Crout versions of ILU for general sparse matrices *

Na Li† Yousef Saad† Edmond Chow‡

April 7, 2002

Abstract

This paper presents an efficient implementation of incomplete LU (ILU) factorizations that are derived from the Crout version of Gaussian elimination (GE). At step k of the elimination, the k-th row of U and the k-th 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. It is shown that this version of ILU has many practical advantages. In particular, its data structure allows efficient implementation of more rigorous and effective dropping strategies. Numerical tests show that the method is far more efficient than standard threshold-based ILU factorizations computed row-wise or column-wise.

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.

1 Introduction

The rich variety of existing Gaussian elimination algorithms has often been exploited, for example, to extract the most efficient variant for a given computer architecture. It was noted in [11] that 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 form 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:

1. for $i = 2 : n$
2. for $k = 1 : i - 1$
3. for $j = k + 1 : n$

*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.

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

‡Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, L-560, Box 808, Livermore, CA 94551, email: 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 No. W-7405-Eng-48.

4. $\qquad a_{ij} = a_{ij} - a_{ik} * a_{kj}$

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, which is the JKI version of GE.

Bordering methods can be viewed as modifications of the delayed-update methods. The loop starting in Line 3 of the above IKJ version is shortened into "for $j = k+1 : i-1$" which computes the k-th row of L. A similar loop is then added to compute the k-th *column* of U. In theory, 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 factorizations can be derived from any of these variants, see, e.g., [14, 2, 1] although the IKJ version has often been preferred because it greatly simplifies the data structure required by the implementation. In fact, a key factor in selecting one of the options is the convenience of the data structure that is used. For example, the KIJ algorithm requires rank-one updates and this causes up to $n-k$ rows and columns to be altered at step k. Since the matrices are stored in sparse mode, this is in not efficient for handling the fill-ins introduced during the factorization [6]. However, other methods were developed as well. For example, a technique based on bordering was advocated in [5].

This papers considers incomplete LU factorizations based on yet another version of Gaussian elimination known as the Crout variant, which which can seen as a combination of the IKJ algorithm shown above to compute the U part and a transposed version to compute the L part. The k-th 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 threshold-based incomplete factorizations, is increasing order by column number (since the nonzero pattern of the factorization is not known beforehand). The entries in this order must be found via searches, which are further 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 a high number of fill-ins are required, which is the case for more difficult problems, the cost of searching for the leftmost pivot may make the factorization ineffective. An alternative is to maintain the current row in a binary tree and to utilize binary searches. This strategy was mentioned in [14] and was recently implemented by Bollhoefer in ILUT and ILUTP [4]. This code will be used for comparisons later in this paper.

In this paper we will make the case that among the various Gaussian elimination algorithms, the Crout formulation provides perhaps the most practically useful option when developing incomplete LU factorizations. It was observed in [10] that the Crout-based incomplete factorization is effective in the symmetric positive definite case, as it bypasses the need for the costly searches mentioned above. As will be seen, this can be easily generalized to the nonsymmetric case. Moreover, this version has one other compelling advantage, namely that it leads to an efficient implementation of inverse-based dropping strategies [3]. These strategies have been shown to be effective in [3] and this will be verified in the numerical experiments section of this paper.

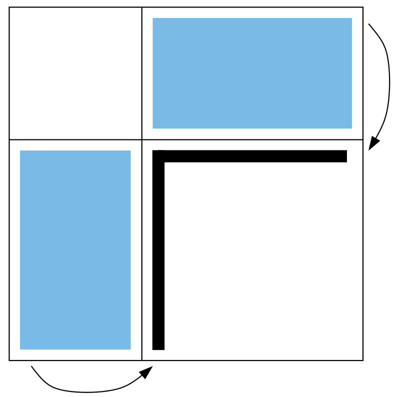

Figure 1: The computational pattern for the Crout factorization. The dark area shows the parts of the factors being computed at the k-th step. The shaded areas show the parts of the factors being accessed at the k-th step.

2.1 The Crout formulation

The Crout formulation can be viewed as yet another "delayed-update" form of GE. At step k 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 k-th 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 A stored such that its lower triangular part is stored by columns and its upper triangular part is stored by rows. The computational pattern for the factorization is shown in Figure 1.

ALGORITHM **2.1** *Crout LU Factorization*

For $k = 1 : n$ *Do :*
    For $i = 1 : k-1$ *and if* $a_{ki} \neq 0$ *Do :*
        $a_{k,k:n} = a_{k,k:n} - a_{ki} * a_{i,k:n}$
    EndDo
    For $i = 1 : k-1$ *and if* $a_{ik} \neq 0$ *Do :*
        $a_{k+1:n.k} = a_{k+1:n,k} - a_{ik} * a_{k+1:n,i}$
    EndDo
    $a_{ik} = a_{ik}/a_{kk}$ *for* $i = k+1, ..., n$
EndDo

The k-th step of the algorithm generates the k-th row of U and the k-th column of L. This step is schematically represented in Figure 2. Notice now that the updates to the k-th row of U (resp. the k-th column of L) can be made in any order. There is also a certain symmetry in the data structure representing L and U since the U matrix is accessed by rows and the L matrix by columns. By adapting Algorithm 2.1 for

