Parallel Multilevel Sparse Approximate Inverse Preconditioners in Large Sparse Matrix Computations

Kai Wang,* Jun Zhang,† Chi Shen‡
Laboratory for High Performance Scientific Computing and Computer Simulation
Department of Computer Science
University of Kentucky
Lexington, KY 40506-0046, USA
kwang0@csr.uky.edu, jzhang@cs.uky.edu, cshen@csr.uky.edu

ABSTRACT

We investigate the use of the multistep successive preconditioning strategies (MSP) to construct a class of parallel multilevel sparse approximate inverse (SAI) preconditioners. We do not use independent set ordering, but a diagonal dominance based matrix permutation to build a multilevel structure. The purpose of introducing multilevel structure into SAI is to enhance the robustness of SAI for solving difficult problems. Forward and backward preconditioning iteration and two Schur complement preconditioning strategies are proposed to improve the performance and to reduce the storage cost of the multilevel preconditioners. One version of the parallel multilevel SAI preconditioner based on the MSP strategy is implemented. Numerical experiments for solving a few sparse matrices on a distributed memory parallel computer are reported.

General Terms

Sparse matrix

Keywords

Parallel preconditioning, sparse approximate inverse, multilevel preconditioning

1. INTRODUCTION

We consider the solution of linear systems of the form $Ax = b$, where b is the right-hand side vector, x is the unknown vector, and A is a large sparse nonsingular matrix of order n. For solving this class of problems, preconditioned Krylov subspace methods are considered to be one of the most promising candidates [1, 11]. A preconditioned Krylov subspace method consists of a Krylov subspace solver and a preconditioner. It is believed that the quality of the preconditioner influences and in many cases dictates the performance of the preconditioned Krylov subspace solver [9, 19]. As the order of the sparse linear systems of interest continues to grow, parallel iterative solution techniques that can utilize the computing power of multiple processors have to be employed. So developing robust parallel preconditioners that are suitable for distributed memory architectures becomes an interesting problem.

Sparse approximate inverse (SAI) is a class of preconditioning techniques which can be used for solving large sparse linear systems on parallel systems [2, 3]. They possess high degree of parallelism in the preconditioner application phase and are shown to be efficient for certain type of problems. Based on the success achieved by applying multilevel structure to ILU preconditioners [12, 17, 21], the idea of combining strengths of the multilevel methods and the SAI techniques looks attractive. In fact, some authors have already proposed to improve the robustness of SAI techniques by using multilevel structures or to enhance the parallelism of multilevel preconditioners by using SAI [4, 8, 16, 20]. But none of these studies is done on a distributed memory computer system.

Recently, a multistep successive preconditioning strategy (MSP) was proposed in [15] to compute robust preconditioners based on SAI. MSP computes a sequence of low cost sparse matrices to achieve the effect of a high accuracy preconditioner. The resulting preconditioner has a lower storage cost and is more robust and more efficient than the standard SAI preconditioners.

In this paper, we investigate the use of the MSP strategy to construct a class of multilevel SAI preconditioners. We use forward and backward preconditioning strategy to improve the performance of the multilevel preconditioner. In addition MSP provides a convenient approach to creating approximate Schur complement matrices with different accuracy. We implement a two Schur complement matrix preconditioning strategy to reduce the storage cost of the multilevel preconditioner.

This paper is organized as follows. Section 2 outlines the procedure for constructing a multilevel preconditioner based on MSP. Section 3 discusses some implementation details and strategies to improve the performance of our multilevel preconditioner. Section 4 reports some numerical experiments with the multilevel pre-

*K. Wang's research work was funded by the U.S. National Science Foundation under grants CCR-9902022 and ACR-0202934.

†J. Zhang's research work was supported in part by the U.S. National Science Foundation under grants CCR-9902022, CCR-9988165, CCR-0092532, and ACR-0202934, by the U.S. Department of Energy Office of Science under grant DE-FG02-02ER45961, by the Japanese Research Organization for Information Science & Technology, and by the University of Kentucky Research Committee.

‡C. Shen's research work was funded by the U.S. National Science Foundation under grants CCR-9902022 and CCR-0092532.

SC '03, November 15-21, 2003, Phoenix, Arizona, USA

conditioners on a distributed memory parallel computer. A brief summary is given in Section 5.

2. PRECONDITIONER CONSTRUCTION

We recount the multistep successive preconditioning (MSP) strategy introduced in [15], and explain the concept of multilevel preconditioning techniques briefly. We then discuss the idea of using MSP in the multilevel structure to construct a multilevel SAI preconditioner. Our aim is to build a hybrid preconditioner with increased robustness and inherent parallelism.

