Parallel Threshold-based ILU Factorization *

George Karypis and Vipin Kumar

University of Minnesota, Department of Computer Science

Minneapolis, MN 55455, Technical Report #96-061

{karypis, kumar}@cs.umn.edu

Last updated on November 4, 1996 at 3:48pm

Abstract

Factorization algorithms based on threshold incomplete LU factorization have been found to be quite effective in preconditioning iterative system solvers. However, their parallel formulations have not been well understood and they have been considered to be unsuitable for distributed memory parallel computers. In this paper we present a highly parallel formulation of such factorization algorithms. Our algorithm utilizes parallel multilevel k-way partitioning and independent set computation algorithms to effectively parallelize both the factorization as well as the solution of the resulting triangular systems, used in the application of the preconditioner. Our experiments on Cray T3D show that significant speedup can be achieved in both operations; thus, allowing threshold incomplete factorizations to be successfully used as preconditioners in parallel iterative solvers for sparse linear systems.

1 Introduction

The sparse linear systems arising in finite element applications are commonly solved using iterative methods. In particular, as the size of these problems increases, the increased computational and memory requirements of these problems render in-core direct solution methods unusable, leaving iterative methods as the only viable alternative for solving these problems in core.

The major computational kernels of an iterative method are (i) computation of preconditioner, (ii) multiplication of a sparse matrix with a vector, and (iii) application of the preconditioner. Threshold-based incomplete LU factorization have been found to be quite effective in preconditioning iterative system solvers [12]. However, because these factorizations allow the fill elements to be created dynamically, their parallel formulations had not been well understood, and they have been considered to be unsuitable for distributed-memory parallel computers [11]. Furthermore, solution of

*This work was supported by NSF CCR-9423082 and by Army Research Office contract DA/DAAH04-95-1-0538, and by Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute, Cray Research Inc, and by the Pittsburgh Supercomputing Center. Related papers are available via WWW at URL: **http://www.cs.umn.edu/~karypis**

the resulting sparse triangular system (which is required for the application of the preconditioner) is generally more difficult to parallelize than the multiplication of a sparse matrix with a vector. We present a highly parallel formulation of the ILUT factorization algorithm for distributed memory parallel computers. This algorithm uses our parallel multilevel k-way graph partitioning algorithm in conjunction with a parallel maximal independent subset algorithm to parallelize both the factorization as well as the solution of the resulting triangular factors.

Parallel formulations of the threshold-based incomplete factorization has limited scalability and higher memory requirements than the sequential algorithm, because it needs to form the reduce matrix for the interface nodes. Note that the amount of fill in the resulting sparse triangular system is the same.

We also develop a modified ILUT factorization algorithm (ILUT*) that requires less time and is more scalable than ILUT. Our experiments on Cray T3D show that our parallel ILUT* algorithm achieve a high degree of concurrency, and when used as a preconditioner, it is comparable in quality to the unmodified ILUT algorithm. Furthermore, our experiments using the GMRES iterative solver show that the amount of time spent in computing the factorization using the ILUT* algorithm is usually much less than the amount of time required to solve the systems.

We show that highly parallel graph partitioning algorithms in conjunction with parallel algorithms for computing maximal independent sets can be used to develop scalable parallel formulations of incomplete factorizations. A good domain decomposition of the underlying finite element mesh allows the problem to be effectively mapped onto the processors of the parallel computer so that the amount of communication required by each of the above computational kernels is significantly decreased.

2 Preconditioners Based on Incomplete LU Factorizations

It is well known that the rate of convergence of iterative methods depends on the spectral properties of the coefficient matrix [1, 14]. This convergence rate can be greatly improved by transforming the original system $Ax = b$ to an equivalent one (*i.e.*, one that has the same solution), that has more favorable spectral properties. A ***preconditioner*** is a matrix M, that attempts to perform such a transformation. The system $M^{-1}Ax = M^{-1}b$ has the same solution as the original system, but the spectral properties of its coefficient matrix $M^{-1}A$ may be more favorable. For efficient implementation, the matrix M must be such that it is relatively inexpensive to solve linear systems of the form $My = c$.

A broad class of widely used preconditioners is based on incomplete factorizations of the coefficient matrix (ILU) [10, 15, 16, 3, 2, 12]. A factorization is called *incomplete* if during the factorization process certain *fill* elements are ignored. Such preconditioners are given in the form $M = LU$ where L and U are the lower- and upper-triangular matrices of the incomplete factorization. Incomplete factorizations are usually computed by taking a set S of matrix elements, and keeping all positions outside this set equal to zero during the factorization process. The set S can be determined statically prior to the actual incomplete factorization (*e.g.*, using a subset of the non-zero entries of the original matrix) as in ***static-sparsity-pattern*** based incomplete factorization. Alternatively, the set S can be determined dynamically as the factorization progresses as in ***threshold-based*** incomplete factorizations.

Static-sparsity-pattern incomplete LU factorizations based on *levels of fill* [10, 15] are used extensively. In these schemes, the set S is constructed by allowing fill entries to propagate up to k levels, for some constant k. These factorizations are referred to as ILU(k). Out of the class of ILU(k) factorizations, the zero-fill incomplete factorization (ILU(0)) is used most frequently due to its simplicity and low computational requirements. In essence, ILU(0) restricts the set of non-zero elements S to be that of the coefficient matrix, so no fill elements are created in the L and U factors of the matrix.

ILU(k) factorizations for small k (*i.e.*, k in the range of 0 to 2) can be constructed and applied relatively inexpensively. However, for certain ill-conditioned systems, ILU(k) factorizations with large fill levels (*i.e.*, k in the range of 5 to 20) may be required to significantly accelerate the convergence of iterative methods. Unfortunately, for moderately large values of k, the application of the ILU(k) preconditioner becomes expensive, as the L and U factors become quite dense. Furthermore, ILU(k) factorizations are insensitive to the magnitude of the elements, since fill elements are dropped only based upon the structure of the original matrix. This can cause preconditioners to be ineffective for matrices arising in many realistic applications [14].

To strike a balance between reduced computation requirements and increased effectiveness, threshold-based incomplete LU factorizations have been developed [16, 3, 2, 12]. In these factorization schemes, a fill element is dropped during the Gaussian elimination processes based only on the magnitude of the element rather than its location. These schemes usually lead to preconditioners that have moderate computational requirements and are generally more robust than static sparsity pattern incomplete factorizations.

2.1 ILUT Factorization Algorithm

One example of threshold based factorizations is the ILUT(m, t) incomplete factorization algorithm [12]. ILUT employs a dual dropping strategy that is able to control the computational requirements during the factorization as well as during the application of the preconditioner. In general, the ILUT(m,t) algorithm drops any elements whose magnitude is smaller than a threshold t, and out of the remaining non-zero elements in any given row, it keeps the largest m elements in L and the largest m elements in U.

ILUT(m, t) algorithm

```
1.   for i = 1, ..., n
2.       w = a_{i,*}
3.       for k = 1, ..., i - 1
4.           if w_k ≠ 0
5.               w_k = w_k / a_{k,k}
6.               Apply 1st dropping rule to w_k
7.               if w_k ≠ 0
8.                   w = w - w_k * u_{k,*}
9.               endif
10.          endif
11.      endfor
12.      Apply 2nd dropping rule to row w
13.      l_{i,j} = w_j for j = 1, ..., i - 1
14.      u_{i,j} = w_j for j = i, ..., n
15.      w = 0
16.  endfor
```

Algorithm 2.1: The ILUT(m,t) factorization algorithm.

Algorithm 2.1 shows the algorithm for performing the ILUT factorization. In this algorithm, w is a full-length working row which is used to accumulate linear combinations of sparse rows in the elimination and w_k is the kth entry of this row. The ith row of the matrix is denoted by $a_{i,*}$ and the triangular factors are stored in l and u.

