Numerical Linear Algebra with Applications, Vol. 1(4), 387–402 (1994)

ILUT: a Dual Threshold Incomplete LU Factorization

Yousef Saad

University of Minnesota, Department of Computer Science, 200 Union Street S.E., Minneapolis, MN 55455, USA

In this paper we describe an Incomplete LU factorization technique based on a strategy which combines two heuristics. This ILUT factorization extends the usual ILU(0) factorization without using the concept of level of fill-in. There are two traditional ways of developing incomplete factorization preconditioners. The first uses a symbolic factorization approach in which a level of fill is attributed to each fill-in element using only the graph of the matrix. Then each fill-in that is introduced is dropped whenever its level of fill exceeds a certain threshold. The second class of methods consists of techniques derived from modifications of a given direct solver by including a dropoff rule, based on the numerical size of the fill-ins introduced. traditionally referred to as threshold preconditioners. The first type of approach may not be reliable for indefinite problems, since it does not consider numerical values. The second is often far more expensive than the standard ILU(0). The strategy we propose is a compromise between these two extremes.

KEY WORDS Preconditioning Incomplete LU Threshold strategies

1. Introduction

A widely recognized weakness of iterative solvers is their lack of robustness. This difficulty is currently causing serious impediments to the acceptance of iterative solvers in industrial applications in spite of their intrinsic appeal for very large linear systems. Incomplete LU factorizations, combined with a good Krylov subspace projection process, are often regarded as the best 'general purpose' iterative solvers. In general, the reliability of such methods for solving problems from various origins depends much more on the quality of the preconditioner than on the iterative accelerator. An interesting observation is that recent research has been devoted in great part to developing 'robust' iterative accelerators while 'robust' preconditioners have by and large been neglected mainly because of the inherent lack of theory to support these techniques. Yet these techniques are vital to the success of iterative methods in real applications.

The simplest way to define a preconditioner is in the form of an incomplete factorization $A = LU + E$ where L and U have the same nonzero structure as the lower and upper

CCC 1070–5325/94/040387–16

Received 29 September 1992
Revised May, 1994

parts of A respectively. This so-called ILU(0) incomplete factorization is rather easy and inexpensive to compute. On the other hand, it leads to an approximation that is often too crude and as a result the Krylov subspace accelerator may require too many iterations to converge or may even diverge. To remedy this, alternative incomplete factorizations which allow more fill-in in L and U have been defined in several different ways. In general, the more accurate ILU factorizations require fewer iterations to converge. On the other hand the preprocessing cost to compute the incomplete factorization increases. However, if only because of the improved robustness, these tradeoffs are generally in favor of the more accurate factorizations. This is especially true when several systems with the same matrix must be solved since one can amortize the preprocessing cost over several systems.

There have been two distinct ways of developing incomplete factorization preconditioners with improved accuracy. The first approach is based on a symbolic factorization view, i.e., it only requires the nonzero structure of the matrix to determine which fill-ins to drop. A level of fill is recursively attributed to each fill-in element, from the levels of fill-in of its parents, in the Gaussian elimination process. Then each fill-in that is introduced and whose level exceeds a certain threshold is dropped. The second common approach is to modify a given direct solver by including a dropoff rule, based on the numerical size of the fill-ins introduced [8,10,5,4,15,14]. The level-of-fill approach may not be reliable for some indefinite problems, since the strategy ignores the numerical values. Although the relation between the size of the dropped elements and the number of iterations required to achieve convergence is far from being understood, *on the average* dropping small elements is more likely to produce a better quality preconditioner than dropping large elements. However, experience reveals that this is not always true. Another drawback of the level-of-fill approach is that it is difficult to predict the amount of fill-in that will be generated. This is also a problem with the second approach. However, a more serious problem with the second approach is cost. By their nature these techniques are usually far more expensive than those of the first approach.

In this paper we derive a class of incomplete LU factorization preconditioners that lie somewhere in between these two approaches. Unlike the level-of-fill strategy, the new algorithm relies on numerical values for dropping elements. In addition, it can limit the fill-in arbitrarily. Moreover, the data structure required for the implementation is much simpler and the overhead related to the preprocessing is far smaller than those derived from direct solvers.

The emphasis of this paper is on implementation and experimentation rather than on theory. We will show how to implement a specific class of preconditioners for general sparse linear systems. We will then test these preconditioners on a few sample problems.

2. Standard incomplete LU (ILU) factorizations

Consider a general sparse matrix A whose elements are a_{ij}, $i, j = 1, \ldots, N$. The incomplete LU factorization process computes a lower triangular matrix L and an upper triangular matrix U such that $A = LU + E$, where E is some error matrix. For the sake of completeness and in order to introduce the basic implementations of incomplete factorizations we first give a description of the ILU(0) factorization.

The incomplete factorization technique with no fill-in consists of performing the i, j, k version of Gaussian elimination, and dropping any element in L and U that falls outside the pattern of A. In the following we will denote by $b_{i,*}$ the ith row of a given matrix B.

We will denote by $NZ(B)$, the set of pairs (i, j), $1 \leq i, j \leq N$ such that $b_{i,j} \neq 0$.

