The Nonsingularity of Sparse Approximate Inverse Preconditioning and Its Performance Based on Processor Virtualization *

Kai Wang[†] Orion Lawlor [‡] and Laxmikant V. Kale [§]

Parallel Programming Lab
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, 61801, USA

Abstract

In this paper, we analyze the properties of the sparse approximate inverse preconditioner, and prove that for a strictly diagonally dominant M matrix, the computed preconditioning matrix can be guaranteed to be nonsingular if it is nonnegative. Then we investigate the use of the processor virtualization technique to parallelize the sparse approximate inverse solver. Numerical experiments on a distributed memory parallel computer show that the efficiency of the resulting preconditioner can be improved by virtualization.

Key words: Processor virtualization, Nonsingularity, MPI, AMPI, Sparse Approximate Inverse, Preconditioning.

1 Introduction

The need to solve very large sparse linear systems arises from many important applications, and has been driving the development of sparse linear system solvers for parallel computers. Direct solvers, based on sparse matrix factorization, are extremely robust, but their memory and floating point operation requirements grow faster than a linear function of the order of the matrix, because original zeros fill in during the factorization. Preconditioned Krylov subspace methods, by contrast, are considered to be some of the most suitable candidates for solving large sparse linear systems [1, 34].

*This work was supported in part by the National Science Foundation (ITR 0205611, DMR 0121695)
[†]E-mail: wangkai@cs.uiuc.edu. URL: http://charm.cs.uiuc.edu/∼wangkai.
[‡]E-mail:olawlor@uiuc.edu. URL: http://charm.cs.uiuc.edu/users/olawlor.
[§]E-mail: kale@cs.uiuc.edu. URL: http://charm.cs.uiuc.edu.

Simple parallel preconditioners such as Jacobi or block Jacobi methods, although easy to implement, have the inherent weakness of being not robust for difficult problems. Their lack of robustness prevents them from being used in industrial-strength, standard software packages. Other parallel preconditioners based on the multicoloring strategy also have restricted applicability, as only limited parallelism can be extracted by this strategy. Domain decomposition based methods have been exploited extensively in parallel linear system solvers and preconditioners [5, 10, 28, 39]. Important progress has been made recently concerning the parallelization of incomplete LU (Cholesky) factorization preconditioners [21, 30, 33]. Furthermore, there are two additional 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 a detailed discussion of several sequential and parallel multilevel ILU preconditioning techniques, we refer readers to [2, 3, 35, 36, 37, 38].

In this paper, we will examine another class of parallelizable preconditioning techniques which compute a sparse approximate inverse (SAI) of the original matrix. These preconditioners possess a high degree of parallelism in the preconditioner application phase, and are shown to be efficient for certain type of problems. Many algorithms have been proposed to construct SAI preconditioners [8, 15, 17, 18, 19, 40, 41, 43, 46]. A typical one is to compute the preconditioner matrix M by minimizing the Frobenius norm [15, 19]. This method is inherently parallel and can be implemented on distributed memory computer systems. However, unlike an ILU type preconditioner, it is difficult to prove that a SAI preconditioner is nonsingular.

SAI preconditioners have been successfully parallelized in practice. Two software packages were developed independently based on two different sparsity pattern generation algorithms. ParaSails is based on a static sparsity pattern computation and is developed by Chow [13, 14]. SPAI_3.0 is based on a dynamic sparsity pattern computation and was developed by Barnard *et al.* [4]. They are written using the usual parallel message passing interface standard (MPI). The performance of these methods is studied and compared in [42].

The MPI standard [31] is currently the most popular programming model for parallel application development. In MPI, to run on a parellel machine with P processors, the programmer normally divides the computation into exactly P processes, which run on the P processors. For complex dynamic applications, significant programmer effort can be required to divide the computation with good load balance and communication performance.

The processor virtualization concept has been proposed as a way to remedy these difficulties [22, 23, 24, 26, 27]. The processor virtualization method has the programmer decompose the computation according to the nature of the problem, instead of the number of physical processors available. The programmer thus divides the problem into a large number of objects, which are called virtual processors, and the runtime system is responsibile for mapping these virtual processors to different physical processors. This empowers the runtime system to do resource management, including automatic load balancing and communication optimization, by migrating the virtual processors across physical processors. It simplifies the programmer's task by substantially removing the constraint of physical processors from the algorithm design process [27].

The processor virtualization technique has been successfully employed and evaluated in many dynamic applications which are notoriously difficult to parallelize [25, 32, 47]. But its performance on other important parallel applications, including parallel preconditioning, remains unknown. Hence the latter part of this study is an investigation into the effect of processor virtualization on a parallel SAI solver.

The paper is organized as follows. In Section 2, we introduce the basic technique and properties of the SAI preconditioner. We then analyze the nonsingularity of the computed SAI matrix in Section 3, treating separately the special case when the original matrix is an M matrix. In Section 4, we explain the idea of processor virtualization and its application to a SAI solver. Numerical results are presented in Section 5, which show the benefits of processor virtualization for a SAI solver. Section 6 contains some brief concluding remarks.

2 Sparse approximate inverse preconditioners

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 solver applied directly to (1) may be slow if the matrix A is ill-conditioned. In order to speed up the convergence rate of such iterative methods, we transform (1) into an equivalent system

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

where M, the preconditioner, is any nonsingular matrix of order n. A Krylov subspace solver applied to the transformed system will converge faster than the original system if the condition number of MA is better than that of A. In particular, if M is a good approximation to A^{-1} in some sense, then MA should be a good approximation to the identity matrix I. A Krylov solver applied to the identity matrix converges in one step.

A sparse approximate inverse is simply a sparse matrix M which is a good approximation to A^{-1}. The major driving force behind the search for efficient sparse approximate inverse preconditioners is their potential advantages in parallel computing. The idea is that once computed, a sparse preconditioner matrix M can be applied via a simple matrix-vector product, which can be implemented efficiently on a parallel computer [29]. The ease and efficiency of this parallel operation compares favorably with the highly sequential nature of the triangular solution procedures used by incomplete LU factorization preconditioning techniques.

