INCOMPLETE LU FACTORIZATION: A MULTIFRONTAL APPROACH

YOGIN E. CAMPBELL * AND TIMOTHY A. DAVIS †

Technical Report TR-95-024, Computer and Information Sciences Department, University of Florida, Gainesville, FL, 32611 USA. October, 1995.

Key words. Unsymmetric-pattern multifrontal method, sparse matrices, incomplete factorization, iterative methods.

AMS (MOS) subject classifications. 05C50, 65F50, 65F05.

Abstract.

We present a new class of preconditioners based on an unsymmetric multifrontal incomplete LU factorization algorithm. In this algorithm entire update rows and columns of the frontal matrices are dropped based on a level drop strategy. We discuss some of the successes and difficulties inherent in applying this drop strategy. In general, we found the memory usage of this incomplete multifrontal factorization algorithm to be smaller than its complete multifrontal counterpart. We present numerical results to show the quality of the factors when used as preconditioners with the conjugate gradient squared iterative method.

1. Introduction. One of the effective techniques used to accelerate the convergence rate of an iterative method when solving a system of linear equations is the application of a preconditioner to the system. The main aim in applying a preconditioner, M, is to transform the original linear system of equations, $Ax = b$, into the new system, $M^{-1}Ax = M^{-1}b$, on which the iterative method has a faster convergence rate. Of course, not every preconditioner will produce a faster converging transformed system. The iterative method typically has a faster convergence rate on the transformed system, $M^{-1}Ax = M^{-1}b$, if the eigenvalue spectrum of the preconditioned coefficient matrix, $M^{-1}A$, is more densely clustered than the original coefficient matrix A [12].

Much research has been done on the effectiveness of the preconditioners constructed using the incomplete LU or Cholesky factorization approach [2, 3, 4, 14, 13, 15, 16, 18]. The focus in constructing effective preconditioners based on the incomplete factorization approach has been to determine the best (or most useful) set of elements to drop (or to keep) as the factorization algorithm progresses. Several dropping heuristics have been developed, each with the goal of selecting the maximum set of entries to discard in order to produce the most accurate preconditioner. These heuristics include dropping by value, dropping by position, dropping based on storage constraints, or some hybrid form of the previous three strategies.

In this paper we discuss a new class of preconditioners based on the incomplete LU factorization of general unsymmetric matrices [5]. Our incomplete factorization algorithm is derived from the unsymmetric-pattern multifrontal algorithm of Davis and Duff [9, 7]. The dropping strategy based on fill levels. The basic idea is to associate a fill level with each entry in the coefficient matrix and a prescribed way of updating those fill levels as the incomplete factorization progresses. Entire rows and columns of entries in the contribution blocks with levels greater than the level tolerance are dropped at each step of the factorization.

* email: yec@cis.ufl.edu.

† Computer and Information Science and Engineering Department, University of Florida, Gainesville, Florida, USA. (904) 392-1481, email: davis@cis.ufl.edu. Technical reports and matrices are available via the World Wide Web at http://www.cis.ufl.edu/~davis, or by anonymous ftp at ftp.cis.ufl.edu:cis/tech-reports.

In Section 2, we begin with a general description of the level dropping heuristic used in this paper. Section 3 gives a step by step discussion of the unsymmetric-pattern multifrontal (complete) LU factorization algorithm. For more details on the unsymmetric-pattern multifrontal algorithm we refer the reader to [9, 7]. In Section 4 we develop the unsymmetric multifrontal *incomplete* algorithm. This is followed in Section 5 by the results and discussion of our numerical experiments. Here we focus on the quality of the incomplete factors used as preconditioners in the conjugate gradient squared iterative method. Section 6 contains our concluding remarks.

2. The level dropping heuristic. Every dropping heuristic is based on the idea that a measure of importance (as far as the quality of the preconditioner constructed is concerned) can be assigned to the entries of the LU factors. Thus, in the drop-by-value heuristic, entries smaller than the drop tolerance value are considered to be of lesser importance than those entries greater than the dropping tolerance value. Similarly, entries in certain locations are the preferred entries in the drop by location scheme. In the *fill level* dropping heuristic we use, the *measure of the importance* of an entry a_{ij} in the LU factors is based on which other entries are used to numerically update a_{ij} as the factorization progresses. That is, an entry is assigned an initial fill level (or level) and this fill level is updated each time the entry is numerically updated. Entries with lower fill levels are considered more important than those with higher fill levels. The original nonzero entries are initially assigned the lowest fill level possible (zero), while original zero entries are initially assigned the highest fill level possible. The fill level of the original entries remain at zero throughout the incomplete factorization. For the original zero entries the procedure to modify the fill levels are as follows. Recall that in the Gaussian elimination process, the eventual numerical value of an entry is due to updates done at each step of the elimination. The eventual fill level of an original zero entry is therefore due to the fill levels of all the update terms applied to that original zero entry. This eventual fill level is defined to be the *minimum* of all the fill levels of the update terms that contribute to the numerical value of the original zero entry. If an update term is created by two entries with fill levels f_1 and f_2, we define the fill level of that update term to be $\max(f_1, f_2) + 1$.

In general we have the following formula to compute the fill levels. Let f_{ij}^k be the level of the entry at position (i, j) at step k of the ILU factorization. Then the updates to f_{ij}^k that accompany the numerical updates is given by:

$$f_{ij}^{[k+1]} = \min(f_{ij}^{[k]}, \max_k(f_{ik}^{[k]}, f_{kj}^{[k]}) + 1) \qquad i, j > k > 0 \ . \tag{2.1}$$

The initial ($k = 0$) level settings are

$$f_{ij}^{[0]} = \begin{cases} 0 & a_{ij} \neq 0 \\ \infty & a_{ij} = 0 \ . \end{cases} \tag{2.2}$$

The fill levels of the original entries remain zero throughout the factorization. The fill levels of the original zero entries, however, are initially set to a high value and, by the *min* function these fill levels can decrease as the factorization progresses. Note that in the multifrontal approach the fill level updates, like their numerical counterparts, are applied piecemeal. The eventual fill level of an original zero entry therefore remains undetermined until the entry enters a pivot row or pivot column (or "infinite" if it never becomes nonzero). As an example, if a fill-in entry currently has a fill-level of 10 and it is (numerically) updated by entries with fill levels 6 and 3, its fill level after the update will be

$$\min(10, \max(6, 3) + 1) = 7.$$

```
0:    initializations
      while (factoring A) do
1a:     do global pivot search for seed pivot
1b      form frontal matrix F
        while (pivots found within F) do
2a:       assemble prior contribution blocks and original rows into F
2b:       compute the degrees of rows and columns in C (contribution block of F)
2c:       numerically update part of C (levels 2 & 3 BLAS)
2d        do local pivot search within C
        endwhile
3a:     complete update of C using level 3 BLAS
3b:     save C, LU factors, and quotient graph information
      endwhile
```

FIG. 3.1. *Outline of the unsymmetric-pattern multifrontal (complete) algorithm*

The fill level of this fill-in entry can eventually go all the way down to 1, but never higher than 7.