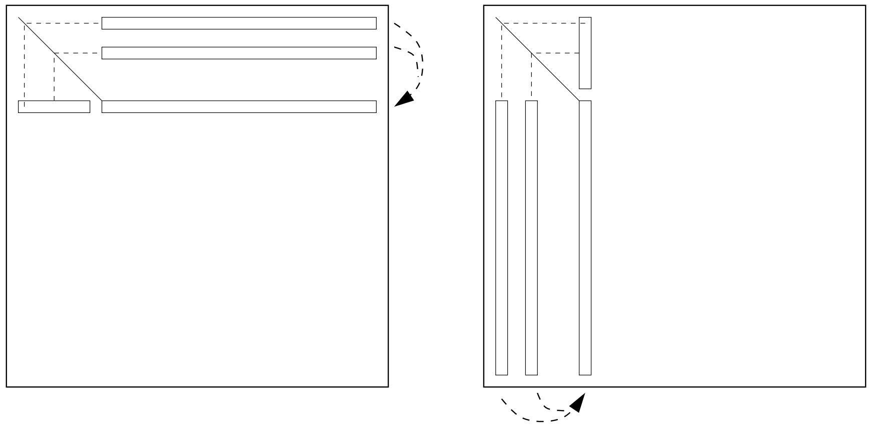

Figure 2: Construction of the k-th row of U (left side) and the k-column of L (right side).

sparse computations and by adding a dropping strategy, the following Crout version of ILU (termed 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* $\{i \mid 1 \leq i \leq k-1$ *and* $u_{ik} \neq 0\}$ *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 done 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 i-th row of U is required, and similarly, only the section $(k+1 : n)$ of the i-th 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 k-th 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 length n arrays: $Ufirst$, *Ulist*, $Lfirst$, and *Llist*, which 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 $i = 1, ..., k - 1$;

b. *Ulist*(k) points to a linked list of rows that will update row k;

c. $Lfirst(i)$ points to the first entry with row index greater than or equal to k in column i of L, where $i = 1, ..., k - 1$;

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

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

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

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 and will be considered in our future work.

3 Dropping Strategies

Any dropping rule can be applied in Lines 10 and 11 of Algorithm 2.2. In this section, we consider a number of different options which were implemented and tested. The most straightforward of these options is a threshold-based technique that is similar to the one used in ILUT. A very important consideration with ILUC is that its data structure allows options that were not practically feasible with standard IKJ implementations of ILU or their column-based equivalent. In particular, note that at step k the first k columns (resp. rows) of L (resp. U) are available. In particular, this enables us to obtain dropping techniques that utilize estimates of the inverse factors which were shown to be quite successful in [3]. These two techniques are considered in turn. Several other strategies were also tested but lead to mixed results.

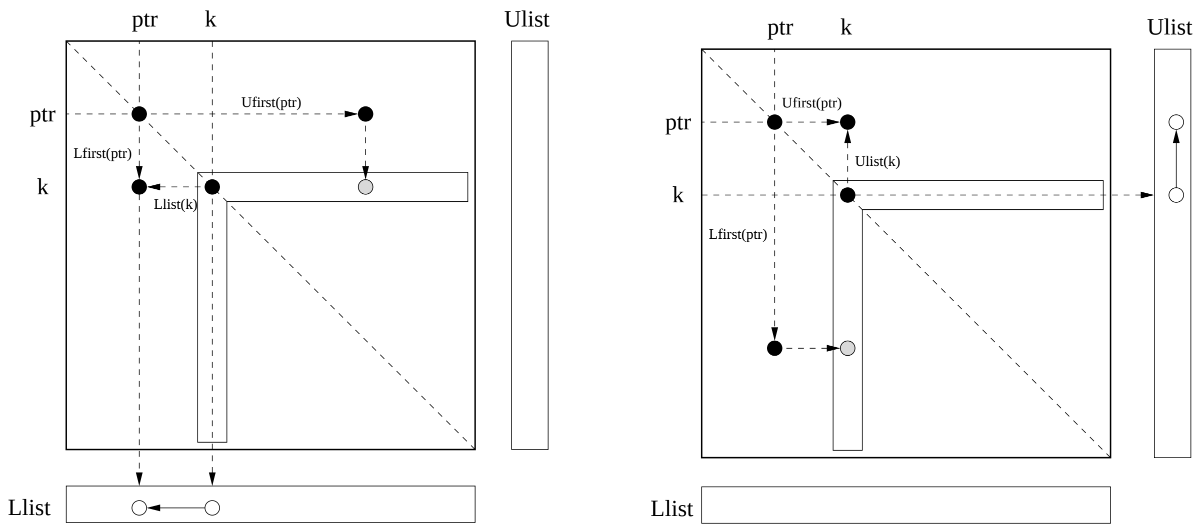

Figure 3: Procedure for updating row k of U and column k of L.

3.1 Standard dual criterion dropping strategy

The dual 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 k-th column of L or the k-th row of U respectively) is dropped.

