Available online at www.sciencedirect.com
SCIENCE DIRECT®
J. Parallel Distrib. Comput. 65 (2005) 331–346
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Distributed block independent set algorithms and parallel multilevel ILU preconditioners☆

Chi Shen, Jun Zhang*, Kai Wang

Department of Computer Science, Laboratory for High Performance Scientific Computing and Computer Simulation, University of Kentucky, 773 Anderson Hall, Lexington, KY 40506-0046, USA

Received 22 May 2003; received in revised form 15 October 2004
Available online 11 January 2005

Abstract

We present a class of parallel preconditioning strategies utilizing multilevel block incomplete LU (ILU) factorization techniques to solve large sparse linear systems. The preconditioners are constructed by exploiting the concept of block independent sets (BISs). Two algorithms for constructing BISs of a sparse matrix in a distributed environment are proposed. We compare a few implementations of the parallel multilevel ILU preconditioners with different BIS construction strategies and different Schur complement preconditioning strategies. We also use some diagonal thresholding and perturbation strategies for the BIS construction and for the last level Schur complement ILU factorization. Numerical experiments indicate that our domain-based parallel multilevel block ILU preconditioners are robust and efficient.

MSC: 65F10; 65F50; 65N55; 65Y05

Keywords: Parallel multilevel preconditioning; Block independent set; Sparse matrices; Schur complement techniques

1. Introduction

Algorithmic scalability is important in solving very large-scale problems on high-performance computers [7]. It is well known that suitably implemented multilevel or multiscale algorithms can demonstrate ideal scalability for solving model problems [5,6,24,39]. Although, it is generally believed that it may be impossible to obtain optimal algorithmic scalability for solving general problems, the quest for near-optimal algorithms arising from multilevel and multigrid approaches has been going on for years [1,4,9,16].

For solving large sparse linear systems, multilevel preconditioning techniques may take advantage of the fact that different parts of the error spectrum can be treated independently on different level scales. A class of high-accuracy multilevel preconditioners that combine the inherent parallelism of domain decomposition, the robustness of ILU factorization, and the scalability potential of multigrid method has been developed by a few authors. Several multilevel ILU preconditioners, such as ILUM, BILUM, and BILUTM [28,32,33], have been tested to show promising convergence results and scalability for solving certain sparse matrices. Interlevel iteration strategies are introduced in [31,42,44] to improve outer Krylov iterations. Diagonal thresholding strategies for enhancing factorization stability are implemented in [31,34,35] and are analyzed in [34]. Singular value decomposition-based block diagonal ILU factorization strategies are implemented in [36,43]. Similar multilevel approaches have been explored in [2,3,12]. Some grid-based

☆ This research work was supported in part by the US National Science Foundation under Grants CCR-9902022, CCR-9988165, CCR-0092532, and ACI-0202934, by the US Department of Energy Office of Science under Grant DE-FG02-02ER45961, by the Japanese Research Organization for Information Science and Technology, and by the University of Kentucky Research Committee.

* Corresponding author. Fax: +1 859 323 1971.
E-mail addresses: cshen@cs.uky.edu (C. Shen), jzhang@cs.uky.edu (Jun Zhang), kwang0@cs.uky.edu (Kai Wang)
URL: http://www.cs.uky.edu/~jzhang.

0743-7315/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2004.10.007

multilevel ILU factorization approaches which are more akin to the classic algebraic multigrid methods are proposed in [23,41,44].

The parallelization of multilevel ILU preconditioning methods and their implementations on distributed memory parallel computers are a nontrivial issue [29]. Most reported parallel implementations of multilevel preconditioning methods are limited to two levels [20,30,37], with the exception of the work of Karypis and Kumar [18]. However, the parallel multilevel preconditioner developed in [18] is more like a domain decomposition preconditioner. At the finest level, each processor holds only one large submatrix corresponding to the interior nodes of the local domain. At the following levels, Luby's parallel maximal independent set algorithm [21] is used to find a (point) independent set of the reduced (Schur complement) system. The recursion with the Schur complement-type reduction is again exploited. This approach, with unbalanced independent sets at the fine and coarse levels, may have two obvious problems. One problem is the inaccurate ILU factorization of the very large blocks on the finest level. The second problem is the slow reduction of the Schur complement matrix size due to the small (point) independent sets formed on the coarse levels.

The difficulty associated with the parallel implementations of multilevel block ILU preconditioning methods is related to computing the block independent set (BIS) from a distributed sparse matrix. For the parallel 2-level block ILU preconditioning methods such as PBILU2 [37], the BIS can be constructed in a single processor before the coefficient matrix is distributed to other processors. This is, of course, to assume that the matrix is initially read or stored in a single processor. Otherwise a large single local block can be formed in each processor as in [18].

Previously developed PBILU2 [37] is a 2-level implementation of the parallelized BILUTM for distributed memory computer systems. In PBILU2, new data structures and novel implementation strategies are used to construct a local submatrix and a local Schur complement matrix on each processor. The preconditioner constructed is shown to be fast, memory efficient, and robust for solving certain large sparse matrices [37].

This paper focuses on two algorithms for constructing BIS from a distributed sparse matrix, which are used to build a class of truly parallel multilevel block ILU preconditioners (PBILUM). The two proposed distributed BIS algorithms are distinct. One is based on a parallel BIS search and the other is based on a sequential BIS search analogous to that used in BILUTM. However, the construction of the global BIS in both algorithms is carried in parallel.

In order to enhance the robustness and stability of the parallel block ILU factorization, we implement some diagonal thresholding and perturbation strategies in PBILUM for the BIS construction at each level and for the Schur complement ILU factorization at the coarsest level [34,41]. The experimental results show that these diagonal thresholding and perturbation strategies help improve robustness of the parallel multilevel block ILU factorization.

This paper is organized as follows. In Section 2, we outline a general framework of the 2-level block ILU preconditioning techniques (PBILU2). In Section 3, we devise two different BIS search algorithms and other components of PBILUM. Diagonal thresholding and perturbation strategies are given in Section 4. Section 5 contains a comparison of several variants of the PBILUM preconditioners for solving several distributed sparse linear systems. Finally, concluding remarks are given in Section 6.

2. A 2-level block ILU preconditioner

PBILUM is a parallel multilevel block ILU preconditioner built on the framework of the parallel 2-level block ILU preconditioning techniques (PBILU2) described in [37]. The BIS algorithm in PBILU2 is a sequential greedy algorithm introduced in [32,33]. For completeness, we give a short overview of the main steps in constructing PBILU2.

2.1. Distributed matrix based on BIS

Let us consider a sparse linear system $Ax = b$, where A is a nonsingular sparse matrix of order n. There are several heuristic strategies to find BISs from a sparse matrix, although the problem of finding the largest BIS may be difficult [32]. The simplest approach is to couple the nearest nodes together. This algorithm will be discussed in Sections 3 and 4.

Assume that a BIS with a uniform block size k has been found, the coefficient matrix A is permuted into a block system

$$\left(\begin{array}{cccc|c} B_1 & & & F_1 & \\ & B_2 & & F_2 & \\ & & \ddots & \vdots & \\ & & B_m & F_m & \\ \hline & E_1 & & C_1 & \\ & E_2 & & C_2 & \\ & \vdots & & \vdots & \\ & E_m & & C_m & \end{array}\right)\left(\begin{array}{c} u_1 \\ u_2 \\ \vdots \\ u_m \\ \hline y_1 \\ y_2 \\ \vdots \\ y_m \end{array}\right)=\left(\begin{array}{c} f_1 \\ f_2 \\ \vdots \\ f_m \\ \hline g_1 \\ g_2 \\ \vdots \\ g_m \end{array}\right), \tag{1}$$

where m is the number of processors used in the computation. The global vector of unknowns x is split into two subvectors $(u, y)^T$, where $u = (u_1, \ldots, u_m)^T$, $y = (y_1, \ldots, y_m)^T$. The right-hand side vector b is also split conformally into subvectors f and g. Each block diagonal submatrix B_i may contain several independent blocks.

For computing the global Schur complement matrix in parallel, the submatrix E needs to be partitioned in two forms: one is by rows and the other by columns (see [37]

for details), i.e.,

$$E = (M_1 \quad M_2 \quad \cdots \quad M_m) = (E_1 \quad E_2 \quad \cdots \quad E_m)^T. \tag{2}$$

The submatrices B_i, F_i, E_i, M_i, C_i and the subvectors f_i and g_i are assigned to the same processor i. u_i and y_i are the local parts of the unknown vectors. When this processor-data assignment is done, each processor holds several rows of the equations. For detailed discussions on our motivations and implementations, see [37].

2.2. Parallel construction of Schur complement matrix

We now consider a block LU factorization of (1) in the form of

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix} = \begin{pmatrix} I & 0 \\ EB^{-1} & I \end{pmatrix} \begin{pmatrix} B & F \\ 0 & S \end{pmatrix}, \tag{3}$$

where S is the global Schur complement. Based on Eqs. (1) and (3), we have

$$S = \begin{pmatrix} C_1 \\ \vdots \\ C_m \end{pmatrix} - \sum_{i=1}^{m} M_i B_i^{-1} F_i. \tag{4}$$

In the ith processor, a local matrix A_i is formed from the submatrices assigned to this processor and an ILU factorization on this local matrix is performed. This local matrix on the processor i looks like

$$A_i = \begin{pmatrix} B_i & F_i \\ M_i & \tilde{C}_i \end{pmatrix} = \left(\begin{array}{c|c} B_i & F_i \\ \hline & 0 \\ & \vdots \\ M_i & C_i \\ & \vdots \\ & 0 \end{array}\right). \tag{5}$$

We perform a restricted Gaussian elimination on the local matrix A_i [33,37]. First, we perform an (I)LU factorization on the upper part (B_i, F_i) of A_i. The submatrix B_i is factored as $L_{B_i} U_{B_i}$. We then continue the Gaussian elimination to the lower part $(M_i, \tilde{C}_i)$. We can obtain a new reduced submatrix $\tilde{C}_i$, and it forms a piece of the global Schur complement matrix. If the factorization procedure is exact, the global Schur complement matrix can be formed by summing all these submatrices $\tilde{C}_i$, i.e., $S = \tilde{C}_1 + \tilde{C}_2 + \cdots + \tilde{C}_m$. Each submatrix $\tilde{C}_i$ is partitioned into m parts (using the same partitionings as the original submatrix C), and the corresponding parts are scattered to the relevant processors. After receiving and summing all parts of the submatrices which have been scattered from different processors, the local part of the Schur complement matrix, S_i, is formed. Here S_i holds a few rows of the global Schur complement matrix. Note that $S_i \neq \tilde{C}_i$.

