Efficient Parallel Computation of ILU(k) Preconditioners

David Hysom and Alex Pothen[1]
Department of Computer Science
Old Dominion University
Norfolk, VA 23529-0162
{hysom, pothen}@cs.odu.edu

1 Introduction and Background

Solving large, sparse systems of linear equations of the form $Ax = b$ is a key component in many scientific and engineering numerical computations. Such systems typically arise during the discretization or linearization of systems of partial differential equations (PDEs) in the spatial and time domains. Since the LU factors tend to be dense even when A is sparse, iterative solution methods are increasingly the solution methods of choice.

It is frequently desirable to improve a system's convergence properties by preconditioning. A preconditioner M transforms the linear system into a related system possessing better convergence properties, $M^{-1}Ax = M^{-1}b$. Parallel algorithms have recently been developed for computing approximate inverse preconditioners, but most results show them to be less effective, in terms of iterations required for convergence, than are serial preconditioners based on incomplete factorizations (ILU) of A. However, while ILU preconditioners are popular, effective, and robust, they are, according to the conventional wisdom, primarily serial in nature.

We have developed and implemented course-grained algorithms that perform well for 2D and 3D problems. High performance is achieved through use of local and global orderings of partitioned matrices, coupled with the constraint that subdomain graphs, which capture communication patterns, be identical for A and the LU factors. Our algorithmic framework supports threshold-based (ILUT) and level based (ILU(k)) factorization methods, with optional partial pivoting (ILUTP, ILUP(K)).

ILU preconditioning is based on the computation of triangular factors L and U, where $LU \approx A$. Preconditioner application reduces to the solution of two triangular systems, $Ly = b$ and $Ux = y$, during each iteration. The factors are commonly grouped together as $F = L + U - I$. Several algorithms for computing the triangular factors, L and U, are known and widely used in serial contexts.

ILUT algorithms perform row-by-row, upward-looking factorizations, discarding elements that are smaller than a given value. Perhaps the most popular formulation is ILUT(τ, p) [14], which employs a dual-dropping strategy. The first parameter τ is the dropping threshold, while the second parameter, p, limits the amount of fill in any row

[1]This work was supported by U. S. National Science Foundation grants DMS-9807172 and ECS-9527169; by the U. S. Department of Energy under subcontract B347882 from the Lawrence Livermore Laboratory; by a GAANN fellowship from the Department of Education; and by NASA under Contract NAS1-19480 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE).

of the matrix. If a row in the original matrix A has r nonzero entries, then a maximum of $r + p$ entries are permitted in the corresponding row of F. The limit p is sometimes applied separately to upper and lower triangular portions of the row. As with other ILU algorithms, both symmetric and non-symmetric variants exist.

In structurally based ILU(k) algorithms, factorization is separated into symbolic and numeric phases. The permitted locations of nonzero entries in the factor are determined during the symbolic phase, based on the level, k. All entries in the original matrix are assigned a level of 0 and are permitted in the factor. During factorization, whenever two permitted entries cause a new entry, the entry is assigned a level based on the levels of the causative entries. If this level is less than or equal to k the entry is permitted in the factor. Since a new entry may be caused by different pairs of existing fill elements, the new entry retains the minimum value of all computed levels. ILU(k) techniques were known as early as the 1960s, and were originally developed in the context of solving finite difference equations for elliptic PDEs. The review article [3] provides historical references.

Two different functions have been used for assigning levels to newly created fill entries. Following most present day implementations, we use the *sum* definition, which assigns the new entry the sum of the levels of the two causative entries, incremented by one. For this definition, the length of a fill path corresponds to the number of times an entry is divided by a pivot value during numerical factorization.

The "classical" algorithm for computing ILU(k) structures operates by mimicking upward-looking, row-oriented factorization. This algorithm is commonly interpreted as a unioning of the rows of F, although it also has graph theoretic interpretations.

The upper triangle of any matrix F can be associated with a directed graph, $G(F_U) = (V, E)$, which has has vertex set V and edge set E. A directed edge $\{v, w\} \in E$ if and only if there is a nonzero matrix entry, $f_{v,w}$ in the upper triangle of F. We assume the reader is familiar with the concept of paths in graphs. We call $G(F_U)$ the *directed adjacency graph* of the upper triangle of F. Similar graphs, $G(F)$ and $G(F_L)$, can be constructed for the complete matrix or the matrix's lower triangle.

The classical algorithm, then, can be interpreted as factoring row j by conducting a graph search in $G(F_U)$ starting from all nodes i, where the problem matrix A contains a nonzero entry $a_{i,j}$.

A newer Graph Search Algorithm, which we have described elsewhere [11], computes structures identical to those of the classic algorithm, but operates by conducting breadth-first searches in the underlying graph of $G(A_L^T) \bigcup G(A_U)$. This algorithm arises from a theorem, cited below, which is an extension of the fill path theorem [12], originally developed in the context of complete factorizations. The fill path theorem states that fill edges can only exist between vertices joined by a fill path.

Definition 1 *A* fill path *is a path joining two vertices v and w, all of whose interior vertices are numbered lower than the minimum of the numbers of v and w.*

Theorem 1 *The edge $\{v, w\}$ is a fill edge of level k if and only if there exists a shortest fill path in G(A) joining v and w whose length is $k + 1$.*

D'Azevedo, Forsyth, and Tang defined fill levels in terms of shortest fill path

lengths in [6], although they used the language of reachability sets rather than fill paths. Hence they knew the result in Theorem 1, but did not state or prove the theorem.

Theorem 1 has an intuitively simple geometric interpretation. Construct a sphere of radius $k+1$ about any node i in a graph. Then a fill entry $f_{i,j}$ can only be permitted in an ILU(k) factor if j is within the sphere. This interpretation will be used in the next section to explain our constraint on the processor graph.

Many other ILU algorithms are known; these include combinations of partial pivoting with ILU(k) and ILUT, preservation of row-sums in the factor, drop-tolerance versions of ILU(k), Fast Graph Search techniques [11], etc.

Although ILU(k) algorithms are widely known and implemented [1, 4, 13], virtually all studies we have come across in the literature have been limited to ILU(0) or ILU(1). This is somewhat unfortunate since our own work, as reported in Section 3, has shown high-level ILU(k) preconditioning to be highly effective for many problems. We define a *high-level ILU(k) preconditioner* as one whose structure results from an ILU(k) factorization, with $k \geq 1$. As a rule of thumb, for problems with several hundred thousand unknowns, we are interested in levels as high as ten or twelve.

