Lawrence Berkeley National Laboratory

Lawrence Berkeley National Laboratory

Title
A Supernodal Approach to Incomplete LU Factorization with Partial Pivoting

Permalink
https://escholarship.org/uc/item/7xn2f7k0

Author
Li, Xiaoye Sherry

Publication Date
2009-09-23

eScholarship.org Powered by the California Digital Library
University of California

A Supernodal Approach to Incomplete LU Factorization with Partial Pivoting*

Xiaoye S. Li† Meiyue Shao‡

June 25, 2009

Abstract

We present a new supernode-based incomplete LU factorization method to construct a preconditioner for solving sparse linear systems with iterative methods. The new algorithm is primarily based on the ILUTP approach by Saad, and we incorporate a number of techniques to improve the robustness and performance of the traditional ILUTP method. These include the new dropping strategies that accommodate the use of supernodal structures in the factored matrix. We present numerical experiments to demonstrate that our new method is competitive with the other ILU approaches and is well suited for today's high performance architectures.

1 Introduction

As the problem size increases with the high fidelity simulations demanding fine details on large, three-dimensional geometries, iterative methods based on preconditioned Krylov subspace techniques are attractive and cheaper alternatives to direct methods. A critical component of the iterative solution techniques is the construction of effective preconditioners. Physics-based preconditioners are quite effective for structured problems, such as those arising from discretized partial differential equations. On the other hand, a class of methods based on incomplete LU decomposition are still regarded as the generally robust "black-box" preconditioners for unstructured systems arising from a wide range of applications areas. A variety of ILU techniques have been studied extensively in the past, including distinct strategies of dropping elements, such as level-of-fill structure-based approach (ILU(k)) [20], numerical threshold-based approach [19], and more recently, numerical inverse-based multilevel approach [1, 2]. The ILU(k) approach assigns a fill-level to each element, which characterizes the length of the shortest fill path leading to this fill-in [13]. The elements with level of fill larger than k are dropped. Intutively, this leads to good approximate factorization only if the fill-ins become smaller and smaller as the sequence of updates proceeds. Implementation of ILU(k) can involve a separate symbolic factorization stage to determine all the elements to be dropped. The threshold-based approach takes

*This research was supported in part by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and in part by the Special Funds for Major State Basic Research Projects (2005CB321700) of China.

†Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720. xsli@lbl.gov.

‡School of Mathematical Sciences and MOE Key Laboratory for Computational Physical Sciences, Fudan University, Shanghai 200433, China. Most of the work was performed while this author was visiting Lawrence Berkeley National Laboratory.

into account the numerical size of the elements in the factors, and drop those elements that are truly small in magnitude. This method tends to produce better approximation and is applicable for a wider range of problems, but its implementation is much more complex because the fill-in pattern must be determined dynamically. One of the most sophisticated threshold-based methods is ILUTP proposed by Saad [19, 20], which combines a dual dropping strategy with numerical pivoting ("T" stands for threshold, and "P" stands for pivoting). The dual dropping rule in ILU(τ, p) first removes the elements that are smaller than τ from the current factored row or column. Among the remaining elements, at most p largest elements are kept in order to control the memory growth. Therefore, the dual strategy is somewhat in between the structural and numerical approaches.

Our method can be considered to be a variant of the ILUTP approach, and we modified our high-performance direct solver SuperLU [6, 7] to perform incomplete factorization. A key component in SuperLU is supernode, which gives several performance advantages over a non-supernodal (e.g., column-wise) algorithm. Firstly, supernodes enable the use of higher level BLAS kernels which improves data reuse in the numerical phase. Secondly, symbolic factorization traverses the supernodal directed graph to determine the nonzero structures of L and U. Since the supernodal graph can be much smaller than the nodal (column) graph, the speed of this phase is drastically improved. Lastly, the amount of indirect addressing is reduced while performing the scatter/gather operations for compressed matrix representation. Although the average size of the supernodes in an incomplete factor is expected to be smaller than in a complete factor because of dropping, we attempted to retain supernodes as much as possible. We have adapted the dropping rules to incorporate the supernodal structures as they emerge during factorization. Therefore, our new algorithm has combined benefits of retaining numerical robustness of ILUTP as well as achiving fast construction and application of the ILU preoconditioner. In addition, we developed a number of new heuristics to enrich the existing dropping rules. We show that these new heuristics are helpful in improving the numerical robustness of the original ILUTP method.

A number of researchers have used blocking techniques to construct incomplete factorization preconditioners. But the extent to which the blocking was applied is rather limited. For example, in device simulation, Fan et al. used the blocks of size 4x4, which occurs naturally when each grid point is associated with 4 variables after discretization of the coupled PDEs [10]. Chow and Heroux used a predetermined block partitioning at a coarse level, and exploited fine-grain dense blocks to perform LU or ILU of the sparse diagonal blocks [3]. Hénon et al. developed a general scheme for identifying supernodes in ILU(k) [12], but it is not directly applicable to threshold-based dropping. Our algorithm is most similiar to the method proposed by Gupta and George [11], and we extended it to the case of unsymmetric factorization with partial pivoting.

Our contributions can be summarized as follows. We adapted the classic dropping strategies of ILUTP in order to incorporate supernode structures and to accommodate dynamic supernodes due to partial pivoting. For the secondary dropping strategy, we proposed an area-based fill control method, which is more flexible and numerically robust than the traditional column-based scheme. Furthermore, we incorporated several heuristics for adaptively modifying various threshold parameters as the factorization proceeds, which improves the robustness of the algorithm.

The remainder of the paper is organized as follows. In Section 2 we describe the computer systems, the test matrices, and the performance metrics that will be used to evaluate the new algorithm. Section 3 describes in detail the new supernodal ILU algorithm together with various dropping strategies, and presents the numerical results. In Section 4 we give some remarks of the implementational and software issues. Finally in Section 5 we compare our new code with ILUPACK, which uses an inverse-based ILU algorithm.

2 Notations and Experimental Setup

We will use the following notations throughout the paper. We use A to denote the coefficient matrix of the original linear system, L and U to denote the triangular factors. The matrix $F = L + U - I$ represents the filled matrix containing both L and U, and $M = LU$ is the preconditioning matrix. D, D_r, D_c represent diagonal matrices, P, P_r, P_c represent permutaion matrices. $\#(\mathcal{S})$ denotes the number of elements in the set $\mathcal{S}$. We use Matlab notation for integer ranges: $(s : t)$ refers to a range of integers $(s, s+1, \ldots, t)$. We use $\text{nnz}(A)$ to denote the number of nonzeros in matrix A. The *fill ratio* refers to the ratio of the number of nonzeros in the filled matrix F over that in the original matrix A. Sometimes we need to refer to the fill ratio of certain column j, i.e., $\text{nnz}(F(:, j))/\text{nnz}(A(:, j))$. The fill ratio is a direct indicator of the memory requirement. The number of operations is also related to the fill ratio, although it usually grows more linearly.

We use two platforms to evaluate our algorithms. The first is an Opteron cluster running a Linux operating system at the National Energy Research Scientific Computing (NERSC) Center.[1] Each node contains dual Opteron 2.2 GHz processors, with 5 GBytes usable memory. We use only one processor of a node. The processor's theoretical peak floating-point performance is 4.4 Gflops/sec. We use PathScale `cc` compiler with the following optimization flags: `-O3 -OPT:IEEE_arithmetic=1 -OPT:IEEE_NaN_inf=ON`, which conforms to the IEEE-754 standard.

The second machine is a Dell PowerEdge 1950 server running a Linux operating system in the MOE Key Laboratory for Computational Physical Sciences at Fudan University. It contains 2 quad-core Xeon 2.5 GHz processors, with 32 GBytes shared memory. We use only one processor but all the memory available. We use Intel's `icc` compiler with `-O3` optimization flag.

