Parallel implementation and performance characteristics of least squares sparse approximate inverse preconditioners*

Edmond Chow[†]

May 25, 2000

Abstract

This paper describes and tests a parallel, message passing code for constructing sparse approximate inverse preconditioners using Frobenius norm minimization. The sparsity patterns of the preconditioners are chosen as patterns of powers of sparsified matrices. Sparsification is necessary when powers of a matrix have a large number of nonzeros, making the approximate inverse computation expensive. For our test problems, the minimum solution time is achieved with approximate inverses with fewer than twice the number of nonzeros of the original matrix. Additional accuracy is not compensated by the increased cost per iteration. The results lead to further understanding of how to use these methods and how well these methods work in practice. In addition, this paper describes programming techniques required for high performance, including one-sided communication, local coordinate numbering, and load repartitioning.

1 Introduction

A *sparse approximate inverse* approximates the inverse of a matrix A by a sparse matrix M. This approximation is based on the assumption that for many matrices arising from the discretization of partial differential equations (PDEs), the inverses contain many small entries that may be neglected. The primary motivation for the development of sparse approximate inverses has been parallel preconditioning, i.e., the parallel, iterative solution of the linear system $Ax = b$ by a transformed system, for example,

$$MAx = Mb$$

where $M \approx A^{-1}$. Each step of the iterative method involves a parallel sparse matrix-vector multiplication involving M. For symmetric positive definite (SPD) problems, A^{-1} is approximated by the factorization $G^T G$, where G is a sparse lower triangular matrix approximating the inverse of the lower triangular Cholesky factor, L, of A.

*Tech. Report UCRL-JC-138883, Lawrence Livermore National Laboratory, Livermore, CA, 2000 (submitted). This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

[†]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, L-560, Box 808, Livermore, CA 94551 (echow@llnl.gov).

In this paper, we consider parallel methods that construct M or G by minimizing the Frobenius norm of the residual matrix $(I - MA)$ or $(I - GL)$, respectively. In the unfactorized case (see 2.3.1 for the factorized case), the objective function can be minimized in parallel because it can be decoupled as the sum of the squares of the 2-norms of the individual rows

$$\|I - MA\|_F^2 = \sum_{i=1}^{n} \|e_i^T - m_i^T A\|_2^2 \tag{1}$$

in which e_i^T and m_i^T are the ith rows of the identity matrix and of the matrix M, respectively. Thus, minimizing (1) is equivalent to minimizing the individual functions

$$\|e_i^T - m_i^T A\|_2, \quad i = 1, 2, \ldots, n. \tag{2}$$

If no restriction is placed on M, the exact inverse will be found. To find an economical approximation, each row in M is constrained to be sparse, by either specifying the nonzero entries in M *a priori* before the minimizations, or during the minimizations with an adaptive procedure. When the m_i are sparse, each function in (2) can be minimized using a least squares method. If A is nonsingular, then the least squares matrices have full rank.

Adaptive procedures are based on updating a sparsity pattern and solving the resulting least squares problem exactly or inexactly. This process is repeated until a threshold on the residual norm has been satisfied, or a maximum number of nonzeros has been reached (Chow & Saad, 1998; Cosgrove, Díaz, & Griewank, 1992; Grote & Huckle, 1997). These techniques have been implemented in parallel by Deshpande, Grote, Messmer, and Sawyer (1996), Barnard and Clay (1997), Barnard, Bernardo, and Simon (1999) and Huckle (1996b).

Adaptive techniques can produce accurate approximate inverses, but these techniques can be expensive, and the use of *a priori* patterns is often just as effective (Chow, 2000). Banded patterns for banded matrices (Benson, Krettmann, & Wright, 1984; Grote & Simon, 1993) and the pattern of A, A^2, and other variants for more general matrices may be used (Field, 1997, 1998; Huckle, 1997). For dense matrices, sparsification to produce patterns for a sparse approximate inverse has been successful (Kolotilina, 1988; Vavasis, 1992; Chen, 1998; Carpentieri, Duff, & Giraud, 2000).

Patterns of powers of sparsified matrices

In this paper, we use *a priori* patterns that are patterns of powers of sparsified matrices (Cosnuau, 1996; Alléon, Benzi, & Giraud, 1997; Tang & Wan, 1998; Chow, 2000; Field, 1999), i.e., patterns of $\tilde{A}^L$, where $\tilde{A}$ is a matrix constructed from A, for example, by numerically dropping entries in A that are small in magnitude. L is a small positive integer, and $L - 1$ is known as the *level* of the pattern. Dropping small entries has the effect of identifying the most important interactions described by a matrix, and the use of matrix powers incorporates "nearby" interactions.

Consider the discrete Green's function in Figure 1(a) for an anisotropic PDE at a point near the center of a square domain. The discrete Green's function plots a row of the inverse of the discretization matrix—thus we seek a pattern that approximates the pattern of the largest entries in the Green's function. Figure 1(b) plots the approximation to the Green's function when the pattern of the original matrix is used as the pattern of the approximate inverse. Note that some small entries in the inverse are included in the pattern before other larger entries. Figure 1(c) plots the approximation to the Green's function when the pattern of a sparsified discretization matrix is used. Sparsification detects the anisotropy. Figure 1(d) plots the approximation using the pattern of the square of the sparsified matrix (level 1 pattern). Now all large entries in the inverse are identified. Note, however, that the way the matrix is sparsified must be balanced with the matrix power used.

In the factorized case, where $G^TG \approx A^{-1}$, the pattern for G should be chosen such that the pattern of G^TG is close in some sense to good patterns for approximating A^{-1}. Given a good pattern for approximating A^{-1}, then the lower triangular part of this pattern is a good pattern for G since the pattern of G^TG includes the given pattern.

2 Algorithmic Stages

This section describes the algorithmic stages for computing sparse approximate inverses using least squares minimization and patterns of powers of sparsified matrices. We also include a post-processing stage called *filtration* (Kolotilina, Nikishin, & Yeremin, 1999). These stages are outlined in Algorithms 2.1 and 2.2 for the nonfactorized and SPD factorized cases, respectively. The input parameters are the threshold *thresh* used in stage 1, the parameter L (equal to the *level* of the pattern plus one) in stage 2, and the filter value *filter* used in stage 4.

