J. Appl. Math. & Computing Vol. 11 *(2003), No. 1-2, pp. 59-80*

A MULTILEVEL BLOCK INCOMPLETE CHOLESKY PRECONDITIONER FOR SOLVING NORMAL EQUATIONS IN LINEAR LEAST SQUARES PROBLEMS

JUN ZHANG AND TONG XIAO

ABSTRACT. An incomplete factorization method for preconditioning symmetric positive definite matrices is introduced to solve normal equations. The normal equations are form to solve linear least squares problems. The procedure is based on a block incomplete Cholesky factorization and a multilevel recursive strategy with an approximate Schur complement matrix formed implicitly. A diagonal perturbation strategy is implemented to enhance factorization robustness. The factors obtained are used as a preconditioner for the conjugate gradient method. Numerical experiments are used to show the robustness and efficiency of this preconditioning technique, and to compare it with two other preconditioners.

AMS subject classifications : 65F10, 65F20, 65F25, 65F50.
Key words and phrases : incomplete Cholesky factorization, multilevel IC preconditioner, normal equation, conjugate gradient.

1. Introduction

We discuss preconditioning techniques for solving linear least squares problems with rectangular matrices. A linear least squares problem is to find $x \in \Re^n$ which minimizes the value of

$$\|b - Ax\|_2, \tag{1}$$

where $A \in \Re^{m\times n}$, $m > n$ is a large sparse rectangular matrix and $b \in \Re^m$ is an arbitrary known vector. Linear least squares problems may be encountered in many scientific and engineering applications such as linear programming, geodetic survey problems, augmented Lagrange methods for computational fluid dynamics, and the natural factor method in structural engineering

Received June 20, 2002. Revised October 8, 2002.
This research was supported in part by the U.S. National Science Foundation under grants CCR-9902022, CCR-9988165, and CCR-0092532.

analysis [3, 6, 7, 11, 15, 20]. Because of the great increase in the capacity of automatic data capturing, least squares problems of large size are now routinely solved in these and other applications [4].

There are a few methods to solve (1). Both iterative and direct methods are discussed in [4]. For large sparse matrices, iterative methods seem to be more attractive. One iterative strategy is LSQR algorithm of Paige and Saunders [19], which is based on the Lanczos bidiagonalization procedure. Due to the space limit, we will not discuss this approach here, but refer the reader to [19, 4] for details.

Another iterative approach is to use conjugate gradient (CG) method to solve the corresponding normal equation

$$A^T A x = A^T b, \tag{2}$$

where the dimension of $A^T A$ is n. This approach, called CGLS, is well known because $A^T A$ is symmetric positive definite (SPD) when A is a real matrix with full rank [4, 23]. In theory the conjugate gradient (CG) method is convergent for any SPD coefficient matrix and has a rate of convergence which is generally superior to other classical iterative methods [10, 12]. It is known that the convergence rate of CG is related to the condition number of the coefficient matrix [23]. We note that (2) is typically used to solve the least squares problem (1) for overdetermined systems, i.e., when A is a rectangular matrix of size $m \times n$, and $m > n$. The drawback of this approach is that the condition number of the normal equation is the square of the condition number of A, thus CG may take many iterations to converge. A good preconditioner is needed to speedup the CG convergence. In fact, some recent research interest in iterative solution of least squares problems has been towards identifying effective preconditioners for different applications [4].

There have been several preconditioning techniques proposed for the CG methods with SPD systems. These include point and block Jacobi preconditioners, SSOR incomplete Cholesky (IC) factorization, polynomial preconditioning, and incomplete orthogonal factorization [2, 4, 29]. There has also been recent interest in developing more robust techniques for preconditioning least squares problems, such as the CIMGS preconditioning [29] and preconditioning based on LU factorization [5] and the associated block SOR methods [8].

In this article, we first discuss two existing strategies to precondition rectangular matrices based on the normal equation approach. We then develop a new multilevel block incomplete Cholesky (BICM) preconditioner that reduces the computational cost substantially. The BICM preconditioner is based on the ideas from the BILUM and BILUTM preconditioners for general sparse matrices [24, 25] and on some diagonal stabilization strategy [17]. Block independent set concept and the Schur complement construction strategies are also reviewed.

The BICM preconditioning technique is motivated for solving the normal equations, but it is a general preconditioning technique for solving sparse SPD linear systems.

We organize our article as follows. Section 2 discusses two existing preconditioning techniques, incomplete Cholesky (IC) and compressed incomplete modified Gram-Schmidt (CIMGS) factorizations, for normal equations. We introduce background on block independent set in Section 3. The multilevel block IC preconditioner is outlined in Section 4. Some numerical results and interpretation of these results are included in Section 5. Section 6 contains the summary remarks and some ideas on future studies.

2. IC and CIMGS Factorizations

Throughout this article, we assume that A is a rectangular matrix with full column rank. There are several incomplete factorization methods for solving (2) [13]. In this section, we briefly review two incomplete factorization type preconditioners for an SPD matrix.

2.1 Incomplete Cholesky factorization

A well known approach to preconditioning SPD matrices is incomplete Cholesky factorization (IC) [18]. Let $B = A^TA$, where B is SPD. By an incomplete Cholesky factorization we mean to establish a relation of the form $B = LL^T + R$, where L is a lower triangular matrix with positive diagonal elements but $R \neq 0$. Usually the factor L is much less sparse than B because of fill-in. A common approach to reducing density is to suppress the fill-in, or part of it, that occurs during the factorization. In order to do this, we may apply a certain drop set W to the factor L, which means that the sparsity of the factor L can be determined by a drop set. A drop set W determines which elements of the target incomplete factor will not be retained in the factorization. One way to select a drop set is to let the factor L have the same sparsity pattern as that of the lower triangular part of B. In this case, the drop set is selected as those positions corresponding to the zero elements in B.

Another way is to determine the drop set W dynamically, which means that drop set is selected as the factorization proceeds. We may use a drop tolerance to select retained elements. Given a threshold tolerance $\tilde{\epsilon}$, the drop tolerance ϵ for the ith row is determined as $\epsilon = \left(\sum_{j\in nz(i)} |b_{ij}| / nnz(i)\right)\tilde{\epsilon}$, where $nz(i) = \{j \ |b_{ij} \neq 0\}$ is the nonzero pattern of the ith row of B, $nnz(i)$ is the cardinality of the set $nz(i)$.

When the magnitude of a computed element is smaller than ϵ, this element is dropped, or we say that this position is selected into W. The diagonal element is not dropped even if its magnitude is smaller than ϵ. The algorithm of dynamic IC factorization is as follows:

Algorithm 1. Algorithm for dynamic incomplete Cholesky factorization.

```
1.  $l_{11} = b_{11}^{1/2}$
2.  for $i = 2$ to $n$, do
3.      for $j = 1$ to $i - 1$, do
4.          $l_{ij} = \left(b_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk}\right) / l_{jj}$
5.          if $|l_{ij}| < \epsilon$ then $l_{ij} = 0$
6.      end do
7.      $l_{ii} = \left(b_{ii} - \sum_{k=1}^{i-1} l_{ik}^2\right)^{1/2}$
8.  end do
```

This algorithm is modified from the IC factorization procedure in [9]. The difficulty with the IC preconditioner is that an incomplete Cholesky decomposition may not necessarily be carried out. It may break down for SPD matrices because the square root of a negative number is required to compute a diagonal element as in line 7 of Algorithm 1. Some improved IC factorization algorithms to deal with breakdown have been discussed in [1, 14, 17].