To evaluate our new ILU strategies, we have chosen 54 matrices from the following sources: Matrix Market [17], University of Florida Sparse Matrix collection [5], and five matrices from the fusion device simulation [14]. We use COLAMD for column permutaion and MC64 for equilibration and row permutation. The iterative solver is restarted GMRES with our ILU as a right preconditioner (i.e. solving $PAM^{-1}y = Pb$). The stopping criterion is $\|r_k = b - Ax_k\|_2 \le \delta \|b\|_2$, here we use $\delta = 10^{-8}$ which is in the order of the square root of IEEE double precision machine epsilon. We set the dimension of the Krylov subspace to be 50 and maximum iteration count to be 1000. We test ILU(τ) with different values of τ, such as 10^{-4}, 10^{-6}, and 10^{-8}.

To compare the effectiveness of different solvers, or different ILU parameter configurations, we will use the performance profiles similar to what was proposed by E. Dolan and J. Moré in [8] to present the data. The idea of performance profile is as follows. Given a set $\mathcal{M}$ of matrices and a set $\mathcal{S}$ of solvers, for each matrix $m \in \mathcal{M}$ and solver $s \in \mathcal{S}$, we use $fr(m, s)$ and $t(m, s)$ to denote the fill ratio and total time needed to solve m by s. If s fails to solve m for any reason (e.g. out of memory, or exceeding maximum iteration limit), we set $fr(m, s)$ and $t(m, s)$ to be $+\infty$ (in practice, a very large number that is outside the range of our interest is sufficient). Then, for each solver s, we define the following two cumulative distribution functions as the profiles of fill ratio and time ratio, respectively.

$$\gamma_s(x) = \frac{\#\{m \in \mathcal{M} : fr(m, s) \le x\}}{\#(\mathcal{M})}, \quad x \in \mathbb{R}$$

and

$$\theta_s(x) = \frac{\#\left\{m \in \mathcal{M} : \frac{t(m,s)}{\min_{s \in \mathcal{S}}\{t(m,s)\}} \le x\right\}}{\#(\mathcal{M})}, \quad x \in \mathbb{R}$$

[1] `http://www.nersc.gov/nusers/systems/jacquard`

Intuitively, $\gamma_s(x)$ shows the fraction of the problems that s could solve within the fill ratio x, and $\theta_s(x)$ shows the fraction of the problems that s could solve within a multiple of x of the best solution time among all the solvers. Therefore, the plots of different solvers in the x-γ or x-θ coordinate could differentiate the strength of the solvers in the criteria of fill ratio or time ratio—the higher the curve, the more problems the corresponding solver could solve under the same fill or time limit.

We caution that even though performance profile is a powerful tool to present overall performance summary data of different solvers, it cannot show the difference of the solvers for each individual matrix. Therefore, in addition to performance profile, we will show other specific results when there is a need.

3 Incomplete Factorization with Supernodes

3.1 Sketch of the algorithm

Our base algorithm framework is the left-looking, partial pivoting, supernodal sparse LU factorization algorithm implemented in SuperLU [6, 7]. A key concept in SuperLU is to exploit dense blocks appeared in the L and U factors. In particular, we define a supernode in L to be a range $(r : s)$ of columns with the triangular block on the diagonal being full, and the identical nonzero structure elsewhere among the columns. Using the same supernode partition to the rows of U, the nonzero structure of each column in U consists of a number of dense segments. Thus, the compressed data structure for L consists of a collection of supernodes as dense submatrices, and that for U consists of a collection of dense subvectors. For convenience, each diagonal block of U associated with each supernode is treated as full and stored together with the L supernodal structure. Moreover, we merge several columns at the fringe of elimination tree into one supernode regardless of their row structures. We call these relaxed supernodes, which help increase the average supernode size at the expense of storing a small number of explicit zeros.

The factorization algorithm is left-looking, with supernode-panel update kernel. A panel is simply a set of consective columns, and is an algorithmic blocking parameter used to enhance data reuse in the memory hierarchy; it enables use of Level 3 BLAS. At each step of panel factorization, we obtain a panel in the U factor and a panel in the L factor.

There are some preprocessing steps before the factorization kernel. In SuperLU, these include row/column equilibration and sparsity-preserving reordering of the columns. In the case of incomplete factorization, we found that it is often beneficial to include another preprocessing step to make the initial matrix more diagonal dominant (e.g., via a maximum weighted bipartite matching algorithm). For this, we use the code MC64 developed by Duff and Koster [9]. MC64 finds a permutation and the row/column scalings so that the scaled and permuted matrix has entries of modulus 1 on the diagonal and off-diagonal entries of modulus at most 1. (Mayer calls this I-matrix [16].) *We tested 54 matrices with or without MC64, and found that using MC64, the ILU-preconditioned GMRES converged for 48 matrices with average 13 iterstions, whereas without MC64, 47 matrices succeeded and the average iteration count is 34.* Moreover, without MC64, the number of zero pivots encountered is also larger.

Our incomplete factorization algorithm retains most of the algorithmic ingredients from SuperLU, with added dropping rules that are applied to the L and U factors on-the-fly. The description of the high level algorithm is given as Algorithm 1. The steps marked as **bold** correspond to the new steps introduced to perform ILU. Since partial pivoting with row interchange is used, the resulting factorization is performed on the matrix $P_r P_0 D_r A D_c P_c^T$, where D_r and D_c are diagonal scaling matrices, P_0 is the

row permutation matrix returned from MC64, P_c is the column permutation matrix for sparsity preservation, and P_r is the row permutation matrix from partial pivoting. D_r, D_c, P_0 and P_c are obtained before factorization, and P_r is obtained during factorization. In the following sections, we describe our adaptation of the dropping rules to the situation when supernodes are present, and our way to handle zero-pivot breakdowns.

Algorithm 1. *Left-looking, supernode-panel ILU algorithm*

1. Preprocessing

   1.1) **(optional)** Use MC64 to find a row permutation P_0 and row and column scaling factors D_r and D_c such that $P_0 D_r A D_c$ is an I-matrix;

   1.2) If step 1.1) is not performed, do a simple LAPACK-style row/column equilibration to obtain $D_r A D_c$;

   1.3) Compute a column permutation P_c to preserve sparsity of the LU factorization of $P_0 D_r A D_c P_c^T$;

2. Factorization of $P_0 D_r A D_c P_c^T$

   FOR each panel of columns DO

   2.1) *Symbolic factorization:* determine which supernodes to the left will update the current panel and a topological order of updates;

   2.2) *Panel factorization:*

   FOR each updating supernode DO

   Apply triangular solve to obtain the U part;

   Apply matrix-matrix multiplication to obtain the L part;

   END FOR

   2.3) *Inner factorization:*

   FOR each column j in the panel DO

   Update the current column j;

   Apply the dropping rule to the U part;

   Find pivot in this column;

   (optional) Modify the diagonal entry to handle zero-pivot breakdown;

   Determine supernode boundary;

   IF column j starts a new supernode THEN

   Apply the dropping rule to the newly formed supernode $(s : j - 1)$ **in** L**;**

   END IF

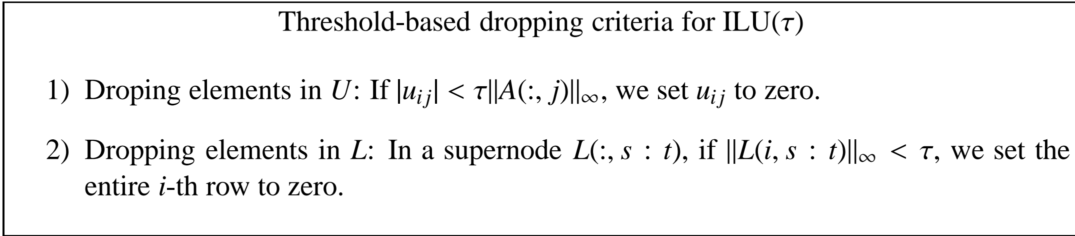
   END FOR

   END FOR

3.2 Threshold-based dropping criteria

Our primary dropping criteria are threshold-based and akin to the ILUTP variants [19, 20]. That is, while performing Gaussian elimination with partial pivoting, we set to zero the entries in L and U with modulus smaller than a prescribed threshold τ, where $\tau \in [0, 1]$.

