

MSP: A Class of Parallel Multistep Successive Sparse Approximate Inverse Preconditioning Strategies*

Kai Wang[†] and Jun Zhang[‡]

Laboratory for High Performance Scientific Computing and Computer Simulation,
Department of Computer Science, University of Kentucky,
Lexington, KY 40506-0046, USA

January 10, 2002

Abstract

We develop new concepts and parallel algorithms of multistep successive preconditioning strategies to enhance efficiency and robustness of standard sparse approximate inverse preconditioning techniques. The key idea is to compute a series of simple sparse matrices to approximate the inverse of the original matrix. Studies are conducted to show the advantages of such an approach in terms of both improving preconditioning accuracy and reducing computational cost. Numerical experiments using one prototype implementation to solve a few general sparse matrices on a distributed memory parallel computer are reported.

Key words: Sparse matrices, parallel preconditioning, sparse approximate inverse.

AMS subject classifications: 65F10, 65F50, 65N55, 65Y05.

1 Introduction

The need for solving very large sparse linear systems arising from many important applications has been pushing the development of sparse linear system solvers for parallel computers. Direct solvers, based on a factorization of the sparse matrices, are extremely

*Technical Report No. 331-02, Department of Computer Science, University of Kentucky, Lexington, KY, 2002. This research was supported in part by the U.S. National Science Foundation under grants CCR-9902022, CCR-9988165, and CCR-0092532, in part by the Japanese Research Organization for Information Science & Technology, and in part by the University of Kentucky Research Committee.

[†]E-mail: kwang0@csr.uky.edu.

[‡]E-mail: jzhang@cs.uky.edu. URL: <http://www.cs.uky.edu/~jzhang>.

robust, but their memory and floating point operation requirements grow as a nonlinear function of the matrix dimension, because original zeros fill in during factorization. While preconditioned Krylov subspace methods are considered to be some of the most suitable candidates for solving large sparse linear systems and the parallelization of many popular Krylov subspace accelerators no longer poses a big problem, the quest for robust parallel preconditioners is still challenging [26].

Simple parallel preconditioners such as Jacobi or block Jacobi methods, although are easy to implement, have inherent weakness of being not robust for difficult problems. Their lack of robustness inhibits them from being used in industrial standard software packages. Other parallel preconditioners based on multicoloring strategy may have restricted applicability as the parallelism extracted from this strategy is limited. Domain decomposition based methods have been exploited extensively in parallel linear system solvers and preconditioners [4, 10, 21, 31]. Important progress has been made recently concerning the parallelization of incomplete LU (Cholesky) factorization preconditioners [20, 24, 25]. Furthermore, there are additional two classes of more advanced parallelizable preconditioners that seem to be more robust than the simple preconditioners. One is based on multilevel block incomplete LU (ILU) factorization, which is built on successive block independent set ordering and block ILU factorization. For detailed discussions on several sequential and parallel multilevel ILU preconditioning techniques, we refer readers to [1, 2, 27, 28, 29, 30].

In this paper, we will be concerned with another class of parallelizable preconditioning techniques based on computing a sparse approximate inverse of the original matrix [7, 14, 18, 33]. These preconditioners have the property of possessing high degree of parallelism in the solution process and are shown to be efficient for certain type of problems. However, for large classes of general sparse matrices, straightforward implementations of many sparse approximate inverse techniques lead to inefficient preconditioners on distributed memory parallel computers. Realistic parallel implementation to compute these preconditioners is also a big problem, due to the nature of interprocessor communications associated with the dynamic sparsity pattern search.

Some recently proposed parallel implementation of sparse approximate inverse preconditioner construction may require the sparsity pattern to be specified *a priori* [12]. For certain matrices, this may lead to poor sparse approximate inverse preconditioners, or to very high computational cost if some higher level static sparsity pattern is specified. The balance between preconditioner construction cost and preconditioner accuracy presents a challenge to practical efficient parallel implementation of many sparse approximate inverse techniques.

In this paper, we investigate a class of multistep successive sparse approximate inverse preconditioning techniques. A sequence of sparse matrices are computed cheaply using an existing parallel sparse approximate inverse technique. The product of these sparse matrices is used to approximate the true inverse of the original matrix. Thus, instead of computing a costly high accuracy sparse approximate inverse preconditioner in one shot, we compute a series of cheap sparse approximate inverse preconditioners to achieve the effect of a high accuracy preconditioner. The sparsity pattern is adjusted when a new

sparse approximate inverse matrix is computed.

We develop algorithms to compute multistep successive sparse approximate inverse preconditioners efficiently on high performance parallel computers. Specifically we design and test some prototype multistep successive sparse approximate inverse preconditioners to show that they are both robust and scalable with different number of processors.

This paper is organized as follows. In Section 2, we introduce some basic knowledge on sparse approximate inverse preconditioning techniques. We then give detailed discussion on our motivation, ideas, and computational strategies for multistep successive preconditioning in Section 3. The implementation details are discussed in Section 4. In Section 5, we give some numerical results to demonstrate the advantages of the new preconditioning strategies. Section 6 contains some brief concluding remarks.

2 Background

Many scientific and engineering models are established on the basis of a few partial differential equations governing certain physical properties and quantities. Computer simulations and modelings of such problems require discrete solution of these partial differential equations. Most sparse matrices are derived from discretizing linear or nonlinear partial differential equations by finite difference, finite element, finite volume, or other methods on structured or unstructured domains. Thus, real life sparse matrices may be very large and may not have regular structures. Efficient preconditioned iterative solution of general sparse matrices is therefore an important step in conducting large scale computer simulation and modeling. One current research focus along this line is to construct robust parallel preconditioners for unstructured sparse matrices, so that the preconditioned linear systems can be solved efficiently by an iterative accelerator on parallel computers.

2.1 Sparse approximate inverse preconditioning

A sparse approximate inverse is a sparse matrix M which is a good approximation to A^{-1} , the inverse of a general nonsingular sparse matrix A . The major driving force behind the search for efficient sparse approximate inverse preconditioners is their potential advantages in parallel computing. The idea is that, if such a matrix M can be constructed by some means, the preconditioning process is just a matrix-vector operation and is relatively easy to implement on high performance parallel computers [23], compared to the inherent sequential nature of the triangular solution procedures in the incomplete LU factorization preconditioning techniques.