2.1 Multistep successive preconditioning

In order to speed up the convergence rate of the iterative methods, we may transform the original linear system into an equivalent one $MAx = Mb$, where M is a nonsingular matrix of order n. If M is a good approximation to A^{-1} in some sense, M is called a sparse approximate inverse (SAI) of A [2, 3]. In many cases, the inverse of a sparse matrix may be a dense matrix, a high accuracy SAI preconditioner may have to be a dense matrix. The basic idea behind the multistep successive preconditioning (MSP) is to find a multi-matrix form preconditioner and to achieve a high accuracy sparse inverse step by step. In each step we compute an SAI inexpensively and hope to build a high accuracy SAI preconditioner in a few steps. MSP can be built from almost any existing SAI techniques [15]. The following is an MSP algorithm with a static sparsity pattern based SAI.

ALGORITHM 2.1. *Multistep Successive SAI Preconditioning [15].*

0. *Given the number of steps $l > 0$, and a threshold tolerance ϵ*
1. *Let $A_1 = A$*
2. *For $i = 1, \ldots, l$, Do*
3. *Sparsify A_i with respect to ϵ*
4. *Compute an SAI according to the sparsified sparsity pattern of A_i, $M_i \approx A_i^{-1}$*
5. *Drop small entries of M_i with respect to ϵ*
6. *Compute $A_{i+1} = M_i A_i$*
7. *EndDo*
8. *Sparsify A_l with respect to ϵ*
9. *Compute an SAI according to the sparsified sparsity pattern of A_l, $M_l \approx A_l^{-1}$*
10. *Drop small entries of M_l with respect to ϵ*
11. *$\prod_{i=1}^{l} M_i$ is the desired preconditioner for $Ax = b$*

In this algorithm, we use the sparsified pattern of the matrix A (or $A^2, A^3, \cdots$) to achieve higher accuracy [5]. Here "sparsified" refers to a preprocessing phase in which certain small entries of the matrix are removed before its sparsity pattern is extracted. In order to keep the computed matrix sparse, small size entries in the computed matrix M_i are dropped (postprocessing phase) at each step of MSP. We note here that Algorithm 2.1 is slightly different from the one developed in [15], in which different parameters are used for the preprocessing and postprocessing phases. Since these parameters are usually chosen to be of the same value [15], only one parameter is used in Algorithm 2.1.

Algorithm 2.1 generates a sequence of sparse matrices $M_1, M_2, \cdots, M_l$ inexpensively. They together form an SAI for A, i.e., $M_l M_{l-1} \cdots M_1 \approx A^{-1}$. From the numerical results in [15] we know that, in addition to enhanced robustness, MSP outperforms standard SAI in both the computational and storage costs.

2.2 Multilevel preconditioning

For an illustration purpose, we show in Fig. 1 the recursive matrix structure of a 4 level preconditioner. Usually, the construction of a multilevel preconditioner consists of two phases. First, at each level the matrix is permuted into a two by two block form, according to some criterion or ordering strategy,

$$A_\alpha \sim P_\alpha A_\alpha P_\alpha^T = \begin{pmatrix} D_\alpha & F_\alpha \\ E_\alpha & C_\alpha \end{pmatrix}, \tag{1}$$

where P_α is the permutation matrix and α is the level reference. For simplicity, we denote both the permuted and the unpermuted matrices by A_α. Second, the matrix is decomposed into a two level structure by a block LU factorization,

$$\begin{pmatrix} D_\alpha & F_\alpha \\ E_\alpha & C_\alpha \end{pmatrix} = \begin{pmatrix} I_\alpha & 0 \\ E_\alpha D_\alpha^{-1} & I_\alpha \end{pmatrix} \begin{pmatrix} D_\alpha & F_\alpha \\ 0 & A_{\alpha+1} \end{pmatrix}, \tag{2}$$

where I_α is the generic identity matrix at level α. $A_{\alpha+1} = C_\alpha - E_\alpha D_\alpha^{-1} F_\alpha$ is the Schur complement matrix, which forms the reduced system. The whole process, permuting matrix and performing block LU factorization, can be repeated with respect to $A_{\alpha+1}$ recursively to generate a multilevel structure. The recursion is stopped when the last reduced system $A_{\mathcal{L}}$ is small enough to be solved effectively. In multilevel preconditioning, the Schur complement matrix $A_{\alpha+1}$ is usually computed approximately to preserve sparsity [12, 13, 21].

