ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439-4801

AN IMPROVED INCOMPLETE CHOLESKY FACTORIZATION*

Mark T. Jones and Paul E. Plassmann

Mathematics and Computer Science Division
Preprint MCS-P206-0191
January 1991
(Revised July 1992)

ABSTRACT

Incomplete factorization has been shown to be a good preconditioner for the conjugate gradient method on a wide variety of problems. It is well known that allowing some fill-in during the incomplete factorization can significantly reduce the number of iterations needed for convergence. Allowing fill-in, however, increases the time for the factorization and for the triangular system solves. In addition, it is difficult to predict *a priori* how much fill-in to allow and how to allow it. The unpredictability of the required storage/work and the unknown benefits of the additional fill-in make such strategies impractical to use in many situations. In this paper we motivate, and then present, two "black-box" strategies that significantly increase the effectiveness of incomplete Cholesky factorization as a preconditioner. These strategies require no parameters from the user and do not increase the cost of the triangular system solves. Efficient implementations for these algorithms are described. These algorithms are shown to be successful for a variety of problems from the Harwell-Boeing sparse matrix collection.

* This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

1. Introduction. One of the most successful iterative methods for solving large sparse symmetric positive definite linear systems is the preconditioned conjugate gradient method [7]. The incomplete Cholesky factorization, proposed by Meijerink and van der Vorst [12] for use on symmetric M-matrices, has been shown to be a very effective preconditioner for a wide variety of problems. However, the incomplete Cholesky factorization can fail for symmetric positive definite matrices [10]. To alleviate this problem, Manteuffel [11] introduced the shifted incomplete factorization algorithm.

No-fill incomplete Cholesky factorization computes incomplete factors, $\hat{L}\hat{L}^T$, of the original matrix A, where $\hat{L}$ has nonzeros only in those positions that correspond to nonzero positions in A [12]. To improve the quality of the preconditioner, many strategies for altering the pattern of the nonzeros in the incomplete factor have been proposed. For the purposes of this paper, we classify these strategies as either fixed fill-in or drop-tolerance strategies. We discuss these strategies in more detail in Section 2. Unfortunately, these strategies have the disadvantage that they are not suitable for a "black-box" implementation; it is difficult to *a priori* choose the effective parameters and/or the required storage.

In this paper we propose an approach that incorporates aspects of both of these strategies. Our approach allows a fixed number of nonzeros in each row or column; the number of nonzeros kept in the factor is the number of nonzeros that were originally in that row or column. However, the original sparsity pattern is ignored, and only the nonzeros with the largest magnitude are retained. In this way, the space required for factorization is fixed, yet by keeping the nonzeros largest in magnitude, we can capture some of the benefits of the drop-tolerance methods without having to choose a drop-tolerance level. Row and column versions of this algorithm, along with efficient implementations, are described in Sections 3 and 4.

The major advantage of our approach is that it is a "black-box" strategy. It is evident that this strategy will not always result in less overall work than other strategies for incomplete factorization. If one is able to pick the correct level of fill-in or a good drop-tolerance, it is likely that some other strategy will outperform the strategy proposed here. However, for general sparse matrices the selection of these parameters is very difficult.

In this paper we give the motivation, description, and demonstration of efficient software for new methods without this drawback. We show that the row and column algorithms mentioned above are black-box, low-risk algorithms that can be implemented in a very efficient manner. We present experimental results that show that these methods should be preferred to the no-fill incomplete Cholesky factorization algorithm when factoring matrices of moderate to high condition number. In Section 2 we review a number of previously proposed techniques for incomplete factorization. We motivate and describe the two new algorithms in Section 3. In Section 4 we discuss the efficient implementation of these algorithms. We give experimental results for the algorithms in Section 5. In Section 6 we summarize and draw conclusions.

2. Other Techniques. Fixed fill-in strategies for incomplete factorization fix the non-zero pattern of the incomplete factor prior to the numeric factorization phase. The pattern may be determined by a symbolic factorization step or may be known *a priori* if the matrix has a very regular structure. For example, fixed fill-in strategies for naturally-ordered matrices arising from finite-difference stencils have been proposed that allow extra diagonals of fill-in [12, 13]. Gustafsson has proposed modifications to the methods in [12] for which has he proved have better convergence than the standard incomplete factorizations [6]. This concept of extra diagonals is only useful within the context of naturally-ordered matrices arising from finite-difference stencils. For general sparse matrices, the extra diagonal concept can be generalized to the concept of "levels" of fill-in. For example, level 1 fill-in would allow nonzeros corresponding to the original matrix nonzeros and any nonzeros directly introduced by the elimination of original nonzeros. Level 2 fill-in would include the level 1 fill and any nonzeros directly introduced by the elimination of the level 1 nonzeros. For general sparse matrices, this method requires a symbolic factorization step to determine the position of nonzeros and the amount of storage required.