3. Parallel multilevel preconditioner

In algebraic multilevel preconditioning techniques, the reduced systems are recursively constructed as the Schur complement matrices. The recursive structure of the multilevel preconditioning matrices is depicted in Fig. 1.

Based on Eq. (3), the multilevel factorization at the lth level is in the form of

$$P_l S_l P_l^T = \begin{pmatrix} B_l & F_l \\ E_l & C_l \end{pmatrix} \approx \begin{pmatrix} I & 0 \\ E_l (L_{B_l} U_{B_l})^{-1} & I \end{pmatrix} \begin{pmatrix} B_l & F_l \\ 0 & S_{l+1} \end{pmatrix},$$

where $L_{B_l} U_{B_l}$ is an ILU factorization of the block diagonal matrix B_l. This process is similar to the BILUTM factorization [33]. The parallel implementation of this generic process is described in the following algorithm:

ALGORITHM 3.1. PBILUM factorization.

1. **If** l is not the last level, **Then**
2. Find a global block independent set.
3. Permute the local matrix in each processor.
4. Generate the local reduced matrix for the next level in parallel.
5. **Else**
6. Perform ILUT factorization on the block diagonal reduced matrix.
7. **EndIf**

The parallel implementations of Steps 4 and 6 have already been discussed in Section 2.2 and in [37]. One of the most important steps in parallel multilevel block ILU preconditioning techniques is to develop algorithms for constructing BIS efficiently from a distributed sparse matrix. Two approaches to accomplishing this task are discussed in the next two subsections. Based on these two approaches, the parallel multilevel preconditioning methods with Schur complement approximating strategies are discussed in the last two subsections.

3.1. Sequential BIS search

One of the simplest approaches to finding a BIS is the sequential search, utilizing the greedy algorithm developed in [32]. Once a distributed matrix is formed, each processor

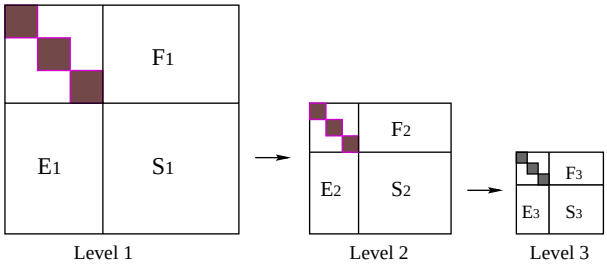

Fig. 1. Recursive structure of the multilevel preconditioning matrices.

holds several rows of it. The nonzero structure (graph) of the local matrix and the array of the diagonal thresholding measure of each row (see Section 4) are sent to one processor. We can find a BIS in this processor, then permute this matrix based on BIS, see the permuted matrix in Eq. (1) for an illustration, and distribute this matrix to other processors. The procedure is detailed in Algorithm 3.2. It is the same as in the 2-level case (PBILU2 [37]). The preconditioner constructed should be algorithmically equivalent to BILUTM [33].

ALGORITHM 3.2. Sequential BIS search with criterion β.

0. Set $k = 0$ and $T = \emptyset$, given the block size *bsize*.
1. **For** $j = 1, \ldots, n$, (global iteration)
2. **If** (node j is marked **or** $\omega(j) < \beta$), **Then**
3. goto Step 1 with $j = j + 1$.
4. **EndIf**
   (Find the nearest-neighboring nodes of the current node j)
5. Let $k = k + 1$ (begin a new block B_k initialized as $\emptyset$).
6. $B_k = B_k \cup \{j\}$.
7. Mark and add this node to a temporary set: $T = T \cup \{j\}$.
8. **Do** while ($B_k <$ *bsize*)
9. **If** ($T \neq \emptyset$), **Then**
10. Choose any node v from T.
11. **If** (adj$(v) \neq \emptyset$), **Then**
12. Find a node $s \in$ adj(v).
13. **If** ($\omega(s) < \beta$), **Then**
14. Add this node to the current block: $B_k = B_k \cup \{s\}$.
15. Add this node to the temporary set: $T = T \cup \{s\}$.
16. Mark and remove this node from adj(v).
17. **Else**
18. Remove this node from adj(v).
19. **EndIf**
20. **If** ($B_k <$ *bsize*), goto Step 11.
21. **Else**
22. Remove node v from the set T.
23. **EndIf**
24. **Else**
25. Unmark all nodes at the jth iteration, and let $k = k - 1$.
    goto Step 1 with $j = j + 1$.
26. **EndIf**
27. **EndDo**
28. **EndFor**

In Algorithm 3.2, the notation adj(j) is the set of the adjacent nodes of the node j. This algorithm is similar to that proposed in [32]. The only difference between them is that we use a diagonal thresholding strategy to avoid some rows (nodes) with small diagonal entries in the BIS [34,35]. The parameter $\omega(i)$ is the diagonal dominance measure of the ith row and β is the diagonal thresholding criterion which is computed according to the current level matrix. For more details, see Section 4.

The main disadvantage of this algorithm is that it may be expensive to implement. Since the process of searching the BIS is carried out in a single processor, the other processors are idle. Furthermore, in the subsequent global permutation phase, a large amount of data may need to be exchanged among the processors. The communication cost may be high. However, due to the global nature of the algorithm, the size of the BIS found sequentially is usually larger than that found by the parallel algorithm, to be discussed later. This may lead to a smaller reduced system and a faster reduction of matrix size between levels. The quality of the constructed preconditioner approaches that of the (sequential) BILUTM [33].

3.2. Parallel BIS search

Developing efficient parallel algorithms for constructing maximal independent set has been an interesting topic in graph theory and has many potential applications in parallel computing and graph partitioning [18,19,21]. However, the issue of developing parallel algorithms for constructing BIS from a distributed sparse matrix graph does not seem to be studied extensively, to the best of our knowledge. We now describe a parallel procedure to construct a global BIS from a distributed matrix.

At the beginning, if the original matrix is read in one processor, it is better to find a BIS and perform the matrix permutation in this processor. The permuted matrix is then distributed to individual processors. In this case, our algorithm will be applied to the distributed Schur complement matrix. In the following, we assume that the matrix has already been distributed onto individual processors.

For the sake of reducing communications among processors, we only search a BIS from the local block diagonal matrix instead of from the global matrix. We can apply Algorithm 3.2 to each local matrix in each processor independently. After this is done, a local BIS set is constructed and resides in a local processor, see Fig. 2.

However, the union of these local BISs may not form a global BIS. Some nodes in a local BIS may be coupled to the nodes in local BISs of the other processors. In Fig. 2, the solid dots with the downward arrows represent those offdiagonal entries that destroy the global independence of certain local independent blocks. In order to form a global BIS, those nodes should be removed from the local BISs. We propose two approaches for this purpose. At least one communication step is needed to exchange local BIS information among the processors.

3.2.1. Node removal approach

With all permutation information, each processor can determine the nodes in its local BIS that are not globally inde-

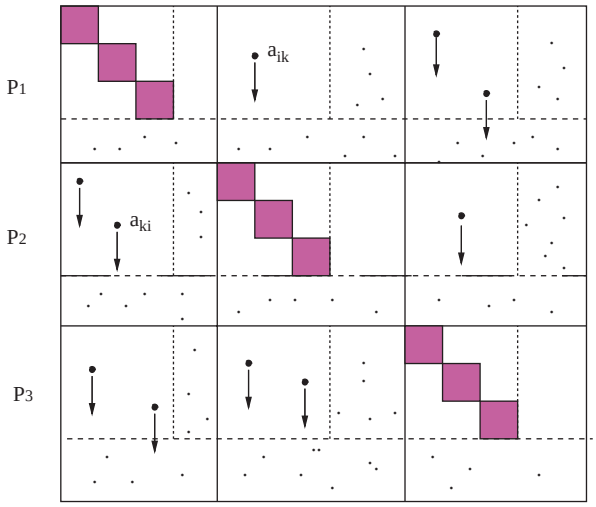

Fig. 2. Parallel search of local BISs.

pendent and remove those nodes from its local BIS. After all processors finish this step independently, the union of all distributed local BISs forms a global BIS.

There is a potential double node removal problem in the just described procedure. For e.g., in Fig. 2, suppose the entry a_{ik} in the processor 1 and the entry a_{ki} in the processor 2 are both nonzero. Independently, the processors 1 and 2 will remove the node i and the node k from their local BISs, respectively. Although this double node removal problem does not affect the global independence of the global BIS, the size of the global BIS is unnecessarily reduced. The reason is that *either* removing the node i from the BIS of the processor 1 *or* removing the node k from the BIS of the processor 2 is sufficient to guarantee the global independence of the local BISs. However, since the node removal is carried out in individual processors independently, this information is not shared among the processors.