2.2 Compressed incomplete modified Gram-Schmidt

Another preconditioner for SPD systems is the compressed incomplete modified Gram-Schmidt (CIMGS) factorization [21, 29].

When a preconditioner is applied to A, a natural target is to make the preconditioned matrix $\tilde{A}$ close to orthogonal because then $\tilde{A}^T\tilde{A} \approx I$. This suggests using an incomplete orthogonalization factorization. CIMGS is based on an incomplete modified Gram-Schmidt (IMGS) factorization, both are discussed in detail in [28, 29].

Incomplete Gram-Schmidt methods give a factorization $A = QR$ with Q not necessarily orthogonal, and R is an upper triangular matrix [13, 21, 28]. In general, this factorization will always succeed in producing a nonsingular upper triangular factor R when A has full rank. IMGS is robust and is potentially effective at reducing the number of CG iterations [21, 29]. Its main weakness is the high cost of computing the preconditioner.

The algorithm CIMGS produces in exact arithmetic the same preconditioner as IMGS does, while greatly reducing the computational cost. The basic idea is to compress the information in the column vectors of A into a product form without losing the information needed for the computation of the factor R. When performing complete MGS (or CMGS) on A, one obtains $A^TA = R^TR$ in exact arithmetic.

If we use the same drop tolerance ϵ as we use in the dynamic IC factorization to select retained elements, we may describe the CIMGS algorithm in the following way [29]:

Algorithm 2. Algorithm for dynamic CIMGS.

```
1.  for $i = 1, 2, \cdots, n$, do
2.      for $k = 1, 2, \cdots, i-1$, do
3.          for $j = i, i+1, \cdots, n$, do
4.              if $|b_{ki}| > \epsilon$ or $|b_{kj}| > \epsilon$, then
5.                  $b_{ij} = b_{ij} - b_{ki}b_{kj}$
6.              end if
7.          end do
8.      end do
9.      if $b_{ii} > 0$, then
10.         $r_{ii} = b_{ii} = \sqrt{b_{ii}}$
11.         for $j = i+1, \cdots, n$, do
12.             $b_{ij} = b_{ij}/b_{ii}$
13.             $r_{ij} = \begin{cases} 0, & |b_{ij}| < \epsilon \\ b_{ij}, & otherwise \end{cases}$
14.         end do
15.     else
16.         quit (break down)
17.     end if
18. end do
```

Because B is symmetric, we only work on the upper triangular part of B to get the factor R. Also there is no need to form B explicitly, all we need is to access one row of B at a time. The disadvantage of CIMGS is that it may require more intermediate storage than IC with the same sparsity pattern because the factorization may store elements in a row that are in the drop set but are needed for the modifications required for later rows. It is remarked by Wang et al. [29] that this temporary storage cost is bounded by the storage cost of a corresponding complete Cholesky factorization [29]. It seems that CIMGS performs a factorization on a more complete matrix than IC does, which lends its robustness, as the (complete) Cholesky factorization on an SPD matrix is (theoretically) guaranteed to succeed.

3. Block Independent Set

The concept of block independent set is proposed by Saad and Zhang [24, 25] for developing robust and parallelizable multilevel block ILU preconditioners for solving general sparse matrices. Consider a collection of nonempty subsets $V_j = \{v_{j_1}, v_{j_2}, \ldots, v_{j_k}\} \neq \emptyset$ of the vertex set V which are mutually exclusive, i.e., $V_j \cap V_i = \emptyset$, if $j \neq i$. A quotient graph is a graph whose vertices are the subsets $V_j, j = 1, \dots, l$. It is generalized by coalescing all the nodes in each subset V_i into a supervertex and defining an edge from any supervertex V_i to another supervertex V_j if there is an edge from a vertex in V_i to a vertex in V_j, i.e., $V_i \to V_j$ if $\exists k_i \in V_i$ and $\exists k_j \in V_j$, such that $a_{k_i,k_j} \neq 0$. A block independent set is simply an independent set defined on this quotient graph [24].

Definition 1. *Let $V_1, V_2, \dots, V_l$ be a collection of mutually exclusive nonempty subsets of V. The set $S = \{V_1, V_2, \dots, V_l\}$ is said to be a block independent set if any two distinct subsets V_j and V_k in S are not adjacent in the quotient graph.*

It is not necessary that $V_1, V_2, \dots, V_l$ have the same cardinality (size). However, we will deal with subsets of constant cardinality k in this article. Various heuristic strategies may be used to find a block independent set with different properties [24]. A simple and usually efficient strategy is a greedy algorithm, which groups the nearest nodes together. This algorithm to find an independent set of size k is described below.

A block of size k will be found by coupling a given node j with $(k-1)$ of its nearest neighbors. In the following algorithm, adj(j) denotes the set of all neighboring nodes of node j (excluding j). The algorithm will mark every node that is visited. We denote by adj$_*(j)$ the subset of adj(j) consisting of unmarked nodes. In the algorithm, we use V_0 to represent a generic vertex set of size k.

Algorithm 3. Greedy algorithm for blocking elements.

```
1.  set $m = 0,\ V_0 = \emptyset$
2.  for $j = 1, 2, \dots, n$, do
3.      if node $j$ is not marked, then
4.          let $m = m + 1,\ V_0 = V_0 \cup \{j\}$
5.          if $m < k$, then
6.              if $adj_*(j) \neq \emptyset$, then
7.                  choose a node $j_m$ from $adj_*(j)$
8.                  $V_0 = V_0 \cup \{j_m\}$, mark $j_m$
9.                  $m = m + 1$
10.             else
11.                 if $V_0 - \{j\} \neq \emptyset$, then
12.                     for $i = j_1, j_2, \cdots, j_m$, do
13.                         if $adj_*(i) \neq \emptyset$ and $m < k$, then
14.                             choose a node $s$ from $adj_*(i)$
15.                             $V_0 = V_0 \cup \{s\}$, mark $s$
16.                             $m = m + 1$
17.                         end if
18.                     end do
19.                 end if
20.             end if
21.         end if
```

```
22.         mark $j$ and all vertices in $adj(j)$ and all neighboring nodes of $V_0$
23.     end if
24. end do
```

Note that upon termination all nodes will be marked. The set of all nodes in all V_i's will constitute the block independent set and the remaining nodes constitute the vertex cover. As the algorithm is described, each block of the block independent set will be of size k.

4. Block ICM Factorization

When looking for a block independent set of B, we do not explicitly form the elements of $B = A^TA$, which could require a potentially large number of floating point operations and a lot of memory. We only form the pattern of B, i.e., the row and column information in which each nonzero element is located. (The pattern of B can be computed cheaply.) Then we apply the Algorithm 3 to find a block independent set of B. After the block independent set is found and the corresponding permutation matrix P is obtained, we permute the original matrix A into $\bar{A}$ and perform a block IC factorization on $\bar{A}^T\bar{A}$. Below we show that we will get the same permuted matrix of B by applying the permutation matrix P on A instead of applying it on B directly.

