A Supernodal Approach to Incomplete LU Factorization with Partial Pivoting[*]

Xiaoye S. Li[†] Meiyue Shao[‡]

May 26, 2010

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 new dropping strategies that accommodate the use of supernodal structures in the factored matrix and an area-based fill control heuristic for the secondary dropping strategy. 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 problem sizes increase with 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, methods based on incomplete LU decomposition are still regarded as generally robust "black-box" preconditioners for unstructured systems arising from a wide range of application areas. A variety of ILU techniques have been studied extensively in the past, including distinct dropping strategies, such as the level-of-fill structure-based approach (ILU(k)) [28], numerical threshold-based approach [27], and more recently, numerical inverse-based multilevel approach [3, 4]. 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 [20]. The elements with level of fill larger than k are dropped. Intuitively, 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

[*]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. D-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.

to be dropped. The threshold-based approach takes into account the numerical size of the elements in the factors, and drops those elements that are smaller than a numerical tolerance. A problem with threshold-based approaches is the difficulty of choosing an appropriate drop tolerance. 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 [27, 28], which combines a dual dropping strategy with numerical pivoting ("T" stands for threshold, and "P" stands for pivoting). The dual dropping rule in ILUTP(p,τ) 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 [9, 10] to perform incomplete factorization. A key component in SuperLU is the use of supernodes, 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 can be significantly 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 have 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 the combined benefits of retaining numerical robustness of ILUTP as well as achieving fast construction and application of the ILU preconditioner. 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 four-by-four blocks which occurs naturally when each grid point is associated with four variables after discretization of the coupled PDEs [14]. 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 [6]. Hénon et al. developed a general scheme for identifying supernodes in ILU(k) [18], but it is not directly applicable to threshold-based dropping. Our algorithm is most similar to the method proposed by Gupta and George [16], and we extend it to the case of unsymmetric factorization with partial pivoting, see Section 3.2.

Our contributions can be summarized as follows. We adapt 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 propose an area-based fill control method, which is more flexible and numerically robust than the traditional column-based scheme. Furthermore, we incorporate several heuristics for adaptively modifying various threshold parameters as the factorization proceeds, which improves the robustness of the algorithm. Finally, the implementation of the algorithm has already been incorporated in the SuperLU Version 4.0 release, downloadable at `http://crd.lbl.gov/~xiaoye/SuperLU/`.

The remainder of the paper is organized as follows. In Section 2 we describe 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 on the implementational and software issues. Finally in Section 5 we compare our new code with the other codes that use different approximate factorization algorithms.

2 Notations and Experimental Setup

We use A to denote the $n \times n$ coefficient matrix of the original linear system, L and U to denote the incomplete 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 permutation matrices. $\#\mathcal{S}$ denotes the cardinality of the set $\mathcal{S}$. We use array section notation in Fortran and MATLAB $(s : t)$ to refer to a range of integers $(s, s+1, \ldots, t)$. We use $\mathrm{nnz}(A)$ 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 a certain column j, i.e., $\mathrm{nnz}(F(:, j))/\mathrm{nnz}(A(:, j))$. The fill ratio is a direct indicator of the memory requirement, because our code requires very little working storage. The number of operations is also related to the fill ratio, although it usually grows more linearly.

Our test machine is a Cray XT5 with 664 compute nodes operated at the National Energy Research Scientific Computing (NERSC) Center.[a]. Each node contains two 2.4 GHz AMD Opteron quad-core processors, with 16 GBytes memory. We use only one core of a node. Each core's theoretical peak floating-point performance is 9.6 Gflops/sec. We use PathScale `pathcc` and `pathf90` compilers with `-O3 -fPIC` optimization flag.

We have used 232 matrices to evaluate our new ILU strategies. These include 227 matrices from the University of Florida Sparse Matrix collection [8], and 5 matrices from the fusion device simulation [21]. These are all unsymmetric matrices of medium to large size. In particular, the 227 matrices are all the real unsymmetric matrices in the UF collection which are of dimension 5K–1M and have condition numbers below 10^{15}. The right-hand side vector is chosen such that the true solution vector is of all ones. The iterative solver is restarted GMRES with our ILU as a right preconditioner (i.e. solving $P_r A M^{-1} y = P_r b$). The initial guess is a vector of all zeros. The stopping criterion is $\|r_k = b - Ax_k\|_2 \leq \delta \, \|b\|_2$, 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 500. We test ILUTP(τ) with different values of τ, such as 10^{-4}, 10^{-6}, and 10^{-8}.

We mainly use two performance metrics to assess the algorithms: memory requirement as reflected by the amount of fill in the preconditioners and the total solution time. To compare different solvers, or different ILU parameter configurations, we use the performance profiles similar to what was proposed by E. Dolan and J. Moré in [11] 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.

[a] `http://www.nersc.gov/nusers/systems/hopper/`

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

and

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

Intuitively, $Prob_f(s,x)$ shows the fraction of the problems that s could solve within the fill ratio x, and $Prob_t(s,x)$ shows the fraction of the problems that s could solve within a factor of x of the best solution time among all the solvers. Therefore, the plots of different solvers in the x–$Prob_f$ or x–$Prob_t$ coordinate can be used to 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 the performance profile is a powerful tool to present visually the overall performance comparison of different solvers, it cannot show the difference of the solvers for each individual matrix. Therefore, in addition to the performance profiles, we will show other specific results when there is a need.

3 Incomplete Factorization with Supernodes

3.1 Left-looking supernodal ILUTP

Our base algorithm framework is the left-looking, partial pivoting, supernodal sparse LU factorization algorithm implemented in SuperLU [9, 10]. The factorization algorithm proceeds from left to right, using a supernode-panel updating kernel. A panel is simply a set of consecutive columns, which is used to enhance data reuse in memory hierarchy; it enables use of Level 3 BLAS. The panel size is an algorithmic blocking parameter. At each step of panel factorization, we obtain a panel in the U factor and a panel in the L factor.

Several preprocessing steps are used in SuperLU before the factorization kernel, including 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 diagonally dominant (e.g., via a maximum weighted bipartite matching algorithm). For this, we use the HSL subroutine MC64 [12, 19], which is based on the algorithm developed by Olschowska and Neumaier [25]. The algorithm 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 bounded by 1. We tested 232 matrices with or without MC64, and found that using MC64, the ILU-preconditioned GMRES converge for 203 matrices with average 12 iterations, whereas without MC64, only 170 matrices converge and the average iteration count is 11.