3. The unsymmetric multifrontal (complete) LU algorithm. Our unsymmetric multifrontal incomplete LU factorization algorithm is derived from the unsymmetric-pattern multifrontal factorization algorithm of Davis and Duff (Harwell-Boeing subroutine MA38 or UMFPACK Version 2.0) [9, 7]. Some of the main features of the Davis/Duff algorithm are summarized below.

First, this algorithm has no separate analyze and factorize phases. Pivot selection and elimination are both done in a single pass since it is assumed from the outset that pivoting for numerical stability is necessary. Second, the frontal matrices are unsymmetric and rectangular rather than square. Third, because the frontal matrices are unsymmetric, some contribution blocks must be assembled into more than one subsequent frontal matrices. The dependencies among the frontal matrices are, therefore, given by a directed acyclic graph (assembly dag). Fourth, advantage is taken of repeated structures in the matrix by factorizing more than one pivot in each frontal matrix: relaxed amalgamation. This relaxed amalgamation allows the use of dense matrix kernels in the innermost loops (level 3 BLAS [10]). Fifth, pivot selection and degree update are based on an unsymmetric analogue of the (symmetric) approximate minimum degree ordering algorithm [1]. Finally, aggressive assembly is done: as many rows and columns of entries as possible from the original matrix and unassembled contribution blocks are assembled into the current frontal matrix. An outline of the algorithm is shown in Figure 3.1.

3.1. Step 1a: selecting a seed pivot. The creation of a new front begins with the selection of a pivot via a global pivot search. Each global (*seed*) pivot is chosen from the entries of the remaining active submatrix. This choice of seed pivot is based on two criteria: fill-in control and numerical stability. To help in selecting a pivot with low fill-in, the upper bound degrees of the remaining rows and columns in the active submatrix are maintained (see [1] for a discussion of how these upper bound degrees are efficiently computed). Let d_x and $\overline{d}_x$ be the true and approximate degree of the row or column x, respectively, where $d_x \leq \overline{d}_x$. The search for the seed pivot is focused on a small set of columns (typically 4) with the smallest upper bound degrees. From this set of columns, the seed pivot a_{rc}, is selected such that the approximate

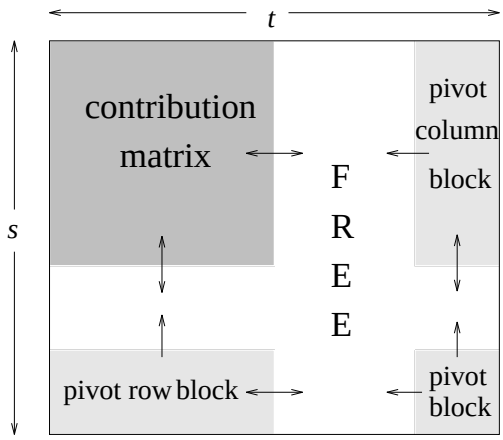

FIG. 3.2. *The frontal matrix data structure*

Markowitz cost, $(d_c - 1)(\overline{d}_r - 1)$, is a minimum, and $|a_{rc}| \geq u \max_i |a_{ic}|$, $0 < u \leq 1$ (with u typically set to 0.1).

3.2. Step 1b: allocating memory for the current front. After a seed pivot a_{rc} is found, memory of size $s \times t$ is allocated for the current frontal matrix F in step 1b. The values of s and t are determined by a growth-factor parameter g with $s = gd_r$ and $t = gd_c$. The growth-factor essentially controls the size to which a given frontal matrix is allowed to grow beyond the seed pivot frontal matrix size. Allowing the frontal matrix to extend beyond its initial size is the technique used to factorize several pivots within a given front. The effect of making g larger is to allow fewer, larger fronts to be created during the course of the algorithm. The level 2 and level 3 BLAS give much better performance on these larger fronts, increasing the megaflop performance of the algorithm. But, as a front grows, more fill-ins are likely to occur, resulting in an increase in both storage and CPU time. Some care is thus needed in setting the value of g to obtain low storage and low factorization time simultaneously. The actual layout of the $s \times t$ work array is shown in Figure 3.2.

3.3. Step 2a: assembly and memory usage. Once the global seed pivot is chosen and the work array is setup for the frontal matrix, steps 2a-2d are done repeatedly until the current front cannot be further extended - because of a numerical pivot failure, work array constraints or the contribution block is empty. In step 2a, entries from the original matrix A and prior unassembled contribution blocks are assembled. Aggressive assembly is done in this step in the sense that all possible row or column of entries that can be assembled into the frontal matrix are actually assembled. This allows the memory allocated for these assembled entries to be deallocated, and thus, provides a very important way of reducing the amount of memory used.

3.4. Step 2b: degree update. The approximate degree update of the rows and columns in the current contribution block is done here in preparation for the *local* pivot search in step 2d. Finding a tight upper bound for the degrees of the rows and columns in the contribution matrix involves the use of an efficient scheme described in [1]. As shown in [1], the tight upper bounds on the degrees usually result in low fill-in.

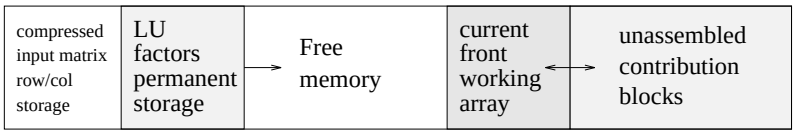

FIG. 3.3. *Partitioning of real memory*

3.5. Step 2c: numerical update using the level 3 BLAS. Forming the Schur complement to update the contribution block is done in step 2c. Rather than doing the pivot update on each pass of the algorithm, however, the updates are postponed until "enough" pivots have been factorized to allow for the effective use of level 3 BLAS (usually around 16 on the CRAY-YMP).

3.6. Step 2d: the local search to extend the current frontal matrix. The local pivot search done in step 2d is based on the same partial pivoting idea used for the global pivot search. Here, the local pivot selected is the entry in the current contribution matrix having minimal approximate Markowitz cost among those entries that are numerically acceptable. The algorithm breaks out of the second while loop when the current front can no longer be extended. This can occur for any of the following reasons. Perhaps no numerically acceptable pivot is found within the contribution block. Or, perhaps, a pivot a'_{rc} is found, but the length of the column or row is too long, i.e., $d_c > s$ or $d_r > t$. Moving the computed LU factors (the light-grey shaded areas in Figure 3.2) out of the front to permanent storage can free-up enough memory to allow the extension to proceed if $s-k < d_c < s$ and $t-k < d_r < t$, where k is the number of pivots currently held in the front. Finally, maybe the front cannot be extended because the size of the contribution matrix has been reduced to zero.