An alternative to fixed fill-in strategies is a drop-tolerance strategy. In these strategies nonzeros are included in the incomplete factor if they are larger than some threshold tolerance. A simple drop-tolerance strategy is to set the tolerance for element a_{ij} to be $c\sqrt{a_{ii}a_{jj}}$, where c is a constant [14]. Neither the pattern or number of nonzeros in the incomplete factor can be determined *a priori* for such a strategy; they are determined during the factorization. More complex strategies where the tolerance varies during the factorization according to the amount of fill-in included at each stage of the factorization have been proposed [1, 14]. These more complex strategies keep the storage required by the incomplete factor within the fixed amount of storage allocated by the user. The number of nonzeros in the factor at step k is compared with the amount of space that the user has allocated for the incomplete factor. If the number of nonzeros in the factor at a particular step does not match the number predicted by a proposed formula, then the tolerance threshold is altered. The only parameters that must be chosen, aside from the initial c, are the formula for altering c and the amount of storage to allocate for the incomplete factor.

Fixed fill-in and drop-tolerance strategies were evaluated on grid problems with with five-point and nine-point stencils by Duff and Meurant [3]. They compared a level 1 fill-in algorithm to a no-fill incomplete Cholesky factorization and found that the extra work in the factorization and forward/back substitutions was greater than the work saved by the reduced number (if any) of iterations. A drop-tolerance strategy was also evaluated and was found to be cost-effective if the right drop-tolerance was selected, but was found to be more difficult from an implementation standpoint.

Freund and Nachtigal [4] have independently developed an algorithm that is similar to our proposed algorithm. Their algorithm combines drop-tolerances with a cap on the number of nonzeros that can be kept per row. For example, if

the drop-tolerance determines that 30 elements are to be kept, but the cap is 20, then only the 20 largest elements are kept. This algorithm is not a black-box because of the tolerance setting, but most likely it is not as sensitive to the tolerance setting as a drop-tolerance algorithm alone. The drop-tolerance can be viewed as a means for saving work if it is set at the proper level. If the drop-tolerance is set to zero and the number of nonzeros per row were set to be the number of nonzeros per row in the original matrix, the Freund and Nachtigal algorithm is similar to our algorithm. However, it differs in two significant aspects. First, their strategy does not preserve symmetry. Second, our implementation does not need to check drop-tolerances and therefore should be more efficient.

In summary, each one of these modifications to the no-fill algorithm requires extra work during the factorization phase. It is not possible to predict *a priori* the amount of extra work required by the factorization phase for general sparse matrices. Each strategy differs in the amount of storage required, the amount of work in the forward/backward substitution steps, and the sensitivity of the method to parameter settings. In Table 1 we have summarized these three aspects of the algorithms discussed above. We note that the storage requirement given in the table is determined assuming that the original matrix is already being stored in a sparse format. We denote the order of the original matrix by n and by nz the number of nonzero elements in the strict lower triangle of the matrix. An efficient implementation of the factorization may require additional low-order storage over the amount given in the table.

3. New Algorithms. First, we provide the motivation for the row and column factorization strategies by discussing incomplete factorization from a different viewpoint from that normally used. In this paper all norms are assumed to be the two-norm. Given the symmetric positive definite matrix A, let $A = LL^T$ be the complete Cholesky factorization and $A = \hat{L}\hat{L}^T - R$ be an incomplete factorization. The size and structure of R have typically been examined for insights into the goodness of the preconditioner [3]. Ultimately, however, the goal is to reduce the condition number of the matrix $B = \hat{L}^{-1}A\hat{L}^{-T}$ [9]. If we define $E = L - \hat{L}$ and $X = \hat{L}^{-1}E$, then B can be expressed as perturbation of the identity in terms of the lower triangular matrix X,

$$B = I + X + X^T + XX^T \ . \tag{1}$$

Thus, as $\|X\|$ is reduced, B approaches the identity. We can assume that A is scaled so that its diagonal is the identity matrix and that any shift used in the incomplete factorization is positive. Therefore, because element (1,1) of $\hat{L}$ is ≥ 1, $\|\hat{L}\| \geq 1$. Thus we may bound the norm of X by

$$\|X\| \leq \kappa(\hat{L})\|E\| \ , \tag{2}$$

where $\kappa(\hat{L})$ is the condition number of $\hat{L}$. This bound on $\|X\|$ is not tight, however, we are primarily interested in the relationship that as $\|E\|$ decreases

