RC25577 (WAT1511-047) November 16, 2015
Computer Science

IBM Research Report

Enhancing Performance and Robustness of ILU Preconditioners through Blocking and Selective Transposition

Anshul Gupta
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

Research Division
Almaden – Austin – Beijing – Brazil – Cambridge – Dublin – Haifa – India – Kenya – Melbourne – T.J. Watson – Tokyo – Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Many reports are available at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

ENHANCING PERFORMANCE AND ROBUSTNESS OF ILU PRECONDITIONERS THROUGH BLOCKING AND SELECTIVE TRANSPOSITION

ANSHUL GUPTA*

Abstract. Incomplete factorization is one of the most effective general-purpose preconditioning methods for Krylov subspace methods for solving 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 for incomplete factorization based on the Crout variation of dense LU factorization is presented. The algorithm is suitable for incorporating threshold-based dropping as well as partial pivoting. 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 GMRES 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. 65F10, 65F50

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 a well known fact 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 [17] and multifrontal [14, 37] 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 [18]. In the context of incomplete factorization, which has long been used to precondition Krylov subspace methods [1, 43], 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 [26], and for many problems, a direct solution turns out to be faster than an iterative solution, even when complete factorization involves significantly more memory and floating-point operations than incomplete factorization [18, 19, 20]. 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 sparse incomplete factors can be practically computed, and the resulting preconditioning is often not effective for hard problems. Performing sparse operations on dense blocks instead of individual elements during incomplete factorization and the solution phases can potentially close the performance gap with complete factorization.

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

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 [21], is based on the Crout variation of LU factorization and supports threshold-based dropping for fill reduction as well as partial pivoting for numerical stability. Li *et al.* [34] 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.

Blocked incomplete factorization has several benefits other than the obvious time saving in the preconditioner set-up phase of the solution process. First, denser and more robust factors, which would have been impractical to compute with conventional methods, can now be computed economically. Secondly, 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 [44] 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 combination of high-density ILU and GMRES(m) with a small m can be used instead of a combination of low-density ILU and GMRES(m) with a large m, without changing the overall memory footprint. On a diverse suite of test problems, we observed that a value in the range of 30 to 40 appeared to be a good choice for m. Our experiments seem to indicate that it is better to use the memory in the ILU preconditioner that increasing m beyond this range. On the other hand, if memory is scarce, then a sparse preconditioner is preferable to reducing m below this range. 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. We show that this is enabled by blocking and would not be possible with conventional row- or column-based ILU factorization.

Dense blocks have been used successfully in the past [4, 15, 28, 31, 32, 35, 41] to enhance the performance of incomplete factorization preconditioners. However, with a few exceptions [35, 41], these block algorithms have been applied to relatively simpler level-based incomplete factorization of matrices that have a naturally occurring block structure. The BILUC algorithm employs the more powerful threshold-based dropping, and not only detects and uses dense blocks in matrices in which such blocks occur naturally, but is also able to realize the benefits of blocking for those matrices that have poor or no block structure to begin with. The algorithm successfully employs classical complete factorization techniques of partial pivoting within column

and row blocks (supernodes) [38] and delayed pivoting for numerical accuracy. It uses a combination of graph-partitioning for parallelism and nested-dissection for fill reduction. The BILUC algorithm is most similar to Li and Shao's [35] left-looking serial algorithm, which itself is similar to our block incomplete Cholesky factorization algorithm [26].

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 of the form $Ax = b$ can be obtained by either 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), often the quality of $A^T = U^T L^T$ factorization can be vastly 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. The choice of factorization; i.e., whether A or A^T is factored effects both. In fact, when factorization is incomplete and is used for preconditioning, the impact is magnified because a poorer preconditioner may 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 identically.

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 four cores of a 4.2 GHz Power 6 system running AIX with 96 GB of memory. In all our experiments, the right-hand side vector b of the sparse system $Ax = b$ to be solved is set such that the solution x is all ones. A maximum of 2000 inner iterations of restarted GMRES [44] were permitted in any experiment, and were terminated when the relative residual norm dropped below 10^{-8}. This choice of threshold for the residual norm was based on the largest value that resulted in a reasonable (typically, in the range of 10^{-3} to 10^{-6}) error norm for the problems in our test suite. 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 [39]. The software implementation is a part of the Watson Sparse Matrix Package (WSMP) library [25], whose object code and documentation is available for testing and benchmarking at *http://www.research.ibm.com/projects/wsmp*.

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 must precede the incomplete factorization. In Section 3, we describe the BILUC algorithm in detail and present some experimental results to demonstrate the effectiveness of the blocking scheme. In Section 4, we discuss a reasonably effective way of boosting the robustness of BILUC (or any ILU) preconditioner, and present experimental results to demonstrate the effectiveness of the heuristic. Section 5 contains concluding remarks.

2. Overview of Preconditioning Scheme. Figure 2.1 gives an overview of the shared-address-space parallel preconditioning scheme based on BILUC factorization. Note that the parallelization strategy is aimed at exploiting only a moderate degree of