3.7. Step 3: closing off processing on the current front. An unsuccessful pivot search signals the end of factorization using the current front; the final processing on the current front and preparation for subsequent fronts are done in steps 3a and 3b. In these two steps, the update of the current contribution block C is completed, C is then put onto a heap for later assembly, the LU factors computed within the current frontal matrix are stored, and, finally, information on the row and column structure of C are saved to allow the easy assembly of C into subsequent frontal matrices (the quotient graph information [1]).

3.7.1. Data structures. The real memory used by the unsymmetric-pattern multifrontal algorithm is partitioned as shown in Figure 3.3. (This memory layout is actually for an early version of the algorithm embodied in MA38/UMFPACK2.0.) A similar layout is used for the integer memory used by the algorithm. The memory layout of the frontal matrix working array is illustrated in Figure 3.2. The memory is partitioned into five sections: contribution matrix, pivot block, pivot row block, pivot column block, and an unused area into which the other four blocks can grow as indicated by the arrows. This data structure allows both the contribution matrix area and the LU factors areas to grow and shrink easily within the frontal working array. Davis first presented this arrangement in [6, 8].

4. The unsymmetric multifrontal incomplete LU algorithm. The steps involved in the unsymmetric multifrontal incomplete L algorithm are very similar to those discussed in Section 3 for the the unsymmetric multifrontal (complete) LU

0: initializations
set level tolerance
while (computing the ILU factors of **A**)
1a: do global pivot search for seed pivot $\mathbf{a_{rc}}$
1b: compute and partition level sets of **r** and **c**
1c: form *reduced* frontal matrix **F**
while (pivots found within **F**) **do**
2a: assemble prior contribution blocks and original rows into **F**
2b: compute degrees of rows and cols in **C** (the contribution block of **F**)
2c: numerically update part of **C**
2d: do local pivot search within **C**: pivot $\mathbf{a_{r'c'}}$
2e: compute and partition level sets of $\mathbf{r}'$ and $\mathbf{c}'$
endwhile
3a: complete update of **C**
3b: save **C**, *LU* factors, quotient graph information, and level sets
endwhile

FIG. 4.1. *The unsymmetric-pattern multifrontal incomplete LU algorithm*

algorithm. We had three basic considerations in mind when tailoring the complete factorization algorithm to do an incomplete *LU* (ILU) factorization. First and foremost, the ILU algorithm should use less memory than the complete factorization algorithm uses on a given matrix. This is vital if constructing the ILU factors is to be feasible for larger problems than the complete algorithm would have storage for. Second, the number of fill-ins should be significantly reduced in the ILU factors (after all, this is the primary reason for constructing the incomplete factors in the first place). The amount of fill-in affects not only the storage and CPU construction costs of the ILU preconditioner, but also the cost per iteration of using the preconditioner. Finally, we wanted to maintain the use of the dense matrix kernels in the innermost loops, the level 2 and level 3 BLAS primitives. The effective use of these primitives is of great importance in keeping the CPU time low in the full factorization, and we hoped to carry this over to the ILU algorithm. But, as we shall see, we had to settle for less than optimal use of the BLAS in the interest of keeping memory usage relatively low. An outline of the ILU algorithm is shown in Figure 4.1.

As in the complete algorithm, the incomplete algorithm is structured around the use of a global phase and a local phase. We must now, however, consider the effects of the level drop strategy on the pivot search, the assembly process, the degree update, the memory usage, and the efficiency of the BLAS primitives. We proceed as we did for the complete algorithm, by giving a step-by-step explanation of the incomplete algorithm, pointing out, as we go along, the main differences between the incomplete and complete algorithms.

4.1. Step 1a: selecting the seed pivot. The global pivot search step is similar to the global pivot search in the complete *LU* factorization, except now its is even less certain that the seed pivot satisfies the local fill-reducing criterion. The difficulty of enforcing the local fill-reducing criterion arises from the fact the approximate degrees on which the pivot search is based do not take into account the dropping of rows and columns in the contribution matrix due to the level tolerance criterion. Figure 4.2 shows the general partitioning of a row or column based on the level tolerance. This partitioning scheme naturally gives rise to the partitioning of the frontal matrix shown

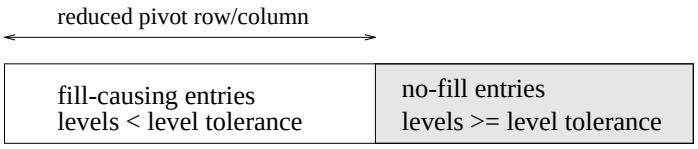

FIG. 4.2. *Partitioning of a row/column based on the level tolerance*

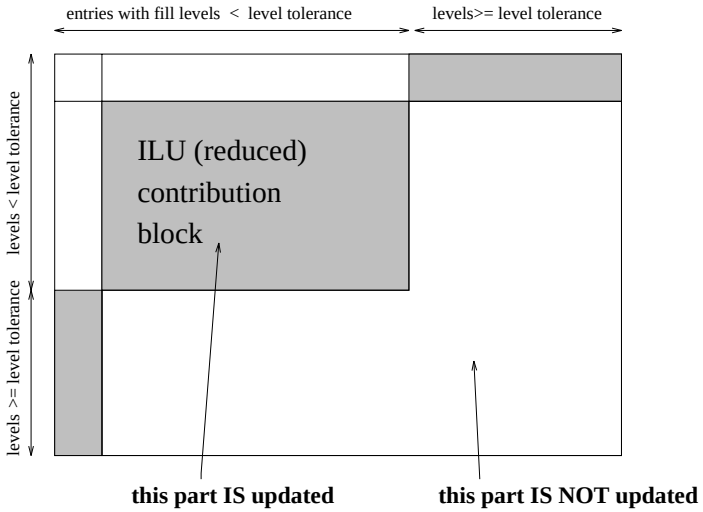

FIG. 4.3. *Structure of the ILU frontal matrix*

in Figure 4.3. Clearly, one effect of the level tolerance partitioning is to reduce the size of the contribution matrix and, thus, the potential for fill-in. For example, a pivot column, c_1, of approximate degree say 25, may be favored over a column, c_2, with approximate degree 35, on which to do a pivot search. The problem is that of the 35 entries in column c_2, 20 may have levels above the level tolerance, while only 5 entries in c_1 may have levels above the level tolerance. Assuming that the minimum degree row is of the same length, m, in both c_1 and c_2, this argument implies that c_2, with a *reduced* contribution block size of $15m$, is actually a better choice than c_1 with a *reduced* contribution block size of $20m$, as far as the local fill-reducing criterion is concerned. Figure 4.4 highlights this problem. Subfigures **A** and **B** show the situation in a full factorization, while subfigures **C** and **D** show how it is in the incomplete case. Notice that in the complete factorization, p_1 would be a better choice than p_2, but the reverse is true in the incomplete case.

Computing the level sets of all columns and rows when doing a global (or local) pivot search is, of course, too expensive. We attempt to reduce this effect by computing the level sets of the four or so columns with the smallest approximate degrees and rate them according their "reduced" degrees; by "reduced" degree we mean the number of entries in the column with levels less than the level tolerance (the unshaded part in Figure 4.2).