Algorithm 2.1. ILU(0)

For $i = 2, \ldots, N$ Do:
  Define $u_{i,*} = a_{i,*}$
  For $k = 1, \ldots, i-1$ and if $(i, k) \in NZ(A)$ Do:
    Compute the pivot $l_{ik} = u_{ik}/u_{kk}$
    For $j = k+1, \ldots, n$ and if $(i, j) \in NZ(A)$, Do:
      compute $u_{ij} = u_{ij} - l_{ik}u_{kj}$.
    endfor
  endfor
endfor

The existence of the above ILU(0) factorization is guaranteed under a few simple and restrictive assumptions on the coefficient matrix [9]. In some cases, one can put the factorization in the form

$$A = (D - E)D^{-1}(D - F)$$

in which E and F are the strict lower and strict upper triangular parts of A, and D is some diagonal matrix. This means that it suffices in these cases to find a recursive formula for the elements in D and that only an extra diagonal of storage is required. The equivalence between the two incomplete factorizations is valid when the product of the strict lower part and the strict upper part of A consists only of diagonal elements and fill-ins. This is true for example for standard 5-point difference approximations to second-order partial differential operators.

By definition, the L and U matrices in ILU(0) have together the same number of nonzero elements as the original matrix A. The accuracy of this incomplete factorization may be insufficient to yield an adequate rate of convergence. The idea of high accuracy ILU factorization has been proposed by Meijerink and van der Vorst [9] at least for structured matrices arising from 5-point and 7-point matrices. Thus, ILU(1) allows the 'first order fill-ins' which are located in specific diagonals, to be kept. One problem with this is that the definition does not extend to general sparse matrices. Later, Watts [13] generalized this definition by introducing the concept of *level of fill-in*. The idea is quite simple. Any nonzero element in A is initially given a level of fill-in equal to 0. When a fill-in element f_{ij} is created in position (i, j) by a formula such as

$$f_{ij} = -l_{ik} \times u_{kj}$$

its level of fill-in is defined by

$$\text{level}(f_{ij}) = \text{level}(l_{ik}) + \text{level}(u_{kj}) + 1 \tag{2.1}$$

This systematic definition allows one to derive a strategy for discarding elements. Typically, for diagonal dominant matrices, the higher the level of fill-in of an element the smaller its magnitude and this suggests dropping any fill-in element whose level is higher than a certain integer p. However, for more general matrices the size of an element is not necessarily related to its level of fill-in. We will describea variant of ILU(p) which performs the symbolic and

the numeric factorization at the same time.

Algorithm 2.2. ILU(p)
For all nonzero elements a_{ij} define $u_{ij} = a_{ij}$, $lev(u_{ij}) = 0$.

For $i = 2, \ldots, N$ Do
  For each $k = 1, \ldots, i-1$ and if $u_{ik} \neq 0$ do
    Compute $l_{ik} = u_{ik}/u_{kk}$ and set $lev(l_{ik}) = lev(u_{ik})$.
    Compute $u_{i,*} = u_{i,*} - l_{ik}u_{k,*}$.
    Update the levels of $u_{i,*}$ using (2.1)
    Replace any element in row i with $lev(u_{ij}) > p$ by zero.
  EndFor
EndFor

There are a number of drawbacks to the above algorithm. First, the amount of fill-in and computational work for obtaining the ILU(p) factorization is not predictable for $p>0$. Second, the cost of updating the levels can be quite high. Most importantly, for indefinite matrices the level of fill-in may not be a good indicator of the size of the elements that are being dropped. Thus, we may drop large elements and obtain an inaccurate incomplete factorization, in the sense that $E = A - LU$ will be larger. Our experience reveals that *on the average* this will lead to a larger number of iterations to achieve convergence, although there are certainly instances, including a few shown in section 6, where this is not the case. The technique which will be described in the next section has been developed to remedy the above mentioned three difficulties, by focusing on producing incomplete factorizations will small error E.

3. Strategies using numerical thresholds and ILUT

As was seen in the previous sections, the standard implementations of ILU factorizations are based on what is commonly referred to as the i, k, j version of Gaussian elimination, which for dense matrices takes the following form.

Algorithm 3.1. Gaussian elimination—IKJ variant

```
do i=2, n
    do k=1,i-1
        lik = a(i,k) / a(k,k)
        do j=k+1,n
            a(i,j) = a(i,j) - lik * a(k,j)
        enddo
        a(i,k)=lik
        enddo
enddo
```

The above algorithm is in place in that the ith row of A is overwritten by the ith rows of the L and U matrices of the factorization. Each step i of the algorithm generates the ith

row of L and the ith row of U at the same time. The previous rows 1, 2, ..., $i-1$ of L and U are accessed at step i but they are not modified.

In the context of sparse matrices the advantage of this version for incomplete factorizations is that the data structure required is quite simple since the rows of L and U are generated in succession. The modification of the above algorithm that leads to an incomplete factorization requires only (1) to account for sparsity and (2) to include a dropping strategy for getting rid of some fill-ins. In what follows *applying a dropping rule to an element* will only mean 'to replace the element by zero if it satisfies certain criteria of smallness'. Clearly, the zeroed element will then not be stored. A dropping rule can be applied to a whole row by applying the same rule to all the elements of the row.

Algorithm 3.2. Generic incomplete LU factorization with threshold

```
0   row(1:n) = 0
1   do i=2, n
2       row(1:n) = a(i,1:n)                    /* sparse copy */
3       for (k=1,i-1 and where row(k) is nonzero) do
4           row(k) := row(k) / a(k,k)
5           apply a dropping rule to row(k)
6           if (row(k) .ne. 0) then            /* sparse update */
7               row(k+1:n)=row(k+1:n)-row(k)*u(k,k+1:n)
8           endif
9       enddo
10      apply a dropping rule to row(1:n)
11      l(i,1:i-1) = row(1:i-1)                /* sparse copy */
12      u(i,i:n) = row(i:n)                    /* sparse copy */
13      row(1:n) = 0                           /* sparse set-to-zero operation*/
14      enddo
15  enddo
```

A possible implementation would be to use a full vector for row and a companion pointer, which points to the positions of its nonzero elements. This enables efficient sparse vector updates in line 7 [6]. The vector row fills with nonzero elements after the completion of each outer loop i so that it is necessary to zero-out the elements that were just filled in during the course of this loop, as is done in line 13.

