SIAM J. SCI. COMPUT.
Vol. 25, No. 2, pp. 716–728

CROUT VERSIONS OF ILU FOR GENERAL SPARSE MATRICES*

NA LI†, YOUSEF SAAD†, AND EDMOND CHOW‡

Abstract. This paper presents an efficient implementation of the incomplete LU (ILU) factorization derived from the Crout version of Gaussian elimination. At step k of the elimination, the kth row of U and the kth column of L are computed using previously computed rows of U and columns of L. The data structure and implementation borrow from already known techniques used in developing both sparse direct solution codes and incomplete Cholesky factorizations. This version of ILU can be computed much faster than standard threshold-based ILU factorizations computed rowwise or columnwise. In addition, the data structure allows efficient implementations of more rigorous and effective dropping strategies.

Key words. incomplete LU factorization, ILU, sparse Gaussian elimination, Crout factorization, preconditioning, ILU with threshold, ILUT, iterative methods, sparse linear systems

AMS subject classifications. 65F10, 65N06

DOI. 10.1137/S1064827502405094

1. Introduction.

The rich variety of existing Gaussian elimination algorithms has often been exploited to extract the most efficient variant for a given computer architecture. As noted in [11], these variants can be unraveled from the orderings of the three main loops in Gaussian elimination. A short overview here serves the purpose of introducing notation. Gaussian elimination is often presented in the following form:

```
1.    for   k = 1 : n − 1,
2.          for   i = k + 1 : n,
3.                for   j = k + 1 : n,
4.                      a_ij = a_ij − a_ik ∗ a_kj,
```

where some calculations (e.g., pivots) have been omitted for simplicity. This is an "outer product" form and will be referred to as the KIJ version, due to the ordering of the three loops. Swapping the first and second loops results in the IKJ version, we get the following:

```
1.    for   i = 2 : n,
2.          for   k = 1 : i − 1,
3.                for   j = k + 1 : n,
4.                      a_ij = a_ij − a_ik ∗ a_kj,
```

which is often referred to as the "delayed-update" version, and sometimes the Tinney–Walker algorithm (see, e.g., [6]). There is also a column variant of the delayed-update version, which is the JKI version of Gaussian elimination.

*Received by the editors April 8, 2002; accepted for publication (in revised form) December 10, 2002; published electronically November 11, 2003. This work was supported by the Army Research Office under grant DAAD19-00-1-0485, in part by the NSF under grants NSF/ACI-0000443 and NSF/INT-0003274, and by the Minnesota Supercomputer Institute.

http://www.siam.org/journals/sisc/25-2/40509.html

†Department of Computer Science and Engineering, University of Minnesota, 200 Union Street S.E., Minneapolis, MN 55455 (nli@cs.umn.edu, saad@cs.umn.edu).

‡Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, L-560, Box 808, Livermore, CA 94551 (echow@llnl.gov). The work of this author was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract W-7405-Eng-48.

Bordering methods can be viewed as modifications of the delayed-update methods. The loop starting at line 3 of the above IKJ version is shortened into "for $j = k+1 : i-1$" which computes the kth row of L. A similar loop is then added to compute the kth *column* of U. These different implementations of Gaussian elimination all yield the same (complete) factorization in exact arithmetic. However, their computational patterns rely on different matrix kernels, and this gave rise to specialized techniques for different computer architectures. In the context of incomplete factorizations, these variants result in important practical differences.

Incomplete LU (ILU) factorizations can be derived from any of these variants (see, e.g., [14, 3, 2]), although the IKJ version has often been preferred because it greatly simplifies the data structure required by the implementation. The KIJ algorithm, for example, requires rank-one updates and thus up to $n-k$ rows and columns need to be altered at step k. Since the matrices are stored in sparse mode, this is not an efficient way to handle the fill-ins introduced during the factorization [6]. ILU factorizations based on bordering have also been proposed [5].

This paper considers ILU factorizations based on a version of Gaussian elimination known as the Crout variant, which can be seen as a combination of the IKJ algorithm shown above to compute the U factor and a transposed version to compute the L factor. The kth step will therefore compute the pieces $U(k, k : n)$ and $L(k : n, k)$ of the factorization. This version of Gaussian elimination was used in the Yale sparse matrix package (YSMP) [7] to develop sparse Cholesky factorizations. More recently, the Crout version was also used to develop an efficient incomplete Cholesky factorization in [10]. The current paper extends this method to nonsymmetric matrices and explores effective dropping strategies that are enabled by the Crout variant.

2. The Crout LU and ILU. The main disadvantage of the standard delayed-update IKJ factorization is that it requires access to the entries in the current row of L by topological order [8]. One topological order, which is perhaps most appropriate for incomplete factorizations where the nonzero pattern of the factorization is not known beforehand, is increasing order by column number. Searches in the current row must be used to find the entries in this order. These searches are complicated by the fact that the current row is dynamically being modified by the fill-in process. In SPARSKIT [12], a simple linear search is used, which is suitable in the case of small amounts of fill-in. When large amounts of fill-in are required, which is the case for more difficult problems, the cost of searching for the leftmost pivot may make the factorization too expensive. An alternative is to maintain the entries in the current row in a binary tree and to utilize binary searches. This strategy was mentioned in [14] and was recently implemented by Bollhöfer in ILUT and ILUTP [4]. This code will be used for comparisons later in this paper.

