AN ALGEBRAIC MULTILEVEL MULTIGRAPH ALGORITHM

RANDOLPH E. BANK* AND R. KENT SMITH†

Abstract. We describe an algebraic multilevel multigraph algorithm. Many of the multilevel components are generalizations of algorithms originally applied to general sparse Gaussian elimination. Indeed, general sparse Gaussian elimination with minimum degree ordering is a limiting case of our algorithm. Our goal is to develop a procedure which has the robustness and simplicity of use of sparse direct methods, yet offers the opportunity to obtain the optimal or near-optimal complexity typical of classical multigrid methods.

Key words. algebraic multigrid, incomplete LU factorization, multigraph methods

AMS subject classifications. 65M55, 65N55

1. Introduction. In this work, we develop a multilevel multigraph algorithm. Algebraic multigrid methods are currently a topic of intense research interest [17, 18, 20, 46, 12, 48, 38, 11, 44, 3, 4, 1, 2, 5, 16, 7, 29, 28, 27, 42, 41, 21]. An excellent recent survey is given in Wagner [49]. In many "real world" calculations, direct methods are still widely used [6]. The robustness of direct elimination methods and their simplicity of use often outweigh the apparent benefits of fast iterative solvers. Our goal here is to try to develop an iterative solver that can compete with sparse Gaussian elimination in terms of simplicity of use and robustness and to provide the potential of solving a wide range of linear systems more efficiently. While we are not yet satisfied that our method has achieved this goal, we believe that it is a reasonable first step. In particular, the method of general sparse Gaussian elimination with minimum degree ordering is a point in the parameter space of our method. This implies that in the worst case, our method defaults to this well-known and widely used method, among the most computationally efficient of general sparse direct methods [26]. In the best case, however, our method can exhibit the near optimal order complexity of the classical multigrid method.

Our plan is to take well studied, robust, and widely used procedures and data structures developed for sparse Gaussian elimination, generalize them as necessary, and use them as the basic components of our multilevel solver. The overall iteration follows the classical multigrid V-cycle in form, in contrast to the algebraic hierarchical basis multigraph algorithm developed in [11].

In this work we focus on the class of matrices which are structurally symmetric; that is, the pattern of nonzeros in the matrix is symmetric, although the numerical values of the matrix elements may render it nonsymmetric. Such structurally symmetric matrices arise in the discretizations of partial differential equations, say, by the finite element method. For certain problems, the matrices are symmetric and positive definite, but for others the linear systems are highly nonsymmetric and/or indefinite. Thus in practice this represents a very broad class of behavior. While our main interest is in scalar elliptic equations, as in the finite element code $PLTMG$ [8], our algorithms can formally be applied to any structurally symmetric, nonsingular, sparse matrix.

Sparse direct methods typically have two phases. In the first (initialization) phase,

*Department of Mathematics, University of California at San Diego, La Jolla, CA 92093. The work of this author was supported by the National Science Foundation under contract DMS-9706090.

†Bell Laboratories, Lucent Technologies, Murray Hill, NJ 07974.

equations are ordered, and symbolic and numerical factorizations are computed. In the second (solution) phase, the solution of the linear system is computed using the factorization. Our procedure, as well as other algebraic multilevel methods, also breaks naturally into two phases. The initialization consists of ordering, incomplete symbolic and numeric factorizations, and the computation of the transfer matrices between levels. In the solution phase, the preconditioner computed in the initialization phase is used to compute solution using the preconditioned composite step conjugate gradient (CSCG) or the composite step biconjugate gradient (CSBCG) method [9].

Iterative solvers often have tuning parameters and switches which require a certain level of a priori knowledge or some empirical experimentation to set in any particular instance. Our solver is not immune to this, although we have tried to keep the number of such parameters to a minimum. In particular, in the initialization phase, there are only three such parameters:

- ϵ, the drop tolerance used in the incomplete factorization (called *dtol* in our code).
- $maxfil$, an integer which controls to overall fill-in (storage) allowed in a given incomplete factorization.
- $maxlvl$, an integer specifying the maximum number of levels.

(The case $\epsilon = 0$, $maxfil = N$, $maxlvl = 1$ corresponds to sparse Gaussian elimination.) In the solution phase, there are only two additional parameters:

- tol, the tolerance used in the convergence test.
- $maxcg$, an integer specifying the maximum number of iterations.

Within our code, all matrices are generally treated within a single, unified framework; e.g., symmetric positive definite, nonsymmetric, and indefinite problems generally do not have specialized options. Besides the control parameters mentioned above, all information about the matrix is generated from the sparsity pattern and the values of the nonzeros, as provided in our sparse matrix data structure, a variant of the data structure introduced in the Yale sparse matrix package [23, 10]. For certain block matrices, the user may optionally provide a small array containing information about the block structure.

This input limits the complexity of the code, as well as eliminates parameters which might be needed to further classify a given matrix. On the other hand, it seems clear that a specialized solver directed at a specific problem or class of problems, and making use of this additional knowledge, is likely to outperform our algorithm on that particular class of problems. Although we do not think our method is provably "best" for any particular problem, we believe its generality and robustness, coupled with reasonable computational efficiency, make it a valuable addition to our collection of sparse solvers.

The rest of this paper is organized as follows. In section 2, we provide a general description of our multilevel approach. In section 3, we define the sparse matrix data structures used in our code. Our incomplete factorization algorithm is a standard drop tolerance approach with a few modifications for the present application. These are described in section 4. Our ordering procedure is the minimum degree algorithm. Once again, our implementation is basically standard with several modifications to the input graph relevant to our application. These are described in section 5. In section 6, we describe the construction of the transfer matrices used in the construction of the coarse grid correction. Information about the block structure of the matrix, if any is provided, is used only in the coarsening procedure. This is described in section 7. Finally, in section 8, we give some numerical illustrations of our method on a variety

of (partial differential equation) matrices.

2. Matrix formulation. Let A be a large sparse, nonsingular $N \times N$ matrix. We assume that the sparsity pattern of A is symmetric, although the numerical values need not be. We will begin by describing the basic two-level method for solving

$$Ax = b. \tag{2.1}$$

Let B be an $N \times N$ nonsingular matrix, called the *smoother*, which gives rise to the basic iterative method used in the multilevel preconditioner. In our case, B is an approximate factorization of A, i.e.,

$$B = (L + D)D^{-1}(D + U) \approx P^t AP, \tag{2.2}$$

where L is (strict) lower triangular, U is (strict) upper triangular with the same sparsity pattern as L^t, D is diagonal, and P is a permutation matrix.

Given an initial guess x_0, m steps of the smoothing procedure produce iterates x_k, $1 \le k \le m$, given by

$$\begin{aligned} r_{k-1} &= P^t(b - Ax_{k-1}), \\ B\delta_{k-1} &= r_{k-1}, \\ x_k &= x_{k-1} + P^t\delta_{k-1}. \end{aligned} \tag{2.3}$$

The second component of the two-level preconditioner is the *coarse grid correction*. Here we assume that the matrix A can be partitioned as

$$\hat{P}A\hat{P}^t = \begin{pmatrix} A_{ff} & A_{fc} \\ A_{cf} & A_{cc} \end{pmatrix}, \tag{2.4}$$

where the subscripts f and c denote *fine* and *coarse*, respectively. Similar to the smoother, the partition of A in fine and coarse blocks involves a permutation matrix $\hat{P}$. The $\hat{N} \times \hat{N}$ coarse grid matrix $\hat{A}$ is given by

$$\begin{aligned} \hat{A} &= \begin{pmatrix} V_{cf} & I_{cc} \end{pmatrix} \begin{pmatrix} A_{ff} & A_{fc} \\ A_{cf} & A_{cc} \end{pmatrix} \begin{pmatrix} W_{fc} \\ I_{cc} \end{pmatrix} \\ &= V_{cf}A_{ff}W_{fc} + V_{cf}A_{fc} + A_{cf}W_{fc} + A_{cc}. \end{aligned} \tag{2.5}$$

The matrices V_{cf} and W_{fc}^t are $\hat{N} \times (N - \hat{N})$ matrices with identical sparsity patterns; thus $\hat{A}$ has a symmetric sparsity pattern. If $A^t = A$, we require $V_{cf} = W_{fc}^t$, so $\hat{A}^t = \hat{A}$.