The structural and numerical relationships between ILU(k) and ILUT factors, particularly for higher-level ILU(k), have not been well studied, although for some simple cases (certain symmetric, diagonally-dominant matrices) the algorithms can be shown to produce identical factors. Published works have tended to report iterations and solution timings, but element-by-element comparisons between the factors themselves have apparently not been undertaken.

The relationships between matrix orderings and ILU preconditioning performance is a complex issue which has, and continues to be, studied by numerous researchers [2, 5, 7, 8, 9]. At least four interrelated effects can be identified in parallel contexts. First, matrix ordering can be used to provide parallelism, i.e, as a partitioning method. Second, from a structural viewpoint, altering a matrix's ordering and ILU(k) or ILUT(τ, p) parameters changes the amount and/or pattern of fill; this affects the amount of work required during setup and application phases. Since altering the fill count and/or pattern also changes a preconditioner's numerical properties, the third and fourth effects are alterations in convergence properties and numerical stability.

Since our algorithmic approach intrinsically changes a problem's ordering, at various places in the following sections we comment on how this effects parallelism, total work (solution time), and convergence properties. We do not address the stability issue in this work.

2 Parallel Algorithmic Framework

Our approach is based on three constraints, two of which are merely reflective of real-world problems and hardware. First, we stipulate that the number of processors be small in relation to the problem size: $p \ll N$. This is typically the case for existing and planned hardware platforms (e.g, ASCI Teraflop machines) where we may have thousands of processors but need to solve equations with millions of unknowns.

Second, we assume the matrix A is well-partitionable. This is analogous to physical domain decomposition such that subdomains have large volume to surface ratios.

To generalize this notion, we call node i (and its associated matrix row) *interior* if, for every nonzero entry a_{ij} or a_{ji}, nodes i and j belong to the same subdomain. Conversely, i is a *boundary* node if there is at least one edge a_{ij} or a_{ji} such that nodes i and j belong to different subdomains. In general, then, a matrix is well-partitioned if all subdomains have large interior to boundary node ratios. Spatially discretized partial differential equations are frequently well-partitionable.

Prior to stating our final constraint we define a *subdomain graph* as a graph whose nodes correspond to subgraphs (subdomains) and whose edgeset contains edge (r, s) if subgraphs r and s are joined by one or more edges.

Our third constraint, then, is that the subdomain graph of the factor F and the matrix A be identical. This constraint permits the determination of all communication patterns prior to actual factorization phases, as discussed below.

Within each subdomain we require that interior nodes be ordered before boundary nodes. By the fill path theorem [12], this ensures that, if an edge $f_{i,j}$ arises during factorization and crosses subdomain boundaries, then nodes i and j must have been boundary nodes in the graph of A. In other words, ordering interior nodes first in the graph of A ensures that interior nodes can not be converted to boundary nodes in the graph of F. This permits the independent factorization of interior nodes within each subdomain. It also preserves parallelism during preconditioner application, since subdomain interior nodes can be evaluated independently during the triangular solves.

The independence of subdomain interiors also allows selection of ILU algorithms and parameters (e.g, level, for ILU(k)) on a subdomain by subdomain basis. Hence, our framework is ideally suited for adaptive preconditioning in multi-physics problems, wherein different models are used for different physical subdomains.

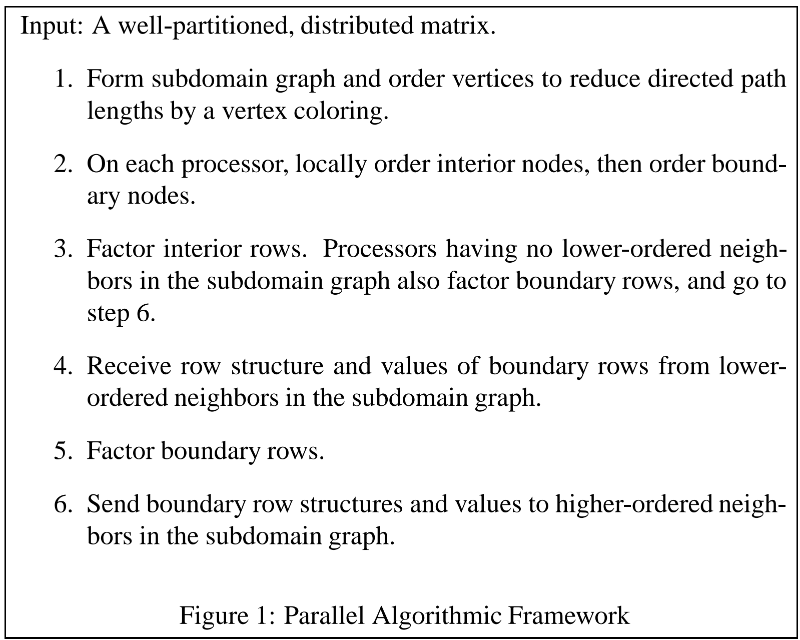
For the remainder of this paper we assume a one-to-one and onto mapping between subdomains and processors. Also, due to the intimate connection between matrices and graphs in the context of matrix factorization, we use the phrases *eliminating a node* and *factoring a row* synonymously.

Figure 1 contains a high-level overview of our algorithm. Comments on particulars follow.

The first step, which induces a global matrix reordering, generally requires global communication. However, due to the constraint $p \ll N$, this computation is not time-constraining. If special topological information is available, e.g., a regular 2D or 3D grid has been used for discrectization, this step can be accomplished without communication.

Steps four through six can potentially give rise to sequential bottlenecks. If processors r and s are neighbors in the subdomain graph and r is ordered before s, then processor s cannot complete factorization of its boundary rows until it receives factored boundary row structural and numerical values from r. Regular 2D and 3D grids can easily be ordered in red black fashion so that dependency paths for the subdomain graph of A are at most of length one. Additional dependencies, however, can arise during factorization, as illustrated in Figure 2.

It is not always possible to compute in advance where these dependencies will arise. Further, if all dependencies are permitted, numerous synchronization points may be required in order to determine which processors must send what to whom. Our third constraint, that subdomain graphs of A and F be identical, dispenses with the need for



Input: A well-partitioned, distributed matrix.

1. Form subdomain graph and order vertices to reduce directed path lengths by a vertex coloring.
2. On each processor, locally order interior nodes, then order boundary nodes.
3. Factor interior rows. Processors having no lower-ordered neighbors in the subdomain graph also factor boundary rows, and go to step 6.
4. Receive row structure and values of boundary rows from lower-ordered neighbors in the subdomain graph.
5. Factor boundary rows.
6. Send boundary row structures and values to higher-ordered neighbors in the subdomain graph.