2.1. The Crout formulation. The Crout formulation can be viewed as yet another "delayed-update" form of Gaussian elimination. At step k of the following algorithm, the entries $a_{k+1:n,k}$ (in the unit lower triangular factor, L) and $a_{k,k:n}$ (in the upper triangular factor, U) are computed and the rank-one update which characterizes the KIJ version is postponed. At the kth step, all the updates of the previous steps are applied to the entries $a_{k+1:n,k}$ and $a_{k,k:n}$. Thus it is natural to store L by columns and U by rows, and to have the lower and upper triangular parts of A stored similarly. The computational pattern for the factorization is shown in Figure 1.

ALGORITHM 2.1. *Crout LU Factorization*

1. *For* $k = 1 : n$ *Do :*

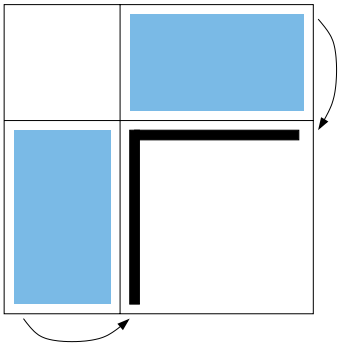

FIG. 1. *Computational pattern of the Crout factorization. The dark area shows the parts being computed at the kth step. The shaded areas show the parts being accessed at the kth step.*

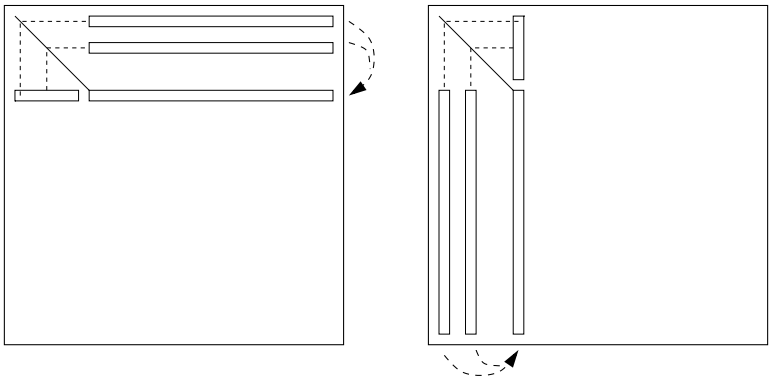

FIG. 2. *Construction of the kth row of U (left) and the k-column of L (right).*

2. *For* $i = 1 : k-1$ *and if* $a_{ki} \neq 0$ *Do :*
3. $a_{k,k:n} = a_{k,k:n} - a_{ki} * a_{i,k:n}$
4. *EndDo*
5. *For* $i = 1 : k-1$ *and if* $a_{ik} \neq 0$ *Do :*
6. $a_{k+1:n.k} = a_{k+1:n,k} - a_{ik} * a_{k+1:n,i}$
7. *EndDo*
8. $a_{ik} = a_{ik}/a_{kk}$ *for* $i = k+1, ..., n$
9. *EndDo*

The kth step of the algorithm generates the kth row of U and the kth column of L. This step is schematically represented in Figure 2. Notice now that the updates to the kth row of U (resp., the kth column of L) can be made in any order. There is also a symmetry in the data structure representing L and U since the U matrix is accessed by rows and the L matrix is accessed by columns. By adapting Algorithm 2.1 for sparse computations and by adding a dropping strategy, the following Crout version of ILU (denoted ILUC) is obtained.

ALGORITHM 2.2. *ILUC—Crout version of ILUC*

1. *For* $k = 1 : n$ *Do :*
2. *Initialize row* z: $z_{1:k-1} = 0, \quad z_{k:n} = a_{k,k:n}$
3. *For* $\{i \mid 1 \leq i \leq k-1$ *and* $l_{ki} \neq 0\}$ *Do :*

4. $z_{k:n} = z_{k:n} - l_{ki} * u_{i,k:n}$
5. *EndDo*
6. *Initialize column w:* $w_{1:k} = 0, \quad w_{k+1:n} = a_{k+1:n,k}$
7. *For* $\{i \mid 1 \le i \le k-1$ *and* $u_{ik} \ne 0\}$ *Do :*
8. $w_{k+1:n} = w_{k+1:n} - u_{ik} * l_{k+1:n,i}$
9. *EndDo*
10. *Apply a dropping rule to row z*
11. *Apply a dropping rule to column w*
12. $u_{k,:} = z$
13. $l_{:,k} = w/u_{kk}, \quad l_{kk} = 1$
14. *Enddo*

The operations in lines 4 and 8 are sparse vector updates and must be performed in sparse mode.

2.2. Implementation. There are two potential sources of difficulty in the sparse implementation of the algorithm just described.

1. Consider lines 4 and 8. Only the section $(k : n)$ of the ith row of U is required, and similarly only the section $(k + 1 : n)$ of the ith column of L is needed. Accessing entire rows of U or columns of L and then extracting only the desired part is an expensive option.
2. Consider lines 3 and 7. The nonzeros in row k of L must be accessed easily, but L is stored by columns. Similarly, the nonzeros in column k of U must be accessed easily, but U is stored by rows.

A solution to these difficulties was presented for the symmetric case in [7] and later in [10]. Here we extend this technique to nonsymmetric problems. The extension is straightforward, except that we can no longer use the optimizations available when L and U have the same nonzero pattern.

To address the first difficulty, consider the factor U and assume its nonzeros in each row are stored in order by column number. Then a pointer for row j, with $j < k$, can be used to signal the starting point of row j needed to update the current row k. The pointers for each row are stored in a pointer array called $Ufirst$. This pointer array is updated after each elimination step by incrementing each pointer to point to the next nonzero in the row, if necessary. A pointer for row k is also added after the kth step. There is a similar pointer array for the L factor called $Lfirst$.