ALGORITHM **2.1** *Nonfactorized sparse approximate inverse,* $M \approx A^{-1}$

1. *Threshold* A *to produce* $\tilde{A}$
2. *Compute the pattern* $\tilde{A}^L$ *for* M
3. *Compute the nonzero entries in* M*, by minimizing* $\|I - MA\|_F^2$
4. *Filtration: drop small entries in* M

ALGORITHM **2.2** *Factorized sparse approximate inverse,* $G^TG \approx A^{-1}$

1. *Threshold* A *to produce* $\tilde{A}$
2. *Compute the pattern* $\tilde{A}^L$ *and let the pattern of* G *be the lower triangular part of the pattern of* $\tilde{A}^L$
3. *Compute the nonzero entries in* G*, as in section 2.3.1*
4. *Filtration: drop small entries in* G *and rescale*

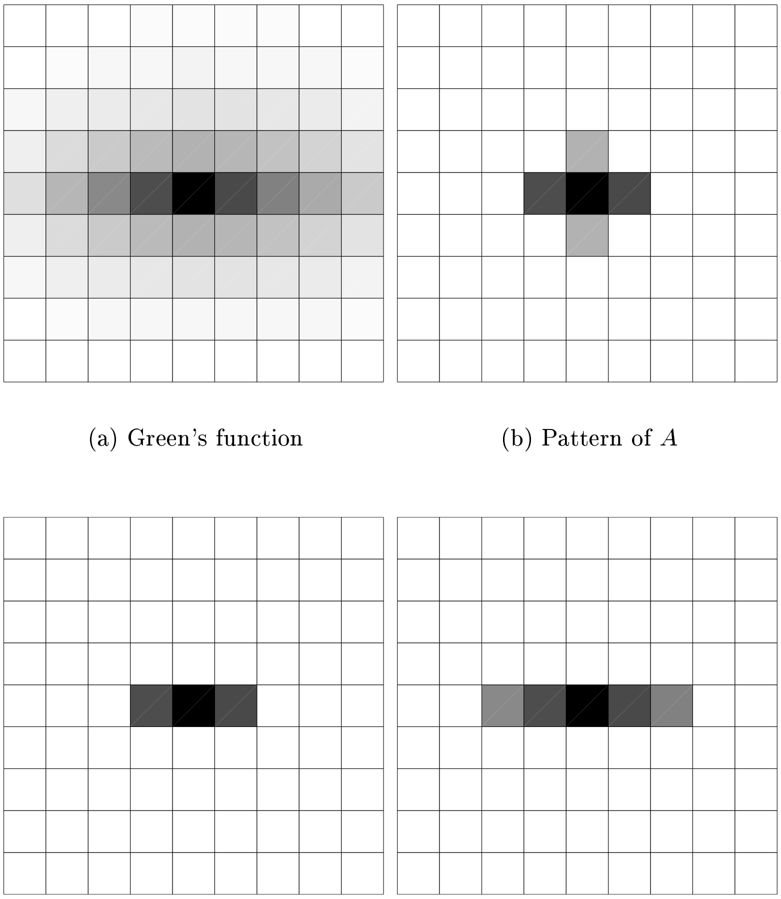
2.1 Thresholding A

A binary matrix $\tilde{A}$ is constructed from a matrix A such that

$$\tilde{A}_{ij} = \begin{cases} 1 & \text{if } i = j \text{ or } |(D^{-1/2}AD^{-1/2})_{ij}| > \mathit{thresh} \\ 0 & \text{otherwise} \end{cases}$$

where the diagonal matrix D is

$$D_{ii} = \begin{cases} |A_{ii}| & \text{if } |A_{ii}| > 0 \\ 1 & \text{otherwise} \end{cases}$$



(a) Green's function

(b) Pattern of A

(c) Pattern of sparsified A

(d) Level 1 pattern

Figure 1: Patterns illustrated using discrete Green's functions.

and *thresh* is a nonnegative threshold. The scaling with D is necessary if A is poorly scaled. The diagonal part of the sparsity pattern should be retained. On a parallel computer, if A is partitioned by rows, interprocessor communication is needed for the nonlocal components of D.

Denser approximate inverses M tend to be more accurate, but the cost of the preconditioning operation is higher, proportional to the number of nonzeros in M. Denser approximate inverses are also more expensive to construct. Thus the threshold *thresh* and the level parameter should be chosen to balance accuracy and cost.

2.2 Computing the sparsity pattern

The kernel operation in computing the pattern of $\tilde{A}^L$ is the merging of the patterns of sparse rows. This operation can be implemented with searches, or better implemented by storing one of the sparse rows in dense format so that it is quick to check whether an entry is nonzero. When rows are partitioned over several processors, the indices must be merged using *local coordinates* in order for the implementation to be scalable in memory. This is discussed in section 3.3. In addition, a processor will occasionally need to merge a row stored on another processor. The second processor will be unaware that it needs to communicate this row to the first processor. This type of communication is called *one-sided* or *single-sided* communication and is discussed in section 3.2. Finally, computing the pattern of $\tilde{A}^L$ may be organized in parallel in many ways. This is discussed in section 3.1.

2.3 Computing the nonzero entries

2.3.1 Factorized case

For symmetric positive definite (SPD) problems, minimizing $\|I - GL\|_F^2$ can be accomplished without knowing the Cholesky factor L by solving the normal equations

$$\{GLL^T\}_{ij} = \{L^T\}_{ij}, \quad (i,j) \in \mathcal{S}_L \tag{3}$$

where $\mathcal{S}_L$ is a lower-triangular nonzero pattern for G. Equation (3) can be replaced by

$$\{\tilde{G}A\}_{ij} = I_{ij}, \quad (i,j) \in \mathcal{S}_L \tag{4}$$

where $\tilde{G} = D^{-1}G$ and D is the diagonal of L. Note that each row of $\tilde{G}$ can be computed independently by solving a small SPD linear system.

The preconditioned matrix has the form

$$GAG^T = D\tilde{G}A\tilde{G}^T D \tag{5}$$

but since D is not generally known, it is chosen such that the diagonal of GAG^T is all ones.

Let $\mathrm{diag}(X)$ denote the diagonal matrix that is the diagonal of the matrix X. Then a few simple propositions can be stated.

Proposition 2.1 (Kolotilina & Yeremin, 1993) *If* $\mathrm{diag}(D\tilde{G}A\tilde{G}^T D) = I$, *then* $D = \mathrm{diag}(\tilde{G})^{-1/2}$.

Proposition 2.2 *The preconditioned matrix is invariant to diagonal scalings of* A.

PROOF. Let $\tilde{G}$ denote the solution of (4) and let $\tilde{H}$ denote the solution of

$$\{\tilde{H}DAD\}_{ij} = I_{ij}, \quad (i,j) \in \mathcal{S}_L \tag{6}$$

where D is a diagonal matrix. Without loss of generality, consider the computation of $\tilde{h}^T$, the ith row of $\tilde{H}$, and let $\mathcal{J} = \{j \mid (i,j) \in \mathcal{S}_L\}$. (We have dropped reference to i for notational convenience.) Denote by a subscript $\mathcal{J}$ the restriction onto the indices $\mathcal{J}$. Then for $(i,j) \in \mathcal{S}_L$, the nonzero entries of $\tilde{h}^T$ can be computed by solving

$$\begin{aligned} \tilde{h}_{\mathcal{J}}^T D_{\mathcal{J}\mathcal{J}} A_{\mathcal{J}\mathcal{J}} D_{\mathcal{J}\mathcal{J}} &= (e_i^T)_{\mathcal{J}} \\ D_{ii}\tilde{h}_{\mathcal{J}}^T D_{\mathcal{J}\mathcal{J}} A_{\mathcal{J}\mathcal{J}} &= (e_i^T)_{\mathcal{J}} \\ \tilde{g}_{\mathcal{J}}^T A_{\mathcal{J}\mathcal{J}} &= (e_i^T)_{\mathcal{J}} \end{aligned}$$

where $\tilde{g}^T = D_{ii}\tilde{h}_{\mathcal{J}}^T D_{\mathcal{J}\mathcal{J}}$. Thus in general, $\tilde{G} = D\tilde{H}D$.