As we can see in lines 6 and 12 of Algorithm 2.1, the ILUT factorization algorithm applies two different dropping rules during the factorization. These dropping rules are used to replace non-zero elements with zero elements if they satisfy certain conditions. In the application of the first dropping rule in line 6, an element w_k is dropped if its magnitude is less than the relative tolerance t_i obtained by multiplying t by the original 2-norm of the ith row. In the application of the second dropping rule in line 12, first we drop all elements in the row with a magnitude that is below the relative tolerance t_i, and then we keep only the largest m elements in the L part of the row and the m largest elements in the U part of the row in addition to the diagonal element which is always kept.

The ILUT algorithm can be implemented quite efficiently by using appropriate data structures [12, 14]. A common implementation is to use a full vector for w and a companion pointer which points to the positions of its non-zero elements. As a result, lines 13 and 14 are sparse copy operations, and setting $w = 0$ at the end of the Gaussian elimination (line 15) is done using a sparse operation.

3 Parallel Sparse Factorization Algorithms

Sparse factorization algorithms (both complete and incomplete) can be performed on parallel computers as follows. First, a high-quality graph partitioning algorithm is used to distribute the matrix A among the processors. The nodes (rows) assigned to each processor are classified as interior (rows) or interface nodes (rows), depending on whether or not a node is connected to only local nodes. The partitioning algorithm minimizes the number of interface nodes by reducing the edge-cut. Then, each processor independently factors its set of interior nodes. This is a completely local operation and requires no communication. Next, the unknowns corresponding to all the factored interior nodes are eliminated from the interface rows, forming a reduced matrix A^I, corresponding only to the interface nodes. Finally, all the processors cooperate to factor A^I.

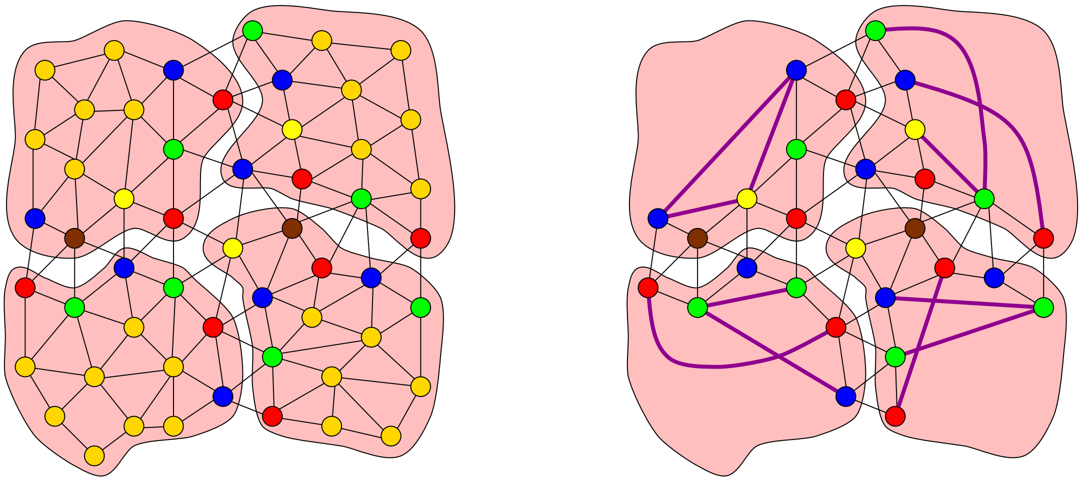
Performing the factorization of A^I in parallel is where the complexity of parallel sparse factorization algorithms resides. Any efficient parallel factorization algorithm has to utilize a highly parallel algorithm for factoring A^I.

Parallel factorization of A^I is often done in phases as follows. During each phase l ($l = 0, 1, \ldots$), the processors cooperate on factoring a set S_l of rows of A^I_l ($A^I = A^I_0$), eliminating the corresponding unknowns from the remaining rows of A^I_l, and forming the next level reduced matrix A^I_{l+1}. Matrix A^I_{l+1} corresponds to the as yet unfactored nodes of A^I. This process continues until all the nodes of A^I have been factored.

A key differentiating feature between various sparse factorization algorithms is how the sets of vertices S_l are computed and whether or not all these sets can be computed prior to performing the actual numerical factorization. The ability to compute the sets S_l prior to factorization is useful for two reasons. First, this information can be used to map the computation onto the processors so that load balance is maintained and communication overhead is minimized during parallel factorization. Second, the sequence of reduced matrices A^I_l does not need to be formed explicitly, which reduces the amount of data movement as well as the memory involved in the factorization; hence, improving the overall performance and scalability of the algorithm. In complete factorization algorithms, the sets S_l are constructed using graph partitioning (*i.e.*, separators in nested dissection orderings) prior to the numerical factorization and a careful mapping of A^I on to processors leads to a highly scalable factorization algorithm [4].

In incomplete factorization algorithms, the sets S_l are often computed as independent set of nodes of the successively reduced matrices A^I_l [5, 9, 14, 11]. Since nodes in S_l are independent (*i.e.*, there is no direct connection between them), all of them can be factored concurrently. In parallel ILU(0) factorizations, the sparsity structure of each reduced matrix A^I_l is known priori since no fill is allowed during the factorization (*i.e.*, obtained from A by simply retaining the non-zero elements between nodes in A^I_l). Hence, these matrices do not have to be explicitly formed in order to compute maximal independent sets. Instead, a coloring of A^I is computed, and each set S_l corresponds to the set of nodes of A^I with the same color. Thus, in ILU(0) (and in all fixed sparsity pattern incomplete factorization algorithms) all sets S_l

can be computed prior to numerical factorization. Figure 1(a) illustrates this idea of coloring the nodes corresponding to A^I (*i.e.*, interface nodes). In this example, the matrix corresponding to A is partitioned into four domains, and each domain is assigned to one of four processors. Each processor factors the nodes internal to its domains, and then the interface nodes are factored iteratively, by concurrently factoring the nodes of each color. However, during the ILUT factorization, the sparsity structure of the matrix A^I changes dynamically. This is illustrated in Figure 1(b). As the factorization of the internal nodes progresses, this creates some fill that adds new dependencies among the interface nodes. As a result, interface nodes that have the same color are not any more independent. Thus, the sets of nodes to be factored concurrently (*i.e.*, the sets S_l) need to be computed dynamically as the factorization progresses [11].



(a) Coloring of nodes for ILU(0) **(b) Fill creates dependencies in ILUT**

Figure 1: Differences between ILU(0) and ILUT factorization algorithms. Since during ILU(0), the sparsity structure of the matrix does not change, a coloring (shown in (a)) of the interface vertices is sufficient to extract concurrency during their factorization. However, ILUT factorization allows fill (shown in (b)) making impossible to use this coloring to determine independent set of nodes and extract concurrency. The dependencies introduced by the added fill is shown using colored edges.

4 Parallel ILUT Factorization

Our parallel formulation of the ILUT algorithm for distributed-memory parallel computers is based on the general framework described in Section 3. It successively computes independent sets of the reduced matrices to extract concurrency during the parallel factorization of the interface nodes. Specifically, our algorithm consists of two phases. In the first phase, each processor computes an ILUT factorization of the interior nodes associated with its domain. During the second phase, a global reduced matrix A^I is formed that corresponds to the interface nodes, and A^I is factored in an iterative fashion using independent sets I_l to extract concurrency (*i.e.*, $S_l = I_l$). The process of finding an independent set and forming a reduced system continues until all the interface nodes have been factored. Figure 2 illustrates this process in the example graph (matrix) shown in Figure 1. In this example, a total of six independent sets are computed in order to factor the interface nodes.

The number of reduced matrices required to perform the entire factorization depends on the initial sparsity structure of the matrix, the maximum number of non-zeros allowed in each row, and the threshold of the incomplete factorization. In general, the number of reduced matrices required by ILUT is greater than that required by ILU(0), and it increases as the amount of fill allowed in the factorization increases.