To address the second difficulty, consider again the factor U, and the need to traverse column k of U, although U is stored by rows. An implied linked list for the nonzeros in column k of U is used, called $Ulist$. $Ulist(k)$ contains the first nonzero in column k of U, and $Ulist(Ulist(k))$ contains the next nonzero, etc. At the end of step k, $Ulist$ is updated so that it becomes the linked list for column $k + 1$. $Ulist$ is updated when $Ufirst$ is updated: when $Ufirst(i)$ is incremented to point to a nonzero with column index c, then i is added to the linked list for column c. For the L factor, there is a linked list called $Llist$.

In summary, we use four arrays of length n, $Ufirst$, $Ulist$, $Lfirst$, and $Llist$, which constitute what we call a *bi-index* structure. Figure 3 illustrates the relationship between the arrays in the bi-index structure.

a. $Ufirst(i)$ points to the first entry with column index greater than or equal to k in row i of U, where $i = 1, \ldots, k-1$;
b. $Ulist(k)$ points to a linked list of rows that will update column k;
c. $Lfirst(i)$ points to the first entry with row index greater than or equal to k in column i of L, where $i = 1, \ldots, k-1$;

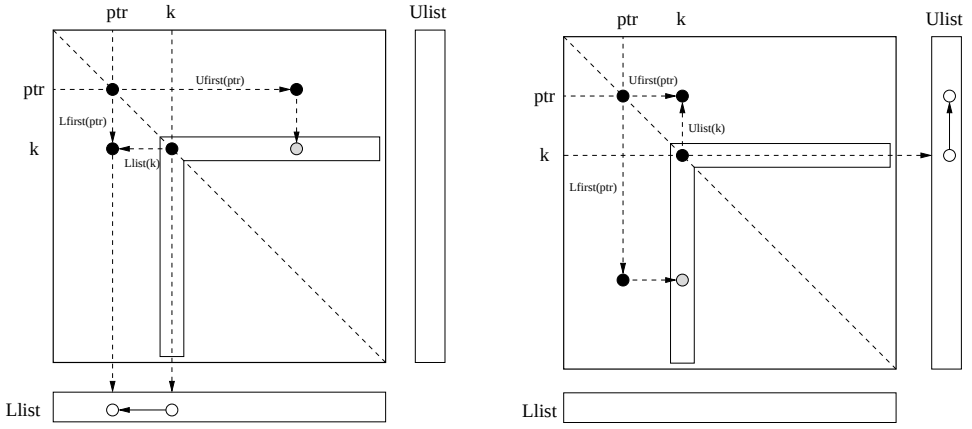

FIG. 3. *Procedure for updating row k of U and column k of L.*

d. $Llist(k)$ points to a linked list of columns that will update row k.

In [10] the entire diagonal of the LDL^T factorization is updated at the end of each elimination step. In contrast, Eisenstat, Schultz, and Sherman [7] update only the kth entry of D at the kth step. In the symmetric case, there is no additional cost incurred in updating all the entries in D. In the nonsymmetric case, this update, which can be written as

$$u_{i,i} := u_{i,i} - l_{i,k}u_{k,i}, \quad i = k+1, \ldots, n,$$

may increase the computational cost slightly because $l_{:,k}$ and $u_{k,:}$ do not in general have the same pattern. However, the option of having the entire updated diagonal at each step may be attractive when developing other possible variants of the algorithm.

3. Dropping strategies. Any dropping rule can be applied in lines 10 and 11 of Algorithm 2.2. In this section, two options are described.

3.1. Standard dual criterion dropping strategy. The dual criterion dropping strategy, similar to the one in ILUT [13, 14], consists of the following two steps.

1. Any element of L or U whose magnitude is less than a tolerance τ (relative to the norm of the kth column of L or the kth row of U, respectively) is dropped.
2. Then only the "Lfil" largest elements in magnitude in the kth column of L are kept. Similarly, the "Lfil" largest elements in the kth row of U in addition to the diagonal element are kept. This controls the total storage that can be used by the preconditioner.

3.2. Dropping based on condition number estimators. In order to reduce the impact of dropping an element on the subsequent steps of an incomplete factorization, it is useful to devise dropping strategies that estimate and utilize the norms of the rows of L^{-1} and the columns of U^{-1}. Such techniques were shown to be effective by Bollhöfer [1]. The ILUC data structure allows this important option to be easily implemented; it is not practically feasible with standard IKJ implementations or their column-based equivalents.

The guiding criterion is to drop an entry l_{jk} at step k when it satisfies

$$|l_{jk}| \ \|e_k^{\mathrm{T}} L^{-1}\| \leq \epsilon,$$

where e_k denotes the kth unit vector, and ϵ is the ILU drop tolerance. A similar criterion is used for the U part: drop u_{kj} when $|u_{kj}|\|U^{-1}e_k\| \leq \epsilon$. In what follows we discuss only the strategy for the L part. The justification for this criterion given in [1] was based on exploiting the connection with the approximate inverse. Here we use a similar, although somewhat simpler, argument.