Now, from proposition 2.1, define the scalings

$$\begin{aligned} D_1 &= \operatorname{diag}(\tilde{G})^{1/2} \\ D_2 &= \operatorname{diag}(\tilde{H})^{1/2} \\ &= \operatorname{diag}(D^{-1}\tilde{G}D^{-1})^{1/2} \\ &= D_1 D^{-1} \end{aligned}$$

The preconditioned matrix for A is

$$\begin{aligned} GAG^T &= D_1^{-1}\tilde{G}A\tilde{G}^T D_1^{-1} \\ &= D_1^{-1}(D\tilde{H}D)A(D\tilde{H}^T D)D_1^{-1} \\ &= D_2^{-1}\tilde{H}(DAD)\tilde{H}^T D_2^{-1} \\ &= H(DAD)H^T \end{aligned}$$

which is the preconditioned matrix for DAD. □

Proposition 2.3 *If G is the computed approximate inverse factor for A, and H is the factor for the symmetrically scaled matrix DAD, then $G = HD$.*

PROOF. This follows immediately from the proof of proposition 2.2. □

Proposition 2.4 *The diagonal entries of G are positive.*

PROOF. For each i, row i of $\tilde{G}$ is the ith row of the inverse of $A_{\mathcal{J}\mathcal{J}}$. Since $A_{\mathcal{J}\mathcal{J}}^{-1}$ is SPD, the ith entry of this row is positive. Finally, G is just a positive scaling of $\tilde{G}$. □

The ith row of G can be computed independently of the other rows by solving a small linear system with the sparse matrix

$$A_{\mathcal{J}\mathcal{J}}, \quad \mathcal{J} = \{j \mid (i,j) \in \mathcal{S}_L\}.$$

The system can be solved by using a dense direct method, a sparse direct method (Field, 1997), or approximately by using an iterative method (Kolotilina, Nikishin, & Yeremin, 1992).

2.3.2 Nonfactorized case

In the nonfactorized case, the nonzero entries in the approximate inverse M may be computed by minimizing (2) using dense QR factorization. See Huckle (1996a) for methods that treat the least squares matrix as sparse.

Computing a nonfactorized approximate inverse requires about 32 times more flops than computing an SPD factorized approximate inverse. If m is the number of nonzeros in a row of the approximate inverse or approximate inverse factor, then computing the row requires at least $4m^3/3$ flops for the nonfactorized form using QR factorization, or $m^3/3$ flops for the factorized form using Cholesky factorization. In addition, for a given sparsity pattern, the number of nonzeros per row is nominally halved in the factorized case, giving the factor of 32. Thus computing a nonfactorized approximate inverse is much more expensive than computing a factorized approximate inverse, and much sparser approximate inverses are required in the former case to make them economical to compute. Therefore, it is advantageous to compute nonsymmetric and/or indefinite preconditioners in factorized form as well.

2.4 Filtration

The sparse approximate inverse M (or the factor G) may contain many nonzero entries that are small in magnitude. These entries may be dropped to reduce the cost of storing and multiplying by M (Tang & Wan, 1998; Kolotilina et al., 1999). In the nonsymmetric case, the perturbation to the preconditioning operation when nonzeros are dropped is proportional to the size of these nonzeros. Experimentally, the number of nonzeros in M may be significantly reduced with little degradation in convergence rate of the iterative method. Filtration is analogous to the dropping that is performed in some adaptive approximate inverse techniques.

Entries in M may be dropped if they are smaller in magnitude than a threshold called the *filter value*. To make this threshold independent of the scaling of A, entries in M are dropped if the corresponding entry in $D^{-1}MD^{-1}$ is small, where $D = \mathrm{diag}(A)^{-1/2}$. In the factorized case, we use proposition 2.3 and drop a nonzero in G if the corresponding nonzero in GD^{-1} is small in magnitude compared to the filter value. Thus M and G do not actually have to be scaled to perform the filtration. In both the nonfactorized and factorized cases, the diagonal entries are never dropped.

In the factorized case, Kolotilina et al. (1999) recommends rescaling G after small entries are dropped to $\hat{G} = D_G G$ such that the diagonal of the preconditioned matrix $\hat{G}A\hat{G}^T$ is all ones. Unlike the scaling in (5), this involves constructing the diagonal of GAG^T. If G is stored by rows, and the local processor stores the rows of A that were needed to compute the local part of G, then the diagonal of GAG^T can be computed without interprocessor communication. Rescaling may not be important, however, if the filter value is small.

3 Parallel implementation

3.1 Parallel computation of sparse matrix powers

When the matrix $\tilde{A}$ is stored and partitioned by rows on a parallel computer, powers of $\tilde{A}$ can be computed by combining the rows of $\tilde{A}$. Denote the kth row of $\tilde{A}^L$ by $\tilde{A}^L_{k,:}$. Then,

$$\tilde{A}^L_{k,:} = \tilde{A}^{L-1}_{k,:}\tilde{A} \tag{7}$$

which is the combination of the rows of $\tilde{A}$ corresponding to the nonzeros in $\tilde{A}^{L-1}_{k,:}$. Interprocessor communication is required if the required rows of $\tilde{A}$ are not stored locally.

An alternative to (7) is to use

$$\tilde{A}^L_{k,:} = \tilde{A}_{k,:}\tilde{A}^{L-1}$$

which is the combination of the rows of $\tilde{A}^{L-1}$ corresponding to the nonzeros in $\tilde{A}_{k,:}$. Here, which rows of each $\tilde{A}^{L-1}$ that need to be sent to other processors remains fixed (and are known by the sending processor when $\tilde{A}$ has symmetric structure). The volume of the communication can be much larger, however, and all rows of $\tilde{A}^{L-1}$ must be available.

The required interprocessor communication can be visualized if $\tilde{A}$ is interpreted as a graph, with its nodes partitioned among processors. Computing the pattern of sparse matrix powers is equivalent to computing level sets about each node in the graph. To compute $\tilde{A}^L$, information is required from the $L-1$st level neighbors. Many other scientific computing algorithms only require information from the first level neighbors.

Parallel organization

There are several alternatives to how the parallel computation of the pattern of sparse matrix powers may be organized. First, each row of $\tilde{A}^L$ may be computed independently of each other. This is efficient on data movement because all the merging required to form a row can be performed at once. Processors must communicate *asynchronously*, however, when rows from another processor are needed in the computation. In addition, this communication is *one-sided.* This approach also precludes taking advantage of any symmetry in the pattern of $\tilde{A}$.

To avoid asynchronous communication, the pattern of each power of $\tilde{A}$ may be computed in sequence. All processors can exchange messages before the new power of $\tilde{A}$ is computed. This may create periods of greater network contention, but there are also fewer and longer messages. In the symmetric case, if a processor makes requests to r other processors, this processor will also receive requests from these r processors. The number of rows exchanged between each pair of processors, however, is generally not the same. In the nonsymmetric case, the number of requests that a processor will receive cannot be computed by the processor. A global communication is needed for each processor to determine how many requests it will receive. This parallel organization, requires storage of the previous power of $\tilde{A}$ and has much more data movement.