We can view ILU(0) as a particular case of the above algorithm. The dropping rule for ILU(0) is simply to drop elements that are in positions not belonging to the original structure of the matrix.

In the factorization ILUT(p, τ) we use the following rule.

1. In line 5, an element $row(k)$ is dropped (i.e., replaced by zero) if it is less than the relative tolerance τ_i obtained by multiplying τ by the original of the ith row (e.g. the 2-norm).
2. In line 10, a dropping rule of a different type is applied. First, we drop again any element in the row with a magnitude that is below the relative tolerance τ_i. Then we keep only the p largest elements in the L part of the row and the p largest elements in the U part of the row in addition to the diagonal element which is always kept.

The result of the second step is that the number of elements per row is controlled. In fact, roughly speaking, we can view p as a parameter that helps control memory usage,

while τ allows us to reduce computational cost. There are several possible variations on the implementation of step 2. For example we can simply keep a number of elements equal to $nu(i) + p$ in the upper part and $nl(i) + p$ in the lower part of the row, where $nl(i)$ and $nu(i)$ are the number of nonzero elements in the L part and the U part of the ith row of A respectively. Note that no pivoting is ever performed. Partial pivoting may be incorporated at the expense of substantially complicating the code. In [11] we implemented and tested a version of ILUT which performs *column pivoting*, allowing us to use essentially the same simple data structure as with the non-pivoting code. An additional permutation array is required to record the corresponding permutation of the variables. We found from recent experience with pivoting, that the rewards gained from pivoting are variable. If a matrix is not close to being diagonally dominant, it may be preferable to use an incomplete factorization that is derived from a direct solver, or a different iterative technique altogether [11], such as one based on preconditioning the normal equations.

Another issue that is not addressed here is that of using various standard permutations employed in the context of direct solvers to reduce fill-in. The two most popular of these are the minimal degree ordering and the nested dissection ordering. We found that reordering in the context of incomplete factorizations can also be beneficial *provided enough accuracy is used*. For example, when a red-black ordering is used, ILU(0) can lead to poor performance compared with that obtained from ILU(0) using the natural ordering. On the other hand if we use ILUT with slightly more fill-in, then the performance starts improving again. In fact an interesting observation we made in [12] is that the performance of ILUT for the red-black ordering *eventually outperforms* that of ILUT for the natural ordering using the same parameters p and τ.

4. Analysis

Existence theorems for the ILUT factorization are similar to those of other incomplete factorizations. If the diagonal elements of the original matrix are positive while the off-diagonal elements are negative, then under certain conditions of diagonal dominance the matrices generated during the elimination will have the same property. If the original matrix is diagonal dominant then the transformed matrices will also have the property of being diagonally dominant under certain conditions. These properties will be analyzed in detail in this section.

We will denote the row vector row resulting from line 4 of Algorithm 3.2 by $u_{i,*}^{k+1}$. Note that $u_{i,j}^{k+1} = 0$ for $j \leq k$. Lines 3 to 10 in the algorithm, involve a sequence of operations of the form

$$l_{ik} := u_{ik}^k / u_{kk} \tag{4.1}$$

if l_{ik} small enough set $l_{ik} = 0$

else:

$$u_{i,j}^{k+1} := u_{i,j}^k - l_{ik} u_{k,j} - r_{ij}^k \quad j = k+1, \ldots, N \tag{4.2}$$

for $k = 1, \ldots, i-1$, in which initially $u_{i,*}^1 := a_{i,*}$ and where r_{ij}^k is an element subtracted from a fill-in element which is being dropped. It should be equal either to zero (no dropping) or to $u_{i,*}^k - l_{ik} u_{k,*}$ when the element $u_{i,j}^{k+1}$ is being dropped. At the end of the ith step of

Gaussian elimination (outer loop in Algorithm 3.2 we obtain the ith row of U,

$$u_{i,*} \equiv u^i_{i-1,*} \tag{4.3}$$

and the following relation is satisfied.

$$a_{i,*} = \sum_{k=1}^{i} l_{k,j} u^k_{i,*} + r_{i,*}$$

where $r_{i,*}$ is the sum of all the fill-in rows.

The existence result which we will prove will only be valid for certain modifications of the basic ILUT(p,τ) described earlier. We will consider an ILUT strategy which uses one of the two following modifications.

4.1. *Modifications*

- Elements generated in the original nonzero pattern of A are not subject to the dropping rule, regardless of their size. Elements in other locations are subject to the same dropping rule as before.
- For any $i < N$ let a_{i,j_i} be the element of largest modulus among the elements $a_{i,j}$, $j = i+1, \ldots, N$. Then elements generated in position (i, j_i) during the ILUT procedure are not subject to the dropping rule.

The first modification clearly implies that the condition required by the second is satisfied. In other words the second modification is the weaker of the two and we can restrict ourselves to proving the main result with this modification as the underlying assumption. Note that these modifications aim at preventing elements generated in position (i, j_i) from ever being dropped and that there may be many alternative strategies that can lead to the same property.