It is well known that for ILU preconditioners, the error made in the inverses of the factors is more important to control than the errors in the factors themselves, because when $A = LU$, and

$$\tilde{L}^{-1} = L^{-1} + X \qquad \tilde{U}^{-1} = U^{-1} + Y,$$

then the preconditioned matrix is given by

$$\tilde{L}^{-1}A\tilde{U}^{-1} = (L^{-1} + X)A(U^{-1} + Y) = I + AY + XA + XY.$$

This means that if the errors X and Y in the inverses of L and U are small, then the preconditioned matrix will be guaranteed to be close to the identity matrix. In contrast, small errors in the factors themselves may yield arbitrarily large errors in the preconditioned matrix.

Let L_k denote the matrix composed of the first k rows of L and the last $n-k$ rows of the identity matrix. Consider a term l_{jk} with $j > k$ that is dropped at step k. Then the resulting perturbed matrix $\tilde{L}_k$ differs from L_k by $l_{jk}e_je_k^T$. Noticing that $L_ke_j = e_j$ we have

$$\tilde{L}_k = L_k - l_{jk}e_je_k^T = L_k(I - l_{jk}e_je_k^T)$$

from which we can obtain the following relation between the inverses:

$$\tilde{L}_k^{-1} = (I - l_{jk}e_je_k^T)^{-1}L_k^{-1} = L_k^{-1} + l_{jk}e_je_k^TL_k^{-1}.$$

Therefore, the inverse of L_k will be perturbed by l_{jk} times the kth row of L_k^{-1}. This perturbation will affect the jth row of L_k^{-1}. Hence, using the infinity norm, for example, it is important to limit the norm of this perturbing row which is $\|l_{jk}e_je_k^TL_k^{-1}\|_\infty = |l_{jk}|\ \|e_k^TL_k^{-1}\|_\infty$.

However, the matrix L^{-1} is not available and it is not feasible to compute it. Instead, in [1] standard techniques used for estimating condition numbers [9] are adapted for estimating the norm of the kth row of L^{-1} (resp., kth column of U^{-1}). In this paper we use only the simplest of these techniques. The idea is to construct a vector b with entries $+1$ or -1, by following a greedy strategy to try to make $L^{-1}b$ large at each step. Since the first k columns of L are available, this is easy to achieve. The problem to estimate $\|e_k^TL^{-1}\|_\infty$ can be reduced to that of dynamically constructing a right-hand side b to the linear system $Lx = b$ so that the kth component of the solution is the largest possible. Thus, if b is the current right-hand side at step k, we write

$$\|e_k^{\mathrm{T}}L^{-1}\|_\infty \approx \tfrac{\|e_k^{\mathrm{T}}L^{-1}b\|_\infty}{\|b\|_\infty},$$

where $\|e_k^{\mathrm{T}}L^{-1}\|$ was estimated as the kth component of the solution x of the system $Lx = b$. The implementation given next uses the simplest criterion which amounts to selecting $b_k = \pm 1$ at each step k, in such a way as to maximize the norm of the kth component of $L^{-1}b$. The notation for the algorithm is as follows. At the k step we have available the first $k-1$ columns of L. The kth component of the solution x is

$$\xi_k = b_k - e_k^TL_{k-1}x_{k-1}.$$

This makes the choice clear: if ξ_k is to be large in modulus, then its sign should be opposite that of $e_k^T L_{k-1} x_{k-1}$. Once b_k is selected, x_k is then known and all the $e_j^T L_k x_k$ are updated. These scalars are called ν_j below. Details may be found in [9].

ALGORITHM 3.1. *Estimating the norms* $\|e_k^T L^{-1}\|_\infty$

1. *Set* $\xi_1 = 1$, $\nu_i = 0, i = 1, \ldots, n$
2. *For* $k = 2, \ldots, n$ *do*
3. $\quad \xi_+ = 1 - \nu_k$; $\xi_- = -1 - \nu_k$;
4. $\quad$ *if* $|\xi_+| > |\xi_-|$ *then* $\xi_k = \xi_+$ *else* $\xi_k = \xi_-$
5. $\quad$ *For* $j = k+1 : n$ *and for* $l_{jk} \neq 0$ *Do*
6. $\quad\quad \nu_j = \nu_j + \xi_k l_{jk}$
7. $\quad$ *EndDo*
8. *EndDo*

The paper [1] also presents an improved variant of this algorithm which is also derived from a dense version described in [9]. In this variant, the ξ_k's are selected to encourage growth not only in the solution x_k but also in the ν_i's. Calling p_j the vector with components ν_i at step j, this is achieved by using as a criterion for selecting ξ_k the weight

$$|\xi_k| + \|p_k\|_1 \tag{1}$$

which depends on the choice made for ξ_k. Note that $\|p_k\|_1 = \|p_{k-1} + \xi_k l_{:,k}\|_1$, so we need to compute the weight (1) for both of the choices in line 3 and select the choice that gives the largest weight.

4. Experimental results. The performance of ILUC was compared with various implementations of the standard ILU with threshold (ILUT) [13]. The computional codes were written in C, and the experiments were conducted on a 866 MHz Pentium III computer with 1 GB of main memory. All codes were compiled with the -O3 optimization option.

Ten nonsymmetric test matrices were used, five of which have a symmetric pattern and the other five a nonsymmetric pattern. The degree of structural symmetry of the matrices can be described using a measure called the relative symmetry match (RSM) [12]. This measures the total number of matches between $a_{ij} \neq 0$ and $a_{ji} \neq 0$ divided by the total number of nonzero elements (RSM = 1 for matrices with symmetric patterns). Some generic information on these matrices is given in Table 1, where n is the dimension of the matrix and nnz the total number of nonzero elements. All matrices are available from the Matrix Market[1] except BARTH1A and CAVA0000.[2] The BARTHT1A matrix was supplied by T. Barth of NASA Ames. CAVA0000 is the first of a sequence of matrices resulting from the simulation of a flow in a driven cavity. This matrix was obtained using 40 quadrilateral elements in each direction, with biquadratic functions for velocities and linear (discontinuous) functions for pressures. The resulting linear systems are indefinite and can be difficult to solve. Though the Reynolds number associated with the physical problem is zero, the resulting matrix is nonsymmetric due to the discretization used. Note that the matrix LNS3937 was also used as a test matrix in [1].