Figure 1: Parallel Algorithmic Framework

this sort of synchronization.

The constraint, however, may alter the factor's structural and numerical properties by preventing the inclusion of some fill edges. Figure 3 is intended to provide an intuitive characterization of prohibited fill edges. For a typical 2D grid, we see that prohibited edges can arise near points where subdomains touch corners but are not neighbors in the subdomain graph. Theorem 1 tells us that these edges can only arise within a radius of $k + 1$ edges about such points.

If desirable, the subdomain graph constraint can be relaxed, as long as the structure of the subdomain graph of F can be determined before factorization begins. Acceptable subdomain graphs for F, for example, can be computed by performing ILU(k) factorizations on the subdomain graph of A.

Several other variants of our algorithm are possible. For example, in earlier work on serial preconditioning we developed algorithms that compute symbolic factors based solely on the nonzero structure of A [11]. Incorporating these algorithms into our framework permits complete separation of symbolic and numeric stages for ILU(k) factorizations. This may be especially advantageous since we have found–with the exception of ILU(0)–that the cost for symbolic factorization tends to be equal, or slightly greater than, that for numeric factorization. These algorithms are also particularly suitable for implementation in shared memory environments.

We have recently become aware of work by Karypis and Kumar [10], which is similar to ours in that they employ partitioning, followed by elimination of interior nodes, followed by elimination of boundary (in their terminology, *interface*) nodes. Their experimental results are encouraging, since they support our thesis that this general

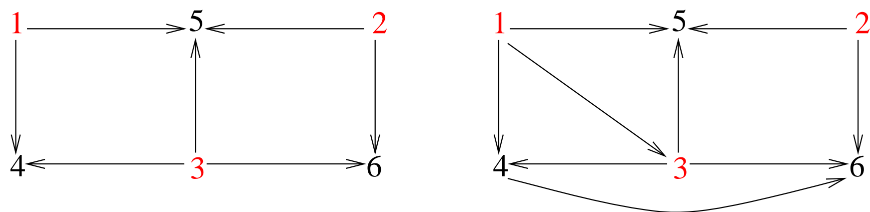


Figure 2: Left: ordered processor graph of A; all dependency paths are of length one. Right: some of the additional dependencies that can arise during factorization; there is now a path length of three, 1, 3, 4, 6, which necessitates three non-overlapping communication phases for transmittal of boundary row structure and values.

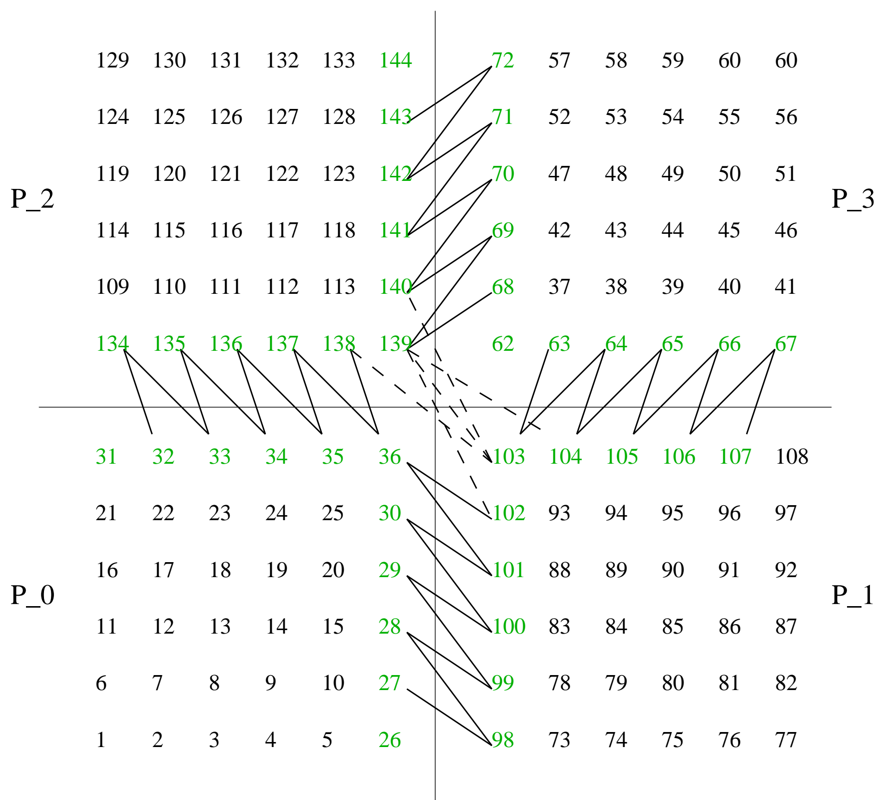


Figure 3: 2D grid after global and local reordering phase. Level 1 and level 2 fill edges which cross subdomain boundaries are shown. The four dotted lines between processors P_2 and P_4's subdomains indicate fill edges that are prohibited due to the subdomain graph constraint. All level 0, and level 1 and 2 edges not crossing subdomain boundaries, have been omitted for clarity.

approach is effective, highly parallel, and scalable. We note the following differences between their work and ours.

Karypis and Kumar's work centers on ILUT, while we are developing object-oriented code which supports tailored selection of numerous ILU variants on a subdomain-by-subdomain basis. Additionally, our use of the fill path Theorem [12], along with Theorem 1, its counterpart for ILU(k), enables us to present (see Section 4) theoretical performance analyses. Our subdomain graph constraint enables an easy computation of communication patterns; in contrast, Karypis and Kumar compute subdomain dependencies in an ad hoc manner, as factorization progresses.

3 Results

Results in this section are based on the following model problems.

Problem 1. Laplace's equation in two or three dimensions

$$\triangle u = 0.$$

Problem 2. Convection-diffusion equation with convection in the xy plane

$$-\varepsilon\triangle u + \frac{\partial}{\partial x}e^{xy}u + \frac{\partial}{\partial y}e^{-xy}u = g.$$

Homogeneous boundary conditions were used for both problems. Derivative terms were discretized on the unit square or cube, using 3-point central differencing on regularly spaced $n_x \times n_y \times n_z$ grids ($n_z = 1$ for 2D). The zero vector was used for the right-hand side of the resulting systems, $Ax = b$, and random vectors with values in the range $[-1, 1]$ were used for initial guesses. The coefficient for Problem 2 was $\varepsilon = 1/500$, which yields a moderately unsymmetric system of equations.