Let

$$\hat{V} = \begin{pmatrix} V_{cf} & I_{cc} \end{pmatrix} \hat{P}, \qquad \hat{W} = \hat{P}^t \begin{pmatrix} W_{fc} \\ I_{cc} \end{pmatrix}. \tag{2.6}$$

In standard multigrid terminology, the matrices $\hat{V}$ and $\hat{W}$ are called *restriction* and *prolongation*, respectively. Given an approximate solution x_m to (2.1), the coarse grid correction produces an iterate x_{m+1} as follows.

$$\begin{aligned} \hat{r} &= \hat{V}(b - Ax_m), \\ \hat{A}\hat{\delta} &= \hat{r}, \\ x_{m+1} &= x_m + \hat{W}\hat{\delta}. \end{aligned} \tag{2.7}$$

As is typical of multilevel methods, we define the *two-level preconditioner* M implicitly in terms of the smoother and coarse grid correction. A single cycle takes an initial guess x_0 to a final guess x_{2m+1} as follows:

Two-Level Preconditioner

(i) x_k for $1 \le k \le m$ are defined using (2.3).
(ii) x_{m+1} is defined using (2.7).
(iii) x_k for $m+2 \le k \le 2m+1$ are defined using (2.3).

The generalization from two-level to multilevel consists of applying recursion to the solution of the equation $\hat{A}\hat{\delta} = \hat{r}$ in (2.7). Let ℓ denote the number of levels in the recursion. Let $\hat{M} \equiv \hat{M}(\ell)$ denote the preconditioner for $\hat{A}$; if $\ell = 2$, then $\hat{M} = \hat{A}$. Then (2.7) is generalized to

$$\begin{aligned} \hat{r} &= \hat{V}(b - Ax_m), \\ \hat{M}\hat{\delta} &= \hat{r}, \\ x_{m+1} &= x_m + \hat{W}\hat{\delta}. \end{aligned} \tag{2.8}$$

The general ℓ level preconditioner M is then defined as follows:

ℓ-Level Preconditioner

(i) if $\ell = 1$, $M = A$; i.e., solve (2.1) directly.
(ii) if $\ell > 1$, then, starting from initial guess x_0, compute x_{2m+1} using (iii)–(v):
(iii) x_k for $1 \le k \le m$ are defined using (2.3).
(iv) x_{m+1} is defined by (2.8), using $p = 1$ or $p = 2$ iterations of the $\ell - 1$ level scheme for $\hat{A}\hat{\delta} = \hat{r}$ to define $\hat{M}$, and with initial guess $\hat{\delta}_0 = 0$.
(v) x_k for $m+2 \le k \le 2m+1$ are defined using (2.3).

The case $p = 1$ corresponds to the symmetric *V-cycle*, while the case $p = 2$ corresponds to the symmetric *W-cycle*. We note that there are other variants of both the V-cycle and the W-cycle, as well as other types of multilevel cycling strategies [30]. However, in this work (and in our code) we restrict attention to just the symmetric V-cycle with $m = 1$ presmoothing and postsmoothing iterations.

For the coarse mesh solution ($\ell = 1$), our procedure is somewhat nontraditional. Instead of a direct solution of (2.1), we compute an approximate solution using one smoothing iteration. We illustrate the practical consequences of this decision in section 8.

If A is symmetric, then so is M, and the ℓ-level preconditioner could be used as a preconditioner for a symmetric Krylov space method. If A is also positive definite, so is M, and the standard conjugate gradient method could be used; otherwise the CSCG method [9], SYMLQ [43], or a similar method could be used. In the nonsymmetric case, the ℓ-level preconditioner could be used in conjunction with the CSBCG method [9], GMRES [22], or a similar method.

To complete the definition of the method, we must provide algorithms to

- compute the permutation matrix P in (2.2);
- compute the incomplete factorization matrix B in (2.2);
- compute the fine-coarse partitioning ($\hat{P}$) in (2.4);
- compute the sparsity patterns and numerical values in the prolongation and restriction matrices in (2.6).

3. Data structures. Let A be an $N \times N$ matrix with elements A_{ij} and a symmetric sparsity structure; that is, both A_{ij} and A_{ji} are treated as nonzero elements (i.e. stored and processed) if $|A_{ij}| + |A_{ji}| > 0$. All diagonal entries A_{ii} are treated as nonzero regardless of their numerical values.

Our data structure is a modified and generalized version of the data structure introduced in the (symmetric) Yale sparse matrix package [23]. It is a rowwise version of the data structure described in [10]. In our scheme, the nonzero entries of A are

stored in a linear array a and accessed through an integer array ja. Let η_i be the number of nonzeros in the strict upper triangular part of row i and set $\eta = \sum_{i=1}^{N} \eta_i$. The array ja is of length $N+1+\eta$, and the array a is of length $N+1+\eta$ if $A^t = A$. If $A^t \neq A$, then the array a is of length $N+1+2\eta$. The entries of $ja(i)$, $1 \leq i \leq N+1$, are pointers defined as follows:

$$\begin{aligned} ja(1) &= N+2, \\ ja(i+1) &= ja(i) + \eta_i, \quad 1 \leq i \leq N. \end{aligned}$$

The locations $ja(i)$ to $ja(i+1)-1$ contain the η_i column indices corresponding to the row i in the strictly upper triangular matrix.

In a similar manner, the array a is defined as follows:

$$\begin{aligned} a(i) &= A_{ii}, \quad 1 \leq i \leq N, \\ a(N+1) &\quad \text{is arbitrary}, \\ a(k) &= A_{ij}, \quad 1 \leq i \leq N, \quad j = ja(k), \quad ja(i) \leq k \leq ja(i+1)-1. \end{aligned}$$

If $A^t \neq A$, then

$$a(k+\eta) = A_{ji}, \quad 1 \leq i \leq N, \quad j = ja(k), \quad ja(i) \leq k \leq ja(i+1)-1.$$

In words, the diagonal is stored first, followed by the strict upper triangle stored rowwise. If $A^t \neq A$, then this is followed by the strict lower triangle stored columnwise. Since A is structurally symmetric, the column indexes for the upper triangle are identical to the row indexes for the lower triangle, and hence they need not be duplicated in storage.

As an example, let

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} & 0 & 0 \\ A_{21} & A_{22} & 0 & A_{24} & 0 \\ A_{31} & 0 & A_{33} & A_{34} & A_{35} \\ 0 & A_{42} & A_{43} & A_{44} & 0 \\ 0 & 0 & A_{53} & 0 & A_{55} \end{pmatrix}.$$

Then

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ja	7	9	10	12	12	12	2	3	4	4	5					
a	A_{11}	A_{22}	A_{33}	A_{44}	A_{55}		A_{12}	A_{13}	A_{24}	A_{34}	A_{35}	A_{21}	A_{31}	A_{42}	A_{43}	A_{53}
	Diagonal						Upper triangle					Lower triangle				

Although the YSMP data structure was originally devised for sparse direct methods based on Gaussian elimination, it is also quite natural for iterative methods based on incomplete triangular decomposition. Because we assume that A has a symmetric sparsity structure, for many matrix calculations a single indirect address computation in ja can be used to process both a lower and a upper triangular element in A. For example, the following procedure computes $y = Ax$:

procedure *mult(N, ja, a, x, y)*
    $lmtx \leftarrow ja(N+1) - ja(1)$
    $umtx \leftarrow 0$
    for $i \leftarrow 1$ to N
        $y(i) \leftarrow a(i)x(i)$

```
        end for
        for i ← 1 to N
            for k ← ja(i) to ja(i+1) − 1
                j ← ja(k)
                y(i) ← y(i) + a(k + umtx)x(j)
                y(j) ← y(j) + a(k + lmtx)x(i)
            end for
        end for
    end mult
```

For symmetric matrices, set $lmtx \leftarrow 0$, $umtx \leftarrow 0$. Also, $y = A^t\, x$ may be readily computed by setting $lmtx \leftarrow 0$, $umtx \leftarrow ja(N+1) - ja(1)$.