Artificial right-hand sides were generated, and GMRES(60) was used to solve the systems using a zero initial guess. The iterations were stopped when the residual norm was reduced by 8 orders of magnitude or when the maximum iteration count of

[1] http://math.nist.gov/MatrixMarket/

[2] These matrices are available from the authors.

TABLE 1
Information on the 10 matrices used for tests.

Matrix	RSM	n	nnz
BARTHT1A	1.0000	14075	481125
RAEFSKY1	1.0000	3242	294276
RAEFSKY2	1.0000	3242	294276
RAEFSKY3	1.0000	21200	1488768
VENKAT25	1.0000	62424	1717792

Matrix	RSM	n	nnz
UTM.3060	0.5591	3060	42211
UTM.5940	0.5624	5940	83842
SHERMAN5	0.7803	3312	20793
LNS3937	0.8686	3937	25407
CAVA0000	0.9773	17922	567467

TABLE 2
Performance of ILUC and c-ILUT on symmetric pattern matrices, $\tau = 0.001$.

Matrix	Lfil, τ	Pre-alg	Sec_{pre}	Sec_{it}	Sec_{tot}	Its	Ratio
BARTHT1A $\gamma \approx 17.1$	$2.5\gamma \approx 42$	ILUC	1.02	2.56	3.58	32	1.979
	$\tau = 0.01$	c-ILUT	2.19	-	-	-	2.777
	$5.0\gamma \approx 85$	ILUC	2.84	1.92	4.76	16	4.053
	$\tau = 0.001$	c-ILUT	7.88	12.67	20.55	94	4.641
RAEFSKY1 $\gamma \approx 45.4$	$2.5\gamma \approx 113$	ILUC	1.39	0.73	2.12	17	2.013
	$\tau = 0.01$	c-ILUT	2.48	0.74	3.22	17	2.158
	$5.0\gamma \approx 226$	ILUC	8.48	1.08	9.56	15	4.385
	$\tau = 0.001$	c-ILUT	13.16	0.94	14.10	13	4.473
RAEFSKY2 $\gamma \approx 45.4$	$2.5\gamma \approx 113$	ILUC	1.50	0.70	2.20	16	2.111
	$\tau = 0.01$	c-ILUT	2.88	0.75	3.63	17	2.184
	$5.0\gamma \approx 226$	ILUC	8.71	0.95	9.66	13	4.465
	$\tau = 0.001$	c-ILUT	14.44	0.96	15.40	13	4.550
RAEFSKY3 $\gamma \approx 35.1$	$2.5\gamma \approx 87$	ILUC	2.94	2.64	5.58	15	1.227
	$\tau = 0.01$	c-ILUT	5.43	3.51	8.94	17	1.735
	$5.0\gamma \approx 175$	ILUC	14.38	2.55	16.93	10	2.493
	$\tau = 0.001$	c-ILUT	29.98	2.81	32.79	10	2.909
VENKAT25 $\gamma \approx 13.8$	$2.5\gamma \approx 34$	ILUC	5.23	18.63	23.86	47	2.415
	$\tau = 0.01$	c-ILUT	9.90	19.97	29.87	51	2.332
	$5.0\gamma \approx 68$	ILUC	16.10	15.71	31.81	29	4.851
	$\tau = 0.001$	c-ILUT	52.50	17.80	70.30	34	4.544

300 was reached. Diagonal scaling is performed for all matrices except when otherwise indicated. This consists of scaling each row by its 2-norm and, after this is done, to scale each column similarly.

Table 2 compares the times to build the preconditioners ("Sec_{pre}"), the iteration times ("Sec_{it}"), and the total times ("Sec_{tot}") for ILUC and c-ILUT (column version of ILUT) on the matrices from the set that have symmetric patterns. "Lfil" is the dropping parameter described in section 3.1. Analogous results were also obtained with a row version of ILUT (r-ILUT), but they are not reported here, as they were quite similar to those of the column version. The parameter "Lfil" is selected as a multiple of the ratio $\gamma = \frac{nnz}{2n}$, half the average number of nonzero elements per row in the original matrix. Two sets of results are shown for each matrix. The first set of runs uses $\tau = 0.01$ and $\gamma = 2.5$ while the second uses $\tau = 0.001$ and $\gamma = 5$. "Its" denotes the number of iterations to convergence. The symbol "-" in the table indicates that convergence was not obtained in 300 iterations. "Ratio" denotes the fill-factor, i.e., the value of $nnz(L + U)/nnz(A)$, which is a useful indicator of the sparsity of the ILU factors. Notice that the times to compute the ILUC factorization are significantly smaller than those for c-ILUT. The difference is more significant for those factorizations with larger degree of fill.