There exist several techniques to construct sparse approximate inverse preconditioners. They can be roughly categorized into three classes [8], sparse approximate inverses based on Frobenius norm minimization [14, 18], sparse approximate inverses computed from an ILU factorization [16], and factored sparse approximate inverses [7, 33, 34]. Each of these classes contains a variety of different constructions and each of them has its own merits and drawbacks. Since our new concept of multistep successive preconditioning

strategies can be applied to almost all of these sparse approximate inverse preconditioning techniques, we will not go into details to elaborate every but one particular sparse approximate inverse approach.

The sparse approximate inverse technique that we discuss here is based on the idea of least squares approximation. This is also the one that initially motivated research in sparse approximate inverse preconditioning [5, 6]. Consider a sparse linear system

$$Ax = b, \tag{1}$$

where A is a nonsingular general square matrix of order n . The convergence rate of a Krylov subspace accelerator applied directly on (1) may be slow due to the potential ill-conditioning of the matrix A . In order to speed up convergence rate of iterative methods, we may transform (1) into an equivalent system

$$MAx = Mb, \tag{2}$$

where M is a nonsingular matrix of order n . If M is a good approximation to A^{-1} in some sense, then MA can be a good approximation to the identity matrix I . It follows that the equivalent system (2) will be easier to solve, compared to solving (1), by a Krylov subspace accelerator.

A particular class of sparse approximate inverse preconditioners is constructed based on the Frobenius norm minimization idea. Since we want M to be a good approximation to A^{-1} , it is ideal if $MA \approx I$. This approach is to approximate A^{-1} from the left, and M is called left preconditioner. It is also possible to approximate A^{-1} from the right, so that $AM \approx I$, which is termed as right preconditioner. In the case of right preconditioning, the equivalent preconditioned system analogous to (2) is

$$AMy = b, \quad \text{and} \quad x = My. \tag{3}$$

In fact, the right preconditioning approach is easier for us to illustrate the Frobenius norm minimization idea, which will be described in detail in the following paragraphs.

In order to have $AM \approx I$, we want to minimize the functional

$$f(M) = \min_M \|AM - I\| \tag{4}$$

for all possible nonsingular square matrices M of order n , with respect to a certain norm. Without any constraint on M , the minimization problem (4) has an obvious solution, i.e., $M = A^{-1}$. This obvious solution is undesirable for at least two reasons. First, the computational cost for solving the unconstrained minimization problem (4) is prohibitively high. Second, for most sparse matrices A , their inverses A^{-1} are dense, which will cause memory problem for large scale matrices encountered in many practical applications.

Thus we are interested in a constrained minimization such that M has a certain sparsity pattern (nonzero structure), i.e., only certain entries of M are allowed to be nonzero. Given a set of sparsity patterns (this set is usually unknown *a priori*) Ω , we minimize the functional

$$f(M) = \min_{M \in \Omega} \|AM - I\|. \tag{5}$$

Although any norms can potentially be used in the above definition, a particularly convenient norm is the Frobenius norm which is defined for a matrix $A = (a_{ij})_{n \times n}$ as $\|A\|_F = \sqrt{\sum_{i,j=1}^n a_{ij}^2}$. Hence, the minimization problem (5) can proceed as (using square for convenience)

$$\|AM - I\|_F^2 = \sum_{k=1}^n \|(AM - I)e_k\|_2^2 = \sum_{k=1}^n \|Am_k - e_k\|_2^2, \quad (6)$$

where m_k and e_k are the k th column of the matrix M and that of the identity matrix I , respectively. It follows that the minimization problem (5) is equivalent to minimizing the individual functions

$$\|Am_k - e_k\|_2, \quad k = 1, 2, \dots, n \quad (7)$$

with certain restriction placed on the sparsity pattern of m_k . In other words, each column of M can be computed independently. For a moment, we assume that the sparsity pattern of m_k is given *a priori*, i.e., there are a few, say n_2 , entries of m_k at certain locations are allowed to be nonzero, the rest of the entries are forced to be zero. Denote those n_2 nonzero entries of m_k as \tilde{m}_k . We choose the n_2 columns of A corresponding to \tilde{m}_k , denote it by A_k . Since A is sparse, the submatrix A_k has many rows that are identically zero. After removing those zero rows, we have a reduced matrix \tilde{A}_k with n_1 not identically zero rows. The individual minimization problem (7) is reduced to a least squares problem of order $n_1 \times n_2$

$$\min_{\tilde{m}_k} \|\tilde{A}_k \tilde{m}_k - \tilde{e}_k\|_2, \quad k = 1, 2, \dots, n. \quad (8)$$

We note that the matrix \tilde{A}_k is usually a very small rectangular matrix. It has full rank if A is nonsingular.

There are a variety of methods available to solve the least squares problem (8). One approach, proposed by Grote and Huckle [18], is to solve (8) using a QR factorization as

$$\tilde{A}_k = Q_k \begin{pmatrix} R_k \\ 0 \end{pmatrix},$$

where R_k is a nonsingular upper triangular $n_2 \times n_2$ matrix. Q_k is an $n_1 \times n_1$ orthogonal matrix, such that $Q_k^{-1} = Q_k^T$. The least squares problem (8) is solved by first computing $\tilde{c}_k = Q_k^T \tilde{e}_k$ and then obtaining the solution as $\tilde{m}_k = R_k^{-1} \tilde{c}_k(1 : n_2)$. In this way, \tilde{m}_k can be computed for each $k = 1, 2, \dots, n$, independently. This yields an approximate inverse matrix M , which minimizes $\|AM - I\|_F$ for the given sparsity pattern.

The inherent parallelism is obvious in the process of computing \tilde{m}_k independently of each other. It can be implemented straightforwardly on a model Parallel Random Access Machine without considering communication cost [23]. However, in realistic implementations on distributed memory parallel computers, the communication cost associated with data movement does pose some difficulty for naive implementations.

2.2 Static and dynamic sparsity patterns

The remaining problem for constructing a sparse approximate inverse preconditioner seems to decide how to choose a good sparsity pattern for M . There are a few heuristic strategies,

both static and dynamic sparsity pattern approaches have been proposed [14, 15, 22].

The dynamic sparsity pattern strategies can usually compute better sparse approximate inverse preconditioners, given a certain sparsity ratio (density) for M . However, dynamic strategies are usually expensive and more difficult to implement on parallel computers, although the implementation by Barnard et al. [3] is certainly attractive. Let us take a look at, for example, the dynamic strategy proposed by Grote and Huckle [18]. Given an initial sparsity pattern the least squares problem (8) is solved and an initial \tilde{m}_k is computed. Another subminimization is then conducted to find which of the zero positions in m_k can be augmented into \tilde{m}_k so that the residual in (8) can be reduced most. The difficulty is associated with the matrix \tilde{A}_k . As \tilde{m}_k is augmented, the corresponding rows and columns of \tilde{A}_k have to be augmented also. These rows and columns are not necessarily in the local processor, and have to be transferred from other processors on a distributed memory architecture. If this sparsity pattern search procedure is to be executed for a few times for each k , the communication cost is likely to present a big problem on a distributed memory computers with many processors on which many \tilde{m}_k are computed and updated simultaneously.