The preconditioner application process consists of a level by level forward elimination, the coarsest level solution, and a level by level backward substitution. Suppose the right hand side vector b and the solution vector x are partitioned according to the permutation in (1), we have, at each level,

$$x_\alpha = \begin{pmatrix} x_{\alpha,1} \\ x_{\alpha,2} \end{pmatrix}, \qquad b_\alpha = \begin{pmatrix} b_{\alpha,1} \\ b_{\alpha,2} \end{pmatrix}.$$

The forward elimination is performed by solving a temporary vector y_α, i.e., for $\alpha = 0, 1, \ldots, \mathcal{L}-1$, by solving

$$\begin{pmatrix} I_\alpha & 0 \\ E_\alpha D_\alpha^{-1} & I_\alpha \end{pmatrix} \begin{pmatrix} y_{\alpha,1} \\ y_{\alpha,2} \end{pmatrix} = \begin{pmatrix} b_{\alpha,1} \\ b_{\alpha,2} \end{pmatrix},$$

with

$$\begin{cases} y_{\alpha,1} &= b_{\alpha,1}, \\ y_{\alpha,2} &= b_{\alpha,2} - E_\alpha D_\alpha^{-1} y_{\alpha,1}. \end{cases}$$

The last reduced system may be solved to a certain accuracy by a preconditioned Krylov subspace iteration to get an approximate solution $x_{\mathcal{L}}$. After that, a backward substitution is performed to obtain the preconditioning solution by solving, for $\alpha = \mathcal{L}-1, \ldots, 1, 0$,

$$\begin{pmatrix} D_\alpha & F_\alpha \\ 0 & A_{\alpha+1} \end{pmatrix} \begin{pmatrix} x_{\alpha,1} \\ x_{\alpha,2} \end{pmatrix} = \begin{pmatrix} y_{\alpha,1} \\ y_{\alpha,2} \end{pmatrix},$$

with

$$\begin{cases} x_{\alpha,2} &= A_{\alpha+1}^{-1} y_{\alpha,2}, \\ x_{\alpha,1} &= D_\alpha^{-1}(y_{\alpha,1} - F_\alpha x_{\alpha,2}), \end{cases}$$

where $x_{\alpha,2}$ is actually the coarser level solution.

2.3 Multilevel preconditioner based on MSP

A straightforward way to build a multilevel SAI preconditioner is to compute an SAI matrix M_α for the submatrix D_α, and to use M_α to substitute D_α^{-1} in Eq. (2). We have

$$\begin{pmatrix} D_\alpha & F_\alpha \\ E_\alpha & C_\alpha \end{pmatrix} \approx \begin{pmatrix} I_\alpha & 0 \\ E_\alpha M_\alpha & I_\alpha \end{pmatrix} \begin{pmatrix} D_\alpha & F_\alpha \\ 0 & A_{\alpha+1} \end{pmatrix},$$

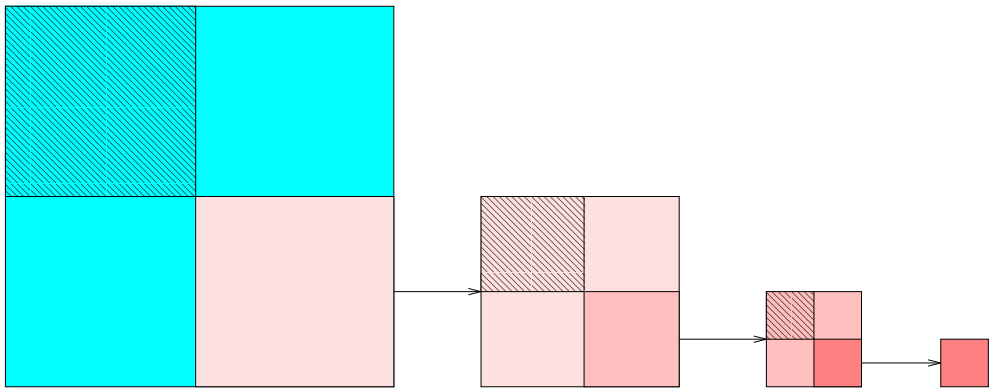

Figure 1: Recursive matrix structure of a 4 level preconditioner.