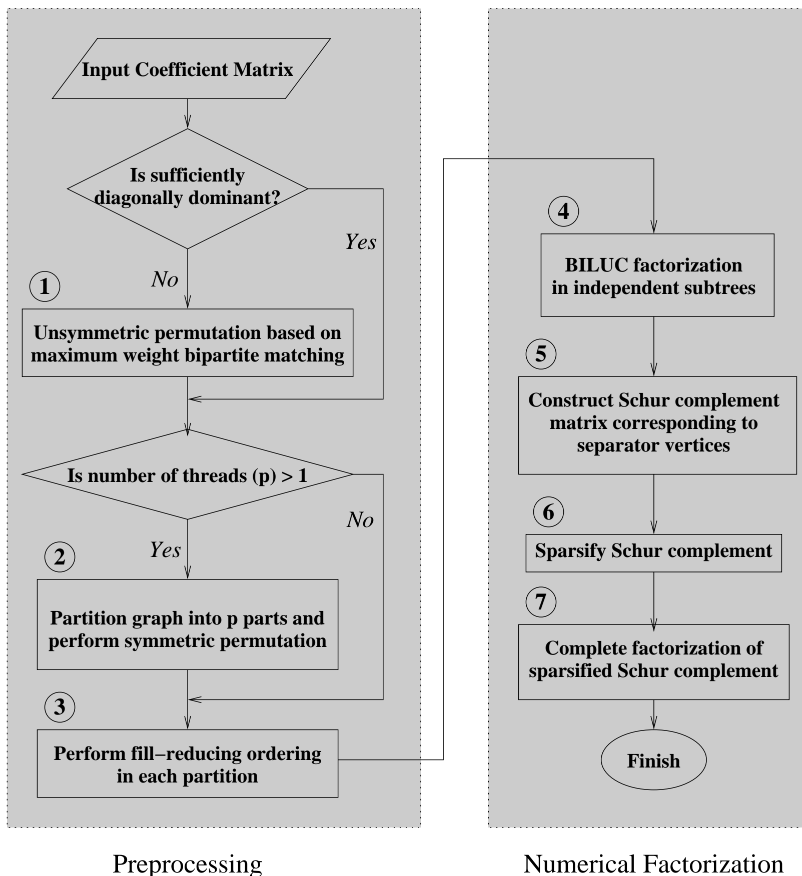

FIG. 2.1. *An overview of BILUC-based preconditioning*

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 [30, 33] 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 after the overview, the remainder of the paper will focus on enhancing the robustness and performance of BILUC factorization. However, this mildly 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. We use standard preconditioned Krylov subspace solvers such as GMRES [44], BiCGStab [46], and TFQMR [16]. In this paper, we primarily focus on preprocessing and preconditioner generation.

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

First, if the matrix has poor diagonal dominance, then it is subject to an unsymmetric row or column permutation to improve its diagonal dominance. This step uses maximum weight bipartite matching [12, 27] to compute a permutation of rows or columns of the matrix to maximize the product of its diagonal entries.

Next, an undirected graph of the matrix is constructed and partitioned [23] into p parts, where p is the number of parallel threads being used. The partitioning 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) [6, 11] ordering is used if the incomplete factors are expected to be relatively sparse (based on the dropping criteria); otherwise, nested dissection [17, 23] ordering is used. The choice of ordering is based on our and others' [13] observation that RCM generally performs better with low fill in and that 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 possible unsymmetric permutation to improve diagonal dominance. 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 the third permutations depend solely on the sparsity pattern of the matrix, the first step of unsymmetric permutation to enhance diagonal dominance depends on the numerical values of the nonzero entries in it. This permutation, 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 [36] from the the structures of $B_i + B_i^T$, where B_i is the i-th diagonal block corresponding to the i-th domain ($1 \leq i \leq p$) of the coefficient matrix. An elimination tree defines the task and data dependencies in the factorization process, and along with symbolic factorization [17], its construction is a critical preprocessing step in complete factorization. In the context of incomplete factorization with threshold-based dropping, an a priori symbolic factorization to determine the location and number of nonzeros in the factor matrices is not performed. It is not useful because only a small fraction of nonzeros entries are saved and the location of discarded and saved entries is unpredictable before the actual factorization.

Figure 2.2 shows the correspondence between the partitioned (symmetrized) graph, the elimination tree, and the reordered matrix for the case of 4 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

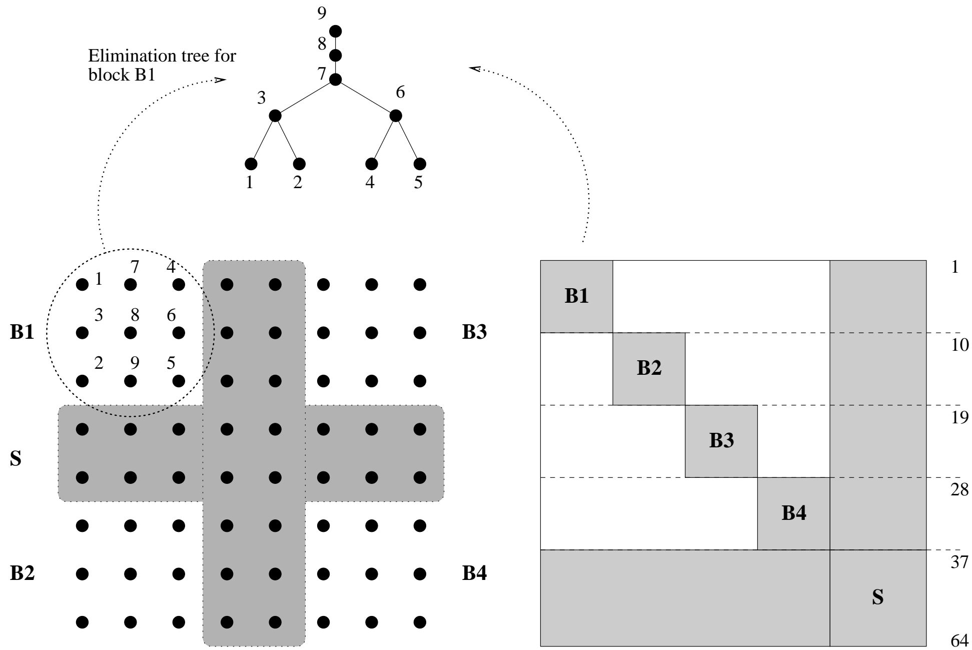

FIG. 2.2. *Correspondence between the partitioned (symmetrized) graph, the elimination tree, and the reordered matrix.*

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 formed corresponding to the remaining graph vertices that have edges traversing partition boundaries. This Schur complement (matrix S in Figure 2.2) is then further sparsified through a round of dropping, and the sparsified matrix is factored using a parallel direct solver [24]. 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 portion 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 operating in a shared address space. This is because a larger portion of the matrix is presparsified 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 the factors as 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 *et al.*'s ILUC algorithm [34], 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 (DAGs) [22]. 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 [36]. 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 using a suboptimal dependency graph in the case of sparse 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 corresponding to straight portions of the elimination tree. In these straight portions, the parents have one child each, for example, indices 7, 8, and 9 in Figure 2.2. If a vertex has no child, then it is a leaf, and when it has more than one child, then the tree branches. 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 3.1 illustrates assembly and factor blocks and their relation to the elimination tree.

Although assembly and factor blocks consists 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.