It seems that a static (*a priori*) sparsity pattern can be more attractive to implement on distributed memory parallel computers. This is, however, not an easy answer. Although communications are still needed to assemble local matrix \tilde{A}_k , the biggest problem for static sparsity pattern is that, for general sparse matrices, there is no useful information to determine whether a static pattern is a good one before a sparse approximate inverse is computed and tested. On the other hand, for certain matrices, numerical experiments show that some static sparsity patterns may be good. For example, banded sparsity patterns can be used for banded matrices [6, 19].

A particularly useful and effective strategy is to use the sparsity pattern of the matrix A or A^T . Chow [13] proposes to use sparsified patterns of A as the sparsity pattern for M . Here “sparsified” means that certain small entries of A are removed before its sparsity pattern is extracted. For achieving higher accuracy, the sparsity patterns of (sparsified) A^2, A^3, \dots , may be used. Here the matrices A^2, A^3, \dots , are not explicitly computed, only their sparsity patterns are extracted from that of the matrix A with binary operations. Several auxiliary strategies are proposed to make such an approach practically useful. A software package, ParaSails, which implements the static sparsity pattern sparse approximate inverse preconditioning, has been announced to the public [11, 12].

3 Multistep Successive Preconditioning

It has been noticed that high accuracy sparse approximate inverse preconditioners may be difficult and very expensive to compute using a static sparsity pattern as in ParaSails [12, 25]. Experimental results indicate that, compared to incomplete Cholesky factorization, sparse approximate inverse preconditioning wins only when the factorization is not required to be very accurate [25]. This is because it is very difficult to determine a good static sparsity pattern *a priori*. The following is some test data using ParaSails with differ-

Sparsity Pattern	Sparsity Ratio	Iteration	Setup Time	Solution Time
A	0.25	754	2.0	39.3
A^2	0.47	539	40.0	33.7
A^3	0.80	243	491.0	20.4

Table 1: Test results using ParaSails from [11].

ent levels of sparsity patterns from Chow’s paper [11]. It is to solve a symmetric positive definite matrix with $n = 12,205$ and about 1.4 million nonzeros.

We can see that use of higher level sparsity patterns, such as those of A^2 and A^3 , does lead to better sparse approximate inverse preconditioners. This is indicated by the reduction in the number of preconditioned iterations (column 3). However, the CPU time in seconds needed to construct the preconditioners with higher accuracy (the Setup Time, column 4) increases substantially. The reduction in the solution time (column 5) does not compensate for the huge increase in setup time. Hence, it is difficult to justify in this case to compute higher accuracy (more robust) sparse approximate inverse preconditioners. Unless, as Chow points out [11], we are in a situation that the higher accuracy sparsity pattern will be used in latter computation, the initial high cost of extracting high accuracy sparsity pattern may be amortized.

We can approach the problem of choosing a suitable sparsity pattern in another way. Suppose a (*simple and cheap*) sparse approximate inverse preconditioner M_0 is computed for the matrix A , using any available sparse approximate inverse construction techniques, e.g., ParaSails with the sparsified pattern of A . If somehow we find that M_0 is not very efficient, we can compute another sparse approximate inverse preconditioner M_1 for the preconditioned linear system

$$M_0Ax = M_0b. \tag{9}$$

Here, for discussion convenience, we return back to use left preconditioning. We note that the systems (1) and (9) are equivalent, if M_0 is nonsingular as assumed. Thus, we compute another (*simple and cheap*) sparse approximate inverse preconditioner M_1 for the matrix $A_1 = M_0A$. The combined preconditioner is then M_1M_0 for the matrix A . Here are a few comments to justify our successive sparse approximate inverse preconditioner in the form of product matrix M_1M_0 .

- If we use the sparsity pattern (sparsified, if applicable) of A for M_0 , which is cheap as we see from the experimental data in Table 1, the sparsity pattern of $A_1 = M_0A$ is similar to that of A^2 . However, computing a level 2 sparse approximate inverse preconditioner for A using the sparsity pattern of A^2 is different from computing a simple level 1 sparse approximate inverse preconditioner for A_1 using the sparsity pattern of A_1 . The latter is much cheaper.
- The computation of $A_1 = M_0A$ can be done efficiently on parallel computers. If p is the average number of nonzeros in each row of A , the cost of computing A_1 is approximately equal to p folds of applying M_0 on a dense vector, or less than that of

$p/2$ preconditioned iteration steps, assuming that M_0 uses the sparsity pattern of A . Moreover, when we sparsify A_1 using a threshold parameter, the obtained sparsity pattern is more accurate than that of A^2 , as it reflects the true pattern of A_1 . The pattern of A^2 is computed from that of A using binary operations on the graph of A , without considering the size of the entries of A^2 . Thus some useful information may get lost.

- If M_0 is an approximation to A^{-1} , albeit not a very good one, then $A_1 = M_0A$ tends to be closer to I than A does, or A_1 tends to be more diagonally dominant than A does. Thus, computing a sparse approximate inverse for A_1 is usually easier than computing one for A , given the same conditions. In Section 5.2, we give some test experimental results to justify this argument.
- Intuitively the inverse A^{-1} of a sparse matrix A is dense. Then usually an accurate approximation M for A^{-1} should be a dense matrix. This conflicts with our initial goal which is to find a sparse approximate inverse matrix M . However, if using the product of two sparse matrices M_1M_0 to approximate A^{-1} , we expect that M_1M_0 may be capable of holding more information than a single matrix M is. In this viewpoint, using M_1M_0 as a preconditioner is to some extent like using the factored sparse approximate inverse preconditioners [7, 33, 34].

A reader with recursive thinking will have already figured out the next step in the successive sparse approximate inverse procedure. If M_1M_0 is not good enough for preconditioning the matrix A in question, we compute a third sparse approximate inverse matrix M_2 for the product matrix $A_2 = M_1M_0A$. This procedure can be continued for a few times to obtain a sequence of sparse matrices M_0, M_1, \dots, M_l , such that $M_lM_{l-1} \cdots M_1M_0 \approx A^{-1}$. If each M_i is good (but not necessarily very good) in some sense, we may expect that

$$\lim_{l \rightarrow \infty} M_lM_{l-1} \cdots M_1M_0 = A^{-1}.$$

The algorithm for computing a multistep successive sparse approximate inverse preconditioner can be written as follows.