The approximate Schur complement matrix is computed as $A_{\alpha+1} = C_\alpha - E_\alpha M_\alpha F_\alpha$. Continue doing this for $A_{\alpha+1}$ at the next level, a multilevel preconditioner based on SAI can be constructed. Correspondingly, the forward and backward substitutions in the preconditioner application phase change to

$$\begin{cases} y_{\alpha,1} &= b_{\alpha,1}, \\ y_{\alpha,2} &= b_{\alpha,2} - E_\alpha M_\alpha y_{\alpha,1}, \end{cases} \tag{3}$$

and

$$\begin{cases} x_{\alpha,2} &= A_{\alpha+1}^{-1} y_{\alpha,2}, \\ x_{\alpha,1} &= M_\alpha (y_{\alpha,1} - F_\alpha x_{\alpha,2}). \end{cases} \tag{4}$$

Since the next level (Schur complement) matrix $C_\alpha - E_\alpha M_\alpha F_\alpha$ is computed according to M_α, we prefer an accurate SAI of D_α to avoid the error dilation during the construction of the multilevel preconditioner.

Through suitable permutation, it is possible to find a D_α with some special structure so that a sparse approximate inverse of D_α can be computed inexpensively and accurately. An (block) independent set strategy is used in [10, 12, 13, 14] for building the multilevel ILU preconditioners, in which D_α consists of small block diagonal matrices. Thus an accurate (I)LU factorization can be applied to these blocks independently. An independent set related strategy to find a well-conditioned D_α is also used in [16] to construct a multilevel factored SAI preconditioner. Unfortunately block independent set algorithms may be difficult to implement on distributed memory parallel computers.

For SAI based multilevel preconditioners, there is no need to exploit independent set ordering to extract parallelism, although a block diagonal matrix is certainly easier to invert [20]. What we want is to form a well-conditioned D_α. A diagonally dominant matrix is well-conditioned and may be inverted accurately. This suggests us to find a D_α matrix with a good diagonal dominance property so that D_α^{-1} can be computed inexpensively and accurately. In our implementation, at each level we use a diagonal dominance based strategy to force the rows with small size diagonal entries into the next level system and keep the relatively large diagonal entries in the current level. At the next level another well-conditioned subsystem is found by pushing the rows with unfavorable property into its next level system. This diagonal dominance based strategy is more like a divide and conquer strategy. Each time a difficult to solve problem is divided into two parts. One part is easier to solve than the other. We solve the easier part and employ the Schur complement strategy to deal with the other part.

We can improve the approximation of D_α^{-1} by using MSP. At each level, we compute a series of sparse matrices such that

$$M_{\alpha l} M_{\alpha l-1} \cdots M_{\alpha 1} \approx D_\alpha^{-1}, \tag{5}$$

where l is the number of steps. The corresponding Schur complement matrix can be formed as

$$C_\alpha - E_\alpha M_{\alpha l} M_{\alpha l-1} \cdots M_{\alpha 1} F_\alpha. \tag{6}$$

3. IMPLEMENTATION DETAILS

To solve a sparse linear system on a parallel computer, the coefficient matrix is first partitioned by a graph partitioner and is distributed to different processors (approximately) evenly. Suppose the matrix is distributed to each processor according to a row-wise partitioning, each processor holds k rows of the global matrix to form a local matrix.

Matrix permutation. We give a simple diagonal dominance based strategy to find a well-conditioned D_α matrix. This can be accomplished by computing a diagonal dominance measure for each row of the matrix based on the diagonal value and the sum of the absolute nonzero values of the row [17], i.e., $t_i = |a_{ii}| / \sum_{j \in \mathrm{Nz}(i)} |a_{ij}|$. Here $\mathrm{Nz}(i)$ is the index set of the nonzeros of the ith row. If the ith row is a zero row (locally) in a processor, we set $t_i = 0$. Then the rows with the large diagonal measures are permuted to form the upper block matrices D_α.

Let ϕ be a parameter between 0 and 1, which is referred to as the reduction ratio. We keep the $k \cdot \phi$ rows with the largest diagonal dominance measures at the current level and let $k \cdot (1 - \phi)$ rows go to the next level. When ϕ is close to 1, the reduced system (next level matrix) will be small. We can maintain load balancing by using the same ϕ in each processor.

We should also point out that in our implementation, the number of levels is not an input parameter like in the other multilevel methods, e.g., BILUM [12]. The multilevel setup algorithm builds the multilevel structure automatically, using ϕ as the constraint. One option is to let the construction phase stop when each processor has only 1 unknown. The last reduced system may be easy to solve. But this option may generate too many levels.