There exist several techniques to construct sparse approximate inverse preconditioners. They can be roughly categorized into three classes [9]: sparse approximate inverses based on Frobenius norm minimization [15, 19, 41], sparse approximate inverses computed from an ILU factorization [16], and factored sparse approximate inverses [8, 45, 46]. Each of these classes contains a variety of different constructions and each of them has its own merits and drawbacks. 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 [6, 7].

We discuss here a particular class of sparse approximate inverse preconditioners that are constructed based on a minimization of the Frobenius norm. 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 the left preconditioner. It is also possible to approximate A^{-1} from the right, so that $AM \approx I$, which is termed as the right preconditioner. In the case of the right preconditioning, the equivalent preconditioned system analogous to (2) is

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

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 the obvious solution $M = A^{-1}$. This solution is undesirable for at least two reasons. First, inverting a matrix is much slower than performing a linear solve. Second, for most sparse matrices A, their inverses A^{-1} are mostly dense, which will cause serious memory problems for the large matrices encountered in many practical applications.

Thus we are interested in a constrained minimization such that M has a certain sparsity pattern, or nonzero structure—that is, only certain entries of M are allowed to be nonzero. Given a sparsity pattern Ω (which could be fixed or depend on the original matrix), we minimize the functional

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

Although any norm could be used in the above definition, a particularly convenient norm is the Frobenius norm, defined for a matrix $A = (a_{ij})_{n\times n}$ as $\|A\|_F = \sqrt{\sum_{i,j=1}^{n} a_{ij}^2}$ [34]. With the Frobenius norm, the minimization problem (5) can be decoupled into n independent subproblems and can proceed as (using square for convenience)

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

where m_k and e_k are the kth columns of M and I, respectively. It follows that the minimization problem (5) is equivalent to minimizing the individual functions

$$\|Am_k - e_k\|_2, \qquad k = 1, 2, \ldots, n \tag{7}$$

with certain restrictions placed on the sparsity pattern of m_k. In other words, each column of M can be computed independently.

If the sparsity pattern of m_k allows, say, n_2 nonzero entries, the rest of the entries are forced to be zero. Denote the n_2 nonzero entries of m_k by $\tilde{m}_k$ and the n_2 corresponding columns of A by A_k. Since A is sparse, its submatrix A_k has many rows that are identically zero. If we remove the zero rows, we have a reduced matrix $\tilde{A}_k$ with n_1 rows and n_2 columns. The individual minimization problem (7) is thus reduced to a least squares problem of order $n_1 \times n_2$

$$\min_{\tilde{m}_k} \|\tilde{A}_k \tilde{m}_k - \tilde{e}_k\|_2, \qquad k = 1, 2, \ldots, n. \tag{8}$$

We note that the matrix $\tilde{A}_k$ is usually a very small rectangular matrix. It has full rank if A is nonsingular. $\tilde{m}_k$ can be computed by QR factorization or the normal equations [19] for each column k independently. These solves yield an approximate inverse matrix M, which minimizes $\|AM - I\|_F$ for the given sparsity pattern.

The parallelism inherent in the technique is the computation of the columns $\tilde{m}_k$ independently of each other. It can be implemented efficiently on any modern parallel machine [29].

3 Nonsingularity of the sparse approximate inverse matrix

If our preconditioner is a singular matrix, a solution of the transformed system (2) may not correspond to a solution of the original system (1). Hence the preconditioning matrix must be nonsingular.

Unfortunately, unlike the ILU type preconditioners, whose nonsingularity can be guaranteed by forcing the diagonal elements of the triangular matrices (L and U) to be nonzero, it is difficult to prove that an SAI preconditioning matrix is nonsingular. In most numerical experiments, we find out that the computed approximate inverse matrices turn out to be nonsingular. However, there have been no practically useful methods to determine if a computed SAI matrix is nonsingular or not. Existing theorems require the computed SAI matrices to satisfy certain strict conditions to be nonsingular [34]. Meeting these conditions usually requires the sparsity pattern to include an impractical number of nonzeros.

In this section, we discuss the nonsingularity of the SAI matrix. We assume the original matrix A is a nonsingular matrix, and we assume the diagonal is included in the selected sparsity pattern. First we will give some definitions and concepts which will be used in the following discussion.

3.1 Definitions

- **Strictly column diagonally dominant matrix:**

A matrix A is called strictly column diagonally dominant if

$$| a_{ii} | > \sum_{j=1, j\neq i}^{n} | a_{ji} |$$

A matrix is strictly row diagonally dominant if its transpose is strictly column diagonally dominant. It is well know that any strictly column or row diagonally dominant matrix is nonsingular.

- **Nonnegative and nonpositive matrix:**

  A matrix A is a nonnegative matrix if for each element a_{ij} in the matrix, we have $a_{ij} \geq 0$. We can write it as $A \geq 0$. A matrix A is a nonpositive matrix if $-A$ is a nonnegative matrix.

3.2 Sparse approximate inverse for general matrices

From the introduction in Section 2, we know that the approximate inverse process is to compute n independent small minimization problems according to a selected sparsity pattern. The solution of each minimization problem is a vector $\tilde{m}_k$ with length n_k, where $n_k \leq n$, and is equal to the number of nonzeros in the sparsity pattern. n is the dimension of the original matrix A.

To be convenient for discussion, we permute $\tilde{A}_k$ to A_k, so that A_k can be written in the block form

$$\begin{pmatrix} B_k \\ E_k \end{pmatrix}. \tag{9}$$

Here we define B_k by dropping both the rows and columns of A corresponding to the zeros in our sparsity pattern for the k'th column, then permuted so the kth diagonal element of A is the first diagonal element of B_k. Hence B_k is a primary submatrix of A of rank n_k. E_k is the rectangular matrix, which is formed by the remaining off-diagonal rows of $\tilde{A}_k$.