ALGORITHM 3.1

0. Given step number “step”, threshold tolerance “thre”, and filter parameter “filt”
1. Let $l = \text{step}$, $\tau = \text{thre}$, $\epsilon = \text{filt}$, $A_0 = A$
2. For ($i = 1; i \leq l; i++$)
3. Sparsify A_{i-1} with respect to τ
4. Compute a sparse approximate inverse $M_{i-1} \approx A_{i-1}^{-1}$
5. Drop small entries of M_{i-1} with respect to ϵ
6. Compute $A_i = M_{i-1} A_{i-1}$
7. End
8. Sparsify A_l with respect to τ
9. Compute a sparse approximate inverse $M_l \approx A_l^{-1}$
10. Drop small entries of M_l with respect to ϵ
11. $\prod_{i=0}^l M_i$ is the preconditioner for $Ax = b$

Because the matrix M_i becomes denser and denser as i increases, in each step we keep them sparse by dropping certain small size entries before and after the sparse approximate inverse computation. This is indicated in Lines 3, 5, 8 and 10 in the algorithm. Such sparsification procedures are called preprocessing (τ or “thre”) and postprocessing (ϵ or “filt”) phases respectively. We note that when $\text{step} = 0$, the algorithm is the same as a standard sparse approximate inverse algorithm.

4 Implementation Details

The success of general sparse linear system solvers depends on sophisticated implementations as heavily as on the innovative underlying ideas. In this section, we discuss several implementation issues that need to be addressed to build efficient software for distributed memory parallel computers.

Matrix distribution. The matrix will be distributed to individual processors either by rows or by columns. In some applications, the computational domain may already be partitioned and the matrix may be generated locally in each processor. We should take these situations into consideration when we design particular algorithms and codes for particular applications. However, the matrix partition and distribution itself is not a main topic of this paper.

Sparse approximate inverse. There are a few sparse approximate inverse algorithms published by different researchers [7, 14, 34]. There are also a few parallel implementations of some sparse approximate inverse preconditioned iterative solvers [3, 11, 12]. We can use any existing sparse approximate inverse packages, such as ParaSails of Chow and SPAL1.1 of Barnard et. al. [3], as the backbones for our multistep successive sparse approximate inverse preconditioned iterative solvers.

Maintaining sparsity. In Chow’s implementation [11, 12], the matrix A is first sparsified, i.e., is first removed of small size entries (using the threshold parameter “thre”), and then its graph structure is extracted to build sparsity pattern for M . Higher level sparsity patterns are built from the sparsity patterns of those of A^2, A^3, \dots , without actually computing A^2, A^3, \dots . The patterns of A^2, A^3, \dots , are obtained from that of sparsified A with binary operations on the graphs. Because, in standard sparse approximate inverse computation, the values of the matrix A^2, A^3, \dots , are not used. In our case, the values of the matrix $A_1 = M_0A$, as well as its sparsified graph, will be used to construct the next step preconditioner M_1 . Sparsification strategies (e.g., removing small size entries) can be applied to A, M_0 and A_1 . Thus, our multistep strategies may actually be cheaper than those implemented in Chow’s ParaSails. It is intuitively correct that our strategies are more likely to build better sparsity patterns step by step, if the sizes of the entries are indicator of certain locations of large entries of the sparse approximate inverse matrix. If this heuristic is not valid, then the original basis for ParaSails is also flawed. Hence, we ex-

pect that the multistep successive sparse approximate inverse preconditioning strategy is *unlikely* to generate worse preconditioners than the original standard sparse approximate inverse preconditioning technique used as its backbone.

Parallel implementation issues. A processor computing a certain column m_k may require rows of A (\tilde{A}_k) stored on other processors. These other processors cannot in general predict that the first processor needs to communicate. This one-sided communication can be accomplished with specialized software or can be simulated with message passing.

In a message passing environment, processors send requests for rows that they need. Processors must also occasionally probe for and service such requests from other processors. This is the technique used by Barnard et. al. [3]. 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 [11, 13].

5 Experimental Results

We conduct a few numerical experiments using a preliminary prototype code with multistep successive sparse approximate inverse preconditioning strategies outlined in the previous sections. This particular implementation uses ParaSails of Chow [12] as the backbone to build our multistep sparse approximate inverse preconditioner. For this reason, we refer to our preconditioner here as *MultiSails* with different steps. It is also our understanding that our idea of multistep successive preconditioning can be applied to many existing sparse approximate inverse techniques and codes as well. The MultiSails code is mostly written in C programming language, with interprocessor communications being handled by MPI.

The computations are carried out on a 32 processor (750MHz) subcomplex of an HP superdome (supercluster) at the University of Kentucky. Unless otherwise indicated explicitly, 4 processors are used in our numerical experiments. When computing $M_i \approx A_i^{-1}$ in each step, we use the (sparsified) pattern of A_i as the *a priori* sparsity pattern.

In all tables containing numerical results, “ n ” denotes the dimension of the matrix; “ nnz ” represents the number of nonzeros in the sparse matrix; “ np ” is the number of processors used; “ $iter$ ” shows how many iterations it takes for the preconditioned GMRES(50) to reduce residual norm by 8 orders of magnitude. We also set an upper bound of 5000 for the GMRES iteration; a symbol “-” in a table indicates lack of convergence. Similarly, “ s -ratio” stands for the sparsity ratio. In MultiSails, this is the sum of the number of nonzero entries of each M_i divided by the number of nonzero entries of original matrix A . “ $setup$ ” is the total CPU time in seconds for constructing the preconditioner; “ $solve$ ” is the total CPU time in seconds for solving the given sparse matrix using the preconditioner; “ $total$ ” is the sum of “ $setup$ ” and “ $solve$ ”. “ $thre$ ” and “ $flit$ ” are two parameters used by Chow in ParaSails [11]. In MultiSails we also use these two parameters to keep the memory cost small in each step as in Algorithm 3.1. The content in the parentheses following “PS” indicates the *a priori* pattern used in ParaSails, e.g., PS(A^2) means that we use the sparsity

pattern of A^2 . Similarly, we use “MS” to denote MultiSails, the number in the followed parentheses is the step number, e.g., MS(2) means a 2 step MultiSails preconditioner.