4.2. Step 1b: computing the row and column level sets. In steps 1b and 2e the levels of the entries in the pivot row and column is computed. The pivot row/column is then partitioned as per Figure 4.2.We discussed in Section 2 how the

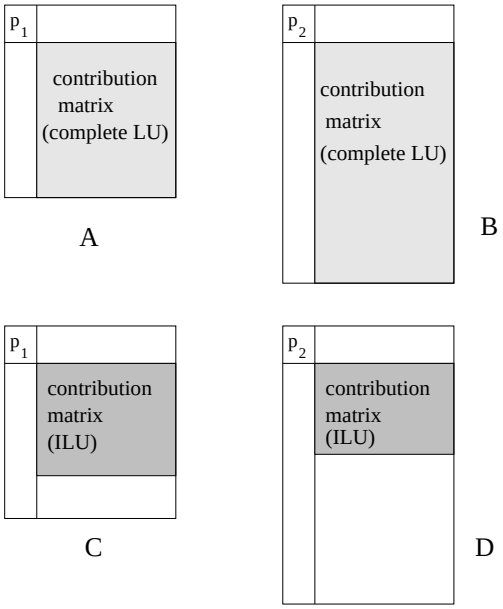

FIG. 4.4. *Shrinking of the contribution block due to level partitioning*

fill level of an entry is computed. The fill level updates to an entry remain *implicit* (not actually done) until the pivot row or column to which the entry belongs become pivotal. Explicitly updating the fill level as each numerical update is done would require storage for fill levels of all entries in the current contribution block and all unassembled contribution blocks. Let C_r and C_c be the number of rows and columns in the contribution matrix C, respectively. The storage involved to keep track of level updates (using the implicit approach) is then $(C_r + C_c)$, rather than $(C_r C_c)$, for the current frontal matrix and each unassembled contribution matrix C, if the explicit approach were used.

4.3. Step 1c: allocating memory for the ILU frontal matrix. In step 1c we allocate storage for the ILU frontal matrix shown in Figure 4.3. Note that as a result of the smaller ILU contribution matrix (relative to what it would have been in a complete factorization), the ILU frontal matrix size is always less than or equal to the size of the corresponding complete LU frontal matrix size. Based on the partitioning of the row and column level sets, the "incomplete" frontal matrix is now partitioned into the pivot row, the pivot column, the reduced contribution block (updates and assembly occur in this area), and the "wing" area (the lightly shaded area in Figure 4.3). No updates or assembly occur in the "wing" area and memory need not be allocated for this part. The two-dimensional ILU frontal matrix work array is once again of size $s \times t$. Now, however, $s = g\tilde{d}_c$ and $t = g\tilde{d}_r$, where $\tilde{d}_c \leq d_c$ and $\tilde{d}_r \leq d_r$ are the reduced lengths of the pivot column and pivot row, respectively.

In the *complete* factorization implementation, we found that a growth-factor g of two gives good results (storage and CPU time) on matrices from many disciplines. As the value of g approaches one both storage and CPU time increases. Storage increases because more fronts are formed, and more contribution blocks need to be stored. CPU time increases primarily because smaller blocks are involved in the dense BLAS computations. On the other hand, as g is set to values increasingly greater than

two, fewer (and larger) fronts are formed allowing the level 2 and level 3 BLAS to perform increasingly better. But, the amount of fill-ins becomes greater, requiring an increasing number of floating point operations to do the factorization, and an increasing amount of memory to store. The end result is once again increased storage and CPU costs. For the ILU algorithm, an additional factor is the level tolerance setting.

For a given tolerance setting, three factors influence the behavior of the ILU algorithm as the parameter g (and, therefore, the size of the working array for the front) is varied. Larger values of g encourages the formation of larger-sized fronts, resulting in better performance of the level 2 and level 3 BLAS since larger-sized blocks are involved. The algorithm then has a better megaflop performance. (Smaller values of g have the opposite effect.) Inasmuch as a high megaflop performance rate is desirable, we run into the following difficulties having to do with memory usage and fill-ins. The larger fronts typically leave behind larger unassembled contribution blocks. The rate of absorption of unassembled contribution blocks into subsequent fronts tends to be lower than in the complete factorization. This means that these unassembled contribution blocks take longer to be assembled into subsequent fronts resulting in higher memory usage. This problem can be especially severe for lower values of the level tolerance. Finally, larger values of g and, therefore, larger fronts generally mean higher levels of fill-in. While true even in the complete LU algorithm, in the ILU algorithm the effect can be more pronounced because of the following effect. The number of pivots factorized within a given size of the working array in the ILU algorithm is usually larger than for the same work array size in the complete algorithm. This occurs because the size of the work array only needs to be large enough to contain the reduced contribution block and reduced pivot column and row. Thus, for a given work array size, the "seed" frontal matrix in the ILU case has more room for extension than in the complete case. Fortunately, the fill-in per pivot in the front is reduced due to the smaller size of the contribution block for the ILU case, and this reduction tends to lessen the effect of the increased number of pivots factorized within a given front.

4.4. Step 2a: assembling into the reduced frontal matrix. The main thrust of step 2a in the incomplete algorithm is much the same as for the complete algorithm: to assemble as many entries as possible from the prior unassembled contribution blocks and the original matrix in an effort to free up as quickly as possible the maximum amount of memory. This can be less effective in the incomplete case than in the complete case because of the reduced size of the contribution matrix in the incomplete factorization. The net effect is that unassembled contribution matrices tend to stay around much longer as explained in Section 4.3.

4.5. Steps 2b and 2c: degree and numerical updates. The approximate degree update, step 2b, is based on the reduced size of the contribution matrix. As explained earlier, the numerical update of the reduced contribution matrix, step 2c, tends to involve smaller-sized blocks in the level 2 and level 3 BLAS. This can result in higher overall CPU time to do the numerical updates.

4.6. Step 2d: extending the frontal matrix. To effectively take advantage of dense matrix kernels, and also to reduce the amount of memory used, the frontal matrix is extended by searching for pivots within the reduced contribution matrix (local pivot search). The criteria for exiting the local pivot search loop are modified somewhat from what they are in the complete algorithm. A numerical failure is

$$A = \begin{bmatrix} a_{11} & 0 & 0 & a_{14} & a_{15} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & a_{25} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 0 & a_{37} \\ a_{41} & 0 & 0 & a_{44} & a_{45} & a_{46} & 0 \\ 0 & a_{52} & a_{53} & 0 & a_{55} & a_{56} & 0 \\ 0 & 0 & 0 & 0 & 0 & a_{66} & a_{77} \\ a_{71} & 0 & a_{73} & 0 & a_{75} & 0 & a_{77} \end{bmatrix}$$