We propose some strategies to alleviate the double node removal problem. These strategies can be implemented using the global information exchanged. Each processor should know which nodes in its local BIS are coupled with *which* nodes in the local BISs of *which* other processors. Then each processor can independently perform the node removal procedure based on the following rules:

R1: If a node is coupled to more than one node of BISs in other processors, this node is to be removed from the local BIS;

R2: A processor with a larger local BIS has a higher priority to remove some nodes of its BIS which are coupled with other nodes in the local BISs of the other processors which have a lower priority to do so. For e.g., in our previous discussion, if the processor 1 has a larger BIS than the processor 2 does, the processor 1 will remove the node i from its BIS;

R3: In the case of a tie, the node in the processor with the least label (identification number) is removed from the local BIS.

Of course, none of these rules is optimal. For e.g., Rule R1 may still cause some nodes to be removed unnecessarily. There is room for further improvements. The motivation to eliminate double node removal problem is to find larger BIS at each level and have smaller size of the last reduced matrix. So that the computational cost in the solution phase will be less and thus the overall parallel performance can be improved and increased. The implementations of these improvements may be expensive, since more global information exchange is needed. In this paper, the simple node removal approach without fixing the double node removal problem is used in our experimental tests.

3.2.2. *Entry dropping approach*

We may use a predetermined thresholding parameter σ (preset or computed in the code, not as an input parameter). When an offdiagonal entry a_{ij} couples a node in the local BIS to one node in the local BIS of another processor, the absolute value of this entry is compared with the thresholding parameter σ. If $|a_{ij}| \leqslant \sigma$, this entry is dropped. This strategy is equivalent to setting a second dropping process to further sparsify the matrix. This strategy will not severely hurt the accuracy of the preconditioner, if σ is not too large, since the factorization performed after the construction of the BIS is incomplete, which will remove some couplings due to the dropping strategies imposed in the ILU factorization. This delayed dropping strategy can also help keep some small entries that are inside the BIS, so that more accurate factorization may be performed, compared to the case that a larger dropping parameter is set initially in the ILU factorization. If $|a_{ij}| > \sigma$, we have to use the node removal approach to deal with this node.

Fig. 3 shows a global BIS. Compared to Fig. 2, the sizes of the local blocks in Fig. 3 are smaller than those of the blocks in Fig. 2, as some nodes are removed during the construction of the global BIS.

A generic algorithm for parallel BIS search is described in Algorithm 3.3.

ALGORITHM 3.3. Parallel BIS search.

0. In the ith processor, do:
1. Use the Algorithm 3.2 to find a local BIS in the local block diagonal matrix.
2. Find the local permutation array $\tilde{P}_i$.
3. Exchange its permutation array $\tilde{P}_i$ with other processors.
4. Remove (or drop) those nodes (or entries) which are not globally independent.
5. Find the new local permutation array P_i.

This parallel algorithm for finding BISs has a number of advantages, compared with the sequential one. The commu-

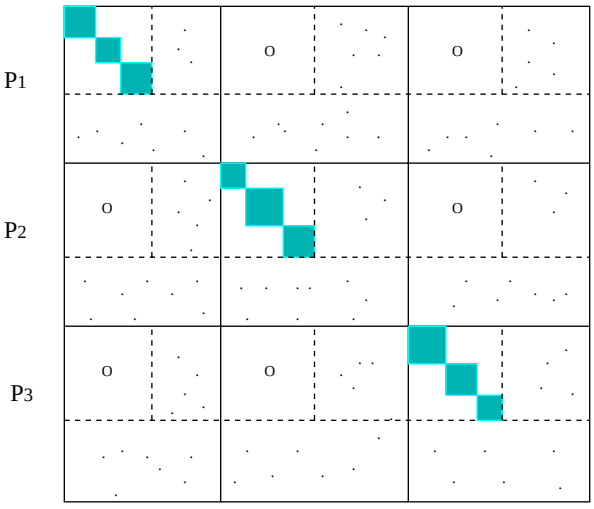

Fig. 3. A global BIS, empty circles are zero blocks.

nication cost is low in the construction phase, as the matrix is permuted locally. When we perform forward and backward solution procedures in applying the multilevel preconditioner constructed by the parallel BIS search algorithm, the permutations of some vectors between levels are local. However, since we only perform BIS search in the local matrix, the size of the BISs constructed may be smaller than those constructed by the sequential algorithm. It is more difficult to maintain a uniform block size for the BIS constructed by the parallel algorithm, due to the node removal procedure implemented to ensure global independence of the global BIS.

3.3. Induced parallel multilevel preconditioner

As it was discussed in [37], with the main factorization step shown in Eqs. (3) and (4), the preconditioning step $LU\, e = r$ can be implemented in three steps (with the sub-vector notations):

$$\begin{cases} \tilde{g} = g - E\, B^{-1}\, f, \\ S\, y = \tilde{g}, \\ B\, u = f - F\, y. \end{cases} \tag{6}$$

Here $e = (u, y)^T, r = (f, g)^T$. Thus the parallel multilevel preconditioning procedure can be written as follows, based on our matrix factorization (3) and the preconditioning step (6).

ALGORITHM 3.4. One application of the PBILUM preconditioner.

Forward elimination step:
1. For $l = 1, \ldots, (L-1)$.
2. Compute $\tilde{g}_{l,i} = g_{l,i} - E_{l,i}\,(B_{l,1}^{-1} f_{l,1}, B_{l,2}^{-1} f_{l,2}, \ldots, B_{l,m}^{-1} f_{l,m})^T$.
3. Permute the right-hand side $\tilde{g}_{l,i}$ to $(f_{l+1,i}, g_{l+1,i})^T$ for the lower level use.

Coarsest level solution:
4. Solve $S_{L,i}\, y_L = \tilde{g}_{L,i}$ iteratively.

Backward solution step:
5. For $l = (L-1), 1, -1$.
6. Perform matrix–vector product: $\tilde{f}_{l,i} = F_{l,i}\, y_l$.
7. Solve $(L_{B_{l,i}} U_{B_{l,i}})\, u_{l,i} = f_{l,i} - \tilde{f}_{l,i}$.
8. Permute vector $(u_{l,i}, y_{l,i})^T$ back to $y_{l-1,i}$ for the upper level use.

In Algorithm 3.4, L denotes the number of total levels, l is the level reference number. It is seen that at least two communication steps are needed at each level. One is for the matrix–vector product involving the submatrix $E_{l,i}$ and the vector $(B_{l,1}^{-1} f_{l,1}, B_{l,2}^{-1} f_{l,2}, \ldots, B_{l,m}^{-1} f_{l,m})^T$. Another is the matrix–vector product involving the submatrix $F_{l,i}$ and the vector y_l.

At each level, the matrix is reordered according to the BIS ordering, some permutations of the vectors are also needed in the preconditioning phase at Steps 3 and 8 in Algorithm 3.4. Based on Algorithm 3.3, these permutations are performed locally and independently by each processor without any communication. We denote this version of the parallel multilevel preconditioner as PBILUM_P. And PBILUM_Pd is the preconditioner PBILUM_P with the entry dropping rule. Analogously, the multilevel preconditioner associated with Algorithm 3.2 is denoted as PBILUM_S. Even though Algorithm 3.2 for constructing a BIS is conceptually easier than the parallel one, the preconditioning application phase is more complex to implement. In PBILUM_S, the permutations associated with the BIS ordering are not local. Moreover, unlike in PBILUM_P, the nodes of PBILUM_S in each processor change from level-to-level. This may increase communications. However, the BISs constructed by Algorithm 3.2 are larger and have better quality, and PBILUM_S achieves higher degree of load balancing at each level.

3.4. Multilevel Schur complement preconditioning strategy

If Algorithm 3.4 includes an iteration procedure at a certain level or at all levels, it may lead to a class of more accurate preconditioning algorithms, which involves inner Krylov iteration steps. The multilevel preconditioner solves the given Schur complement matrix using the lower level parts of PBILUM as the preconditioner. As it was discussed in [31,42,44], there are advantages to apply a Krylov iteration step to the reduced system. It can be beneficial to find more accurate solutions to the coarse level systems. However, the amount of the computations required to carry out each preconditioning step is increased. In general, the costs of both memory and computations are increased. Instead of employing Krylov iterations at every levels, we only have it for the first-level Schur complement system. This preconditioning strategy will be abbreviated as P_SchPre and

S_SchPre for PBILUM_P and PBILUM_S, respectively, following the notation of [42]. Based on Algorithm 3.4, the Schur complement preconditioning algorithm is described below:

ALGORITHM 3.5. PBILUM with Schur complement preconditioning.

Forward elimination step:

1. Solve $\tilde{g}_{1,i} = g_{1,i} - E_{1,i}\,(B_{1,1}^{-1} f_{1,1}, B_{1,2}^{-1} f_{1,2}, \ldots, B_{1,m}^{-1} f_{1,m})^T$.
2. Permute the right-hand side $\tilde{g}_{1,i}$ to $(f_{2,i}, g_{2,i})$.
3. Solve the reduced system $S_{1,i}\, y_1 = \tilde{g}_{1,i}$ to a given accuracy by using GMRES preconditioned by the lower levels of PBILUM.
4. call the Algorithm 3.4 from level 2 to L and get $y_{1,i}$.

Backward solution step:

5. Perform matrix–vector product: $\tilde{f}_{1,i} = F_{1,i}\, y_1$.
6. Solve $(L_{B_{1,i}} U_{B_{1,i}}) u_{1,i} = f_{1,i} - \tilde{f}_{1,i}$. Permute vector $(u_{1,i}, y_{1,i})^T$.

In a multilevel preconditioner, let S_1 be the first exact reduced system (exact Schur complement matrix). Since there is a Krylov iteration at this level, a matrix–vector multiplication and a preconditioning step are performed. Thus we need the matrix S_1 in one form or another. There are several strategies to deal with this matrix [42]. The simplest one is just to compute and keep the Schur complement matrix, and use it to do the iteration. Since the reduced system is formed based on the restricted Gaussian elimination, an approximate matrix $\bar{S}_1$ to S_1 is actually computed. Meurant [22] proposes to compute two approximate Schur complement matrices using different drop tolerances τ_1 and τ_2 ($\tau_1 < \tau_2$). The more accurate one (using τ_1) is denser and is used to perform the matrix–vector product. The other one (using τ_2) is more sparse and is used to construct the next level preconditioner. We denote this Schur complement preconditioning algorithm as P_SchPre_2 or S_SchPre_2. It needs extra memory to store the approximate matrix $\bar{S}_1$.