2. Then, only the "Lfil" largest elements in magnitude in the k-th column of L are kept. Similarly the "Lfil" largest elements in the k-th row of U in addition to the diagonal element are kept. This controls the total storage that can be used by the preconditioner.

The above strategy is referred to as the "standard strategy" in the experiments. It is often observed that it is more effective to use a drop tolerance only, rather than to force a limited fill-in. This means that better results are often achieved by taking a large value of "Lfil" and varying the parameter τ to achieve a given amount of fill-in.

3.2 Dropping based on condition number estimators

In order to reduce the impact of dropping an element on the subsequent steps of Gaussian elimination, it is useful to devise dropping strategies which estimate the norms of the rows of L^{-1} and the columns of U^{-1}. Such techniques were proved to be quite effective in the ILU context by Bollhoefer [3]. 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 k-th 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 the sequel we only discuss the strategy for the L part. The justification for this criterion given in [3] 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 $l_{jk} e_j e_k^T$. Noticing that $L_k e_j = e_j$ we have

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

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

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

Therefore, the inverse of L_k will be perturbed by l_{jk} times the k-th row of L_k^{-1}. This perturbation will affect the j-th 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 $\|l_{jk} e_j e_k^T L_k^{-1}\|_\infty = |l_{jk}| \; \|e_k^T L_k^{-1}\|_\infty$.

However, the matrix L^{-1} is not available and it is not feasible to compute it. Instead, in [3] standard techniques used for estimating condition numbers [9] are adapted for estimating the norm of the k-th row of L^{-1} (resp. k-th column of U^{-1}). In this paper we only use 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 $\|e_k^T L^{-1}\|_\infty$ can be reduced to that of dynamically constructing a right-hand side b to the linear system $Lx = b$ so that the k-th component of the solution is the largest possible. Thus, if b is the current right-hand side at step k, we write,

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

where $\|e_k^{\mathrm{T}} L^{-1}\|$ was estimated as the k-th 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 k-th 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 k-th component of the solution x is

$$\xi_k = b_k - e_k^T L_{k-1} x_{k-1}$$

This makes the choice clear: if ξ_k is to be large in modulus, then its sign should be of the opposite sign as as $e_k^T L_{k-1} x_{k-1}$. 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 $\qquad \nu_j = \nu_j + \xi_k l_{jk}$
7 $\quad$ *EndDo*
8. *EndDo*

The paper [3] 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

$$|\xi_k| + \|p_k\|_1 \tag{2}$$

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 (2) for both of the choices in Line 3 and select the choice that gives the largest weight.

4 Diagonal compensation strategies

It is sometimes helpful to modify the diagonal entries in the ILU factorization to compensate for the elements being dropped during factorization. In the simplest row-oriented ILU techniques, the sum of all elements being dropped in computing a given row of the L, U pair, is added to the diagonal entry. This makes the product LU and A have the same row-sum, and as a result the preconditioned matrix will have one eigenvalue equal to one with associated eigenvector the vector of all ones.

In the context of ILUC, we can also enforce a similar condition. In fact it is possible, as well as natural, to enforce both a row-sum and a column-sum condition. Consider the equation which defines the k-th column of L for the equivalent $A = LDU$ factorization,

$$\tilde{l}_{k+1:n,k} = a_{k+1:n,k} - \sum_{j=1}^{k-1} u_{j,k} d_{j,j} l_{k+1:n,j}. \tag{3}$$

After this column is calculated it undergoes dropping and then scaling,

$$\hat{l}_{k+1:n,k} := \tilde{l}_{k+1:n,k} + s_{k+1:n,k}, \qquad l_{k+1:n,k} := \hat{l}_{k+1:n,k}/d_k.$$

As a result we have

$$a_{k+1:n,k} = \sum_{j=1}^{k} u_{j,k} d_{j,j} l_{k+1:n,j} + s_{k+1:n,k}.$$

Therefore, it may be possible to enforce a column-sum condition on the strict lower part of A, and in a similar fashion, a row-sum condition on the strict upper part of A. We can do better with a little additional work. The above relation can be extended to the entire column

$$a_{:,k} = \sum_{j=1}^{k} u_{j,k} d_{j,j} l_{:,j} + s_{:,k}.$$

While $s_{k+1:n}$ is available at step k, the elements $s_{1:k}$ represent terms dropped from the U-part in earlier steps. It is possible to keep a running sum of these elements for each column as the algorithm proceeds. If e is a vector of all ones, then

$$e^T a_{:,k} = e^T \sum_{j=1}^{k} u_{j,k} d_{j,j} l_{:,j} + e^T s_{1:k,k} + e^T s_{k+1:n,k}.$$

The term $e^T s_{k+1:n,k}$ is the sum of elements dropped while computing $l_{:,k}$ and is therefore easily available. The second term, $e^T s_{1:k,k}$, is the sum of elements dropped in previous steps in the U part of the matrix. Note that $e^T s_{1:k,k} = e^T s_{1:k-1,k}$, since no elements are dropped from the diagonal. This second sum is available provided we maintain and update a row-vector which runs all the column-sums of the terms dropped for each column. Thus, once the row $U_{k,:}$ is determined, this row, call it r_{sum} will be updated by adding to it all elements dropped while building $U_{k,:}$. Similarly, a column, say t_{sum}, is maintained which adds up all the terms dropped when computing the successive columns of L.