Our incomplete factorization algorithm retains most of the algorithmic ingredients from SuperLU, with the added dropping rules that are applied to the L and U factors on-the-fly [22]. The description of the algorithm is given as Algorithm 1. The steps marked as **bold** correspond to the new steps introduced to perform ILU. Note that the 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 the factorization, and P_r is obtained during factorization.

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 the elements on the diagonal of $P_0 D_r A D_c$ has the largest absolute
        values.
   1.2) If Step 1.1) is not performed, do a simple row & column equilibration to obtain $D_r A D_c$
        in which the element of largest absolute value in each row and column has value 1;
   1.3) Compute a fill-reduction column permutation $P_c$;

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 $L(:, s : j-1)$,
                where $s$ is the first column of this supernode;
            END IF
        END FOR
    END FOR
```

Our primary dropping criteria are threshold-based and akin to the ILUTP variants [27, 28]. 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 relative 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 standard 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.

The lower triangular matrix L is stored as a collection of supernodes. Recall that a supernode consists of a range $(r : s)$ of columns in L with the triangular block on the diagonal being full, and the identical nonzero structure elsewhere among the columns. Our goal is to retain the supernodal structure to the largest extent as in the complete factorization. To this end, we either keep or drop an entire row of a supernode when it is formed at the current step. This is similar to what was first proposed in [16, 23] for incomplete Cholesky factorizations. This supernodal dropping criterion is the second rule shown in Figure 1. Since partial pivoting is used, the magnitude of the elements in L is bounded by 1, and so the absolute quantity is the same as the relative quantity.

Threshold-based dropping criteria for ILUT(τ)

1) Dropping elements in U: If $|u_{ij}| \leq \tau \|A(:, j)\|_\infty$, we set u_{ij} to zero.
2) Dropping elements in L: In a supernode $L(:, s : t)$, if $RowSize(L(i, s : t)) \leq \tau$, we set the entire i-th row to zero, where s and t are the first and last columns of the supernode.

Figure 1: The threshold-based dropping criteria, with L-supernode.

Note that we use $RowSize()$ function to determine whether the size of a row as a whole is considered to be small and can be dropped. There are several possibilities to define this metric. In our code, we provide three choices: (1) $RowSize(x) \overset{\text{def}}{=} \|x\|_1/m$, (2) $RowSize(x) \overset{\text{def}}{=} \|x\|_2/\sqrt{m}$, and (3) $RowSize(x) \overset{\text{def}}{=} \|x\|_\infty$, where m is the number of columns in the supernode (i.e., $m = t - s + 1$). The first two metrics can be interpreted as "generalized mean" of the vector elements, whereas the last one is a standard vector norm. The following relations hold for the three metrics above:

$$\frac{\|x\|_1}{m} \leq \frac{\|x\|_2}{\sqrt{m}} \leq \|x\|_\infty$$

In case (1), a row can be dropped when the average magnitude of the elements is small, but some elements much larger than τ can also be dropped. This is the same criterion used by Gupta and George [16] in their IC algorithm. Our experiments showed that this metric is much worse than (3) for ILU probably because many large elements are dropped. In case (3), the use of ∞-norm implies that when row i is dropped, the magnitude of every element in this row is smaller than τ, and hence these elements would also be dropped in a column-wise algorithm. Therefore, from local viewpoint, this supernodal dropping rule retains more elements compared to a column-wise algorithm. In other words, in terms of the amount of dropping, (1) is most aggressive and (3) is most conservative.

Throughout the rest of the paper, the supernodal ILUTP variants are used. In Section 3.3, we will show the numerical results comparing our supernodal ILU to the column-wise ILU when setting supernode size to be one.

3.2 Adaptive, area-based dropping rules to limit memory use

ILUTP(τ) works well if there is sufficient memory, but it may still suffer from too much fill. We employ two new ideas to further control the amount of fill. One is dynamically adjusting parameter p in the dual dropping rule, another is dynamically adjusting τ. We use an area-based fill estimation to adjust both parameters, which will become clear soon.

Several methods were proposed earlier for the secondary dropping rule using parameter p. In Saad's original ILUT(p, τ) approach [27], p is the maximum number of nonzeros (not the level-of-fill) allowed in each row of F (in a row-wise algorithm), and is fixed on input. 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 given by a user [16]. 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 ILUT(p, τ). According to our experience, Gupta's heuristic depends largely on the distribution of the locations of the nonzeros in F, and the fill ratio can be either very large or very small. The performance benefit realized by not sorting is also limited. In fact, sorting is not necessary here; a selection algorithm is more suitable, because we only need to find the p-th largest element. If the number of nonzero entries in $F(:, j)$ is k, the p-th largest element can be found by quickselect in $O(k)$ time on average, which is the same as interpolation. Even for a sorting-based selection algorithm, the complexity is merely $O(k \log k)$ on average using quicksort and is reasonably fast in practice. Our implementation uses quickselect.