Forward and backward preconditioning. When examining the forward and backward steps in (3) and (4), we find that the operation $\tilde{x}_\alpha = M_\alpha b_\alpha$ appears twice. In exact form, this operation should be $x = D_\alpha^{-1} b$. So the value $\tilde{x}_\alpha$ is only an approximation of the true value x_α. The more accurately that $\tilde{x}_\alpha$ approxi-

mates x_α, the better a preconditioner we have. We can improve the computed value $\tilde{x}_\alpha$ by a preconditioned GMRES iteration on $M_\alpha D_\alpha x_\alpha = M_\alpha b_\alpha$ and using $\tilde{x}_\alpha$ as the initial guess. We call this preconditioning iteration as a forward and backward preconditioning (FBP) iteration.

Because the reduced systems (Schur complement matrices) are not computed exactly, there is no need to perform many FBP iterations to obtain a very accurate value of $\tilde{x}_\alpha$. A few sweeps are sufficient to make the approximate inverse of D_α comparably accurate with respect to other parts of the preconditioning matrix.

Schur complement preconditioning. When using MSP to compute the SAI of a matrix, a larger number of steps will produce a better approximation [15]. The final form of the preconditioner is a multi-matrix form and these matrices are stored individually. The combined storage cost of MSP is not too large if each matrix is sparse. This is one of the advantages of MSP over the standard SAI [15]. When using MSP to generate a multilevel preconditioner, these sparse matrices have to be multiplied out to compute the reduced system $A_{\alpha+1}$ as in (6). This may result in a dense Schur complement matrix.

A compromise can be reached in this situation by computing two Schur complement matrices with different accuracy by using different drop tolerances [8]. The more sparse one is used as the coarse level system to generate the coarse level preconditioner, and is discarded after serving that purpose. The more accurate and denser Schur complement matrix is kept as a part of the preconditioning matrix and is used in the preconditioner application phase. In our multilevel MSP preconditioner, we use a similar strategy to control the storage cost. Here the two Schur complement matrices are not computed by using different drop tolerances but by using different steps in MSP.

Suppose that MSP generates a series of matrices as in (5). We construct the explicit Schur complement matrix (for the reduced system) by using only the first few steps of (5), e.g., only $M_{\alpha 1}$, we have $C_\alpha - E_\alpha M_{\alpha 1} F_\alpha$. Because $M_{\alpha 1}$ is usually very sparse according to [15], this Schur complement matrix may be sparse (at least more sparse than the Schur complement matrix (6)) and can be computed inexpensively. In the preconditioning phase, we may use the more accurate Schur complement matrix (6) in an implicit form. To further improve the accuracy of the Schur complement solution, we may iterate on the implicit Schur complement matrix (6) with the lower level preconditioner. This strategy is called Schur complement preconditioning [18]. During the Schur complement preconditioning phase, we only perform a series of matrix vector products. We can see that if each of these matrices is sparse, the combined storage cost is not too high.

4. EXPERIMENTAL RESULTS

We implement our parallel multilevel MSP preconditioner (MMSP) based on the strategies outlined in the previous sections. At each level, we use a diagonal dominance measure based strategy to permute the matrix into a two by two block form. A static sparsity pattern based MSP is used to compute an SAI of D_α. During the preconditioning phase, we perform forward and backward preconditioning (FBP) iterations to improve the performance of MMSP. The last level reduced system is solved by a GMRES iteration preconditioned by MSP. We use the MSP code developed in [15] to build our MMSP code, which is written in C with a few LAPACK routines written in Fortran. The interprocessor communications are handled by MPI. We conduct a few numerical experiments to show the performance of MMSP. We also compare MMSP with MSP to show the improved robustness and efficiency due to the introduction of the multilevel structure.

The computations are carried out on a 32 processor (750 MHz) subcomplex of an HP superdome (supercluster) with distributed memory at the University of Kentucky. Unless otherwise indicated explicitly, four processors are used in our numerical experiments.

For all preconditioning iterations, which include the outer (main) preconditioning iterations, FBP iterations, Schur complement preconditioning iterations, and the coarsest level solver, we use a flexible variant of restarted parallel GMRES (FGMRES).

In all tables containing numerical results, "ϕ" is the reduction ratio; "step" indicates the number of steps used in MSP; "iter" shows the number of outer iterations for the preconditioned FGMRES(50) to reduce the 2-norm residual by 8 orders of magnitude. We also set an upper bound of 2000 for the FGMRES iteration, a symbol "-" in a table indicates lack of convergence; "density" stands for the sparsity ratio; "setup" is the total CPU time in seconds for constructing the preconditioner; "solve" is the total CPU time in seconds for solving the given sparse linear system; "total" is the sum of "setup" and "solve"; "ϵ" is the parameter used in MMSP and MSP to sparsify the computed SAI matrices.