Although much present day scientific modeling results in systems of Partial Differential Equations (PDEs) far more complicated than our model problems, most of our parallel code's behavior can be understood by examining these simple problems in conjunction with ILU(k) preconditioning. ILU(k) preconditioning is amenable to performance analysis since the structures of ILU(k) preconditioners are identical for *any* PDE that has been discretized on a 2D or 3D grid with a given stencil. The structure depends on the grid and the stencil only, and is not affected by numerical values. Identical structures imply identical symbolic factorization costs, as well as identical flop counts during the numerical factorization and solve phases. In parallel contexts, communication patterns and costs are also identical. While preconditioner effectiveness–the number of iterations until the stopping criteria is reached–differs with the numerics of the particular problem being modeled, the parallelism available in the preconditioner does not.

The structure of ILUT preconditioners, on the other hand, is a function of the grid, the stencil, and the numerics. Changing the problem, particularly for non-diagonally dominant cases, can alter the preconditioner structure, even when the grid and stencil remain the same. Moreover, even for diagonally dominant problems, the structure of ILUT(τ, p) preconditioners is a function of two parameters, τ and p, which makes for

a much more complicated experimental space than when structure is controlled by the single parameter, k.

These are the primary reasons why, even though our algorithmic framework is suitable with use of ILUT(τ, p) and other ILU variants, we find performance evaluation more meaningful when carried out in the context of ILU(k). In addition, most ILU(k) implementations separate symbolic from numeric factorization, thus permitting an easy determination of the time spent in each stage. We typically see that the symbolic phase–during which no floating point operations are issued–takes as long or longer than the numeric phase. These relationships are very difficult to determine for ILUT, since the symbolic and numeric phases must be interleaved on a row-by-row basis.

In addition to demonstrating that our algorithm can provide high degrees of parallelism, several other issues must be addressed. Our results show that available parallelism increases with level, so we need to examine the effectiveness of high-level preconditioners for reducing total solution time. Since memory requirements are directly related to fill count, we should consider whether the storage costs imposed by high-level preconditioners are acceptable. As there is not much point in parallelizing operations which only account for very small proportions of execution time, we need to examine how much run time is typically consumed by preconditioner computation. Finally, since matrix orderings can significantly effect execution time for both ILU(k) and ILUT, we should look at how the matrix orderings required by our approach affect convergence behavior.

3.1 Preconditioner Effectiveness

In this subsection we consider the effectiveness of high-level preconditioners; look at the relationships between factorization time, total execution time, and level; examine storage costs; and consider how the orderings imposed by our algorithms effect convergence. These topics are fairly easy to address since they can be adequately examined in sequential environments. The results are important, however, because they lay the groundwork for interpreting our code's performance in parallel environments, presented in the next subsection.

Experiments in this subsection were conducted on a Sun Ultra-30, with 1024 Megabytes of main memory and a CPU clock rate of 300 MHz. Problem 1 was solved using the Conjugate Gradient method and left preconditioning. Problem 2 was solved using GMRES with restart=30, Unmodified Gram-Schmidt, and 1 step iterative refinement orthogonalization. The solvers were based on optimized code from the PETSc library, with our own routines used to measure CPU time. Stopping criteria was $\| r_k \|_2 \leq \mathbf{rtol}$ with **rtol**, the relative decrease in the residual norm, ranging from 10^{-5} to 10^{-7}.

Since Problem 1 is symmetric, it could be preconditioned using Incomplete Cholesky Factorization (IC(k)) rather than ILU(k). However, if the problem were augmented by the addition of a first derivative term the resulting system would unsymmetric and we would be forced to use ILU(k), even if the first-order term had near-negligible coefficients. We therefore believe that reporting on ILU(k) rather than IC(k) makes the results more general.

Table 1: Convergence behavior for Laplace's equation (Problem 1) on 2D, 360×360 grid.

level	nzF/nzA	$rtol = 10^{-5}$		$rtol = 10^{-7}$	
		solution time	setup ratio (%)	solution time	setup ratio (%)
0	1.0	10.1	5	33.2	1.6
1	1.4	7.8	12	24.6	3.6
2	1.8	7.1	15	21.6	4.9
3	2.6	6.4	25	19.1	8.2
4	3.4	6.3	32	17.6	11.6
5	4.2	6.9	39	16.8	16.4
6	4.9	6.5	50	16.4	20.0
7	5.7	7.4	54	15.7	25.0
8	6.5	8.0	62	16.4	30.0
9	7.3	8.8	65	16.6	34.8

Total solution times are reported in preference to iteration counts since work-per-iteration is largely dependent on the total amount of fill permitted in the preconditioner. Also, evaluating preconditioner effectiveness by iteration count comparison ignores preconditioner setup (factorization) time, which can constitute a considerable proportion of total execution time.

We begin by reviewing some statistics concerning ILU(k) behavior in general. Table 1 summarizes ILU(k) behavior for Problem 1. The table shows total solution time, the percentage of total solution time devoted to preconditioner factorization, and the ratio of nonzeros in the preconditioner to that in the problem (nzF/nzA).

There are three points to be noted in this data. First, the most time-efficient solutions require relatively high fill levels. Second, as greater resolution is required (smaller rtol), the optimal preconditioner level increases. Changing the stopping criteria by a factor of 100, from $rtol = 10^{-5}$ to $rtol = 10^{-7}$, results in a change from ILU(4) to ILU(7), for fastest solution. Finally, for the most time-efficient solutions, between 1/3 and 1/4 of execution time was spent in the preconditioner setup phase (symbolic and numeric factorization). Application of Amdahl's Law tells us that, if we can not effectively parallelize ILU(k) factorization, we can never obtain more than a three-fold speedup.

Table 2 contains similar statistics for Problem 2. The same trends noted for Problem 1 are evident here, although the fastest solutions required an even greater fill level.

It is illuminating to note how the amounts of fill in ILU(k) preconditioners compare with the amounts commonly permitted in ILUT preconditioners. In [2], Benzi, et. al., compared preconditioner performance between ILU(0), ILU(1), ILUT(.005,5), and ILUT(.001,10) [2], for a variety of problems and matrix orderings. We noticed that, for most of the cases reported, the amount of fill permitted in the ILUT precondition-

[2]more accurately, they used SILUT, a modification of the ILUT algorithm which gives rise to symmetric preconditioners when the original matrix is symmetric.