Assume the dimension of A is $m \times n$ and $m > n$. Thus B is an $n \times n$ matrix and the permutation matrix P is also $n \times n$. Because A is a rectangular matrix, we generate the row permutation matrix Q as

$$Q = \begin{pmatrix} P & 0 \\ 0 & I_{m-n} \end{pmatrix},$$

where I_{m-n} is an identity matrix of dimension $(m-n) \times (m-n)$. We permute the matrix A into $\bar{A} = QAP^T$. Note that

$$\bar{A}^T\bar{A} = (QAP^T)^T(QAP^T) = PA^TQ^TQAP^T = PA^TAP^T = PBP^T.$$

Here we used the fact that a permutation matrix is orthonormal: $Q^TQ = I$.

The matrix B is permuted into a two by two block form where the left top block D is a block diagonal matrix $D = \text{diag}(\tilde{D}_1, \tilde{D}_2, \cdots, \tilde{D}_l)$, and each $\tilde{D}_i$ is a $k \times k$ submatrix. We may eliminate the unknowns of the independent set to obtain a reduced system, and the process may be repeated with the reduced system [24]. We construct a series of reduced systems of the form $B_{j+1} = C_j - E_jD_j^{-1}E_j^T$, with j being the level index. In exact arithmetic, these reductions are implicitly equivalent to a block Cholesky factorization

$$P_jB_jP_j^T = \begin{pmatrix} D_j & E_j^T \\ E_j & C_j \end{pmatrix} = \begin{pmatrix} L_j & 0 \\ E_jL_j^{-T} & I_j \end{pmatrix} \times \begin{pmatrix} L_j^T & L_j^{-1}E_j^T \\ 0 & B_{j+1} \end{pmatrix}, \tag{3}$$

where L_j is a lower triangular matrix which is obtained by a block Cholesky factorization on D_j with $D_j = L_jL_j^T$. B_{j+1} is the Schur complement with respect to C_j and $B_{j+1} = C_j - E_j(L_jL_j^T)^{-1}E_j^T$. I_j is the identity matrix on level j. P_j is the permutation matrix corresponding to the independent set ordering.

The solution process with the above factorization consists of level-by-level forward eliminations, followed by a solution on the last reduced system B_L. Here L is the number of reductions or levels. The solution of the original linear system is obtained by level-by-level backward substitutions (with suitable permutation).

In real factorization, we do not perform matrix multiplication to get the Schur complement matrix at each level. We can compute it through the Cholesky factorization. Assume that the first s equations are associated with the block independent set as in the middle of (3) with the subscript j suppressed, i.e., the dimension of D is $s \times s$. If we perform Cholesky factorization on the upper part (the first s rows) of the matrix, i.e., on the submatrix $(D,\ E^T)$, we transform the upper part $(D,\ E^T)$ into $(LL^T,\ L^{-1}E^T)$. We then continue the elimination to the lower part, but the elimination is only performed with respect to the submatrix E, i.e., we only eliminate those elements $a_{i,k}$ for which $s < i \leq n, 1 \leq k \leq s$. Corresponding linear combinations are also performed with respect to the C submatrix. This is illustrated in the following algorithm for computing block Cholesky factorization with the Schur complement.

Algorithm 4. Algorithm for computing block Cholesky factorization and Schur complement.

```
1.  $l_{11} = b_{11}^{1/2}$
2.  for $i = 2$ to $n$, do
3.      for $j = 1$ to $i-1$, do
4.          if $j \leq s$, then
5.              $l_{ij} = (b_{ij} - \sum_{k=1}^{j-1} l_{ik}l_{jk})/l_{jj}$
6.          else
7.              $l_{ij} = b_{ij} - \sum_{k=1}^{s} l_{ik}l_{jk}$
8.          end if
9.      end do
10.     $l_{ii} = \begin{cases} (b_{ii} - \sum_{k=1}^{i-1} l_{ik}^2)^{1/2}, & if\ i \leq s \\ b_{ii} - \sum_{k=1}^{s} l_{ik}^2, & otherwise \end{cases}$
11. end do
```

Since B is SPD, we may take the advantage of only forming and performing factorization on the lower triangular part of the matrix. The Algorithm 4 performs a block factorization of the form

$$\begin{pmatrix} D & E^T \\ E & C \end{pmatrix} = \begin{pmatrix} L & 0 \\ EL^{-T} & I \end{pmatrix} \times \begin{pmatrix} L^T & L^{-1}E^T \\ 0 & B_1 \end{pmatrix}. \tag{4}$$

Proposition 1. *The matrix B_1 computed by Algorithm 4 in (4) is the Schur complement of B with respect to C.*

Proof. For illustration purpose, we perform Cholesky factorization on the whole matrix. The part of the matrix after the upper part factorization (with respect to the block independent set) can be written as $(L^T,\ L^{-1}E^T)$. This upper part is used to eliminate the lower part. We may write the active part of the matrix B as

$$\begin{pmatrix} L^T & L^{-1}E^T \\ E & C \end{pmatrix}. \tag{5}$$

In order to eliminate E from the lower part of (5), we subtract the lower part $(E,\ C)$ by the upper part $(L^T,\ L^{-1}E^T)$ multiplied by EL^{-T}, i.e.,

$$EL^{-T}(L^T,\ L^{-1}E^T) = (E,\ EL^{-T}L^{-1}E^T).$$

The submatrix E is eliminated. At the same time, the submatrix C is modified as

$$B_1 = C - EL^{-T}L^{-1}E^T = C - E(LL^T)^{-1}E^T = C - ED^{-1}E^T.$$

The actual computation of the Cholesky factorization is equivalent, but is slightly different, with operations only performed on the lower triangular part of B and on L. □

Note that the reduced system B_1 obtained from Algorithm 4 includes only the lower triangular part and the diagonal. This algorithm gives an exact block Cholesky factorization with a Schur complement. In order to obtain an incomplete factorization, dropping strategies similar to those used in Algorithm 1 can be applied to Algorithm 4, resulting in a block IC factorization with an approximate Schur complement B_1. We describe one level of the BICM factorization in the following algorithm, given a certain drop tolerance ϵ and the size of the block independent set s.