FIG. 4.5. *Example matrix*

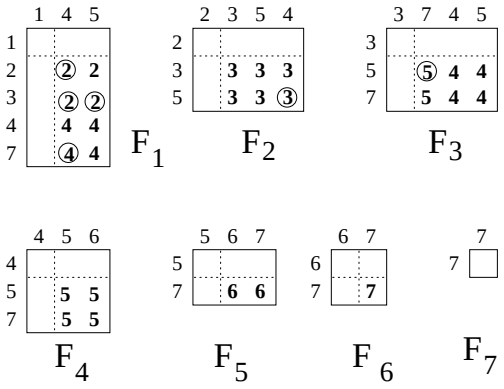

FIG. 4.6. *Progress of the complete algorithm on the matrix in Fig. 4.5*

bypassed if it occurs less than a set number of times within a front (typically 3). That is, rather than allowing a local pivot failure due to the size of the pivot value being too small, we replace the pivot value by some specified value (typically 1.0). Kershaw in [14] used this strategy to avoid pivot failures and/or the loss of positive-definiteness of the incomplete Cholesky factors. Our rationale for using it here is different since we are not attempting to maintain positive-definiteness in the ILU factors and the ILU algorithm will not necessarily fail at this stage if we permitted a local pivot failure. Our main concern is with memory usage, since the contribution block left behind (if a failure is permitted) tends to remain allocated longer because of the slower rate of assembly in the incomplete algorithm. If this kind of local failure is infrequent (as we found for the matrices tested) the effect of replacing the small pivot value by some larger value should have little effect on the quality of the preconditioner. The local pivot search loop is exited if the length of the *reduced* local pivot column or row exceed the size of the working area.

4.7. Step 3: final processing on the current front. The only difference between step 3 in the ILU code and complete code is the need to store the level set information for the last local pivot. Note that the row and column level sets of the last local pivot are the only pieces of level information that need to be saved from the just completed front.

4.8. An example. Consider the unsymmetric matrix A, shown in Figure 4.5. Assuming that the pivots are chosen in sequence from the diagonal, Figure 4.6 shows the sequence of simple fronts, F_1 through F_7, that are formed and factorized as the

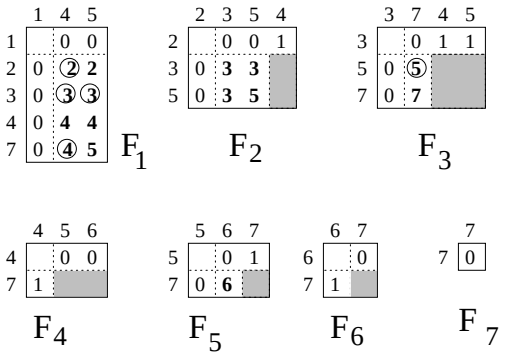

FIG. 4.7. *Progress of the incomplete algorithm on the example matrix*

algorithm progresses. The bold-typed numerals indicate into which frontal matrix the entries in the contribution blocks are assembled. For example, the entries in the contribution block of F_1 labeled **2** and **4** are assembled into F_2 and F_4, respectively. A contribution matrix is deallocated when all its entries have been assembled into subsequent frontal matrices. (We found doing garbage collection to reclaim the assembled parts of a contribution block to be too expensive using our current data structures. With some modification to the data structures and code logic, this more extensive garbage collection can be done.) Consequently, the entire contribution block belonging to F_1 occupies storage until after the assembly into F_4 is done. The small circles indicate the fill-in entries.

Figure 4.7 shows the progression of frontal matrices formed and factorized using an *incomplete* factorization, assuming the same choice of pivots (along the diagonal) as for the complete factorization and a level tolerance of value one is used. The gray-shaded areas indicate the parts of the contribution matrices that are dropped. The numbers within the pivot rows and columns give the levels of the associated pivot entry. For example, the entry at position $(1,4)$ has level 0, while that at $(2,4)$ has level 1. These levels are computed using the formula in Equation (2.1). The bold-typed numerals in the reduced contribution matrices once again indicate when the entries are assembled.

The following observations can be made by inspection of Figures 4.6 and 4.7. The number of fill-ins has dropped from six to five because column four of the contribution block of frontal matrix F_2 has been dropped in the ILU factorization. Typically, less memory is required in the incomplete case for a given frontal matrix because of the reduced size of the contribution matrix. Aggressive assembly in the incomplete algorithm is hampered by the reduced sizes of the contribution blocks. This is evident from the fact that, for example, the contribution block C_1 in the incomplete factorization, is not *fully* assembled until after F_5 is formed, while in the complete factorization C_1 is fully assembled into F_4.

We should point out that the pivot sequence is usually different in the ILU algorithm, making precise comparisons of memory usage and fill-in between the complete and incomplete algorithms difficult. In addition, both memory usage and fill-in depend on the level tolerance setting. We would expect, though, that both the number of fill-ins and memory usage will increase as the level tolerance is increased. We further discuss this issue in the next section when we present the results of the numerical experiments.

TABLE 5.1
Matrix Statistics

name	n	nz	sym.	cond(A)	discipline	comments
PORES3	532	3474	0.260	6.6×10^{5}	petroleum eng.	
MCFE	765	24382	.699	5.4×10^{14}	astrophysics	radiative transfer
ORSIRR2	886	5970	.000	4.6×10^{5}	petroleum eng.	$21\times21\times5$ irregular grid
SAYLR4	3564	22316	1.000	–	petroleum eng.	$33\times6\times18$ 3D grid
GEMAT11	4929	33185	.001	–	electric power	linear program. basis
SHERMAN3	5005	20033	1.000	–	petroleum eng.	$35\times11\times13$ grid

5. Numerical experiments and results. We conducted several numerical experiments to determine the quality of the incomplete factors when used as preconditioners. The important measures of the quality of a preconditioner M include the condition number and eigenvalue spectrum of the preconditioned matrix $\tilde{A}$ ($\tilde{A} = M^{-1}A$), the norm of the remainder matrix R ($R = M - A$), the CPU time and storage costs to construct M, the cost per iteration to use M, and the rate of convergence of the preconditioned system. Ideally, a good preconditioner will result in a clustering of the eigenvalues of $\tilde{A}$ around a one or a few groups of values, and the condition number of the preconditioned matrix $\tilde{A}$ being small (preferably of order one). In addition the norm(R) should be small; the closer to zero the better. A good preconditioner should also be cheap to compute and cheap to use, and should significantly accelerate the convergence rate of the iterative algorithm. Simultaneously obtaining two or more of these desired features in a preconditioner is difficult at best and is usually contradictory in nature. For example, a preconditioner that is cheap to construct often does not accelerate the convergence rate as much as one might like. The general goal in constructing preconditioners, therefore, amounts to balancing the conflicting requirements of these quality parameters to suit the particular problem class or computer architecture (or both).

5.1. The test matrices. We were guided in our choice of test matrices by three considerations. We wanted to test problems from several disciplines. We needed to choose matrices small enough to be able to compute inverses, condition numbers, and eigenvalues. We had to use medium to large sized matrices to more clearly determine the effect of the level tolerance on memory usage and CPU time. Table 5.1 gives the set of small and medium-sized matrices on which we report results.