Since our compressed storage is column oriented for both L and U, the dropping rule is also column oriented. The upper triangular matrix U is stored in a normal compressed column format, we can easily remove the small elements while storing the newly computed column into the compressed storage, using the first criterion given in Figure 1.



Threshold-based dropping criteria for ILU(τ)

1) Droping elements in U: If $|u_{ij}| < \tau \|A(:, j)\|_\infty$, we set u_{ij} to zero.
2) Dropping elements in L: In a supernode $L(:, s : t)$, if $\|L(i, s : t)\|_\infty < \tau$, we set the entire i-th row to zero.

Figure 1: The threshold-based dropping criteria.

The lower triangular matrix L is stored as a collection of supernodes. Our goal is to retain the supernodal structure to the largest extent as in the complete factorization. In a naive implementation of ILU, we may apply the traditional dropping to each individual column. But it is usually not the case that the nonzeros in the same row of the consituent columns within a supernode (resulting from a complete factorization) are dropped. This implies that after dropping, the nonzero structures among the columns in the original supernode may be different, thus, we will need to regroup the columns into smaller supernodes, resulting in performance penalty. Instead, we adopt an alternative approach that better respects the supernode structure. That is, we either keep or drop an entire row of a supernode when it is formed at the crrent step. This is similar to what was first proposed by Gupta and George for incomplete Cholesky factorization [11]. Our dropping criterion is the second rule shown in Figure 1. Note that since partial pivoting is used, the magnitude of the elements in L is bounded above by one, and so the absolute quantity is the same as the relative quantity. The use of ∞-norm for row i of a supernode implies that when row i is dropped, the magnitude of every element in this row is smaller than τ. Therefore, in a traditional column-wise algorithm, these elements should be dropped as well. On the other hand, there might be a row j such that $\|L(j, s : t)\|_\infty > \tau$ and hence row j is retained in our supernodal version, even though some elements in this row may be smaller than τ in magnitude and would have been dropped in a column-wise algorithm. To summarize, in local viewpoint, this supernodal dropping rule leads to fewer elements being dropped compared with a column-wise algorithm. But we cannot give such an analytical comparison for the entire factorization, because any difference in the current step could affect the dropping in the later factorization, We did an experiment to compare the supernodal ILU and the column-wise ILU (setting maximum supernode size to be one). *For 54 matrices, GMRES with supernodal ILU converged for 47 matrices, while the column-wise ILU succeeded with only 43 matrices. The average fill ratios of the supernodal and column-wise ILU are 13.2 and 9.8, respectively. For the 43 matrices that both versions succeeded, the supernodal ILU version is around 2.3 times faster in total GMRES solution time than the scalar version on the Dell Xeon server.* This shows that our supernodal version is numerically superior achieves higher performance than the scalar ILU.

3.3 Secondary dropping to control fill-in adaptively

ILU(τ) works well if there is sufficient memory, but it may still incur too much fill. A secondary dropping can be used to alleviate the problem.

Several methods were proposed earlier in this regard. In Saad's ILU(τ,p) approach [19], p is the largest number of nonzeros (not the level-of-fill) allowed in echo row of F (in a row-wise algorithm). Gupta and George suggested using $p(j) = \gamma \cdot \text{nnz}(A(:, j))$ for the j-th column instead of a constant, where γ is an upper bound of the fill ratio defined by a user [11]. They also proposed a method of computing a secondary dropping tolerance by an interpolation formula rather than sorting the largest p entries, which is cheaper than the original ILU(τ,p). According to our experience, Gupta's heuristic depends largely on the distribution of the nonzero modulos in F, and the fill ratio can be either very large or very small. The benefit of avoid sorting is also limited, as sorting is not very expensive. Suppose we use *quicksort*, the complexity is only $O(k \log k)$, where k is the number of nonzero entries in $F(:, j)$. There is also a linear time algorithm to find the p-th largest number in an array [4], but in practice, it may not be as fast as quicksort.

We now present a new strategy for choosing p. Given a user-desired upper bound of the overall fill ratio γ, we define an upper bound function $f(j)$ for each column j, $f : [1, n] \to [1, \gamma]$, which satisfies $f(n) \le \gamma$. Then at the j-th column, if the current fill ratio

$$\frac{\text{nnz}(F(:, 1 : j))}{\text{nnz}(A(:, 1 : j))} \tag{1}$$

exceeds $f(j)$, we choose a maximum possible value p such that when we keep the largest p elements, the current fill ratio is bounded by $f(j)$. This criterion can be adapted to our supernodal algorithm as follows. For a supernode with k columns, p may be computed as

$$p = \max\left\{\frac{f(j) \cdot \text{nnz}(A(:, 1 : j)) - \text{nnz}(F(:, 1 : j - k))}{k},\ k\right\}. \tag{2}$$

In other words, if we keep the largest p rows of this supernode, the current fill ratio is guaranteed not to exceed $f(j)$. The second k term in $\max\{\ldots\}$ is to ensure that we do not drop any row in the diagonal block of the supernode.

This is also an ILU(τ,p) approach with adaptive p, similar to Gupta's scheme. However, our fill ratio definition (1) is *area-based* instead of column-based, because we count all the fill-ins from column 1 to column j. That is, we only monitor the overall memory growth instead of that of each individual column. This is more flexible than the column-based method in that it allows larger amount of fill for certain columns so long as the cumulative fill ratio in the previous columns is small. At the end of factorization, the total fill ratio is still bounded by γ because of the condition $f(n) \le \gamma$.

The above description of area-based strategy is generic, and may be used in any implementation. We now introduce a specific $f(j)$ that is suitable for the SuperLU implementation. Since L and U are stored in different data structures, and dropping of L is invoked after a complete supernode is formed, it is sensible to use different secondary droppping rules for L and U. For a column-based method, at the j-th column, the simplest way is to split γ proportionally with $j : (n - j)$ ratio for $U(:, j)$ and $L(:, j)$. For our area-based approach, we may choose two functions, $f_L(j)$ for L and $f_U(j)$ for U, so long as $f_L(n) + f_U(n) \le \gamma$. A simple way is to assign $f_L(n)$ and $f_U(n)$ to be the areas of $L(:, j)$ and $U(:, j)$ relative to $F(:, 1 : j)$, as follows:

$$f_U(j) = \frac{j}{2n}\gamma, \quad f_L(j) = \left(1 - \frac{j}{2n}\right)\gamma. \tag{3}$$

Then we split the fill quota proportionally with $f_U(j) : f_L(j)$ ratio.

A problem with this is that dropping in U could be very constraining for small j. Then, we can simply use $f_U(j) = \gamma/2$. With this, even though it could happen that $f_L(j) + f_U(j) \geq \gamma$ in the middle of the factorization, $f_L(n) + f_U(n) \leq \gamma$ still holds in the end, and the total memory is still bounded. Since we do not apply the dropping rules toward the end in order to reduce the number of zero pivots (see Section 3.4), we need to reserve some quota for them by reducing $f_L(j) + f_U(j)$. In addition, we need to allow more fill-ins in L than in U, because the dense diagonal blocks are stored in L, and some small entries in L are located in the rows with large norm, hence are not dropped. As a result, we propose to use

$$f_U(j) = 0.9 \times \frac{\gamma}{2}, \quad f_L(j) = \left(1 - \frac{j}{2n}\right)\gamma. \tag{4}$$

In conjunction with the dynamic, area-based strategy for choosing p, we devised an adaptive scheme for choosing τ as well. The idea is that when the current fill ratio is large, we increase τ, forcing more droppings. Otherwise we decrease τ to retain more entries. Specifically, let $\tau(1) = \tau_0$ be the user-input threshold, at column j, if the fill ratio given by Equation (1) is larger than $f(j)$, we set $\tau(j+1) = \min\{1, 2\tau(j)\}$, otherwise, we set $\tau(j+1) = \max\{\tau_0, \tau(j)/2\}$. That is, we maintain $\tau(j) \in [\tau_0, 1]$. Our adaptive ILU($\tau(j)$,$p(j)$) is a simple heuristic which does not require sorting.