Figure 4 shows the times required for computing three preconditioners as a function of "Lfil" for the matrix RAEFSKY3. In this particular test, the matrix was not

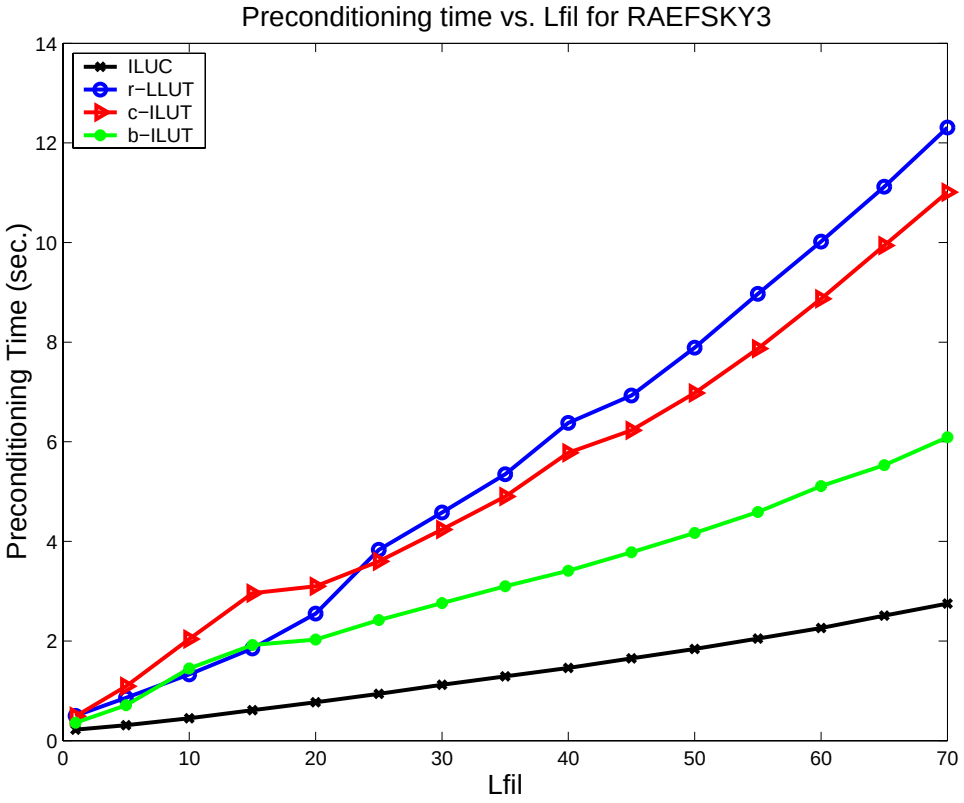

FIG. 4. *Precondition time vs. Lfil for ILUC, r-ILUT, c-ILUT, and b-ILUT* ($\tau = 0.001$).

diagonally scaled. The drop tolerance was $\tau = 0.001$. The figure also shows the times for an IKJ version of ILUT (i.e., r-ILUT) when searching for the leftmost pivot is accomplished using binary search trees. This version of ILUT [4], which is labeled b-ILUT in the figure, was coded in FORTRAN. The figure shows that the ILUC setup is faster than all the other variants, and that this setup time increases more slowly with increasing amounts of fill-in.

Table 3 shows results which are analogous to those of Table 2 for the five matrices in the test set that have nonsymmetric patterns. The time to compute the preconditioner is still generally smaller than with the other versions. ILUC did help GMRES achieve convergence in the case of the matrix UTM.5940. In a few other cases, it caused GMRES to converge slowly or to fail to converge, while r-ILUT and/or c-ILUT yielded good convergence. This is illustrated by the results with UTM.3060 and CAVA0000. For the matrix LNS3973 convergence was not achieved by either method, even when a rather high Lfil was used. As will be seen next, the inverse-based dropping strategy will do much better for this case.

The next tests compare the two dropping strategies described in section 3, namely the standard threshold-based technique (termed "standard") with the technique based on norm estimates of the inverse triangular factors (termed "inverse-based").

Table 4 shows the results for those matrices in the set which have a symmetric pattern and Table 5 shows the results for the matrices with nonsymmetric patterns. In order to obtain a better comparison of the effect of the dropping strategy, we set

TABLE 3
Performance of ILUC and c-ILUT on nonsymmetric pattern matrices, $\tau = 0.001$.

Matrix	Lfil, τ	Pre-alg	Sec_{pre}	Sec_{it}	Sec_{tot}	Its	Ratio
UTM.3060 $\gamma \approx 6.9$	$2.5\gamma \approx 17$	ILUC	0.09	1.05	1.14	102	2.238
	$\tau = 0.01$	c-ILUT	0.08	0.48	0.56	52	2.103
	$5.0\gamma \approx 34$	ILUC	0.19	0.42	0.61	31	4.337
	$\tau = 0.001$	c-ILUT	0.25	0.38	0.63	31	4.052
UTM.5940 $\gamma \approx 7.1$	$2.5\gamma \approx 17$	ILUC	0.19	-	-	-	2.194
	$\tau = 0.01$	c-ILUT	0.19	-	-	-	2.070
	$5.0\gamma \approx 35$	ILUC	0.44	3.05	3.49	107	4.453
	$\tau = 0.001$	c-ILUT	0.64	-	-	-	4.164
SHERMAN5 $\gamma \approx 3.1$	$2.5\gamma \approx 7$	ILUC	0.02	0.11	0.13	24	0.986
	$\tau = 0.01$	c-ILUT	0.02	0.08	0.10	21	1.156
	$5.0\gamma \approx 15$	ILUC	0.04	0.10	0.14	16	2.009
	$\tau = 0.001$	c-ILUT	0.05	0.07	0.12	14	1.961
LNS3973 $\gamma \approx 3.2$	$2.5\gamma \approx 8$	ILUC	0.03	-	-	-	1.484
	$\tau = 0.01$	c-ILUT	0.04	-	-	-	1.837
	$5.0\gamma \approx 16$	ILUC	0.05	-	-	-	2.529
	$\tau = 0.001$	c-ILUT	0.09	-	-	-	3.355
CAVA0000 $\gamma \approx 15.8$	$2.5\gamma \approx 39$	ILUC	1.55	-	-	-	2.403
	$\tau = 0.01$	c-ILUT	2.40	14.31	16.71	118	2.441
	$5.0\gamma \approx 79$	ILUC	4.40	7.93	11.33	45	4.850
	$\tau = 0.001$	c-ILUT	11.94	5.60	17.54	33	4.918