Following Axelsson and Barker [3], we will call an $\hat{M}$ matrix a matrix H whose entries h_{ij} satisfy the following three conditions:

$$h_{ii} > 0 \quad \text{for} \quad 1 \le i < N \quad \text{and} \quad h_{NN} \ge 0 \tag{4.4}$$

$$h_{ij} \le 0 \quad \text{for} \quad i, j = 1, \ldots, N \quad \text{and} \quad i \ne j \tag{4.5}$$

$$\sum_{j=i+1}^{N} h_{ij} < 0, \quad \text{for} \quad 1 \le i < N \tag{4.6}$$

The third condition is simply a requirement that there be at least one nonzero element to the right of the diagonal element, in each row except the last. The row ssum for the ith row is defined by

$$rs(h_{i,*}) = \sum_{j=1}^{N} h_{i,j}$$

A given row of an $\hat{M}$ matrix H is *diagonally dominant* if its row sum is nonnegative. An $\hat{M}$ matrix H is said to be diagonally dominant if all its rows are diagonally dominant.

In the following we establish an existence result for ILUT that is similar to Theorem 1.14 in Axelsson and Barker [3] for ILU(0). The underlying assumption is that an ILUT strategy is used with modification 2.

Theorem 4.1. *If the matrix A is a diagonally dominant $\hat{M}$ matrix, then the rows $u_{i,*}^k$, $k = 0, 1, 2, \ldots, i$ defined by (4.2) starting with $u_{i,*}^0 = 0$ and $u_{i,*}^1 = a_{i,*}$ satisfy the following relations for $k = 0, 1, \ldots, i-1$:*

$$u_{ij}^{k+1} \leq 0 \quad j \neq i \tag{4.7}$$

$$rs(u_{i,*}^{k+1}) \geq rs(u_{i,*}^k) \geq 0, \tag{4.8}$$

$$u_{ii}^{k+1} > 0 \quad \text{when} \quad i < N \quad \text{and} \quad u_{NN}^{k+1} \geq 0 \tag{4.9}$$

Proof The result can be proved by induction on k. It is trivially true for $k = 0$. To prove that the relation (4.7) is satisfied we start from the relation,

$$u_{i,*}^{k+1} := u_{i,*}^k - l_{ik} u_{k,*} - r_{i*}^k$$

in which $l_{ik} \leq 0, u_{k,j} \leq 0$. Either r_{ij}^k is zero which yields $u_{ij}^{k+1} \leq u_{ij}^k \leq 0$, or r_{ij}^k is nonzero which means that u_{ij}^{k+1} is being dropped, i.e., replaced by zero, and therefore again $u_{ij}^{k+1} \leq 0$. This establishes (4.7). Note that by this argument $r_{ij}^k = 0$ except when the jth element in the row is dropped in which case $u_{ij}^{k+1} = 0$ and $r_{ij}^k = u_{ij}^k - l_{ik} u_{k,j} \leq 0$. Therefore, we always have $r_{ij}^k \leq 0$. Moreover, when an element in position (i, j) is not dropped we have $u_{i,j}^{k+1} := u_{i,j}^k - l_{ik} u_{k,j} \leq u_{i,j}^k$ and in particular by our rule in modification 2, for $i < N$, we will always have for $j = j_i$,

$$u_{i,j_i}^{k+1} \leq u_{i,j_i}^k \tag{4.10}$$

in which j_i is defined in the statement of the modification.

Consider the row sum of u_{i*}^{k+1}. We have

$$\begin{aligned} rs(u_{i,*}^{k+1}) &= rs(u_{i,*}^k) - l_{ik} rs(u_{k,*}) - rs(r_{i*}^k) \\ &\geq rs(u_{i,*}^k) - l_{ik} rs(u_{k,*}) \qquad (4.11) \\ &\geq rs(u_{i,*}^k) \qquad (4.12) \end{aligned}$$

which establishes (4.8).

It remains to prove (4.9). From (4.8) we have, for $i < N$,

$$\begin{aligned} u_{ii}^{k+1} &\geq \sum_{j=k+1,N} -u_{i,j}^{k+1} = \sum_{j=k+1,N} |u_{i,j}^{k+1}| \qquad (4.13) \\ &\geq |u_{i,j_i}^{k+1}| \geq |u_{i,j_i}^k| \geq \ldots. \qquad (4.14) \\ &\geq |u_{i,j_i}^1| = |a_{i,j_i}| \qquad (4.15) \end{aligned}$$

Note that the inequalities in (4.14) are true because u_{i,j_i}^k is never dropped by assumption and as a result (4.10) applies. By the condition (4.6) defining $\hat{M}$ matrices, $|a_{i,j_i}|$ is positive for $i < N$. Clearly when $i = N$ we have by (4.13) $u_{NN} \geq 0$. This completes the proof. ■

We would like to point out that the result given above does not mean that the factorization will only be efficient when the conditions of the theorem are satisfied. In practice the preconditioner can be efficient under far more general conditions.

5. Implementation details

Implementation details of the ILUT preconditioning can be quite important, because a poor implementation may lead to a higher complexity in terms of number of arithmetic operations. The following is a list of the potential difficulties that can lead to inefficiencies in the implementations of ILUT.

1. Generation of the linear combination of rows of A (Line 7 in Algorithm 3.2);
2. Selection of the p largest elements in L and U;
3. Need to access the elements of L in increasing order of columns (in Line 3 of Algorithm 3.2).