5.1 Test Problems

In this section, we introduce the test problems which will be used in our experiments. The right hand sides of all linear systems are constructed by assuming that the solution is a vector of all ones. The initial guess is a zero vector.

Convection-diffusion problem. The two dimensional convection-diffusion problem

$$-u_{xx} - u_{yy} - 10(\sin x \cos \pi y u_x - \cos \pi x \sin y u_y) = 0, \quad (10)$$

is defined on the unit square. Here the so-called Reynolds number value is 10. Dirichlet boundary condition is assumed, but the artificial right hand side mentioned previously is used. The equation is discretized by using the standard 5-point central difference scheme. The resulting matrix is referred to as the 5-point matrix.

A three dimensional convection-diffusion problem (defined on a unit cube)

$$u_{xx} + u_{yy} + u_{zz} + 1000(p(x, y, z)u_x + q(x, y, z)u_y + r(x, y, z)u_z) = 0 \quad (11)$$

is used to generate some large sparse matrices to test the implementation scalability of MultiSails. Here the convection coefficients are chosen as

$$\begin{aligned} p(x, y, z) &= x(x-1)(1-3y)(1-2z), \\ q(x, y, z) &= y(y-1)(1-2z)(1-2x), \\ r(x, y, z) &= z(z-1)(1-2x)(1-2y). \end{aligned}$$

The Reynolds number value for this problem is 1000. Equation (11) is discretized by using the standard 7-point central difference scheme and the 19-point fourth order compact difference scheme [32]. The resulting matrices are referred to as the 7-point and 19-point matrices respectively.

Test matrices. We also use MultiSails to solve a few sparse matrices listed in Table 2.

The FIDAP matrices ¹ were extracted from the test problems provided in the FIDAP package [17]. They arise from coupled finite element discretization of Navier-Stokes equations modeling incompressible fluid flows. Some FIDAP matrices may have zero main diagonals and they are difficult to solve by standard incomplete LU factorization preconditioners without high levels of fill-in.

The RAEFSKY1 and RAEFSKY2 matrices are from modeling incompressible flow in pressure driven pipe, and are available from the University of Florida Sparse Matrix Collection. ² The other matrices are from the well known Harwell-Boeing sparse matrix collection.

¹All FIDAP matrices are available online from MatrixMarket of the National Institute of Standards and Technology (<http://math.nist.gov/MatrixMarket>).

²<http://www.csis.ufl.edu/~davis/sparse>.

Matrices	n	nnz	Description
FIDAP024	2283	48733	nonsymmetric forward roll coating
FIDAP028	2603	77653	two merging liquids with one external interior interface
FIDAP031	3909	115299	dilute species deposition on a tilted heated plate
FIDAP036	3079	53851	chemical vapor deposition
FIDAP037	3565	67591	flow of plastic in a profile extrusion die
FIDAPM08	3876	103076	developing flow, vertical channel (angle = 0, Ra = 1000)
FIDAPM10	3046	53842	2D flow over multiple heat sources in a channel
PORES2	1224	9613	reservoir modeling
SHERMAN1	1000	3750	oil reservoir modeling, black oil simulation, shale barriers
PSMIGR1	3140	543162	demography, US intercounty migration 1965-1970
RAEFSKY1	3242	294276	flow in pressure driven pipe, time = 05
RAEFSKY2	3242	294276	flow in pressure driven pipe, time = 25

Table 2: Information about some sparse matrices used in the experiments.

5.2 Comparison of solving MA and A

First, we use ParaSails, the software package developed by Chow, to compute a sparse approximate inverse matrix M for the matrix A (using the sparsity pattern of A). We then compute another sparse approximate inverse preconditioner for the product matrix MA . The purpose of this comparison is to show that MA usually is more attractive than A to be used to construct a sparse approximate inverse preconditioner, which means Equation (2) may be easier to solve, compared to solving Equation (1).

The test results listed in Table 3 are from solving two dimensional convection-diffusion problem (10). The first column is the number of rows (unknowns) of the matrices. The results in the second and third columns are to solve $Ax = b$ using a sparse approximate inverse preconditioner with the sparsity pattern of A^2 . The results in the fourth and fifth columns are to solve $MAx = Mb$, which can be divided into two steps. First we use the sparsity pattern of A to obtain a sparse matrix $M \approx A^{-1}$, then we solve $MAx = Mb$ using a sparse approximate inverse preconditioner with the sparsity pattern of MA . In the experiments, we set the parameters “thre” and “filt” in ParaSails to be 0, so that it does not drop anything during the preprocessing and postprocessing phases [12]. This implementation makes the sparsity pattern of MA the same as that of A^2 .

We can see from Table 3 that the iteration numbers needed to solve the matrix MA are usually 20% less than that to solve the matrix A directly. That means under the same sparsity pattern or preconditioner density, which is indicated in the “s-ratio” columns, preconditioning matrix MA can get better convergence results than preconditioning matrix A directly. This property motivates us to develop multistep successive sparse approximate inverse preconditioning strategies.

5.3 Properties of MultiSails

In this subsection, we present results from a few numerical experiments to demonstrate some favorable properties of multistep sparse approximate inverse preconditioners.

n	$Ax = b$		$MAx = Mb$	
	s-ratio	iter	s-ratio	iter
100^2	2.58	195	2.58	139
200^2	2.59	354	2.59	249
250^2	2.59	443	2.59	354
300^2	2.59	535	2.59	400
350^2	2.59	576	2.59	427
400^2	2.60	681	2.60	536
450^2	2.60	821	2.60	625
500^2	2.60	864	2.60	688

Table 3: Comparison of preconditioning MA and A for solving the 5-point matrices.

Diagonal dominance property. Figure 1 depicts the relationship between the number of steps in constructing the multistep sparse approximate inverse preconditioner and the ratio of strongly diagonally dominant rows of A_i in solving a few sparse matrices using MultiSails. The number after the matrix name in Figure 1 denotes the iteration number to solve the given matrix using the multistep sparse approximate inverse preconditioner with 7 steps.