The 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. Figure 3.1 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

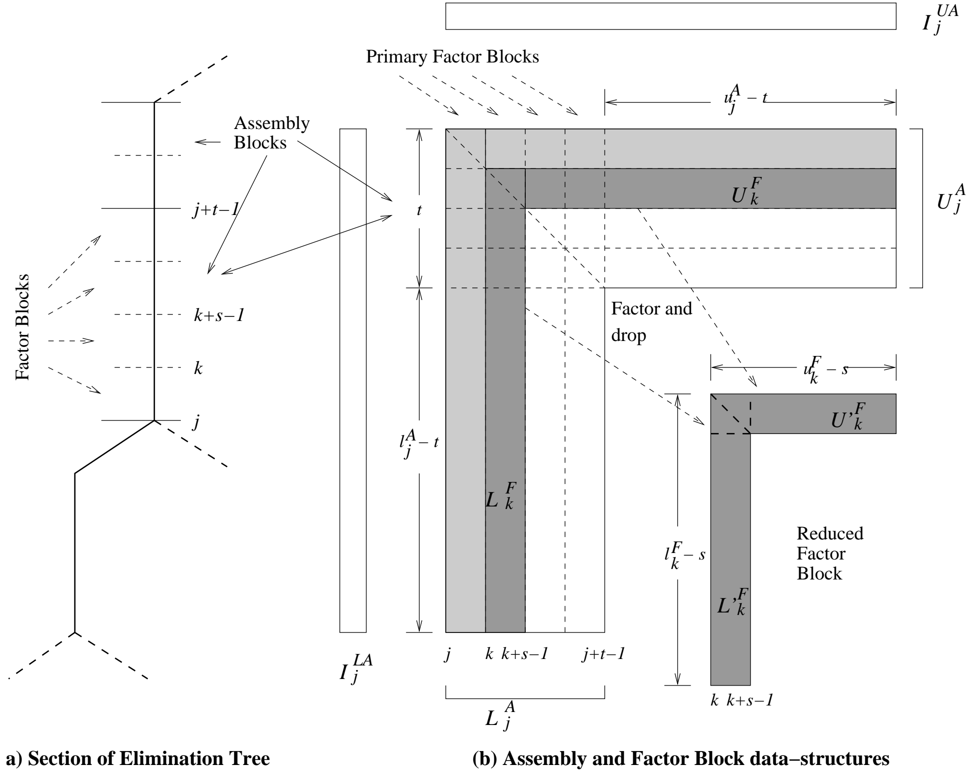

FIG. 3.1. *A section of an elimination tree and related assembly and factor blocks.*

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 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 in the actual implementation so that elements in a row reside in contiguous memory locations. 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.

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

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}

TABLE 3.1
BILUC data structures and the conventions used for their representation in this paper.

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.

Figure 3.1(b) shows one such factor block, its relationship with its assembly block, and the typical BILUC dense data-structures corresponding to typical assembly and factor blocks. The 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, entries 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, $l_k^F \leq l_j^A + j - k$ and $u_k^F \leq u_j^A + j - k$. 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 in the original matrix corresponding to each row of $L_k'^F$ and each column of $U_k'^F$.

Table 3.1 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 3.2 shows the assembly and factor blocks corresponding to Figure 3.1(b) with local indices only.

LU factorization within an assembly block proceeds in a manner very similar to that in complete supernodal [38] factorization. Partial pivoting is performed based on a user-defined pivot threshold α. For most matrices that are not strictly diagonally

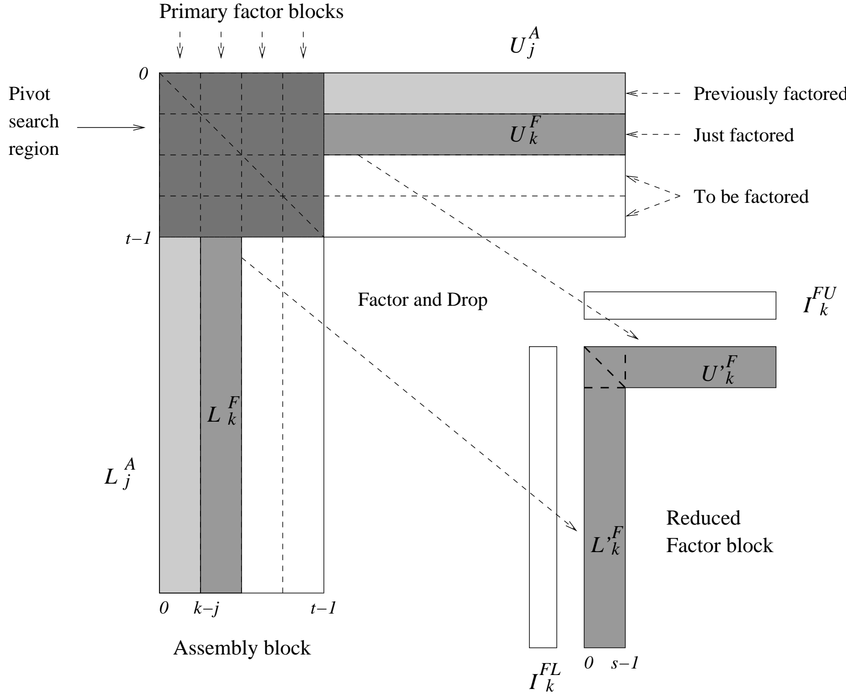

FIG. 3.2. *A typical assembly and factor block in the BILUC algorithm.*

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 is stable without pivoting [5].

When attempting to factor column i $(0 \leq i < t)$ in an assembly block, the BILUC algorithm scans column i of L_j^A to determine $g = \max_{i \leq m < t} |L_j^A(m,i)|$ and $h = \max_{t \leq m < l_j^A} |L_j^A(m,i)|$. If the maximum magnitude g in column i within the pivot block is greater than or equal to α times the maximum magnitude h beyond the pivot block (i.e., $g > \alpha h$), 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 within the pivot block in column i, then subsequent columns are searched. If a suitable pivot element is found, 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 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 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 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 pivot growth. In WSMP's BILUC implementation, we use two piv-

oting 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 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 in the case when only α was used as a single threshold.

Factorization in an assembly block takes place in units of factor blocks. When a complete primary factor block is factored, then it undergoes sparsification via the dropping strategy discussed in Section 3.3. 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 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. WSMP implements a dual dropping strategy of the form introduced by Saad [42]. 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.

The factorization process described in Section 3.2 is essentially the same as the one used is used in WSMP's general direct solver [24]. It is used in the BILUC algorithm in conjunction with the dropping and downdating strategy described below.

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 $dscr_L$ is assigned to each row of L_k^F and $dscr_U$ to each column of U_k^F in the primary factor block. Specifically,

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

and

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

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 and columns of primary factor blocks 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 column-wise, each diagonal entry $L_k^F(m,m)$ in Equation 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 $0 \leq m < s$. 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 [42, 43], or dropping based on Munksgaard's criterion [40]. We found dropping based on magnitude relative to the corresponding diagonal entry to be slightly better than the other two on an average for the problems in our test suite. The BILUC algorithm is well-suited for incorporating dropping based on the growth of inverse of triangular factors [2], and we plan to include that as an option in a future release.