Algorithm 5. Algorithm for block incomplete Cholesky factorization.

```
1.  $l_{11} = b_{11}^{1/2}$
2.  for $i = 2$ to $n$, do
3.      for $j = 1$ to $i-1$, do
4.          if $j \le s$, then
5.              $b_{ij} = (b_{ij} - \sum_{k=1}^{j-1} l_{ik}l_{jk})/l_{jj}$
6.          else
7.              $b_{ij} = b_{ij} - \sum_{k=1}^{s} l_{ik}l_{jk}$
8.          end if
9.          $l_{ij} = \begin{cases} b_{ij}, & if\ |b_{ij}| > \epsilon \\ 0, & otherwise \end{cases}$
10.     end do
11.     $b_{ii} = \begin{cases} b_{ii} - \sum_{k=1}^{i-1} l_{ik}^2, & if\ i \le s \\ b_{ii} - \sum_{k=1}^{s} l_{ik}^2, & otherwise \end{cases}$
12.     if $b_{ii} > 0$, then
```

```
13.         $l_{ii} = \begin{cases} \sqrt{b_{ii}}, & if\ i \le s \\ b_{ii}, & otherwise \end{cases}$
14.      else
15.         quit (break down)
16.      end if
17.  end do
```

A potential drawback of this incomplete factorization procedure is that it will break down if a negative diagonal element is encountered. This is due to the drop of elements during the factorization, which might make the resulting matrix no longer positive definite. In this case, the preconditioner would not necessarily exist [17]. It is known that the procedure of Cholesky factorization prevents the use of pivoting strategies commonly utilized in LU factorizations to deal with stabilization problems [27]. Alternative strategies that result in modifying the factorization matrix in one way or another to prevent factorization breakdown are proposed [1, 17]. One strategy to deal with breakdown is to modify the diagonal elements when we encounter a nonpositive diagonal element and perform the factorization again [17, 16]. This results in a factorization on the perturbed matrix $B + \sigma I$, where $\sigma > 0$. The diagonal shift algorithm that we use in our experiments for all preconditioners is described below [17, 16].

Algorithm 6. Algorithm for handling breakdown.

```
1.   choose a scalar $\sigma > 0$, and a maximum number of stabilizations $N$
2.   for $i = 0, 1, \cdots, N$, do
3.      perform incomplete factorization on $B$ if i=0, otherwise on $B + \sigma I$
4.      if breakdown occurs, then
5.         set $\sigma = 2\sigma$, perform factorization again
6.      else
7.         exit with successful factorization
8.      end if
9.   end do
10.  fail
```

We point out that in BICM, the restart is only performed on the current level of the factorization. The previously computed factors are not touched. For all preconditioning techniques tested in this paper, the diagonal shifting strategy is implemented during the computation of the preconditioners. When a diagonal entry of the matrix B is computed, a diagonal shift is added. We do not shift the diagonal entries of the original matrix A.

BICM preconditioning procedure. The preconditioning process based on the BICM factors includes a series of forward eliminations and backward substitutions. Suppose the right-hand side vector, denoted again as b for convenience, and the solution vector x are partitioned according to the block independent set ordering as in (3), we would have, on each level j

$$x_j = \begin{pmatrix} x_{j,1} \\ x_{j,2} \end{pmatrix}, \qquad b_j = \begin{pmatrix} b_{j,1} \\ b_{j,2} \end{pmatrix}.$$

Let l be the number of reduction levels. The forward elimination is performed by solving for a temporary vector y_j, i.e., for $j = 0, 1, \cdots, (l-1)$, by solving

$$\begin{pmatrix} L_j & 0 \\ E_j L_j^{-T} & I_j \end{pmatrix} \begin{pmatrix} y_{j,1} \\ y_{j,2} \end{pmatrix} = \begin{pmatrix} b_{j,1} \\ b_{j,2} \end{pmatrix},$$

with

$$y_{j,1} = L_j^{-1} b_{j,1}, \qquad y_{j,2} = b_{j,2} - E_j L_j^{-T} y_{j,1}.$$

Upon finishing this series of forward eliminations, we then get a group of intermediate vectors $(y_{j,1}, y_{j,2})^T$, $j = 0, 1, \cdots, (l-1)$. Here we have $y_{j,2} = (b_{j+1,1}, b_{j+1,2})^T$, i.e., part of the solution at level j is used as the right-hand side of level $(j+1)$st elimination.

We then solve the last reduced system as $L_l L_l^T x_l = y_l$. Here, y_l is actually $y_{l-1,2}$ from the previous level forward elimination. The approximate solution on the last level is obtained by performing a forward/backward substitution on $L_l L_l^T$. After that, a series of backward substitutions are performed to obtain the solution by solving

$$\begin{pmatrix} L_j^T & L_j^{-1} E_j^T \\ 0 & 0 \end{pmatrix} \begin{pmatrix} x_{j,1} \\ x_{j,2} \end{pmatrix} = \begin{pmatrix} y_{j,1} \\ y_{j,2} \end{pmatrix},$$

for $j = (l-1), \cdots, 1, 0$, with

$$x_{j,1} = y_{j,1} - L_j^{-1} E_j x_{j,2}, \qquad x_{j,1} = L_j^{-T} x_{j,1}.$$

Here, $x_{j,2}$ is the solution from the substitution at level $(j-1)$, and $x_{l-1,2} = x_l$. The approximate solution of the forward eliminations and backward substitutions will be $(x_{1,1}, x_{1,2})^T$.

5. Numerical Experiments

In the numerical experiments, we compare the robustness and efficiency of IC, CIMGS, and BICM as preconditioners for the conjugate gradient method. We also study some properties of BICM experimentally. We solve the least squares rectangular matrices using the regular conjugate gradient method on the implicit normal equations. All computations are done on one processor of an SGI Power Challenge with 8 processors and 512MB shared memory. The processor speed is 194 MHz. The codes are written in standard Fortran 77 programming language and the computation are done in double precision. Due to the nature of our test environment (parallel system overhead), small CPU timings, e.g., those less than 0.1 seconds, may not be reliable.

The test matrices include 21 matrices from the RRA sets of the Harwell-Boeing collection. All are rectangular matrices. Characteristics of the matrices include the number of rows (m), the number of columns (n), the number of nonzeros in A, and the number of nonzeros in A^TA ($= B$) are listed in Table 1. In our test matrices, some are underdetermined matrices. In order to make use of the advantages of normal equation, we use the transpose A^T of the original matrix as A if $n > m$. Thus the matrix A^TA has the dimension of $\min(m, n)$.

For each matrix we generate a right-hand side vector consistent with a solution vector whose components are all equal to one and the initial guess is a vector of some random numbers. We check the accuracy of the methods using the residual vector norm $\|A^Tb - Bx^k\|_2$, here k is the number of iterations. The computations are terminated when the actual residual vector norm $\|A^Tb - Bx^k\|_2 < 10^{-6}$. We also set an upper bound of 1000 for the PCG iterations. The restart parameter N in Algorithm 6 is set to be 50 and σ is set to be 10^{-5}. In BICM, the maximum number of levels is set to be 3.

TABLE 1. Characteristics of the test matrices.

name	row dimen (m)	column dimen (n)	No. of nnz(A)	No. of nnz(B)
25FV47	821	1876	10705	22968
80BAU3B	2262	12061	23264	22410
BNL2	643	1586	5532	9432
CYCLE	1903	3371	21234	57318
CZPROB	929	3562	10708	15073
D2Q06C	2171	5831	33081	56153
DEGEN3	1503	2604	25432	101859
FFFFF800	524	1028	6401	20706
FINNIS	497	1064	2760	6847
GANGES	1309	1706	6937	16621
GEMAT1	4929	10595	47369	95017
GREENBEA	2392	5598	31070	70071
ILLC1033	1033	320	4732	3974
ILLC1850	1850	712	8758	9126
KEN07	2426	3602	8404	14382
MAROS	846	1966	10137	23670
NUG08	912	1632	7296	28816
PEROLD	625	1506	6148	13491
SCFXM2	660	1200	5469	12312
WELL1033	1033	320	4732	3974
WELL1850	1850	712	8758	9126

In our experiments, we do not form the normal equations explicitly. At the ith step, we form the ith row of A^TA, then perform factorizations for that row before going on to the next row. In BICM factorization, this is only done at the first level. Because the elements in each reduced system after the first level can be considered as part of the preconditioner matrix. The Schur complement submatrix is just the lower right block of the preconditioner matrix.