TABLE 4
Performance of two dropping strategies used by ILUC on symmetric pattern matrices, Lfil $= \infty$.

Matrix	Drop-strategy	Droptol	Sec_{pre}	Sec_{it}	Sec_{tot}	Its	Ratio
BARTHT1A	Inverse-based	0.27	0.77	20.48	21.25	231	1.366
	Standard	0.12	0.63	-	-	-	1.385
	Inverse-based	0.18	0.84	10.08	10.92	105	1.547
	Standard	0.095	0.77	-	-	-	1.587
RAEFSKY1	Inverse-based	0.01	0.48	0.56	1.04	19	0.990
	Standard	0.09	0.43	0.60	1.03	19	1.002
	Inverse-based	0.003	1.47	0.61	2.08	15	1.862
	Standard	0.03	1.43	0.70	2.13	17	1.899
RAEFSKY2	Inverse-based	0.016	0.66	0.75	1.41	21	1.379
	Standard	0.08	0.66	0.61	1.27	18	1.384
	Inverse-based	0.01	1.10	0.75	1.85	18	1.800
	Standard	0.05	1.17	0.67	1.84	16	1.811
RAEFSKY3	Inverse-based	0.026	2.81	3.12	5.93	18	1.166
	Standard	0.025	2.93	2.97	5.90	15	1.170
	Inverse-based	0.01	7.77	2.52	10.29	12	1.766
	Standard	0.01	8.38	2.57	10.95	11	1.764
VENKAT25	Inverse-based	0.08	2.65	34.24	36.89	100	1.589
	Standard	0.04	2.96	42.40	45.36	119	1.596
	Inverse-based	0.06	3.19	27.86	31.05	78	1.816
	Standard	0.08	3.67	38.04	41.71	105	1.824

Lfil to infinity for these tests. This means that the total number of nonzeros in the rows of U or columns of L is controlled only by the drop tolerance. In addition, an effort was made to obtain LU factors that use more or less the same amount of memory for the preconditioners being compared, as reflected by the fill ratios. This was accomplished by a trial and error process, where various drop tolerances were tested for each matrix. As can be seen, there are cases when the inverse-based method drops small elements more precisely than the standard technique, in the sense that

TABLE 5
Performance of two dropping strategies used by ILUC on nonsymmetric pattern matrices, Lfil = ∞.

Matrix	Drop-strategy	Droptol	Sec_{pre}	Sec_{it}	Sec_{tot}	Its	Ratio
UTM.3060	Inverse-based	0.34	0.06	2.22	2.28	223	1.652
	Standard	0.083	0.05	0.90	0.95	98	1.648
	Inverse-based	0.29	0.06	1.76	1.82	176	1.879
	Standard	0.06	0.06	0.56	0.62	56	1.879
UTM.5940	Inverse-based	0.34	0.17	-	-	-	2.156
	Standard	0.051	0.14	4.95	5.09	239	2.134
	Inverse-based	0.35	0.22	-	-	-	2.580
	Standard	0.029	0.17	3.04	3.21	141	2.479
SHERMAN5	Inverse-based	0.80	0.01	0.37	0.38	73	0.308
	Standard	0.78	0.01	0.90	0.91	166	0.308
	Inverse-based	0.67	0.01	0.33	0.34	60	0.339
	Standard	0.65	0.01	0.52	0.53	99	0.339
LNS3937	Inverse-based	1.7	0.08	3.11	3.19	287	3.669
	Standard	0.019	0.07	-	-	-	3.730
	Inverse-based	1.5	0.09	2.36	2.45	214	3.905
	Standard	0.016	0.07	-	-	-	3.915
CAVA0000	Inverse-based	0.15	0.60	25.94	26.54	296	0.932
	Standard	0.16	0.52	-	-	-	0.935
	Inverse-based	0.14	1.68	24.77	26.45	235	1.694
	Standard	0.071	1.63	-	-	-	1.645

elements are dropped when they are least likely to affect convergence of the iteration. This observation is illustrated in Figure 5, which compares the times required by GMRES to converge when the fill-in ratio is varied for the two dropping strategies. The coefficient matrix used for this test is called CONVDIFF2 and was obtained from a convection-diffusion problem. CONVDIFF2 has a symmetric pattern, is of size $n = 205,761$, and has $nnz = 1,436,480$ nonzero entries. It is a member of a sequence of four parameterized matrices with various degrees of nonsymmetry. When measured by the ratio $\|A - A^T\|_F / \|A + A^T\|_F$, the degree of nonymmetry of CONVDIFF2 is equal to 10^{-2}. No diagonal scaling was performed to process the matrix in this test.

Tests with other matrices show a similar behavior for the case when the pattern is symmetric. However, for matrices with nonsymmetric pattern, it can happen that the standard dropping strategy yields better results than the inverse-based dropping. In general, for reasons which are unclear, the ILUC does not appear to perform as well, relatively speaking, for matrices with nonsymmetric patterns.