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 factor block entries that eventually end up being dropped. As Chow and Saad [5] 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, the 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 of $L_k'^F$, $U_k'^F$, I_k^{FL}, and I_k^{FU} 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 non-blocked 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

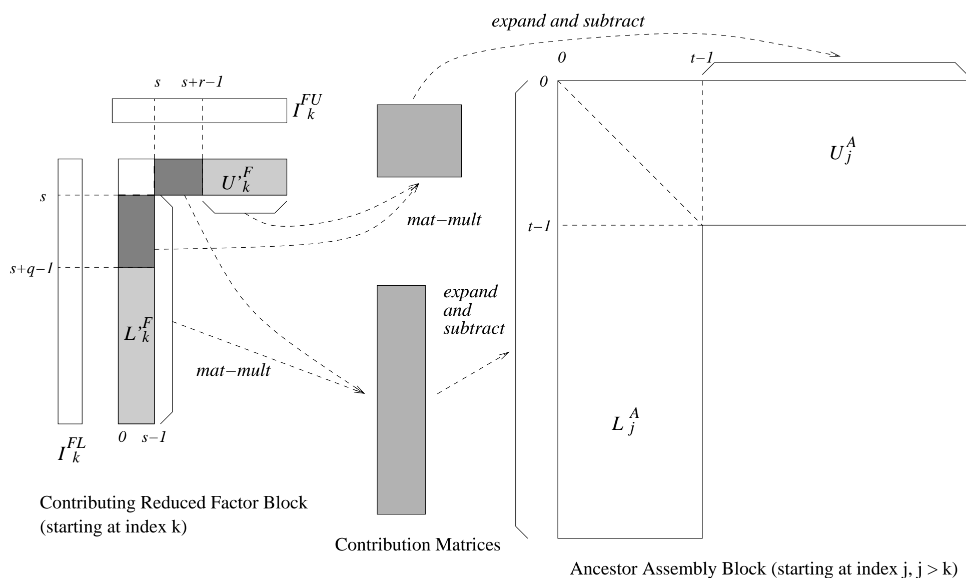

FIG. 3.3. *A reduced factor block updating its first ancestor assembly block.*

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.

3.4. Constructing assembly blocks. After 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 3.1 includes indices $j, j+1, \ldots, j+t-1$. However, the sizes l_j^A and u_j^A and the indices in I_j^{AL} and I_j^{AU} are determined only when the assembly block is actually constructed, which happens just before it is factored.

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.

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. The set of its row indices is simply the union of row indices of columns $j, j+1, \ldots, j+t-1$ of A and the set of its column indices is the union of column indices of rows $j, j+1, \ldots, j+t-1$ of A. All entries in L_j^A and U_j^A that are not in A are filled with with zeros.

All assembly blocks that do not start at a leaf have a linked list of contributing reduced factor blocks associated with them. At the beginning of the BILUC algorithm, all lists are empty. When a reduced factor block starting at index k is created, then it is placed in the linked list of the ancestor assembly block whose pivot block contains the index $v = \min(I_k^{FL}(s), I_k^{FU}(s))$. Here v is the smallest index greater than the pivot indices of the current assembly block. Note that the first s entries $I_k^{FL}(0 : s-1)$ and $I_k^{FU}(0 : s-1)$ are simply $k, k+1, \ldots, k+s-1$. The ancestor assembly block is

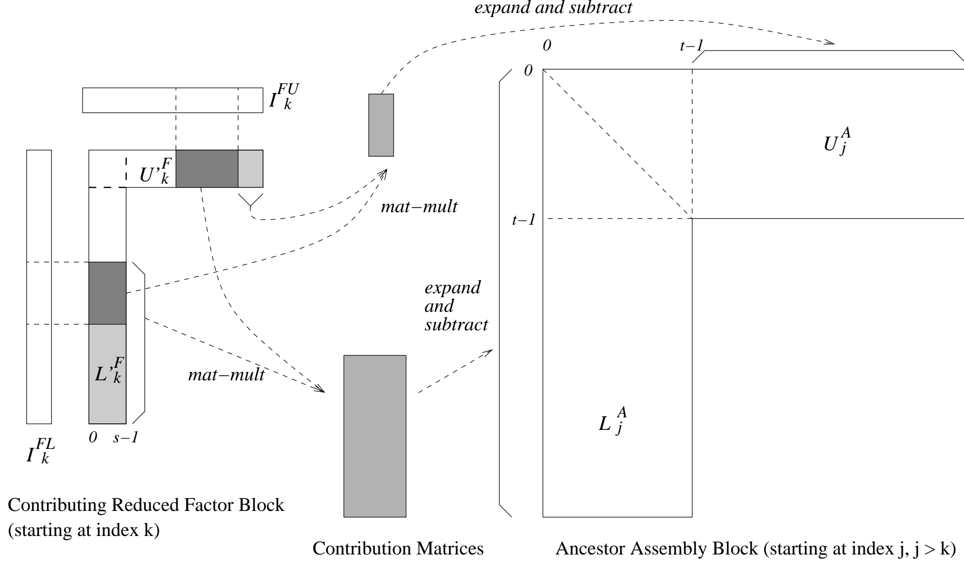

FIG. 3.4. *A reduced factor block updating an ancestor assembly block.*

the first assembly block that the entries in factor block k will update.

When the BILUC process reaches a non-leaf assembly block of size t starting at index j, the first step is to construct I_j^{AL} and I_j^{AU}. I_j^{AL} is the union of (a) $\{j, j+1, \ldots, j+t-1\}$, (b) all indices greater than or equal to $j+t$ in columns $j, j+1, \ldots, j+t-1$ of A, and (c) all indices greater than or equal to $j+t$ in I_k^{FL} for all k such that the reduced factor block starting at k is in the current assembly block's linked list. Similarly, I_j^{AU} is the union of (a) $\{j, j+1, \ldots, j+t-1\}$, (b) all indices greater than or equal to $j+t$ in rows $j, j+1, \ldots, j+t-1$ of A, and (c) all indices greater than or equal to $j+t$ in I_k^{FU} for all k such that the reduced factor block starting at k is in the current assembly block's linked list.

Once I_j^{AL} and I_j^{AU} have been constructed, the size of the assembly block is known. It is then allocated, and 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^A and U_j^A. Figure 3.3 shows how the contribution matrices are computed from a reduced factor block. Let q and r be such that $I_k^{FL}(s+q)$ is the first index in I_k^{FL} greater than or equal to $j+t$, and $I_k^{FU}(s+r)$ is the first index in I_k^{FU} greater than or equal to $j+t$. As shown in the figure, q and r 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^A and U_j^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^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^A. In general, the global row and column indices associated with the contribution matrices are subsets of the indices associated with L_j^A and U_j^A. Therefore, the contribution matrices are expanded to align their row and column index sets with those of L_j^A and U_j^A before subtraction.