For (1) the usual technique is to generate a full row and accumulate the linear combination of the previous rows in it. The row is zeroed again after the whole loop is finished using a sparse set-to-zero operation. A variation on this technique which we adopted is to use only a full integer array $jr(1 : n)$ the values of which are zero except where there is a nonzero element. With this full row we need to maintain a short real vector $u(1 : maxu)$ which contains the real values of the row, as well as a corresponding short integer array $ju(1 : maxu)$ which points to the column position of the real values in the row. When a nonzero element resides in position j of the row, then $jr(j)$ is set to the address k in u, ju where the nonzero element is stored. Thus, $ju(k)$ points to $jr(j)$ and $jr(j)$ points to $ju(k)$ and $u(k)$. This is illustrated in Figure 1.

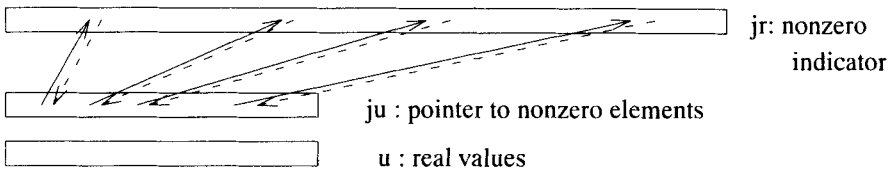

Figure 1. Illustration of data structure used for working row in ILUT

Note that jr holds the information on the row consisting of both the L part and the U part of the LU factorization. When the linear combinations of the rows are performed, we first determine the pivot. Then, unless it is small enough to be dropped according to the dropping rule being used, we proceed with the elimination. If a new element in the linear combination is not a fill-in, i.e., if $jr(j) = k \neq 0$, then we update the real value $u(k)$. If it is a fill in ($jr(j) = 0$) then we append an element to the arrays u, ju and update jr accordingly.

For (2), we have tried several alternatives. The simplest alternative is to use a bubble sort technique, modified to account for the fact that we do not need to sort all the elements but only to obtain the p largest elements in front of the array. This essentially requires exiting after p steps of the main loop of the bubble sort algorithm. If we need to extract the p largest elements from an array of m numbers then the cost of this technique is about $p \times m$ in the worst case. There are better alternatives. The first is to use a heap-sort strategy, again

modified so as to stop when the p largest elements have been extracted. The cost of this is $O(m + p \times \log_2 m)$, i.e., $O(m)$ for the heap construction and $O(\log_2 m)$ for each extraction. Finally, the next technique based on a quick-sort strategy is based on the fact that we need not actually sort the array but only extract the largest p elements. We call this a quick-split technique as opposed to the full quick-sort strategy. The method consists of choosing an element, e.g., $x = u(1)$, in the array $u(1 : m)$, then permuting the data so that $|u(k)| \leq |x|$ if $k \leq mid$ and $|u(k)| \geq |x|$ if $k \geq mid$, where mid is some split point. If $mid = p$ then we exit. Otherwise we split *one of the left or right subarrays* recursively, depending on whether mid is smaller of larger than p. The cost of this strategy *on the average* is $O(m)$. The savings relative to the bubble sort scheme are small for small values of p but they become rather significant for large p and m. For example, on the CRAY-2, when p exceeds 10 and for a three-dimensional Laplace operator matrix it is not too uncommon to save over 30% in time in the ILUT factorization when using quick-split versus a bubble sort. Thus, the sorting does constitute an important part of the computation for large enough p. For small p its impact on the overall ILUT time is rather minimal. Most of the time consumed by the ILUT factorization is spent in the linear combination of sparse rows.

The next implementation difficulty is that the elements in the L part of the row being built are not in increasing order of columns. Since we need to access these elements from left to right in the elimination process, we must scan all elements in the row after the ones already eliminated, and pick the one with smallest column number. This operation can be efficiently organized as a binary search tree which allows easy insertions and searches. However, our current code does not incorporate these improvements which are rather complex to implement and may yield relatively small rewards.

6. Numerical experiments

We tested our ILUT code on many examples. The purpose of this section is to report on just a few results and to comment on our experience with the code. None of the matrices involved in these experiments is symmetric, and none of them is diagonally dominant.

As a first example we consider the partial differential equation:

$$-\Delta u + \gamma \left(e^{xy} \frac{\partial u}{\partial x} + e^{-xy} \frac{\partial u}{\partial y} \right) + \alpha u = f \tag{6.1}$$

with Dirichlet boundary conditions and $\gamma = 10, \alpha = -60$; discretized with centered differences on a $27 \times 27 \times 27$ grid. This leads to a linear system with $N = 25^3 = 15\,625$ unknowns. The experiments have been performed on a CRAY 2. In order to make the experiments realistic, we implemented the level-scheduling technique for backward and forward solution using the jagged diagonal data structure. For details, see [2,1]. We have tested the preconditioned matrix obtained from a natural ordering as well as from permuting the matrix into the red-black ordering.

6.1. *Natural ordering*

We experimented with two ways of varying the parameters τ and p. The first way was to simply fix τ to a certain threshold value, in the current test $\tau = 0.0005$, and vary p, the maximum number of nonzero elements per row allowed in L and U. The execution times

are plotted in Figure 2. Notice how the execution time for performing the ILUT factorization increases with p while at the same time the execution time for the GMRES acceleration decreases. The number of direction vectors used in GMRES is fixed to 10. An optimal value of p is reached at $p = 8$.