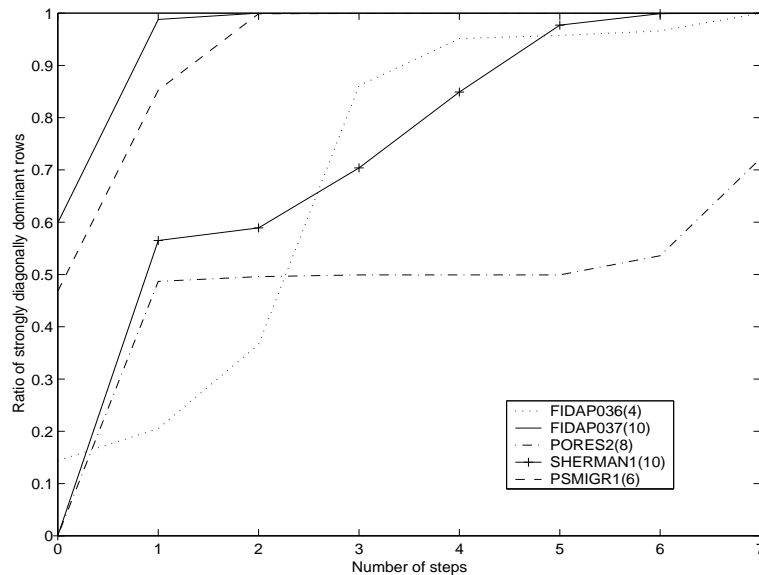


Figure 1: Relationship between the number of steps and the ratio of strongly diagonally dominant rows.

From Figure 1 we can see that when the step number increases, the ratio of strongly diagonally dominant rows of A_i increases quickly. In 4 of the test cases, the strongly diagonal dominance ratio finally reaches 1.0, i.e., 100%, after only a few steps. It is well known that diagonally dominant matrices are comparably easy to solve. So after 7 steps, all these matrices can be solved with the multistep sparse approximate inverse

steps	thre	flt	s-ratio	iter	setup	solve	total
0	0.03	0.03	0.38	-	0.3	-	-
1	0.03	0.03	1.01	1439	1.7	2.5	4.2
2	0.03	0.03	1.43	573	4.4	1.8	6.1
3	0.03	0.03	1.62	392	8.0	1.4	9.4
4	0.03	0.03	1.72	398	14.2	1.7	15.9
5	0.03	0.03	1.78	342	19.5	1.8	21.3

Table 4: Comparison of MultiSails with different number of steps to solve the FIDAP031 matrix.

preconditioner in no more than 10 iterations.

In the experiments we also find that when the strongly diagonally dominant row ratio approaches 1, the structure of A_i tends to be similar to that of the identity matrix I with many small size offdiagonal entries, compared to the magnitudes of the main diagonal entries. It is possible to use a diagonal matrix to approximate the strongly diagonally dominant product matrix. So that we only need to compute the main diagonal entries of the last matrix and its inverse can be computed straightforwardly.

Sparsity ratio and iteration number. Table 4 gives some results from using MultiSails with different steps to solve the FIDAP031 matrix. We note that 0 step here can be regarded as a standard sparse approximate inverse algorithm, i.e., ParaSails in the current case. We point out that if we use an oversparsified pattern of A to construct a preconditioner for A at the 0th step, the resulting preconditioner may not converge. However, this “poor” preconditioner can be used as M_0 in MultiSails as the basis to construct M_1 , and M_1M_0 may make the preconditioned solver converge. In our situation, we think M_1M_0 may still not be good enough, because it converges after 1439 iterations. We then use M_1M_0 as the basis to construct M_2 . In our tests, $M_2M_1M_0$ seems to be a good preconditioner for A , and it converges after 573 iterations. Continue doing this, we find that the multistep preconditioner converges in 342 iterations after 5 steps.

Our other experiments also indicate that a larger step number leads to better convergence results. But it is not the case that the more steps in MultiSails the better the constructed preconditioner. This is because in each step we compute a matrix $M_i \approx A_i^{-1}$. The memory cost of M_i will be counted into the whole memory cost of the preconditioner, as well as the construction cost. This will obviously increase both computational cost and memory cost for MultiSails with a large number of steps. In Table 4 we notice that in the 5th step case, the preconditioner converges in 342 iterations, but the total computational cost is 5 times as much as that in the 1st step case. The reduction in the solution time does not compensate for the increase in the setup time. So unless it does not converge with a lower number of steps or in the case of solving one matrix with many right hand sides, usually we do not recommend too many steps in real applications, even though that may yield better convergence results. In Table 4, we think that 1 or 2 steps is a good compromise between reasonable computational cost and good convergence results.

step	thre	flt	s-ratio	iter	setup	solve	total
1	0.001	0.001	4.58	426	10.0	0.8	10.8
	0.005	0.005	3.37	465	5.2	1.8	7.0
	0.01	0.01	2.71	484	4.5	0.7	5.2
	0.05	0.05	1.17	730	0.6	0.7	1.3
	0.07	0.07	0.85	936	0.5	0.9	1.4
2	0.001	0.001	12.70	99	110.6	0.7	111.3
	0.005	0.005	7.80	145	34.4	0.8	35.2
	0.01	0.01	5.80	169	15.9	0.8	16.7
	0.05	0.05	1.67	482	1.4	0.8	2.2
	0.07	0.07	1.14	646	0.8	0.9	1.7
	0.09	0.09	0.85	889	0.6	1.0	1.5

Table 5: Comparison of MultiSails using different preprocessing and postprocessing parameters to solve the FIDAPM10 matrix.

Table 5 lists experimental results using MultiSails to solve the FIDAPM10 matrix. We let the step number to be 1 and 2, then change the values of “thre” and “flt” parameters in each case to see the relationship between sparsity ratio and iteration number. To be convenient, the values of “thre” and “flt” are always chosen to be equal.

From Table 5 we can see that in each step, the increasing of “thre” and “flt” values leads to a smaller sparsity ratio due to the fact that more entries are dropped during the preprocessing and postprocessing phases. These sparsification strategies may degrade the convergence rate of the preconditioner to some extent and may increase the iteration number. However, with different steps, a smaller sparsity ratio does not necessarily mean a worse convergence property. For instances, in 1 step case, MultiSails with a sparsity ratio of 2.71 converges in 484 iterations; in 2 step case, MultiSails with a sparsity ratio of 1.67 converges in 482 iterations.

5.4 Comparison of ParaSails and MultiSails

In Table 6, we give some comparison results between MultiSails and ParaSails for solving a few sparse matrices.

We see that when constructing a preconditioner, MultiSails usually spends less time than ParaSails to reach the same amount of sparsity ratio. According to our discussions in previous sections, the preconditioner computed from MultiSails is composed of a number of sparse matrices M_i . The number of these sparse matrices is equal to the step number plus 1. Also the whole memory cost of the multistep preconditioner is equal to the sum of the memory cost of these sparse matrices. The memory cost of each sparse matrix is usually small and each of the sparse approximate inverse matrices can be computed very cheaply. So the whole computational cost of these sparse matrices is also small, compared with computing a single sparse matrix with comparable density in the case of ParaSails.