5.2. Numerical experiments information. We used a Sun SPARC station 10 for all experiments. The frontal growth-factor, g, was set to two. We used the preconditioned conjugate gradient square iterative method (CGS) [17] with a maximum of 250 iterations allowed for each run with a preconditioner. Convergence is achieved if the relative 2-norm of the residual is less than 10^{-5} and the relative error in the solution is less than 10^{-3}. Each of the runs used the same right-hand side. For the smaller three matrices in Table 5.1 we were able to compute the 2-norm condition number of the preconditioned matrix and the Frobenius norm of the remainder matrix.

The results are presented in Tables 5.2 through 5.7. Under the level column an I means the unpreconditioned matrix, i.e., $M = I$. The *iters.* column gives the number of iterations before convergence was achieved (250 under this column means that the conjugate gradient method failed to converge). The $cond_2(\tilde{A})$ column shows

TABLE 5.2
Numerical results for PORES3

level	iters.	$\text{cond}_2(\tilde{A})$	$\|\|R\|\|_F$	opcnt $\times 10^6$	mem $\times 10^4$	lunz $\times 10^4$	fronts	ftime (sec)	stime (sec)	ttime (sec)
I	1562	6.6×10^{5}	0.00	–	–	–	–	–	–	–
1	250	1.4×10^{18}	9.1×10^{6}	–	1.99	0.89	145	–	–	–
2	250	1.1×10^{10}	1.6×10^{6}	–	2.04	1.22	131	–	–	–
3	7	3.2×10^{14}	2.9×10^{7}	.355	2.10	0.98	112	.55	.4	.95
4	5	2.7×10^{3}	47.3	.403	2.16	1.15	118	.57	.24	.81
5	4	1.20	8.1×10^{-10}	.355	2.25	1.24	117	.53	.47	1.0
6	1	1.00	2.2×10^{-9}	.339	2.53	1.31	116	.60	.14	.74

TABLE 5.3
Numerical results for MCFE

level	iters.	$\text{cond}_2(\tilde{A})$	$\|\|R\|\|_F$	opcnt $\times 10^6$	mem $\times 10^5$	lunz $\times 10^4$	fronts	ftime (sec)	stime (sec)	ttime (sec)
I	> 2000	5.4×10^{14}	0.00	–	–	–	–	–	–	–
1	4	1.1×10^{29}	6.1×10^{12}	2.85	1.61	7.24	72	2.31	0.78	3.09
2	5	2.8×10^{5}	1.7×10^{12}	8.11	1.33	7.41	67	2.18	1.10	3.28
3	3	43.7	3.1×10^{9}	9.02	1.39	7.45	65	2.40	0.69	3.09
4	1	1.00	181.4	7.84	1.46	7.58	66	2.33	0.29	2.62
5	1	1.00	181.4	7.88	1.53	8.01	64	2.61	0.33	2.61

TABLE 5.4
Numerical results for ORSIRR2

level	iters.	$\text{cond}_2(\tilde{A})$	$\|\|R\|\|_F$	opcnt $\times 10^6$	mem $\times 10^4$	lunz $\times 10^4$	fronts	ftime (sec)	stime (sec)	ttime (sec)
I	426	6.6×10^{5}	0.00	–	–	–	–	–	–	–
1	250	9.7×10^{4}	5.1×10^{5}	–	4.94	2.25	228	–	–	–
2	159	1.1×10^{7}	9.1×10^{4}	6.31	9.73	3.70	209	1.16	15.3	16.50
3	13	1.7×10^{4}	3.3×10^{3}	2.55	7.95	4.19	188	1.73	1.55	3.28
4	8	1.41	.048	3.21	7.93	4.05	180	1.14	1.14	2.28
5	2	1.09	5.9×10^{-9}	5.52	7.93	4.11	171	1.41	.24	1.65
6	1	1.0	6.7×10^{-10}	5.57	7.94	4.14	172	1.71	.18	1.89

TABLE 5.5
Results for the SAYLR4 *matrix*

level	iters.	opcnt $\times 10^7$	mem $\times 10^5$	lunz $\times 10^5$	fronts	ftime (sec)	stime (sec)	ttime (sec)
1–3	250	–	–	–	–	–	–	–
4	15	3.67	5.23	3.24	823	22.8	15.0	37.0
5	4	3.83	5.71	3.54	827	30.7	4.6	35.3
6	5	3.89	5.48	3.97	819	25.8	5.4	31.2
7	3	16.76	6.59	4.07	822	25.8	2.5	28.3
8	1	17.76	6.38	4.66	822	27.4	1.0	28.4

TABLE 5.6
Results for the GEMAT11 *matrix*

level	iters.	opcnt $\times 10^6$	mem $\times 10^5$	lunz $\times 10^4$	fronts	ftime (sec)	stime (sec)	ttime (sec)
1−2	250	–	–	–	–	–	–	–
3	15	3.70	1.32	5.85	983	2.29	5.59	7.88
4	8	2.87	1.29	6.21	989	1.86	2.88	4.74
5	5	1.73	1.30	6.28	993	2.06	1.84	3.90
6	4	1.11	1.35	6.29	994	2.11	1.47	3.58
7	2	1.41	1.39	6.30	995	2.27	.79	3.06
8	1	1.61	1.41	6.83	995	2.37	.49	2.86

TABLE 5.7
Results for the SHERMAN3 *matrix*

level	iters.	opcnt $\times 10^7$	mem $\times 10^5$	lunz $\times 10^5$	fronts	ftime (sec)	stime (sec)	ttime (sec)
1−3	250	–	–	–	–	–	–	–
4	14	9.03	4.47	3.16	2918	15.5	10.7	26.2
5	11	11.12	4.18	3.48	2916	23.8	9.4	33.2
6	7	10.40	4.02	3.32	2909	15.7	8.1	23.8
7	6	12.44	4.01	3.60	2881	19.6	5.3	24.9
8	4	13.44	4.38	3.68	2834	18.6	3.3	21.9
9	1	13.94	4.42	4.01	2703	21.3	1.3	22.6

the 2-norm conditioned number of the preconditioned matrix (this is also equal to the ratio of the largest singular value to the smallest). The Frobenius norm of the remainder matrix is given in the column with heading $||R||_F$. The total number of floating point operations required to carry out the incomplete factorization and to solve the preconditioned system is given under the *opcnt* column. The memory usage column, *mem*, shows the maximum amount memory (in bytes) used during the incomplete factorization. The memory usage reported here includes the memory used to store the L and U factors. The number of nonzero entries in the L and U factors is given under the *lunz* column. The *fronts* column shows the number of fronts created during the factorization (this is equal to the number of global [seed] pivots found). The *ftime*, *stime*, *ttime* columns give the incomplete factorization time, the time to solve the preconditioned system, and the total time (ftime plus stime), respectively.