TABLE 1
Comparison of Algorithms for General Sparse Symmetric Matrices.

No-fill Algorithm

Storage: Only nz reals are required; the row and column positions are the same as those in the original matrix.

Forward/Back Solves: $n + 2 * nz$ floating-point additions and multiplications are required.

Parameters: no parameters; a black box.

Drop Tolerance

Storage: The size of the factor is dependent on the tolerance setting; it can be more or less than that required by the original matrix. The entire sparse data structure must be provided, because the rows and columns are no longer those of the original matrix.

Forward/Back Solves: The amount of work is dependent on the tolerance strategy.

Parameters: The amount of work and the size and effectiveness of the factors are completely dependent on the tolerance strategy.

Level k Fill

Storage: Requires more storage than the original matrix; this amount can be determined by a preprocessing step. The entire sparse data structure must be provided, because the rows and columns are no longer those of the original matrix.

Forward/Back Solves: The amount of work is larger than that required by the no-fill algorithm.

Parameters: The level of fill, k, must be selected; however, the amount of storage cannot be predicted *a priori*; therefore, a preprocessing step is necessary.

New Row and Column Algorithms

Storage: Nonzero storage and column (or row) indices for a sparse matrix of the same size as A must be provided. Pointers to the column (or row) indices are the same as those of the original matrix.

Forward/Back Solves: Same as for no-fill algorithm.

Parameters: No parameters; a black box.

then $\|X\|$ decreases with a corresponding decrease in $\kappa(B)$. This behavior can be observed in the experiments in Section 5.

This relationship between $\|E\|$, $\|X\|$, and $\kappa(B)$ provides the motivation for our method. By reducing the norm of X, we expect to reduce the condition number of B. Unfortunately, since we are considering incomplete factorizations, the size of $\|E\|$ is at least as large as the largest element of L not included in the structure of $\hat{L}$. Equation (2) also presents a different perspective as to why including a shift is important, namely, to reduce the size of the condition number of $\hat{L}$. We note that this is counterintuitive, since for ill-conditioned problems one might assume that a good preconditioner has the property $\kappa(\hat{L}) \approx \kappa(L)$. On the other hand, when $\hat{L}$ is positive definite, shifting the diagonal usually increases $\|E\|$. Thus, these two goals, seeking to minimize the condition number of $\hat{L}$ and the reduction of $\|E\|$, can be conflicting.

Our approach is to attempt to minimize the norm of E, while keeping the incomplete factorization as stable as possible, and thus decrease the condition number of $\hat{L}$. In the row algorithm, we use a Crout-Doolittle factorization: during step k, row k of $\hat{L}$ is updated by the first $k-1$ rows of $\hat{L}$. At step k, row k is stored in a dense vector, and fill-in is allowed. If there were originally m nonzero off-diagonal elements in row k, the largest m elements (in absolute value) in the dense vector are returned to the storage allocated for $\hat{L}$. The diagonal is updated, and the remainder of the dense vector is discarded.

The column strategy is similar. At step k of the factorization, the off-diagonal of column k of $\hat{L}$ is updated by the first $k-1$ columns of $\hat{L}$. As before, column k is stored in a dense vector, and fill-in is allowed. After the updates have taken place, the nonzero elements in the dense vector are used to update the last $n-k$ diagonal elements of $\hat{L}$. Again, if m nonzero off-diagonal elements were originally in column k, the largest m elements are incorporated into the sparse $\hat{L}$ matrix.

Computationally, at least one of these two strategies was always found to require significantly fewer iterations for convergence than the standard shifted preconditioner. However, our results show that the superiority of one of the two strategies varied. We have also considered a variation of the second strategy that kept the off-diagonals in the same positions as in A, but still allowed the elements in the dense vector to update the remaining diagonals. This strategy was sometimes much better, but not consistently or dramatically better than incomplete factorization with no fill-in.