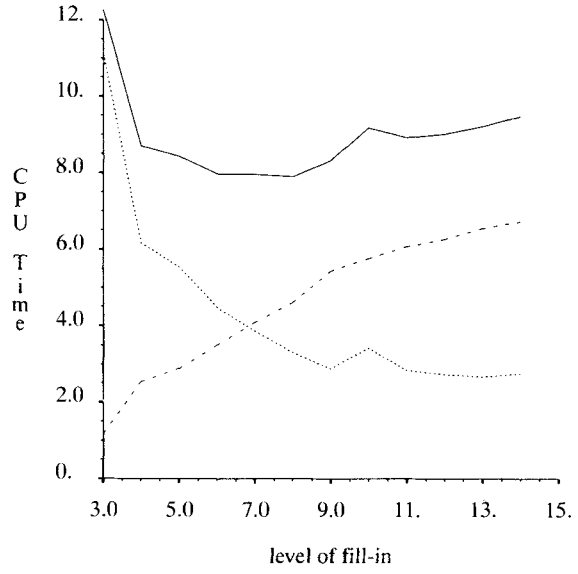

Figure 2. CPU time as a function of the fill-in. Dashed line = ILUT time, dotted line = GMRES time, solid line = total

The second way is to fix p to a maximum value ($p = 15$ in the current experiment) and let τ vary. Specifically, τ is of the form $\tau = \tau_0/2^k$, with $\tau_0 = 0.1$ and the x-axis shows the power k. The motivation for this is that we may wish to select a given accuracy but would like at the same time to limit the amount of memory required. Figure 3 shows the corresponding results. We found that generally speaking, this is a slightly more economical approach in terms of overall execution speeds, when compared with the previous approach of relying on the p parameter to limit the total number of operations.

6.2. *Tests with red-black orderings*

We took the same matrix as in the previous example and reordered it using the red-black ordering. This usually makes the problems more difficult to solve by the simple ILU(0) preconditioner. One of the the purposes of this experiment is to show the importance of higher accuracy in such cases. For the reordered problem both ILU(0) and MILU(0) failed to converge with GMRES(m) using $m = 10$ and $m = 20$. The plot in Figure 4 is the analogue of the one in Figure 3 for this case. The parameters and their meanings are identical. We do not plot the analogue of Figure 2 because of the similarity of the results and the conclusions. We observe that the better performance is reached for the more accurate factorizations.

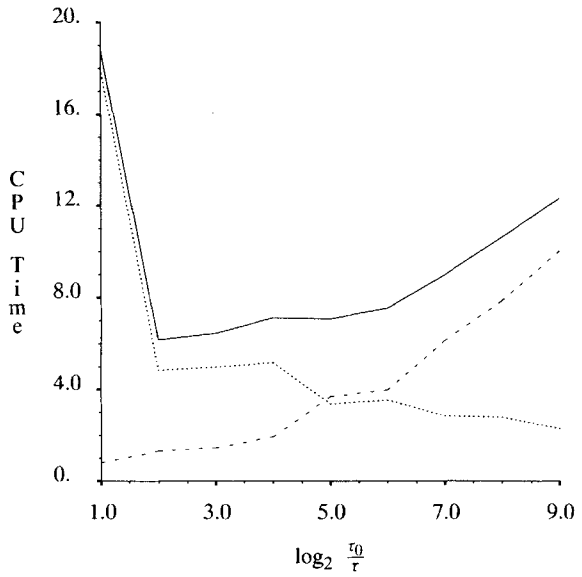

Figure 3. CPU time as a function of the numerical threshold. Dashed line = ILUT time, dotted line = GMRES time, solid line = total

6.3. *Tests with Harwell–Boeing matrices*

Some of the matrices in the Harwell–Boeing collection [7] can cause serious challenges to iterative methods. We have performed a number of tests of ILUT on some of these matrices and made the following observation:

1. In general low accuracy ILU factorizations, e.g., ILU(0) or ILUT with small p and large τ do not constitute a reliable approach.
2. By increasing the accuracy far enough there always comes a point where convergence is reached.
3. Often this point is reached suddenly rather than progressively.
4. There is always a level beyond which it pays very little to increase the accuracy of the preconditioner.

As a result of (3) it is difficult to imagine a practical technique for determining in advance what the best values for the parameters p and τ should be. It suggests that the best strategy is probably to recompute the preconditioner whenever convergence is not satisfactory. Unfortunately, the work in computing the previous ILU cannot be exploited. In addition, the cost will ultimately become close to that of a direct solver, and a rather poor one since preordering is not used to minimize fill-in. Nevertheless, this opens up the possibility of developing simple 'black-box' solvers which will exploit iterative solvers and shift more towards direct solvers when the problem is difficult to solve by a combination such as ILUT-GMRES. The solver can start by performing a reordering of the equations as is usually done for direct solvers. This can be a symbolic reordering only such as nested dissection. Then an outer loop will compute an ILUT factorization and then call the preconditioned GMRES. Here, an ILUT variant that incorporates column pivoting may be a more sensible choice. If

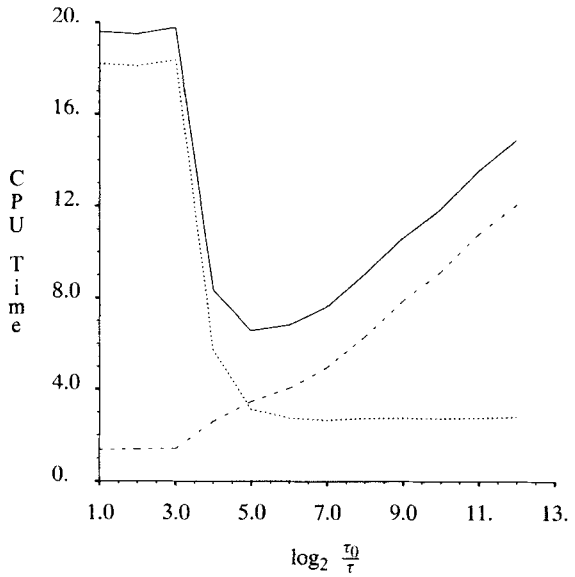

Figure 4. CPU time as a function of the numerical threshold for the red-black ordering. Dashed line = ILUT time, dotted line = GMRES time, solid line = total

convergence is deemed satisfactory, then no further ILU's are computed and the GMRES is run until convergence. Otherwise a more accurate factorization is computed and GMRES is called again. Clearly, the details of this heuristic regarding in particular the selection of the new parameters p, τ each time ILUT is called, are likely to be quite important. For difficult problems, the technique will gracefully degenerate into a technique that has the complexity of a direct solver.