In all tables with numerical results, "bsize" is the size of the uniform blocks, "iter" shows the number of PCG iterations, "fact" shows the CPU time in seconds for an incomplete factorization, "spar" (sparsity ratio) shows the ratio of storage between the preconditioner factors and the lower triangular part of the matrix A^TA, "rest" shows the number of times that the diagonal scaling stabilization is performed in the factorization. In case of BICM, we use the notation (*,*), the first number indicates the level at which the factorization restarts, the second one indicates the number of times of diagonal scaling at this level. "(L,*)" indicates the number of restarts at the last level. "0" means no restart is needed. In addition, "solu" shows the CPU time in seconds using PCG (or plain CG) to solve the linear systems, "totl" is the total solution time in seconds for solving a given matrix, which is the sum of "fact" and "solu". The symbol "no convergence" indicates lack of convergence (more than 1000 iterations), and "breakdown" indicates a breakdown happens when performing (unsuccessful) factorization with more than 50 restarts. We note that in the case of "no convergence", the incomplete factorization is successful, but the PCG does not converge in 1000 iterations.

First, we attempt to solve all 21 test matrices using plain CG without a preconditioner. The test data are listed in Table 2. There are only 8 out of 21 matrices that can be solved by the plain CG method without preconditioning, which implies that plain CG is not a robust solver for this type of rectangular matrices.

Next, we compare IC, CIMGS, and BICM with block size 1 (point multilevel IC). Table 3 lists the number of restarts, the level on which the restart happened (in case of BICM), the factorization cost, and the sparsity ratio of the three preconditioning methods. Table 4 shows the number of iterations, the PCG solution time and the total CPU time for solving a given matrix. The drop tolerance we use in this set of tests is 10^{-4} for all matrices.

From Tables 3 and 4, the following observations can be made statistically based on our experimental data:

- BICM is the fastest in terms of CPU time among the three preconditioning methods. For those tests BICM converges (18 out of 21), it finishes the incomplete factorization much faster than both IC and CIMGS. In the cases that IC does not break down (9 out of 21), IC finishes the factorization faster than CIMGS. In terms of total solution cost (factorization and solution) BICM is also the fastest among the three preconditioning methods. Note that in many cases, the factorization cost of CIMGS is several times higher than that of BICM. (We remark that there are some discrepancies in the reported small CPU timings. As we indicated earlier, the given computer system has a relatively high system overhead cost, very small CPU timings may not be reliable.)
- Among the three preconditioning methods, both CIMGS and BICM are much more robust than IC. IC breaks down in 12 out of the 21 test cases.

TABLE 2. Solving the rectangular matrices using unpreconditioned CG.

matrix	iter	solu
25FV47	no convergence	–
80BAU3B	217	3.65
BNL2	no convergence	–
CYCLE	no convergence	–
CZPROB	134	0.85
D2Q06C	no convergence	–
DEGEN3	no convergence	–
FFFFF800	no convergence	–
FINNIS	384	0.69
GANGES	237	0.97
GEMAT1	no convergence	–
GREENBEA	no convergence	–
ILLC1033	no convergence	–
ILLC1850	no convergence	–
KEN07	172	1.12
MAROS	no convergence	–
NUG08	8	0.034
PEROLD	no convergence	–
SCFXM2	no convergence	–
WELL1033	162	0.36
WELL1850	425	1.83

Comparing the results of Tables 2 and 3 shows that IC preconditioner does not offer significant preconditioning effect on CG. We see that plain CG and IC preconditioned CG can solve 8 and 9 out of 21 test matrices, respectively, although in the case of IC preconditioned CG, the numbers of iterations are smaller. The test results indicate that, without suitable modifications, IC cannot be used as a serious preconditioner for this type of problems. This observation is in agreement with that obtained by Wang et al. [29]. Both CIMGS and BICM finish all factorizations. However, there are two cases in which CIMGS does not converge and three cases in which BICM does not converge (in 1000 iterations). When both of them converge, in 8 cases, BICM takes fewer iterations. While in 6 other cases (including the one CIMGS converges, but BICM does not), CIMGS converges in fewer iterations. In the remaining 5 converged cases, both BICM and CIMGS converge in exactly the same number of iterations. We therefore conclude that BICM and CIMGS have comparable robustness.

TABLE 3. Comparison of IC, CIMGS, and BICM with block size 1.

	IC			CIMGS			BICM		
matrix	rest	fact	spar	rest	fact	spar	rest	fact	spar
25FV47	breakdown			1	24.42	4.26	(1,1)	9.26	4.46
80BAU3B	2	62.67	5.56	0	113.35	5.71	0	14.02	5.98
BNL2	1	6.58	2.89	1	9.71	2.89	(1,1)	1.02	3.09
CYCLE	breakdown			1	90.52	1.94	(1,1), (L,20)	12.00	1.42
CZPROB	0	42.80	24.43	0	56.29	24.75	0	2.19	0.67
D2Q06C	breakdown			0	216.61	2.30	(L,13)	25.10	2.87
DEGEN3	breakdown			0	136.74	5.65	(L,5)	42.42	5.68
FFFFF800	breakdown			no convergence			no convergence		
FINNIS	0	1.59	4.59	0	2.78	4.55	0	0.44	4.61
GANGES	0	8.84	5.36	0	33.06	5.55	0	2.29	4.91
GEMAT1	breakdown			0	777.77	2.62	(L,12)	139.43	2.58
GREENBEA	1	183.62	6.20	1	444.57	6.18	(1,1)	35.96	6.04
ILLC1033	breakdown			0	2.03	1.81	0	0.40	0.96
ILLC1850	breakdown			0	9.38	3.99	0	1.67	3.62
KEN07	1	10.86	2.25	1	16.77	2.27	(L,2)	5.97	11.16
MAROS	breakdown			no convergence			no convergence		
NUG08	breakdown			1	78.21	9.66	(L,7)	14.21	9.81
PEROLD	breakdown			0	6.35	1.08	no convergence		
SCFXM2	breakdown			0	5.26	1.69	(L,20)	1.36	1.74
WELL1033	0	2.00	2.16	0	2.03	2.16	0	0.40	0.95
WELL1850	0	8.06	4.53	0	9.42	4.53	0	1.66	3.66

- As to the diagonal scaling stabilization strategy with restart, IC only succeeds in 5 cases without restart. With one or two restarts, IC succeeds in 4 more cases. CIMGS finishes factorization on 13 matrices without restart, and on all matrices with at most one restart. BICM succeeds in 8 cases without restart. It finishes factorization on all matrices with different number of restarts on different levels, see Table 3. Note that CIMGS does not need restart if there were no rounding errors in the computation [29]. The necessity of CIMGS restart indicates that some of the normal equation matrices are ill-conditioned.
- The storage costs of three preconditioners are comparable, with just a few exceptions. E.g., for solving the CZPROB matrix, both IC and CIMGS require storage space as much as 24 times of that of A^TA. But BICM only requires 67% of the storage space required for storing A^TA. On the other hand, for solving the KEN07 matrix, BICM needs much more storage space than both IC and CIMGS do. Those exceptions can be attributed to the reordering effects embedded in BICM.

