditioner computation. This factorization step, as well as the triangular solutions with respect to this factorization, are also multithreaded. Thus, the entire numerical factorization phase is parallel. However, the two factorization steps do have a synchronization point between them when the Schur complement matrix is assembled. The computation-intensive updates that contribute the numerical values to the Schur complement from each of the domains are still computed independently in parallel; only their assembly and sparsification to construct the input data structures for the second step are serial. Nevertheless, as noted earlier, this approach is suitable for a small or moderate number of threads only. This is because a larger portion of the matrix needs to be sparsified and the factored completely as the number of threads increases. This results in an increase in the overall number of entries that are stored, while reducing effectiveness of preconditioner.

3. The BILUC algorithm. The BILUC algorithm is at the heart of our overall incomplete factorization and solution strategy. While BILUC shares the Crout formulation with Li, Saad, and Chow's ILUC algorithm [38], most of its key features are different. BILUC can be expressed as a recursive algorithm that starts at the root of an elimination tree. The elimination tree serves as the task- and data-dependency graph for the computation. In the parallel case, each thread executes the algorithm starting at the root of the subtree assigned to it; in the serial case, there is only one tree. In the remainder of this section, we will drop the distinction between the subtree of a partition and tree of the whole matrix. Instead, we will discuss the algorithm in the context of a generic matrix A and its elimination tree.

Strictly speaking, for a matrix A with an unsymmetric structure, the task- and data-dependency graphs are directed acyclic graphs [25]. However, all the dependencies can be captured by using the dependency graphs corresponding to the structure of $A + A^T$. Such a dependency graph is the elimination tree [40]. Using a tree adds artificial dependencies and, in theory, may be less efficient than using a minimal dependency graph. On the other hand, the added simplicity and the reduction in bookkeeping costs afforded by the elimination tree more than make up for its suboptimality in the case of incomplete factorization, where the total amount of computation is significantly smaller than in complete LU factorization.

3.1. Block data structures. The BILUC algorithm uses two types of blocks that consist of contiguous rows and columns of the matrix corresponding to straight portions of the elimination tree. In these straight portions, all parents except the last (the one with the smallest index) have one child each. In Figure 2, vertices 7, 8, and 9 form such a straight portion. The two types of blocks are *assembly blocks* and *factor blocks*. The assembly blocks consist of large sections of straight portions of the elimination tree. Each assembly block typically consists of multiple smaller factor blocks. Figure 3 illustrates assembly and factor blocks and their relationship with the elimination tree.

Although assembly and factor blocks consist of multiple vertices in straight portions of the tree (and, therefore, multiple consecutive matrix rows and columns), in BILUC, we identify a block by its starting (smallest) index; i.e., block j refers to block starting at index j. Therefore, in our implementation, we store only the starting index as the sole identifier of an assembly block, along with supplementary information such as the size of the assembly block and the number and sizes of its constituent factor blocks.

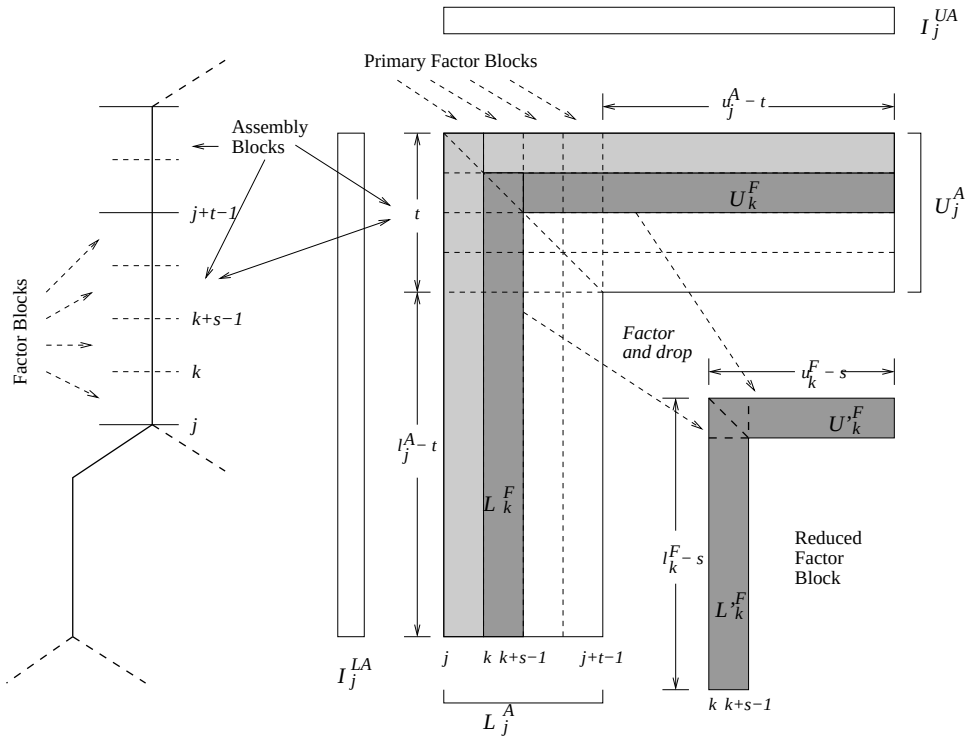

FIG. 3. *A section of an elimination tree and related assembly and factor blocks.*

The maximum size of the assembly blocks is user-defined. It is currently set to 40, but the algorithm chooses a value close to 40 for each assembly block such that it contains a whole number of factor blocks. Of course, this maximum applies only if the number of vertices in a straight portion of the elimination tree exceeds 40. Figure 3 shows one such assembly block and the typical BILUC dense data structure corresponding to it in detail. This block of size t starts at row/column index j. The rows and columns of this block correspond to contiguous vertices $j, j+1, \ldots, j+t-1$ in the elimination tree. In this assembly block, l_j^A is the number of nonzero entries in column j after the block has been fully assembled. This assembly requires contribution from factor blocks containing rows and columns with indices smaller than j. Note that due to sparsity, only a subset of such rows and columns will contribute updates to this assembly block. Similarly, u_j^A is the number of nonzeros in row j. The nonzeros in an assembly block can have arbitrary indices greater than j in the partially factored coefficient matrix, but the assembly blocks are stored in two dense matrices: (1) $l_j^A \times t$ matrix L_j^A with columns $j, j+1, \ldots, j+t-1$, and (2) $t \times u_j^A$ matrix U_j^A with rows $j, j+1, \ldots, j+t-1$. Note that the $t \times t$ diagonal block is a part of both L_j^A and U_j^A. In the actual implementation, this duplication is avoided by omitting this block from U_j^A. Furthermore, U_j^A is stored in its transposed form so that elements in a row reside in contiguous memory locations (assuming column-major format). In addition to L_j^A and U_j^A, the assembly block data structure includes two integer arrays, I_j^{AL} and I_j^{AU} of sizes l_j^A and u_j^A, respectively. These integer arrays store the global indices of the original matrix corresponding to each row of L_j^A and each column of U_j^A.

TABLE 1
BILUC data structures and the conventions used for their representation in this paper. Here $j \le k \le j+t-1$ *and* $1 \le s \le t$.

Data structure	Symbol	Starting index	Length	Width	Index array
Assembly block (L part)	L_j^A	j	l_j^A	t	I_j^{AL}
Assembly block (U part)	U_j^A	j	u_j^A	t	I_j^{AU}
Primary factor block (L part)	L_k^F	k	$l_j^A + j - k$	s	I_j^{AL}
Primary factor block (U part)	U_k^F	k	$u_j^A + j - k$	s	I_j^{AU}
Reduced factor block (L part)	$L_k'^F$	k	l_k^F	s	I_k^{FL}
Reduced factor block (U part)	$U_k'^F$	k	u_k^F	s	I_k^{FU}

The factor blocks are smaller subblocks of the assembly blocks. The factor blocks can either correspond to natural blocks in the coefficient matrix or can be carved out of assembly blocks artificially. Some applications yield matrices whose adjacency graphs have natural cliques. The clique vertices are assigned consecutive indices. For such matrices, each straight portion of the elimination tree would consist of a whole number of sets of vertices corresponding to these cliques. The groups of consecutive rows and columns corresponding to the cliques would then serve as natural factor blocks. For matrices without natural cliques, each assembly block is artificially partitioned into smaller factor blocks of a user-specified size, which is 4 by default in our implementation.

Figure 3(b) shows one such factor block, its relationship with its assembly block, and the BILUC dense data structures corresponding to typical assembly and factor blocks. The factor block shown in the figure is of size s and corresponds to row and column indices $k, k+1, \ldots, k+s-1$ of the coefficient matrix. The *primary factor block* is a part of the assembly block. It consists of two dense matrices, $(l_j^A + j - k) \times s$ matrix L_k^F with columns $k, k+1, \ldots, k+s-1$, and $s \times (u_j^A + j - k)$ matrix U_k^F with rows $k, k+1, \ldots, k+s-1$. L_k^F and U_k^F are submatrices of L_j^A and U_j^A, respectively.

After factorization, certain rows of L_k^F and columns of U_k^F in the primary factor block are dropped (section 3.3) based on the dropping criteria, and the result is a *reduced factor block*. In this reduced factor block, l_k^F is the number of nonzero entries remaining in column k after the primary factor block has been factored and rows of L_k^F with small entries have been dropped. Similarly, u_k^F is the number of nonzero entries remaining in row k after factorization and dropping in U_k^F. As a result of dropping, $l_k^F \le l_j^A + j - k$ and $u_k^F \le u_j^A + j - k$. Like the assembly and primary factor blocks, the resulting reduced factor block is stored in two dense matrices: (1) $l_k^F \times s$ matrix $L_k'^F$ with columns $k, k+1, \ldots, k+s-1$, and (2) $s \times u_k^F$ matrix $U_k'^F$ with rows $k, k+1, \ldots, k+s-1$. In the implementation, the transpose of $U_k'^F$ is stored. Accompanying integer arrays, I_k^{FL} and I_k^{FU} of sizes l_k^F and u_k^F, respectively, store the global indices of the original matrix corresponding to each row of $L_k'^F$ and each column of $U_k'^F$.