So the minimization problems (8) can be rewritten as

$$\min \|A_k m_k - e_1\|_2, \quad k = 1, \ldots, n \tag{10}$$

Here, m_k, e_1 is the corresponding permutation of $\tilde{m}_k$ and $\tilde{e}_k$. The vector m_k is computed by solving the normal equation

$$\begin{array}{rl} & A_k^T A_k m_k = A_k^T e_1 \\ \Rightarrow & (B_k^T B_k + E_k^T E_k) m_k = A_k^T e_1 \\ \Rightarrow & (B_k^T B_k + E_k^T E_k) m_k = B_k^T e_1. \end{array} \tag{11}$$

It is difficult to evaluate the nonsingularity of the SAI matrix M directly, since the columns of M are computed independently, and there is no direct relationship among them. We turn to analyzing the properties of AM, the product of the original matrix A and SAI matrix M; because if AM is nonsingular, then M must be nonsingular.

First the diagonal property of the AM matrix is studied.

Theorem 3.1. *Let $AM = (d_{ij})_{i,j=1\ldots n}$. AM is a matrix with nonnegative diagonal elements, and each diagonal element d_{kk} of AM can be expressed as*

$$d_{kk} = \sum_{j=1}^{n} d_{jk}^2.$$

Proof: From the computation of SAI matrix, the nonzero elements of AM are a permutation of

$$A_k m_k, \quad k = 1, \ldots, n.$$

Here m_k stands for the solution of (11).

In our discussion, we have permuted the kth diagonal element of A to be the first diagonal element of A_k. So the kth diagonal element of AM is

$$d_{kk} = e_1^T A_k m_k = m_k^T A_k^T e_1.$$

which when we express A_k as in (9) is

$$d_{kk} = m_k^T B_k^T e_1.$$

From (11), we know that

$$A_k^T A_k m_k = B_k^T e_1.$$

Multiplying this relation on the left by m_k^T gives

$$d_{kk} = m_k^T B_k^T e_1 = (A_k m_k)^T A_k m_k = \sum_{j=1}^{n} d_{jk}^2.$$

□

Since we are trying to compute M as a sparse inverse of A, AM should approximate the identity matrix I. In the next theorem, we show the diagonal property of $AM - I$.

Theorem 3.2. *Let $AM - I = (c_{ij})_{i,j=1\ldots n}$. $AM - I$ is a matrix with nonpositive diagonal elements, and each diagonal element c_{kk} of $AM - I$ satisfies*

$$c_{kk} = -\sum_{j=1}^{n} c_{jk}^2.$$

Proof: The proof is straightforward, considering $c_{jk} = d_{jk}$ when $j \neq k$, where d_{jk} is the element of AM, we have

$$\begin{array}{rl} & c_{kk} = d_{kk} - 1 \\ \Rightarrow & c_{kk} = -(d_{kk} - 1)^2 - d_{kk} + d_{kk}^2 \\ \Rightarrow & c_{kk} = -c_{kk}^2 - \sum_{j=1}^{n} d_{jk}^2 + d_{kk}^2 \\ \Rightarrow & c_{kk} = -c_{kk}^2 - \sum_{j=1, j\neq k}^{n} d_{jk}^2 \\ \Rightarrow & c_{kk} = -\sum_{j=1}^{n} c_{jk}^2. \end{array}$$

□

The next theorem is related to the Frobenius norm of $AM - I$.

Theorem 3.3. *The Frobenius norm $\|AM - I\|_F$ is the square root of the sum of the absolute values of the diagonal elements of $AM - I$.*

Proof: This can be proved by

$$\begin{array}{ll} & \|AM - I\|_F^2 \\ = & \sum_{k=1}^n \|A_k m_k - e_1\|_2 \\ = & \sum_{k=1}^n \sum_{j=1}^n c_{jk}^2. \end{array}$$

From Theorem 3.2, we get

$$\|AM - I\|_F^2 = -\sum_{k=1}^n c_{kk}.$$

Here c_{kk} is the diagonal element of $AM - I$.

□

Finally, we give a theorem about the nonsingularity of the computed SAI matrix M for a general matrix A.

Theorem 3.4. *If $\sum_{k=1}^n |c_{kk}| < 1$, then M is nonsingular.*

Proof: Theorem 3.3 shows the Frobenius norm of $AM - I$ can be written as

$$\|AM - I\|_F^2 = \sum_{k=1}^n |c_{kk}|.$$

So $\sum_{k=1}^n |c_{kk}| < 1$ implies

$$\|AM - I\|_F =< 1.$$

It is well known that when

$$\|AM - I\|_F = \|I - AM\|_F < 1,$$

then $I - (I - AM) = AM$ is a nonsingular matrix [34], so M is nonsingular.

□

Thus we see that, to verify if an approximate matrix is nonsingular or not, we only need to check the diagonal elements of AM, instead of doing a high cost matrix multiplication.

3.3 Sparse approximate inverse for M matrices

Many sparse linear systems arising from scientific and engineering applications lead to a special kind of matrix, the M matrix. It is thus useful investigate the nonsingularity of the approximate inverse preconditioners for M matrices.

A square matrix A is called an M matrix if $A = \lambda I - G$ with $G \geq 0$ and $\lambda \geq \rho(G)$, where $\rho(G)$ is the spectral radius of the matrix G. If a matrix A is an M matrix, then [11]:

- The diagonal elements of A are nonnegative (≥ 0), and the off-diagonal elements of A are nonpositive (≤ 0).
- The inverse of A is nonnegative.
- All the primary submatrices of A are M matrices.
- There exists a positive diagonal matrix D which makes AD a strictly column diagonally dominant matrix.

The remainder of this section discusses M matrices.