We propose a new adaptive strategy for choosing p, which takes into account the overall fill ratio *up to* the current step. Given a user-desired upper bound of the overall fill ratio γ, we define a continuous, upper bound function $f(j)$ for each column j, $f : [1, n] \to [0, +\infty)$, 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 largest 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 and j being the index of the last column of the current supernode, 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. We call this scheme an *area-based* ILUT(p, τ), with adaptive p. This is more flexible than the column-based method in that it allows larger amount of fill for certain columns as 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 the 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 in L is invoked after a complete supernode is formed, it is sensible to use different $f(j)$ 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, as 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 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 factorization, $f_L(n) + f_U(n) \leq \gamma$ 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

$$f_U(j) = \frac{\gamma}{2} \times 90\%, \quad f_L(j) = \left(1 - \frac{j}{2n}\right)\gamma. \tag{4}$$

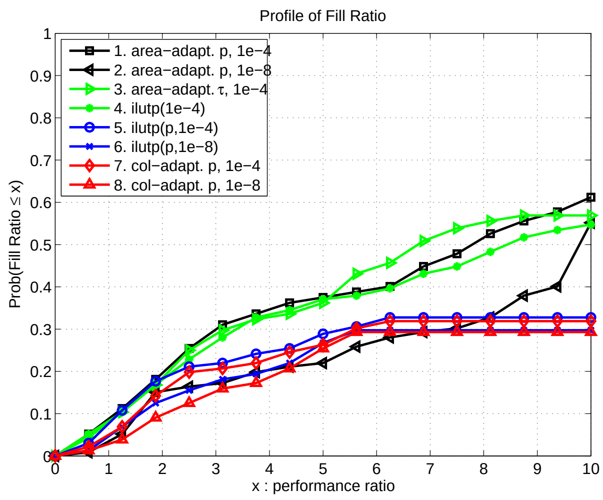
In conjunction with the dynamic, area-based strategy for choosing p, we devised an adaptive scheme for choosing τ as well. Specifically, let $\tau(1) = \tau_0$ be the user-input threshold, at column j, if the fill ratio given by Equation (1) is larger than $f(j)$, we increase τ at the next step: $\tau(j+1) = \min\{1, 2\tau(j)\}$, forcing more elements to be dropped. Otherwise, we decrease τ as $\tau(j+1) = \max\{\tau_0, \tau(j)/2\}$, retaining more entries. Note that we always maintain $\tau(j) \in [\tau_0, 1]$. An advantage of this adaptive τ is that even though the initial τ chosen by the user is not good, the code can self-adjust properly as it proceeds.

We now present the results of the tests comparing various parameter settings. When the secondary dropping is used, we set $\gamma = 10$. Our ILU configurations include the following:

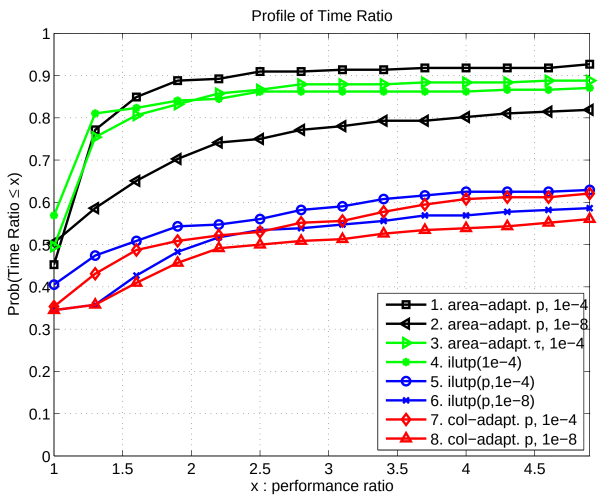
1. area-based adaptive p, $\tau = 10^{-4}$;
2. area-based adaptive p, $\tau = 10^{-8}$;
3. area-based adaptive τ, $\tau_0 = 10^{-4}$, no secondary dropping;
4. ILUTP(τ), $\tau = 10^{-4}$;
5. ILUTP(p, τ), $p = \gamma \cdot \text{nnz}(A)/n$, $\tau = 10^{-4}$;
6. ILUTP(p, τ), $p = \gamma \cdot \text{nnz}(A)/n$, $\tau = 10^{-8}$;
7. column-based adaptive p (i.e., $p(j) = \gamma \cdot \text{nnz}(A(:, j))$), $\tau = 10^{-4}$;
8. column-based adaptive p (i.e., $p(j) = \gamma \cdot \text{nnz}(A(:, j))$), $\tau = 10^{-8}$.

Figure 2 shows the performance profiles of the fill ratio and the time ratio for the 232 test matrices. We see that a small τ such as 10^{-8} is generally not good, because it generates too much fill and takes long to compute the solution. It is therefore not efficient to rely only on the secondary dropping rule. The threshold-based dropping criterion in Figure 1 must play a significant role. If we do not use the secondary dropping, the fraction of the problems solved within a fill constraint is still smaller than after using our area-based secondary dropping, see the curve corresponding to "ilutp(1e-4)" in Figure 2(a). Both the non-adaptive and the adaptive column-based secondary dropping heuristics are much worse than the area-based secondary dropping. This is clearly seen from the bottom four curves in both Figures 2(a) and 2(b).

A key conclusion is that our new area-based scheme is much more robust than the column-based scheme; it is also better than ILUTP(τ) when the fill ratio does not exceed the user-desired γ (10 in these cases). ILUTP(τ) becomes better only when the fill ratio is unbounded (i.e., allowing it to exceed γ). This is consistent with the intuition that an ILU preconditioner tends to be more robust with more fill-ins.



(a) Profile of fill ratio



(b) Profile of time ratio

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

It is possible that the actual fill ratio is larger than the preset parameter γ although this occurs occasionally. There are two reasons for this to occur. 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 factorization. If the user wants the memory to be absolutely under a certain limit, we recommend that a slightly smaller γ be used.

Figure 2(b) shows the runtime comparison of the solvers. 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 significantly more memory than the others. The top three solvers are much better than the others. The area-based adaptive-p or adaptive-τ schemes have quite similar performance, with the former having a slight edge over the latter.

Taking into account both memory and time, we see that the secondary dropping helps achieve a good trade-off, with controlled fill-in and the solver not being much slower. Either the scheme corresponding to the line with “square” (area-based adaptive p, $\tau = 10^{-4}$), or the scheme corresponding to the line with “right triangle” (adaptive τ, $\tau_0 = 10^{-4}$) can be used as a default setting in the code.