To avoid extra memory cost, one strategy proposed in [31,42] is considered. The idea comes from the expanded Eq. (4)

$$S_1 = \begin{pmatrix} C_{1,1} \\ C_{1,2} \\ \vdots \\ C_{1,m} \end{pmatrix} - \begin{pmatrix} E_{1,1} \\ E_{1,2} \\ \vdots \\ E_{1,m} \end{pmatrix} \begin{pmatrix} B_{1,1}^{-1} & & & \\ & B_{1,2}^{-1} & & \\ & & \ddots & \\ & & & B_{1,m}^{-1} \end{pmatrix} \times \begin{pmatrix} F_{1,1} \\ F_{1,2} \\ \vdots \\ F_{1,m} \end{pmatrix}. \quad (7)$$

The submatrices $C_{1,i}$, $E_{1,i}$, and $F_{1,i}$ are in the processor i. For $B_{1,i}^{-1}$, it can be replaced by a solution with $L_{B_{1,i}} U_{B_{1,i}}$, which comes from the restricted Gaussian elimination. If an inexact factorization is performed, we get an approximate matrix $\tilde{S}_1$ to S_1. Instead of performing explicit matrix–vector product $\tilde{S}_{1,i}\, y_1$ in processor i, we can perform this operation by computing $x_i = (L_{B_{1,i}} U_{B_{1,i}})^{-1} F_{1,i}\, y_1$, followed by computing $C_{1,i}\, y_1 - E_{1,i}\, x$ $(= \tilde{S}_{1,i}\, y_1)$. Here $x = (x_1, x_2, \ldots, x_m)^T$. This strategy can avoid storing the reduced system matrix. We denote this Schur complement preconditioning algorithm as P_SchPre_1 or S_SchPre_1. In general, the implicit matrix $\tilde{S}_1$ is more accurate than the explicit approximate matrix $\bar{S}_1$ even with the same drop tolerance and filling. It may lead to faster convergence. However, the computational cost in the preconditioning phase is high due to some matrix–vector products involved at each iteration step.

Implementation cost and algorithmic scalability of the preconditioning Schur complement and Schur complement preconditioning strategies are analyzed and discussed in detail in [42].

4. Diagonal thresholding and perturbation strategies

It is well known that the performance of an ILU preconditioner is dependent on the stability of the ILU factorization, which is affected by the ordering of the matrix [8,13,34,41,45]. Since our matrix ordering is based on the BISs, the stability of the ILU factorization of the blocks is important.

An unstable ILU factorization may be resulted from factoring a matrix with small or zero diagonals [8,13]. In order to improve stability, it is beneficial to only include those nodes with large diagonal entries in the BIS [34,41,45].

There are a few heuristic strategies to implement the diagonal thresholding strategy. A simple approach based on a prescribed diagonal thresholding value is analyzed and implemented in [34]. In [41], a dual reordering strategy is proposed to divide the fine and coarse level nodes. The diagonal thresholding strategies proposed in [35,41,45] are automatic and do not require an input parameter.

In [11,31], a weight array of each row is computed and is used to determine the qualification of each row to be included in the BIS. To account for the vast difference in the matrix properties, we give a carefully designed strategy to compute the relative diagonal dominance measure of each row and a diagonal thresholding criterion β. They are used to determine when it is acceptable to include a node into the BIS. The measure $\omega(i)$ for the ith row is computed as $\omega(i) = |a_{ii}| / \max_{j \neq i} |a_{ij}|$.

In many cases, the number of nodes in the BIS is less or equal to a half of the total unknowns. Thus we consider to use the median diagonal dominance measure as our diagonal thresholding criterion β. The computational cost to compute the measure array and the criterion is $O(nnz)$, where

nnz is the number of nonzeros of the matrix at a given level. Since the matrices are sparse, $nnz \ll n^2$. Alternatively, we can combine the average ω_{avg}, maximum $\omega_{\max}$ and minimum $\omega_{\min}$ weights together to form a diagonal thresholding criterion β. This strategy is described in Algorithm 4.1.

ALGORITHM 4.1. Computing a diagonal thresholding criterion.

1. **For** $i = 1, \ldots, n$, **Do**
2. **If** ($|a_{ii}| \neq 0$ and $\max_{j \neq i} |a_{ij}| = 0$), **Then**
3. $\omega(i) = 1.0$.
4. **Else**
5. $\omega(i) = |a_{ii}| / \max_{j \neq i} |a_{ij}|$.
6. **EndIf**
7. **EndFor**
8. Compute $\omega_{\min}, \omega_{\max}$ and ω_{ave}.
9. $\beta = \min(\omega_{\text{ave}}, (\omega_{\min} + \omega_{\max})/2, 0.1)$.

We also employ a diagonal perturbation strategy on the Schur complement matrix at the coarsest level before performing a block diagonal ILU factorization. The measure of the rows of the block diagonal matrix is computed in the same way as described above. Since this is the last level, it may have some zero or small diagonal entries. We remedy this kind of diagonal entries with a perturbation strategy as in Algorithm 4.2.

ALGORITHM 4.2. Diagonal perturbation with a given criterion α.

1. Compute the diagonal dominance measure $\omega(i)$ by using the Algorithm 4.1.
2. Compute a maximum value of the each row, i.e., $v(i) = \max_{j \neq i} |a_{ij}|$, $i = 1, \ldots, n$.
3. Set $t = (\max(v(1), \ldots, v(n)) + \min(v(1), \ldots, v(n)))/2$.
4. **For** $i = 1, \ldots, n$,
5. **If** $(w(i) < \alpha)$, **Then**
6. $|a_{ii}| = \alpha \times \min(t, v(i))$.
7. **EndIf**
8. **EndFor**

The result of Algorithm 4.2 is that the values of the small diagonal entries are altered, so that a stable ILU factorization may be computed from a perturbed matrix at the last level.

5. Numerical experiments

In the numerical experiments, we compare the performance of the different variants of the PBILUM preconditioners described in the previous sections. We also test the performance of one of a parallel preconditioner, add_arms, in pARMS [20]. Preconditioner add_arms is an Additive Schwarz procedure in which ARMS is used as a preconditioner for solving the local systems [20]. All matrices are considered general sparse and any available structures are not exploited. The right-hand side vector is generated by assuming that the solution is a vector of all ones and the initial guess is a zero vector. We first show the performance of these preconditioners for solving some 2D and 3D convection diffusion problems. Then we test the preconditioner scalability using the 2D and 3D problems. Later some FIDAP matrices from [14] [1] are solved to show the robustness of PBILUM. Finally, we solve a sparse matrix from the numerical simulation of a laminar diffusion flame to compare the performance of PBILUM with different levels. Our main objective is to demonstrate that truly multilevel preconditioners are more robust and sometimes are necessary to solve certain difficult problems.

We use a flexible variant of restarted GMRES(30) (FGMRES) [25,29] in a parallel version from P_SPARSLIB [27]. The exceptions are indicated explicitly. The preconditioner for the coarsest level system is the block Jacobi, which is the same as in PBILU2 [37]. The (outer iteration) computations are terminated when the 2-norm of the residual vector is reduced by a factor of 10^8. The inner iterations on solving the coarsest level system is assumed to have satisfied the convergence test if the 2-norm of the coarsest level residual vector is reduced by a factor of 10^2 or the maximum number of 5 iterations for PBILUM and 2 iterations for add_arms at the coarsest level is reached.

For all PBILUM variants, the first-level BIS from the original matrix is constructed and the matrix is permuted in a single processor before it is distributed to the other processors. The CPU time for this step is included in the preconditioner construction time.

In all tables containing numerical results, "n" is the order of the matrix, "n_c" is the order of the coarsest level system. "np" denotes the number of processors. "iter" is the number of preconditioned FGMRES iterations. "level" is the number of the levels used in PBILUM and add_arms. "sparsity" denotes the sparsity ratio, i.e., the density of the preconditioner with respect to the original matrix [33]. "T_{total}" is the total CPU time in seconds, i.e., the sum of "T_{setup}" and "T_{iter}", which are the CPU times for the preconditioner construction and preconditioned iteration, respectively. "τ" and "p" are the two parameters used in the ILU factorization as the drop tolerance and the fill-in parameter, similar to those used in ILUT [26] and BILUTM [33]. "α" is used in the diagonal perturbation algorithm. σ is a predetermined thresholding parameter used in the entry dropping rule. In our tests, $\sigma = 0.5$. "ε" is a drop tolerance used for sparsifying the reduced systems for constructing the next level preconditioner. Unless specified explicitly, $\varepsilon = 10\tau$, and $bsize = 100$.

The computations are carried out on a 32 processor subcomplex of an HP superdome (supercluster) at the Univer-

[1] These matrices are available online from the Matrix Market of the National Institute of Standards and Technology at `http://math.nist.gov/MatrixMarket/`

sity of Kentucky. Each of the Titanium processors runs at 1.5 GHz with 2 GB local memory. They are connected by a high speed, low latency HP hyperfabric internal interconnect. We use MPI for interprocessor communications and the code is written in Fortran 90 in double precision.

5.1. 2D and 3D convection diffusion problems

The 2D convection diffusion problems are governed by the following partial differential equation:

$$\Delta u + Re(\exp(xy-1)u_x - \exp(-xy)u_y) = 0$$

defined on a unit square, and the 3D problems by

$$\begin{aligned}\Delta u + Re(&x(x-1)(1-2y)(1-2z)u_x\\ &+y(y-1)(1-2z)(1-2x)u_y\\ &+z(z-1)(1-2x)(1-2y)u_z) = 0\end{aligned}$$

defined on a unit cube. Here, Re is the Reynolds number, which represents the strength of the convection against diffusion. These equations are important components of the Navier–Stokes equations used in computational fluid dynamics.