Lemma 3.5. *Let A be an M matrix. If the least squares solution m_k of (10) is nonnegative, then $B_k m_k - e_1$ and $E_k m_k$ are nonpositive.*

Proof: When A is a nonsingular M matrix, B_k is a nonsingular M matrix because it is a primary submatrix of A; and the matrix E_k is a nonpositive matrix as it is formed by the off-diagonal elements of A.

$E_k m_k \leq 0$ is clear because $E_k \leq 0$ and $m_k \geq 0$. Next we prove $B_k m_k - e_1 \leq 0$.

From (11), we get

$$\begin{aligned} & B_k^T B_k m_k + E_k^T E_k m_k - B_k^T e_1 = 0 \\ \Rightarrow \quad & B_k m_k - e_1 = -B_k^{-T} E_k^T E_k m_k. \end{aligned}$$

B_k is a nonsingular M matrix, so B_k^{-1} is a nonnegative matrix. Therefore,

$$-B_k^{-T} E_k^T E_k m_k$$

is a nonpositive vector, so $B_k m_k - e_1$ is nonpositive.

□

Each least square solution of (10) will compute a column of approximate matrix M. Lemma 3.5 implies that when the column of M is ≥ 0, the corresponding column of $AM - I$ will be ≤ 0. So we get the next Lemma

Lemma 3.6. *If the SAI matrix M of an M matrix A is nonnegative, then AM is a matrix with nonnegative diagonal elements and nonpositive off-diagonal elements, and $AM - I$ is a nonpositive matrix.*

From the definition we know that an M matrix can be transformed to a strictly column diagonally dominant M matrix by multiplying by a diagonal matrix D. Next we give a theorem when the original matrix A is a diagonally dominant M matrix.

Theorem 3.7. *Suppose A is a column diagonally dominant M matrix. If the computed sparse approximate inverse preconditioner M is a nonnegative matrix, then AM is a diagonally dominant matrix. Especially, if A is strictly column diagonally dominant, then M is nonsingular, and AM is also an M matrix.*

Proof: Let d_k be one column in AM. According to Lemma 3.6, when M is a nonnegative matrix,

$$d_{kk} \geq 0,$$

and

$$d_{ik} \leq 0, \quad i \neq k.$$

Since $d_{ik} = \sum_{j=1}^{n} a_{ij} m_{jk}$,

$$\begin{array}{rl} & \sum_{i=1}^{n} d_{ik} \\ = & \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} m_j \\ = & \sum_{j=1}^{n} \sum_{i=1}^{n} a_{ij} m_j \\ = & \sum_{j=1}^{n} m_j \sum_{i=1}^{n} a_{ij}. \end{array}$$

A is a diagonally dominant M matrix, so we have

$$\sum_{i=1}^{n} a_{ij} \geq 0.$$

Therefore, we get

$$\sum_{i=1}^{n} d_{ik} = \sum_{j=1}^{n} m_j \sum_{i=1}^{n} a_{ij} \geq 0.$$

That means AM is a diagonally dominant matrix. Obviously, when A is strictly diagonally dominant, AM is also a strictly diagonally dominant M matrix, which implies M is nonsingular.

□

From Theorem 3.7, we can see that for a strictly column diagonally dominant M matrix, the nonsingularity of its approximate inverse matrix can be guaranteed by computing a nonnegative approximate inverse matrix.

4 Processor virtualization in sparse approximate inverse

The goal of processor virtualization is to find an effective division of labor between the programmer and runtime system. Specifically, the human programmer is best at finding and expressing the natural parallelism of the application, but the runtime system can efficiently carry out resource management and many performance optimizations [26, 27]. In the processor virtualization model, the programmer divides the computation into many virtual processors, and the runtime system assigns them to available physical processors. The management and inspection of the virtual processors are also controlled by the runtime system instead of the programmer.

Probably the most obvious advantage of processor virtualization is that the runtime system can do automatic dynamic load balancing, by moving the virtual processors between physical processors. Suppose each physical processor houses many virtual processors. In the simplest setting, the runtime system can monitor the loads on all physical processors and its neighbors. When a physical processor goes idle, the run time system could request additional virtual processors from neighboring physical processors with high load, so that the loads are balanced. A powerful runtime system can make this possible without user supervision.

Processor virtualization has been applied in many different areas. Many of these applications benefit from the automatic load balancing mechanism [25, 32, 47]. However, for SAI preconditioning, load balancing is usually not a serious issue. Since the computation and communication pattern is a fixed matrix vector product, the load can be roughly balanced by assigning the same number of matrix rows to each processor. So in this study, we do not focus on the automatic load balancing advantage of virtualization, but instead on its abilities to improve cache performance and optimize communication.

Better cache performance The parallelization scheme for a SAI solve on virtual processors is the same as for physical processors—the matrix data is distributed to the virtual processors row by row. When the number of virtual processors is larger than the number of physical processors, each virtual processor handles a smaller set of data than each physical processor. A virtual processor may thus have better memory locality during both communication and computation. This blocking effect is the same strategy employed by many sequential cache optimization techniques.

Adaptive overlap of computation and communication For the SAI preconditioning technique, the communication is mainly from its matrix vector product in the solving phase. For example, when we use the GMRES algorithm as the preconditioned Krylov subspace solver, each GMRES iteration involves two matrix vector product operations. As one processor only stores part of the vector, the other parts of the vector need to be acquired from corresponding processors by message communication. Therefore, the time spent on an iteration for one processor does not only depend on itself but also the slowest processor which has part of the vector.

Typical parallel programming models such as MPI support only one process per physical processor. Therefore, if this single process is blocked on a receive, the whole physical processor blocks and is idle. This communication idle time can be traced to two distinct causes. First, a processor B may have to wait for processor A to complete its work, for example because A is a slower processor or more heavily loaded (load imbalance). Second, even after A sends the data, processor B still must wait for the data to arrive across the network (message delay). This is illustrated in Fig. 1, where processor B has finished its first phase computation, but it cannot go to its second phase computation without the message from processor A. So processor B remains idle until the message from A arrives. Both load imbalance and message delay prevent us from taking full advantage of machine's power.