Asynchronous communication and the data movement problem can be avoided by separating the communication from the computation of the rows of $\tilde{A}^L$, as described in Algorithm 3.1.

ALGORITHM **3.1** *Computation of the pattern of $\tilde{A}^L$*

1. For $l = 1, \ldots, L$
2. Request nonlocal rows of $\tilde{A}$ corresponding to nonzeros in $\tilde{A}^l_{i,:}$ that have not already been requested, where i is a local row
3. Service requests for rows
4. Endloop
5. For all local row numbers i
6. For $l = 1, \ldots, L$
7. Compute the pattern of $\tilde{A}^{l+1}_{i,:}$ by merging the pattern of the rows of $\tilde{A}$ corresponding to nonzeros in $\tilde{A}^l_{i,:}$
8. Endloop
9. Endloop

Algorithm 3.1 combines the advantages of the two previous alternatives. However, it is no longer the case (even for symmetric matrices) that the number of requests that will be received can be known without performing a global communication. On the other hand, these global communications are not significant compared to the global communications required in the iterative solution method.

To compute the numerical values in M or G, rows from A are needed. If M has symmetric structure, the processors that need to communicate can be determined beforehand. In general, however, M is not symmetric and G is triangular. Thus we also use global communication here to determine which processors need to communicate.

3.2 One-sided communication

A processor computing its portion of $\tilde{A}^L$ typically requires rows stored on other processors. These other processors generally cannot predict that the first processor needs to communicate. This *one-sided* communication can be accomplished with specialized software (Nieplocha, Littlefield, & Rosing, 1995; Gropp, Huss-Lederman, Nitzberg, & Lusk, 1998), or can be simulated with message passing.

In a message passing implementation, processors send requests for rows that they need. Processors must also occasionally probe for and service such requests. This is the technique used by Barnard et al. (1999). The polling frequency must be chosen carefully to balance the resultant latency and wasted CPU cycles. In a multithreaded environment, one or more threads may be dedicated to servicing these requests (Chow, 2000).

If the asynchronous nature of one-sided communication can be avoided, as in Algorithm 3.1, then it is well-determined when polling is necessary. The communication kernel for synchronous one-sided communication is outlined in Algorithm 3.2.

ALGORITHM **3.2** *Synchronous one-sided communication*

1. Determine num_requests, the number of requests this processor will receive, using global communication
2. Loop $i = 1, \ldots,$ num_requests
3. Send request i (nonblocking, no need to test for completion)
4. Endloop
5. Determine num_replies, the number of replies this processor will receive, using global communication
6. Loop $i = 1, \ldots,$ num_replies
7. Receive a request (use probe, since don't know request size)
8. Construct reply in its own buffer and send reply (nonblocking, or else may deadlock)
9. Endloop
10. Loop $i = 1, \ldots,$ num_requests
11. Receive a reply directly into buffers (use probe, since don't know reply size)
12. Endloop
13. Wait for all sends in line 8 to complete

3.3 Global and local coordinate numbering

A local numbering scheme is used for the matrix row and column indices on a processor. The local numbering scheme numbers the local indices sequentially starting at 0, followed by the external indices. The size of the matrix that each processor sees is much smaller than the global matrix size. This allows sparse matrix operations that require *full-length* arrays (arrays of length the global size) to be implemented scalably.

Adding two sparse vectors is an example of such an operation. One vector is scattered into a full-length array so that its nonzeros can be accessed quickly. The second vector is accumulated into this full-length array, keeping track of new nonzeros, and the result is gathered into a sparse sum. If global coordinate numbering is used, then the full-length array must have length the order of the global matrix and memory usage becomes unscalable with the number of processors. If local coordinate numbering is used, then the full-length array is typically not much larger than the number of rows stored on the processor.

Another operation that requires local coordinate numbering is the construction of $A_{\mathcal{J}\mathcal{J}}$ and the least squares matrices in (2). This operation is costly and must be implemented efficiently. A mapping of the indices to the numbering of $A_{\mathcal{J}\mathcal{J}}$ is required and again is accomplished through a full-length array.

Matrices and other data structures should use local coordinate numbering because it is faster to convert local coordinates to global, rather than the reverse. Indices only need to be converted to global coordinates when they are passed to other processors, or when they are output to the user.

The alternative to full-length arrays is to use searching, i.e., search for a global coordinate index to determine if it is nonzero or to look up associated data. This can be very slow if many indices are involved. Using both global and local numbering schemes is complex but in many cases should be designed into a parallel code in order to achieve scalable implementation and high performance.

Implementation with hash tables

Converting an index from local coordinates to global coordinates can be accomplished efficiently through an array. Efficient conversion from global to local coordinates can be accomplished using hash tables.

A hash table can be used to store a set of objects when the number of keys is small compared to the total number of possible keys. Hashing maps the key, in this case the global coordinate index, to a location inside the table which stores data associated with the index. See, for example, Gonnet and Baeza-Yates (1991), Cormen, Leiserson, and Rivest (1991) for more details. We found that the modulus hash function can result in many hash table collisions on some regularly structured problems, and currently use *multiplicative* hash function (Knuth, 1973) instead. Our hash table stores the keys in the table, with collisions resolved by linear probing, which has better cache behavior than other schemes.

There are two potential difficulties with using hash tables for mapping global indices: an upper limit on the number of unique global indices on a processor must be known beforehand, and the performance is difficult to predict if collisions are frequent. The first problem can be fixed by simply rehashing into a larger table whenever necessary. The second problem can be fixed by choosing a good hash function and rehashing whenever the table is 50 percent full.

3.4 Load Balancing

When the sparse approximate inverse is very sparse, e.g., fewer than 20 nonzeros per row, then its computation is fast and little can be gained in the total solution time by attempting to rectify any load imbalance. When the sparse approximate inverse has many nonzeros per row, however, the computation of the nonzero values in the approximate inverse can be excessive. This stage of the computation costs $O(l^3)$ for a row that contains l nonzeros in the approximate inverse. This high order complexity can cause this stage to be unbalanced, even when each processor owns approximately the same number of rows of the matrix.

The cost of the numerical stage is known beforehand, after the pattern of the approximate inverse has been determined, which is a quick computation. Thus we can then approximately load balance the entire computation by *repartitioning* which rows are computed by each processor.

We define load balance as the ratio of the average workload to the largest workload. Let m denote the average workload, calculated as the average of the sum of the cubes of the lengths of the local rows of $\tilde{A}^L$. To achieve a load balance of at least β, processors with greater than m/β workload must *donate* rows to processors with less than the average workload. The recipient processors accept work up to a total workload of m/β. After the work is performed, the donated rows are returned to their donors.