The FFFFF800 matrix can be solved with a smaller drop tolerance value. In Figure 1 we plot the convergence history of BICM (with block size 1) for solving

TABLE 4. Comparison of IC, CIMGS, and BICM with block size 1 (continued).

	IC			CIMGS			BICM		
matrix	iter	solu	totl	iter	solu	totl	iter	solu	totl
25FV47	breakdown			10	0.43	24.85	16	0.24	9.50
80BAU3B	6	0.49	63.16	5	0.37	113.72	5	0.26	14.18
BNL2	8	0.10	6.68	6	0.08	9.79	6	0.02	1.04
CYCLE	breakdown			60	3.42	93.94	695	10.72	22.72
CZPROB	7	1.17	43.97	6	1.06	57.35	4	0.01	2.20
D2Q06C	breakdown			20	1.46	218.07	84	3.13	28.23
DEGEN3	breakdown			6	1.69	138.43	5	0.61	43.03
FFFFF800	breakdown			no convergence			no convergence		
FINNIS	5	0.06	1.65	5	0.06	2.84	3	0.01	0.45
GANGES	5	0.20	9.04	5	0.20	33.26	3	0.04	2.33
GEMAT1	breakdown			37	5.35	783.12	947	63.12	202.55
GREENBEA	5	1.08	184.70	4	0.89	445.46	4	0.42	36.38
ILLC1033	breakdown			10	0.05	2.08	4	0.005	0.405
ILLC1850	breakdown			5	0.09	9.47	5	0.02	1.69
KEN07	2	0.05	10.91	3	0.06	16.83	2	0.07	6.01
MAROS	breakdown			no convergence			no convergence		
NUG08	breakdown			7	0.89	79.00	3	0.13	14.34
PEROLD	breakdown			175	1.35	7.70	no convergence		
SCFXM2	breakdown			67	0.65	5.91	147	0.37	1.73
WELL1033	3	0.02	2.02	2	0.01	2.04	2	0.004	0.404
WELL1850	3	0.06	8.12	3	0.06	9.48	2	0.01	1.67

the FFFFF800 matrix with three values of the drop tolerance. It is noticed that for drop tolerance values 10^{-5} and 10^{-6}, the convergence curves of BICM are oscillatory. This indicates that the vectors generated by CG procedure lose their orthogonality during the computation, due to the ill conditioning of the (inaccurately) preconditioned matrix. However, we can see that when the drop tolerance value is 10^{-7}, BICM converges very quickly in only 9 iterations.

Tables 5 and 6 compares the number of iterations, the factorization time, the solution time, and the number of times of diagonal scaling (restart) performed, under comparable memory cost for the three preconditioning methods. In this set of tests, we choose some large size blocks for BICM, which is more suitable for parallel implementations [26]. These two tables illustrate the robustness and effectiveness of the three preconditioning methods under limited memory situation. Here we tend to keep the memory cost as small as possible. Therefore, the drop tolerance used in different preconditioning method may be different. The results are not comparable for those matrices which do not show up in the tables.

From the results in Tables 5 and 6, we may see that BICM and CIMGS are again more efficient and more effective than IC. If we are restricted in small

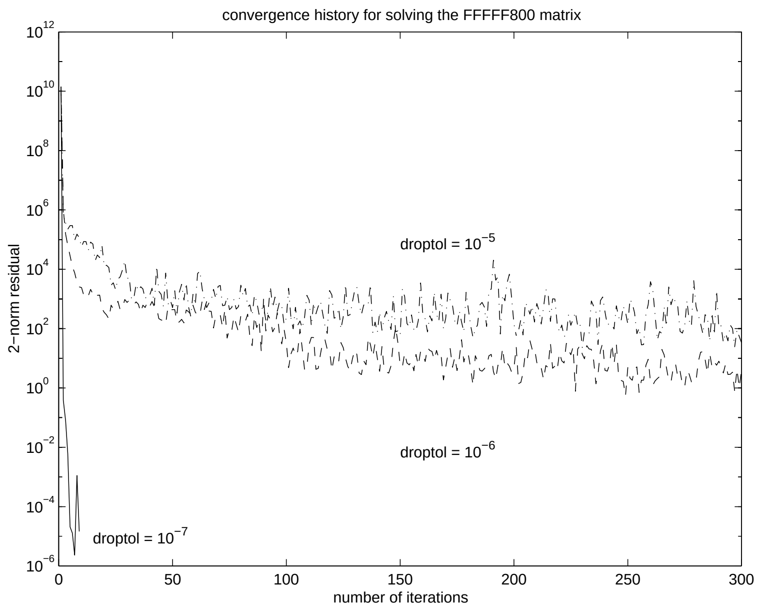

FIGURE 1. Convergence history (2-norm residual) of BICM for solving the FFFFF800 matrix with different drop tolerance values.

memory computations, 11 out of 14 matrices will not be solved with IC factorization. IC factorizations break down on these 11 matrices. There is only 1 matrix with which CIMGS does not converge. BICM converges for all matrices under this set of tests. The reason for better behaved BICM is because it is more controllable than both IC and CIMGS. We may control the memory cost by adjusting several parameters in BICM, such as the block size, the number of levels, and the drop tolerance. But in IC and CIMGS, only the drop tolerance is adjustable. In fact, the block size in BICM may affect its convergence for solving a given matrix. Table 7 lists comparable results of BICM with different block sizes, where “droptol” indicates the drop tolerance used in the factorizations.

From Table 7, we can see that the choice of block size affects the convergence of BICM. With the same drop tolerance, different block size may lead to different rate of convergence. For some matrices, one choice of block size makes the PCG iteration converge faster and takes less memory cost while another choice may lead to no convergence. This observation may look like a disadvantage for BICM with large size blocks, since it indicates that BICM with large size blocks is not suitable for implementations in parallel black-box solvers. However, since IC is far less robust and CIMGS is far more expensive, BICM (at least with block size 1) seems to be a robust and yet cost-effective preconditioner for

Table 5. Comparison of IC, CIMGS, and BICM (with large block size) under comparable memory cost.