Allowing each physical processor to contain many virtual processors can decrease the amount of time wasted. When one virtual processor is blocked, the runtime system can keep the CPU working by picking up another virtual processor to take the control of the CPU. This behavior is illustrated in Fig. 2, where the physical processor A and B now

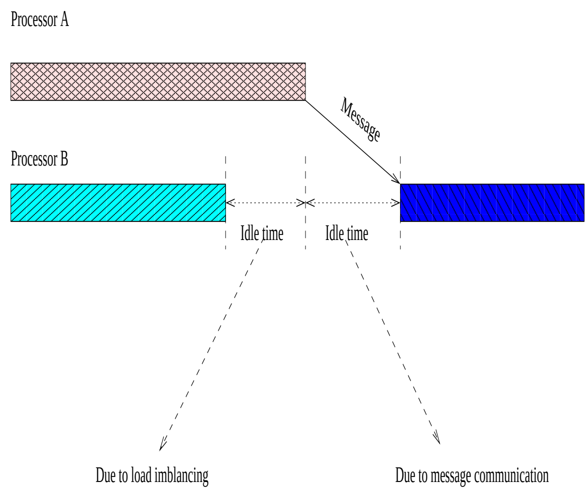

Figure 1: Processor idle time caused by load imbalance and message delay.

contain two virtual processors $A1$, $A2$, $B1$, and $B2$. When the first phase computation of $B1$ and $B2$ are finished, the message from $A1$ arrives, hence $B1$ can start its next phase computation immediately. Compared with Fig. 1, the idle time in Fig. 2 is reduced because the computation and communication in Fig. 2 are overlapped.

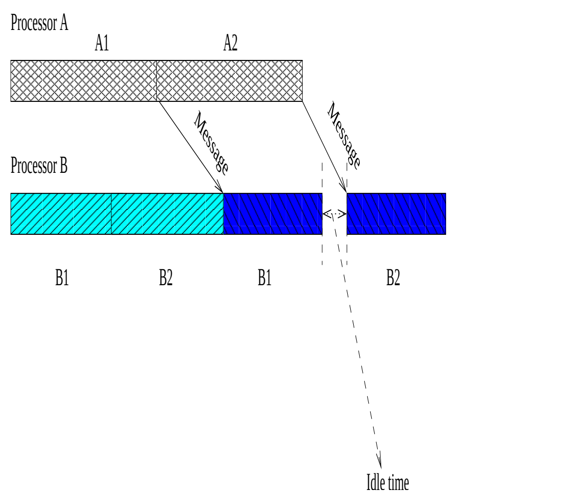

Figure 2: Processor idle time reduction via processor virtualization.

Now the question is: can the communication and computation be overlapped in SAI preconditioning? In other words, can the matrix-vector operations take advantage of the adaptive overlapping mechanism of virtualization? Obviously, when the matrix is a dense matrix, the matrix vector operation tends to require an all to all communication. In this case, the communication and computation can hardly be overlapped, for example, in Fig. 2, if the virtual processor $B1$ requires both messages from $A1$ and $A2$ to start its second phase computation.

Fortunately, SAI preconditioning is for sparse matrices. In each communication step, a virtual processor requires the part of the vector corresponding to the nonzero pattern of the local submatrix. Thus communication is only required between a small set of nearby virtual processors. An extreme example is when the number of virtual processors is equal to the dimension of the matrix. Each virtual processor then has only one row of the matrix, so the number of communications per virtual processor will be less than or equal to the number of nonzero elements in the stored row, which is usually much smaller than the dimension of the matrix (otherwise, it is not a sparse matrix.) This sparse communication pattern makes the adaptive overlap of communication and computation possible.

4.1 Implementation issues

Currently there are at least two parallel programming systems supporting the processor virtualization technique. Charm++ is one of the earliest. It is C++ based and uses parallel objects called Chares to express each virtual processor. Chares communicate via asynchronous invocation of each other's special "remote" methods. For some types of applications, Charm++ has better performance and modularity properties than MPI[24, 26, 27]. However the asynchronous message-passing mechanism is unfamiliar to many programmers, especially in Science and Engineering or using languages other than C++.

To provide processor virtualization within the popular MPI programming interface, Adaptive MPI (AMPI) was developed [20]. AMPI is built on Charm++, but provides the familiar programming model of MPI. In this study, we use AMPI to implement an SAI solver. Details of the Charm++ and AMPI programming systems can be obtained from their website [12], or publications [20, 22, 23, 24].

The processor virtualization concept can be applied to any existing SAI solver. Here we use an SAI solver based on a static sparsity pattern [13, 14] to show the advantages of virtualization. This algorithm uses the sparsified patterns of powers 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, $A^3, \ldots,$ may be used. Here the matrices $A^2, A^3, \ldots,$ are not explicitly computed, only their sparsity patterns are extracted from that of the matrix A with binary operations. A software package, ParaSails, which uses MPI to parallelize the static sparsity pattern SAI preconditioning, has been released to the public [13, 14]. We make ParaSails virtualization capable by replacing the MPI commands in the software to corresponding AMPI commands and compiling it with the AMPI compiler. The following is the SAI algorithm based on the static sparsity pattern [13].

ALGORITHM **4.1.** Construct a static pattern SAI preconditioner.

1. Given a drop tolerance τ and the level of pattern l
2. Drop entries of A that are smaller than τ to get A'
3. Compute an SAI matrix M according to the sparsity pattern of A'^l
4. Drop entries of M that are smaller than τ
5. M is the preconditioner for $Ax = b$

5 Experimental results

In this section, we show the parallel performance of an SAI solver based on the processor virtualization technique introduced in the previous section. The equations to be solved are the discretized convection diffusion equations, which are very important in computational fluid dynamics to model transport phenomena.