4.1 Test problems

We first introduce the test problems used in our experiments. The right hand sides of all linear systems are constructed by assuming that the solution is a vector of all ones. The initial guess is a zero vector.

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

$$u_{xx} + u_{yy} + u_{zz} + 1000\,(p\,u_x + q\,u_y + r\,u_z) = 0 \qquad (7)$$

is used to generate some large sparse matrices to test the scalability of MMSP. Here the convection coefficients are chosen as $p = x(x-1)(1-3y)(1-2z)$, $q = y(y-1)(1-2z)(1-2x)$, $r = z(z-1)(1-2x)(1-2y)$. The Reynolds number for this problem is 1000. Eq. (7) is discretized by using the standard 7 point central difference scheme and the 19 point fourth order compact difference scheme [7]. The resulting matrices are referred to as the 7 point and 19 point matrices respectively.

General sparse matrices. We also use MMSP to solve the sparse matrices listed in Table 1. Most of them can be downloaded from MatrixMarket of the National Institute of Standards and Technology. [1] We remark that, based on our experience, most of these matrices are considered difficult to solve by standard SAI preconditioners.

4.2 Performance of MMSP

Forward and backward preconditioning. Fig. 2 depicts the convergence behavior and the CPU time with respect to the number of FBP iterations for solving the UTM1700B matrix. Here, the FBP iteration performs a few FGMRES(50) iterations to reduce the 2-norm of the relative residual. The number of iterations is an input parameter. From Fig. 2, we can see that when we increase the number of FBP iterations from 0 to 5, the number of outer FGMRES iterations decreases rapidly from more than 2000 to around 200. Correspondingly the total CPU time decreases from more than 20 seconds to around 3 seconds. However, we find that doing more than 5 FBP iterations does not result in significant difference in the convergence of MMSP, the number of outer iterations only decreases to around 100. The CPU time actually increases from 3 to 11 seconds. We conclude that the FBP iteration can improve the

[1] `http://math.nist.gov/MatrixMarket`.

matrices	n	nnz	description
FIDAP012	3973	80151	Flow in lid-driven wedge
FIDAP024	2283	48733	Nonsymmetric forward roll coating
FIDAP040	7740	456226	3D die-swell (square die $Re = 1, Ca = \infty$)
FIDAPM03	2532	50380	Flow past a cylinder in free stream ($Re = 40$)
FIDAPM08	3876	103076	Developing flow, vertical channel (angle = 0, Ra $= 1000$)
FIDAPM09	4683	95053	Jet impingment cooling
FIDAPM11	22294	623554	3D steady flow, heat exchanger
FIDAPM13	3549	71975	Axisymmetric poppet value
FIDAPM33	2353	23765	Radiation heat transfer in a square cavity
UTM1700A	1700	21313	Nuclear fusion plasma simulations
UTM1700B	1700	21509	Nuclear fusion plasma simulations
UTM3060	3060	42211	Nuclear fusion plasma simulations

Table 1: Information about the general sparse matrices used in the experiments (n is the order of a matrix, nnz is the number of nonzero entries).