Also the data in Table 6 show that with the same amount of memory cost (sparsity ratio), MultiSails usually has better convergence performance than ParaSails does. For

Matrices	Preconditioner	thre	flt	s-ratio	iter	setup	solve	total
RAEFSKY1	PS(A)	0.01	0.01	0.24	545	5.1	4.3	9.4
	PS(A^2)	0.02	0.02	0.38	148	210.5	4.1	214.6
	MS(1)	0.05	0.05	0.16	207	1.2	1.5	2.8
	MS(2)	0.02	0.02	0.44	50	10.2	0.2	10.5
RAEFSKY2	PS(A)	0.01	0.01	0.38	786	6.0	7.5	13.5
	PS(A^2)	0.02	0.01	1.02	196	263.5	3.3	266.8
	MS(1)	0.05	0.02	0.32	535	2.9	3.3	6.2
	MS(2)	0.02	0.01	0.93	169	31.9	1.1	33.0
FIDAP024	PS(A^2)	0.0	0.0	4.86	-	9.8	-	-
	PS(A^3)	0.01	0.01	6.93	285	52.7	3.3	55.9
	MS(1)	0.001	0.002	4.47	799	12.2	4.4	16.6
	MS(2)	0.01	0.01	4.87	188	14.4	1.8	16.3
FIDAP028	PS(A^2)	0.0	0.0	4.29	789	19.3	6.7	25.9
	PS(A^2)	0.001	0.001	4.16	835	19.5	7.8	27.3
	MS(1)	0.004	0.004	2.97	255	13.4	2.9	16.2
	MS(1)	0.005	0.005	2.75	330	10.7	3.3	14.0
FIDAPM08	PS(A^2)	0.0	0.0	5.21	-	37.8	-	-
	PS(A^3)	0.0	0.0	12.91	-	377.0	-	-
	MS(2)	0.01	0.01	3.28	729	48.3	3.4	51.7
	MS(3)	0.01	0.01	5.12	291	142.2	2.2	144.5

Table 6: Comparison of ParaSails and MultiSails for solving a few sparse matrices.

solving the FIDAPM08 matrix, ParaSails does not converge when using either A^2 or A^3 as its sparsity pattern. However, MultiSails with a 2 step construction converges with a sparsity ratio 3.28.

5.5 Implementation scalability

According to Algorithm 3.1, the main computational costs in MultiSails are matrix-matrix product and matrix-vector product operations. As it is well known [23], these operations can be performed in parallel efficiently on most distributed memory parallel architectures.

The implementation scalability is tested using a three dimensional convection-diffusion problem (11) with the 7-point standard central difference scheme and the 19-point fourth order compact difference discretization scheme [32]. For the 7-point discretization case we let the matrix dimension to be 100^3 . The nonzero number is 6940000. The matrix is solved by using a 1 step MultiSails. For the 19-point matrix we choose the matrix dimension to be 80^3 . The nonzero number is 9536960. It is solved by using a 2 step MultiSails. Tables 7 and 8 show the computational results with different numbers of processors. We can see that the MultiSails preconditioner scales very well in these two test cases. In particular, we point out that the number of iterations remains to be the same in each case, when the number of processors increases from 4 to 32. This is different from the simple domain decomposition preconditioners whose iteration properties are usually affected by the number of processors (domains) involved [9, 31].

We remark that for the “solve” time listed in the 7th column of Tables 7 and 8,

np	thre	flt	s-ratio	iter	setup	solve	total
4	0.05	0.05	1.74	288	1953.4	232.8	2186.1
8	0.05	0.05	1.74	288	984.1	121.0	1105.0
16	0.05	0.05	1.74	288	501.9	44.4	546.3
24	0.05	0.05	1.74	288	361.7	29.4	391.1
32	0.05	0.05	1.74	288	281.8	24.7	306.5

Table 7: Scalability of MultiSails for solving a 7-point matrix with $n = 100^3$.

np	thre	flt	s-ratio	iter	setup	solve	total
4	0.05	0.05	0.74	171	1049.9	69.4	1119.3
8	0.05	0.05	0.74	171	529.1	31.0	560.1
16	0.05	0.05	0.74	171	269.9	15.0	285.0
24	0.05	0.05	0.74	171	184.3	12.0	196.3
32	0.05	0.05	0.74	171	146.6	8.3	154.8

Table 8: Scalability of MultiSails for solving a 19-point matrix with $n = 80^3$.

the timing results show some superlinear speedup effect for up to 16 processors. This is largely because of the well known cache effect commonly seen in such large scale parallel computer systems.

6 Concluding Remarks

We have proposed a class of multistep successive sparse approximate inverse preconditioning strategies for solving general sparse matrices. A prototype implementation named MultiSails is tested to show favorable convergence properties and computational efficiency of this class of new preconditioning strategies.

Our numerical experiments with several sparse matrices show that as the step number increases, the matrix A_i we compute in each step tends to become more strongly diagonally dominant. Solving these strongly diagonally dominant matrices is relatively easier, compared to solving the original matrix. Usually, multistep preconditioners with a small step number lead to good convergence results, even with a small memory cost. Multistep preconditioners with a large step number may be more robust, but they may also lead to both higher computational cost and higher memory cost. It is our experience that the step number should be chosen as 1 or 2, provided that the preconditioned solver can converge well.

When compared with ParaSails, MultiSails is cheaper in its setup phase. Generally speaking, to compute a series of small memory cost (sparse) matrices may be cheaper than to compute a high memory cost (sparse) matrix. However, a series of small memory cost (sparse) matrices together may hold more information about the inverse of the original matrix, it can make the preconditioned solver more robust and converge faster.

The scalability of MultiSails is also tested in our numerical experiments. The Multi-

Sails preconditioner seems to scale well for the tested two and three dimensional convection-diffusion problems.

In conclusion, the performance of the MultiSails preconditioner is indeed as good as what we expected. But some detailed work still needs to be done in future work, in order to build a software package that may be used in realistic large scale scientific computation and computer simulations. E.g., “thre” and “filt” are two important parameters in this class of algorithms (both MultiSails and ParaSails). However, in our current implementation, we do not consider different situations in different construction steps and always keep them the same. It is possible that we may be able to choose these parameters adaptively in a more sophisticated implementation.