In what follows we make a few observations on some additional experiments with Harwell–Boeing matrices. All right-hand sides are generated as before, i.e., $b = Ae$ where e is the vector consisting of all ones. The initial vectors are always generated randomly and the convergence test is to stop the iteration when the 2-norm of the residual vector has been reduced by a factor of 10^7.

Sherman3. Sherman3 is a matrix of size $N = 5005$ which has a total of $NZ = 20\,033$ nonzero elements. This is in fact a 7-point matrix on a three-dimensional grid of size $35 \times 11 \times 13$ which arises from a reservoir simulation problem [7]. In Figure 5 we show the execution times as a function of the numerical threshold. Again, τ is of the form $\tau = \tau_0/2^k$, with $\tau_0 = 0.1$ and the x-axis shows the power k. The ILU(0) factorization is computable but ILU(0)-GMRES(10) does not converge for this problem.

Pores_2. Pores_2 is a matrix of size $N = 1224$ which has $NZ = 9613$ nonzero elements. It originates from reservoir simulation. For this matrix it is not even possible to compute the ILU(0) and MILU(0) factorizations. The incomplete factorization ILUT with $p = 7$ and $\tau = 0.0001$ required 38 iterations to reduce the initial residual by 10^{-7} with GMRES(10). The number of iterations decreases to 20 if we increase p to 10 and leave τ unchanged.

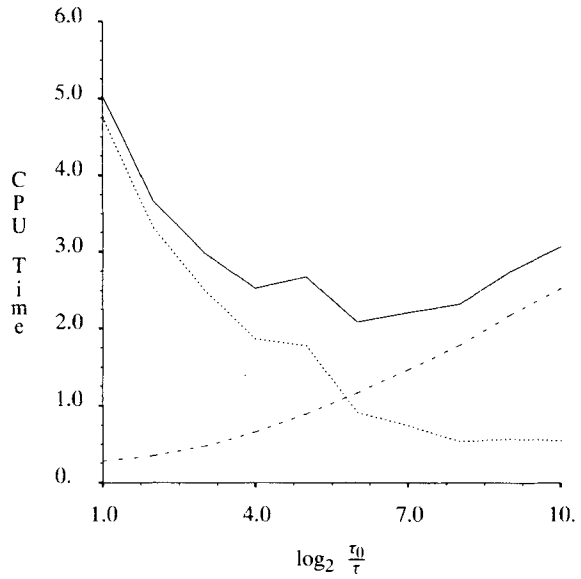

Figure 5. CPU time as a function of the numerical threshold for the Harwell-Boeing matrix Sherman3. Dashed line = ILUT time, dotted line = GMRES time, solid line = total

Sherman2. Sherman2 is a difficult test matrix for iterative methods. The matrix size is only $N = 1080$ but the number of nonzero elements is $NZ = 23\,094$. It is far from diagonally dominant. More precisely, only 6.8% of the rows and 58.7% of the columns are diagonally dominant. The matrix has very large elements (the largest element is approximately 1.34E+9). ILU(0) and MILU(0) were also not computable and terminated with an overflow condition. Similarly, ILUT/GMRES had difficulties converging when the accuracy required is too low. We tested with increments of the accuracy in the following way. First p was set to a maximum of $p = 30$. Then a starting value of $\tau = 0.0001$ was selected and ILUT was computed and tested with GMRES(20) for this value. The test was repeated with smaller values of τ obtained by dividing τ by 10 several times. The last test was with $\tau = 0$. The results are summarized in Table 1 which shows the behavior of the combination ILUT(30,τ)-GMRES(20) as τ varies. The last column in the table shows the actual storage required for the two factors L and U.

Table 1. Performance for the matrix Sherman2 of GMRES(20)-ILUT(30,τ), as τ varies

τ	ILUT time	GMRES time	Total time	Iter	Storage
0.1E-04	0.361E+00	0.230E+01	0.266E+01	120	0.268E+05
0.1E-05	0.417E+00	0.967E+00	0.138E+01	36	0.307E+05
0.1E-06	0.806E+00	0.111E+01	0.192E+01	31	0.407E+05
0.1E-07	0.123E+01	0.772E+00	0.200E+01	19	0.468E+05
0.0E+00	0.637E+01	0.389E+00	0.676E+01	7	0.580E+05

The maximum number of iterations allowed here is 120 so the first line indicates a non-

converging or a slowly converging process. We should point out that the errors obtained in this example are large compared with previous examples but this is expected because the matrix is quite ill-conditioned. The initial residual is around 2×10^9 and the algorithm is stopped as soon as the residual is reduced by a factor of 10^7.