We now present the results of the tests comparing various parameter settings. The ILU configurations include:

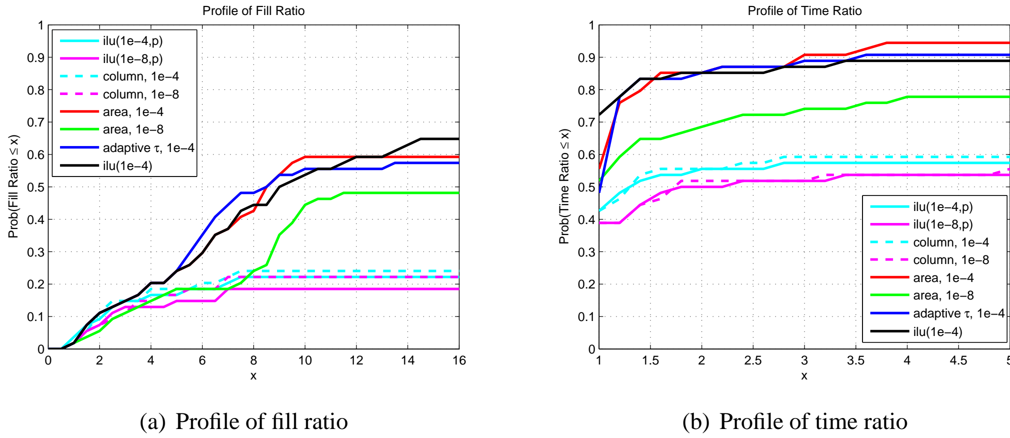
- ILU(τ), $\tau = 10^{-4}$;
- ILU(τ, p), $\tau = 10^{-4}$ or 10^{-8}, $p = \gamma \cdot \text{nnz}(A)/n$;
- column-based adaptive p, $\tau = 10^{-4}$ or 10^{-8};
- area-based adaptive p, $\tau = 10^{-4}$ or 10^{-8};
- area-based adaptive $\tau(j)$, $\tau_0 = 10^{-4}$, no secondary dropping

Figure 2 shows the performance profiles of the fill ratio and the time ratio for the 54 test matrices. We can see that a small τ such as 10^{-8} is generally not good, that is, it is not efficient to use the secondary dropping rule only. The threshold-based dropping criterion in Figure 1 should play a significant role.

A key conclusion is that our new area-based scheme is much more robust than the column-based scheme; it is also better than ILU(τ) when the fill ratio does not exceed the user-desired γ. ILU(τ) becomes better only when the fill ratio is unbounded (i.e., allow it to exceed γ). This is consistent with the intuition that an ILU preconditioner tends to be more robust with more fills.

There are two reasons why the actual fill ratio can be larger than the preset parameter γ. Firstly, our dropping rules do not drop entries in the dense diagonal blocks. Therefore, when there are some large supernodes in L, these blocks would contribute to a large memory growth even if τ is large. Secondly, we never drop any entries in several trailing columns, and usually there are a lot of fill-ins towards the end of the factorization. If the user wants the memory to be absolutely under control, we recommend that a slightly smaller γ be used.

Figure 2(b) shows the runtime comparison of the solvers using the Dell Xeon server. In this plot, the matrices with fill ratio larger than 10 are considered as failure. Thus, the comparison is made under the same memory constraint, and none of the solvers are allowed to consume much more memory than the



(a) Profile of fill ratio

(b) Profile of time ratio

Figure 2: Performance profiles after incorporating the secondary dropping rules; $\gamma = 10$.

others. The top three solvers are much better than the others. Our area-based adaptive-p or adaptive-τ schemes have quite similar performance, with the former having a slight edge over the other one.

Taking into account both memory and time, we can see that the secondary dropping helps achieve a good trade-off, with controlled fill-in and the solver not being much slower. Either our "red" scheme or "blue" scheme can be used as a default setting in the code.

3.4 Handling breakdown due to zero pivots

In the case of LU factorization with partial pivoting, zero pivots may occur due to numerical cancellations when the matrix is nearly singular. However, for an incomplete LU factorization, zeros pivots may occur more often because of dropping, which has nothing to do with numerical cancellation.

To illustrate this, let us consider the following two 2×2 matrices:

$$A_1 = \begin{bmatrix} a & b \\ c & 0 \end{bmatrix}, \qquad A_2 = \begin{bmatrix} c & 0 \\ a & b \end{bmatrix}, \qquad (bc \neq 0).$$

Assume that the column permutation is Identity. Thus, if $|c| < \tau |a|$, the (2,1) entry will be dropped, and the (2,2) entry will become zero, causing ILU to break down. Assuming that a, b, and c are drawn independently from the uniform distribution in $[-1, 1]$, we have:

$$\text{Prob}\{u_{22} = 0\} = \frac{\tau}{4} > 0.$$

In general, let us assume that the nonzero entries of a nonsingular matrix satisfy a uniform distribution in $[-1, 1]$. Then for a given sparsity pattern, if there exist a row permutation P and a column j such that $(PA)(j : n, j) = 0$, the probability of encountering a zero pivot in the j-th column would be positive. We can show that $(\tau/4)^{\text{nnz}(A)-n}$ is a lower bound of the probability. Let $B = PA$ and suppose that j is the minimum column index which satisfies $B(j : n, j) = 0$. The probability that all the pivots of the first $j - 1$ columns are diagonal entries of B and all the off-diagonal entries are dropped (with

this condition,, definitely there will be a zero pivot in the j-th column) is

$$\prod_{i=1}^{j-1}\prod_{k>i}\text{Prob}\{|B(k,i)|<\tau|B(i,i)|\} = \prod_{i=1}^{j-1}\left(\frac{\tau}{4}\right)^{\text{nnz}(B(i+1:n,i))}$$
$$\geq \left(\frac{\tau}{4}\right)^{\text{nnz}(A(:,1:j-1))-(j-1)} \geq \left(\frac{\tau}{4}\right)^{\text{nnz}(A)-n}.$$

The last inequality comes from the fact that there is at least one nonzero element in each column of a nonsingular matrix.

Usually, many zero pivots occur in the last columns, because it is more probable at the end than in the beginning that all the nonzero entries of a column are permuted to the upper triangular part. To mitigate this, we stop dropping when the column index is larger than $\max\{n - 2N_s, n \times 95\%\}$, where N_s is the maximum size of a supernode. That is, the factorization is almost finished. According to our experiement, this helps reduce a large fraction of the zero pivots.

We have devised a simple adaptive mechanism to handle the situation when a zero pivot indeed occurs. At column j, when we encounter $u_{jj} = 0$, we set it to $\hat{\tau}\,\|A(:,j)\|_\infty$ to ensure the factorization can continue and U is nonsingular after the factorization. This is equivalent to adding a small perturbation $\hat{\tau}$ to L_{ij} at the current step. If $\hat{\tau} = \tau$, the perturbation we add to u_{jj} will not exceed the upper bound of the error propagated by the droppings. In our code, we choose $\hat{\tau}(j) = 10^{-2(1-j/n)}$, which is an increasing function with the column index, rather than a constant. This prevents the diagonal entries of U from being too small, which could result in a very ill-conditioned preconditioner.