3.3 Comparison of the supernodal and the column-wise algorithms

We performed the experiments to compare the supernodal ILU and the column-wise ILU (by setting maximum supernode size to be one). We use both metrics (2) $RowSize(x) \overset{\text{def}}{=} \|x\|_2/\sqrt{m}$ and (3) $RowSize(x) \overset{\text{def}}{=} \|x\|_\infty$ to measure the row size in the dropping rule 1) of Figure 1.

Table 1 compares various performance metrics of the supernodal (S-ILU) versus column-wise (C-ILU) ILU. Each number in the table is the average of the corresponding metric taken over the number of matrices that both versions succeed. As can be seen, using either metric in the dropping rule, S-ILU is faster than C-ILU, although the speed advantage is not great compared to that in a complete factorization. This is mainly due to two reasons: 1) because of the row dropping criterion for supernode, S-ILU has more fill-ins and floating-point operations, and 2) the average supernode size is small after dropping (less than 3 columns on average). The more fill-ins in the supernodal versions translate into many more floating-point operations, especially in the conservative case when ∞-norm is used, S-ILU performs over 3.5 times the operations of C-ILU. The larger are the supernodes, the more fill-ins and operations are incurred in S-ILU. From our experiments we found that we have to set a smaller cutoff for the maximum supernode size (*maxsuper*) than in complete factorization. (That is, when a supernode size exceeds *maxsuper* columns, we will break this supernode into two supernodes.) Otherwise the extra operations would offset the BLAS-3 benefit. For example, in S-ILU $maxsuper = 20$ strikes a good balance in overall speed, whereas in the complete LU, we often use $maxsuper = 100$.

In these tests, we excluded one matrix *cage13*, because the C-ILU factorization time is about 1000x slower than the average, and the inclusion of this matrix would severely skew the statistics of the mean metric.

	Factor construction				*GMRES*			Total time
	fill-ratio	s-node size (columns)	flops (10^9)	Factor time	Iters	One Trisolve time	Iter time	
	$RowSize(x) \overset{\text{def}}{=} \frac{\|x\|_2}{\sqrt{m}}$; this includes 138 matrices that both versions succeeded							
S-ILU	4.2	2.80	7.60	39.69	21.6	0.0092	2.93	42.68
C-ILU	3.7	1	2.65	65.15	20.0	0.0079	2.55	67.75
	$RowSize(x) \overset{\text{def}}{=} \|x\|_\infty$; this includes 134 matrices that both versions succeeded							
S-ILU	4.2	2.72	9.45	54.44	20.5	0.0109	3.40	57.90
C-ILU	3.6	1	2.58	74.10	19.8	0.0090	2.88	77.04

Table 1: Comparison of supernodal ILU (S-ILU, with $maxsuper = 20$) and column-wise ILU (C-ILU, with $maxsuper = 1$) using the "mean" statistics about various metrics. The times are in seconds. $\tau = 10^{-4}, \gamma = 10$.

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, zero 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 the 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:

$$\text{Prob}\{u_{22} = 0\} = \frac{\tau}{2} > 0.$$

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 $(PA)(j : n, j) = 0$ and $(PA)(1 : j-1, 1 : j-1)$ is nonsingular, the probability of encountering a zero pivot in the j-th column would be positive. We can show that $(\tau/2)^{\text{nnz}(A)-n}$ 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

$$\begin{aligned}\prod_{i=1}^{j-1}\prod_{k>i}\text{Prob}\{|B(k,i)| < \tau|B(i,i)|\} &= \prod_{i=1}^{j-1}\left(\frac{\tau}{2}\right)^{\text{nnz}(B(i+1:n,i))} \\ &\geq \left(\frac{\tau}{2}\right)^{\text{nnz}(A(:,1:j-1))-(j-1)} \geq \left(\frac{\tau}{2}\right)^{\text{nnz}(A)-n}.\end{aligned}$$

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

On the other hand, if j is the first column index that encounters zero pivot, the matrix must satisfy the condition that $(PA)(j : n, j) = 0$ and $(PA)(1 : j-1, 1 : j-1)$ is nonsingular.

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 experiment, 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 dropping elements. 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; otherwise, it 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 [16] and the multilevel method [3]. 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 [4], see Section 5.

3.5 Modified ILU (MILU)

The MILU techniques were introduced to reduce the effect of dropping by compensating for the discarded elements [28]. For systems arising from the discretization of second order elliptic PDEs, it is generally admitted that the modified incomplete Cholesky (MIC) factorization is more efficient than the unmodified one [13, 15, 17, 24, 29]. Whereas for general systems, very little is shown either theoretically or empirically whether MILU always helps. Our empirical experience shows that sometimes it can be better than the unmodified ILU.

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-sum relation $P_r Ae = \tilde{L}\tilde{U}e$ for a row-wise algorithm or column-sum 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 $f_{ij} = u_{ij}$, whereas for the lower triangular part $f_{ij} = l_{ij}u_{jj}$ because of division by the diagonal entry f_{jj}.

We have modified the above column-wise MILU procedure to accommodate our supernodal dropping criteria. Recall that in Algorithm 1, we apply the dropping rule 2) in Figure 1 only after 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 the diagonal is not compensated enough. Therefore, the column-sum relation is not preserved.

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