The 2D problems are discretized by using the 9-point fourth-order compact difference scheme [17], and the resulting sparse matrices are referred to as the 9-POINT matrices. The 3D problems are discretized by using either the standard 7-point central difference scheme or the 19-point fourth-order compact difference scheme [40], and the resulting sparse matrices are referred to as the 7-POINT or the 19-POINT matrices, respectively.

We first generate a series of 3D 19-POINT matrices with $Re = 0.0$, by increasing the problem size from $n = 20^3$ to 100^3. The parameters are chosen as $\tau_1 = 10^{-2}, \tau_2 = 10^{-1}$, $p = 30$, $\varepsilon = 10^{-1}$. Here, we use τ_1 as a drop tolerance in the restricted Gaussian elimination procedure to compute an approximate Schur complement $\bar{S}_1$. It will be stored and used in the Schur complement preconditioning strategy for the SchPre_2 preconditioners. Then we use τ_2 to sparsify $\bar{S}_1$ for the next level use. We compare the performance of different variants of PBILUM for solving matrix problems of different size. A total of 4 processors are used in this test. Fig. 4 shows that the number of iterations changes as the matrix size increases. We see that the preconditioners with the Schur complement preconditioning strategies (i.e., P_SchPre_1 and P_SchPre_2) converge in a smaller number of iterations, and P_SchPre_1 is the fastest in terms of the number of iterations to converge. The sparsity ratios for those preconditioners are 1.12 for the 2-level PBILUM_P, 0.91 for the 4-level PBILUM_P and 4-level P_SchPre_1, and 2.21 for the 4-level P_SchPre_2. This test demonstrates that the 4-level PBILUM preconditioners (except the 4-level P_SchPre_2) are usually better than the 2-level ones in terms of the convergence rate and the memory cost. We also notice that the 4-level P_SchPre_2 uses more memory space due to the storage of the approximate Schur complement matrix at the first level.

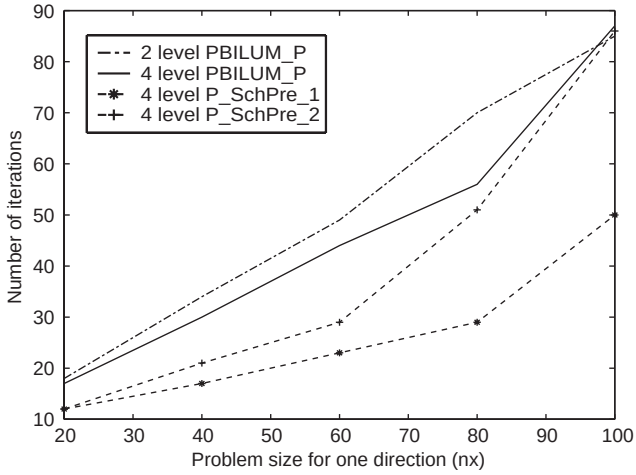

Fig. 4. Comparison of different variants of PBILUM for solving a series of 19-POINT matrices with $Re = 0.0$ ($\tau = 10^{-2}$, $p = 30$, and $\varepsilon = 10^{-1}$).

Then we use 6 processors to solve a series of 2D 9-POINT matrices with $Re = 1.0$. The PBILUM parameters are fixed as $\tau = 10^{-3}$, $p = 50$. The problem size increases from $n = 200^2$ to $n = 1000^2$. The test results are listed in Table 1. "-" means there is no last reduced system size computed.

A few comments on the results in Table 1 are in order. In our experiments, we find that PBILUM with the Schur complement preconditioning strategy usually achieves faster convergence rate than the other strategies do. Since it involves many matrix–vector product operations in the first-level Schur complement iterations, each outer FGMRES iteration needs more CPU time. However, in this test, the number of iterations for the methods with the Schur complement preconditioning strategy is much less than that of other preconditioners, they are seen to be fastest in terms of iteration time. Additive Schwarz does not show good performance in this test.

We now compare the 2-level PBILUM and the 4-level PBILUM. Since the size of the last reduced system of the 4-level PBILUM is much smaller than that of the 2-level one, as the problem size increases, the 4-level PBILUM converges faster and uses less CPU time and memory than the 2-level one does.

For solving the 2D matrices, we see that PBILUM_P (with the parallel BIS search) is faster than PBILUM_S (with the sequential BIS search). The reason is that the size of the last reduced system from PBILUM_P is only slightly larger than that from PBILUM_S. Thus the preconditioner PBILUM_P takes advantage of its high-level parallelism in the solution phase.

Usually, the sparsity ratio of the multilevel PBILUM is smaller than that of the 2-level PBILUM due to its use of a larger drop tolerance $\varepsilon > \tau$ in sparsifying the reduced system for constructing the next level factorization.

Table 1
Solving the 9-POINT matrices with $Re = 1.0$

n	Preconditioner	Level	Iter	T_{total}	Sparsity	n_c
40,000	PBILUM_S	2	25	1.10	3.44	7100
	PBILUM_S	4	27	1.13	3.26	1500
	S_SchPre_1	4	12	1.41	3.26	1500
	PBILUM_P	4	26	2.45	1.06	2849
	P_SchPre_1	4	12	2.98	1.19	2849
	add_arms	4	129	15.74	3.46	—
160,000	PBILUM_S	2	57	11.98	3.53	28,600
	PBILUM_S	4	58	11.84	3.27	6500
	S_SchPre_1	4	20	12.25	3.27	6500
	PBILUM_P	4	53	10.87	3.28	9474
	P_SchPre_1	4	20	11.15	3.28	9474
	add_arms	4	175	137.61	3.46	—
360,000	PBILUM_S	2	94	52.83	3.60	63,800
	PBILUM_S	4	98	45.29	3.29	15,000
	S_SchPre_1	4	27	39.69	3.29	15,000
	PBILUM_P	4	98	44.13	3.30	20,001
	P_SchPre_1	4	27	37.82	3.30	20,001
	add_arms	4	247	489.58	3.46	—
640,000	PBILUM_S	2	133	142.68	3.60	112,700
	PBILUM_S	4	136	111.79	3.30	26,500
	S_SchPre_1	4	37	95.61	3.30	26,500
	PBILUM_P	4	135	107.91	3.32	22,933
	P_SchPre_1	4	36	88.82	3.32	22,933
	add_arms	4	358	1333.15	3.48	—
1,000,000	PBILUM_S	2	166	285.87	3.63	171,900
	PBILUM_S	4	175	224.33	3.35	41,900
	S_SchPre_1	4	49	195.15	3.35	41,900
	PBILUM_P	4	176	224.93	3.35	46,685
	P_SchPre_1	4	47	183.19	3.35	46,685
	add_arms	4	401	2353.74	3.46	—

5.2. Scalability tests

In Fig. 5, we solve a 3D 7-POINT matrix with $Re = 1000.0$ by using different variants of PBILUM with different number of processors. We fix the problem size as $n = 1,000,000$, $nnz = 6,490,000$. The PBILUM parameters are chosen as $\tau = 10^{-2}$, $p = 20$, $\tau = 10^{-2}$, and $p = 30$ for add_arms. We see that the 4-level PBILUM performs better than the 2-level counterpart. The preconditioner add_arms uses more CPU time and needs a larger number of iterations to converge.

Table 2 gives more detailed test data, corresponding to the test cases reported in Fig. 5. Since the test results for S_SchPre_1 are almost the same as P_SchPre_1, we only list the results for S_SchPre_1. In this test, these methods do not show fast convergence rate in terms of the iteration time. The number of iterations for these two methods are not small enough, compared with that of the other methods, so they take more CPU time in the solution phase due to their matrix–vector product operations at each iteration. Due to limited memory, we cannot run add_arms in one processor.

It is clear from Table 2 that, for the 4-level variants, PBILUM_P takes less time than PBILUM_S to construct the preconditioner. But PBILUM_S constructs larger BISs. The difference is not significant when the number of processors is small, and PBILUM_P is faster in terms of overall CPU time. However, when the number of processors is large and

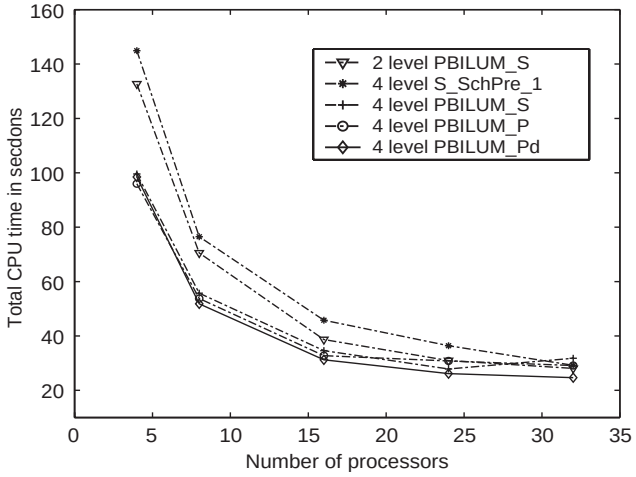

Fig. 5. Total CPU time comparison of different variants of PBILUM for solving a 3D 7-POINT matrix with $Re = 1000.0$ and $n = 1,000,000$.