5.3. Convergence rate, condition number and $||R||_F$. In general, the rate of convergence is exceptionally fast beyond level tolerances of two or three. Note that for the three smaller matrices (Tables 5.2 − 5.4), both the condition number and Frobenius norm of the remainder matrix ($||R||_F$) also decrease rather rapidly with increasing level tolerance. The decrease of $||R||_F$ with increasing level tolerance supports the observation made in [11] that entries with higher levels tend to have smaller numerical values.

5.4. The eigenvalue spectrum. Figures 5.1, 5.2 and 5.3 show the eigenvalue spectrum for the unpreconditioned matrix and for three level tolerances for the three smaller matrices. Note the rapid clustering of the eigenvalues around the value one for progressively higher values of the level tolerance.

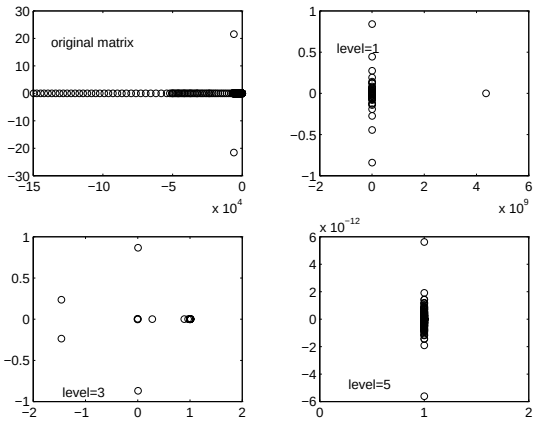

FIG. 5.1. PORES3 *matrix: variation of eigenvalue spectrum with level*

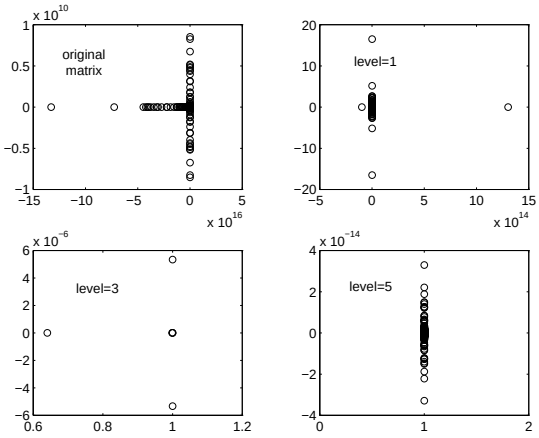

FIG. 5.2. MCFE *matrix: variation of eigenvalue spectrum with level*

5.5. Memory usage. The results reported for memory usage only concern the *real* memory used; for a given matrix, the *integer* memory used varies very little with variation of the tolerance level. The memory used includes internal fragmentation in the unassembled contribution blocks since, as mentioned earlier, we do not attempt to reclaim assembled regions of the contribution blocks. It represents the the maximum amount of memory allocated for the input matrix, the LU factors, the current frontal matrix working array, and the unassembled contribution blocks (with internal fragmentation).

Obtaining consistently low memory usage for the ILU factorization is, perhaps, the most delicate part of the ILU implementation. As explained earlier, the memory usage is both a function of the growth-factor (g) and of the level tolerance. We show the general pattern of memory usage as g is varied for a given value of the level tolerance for the MCFE matrix (Figure 5.4) and the SHERMAN3 matrix (Figure 5.5). From these two figures we see that the memory usage fluctuates as g is varied, but tends to be larger both for values of g much less than 2 (Figure 5.5) and for values of g much greater than 2 (Figure 5.4). This is as we would expect due to the interplay

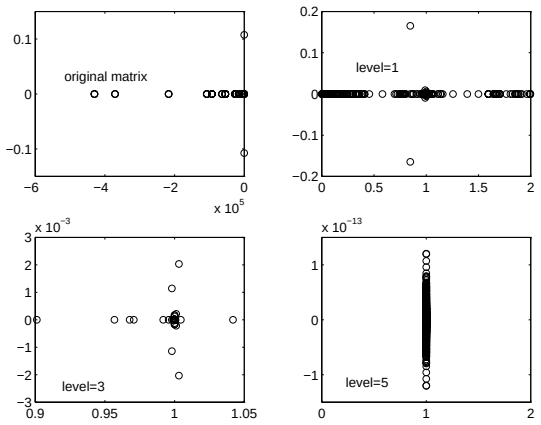

FIG. 5.3. ORSIRR2 *matrix: variation of eigenvalue spectrum with level*

between g, the amount of fill-in, and the sizes of the unassembled contribution blocks discussed earlier. We chose g to be two in all of our experiments. Typically, the memory usage is smaller for smaller values of the level tolerance. The difference between the memory usage for low and high values of the level tolerance is not as significant as we had hoped for (less than 25%). This is not entirely surprising since, as we pointed out earlier, we expected the rate of absorption of the unassembled contribution blocks to be lower than in the complete algorithm. In spite of this, we had hoped that the reduced sizes of the contribution blocks in the incomplete algorithm would have more than compensated for the lower rate of absorption. We simulated the garbage collection to remove the internal fragmentation of the unassembled contribution but this did not improve the situation. (Doing garbage collection does, however, reduce the amount of memory required to do the incomplete factorization from about 5-20 percent for all level tolerances.)

5.6. CPU time. The two important components to the total time are the time to do the ILU factorization to construct the preconditioner and the time to actually solve the preconditioned system. The ILU factorization time is sensitive to the sizes of the blocks used in the BLAS primitives and also to the amount of fill-in. The bigger the block sizes involved in the BLAS calls the more efficient those calls are. The average block size depends on the average size of the fronts formed during the factorization. Consequently, for a given matrix, the smaller the number of fronts formed the larger the average size of the BLAS blocks used and the higher the ILU factorization megaflop performance. For all of the test matrices, the number of fronts formed during the factorization typically decreases as the level tolerance value is increased. This implies that, as far as the efficiency of the level 2 and level 3 BLAS (and megaflop rating) is concerned, higher level tolerances are preferable. But, the ILU factorization time is also dependent on the number of nonzero entries in the L and U factors and this increases as the level tolerance is raised. In addition, the per iteration solve time of the CGS method increases for increased values of level tolerance owing to the increased fill-in and, therefore, the increased number of floating point operations per iteration.