Table 2: Convergence behavior for Convection-Diffusion equation (Problem 2) on 2D, 360×360 grid.

level	nzF/nzA	$rtol = 10^{-5}$ solution time	$rtol = 10^{-5}$ setup ratio (%)	$rtol = 10^{-7}$ solution time	$rtol = 10^{-7}$ setup ratio (%)
0	1.0	39.2	1.3	165.8	0.3
1	1.4	34.3	2.2	108.2	0.7
2	1.8	33.5	2.7	99.5	0.9
3	2.6	31.4	4.1	82.1	1.5
4	3.4	30.2	5.8	77.6	2.2
5	4.2	28.0	8.7	69.4	3.5
6	4.9	26.4	11.2	56.9	5.2
7	5.7	25.4	13.8	54.3	6.6
8	6.5	23.5	18.8	54.0	8.0
9	7.3	23.6	21.8	48.4	10.5
10	8.1	25.8	25.6	50.0	13.2

Table 3: Comparison of ILU(k) and ILUT fill counts, 2D domain.

grid size	ILUT(.01,5),	ILUT(.001,10)	ILU(1)	ILU(2)	ILU(3)	ILU(4)	ILU(5)
128×128	31	154	16	31	62	93	124
256×256	126	628	64	128	256	383	509
300×300	174	865	88	177	353	528	702
400×400	310	1544	158	316	630	944	1257

ers exceeded that permitted in the ILU(k) factors. This motivated us to wonder what ILU(k) levels would be required to produce amounts of fill comparable to those in the ILUT factors. Tables 3 and 4 show this comparison. The figures for the ILUT levels are from [2]. The figures for ILU(k) were computed using the PETSc library. In these tables we use Benzi's method of reporting fill counts, which is the amount of fill greater than level zero, in either the upper or lower triangle of the factor.

Even though the entries are computed differently, every entry in an ILUT factor can be assigned the level it would have, if it were permitted in an ILU(k) factor. Since The fill counts in the 2D ILUT(.001,10) factors in Table 3 exceed the fill counts for the ILU(5) factors, it follows that the ILUT factors must include at least some level six, or larger, entries.

High-level preconditioners are effective for many problem types. The Driven Cavity set from the SPARSKIT Collection consists of a series of non-symmetric, indefinite matrices which arise from modeling of the incompressible Navier-Stokes equations. They are reportedly difficult to solve iteratively due to the high condition numbers of the coefficient matrices. All the problem can be solved, however, using high-level pre-

Table 4: Comparison of ILU(k) and ILUT fill counts, 3D domain.

grid size	ILUT(.01,5),	ILUT(.001,10)	ILU(1)	ILU(2)	ILU(3)
$16 \times 16 \times 16$	14	38	9	22	47
$32 \times 32 \times 32$	119	313	83	216	475
$40 \times 40 \times 40$	236	615	167	439	971
$50 \times 50 \times 50$	466	1206	336	885	1965

Table 5: Iterative performance for Driven Cavity problem e40r3000. ILU(1) converged very slowly, and was terminated before achieving exit criteria.

level	solution time (s)	PC formation time (%)	residual norm	iterations
1	95.7	8	8.0e-3	NA
2	91.8	14	3.0e-5	299
3	49.2	49	1.8e-5	85
4	58.2	58	3.6e-5	56
5	64.5	64	4.7e-5	29
6	78.7	85	1.5e-5	27

conditioners. Table 5 shows a typical problem, for which the fastest solution time was obtained with ILU(3) preconditioning. This problem can also be solved using high-level ILUT preconditioning, although we found that solution times were slower, with larger amounts of fill required to force convergence.

The use of high-level preconditioning is counter-intuitive in the sense that a primary reason for employing iterative solvers is to reduce storage requirements. There are, however, large gaps between the amounts of fill we have been talking about and that required for direct factorization. Figure 4 shows storage requirements as a function of level, for $k = 0, \ldots, K$, where ILU(K) results in complete factorization. The conclusion is that memory requirements for high-level preconditioning, given the levels we have been discussing, do not begin to approach that needed for direct solution methods.

While the curve in Figure 4 shows fill growth typical of what we have observed for all medium to large matrices studied, a word of caution is in order. Smaller matrices, such as most in the Harwell Boeing collection, tend to have relatively small maximum levels. For such problems, employing ILU(k) for $k > 1$ may result in factors whose sizes approach those for complete factorization.

We now turn to the question of how our algorithm's matrix reorderings affect convergence. Although our algorithm admits a high degree of parallelism (as demonstrated in the following subsection), the parallelism will not be of much utility if an imposed ordering results in a significant degradation of convergence behavior.

There are several ways we might investigate this question. For example, we could completely solve linear systems in parallel environments, using our preconditioner

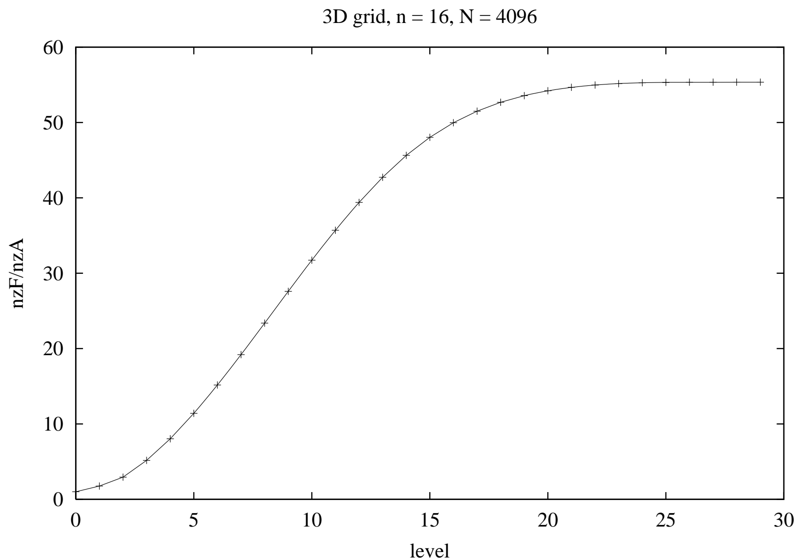


Figure 4: Storage requirements vs. fill level for 16×16 grid

code in conjunction with some iterative software library such as PETSc. This is, of course, the ultimate goal. However, measurements from such a procedure do not allow us to isolate the effects of matrix orderings from other influences. Instead, the measurements would reflect a combination of influences from our algorithm, its implementation, the PETSc library, the MPI specification as implemented on some selected platform/communication network, contention with other users for system resources, and so on.