Table 2
Performance data for solving a 7-POINT matrix, corresponding to Fig. 5

np	Preconditioner	Level	Iter	T^{S}_{seq}/T^{P}_{dis}	T_{con}	T_{setup}	T_{iter}	Sparsity	n_c
1	PBILUM	4	70	4.96	30.25	35.21	307.93	2.08	74,400
4	PBILUM_S	2	62	3.27	8.82	12.09	124.14	2.43	330,700
	S_SchPre_1	4	26	6.11	11.19	17.30	127.71	2.11	76,300
	PBILUM_S	4	70	6.09	11.71	17.80	84.49	2.11	76,300
	PBILUM_P	4	69	4.97	9.30	14.28	82.71	2.08	96,631
	PBILUM_Pd	4	73	4.85	8.37	13.22	85.21	2.08	77,513
	add_arms	4	79			81.42	546.25	2.68	—
8	PBILUM_S	2	62	3.56	5.41	8.97	61.88	2.42	330,700
	S_SchPre_1	4	26	6.25	8.17	14.42	62.11	2.11	77,500
	PBILUM_S	4	70	6.29	8.33	14.62	40.87	2.11	77,500
	PBILUM_P	4	70	4.97	5.56	10.53	42.75	2.08	124,759
	PBILUM_Pd	4	74	4.38	5.22	9.60	41.76	2.08	84,350
	add_arms	4	104			64.95	346.02	2.52	—
16	PBILUM_S	2	59	4.24	3.87	8.11	32.37	2.38	330,700
	S_SchPre_1	4	26	8.07	5.86	13.93	32.03	2.13	78,200
	PBILUM_S	4	71	8.25	5.74	13.99	20.15	2.11	78,200
	PBILUM_P	4	69	5.34	3.87	9.21	23.79	2.09	175,270
	PBILUM_Pd	4	76	5.21	3.28	8.49	22.25	2.08	91,697
	add_arms	4	104			54.27	165.84	2.52	—
24	PBILUM_S	2	59	4.87	3.40	8.27	24.61	2.36	330,700
	S_SchPre_1	4	26	9.87	5.08	14.95	21.75	2.14	80,200
	PBILUM_S	4	68	9.92	5.12	15.04	13.29	2.14	80,200
	PBILUM_P	4	68	5.97	3.08	9.05	21.03	2.08	228,496
	PBILUM_Pd	4	80	5.88	2.96	8.84	17.80	2.08	105,325
	add_arms	4	109			59.42	109.88	2.60	—
32	PBILUM_S	2	58	5.63	3.76	9.39	24.36	2.36	330,700
	S_SchPre_1	4	26	11.04	4.41	15.45	16.60	2.14	80,400
	PBILUM_S	4	70	11.26	4.50	15.76	10.10	2.14	80,400
	PBILUM_P	4	70	6.63	2.32	8.95	20.47	2.06	251,788
	PBILUM_Pd	4	84	6.63	2.18	8.81	16.02	2.08	114,999
	add_arms	4	117			59.78	73.86	2.88	—

the problem size in each processor is small, PBILUM_S may be slightly faster in terms of the total CPU time. We also find that PBILUM_Pd uses less iteration time than PBILUM_P does due to its smaller size of the last reduced matrix, even though it may need a larger number of iterations to converge. This test shows that it is worth putting our future efforts on finding large BIS at each level. A good property of PBILUM is that its convergence rate is almost independent of the number of the processors employed for this set of tests.

In this test, the setup time in the preconditioner construction phase does not scale well. For PBILUM_S, some sequential operations, such as BIS search, matrix permutation and matrix distribution at every level, are involved in this phase. We use T^{S}_{seq} to denote the total computational time of BIS search and matrix permutation for all level in one processor. The BIS search step may cause computational bottleneck due to the large amount of data processed in one processor and the other processors have nothing to do while waiting. Communication bottleneck may also occur at the matrix distribution step, in which the master processor sends and receives data from the other processors. Many other processors may try to send data to the master processor at the same time. In this test, the communication time at this step does not decrease, it is about 2.60 s, as the number of processor increases. For PBILUM_P, the matrix is read in one processor and then distributed to other processor at the first level, (we use T^{P}_{dis} to represent the timings of this step), the serial step decreases the overall performance in the preconditioner construction phase. This step can be eliminated by generating submatrix in each processor. As we see in Table 2, if this part of the timing is removed, the preconditioning construction time, T_{con}, scales well. The important information in Table 2 is that PBILUM_P is several times faster than PBILUM_S in the procedure of BIS search and matrix permutation, see the timing T^{S}_{seq}/T^{P}_{dis} in the column of Table 2.

Since we use some collective communication calls, such as All-to-All broadcast or personalized communication, in the preconditioner construction phase, they are likely to cause bottlenecks. Because the communication time is slightly increased as the number of processors increased.

Comparing the test results listed in Table 1 with that in Table 2, for the same number of processors, the sizes of the last reduced system for the 4-level PBILUM_S and PBILUM_P are almost the same for the 9-POINT matrix, but are quite different for the 7-POINT matrix. The reason is that the 9-POINT matrix has a strong block diagonal structure, but the 7-POINT matrix has a block off-diagonal structure. With a strong block diagonal structure, PBILUM_P can find larger size of BIS than that without such kind of structure. PBILUM_P can take advantage of its high parallelism and can have better overall parallel performance than PBILUM_S.

We then do algorithmic scalability tests by generating a series of 2D 9-POINT matrices with $Re = 10.0$ and 3D 19-POINT matrices with $Re = 100.0$. The problem size is approximately fixed as 200×200 and $30 \times 30 \times 30$ in each processor for the 2D and 3D problems, respectively. We use square and cubic processor grids of increasing size for the 2D and 3D problem tests. Since the maximum number of processors can be used exclusively is 32 on this parallel computer, we use 2^2, 3^2, 4^2, 5^2 and 2^3, 3^3 as the number of processors for the 2D problem and the 3D problem respectively. The left panel of Fig. 6 shows that the number of iterations changes as the number of processors increases for solving the 2D problem with $p = 50$, $\tau = 10^{-3}$, and GMRES(50). There is almost linear behavior observed in terms of the number of iterations and the number of processors in Fig. 6. But compared with the block Jacobi-typed method, add_arms, the rate of increase is quite different. For example, if 4 processors are used, the number of outer iterations of add_arms is 176, the total CPU time is 156.12 s. For 9 processors, there are 308 iterations in 287 s of total CPU time. For 16 processors, it takes 404 iterations and 392 s in total CPU time. Because add_arms runs out of memory space for 25 processors, we do not add the comparison information of add_arms in Fig. 6. From this test, we confirm that S_SchPre_1 and P_SchPre_1 converge much faster than the other preconditioners in terms of the number of iterations. The right panel of Fig. 6 compares the total CPU time used by the PBILUM preconditioners. In this test, the 4-level PBILUM_P has slightly better performance in terms of the solution time due to its efficient preconditioner construction algorithm.

For the 3D problem, the parameters used in this test are the same as those used in the 7-POINT matrix test. The test results are listed in Table 3. Again, due to the smaller size of the last reduced matrix, PBILUM_Pd shows better convergence rate in terms of total CPU time. The Schur complement preconditioning strategies do not perform very well in this set of tests.

5.3. FIDAP matrices

This set of test matrices were extracted from the test problems provided in the FIDAP package [14]. As many of these matrices have small or zero diagonals, they are difficult to solve using standard ILU preconditioners. We solve 28 of the largest FIDAP matrices ($n > 2000$). The descriptions of these matrices can be found in [41]. For this set of tests, we use FGMRES(50) as our iterative solver. We use the diagonal perturbation strategy with the parameter α for the last level reduced system. The range of the α values is from 10^{-2} to 10^{-4}. Before the preconditioner construction, most of the FIDAP matrices are scaled. We first make the 2-norm of each column of a matrix to be unity, then we scale the rows so that the 2-norm of each row is unity. Compared with the previous tested problems, the FIDAP matrices are small. We only use 2 processors in our tests.

As we know, in PBILUM, the last reduced system is solved by a block Jacobi-preconditioned GMRES(5) method. The smaller size of the last reduced matrix is, the better preconditioner we will have. To have smaller size of the last reduced matrix, one heuristic way is to use more levels. Usually, multilevel preconditioners may have better convergence rate than 2-level preconditioners.

We first test the 2-level PBILUM. If a matrix cannot be solved or needs a lot of iterations to solve, we increase the number of levels until it is solved without many iterations, or until we reach level 6.

For the 2-level preconditioners, PBILUM_P and PBILUM_S are the same. For the multilevel cases, the test results are almost the same for both preconditioners with only two processors. We only list the test results of the 28 FIDAP matrices for PBILUM_S in Table 4. We find that our preconditioners can solve most of the 28 FIDAP matrices with a low sparsity ratio. The robustness of our parallel preconditioners is impressive, since the previous best result that we are aware of is that 27 of the largest FIDAP matrices are solved by a sequential multilevel preconditioner with a dual reordering strategy [41]. We point out that 13 FIDAP matrices can be solved more efficiently by using the multilevel PBILUM than by the 2-level one.

5.4. Flame matrix

We generate a sparse matrix from the 2D numerical simulation of an axisymmetric laminar diffusion flame [10]. We use the vorticity–velocity formulation of the Navier–Stokes equations, which is described in detail in [15]. Let us define the velocity vector $v = (v_r, v_z)$ with the radial (v_r) and the axial (v_z) components. The normal component of the vorticity is $w = \frac{\partial}{\partial z} v_r - \frac{\partial}{\partial z} v_z$. The vorticity transport equation is formed by taking the curl of the momentum equations, which eliminates the partial derivatives of the pressure field. A Laplace equation is obtained for each velocity component by taking the gradient of w and using the continuity equation.