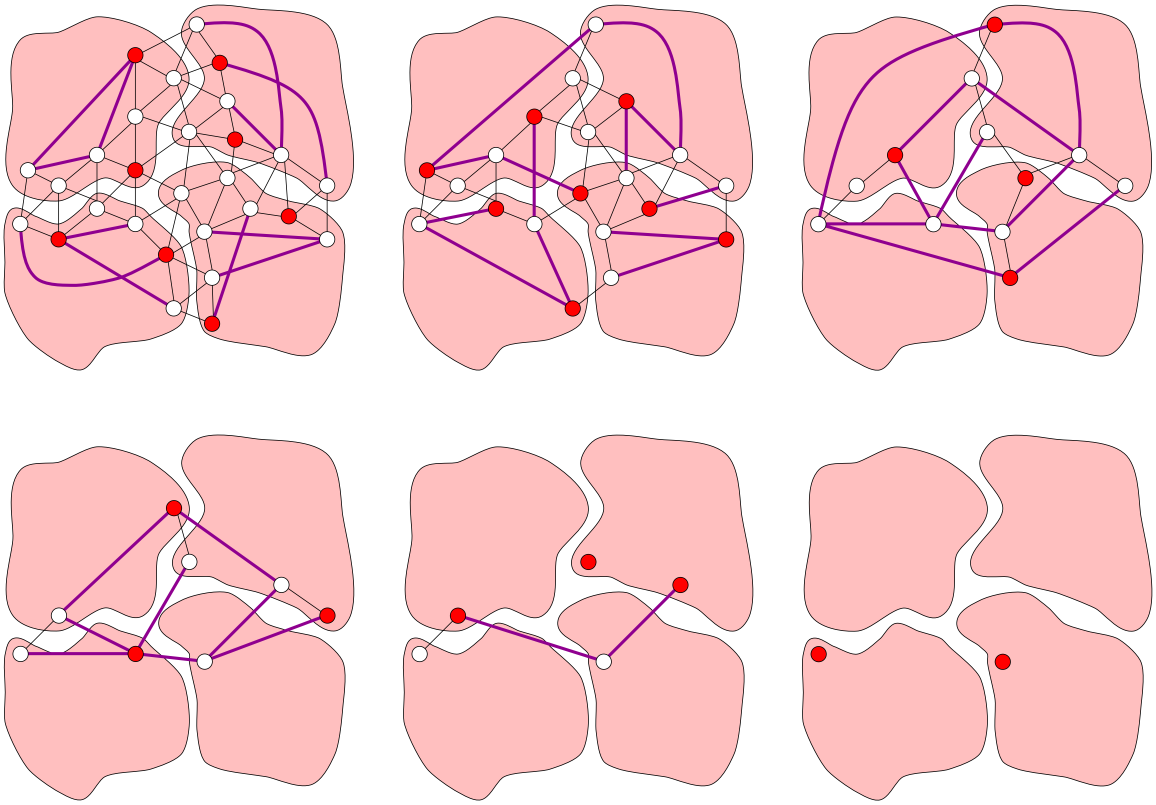


Figure 2: The process of factoring the interface nodes by repeatedly factoring an MIS of nodes forming the reduced system (*i.e.*, new dependency graph). The nodes in the MIS are colored and the new dependencies are shown with colored edges.

The serial implementation of the algorithm to factor the interface nodes works as follows. After I_l is found by using an independent set algorithm, matrix A_l^I is permuted so that the rows corresponding to nodes in I_l are numbered first. The algorithm proceeds to factor the nodes in I_l, using the ILUT algorithm shown in Algorithm 2.1. Having done that, then the next level reduced matrix A_{l+1}^I is formed using the algorithm shown in Algorithm 4.1. This algorithm is derived from the ILUT algorithm and has similar structure. For each row i in $A_l^I \backslash I_l$, the algorithm performs linear combinations (line 8) only with rows that belong in I_l (as it can be seen by the range of values for k in line 3). After all these combinations have been performed, the working vector w is merged with the ith row of L (line 13). The dropping rule applied on line 15, affects only the elements of w whose indices are smaller than n_{l+1}, that is, those that correspond to nodes that already have been factored. For those elements, this rule removes any elements whose magnitude is smaller than the threshold t of the factorization, and of the remaining ones it keeps only the m larger. This dropping rule is similar to the 2nd rule used in the ILUT algorithm, but it only affects the columns of L that correspond to factored nodes. Note that this dropping rule correctly enforces the requirements of the ILUT algorithm of keeping the larger m values in each row of L that are above the threshold. On line 16, the elements of w that correspond to the ith row of the L factor are copied back, and the elements of w that correspond to the unfactored portion of the matrix are copied back to the ith row of A_l^I. This row now becomes the ith row of the next level reduced matrix A_{l+1}^I. Note that the dropping test in line 6 is similar to that used by the ILUT described in Section 2.1.

Our parallel algorithm for computing the ILUT factorization of matrix A^I corresponding to the interface nodes is implemented as follows. During iteration l, an independent set I_l of A_l^I is computed in parallel using the algorithm described in Section 4.1. Every processor performs the ILUT factorization of the locally stored nodes of I_l. Since

Eliminating the first $n_{l+1} - n_l + 1$ unknowns of A_l^I to form the reduced A_{l+1}^I matrix of size $(n - n_{l+1} + 1) \times (n - n_{l+1} + 1)$.
A_l^I has already been permuted so that the rows in I_l are numbered first.

for $i = n_{l+1}, \ldots, n$
  $w = a_{i,*}$
  for $k = n_l, \ldots, n_{l+1} - 1$
    if $w_k \neq 0$
      $w_k = w_k / a_{k,k}$
      Apply 1st dropping rule to w_k
      if $w_k \neq 0$
        $w = w - w_k * u_{k,*}$
      endif
    endif
  endfor
  $w = w \cup l_{i,*}$
  $l_{i,*} = a_{i,*} = 0$
  Apply 3rd dropping rule to row w
  $l_{i,j} = w_j$ for $j = 1, \ldots, n_{l+1} - 1$
  $a_{i,j} = w_j$ for $j = n_{l+1}, \ldots, n$
  $w = 0$
endfor

Algorithm 4.1: The algorithm that used to form successive reduced matrices.

these nodes are independent, this factorization requires only creating the rows of the upper triangular matrix U for each row. After that, each processor computes the rows of the next level reduced matrix A_{l+1}^I that correspond to the locally stored nodes, using the algorithm shown in Algorithm 4.1. In particular, for each row i, the processor needs to perform linear combinations with all rows u_k such that $a_{i,k} \neq 0$ and $n_l \leq k < n_{l+1}$. Some of these u_k rows may be stored on other processors; thus they need to be communicated. However, because the nodes in I_l are independent, the above linear combinations will not create any fill elements in $a_{i,k}$ for $n_l \leq k < n_{l+1}$; thus, the rows that are required can be determined and communicated prior to performing any computation. After all the required rows of U are received, each processor essentially executes the algorithm shown in Algorithm 4.1 on the locally stored nodes, updating the appropriate rows of L and forming the next level reduced matrix A_{l+1}^I. The algorithm terminates when the rows in the reduced matrix are independent, in which case all of them can be eliminated in parallel.

This parallel ILUT algorithm uses our parallel multilevel k-way partitioning algorithm [6] to produce a very good initial domain decomposition with a small number of interface nodes. Consequently, the distributed phase (*i.e.*, second phase) of the algorithm factors only a small number of nodes.

4.1 Computing Maximal Independent Sets

The independent sets of the successively reduced matrices A_l^I are computed using a parallel formulation of Luby's [8] algorithm. A maximal independent set I of a set of vertices S is computed via Luby's algorithm in an incremental fashion as follows. A random number is assigned to each vertex, and if a vertex has a random number that is smaller than all of the random numbers of the adjacent vertices, it is then included in I. Now this process is repeated for the vertices in S that are neither in I nor adjacent to vertices in I, and I is augmented similarly. This incremental augmentation of I ends when no more vertices can be inserted in I. It is shown in [8] that one iteration of Luby's algorithm requires a total of $O(\log |S|)$ such augmentation steps to find a maximal independent set of a S. Since the majority of the independent vertices are discovered during the first few iterations of this algorithm, our parallel independent set algorithm performs only five such augmentation steps. This reduces the run time of the algorithm without significantly reducing the size of the computed independent sets.