We therefore elected to examine ordering effects in a more tightly controlled environment. Problems were generated, and local and global orderings carried out, using our parallel code. Each subdomain's interior nodes were locally numbered using natural ordering. At this point the matrices were written to file, after which convergence behavior was studied on a uni-processor platform.

For each series of runs we generated problems for 2D grids ranging in size from 128×128 to 600×600 grid-points, and optimally partitioned the grids amongst 1, 4, 9, 16, and 25 processors. Our findings are summarized in Figures 5 and 6. The figures show that the partitioning and subsequent ordering does affect convergence behavior, but that the effect is relatively minor. Further, the effect decreases as the problem size increases, relative to the number of processors. The fastest solution time on the 240×240 grid was 18% slower for 25 processors than for a single processor, while there was only a 3.8% difference on the 600×600 grid.

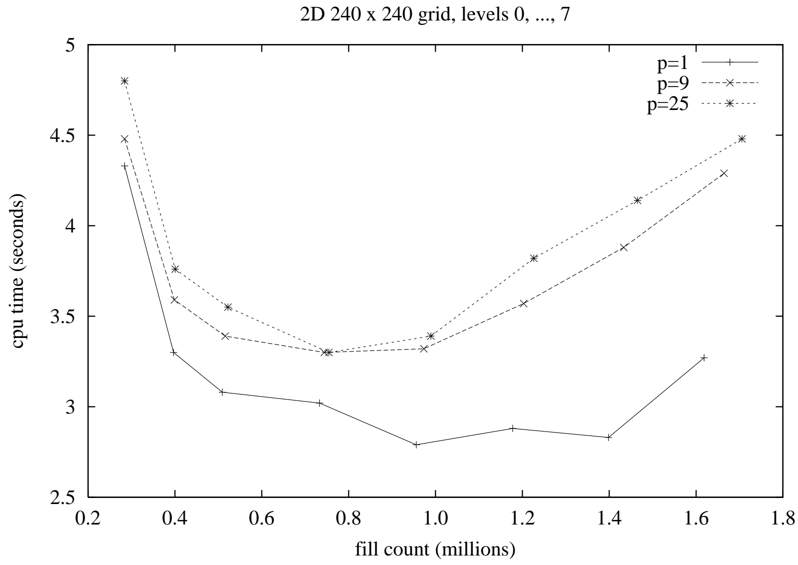


Figure 5: Effect of matrix partitioning and ordering, 2D domain

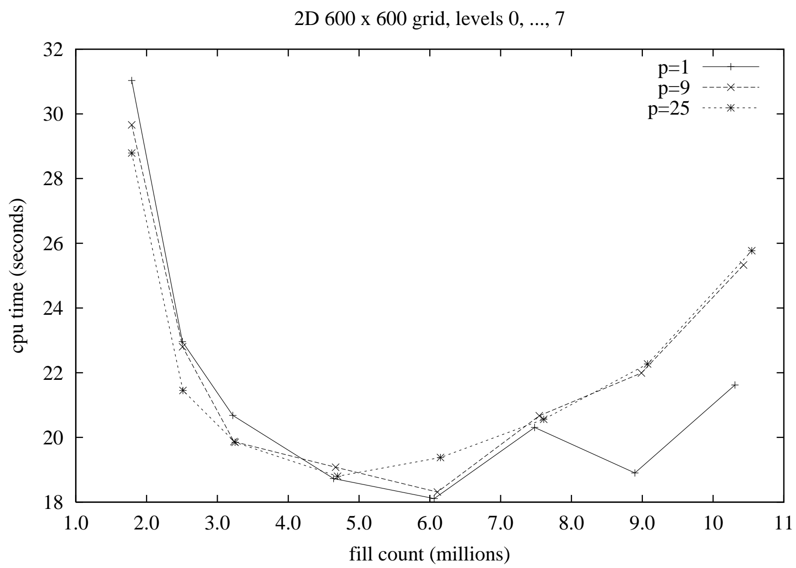


Figure 6: Effect of matrix partitioning and ordering, 2D domain

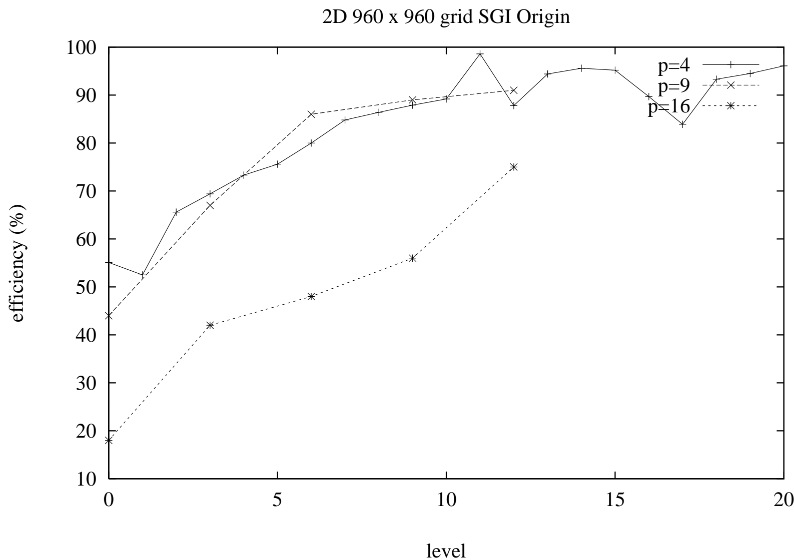


Figure 7: Symbolic factorization performance on 4, 9, and 16 processors, SGI Origin

3.2 Parallel Performance

The present version of our code uses MPI library calls for communication, though we are investigating shared-memory implementations.

Figure 7 shows symbolic factorization performance for ILU(k) symbolic factorization on an SGI Origin2000. We have experienced considerable contention with other users on this machine, as evidenced in the noise spikes in the four-processor line. Figure 8 shows similar results on the *Coral* Beowulf cluster at ICASE

Careful readers may note that Figure 7 plots level on the x-axis, while Figure 8 plots nzF/nzA. This difference is more reflective of the author's fickleness than of any substantive distinction. When a given problem is partitioned amongst differing numbers of processors while fill level is set at some value greater than one, fill counts in the computed preconditioners will vary. The variance is slight, however, given the conditions assumed throughout this paper. Our opinion as to how this data ought best be represented continues to oscillate between the options of plotting efficiency vs.time, or efficiency vs. fill level.