The data structure for storing $B = (L+D)D^{-1}(D+U)$ is quite analogous to that for A. It consists of two arrays, ju and u, corresponding to ja and a, respectively. The first $N+1$ entries of ju are pointers as in ja, while entries $ju(i)$ to $ju(i+1)-1$ contain column indices of the nonzeros of row i in of U. In the u array, the diagonal entries of D are stored in the first N entries. Entry $N+1$ is arbitrary. Next, the nonzero entries of U are stored in correspondence to the column indices in ju. If $L^t \neq U$, the nonzero entries of L follow, stored columnwise.

The data structure we use for the $N \times \hat{N}$ matrix $\hat{W}$ and the $\hat{N} \times\ N$ matrix $\hat{V}$ are similar. It consists of an integer array jv and a real array v. The nonzero entries of $\hat{W}$ are stored rowwise, including the rows of the block I_{cc}. As usual, the first $N+1$ entries of jv are pointers; entries $jv(i)$ to $jv(i+1)-1$ contain column indices for row i of $\hat{W}$. In the v array, the nonzero entries of $\hat{W}$ are stored rowwise in correspondence with jv but shifted by $N+1$ since there is no diagonal part. If $\hat{V}^t \neq \hat{W}$, this is followed by the nonzeros of $\hat{V}$, stored columnwise.

4. ILU factorization. Our incomplete $(L+D)D^{-1}(D+U)$ factorization is similar to the row elimination scheme developed for the symmetric YSMP codes [23, 26]. For simplicity, we begin by discussing a complete factorization and then describe the modifications necessary for the incomplete factorization. Without loss of generality, assume that the permutation matrix $P = I$, so that $A = (L+D)D^{-1}(D+U)$.

After k steps of elimination, we have the block factorization

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} D_{11}+L_{11} & 0 \\ L_{21} & I \end{pmatrix} \begin{pmatrix} D_{11}^{-1} & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} D_{11}+U_{11} & U_{12} \\ 0 & I \end{pmatrix}, \tag{4.1}$$

where A_{11} is $k \times k$ and A_{22} is $N-k \times N-k$. We assume that at this stage, all the blocks on the right-hand side of (4.1) have been computed except for the Schur complement S, given by

$$S = A_{22} - L_{21}D_{11}^{-1}U_{12}. \tag{4.2}$$

Our goal for step $k+1$ is to compute the first row and column of S, given by

$$\begin{aligned} Se_1 &= A_{22}e_1 - L_{21}(D_{11}^{-1}U_{12}e_1), \\ S^te_1 &= A_{22}^te_1 - U_{12}^t(D_{11}^{-1}L_{21}^te_1). \end{aligned} \tag{4.3}$$

Because A and $(L+D)D^{-1}(D+U)$ have symmetric sparsity patterns, and our data structures take advantage of this symmetry, it is clear that the algorithms for computing Se_1 and S^te_1 are the same and in practice differ only in the assignments of shifts for the u and a arrays, analogous to $lmtx$ and $umtx$ in procedure mult. Thus

we will focus on the computation of just Se_1. At this point, we also assume that the array ju has been computed in a so-called *symbolic factorization* step.

The major substeps are as follows:

1. Copy the first column of A_{22} (stored in the data structures ja and a) into an expanded work vector z of size N.
2. Find the *multipliers* given by nonzeros of $D_{11}^{-1}U_{12}e_1$.
3. For each multiplier $\gamma = e_k^t D_{11}^{-1}U_{12}e_1$, update z using column k of L_{21} (i.e., $\gamma L_{21}e_k$).
4. Copy the nonzeros in z into the data structures ju and u.

In step 1, we need to know the nonzeros of the first column of A_{22}, which is precisely the information easily accessible in the ja and a data structures. In step 3, we need to know the nonzeros in columns of L_{21}, which again is precisely the information easily available in our data structure. In step 4, we copy a column of information into the lower triangular portion of the ju and u data structures. Indeed, the only difficult aspect of the algorithm is step 2, in which we need to know the sparsity structure of the first *column* of U_{12}, information that is *not* readily available in the data structure. This is handled in a standard fashion using a dynamic linked list structure and will not be discussed in detail here.

To generalize this to the incomplete factorization case, we first observe that the ju array can be computed concurrently with the numeric factorization simply by creating a list of the entries of the expanded array z that are updated in step 3. Next, we note that one may choose which nonzero entries from z to include in the factorization by choosing which entries to copy to the ju and u data structures in step 4. We do this through a standard approach using a drop tolerance ϵ. In particular, we neglect a pair of off-diagonal elements if

$$\max |L_{ij}|, |U_{ji}| \leq \epsilon \sqrt{|D_{jj}A_{ii}|}, \tag{4.4}$$

$j = k + 1$ and $i > j$. Note D_{ii} has not yet been computed. It is well known that the fill-in generated through the application of a criterion such as (4.4) is a highly nonlinear and matrix dependent function of ϵ. This is especially problematic in the present context, since control of the fill-in is necessary in order to control the work per iteration in the multilevel iteration.

Several authors have explored possibilities of controlling the maximum number of fill-in elements allowed in each row of the incomplete decomposition [35, 47, 31]. However, for many cases of interest, and in particular for matrices arising from discretizations of partial differential equations ordered by the minimum degree algorithm, most of the fill-in in a complete factorization occurs in the later stages, even if all the rows initially have about the same number of nonzeros. Thus while it seems advisable to try to control the total fill-in, one should adaptively decide how to allocate the fill-in among the rows of the matrix. In our algorithm, in addition to the drop tolerance ϵ, the user provides a parameter $maxfil$, which specifies that the total number of nonzeros in U is not larger than $maxfil \cdot N$.

Our overall strategy is to compute the incomplete decomposition using the given drop tolerance. If it fails to meet the given storage bound, we increase the drop tolerance and begin a new incomplete factorization. We continue in this fashion until we complete a factorization within the given storage bound. Of course, such repeated factorizations are computationally expensive, so we developed some heuristics which allow us to predict a drop tolerance which will satisfy the storage bound.

As the factorization is computed, we make a histogram of the approximate sizes of all elements that exceed the drop tolerance and are accepted for the factorization. Let m denote the number of bins in the histogram; $m = 400$ in our code. Then for each pair of accepted off-diagonal elements, we find the largest $k \in [1, m]$ such that

$$\rho^{k-1} \leq \frac{\max |L_{ij}|, |U_{ji}|}{\epsilon \sqrt{|D_{jj} A_{ii}|}}. \tag{4.5}$$

Here $\rho > 1$ ($\rho = 10^{4/m}$ in our code). The histogram is realized as an integer array h of size m, where h_ℓ is the number of accepted elements that exceeded the drop tolerance by factors between $\rho^{\ell-1}$ and ρ^ℓ for $1 \leq \ell \leq m-1$; h_m contains the number of accepted elements exceeding the drop tolerance by ρ^{m-1}. If the factorization reaches the storage bound, we continue the factorization but allow no further fill-in. However, we continue to compute the histogram based on (4.5), profiling the elements we would have accepted had space been available. Then using the histogram, we predict a new value of ϵ such that the total number of elements accepted for U is no larger than $maxfil \cdot N/\theta$. Such a prediction of course cannot be guaranteed, since the sizes and numbers of fill-in elements depend in a complicated fashion on the specific history of the incomplete factorization process; indeed, the histogram cannot even completely profile the remainder of the factorization with the existing drop tolerance, since elements that would have been accepted could introduce additional fill-in at later stages of the calculation as well as influence the sizes of elements computed at later stages of the factorization. In our implementation, the factor θ varies between $\theta = 1.01$ and $\theta = 1.4$, depending on how severely the storage bound was exceeded. Its purpose is to introduce some conservative bias into the prediction with the goal that the actual fill-in accepted should not exceed $maxfil \cdot N$.