This algorithm is able to correctly find an independent set provided that the set of nodes in T forms an undirected graph, *i.e.*, the reduced matrices A_l^I's are structurally symmetric. However, since the sparsity structure of each reduced matrix A_l^I depends on the magnitude of the nonzero entries, these matrices are not in general structurally symmetric. For example consider a two node graph v, u, that are connected via the directed edge (u, v). If 2 is the random number associated with v and 1 is the random number associated with u, then both v and u will be inserted in the independent set. Consequently, the computed set of nodes may not be independent. However, an independent set can still be computed if Luby's algorithm is modified as follows. Vertices are included in I by using the following two step process which are separated by a barrier synchronization. In the first step, vertices that have a random number smaller than their adjacent vertices are first inserted in I. During the second step, all the vertices in I that are adjacent to any vertices also in I are removed from I. Now, the vertices left in I are guaranteed to be independent. This modification to Luby's algorithm requires only an additional communication step involving a set of the interface nodes; thus, it does not increase the asymptotic complexity of the parallel algorithm.

Luby's algorithm can be implemented quite efficiently on a shared memory parallel computer, since for each vertex v, a processor can easily determine if the random value assigned to v is the smaller among all the random values assigned to the adjacent vertices. However, on a distributed memory parallel computer, for each vertex, random values associated with adjacent vertices that are not stored on the same processor needs to be explicitly communicated. In our implementation of Luby's algorithm, prior to computing an independent set, we perform a *communication setup* phase, in which appropriate data structures are created to facilitate this exchange of random numbers. In particular, we pre-determine which vertices are located on a processor boundary (*i.e.*, a vertex connected with vertices residing on different processors), and which are internal vertices (*i.e.*, vertices that are connected only to vertices on the same processors).

4.2 Modified ILUT Factorization (ILUT*)

Recall that during the second phase of our parallel formulation of the ILUT factorization algorithm, a sequence of successively smaller reduced matrices is formed in order to compute independent sets. Even though the L and U factors in ILUT(m, t) have only up to m non-zero elements per row, the number of non-zero entries in the reduced matrices can be significantly larger than that, since the restriction regarding the maximum number of allowed non-zeros is only enforced when a row is actually factored (3rd dropping rule in Algorithm 4.1. In particular, for small tolerance values t, the reduced matrices become quite dense. In many cases there are a few hundred non-zero elements per row, even though we only keep somewhere in the range of 5 to 10 of them in the L and U factors.

Even though forming these reduced matrices is critical to extract concurrency during the factorization of the interface nodes (Section 3), they can significantly affect the performance of the parallel formulation of the ILUT algorithm for the following two reasons. First, these quite dense reduced matrices need to be constructed in each successive step of the algorithm, which significantly increases the memory requirements and the amount of time spent in essentially copying data. Second, as these reduced matrices become denser, the size of the independent sets decrease significantly; hence, the number of iterations required to factor the interface nodes increases accordingly, as well. This reduces the amount of concurrency and increases the synchronization cost. Note that despite the above overheads, the number of floating point operations performed by both the serial and the parallel algorithm are the same (assuming that the same ordering was used).

We have implemented a modification on the original serial ILUT(m, t) algorithm to address this problem. Our modified algorithm, ILUT* (m, t, k) is similar to ILUT, with the following difference. Instead of keeping in the reduced matrices A_l^I, all the non-zero entries whose magnitudes are greater than t, it keeps only the km larger magnitude entries. Essentially, ILUT* modifies the third dropping rule of line 15 of the algorithm used to construct the reduce matrices (Algorithm 4.1), by putting an upper bound on the maximum number of elements kept in each row of the reduced matrices. Our experiments in Section 6 show that for small values of k (*i.e.*, 2 to 4), ILUT* produces factorizations whose quality is comparable to that of ILUT, and it can be computed in parallel much faster than ILUT.

5 Parallel Forward and Backward Substitutions

For parallel preconditioning techniques that are based on incomplete LU factorizations, it is of utmost importance to develop highly parallel formulations of the forward and backward substitutions. The reason is that these are required in the application of the preconditioner in each iteration of the iterative solution method. In fact, preconditioning techniques based on incomplete factorizations are not as widely used on parallel computers, mainly due to the lack of highly parallel triangular solvers.

Our parallel formulation of the triangular solvers utilizes the structure imposed on the triangular systems by our parallel ILUT factorization. Figure 3 illustrates the sparsity structure of L and U after they are permuted according to the order that the factorization was performed. This structure represents a domain decomposition into four domains (*i.e.*, four processors) and two independent set computations. The various parts of the triangular factors are color-coded to indicate the processors on which they reside. Note that the shaded parts of the triangular factors correspond to sparse sub-matrices.

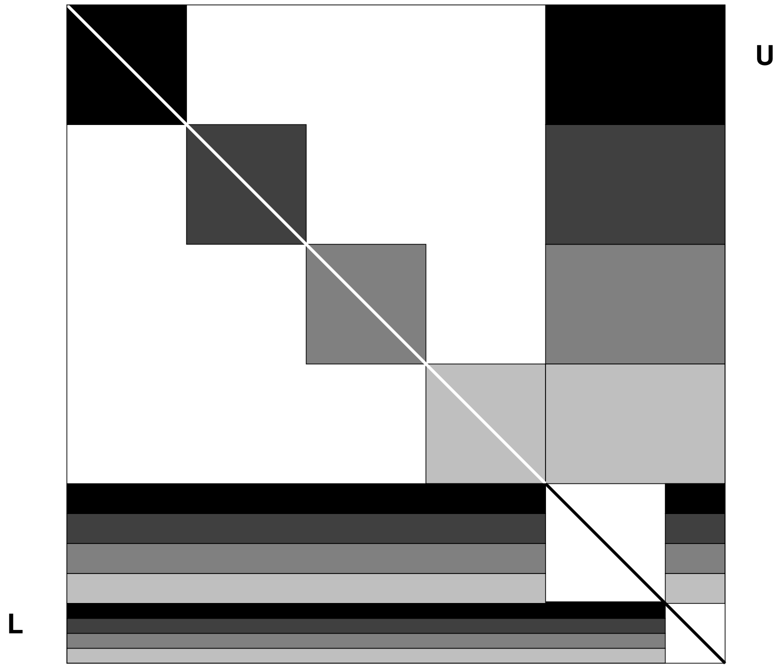


Figure 3: An example of the structure of the lower and upper triangular matrices resulting from the parallel ILUT factorization for 4 processors. The factors are color-coded indicating the processors on which they reside.

Our parallel formulation of the forward and backward substitutions utilizes the same two phase approach used while computing the ILUT factorization. Consider the forward substitution $Lx = b$. During the first phase, each processor solves for the values of the vector x associated with the nodes that are interior to its domain. The second phase consists of q iterations where q is the number of independent sets required to factor the interface nodes in the parallel ILUT factorization. During iteration l, each processor solves for the locally stored nodes of the independent set I_l, and sends these values of x to the processors that have non-zeros along the x column. The backward substitution is performed in a similar fashion but in reverse order. First, the solution of the interface nodes is computed in an iterative fashion, followed by the computation of the solution of the interior nodes.

The amount of communication performed during either forward or backward substitution, is proportional to the number of interface nodes, which is similar to the communication requirements of a matrix-vector multiplication. However, both forward and backward substitutions have q implicit synchronization points that correspond to the q independent sets used to factor the matrix. Thus, if q is relatively small, the performance of the forward and backward substitution is similar to that of a matrix-vector multiplication. In particular, for the ILUT* factorization our experiments show that the cost of performing a forward and a backward substitution is in general, only 30% higher than that of performing the corresponding matrix-vector multiplication, if the matrices in the two cases have similar number of non-zeros.

6 Experimental Results

We implemented our parallel ILUT factorization algorithm on a 128-processor Cray T3D. The T3D is a distributed memory parallel computer, each processor is a 150Mhz Dec Alpha (EV4), and the processors are interconnected via a three dimensional torus network that has a peak unidirectional bandwidth of 150Bytes per second, and a small latency.