To summarize, the dynamics of the level tolerance on memory usage and CPU

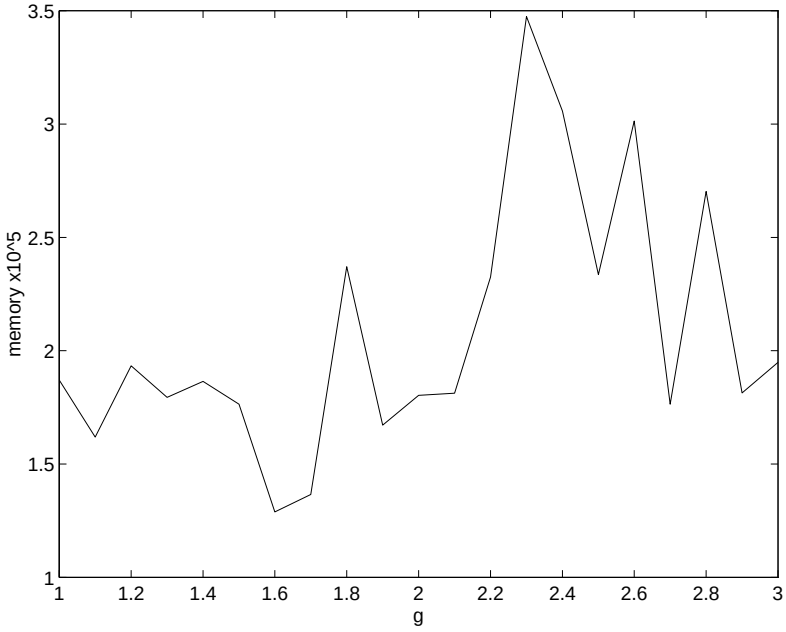

FIG. 5.4. MCFE *matrix: effect of g on memory*

time is such that to keep the memory usage low one needs to choose a low level tolerance. The lower the value of level tolerance, however, the higher the ILU factorization time, and the slower the convergence rate of the CGS algorithm. A small value of the level tolerance will thus result in an overall higher CPU cost to solve the system, as reflected in the results in Tables 5.2 through 5.7. The lowest total times to solve the system is always at the higher level tolerances, while the smallest memory usage occur at the lower level tolerances.

6. Conclusions. We have shown that constructing preconditioners based on a multifrontal incomplete LU factorization is feasible. These preconditioners are generally of good quality on a wide range of problems based on the increased convergence rate obtained when using them to accelerate the conjugate gradient squared iterative algorithm. The major difficulty in constructing these preconditioners is in controlling the memory usage. We were able to reduce the amount of memory used by limiting the sizes to which the frontal matrices can grow and by avoiding unnecessary pivot failures. Further reduction in the amount of memory used is possible by doing garbage collection to reduce internal fragmentation in the unassembled contribution block. However, the difference in memory usage between the lower and higher level tolerance settings is still only about 5 to 25 percent. We attribute this to the decreased assembly rate of unassembled contribution matrices.

7. Acknowledgements. Support for this project was provided by the National Science Foundation (DMS-9223088 and DMS-9504974), and by CRAY Research, Inc., through the allocation of supercomputing resources.

REFERENCES

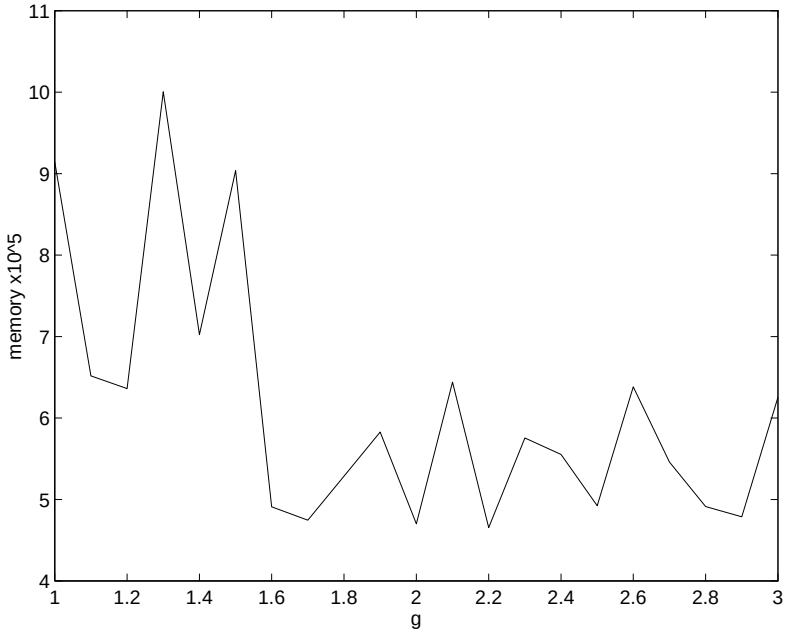

FIG. 5.5. SHERMAN3 *matrix: effect of g on memory*

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Analysis and Application*, (to appear). Also CISE Technical Report TR-94-039.

[2] O. Axelsson. A survey of preconditioned iterative methods for linear systems of algebraic equations. *Bit*, 25:166–187, 1985.

[3] O. Axelsson and N. Munksgaard. A class of preconditioned conjugate gardients methods for the solution of a mixed finite-element discretization of the biharmonic operator. *Int. J. Numer. Math. Eng.*, 14:1001–1019, 1978.

[4] O. Axelsson and N. Munksgaard. Analysis of incomplete factorizations with fixed storage allocation. In D. J. Evans, editor, *Preconditioning Methods: Analysis and Applications*, pages 219–241. Gordon and Breach, New York, 1983.

[5] Y. E. Campbell. *Multifrontal algorithms for sparse inverse subsets and incomplete LU factorization*. PhD thesis, Computer and Information Science and Engineering Department, Univ. of Florida, Gainesville, FL, November 1995. Also CISE Technical Report TR-95-025.

[6] T. A. Davis. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. Technical Report TR-94-005, University of Florida, Gainesville, FL, 1994.

[7] T. A. Davis and I. S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. Technical Report TR-95-020, Computer and Information Science and Engineering Department, Univ. of Florida, 1995.

[8] T. A. Davis and I. S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. Technical Report TR-95-020, University of Florida, Gainesville, FL, 1995.

[9] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM J. Matrix Analysis and Application*, (to appear). Also CISE Technical Report TR-94-038.

[10] J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[11] I.S. Duff and G. A. Meurant. The effect of ordering on preconditioned conjugate gradients. *BIT*, 29:635–657, April 1989.

[12] A. Jennings and G. A. Malik. Partial elimination. *J. Inst. Math. Applics.*, 20:307–316, 1977.

[13] M. T. Jones and P. E. Plassman. An improved Cholesky factorization. Technical Report Preprint MCS-P206-0191, Argonne National Laboratory, Argonne, Illinois, 1992.

[14] D. S. Kershaw. The incomplete Cholesky-conjugate gradient method for the iterative solution of systems of linear equations. *J. Comput. Phys.*, 26:43–65, 1978.

[15] J. Meijerink and A. Van Der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Mathematics of Computation*, 31:134–155, 1977.

[16] N. Munksgaard. Solving sparse symmetric sets of linear equations by preconditioned conjugate gardients. *ACM Transactions on Mathematical Software*, 6:206–219, 1980.

[17] P. Sonnveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 10:36–52, 1989.

[18] R. S. Varga. *Factorizations and normalized iterative methods, in Boundary Problems in Differential Equations (edited by R.E. Langer).* The University of Wisconsin Press, Madison, Wisconsin, 1960.

Note: all University of Florida technical reports in this list of references are available in postscript form via anonymous ftp to `ftp.cis.ufl.edu` in the directory `cis/tech-reports`, or via the World Wide Web at `http://www.cis.ufl.edu/~davis`.