Finally, we note that there is no comprehensive theory regarding the stability of incomplete triangular decompositions. For certain classes of matrices (e.g., M-matrices and H-matrices), the existence of certain incomplete factorizations has been proved [39, 25, 24, 40, 51]. However, in the general case, with potentially indefinite and/or highly nonsymmetric matrices, one must contend in a practical way with the possibility of failure or near failure of the factorization. A common approach is to add a diagonal matrix, often a multiple of the identity, to A and compute an incomplete factorization of the shifted matrix. One might also try to incorporate some form of diagonal pivoting; partial or complete pivoting could potentially destroy the symmetric sparsity pattern of the matrix. However, any sort of pivoting greatly increases the complexity of the implementation, since the simple but essentially static data structures ja, a, ju, and u are not appropriate for such an environment.

Our philosophy here is to simply accept occasional failures and continue with the factorization. Our ordering procedure contains some heuristics directed towards avoiding or at least minimizing the possibility of failures. And when they do occur, failures often corrupt only a low dimensional subspace, so a Krylov space method such as conjugate gradients can compensate for such corruption with only a few extra iterations. In our implementation, a failure is revealed by some diagonal entries in D becoming close to zero. Off-diagonal elements L_{ji} and U_{ij} are multiplied by D_{ii}^{-1}, and the solution of $(L + D)D^{-1}(D + U)x = b$ also involves multiplication by D_{ii}^{-1}. For purposes of calculating the factorization and solution, the value of D_{ii}^{-1} is modified near zero as follows:

$$D_{ii}^{-1} = \begin{cases} 1/D_{ii} & \text{for } |D_{ii}| > \alpha, \\ D_{ii}/\alpha^2 & \text{for } |D_{ii}| \leq \alpha. \end{cases} \tag{4.6}$$

Here α is a small constant; in our implementation, $\alpha = \mu\|A\|$, where μ is the machine epsilon. Although many failures could render the preconditioner well-defined but essentially useless, in practice we have noted that D_{ii}^{-1} is rarely modified for the large class of finite element matrices which are the main target of our procedure.

5. Ordering. To compute the permutation matrix P in (2.2), we use the well-known minimum degree algorithm [45, 26]. Intuitively, if one is computing an incomplete factorization, an ordering which tends to minimize the fill-in in a complete factorization should tend to minimize the error

$$E = P^t AP - (L + D)D^{-1}(D + U).$$

For particular classes of matrices, specialized ordering schemes have been developed [34, 15, 37, 36]. For example, for matrices arising from convection dominated problems, ordering along the flow direction has been used with great success. However, in this general setting, we prefer to use just one strategy for all matrices. This reduces the complexity of the implementation and avoids the problem of developing heuristics to decide among various ordering possibilities. We remark that for convection dominated problems, minimum degree orderings perform comparably well to the specialized ones, provided some (modest) fill-in is allowed in the incomplete factorization. For us, this seems to be a reasonable compromise.

Our minimum degree ordering is a standard implementation, using the quotient graph model [26] and other standard enhancements. A description of the *graph* of the matrix is the main required input. Without going into detail, this is essentially a small variant of the basic ja data structure used to store the matrix A. We will denote this modified data structure as jc. Instead of storing only column indices for the strict upper triangle as in ja, entries $jc(i)$ to $jc(i+1)-1$ of the jc data structure contain column indices for *all* off-diagonal entries for row i of the matrix A.

We have implemented two small enhancements to the minimum degree ordering; as a practical matter, both involve changes to the input graph data structure jc that is provided to the minimum degree code. First, we have implemented a drop tolerance similar to that used in the the factorization. In particular the edge in the graph corresponding to off-diagonal entries A_{ij} and A_{ji} is not included in the jc data structure if

$$\max |A_{ij}|, |A_{ji}| \leq \epsilon \sqrt{|A_{jj}A_{ii}|}. \tag{5.1}$$

This excludes many entries which are likely to be dropped in the subsequent incomplete factorization and hopefully will result in an ordering that tends to minimize the fill-in created by the edges that are kept.

The second modification involves some modest a priori diagonal pivoting designed to minimize the number failures (near zero diagonal elements) in the subsequent factorization. We first remark that pivoting or other procedures based on the values of the matrix elements (which can be viewed as weights on graph edges and nodes) would destroy many of the enhancements which allow the minimum degree algorithm to run in almost linear time. Our modification is best explained in the context of a simple 2×2 example. Let

$$A = \begin{pmatrix} 0 & c \\ b & a \end{pmatrix}$$

with $a, b, c \neq 0$. Clearly, A is nonsingular, but the complete triangular factorization of A does not exist. However,

$$P^t A P = \begin{pmatrix} a & b \\ c & 0 \end{pmatrix} = \begin{pmatrix} a & 0 \\ c & -bc/a \end{pmatrix} \begin{pmatrix} 1/a & 0 \\ 0 & -a/bc \end{pmatrix} \begin{pmatrix} a & b \\ 0 & -bc/a \end{pmatrix}. \tag{5.2}$$

Now suppose that $A_{ii} \approx 0$, $A_{jj}, A_{ij}, A_{ji} \neq 0$. Then these four elements form a submatrix of the form described above, and it seems an incomplete factorization of A is less likely to fail if the P is chosen such that vertex j is ordered before vertex i. This is done as follows: for each i such that $A_{ii} \approx 0$, we determine a corresponding j such that $A_{jj}, A_{ij}, A_{ji} \neq 0$; if there is more than one choice, we choose the one for which $|A_{ij}A_{ji}/A_{jj}|$ is maximized. To ensure that vertex i is ordered after vertex j, we replace the sparsity pattern for the off-diagonal entries for row (column) i with the union of those for rows (columns) i and j. If we denote the set of column indices for row i in the jc array as $adj(i)$, then

$$adj(j) \cup \{j\} \subseteq adj(i) \cup \{i\}. \tag{5.3}$$

Although the sets $adj(i)$ and $adj(j)$ are modified at various stages, it is well known that (5.3) is maintained throughout the minimum degree ordering process [26], so that at every step of the ordering process $deg(j) \leq deg(i)$, where $deg(i)$ is the degree of vertex i. As long as $deg(j) < deg(i)$, vertex j will be ordered before vertex i by the minimum degree algorithm. On the other hand, if $deg(i) = deg(j)$ at some stage of the ordering process, it remains so thereafter, and (5.3) becomes

$$adj(j) \cup \{j\} = adj(i) \cup \{i\}. \tag{5.4}$$

In words, i and j become so-called *equivalent vertices* and will be eliminated at the same time by the minimum degree algorithm (see [26] for details). Since the minimum degree algorithm sees these vertices as equivalent, they will be ordered in an arbitrary fashion when eliminated from the graph. Thus, as a simple postprocessing step, we must scan the ordering provided by the minimum degree algorithm and exchange the order of rows i and j if i was ordered first. Any such exchanges result in a new minimum degree ordering which is completely equivalent, in terms of fill-in, to the the original.

For many types of finite element matrices (e.g., the indefinite matrices arising from Helmholtz equations), this a priori scheme is useless because none of the diagonal entries of A is close to zero. However, this type of problem is likely to produce only isolated small diagonal entries in the factorization process, if it produces any at all. On the other hand, other classes of finite element matrices, notably those arising in from mixed methods, Stokes equations, and other saddle-point-like formulations, have many diagonal entries that are small or zero. In such cases, the a priori diagonal pivoting strategy can make a substantial difference and greatly reduce the numbers of failures in the incomplete triangular decomposition.