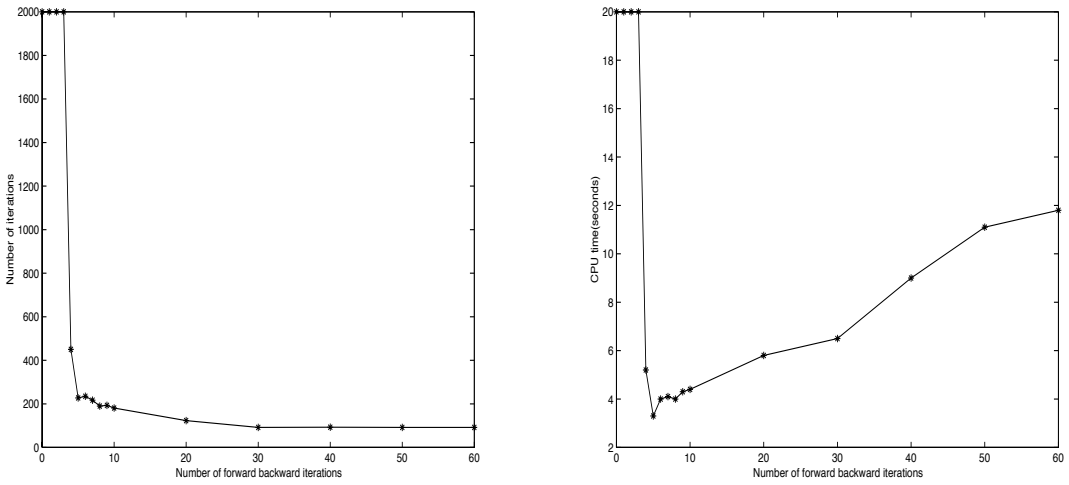

Figure 2: Convergence behavior of MMSP using different number of FBP iterations for solving the UTM1700B matrix ($\phi = 0.67, \text{step} = 2, \epsilon = 0.05, \text{density} = 3.48, \text{level} = 7$). Left: the number of outer iterations versus the number of FBP iterations. Right: the total CPU time versus the number of FBP iterations.

convergence of MMSP. But a large number of FBP iterations is not cost effective, since the other parts of the preconditioner are not computed exactly. In Fig. 2, the best result is obtained with 5 FBP iterations. In the following tests, we fix the number of FBP iterations at 5. We point out that the optimum value of this parameter may be problem dependent, and 5 FBP iterations may not be the best for all problems.

matrices	level	step	density	iter	setup	solve	total
FIDAPM09	8	1	3.40	-	0.9	-	-
	8	2	6.61	-	4.5	-	-
	8	3	9.82	248	11.1	13.8	24.9
FIDAPM33	7	1	3.21	-	0.6	-	-
	7	2	8.14	43	2.0	0.7	2.6
	7	3	14.93	31	5.4	0.7	6.2

Table 2: Comparison of MMSP with different MSP steps for solving two FIDAP matrices ($\phi = 0.67, \epsilon = 0.01$).

Schur complement preconditioning. The data in Table 2 show the influence of different MSP steps on the performance of MMSP. For the FIDAPM09 matrix, MMSP does not converges with 1 and 2 MSP steps. It converges with 3 MSP steps in 248 iterations. For the FIDAPM33 matrix, MMSP converges with 2 and 3 MSP steps, but fails in the 1 MSP step case. Just as we expected, a larger number of MSP steps builds a more robust MMSP preconditioner.

From Table 2, we also see that the storage cost of MMSP with 3 MSP steps is large and the implementation may be impractical in large scale applications. We rerun the two test problems in Table 2 using the two Schur complement matrix strategy, which is only implemented at the first level. The test results are shown in Table 3, where "step" is the number of MSP steps in the implicit Schur complement matrix. We only allow at most 50 Schur complement preconditioning iterations.

From Tables 2 and 3 we can see that the two Schur complement matrix strategy reduces the sparsity ratio of MMSP for solving the FIDAPM09 matrix from 11.1 to 4.79. For solving the FIDAPM33 matrix, the sparsity ratio of the 2 MSP step case is reduced from 8.14 to 4.51 and that of the 3 MSP step case is reduced from 14.93 to 6.65. In addition, the setup (construction) time is also reduced to some extent. We consider the two Schur complement matrix strategy as an effective way to reduce the memory cost of MMSP. How-

matrices	level	step	density	iter	setup	solve	total
FIDAPM09	8	3	4.79	94	2.5	66.6	69.1
FIDAPM33	7	2	4.51	19	0.7	9.1	9.9
	7	3	6.65	15	1.8	7.7	9.5

Table 3: Results of the two Schur complement matrix strategy, comparing to Table 2.

ever, the solution time increases because the Schur complement preconditioning strategy utilizes a lot of matrix vector products in the preconditioning phase. We provide the Schur complement preconditioning strategy as an option in our MMSP code in case we have to use a large number of MSP steps for some difficult problems and if the memory cost is more critical than the CPU time.

4.3 Comparison of MSP and MMSP

In Table 4, we compare MSP and MMSP for solving a few sparse matrices. For MSP, we adjust the parameter ϵ and the number of steps and try to give the best performance results for solving these matrices. For MMSP we fix $\epsilon = 0.05$ and $\text{step} = 2$. The number in the parentheses of MSP is the number of steps, and the number in the parentheses of MMSP is the number of MMSP levels.

Only 2 of the 9 tested matrices can be solved by MSP. The MMSP can solve these two matrices with smaller sparsity ratios and only 10 percent of the CPU time. In addition, MSP fails to solve the other 7 matrices, which can be solved by MMSP effectively.