Adding a small perturbation on the zero diagonal is a simple remedy to enable the factorization to complete. This is an acceptable solution when not many zero pivots occur, otherwise, the preconditioner can be quite ill-conditioned even though the factorization completes, making this ineffective. Some other methods were proposed to handle the breakdown, such as the delayed pivoting [11] and the multilevel method [1]. We plan to investigate them in the future. But our comparison showed that our current ILU scheme is very competitive with a multilevel ILU scheme as in ILUPACK [2], see Section 5.

3.5 Relaxed pivoting with diagonal threshold

For some matrices with band structure or close to diagonally dominant, sometimes we can trade partial pivoting for a sparser factorization. Therefore, we provide a relaxed pivoting strategy which gives preference to the diagonal entries. We use a threshold parameter $\eta \in [0, 1]$ to facilitate this. If $|f_{jj}| \geq \eta \max_{i\geq j}\{|f_{ij}|\}$, we use the diagonal entry f_{jj} as the pivot. Thus, $\eta = 1.0$ corresponds to partial pivoting, and $\eta = 0.0$ corresponds to diagonal pivoting. Usually, η cannot be too small if the numerical property of the matrix is unknown because the magnitude of the entries in L can grow as much as η^{-1}. In general, even though the pivot growth can be a bit larger than one, the dropped entries are still relatively small.

Tables 1 and 2 present the numerical results with varying η, without (i.e., ILU(τ)) or with secondary dropping (i.e., ILU(τ, p)).

When secondary dropping is not used (Table 1), (threshhold) pivoting is more reliable than no pivoting at all, because general matrices are not close to being diagonally dominant, even the I-matrices may not be. As long as $\eta > 0$, the numbers of solvable problems are about the same. Except for $\eta = 1.0$, larger η tends to maintain sparsity slightly better. The fill ratio becomes large when η is very small, because we use an absolute dropping threshold for L, which results in less dropping with smaller η. For

	Diag_thresh (η)	1.0	0.1	0.01	0.001	0
ILU(10^{-4})	Number of sucesses	47	48	47	48	44
	Average fill ratio	12.6	12.0	12.6	12.9	11.7
ILU(10^{-6})	Number of sucesses	51	51	51	51	45
	Average fill ratio	28.6	28.6	28.9	29.1	29.9

Table 1: Effect of Diag_thresh (η) with ILU(τ).

	Diag_thresh (η)	1.0	0.1	0.01	0.001	0	0.8	0.5
$\tau = 10^{-4}$	Number of successes	32	33	33	35	34	34	33
	Average fill ratio	5.6	4.7	4.7	4.8	4.7	5.2	4.8
$\tau = 10^{-6}$	Number of successes	25	29	31	29	27	26	29
	Average fill ratio	5.8	6.0	6.1	5.9	5.9	5.9	6.0

Table 2: Effect of Diag_thresh (η) with ILU(τ, p), using area-based secondary dropping, $\gamma = 10$.

some matrices, the fill ratios are quite large if we use partial pivoting. However, they can be solved if we use a relaxed pivoting scheme. For example, for the fusion matrix **matrix181** of dimension 589,698, ILU factorization runs out of memory if partial pivoting is used, but it can be solved with $\eta = 0.1$. We also did experiments when MC64 is not used for ILU(10^{-4}) with $\eta = 0.1$. There were two more failures but the average fill ratio is only 8.4. Among those for which the preconditioner works, the average fill ratio is 12.8, which is not so small. As a result, we recommend that MC64 is always used, and $\eta = 0.1$ is used as a default in our code.

When secondary dropping is used (Table 2), the situation is not very conclusive, and it is difficult to choose a good η. This is mainly because the influence of drop tolerance becomes insignificant in the presence of secondary dropping. But we can see clearly that the average fill ratios are less than half of those in Table 1, and the numbers of problems successfully solved are quite smaller.

All we can advise to the users is, when memory is not at a premium, it is better not to use secondary dropping.

3.6 Three variants of MILU

Through experiments, we found that the numerical quality with the pure threshold-based dropping rule is not very satisfactory, and then we looked into the Modified ILU (MILU) method.

The MILU techniques were introduced to reduce the effect of dropping by compensating for the discarded elements [20]. The basic idea is to add up the dropped elements in a row or column to the diagonal of U. The commonly used strategy has an appealing property that it preserves the row sums relation $P_r Ae = \tilde{L}\tilde{U}e$ for a row-wise algorithm or column sums relation $e^T P_r A = e^T \tilde{L}\tilde{U}$ for a column-wise algorithm, where $\tilde{L}$ and $\tilde{U}$ are incomplete factors. Algorithm 2 gives the procedure to perform a column-wise MILU with partial pivoting. Note that for the upper triangular part f_{ij} is equal to u_{ij}, whereas for the lower triangular part $f_{ij} = l_{ij}u_{jj}$ because of scaling.

In order to accommodate our supernodal dropping criteria, we need to modify the above column-wise MILU procedure. Recall that in Algorithm 1, we apply the dropping rule 2) in Figure 1 as a new supernode in L is formed. The consequence of this "delayed" dropping is that at the time a column is processed for pivoting, the computed sum s may contain fewer dropped entries and less amount is compensated on the diagonal. Therefore, the column sums relation is not preserved. This drawback

can be circumvented if we redesign the symbolic factorization algorithm to allow different supernode partitions. Then, we could have an implementation that is numerically faithful to Algorithm 2, but the performance of the code would suffer because the average size of the supernode would be smaller. Instead, we decided to use a simpler adaptation of the code as follow. For each column of U, we accumulate in s the sum of the dropped entries. Then, s is not only added to the diagonal but is also used in choosing the proper pivot in the lower triangular part. We call this supernodal version of MILU to be SMILU-1, which is outlined in Algorithm 3.

The SMILU-1 algorithm ensures that the pivot has the largest magnitude after droppings are performed in the upper triangular part. However, we cannot guarantee that the pivot still has a relatively large absolute value after the entries in the lower triangular are dropped. The pivot could become small or even zero after we apply the dropping rule to L (that is, after applying (4) in Algorithm 3.) This may cause the factor U to be ill-conditioned or even singular, resulting in an unstable preconditioner. The SMILU-2 algorithm in Algorithm 4 provides a remedy for this problem. Here, we ensure that the magnitude of the pivot is nondecreasing after diagonal compensations, thereby avoiding small pivots.

An alternative method is to accumulate the one-norm of the dropped vector. This will make the pivots have larger absolute values compared to what would be in SMILU-2, and we expect the condition number of U would be smaller. We call this SMILU-3, shown in Algorithm 5.

For SMILU-2 and SMILU-3, $|f_{ij} + \mathrm{sign}(f_{ij})s| \equiv |f_{ij}| + s$, since s is always nonnegative. As a result, the trick we use in SMILU-1 to select proper pivot for partial pivoting (Step (2) in Algorithm 3) is no longer needed here.

In Table 3, we compare the performance of various ILU algorithms and direct solver SuperLU, using the 54 test matrices. We classify the failures in three categories: "slow" means the stopping criterion is not met, although the residual norm is still decreasing while the maximum iteration count is exceeded (i.e, $\delta < \|r\|_2/\|b\|_2 < 1$), "diverge" means $\|r\|_2 \geq \|b\|_2$, and "memory" means the code runs out of memory. We used the Dell Xeon server that has 32 GBytes memory. Because of large amount of memory available, SuperLU succeeded with all but one problem, for which the code ran out of memory. As can be seen, when τ is sufficiently small, e.g., $\tau = 10^{-6}$ or $\tau = 10^{-8}$, the ILU algorithms can solve the same number of problems as SuperLU. ILU(τ) usually works very well, however, when it fails, it is often due to a lot of zero pivots.