Aside from being better preconditioners, these two strategies often allow for a more stable incomplete factorization. If we use shifted incomplete factorization [11] to form a positive definite preconditioner, the shift required to make the factorization positive definite is often smaller for our two strategies than it is for incomplete factorization without fill-in. Therefore, if one uses a strategy of performing the factorization with increasing shifts until the factorization is positive definite, then much time can be saved with our strategies, because an acceptable factorization will be found much sooner.

4. Efficient Implementation. The efficient implementation of the column-oriented approach is simple and is therefore described first. First, we give a description of the data structure used to store the matrix, A. The nonzeros of A are stored in four arrays: (1) *diag*, which stores the n diagonal elements; (2) *nonz*, which stores the off-diagonal nonzeros in column-major order; (3) ja, where $ja(i)$ is the row value for element i in *nonz*; and (4) ia, where $ia(j)$ points to the beginning of column j in *nonz* and ja. The first concern in designing the algorithm is to ensure that $O(n^2)$ overhead operations are not necessary. To achieve this, we use two n-length arrays: *first*, where *first*(j) points to the position in *nonz* and ja of the next entry in column j to be used during the factorization, and *list*, where *list*(j) points to a linked list of columns that will update column j. Two additional n-length arrays are used to allow for fill-in at step j of the factorization: *tcolumn*, which stores the nonzero values in the current column, and *trow*, which stores the row values for the current column. The column-based algorithm is shown in Figure 1.

```
1)  DO j = 1, ..., n
2)      Load nonzeros from column j in nonz into tcolumn;
3)      Load row values from column j in ja into trow;
4)      ptr = list(j);
5)      WHILE (ptr ≠ 0) DO
6)              Update tcolumn using column ptr starting at
7)                  position first(ptr) in nonz;
8)              first(ptr) = first(ptr) + 1; {advance to next row in column}
9)              IF (first(ptr) < ia(ptr + 1)) THEN
10)                 tptr = ptr;
11)                 ptr = list(ptr); {move ptr to the next column in list}
12)                 list(tptr) = list(ja(first(tptr))); {add this column to a new list}
13)                 list(ja(first(tptr))) = tptr;
14)             ENDIF
15)     ENDWHILE
16)     Divide the elements in the tcolumn by diag(j);
17)     Update the remaining diagonals with tcolumn;
18)     Copy the largest ia(j + 1) − ia(j) values in tcolumn into nonz and ja;
19) ENDDO
```

FIG. 1. *The column-based algorithm.*

A similarly efficient implementation can be constructed for the no-fill incomplete factorization algorithm, but in this case the *trow* variable is not needed, and ja is not modified.

To construct a row-based implementation of the Crout-Doolittle algorithm that meets our goal of not taking $O(n^2)$ overhead operations is more challenging and requires more storage. The same data structure used in the column-based

```
1)  DO j = 1, ..., n
2)       Load nonzeros from row j in nonz into trow;
3)       Load column values from row j in ja into tcolumn;
4)       k = the column number of the first nonzero in row j;
5)       WHILE (k ≠ j) DO
6)            trow(k) = trow(k)/diag(k);
7)            diag(j) = diag(j) − trow(k) * trow(k);
8)            use column k to update trow; {find column k by traversing
9)              the list beginning at cptr(k)}
10)           k is the column number of the next nonzero in row j;
11)      ENDWHILE
12)      Copy the largest ia(j + 1) − ia(j) values in trow into nonz and ja;
13)      Add this row to the column linked list structure;
14) ENDDO
```

FIG. 2. *The row-based algorithm.*

algorithm is used to store A in row-major form. In addition, *trow* and *tcolumn* arrays are used to store the current row. However, for the sake of efficiency, linked lists must be constructed that, at step j, store the columns of A up to row $j-1$. Three arrays are required: *cptr*, where $cptr(j)$ points to the beginning of column j; *next*, where $next(i)$ points to the element following the element in position i,; and *rval*, where $rval(i)$ is the row value of the element in position i. The last two arrays are the same length as *nonz*, a sizable amount of storage. Additional efficiency can be gained by using a vector of length n to quickly determine if a nonzero exists in specific position of the current row. The row-based algorithm is shown in Figure 2.