Let ρ be the density, μ the viscosity, g the gravity vector, div(v) the cylindrical divergence of the velocity vector, S the conserved scalar, and D a diffusion coefficient. The

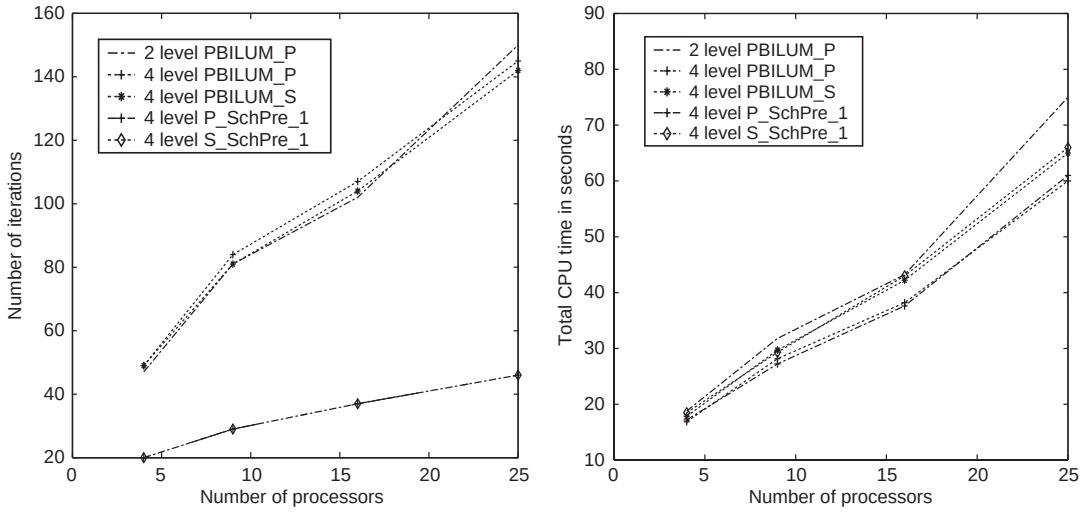

Fig. 6. Convergence behavior of the PBILUM preconditioners for solving the 9-POINT matrices with a fixed subproblem size in each processor. Left panel: the number of iterations versus the number of processors. Right panel: the total CPU time versus the number of processors.

Table 3
3D 19-POINT matrices with a fixed subproblem in each processor

Preconditioner	Level	np = 8			np = 27		
		Iter	T_{total}	Sparsity	Iter	T_{total}	Sparsity
PBILUM_S	2	46	12.34	1.14	68	38.87	1.17
PBILUM_P	4	46	10.53	0.92	75	33.35	0.92
PBILUM_S	4	45	11.15	0.92	75	33.69	0.93
P_SchPre_1	4	23	15.93	0.92	45	62.55	0.92
S_SchPre_1	4	23	16.20	0.92	46	46.78	0.93
PBILUM_Pd	4	46	10.47	0.91	75	31.60	0.92

components of $\overline{\nabla}\beta$ are $\left(\frac{\partial}{\partial z}\beta, -\frac{\partial}{\partial r}\beta\right)$. The governing system of partial differential equations (PDEs) for the laminar diffusion flames can be written as the following [10]:

$$\frac{\partial^2 v_r}{\partial r^2} + \frac{\partial^2 v_r}{\partial z^2} = \frac{\partial w}{\partial z} - \frac{1}{r}\frac{\partial v_r}{\partial r} + \frac{v_r}{r^2} - \frac{\partial}{\partial r}\left(\frac{v\cdot\nabla\rho}{\rho}\right),$$

(Radial velocity Eq.)

$$\frac{\partial^2 v_z}{\partial r^2} + \frac{\partial^2 v_z}{\partial z^2} = -\frac{\partial w}{\partial r} - \frac{1}{r}\frac{\partial v_r}{\partial z} - \frac{\partial}{\partial z}\left(\frac{v\cdot\nabla\rho}{\rho}\right),$$

(Axial velocity Eq.)

$$\frac{\partial^2 \mu w}{\partial r^2} + \frac{\partial^2 \mu w}{\partial z^2} + \frac{\partial}{\partial r}\left(\frac{\mu w}{r}\right) = \rho v_r \frac{\partial w}{\partial r} + \rho v_z \frac{\partial w}{\partial z} - \frac{\rho v_r}{r} w + \overline{\nabla}\rho\cdot\nabla\frac{v^2}{2} - \overline{\nabla}\rho\cdot g + 2\left(\overline{\nabla}(\operatorname{div}(v))\cdot\nabla\mu - \nabla v_r\cdot\overline{\nabla}\frac{\partial\mu}{\partial r} - \nabla v_z\cdot\overline{\nabla}\frac{\partial\mu}{\partial z}\right),$$

(Vorticity Eq.)

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\rho D\frac{\partial S}{\partial r}\right) + \frac{\partial}{\partial z}\left(\rho D\frac{\partial S}{\partial z}\right) = \rho v_r\frac{\partial S}{\partial r} + \rho v_z\frac{\partial S}{\partial z}.$$

(Shvab–Zeldovich Eq.)

The temperature and major species profiles are recovered from the conserved scalar S as detailed, for instance, in [38].

The matrix that we generate is the Jacobi matrix associated with the nonlinear Newton iteration and has $n = 21,060$ and $nnz = 382,664$. The matrix has a 4×4 block structure, corresponding to the number of coupled governing PDEs. The 2D nonuniform tensor grid is covered with a mesh of 65×81. Mixed 9-point central and upwind difference schemes are used so that each row of the matrix has at most 36 nonzeros. The matrix is scaled as we did for the FIDAP matrices.

We solve the Flame matrix with 2 processors to show the performance difference of PBILUM with different levels. The sparsity ratio for PBILUM with different levels is almost the same, which is around 2.5. In this test, we use $\varepsilon = \tau$.

Except for the 2-level case, we find that the 4-level and 8-level implementations of both PBILUM_S and PBILUM_P achieve almost the same results. However, for the 6-level case, PBILUM_P has almost the same results as for the 8-level case, thus it converges faster than the 6-level PBILUM_S. For simplicity, we only plot the test results of PBILUM_S in Fig. 7. We can see

Table 4
Solving the FIDAP matrices using PBILUM_S and 2 processors

Matrices	*n*	*nnz*	Level	Iter	Sparsity	*bsize*	τ	*p*
FIDAP008	3096	106,302	3	17	3.50	100	10^{-8}	300
FIDAP009	3363	99,397	4	22	4.46	100	10^{-10}	300
FIDAP010	2410	54,816	3	8	2.87	100	10^{-8}	300
FIDAP012	3973	80,151	2	17	2.59	100	10^{-3}	50
FIDAP013	2568	75,628	4	4	4.00	50	10^{-12}	300
FIDAP014	3251	66,647	3	24	4.00	50	10^{-8}	100
FIDAP015	6867	9,6421	2	9	3.50	100	10^{-12}	200
FIDAP018	5773	69,335	2	8	3.92	100	10^{-8}	300
FIDAP019	12,005	259,863	4	5	3.60	100	10^{-10}	500
FIDAP020	2203	69,579	2	27	1.78	50	10^{-3}	50
FIDAP024	2283	48,733	4	26	3.13	50	10^{-5}	50
FIDAP028	2603	77,653	2	41	1.90	100	10^{-3}	50
FIDAP029	2870	23,754	2	5	1.53	100	10^{-3}	50
FIDAP031	3909	115,299	2	32	3.45	200	10^{-8}	400
FIDAP035	19,716	218,308	2	23	3.41	100	10^{-8}	200
FIDAP036	3079	53,851	4	18	2.73	100	10^{-4}	50
FIDAP037	3565	67,591	4	4	1.38	100	10^{-4}	50
FIDAP040	7740	456,226	2	57	2.40	500	10^{-5}	100
FIDAPM03	2532	50,380	2	27	2.80	500	10^{-4}	50
FIDAPM08	3876	103,076	3	20	2.99	100	10^{-4}	100
FIDAPM09	4683	95,053	2	23	8.70	700	10^{-5}	200
FIDAPM10	3046	53,842	2	26	2.30	700	10^{-5}	30
FIDAPM11	22,294	623,554	4	181	8.99	100	10^{-10}	200
FIDAPM13	3549	71,975	2	139	2.19	700	10^{-5}	30
FIDAPM15	9287	98,519	4	89	2.92	100	10^{-5}	30
FIDAPM29	13,668	186,294	2	82	5.40	500	10^{-5}	70
FIDAPM33	2353	23,765	6	26	4.10	30	10^{-8}	80
FIDAPM37	9152	765,944	2	87	3.77	500	10^{-5}	800

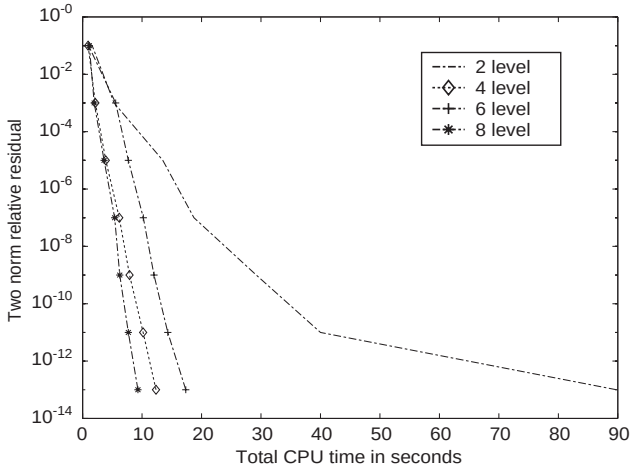

Fig. 7. Performance comparison of PBILUM_S with different levels for solving the Flame matrix ($\tau = 10^{-6}$, $p = 40$).

that the multilevel PBILUM converges faster than the 2-level one.

6. Concluding remarks

We developed two algorithms for constructing BISs from a distributed sparse matrix. Based on the new distributed BIS algorithms, we devised a class of truly parallel multilevel block incomplete LU preconditioning techniques (PBILUM) for solving general sparse linear systems. We discussed a few variants of PBILUM with different implementation strategies. We tested and compared different PBILUM variants and implementations on a few sets of sparse matrices arising from different applications. We showed that the multilevel PBILUM is more robust and converges faster than the 2-level PBILUM for solving large-scale problems. Thus, the efforts put in developing the more complex truly parallel multilevel ILU preconditioners are well paid.