Table 1 summarizes the key BILUC data structures and the convention used in this paper to denote their generic sizes and indices. Note that the table's convention applies to the case when factor block k is a part of assembly block j. The lengths and the index arrays of the primary factor blocks given in this table are not valid for unrelated assembly and factor blocks.

3.2. Incomplete factorization with pivoting. The I arrays described in section 3.1 store the mapping between the indices of the global sparse coefficient matrix and the contiguous indices of the dense assembly and factor block matrices. Therefore, entries in the assembly and factor blocks can be referred to by their local indices during the factorization and dropping steps. Figure 4 shows the assembly and factor blocks corresponding to Figure 3(b) with local indices only.

LU factorization within an assembly block proceeds in a manner very similar to complete supernodal [42] factorization. Partial pivoting is performed based on a user-defined pivot threshold α. For most matrices that are not strictly diagonally dominant, pivoting plays an important role in maintaining stability of the ILU process and, in many cases, may be necessary even if a complete LU factorization of the same matrix is stable without pivoting [8].

When attempting to factor column i $(0 \leq i < t)$ in an assembly block, the BILUC algorithm scans this column of L_j^A to determine $g = \max_{i \leq m < t} |L_j^A(m,i)|$ and $h = \max_{t \leq m < l_j^A} |L_j^A(m,i)|$. Now g is the entry with the largest magnitude in column i within the lower triangular part of the pivot block of L_j^A and h is the entry with the largest magnitude in column i below the pivot block. If $g \geq \alpha h$, then a suitable pivot has been found in column i. The pivot element is brought to the diagonal position via a row interchange and the factorization process moves to column $i+1$. If a suitable pivot is not found in column i, then subsequent columns are searched. If a suitable pivot element is found within the pivot block, then it is brought to the diagonal position $L_j^A(i,i)$ via a column and a row interchange. It is possible to reach a stage where no suitable pivots can be found within the assembly block. In this

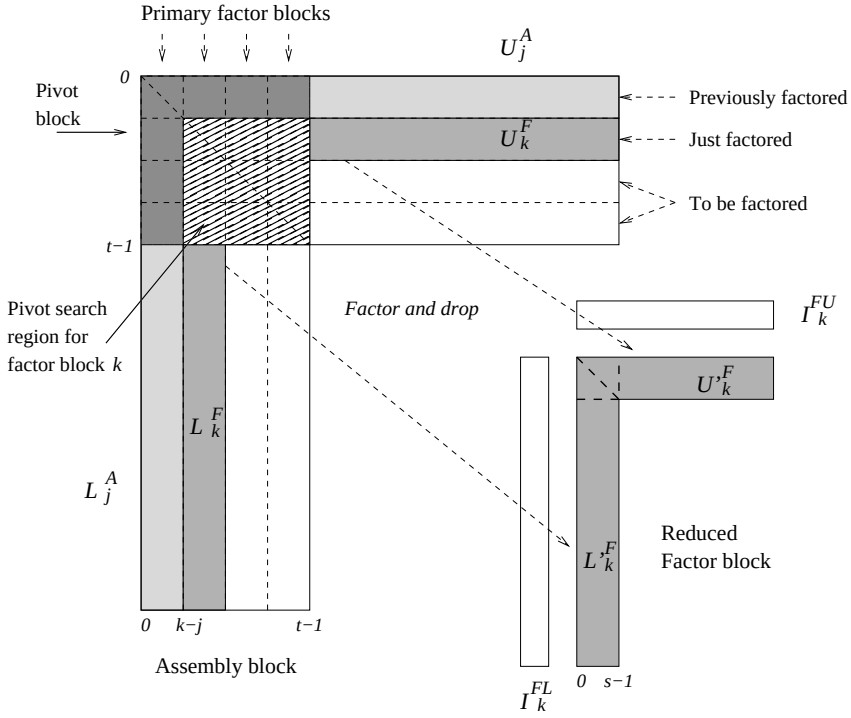

FIG. 4. *A typical assembly and factor block in the BILUC algorithm.*

situation, the unfactored rows and columns of the current assembly block are merged into its parent assembly block. Such delayed pivoting is commonly used in complete multifrontal factorization and increases the number of rows and columns eligible to contribute the pivot element. The reason is that some rows of L_j^A and columns of U_j^A with global indices greater than or equal to $j+t$ would become part of the pivot block in the parent assembly block. Any row-column pair can potentially be delayed until it reaches the root of the elimination tree, where all elements in the remaining unfactored portion of the matrix are eligible pivots.

Excessive delayed pivoting can increase fill-in and the computation cost of factorization. A smaller pivot threshold α can reduce the amount of delayed pivoting at the cost of higher pivot growth. In WSMP's BILUC implementation, we use two pivoting thresholds in order to strike a balance between minimizing growth and fill-in due to delayed pivoting. A secondary pivot threshold β is defined to be equal to 0.1α. The pivot search within the pivot block proceeds as described earlier with the threshold α. However, while searching for the first pivot that satisfies the α threshold, the algorithm keeps track of the largest magnitude element encountered in the pivot block that satisfies the relaxed β threshold. If the end of the pivot block is reached without any element satisfying the α threshold, then the largest entry satisfying the β threshold is used as a pivot, if such an entry is encountered at all. The algorithm resorts to delayed pivoting only if the β threshold too cannot be satisfied within the current pivot block. In our experiments, we observed that the fill-in resulting from this dual threshold pivoting strategy was close to the fill-in when only β was used as a single pivoting threshold. However, the quality of the preconditioner with the dual threshold preconditioner was significantly better and was only slightly worse than the case where only α was used as a single threshold.

Factorization in an assembly block takes place in units of factor blocks. After a primary factor block is fully factored, it undergoes sparsification via the dropping strategy discussed in section 3.3 to yield a reduced factor block. New column and row index sets I^{FL} and I^{FU}, which are subsets of the corresponding I^{AL} and I^{AU}, respectively, are built. Finally, the data structure for the reduced factor block is stored for future updates and for use in the preconditioning steps of the iterative solver. After all the factor blocks in an assembly block are factored and any delayed pivots are merged with the parent assembly block, the memory associated with the current assembly block is released.

3.3. Dropping and downdating. The factorization process described in section 3.2 is essentially the same as the one used is used in WSMP's general direct solver [27]. It is used in the BILUC algorithm in conjunction with the dropping and downdating strategy described below.

WSMP implements a dual dropping strategy of the form introduced by Saad [47]. Two user-defined thresholds τ and γ are used. Threshold τ determines which entries are dropped from the factor blocks based on their magnitudes. Threshold γ is the desired fill factor; i.e., the BILUC algorithm strives to keep the size of the incomplete factor close to γ times the number of nonzeros in the original matrix.

After factoring the rows and columns corresponding to a factor block, the BILUC algorithm performs a dropping and downdating step before moving on to the next factor block in the same assembly block. The $s \times s$ diagonal block is kept intact. Beyond the diagonal block, a drop score $dscr_L(i)$ is assigned to each row i of L_k^F and $dscr_U(i)$ to each column i of U_k^F in the primary factor block. Specifically,

$$dscr_L(i) = \frac{1}{s} \sum_{m=0,s-1} \left| L_k^F(i,m) \right|, s \le i < l_j^A - (k-j) \tag{3.1}$$

and

$$dscr_U(i) = \frac{1}{s} \sum_{m=0,s-1} \left| \frac{U_k^F(m,i)}{U_k^F(m,m)} \right|, s \le i < u_j^A - (k-j). \tag{3.2}$$

An entire row i of L_k^F is dropped if $dscr_L(i) < \tau$. Similarly, an entire column i of U_k^F is dropped if $dscr_U(i) < \tau$. Essentially, rows of L_k^F and columns of U_k^F in the primary factor block k are dropped if the average relative magnitude of these rows and columns is below drop tolerance τ. Note that, for computing the contribution of an element of a factored row or column to the drop score, we consider its magnitude relative to that of the corresponding diagonal entry. Since factorization is performed columnwise, each diagonal entry $L_k^F(m,m)$ in the context of (3.1) is 1; therefore, $dscr_L(i)$ is simply the average magnitude of entries in row i of L_k^F. On the other hand, $dscr_U(i)$ for column i of U_k^F is the average of the magnitude of $U_k^F(m,i)/U_k^F(m,m)$ for $0 \le m < s$. Other dropping strategies have been used for incomplete factorization. These include dropping based on the magnitude of an element relative to the 2-norm or ∞-norm of its column [47, 49] or dropping based on Munksgaard's criterion [44]. We found dropping based on magnitude relative to the corresponding diagonal entry to be slightly better than the other two on average for the problems in our test suite. Any of these dropping strategies are easy to incorporate in the BILUC framework. The BILUC algorithm, because of its Crout formulation, is also well-suited for dropping based on the growth of inverse of triangular factors [3].

After dropping rows and columns based on drop scores, the number of remaining rows and columns in the primary factor block may still exceed γ times the number of entries in row and column k of the original matrix. If this is the case, then additional rows and columns with the smallest drop scores are dropped from L_k^F and U_k^F.

Note that even though factorization is performed in steps of factor blocks of size s, the pivot search spans the entire remaining assembly block, which typically extends beyond the boundary of the current factor block. Therefore, columns of the assembly block beyond the boundary of the current factor block must be updated after each factorization step if entries from these columns are to serve as pivot candidates. Since small entries are not dropped until the entire factor block is factored, the columns of the assembly block beyond the boundary of the factor block may have been updated by entries that eventually end up being dropped. As Chow and Saad [8] point out, dropping after the updates results in a higher error in the incomplete factors than dropping before the updates. Therefore, BILUC needs to undo the effects of the updates by the dropped entries.

In WSMP's BILUC implementation, rows of L_k^F and columns of U_k^F that are eligible for dropping are first tagged. Then, the portion of assembly block L_j^A that has been updated by the factor block L_k^F is downdated by the rows of L_k^F that are tagged for dropping. This ensures that only those entries that are present in the final incomplete factors effectively participate in the factorization process. After the downdating step, reduced factor block $L_k'^F$ is constructed from the primary factor block L_k^F by copying only the untagged rows from the latter to the former. The $U_k'^F$ matrix of the reduced factor block is constructed similarly from U_k^F. Index arrays I_k^{FL} and I_k^{FU} are constructed as subsets of I_j^{AL} and I_j^{AU}, respectively, containing indices

of the rows and columns of the primary factor block retained in the reduced factor block. The reduced factor block comprising L'^F_k, U'^F_k, I^{FL}_k, and I^{FU}_k is stored as part of the incomplete factor and for future updates of the ancestral supernodes.