After extracting the contribution from a reduced factor block starting at k, if both q and r, as described above, exist (i.e., both I_k^{FL} and I_k^{FU} have at least one index

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 $L_k'^F$ and $U_k'^F$ 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. $v = \min(I_k^{FL}(s), I_k^{FU}(s))$;
17. Insert k in linked list of assembly block containing index v;
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. 3.5. *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. Block i refers to a block starting at index i for both factor and assembly blocks.*

greater than or equal to $j+t$), then the reduced factor block is placed in the linked list of an assembly block whose block contains the index $v = \min(I_k^{FL}(s+q), I_k^{FU}(s+r))$. Figure 3.4 illustrates how this reduced factor block will contribute to the next assembly block. The set of indices in I_k^{FL} and I_k^{FU} corresponding to the dark shaded regions of $L_k'^F$ and $U_k'^F$ are always subsets of the pivot indices $j, j+1, \ldots, j+t-1$ of the assembly block that is being updated. Note that j and t are used generically; the ones in Figure 3.4 would have different values than the ones in Figure 3.3.

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 factorization as described in Section 3.2. Some of these reduced factor blocks that contribute to the current assembly block end up in the linked lists of other assembly blocks. 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 3.5 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 is either a leaf of the elimination tree, or it has one or more children. It has 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 of size

	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

TABLE 3.2
Test matrices and their basic information.

t is to recursively invoke itself for all the children assembly blocks (lines 2–4). It then assembles the row and column index sets I_j^{AL} and I_j^{AU}, followed by actually constructing the 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} contain 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. Figures 3.3 and 3.4 illustrate the computation of contribution matrices from the contributing reduced factor blocks and their assimilation into the assembly block. 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. Next, the assembly block is factored in units of its 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 in which 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 3.2 lists the matrices used in our experiments. Most of these matrices are from the University of Florida sparse matrix collection [7]. The remaining ones are from some of the applications that currently use WSMP's general sparse direct solver [24]. The experimental setup is described in Section 1. Recall that we use

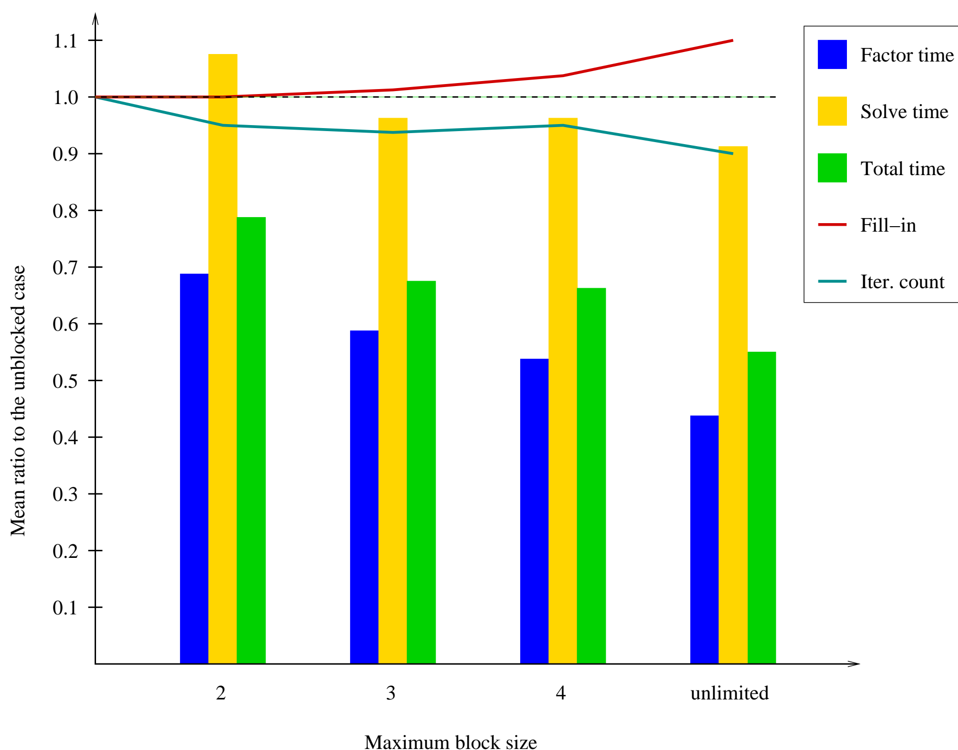

FIG. 3.6. *Some performance metrics as functions of maximum block size relative to the unblocked case.*

Morgan's GMRES variant [39] 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.

For our first experiment, we solved systems using the matrices in Table 3.2 and right-hand side vectors containing all ones. Other than the maximum 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 *et al.*'s ILUC algorithm [34]. Note that WSMP would still use assembly blocks of size 10 in this case, and therefore, is likely to be faster than completely unblocked ILUC. We then solved the systems with the maximum factor block size set to 2, 3, 4, and "unlimited." The assembly block size is 10 times the size of the maximum factor block or 40, whichever is smaller. In the "unlimited" case, the factor block sizes are limited by the sizes of the natural cliques in the graphs of the matrices. This is the default setting in WSMP. GMRES(100,9) was used with at least 100 subspace vectors and at most 9 approximate eigenvectors. The default for the maximum number of approximate eigenvectors to be added to the subspace is set to $\sqrt{m} - 1$ in WSMP. 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 3.6 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 3.6. During incomplete factorization, blocking helps 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 non-blocked 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 [9, 10], 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 3.3 and 3.4 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 a given problem. This means that blocking may have the potential to reduce the overall solution time by more than what Figure 3.6 shows if both BILUC and IIUC use different drop tolerance values.

In our next set of experiments, we compare incomplete factorization times with and without blocking for different drop tolerance values. For these experiments, we chose seven drop tolerance values in the range of 10^{-2} and 10^{-5}. For each of these