5. Conclusion. The new version of ILU presented in this paper has several advantages over standard ILU techniques. The most obvious of these, which provided the primary motivation for this work, is that it leads to an efficient implementation that bypasses the need for searches. These costly searches constitute the main drawback of standard delayed-update implementations. In addition, this new version of ILU enables efficient implementations of some variations that were not practically possible with the other forms of ILUT. For example, the more rigorous dropping strategies based on estimating the norms of the inverse factors described in [1] can easily be implemented and lead to effective algorithms. In the same vein, this version of ILU also allows the implementation of potentially more effective pivoting strategies. This was not considered in this paper but will be the subject of a forthcoming study.

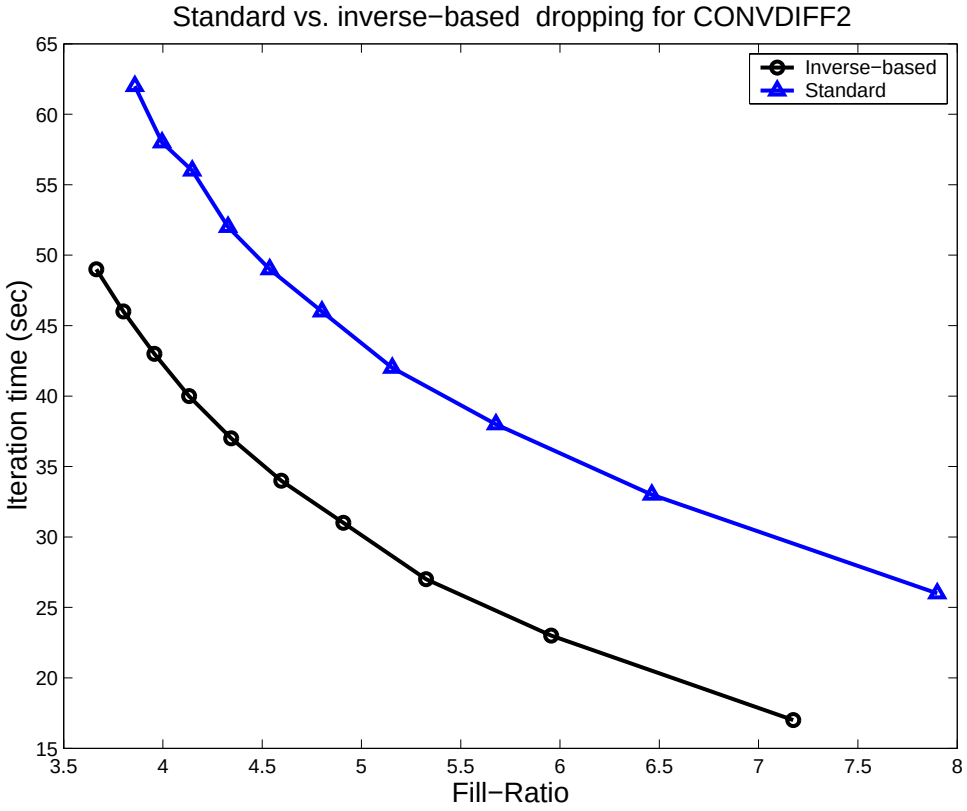

FIG. 5. *Iteration times vs fill-in ratio for inverse-based dropping and standard dropping.*

REFERENCES

[1] M. BOLLHÖFER, *A robust ILU with pivoting based on monitoring the growth of the inverse factors*, Linear Algebra Appl., 338 (2001), pp. 201–213.
[2] M. BOLLHÖFER AND Y. SAAD, *A factored approximate inverse preconditioner with pivoting*, SIAM J. Matrix Anal. Appl., 23 (2002), pp. 692–705.
[3] M. BOLLHÖFER AND Y. SAAD, *On the relations between ILUs and factored approximate inverses*, SIAM J. Matrix Anal. Appl., 24 (2003), pp. 219–237.
[4] M. BOLLHÖFER, *private communication*, 2002.
[5] E. CHOW AND Y. SAAD, *ILUS: An incomplete LU factorization for matrices in sparse skyline format*, Internat. J. Numer. Methods Fluids, 25 (1997), pp. 739–748.
[6] I. S. DUFF, A. M. ERISMAN, AND J. K. REID, *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford, UK, 1986.
[7] S. C. EISENSTAT, M. H. SCHULTZ, AND A. H. SHERMAN, *Algorithms and data structures for sparse symmetric Gaussian elimination*, SIAM J. Sci. Statist. Comput., 2 (1981), pp. 225–237.
[8] J. R. GILBERT AND T. PEIERLS, *Sparse partial pivoting in time proportional to arithmetic operations*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 862–874.
[9] G. H. GOLUB AND C. VAN LOAN, *Matrix Computations*, 3rd ed., The Johns Hopkins University Press, Baltimore, 1996.
[10] M. JONES AND P. PLASSMANN, *An improved incomplete Cholesky factorization*, ACM Trans. Math. Software, 21 (1995), pp. 5–17.
[11] J. M. ORTEGA, *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Press, New York, 1988.

[12] Y. Saad, *SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations*, Technical report RIACS-90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, 1990.
[13] Y. Saad, *ILUT: A dual threshold incomplete ILU factorization*, Numer. Linear Algebra Appl., 1 (1994), pp. 387–402.
[14] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS, New York, 1996.