5 Experimental results

The performance of ILUC was compared to standard ILUT [13] in both row-wise (r-ILUT) and column-wise (c-ILUT) forms. The codes were written in C, and the experiments were conducted on a 866 MHz Pentium III computer with 1 GB of main memory. The codes were compiled with -O3 for optimization.

The test matrices can be described using a measure of structural symmetry 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). All 10 test matrices are nonsymmetric and of those, five have a nonsymmetric pattern. Some generic information about the test matrices is shown in Table 1. The BARTHT1A matrix was supplied by T. Barth of NASA Ames. The SHERMAN2 matrix is from the Boeing-Harwell collection and is available from the Matrix Market.[1] The matrices CAVA0000 and CAVA0100[2] resulted from the simulation of a driven cavity problem. The domain of interest is 2-dimensional and the discretization uses quadrilateral elements with bi-quadratic functions for velocities and linear (discontinuous) functions for pressures. Using 40 elements in each direction yields a matrix of size $n = 17,922$ and $nnz = 567,467$ nonzero elements. These linear systems are indefinite and can be difficult to solve. The Reynolds number was used as a continuation parameter; the test matrices have Reynolds numbers 0 and 100. The other matrices are available from the University of Florida sparse matrix collection.[3] In the table, n is the dimension of the matrix and nnz represents the total number of nonzero elements.

Artificial right-hand sides were generated, and GMRES(60) was used to solve the systems using a random initial guess. The iterations were stopped when the residual norm was reduced by 8 orders of magnitude or when the maximum iteration count of 300 was reached.

Table 2 compares the timings to build the preconditioners ("Pr-sec.") and the iteration timings ("Its sec.") for ILUC, r-ILUT and c-ILUT on the matrices from the set that have symmetric patterns. "Lfil" is the dropping parameter described in Section 3.1. We selected "Lfil" by basing it on the ratio $\gamma = \frac{nnz}{2n}$, which is an average number of nonzeros in each row or column of the upper or lower triangular part of the matrix. "Its" denotes the number of iterations to convergence. The symbol "-" in the table indicates

[1] `http://math.nist.gov/MatrixMarket/`
[2] Matrices available from the authors.
[3] `http://www.cise.ufl.edu/ davis/sparse/`

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
UTM.3060	0.5591	3060	42211
UTM.5940	0.5624	5940	83842
SHERMAN2	0.6862	1080	23094
CAVA0000	0.9773	17922	567467
CAVA0100	0.9773	17922	567467

Table 1: Information on the 10 matrices used for tests

that convergence was not obtained in 300 iterations. "Pr-Mem." denotes the number of memory locations required by the preconditioners and "Ratio" denotes the fill-factor, i.e., the value of $nnz(L+U)/nnz(A)$.

There are two main observations that can be made in Table 2. First, the setup timings for ILUC are significantly smaller than those for r-ILUT and c-ILUT. The difference can be seen to be larger when a larger amount of fill-in is allowed. Second, ILUC may be more robust than r-ILUT and c-ILUT or may require fewer iterations to converge (see BARTHT1A and VENKAT25). For most other problems however, the iterations counts for ILUC and the other variants are similar. In general, the overall solution time is reduced by using ILUC.

Figure 4 shows the timings required for computing three preconditioners as a function of "Lfil" for the matrix RAEFSKY3. The drop tolerance was $\tau = 0.001$. The figure also shows the timings 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, which is referred to as b-ILUT [4], was coded in FORTRAN. The figure shows that ILUC is faster than all the other variants, and that the ILUC 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 5 matrices in the test set that have nonsymmetric patterns. The superiority of ILUC is not as compelling as in Table 2 which involved only matrices with symmetric patterns. The time to compute the preconditioner is still generally smaller than with the other versions. Sometimes, ILUC did help GMRES achieve convergence as shown in the case of the matrix UTM.5940. In other cases, it caused GMRES to fail to converge or to converge slowly, while r-ILUT and/or c-ILUT yielded good convergence. This is illustrated by the results with SHERMAN2 and CAVA0100.

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"). For these tests we added four symmetric-pattern test matrices to study the effect of nonsymmetry. These matrices arise from two-dimensional finite element convection-diffusion problems. They were obtained using linear triangular elements and have 205761 equations and 1436480 nonzeros. The four matrices correspond to different sizes of the convection term, leading to increasing degrees of nonsymmetry; see Table 4.