The repartitioning should be chosen to be an *incremental* modification to the existing partitioning (Hendrickson & Devine, to appear). This reduces both the data transfer to perform the repartitioning and the data communication in the repartitioned problem. Incrementality can be accomplished by transferring work as much as possible between neighboring processors.

proc	unbalanced	balanced
0	51.7	75.9
1	59.8	76.5
2	71.0	78.6
3	119.7	74.5
max	119.7	78.6
ave	75.6	76.4

Table 1: Timings for original and load balanced computations with 4 processors.

Algorithm 3.3 shows how the workload should be repartitioned, given p processors, each with workload c_i. The target load balance β may be set less than 1, to reduce the overhead of repartitioning. This algorithm can be run independently after each processor broadcasts the size of its workload to all other processors. In the algorithm, a processor can donate work to more than one processor and a processor may accept work from more than one other processor. The algorithm does not attempt to minimize the number of processors that must exchange data.

It is also possible to reduce the data transfer by transferring long rows rather than short rows, since the work is proportional to the cube of the length of the rows. The gains, however, may not be large, and we do not utilize this in the algorithm.

ALGORITHM **3.3** *Repartitioning*

Set $m = \frac{1}{p}\sum c_i$ *and set* $upper = m/\beta$
Loop on i *over all processors*
    If ($c_i > upper$) *then must donate work:*
        Set the quantity of work to move: $move = c_i - upper$
        Loop over all other processors, beginning with neighbors
            If ($c_j < m$) *then* j *has below average workload:*
                $accept = min(move, upper - c_j)$
                $c_i = c_i - accept$
                $c_j = c_j + accept$
                If ($c_i \leq upper$) *then exit loop (done balancing processor* i*)*
            Endif
        Endloop
    Endif
Endloop

Table 1 shows an example calculation of a sparse approximate inverse with timings for four processors with no load balancing, and with load balancing set with $\beta = 1$. The total time (maximum time) is reduced in the balanced case. The average time shows that there is not a significant repartitioning overhead for this case.

3.5 Parallel sparse matrix-vector multiplication

In an iterative solver, the sparse matrix-vector multiplication kernel should be as efficient as possible. Any matrix data structure should be designed such that this kernel can be implemented efficiently. See, for example, Saad and Malevsky (1995), Tuminaro, Shadid, and Hutchinson (1998) and Gropp, Kaushik, Keyes, and Smith (1999).

Parallel sparse matrix-vector multiplication should also be implemented with local coordinate numbering, with the external indices from a given processor grouped together in the numbering. Thus external values collected from a processor do not need to be reordered before they are used.

If an approximate inverse factor G is lower triangular and stored row-wise, applying the factorized preconditioner involves multiplications by G and G^T. The indices that are sent by a processor in the G multiplication are the indices that are received by that processor in the G^T multiplication. Thus, setting up the matrix-vector multiplications for a matrix and its transpose can be simplified.

The multiplication by the triangular matrix G typically does not require load balancing because each processor normally has approximately the same number of nonzeros of G. This is clear when G has the pattern of the lower triangular part of a matrix from a PDE discretization.

In any case, whether or not the multiplication by G needs to be load balanced can be determined by comparing its time to the time for multiplying by the coefficient matrix A, scaled by the ratio of the numbers of nonzeros in the two matrices. If G is repartitioned to balance the matrix-vector multiplication, then the result of the multiply needs to be redistributed to match the partitioning of A. It is also possible to find one partitioning for both A and G such that the overall computation is balanced, using a *weighted* partitioning method (Karypis & Kumar, 1998).

3.6 Matrix data structure

Unlike other preconditioners such as incomplete factorizations, each row of the coefficient matrix A is accessed many times in order to construct the preconditioner. Thus A should be stored in a data structure that has efficient access by rows.

To store the preconditioner in contiguous rows so that its access is cache efficient, the storage requirements for the preconditioner should be known beforehand. As an alternative, we store the preconditioner in large, separate blocks of memory. The rows within a block are stored contiguously, and additional blocks are allocated as needed. Some memory is wasted in the final block. All blocks are deallocated when the matrix is no longer needed.

4 Experimental Results

4.1 Detailed study of an elasticity problem

We begin with a detailed study of one problem, a finite element elasticity model of three concentric spherical shells. The two outer shells are hard materials (steel) and the inner shell is a soft material (lucite). "Slide surface" boundary conditions between the shells allow the shells to

	Ratio	Iter	Setup	Solve	Total
diagonal scaling		1992			78.2
pattern of A	1.00	705	3.1	58.4	61.5
pattern of A^2	4.65	470	78.9	121.3	200.2

Table 2: Results using simple *a priori* patterns.

move tangentially relative to one other. The small example linear system that we study from this model has 18205 equations with 1.4 million nonzeros and is SPD.

In all tests described in this paper, we use the factorized preconditioner with preconditioned conjugate gradients (PCG) to reduce the initial residual norm by 8 orders of magnitude. Timings were taken on an IBM SP with PowerPC 604e (332 MHz) processors. For this small problem, 4 processors (1 node) were used.

In the tables, the total solution time is decomposed as the time required for constructing the preconditioner ("Setup") and the time required for the iterative solve phase ("Solve"). All timings are reported in seconds. The sparsity ratio, denoted by "Ratio," is the ratio of the number of nonzeros in the preconditioner to the number of nonzeros in the original matrix and "Iter" is the number of iterations to convergence.

In Table 2, results using simple *a priori* patterns are shown. The lowest total time is achieved by using the pattern of the original matrix A. The more accurate preconditioner using the pattern of A^2 has both higher construction cost and higher iteration cost. The pattern of A^2 averages 350 nonzeros per row, which leads to very high construction costs. The sparsity ratio of 4.65 for the more accurate preconditioner is not compensated by the better convergence rate.

When constructing the preconditioner, computing the pattern takes much less time than computing the values (0.2 and 2.7 seconds, respectively, using the pattern of A, and 1.9 and 75.9 seconds, respectively, using the pattern of A^2).

Table 3 shows that sparsification can produce preconditioners which are cheaper to use and construct. Various sparsification thresholds were tested with the pattern of $\tilde{A}^2$. A threshold of 0 indicates that no dropping was used. Thresholding reduces both the setup and solve timings. It reduces the setup timings because sparser approximate inverses are more economical to construct. It reduces the solve timings because each step of an iterative solver costs less, because the preconditioner is sparser. The number of iterations does not increase substantially when small thresholds are used.

Filtration can further reduce the iteration time. Table 4 shows results using various filter values. When a filter value of 0.1 is used, 78 percent of the nonzeros in the initial preconditioner are dropped, while the number of iterations does not increase substantially. A small increase in the setup time due to filtering can be noted in Table 4. The size of this increase is related to the number of nonzeros that must participate in the rescaling after small entries are dropped. Tests on several problems show that filter values of 0.05 to 0.1 are usually best, corroborating the preliminary result of Kolotilina et al. (1999).

When sparsification and filtration are combined, a smaller fraction of nonzeros is dropped by the filtration procedure. The best filter values are the same, except when very large sparsification

Thresh	Ratio	Iter	Setup	Solve	Total
0.00	4.65	470	78.9	121.3	200.2
0.01	3.52	475	38.8	96.5	135.3
0.05	0.97	504	2.9	42.8	45.8
0.10	0.27	869	0.8	46.6	47.4

Table 3: Results using different sparsification thresholds ($level = 1$).