5. Experimental Results. In this section, the strategies given in the preceding section are compared to incomplete factorization without fill-in. The matrices used for the comparisons are described in Table 2. The first five problems were generated from finite element models from the Computational Structural Mechanics Branch at NASA Langley Research Center. The last five problems are from the Harwell-Boeing Sparse Matrix Collection [2].

A first set of experiments was run using the environment provided by the CLAM package [5]. A second set of experiments were run using a Fortran program on a Sun Sparcstation. For these experiments, each matrix was scaled so that its diagonal was the identity and the right-hand side was chosen to be the normalized vector of ones. The initial guess for the problems was the zero vector. The solutions were sought to a relative accuracy of 0.001, where the relative accuracy at step k is defined as $\|r_k\|_2/\|r_0\|_2$ and r_k is the residual at step k. When the incomplete Cholesky factorization failed, 0.01 was added to the diagonal until

[1] The PLANE problem was too large for our system to determine the condition number.

TABLE 2
Description of Problems.

Name	Size	$nz(L)$	$\kappa(A)$	Description
PLATE	327	8474	1.1E7	finite element model of a plate using 4-node elements
PLANE	2141	30350	* [1]	finite element model of an airplane using various 2-D element types
CUBE	180	4301	4.7E6	finite element model of a cube using 8-node elements
CYL	216	3347	1.3E6	finite element model of a circular cylindrical shell using 4-node elements
PANEL	477	5007	8.6E4	finite element model of a blade-stiffened panel with a discontinuous stiffener using 4-node elements
BCSSTK08	1074	7017	2.6E7	TV studio
BCSSTK09	1083	9760	9.5E3	Square plate clamped
BCSSTK10	1086	11578	5.2E5	Buckling of a hot washer
BCSSTK11	1473	17857	2.2E8	Ore car
BCSSTK19	817	3835	1.3E11	Part of a suspension bridge

the factorization succeeded.

In the first set of experiments, the effect of the strategies on the E, X, and B matrices, as well as the number of iterations, was of interest. The results for the first set of experiments are given in Table 3. All norms shown are 2-norms, $\kappa(A)$ is the condition of A, η is the number of nonzeros that have shifted positions in $\hat{L}$, and α is the amount that the diagonal must be multiplied by to force a positive-definite preconditioner. We note that the row and column strategies are always superior to the standard factorization. The column-based strategy always requires less work than the row-based. However, the improvement of the column-based algorithm over the standard approach, in terms of the number of iterations, is less consistent than the row-based strategy. The column FLOPS gives the number of floating point operations used during the incomplete factorization step. We point out that the number of floating-point operations required per iteration is approximately four times the number of nonzeros in the original matrix; this number is identical for each strategy. It is clear from these results that the reduction of a few iterations in the conjugate gradient algorithm can offset the additional factorization costs. Thus, many floating-point operations can be saved by using these strategies.

In the second set of experiments, the primary concern was the amount of time required by efficient implementations of the strategies. To determine whether the superiority of the column and row algorithms was independent of how the equations were ordered, we studied the effect of reordering the matrices by the mini-

TABLE 3
Comparison of Strategies.

Name	Strategy	# of its.	$\|E\|$	$\|X\|$	$\kappa(B)$	FLOPS	η	α
PLATE	Standard	510	2.3	1.6	2.3E6	6.48E4	0	1.37
PLATE	Row	141	1.0	1.2	8.8E4	2.36E5	1946	1.02
PLATE	Column	337	2.0	3.3	7.3E6	9.59E4	1992	1.19
PLANE	Standard	805	9.9^F	*	*	3.80E5	0	1.04
PLANE	Row	566	7.6^F	*	*	1.38E6	6286	1.02
PLANE	Column	436	6.7^F	*	*	1.05E6	6576	1.01
CUBE	Standard	125	0.6	1.3	1.2E4	9.27E4	0	1.01
CUBE	Row	89	0.4	1.2	8.8E3	2.38E5	1168	1.01
CUBE	Column	86	0.4	1.2	8.5E3	1.39E5	1138	1.01
CYL	Standard	104	1.7	3.0	2.3E4	4.19E4	0	1.32
CYL	Row	62	1.3	2.3	7.2E3	1.33E5	708	1.14
CYL	Column	28	0.6	1.1	2.8E2	6.80E4	681	1.02
PANEL	Standard	49	1.1	1.3	637	4.74E4	0	1.02
PANEL	Row	28	0.6	1.1	79	1.64E5	719	1.01
PANEL	Column	43	0.7	4.6	2502	7.43E4	585	1.02

TABLE 4
Comparison of Strategies Using Different Orderings for BCSSTK08.