Table 5 shows the results for those matrices in the set which have a symmetric patterns and Table 6 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 Lfil to infinity for these tests. This means that the total number

Matrix	Lfil	Pr-alg.	Pr-sec.	Its sec.	Its	Pr-Mem.	Ratio
BARTHT1A $\gamma \approx 17.1$	$2.0\gamma \approx 34$	ILUC	0.920	-	-	815037	1.694
		r-ILUT	1.990	-	-	923060	1.919
		c-ILUT	2.210	-	-	937685	1.949
	$2.5\gamma \approx 42$	ILUC	1.160	7.550	78	986395	2.050
		r-ILUT	2.340	-	-	1113073	2.313
		c-ILUT	2.870	-	-	1147112	2.384
	$3.0\gamma \approx 51$	ILUC	1.450	5.900	55	1166370	2.424
		r-ILUT	2.870	-	-	1319407	2.742
		c-ILUT	3.690	-	-	1378620	2.865
RAEFSKY1 $\gamma \approx 45.4$	$1.0\gamma \approx 45$	ILUC	0.460	0.700	22	289594	0.984
		r-ILUT	1.520	0.570	18	288136	0.979
		c-ILUT	1.620	0.530	18	288168	0.979
	$1.5\gamma \approx 68$	ILUC	0.790	0.760	20	427595	1.453
		r-ILUT	2.810	0.610	16	430724	1.464
		c-ILUT	3.010	0.580	16	430937	1.464
	$2.0\gamma \approx 90$	ILUC	1.250	0.790	18	557364	1.894
		r-ILUT	4.980	0.680	15	562961	1.913
		c-ILUT	5.150	0.620	15	563896	1.916
RAEFSKY2 $\gamma \approx 45.4$	$1.0\gamma \approx 45$	ILUC	0.460	0.860	27	291078	0.989
		r-ILUT	1.950	0.730	23	288367	0.980
		c-ILUT	1.910	0.750	25	288345	0.980
	$1.5\gamma \approx 68$	ILUC	0.790	0.800	21	431255	1.465
		r-ILUT	3.760	0.680	18	431369	1.466
		c-ILUT	3.710	0.730	20	431394	1.466
	$2.0\gamma \approx 90$	ILUC	1.240	0.800	18	560577	1.905
		r-ILUT	5.910	0.730	16	565818	1.923
		c-ILUT	5.910	0.670	16	566890	1.926
RAEFSKY3 $\gamma \approx 35.1$	$1.0\gamma \approx 35$	ILUCT	1.520	2.080	13	1197935	0.805
		r-ILUT	5.320	2.950	17	1467232	0.986
		c-ILUT	4.920	1.790	11	1467145	0.985
	$1.5\gamma \approx 52$	ILUC	2.290	2.180	12	1647174	1.106
		r-ILUT	8.460	2.720	14	1965430	1.320
		c-ILUT	7.470	1.840	10	1966184	1.321
	$2.0\gamma \approx 70$	ILUC	3.350	2.000	10	2076087	1.395
		r-ILUT	12.450	2.650	12	2406591	1.616
		c-ILUT	11.120	2.010	10	2406519	1.616
VENKAT25 $\gamma \approx 13.8$	$1.0\gamma \approx 13$	ILUC	1.580	59.530	160	1668586	0.971
		r-ILUT	3.720	91.780	241	1680052	0.978
		c-ILUT	9.760	-	-	1681197	0.979
	$1.5\gamma \approx 20$	ILUC	2.680	29.610	73	2537005	1.477
		r-ILUT	7.180	41.500	102	2545032	1.482
		c-ILUT	17.160	50.220	126	2547584	1.483
	$2.0\gamma \approx 27$	ILUC	4.210	20.480	48	3405086	1.982
		r-ILUT	10.740	30.230	69	3399380	1.979
		c-ILUT	23.260	27.240	63	3407450	1.984

Table 2: Performance of ILUC, r-ILUT and c-ILUT on symmetric pattern matrices, $\tau = 0.001$