We evaluated the performance of our parallel formulation of the ILUT(m, t), and ILUT* (m, t, k) algorithms, for a wide range of values for m and t. Specifically, we let m, the maximum number of non-zeros kept on each row of L and U, take the values of 5, 10 and 20, and we let the threshold t take the values of 10^{-2}, 10^{-4}, 10^{-6}. In the case of ILUT*, we used $k = 2$, keeping only $2m$ non-zeros on each row of the reduced matrix, as this value provides a reasonable balance between reducing the cost of forming the reduce matrices, and retaining enough non-zero elements to effectively emulate the true ILUT factorization. Because of the large number of choices of parameters m, t, and k, we limit our experiments to two representative matrices G40 and TORSO. G40 corresponds to a PDE discretized with centered differences on a $40 \times 40 \times 40$ grid, leading to a system with 54,872 equations and 164616 non-zeros, whereas TORSO is a finite element matrix with 201,142 equations and 1479989 non-zeros arising in computing the ECG fields of the human thorax using Laplace's equation [7].

Performance of the Factorization Table 1 shows the run time of our parallel ILUT and ILUT* algorithms. For each one of the ILUT and ILUT* algorithms, nine different factorizations are shown that correspond to the different choices for m and t.

From this table we see that as either m increases and/or t decreases, the amount of time required to perform the factorization increases. In some cases, this increase in run-time is small while in some other cases it is quite dramatic. For example on 16 processors, the ILUT(20,10^{-6}) factorization requires almost 100 times more time than the ILUT(5,10^{-2}) factorization. Since as m and t increase, more computation is performed by the incomplete factorization, the increases in the run-time are natural. Also, if the incomplete factorizations lead to better preconditioners with increasing value of m and decreasing value of t, then the higher factorization time can be easily offset by a much faster solution time (Table 3). Comparing the time required by the corresponding ILUT and ILUT* factorizations, we see that in general

	G40				TORSO			
Factorization	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
ILUT(5,10^{-2})	0.476	0.319	0.202	0.133	2.643	1.509	0.926	0.534
ILUT(10,10^{-2})	0.536	0.337	0.215	0.148	3.918	2.285	1.275	0.705
ILUT(20,10^{-2})	0.538	0.337	0.221	0.145	4.539	2.643	1.405	0.793
ILUT(5,10^{-4})	1.806	1.250	0.821	0.539	7.880	4.806	3.072	1.826
ILUT(10,10^{-4})	3.509	2.404	1.700	1.197	17.661	10.628	6.955	4.039
ILUT(20,10^{-4})	6.298	4.371	3.040	2.125	34.349	19.719	12.275	7.317
ILUT(5,10^{-6})	6.452	4.416	3.051	1.945	18.143	11.604	7.193	4.643
ILUT(10,10^{-6})	19.336	14.541	10.149	6.316	58.300	38.439	26.042	17.202
ILUT(20,10^{-6})	45.188	34.295	23.788	15.141	140.330	94.417	64.864	43.073
ILUT* (5,10^{-2},2)	0.442	0.292	0.180	0.112	2.646	1.524	0.927	0.538
ILUT* (10,10^{-2},2)	0.495	0.304	0.190	0.126	3.873	2.256	1.251	0.698
ILUT* (20,10^{-2},2)	0.497	0.301	0.191	0.123	4.497	2.592	1.376	0.770
ILUT* (5,10^{-4},2)	1.471	0.917	0.537	0.321	6.845	3.875	2.275	1.284
ILUT* (10,10^{-4},2)	2.962	1.893	1.189	0.711	15.758	8.901	5.284	2.844
ILUT* (20,10^{-4},2)	5.404	3.503	2.216	1.381	32.208	17.938	10.712	6.035
ILUT* (5,10^{-6},2)	3.488	1.848	0.917	0.473	13.141	6.996	3.725	1.975
ILUT* (10,10^{-6},2)	8.831	4.681	2.345	1.197	39.296	20.794	10.779	5.336
ILUT* (20,10^{-6},2)	22.152	12.007	6.720	3.857	100.016	55.822	28.926	15.585

Table 1: The performance of the parallel factorization algorithms. For each matrix, the run time (in seconds) is shown for 16, 32, 64, and 128 processors.

ILUT requires more time. Furthermore, the factor by which ILUT* outperforms ILUT increases as the number of processors increase. For example, for TORSO, the ILUT(20,10^{-6}) is only 1.4 times slower than ILUT* (20,10^{-6},2) on 16 processors but it is 2.7 times slower on 128 processors. This is because on 128 processors, the interface nodes correspond to a large fraction of the matrix than they do on 16 processors; thus, ILUT* is able to accelerate a larger portion of the overall computation required for factorization.

To study the scalability of our parallel ILUT and ILUT* factorization algorithms, we plotted the speedup relative to 16 processors achieved by our algorithms on 32, 64, and 128 processors. Figures 4 and 5 shows these relative speedup curves for all nine different factorizations of G40 and TORSO, respectively. A number of insightful conclusions can be drawn from these figures. Note first that the overall speedup achieved on TORSO are better than those achieved on G40. Since, TORSO is a much larger problem, the parallel overhead is a smaller portion of the total computation, thus, leading to higher efficiency and better speedup.

Comparing ILUT* with ILUT, we see that the speedups achieved by ILUT* are either comparable or better than those achieved by ILUT. In particular, when $t = 10^{-2}$ (first three plots on each figure), the speedups are almost identical. However, for $t = 10^{-4}$ and particularly for $t = 10^{-6}$, ILUT* performs significantly better than ILUT. This is because, as the threshold parameter of the factorization decreases, the number of fill elements in the reduced matrices whose magnitude is greater than the threshold increases significantly. In the case of ILUT, all these elements are kept while in the case of ILUT*, only km elements are kept in each row. Because the reduced matrices have more non-zeros, the parallel ILUT algorithm has to spent considerably more time in factoring these matrices. Furthermore, since the reduced matrices are denser, the number of independent sets that are required to factor them also increases, further adding to the computational requirements. For example, for TORSO on 128 processors, the number of independent sets required for ILUT(20,10^{-4}) and ILUT(20,10^{-6}) are 131 and 439, respectively. On the other hand, the number of respective independent sets required by ILUT* (20,10^{-4},2) and ILUT* (20,10^{-6},2) are only 105 and 184. Not only they are fewer, but also increase at a much lower rate.

The net effect of these denser reduced matrices is that the time required to factor a row corresponding to an interface