```
(1) Obtain the current filled column $F(:, j)$;
(2) Pivot: Choose pivot row $i$, such that $i = \arg\max_{i \geq j} |f_{ij}|$;  Swap rows $i$ and $j$;
(3) Apply a dropping rule to $F(:, j)$;
(4) Compute the sum of dropped entries in $F(:, j)$: $s = \sum_{\text{dropped}} f_{ij}$;  Set $f_{jj} := f_{jj} + s$;
(5) Separate $U$ and $L$: $U(1 : j, j) := F(1 : j, j)$;   $L(j : n, j) := F(j : n, j)/F(j, j)$;
```

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

```
(1) Obtain the current filled column $F(:, j)$;
(2) Apply dropping rule 1) of Figure 1 to $F(1 : j, j)$, and set $U(1 : j, j) := F(1 : j, j)$;
(3) Compute the sum of the dropped entries in $U(:, j)$: $s = \sum_{\text{dropped}} u_{ij}$;
(4) Pivot: Choose pivot row $i$, such that $i = \arg\max_{i \geq j} |\mathbf{f_{ij}} + \mathbf{s}|$;
            Swap rows $i$ and $j$, and set $u_{jj} := f_{ij} + s$;
(5) IF $j$ starts a new supernode THEN
        Let $(r : t)$ be the newly formed supernode; $(t \equiv j - 1)$
        Apply dropping rule 2) of Figure 1 to the supernode $L(:, r : t)$;
        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;
```

Our supernodal version of the MILU strategy works as follows. For each column of U, we accumulate in s the sum of the dropped entries. Later on, s is not only added to the diagonal but is also used during pivot selection in the lower triangular part. The procedure is outlined in Algorithm 3 and we call it SMILU-1. This algorithm ensures that the pivot has the largest magnitude after the elements are dropped from 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 in a future step when the supernode is formed. The pivot could become small or even zero after we apply the dropping rule to L (i.e., after applying (5) in Algorithm 3.) This may cause the factor U to be ill-conditioned, resulting in an unstable preconditioner. A slightly modified algorithm, which we call SMILU-2, provides a remedy. Here, we take s to be the absolute value of the dropped sum: $s = |\sum_{\text{dropped}} u_{ij}|$, and the diagonal u_{jj} is set to be: $u_{jj} := f_{ij} + \text{sign}(f_{ij})s$. This ensures that the magnitude of the pivot is nondecreasing after diagonal compensation, thereby avoiding small pivots. An alternative method is to accumulate the one-norm of the dropped vector: $s = \sum_{\text{dropped}} |u_{ij}|$. Thus, the pivots would have larger absolute values compared to what would be in SMILU-2, and we expect the condition number of U to be smaller; we call this SMILU-3.

Another twist with MILU is to incorporate diagonal perturbations. In the earlier investigations of MIC (e.g. [24] and the references therein), people found that the unperturbed MIC may lead to unpredictable behavior of the largest eigenvalues of the preconditioned matrix. Different perturbations were introduced which could upper bound the eigenvalues for special problems, such as Stieltjes matrices. One of the best is Notay's DRIC [24], a dynamic, relaxed IC. We adopted this scheme in our MILU variants above. Specifically, the diagonal u_{jj} is modified as follows:

1) Choose $\alpha = n^{-\frac{1}{d}}$ (for discretized PDEs, d can be the dimension of the space.)
2) Set $\alpha_j = s/u_{jj}$, s is computed as in Step (3) of Algorithm 3
3)

$$\omega_j = \begin{cases} \min\left\{\frac{2(1-\alpha)}{\alpha_j}, 1\right\} & \text{if } \alpha_j > 0 \\ \max\left\{\frac{2(1-\alpha)}{\alpha_j}, -1\right\} & \text{otherwise} \end{cases}$$

4) Set $u_{jj} := f_{ij} + s \cdot \omega_j$ in Step (4) of Algorithm 3

Intuitively speaking, this ensures that we do not compensate too much on the diagonal if the sum of the dropped entries is large. Note that Notay provided theoretical arguments that the above choice of ω_j provides upper spectral bound for certain class of matrices, but no theoretical justifications exist for general unsymmetric matrices. We observed only empirically that it improves convergence in some cases.

In Table 2, we compare the performance of various ILU algorithms and direct solver SuperLU. We classify the GMRES convergence history in three categories: *converge*, *slow* (meaning the preset maximum iteration count 500 is exceeded although the residual norm is still decreasing, i.e, $\delta < ||r||_2/||b||_2 < 1$), and *diverge* (meaning $||r||_2 \geq ||b||_2$). The column labeled *memory* means the code runs out of memory. The (unmodified) ILUTP(τ) usually works very well, however, when it fails, it is often due to too many zero pivots. Looking at the *slow* and *diverge* columns, we see that for a number of matrices the MILUs could succeed if they are allowed to continue with more iterations, whereas ILU would fail completely.

Figure 3 shows performance profiles of various ILUTP algorithms and SuperLU. Within a certain time limit, the ILUs can solve many more problems than SuperLU does. The ILUs are also advantageous over SuperLU in terms of fill ratio. SuperLU fails with many problems due to memory exhaustion. From the fill-in point of view, various MILUs are similar, and nearly as good as ILUTP(τ). SMILU-2 is slightly better than the other two MILU schemes. When τ is small, such as $\tau = 10^{-6}$, the difference between the different variants of ILUs is very small, mainly because the number of entries dropped is small. Looking only at the global performance profiles, the MILUs are slightly worse than ILUTP. In general, this is because the MILUs are designed to prevent U from becoming too ill-conditioned by using diagonal perturbations. On the other hand, the perturbations can make the L