Matrix	Lfil	Pr-alg.	Pr-sec.	Its sec.	Its	Pr-Mem.	Ratio
UTM.3060 $\gamma \approx 6.9$	$2.5\gamma \approx 17$	ILUC	0.080	1.860	156	91568	2.169
		r-ILUT	0.150	-	-	94910	2.248
		c-ILUT	0.130	0.840	75	91739	2.173
	$3.0\gamma \approx 20$	ILUC	0.110	0.780	58	106731	2.529
		r-ILUT	0.180	-	-	110196	2.611
		c-ILUT	0.150	0.600	51	105762	2.506
	$3.5\gamma \approx 24$	ILUC	0.130	0.800	56	126396	2.994
		r-ILUT	0.210	-	-	130066	3.081
		c-ILUT	0.180	0.580	47	124268	2.944
UTM.5940 $\gamma \approx 7.1$	$4.0\gamma \approx 28$	ILUC	0.320	5.470	180	289406	3.452
		r-ILUT	0.600	-	-	294066	3.507
		c-ILUT	0.550	-	-	283592	3.382
	$4.5\gamma \approx 31$	ILUC	0.350	3.780	119	318232	3.796
		r-ILUT	0.650	-	-	322226	3.843
		c-ILUT	0.620	-	-	311632	3.717
	$5.0\gamma \approx 35$	ILUC	0.410	3.950	118	356279	4.249
		r-ILUT	0.730	-	-	360560	4.300
		c-ILUT	0.690	-	-	346272	4.130
SHERMAN2 $\gamma \approx 10.7$	$1.0\gamma \approx 10$	ILUC	0.010	-	-	11727	0.508
		r-ILUT	0.030	-	-	16855	0.730
		c-ILUT	0.010	0.190	51	20345	0.881
	$1.5\gamma \approx 16$	ILUC	0.010	1.130	294	13539	0.586
		r-ILUT	0.020	-	-	23587	1.021
		c-ILUT	0.010	0.040	13	26674	1.155
	$2.0\gamma \approx 21$	ILUC	0.020	1.120	293	14045	0.608
		r-ILUT	0.020	0.060	18	26466	1.146
		c-ILUT	0.010	0.020	9	28100	1.217
CAVA0000 $\gamma \approx 15.8$	$2.0\gamma \approx 31$	ILUC	1.330	5.820	48	1103714	1.945
		r-ILUT	3.050	5.150	45	1084765	1.912
		c-ILUT	3.290	4.840	42	1116973	1.968
	$2.5\gamma \approx 39$	ILUC	1.780	42.710	300	1382548	2.436
		r-ILUT	3.830	5.620	43	1357450	2.392
		c-ILUT	4.390	7.710	58	1398952	2.465
	$3.0\gamma \approx 47$	ILUC	2.180	6.930	47	1660151	2.926
		r-ILUT	4.570	4.880	36	1630057	2.873
		c-ILUT	4.980	4.720	36	1675004	2.952
CAVA0100 $\gamma \approx 15.8$	$2.0\gamma \approx 31$	ILUC	1.310	6.540	53	1085455	1.913
		r-ILUT	3.090	5.130	43	1094607	1.929
		c-ILUT	3.250	5.170	45	1083709	1.910
	$2.5\gamma \approx 39$	ILUC	1.710	43.460	300	1359356	2.395
		r-ILUT	3.850	5.340	42	1367359	2.410
		c-ILUT	4.130	5.360	43	1356124	2.390
	$3.0\gamma \approx 47$	ILUC	2.160	28.830	179	1631668	2.875
		r-ILUT	4.620	4.290	32	1638681	2.888
		c-ILUT	5.020	4.690	35	1627911	2.869

Table 3: Performance of ILUC, r-ILUT and c-ILUT on nonsymmetric pattern matrices, $\tau = 0.001$

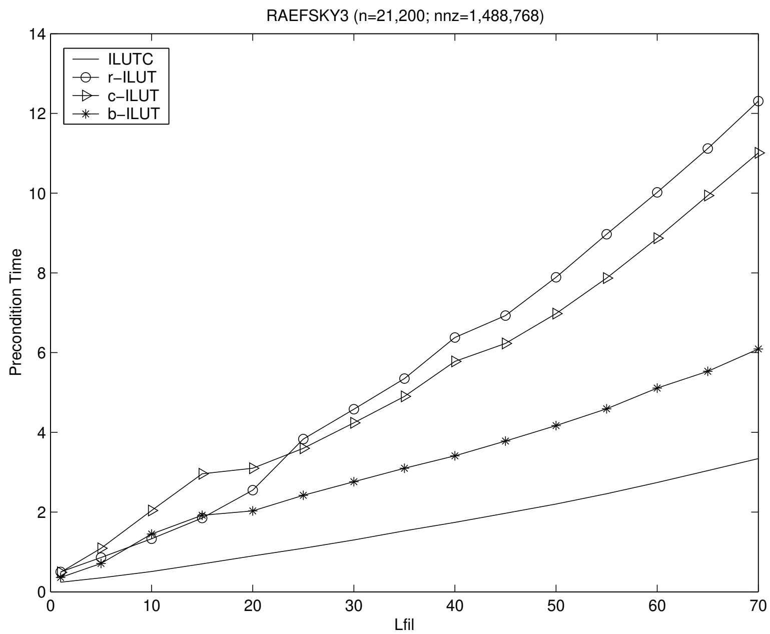

Figure 4: Precondition time vs. Lfil for ILUC, r-ILUT, c-ILUT and b-ILUT ($\tau = 0.001$)

Matrix	η
CONVDIFF0	3×10^{-4}
CONVDIFF1	3×10^{-3}
CONVDIFF2	3×10^{-2}
CONVDIFF3	3×10^{-1}

Table 4: Four symmetric-pattern matrices from a convection-diffusion problem. The matrices have 205761 equations and 1436480 nonzeros. The value η measures the degree of nonsymmetry, $\|A - A^T\|_F / \|A + A^T\|_F$.