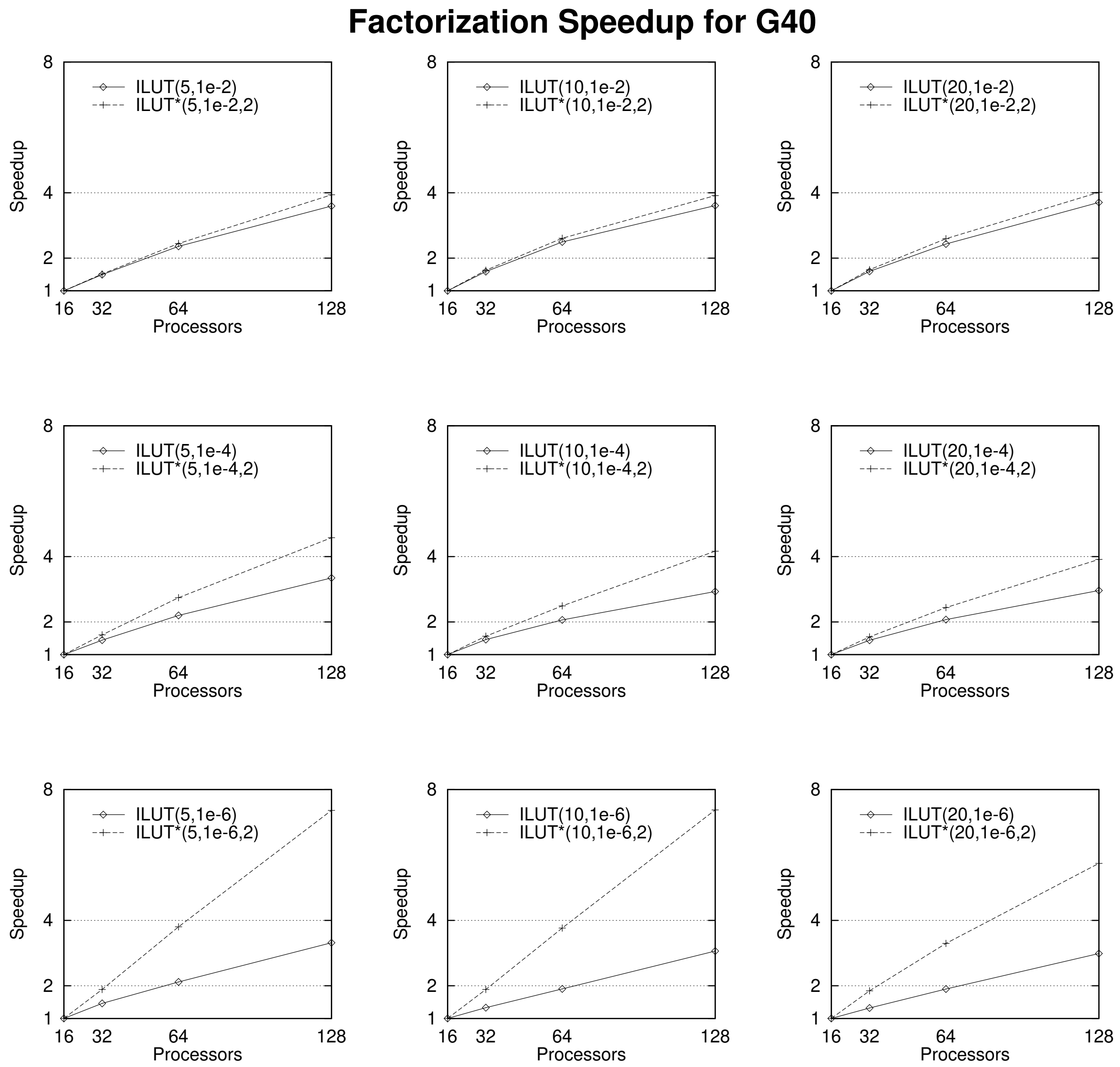


Figure 4: The speedup achieved in computing the ILUT and ILUT* factorizations for G40. The speedup relative to the 16-processor run-time is shown for nine different factorizations for each one of the two algorithms.

node is higher than the time required to factor a row corresponding to an interior node. As the number of processors increases, the number of interior nodes decreases while the number of interface nodes increases, causing the factorization to spent a larger portion factoring 'expensive' rows, which limits the achievable speedup. The ILUT* algorithm though, does not exhibit this problem as long as m is relatively small. However, as m increases, the density of the reduced matrices also increases (each row has up to km elements), leading to similar but less dramatic problems. This can be seen in Figure 5 for $t = 10^{-6}$ (last three plots), in which for $m = 5$ and $m = 10$, the relative speedups are almost linear, but for $m = 20$, the speedup is somewhat lower.

Performance of the Forward and Backward Substitution Table 2 shows the amount of time required to solve the triangular systems produced by the nine different factorizations of TORSO using ILUT and ILUT*. The last row of this table also shows the amount of time required to multiply TORSO by a vector. Note that our matrix-vector multiplication algorithm achieves almost linear speedup indicating that the communication overhead (caused by nodes on the partition boundary) is very small. This demonstrates that our parallel multilevel k-way partitioning algorithm [6] produces high quality partitions in a real application.

	TORSO			
Factorization	$p = 16$	$p = 32$	$p = 64$	$p = 128$
ILUT(5,10^{-2})	0.061	0.033	0.018	0.010
ILUT(10,10^{-2})	0.097	0.049	0.028	0.014
ILUT(20,10^{-2})	0.125	0.066	0.036	0.019
ILUT(5,10^{-4})	0.065	0.035	0.021	0.012
ILUT(10,10^{-4})	0.105	0.059	0.035	0.021
ILUT(20,10^{-4})	0.183	0.102	0.061	0.038
ILUT(5,10^{-6})	0.067	0.038	0.023	0.015
ILUT(10,10^{-6})	0.112	0.066	0.043	0.031
ILUT(20,10^{-6})	0.199	0.119	0.079	0.058
ILUT* (5,10^{-2},2)	0.065	0.035	0.017	0.009
ILUT* (10,10^{-2},2)	0.096	0.051	0.027	0.014
ILUT* (20,10^{-2},2)	0.128	0.066	0.035	0.019
ILUT* (5,10^{-4},2)	0.068	0.034	0.019	0.011
ILUT* (10,10^{-4},2)	0.103	0.057	0.033	0.019
ILUT* (20,10^{-4},2)	0.181	0.100	0.059	0.035
ILUT* (5,10^{-6},2)	0.069	0.035	0.020	0.012
ILUT* (10,10^{-6},2)	0.105	0.059	0.035	0.021
ILUT* (20,10^{-6},2)	0.190	0.106	0.066	0.041
Matrix-Vector	0.064	0.032	0.016	0.009

Table 2: The performance of the parallel forward and backward substitutions algorithms. The run time (in seconds) is shown for 16, 32, 64, and 128 processors, for the forward and backward substitutions for each one of the nine factorizations of ILUT and ILUT*. The last row also shows the amount of time required by the parallel matrix-vector multiplication algorithm.

From Table 2 we see that the amount of time required to solve the triangular systems increases as m increases since the triangular factors contain more elements. Also note that for fixed m, the time required also increases with decreasing t. This is because, when t is large, each row of the triangular factors usually has much fewer than m elements. Comparing the triangular factors produced by ILUT and ILUT* we see that solving the triangular systems produced by the ILUT* factorization usually takes less time, and the difference widens as the number of processors increases. Note that both factorizations produce triangular factors that have roughly the same number of non-zero elements, but ILUT* requires fewer independent sets and hence smaller amount of communication and other overheads..

To study the scalability of our parallel formulation of the forward and backward substitutions we plotted the speedup relative to 16 processors achieved on 32, 64, and 128 processors. Figure 6 shows these relative speedup curves for