In particular, we showed that the algorithm to construct the BISs in parallel is faster than the one to search independent blocks in a serial mode. For very large-scale problems in which each processor has a large local subproblem, PBILUM_P is shown to be better than PBILUM_S. However, when the number of processors is large and the local subproblem is small, PBILUM_P tends to generate a larger last level reduced system and takes longer time in the solution process. Even in these cases, their total computational costs are comparable, as PBILUM_P takes less CPU time in the preconditioner construction phase. PBILUM_P only searches BIS in the local matrix, and has a smaller BIS. It still achieves almost the same convergence rate as PBILUM_S which is based on a sequential BIS search on the whole matrix.

We also tested the Schur complement preconditioning option with the PBILUM preconditioners. Our results show that it is sometimes beneficial to solve the first level (approximate) Schur complement matrix to a certain accuracy. This is especially true if the problem size is very large, as this strategy tends to have better algorithmic scalability.

Our PBILUM preconditioners do not demonstrate optimal scalability, similar to all existing parallel multilevel preconditioners that aim at solving general sparse linear systems. We note that there is room for improvements. First, we did not adjust the parameters in our scalability tests. It is possible to get better test results by choosing different parameters for different problem sizes. We did not attempt that for the reason of maintaining the general purpose of our preconditioners.

Another direction of improvements may come from the strategies used in the parallel BIS construction algorithm. If we can design better and more inexpensive strategies to avoid the double node removal problem in a distributed environment, we will have larger BISs at all levels. This gain may be significant as we see that the inner iteration process to solve the last level reduced system of large size may slow down the overall computational process.

References

[1] O. Axelsson, P.S. Vassilevski, Algebraic multilevel preconditioning methods. I, Numer. Math. 56 (2–3) (1989) 157–177.

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

[3] E.F.F. Botta, F.W. Wubs, Matrix renumbering ILU: an effective algebraic multilevel ILU preconditioner for sparse matrices, SIAM J. Matrix Anal. Appl. 20 (4) (1999) 1007–1026.

[4] D. Braess, Towards algebraic multigrid for elliptic problems of second order, Computing 55 (4) (1995) 379–393.

[5] A. Brandt, Multi-level adaptive solutions to boundary-value problems, Math. Comput. 31 (138) (1977) 333–390.

[6] W.L. Briggs, V.E. Henson, S.F. McCormick, A Multigrid Tutorial, second ed., SIAM, Philadelphia, PA, 2000.

[7] P.N. Brown, R.D. Falgout, J.E. Jones, Semicoarsening multigrid on distributed memory machines, SIAM J. Sci. Comput. 21 (5) (2000) 1823–1834.

[8] E. Chow, Y. Saad, Experimental study of ILU preconditioners for indefinite matrices, J. Comput. Appl. Math. 86 (2) (1997) 387–414.

[9] A.J. Cleary, R.D. Falgout, V.E. Henson, J.E. Jones, T.A. Manteuffel, S.F. McCormick, G.N. Miranda, J.W. Ruge, Robustness and scalability of algebraic multigrid, SIAM J. Sci. Comput. 21 (5) (1998) 1886–1908.

[10] C.C. Douglas, A. Ern, M.D. Smooke, Multigrid solution of flame sheet problems on serial and parallel computers, Parallel Algorithms Appl. 10 (1997) 225–236.

[11] I. Duff, J. Koster, On algorithms for permuting large entries to the diagonal of a sparse matrix, SIAM J. Matrix Anal. Appl. 22 (4) (2001) 973–996.

[12] V. Eijkhout, T.F. Chan, ParPre: a parallel preconditioners package reference manual for version 2.0.17, Technical Report CAM 97-24, Department of Mathematics, UCLA, Los Angeles, CA, 1997.

[13] H.C. Elman, A stability analysis of incomplete LU factorizations, Math. Comput. 47 (175) (1986) 191–217.

[14] M. Engelman, FIDAP: Examples Manual, Revision 6.0, Technical Report, Fluid Dynamics International, Evanston, IL, 1991.

[15] A. Ern, M.D. Smooke, Vorticity-velocity formulation for three-dimensional steady compressible flows, J. Comput. Phys. 105 (1993) 58–71.

[16] M. Griebel, Parallel domain-oriented multilevel methods, SIAM J. Sci. Comput. 16 (5) (1995) 1105–1125.

[17] M.M. Gupta, R.P. Manohar, J.W. Stephenson, A single cell high order scheme for the convection–diffusion equation with variable coefficients, Internat. J. Numer. Methods Fluids 4 (7) (1984) 641–651.

[18] G. Karypis, V. Kumar, Parallel threshold-based ILU factorization, Technical Report 96-061, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1996.

[19] G. Karypis, V. Kumar, Parallel multilevel *k*-way partitioning scheme for irregular graphs, SIAM Rev. 41 (2) (1999) 278–300.

[20] Z. Li, Y. Saad, M. Sosonkina, pARMS: a parallel version of the algebraic recursive multilevel solver, Technical Report UMSI 2002-100, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 2001.

[21] M. Luby, A simple parallel algorithm for the maximal independent set problem, SIAM J. Comput. 15 (4) (1986) 1036–1053.

[22] G. Meurant, A multilevel AINV preconditioner, Numer. Algorithms. 29 (1–3) (2002) 107–129.

[23] A. Reusken, On the approximate cyclic reduction preconditioner, SIAM J. Sci. Comput. 21 (2) (1999) 565–590.

[24] J.W. Ruge, K. Stüben, Algebraic multigrid, in: S. McCormick (Ed.), Multigrid Methods, Frontiers in Applied Mathematics, SIAM, Philadelphia, PA, 1987, pp. 73–130, (Chapter 4).

[25] Y. Saad, A flexible inner–outer preconditioned GMRES algorithm, SIAM J. Sci. Comput. 14 (2) (1993) 461–469.

[26] Y. Saad, ILUT: a dual threshold incomplete LU preconditioner, Numer. Linear Algebra Appl. 1 (4) (1994) 387–402.

[27] Y. Saad, Parallel sparse matrix library (P_SPARSLIB): The iterative solvers module, in: Advances in Numerical Methods for Large Sparse Sets of Linear Equations, vol. 10, Matrix Analysis and Parallel Computing, PCG 94, Keio University, Yokohama, Japan, 1994, pp. 263–276.

[28] Y. Saad, ILUM: a multi-elimination ILU preconditioner for general sparse matrices, SIAM J. Sci. Comput. 17 (4) (1996) 830–847.

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

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

[31] Y. Saad, B. Suchomel, ARMS: an algebraic recursive multilevel solver for general sparse linear systems, Numer. Linear Algebra Appl. 9 (5) (2002) 359–378.

[32] Y. Saad, J. Zhang, BILUM: block versions of multielimination and multilevel ILU preconditioner for general sparse linear systems, SIAM J. Sci. Comput. 20 (6) (1999) 2103–2121.

[33] Y. Saad, J. Zhang, BILUTM: a domain-based multilevel block ILUT preconditioner for general sparse matrices, SIAM J. Matrix Anal. Appl. 21 (1) (1999) 279–299.

[34] Y. Saad, J. Zhang, Diagonal threshold techniques in robust multi-level ILU preconditioners for general sparse linear systems, Numer. Linear Algebra Appl. 6 (4) (1999) 257–280.

[35] Y. Saad, J. Zhang, A multi-level preconditioner with applications to the numerical simulation of coating problems, in: D.R. Kincaid, A.C. Elster (Eds.), Iterative Methods in Scientific Computing IV, IMACS, New Brunswick, NJ, 1999, pp. 437–450.

[36] Y. Saad, J. Zhang, Enhanced multilevel block ILU preconditioning strategies for general sparse linear systems, J. Comput. Appl. Math. 130 (1–2) (2001) 99–118.

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

[38] M.D. Smooke, R.E. Michell, D.E. Keyes, Numerical solution of two-dimensional axisymmetric laminar diffusion flames, Combust. Sci. Tech. 67 (1989) 85–122.

[39] P. Wesseling, An Introduction to Multigrid Methods, Wiley, Chichester, England, 1992.
[40] J. Zhang, An explicit fourth-order compact finite difference scheme for three dimensional convection–diffusion equation, Comm. Numer. Methods Eng. 14 (3) (1998) 209–218.
[41] J. Zhang, A multilevel dual reordering strategy for robust incomplete LU factorization of indefinite matrices, SIAM J. Matrix Anal. Appl. 22 (3) (2000) 925–947.
[42] J. Zhang, On preconditioning Schur complement and Schur complement preconditioning, Electron. Trans. Numer. Anal. 10 (2000) 115–130.
[43] J. Zhang, Preconditioned Krylov subspace methods for solving nonsymmetric matrices from CFD applications, Comput. Methods Appl. Mech. Eng. 189 (3) (2000) 825–840.
[44] J. Zhang, A class of multilevel recursive incomplete LU preconditioning techniques, Korean J. Comput. Appl. Math. 8 (2) (2001) 213–234.
[45] J. Zhang, A grid-based multilevel incomplete LU factorization preconditioning technique for general sparse matrices, Appl. Math. Comput. 124 (1) (2001) 95–115.

Chi Shen received her Ph.D. degree of computer science from University of Kentucky in 2004. She is an Assistant Professor of Computer Science at the Kentucky State University. Her main research interest is in parallel and distributed computing, with applications to scientific computing. Her current research work focuses on iterative and preconditioning techniques for large sparse linear systems, numerical simulation and information retrieval.

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, computational medical imaging, and data mining and information retrieval. 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 and the Department of Energy. He is recipient of National Science Foundation CAREER Award and several other awards.

Kai Wang received a Ph.D. in computer science from the University of Kentucky in 2003. He is currently a postdoctoral researcher at the University of Illinois at Urbana-Champaign. His research interests include high performance computing, parallel and distributed computations and applications.