Dropping entire rows and columns of factor blocks instead of individual entries has an impact on both the size and the quality of the preconditioner because the drop tolerance is applied inexactly in BILUC. We look at this impact experimentally in section 3.6. For the same drop tolerance τ, if fill factor γ is disregarded, then BILUC results in slightly larger factors than ILUC without blocking. The reason for the extra nonzeros in the incomplete factors is that the block columns may retain a substantial number of zero or small entries, which are discarded in the nonblocked version of incomplete factorization with the same drop tolerance. Small or zero entries, for example, can be retained in a row of a supernode that has a drop score greater than the drop tolerance due to a single large entry. Similarly, BILUC may drop entries whose magnitude exceeds the drop tolerance by a factor up to s because a row or column in a primary factor block with a single entry of magnitude $s\tau$ will be dropped if all other entries in that row or column are zero.

The use of small factor blocks (4 or the size of the natural clique, whichever is larger) is important for maintaining the precision of dropping. A small s ensures that large entries are not dropped and too many small entries are not retained. In contrast, large assembly blocks (up to size 40 by default) provide a bulk of the benefit of dense computations and permit pivot search over a larger area of the matrix to minimize costly delayed pivoting. This is the rationale behind BILUC's two-tier blocking scheme.

3.4. Constructing assembly blocks. As mentioned previously, BILUC is a recursive algorithm. It follows the elimination tree in depth-first order. So far, we have described the various steps involved in processing an assembly block, i.e., factorization with pivoting, dropping, downdating, and constructing the reduced factor blocks. In this section, we describe how new assembly blocks are constructed using previously computed reduced factor blocks.

Once the elimination tree is constructed, the global indices of the pivot block of each assembly block are known; for example, the pivot block of the assembly block shown in Figure 3 includes indices $j, j+1, \ldots, j+t-1$. However, the sizes l^A_j and u^A_j and the indices in I^{AL}_j and I^{AU}_j are determined only when the assembly block is actually constructed, which happens just before it is factored. The reason is that the indices in I^{AL}_j and I^{AU}_j depend on the indices of the reduced factor blocks that update the assembly block j, and the sizes and indices of these reduced factor blocks cannot be predicted in advance due to dynamic pivoting and dropping.

If the starting index j of an assembly block is a leaf in the elimination tree, then the assembly block simply consists of the corresponding rows and columns of the coefficient matrix. I^{AL}_j is simply the union of row indices of columns $j, j+1, \ldots, j+t-1$ of A and I^{AU}_j is the union of column indices of rows $j, j+1, \ldots, j+t-1$ of A. All entries in L^A_j and U^A_j that are not in A are filled with with zeros.

Each assembly block that does not start at a leaf has a linked list of contributing reduced factor blocks associated with it. At the beginning of the BILUC algorithm, all lists are empty. Consider a factor block k that is a part of an assembly block j of size t. Let $v = I^{FL}_k(q_1)$ be the smallest index greater than or equal to $j+t$ in I^{FL}_k and $w = I^{FU}_k(r_1)$ be the smallest index greater than or equal to $j+t$ in I^{FU}_k. When the reduced factor block k is created, then it is placed in the linked list of the ancestor assembly block whose pivot block contains the index $z = \min(v, w)$. Here z is the smallest index greater than the pivot indices of assembly block j. The ancestor

assembly block is the first assembly block that will be updated by factor block k. Let the starting index of the ancestor block be j_1 and its size be t_1. Then $j_1 \le z < j_1 + t_1$.

When the BILUC process reaches a nonleaf assembly block j_1 (i.e., j_1 is ancestor of at least one other assembly block) of size t_1, the first step is to construct $I_{j_1}^{AL}$ and $I_{j_1}^{AU}$. $I_{j_1}^{AL}$ is the union of (1) $\{j_1, j_1+1, \ldots, j_1+t_1-1\}$, (2) all indices greater than or equal to $j_1 + t_1$ in columns $j_1, j_1+1, \ldots, j_1+t_1-1$ of A, and (3) all indices greater than or equal to j_1 in I_k^{FL} for all k such that the reduced factor block k is in the linked list of assembly block j_1. Similarly, $I_{j_1}^{AU}$ is the union of (1) $\{j_1, j_1+1, \ldots, j_1+t_1-1\}$, (2) all indices greater than or equal to $j_1 + t_1$ in rows $j_1, j_1+1, \ldots, j_1+t_1-1$ of A, and (3) all indices greater than or equal to j_1 in I_k^{FU} for all k such that the reduced factor block k is in the linked list of assembly block j_1.

Once $I_{j_1}^{AL}$ and $I_{j_1}^{AU}$ have been constructed, the size of assembly block j_1 is known. It is then allocated and is populated with corresponding entries from A, while the remaining entries are initialized to zeros. Next, contribution matrices from each of the reduced factor blocks in its linked list are computed and are subtracted from $L_{j_1}^A$ and $U_{j_1}^A$. Figure 5 shows how the contribution matrices are computed from a reduced factor block. Let q_2 and r_2 be such that $I_k^{FL}(q_2)$ is the first index in I_k^{FL} greater than or equal to $j_1 + t_1$, and $I_k^{FU}(r_2)$ is the first index in I_k^{FU} greater than or equal to $j_1 + t_1$. As shown in the figure, q_1, q_2 and r_1, r_2 identify the portions of $L_k'^F$ and $U_k'^F$ that would be multiplied to create the contribution matrices to be subtracted from $L_{j_1}^A$ and $U_{j_1}^A$. The darker shaded portion of $L_k'^F$ is multiplied with the lighter shaded portion of $U_k'^F$ to compute the contribution matrix for $U_{j_1}^A$. Similarly, the darker shaded portion of $U_k'^F$ is multiplied with the lighter shaded portion of $L_k'^F$ to compute the contribution matrix for $L_{j_1}^A$. In general, the global row and column indices associated with the contribution matrices are subsets of the index sets $I_{j_1}^{AL}$ and $I_{j_1}^{AU}$ associated with $L_{j_1}^A$ and $U_{j_1}^A$, respectively. Therefore, the contribution matrices are expanded to align their global row and column index sets with $I_{j_1}^{AL}$ and $I_{j_1}^{AU}$ before subtraction.

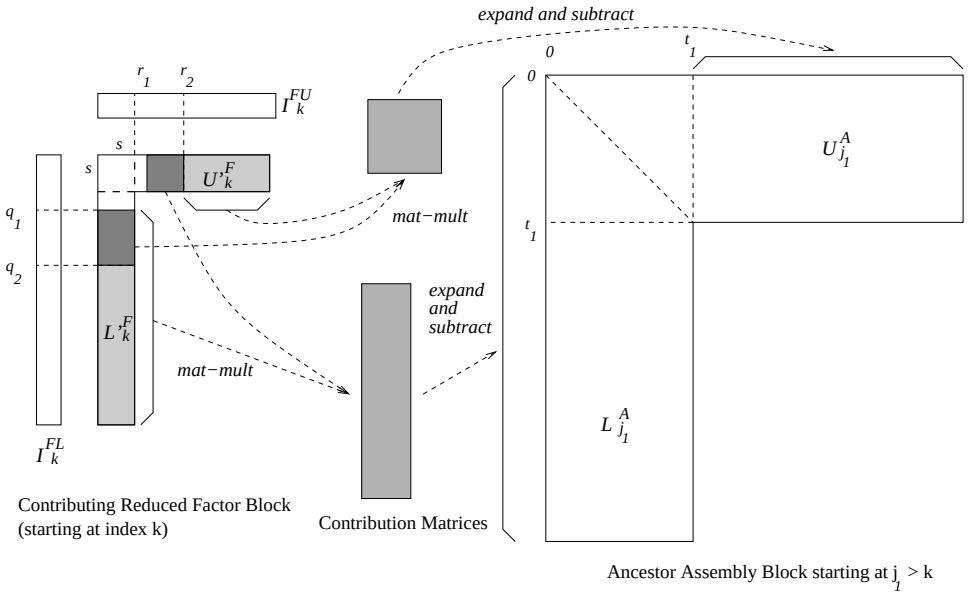

FIG. 5. *Reduced factor block k updating its first ancestor assembly block* j_1.

After extracting the contribution from reduced factor block k, if both q_2 and r_2, as described above, exist (i.e., both I_k^{FL} and I_k^{FU} have at least one index greater than or equal to $j_1 + t_1$), then the reduced factor block is placed in the linked list of assembly block j_2 of size t_2 such that $z_2 = \min(I_k^{FL}(q_2), I_k^{FU}(r_2))$ lies in the pivot block of j_2. Assembly block j_2 would be the second assembly block to be updated by factor block k. This update would take place when the BILUC process reaches j_2 during its DFS traversal of the elimination tree. Figure 6 illustrates this update process, which is very similar to the first update by factor block k shown in Figure 5, except different portions of $L_k'^F$ and $U_k'^F$ are used.

When all the reduced factor blocks in the linked list of the assembly block being constructed are processed, then the assembly block is ready for the factorization process described in section 3.2. Some of these contributing factor blocks end up in the linked lists of other assembly blocks. For other contributing blocks, this may be the last assembly block. After factorization, dropping, and downdating, the current assembly block yields its own fresh set of reduced factor blocks which are placed in the appropriate linked lists.

3.5. BILUC—putting it all together. Figure 7 summarizes the BILUC algorithm whose various steps are described in detail in sections 3.2–3.4. The recursive algorithm has the starting index of an assembly block as its primary argument. It is first invoked simultaneously (in parallel) in each domain with the root assembly block of the elimination tree corresponding to the respective submatrix. The starting index of an assembly block either is a leaf of the elimination tree or has one or more children. It has exactly one child in the straight portion of the elimination tree, and when it has more than one child, then the tree branches.