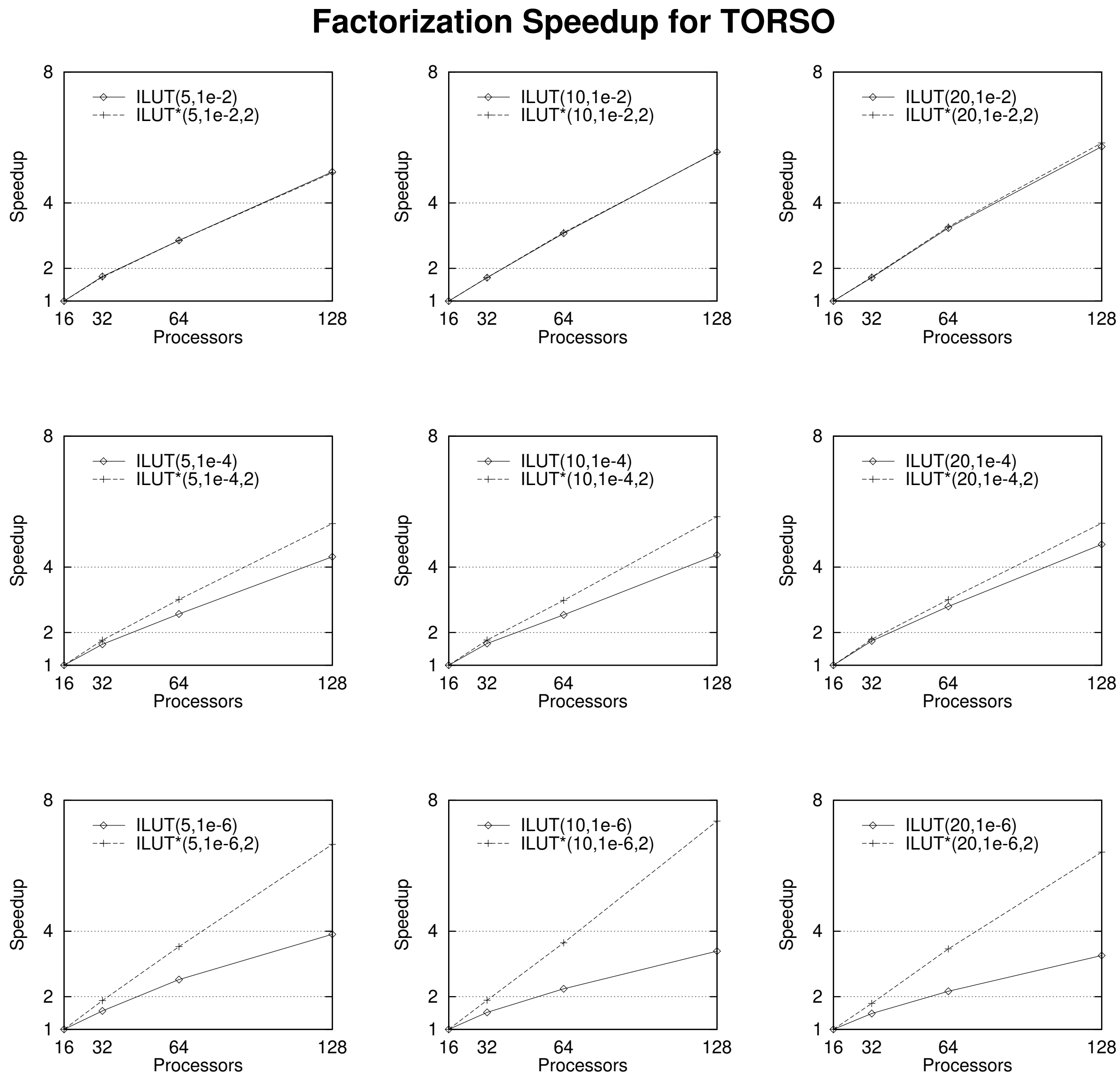


Figure 5: The speedup achieved in computing the ILUT and ILUT* factorizations for TORSO. The speedup relative to the 16-processor run-time is shown for nine different factorizations for each one of the two algorithms.

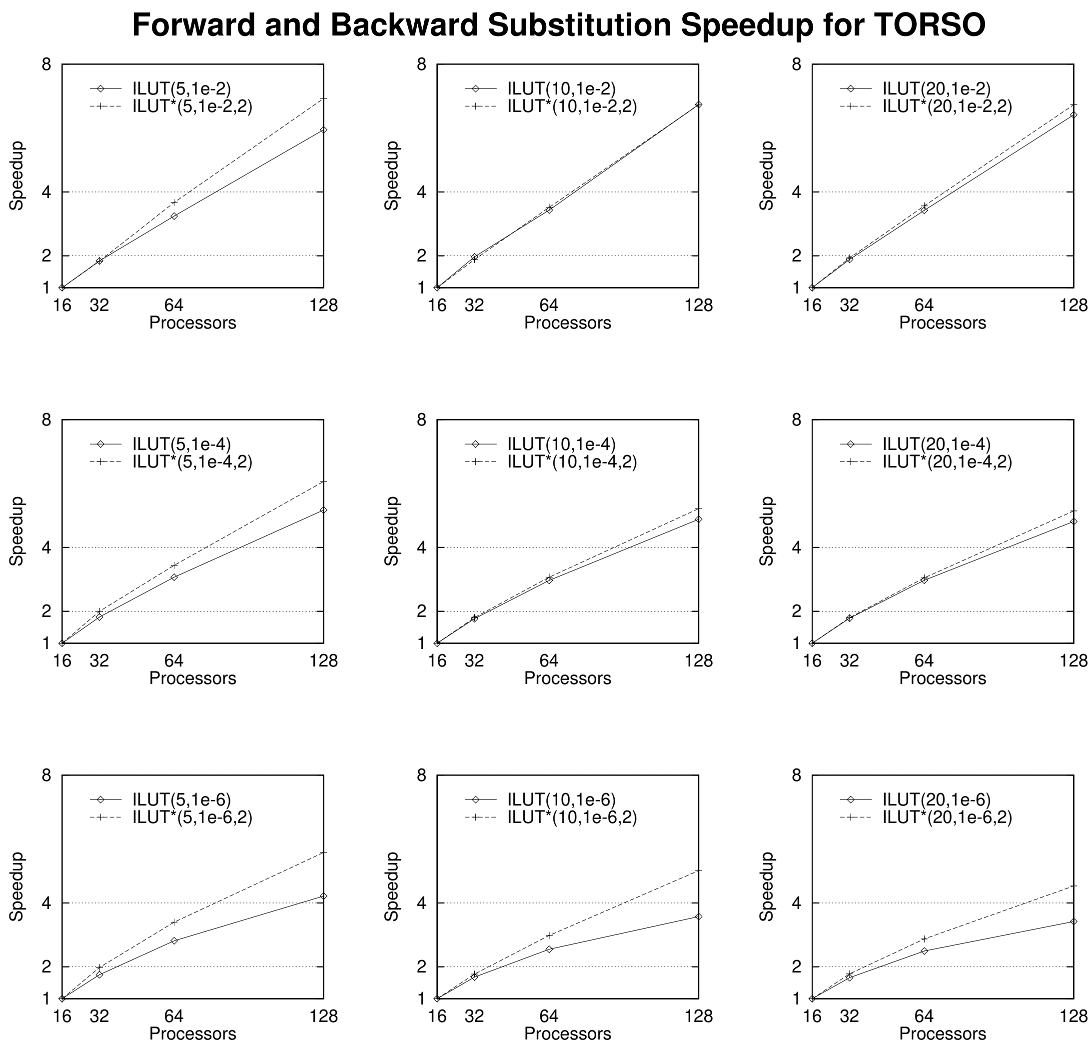


Figure 6: The speedup achieved in performing the forward and backward substitutions for TORSO. The speedup relative to the 16-processor run-time is shown for the nine different substitutions, each corresponding to a different ILUT and ILUT* factorizations.

TORSO. From this figure we see that as m increases and/or t decreases, the speedup that is achieved by our parallel formulation decreases. This is because during the factorization, the number of independent sets required to factor the interface nodes increases; thus, increasing the number of implicit synchronization points required in forward and backward substitutions (as discussed in Section 5). Also, because the number of required independent sets are fewer for ILUT*, the speedups achieved in solving the triangular factors produced by ILUT* are better than those achieved in solving those produced by ILUT.

To compare the performance of solving the triangular systems produced by ILUT and ILUT* to that achieved by the matrix-vector multiplication algorithm, consider the ILUT(20,10^{-6}) and ILUT* (20,10^{-6},2) factorizations. Due to the small threshold value, the average number of non-zeros on each row of the triangular factors is 20. The matrix-vector product achieves 6.17MFlops and 6.0MFlops per processor on 16 and 128 processors, respectively. The corresponding per processor MFlop performance for the ILUT triangular solvers are 5.18 and 2.48, and for the ILUT* are 5.42 and 3.51. That is, on 16 processors the performance achieved by the triangular solvers for ILUT and ILUT* are 1.19 and 1.14 times slower than the matrix-vector product, and on 128 processors they are 2.42 and 1.71 times slower, respectively. Thus, the ILUT* factorization leads to triangular systems whose solution performance compared to the performance achieved by the matrix-vector multiplication, is only slightly lower on 16 processors, and decreases moderately as the number of processors increases.

Comparing the speedup achieved by the parallel forward and backward substitutions to those produced by the factorization algorithms (Figures 5 and 6) we see that in general the factorization algorithms achieve better speedup as m increases and/or t decreases. This is because, as the amount of fill increases, the amount of computation required by the factorization algorithms increases more than linearly. This can compensate for the overhead due to extra independent set computations. In contrast, the amount of computation for the forward and backward substitution increases only linearly. Nevertheless, our parallel algorithms for solving the triangular systems are still able to achieve very good speedups.

Preconditioning Performance To compare the quality of the factorizations produced by the parallel ILUT and ILUT* algorithms we used them as preconditioners in a GMRES iterative solver [13]. Table 3 shows the amount of time and the total number of matrix-vector products required to solve our two test matrices on 128 processors. For both matrices, we construct the right hand sides to be of the form $b = Ae$, where e is a vector of all ones, and the initial solution consisting of all zeros. The iterations were stopped as soon as the residual norm was reduced by a factor of 10^{-8}.