6. Computing the transfer matrices. There are three major tasks in computing the prolongation and restriction matrices $\hat{V}$ and $\hat{W}$ of (2.6). First, one must determine the sparsity structure of these matrices; this involves choosing which unknowns are *coarse* and which are *fine*. This reduces to determining the permutation matrix $\hat{P}$ of (2.4). Second, one must determine how coarse and fine unknowns are related, the so-called *parent-child relations* [49]. This involves computing the sparsity

patterns for the matrices V_{cf} and W_{fc}. Third, one must compute the numerical values for these matrices, the so-called *interpolation coefficients* [50].

There are many existing algorithms for coarsening graphs. For matrices arising from discretizations of partial differential equations, often the sparsity of the matrix A is related in some way to the underlying grid, and the problem of coarsening the *graph* of the matrix A can be formulated in terms of coarsening the *grid*. Some examples are given in [14, 13, 17, 18, 46, 12, 49]. In this case, one has the geometry of the grid to serve as an aid in developing and analyzing the coarsening procedure. There are also more general graph coarsening algorithms [32, 33, 19], often used to partition problems for parallel computation. Here our coarsening scheme is based upon another well-known sparse matrix ordering technique, the reverse Cuthill–McKee algorithm. This ordering tends to yield reordered matrices with minimal bandwidth and is widely used with generalized band elimination algorithms [26]. We now assume that the graph has been ordered in this fashion and that a jc data structure representing the graph in this ordering is available. Our coarsening procedure is just a simple postprocessing step of the basic ordering routine, in which the N vertices of graph are marked as $COARSE$ or $FINE$.

```
procedure coarsen(N, jc, type)
    for i ← 1 to N
        type(i) ← UNDEFINED
    end for
    for i ← 1 to N
        if type(i) = UNDEFINED, then
            type(i) ← COARSE
            for j ← jc(i) to jc(i + 1) − 1
                type(jc(j)) ← FINE
            end for
        end if
    end for
end coarsen
```

This postprocessing step, coupled with the the reverse Cuthill–McKee algorithm, is quite similar to a greedy algorithm for computing maximal independent sets using breadth-first search. Under this procedure, all coarse vertices are surrounded only by fine vertices. This implies that the matrix A_{cc} in (2.4) is a diagonal matrix. For the sparsity patterns of matrices arising from discretizations of scalar partial differential equations in two space dimensions, the number of coarse unknowns $\hat{N}$ is typically on the order of $N/4$ to $N/5$. Matrices with more nonzeros per row tend to have smaller values of $\hat{N}$. To define the parents of a coarse vertex, we take all the connections of the vertex to other fine vertices; that is, the sparsity structure of V_{cf} in (2.5) is the same as that of the block A_{cf}.

In our present code, we pick V_{cf} and W_{fc} according to the formulae

$$\begin{aligned} W_{fc} &= -R_{ff} D_{ff}^{-1} A_{fc}, \\ V_{cf} &= -A_{cf} D_{ff}^{-1} \tilde{R}_{ff}. \end{aligned} \tag{6.1}$$

Here D_{ff} is a diagonal matrix with diagonal entries equal to those of A_{ff}. In this sense, the nonzero entries in V_{cf} and W_{fc} are chosen as multipliers in Gaussian elimination. The nonnegative diagonal matrices R_{ff} and $\tilde{R}_{ff}$ are chosen such that nonzero rows of W_{fc} and columns of V_{cf}, respectively, have unit norms in ℓ_1.

Finally, the coarsened matrix $\hat{A}$ of (2.5) is "sparsified" using the drop tolerance and a criterion like (5.1) to remove small off-diagonal elements. Empirically, applying a drop tolerance to $\hat{A}$ at the end of the coarsening procedure has proved more efficient, and more effective, than trying to independently sparsify its constituent matrices. If the number of off-diagonal elements in the upper triangle exceeds $maxfil \cdot \hat{N}$, the drop tolerance is modified in a fashion similar to the incomplete factorization. The off-diagonal elements are profiled by a procedure similar to that for the incomplete factorization, but in this case the resulting histogram is exact. Based on this histogram, a new drop tolerance is computed, and (5.1) is applied to produce a coarsened matrix satisfying the storage bound.

7. Block matrices. Our algorithm provides a simple but limited functionality for handling block matrices. Suppose that the $N \times N$ matrix A has the $K \times K$ block structure

$$A = \begin{pmatrix} A_{11} & \dots & A_{1K} \\ \vdots & \ddots & \vdots \\ A_{K1} & \dots & A_{KK} \end{pmatrix}, \tag{7.1}$$

where subscripts for A_{ij} are block indices and the diagonal blocks A_{jj} are square matrices. Suppose A_{jj} is of order N_j; then $\sum_{j=1}^{K} N_j = N$.

The matrix A is stored in the usual ja and a data structures as described in section 3 with no reference to the block structure. A small additional integer array ib of size $K+1$ is used to define the block boundaries as follows:

$$\begin{aligned} ib(1) &= 1, \\ ib(j+1) &= ib(j) + N_j, \quad 1 \le j \le K. \end{aligned}$$

In words, integers in the range $ib(j)$ to $ib(j+1)-1$, inclusive, comprise the index set associated with block A_{jj}. Note that $ib(K+1) = N+1$.

This block information plays a role only in the coarsening algorithm. First, the reverse Cuthill–McKee algorithm described in section 6 is applied to the block diagonal matrix

$$\bar{A} = \begin{pmatrix} A_{11} & & \\ & \ddots & \\ & & A_{KK} \end{pmatrix} \tag{7.2}$$

rather than A. As a practical matter, this involves discarding graph edges connecting vertices of different blocks in the construction of the graph array jc used as input. Such edges are straightforward to determine from the information provided in the ib array. The coarsening algorithm applied to the graph of $\bar{A}$ produces output equivalent to the application of the procedure independently to each diagonal block of $\bar{A}$. As a consequence, the restriction and prolongation matrices automatically inherit the block structure A. In particular,

$$\hat{V} = \begin{pmatrix} \hat{V}_{11} & & \\ & \ddots & \\ & & \hat{V}_{KK} \end{pmatrix} \quad \text{and} \quad \hat{W} = \begin{pmatrix} \hat{W}_{11} & & \\ & \ddots & \\ & & \hat{W}_{KK} \end{pmatrix}, \tag{7.3}$$

where $\hat{V}_{jj}$ and $\hat{W}_{jj}$ are are rectangular matrices ($\hat{N}_j \times N_j$ and $N_j \times \hat{N}_j$, respectively), having the structure of (2.6) that would have resulted from the application of the

algorithm independently to A_{jj}. However, like the matrix A, $\hat{V}$ and $\hat{W}$ are stored in the standard jv and v data structures described in section 3 without reference to their block structures.

The complete matrix A is used in the construction of the coarsened matrix $\hat{A}$ of (2.5). However, because of (7.1) and (7.3)

$$\hat{A} = \hat{V} A \hat{W} = \begin{pmatrix} \hat{A}_{11} & \dots & \hat{A}_{1K} \\ \vdots & \ddots & \vdots \\ \hat{A}_{K1} & \dots & \hat{A}_{KK} \end{pmatrix},$$

so $\hat{A}$ also automatically inherits the $K \times K$ block structure of A. It is not necessary for the procedure forming $\hat{A}$ to have any knowledge of its block structure, as this block structure can be computed a priori by the graph coarsening procedure. Like A, $\hat{A}$ is stored in standard ja and a data structures without reference to its block structure. Since the blocks of A have arbitrary order, and are essentially coarsened independently, it is likely that eventually some of the $\hat{N}_j = 0$. That is, certain blocks may cease to exist on coarse levels. Since the block information is used only to discard certain edges in the construction of the graph array jc, "0×0" diagonal blocks present no difficulty.