The first key step in the BILUC algorithm for a given assembly block j is to recursively invoke itself for all the children assembly blocks of j. It then assembles the row and column index sets I_j^{AL} and I_j^{AU}, followed by actually constructing the

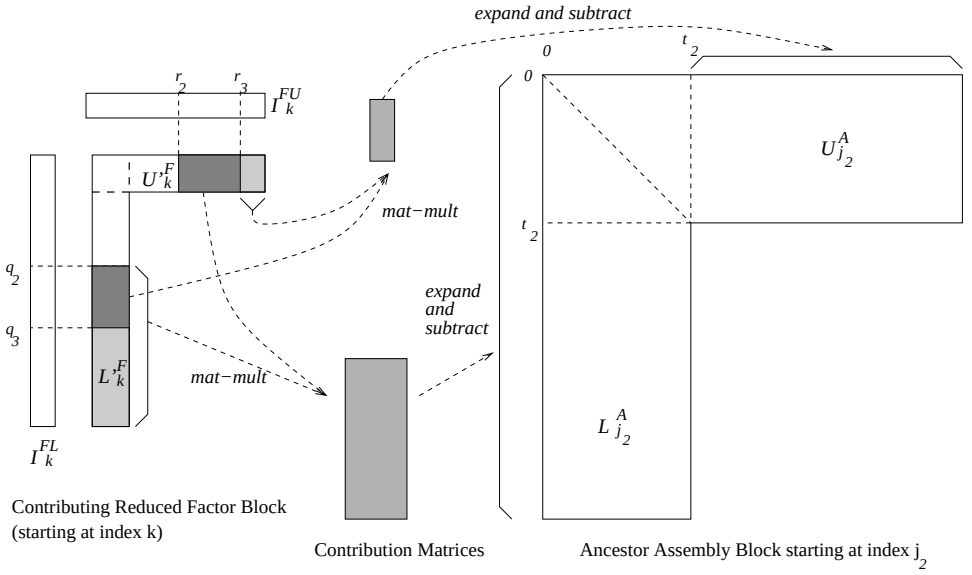

FIG. 6. *Reduced factor block k updating its second ancestor assembly block j_2.*

```
1.  begin function BILUC (j)
2.     for each assembly block i that is a child of assembly block j
3.        BILUC (i);
4.     end for
5.     Initialize row and col index sets I_j^{AL} and I_j^{AU};
6.     Allocate L_j^A and U_j^A based on sizes of I_j^{AL} and I_j^{AU};
7.     Initialize L_j^A and U_j^A by copying entries from A;
8.     for each k', such that reduced factor block k' is in j's linked list
9.        Remove k' from j's linked list;
10.       Compute contribution matrices from L'^F_{k'} and U'^F_{k'} and update L_j^A and U_j^A;
11.       Insert k' in linked list of its next assembly block, if any;
12.    end for
13.    for each k, such that k is a primary factor block in j
14.       Perform factorization with pivoting on block k;
15.       Create reduced factor block k after dropping and downdating;
16.       Let v be the smallest index ≥ j + t in I_k^{FL} and I_k^{FU};
17.       Insert k in linked list of assembly block containing index v in its pivot block;
18.    end for
19.    Merge unfactorable portions of L_j^A and U_j^A in j's parent assembly block.
20.    return;
21. end function BILUC
```

FIG. 7. *Outline of the recursive BILUC algorithm. Invoking BILUC with the root assembly block of an elimination subtree computes a block ILU factorization of the submatrix associated with the subtree.*

assembly block L_j^A and U_j^A, as described in section 3.4. L_j^A and U_j^A are constructed from the entries of A that lie within these blocks and from the contribution matrices from reduced factor blocks of some of j's descendants in the elimination tree. A reduced factor block k' contributes to assembly block j if and only if $I_{k'}^{FL}$ or $I_{k'}^{FU}$ contains at least one index within $j, j+1, \ldots, j+t-1$. All such reduced factor blocks would have already been placed in the linked list of assembly block j by the time the BILUC algorithm reaches this stage. After the contribution from reduced factor block k in the linked list of assembly block j is used in the construction of L_j^A and U_j^A, the factor block is placed in the linked list of the next assembly block that it will contribute to, if such an assembly block exists. When contributions from all factor blocks in the assembly block j's linked list have been absorbed, the assembly block is factored in units of its own factor blocks. The end result of this process, described in detail in sections 3.2 and 3.3, is a set of fresh reduced factor blocks. These are placed in the linked lists of their respective first target assembly blocks that they will update. Finally, any unfactorable portions of L_j^A and U_j^A where a suitable pivot could not be found are merged with the assembly block that is the parent of j in the elimination tree. This completes the BILUC process for a given assembly block identified by its first index j.

3.6. Experimental results. We now describe the results of our experiments with the BILUC algorithm highlighting the impact of blocking and block sizes on memory consumption, convergence, factorization time, solution time, and overall performance. Table 2 lists the matrices used in our experiments. Most of these matrices are from the University of Florida sparse matrix collection [10]. The remaining ones

TABLE 2
Test matrices and their basic information.

	Matrix	Dimension	Nonzeros	Application
1.	Jacobian	137550	9050250	Circuit simulation
2.	af23560	23560	484256	CFD
3.	bbmat	38744	1771722	CFD
4.	ecl32	51993	380415	Semiconductor device simulation
5.	eth-3dm	31789	1633499	Structural engineering
6.	fullJacobian	137550	17500900	Circuit simulation
7.	matrix-3	125329	2678750	CFD
8.	mixtank	29957	1995041	CFD
9.	nasasrb	54870	2677324	Structural engineering
10.	opti_andi	41731	542762	Linear programming
11.	poisson3Db	85623	2374949	3D Poisson problem
12.	venkat50	62424	1717792	Unstructured 2D Euler solver
13.	xenon2	157464	3866688	Material science
14.	matrix12	2757722	38091058	Semiconductor device simulation
15.	matrixTest2_10	1035461	5208887	Semiconductor device simulation
16.	seid-cfd	35168	14303232	CFD

are from some of the applications that currently use WSMP's general sparse direct solver [27]. The experimental setup is described in section 1.4. Recall that we use Morgan's GMRES variant [43] that augments the saved subspace with approximate eigenvectors corresponding to a few smallest eigenvalues of the matrix. For our implementation, GMRES(k,l) denotes restarted GMRES with at least k subspace vectors and at most l eigenvectors. The total space allocated for subspace and approximate eigenvectors is $m = k + 2l$. The reason why l eigenvectors require $2l$ space is that each eigenvector can have a real and an imaginary part. Unused eigenvector space is used for storing additional subspace vectors; therefore, the actual number of inner GMRES iterations before a restart is triggered is between k and m. The default for the maximum number l of approximate eigenvectors is set to $\sqrt{k} - 1$ in WSMP.

For our first experiment, we solved systems using the matrices in Table 2 and RHS vectors containing all ones. Other than the maximum factor block sizes, default values of all other parameters were used. We first turned blocking off by setting the maximum factor block size to 1, which would result in an algorithm similar to Li, Saad, and Chow's ILUC algorithm [38]. We then solved the systems with the maximum factor block size set to 2, 3, 4, and "unlimited." In the "unlimited" case, natural cliques are used as factor blocks if they contain more than 4 vertices; otherwise, factor blocks of size 4 are used. In WSMP, the assembly block size is chosen to be the smaller of 40 and 10 times the size of the maximum factor block. Note that in the case of maximum factor block size of 1, which we use as our ILUC emulation, WSMP would still use assembly blocks of size 10 and is likely to be somewhat faster than the original unblocked ILUC algorithm. As a result, BILUC's performance advantage over conventional unblocked incomplete factorization may be even bigger than what our experiments show.

We use GMRES(100,9) with at least 100 subspace vectors and at most 9 approximate eigenvectors. We observed the preconditioner generation (incomplete factorization), solution (GMRES iterations), and the total time, as well as factorization fill-in and the number of GMRES iterations. For each of these metrics, we computed the ratio with respect to the unblocked case. Figure 8 shows the average of these ratios over the 16 test matrices.

Blocking has the most significant impact on preconditioner generation time, as shown by the blue bars in Figure 8. During incomplete factorization, blocking helps

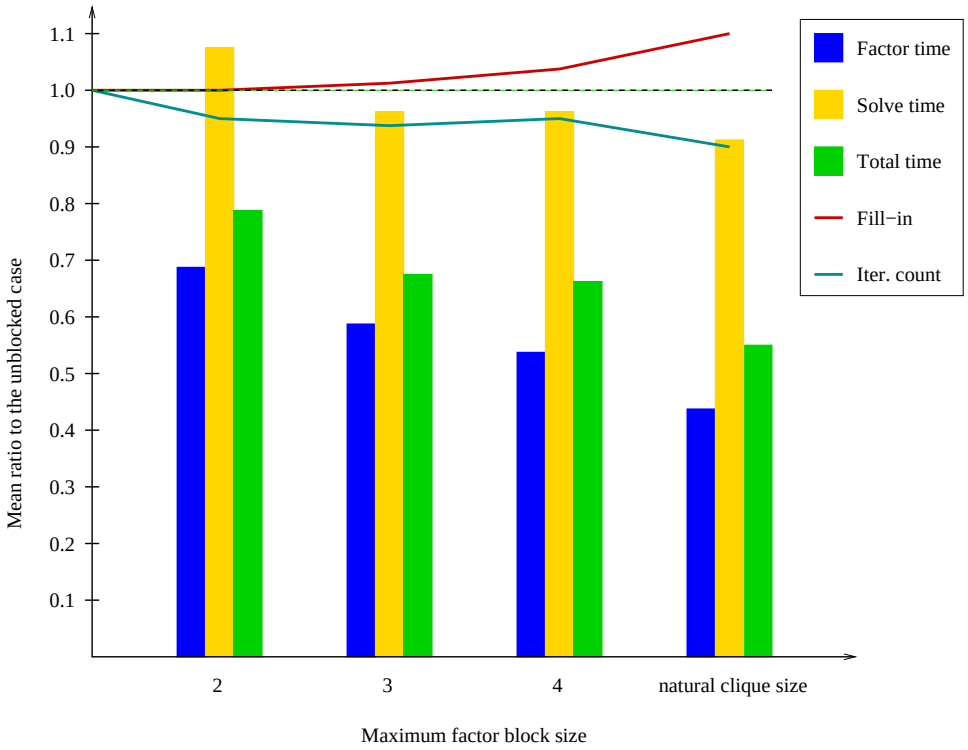

FIG. 8. *Some performance metrics as functions of maximum block size relative to the unblocked case.*

in two ways. First, the use of blocks reduces the overhead due to indirect addressing because a single step of indirect addressing affords access to a whole block of nonzeros instead of a single element. Since a static symbolic factorization cannot be performed for incomplete factorization, updating a sparse row (column) with another requires traversing the index sets of both rows (columns). Consider the updating of an assembly block of width t by a reduced factor block of width s. This would require a single traversal of a pair of sets of indices. The same set of updates in a conventional nonblocked incomplete factorization can require a traversal of up to st pairs of index sets because each of the t rows and columns of the assembly block could potentially be updated by all s rows and columns of the factor block. The second benefit of blocking is that it permits the use of higher level BLAS [12, 13], thus improving the cache efficiency of the implementation. Note that when we refer to the use of higher level BLAS, we do not necessarily mean making calls to a BLAS library. Typically, the blocks in sparse incomplete factors are too small for BLAS library calls with high fixed overheads to be efficient. The key here is to use the blocks to improve spatial and temporal locality for better cache performance, which we achieve through our own implementation of lightweight BLAS-like kernels, instead of making calls to an actual BLAS library. Figures 5 and 6 show how matrix-matrix multiplication is the primary computation in the update process.