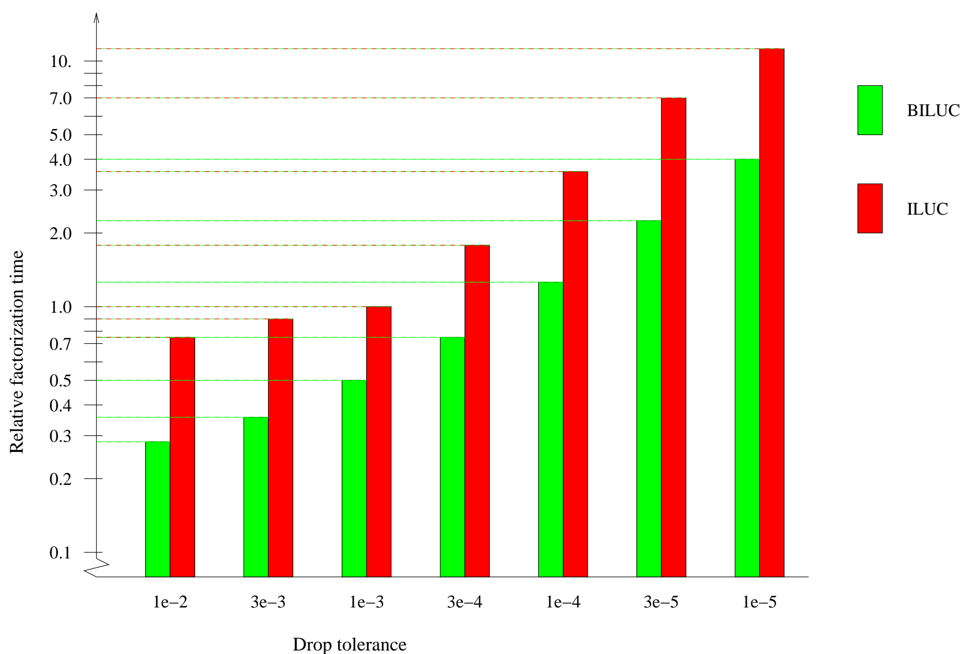

FIG. 3.7. *Relative factorization times of ILUC and BILUC algorithms as functions of drop tolerance.*

drop tolerance values, Figure 3.7 shows the average of the 16 matrices' incomplete factorization times of ILUC and BILUC algorithms normalized with respect to the ILUC time with drop tolerance of 10^{-3}. The results show that BILUC is typically 2–3 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 and order of magnitude lower and still compute an incomplete factorization in about the same time as ILUC.

A smaller drop tolerance 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 [44], 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

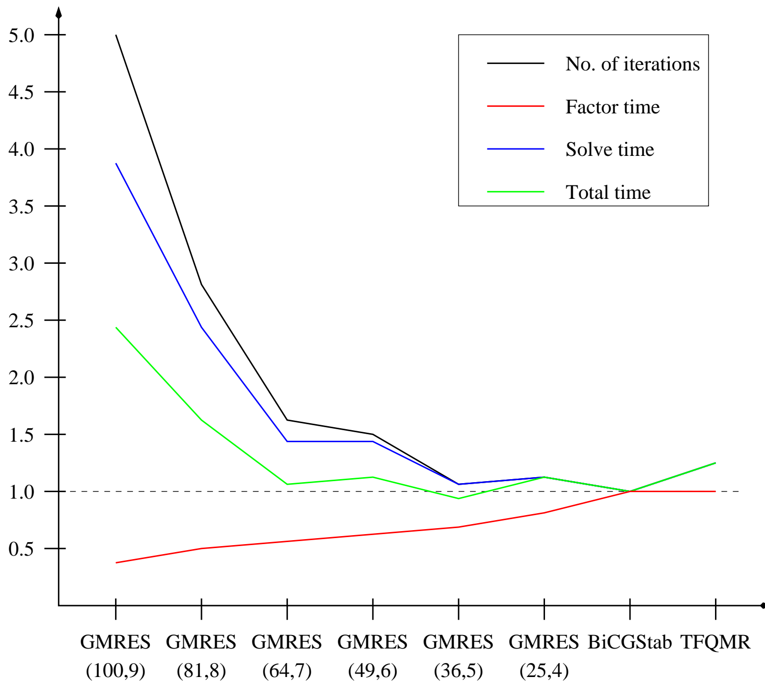

FIG. 3.8. *Relative performance of GMRES variants, BiCGStab, and TFQMR under constant total preconditioner and solver memory.*

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

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

In the last set of experiments for this section, we 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 GMRES restart parameter and the drop tolerance (and therefore, the effective γ) according to Equation 3.4 so that the total memory requirement stayed constant. 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 Equations 3.3—3.5. In WSMP, $l = \sqrt{k}$ by default.

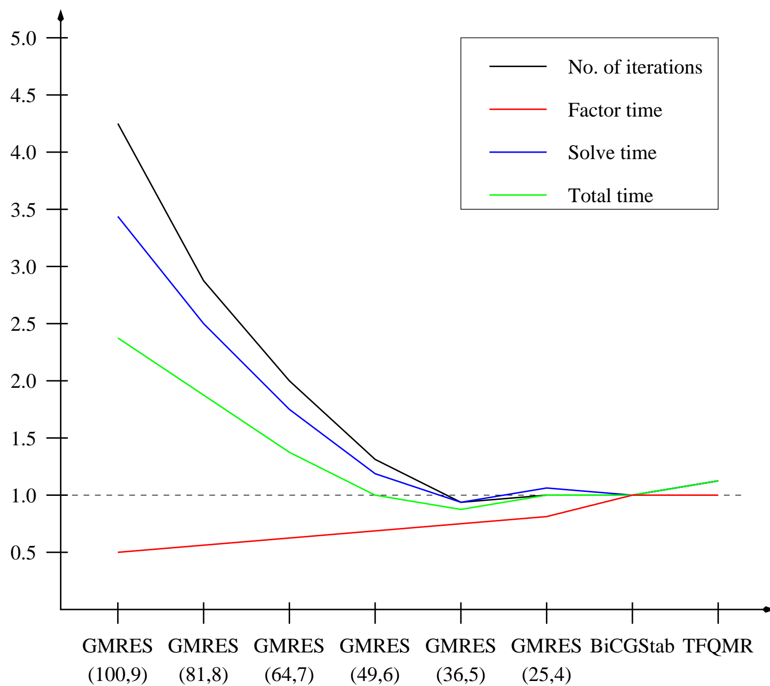

FIG. 3.9. *Relative performance of GMRES variants, BiCGStab, and TFQMR under constant total preconditioner and solver memory.*

In addition to GMRES with different restart parameters, our experiments also include short recurrence methods such as BiCGStab [46] and TFQMR [16] 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. We conducted two sets of experiments with a fixed overall memory. In the first set, for each matrix, we determined the drop tolerance that (roughly) led to the best overall time for BiCGStab, 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 while increasing the drop tolerance such that γ satisfied Equation 3.5. The results were normalized with respect to BiCGStab results. Figure 3.8 shows 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 sparse preconditioners and large restart parameter values.