From Table 3 we see that ILUT and ILUT* are quite comparable. In general, ILUT requires fewer iterations than ILUT* for G40, while for TORSO the results are quite mixed. For example, for GMRES(20), for $m = 10$ and $t = 10^{-4}$, ILUT requires 596 matrix-vector products while ILUT* requires only 486, but for $m = 5$ and $t = 10^{-4}$, ILUT converges after 967 products, whereas it takes ILUT* 1276 matrix-vector products to converge. Comparing the actual run-times, we see that for $t = 10^{-6}$, the amount of time required by ILUT* is smaller than that required by ILUT for all cases. This is despite the fact that for G40 ILUT* performs more iterations. In some cases, ILUT* is over 50% faster than ILUT. This is because, the solution of the triangular system of the ILUT* factorization requires less time than that of the ILUT, leading to an overall faster solution time.

The last row of Table 3 also shows the amount of time required by the diagonal preconditioner. In the case of GMRES(20) neither system converged whereas for GMRES(50), G40 converged after 558 matrix-vector products, and TORSO converged after 5,845. Comparing the run-times of the ILUT and ILUT* preconditioners to that of the diagonal, we see that for G40, the ILUT preconditioners are 2 to 5 times faster, and for TORSO, they are 5 to 16 times faster. If the time required to perform the factorization is taken into account (Table 1), then for G40, ILUT* $(10,10^{-2},2)$ (requires a total of 1.059 seconds) is about 3.6 times faster, and for TORSO, ILUT* $(20,10^{-4},2)$ (requires a total of 15.297 seconds) is about 10 times faster compared to the diagonal preconditioner.

	G40				TORSO			
	GMRES(20)		GMRES(50)		GMRES(20)		GMRES(50)	
Preconditioner	Time	NMV	Time	NMV	Time	NMV	Time	NMV
ILUT$(5,10^{-2})$	1.335	194	0.997	95	23.937	938	23.672	676
ILUT$(10,10^{-2})$	1.314	176	0.796	77	18.957	629	16.218	403
ILUT$(20,10^{-2})$	1.073	149	0.810	77	11.574	339	11.273	248
ILUT$(5,10^{-4})$	1.547	175	1.143	92	27.427	967	27.333	700
ILUT$(10,10^{-4})$	1.151	93	0.808	49	22.542	596	15.935	333
ILUT$(20,10^{-4})$	1.032	54	0.780	36	15.118	289	9.432	148
ILUT$(5,10^{-6})$	1.862	159	1.374	91	33.727	1081	29.644	725
ILUT$(10,10^{-6})$	1.911	93	1.219	49	24.675	544	19.338	348
ILUT$(20,10^{-6})$	2.029	54	1.474	36	19.869	276	12.298	149
ILUT* $(5,10^{-2},2)$	1.316	202	0.942	93	24.511	985	24.903	713
ILUT* $(10,10^{-2},2)$	1.110	158	0.869	86	14.526	495	15.603	408
ILUT* $(20,10^{-2},2)$	1.044	156	0.884	86	16.112	473	10.881	246
ILUT* $(5,10^{-4},2)$	1.345	164	1.090	92	34.317	1276	26.255	716
ILUT* $(10,10^{-4},2)$	1.144	103	1.156	82	16.899	486	14.906	335
ILUT* $(20,10^{-4},2)$	0.992	60	0.778	40	14.979	300	9.262	153
ILUT* $(5,10^{-6},2)$	1.371	159	1.113	92	27.143	983	26.216	699
ILUT* $(10,10^{-6},2)$	1.231	99	0.945	61	16.876	472	15.394	337
ILUT* $(20,10^{-6},2)$	1.293	59	0.981	39	13.047	232	9.845	146
Diagonal			3.653	558			149.622	5845

Table 3: The performance of GMRES to solve G40 and TORSO on 128 processors, using parallel ILUT and ILUT* as preconditioner. The column labeled 'Time' is the run time (in seconds) of GMRES (does not include the amount of time required to compute the incomplete factorizations). The column labeled 'NMV' is the number of matrix-vector operations performed. The systems were solved with a tolerance of 10^{-8}.

7 Conclusions

Preconditioners for sparse iterative solvers derived from threshold-based ILU factorizations are widely used on serial as well as vector-supercomputers, but were considered unsuited for execution on highly parallel distributed memory architectures. Our work has shown that the computation of these factorizations as well as the solution of the resulting triangular systems (which are required during the application of the preconditioner) can be performed effectively on distributed memory parallel computers.

In particular, our experiments has shown that our modification to ILUT algorithm (ILUT*), achieves good speedup with an increasing number of processors. the modifications ILUT* are critical for obtaining good performance on parallel computers with slower communication networks (such as workstation clusters), especially when m (*i.e.*, number of non-zeros retained) increases and t (*i.e.*, the threshold of the factorization) decreases. Under these conditions, the reduced matrices produced by ILUT have many non-zeros per row, leading to a large number of independent sets that have a small number of rows; thus, increasing the number of synchronization steps required to factor the interface nodes.

The preconditioning quality of ILUT* (relative to ILUT) depends on the value of k (that determines the number of non-zeros to keep in each row of the reduced matrix). As k increases, factorizations produced by ILUT* become similar to those produced by ILUT. Our experiments has shown that for our test matrices, $k = 2$ lead to factorizations whose preconditioning ability are comparable to those of ILUT. A more comprehensive study is required to characterize the convergence characteristics of ILUT* relative to ILUT for different values of k.

As the desired ILUT and ILUT* factorizations become denser, an alternative parallel formulation can be developed that utilizes graph partitioning to extract concurrency instead of independent sets of rows. Such a scheme will compute a p-way partitioning of the graph corresponding to the interface rows (A^I). Then, the rows that are internal to each domain will be factored concurrently and the second level reduced matrix corresponding to the new interface nodes can be formed. These reduced matrices can now be factored in a similar fashion.

References

[1] Owe Axelsson. *Iterative Solution Methods*. Cambridge University Press, New York, NY, 1994.

[2] E. F. D'Azevedo, F. A. Forsyth, and W. P. Tang. Towards a cost effective ilu preconditioner with high level fill. *BIT*, 1992.

[3] K. A. Gallivan, A. Sameh, and Z. Slatev. A parallel hybrid sparse linear system solver. *Computing Systems in Engineering*, 1:183–195, June 1990.

[4] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. Technical Report 94-63, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. To appear in *IEEE Transactions on Parallel and Distributed Computing*. Available on WWW at URL http://www.cs.umn.edu/˜karypis/papers/sparse-cholesky.ps.

[5] Mark T. Jones and Paul E. Plassmann. Scalable iterative solution of sparse linear systems. *Parallel Computing*, 20:753–773, 1994.

[6] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. Technical Report TR 96-036, Department of Computer Science, University of Minnesota, 1996. Also available on WWW at URL http://www.cs.umn.edu/˜karypis/papers/mlevel_kparallel.ps. A short version appears in Supercomputing 96.

[7] R. N. Klepfer, C.R. Jhonson, and R.S. MaxLeod. The effects of inhomogeneities and anisotropies on electrocardiographic fields: A three-dimensional finite element study. In *IEEE Engineering in Medice and Biology Society, 17th Annual International Conference*, 1995.

[8] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986.

[9] S. Ma and Y. Saad. Distributed ILU(0) and SOR preconditioners for unstructured sparse linear systems. Technical Report 94–027, Army High Performance Computing Research Center, University of Minnesota, Minneapolis, MN, 1994.

[10] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems fo which the coefficient matrix is a symmetric m-matrix. *Math. Comp.*, 31(137):148–162, 1977.

[11] Y. Saad. ILUM: A parallel multielimination ilu preconditioner for general sparse matrices. Technical Report CS-92-241, Department of Computer Science, University of Minnesota, 1992.

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

[13] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7:856–869, 1986.

[14] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing, Boston, MA, 1996.

[15] J. W. Watts-III. A conjugate gradient truncated direct method for the iterative solution of the reservoir pressure equation. *Society of Petroleum Engineer Journal*, 21:345–353, 1981.

[16] 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 Journal on Scientific and Statistical Computing*, 10:1186–1199, 1989.