Filter	Ratio	Iter	Setup	Solve	Total
0.001	0.94	504	4.7	43.1	47.8
0.010	0.72	503	4.1	37.8	41.9
0.050	0.35	548	3.5	31.3	34.8
0.100	0.21	579	3.3	29.0	32.3
0.200	0.10	719	3.2	34.7	37.9

Table 4: Results using different filter values ($thresh = 0.05$, $level = 1$).

thresholds are used, in which case filtration does not significantly reduce the number of nonzeros, although the convergence rate deteriorates.

It is possible to make approximate inverses very accurate and sparse by computing an initial approximate inverse that has high sparsity ratio, and then applying the filtration procedure to produce a very sparse approximate inverse. The initial approximate inverse is expensive to compute, but the very sparse filtered pattern can be very accurate. In a sequence of solves when the matrix does not change significantly, the filtered pattern can be reused to generate very accurate preconditioners at low cost. In Table 5, the sparsity ratio before filtering was 6.89, and the values for this preconditioner were expensive to compute. Filtration produced a preconditioner with sparsity ratio 0.84. This filtered pattern was used to generate a preconditioner for the same matrix very cheaply.

In Table 6, we show results for various combinations of threshold and level. As mentioned in section 2.1, these parameters must be balanced for best results; the best level depends on the threshold, and vice-versa.

Thresh	Level	Filter	Ratio	Iter	Setup	Solve	Total
0.05	3	0.05	0.84	146	240.1	11.5	251.5
pattern from above			0.84	156	3.1	11.8	14.9

Table 5: Accurate sparse approximate inverse patterns.

Thresh	Level	Ratio	Iter	Setup	Solve	Total
0.01	0	0.25	754	2.0	39.3	41.3
0.01	1	0.47	539	40.0	33.7	73.7
0.01	2	0.80	243	491.2	20.4	511.6
0.05	0	0.17	804	0.9	40.7	41.5
0.05	1	0.35	548	3.5	31.3	34.8
0.05	2	0.57	252	33.5	17.9	51.3
0.05	3	0.84	146	240.1	11.5	251.5
0.10	0	0.09	1062	0.6	47.4	48.0
0.10	1	0.18	886	1.1	43.1	44.2
0.10	2	0.28	729	2.3	40.3	42.6
0.10	3	0.38	569	6.7	33.5	40.2
0.10	4	0.50	507	27.3	32.3	59.7
0.10	5	0.62	456	104.6	32.3	136.9

Table 6: Results varying *thresh* and *level* (*filter* = 0.05).

Thresh	Level	Filter	Ratio	Iter	Setup	Solve	Total
0.05/0.2	2	0.05	0.36	316	2.8	18.2	21.0

Table 7: Result using the pattern of $A_1 A_2^3 + A_2^3 A_1$ where A_1 was created with threshold 0.05 and A_2 was created with threshold 0.2. The sparsity ratio before filtering was 0.71.

4.2 Patterns for PDE systems problems

A system of m PDEs is often discretized with m variables at each grid point, leading to matrices containing block entries of size $m \times m$. It is possible to retain this block structure in a sparse approximate inverse to take advantage of efficient computations with blocks. However, the patterns of the largest entries in the inverses of these matrices generally do not have block structure; the inverses often show stronger couplings between like variables instead of variables at the same grid point.

For these problems, we propose using *dual threshold patterns*, or patterns of matrices of the form

$$A_1 A_2^L + A_2^L A_1$$

where A_1 and A_2 are sparsified matrices using two different thresholds. The matrix A_2 should only capture the couplings between like variables. Since this matrix is very sparse, high powers of this matrix are feasible and emulate the strong couplings between like variables in the inverse. The matrix A_1 should be denser and represent more global couplings.

For the problem studied in section 4.1, the best timing can be further reduced by using these dual threshold patterns, as shown in Table 7.

Thresh	Level	Filter	Ratio	Iter	Setup	Solve	Total
0.00	4	0.00	12.35	1422	163.8	613.6	777.3
0.01	4	0.00	12.26	1386	163.3	632.3	795.6
0.00	4	0.01	8.27	1396	174.5	441.2	615.7
0.01	4	0.01	8.23	1413	171.5	454.8	626.3
0.01/0.10	3	0.01	6.36	1590	102.6	386.8	489.3

Table 8: Results for s3dkt3m2. The last row shows that dual threshold patterns can be used to generate sparser preconditioners for PDE systems problems. The sparsity ratio before filtering for the dual threshold pattern was 10.00.

As another example of the use of dual threshold patterns, we consider problem "s3dkt3m2," which is a very difficult problem for sparse approximate inverse preconditioners. The problem, from Reijo Kouhia, models a cylindrical shell (radius/thickness=1000). The matrix is of order 90449 and has 3.8 million nonzeros. This problem could not be solved with sparse approximate inverses with levels up to 3 and with thresholds and filter values much larger than 0.01. The results are shown in Table 8, using 16 processors.

The sparsity ratios in the first two rows of Table 8 are similar, meaning that sparsification was not helpful to reduce the cost of constructing the preconditioner. This is a difficulty with systems problems. The last row in Table 8 shows that a sparser and less costly preconditioner can be constructed using a dual threshold pattern.

4.3 Comprehensive tests

We collected large SPD matrices from various sources and selected large representative problems and problems that have been tested with approximate inverses by others (Benzi, Kouhia, & Tůma, 1998; Field, 1997; Kolotilina et al., 1999; Benzi, Cullum, & Tůma, 1999). Most of the problems are from structures or solid mechanics applications, but radiation diffusion and pressure Poisson problems are also represented. Table 9 lists the test matrices. Four of these problems could be solved using diagonal preconditioning and are listed in Table 10 for comparison.

We note that there were several problems that we could not solve efficiently. These problems were "bcsstk35" (automobile seat frame and body attachment), "bcsstk37" (track ball), "ct20stif" (engine block), "vibrobox" (Vibroacoustic problem). These problems are not shown in Table 9.

The test results are shown in Table 11. For each matrix, the first row in the table is the result using the pattern of the original matrix A, and the second row shows the best result based on total time. Thresholds of 0, 0.01, 0.02, 0.05, 0.1, 0.2 and levels of 0, 1, 2, 3 were systematically tested. A filter value of 0.05 and a load balance parameter of 0.9 was used in all cases. "Ratio1" and "Ratio2" are the sparsity ratios before and after filtration, respectively. Initial guesses of zero were used and random right-hand sides were used if right-hand sides were not provided. Timings were taken on 4 processors (1 node) on an IBM SP except for problems "pwtk" and "smt" where 16 processors (4 nodes) were used. The results show that best sparsity ratio after

	n	nnz	nnz/n	
sls	18205	1358897	74.6	spherical shell, 2 slides
bcsstk17	10974	428650	39.0	elevated pressure vessel
bcsstk36	23052	1143140	49.5	shock absorber assembly
cfd2	123440	3087898	25.0	pressure matrix, Rothberg
small_ic	15492	1213038	78.3	integrated circuit
eltgroth	68921	1543652	22.3	radiation diffusion
msc23052	23052	1154814	50.0	MSC/Nastran problem
nasasrb	54870	2677324	48.7	shuttle rocket booster
pwtk	217918	11634424	53.3	pressurized wind tunnel
skirt	45361	2533123	55.8	structure from NASA
smt	25710	3753184	145.9	surface mounted transistor

Table 9: Test matrices (all SPD).