In our second 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 Equation 3.5. The results of this experiment are shown in Figure 3.9. 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 used 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 because very small subspaces are ineffective in advancing the GMRES algorithm even with a good preconditioner. 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 [29]. 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 a 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 world problem is unlikely to result in pivot growth anywhere close to the worst case bounds; however, it is reasonable to expect that a lower 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. 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 2.1 that if the coefficient matrix is insufficiently diagonally dominant, then the first preprocessing step is to reorder it via an unsymmetric permutation based on maximum weight bipartite matching [27] to maximize the magnitude of the product of the diagonal entries. Let $\mathcal{D}_i^R = |A_{ii}| / \sum_{j \neq i} |A_{ij}|$ be the measure of diagonal dominance of row i and $\mathcal{D}_i^C = |A_{ii}| / \sum_{j \neq i} |A_{ji}|$ be the measure of diagonal dominance of column i of the coefficient matrix. The unsymmetric permutation in Step 1 of the algorithm in Figure 2.1 is performed if the minimum of all $\mathcal{D}_i^R$ and $\mathcal{D}_i^C$ is less than 0.1 or if their geometric mean is less than 0.25. If the matrix is reordered, then $\mathcal{D}_i^R$ and $\mathcal{D}_i^C$ values are recomputed for all $0 < i \leq n$ after the reordering.

Next, we compare the product of $\min(\mathcal{D}_i^R)$ and the geometric mean of all $\mathcal{D}_i^R$s with the product of $\min(\mathcal{D}_i^C)$ and the geometric mean of $\mathcal{D}_i^C$s. If the former is smaller, then we proceed with the conventional incomplete LU 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 code, we factor A^T instead of A, effectively interchanging columns of A for pivoting when needed. To summarize, we use the same

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	57	70415	7.3	5.9	T*	**37**	**63491**	**6.4**	**3.3**
2	**8**	**4685**	**.33**	**.04**	N*	9	4779	.34	0.6
3	**49**	**17702**	**5.6**	**1.9**	N*	fail	-	-	-
4	**23**	**4311**	**.38**	**.19**	T	25	5510	.57	.31
5	20	29937	12.3	.95	T*	**27**	**30423**	**11.6**	**1.3**
6	44	78981	8.3	5.1	T*	**41**	**67060**	**6.2**	**4.3**
7	fail	-	-	-	T*	**27**	**13971**	**1.1**	**1.2**
8	12	27740	7.6	.57	T*	**12**	**24579**	**6.7**	**.48**
9	**195**	**12903**	**.75**	**4.6**	N*	260	12790	.72	5.1
10	6	27496	9.2	.82	T*	**4**	**26269**	**8.4**	**.56**
11	22	16411	1.12	.81	N	**22**	**16273**	**1.12**	**.81**
12	8	11557	.59	.15	T*	**8**	**11880**	**.58**	**.15**
13	17	41178	4.5	1.3	N*	17	41178	4.5	1.3
14	**171**	**493052**	**94.**	**184.**	T	232	533310	101.	258.
15	fail	-	-	-	T*	**171**	**44158**	**3.90**	**30.4**
16	23	126363	18	3.5	N	**29**	**115984**	**14.5**	**4.4**

TABLE 4.1
Impact of selective transposition on number of GMRES iterations, incomplete factor size, factorization time, and solution time for matrices in Table 3.2.

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 [45] 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 selectively performing row equilibration, followed by standard incomplete LU factorization with column 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 that follows.

4.2. Experimental results. Table 4.1 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. 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 a *. The results show that in 12 out of the 16 cases, our heuristic made the correct choice. Moreover, it averted all three failures, and 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.

The test matrices used in Table 4.1 come from a variety of applications, and in many cases, the difference between rowwise and columnwise diagonal dominance in the coefficient matrix is insignificant. In order to demonstrate the effectiveness of selective transposition more conclusively, we gathered another set of matrices from an elctro-thermal semiconductor device simulation tool FIELDAY [3], which solves

	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

TABLE 4.2
FIELDAY matrices and their basic information.

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	408	3073	0.13	1.96	T*	**158**	**2988**	**0.12**	**0.72**
2	fail	-	-	-	T*	**40**	**8475**	**0.95**	**1.60**
3	**171**	**493052**	**94.1**	**184.**	T	232	533310	99.6	257.
4	**16**	**18476**	**1.26**	**1.05**	T	20	18543	1.27	1.46
5	fail	-	-	-	T*	**171**	**44158**	**3.82**	**30.5**
6	fail	-	-	-	T*	**177**	**89700**	**8.13**	**63.2**
7	**28**	**11675**	**.79**	**1.04**	N*	29	11766	0.80	1.03
8	fail	-	-	-	T*	**101**	**26117**	**2.13**	**10.4**
9	13	2508	0.18	0.09	T*	**13**	**2502**	**0.18**	**0.08**
10	fail	-	-	-	T*	**49**	**12784**	**1.42**	**2.87**

TABLE 4.3
Impact of selective transposition on number of GMRES iterations, incomplete factor size, factorization time, and solution time of FIELDAY problems.

6 coupled PDEs governing carrier mass and energy transport in 3D semiconductor structures. The details of these matrices can be found in Table 4.2. We chose this application because it often tends to generate matrices for which transposition is critical for the success of ILU preconditioning.

Table 4.3 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 [45]. However, a reliable predictor 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. Concluding Remarks and Future Work. We have introduced techniques to improve the reliability and performance of incomplete LU factorization-based preconditioners for solving general sparse systems of linear equations. Along with its sister publication [26] 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 [26]. 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. We hope that this block framework, along with other similar recently proposed blocking approaches [35, 41], would have a similar effect on the development of future incomplete factorization algorithms and software that supernodal [17] and multifrontal [14, 37] techniques had on complete factorization.

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 3.7), 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.

The selective transposition heuristic proposed in this paper could extend the results of Almeida et al. [8], 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 [45] on the relation between the direction of pivoting and the type of equilibration, we think that it is possible give more precise guidance on how to use scaling beneficially while factoring general sparse matrices. It would be interesting to investigate if an effective a priori decision between row or column equilibration can be made using the same heuristic that we use for deciding whether or not to transpose the matrix prior to factorization.

One of the goals of this work was to produce an industrial strength iterative solver for general systems that can reliably replace a direct solver in many real applications, particularly those in which iterative solvers are not traditionally used due to robustness or performance concerns. The software can be downloaded from *http://www.research.ibm.com/projects/wsmp* for testing and benchmarking. Enhancing and extending the block framework to distributed-memory platforms is a key piece of future work.

Acknowledgements. 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] Michele Benzi. Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics*, 182(2):418–477, 2002.
[2] Matthias Bollhöfer. A robust and efficient ILU that incorporates the growth of the inverse triangular factors. *SIAM Journal on Scientific Computing*, 25(1):86–103, 2003.
[3] 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.
[4] Edmond Chow and Michael A. Heroux. An object-oriented framework for block preconditioning. *ACM Transactions on Mathematical Software*, 24(2):159–183, 1998.
[5] Edmond Chow and Yousef Saad. Experimental study of ILU preconditioners for indefinite matrices. *Journal of Computational and Applied Mathematics*, 86(2):387–414, 1997.
[6] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference of the ACM*, pages 152–172, 1969.