3-D Convection-diffusion problem. A three dimensional steady-state convection-diffusion problem is defined on a unit cube as

$$u_{xx} + u_{yy} + u_{zz} + 1000\,(p(x,y,z)\,u_x + q(x,y,z)\,u_y + r(x,y,z)\,u_z) = 0. \qquad (12)$$

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 for this problem is 1000. Eq. (12) is discretized by using the standard 19-point fourth order compact difference scheme [44]. A typical row or column thus has 19 nonzero entries.

The solver We can see from Algorithm 4.1 that there are two parameters in the algorithm, τ and l, which have an important influence on the convergence performance of the resulting system. In a real world application, they should be tuned carefully. However, in this study, we only focus on the parallel performance of the SAI solver. In all our reported numerical results, the drop tolerance parameter τ is fixed to be zero, and the level of pattern is set to be 1.

We run the tests on the Tungsten Xeon machine in NCSA using up to 16 nodes. Each node has 3GB memory and dual Intel Xeon 3.06 processors. The processor is installed with a 512 KB L2 cache and 1MB L3 cache.

For all the results reported in the tables and figures. “degree” means the number of virtual processors assigned to each physical processor. “MPI” denotes the time used for native MPI program solving the same problem. “setup” means the time spent constructing the SAI preconditioner; “solve” is the time spent on the preconditioned GMRES(50) iteration; “total” is the sum of these two values.

5.1 Virtualization overhead

We first compare the performance of the SAI solver when using different number of virtual processors on only one physical processor. The purpose of the test is to study the overhead of virtualization. Here we point out that, when using one virtual processor in one physical processor, the program is actually doing a serial computation.

Degree	setup	solve	all
1	4.1	20.4	24.5
2	4.1	20.7	24.8
3	4.2	21.6	25.8
4	4.3	22.2	26.5
5	4.3	22.4	26.7
6	4.4	22.7	27.0
7	4.4	23.1	27.5
8	4.5	23.6	28.0
MPI	4.1	20.4	24.5

Table 1: Raw data of Fig. 3.

Fig. 3 shows the result for solving the three dimensional convection-diffusion equation with $40,000$ unknowns and $722,206$ nonzeros. Its raw data are listed in Table 1. The time reported here is after 1000 GMRES iterations.

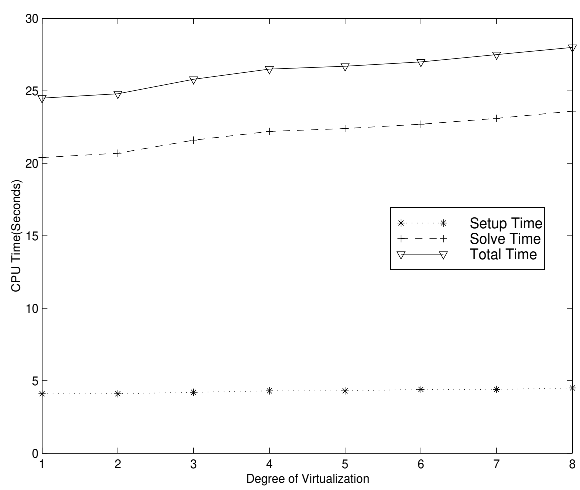

Figure 3: Virtualization overhead on one physical processor case. Number of Unknowns $= 40,000$. 1000 iterations. 1 physical processor.

From Fig. 3 and Table 1 we can see that when using more virtual processors, the virtualization overhead makes both the setup time and solving time increase slightly. The total CPU time of using 8 virtual processors is 3.5 seconds (14 percent) slower than without using virtualization.

5.2 Cache performance

The test in this paragraph is to demonstrate the cache performance effect of virtualization. The results shown in Fig. 4 and Table 2 are from solving a larger problem size of $160,000$

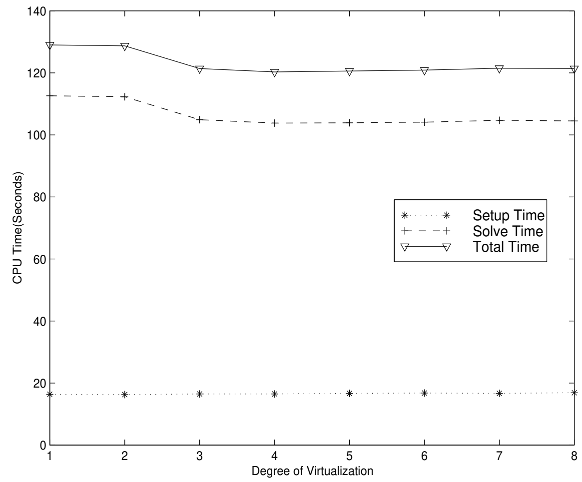

Figure 4: Cache performance on one physical processor case. Number of Unknowns = $160,000$. 1000 iterations. 1 physical processor.

Degree	setup	solve	all
1	16.4	112.6	129.0
2	16.3	112.3	128.7
3	16.5	104.9	121.4
4	16.5	103.8	120.3
5	16.7	103.9	120.6
6	16.8	104.1	120.9
7	16.7	104.7	121.5
8	16.9	104.5	121.4
MPI	16.3	112.4	128.7

Table 2: Raw data of Fig. 4.

unknowns with $2,951,656$ nonzeros on one physical processor.

We can see from Fig. 4 and Table 2 that the setup time increases with more virtual processors due to the virtualization overhead. However, the solve time drops from 112.6 seconds to 103.8 seconds because of the improved cache performance when the degree of virtualization is increased from 1 to 4. More virtual processors make each virtual processor handle less data, which fits better in cache. Here we can see the cache performance outperforms the influence of the virtualization overhead and improves the total CPU time by 6 percent.