Figure 3 shows performance profiles of various ILU algorithms and SuperLU, using the Opteron cluster. For a certain time limit, various ILUs can solve many more problems than SuperLU. The ILUs are also advantageous over SuperLU in terms of fill ratio. Since the Opteron cluster has a relatively smaller amount of memory than the Dell Xeon server, SuperLU fails with more problems due to memory exhaustion. SMILU-2 and SMILU-3 are quite comparable, especially for the matrices with small fill ratio. They are designed for avoiding the factor U to be ill-conditioned, but the L and U factors can be far from PA, which could result in slow convergence. We can see from the figure that when τ is small, such as $\tau = 10^{-6}$, or 10^{-8}, the difference between the different variants of ILUs are very small, mainly because the number of entries dropped is small.

4 Comments on Software

In this section, we describe a few implementational difficulties encountered while performing incomplete factorization, and summarize the new parameters introduced to the ILU routine.

Algorithm 2. *Classic column-wise MILU for column j*

(1) Obtain the current filled column $F(:, j)$;
(2) Compute the sum of dropped entries in $F(:, j)$: $s = \sum_{\text{dropped}} f_{ij}$;
(3) Set f_{jj} to be $f_{jj} + s$;
(4) Pivot: row interchange in the lower triangular part $F(j : n, j)$;
(5) Separate U and L: $U(1 : j, j) = F(1 : j, j)$; $\quad L(j : n, j) = F(j : n, j)/F(j, j)$;

Algorithm 3. *SMILU-1: Supernodal MILU for column j*

(1) Compute the sum of the dropped entries in $U(:, j)$: $s = \sum_{\text{dropped}} u_{ij}$;
(2) Choose pivot row i, such that $i = arg \max_{i \geq j} |f_{ij} + s|$;
(3) Swap rows i and j, and set $u_{jj} := f_{ij} + s$;
(4) IF j starts a new supernode THEN
    Let $(r : t)$ be the newly formed supernode; $(t \equiv j - 1)$
    For each column k in the supernode $(r \leq k \leq t)$:
        compute the sum of the dropped entries: $S_k = \sum_{i \text{dropped}} l_{ik}$;
        set $u_{kk} := u_{kk} + S_k \cdot u_{kk}$;
    END IF;

Algorithm 4. *SMILU-2: Supernodal MILU for column j*

(1) Compute the sum of the dropped entries in $U(:, j)$: $s = |\sum_{\text{dropped}} u_{ij}|$;
(2) Choose pivot row i, such that $i = arg \max_{i \geq j} |f_{ij}|$;
(3) Swap rows i and j, and set $u_{jj} := f_{ij} + \text{sign}(f_{ij})s$;
(4) IF j starts a new supernode THEN
    Let $(s : t)$ be the newly formed supernode; $(t \equiv j - 1)$
    For each row i that is dropped, set $u_{kk} := u_{kk} + |l_{ik}|u_{kk}$, for $s \leq k \leq t$;
    END IF;

Algorithm 5. *SMILU-3: Supernodal MILU for column j*

(1) Compute the sum of the dropped entries in $U(:, j)$: $s = \sum_{\text{dropped}} |u_{ij}|$;
(2) Choose pivot row i, such that $i = arg \max_{i \geq j} |f_{ij}|$;
(3) Swap rows i and j, and set $u_{jj} := f_{ij} + \text{sign}(f_{ij})s$;
(4) IF j starts a new supernode THEN
    Let $(s : t)$ be the newly formed supernode; $(t \equiv j - 1)$
    For each row i that is dropped, set $u_{kk} := u_{kk} + |l_{ik}|u_{kk}$, for $s \leq k \leq t$;
    END IF;

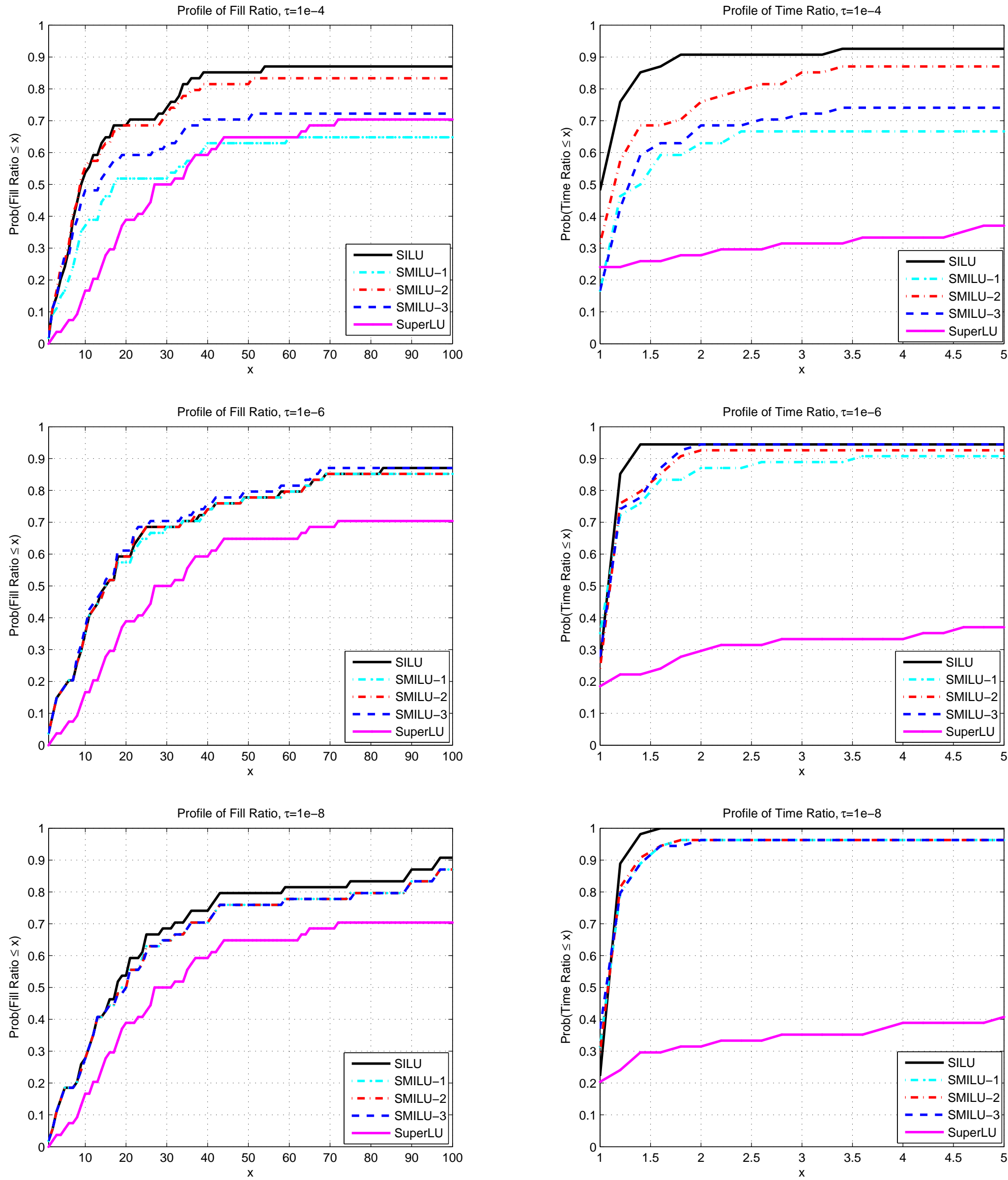


Figure 3: Performance profiles of various of ILU algorithms with $\tau = 10^{-4}, 10^{-6}$, or 10^{-8}, using the Opteron cluster. The left column is the profile w.r.t. the fill ratio, and the right column is the profile w.r.t. the time ratio. Secondary dropping is turned off.

		converge	slow	diverge	memory	zero pivots
$\tau = 10^{-4}$	ILU	47	3	4	0	7786
	SMILU-1	35	15	4	0	9
	SMILU-2	44	6	4	0	9
	SMILU-3	38	14	2	0	9
$\tau = 10^{-6}$	ILU	51	0	3	0	685
	SMILU-1	49	3	1	1	0
	SMILU-2	50	3	1	0	0
	SMILU-3	49	5	0	0	0
$\tau = 10^{-8}$	ILU	52	1	0	1	0
	SMILU-1	50	3	0	1	0
	SMILU-2	50	3	0	1	0
	SMILU-3	51	3	0	0	0
($\tau = 0.0$)	SuperLU	53	0	0	1	0

Table 3: Comparison of various ILU(τ) algorithms and SuperLU on the Dell Xeon server. The last column "zero pivots" indicates the number of zero pivots encountered during ILU factorization. Secondary dropping is turned off.