[7] Timothy A. Davis. The university of Florida sparse matrix collection. Technical report, Department of Computer Science, University of Florida, Jan 2007.

[8] Valmor F. de Almeida, Andrew M. Chapman, and Jeffrey J. Derby. On equilibration and sparse factorization of matrices arising in finite element solutions of partial differential equations. *Numerical Methods for Partial Differential Equations*, 16(1):11–29, 2000.

[9] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[10] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[11] Iain S. Duff, Albert M. Erisman, and John K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, Oxford, UK, 1990.

[12] Iain S. Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2001.

[13] Iain S. Duff and G. A. Meurant. The effect of ordering on preconditioned conjugate gradient. *BIT Numerical Mathematics*, 29:635–657, 1989.

[14] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9(3):302–325, 1983.

[15] Qing Fan, Peter A. Forsyth, J. R. F. McMacken, and Wei-Pai Tang. Performance issues for iterative solvers in device simulation. *SIAM Journal on Scientific Computing*, 17(1):100–117, 1996.

[16] Roland W. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM Journal on Scientific and Statistical Computing*, 14(2):470–482, 1993.

[17] Alan George and Joseph W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, NJ, 1981.

[18] Thomas George, Anshul Gupta, and Vivek Sarin. An empirical analysis of iterative solver performance for SPD systems. *ACM Transactions on Mathematical Software*, 38(4), 2012. A detailed version available as RC 24737, IBM T. J. Watson Research Center, 2009.

[19] Thomas George, Anshul Gupta, and Vivek Sarin. An experimental evaluation of iterative solvers for large SPD systems of linear equations. In *10th Copper Mountain Conference on Iterative Methods*, April 2008. Available at *http://www.cs.umn.edu/~agupta/doc/copper08.pdf.*

[20] John R. Gilbert and Sivan Toledo. An assessment of incomplete-LU preconditioners for nonsymmetric linear systems. *Informatica*, 24(3):409–425, 2000.

[21] Anshul Gupta. Improving performance and robustness of incomplete factorization preconditioners. plenary talk at SIAM Conference on Applied Linear Algebra (SIAM LA), Valencia, Spain.

[22] Anshul Gupta. Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 24(2):529–552, 2002.

[23] Anshul Gupta. Fast and effective algorithms for graph partitioning and sparse matrix ordering. *IBM Journal of Research and Development*, 41(1/2):171–183, January/March, 1997.

[24] Anshul Gupta. WSMP: Watson sparse matrix package (Part-II: Direct solution of general systems). Technical Report RC 21888, IBM T. J. Watson Research Center, Yorktown Heights, NY, November 2000. *http://www.research.ibm.com/projects/wsmp.*

[25] Anshul Gupta. WSMP: Watson sparse matrix package (Part-III: Iterative solution of sparse systems). Technical Report RC 24398, IBM T. J. Watson Research Center, Yorktown Heights, NY, November 2007. *http://www.research.ibm.com/projects/wsmp.*

[26] Anshul Gupta and Thomas George. Adaptive techniques for improving the performance of incomplete factorization preconditioning. *SIAM Journal on Scientific Computing*, 32(1):84–110, 2010.

[27] Anshul Gupta and Lexing Ying. On algorithms for finding maximum matchings in bipartite graphs. Technical Report RC 21576, IBM T. J. Watson Research Center, Yorktown Heights, NY, October 1999.

[28] Pascal Hénon, Pierre Ramet, and Jean Roman. On finding approximate supernodes for an efficient block-ILU(k) factorization. *Parallel Computing*, 34(6-8):345–362, 2008.

[29] Nicholas J. Higham and Desmond J. Higham. Large growth factors in Gaussian elimination with pivoting. *SIAM Journal on Matrix Analysis and Applications*, 10(2):155–164, 1989.

[30] David Hysom and Alex Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM Journal on Scientific Computing*, 22(6):2194–2215, 2000.

[31] Mark T. Jones and Paul E. Plassmann. Blocksolve95 users manual: Scalable library software for the parallel solution of sparse linear systems. Technical Report ANL-95/48, Argonne National Laboratory, Argonne, IL, 1995.

[32] Igor E. Kaporin, L. Yu. Kolotilina, and A. Yu. Yeremin. Block SSOR preconditionings for high-order 3D FE systems. II Incomplete BSSOR preconditionings. *Linear Algebra and its Applications*, 154-156:647–674, 1991.

[33] George Karypis and Vipin Kumar. Parallel threshold-based ILU factorization. Technical Report TR 96-061, Department of Computer Science, University of Minnesota, 1996.

[34] Na Li, Yousef Saad, and Edmond Chow. Crout versions of ILU for general sparse matrices. *SIAM Journal on Scientific Computing*, 25(2):716–728, 2003.

[35] Xiaoye S. Li and Meiyue Shao. A supernodal approach to incomplete LU factorization with partial pivoting. *ACM Transactions on Mathematical Software*, 37(4), 2011.

[36] Joseph W.-H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[37] Joseph W.-H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, 1992.

[38] Joseph W.-H. Liu, Esmond G.-Y. Ng, and Barry W. Peyton. On finding supernodes for sparse matrix computations. *SIAM Journal on Matrix Analysis and Applictations*, 14:242–252, 1993.

[39] Ronald B. Morgan. A restarted GMRES method augmented with eigenvectors. *SIAM Journal on Matrix Analysis and Applications*, 16(4):1154–1171, 1995.

[40] N. Munksgaard. Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients. *ACM Transactions on Mathematical Software*, 6(2):206–219, 1980.

[41] Esmond G.-Y. Ng, Barry W. Peyton, and Padma Raghavan. A blocked incomplete cholesky preconditioner for hierarchical-memory computers. In D. R. Kincaid and A. C. Elster, editors, *Iterative Methods in Scientific Computation IV, IMACS Series in Computational and Applied Mathematics*, pages 211–221. Elsevier, 1999.

[42] Yousef Saad. ILUT: A dual threshold incomplete *LU* factorization. *Numerical Linear Algebra with Applications*, 1(4):387–402, 1994.

[43] Yousef Saad. *Iterative Methods for Sparse Linear Systems, 2nd edition*. SIAM, 2003.

[44] Yousef Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving non-symmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7:856–869, 1986.

[45] Robert D. Skeel. Effect of equilibration on residual size for partial pivoting. *SIAM Journal on Numerical Analysis*, 18(3):449–454, 1981.

[46] Henk A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.