Finally, we remark that the concepts of multistep successive preconditioning can be applied to other preconditioning techniques. It is also possible to construct multistep successive ILU preconditioners, or to construct multistep hybrid successive preconditioners using several different preconditioning techniques in each step.

References

- [1] O. Axelsson and P. S. Vassilevski. Variable-step multilevel preconditioning methods. I. selfadjoint and positive definite elliptic problems. *Numer. Linear Algebra Appl.*, 1(1):75–101, 1994.
- [2] R. E. Bank and C. Wagner. Multilevel ILU decomposition. *Numer. Math.*, 82(4):543–576, 1999.
- [3] S. T. Barnard, L. M. Bernardo, and H. D. Simon. An MPI implementation of the SPAI preconditioner on the T3E. *Int. J. High Performance Comput. Appl.*, 13:107–128, 1999.
- [4] T. Barth, T. F. Chan, and W.-P. Tang. A parallel non-overlapping domain-decomposition algorithm for compressible fluid flow problems on triangulated domains. In *Domain Decomposition Methods, 10*, Contemp. Math., 218, pages 23–41, Providence, RI, 1998. Amer. Math. Soc.
- [5] M. W. Benson and P. O. Frederickson. Iterative solution of large sparse linear systems arising in certain multidimensional approximation problems. *Utilitas Math.*, 22:127–140, 1982.
- [6] M. W. Benson, J. Krettmann, and M. Wright. Parallel algorithms for the solution of certain large sparse linear systems. *Int. J. Comput. Math.*, 16:245–260, 1984.
- [7] M. Benzi and M. Tuma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM J. Sci. Comput.*, 19(3):968–994, 1998.
- [8] M. Benzi and M. Tuma. A comparative study of sparse approximate inverse preconditioners. *Appl. Numer. Math.*, 30(2-3):305–340, 1999.

- [9] X.-C. Cai, W. D. Gropp, and D. E. Keyes. A comparison of some domain decomposition and ILU preconditioned iterative methods for nonsymmetric elliptic problems. *Numer. Linear Algebra Appl.*, 1(5):477–504, 1994.
- [10] T. F. Chan, S. Go, and J. Zou. Multilevel domain decomposition and multigrid methods for unstructured meshes: algorithms and theory. Technical Report CAM 95-24, Department of Mathematics, UCLA, Los Angeles, CA, 1995.
- [11] E. Chow. Parallel implementation and performance characteristics of least squares sparse approximate inverse preconditioners. Technical Report UCRL-JC-138883, Lawrence Livermore National Laboratory, Livermore, CA, 2000.
- [12] E. Chow. ParaSails Users' Guide. Technical Report UCRL-JC-137863, Lawrence Livermore National Laboratory, Livermore, CA, 2000.
- [13] E. Chow. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM J. Sci. Comput.*, 21(5):1804–1822, 2000.
- [14] E. Chow and Y. Saad. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM J. Sci. Comput.*, 19(3):995–1023, 1998.
- [15] J. D. F. Cosgrove, J. C. Diaz, and A. Griewank. Approximate inverse preconditionings for sparse linear systems. *Int. J. Comput. Math.*, 44:91–110, 1992.
- [16] A. C. N. van Duin. Scalable parallel preconditioning with the sparse approximate inverse of triangular matrices. *SIAM J. Matrix Anal. Appl.*, 20:987–1006, 1999.
- [17] M. Engelman. FIDAP: Examples Manual, Revision 6.0. Technical report, Fluid Dynamics International, Evanston, IL, 1991.
- [18] M. Grote and T. Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM J. Sci. Comput.*, 18:838–853, 1997.
- [19] M. Grote and H. D. Simon. Parallel preconditioning and approximate inverse on the Connection machines. In R. F. Sincovec, D. E. Keyes, M. R. Leuze, L. R. Petzold, and D. A. Reed, editors, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 519–523, Philadelphia, PA, 1993. SIAM.
- [20] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J. Sci. Comput.*, 22(6):2194–2215, 2001.
- [21] D. E. Keyes and W. D. Gropp. A comparison of domain decomposition techniques for elliptic partial differential equations and their parallel implementation. *SIAM J. Sci. Statist. Comput.*, 8(2):S166–S202, 1987.
- [22] L. Y. Kolotilina. Explicit preconditioning of systems of linear algebraic equations with dense matrices. *J. Soviet Math.*, 43:2566–2573, 1988.
- [23] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*. Benjamin/Cummings Pub. Co., Redwood City, CA, 1994.

- [24] M. Magolou monga Made and H. A. van der Vorst. Parallel incomplete factorization with pseudo-overlapped subdomains. *Parallel Comput.*, 27(8):989–1008, 2001.
- [25] P. Raghavan, K. Teranishi, and E. Ng. Towards scalable preconditioning using incomplete Cholesky factorization. In *Proceedings of the 2001 Conference on Preconditioning Techniques for Large Scale Matrix Problems in Industrial Applications*, pages 63–65, Tahoe City, CA, 2001.
- [26] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing, New York, NY, 1996.
- [27] Y. Saad and M. Sosonkina. Distributed Schur complement techniques for general sparse linear systems. *SIAM J. Sci. Comput.*, 21(4):1337–1356, 1999.
- [28] Y. Saad and J. Zhang. BILUM: block versions of multielimination and multilevel ILU preconditioner for general sparse linear systems. *SIAM J. Sci. Comput.*, 20(6):2103–2121, 1999.
- [29] Y. Saad and J. Zhang. BILUTM: a domain-based multilevel block ILUT preconditioner for general sparse matrices. *SIAM J. Matrix Anal. Appl.*, 21(1):279–299, 1999.
- [30] C. Shen and J. Zhang. Parallel two level block ILU preconditioning techniques for solving general sparse linear systems. Technical Report No. 310-00, Department of Computer Science, University of Kentucky, Lexington, KY, 2000.
- [31] B. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, New York, NY, 1996.
- [32] J. Zhang. An explicit fourth-order compact finite difference scheme for three dimensional convection-diffusion equation. *Commun. Numer. Methods Engrg.*, 14:209–218, 1998.
- [33] J. Zhang. A parallelizable preconditioner based on a factored sparse approximate inverse technique. In Y. Saad, D. Pierce, and W.-P. Tang, editors, *Proceedings of the 1999 International Conference on Preconditioning Techniques for Large Sparse Matrix Problems in Industrial Applications*, pages 193–199, Minneapolis, MN, 1999. University of Minnesota.
- [34] J. Zhang. A sparse approximate inverse technique for parallel preconditioning of general sparse matrices. *Appl. Math. Comput.*, 2001. in press.