	IC			CIMGS			BICM			
matrix	rest	fact	spar	rest	fact	spar	bsize	rest	fact	spar
25FV47	breakdown			1	24.42	4.26	100	(1,14), (L,20)	13.52	4.06
80BAU3B	0	57.86	1.02	0	103.55	1.05	150	0	12.27	1.56
BNL2	breakdown			1	9.14	1.06	200	(1,16), (2,16)	3.73	1.08
CYCLE	1	50.02	3.35	1	93.64	3.13	200	(1,16), (L,21)	24.86	3.37
FFFFF800	breakdown			0	8.11	4.17	50	0	1.92	4.62
FINNIS	breakdown			0	2.40	0.97	170	(1,21)	1.58	0.66
GANGES	breakdown			0	25.96	1.68	150	(1,1)	2.12	1.78
GREENBEA	breakdown			1	394.70	1.11	220	(1,1)	33.73	1.31
ILLC1033	breakdown			0	2.03	1.81	250	(1,5), (1,6)	1.71	1.82
ILLC1850	breakdown			0	9.24	2.43	400	(1,8), (L,11)	12.36	2.51
NUG08	breakdown			breakdown			100	(1,16), (L,23)	2.96	0.88
SCFXM2	breakdown			0	5.26	1.69	250	(1,14), (L,23)	12.12	1.72
WELL1033	breakdown			0	2.03	1.67	250	(L.9)	0.46	1.86
WELL1850	0	7.47	1.59	0	8.91	1.60	200	(2,12)	1.67	1.57

Table 6. Comparison of IC, CIMGS, and BICM (with large block size) under comparable memory cost (continued).

	IC			CIMGS			BICM		
matrix	iter	solu	totl	iter	solu	totl	iter	solu	totl
25FV47	breakdown			10	0.43	24.85	153	2.03	15.55
80BAU3B	17	0.39	58.25	16	0.17	103.72	16	0.15	12.42
BNL2	breakdown			34	0.21	9.35	65	0.10	3.83
CYCLE	5	0.48	50.51	10	0.90	94.54	750	29.19	54.05
FFFFF800	breakdown			81	2.68	10.79	7	0.09	2.01
FINNIS	breakdown			31	0.12	2.52	81	0.06	1.64
GANGES	breakdown			13	0.18	26.04	16	0.07	2.19
GREENBEA	breakdown			14	0.67	395.37	24	0.49	34.22
ILLC1033	breakdown			10	0.05	2.08	81	0.08	1.79
ILLC1850	breakdown			9	0.11	9.35	82	0.21	12.52
NUG08	breakdown			breakdown			26	0.10	3.06
SCFXM2	breakdown			67	0.64	5.90	163	0.42	12.54
WELL1033	breakdown			9	0.04	2.07	16	0.02	0.48
WELL1850	10	0.09	7.56	7	0.07	8.98	62	0.14	1.81

solving rectangular matrices. The increased degree of parallelism inherited in BICM is also an advantage that may distinguish it from IC and CIMGS when implemented on high performance computers [26].

Figure 2 illustrates the effect of block size and PCG convergence with four matrices: 80BAU3b, NUG08, WELL1033, and WELL1850. We can see that the

TABLE 7. Test results of BICM with different block sizes.

		bsize=100		bsize=150		bsize=200		bsize=250	
matrix	droptol	iter	spar	iter	spar	iter	spar	iter	spar
25FV47	10^{-4}	14	6.46	13	6.96	19	6.90	114	7.67
80BAU3B	10^{-2}	17	1.61	16	1.57	17	1.53	17	1.61
BNL2	10^{-3}	65	2.93	87	3.11	12	2.47	14	2.21
CYCLE	10^{-5}	no convergence		no convergence		662	3.73	no convergence	
CZPROB	10^{-2}	19	3.99	19	3.05	19	2.55	19	2.11
D2Q06C	10^{-4}	no convergence		542	6.00	538	5.35	no convergence	
DEGEN3	10^{-2}	75	4.84	116	3.77	134	4.21	148	3.82
FFFFF800	10^{-10}	8	5.42	8	5.35	7	5.41	8	5.34
FINNIS	10^{-3}	13	3.76	10	3.13	18	3.16	11	3.26
GANGES	10^{-2}	65	1.50	16	1.72	170	1.08	143	1.04
GEMAT1	10^{-5}	284	14.70	no convergence		no convergence		no convergence	
GREENBEA	10^{-2}	139	1.70	42	1.60	29	1.53	84	1.34
ILLC1033	10^{-4}	32	6.74	24	5.48	18	3.38	24	2.39
ILLC1850	10^{-2}	682	10.47	944	11.88	no convergence		237	6.85
KEN07	10^{-1}	57	0.77	48	0.76	50	0.77	50	0.76
MAROS	10^{-8}	no convergence		no convergence		861	6.65	8	6.21
NUG08	10^{-1}	56	0.93	18	1.13	98	0.77	51	0.89
PEROLD	10^{-7}	14	3.77	11	3.65	16	3.75	17	3.69
SCFXM2	10^{-4}	no convergence		340	1.06	331	0.98	163	1.71
WELL1033	10^{-2}	34	5.31	38	3.78	25	2.61	16	1.96
WELL1850	10^{-2}	51	8.34	72	7.52	71	6.90	33	5.81

curves fluctuate except for the matrix 80BAU3B. The information we depicts in Figure 2 is in agreement with that contained in Table 7.

6. Summary Remarks and Future Studies

We have developed a multilevel block IC factorization preconditioner for symmetric positive definite matrices arising from solving rectangular matrices, and compared it with the existing IC and CIMGS factorizations. This new method (BICM) is originated from IC factorization, with the addition of a multilevel reordering strategy and a diagonal scaling stabilization strategy. BICM factorization is the fastest among the three preconditioning methods. It takes much less time to perform the incomplete factorization than IC and CIMGS factorizations do. One explanation is that when encountering breakdown, we add a positive scalar to the diagonal elements of the matrix and then repeat the factorization only at the current level. But in IC or CIMGS, we repeat the factorization on the whole matrix when breakdown occurs. Moreover, in BICM the bandwidth of the matrix D is reduced a lot due to the reordering of the matrix with diagonal blocks, this saves factorization time, compared to IC and CIMGS.

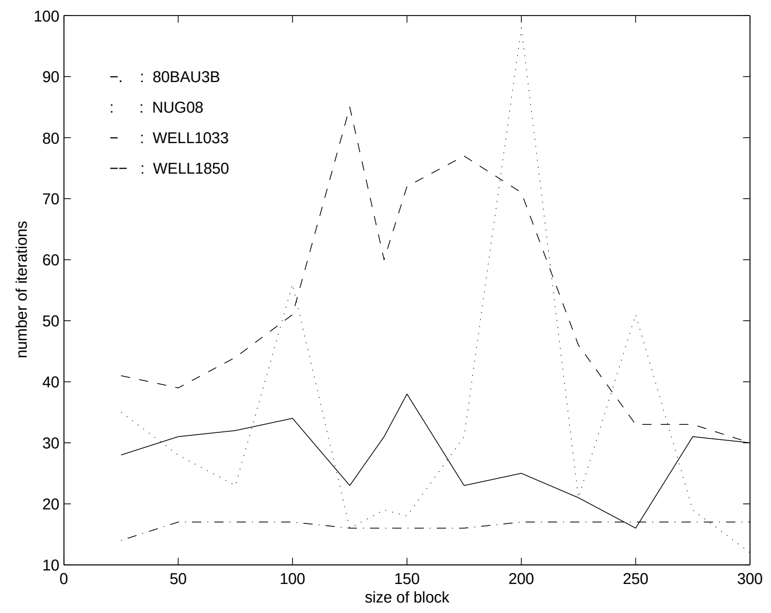

FIGURE 2. Number of iterations as a function of block size of BICM.