We have tested debugging versions of our code using two different grid partitioning strategies. With respect to the underlying grid, **block partitioning** divides the grid into square (cubic) sub-grids, while **striped partitioning** divides the grid into horizontal slices. Blocked partitioning is optimal in the sense that the ratio of interior/boundary nodes remains constant when problem size is scaled linearly with the number of processors. Striped partitioning would arise in an application wherein grid points were naively numbered globally, in natural order. Table 6 compares efficiency for symbolic factorization for blocked vs. striped partitioning. Efficiency for the striped cases is in-

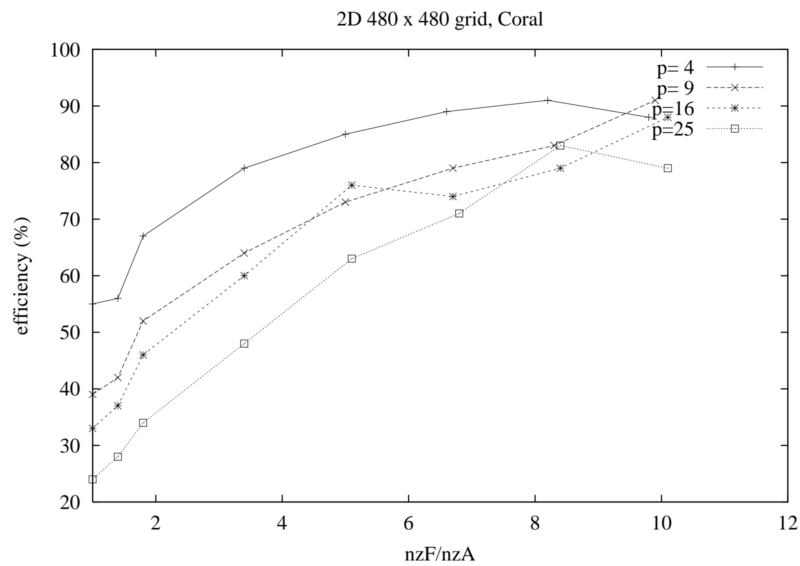


Figure 8: Symbolic factorization performance on 4, 9, 16, and 25 processors, on ICASE Coral Beowulf cluster. The cluster, which operates under RedHat Linux, consists of thirty-two 400MHz Pentium-II processors with fast ethernet switching and 12GB total RAM.

Table 6: Performance comparison between blocked and striped partitioning for 480×480 grid; data is for symbolic factorization on an SGI Origin2000, for 4 and 16 processors.

	Efficiencies (%)			
	p = 4		p = 16	
level	blocked	striped	blocked	striped
0	51	57	22	21
1	58	56	26	23
2	69	65	33	31
4	79	76	45	43
6	83	78	62	54
8	88	77	57	53
10	93	82	68	66
12	92	76	78	70

ferior to that for the blocked cases, which is expected from the scaling considerations. However, these results indicate that our methods can produce favorable results even for matrices whose underlying grids are poorly partitioned.

4 Performance Analysis

In this section we present simplified theoretical analyses of algorithmic behavior for matrices arising from PDEs discretized on 2D grids with five-point stencils and 3D grids with seven-point stencils. Since our arguments are structural in nature, we assume ILU(k) is the factorization method of choice. After a word about nomenclature, we begin with the 2D case.

The word *grid* refers to the grid (mesh) of unknowns; for Regular 2D and 3D grids with five and seven point stencils, respectively, this is identical to the undirected graph, $G(A)$. We remind the reader that we use the terms *eliminating a node* and *factoring a row* synonymously.

We assume the grid has been block-partitioned, with each subdomain consisting of a square subgrid of dimension $c \times c$. We also assume the subdomain grid has dimensions $\sqrt{p} \times \sqrt{p}$, so there are p total processors. There are thus $N = c^2 p$ nodes in the grid, and subdomains have at most $4c = 4\sqrt{\frac{N}{p}}$ boundary nodes.

If subdomain interior nodes are locally numbered in natural order and $k \ll c$, matrix rows in the factor F can asymptotically be considered as having $2k$ (strict) upper triangular and $2k$ (strict) lower triangular nonzero entries. The justification for this statement arises from consideration of Theorem 1; the geometric intuition is illustrated in Figure 9.

Assuming the classic ILU(k) algorithm is used for symbolic factorization, both symbolic and numeric factorization of row j entails $4k^2$ work. This is because, for each lower triangular entry $f_{j,i}$ in matrix row j, factorization requires "touching" each