		converge	*slow*	*diverge*	*memory*	zero pivots	avg. iterations
$\tau = 10^{-4}$	ILUTP(τ)	142	60	28	2	2216	21
	SMILU-1	131	69	30	2	589	27
	SMILU-2	137	70	22	3	222	28
	SMILU-3	135	73	22	2	219	40
$\tau = 10^{-6}$	ILUTP(τ)	133	51	46	2	1737	35
	SMILU-1	125	72	33	2	1058	34
	SMILU-2	127	71	31	3	296	30
	SMILU-3	129	73	28	2	289	33
($\tau = 0.0$)	SuperLU	222	0	0	10	0	1

Table 2: Comparison of various (M)ILUTP(τ) algorithms and SuperLU. The column “zero pivots” indicates the number of zero pivots encountered during ILU factorization. The secondary dropping is turned off.

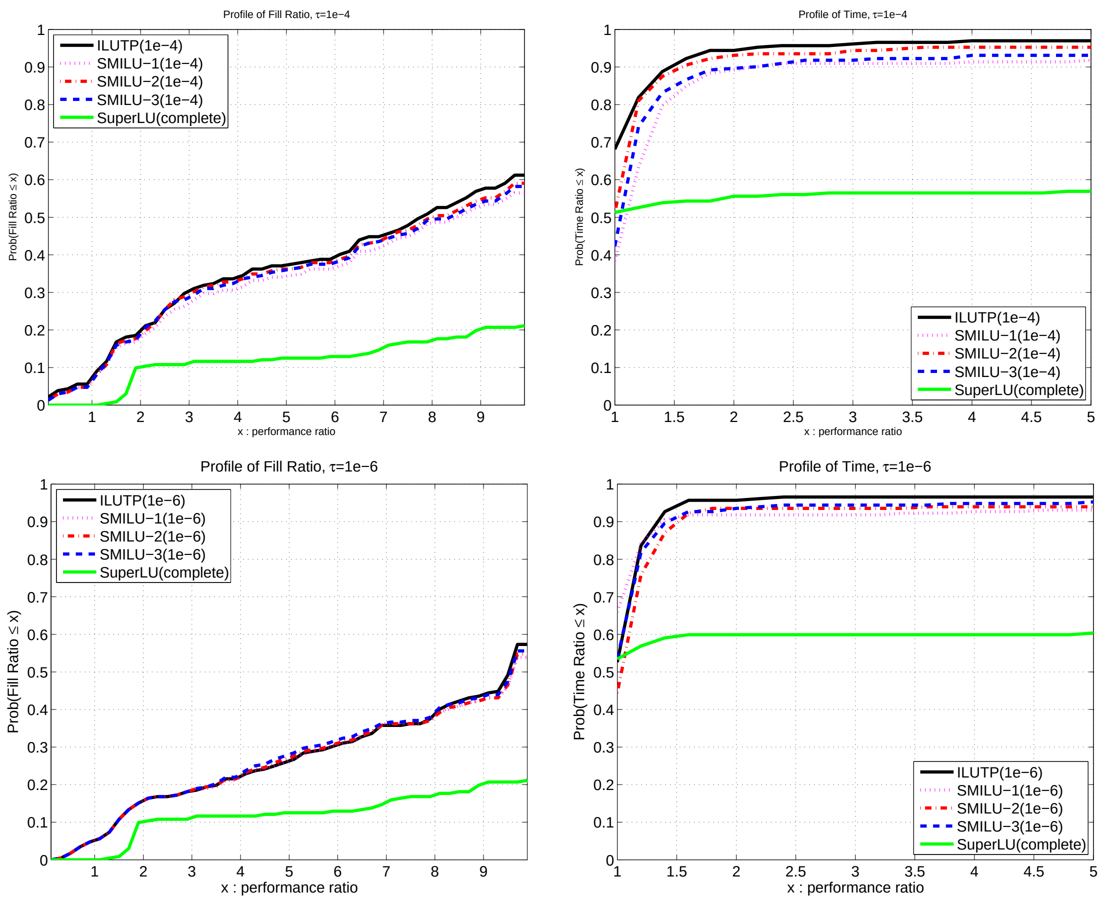


Figure 3: Performance profiles of the unmodified ILU and the MILU algorithms with $\tau = 10^{-4}$ or 10^{-6}. The left column is the profile w.r.t. the fill ratio, and the right column is the profile w.r.t. the time ratio.

and U factors can be far from PA (i.e., less accurate [2]), which could result in a poor preconditioner. Nonetheless, MILU can be useful for some matrices when the pure ILU fails due to too many zero pivots, as shown in Table 2. Therefore, we still see the merit of MILU being available as an option even though we do not recommend it to be used as a default in the software.

3.6 Threshold partial pivoting

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 that gives preference to the diagonal entries. We use a threshold parameter $\eta \in [0, 1]$ to facilitate this. If $|f_{jj}| \geq \eta \max_{i \geq j}\{|f_{ij}|\}$, the diagonal entry f_{jj} is used 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 3 and 4 present the numerical results when varying the pivot tolerance η, without and with secondary dropping, respectively.

When secondary dropping is not used (Table 3), (threshold) pivoting is more reliable than no pivoting at all, because general matrices are not close to being diagonally dominant.

When secondary dropping is used (Table 4), 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 the average fill ratios are usually less than one third of those in Table 3, the numbers of problems successfully solved are much smaller, and the iteration counts are much larger.

Based on our experience, it is better not to use secondary dropping when memory is not at a premium. As a comparison, for complete factorization with $\eta = 0.1$, the average fill ratio is 90.7 and the maximum fill ratio is 1577.2.

	Diag_thresh (η)	1.0	0.1	0.01	0.001	0
$\tau = 10^{-4}$	Number of successes	199	203	201	201	191
	Average fill ratio	17.3	17.0	15.3	15.6	10.0
	Maximum fill ratio	371	312	98	100	92
	Average iterations	7.8	11.7	11.5	15.4	14.7
$\tau = 10^{-6}$	Number of successes	209	209	208	205	203
	Average fill ratio	34.5	34.7	33.0	31.5	28.6
	Maximum fill ratio	1200	1200	1200	1200	1200
	Average iterations	4.1	3.7	3.8	3.3	6.6

Table 3: Effect of Diag_thresh (η) with ILUTP(τ).

4 Comments on the Software

In this section, we describe a few implementation difficulties encountered while developing incomplete factorization, and summarize the input parameters introduced to the ILU routine.

	Diag_thresh (η)	1.0	0.1	0.01	0.001	0
$\tau = 10^{-4}$	Number of successes	134	142	142	141	148
	Average fill ratio	4.2	4.2	4.2	4.2	4.0
	Maximum fill ratio	9.6	9.6	9.6	9.6	9.6
	Average iterations	15	21	22	23	26
$\tau = 10^{-6}$	Number of successes	124	133	133	131	132
	Average fill ratio	5.2	5.4	5.4	5.3	5.4
	Maximum fill ratio	9.6	9.6	9.6	9.6	9.6
	Average iterations	29	35	36	31	28

Table 4: Effect of Diag_thresh (η) with ILUTP(τ, p), using area-based secondary dropping, $\gamma = 10$.

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 factorization 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 $t < \min(r, j)$ [9]. 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:[b]

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

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

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

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 "$\otimes$" indicates a numerical dropping in ILU.

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

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

Options	Default
drop tolerance (τ)	10^{-4}
$RowSize()$	∞-norm
secondary dropping	area-based adaptive-p
fill-ratio bound (γ)	10
SMILU	No
MC64	ON
equilibration	ON
column permutation	COLAMD [7]
diag_thresh (η)	0.1

Table 5: Default values of the parameters of the ILU routine `xGSITRF`.

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 runtime 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). We group several columns at the bottom of the column elimination tree into an artificial supernode [9]. 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`. It 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 listed in Table 5, 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 adjust the parameters if the defaults do not work:

- Equilibration is necessary, and MC64 is usually helpful.
- If there are many zero pivots and the preconditioner is too ill-conditioned, you could try the modified variants SMILU-2 or SMILU-3.