In terms of total solution cost for solving a rectangular matrices, BICM is also much faster than both IC and CIMGS.

In general, both BICM and CIMGS are more efficient and more effective than IC. BICM with block size 1 is shown to be as robust as CIMGS. When the three preconditioners are tested under limited memory situations, BICM with large size blocks seems to be more efficient than IC, but in general takes more iterations to converge than CIMGS does. However, in most cases, the total CPU time consumed by BICM is still the least among the three preconditioning methods.

In order to deal with the breakdown problem, we apply a diagonal scaling strategy and restart the factorizations repeatedly. From the test results, we see that this strategy works well for BICM and CIMGS. We also tried another strategy to handle breakdown, i.e., to substitute the diagonal element in question by 1 when encountering breakdown. This strategy is not as effective as the one we use in the computations presented in this article.

When searching for a block independent set in BICM, the algorithm we use in our test is the greedy algorithm (Algorithm 3). We also tried another method to search for block independent set, in which we choose those nodes which have relatively large size diagonals as nodes in the block independent set, hoping to reduce the possibility of breakdown. But this method did not show much improvement in our current implementation.

We only implemented one dropping strategy based on the size of the computed matrix elements. Other dropping strategies based on matrix structure may be useful in certain applications. There are also possibilities to use hybrid dropping strategies or dual dropping strategies [22] to better control the fill-in. These options will be investigated in our future studies.

The implementation of a parallel diagonal scaling stabilization strategy will be interesting. Since individual blocks are distributed in different processors [26], they are factored independently with restart if necessary. Thus different processors may perform different numbers of restart.

References

[1] M. A. Ajiz and A. Jennings, *A robust incomplete Choleski-conjugate gradient algorithm*, Int. J. Numer. Methods Engrg., **20** (1984), 949-966.

[2] S. Ashby, *Minimax polynomial preconditioning for Hermitian linear systems*, SIAM J. Matrix Anal. Appl., **12** (1991), 766-789.

[3] M. W. Berry and R. J. Plemmons, Âlgorithms and experiments for structural mechanics on high-performance architectures, Comput. Methods Appl. Mech. Engrg., **64** (1987), 487-507.

[4] A. Björck, *Numerical Methods for Least-Squares Problems*, SIAM, Philadelphia, PA, 1996.

[5] A. Björck and J. Y. Yuan, *Preconditioners for least squares problems by LU factorization*, Elect. Trans. Numer. Anal., **8** (1999), 26-35.

[6] I. Chio, C. L. Monna, and D. Shanno, *Future development of a primal-dual interior point method*, ORSA J. Comput., **2** (1990), 304-311.

[7] M. Fortin and R. Glowinski, *Augmented Lagrange Methods: Applications to the Numerical Solution of Boundary-Value Problems*, North-Holland, Amsterdam, 1983.

[8] R. Freund, *A note on two SOR methods for sparse least squares problems*, Linear Algebra Appl., **88/89** (1987), 211-221.

[9] G. H. Golub and J. M. Ortega, *Scientific Computing: An Introduction with Parallel Computing*, Academic Press, Boston, MA, 1993.

[10] G. H. Golub and C. F. van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.

[11] M. T. Heath, R. J. Plemmons, and R. C. Ward, *Sparse orthogonal schemes for structural optimization using force method*, SIAM J. Sci. Statist. Comput., **5** (1984), 514-532.

[12] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, J. Research Nat. Bur. Standards, **49** (1952), 409-436.

[13] A. Jennings and M. A. Ajiz, *Incomplete methods for solving $A^TAx = b$*, SIAM J. Sci. Statist. Comput., **5** (1984), 978-987.

[14] M. T. Jones and P. E. Plassmann, *An incomplete Cholesky factorization*, ACM Trans. Math. Software, **21** (1995), 5-17.

[15] E. G. Kolata, *Geodesy: dealing with an enormous computer task*, Science, **200** (1978), 421-422.

[16] C. Lin and J. J. Moré, *Incomplete Cholesky factorization with limited memory*, SIAM J. Sci. Comput., **21** (1999), 24-45.

[17] T. A. Manteuffel, *An incomplete factorization technique for positive definite linear systems*, Math. Comput., **34** (1980), 473-497.

[18] J. A. Meijerink and H. A. van der Vorst, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Math. Comput., **31** (1977), 148-162.

[19] C. C. Paige and M. A. Saunders, *LSQR: an algorithm for sparse linear equations and sparse least squares*, ACM Trans. Math. Software, **8** (1982), 43-71.

[20] J. R. Rice, *PARVEC workshop on very large least squares problems and supercomputers*, Technical Report CSD-TR 464, Department of Computer Science, Purdue University, West Lafayette, IN, 1983.

[21] Y. Saad, *Preconditioning techniques for nonsymmetric and indefinite linear systems*, J. Comput. Appl. Math., **24** (1988), 89-105.

[22] Y. Saad, *ILUT: a dual threshold incomplete LU preconditioner*, Numer. Linear Algebra Appl., **1** (1994), 387-402.

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

[24] Y. Saad and J. Zhang, *BILUM: block versions of multielimination and multilevel ILU preconditioner for general sparse linear systems*, SIAM J. Sci. Comput., **20** (1999), 2103-2121.

[25] Y. Saad and J. Zhang, *BILUTM: a domain-based multilevel block ILUT preconditioner for general sparse matrices*, SIAM J. Matrix Anal. Appl., **21** (1999), 279-299.

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

[27] G. W. Stewart, *Matrix Algorithms, Volume I: Basic Decompositions*, SIAM, Philadelphia, PA, 1998.

[28] X. Wang, *Incomplete Factorization Preconditioning for Linear Least Squares Problems*, PhD thesis, University of Illinois, Urbana-Champaign, IL, 1993.

[29] X. Wang, K. A. Gallivan, and R. Bramley, *CIMGS: an incomplete orthogonal factorization preconditioner*, SIAM J. Sci. Comput., **18** (1997), 516-536.

Jun Zhang received a Ph.D. from The George Washington University in 1997. He is an Associate Professor of Computer Science and Director of the Laboratory for High Performance Scientific Computing and Computer Simulation at the University of Kentucky. His research interests include large scale parallel and scientific computing, numerical simulation, iterative and preconditioning techniques for large scale matrix computation. Dr. Zhang is associate editor and on the editorial boards of three international journals in computer simulation and computational mathematics, and is on the program committees of a few international conferences. His research work is currently funded by the U.S. National Science Foundation, the U.S. Department of Energy Office of Science, Japanese Research Organization for Information Science & Technology (RIST), and the University of Kentucky Research Committee. He is recipient of the U.S. National Science Foundation CAREER Award and several other awards.

Tong Xiao received an M.S. degree in Computer Science from the University of Kentucky in 2000.

Laboratory for High Performance Scientific Computing and Computer Simulation, Department of Computer, University of Kentucky, 773 Anderson Hall, Lexington, KY 40506–0046, USA
`E-mail: jzhang@cs.uky.edu URL: http://www.cs.uky.edu/~jzhang`