4.1 Difficulty with symmetric pruning

Symmetric pruning is a technique to find a smaller graph (symmetric reduction) in place of $G(L^T)$ and that maintains the path-preserving property. Using symmetric reduction can speed up the depth-first search traversals (i.e., the symbolic factorization) which are interleaved with the numerical factoriation steps. Specifically, at step j, the symmetric reduction of the current factor $L(:, 1 : j)$ is obtained by removing all nonzeros l_{rs} for which $l_{ts}u_{st} \neq 0$ for some $t < \min(r, j)$ [6]. That is, in L, the nonzeros below the first matching nonzero pair in column and row $s < j$ of the factor $F(1 : j, 1 : j)$ can be removed. Consider the following 4×4 matrix A, the filled matrix F (using the given elimination order), and the symmetric reduction R:

$$A = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & & \\ & & \bullet & \\ \bullet & & & \end{bmatrix}, \quad F = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \circ & \circ \\ & & \bullet & \\ \bullet & \circ & \circ & \circ \end{bmatrix}, \quad R = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \circ & \circ \\ & & \bullet & \\ \ominus & \circ & \circ & \circ \end{bmatrix}.$$

In F and R, a symbol "$\circ$" indicates a fill-in entry. In R, a symbol "$\ominus$" indicates a removed entry from symmetric pruning, that is, l_{41} is removed due to the matching nonzero pair l_{21} and u_{12}. If $G(F)$ is used in the depth-first traversal, the entry l_{43} is obtained by the following path:[2]

$$3 \xrightarrow{A} 1 \xrightarrow{F} 4$$

When using the reduced graph $G(R)$, the above path is replaced by the following one, and the reachability is maintained:

$$3 \xrightarrow{A} 1 \xrightarrow{R} 2 \xrightarrow{R} 4$$

[2]We use the convention that an edge is directed from a column to a row of the matrix.

However, in an incomplete factorization, if the magnitude of l_{42} is smaller than the threshold, it would be dropped both in F and in R. Then the edge $2 \xrightarrow{R} 4$ does not exist anymore. The entry l_{43} would be missing if R is used for the depth-first search, and similarly for l_{44}. The erroneous R is shown below, where "⊗" indicates a numerical dropping in ILU.

$$R_{ilu} = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \circ & \circ \\ & & \bullet & \\ \ominus & \otimes & & \end{bmatrix}.$$

We thought about several ways to mitigate this problem, such as delayed pruning or protecting pruned entries from dropping. But their implementations would incur nontrivial costs in tuntime and memory. We did some tests to evaluate the benefit of pruning. For complete factorization, even if the pruned graph is very small, i.e. size of R less than 5% of that of F, the total speedup is usually no more than 20%. For incomplete factorization, since the fill ratio is often much smaller (i.e., F is already quite small), we expect the benefit of pruning would be less. Therefore, we decided not to use any reduced graph.

4.2 Zero pivots and relaxed supernodes

In SuperLU's complete factorization, we use relaxed supernodes to increase the average size of supernodes (or block size). It groups several columns at the fringe of the column elimination tree into an artificial supernode [6]. The column elimination tree is the elimination tree (etree) of $|A|^T|A|$, which shows the columns' dependencies for any row permutation (partial pivoting). That is, the relaxed supernodes at the bottom of the etree will not be modified by any other columns outside these supernodes. Given a postordered etree, this means that the nonzero row structure of a column $L(:, j)$ must be disjoint from that of a later supernode $(r : s) > j$. Otherwise, there exists a numerical assignment such that a common row i can be selected as a pivot at step j, making supernode $(r : s)$ dependent on column j. Therefore, selecting any pivot column j has no impact on supernode $(r : s)$.

On the other hand, in an incomplete factorization, if zero pivot occurs in column j due to dropping, we cannot choose a random row below the diagonal as pivot, because it could overlap with a row in the future relaxed supernodes, which in essence changes the etree structure and dependency. Therefore, we must choose a pivot row which does not appear in any later relaxed supernode.

4.3 Tunable parameters in the ILU routine

The new ILU routine is named `xGSITRF`, which takes `options` structure as the first argument, which contains a set of parameters to control how the ILU decomposition will be performed. The default values of these parameters are given in Table 4, which are set by calling the routine `ilu_set_default_options()`. The users may modify these values based on their problems need.

Based on our experience, we provide the following guidelines regarding how to choose the parameters if the defaults do not work:

- Equilibration is necessary, and MC64 is usually helpful.
- If zero pivots occur and the preconditioner is too ill-conditioned, you should try modified variants SMILU-2 or SMILU-3.

Options	Default
MC64	ON
equilibration	ON
drop tolerance (τ)	10^{-4}
fill-ratio bound (γ)	10
diag_thresh (η)	0.1
column permutation	COLAMD
SMILU	SMILU-2
secondary dropping	area-based

Table 4: Default values of the parameters of the ILU routine `xGSITRF`.

- If the fill ratio is still small, you may try a smaller τ.
- If you run out of memory, you may try a smaller γ and a smaller η.

5 Comparison with ILUPACK

There are a number of ILU preconditioning packages for unsymmetric matrices, such as SPARSKIT [18], ILUPACK [2], ILU++ [15]. We have compared our algorithm with ILUPACK V2.1, which has been gaining popularity.

We choose to compare with ILUPACK mainly because it uses a very different approach than ours; it is an inversed-based method, and uses a relatively new multilevel approach to handle small pivots. The inverse-based approach attempts to control the size of the inverse of the preconditioner so that the preconditioner has a small condition number. This objective is achieved indirectly in ILUPACK: at step k of factorization, the algorithm monitors the norm of the k-th row of L^{-1}. If that exceeds the prescribed bound ν, implying no suitable pivot can be chosen at this step, then row k and column k is moved to the end, and the factorization continues to the next row/column. After all fthe good pivots are chosen, the current level is considered to be complete, the factorization starts a new level, which is comprised of all the delayed rows and columns from the previous level.

Our tests were carried out on the Dell Xeon cluster. For ILUPACK, we downloaded the precompiled 64-bit libraries, and compiled only the main() function using "`gcc -O3`". For our code, we used "`icc -O3`" to compile, and linked with `GotoBLAS` library.

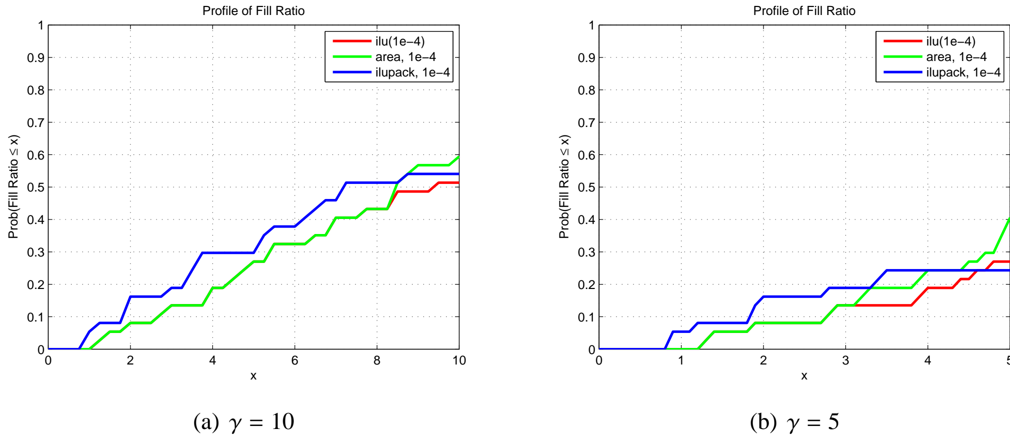
In our experiments, we tried to keep the same parameter settings for both codes:

- Our ILU: $\tau = 10^{-4}$, area-based secondary dropping $\gamma = 5$ or 10, diagonal threshold η=0.1;
- ILUPACK: $\tau = 10^{-4}$, $\nu = 5$, secondary dropping $\gamma = 5$ or 10 (corresp. to "`param.elbow`" in the code.)