Blocking has a less dramatic effect on GMRES iteration time. Some gains in efficiency are offset by increase in operation count due to slightly larger factors that result from blocking. On the other hand, the total iteration count tends to drop slightly as blocks get larger because more nonzeros are stored. In our experiments,

the net effect of all these factors was that solution time increased for very small blocks, for which the bookkeeping overhead associated with blocking seems to have more than offset the small gains. For larger block size, the solution time fell. The overall time to solve the systems recorded almost 50% reduction on an average in our test suite.

Within limits, there is a trade-off between incomplete factorization time and the iterative solution time in a typical preconditioned Krylov method. Denser, costlier preconditioners would generally result in fewer iterations, and vice versa, as long as the added density reduces error due to dropping and the increase in solution time with respect to the denser preconditioner does not dominate the time saved due to fewer iterations. There would be an optimum preconditioner density (and a corresponding optimum drop tolerance) for which the total of factorization and solution time would be minimum. Since blocking can significantly reduce incomplete factorization time, but has a more muted effect on GMRES iterations' time, it has the potential to change the optimum drop tolerance and preconditioner density for achieving the least overall factorization and solution time.

In our next set of experiments, we compare ILUC and BILUC for different drop tolerance values. For these experiments, we chose seven drop tolerance (τ) values in the range of 10^{-2} and 10^{-5}. For each of these drop tolerance values, the bars in Figure 9 show the average of the 16 matrices' incomplete factorization times normalized with respect to the ILUC factorization time with $\tau = 10^{-3}$ on the logarithmic scale on the left. The line graphs show the total factorization and solution time relative to the total ILUC time with $\tau = 10^{-3}$ on the linear scale on the right. The numbers of cases out of 16 in which the algorithms failed to converge are shown at the top.

The results show that BILUC factorization is typically two or three times faster than ILUC and its speed advantage over ILUC is more pronounced at smaller drop tolerances. The figure also shows that BILUC can typically use a drop tolerance that is at least an order of magnitude lower and still compute an incomplete factorization

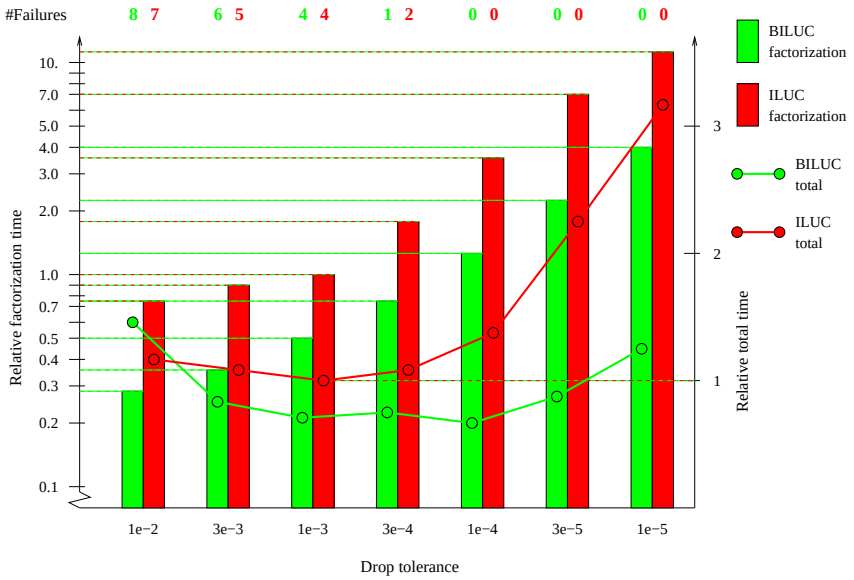

FIG. 9. *Relative factorization times of ILUC and BILUC algorithms as functions of drop tolerance.*

in about the same time as ILUC. The impact of this can be seen on the robustness and the total of factorization and solution time of the preconditioners. A drop tolerance of 10^{-3} results in the best overall time for ILUC with a 1200-iteration limit on restarted GMRES. However, it can solve only 75% of the problems in the test suite with this drop tolerance. Both ILUC and BILUC are able to solve 100% of the problems with $tau = 10^{-4}$. However, ILUC is about 50% slower at this value of τ compared to $tau = 10^{-3}$. On the other hand, for our test suite, $tau = 10^{-4}$ coincides with the smallest total time for BILUC, which is more than twice as fast as ILUC for this value of τ.

Using smaller drop tolerances, which BILUC is well-suited for, can reduce the number of iterations and help some hard-to-solve problems converge. However, reducing the drop tolerance increases the size of incomplete factors and total memory consumption. The BILUC algorithm's ability to efficiently work with small drop tolerances has interesting implications for restarted GMRES [50], which is most often the Krylov subspace method of choice for solving general sparse linear systems. Consider a sparse $n \times n$ coefficient matrix A with nnz_A nonzeros. Let d be the average row-density of A, i.e., $nnz_A = dn$. Let γ be the effective fill factor, i.e., the number of nonzeros in the incomplete factor, $nnz_F = \gamma nnz_A = \gamma dn$. The number of words of memory M required to solve a system $Ax = b$ using GMRES(m) preconditioned with the incomplete factor of A, where m is the restart parameter (or the number of subspace vectors stored) can roughly be expressed as

$$M = n \times (K + m + d + \gamma d). \tag{3.3}$$

Here K is a small constant, typically around 3 in most implementations. Now the drop tolerance can be changed to obtain a denser or a sparser incomplete factorization with a different effective fill factor of γ' while keeping the overall memory consumption M unchanged if we alter the restart parameter of GMRES to m' such that

$$n \times (K + m + d + \gamma d) = n \times (K + m' + d + \gamma' d)$$

or

$$m' = m + d(\gamma - \gamma'). \tag{3.4}$$

Similarly, if we change the restart parameter from m to m', then we can keep the overall memory constant by changing the effective fill factor to γ' given by

$$\gamma' = \gamma + (m - m')/d. \tag{3.5}$$

We conducted two sets of experiments to study the trade-off between fill-in and the restart parameter. For these experiments, we measured the number of iterations and factorization, solve, and total times while adjusting the drop tolerance (and therefore, the effective γ) and the GMRES restart parameter according to (3.4) and (3.5) so that the total memory remained unchanged. For these experiments, we control the fill-in by changing only the drop tolerance. The total memory allocated for subspace and approximate eigenvectors is $m = k + 2l$, and this is the m that corresponds to the one in (3.3)—(3.5).

In addition to GMRES with different restart parameters, our experiments also include short recurrence methods such as BiCGStab [54] and TFQMR [19] that do not store the subspace explicitly and use only a small fixed number of vectors for working storage. We use $m = 3$ for BiCGStab and TFQMR. For one set of experiments, we

determined the drop tolerance that (roughly) led to the best overall time for BiCGStab for each matrix and measured the corresponding effective fill factor. We then ran TFQMR with the same preconditioner that BiCGStab used and GMRES(k,l) for six different restart values m' while adjusting the drop tolerance such that γ' satisfied (3.5). The results were normalized with respect to BiCGStab results. Figure 10 shows a plot of the geometric means of these normalized values over all 16 test cases. The results indicate that denser preconditioners combined with small restart parameter values, or even a short-recurrence method, resulted in significantly faster convergence and overall solution compared to the combination of sparser preconditioners and larger restart parameter values.

For our next experiment, we started with the best drop tolerance for GMRES (100,9) and used the corresponding effective fill factor as our base γ. We then reduced restart parameter and the drop tolerance it such that the effective fill factor γ' satisfied (3.5). The results of this experiment are shown in Figure 11. Once again, the results indicate that investing in a denser ILU preconditioner rather than subspace vectors is a better use of memory. In fact, BiCGStab seems to work almost as well as GMRES when memory for storing the subspace vectors is diverted to create denser ILU preconditioners. Among GMRES variants, moderate restart values dramatically outperform large restart values when memory is taken into account.