There are a few interesting observations that can be made from the above table. First, note that even for very small values of τ there are differences in performance. For example, unexpectedly, there is a big difference between τ= 0.1E-07 and $\tau = 0.0$. In other words, the upper limit of p=30 is probably not reached most of the time for $\tau \neq 0$. A second observation is that setting a very small value for τ can be quite expensive.

Observe that roughly speaking, the more memory we use the better the performance of the iterative solver. The storage requirement is far from reaching the maximum allowed by p which would give us room for roughly $N \times 30 \times 2 + N = 65\,880$ locations. In addition, the storage that would be required by a skyline code on this example would be approximately 342 585 which is about 8.4 times as large as that obtained with τ=0.1E-06. However, if we were to compare with more efficient direct solvers, the advantage of the iterative solvers might simply be lost in this particular example unless efficient reordering techniques similar to the ones used for direct solvers are also used before ILUT is attempted.

Saylr3. Saylr3 is a 7-point matrix associated with a $10 \times 10 \times 10$ grid. Its size is $N = 1000$ and the number of its nonzero elements is $NZ = 3750$. The reason this matrix causes difficulties for iterative solvers is that it is exactly singular since it contains two rows that are linearly dependent. ILU(0) and MILU(0) both ended with an error because of a zero pivot in row 989. Our initial version of ILUT also failed to compute the incomplete factorization, ending with a similar error message. However, the current version replaces a zero diagonal element by a small element. Specifically, it inserts the value (0.001+tol)*norm(row). This new version is able to compute the ILUT factorization and to make GMRES(10) converge quite well for this problem. Note that when the right-hand side is consistent, i.e., if it is in the range of the columns of A then GMRES is insensitive to the singularity of the (preconditioned) matrix, since it attempts to compute a least squares solution. If the right-hand side is consistent, as is the case in this example then GMRES, when it converges, will find a zero-residual solution whenever the initial guess is not taken to be a zero-vector.

7. Conclusion

When incomplete factorization preconditioners are used in combination with an iterative process such as the conjugate gradient method, they tend to dictate the convergence behavior more than the iterative process itself. The standard ILU(0) and SSOR preconditioners rely more on the accelerator, i.e., the conjugate gradient method, to achieve convergence. As is shown in the numerical experiments this may not be reliable for hard problems. Incomplete factorization preconditioners which rely on threshold dropping can be expensive. However, with a good implementation the additional cost can often be offset by the gain in the acceleration part of the process. In addition, the gains for the common situation where there are many linear systems to solve with the same matrix can be quite substantial. The only disadvantage of ILUT is that it is difficult to optimize on high-performance computers. In spite of this difficulty, the contribution of CPU time incurred in the preprocessing phase can be kept to a reasonable level compared with the rest of the computation which can be

highly optimized. Nevertheless, a parallel version of ILUT would certainly be quite useful and work in this direction is currently under way.

Acknowledgements

The author would like to acknowledge the support of the Minnesota Supercomputer Institute which provided the computer facilities and an excellent research environment to conduct this research.

REFERENCES

1. E. C. Anderson. *Parallel implementation of preconditioned conjugate gradient methods for solving sparse systems of linear equations.* Technical Report 805, CSRD, University of Illinois, Urbana, IL, 1988. (MS thesis).
2. E. C. Anderson and Y. Saad. *Solving sparse triangular systems on parallel computers. International Journal of High Speed Computing*, 1, 73–96, 1989.
3. O. Axelsson and V. A. Barker. *Finite Element Solution of Boundary Value Problems.* Academic Press, Orlando, FL, 1984.
4. E. F. D'Azevedo, F. A. Forsyth, and W. P. Tang. *Ordering methods for preconditioned conjugate gradient methods applied to unstructured grid problems. SIAM J. Math. Anal. Appl.*, 13, 944–961, 1992.
5. E. F. D'Azevedo, F. A. Forsyth, and W. P. Tang. *Towards a cost effective ILU preconditioner with high level fill. BIT*, 31, 442–463, 1992.
6. I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Clarendon Press, Oxford, 1986.
7. I. S. Duff, R. G. Grimes, and J. G. Lewis. *Sparse matrix test problems. ACM trans. Math. Soft.*, 15, 1–14, 1989.
8. K. Gallivan, A. Sameh, and Z. Zlatev. *A parallel hybrid sparse linear system solver. Computing Systems in Engineering*, 1(2-4), 183–195, 1990.
9. J. A. Meijerink and H. A. van der Vorst. *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. Math. Comp.*, 31(137), 148–162, 1977.
10. O. Osterby and Z. Zlatev. *Direct methods for sparse matrices.* Springer Verlag, New York, 1983.
11. Y. Saad. *Preconditioning techniques for indefinite and nonsymmetric linear systems. Journal of Computational and Applied Mathematics*, 24, 89–105, 1988.
12. Y. Saad. *Supercomputer implementations of preconditioned Krylov subspace methods.* In *Algorithmic Trends in Computational Fluid Dynamics*, pages 107–136. Springer Verlag, New-York, 1993..
13. J. W. Watts-III. *A conjugate gradient truncated direct method for the iterative solution of the reservoir simulation pressure equation. Society of Petroleum Engineer Journal*, 21, 345–353, 1981.
14. D. P. Young, R. G. Melvin, F. T. Johnson, J. E. Bussoletti, L. B. Wigton, and S. S. Samant. *Application of sparse matrix solvers as effective preconditioners. SIAM J. Scient. Stat. Comput.*, 10, 1186–1199, 1989.
15. Z. Zlatev. *Use of iterative refinement in te solution of sparse linear systems. SIAM J. Numer. Anal.*, 19, 381–399, 1982.