The ordering algorithms are different: our code uses a column reordering method such as Column Approximate Minimum Degree, for which the underlying graph model is the adjacency graph of $|A|^T|A|$. ILUPACK uses a symmetric reordering such as Approximate Minimum Degree, for which the underlying graph model is the adjacency graph of $|A|^T + |A|$.

From our 54 test matrices, we chose 37 which are available in Harwell-Boeing Figure 4(a) shows the performance profiles of the two preconditioners with $\gamma = 10$ in our secondary dropping. For smaller

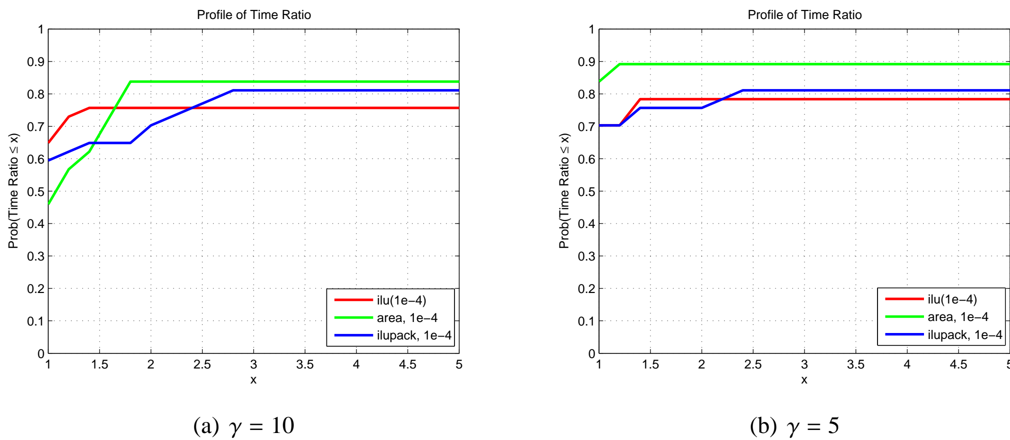
allowable fill ratios, ILUPACK could solve a few more problems than our ILU does. However, when the fill ratio is close to the prescribed limit γ, our code can solve more problems. If we use a smaller γ, the curves of profile will be changed, see Figure 4(b), and they are different from just cutting off at $\gamma = 5$ in the left figure.



(a) $\gamma = 10$ (b) $\gamma = 5$

Figure 4: Comparison of fill ratio between our supernodal ILU and ILUPACK.

Figure 5 compares the runtime of ILUPACK with two of our ILU variants, one is ILU(τ), another is ILU(τ, p). This shows that our area-based adaptive ILU(τ, p) is superior to the other solvers.



(a) $\gamma = 10$ (b) $\gamma = 5$

Figure 5: Comparison of runtime between our supernodal ILU and ILUPACK.

To explore some detailed information about these solvers, we examined 11 hard problems for which at least one solver fails and at least one solver workss. The success-failure instances are tabulated in Table 5. Our area-based approache is better than the orginal ILU(τ) approach. ILUPACK solved different set of problems, but ours can solve a few more. Overall, our approache is at least competitive with ILUPACK.

Matrix	ILU(τ)	ILU(τ, p), area-based, static τ	ILUPACK
NASASRB	∘	×	∘
ecl32	×	×	∘
gemat11	∘	∘	×
jpwh_991	×	∘	∘
onetone2	∘	∘	×
twotone	∘	∘	×
vavasis1	×	×	∘
vavasis2	×	×	∘
wang3	×	∘	×
wang4	×	∘	×
xenon2	×	∘	×
total solved	4	7	5

Table 5: The selected matrices for which at least one solver works (shown as "∘") and one solver fails (shown as "×"). We set $\tau = 10^{-4}$ and $\gamma = 10$.

6 Conclusions

We adapted the classic dropping strategies of ILUTP in order to incorporate supernode structures and to accommodate dynamic supernodes due to partial pivoting. For the secondary dropping strategy, we proposed an area-based fill control mechanism which is more flexible and numerically more stable than the traditional column-based scheme. Furthermore, we incorporated several heuristics for adaptively modifying various threshold parameters as the factorization proceeds, which improves the robustness of the algorithm. The numerical experiments show that our new supernodal ILU algorithm is competitive with an inversed-based ILU method as implemented in ILUPACK. The new ILU routine will be available in SuperLU Version 4.0.

In the future, we plan to investigate different methods for handling zero pivots in order to enhance stability of the factorization, add more adaptivity, and study the preconditioning effect with the other iterative solvers.

References

[1] M. Bollhöfer and Y. Saad. Multilevel preconditioners constructed from inverse-based ILUs. *SIAM J. Scientific Computing*, 27(5):1627–1650, 2006.

[2] M. Bollhöfer, Y. Saad, and O. Schenk. ILUPACK - preconditioning software package. `http://ilupack.tu-bs.de`, TU Braunschweig, 2006.

[3] EDMOND CHOW and MICHAEL A. HEROUX. An object-oriented framework for block preconditioning. *ACM Trans. Mathematical Software*, 24:159–183, 1998.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[5] Timothy A. Davis. University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices.

[6] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999.

[7] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. SuperLU Users' Guide. Technical Report LBNL-44289, Lawrence Berkeley National Laboratory, September 1999. http://crd.lbl.gov/~xiaoye/SuperLU/. Last update: September 2007.

[8] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–203, 2002.

[9] Iain S. Duff and Jacko Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Analysis and Applications*, 20(4):889–901, 1999.

[10] QING FAN, P. A. FORSYTH, J. R. E MCMACKEN, and WEI-PAI TANG. Performance issues for iterative solvers in device simulation. *SIAM J. Scientific Computing*, 17(1):100–117, 1996.

[11] A. Gupta and T. George. Adaptive techniques for improving the performance of incomplete factorization preconditioning. Technical Report RC 24598(W0807-036), IBM Research, Yorktown Heights, NY, 2008.

[12] P. Hénon, P. Ramet, and J. Roman. On finding approximate supernodes for an efficient ILU(k) factorization. *Parallel Computing*, 34:345–362, 2008.

[13] David Hysom and Alex Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J. Scientific Computing*, 22(6):2194–2215, 2001.

[14] S. C. Jardin, J. Breslau, and N. Ferraro. A high-order implicit finite element method for integrating the two-fluid magnetohydrodynamic equations in two dimensions. *Journal of Computational Physics*, 226:2146–2174, 2007.

[15] J. Mayer. ILU++. http://iamlasun8.mathematik.uni-karlsruhe.de/~ae04/iluplusplus.html, 2007.

[16] J. Mayer. Symmetric permutations for I-Matrices to delay and avoid small pivots during factorization. *SIAM J. Scientific Computing*, 30(2):982–996, 2008.

[17] Matrix Market. http://math.nist.gov/MatrixMarket/.

[18] Y. Saad. SPARSKIT: A basic tool-kit for sparse matrix computations. http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html.

[19] Y. Saad. ILUT: A dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, 1(4):387–402, 1994.

[20] Yousef Saad. *Iterative methods for sparse linear systems*. PWS Publishing Company, Boston, MA, 1996.