Matrix	Drop-strategy	Droptol	Pr-sec.	Its sec.	Its	Ratio
BARTHT1A	Inverse-based	0.1	2.880	6.460	49	3.501
	Standard	0.01	2.820	8.480	60	3.382
	Inverse-based	0.01	28.560	5.100	19	9.660
	Standard	0.001	35.140	6.490	22	10.890
RAEFSKY1	Inverse-based	0.01	0.500	0.560	18	0.924
	Standard	0.1	0.470	0.670	20	1.096
	Inverse-based	0.001	7.710	0.700	10	3.670
	Standard	0.01	6.190	0.820	12	3.522
RAEFSKY2	Inverse-based	0.01	1.060	0.730	17	1.736
	Standard	0.1	0.630	0.700	19	1.325
	Inverse-based	0.001	13.270	0.810	9	5.074
	Standard	0.01	8.300	0.900	12	4.030
RAEFSKY3	Inverse-based	0.1	7.070	13.510	57	1.391
	Standard	0.1	-	-	-	1.083
	Inverse-based	0.01	18.180	2.360	9	2.280
	Standard	0.01	14.800	2.190	8	2.402
VENKAT25	Inverse-based	0.1	7.330	63.920	127	2.539
	Standard	0.1	2.900	37.000	91	1.522
	Inverse-based	0.01	68.450	24.710	25	9.300
	Standard	0.01	30.800	15.180	21	6.175
CONVDIFF0	Inverse-based	0.01	5.800	100.550	95	3.644
	Standard	0.01	5.840	165.450	132	3.909
	Inverse-based	0.001	15.820	38.340	34	7.137
	Standard	0.001	21.320	83.010	46	9.391
CONVDIFF1	Inverse-based	0.01	5.930	123.110	97	3.646
	Standard	0.01	5.950	153.840	116	3.910
	Inverse-based	0.001	15.840	36.860	33	7.139
	Standard	0.001	19.910	71.250	43	9.389
CONVDIFF2	Inverse-based	0.01	6.170	52.120	49	3.665
	Standard	0.01	6.100	82.310	60	3.924
	Inverse-based	0.001	16.110	16.540	17	7.174
	Standard	0.001	20.900	32.700	22	9.323
CONVDIFF3	Inverse-based	0.1	2.400	98.260	100	1.523
	Standard	0.1	2.450	102.970	101	1.744
	Inverse-based	0.01	6.340	12.520	17	3.848
	Standard	0.01	6.670	19.840	21	4.221

Table 5: Performance of two dropping strategies used by ILUC on symmetric pattern matrices, Lfil $= \infty$

Matrix	Drop-strategy	Droptol	Pr-sec.	Its sec.	Its	Ratio
UTM.3060	Inverse-based	0.6	-	-	-	1.224
	Standard	0.1	0.050	2.130	150	1.455
	Inverse-based	0.06	0.210	0.680	37	4.251
	Standard	0.01	0.180	0.590	31	4.178
UTM.5940	Inverse-based	0.1	0.850	4.720	115	6.251
	Standard	0.01	0.500	2.650	54	5.217
	Inverse-based	0.01	5.530	2.290	31	15.279
	Standard	0.001	4.430	2.080	25	14.771
SHERMAN2	Inverse-based	0.1	0.030	0.030	7	1.444
	Standard	5e-5	0.020	0.060	14	1.155
	Inverse-based	0.01	0.060	0.010	2	2.196
	Standard	5e-6	0.050	0.020	5	1.927
CAVA0000	Inverse-based	0.0008	22.660	6.640	27	6.824
	Standard	0.01	38.580	40.390	128	6.491
	Inverse-based	0.005	45.580	7.620	24	9.539
	Standard	0.001	49.480	28.060	74	10.013
CAVA0100	Inverse-based	0.1	19.910	24.920	88	5.500
	Standard	0.6	31.330	51.800	204	5.905
	Inverse-based	0.01	6.680	5.010	23	4.310
	Standard	0.06	65.230	14.440	46	8.832

Table 6: Performance of two dropping strategies used by ILUC on nonsymmetric pattern matrices, Lfil $= \infty$

of nonzeros in the rows of U or columns of L is limited 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, the inverse-based method seems to drop small elements more precisely than the standard technique, in the sense that elements are dropped when they are least likely to affect convergence of the iteration. The tests also show that, in most cases, fewer GMRES iterations are needed to converge with the inverse-based dropping version.

This observation is further illustrated by the plots shown in Figure 5 which compare iteration times required by GMRES to converge when the fill-in ratio is varied for the two strategies. This is done for the four matrices BARTHT1A, CONVDIFF2 (symmetric patterns) and UTM.5940 and CAVA0100 (nonsymmetric patterns). Of the four cases, only UTM.5940 showed poorer overall performance for the inverse-based dropping. For reasons which are unclear, ILUC does not perform as well, relatively speaking, for matrices with nonsymmetric patterns.

6 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. Perhaps more significant is the advantage that this new version of ILU enables efficient implementations of some variations that were not practically possible with standard ILUT. For example, the more rigorous dropping strategies based on estimating the norms of the inverse