The setup time does not show much cache benefit because the main computation during the setup phase is the small least square solves, which are implemented by the highly cache optimized BLAS/LAPACK routines in the SAI solver. In addition, each least square solve is only to compute one column of the SAI matrix, so the memory usage

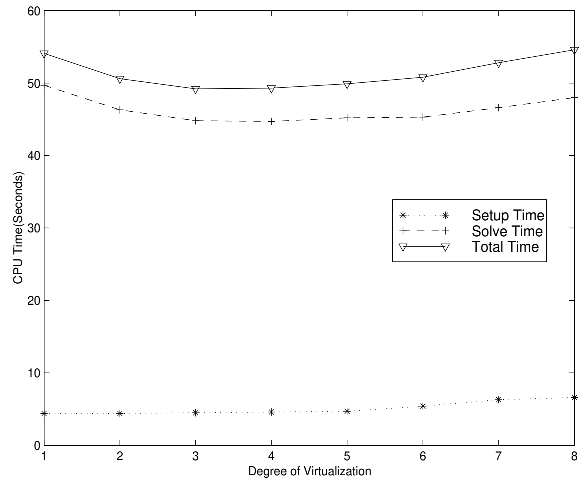

Figure 5: Performance of adaptive overlapping. Number of Unknowns = 640000. 1000 iterations. 16 physical processors. Upper: Relationship between the degree of virtualization and total time. Lower left: Relationship between the degree of virtualization and setup time. Lower right: Relationship between the degree of virtualization and solving time.

is related to the number of nonzeros in the sparsity pattern, which is bounded by 19 in our numerical experiments and not influenced by the problem size.

5.3 Adaptive overlapping

The adaptive overlap of communication and computation happens when one virtual processor blocks for a receive, and the runtime system switches to another virtual processor. Theoretically, adaptive overlap can save CPU time since a better overall processor utilization is expected. To show its performance, we first should eliminate the influence of the cache effect. From the experiments in the previous subsection, we see that when a physical processor is assigned a data set size of 40,000, the whole CPU time only increases as we add virtual processors. It implies that there is no cache benefit, or the cache benefit cannot compensate for the virtualization overhead.

So we run our next test on 16 physical processors, and assign 40,000 unknowns to each physical processor, which means a total problem size of 640,000. The results are shown in Fig. 5 and Table 3.

From Fig. 5 we can see that when use 4 virtual processors per physical processor, the solving time is decreased by 5 seconds, which is 10 precent faster than the non-virtualizatized case. Since each physical processor contains only a 40,000 unknowns, the speedup here can be regarded as purely from the adaptive overlap of communication and computation.

Degree	setup	solve	all
1	4.4	49.7	54.1
2	4.4	46.3	50.6
3	4.5	44.8	49.2
4	4.6	44.7	49.3
5	4.7	45.2	49.9
6	5.4	45.3	50.8
7	6.3	46.6	52.8
8	6.6	48.0	54.6
MPI	4.4	49.8	54.2

Table 3: Raw data of Fig. 5.

Degree	setup	solve	all
1	17.8	252.8	270.6
2	18.0	249.1	267.1
3	18.4	233.0	251.4
4	18.8	221.7	240.5
5	20.1	216.0	236.1
6	21.4	216.7	238.1
7	21.6	218.8	240.5
8	22.3	219.8	242.1
MPI	18.0	252.6	270.6

Table 4: Raw data of Fig. 5.

5.4 Interleaved performance

With a larger problem size, we show the performance of the SAI solver with both improved cache performance and adaptive overlapping of communication and computation. The results in Fig. 6 and Table 4 are from solving a three dimensional problem size with $5,120,000$ unknowns.

The data in Fig. 6 and Table 4 illustrate that when the degree of virtualization is 5, the solving time decreases from 252.8 to 216.0 seconds, which is 14.5 percent improvement compared with the non-virtualized case. The total CPU time can be saved 34.5 seconds. The speedup here are both from the improved cache performance and adaptive overlapping of communication and computation.

6 Conclusion

In this paper, new stronger guarantees for the nonsingularity of the SAI preconditioning matrix were presented. For a general sparse matrix, the nonsingularity of its SAI matrix M can be checked by only computing the diagonal elements of $AM - I$. For a strictly

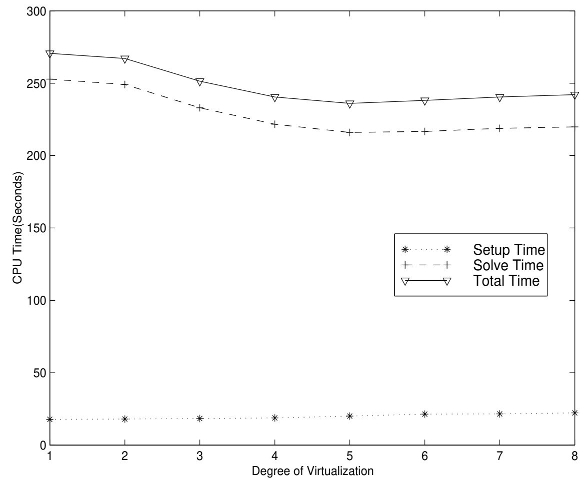

Figure 6: Interleaved performance. Number of Unknowns = $5,120,000$. 1000 iterations. 32 physical processors.

diagonally dominant M matrix, the nonsingularity of the resulting preconditioned system can be guaranteed by computing a nonnegative SAI matrix.

We also demonstrated that the processor virtualization technique can be used to parallelize the SAI preconditioning process. The numerical results demonstrate that the parallel performance of the SAI preconditioning can be improved by using a proper number of virtual processors in a physical processor. Two reasons account for this speedup: first is improved cache performance, because each virtual processor handles less data than a physical processor; second, virtualization allows many processes in each processor, which decreases the probability of the processor idling when one process blocks for a receive.