	Iter	Solve
bcsstk17	2995	46.2
sls	1992	78.2
eltgroth	206	17.6
smt	2053	299.8

Table 10: Problems that could be solved with diagonal preconditioning, Cf. Table 11.

	Thresh	Level	Ratio1	Ratio2	Iter	Setup	Solve	Total
sls					705	3.0	58.7	61.8
	0.05	1	0.97	0.35	547	3.4	34.0	37.4
bcsstk17					619	0.6	18.0	18.6
	0.20	2	0.76	0.47	458	0.8	10.2	11.0
bcsstk36					1524	1.6	135.3	136.9
	0.02	1	2.84	1.47	664	10.3	73.4	83.7
cfd2					2566	3.1	680.4	683.6
	0.00	1	4.22	1.83	1131	21.3	381.4	402.7
small_ic					2577	3.4	253.6	257.0
	0.10	2	2.01	0.56	702	12.7	57.7	70.4
eltgroth					116	1.8	15.7	17.5
	0.01	1	0.56	0.45	81	2.5	8.4	10.9
msc23052					1321	3.4	176.4	179.8
	0.05	1	2.35	1.14	584	11.8	89.2	101.0
nasasrb					2567	3.4	383.6	387.0
	0.10	2	2.27	1.06	1180	15.4	207.8	223.3
pwtk					†	4.1	†	†
	0.00	1	2.78	1.32	1943	25.9	531.2	557.1
skirt					2800	3.6	489.3	492.9
	0.10	2	3.08	0.79	1177	39.0	185.3	224.3
smt					561	4.4	38.2	42.6
	0.10	1	0.85	0.29	300	5.3	14.7	19.9

Table 11: Comprehensive test results. For each matrix, the first row in the table is the result using the pattern of the original matrix A and the second row shows the best result based on total time.

filtration is less than 2 for this set of problems.

4.4 Implementation scalability

The implementation scalability was tested using a 3-D constant coefficient anisotropic diffusion problem

$$\begin{aligned} au_{xx} + bu_{yy} + cu_{zz} = 1 \quad &\text{in} \quad \Omega = (0,1)^3 \\ u = 0 \quad &\text{on} \quad \partial\Omega \end{aligned}$$

discretized using finite differences on a uniform mesh, with the anisotropic parameters $a = 0.1$, $b = 1$, and $c = 10$.

The results are shown in Table 12. The local problem size was $60 \times 60 \times 60$ grid points, and the global problem size N was scaled up with the number of processors, "Nproc." A level 3 pattern with sparsification gave a sparsity ratio of 1.25. As desired, the setup time scales better

Nproc	N	Iter	Patt	Values	Setup	Solve	/Iter
1	216,000	107	2.2	9.5	12.1	75.3	0.70
8	1,728,000	204	2.6	10.4	13.8	247.9	1.22
27	5,832,000	305	2.9	11.0	14.6	392.7	1.29
64	13,824,000	399	3.2	11.4	15.4	536.6	1.34
125	27,000,000	497	3.2	11.3	15.5	670.9	1.35
216	46,656,000	595	3.6	11.5	15.8	856.4	1.44
343	74,088,000	694	4.0	11.6	16.3	1092.7	1.57
512	110,592,000	790	4.0	12.4	17.4	1278.8	1.62
729	157,464,000	884	4.4	12.8	18.4	1485.7	1.68
1000	216,000,000	979	4.2	12.1	17.1	1710.7	1.75

Table 12: The implementation scalability is tested by increasing the problem size N with the number of processors "Nproc." As desired, the setup time scales better than the time per iteration, which has more global communications.

than the time per iteration, since it has fewer global communications. Timings for components of the setup phase (computing the pattern and values) are also shown. In addition, the number of iterations to convergence varies as the cube root of N, which is in agreement with theory.

5 Conclusions

This paper describes and tests the use of patterns of powers of sparsified matrices for least squares sparse approximate inverse preconditioners and describes the parallel implementation of the software. To obtain high performance, load balancing and the use of local coordinate numbering is necessary. Load balancing is accomplished by repartitioning based on the workload on each processor after the sparsity pattern has been determined.

Sparsification of the matrix before taking matrix powers (called *levels*) is necessary when powers of a matrix have a large number of nonzeros, making the computation of the approximate inverse excessively expensive. Difficult problems that need high matrix powers will typically require sparsification.

To use higher matrix powers, a higher threshold must often also be used. This threshold must not be too high or else the approximate inverse will be too inaccurate. Thus, the threshold and level must be balanced with each other. In our tests, patterns up to level 4 were practical, and thresholds less than 0.3 were accurate. Higher levels are feasible for matrices that have very few nonzeros per row.

The test results also show that the minimum solution time is usually achieved by using very sparse approximations, since denser approximations require more time per iteration and this additional time is usually not compensated by the improved preconditioner accuracy. Also, nonfactorized approximate inverses require much more computation than factorized forms and must be even sparser to be practical to compute.

Computing the pattern for a sparse approximate inverse is much less expensive than computing its values. In practical usage, when a good set of parameters is not known, the cost of computing the approximate inverse can be determined after the pattern has been computed. This technique can be used to abort a lengthy computation of approximate inverses with large numbers of nonzeros.

Filtration of the small nonzeros in the approximate inverse reduces the time per iteration, sometimes dramatically. A filter value between 0.05 to 0.1 does not significantly degrade accuracy and gives good overall results. The use of higher powers or lower thresholds combined with filtration can produce patterns that yield very accurate approximate inverses. These patterns may be reused in a sequence of linear solves.

To solve very large problems, the input parameters may be tuned using smaller versions of the same problem. The sparsity ratios should be similar across problem sizes and the preconditioner construction times should scale linearly. However, the convergence rate would be worse for larger problems and a larger fraction of the time would be spent in the iterative method. Tuning using a smaller problem should take this effect into account by assuming a worse convergence rate for the smaller problem.

The software described in this paper is available at `http://www.llnl.gov/CASC/parasails/`. The software was designed for both SPD factorized and general nonfactorized problems.

References

Alléon, G., Benzi, M., & Giraud, L. (1997). Sparse approximate inverse preconditioning for dense linear systems arising in computational electromagnetics. *Numerical Algorithms*, *16*, 1–15.

Barnard, S. T., Bernardo, L. M., & Simon, H. D. (1999). An MPI implementation of the SPAI preconditioner on the T3E. *Intl. J. High Perf. Comput. Appl.*, *13*, 107–128.

Barnard, S. T., & Clay, R. (1997). A portable MPI implementation of the SPAI preconditioner in ISIS++. In *Proceedings eighth SIAM conference on parallel processing for scientific computing.* Minneapolis, MN.