- 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 the Other Preconditioners

A number of preconditioning packages using incomplete factorization algorithms were developed for unsymmetric matrices. In this section, we compare the performance of our new ILU preconditioner with SPARSKIT [26], ILUPACK [4] and ParaSails [5], which are representatives of a wide algorithmic spectrum. These preconditioners are used in the standard GMRES iterative solver.

The algorithm in SPARSKIT is the original ILUTP algorithm proposed by Saad, and is the closest to our algorithm. The primary differences are: 1) SPARSKIT performs factorization row by row and does not use supernode, whereas ours is a column-wise algorithm and exploits supernode; 2) The secondary dropping in SPARSKIT is row-based and p is a fixed constant, whereas ours is area-based with adaptive p.

ILUPACK uses a very different approach to ours and SPARSKIT; it is an inverse-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 conditioner number of the preconditioner is under control. This objective is achieved indirectly: 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 the suitable pivots are chosen, the current level is considered to be complete, and the factorization starts a new level, which is comprised of all the delayed rows and columns from the previous level.

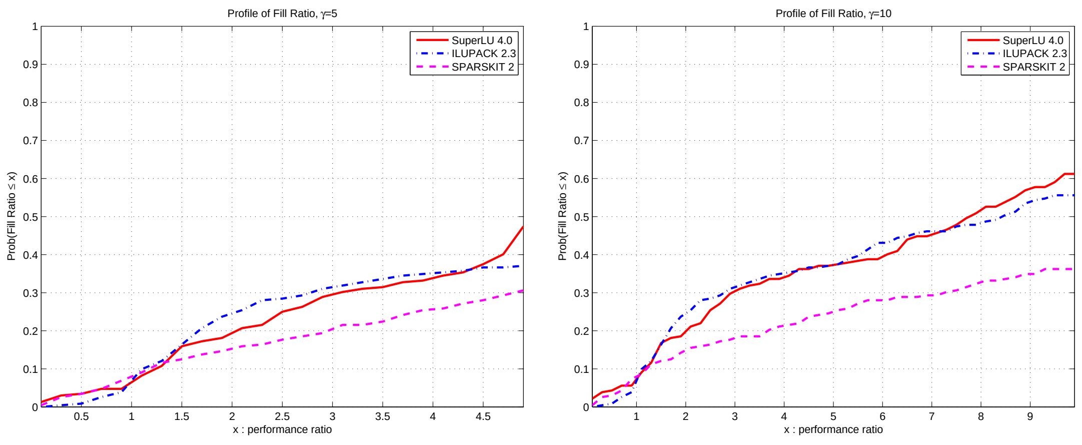
ParaSails is a parallel implementation of a sparse approximate inverse preconditioner, which aims at finding an M with a prescribed sparsity pattern and minimize $||I - AM||_F^2$ [5]. (We only use one processor in our test.) The use of Frobenius norm permits decoupling the minimization problem into n small independent least-squares problems. ParaSails uses the patterns of powers of sparsified matrices as the approximate inverse patterns. The algorithm involves two preprocessing steps: 1) sparsification of A to $\widetilde{A}$ controlled by parameter $0 \leq thresh \leq 1.0$, and 2) obtain the pattern of $\widetilde{A}^{nlevels}$ as M's pattern. Lower values of *thresh* (fewer elements are dropped) and higher values of *nlevels* (keep high level of neighbors) usually result in more accurate but more expensive preconditioners. With the default settings: $thresh = 0.1$ and $nlevels = 1$, only 39 matrices converged with the preconditioned GMRES. The construction of the preconditioner M succeeded with 211 matrices. It took over 63 hours for the solver to complete with those 211 matrices (either converged or exhaustion of 500 iterations), in which 62 hours is for preconditioner construction. We tried to decrease the value of *thresh* and increase the value of *nlevels*, which helps GMRES to succeed with a few more matrices, but the preconditioner construction time becomes prohibitively long. So ParaSails is not competitive with the other preconditioners we are evaluating.

In our comparisons, we use the latest versions of SPARSKIT (version 2.0) and ILUPACK (version 2.3). We use "`pathf90 -O3 -fPIC`" to compile SPARSKIT which is in Fortran, "`pathcc -O3`

`-fPIC`” to compile SuperLU and ILUPACK which are in C, and linked with the AMD Core Math Library (ACML) for the BLAS routines. In our experiments, we try to keep the similar parameter settings for all three codes:

- SuperLU 4.0: $\tau = 10^{-4}$, area-based secondary dropping with $\gamma = 5$ or 10, diagonal threshold $\eta = 0.1$;
- SPARSKIT: $\tau = 10^{-4}$, secondary dropping with $p = \gamma \cdot nnz(A)/n(A)$, where $\gamma = 5$ or 10, diagonal threshold $\eta = 0.1$;
- ILUPACK: $\tau = 10^{-4}$, $\nu = 5$, $\gamma = 5$ or 10 (corresp. to “`param.elbow`” in the code.)

We use the default sparsity reordering options, which are different among the three codes due to different numerical pivoting strategies. our ILU performs partial pivoting with row swappings and uses a column reordering method such as Column Approximate Minimum Degree [7], for which the underlying graph model is the adjacency graph of $|A|^T|A|$. This attempts to minimize an upper bound on the fill *regardless of row interchanges*. ILUPACK does not perform pivoting and uses a symmetric reordering method such as Approximate Minimum Degree [1], for which the underlying graph model is the adjacency graph of $|A|^T + |A|$. SPARSKIT performs partial pivoting with row interchanges, but does not perform column exchanges.

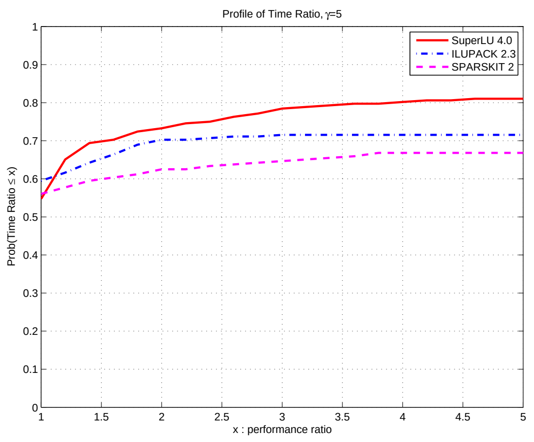


(a) $\gamma = 5$ (b) $\gamma = 10$

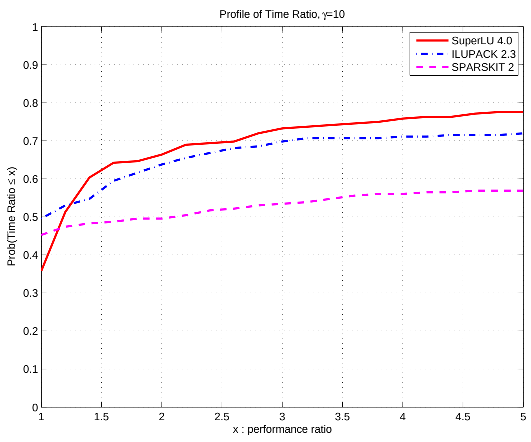
Figure 4: Comparison of fill ratio between our supernodal ILUTP, ILUPACK and SPARSKIT.

Figure 4(a) shows the fill profiles of the three preconditioners with $\gamma = 5$ in the secondary dropping. For smaller allowable fill ratio, ILUPACK could solve a few more problems than our ILU. However, when the fill ratio is close to the prescribed limit γ, our code can solve more problems. Both ours and ILUPACK are better than SPARSKIT. For a larger γ, the curves of profile are changed, see Figure 4(b). The left plots are different than just cutting off at $\gamma = 5$ in the right figure.

Figure 5 compares the runtime of the GMRES solver using the three preconditioners. This shows that our area-based adaptive ILUTP(τ, p) is generally superior to the other two preconditioners. Only at the leftmost part of Figure 5(b), when allowing the similar amount of time, ILUPACK and SPARSKIT are slightly better.