In our tests, no benefit was shown during the setup phase, because the standard SAI computation does not involve much communication and the memory usage for each least squares computation is fixed. But for more complex computation, like the multistep successive preconditioning (MSP) [41], whose setup phase consists of many steps, and each step does matrix products, the performance of its setup phase can expected to be improved by virtualization. Our future work will include a study of the performance of these more complicated solvers.

References

[1] O. Axelsson. *Iterative Solution Methods*. Cambridge Univ. Press, Cambridge, 1994.

[2] 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.

[3] R. E. Bank and C. Wagner. Multilevel ILU decomposition. *Numer. Math.*, 82(4):543–576, 1999.

[4] S. T. Barnard, L. M. Bernardo, and H. D. Simon. An MPI implementation of the SPAI preconditioner on the T3E. *Int. J. High Perfor. Comput. Appl.*, 13:107–128, 1999.

[5] 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.

[6] 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.

[7] 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.

[8] M. Benzi and M. Tuma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM J. Sci. Comput.*, 19(3):968–994, 1998.

[9] M. Benzi and M. Tuma. A comparative study of sparse approximate inverse preconditioners. *Appl. Numer. Math.*, 30(2-3):305–340, 1999.

[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] A. Berman, R. J. Plemmons. *Nonnegative Matrices In The Mathematical Sciences.* Academic Press, New York, NY, 1979.

[12] Homepage of Parallel Programming Lab at Department of Computer Science, University of Illinois at Urbana-Champaign: http://charm.cs.uiuc.edu.

[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. Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns. *Int. J. High Perf. Comput. Appl.*, 15:56–74, 2001.

[15] E. Chow and Y. Saad. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM J. Sci. Comput.*, 19(3):995–1023, 1998.

[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] N. I. M. Gould and J. A. Scott. Sparse approximate-inverse preconditioners using norm-minimization techniques. *SIAM J. Sci. Comput.*, 19(2):605–625, 1998.

[18] G. A. Gravvanis. An approximate inverse matrix technique for arrowhead matrices. *Int. J. Computer Math.*, 70:35–45, 1998.

[19] M. Grote and T. Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM J. Sci. Comput.*, 18:838–853, 1997.

[20] C. Huang and O .Lawlor and L. Kale. Adaptive MPI Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03), College Station, Texas, October, 2003.

[21] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J. Sci. Comput*, 22(6):2194–2215, 2001.

[22] L. Kale, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM parallel programming language and system: Part I – Description of language features, *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[23] L. Kale, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM parallel programming language and system: Part II – The runtime system, *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[24] L. Kale, S. Krishnan. Charm++: Parallel programming with message-driven objects, In *Parallel Programming Using C++*, G. V. Wilson and P. Lu, editors, MIT Press, 1996, 175-213.

[25] L. Kale, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz. J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten, NAMD2: Greater scalability for parallel molecular dynamics, *J. of Comp. Phy.*, 151:283-312, 1999.

[26] L. Kale. The virtualization model of parallel programming: runtime optimizations and the state of art, In *LACSI 2002*, Albuquerque, October 2002.

[27] L. Kale. Performance and Productivity in Parallel Programming via Processor Virtualization, Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10), Madrid, Spain, February, 2004.

[28] 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.

[29] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* Benjamin/Cummings Pub. Co., Redwood City, CA, 1994.

[30] M. Magolu monga Made and H. A. van der Vorst. Parallel incomplete factorization with pseudo-overlapped subdomains. *Parallel Comput.*, 27(8):989–1008, 2001.

[31] MPI website: http://www-unix.mcs.anl.gov/mpi/.

[32] M. Nelson, W. Humphrey, A. Gursoy, A .Dalke, L. Kale, R. Skeel, and K. Schulten. NAMD - a parallel, object-oriented molecular dynamics program, *International Journal Supercomputing Applications and High Performance Computing*, Winter 1996, Volume 10, number 4.

[33] 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.

[34] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing, New York, NY, 1996.

[35] Y. Saad and M. Sosonkina. Distributed Schur complement techniques for general sparse linear systems. *SIAM J. Sci. Comput.*, 21(4):1337–1356, 1999.

[36] 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.

[37] 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.

[38] C. Shen and J. Zhang. Parallel two level block ILU preconditioning techniques for solving large sparse linear systems. *Paral. Comput.*, 28(10):1451–1475, 2002.

[39] B. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, New York, NY, 1996.

[40] K. Wang and J. Zhang. Multigrid treatment and robustness enhancement for factored sparse approximate inverse preconditioning. *Appl. Num. Math.*, 43(4):483-500, 2002.

[41] K. Wang and J. Zhang. MSP: a class of parallel multistep successive sparse approximate inverse preconditioning strategies. *SIAM J. Sci. Comput.*, 24(4):1141-1156, 2003.

[42] K. Wang, S.B. Kim, and J. Zhang. A Comparative Study on Dynamic and Static Sparsity Patterns in Parallel Sparse Approximate Inverse Preconditioning. *Journal of Math. Modelling and Algorithms.*, 2(3):203-215,2003.

[43] K. Wang, J. Zhang, and C. Shen. Parallel Multilevel Sparse Approximate Inverse Preconditioners in Large Sparse Matrix Computations. To Appear in *proceedings of Supercomputing 2003: Igniting Innovation.*, November 15 - 21, 2003, Phoenix, Arizona, USA

[44] J. Zhang, An explicit fourth-order compact finite difference scheme for Three Dimensional Convection-diffusion Equation, *Commun. Numer. Methods Engrg.*, 14:209-218, 1998.

[45] 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.

[46] J. Zhang. A sparse approximate inverse technique for parallel preconditioning of general sparse matrices. *Appl. Math. Comput.*, 130(1):63–85, 2002.

[47] G. Zheng, G. Kakulapati, L. Kale, BigSim: A parallel simulator for performance prediction of extremely large parallel machines, In IPDPS, April, Santa Fe, New Mexico, 2004.