In both sets of constant-memory experiments, after bottoming out at $m = 46$ (i.e., $k = 36, l = 5$), GMRES iteration count starts increasing as more memory is diverted to the incomplete factors from subspace vectors to the factors. This is probably

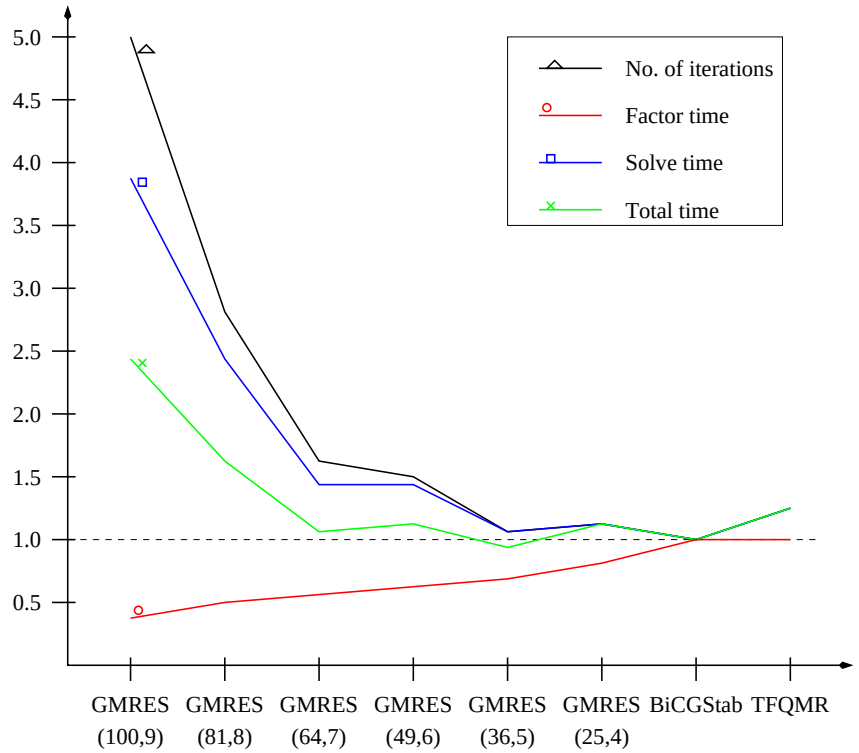

FIG. 10. *Relative performance of restarted GMRES variants, BiCGStab, and TFQMR under constant total preconditioner and solver memory.*

because very small subspaces are ineffective in advancing the GMRES algorithm even with good preconditioners. As a result, BiCGStab usually outperforms GMRES with a small restart parameter in the constant-memory scenario.

4. Selective transposition. The BILUC algorithm performs threshold partial pivoting, as described in section 3.2. The worst-case pivot growth for Gaussian elimination with partial pivoting is 2^{n-1} for an $n \times n$ dense matrix [32]. For incomplete factorization of a sparse matrix, the worst case would be 2^{l-1}, where l is the maximum number of nonzeros in a row of L or column of U. When using a threshold $\alpha < 1$, the worst-case pivot growth could be higher by another factor of $\frac{1}{\alpha^{l-1}}$. Recall that threshold α permits an entry $L_j^A(i,i)$ in the diagonal position as long as its magnitude is greater than α times the largest magnitude of any entry in column i of L_j^A. In practice, factoring a matrix from a real problem is unlikely to result in pivot growth anywhere close to the worst-case bounds; however, it is reasonable to expect that a smaller ratio of $|L_j^A(i,i)|$ to the largest magnitude of any entry in column i of L_j^A is likely to result in higher growth. This ratio is likely to be lower for matrices with a smaller degree of diagonal dominance along the columns. Smaller diagonal dominance along columns is also likely to increase the number of row interchanges to meet the pivoting threshold and therefore result in higher fill-in and overall factorization time.

We conjectured that row pivoting (i.e., scanning columns to search for pivots and interchanging rows) would yield smaller and more effective incomplete factors for matrices with higher average diagonal dominance along the columns than along the rows.

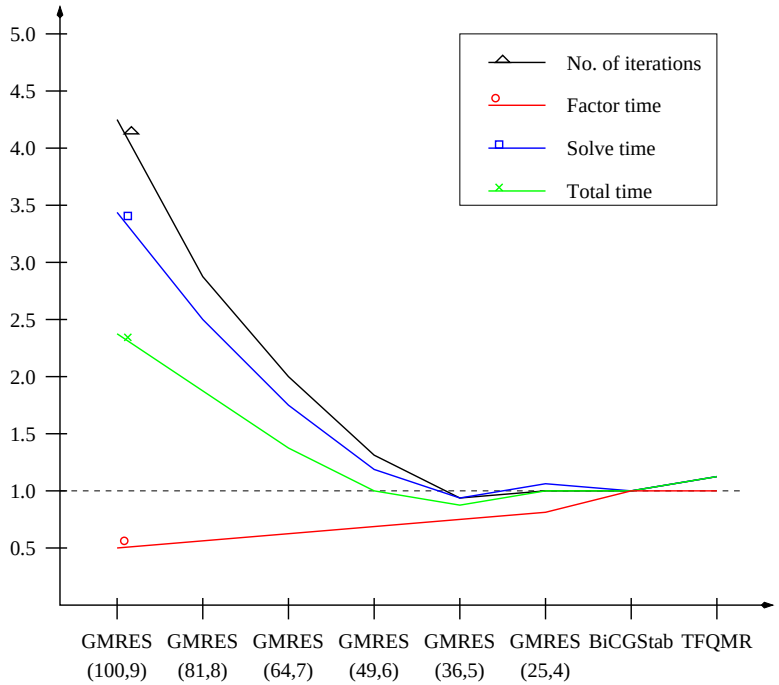

FIG. 11. *Relative performance of GMRES variants, BiCGStab, and TFQMR under constant total preconditioner and solver memory.*

Similarly, column pivoting (i.e., scanning rows to search for pivots and interchanging columns) would be more effective if the average diagonal dominance was higher along the rows than along columns. We verified the conjecture experimentally. Not surprisingly, it turned out to be valid for both complete and incomplete LU factorization with threshold partial pivoting. Section 4.1 describes how this observation was used in WSMP to improve the computation time and the quality of the preconditioners.

4.1. Methodology. Recall from Figure 1 that if the coefficient matrix is insufficiently diagonally dominant, then the first preprocessing step is to reorder it via an unsymmetric permutation based on MWBP [30] to maximize the product of the magnitudes of the diagonal entries. Let $\phi_i^R = |A_{ii}|/\sum_{j\neq i}|A_{ij}|$ be the measure of diagonal dominance of row i and let $\phi_i^C = |A_{ii}|/\sum_{j\neq i}|A_{ji}|$ be the measure of diagonal dominance of column i of an irreducible nonsingular coefficient matrix A. If the minimum of all ϕ_i^R's and ϕ_i^C's is less than 0.1 or if their geometric mean is less than 0.25, then we consider the matrix to be insufficiently diagonally dominant, and the unsymmetric permutation in step 1 of the algorithm in Figure 1 is performed. If the matrix is thus reordered, then ϕ_i^R and ϕ_i^C values are recomputed for all $0 < i \leq n$ after the reordering.

Next, we compare the product of $\min(\phi_i^R)$ and the geometric mean of all ϕ_i^R's with the product of $\min(\phi_i^C)$ and the geometric mean of ϕ_i^C's. If the former is smaller, then we proceed with the conventional ILU factorization with row pivoting. If the opposite is true, then we simply switch to using columns of A to populate the block rows U^A of the assembly blocks and the rows of A to populate the block columns L^A of the assembly blocks. Thus, without making any changes to the underlying factorization algorithm or the software, we factor A^T instead of A, effectively interchanging columns of A for pivoting when needed. To summarize, we use the same row-pivoting-based algorithm on either A or A^T, depending on which orientation we expect to result in fewer interchanges and smaller growth.

Skeel [53] showed that column pivoting with row equilibration satisfies a similar error bound as row pivoting without equilibration. The same relation holds between row pivoting with column equilibration and column pivoting without equilibration. Therefore, in theory, it may be possible to achieve the same effect as factoring the transpose by performing column equilibration, followed by standard ILU factorization with row pivoting. We did not explore selective equilibration, partly because our Crout-based implementation incurs no cost for switching between A and A^T and yields excellent results, as shown in section 4.2.

4.2. Experimental results. Table 3 shows the impact of selective transposition on the size and effectiveness of BILUC factors of the 16 test matrices in our suit of diverse test matrices. BILUC factorization of each matrix was computed, both with and without transposition, and used in restarted GMRES. Number of GMRES iterations, number of nonzeros in the incomplete factor, factorization time, and solution time were measured and tabulated. For the experiments in this section, an RHS vector with random numbers between 0.0 and 1.0 was used. The results of the configuration with the fastest combined factorization and solution are shown in bold. The middle "N or T" column indicates the choice that the solver made based on the heuristic discussed in section 4.1. The cases in which the choice led to the better configuration are marked with an *. The results show that in 13 out of the 16 cases, our heuristic made the correct choice. Moreover, it averted all three failures. In the cases in which the heuristic led to an increase in overall time, the difference between factoring the original or the transpose of the coefficient matrix was relatively small.

TABLE 3
*Impact of selective transposition on number of GMRES iterations, incomplete factor size (*million nonzeros*), factorization time (*seconds*), and solution time (*seconds*) for matrices in Table* 2.

Matrix	N: No transpose				N or T	T: Transpose			
	Iter. count	Factor size	Factor time	Solve time		Iter. count	Factor size	Factor time	Solve time
1	44	70.4	7.3	4.4	T*	**39**	**63.5**	**6.4**	**3.4**
2	**10**	**4.69**	**.33**	**.05**	N*	11	4.78	.34	0.06
3	**55**	**17.7**	**5.6**	**1.9**	N*	fail	-	-	-
4	**25**	**4.31**	**.38**	**.20**	T	27	5.51	.57	.31
5	22	29.9	12.3	1.0	T*	**28**	**30.4**	**11.6**	**1.2**
6	49	79.0	8.3	6.0	T*	**43**	**67.1**	**6.2**	**4.4**
7	fail	-	-	-	T*	**28**	**14.0**	**1.1**	**1.2**
8	14	27.7	7.6	.62	T*	**14**	**24.6**	**6.7**	**.54**
9	260	12.9	.75	6.0	N	**260**	**12.8**	**.72**	**5.1**
10	6	27.5	9.2	.73	T*	**5**	**26.3**	**8.4**	**.60**
11	**26**	**16.4**	**1.12**	**.88**	N*	26	16.3	1.12	.91
12	10	11.6	.59	.21	T*	**9**	**11.9**	**.58**	**.20**
13	26	41.2	4.5	1.9	N*	26	41.2	4.5	1.9
14	285	493.	94.	327.	T*	**260**	**533.**	**101.**	**288.**
15	fail	-	-	-	T*	**57**	**44.2**	**3.90**	**9.9**
16	31	126.	18	4.5	N	**33**	**116.**	**14.5**	**4.9**

TABLE 4
FIELDAY matrices and their basic information.