Ordering	Strategy	Factorization Time	α	Number of Iterations	Iteration Time
Given	Standard	0.35	1.00	17	1.33
Given	Column	1.57	1.01	13	1.03
Given	Row	12.05	1.00	11	0.89
MDO	Standard	0.20	1.00 (1.01)	26 (25)	2.61 (2.59)
MDO	Column	0.51	1.00	16	1.91
MDO	Row	1.30	1.00	60	0.67
RCM	Standard	0.24	1.00	20	2.23
RCM	Column	0.56	1.01	10	1.16
RCM	Row	4.92	1.01	14	1.44

TABLE 5
Comparison of Strategies Using Different Orderings for BCSSTK09.

Ordering	Strategy	Factorization Time	α	Number of Iterations	Iteration Time
Given	Standard	0.23	1.06 (1.09)	111 (53)	10.43 (5.40)
Given	Column	0.51	1.00	22	2.14
Given	Row	1.55	1.00	17	1.68
MDO	Standard	0.25	1.11 (1.16)	92 (69)	8.72 (6.54)
MDO	Column	0.61	1.05 (1.06)	51 (49)	4.87 (4.68)
MDO	Row	2.87	1.17	77	7.31
RCM	Standard	0.22	1.09 (1.13)	66 (56)	6.23 (5.28)
RCM	Column	0.49	1.00	33	3.16
RCM	Row	1.30	1.00	24	2.33

TABLE 6
Comparison of Strategies Using Different Orderings for BCSSTK10.

Ordering	Strategy	Factorization Time	α	Number of Iterations	Iteration Time
Given	Standard	0.32	1.09	71	7.61
Given	Column	0.62	1.01	32	3.50
Given	Row	0.91	1.01	26	2.85
MDO	Standard	0.34	1.07 (1.10)	116 (113)	12.41 (12.09)
MDO	Column	0.70	1.06 (1.07)	78 (74)	8.37 (7.95)
MDO	Row	1.22	1.01 (1.02)	45 (44)	4.89 (4.77)
RCM	Standard	0.32	1.04	59	6.33
RCM	Column	0.60	1.01	27	2.95
RCM	Row	0.86	1.17	100	10.68

TABLE 7
Comparison of Strategies Using Different Orderings for BCSSTK11.

Ordering	Strategy	Factorization Time	α	Number of Iterations	Iteration Time
Given	Standard	0.51	1.03 (1.04)	623 (622)	99.90 (99.90)
Given	Column	1.15	1.05	610	98.12
Given	Row	2.65	1.02	415	66.56
MDO	Standard	0.55	1.06	717	115.87
MDO	Column	1.22	1.10	828	133.84
MDO	Row	2.77	1.02 (1.03)	540 (508)	87.07 (81.92)
RCM	Standard	0.52	1.06	688	109.97
RCM	Column	1.12	1.07	672	107.65
RCM	Row	3.32	1.02	448	71.65

TABLE 8
Comparison of Strategies Using Different Orderings for BCSSTK19.

Ordering	Strategy	Factorization Time	α	Number of Iterations	Iteration Time
Given	Standard	0.09	1.14 (1.18)	1375 (1129)	61.17 (50.25)
Given	Column	0.15	1.04 (1.05)	689 (596)	30.67 (26.52)
Given	Row	0.95	1.01	341	15.29
MDO	Standard	0.10	1.15 (1.17)	1462 (1182)	64.70 (52.31)
MDO	Column	0.16	1.17 (1.18)	1283 (1184)	56.77 (52.38)
MDO	Row	0.24	1.07 (1.09)	899 (866)	39.86 (38.39)
RCM	Standard	0.09	1.18 (1.20)	1263 (1207)	56.32 (53.80)
RCM	Column	0.14	1.07 (1.09)	844 (839)	37.63 (37.41)
RCM	Row	0.19	1.15 (1.17)	1126 (1074)	50.35 (48.02)