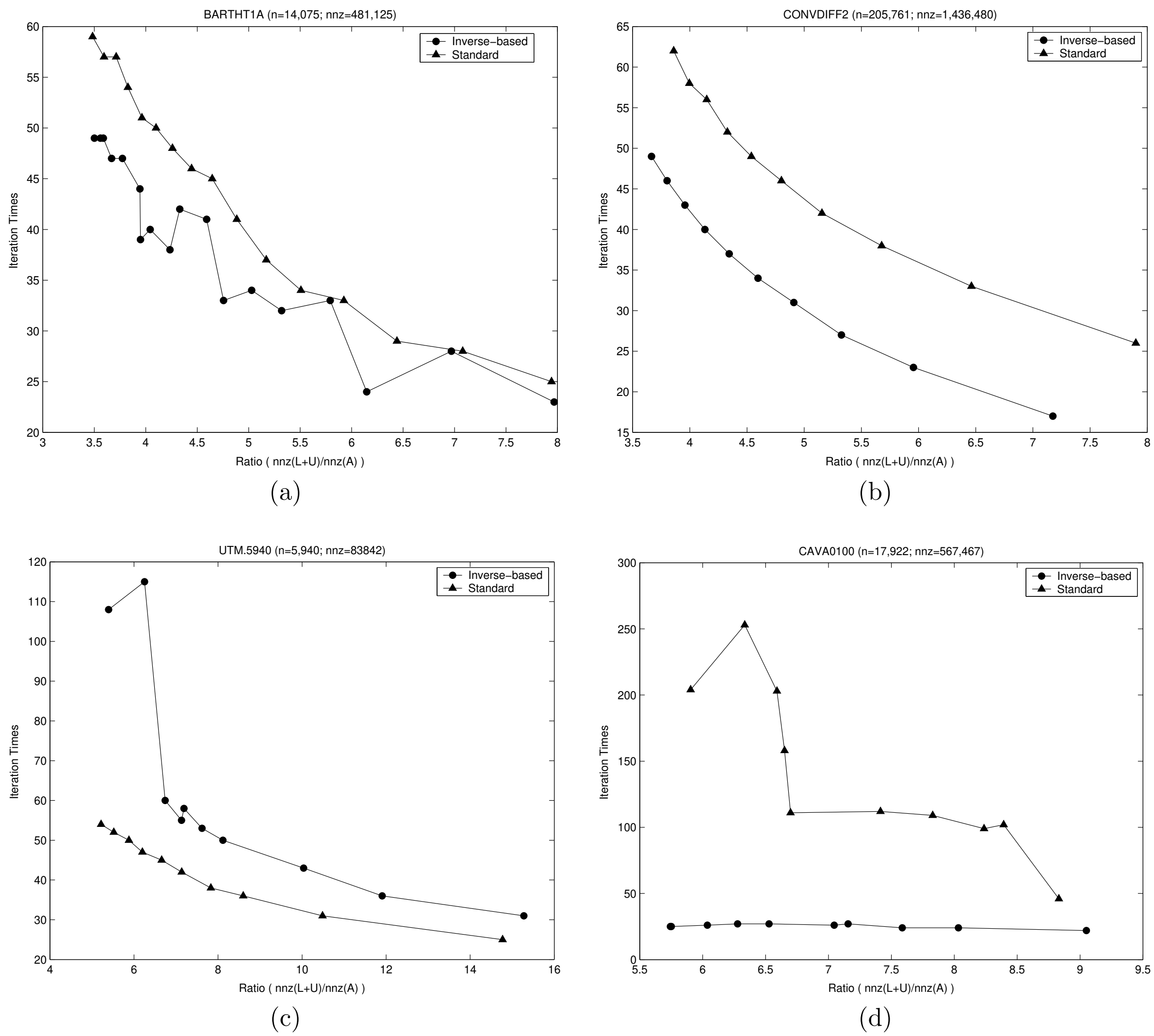

Figure 5: *Iteration times vs fill-in ratio for inverse-based dropping and standard dropping*

factors described in [3] 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.

References

[1] M. Bollhöfer and Y. Saad. A factored approximate inverse preconditioner with pivoting. *SIAM Journal on Matrix Analysis and Applications*, 2001. to-appear.

[2] M. Bollhöfer and Y. Saad. On the relations between ilus and factored approximate inverses. Technical Report umsi-2001-67, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 2001. To appear, SIMAX.

[3] Matthias Bollhöfer. A robust ILU with pivoting based on monitoring the growth of the inverse factors. *Linear Algebra and its Applications*, 338(1–3):201–218, 2001.

[4] Matthias Bollhöfer. Binary search tree implementation of ILUT and ILUTP. Personal Communication, 2002.

[5] E. Chow and Y. Saad. ILUS: an incomplete LU factorization for matrices in sparse skyline format. *International Journal for Numerical Methods in Fluids*, 25:739–748, 1997.

[6] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, 1986.

[7] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman. Algorithms and data structures for sparse symmetric Gaussian elimination. *SIAM Journal on Scientific Computing*, 2:225–237, 1981.

[8] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to artithmetic operations. *SIAM Journal on Scientific Computing*, 9:862–874, 1988.

[9] G. H. Golub and 3rd edn C. Van Loan. *Matrix Computations*. The John Hopkins University Press, Baltimore, 1996.

[10] M. Jones and P. Plassman. An improved incomplete Choleski factorization. *ACM Transactions on Mathematical Software*, 21:5–17, 1995.

[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, Moffet Field, CA, 1990.

[13] Y. Saad. ILUT: a dual threshold incomplete ILU factorization. *Numerical Linear Algebra with Applications*, 1:387–402, 1994.

[14] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS publishing, New York, 1996.