Benson, M. W., Krettmann, J., & Wright, M. (1984). Parallel algorithms for the solution of certain large sparse linear systems. *Intl. J. Comput. Math.*, *16*, 245–260.

Benzi, M., Cullum, J. K., & Tůma, M. (1999). *Robust approximate inverse preconditioning for the conjugate gradient method* (Tech. Rep. No. LA-UR-99-2899). Los Alamos, NM: Los Alamos National Laboratory.

Benzi, M., Kouhia, R., & Tůma, M. (1998). An assessment of some preconditioning techniques in shell problems. *Communications in Numerical Methods in Engineering*, *14*, 897–906.

Carpentieri, B., Duff, I. S., & Giraud, L. (2000). *Sparse pattern selection strategies for robust Frobenius norm minimization preconditioners in electromagnetism* (Tech. Rep. No. TR/PA/00/05). Toulouse, France: CERFACS.

Chen, K. (1998). On a class of preconditioning methods for dense linear systems from boundary elements. *SIAM J. Sci. Comput.*, *20*, 684–698.

Chow, E. (2000). A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM J. Sci. Comput.*, *21*, 1804–1822.

Chow, E., & Saad, Y. (1998). Approximate inverse preconditioners via sparse-sparse iterations. *SIAM J. Sci. Comput.*, *19*, 995–1023.

Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1991). *Introduction to algorithms.* Cambridge, MA: The MIT Press.

Cosgrove, J. D. F., Díaz, J. C., & Griewank, A. (1992). Approximate inverse preconditioning for sparse linear systems. *Intl. J. Comput. Math.*, *44*, 91–110.

Cosnuau, A. (1996). *Etude d'un préconditionneur pour les matrices complexes dense issues des équations de Maxwell en formulation intégrale* (Tech. Rep. No. 142 328.96/DI/MT). France: ONERA.

Deshpande, V., Grote, M. J., Messmer, P., & Sawyer, W. (1996). Parallel implementation of a sparse approximate inverse preconditioner. In A. Ferreira, J. Rolim, Y. Saad, & T. Yang (Eds.), *Parallel algorithms for irregularly structured problems (irregular '96), santa barbara, ca, 1996* (Vol. 1117, pp. 63–74). Springer-Verlag, Berlin.

Field, M. R. (1997). *An efficient parallel preconditioner for the conjugate gradient algorithm* (Tech. Rep. No. HDL-TR-97-175). Dublin: Hitachi Dublin Laboratory, Trinity College.

Field, M. R. (1998). *Improving the performance of parallel factorised sparse approximate inverse preconditioners* (Tech. Rep. No. HDL-TR-98-199). Dublin: Hitachi Dublin Laboratory, Trinity College.

Field, M. R. (1999). *A parallel factorised sparse approximate inverse preconditioner with improved choice of sparsity pattern* (Tech. Rep. No. HDL-TR-99-214). Dublin: Hitachi Dublin Laboratory, Trinity College.

Gonnet, G. H., & Baeza-Yates, R. (1991). *Handbook of algorithms and data structures* (2nd ed.). Reading, MA: Addison-Wesley.

Gropp, W., Huss-Lederman, S., Nitzberg, B., & Lusk, E. (1998). *MPI the complete reference: The MPI-2 extensions.* Cambridge, MA: The MIT Press.

Gropp, W., Kaushik, D., Keyes, D., & Smith, B. (1999). Toward realistic performance bounds for implicit CFD codes. In *Proceedings of parallel CFD99.* Halifax, Nova Scotia.

Grote, M., & Huckle, T. (1997). Parallel preconditioning with sparse approximate inverses. *SIAM J. Sci. Comput.*, *18*, 838–853.

Grote, M., & Simon, H. D. (1993). Parallel preconditioning and approximate inverses on the Connection Machine. In R. F. Sincovec, D. E. Keyes, L. R. Petzold, & D. A. Reed (Eds.), *Parallel processing for scientific computing* (Vol. 2, pp. 519–523). SIAM, Philadelphia, PA.

Hendrickson, B., & Devine, K. (to appear). Dynamic load balancing in computational mechanics. *Comp. Meth. Applied Mechanics and Engineering.*

Huckle, T. (1996a). *Efficient computation of sparse approximate inverses* (Tech. Rep. No. 342/04/96). München, Germany: Technische Universität München.

Huckle, T. (1996b). PVM-implementation of sparse approximate inverse preconditioners for solving large sparse linear equations. In *Lecture notes in comput. sci.* (Vol. 1156, pp. 166–173). Springer-Verlag, New York.

Huckle, T. (1997). Approximate sparsity patterns for the inverse of a matrix and preconditioning. In R. Weiss & W. Schönauer (Eds.), *Preliminary proceedings IMACS world congress on scientific computation 1997.* Berlin.

Karypis, G., & Kumar, V. (1998). Multilevel algorithms for multi-constraint graph partitioning. In *Supercomputing 98.* Orlando, FL.

Knuth, D. E. (1973). *Sorting and searching* (Vol. 3). Reading, MA: Addison-Wesley.

Kolotilina, L. Y. (1988). Explicit preconditioning of systems of linear algebraic equations with dense matrices. *J. Soviet Math.*, *43*, 2566–2573.

Kolotilina, L. Y., Nikishin, A. A., & Yeremin, A. Y. (1992). Factorized sparse approximate inverse (FSAI) preconditionings for solving 3D FE systems on massively parallel computers. II. Iterative construction of FSAI preconditioners. In R. Beauwens & P. de Groen (Eds.), *Proceedings IMACS intl. symp. iterative methods in linear algebra* (pp. 311–312). Brussels, Belgium.

Kolotilina, L. Y., Nikishin, A. A., & Yeremin, A. Y. (1999). Factorized sparse approximate inverse preconditionings. IV: Simple approaches to rising efficiency. *Num. Lin. Alg. Appl.*, *6*, 515–531.

Kolotilina, L. Y., & Yeremin, A. Y. (1993). Factorized sparse approximate inverse preconditionings I. Theory. *SIAM J. Matrix Anal. Appl.*, *14*, 45–58.

Nieplocha, J., Littlefield, R., & Rosing, M. (1995). Beyond message passing: A case for one-sided communication in mpi. In *MPI developers conference 1995.* University of Notre Dame, IN.

Saad, Y., & Malevsky, A. (1995). P_SPARSLIB: A portable library of distributed memory sparse iterative solvers. In *Proceedings of parallel computing technologies (PaCT-95), 3rd international conference.* St. Petersburg.

Tang, W.-P., & Wan, W. L. (1998). *Sparse approximate inverse smoother for multi-grid* (Tech. Rep. No. CAM 98-18). Los Angeles, CA: Department of Mathematics, University of California.

Tuminaro, R. S., Shadid, J. N., & Hutchinson, S. A. (1998). Parallel sparse matrix vector multiply software for matrices with data locality. *Concurrency: Practice and Experience*, *10*, 229–247.

Vavasis, S. A. (1992). Preconditioning for boundary integral equations. *SIAM J. Matrix Anal. Appl.*, *13*, 905–925.