	Matrix	Dimension	Nonzeros
1.	case2	44563	661778
2.	m32bitf	194613	2225869
3.	matrix12	2757722	38091058
4.	matrixTest2_1	345150	2002658
5.	matrixTest2_10	1035461	5208887
6.	matrixTest2_19	2070922	10774594
7.	matrixTest3_1	231300	1364278
8.	matrixTest3_10	693904	3500181
9.	matrix-0	64042	326339
10.	matrix-11	384260	1687754

The test matrices used in Table 3 come from a variety of applications, and in many cases, the difference between rowwise and columnwise diagonal dominance in the coefficient matrix was not substantial. In order to demonstrate the effectiveness of selective transposition more conclusively, we gathered another set of matrices from an electrothermal semiconductor device simulation tool FIELDAY [5], which solves six coupled PDEs governing carrier mass and energy transport in three-dimensional (3D) semiconductor structures. The details of these matrices can be found in Table 4. We chose this application because it often tends to generate matrices for which transposition is critical for the success of ILU preconditioning.

Table 5 shows the effect of selective transposition on BILUC preconditioning on 10 linear systems derived from FIELDAY's application on real 3D semiconductor device simulation problems. FIELDAY matrices seem to have an overwhelming preference for transposition. Our heuristic not only made the correct choice in 80% of the test cases but it also avoided all the failures. Note that it may be possible to use column equilibration to achieve performance and robustness improvement similar to that offered by selective transposition for these matrices [53]. However, a reliable predictor

TABLE 5
*Impact of selective transposition on number of GMRES iterations, incomplete factor size (*million nonzeros*), factorization time (*seconds*), and solution time (*seconds*) of FIELDAY problems.*

Matrix	N: No transpose				N or T	T: Transpose			
	Iter. count	Factor size	Factor time	Solve time		Iter. count	Factor size	Factor time	Solve time
1	fail	-	-	-	T*	**280**	**2.99**	**0.12**	**1.33**
2	fail	-	-	-	T*	**40**	**8.48**	**0.95**	**1.60**
3	285	493.	94.	327.	T*	**260**	**533.**	**101.**	**288.**
4	**26**	**18.5**	**1.26**	**1.64**	T	25	18.5	1.27	1.83
5	fail	-	-	-	T*	**57**	**44.2**	**3.90**	**9.9**
6	fail	-	-	-	T*	**59**	**89.7**	**8.13**	**20.8**
7	39	11.7	0.79	1.56	N	**40**	**11.8**	**0.80**	**1.49**
8	fail	-	-	-	T*	**50**	**26.1**	**2.13**	**5.07**
9	17	2.51	0.18	0.10	T*	**17**	**2.50**	**0.18**	**0.10**
10	fail	-	-	-	T*	**36**	**12.8**	**1.42**	**2.10**

for when to use column equilibration would still be required, and it seems that our general-purpose heuristic based on the product of the smallest and the geometric mean of diagonal dominance in each orientation would be effective.

5. Comparison with contemporary ILU preconditioners. Section 3.6 contains an extensive comparison between BILUC and our implementation of the ILUC [38] algorithm. Since Li, Saad, and Chow [38] compare ILUC with conventional threshold-based ILU factorization (ILUT) [2], the results in section 3.6 also serve as an indirect comparison with ILUT. In this section, we present a brief direct or indirect comparison with other recently published ILU algorithms for which either specific performance data is available or the software is available to obtain performance data experimentally. The following solvers are compared:

- VBARMS by Carpentieri, Liao, and Sosonkina [6], who compare it with ARMS [51] in their paper. VBARMS is also introduced in section 1.
- HSL_MI30 by Scott and Tůma [52], who compare it with SYM-ILDL [23] in their paper. HSL_MI30 and SYM-ILDL are two state-of-the-art preconditioners for symmetric indefinite systems that require pivoting. Although BILUC is designed for unsymmetric matrices and unlike HSL_MI30 and SYM-ILDL does not take advantage of symmetry to reduce factorization operation count by half, we include this comparison to show that BILUC's efficiency makes it competitive even for symmetric indefinite systems.
- SuperLU's threshold-based incomplete factorization by Li and Shao [39], who compare SuperLU with ILUPACK [4] and SPARSKIT [46]. This preconditioner is introduced in section 1.

From the papers [6, 52] on VBARMS and HSL_MI30, we picked the best reported results for a few of the largest or the hardest (as reported by the authors) problems and put them in Table 6 along with the corresponding results obtained by using the best drop tolerance in the BILUC algorithm under our experimental setup. For BILUC's data in this section, we used a single thread and an RHS such that the solution is a unit vector to match the experimental scenarios under which the other solvers were used. Although the factorization and solution times in the first seven rows in Table 6 come from different machines, these machines are similar in speed. For SuperLU [39], exact performance statistics on specific matrices were unavailable; however, we could obtain the software. We compared its supernodal threshold-based incomplete factorization

TABLE 6

Comparison of BILUC with other contemporary ILU preconditioners based on incomplete factor size (millions of nonzeros), factorization time (seconds), preconditioned restarted GMRES time (seconds), and iteration count. †: best of VBARMS [6] *and ARMS*2 [51]*; ‡: best of HSL_MI*30 [52] *and SYM-ILDL* [23]*; ⋄: SuperLU* [39].

Matrix	WSMP's BILUC preconditioner				Other ILU preconditioners				
	Factor size	Factor time	Solve time	Iter. count	Factor size	Factor time	Solve time	Iter. count	Comment
nasasrb	4.59	2.02	3.30	52	6.46	13.9	11.0	94	†
venkat01	5.11	0.52	0.45	12	0.81	0.19	0.51	40	
xenon1	3.86	1.36	1.20	25	5.21	10.5	12.2	220	
c-71	2.86	2.05	1.02	33	1.64	1.10	1.53	53	‡
c-big	8.30	4.01	7.94	59	5.96	5.00	16.3	56	
darcy003	19.2	2.40	1.75	9	6.64	3.39	4.09	35	
stokes128	9.51	1.50	0.65	13	0.85	0.33	1.83	149	
ML_Laplace	92.7	21.6	67.0	90	176.	679.	15.4	13	⋄
Transport	133.	24.4	134.	95	199.	3241	77.0	37	
atmosmodd	113.	27.0	3.6	31	81.0	1088	229.	190	
cage13	35.0	34.8	2.04	5	54.8	1.3E4	3.48	7	
largebasis	9.0	0.93	1.65	10	6.68	2.31	2.77	22	

with WSMP's [28] BILUC on a single thread of the same Power 6 machine with default values of drop tolerance and other options for both solvers. The GMRES restart parameter was set to 100 for both solvers and augmentation of the subspace vectors with estimates of eigenvectors corresponding to the smallest eigenvalues was turned off in WSMP.

The comparison with VBARMS shows that other than the smallest problem, *venkat*01, BILUC generates sparser preconditioners that are faster to compute and are much more effective, presumably because of pivoting. To test this hypothesis, we resolved *nasasrb* and *xenon*1 using BILUC with pivoting turned off. The factorization and solution times for *xenon*1 were mostly unchanged, but *nasasrb* took 272 iterations and 20.8 seconds to solve without pivoting. Although the sum of BILUC's factorization and solution time was still smaller than that of VBARMS, this shows that BILUC's ability to perform unrestricted pivoting plays a significant role in making it an effective preconditioner.

The comparison with HSL_MI30 and SYM-ILDL shows that although BILUC computes an ILU factorization of the full matrix, its blocking and pivoting schemes make it competitive with state-of-the-art incomplete LDL^T factorization algorithms that factor only triangular portions of symmetric indefinite matrices.

The last five rows of Table 6 show a comparison with SuperLU's threshold-based incomplete factorization for relatively large problems from the Florida Sparse Matrix Collection [10]. BILUC uses large assembly blocks in order to maximize the computational benefits of blocking and uses smaller blocks for factoring to maintain the precision of dropping entries during incomplete factorization. SuperLU uses the same supernodes for each type of operation and therefore faces a trade-off between effective blocking and effective dropping. In order to avoid dropping larger entries when entire rows of supernodes are dropped, it must use a conservative dropping scheme, which can lead to costly and dense factors. SuperLU's pivot search is also confined to the supernodes. The benefits of BILUC's two-tier blocking with unconstrained pivoting are apparent from results of solving some of the larger systems, for which it is more than an order of magnitude faster than SuperLU.

Table 6 and Figure 9 show that BILUC is competitive with or outperforms most currently available preconditioners based on ILU factorization. As expected, BILUC's advantage over other ILU-based preconditioners increases as the problems get larger.

6. Concluding remarks and future work. We have introduced techniques to improve the reliability and performance of ILU factorization-based preconditioners for solving general sparse systems of linear equations. Along with its sister publication [29] for symmetric systems, this paper presents a comprehensive block framework for incomplete factorization preconditioning. This framework almost invariably leads to faster and more robust preconditioning. In addition, it goes a long way in alleviating the curse of fill-in, whereby, in the absence of blocking, incomplete factorization time grows rapidly as the density of the preconditioner increases [29]. Blocking makes it practical to compute denser and more robust incomplete factors. Blocking is also likely to render incomplete factorization-based preconditioning more amenable to multicore and accelerator hardware.

In conventional ILU factorization preconditioning, drop tolerance and fill factor are the typical tuning parameters that control the trade-offs between memory, run time, and robustness. Since the block version of ILU factorization permits efficient computation of preconditioners with a wider range of densities (Figure 9), it enables the inclusion of the GMRES restart parameter among the parameters to be tuned simultaneously to optimize the overall solution time as well as memory consumption. To the best of our knowledge, this is the first attempt to study the impact of splitting a fixed amount of memory in different ratios between restarted GMRES and its preconditioner. We find that if the memory for storing GMRES's subspace vectors is diverted toward computing denser and stronger ILU preconditioners, then BiCGStab performs almost as well as restarted GMRES on our suite of test problems. BiCGStab and restarted GMRES delivering similar overall performance for the same amount of memory may have important implications for the choice of Krylov subspace method on highly parallel computers, on which the orthogonalization step of GMRES could be a scalability bottleneck [56].