8. Numerical experiments. In this section, we present a few numerical illustrations. In our first sequence of experiments, we consider several matrices loosely based on the classical case of 5-point centered finite difference approximations to $-\Delta u$ on a uniform square mesh. Dirichlet boundary conditions are imposed. This leads to the $n \times n$ block tridiagonal system

$$A = \begin{pmatrix} T & -I & & & \\ -I & T & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & T & -I \\ & & & -I & T \end{pmatrix}$$

with T the $n \times n$ tridiagonal matrix

$$T = \begin{pmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{pmatrix}.$$

This is a simple test problem easily solved by standard multigrid methods. In contrast to this example we also consider the block tridiagonal system

$$\bar{A} = 8I - A.$$

Both A and $\bar{A}$ have the same eigenvectors and the same eigenvalues, although the association of eigenvectors and eigenvalues are reversed in the case of $\bar{A}$. That is, the so-called *smooth* eigenvectors are associated with large eigenvalues, while *rough* eigenvectors are associated with smaller eigenvalues. Although $\bar{A}$ does not arise naturally in the context of numerical discretizations of partial differential equations, it is of interest because it defies much of the conventional wisdom for multigrid methods.

Third, we consider block 3×3 systems of the form

$$S = \begin{pmatrix} A & 0 & C_x \\ 0 & A & C_y \\ C_x^t & C_y^t & -D \end{pmatrix},$$

where A is the discrete Laplacian and D is a symmetric positive definite "stabilization" matrix with a sparsity pattern similar to A. However, the nonzeros in D are of size $O(h^2)$, compared to size $O(1)$ nonzero elements in A. C_x and C_y also have sparsity patterns similar to that of A, but these matrices are nonsymmetric and their nonzero entries are of size $O(h)$. Such matrices arise in stabilized discretizations of the Stokes equations. One third of the eigenvalues of S are negative, so S is quite indefinite. In addition to the ja and a arrays, for the matrix S we also provided an ib array as described in section 7 to define its 3×3 block structure. We emphasize again that this block information is used only in the computation of the graph input to the coarsening procedure and is not involved in any aspect of the incomplete factorization smoothing procedure. With many small diagonal elements, this class of matrices provides a good test of the a priori pivoting strategy used in conjunction with the minimum degree ordering.

In Table 8.1, *Levels* refers to the number of levels used in the calculation. In our implementation the parameter $maxlvl$, which limits the number of levels allowed, was set sufficiently large that it had no effect on the computation. The drop tolerance was set to $\epsilon = 10^{-2}$ for all matrices. The fill-in control parameter $maxfil$ was set sufficiently large that it had no effect on the computation. The initial guess for all problems was $x_0 = 0$.

In Table 8.1, the parameter *Digits* refers to

$$Digits = -\log \frac{\|r_k\|}{\|r_0\|}. \tag{8.1}$$

In these experiments, we asked for six digits of accuracy. The column labeled *Cycles* indicates the number of multigrid cycles (accelerated by CSCG) that were used to achieve the indicated number of digits. Finally, the last two columns, labeled *Init.* and *Solve*, record the CPU time, measured in seconds, for the initialization and solution phases of the algorithm, respectively. Initialization includes all the orderings, incomplete factorizations, and computation of transfer matrices used in the multigraph preconditioner. Solution includes the time to solve (2.1) to at least six digits given the preconditioner. These experiments were run on an SGI Octane R10000 250mhz, using double precision arithmetic and the f90 compiler.

In analyzing these results, it is clear that our procedure does reasonably well on all three classes of matrices. Although it appears that the rate of convergence is not independent of N, it seems apparent that the work is growing no faster than logarithmically. CPU times for larger vales of N are affected by cache performance as well as the slightly larger number of cycles.

For the highly indefinite Stokes matrices S, it is important to also note the robustness, that the procedure solved all of the problems. With more nonzeros per row on average, the incomplete factorization was more expensive to compute than for the other cases. This is reflected in relatively larger initialization and solve times.

In our next experiment, we illustrate the effect of the parameters $maxlvl$ and ϵ. For the matrix A with $N = 160000$, we solved the problem for $\epsilon = 10^{-k}$, $1 \le k \le 3$, and $1 \le maxlvl \le 7$. We terminated the iteration when the solution had six digits,

TABLE 8.1
Performance comparison.

n	N	Levels	Digits	Cycles	Init.	Solve
Discrete Laplacian A, $\epsilon = 10^{-2}$						
10	100	6	6.3	2	4.4e-3	1.2e-3
20	400	7	8.2	3	2.1e-2	6.9e-3
40	1600	8	8.6	4	9.4e-2	3.7e-2
80	6400	8	6.6	4	4.1e-1	2.0e-1
160	25600	9	6.9	5	1.9e 0	1.2e 0
320	102400	11	7.1	6	9.6e 0	7.4e 0
$\bar{A} = 8I - A$, $\epsilon = 10^{-2}$						
10	100	6	8.8	2	4.2e-3	1.2e-3
20	400	7	6.3	2	1.9e-2	5.0e-3
40	1600	8	8.1	3	9.2e-2	3.0e-2
80	6400	8	7.2	3	4.0e-1	1.6e-1
160	25600	9	6.8	3	1.9e 0	7.9e-1
320	102400	11	6.6	3	9.5e 0	4.2e 0
Stokes matrix S, $\epsilon = 10^{-2}$						
10	300	6	7.4	2	3.0e-2	5.3e-3
20	1200	7	8.2	3	2.3e-1	4.5e-2
40	4800	8	7.9	5	1.5e 0	5.1e-1
80	19200	9	6.5	5	8.1e 0	2.6e 0
160	76800	9	6.0	8	41.4e 0	20.6e 0

as measured by (8.1). We also provide the total storage for the ja and ju arrays for all matrices, measured in thousands of entries. Since the matrices are symmetric, this is also the total (floating point) storage for all matrices A and approximate LDU factorizations.

Here we see that our method behaves in a very predictable way. In particular, decreasing the drop tolerance or increasing the number of levels improves the convergence behavior of the method. On the other hand, the timings do not always follow the same trend. For example, for the case $\epsilon = 10^{-3}$ increasing the number of levels from $maxlvl = 1$ to $maxlvl = 2$ decreases the number of cycles but increases the time. This is because for $maxlvl = 1$, our method defaults to the standard conjugate gradient iteration with the incomplete factorization preconditioner. When $maxlvl > 1$, one presmoothing and one postsmoothing step are used for the largest matrix. With the additional cost of the recursion, the overall cost of the preconditioner is more than double the cost for the case $maxlvl = 1$.

We also note that, unlike the classical multigrid method, where the coarsest matrix is solved exactly, in our code we have chosen to approximately solve the coarsest system using just one smoothing iteration using the incomplete factorization. When the maximum number of levels are used, as in Table 8.1, the smallest system is typically 1×1 or 2×2, and this is an irrelevant remark. However, in the case of Table 8.2, the fact that the smallest system is not solved exactly significantly influences the overall rate of convergence. This is why, unlike methods where the coarsest system is solved exactly, increasing the number of levels tends to improve the rate of convergence. In the case $\epsilon = 10^{-1}$, the coarsest matrix had an exact LDU factorization for the case $maxlvl = 5$ (because the matrix itself was nearly diagonal), and setting $maxlvl > 5$ did not increase the number of levels. The cases $\epsilon = 10^{-2}$

TABLE 8.2
Dependence of convergence of ϵ and maxlvl, discrete Laplacian A, $N = 160000$.

ϵ	*maxlvl*	Digits	Cycles	Init.	Solve	$\sum \lvert ja \rvert$	$\sum \lvert ju \rvert$
	1	6.0	401	4.3	182.7	479	643
	2	6.0	166	9.3	156.1	878	962
	3	6.1	96	13.2	116.9	1077	1119
10^{-1}	4	6.1	79	15.0	107.3	1176	1178
	5	6.0	75	15.8	106.6	1225	1188
	6	–	–	–	–	–	–
	7	–	–	–	–	–	–
	1	6.0	119	5.8	62.4	479	1236
	2	6.1	56	12.1	64.9	878	2106
	3	6.0	32	14.6	49.3	977	2323
10^{-2}	4	6.4	18	15.1	29.2	1002	2376
	5	6.5	9	15.3	15.5	1008	2388
	6	7.2	7	15.2	12.6	1010	2390
	7	6.1	6	15.3	10.9	1011	2391
	1	6.0	41	8.0	24.9	479	1999
	2	6.1	22	16.6	31.7	878	3649
	3	6.6	13	19.4	25.4	977	4053
10^{-3}	4	6.5	7	20.2	15.2	1002	4147
	5	6.0	4	20.3	9.7	1008	4167
	6	6.5	4	20.3	9.5	1010	4170
	7	6.5	4	20.4	9.4	1011	4171
≈ 0	1	11.1	1	52.4	1.8	479	5626

and $\epsilon = 10^{-3}$ used a maximum of 10 and 9 levels, respectively, but the results did not change significantly from the case $maxlvl = 7$.