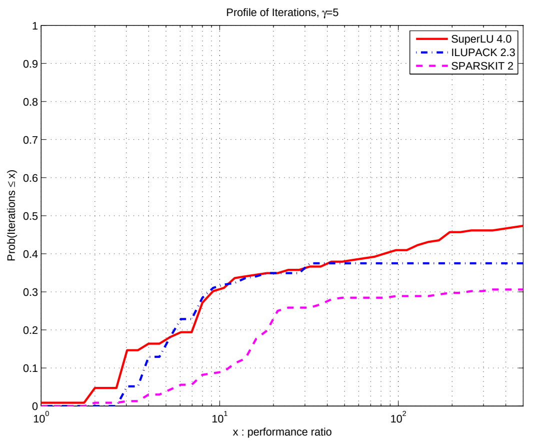


(a) $\gamma = 5$

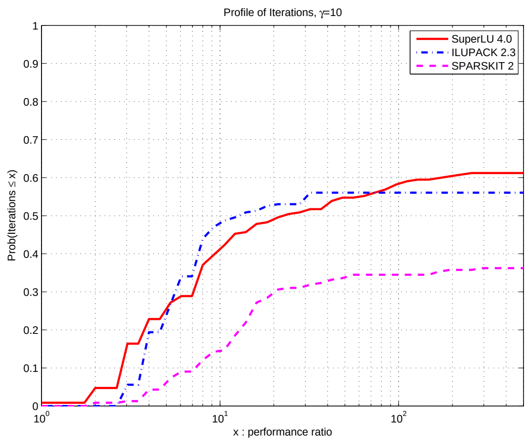


(b) $\gamma = 10$

Figure 5: Comparison of runtime of the GMRES solver using the three preconditioners.



(a) $\gamma = 5$



(b) $\gamma = 10$

Figure 6: Comparison of iteration count of the GMRES solver using the three preconditioners.

Figure 6 compares the number of iterations of the GMRES solver using the three preconditioners. This is very much correlated to the runtime profiles shown in Figure 5. For $\gamma = 5$, our ILUTP preconditioner and ILUPACK can solve almost the same number of problems within certain iterations, and can solve more than SPARSKIT does.

Looking only at the performance profiles in Figures 4–6, our code and ILUPACK seem to be comparable: ours is slightly faster and ILUPACK maintains slightly lower fill. On the other hand, none of the preconditioners can succeed with all the problems, even though they are designed as general-purpose preconditioners. This is different from direct methods, which almost always succeed as long as there is enough memory. Therefore, in addition to performance profiles, we need to examine preconditioners from different perspectives. Among the 232 problems, our code succeeds with 142 problems, and ILUPACK succeeds with 130 problems. Both codes succeed with 100 problems. This shows that even though the two preconditioners have similar success rate, they succeed with different sets of problems, and so the two methods can be considered complimentary to one another, and both have practical values in existence.

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 the inverse-based multilevel ILU method implemented in ILUPACK. The new ILU routine is already released to public in SuperLU Version 4.0, which can be downloaded at `http://crd.lbl.gov/~xiaoye/SuperLU/`.

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

Acknowledgements

We owe a great deal to the anonymous referees whose careful reading of the initial manuscript and valuable comments have helped us improve the presentation greatly.

References

[1] P. R. Amestoy, T. A. Davis, and Iain S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Analysis and Applications*, 17(4):886–905, 1996. Also University of Florida TR-94-039.

[2] Michele Benzi. Preconditioning techniques for large linear systems: A survey. *J. Comp. Phys.*, 182:418–477, 2002.

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

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

[5] Edmond Chao. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM J. Scientific Computing*, 21(5):1804–1822, 2000.

[6] Edmond Chow and Michael A. Heroux. An object-oriented framework for block preconditioning. *ACM Trans. Mathematical Software*, 24:159–183, 1998.

[7] T. A. Davis, J. R. Gilbert, S. Larimore, and E. Ng. A column approximate minimum degree ordering algorithm. *ACM Trans. Mathematical Software*, 30(3):353–376, 2004.

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

[9] 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.

[10] 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.

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

[12] 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.

[13] T. Dupont, R. P. Kendall, and Jr. H. H. Rachford. An approximate factorization procedure for solving self-adjoint elliptic difference equations. *SIAM Journal on Numerical Analysis*, 5:559–573, 1968.

[14] 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.

[15] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*. SIAM, Philadelphia, PA, 1997.

[16] 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.

[17] I. Gustaffson. A class of first order factorization methods. *BIT*, 18:142–156, 1978.

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

[19] HSL. `http://www.cse.scitech.ac.uk/nag/hsl/hsl.shtml`, October 2007.

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

[21] 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.

[22] X.S. Li, M. Shao, I. Yamazaki, and E.G. Ng. Factorization-based sparse solvers and preconditioners. In *Proceedings of SciDAC 2009 Conference, Journal of Physics: Conference Series 180 (2009) 012015. Institute of Physics Publishing.*, San Diego, June 14-18 2009.

[23] Esmond G. Ng, Barry W. Peyton, and Padma Raghavan. A blocked incomplete cholesky preconditioner for hierarchical-memory computers. In David R. Kincard and Anne C. Elster, editors, *Proceedings of Iterative Methods in Scientific Computations IV*, pages 211–222. IMACS, 1999.

[24] Y. Notay. DRIC: a dynamic version of the RIC method. *Numerical Linear Algebra with Applications*, 1(6):511–532, 1994.

[25] Markus Olschowka and Arnold Neumaier. A new pivoting strategy for Gaussian elimination. *Linear Algebra and its Applications*, 240:131–151, 1996.

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

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

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

[29] H. van der Vorst. The convergence behaviour of preconditioned CG and CGS in the presence of rounding errors. In O. Axelsson and L.Y. Kolotilina, editors, *Preconditioned conjugate gradient methods*. vol. 1457, Lecture notes in Math., Springer Verlag, 1990.