The selective transposition heuristic proposed in this paper could extend the results of Almeida, Chapman, and Derby [11], who observed that in many cases, judicious use of equilibration could help to preserve the original ordering of the sparse matrix by reducing the amount of pivoting. Combined with Skeel's results [53] on the relation between the direction of pivoting and the type of equilibration, we think that it is possible to give more precise guidance on how to use scaling beneficially while factoring general sparse matrices.

Acknowledgments. The author would like to thank Haim Avron, Thomas George, Rogeli Grima, Felix Kwok, and Lexing Ying. Pieces of software written by them over the years are included in WSMP's iterative solver package.

REFERENCES

[1] M. Benzi, *Preconditioning techniques for large linear systems: A survey*, J. Comput. Phys., 182 (2002), pp. 418–477.
[2] M. Bollhöfer, *A robust ILU with pivoting based on monitoring the growth of the inverse factors*, Linear Algebra Appl., 338 (2001), pp. 201–213.
[3] M. Bollhöfer, *A robust and efficient ILU that incorporates the growth of the inverse triangular factors*, SIAM J. Sci. Comput., 25 (2003), pp. 86–103.
[4] M. Bollhöfer, Y. Saad, and O. Schenk, *ILUPACK V*2.2., https://www.math.tu-berlin.de/ilupack (2006).

[5] E. Buturla, J. Johnson, S. Furkay, and P. Cottrell, *A new 3D device simulation formulation*, in Proceedings of Numerical Analysis of Semiconductor Devices and Integrated Circuits, 1989.

[6] B. Carpentieri, J. Liao, and M. Sosonkina, *VBARMS: A variable block algebraic recursive multilevel solver for sparse linear systems*, J. Comput. Appl. Math., 259 (2014), pp. 164–173.

[7] E. Chow and M. A. Heroux, *An object-oriented framework for block preconditioning*, ACM Trans. Math. Software, 24 (1998), pp. 159–183.

[8] E. Chow and Y. Saad, *Experimental study of ILU preconditioners for indefinite matrices*, J. Comput. Appl. Math., 86 (1997), pp. 387–414.

[9] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, in Proceedings of the 24th National Conference of the ACM, 1969, pp. 152–172.

[10] T. A. Davis, *The University of Florida sparse matrix collection*, Tech. report, Department of Computer Science, University of Florida, 2007, https://www.cise.ufl.edu/research/sparse/matrices.

[11] V. F. de Almeida, A. M. Chapman, and J. J. Derby, *On equilibration and sparse factorization of matrices arising in finite element solutions of partial differential equations*, Numer. Methods Partial Differential Equations, 16 (2000), pp. 11–29.

[12] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Software, 16 (1990), pp. 1–17.

[13] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson, *An extended set of FORTRAN basic linear algebra subprograms*, ACM Trans. Math. Software, 14 (1988), pp. 1–17.

[14] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, Oxford, UK, 1990.

[15] I. S. Duff and J. Koster, *On algorithms for permuting large entries to the diagonal of a sparse matrix*, SIAM J. Matrix Anal. Appl., 22 (2001), pp. 973–996.

[16] I. S. Duff and G. A. Meurant, *The effect of ordering on preconditioned conjugate gradient*, BIT, 29 (1989), pp. 635–657.

[17] I. S. Duff and J. K. Reid, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Trans. Math. Software, 9 (1983), pp. 302–325.

[18] Q. Fan, P. A. Forsyth, J. R. F. McMacken, and W.-P. Tang, *Performance issues for iterative solvers in device simulation*, SIAM J. Sci. Comput., 17 (1996), pp. 100–117.

[19] R. W. Freund, *A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems*, SIAM J. Sci. Statis. Comput., 14 (1993), pp. 470–482.

[20] A. George and J. W.-H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[21] T. George, A. Gupta, and V. Sarin, *An empirical analysis of iterative solver performance for SPD systems*, ACM Trans. Math. Software, 38 (2012).

[22] J. R. Gilbert and S. Toledo, *An assessment of incomplete-LU preconditioners for nonsymmetric linear systems*, Informatica, 24 (2000), pp. 409–425.

[23] C. Greif, S. He, and P. Liu, *SYM-ILDL: C++ Package for Incomplete Factorization of Symmetric Indefinite Matrices*, https://github.com/inutard/matrix-factor (2013).

[24] A. Gupta, *Improving performance and robustness of incomplete factorization preconditioners*, presented at SIAM Conference on Applied Linear Algebra, Valencia, Spain, 2012.

[25] A. Gupta, *Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices*, SIAM J. Matrix Anal. Appl., 24 (2002), pp. 529–552.

[26] A. Gupta, *Fast and effective algorithms for graph partitioning and sparse matrix ordering*, IBM J. Research Development, 41 (1997), pp. 171–183.

[27] A. Gupta, *WSMP: Watson Sparse Matrix Package Part* II*—Direct Solution of General Systems*, Tech. report RC 21888, IBM T. J. Watson Research Center, Yorktown Heights, NY, 2000, http://www.research.ibm.com/projects/wsmp.

[28] A. Gupta, *WSMP: Watson Sparse Matrix Package Part* III*—Iterative Solution of Sparse Systems*, Tech. report RC 24398, IBM T. J. Watson Research Center, Yorktown Heights, NY, 2007. http://www.research.ibm.com/projects/wsmp.

[29] A. Gupta and T. George, *Adaptive techniques for improving the performance of incomplete factorization preconditioning*, SIAM J. Sci. Comput., 32 (2010), pp. 84–110.

[30] A. Gupta and L. Ying, *On Algorithms for Finding Maximum Matchings in Bipartite Graphs*, Tech. report RC 21576, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1999.

[31] P. Hénon, P. Ramet, and J. Roman, *On finding approximate supernodes for an efficient block-ILU(k) factorization*, Parallel Comput., 34 (2008), pp. 345–362.

[32] N. J. Higham and D. J. Higham, *Large growth factors in Gaussian elimination with pivoting*, SIAM J. Matrix Anal. Appl., 10 (1989), pp. 155–164.

[33] D. Hysom and A. Pothen, *A scalable parallel algorithm for incomplete factor preconditioning*, SIAM J. Sci. Comput., 22 (2000), pp. 2194–2215.
[34] C. Janna, M. Ferronato, and G. Gambolati, *A block FSAI-ILU parallel preconditioner for symmetric positive definite linear systems*, SIAM J. Sci. Comput., 32 (2010), pp. 2468–2484.
[35] M. T. Jones and P. E. Plassmann, *Blocksolve95 Users Manual: Scalable Library Software for the Parallel Solution of Sparse Linear Systems*, Tech. report ANL-95/48, Argonne National Laboratory, Argonne, IL, 1995.
[36] I. E. Kaporin, L. Y. Kolotilina, and A. Y. Yeremin, *Block SSOR preconditionings for high-order* 3*D FE systems.* II *Incomplete BSSOR preconditionings*, Linear Algebra Appl., 154-156 (1991), pp. 647–674.
[37] G. Karypis and V. Kumar, *Parallel Threshold-Based ILU Factorization*, Tech. report TR 96-061, Department of Computer Science, University of Minnesota, 1996.
[38] N. Li, Y. Saad, and E. Chow, *Crout versions of ILU for general sparse matrices*, SIAM J. Scit. Comput., 25 (2003), pp. 716–728.
[39] X. S. Li and M. Shao, *A supernodal approach to incomplete LU factorization with partial pivoting*, ACM Trans. Math. Software, 37 (2011).
[40] J. W.-H. Liu, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp.134–172.
[41] J. W.-H. Liu, *The multifrontal method for sparse matrix solution: Theory and practice*, SIAM Rev., 34 (1992), pp. 82–109.
[42] J. W.-H. Liu, E. G.-Y. Ng, and B. W. Peyton, *On finding supernodes for sparse matrix computations*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 242–252.
[43] R. B. Morgan, *A restarted GMRES method augmented with eigenvectors*, SIAM J. Matrix Anal. Appl., 16 (1995), pp. 1154–1171.
[44] N. Munksgaard, *Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients*, ACM Trans. Math. Software, 6 (1980), pp. 206–219.
[45] E. G.-Y. Ng, B. W. Peyton, and P. Raghavan, *A blocked incomplete cholesky preconditioner for hierarchical-memory computers*, in Iterative Methods in Scientific Computation IV, IMACS Series in Computational and Applied Mathematics, D. R. Kincaid and A. C. Elster, eds., Elsevier, Amsterdam, 1999, pp. 211–221.
[46] Y. Saad, *SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations*, Tech. report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.
[47] Y. Saad, *ILUT: A dual threshold incomplete LU factorization*, Numerical Linear Algebra with Applications, 1 (1994), pp. 387–402.
[48] Y. Saad, *Finding exact and approximate block structures for ILU preconditioning*, SIAM J. Sci. Comput., 24 (2003), pp. 1107–1123.
[49] Y. Saad, *Iterative Methods for Sparse Linear Systems,* 2*nd ed.*, SIAM, Philadelphia, 2003.
[50] Y. Saad and M. H. Schultz, *GMRES: A generalized minimal residual algorithm for solving non-symmetric linear systems*, SIAM J. Sci. Statis. Comput., 7 (1986), pp. 856–869.
[51] Y. Saad and B. Suchomel, *ARMS: An algebraic recursive multilevel solver for general sparse linear systems*, Numer. Linear Algebra Appl., 9 (2002), pp. 359–378.
[52] J. Scott and M. Tůma, *On signed incomplete Cholesky factorization preconditioners for saddle-point systems*, SIAM J. Sci. Comput., 36 (2014), pp. A2984–A3010.
[53] R. D. Skeel, *Effect of equilibration on residual size for partial pivoting*, SIAM J. Numer. Anal., 18 (1981), pp. 449–454.
[54] H. A. van der Vorst, *Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput., 13 (1992), pp. 631–644.
[55] N. Vannieuwenhoven and K. Meerbergen, *IMF: An incomplete multifrontal LU-factorization for element-structured sparse linear systems*, SIAM J. Sci. Comput., 35 (2013), pp. A270–A293.
[56] I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, and J. Dongarra, *Improving the performance of CA-GMRES on multicores with multiple GPUs*, in Proceedings of the International Parallel and Distributed Processing Symposium, 2014.