We also include in Table 8.2 the case $\epsilon = 0$, $maxlvl = 1$, sparse Gaussian elimination. (In fact, our code uses $\mu\|A\|$ as the drop tolerance when the user specifies $\epsilon = 0$ to avoid dividing by zero.) Here we see that Gaussian elimination is reasonably competitive on this problem. However, we generally expect the initialization cost for $\epsilon = 0$ to grow like $O(N^{3/2})$. For $maxlvl = 1$ and $\epsilon > 0$, we expect the solution times to grow like $O(N^p)$, $p > 1$. For the best multilevel choices, we expect both initialization and solution times to behave like $O(N) - O(N \log N)$.

In our final series of tests, we study the convergence of the method for a suite of test problems generated from the finite element code *PLTMG* [8]. These example problems were presented in our earlier work [11], where a more complete description of the problems, as well as numerical results for our hierarchical basis multigraph method and the classical AMG algorithm of Ruge and Stüben [46], can be found. As a group, the problems feature highly nonuniform, adaptively generated meshes, relatively complicated geometry, and a variety of differential operators. For each test case, both the sparse matrix and the right-hand side were saved in a file to serve as input for the iterative solvers. A short description of each test problem is given below.

Problem Superior. This problem is a simple Poisson equation

$$-\Delta u = 1$$

with homogeneous Dirichlet boundary conditions on a domain in the shape of Lake Superior. This is the classical problem on a fairly complicated domain. The solution

is generally very smooth but has some boundary singularities.

Problem Hole. This problem features discontinuous, anisotropic coefficients. The overall domain is the region between two concentric circles, but this domain is divided into three subregions. On the inner region, the problem is

$$-\delta\Delta u = 0$$

with $\delta = 10^{-2}$. In the middle region, the equation is

$$-\Delta u = 1,$$

and in the outer region the equation is

$$-u_{xx} - \delta u_{yy} = 1.$$

Homogeneous Dirichlet boundary conditions are imposed on the inner (hole) boundary, homogeneous Neumann conditions on the outer boundary, and the natural continuity conditions on the internal interfaces. While the solution is also relatively smooth, singularities exist at the internal interfaces.

Problem Texas. This is an indefinite Helmholtz equation

$$-\Delta u - 2u = 1$$

posed in a region shaped like the state of Texas. Homogeneous Dirichlet boundary conditions are imposed. The length scales of this domain are roughly 16×16, so this problem is fairly indefinite.

Problem UCSD. This is a simple constant coefficient convection-diffusion equation

$$-\nabla \cdot (\nabla u + \beta u) = 1,$$

$\beta = (0, 10^5)^T$ posed on a domain in the shape of the UCSD logo. Homogeneous Dirichlet boundary conditions are imposed. Boundary layers are formed at the bottom of the region and the top of various obstacles.

Problems Jcn 0 and Jcn 180. The next two problems are solutions of the current continuity equation taken from semiconductor device modeling. This equation is a convection-diffusion equation of the form

$$-\nabla \cdot (\nabla u + \beta u) = 0,$$

$\beta = 0$ in most of the rectangular domain. However, in a curved band in the interior of the domain, $|\beta| \approx 10^4$ and is directed radially. Dirichlet boundary conditions $u = 10^{-5}$ and $u = 10^{10}$ are imposed along the bottom boundary and along a short segment on the upper left boundary, respectively. Homogeneous Neumann boundary conditions are specified elsewhere. The solutions vary exponentially across the domain which is typical of semiconductor problems.

In the first problem, Jcn 0, the convective term is chosen so the device is *forward biased*. In this case, a sharp internal layer develops along the top interface boundary. In the second problem, Jcn 180, the sign of the convective term is reversed, resulting in two sharp internal layers along both interface boundaries.

We summarize the results in Table 8.3. As before, perhaps the most important point is that the method solved all of the problems. While convergence rates are not independent of h, once again the growth appears to be at worst logarithmic.

Below we make some additional remarks.

TABLE 8.3
Performance comparison.

N	Levels	Digits	Cycles	Init.	Solve
Superior, $\epsilon = 10^{-3}$					
5k	7	7.2	3	2.4e-1	1.0e-1
20k	9	7.3	5	1.4e 0	9.4e-1
80k	9	6.1	7	10.5e 0	6.8e 0
Hole, $\epsilon = 10^{-4}$					
5k	7	6.3	3	4.3e-1	1.5e-1
20k	7	8.2	4	2.4e 0	1.3e 0
80k	8	6.1	5	16.1e 0	7.6e 0
Texas, $\epsilon = 10^{-5}$					
5k	7	12.3	2	4.2e-1	1.1e-1
20k	8	8.2	2	3.0e 0	6.9e-1
80k	9	9.8	5	27.4e 0	10.0e 0
UCSD, $\epsilon = 10^{-3}$					
5k	6	11.1	2	2.0e-1	1.4e-1
20k	6	9.7	2	1.2e 0	7.8e-1
80k	7	8.8	2	10.5e 0	4.0e 0
Jcn 0, $\epsilon = 10^{-4}$					
5k	7	6.4	1	4.5e-1	1.7e-1
20k	7	6.5	1	2.3e 0	8.5e-1
80k	8	10.5	2	15.1e 0	6.2e 0
Jcn 180, $\epsilon = 10^{-5}$					
5k	7	12.3	2	4.9e-1	2.8e-1
20k	7	7.6	2	2.6e 0	1.4e 0
80k	8	7.1	3	18.0e 0	9.3e 0

- For all problems, decreasing the drop tolerance will tend to increase the effectiveness of the preconditioner, although it generally will also make the preconditioner more costly to apply. Thus one might optimize the selection of the drop tolerance to minimize the decreasing number of cycles against the increasing cost per cycle. In these experiments, we did not try such systematic optimization, but we did adjust the drop tolerance in a crude way such that more difficult problems performed in a fashion similar to the easy ones.
- Problem Texas is by far the most difficult in this test suite. While we set $maxfil = 35$, the problem with order $80k$ was the only one which came close to achieving this storage limit. Most were well below this limit, and many averaged less than 10 nonzeros per row in L and U factors.
- For the nonsymmetric problems the CSBCG method is used for acceleration. Since the CSBCG requires the solution of a conjugate system with A^t, two matrix multiplies and two preconditioning steps are required for each iteration. As noted in section 3, with our data structures, applying a transposed matrix and preconditioner costs the same as applying the original matrix or preconditioner. Since these are the dominant costs in the CSBCG methods, the cost per cycle is approximately double that for an equivalent symmetric system.

REFERENCES

[1] O. Axelsson and H. Lu, *On eigenvalue estimates for block incomplete factorization methods*, SIAM J. Matrix Anal. Appl., 16 (1995), pp. 1074–1085.

[2] ———, *Conditioning analysis of block incomplete factorizations and its application to elliptic equations*, Numerische Mathematik, 78 (1997), pp. 189–209.

[3] O. Axelsson and M. Neytcheva, *The algebraic multilevel iteration methods - theory and applications*, in Proceedings of the Second International Colloquium in Numerical Analysis, Plovdiv, Bulgaria, 1993, pp. 13–23.

[4] O. Axelsson and B. Polman, *Stabilization of algebraic multilevel iteration methods; additive methods*, University of Nijmegen, Nijmegen, The Netherlands, 1996.

[5] O. Axelsson and P. S. Vassilevski, *Algebraic multilevel preconditioning methods I*, Numer. Math., 56 (1989), pp. 157–177.