mum degree (MDO) and reverse Cuthill-McKee (RCM) heuristics. We included these other orderings to show that the utility of our strategies is not limited to a specific ordering. In addition to using the first acceptable, or "critical," value of α, we also continued to search for an "optimal" α. By optimal, we mean the value of α, after the first acceptable α, that results in the fewest number of iterations. The search for the optimal α is halted when the number of iterations begins to increase; the reasoning for this search pattern is based on the results presented in [11]. When the optimal α differs from the first acceptable α, the results for the optimal value are reported in parentheses. Again, the concern was whether the superiority of the column and row algorithms was independent of whether the "critical" or "optimal" α was found. The results of these comparisons are given in Figures 4–8. From these results we conclude that the row and column strategies will generally require less execution time than the standard strategy, regardless of what ordering is used or whether an "optimal" α is found.

6. Concluding Remarks. We have presented two new strategies for improving the incomplete factorization for use in the preconditioned conjugate gradient algorithm. The size of these incomplete factors is the same as the size of the lower triangle of the original matrix. Thus, the cost of the triangular systems solves is the same as for the standard incomplete factorization. Our experimental observations, obtained with difficult problems from structural engineering applications, show that our strategies can require significantly fewer iterations for convergence than does the standard method. There is a moderate increase in the cost of computing the incomplete factorizations, but this cost is more than offset by the decrease in the number of iterations required. Neither of the two strategies seemed to consistently require fewer iterations than the other; however, we prefer the column strategy over the row strategy because it requires less storage. We have also presented some theoretical motivation, in addition to

the experimental results, to support the claim that these methods are superior to a standard incomplete factorization.

In other recent work, we have shown that coloring heuristics can be used to provide orderings for positive definite matrices arising in practical problems [8]. These orderings provide a large amount of exploitable parallelism for incomplete factorization and the accompanying triangular system solves. A possible drawback to the two strategies presented in this paper is that, if implemented in a straightforward fashion, they will destroy the parallelism inherent in a given ordering. We note that a modification of these strategies that does not destroy this parallelism may be achieved by restricting fill-in to locations consistent with the coloring employed. An efficient parallel implementation of this algorithm is an interesting topic for further research.

REFERENCES

[1] O. Axelsson and N. Munksgaard, *Analysis of incomplete factorizations with fixed storage allocation*, in Preconditioning Methods Theory and Applications, D. Evans, ed., Gordon and Breach, 1983, pp. 219–241.

[2] I. S. Duff, R. Grimes, J. Lewis, and B. Poole, *Sparse matrix test problems*, SIGNUM Newsletter, 17 (1982), p. 22.

[3] I. S. Duff and G. A. Meurant, *The effect of ordering on preconditioned conjugate gradients*, BIT, 29 (1989), pp. 635–657.

[4] R. W. Freund and N. M. Nachtigal, *An implementation of the look-ahead Lanczos algorithm for non-Hermitian matrices: Part II*, Technical report, RIACS, 1990.

[5] W. D. Gropp, D. E. Foulser, and S. Chang, *CLAM User's Guide*, Scientific Computing Associates, 1989.

[6] I. Gustafsson, *A class of first order factorization methods*, BIT, 18 (1978), pp. 142–156.

[7] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, Journal of Research of the National Bureau of Standards, 49 (1952), pp. 409–436.

[8] M. T. Jones and P. E. Plassmann, *Scalable iterative solution of sparse linear systems*, Preprint MCS-P277-1191, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1991.

[9] S. Kaniel, *Estimates for some computational techniques in linear algebra*, Mathematics of Computation, 20 (1966), pp. 369–378.

[10] D. S. Kershaw, *The incomplete Cholesky–conjugate gradient method for the iterative solution of systems of linear equations*, Journal of Computational Physics, 26 (1978), pp. 43–65.

[11] T. A. Manteuffel, *An incomplete factorization technique for positive definite linear systems*, Mathematics of Computation, 34 (1980), pp. 473–497.

[12] J. Meijerink and H. A. van der Vorst, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Mathematics of Computation, 31 (1977), pp. 148–162.

[13] ——, *Guidelines for the usage of incomplete decompositions in solving sets of linear equations as they occur in practical problems*, Journal of Computational Physics, 44 (1981), pp. 134–155.

[14] N. Munksgaard, *Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients*, ACM Transactions on Mathematical Software, 6 (1980), pp. 206–219.