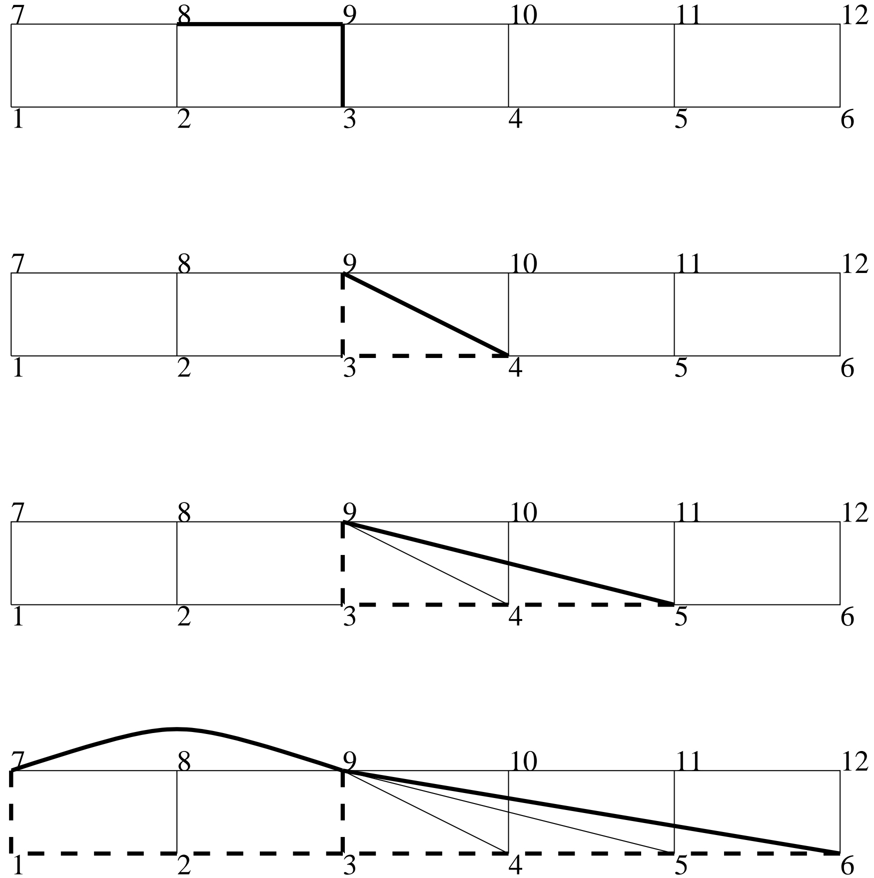


Figure 9: Counting lower triangular fill edges in a naturally ordered grid. From top to bottom, there are two level 0 edges; there is one level 1 edge, due to fill path 9, 3, 4; there is one level 2 edge due to fill path 9, 3, 4, 5; there are two level 3 edges, due to fill paths 9, 3, 4, 5, 6 and 9, 3, 2, 1, 7. Two additional fill edges are created for every level greater than three. Since there is a total of six fill edges for a level 3 factorization, we conclude that asymptotically there are $2k$ lower triangular edges in a level k factorization. By symmetry, the upper triangle must contain the same number of entries.

upper triangular entry in row i.

A red-black ordering of the subdomain grid gives an optimal bipartite division. If red subdomains are numbered before black subdomains, our algorithm simplifies to the following three stages.

1. Red processors eliminate all nodes; black processors eliminate interior nodes.
2. Red processors send boundary-row structure and values to black processors.
3. Black processors eliminate boundary nodes.

If these stages are non-overlapping, the cost of the first stage reduces to the cost of eliminating all nodes in a subdomain. This cost is $4k^2c^2 = \frac{4k^2N}{p}$.

The cost for the second stage is the cost of sending structural and numerical values from the upper-triangular portions of the boundary rows to neighboring processors. If $k \ll c$, Theorem 1 can be used to show that, asymptotically, a processor only needs to forward values from c rows to each neighbor. Assuming a standard, non-contentious communication model wherein α and β represent message startup and cost-per-word respectively, and with time for each operation normalized to unity, the cost for this step is $4(\alpha + 2k\beta c) = 4(\alpha + 2k\beta\sqrt{\frac{N}{p}})$.

Since the cost of factoring a boundary row can be shown to be asymptotically identical to that for factoring an interior row, the cost for eliminating the $4c$ boundary nodes is $(4k^2)(4c) = 16k^2\sqrt{\frac{N}{p}}$.

Speedup can then be expressed as

$$\text{speedup} = \frac{4k^2N}{\frac{4k^2N}{p} + 4(\alpha + 2k\beta\sqrt{\frac{N}{p}}) + 16k^2\sqrt{\frac{N}{p}}},$$

where the numerator represents cost for uni-processor (sequential) execution, and the three denominator terms represent the costs for the corresponding stages of the simplified algorithm for parallel execution.

Two interpretations of this equation are in order. First, for a fixed problem size and number of processors, the parallel computational cost (the first and third terms in the denominator) is proportional to k^2, while the communication cost (the second term in the denominator) is proportional to k. This explains the experimentally observed increase in efficiency with level. Second, if the ratio N/p is large enough, the first term in the denominator will dominate the second and third terms, and efficiency will approach 100%.

For the 3D case we assume partitioning into cubic subgrids of dimension $c \times c \times c$ and a subdomain grid of dimension $p^{1/3} \times p^{1/3} \times p^{1/3}$, which gives $N = c^3p$. Subdomains have at most $6c^2$ boundary nodes. A development similar to that above shows that, asymptotically, matrix rows in the factor F have k^2 (strict) upper and lower triangular entries, so the cost for factoring a row is k^4. Speedup for this case can then be expressed as

$$\text{speedup} = \frac{k^4N}{\frac{k^4N}{p} + 6(\alpha + k^2\beta(\frac{N}{p})^{1/3}) + 6k^4(\frac{N}{p})^{1/3}}.$$

References

[1] S. Balay, W. D. Gropp, L. Curfman McInnes, and B. F. Smith. PETSc home page. http://www.mcs.anl.gov/petsc, 1999.

[2] M. Benzi, W. Joubert, and G Mateescu. Numerical experiments with parallel orderings for ILU preconditioners. *Elec. Trans. on Numer. Anal.*, 8:88–114, 1999.

[3] T. C. Chan and H. A. Van der Vorst. *Approximate and Incomplete Factorizations*, in Parallel Numerical Algorithms. D. E. Keyes, A. Sameh, and V. Venkatakrishnan (eds.), pp. 167–202, Kluwer Academic, Dordrecht, 1997.

[4] E. Chow and M. A. Heroux. Block preconditioning toolkit. http://www-users.cs.umn.edu/ chow/bpkit.html, 1997.

[5] E. Chow and Y. Saad. Experimental study of ILU preconditioners of indefinite matrices. *J. Comput. Appl. Math*, 86:387–414, 1997.

[6] E. F. D'Azevedo, P. A. Forsyth, and W.-P. Tang. Towards a cost-effective ILU preconditioner with high level fill. *BIT*, 32:442–463, 1992.

[7] I. S. Duff and G. A. Meurant. The effect of ordering on preconditioned conjugate gradients. *BIT*, 29, 1983.

[8] S. Dui and A. Lichnewsky. A graph-theory approach for analyzing the effects of ordering on ILU preconditioning. Technical Report 1452, Institut National de Recherche in Informatique et en Automatique, Rocqhencourt, BP105-78153, Le Chesnay Cedex, France, 1991.

[9] V. Eijkhout. Analysis of parallel incomplete point factorizations. *Lin. Algebra. Applic.*, 154–156:723–740, 1991.

[10] G. Karypis and V. Kumar. Parallel threshold-based ILU factorization. Technical Report 061, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 5455, 1998.

[11] A. Pothen and D Hysom. Fast algorithms for incomplete factorization. in prepatation.

[12] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM J. Appl. Math.*, 23(1):176–197, 1978.

[13] Y. Saad. Sparskit, version 2. http://www-users.cs.umn.edu/ saad/software.html, 1990, 1994.

[14] Y. Saad. ILUT: A dual-threshold incomplete LU factorization. *Numer. Linear Algebra Appl.*, pages 387–402, 1994.