[6] I. Babuška, *private communication*, 2000.

[7] Z.-Z. Bai, *A class of hybrid algebraic multilevel preconditioning methods*, Appl. Numer. Math., 19 (1996), pp. 389–399.

[8] R. E. Bank, *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations, Users' Guide 8.0*, Software, Environments and Tools, Vol. 5, SIAM, Philadelphia, 1998.

[9] R. E. Bank and T. F. Chan, *An analysis of the composite step biconjugate gradient method*, Numerische Mathematik, 66 (1993), pp. 295–319.

[10] R. E. Bank and R. K. Smith, *General sparse elimination requires no permanent integer storage*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. 574–584.

[11] R. E. Bank and R. K. Smith, *The incomplete factorization multigraph algorithm*, SIAM J. on Scientific Computing, 20 (1999), pp. 1349–1364.

[12] R. E. Bank and C. Wagner, *Multilevel ILU decomopsition*, Numerische Mathematik, 82 (1999), pp. 543–576.

[13] R. E. Bank and J. Xu, *The hierarchical basis multigrid method and incomplete LU decomposition*, in Seventh International Symposium on Domain Decomposition Methods for Partial Differential Equations (D. Keyes and J. Xu, eds.), AMS, Providence, RI, 1994, pp. 163–173.

[14] ———, *An algorithm for coarsening unstructured meshes*, Numer. Math., 73 (1996), pp. 1–36.

[15] M. Benzi, D. B. Szyld, and A. van Duin, *Orderings for incomplete factorization preconditioning of nonsymmetric problems*, SIAM J. Sci. Comput., 20 (1999), pp. 1652–1670.

[16] D. Braess, *Towards algebraic multigrid for elliptic problems of second order*, Computing, 55 (1995), pp. 379–393.

[17] A. Brandt, S. McCormick, and J. Ruge, *Algebraic multigrid (AMG) for automatic multigrid solution with application to geodetic computations*, tech. rep., Institute for Computational Studies, Colorado State University, Fort Collins CO, 1982.

[18] ———, *Algebraic multigrid (AMG) for sparse matrix equations*, in Sparsity and Its Applications (D. J. Evans, ed.), Cambridge University Press, Cambridge, UK, 1984.

[19] T. F. Chan, S. Go, and J. Zou, *Boundary treatments for multilevel methods on unstructured meshes*, SIAM J. Sci Comput., 21 (1999), pp. 46–66.

[20] A. J. Cleary, R. D. Falgout, V. E. Henson, and J. E. Jones, *Coarse-grid selection for parallel algebraic multigrid*, in Solving irregularly structured problems in parallel (Berkeley, CA, 1998), vol. 1457 of Lecture Notes in Comput. Sci., Springer, Berlin, 1998, pp. 104–115.

[21] J. E. Dendy, *Black box multigrid*, J. Comput. Phys., 48 (1982), pp. 366–386.

[22] S. C. Eisenstat, H. C. Elman, and M. H. Schultz, *Variational iterative methods for nonsymmetric systems of linear equations*, SIAM J. Numer. Anal., 20 (1983), pp. 345–357.

[23] S. C. Eisenstat, M. C. Gursky, M. Schultz, and A. Sherman, *Algorithms and data structures for sparse symmetric Gaussian elimination*, SIAM J. Sci. Statist. Comput., 2 (1982), pp. 225–237.

[24] H. C. Elman, *A stability analysis of incomplete LU factorizations*, Math. Comp., 47 (1986), pp. 191–217.

[25] H. C. Elman and X. Zhang, *Algebraic analysis of the hierarchical basis preconditioner*, SIAM J. Matrix Anal., 16 (1995), pp. 192–206.

[26] A. George and J. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[27] C.-H. Guo, *Incomplete block factorization preconditioning for linear systems arising in the numerical solution of the helmholtz equation*, Applied Numererical Mathematics, 19 (1996), pp. 495–508.

[28] ———, *Incomplete block factorization preconditioning for indefinite elliptic problems*, Numerische Mathematik, 83 (1999), pp. 621–639.

[29] R. Guo and R. D. Skeel, *An algebraic hierarchical basis preconditioner*, Appl. Numer. Math.,

9 (1992), pp. 21–32.
[30] W. Hackbusch, *Multigrid Methods and Applications*, Springer-Verlag, Berlin, 1985.
[31] W. Hackbusch and G. Wittum, *Incomplete Decompositions – Theory, Algorithms and Applications*, vol. 41 of Notes Numer. Fluid Mech., Vieweg, Braunschweig, 1993.
[32] G. Karypis and V. Kumar, *Analysis of multilevel graph partitioning*, Tech. Rep. 95-037, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1995.
[33] ———, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Statist. Comput., (to appear).
[34] S. LeBorne, *Ordering techniques for convection dominated problems on unstructured three dimensional grids*, in 11th International Symposium of Domain Decomposition Methods for Partial Differential Equations, C. Lai, P. Bjorstad, M. Cross, and O.Widlund, eds., 1999, p. to appear.
[35] C.-J. Lin and J. J. Moré, *Incomplete Cholesky factorizations with limited memory*, SIAM J. Sci. Comput., 21 (1999), pp. 24–45.
[36] M.-M. Magolu, *Ordering strategies for modified block incomplete factorizations*, SIAM J. Sci. Comput., 16 (1995), pp. 378–399.
[37] ———, *Taking advantage of the potentialities of dynamically modified block incomplete factorizations*, SIAM J. Sci. Comput., 19 (1998), pp. 1083–1108.
[38] J. Mandel, M. Brezina, and P. Vanek, *Energy optimization of algebraic multigrid bases*, Computing, 62 (1999), pp. 205–228.
[39] T. Mannseth, *An analysis of the robustness of some incomplete factorizations*, SIAM J. Sci. Comput., 16 (1995), pp. 1428–1450.
[40] A. Messaoudi, *On the stability of the incomplete LU-factorizations and characterizations of H-matrices*, Numerische Mathematik, 69 (1995), pp. 321–331.
[41] Y. Notay, *Using approximate inverses in algebraic multilevel methods*, Numerische Mathematik, 80 (1998), pp. 397–417.
[42] ———, *A multilevel block incomplete factorization preconditioning*, Applied Numerical Mathematics, 31 (1999), pp. 209–225.
[43] C. C. Paige and M. A. Saunders, *Solution of sparse indefinite systems of linear equations*, SIAM J. Numer. Anal., 12 (1975), pp. 617–629.
[44] A. Reusken, *A multigrid method based on incomplete Gaussian elimination*, J. Numer. Linear Algebra Appl., 3 (1996), pp. 369–390.
[45] D. J. Rose, *A graph theoretic study of the numeric solution of sparse positive definite systems*, in Graph Theory and Computing, Academic Press, New York, 1972, pp. 183–217.
[46] J. W. Ruge and K. Stüben, *Algebraic multigrid (AMG)*, in Multigrid Methods, S. F. McCormick, ed., vol. 3 of Frontiers Applied Math., SIAM, Philadelphia, PA, 1987, pp. 73–130.
[47] Y. Saad, *ILUT: a dual threshold incomplete LU factorization*, Numer. Linear Algebra Appl., 1 (1994), pp. 387–402.
[48] P. Vanek, M. Brezina, and J. Mandel, *Convergence of algebraic multigrid based on smoothed aggregation*, Tech. Rep. 126, Center for Computational Mathematics, University of Colorado, at Denver, 1994.
[49] C. Wagner, *Introduction to algebraic multigrid*, tech. rep., Interdisziplinäres Zentrum für Wissenschaftliches Rechnen,Universität Heidelberg, 1999.
[50] W. L. Wan, T. F. Chan, and B. Smith, *An energy-minimizing interpolation for robust multigrid methods*, SIAM J. Sci Comput., 21 (1999), pp. 1632–1649.
[51] G. Wittum, *On the robustness of ILU smoothing*, SIAM J. Sci. Comput., 10 (1989), pp. 699–717.