4.4 Scalability tests

The main computational costs in MMSP are the matrix-matrix product and matrix-vector product operations. These operations can be performed in parallel efficiently on most distributed memory parallel architectures.

We use the 3D convection-diffusion problem (7) to test the implementation scalability of MMSP. The results in Fig. 3 are from solving a 7-point matrix with $n = 100^3$ and $nnz = 6940000$ using different number of processors. Due to the local memory limitation of our parallel computer, we can only run the test with at least 4 processors. For easy visualization, we set the speedup in the 4 processor case to be 4. From Fig. 3 we can see that MMSP scales well. In particular, we point out that the convergence behavior of MMSP is different from that of MSP. We know that the number of MSP iterations is not influenced by the number of processors when the problem size is fixed [15]. The number of MMSP iterations is affected by the number of processors. This is because the permutation of the matrix at each level depends on the ordering of the unknowns. Different number of processors results in different ordering of the unknowns for the same problem. As it is well known, the performance of (ILU type) preconditioners is affected by the matrix ordering [6]. Fortunately, the number of MMSP iterations does not seem to be strongly influenced by the number of processors.

In Fig. 4, the scaled scalability of MMSP is tested by solving a series of 19-point matrices. We try to keep the number of unknowns in each processor to be approximately 25^3. When we change the number of processors, the problem size increases at the same time. To be comparable, we also give the scaled scalability of MSP in the same figure. The parameters used are $\text{step} = 1, \text{level} = 10, \epsilon = 0.1$ for MMSP, and $\text{step} = 2, \epsilon = 0.05$ for MSP. From Fig. 4, We find that MMSP shows better scaled scalability than MSP for this test problem. The behavior of MMSP are more stable than that of MSP.

5. SUMMARIES

We have developed a class of parallel multilevel sparse approximate inverse (SAI) preconditioners based on MSP for solving general sparse matrices. A prototype implementation is tested to show the robustness and computational efficiency of this class of multilevel preconditioners.

From the numerical results presented, we can see that the forward and backward preconditioning (FBP) iteration is an effective strategy for enhancing the performance of MMSP. A few FBP iterations improve the convergence of MMSP. A suitable number of FBP iterations makes MMSP converge fast. Moreover, we can use a two Schur complement matrix strategy to reduce the storage cost.

Compared with MSP, MMSP is more robust and costs less to construct. The scalability of MMSP seems to be good. But the convergence of MMSP may be affected by the number of processors employed, due to the local matrix reordering implemented to enhance the factorization stability.

6. REFERENCES

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

[2] M. W. Benson and P. O. Frederickson. Iterative solution of large sparse linear systems arising in certain multidimensional approximation problems. *Utilitas Math.*, 22:127–140, 1982.

[3] M. W. Benson, J. Krettmann, and M. Wright. Parallel algorithms for the solution of certain large sparse linear systems. *Int. J. Comput. Math.*, 16:245–260, 1984.

[4] M. Bollhöfer and M. Volker. Algebraic multilevel methods and sparse approximate inverses. *SIAM J. Matrix Anal. Appl.*, 24(1):191–218, 2002.

[5] E. Chow. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM J. Sci. Comput.*, 21(5):1804–1822, 2000.

[6] I. S. Duff and G. A. Meurant. The effect of reordering on preconditioned conjugate gradients. *BIT*, 29:635–657, 1989.

[7] M. M. Gupta and J. Zhang. High accuracy multigrid solution of the 3D convection-diffusion equation. *Appl. Math. Comput.*, 113(2-3):249–274, 2000.

[8] G. Meurant. A multilevel AINV preconditioner. *Numer. Alg.*, 29(1-3):107–129, 2002.

[9] N. M. Nachtigal, S. C. Reddy, and L. N. Trefethen. How fast are nonsymmetric matrix iterations? *SIAM Matrix Anal. Appl.*, 13(3):778–795, 1992.

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

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

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

matrices	preconditioner	ϵ	density	iter	setup	solve	total
FIDAP024	MSP(3)	0.01	4.87	188	14.4	1.8	16.3
	MMSP(7)	0.05	3.05	39	0.8	0.6	1.4
FIDAPM08	MSP(3)	0.01	3.28	729	48.3	3.4	51.7
	MMSP(8)	0.05	3.02	192	1.1	4.5	5.6
FIDAP012	MSP(-)	-	-	-	-	-	
	MMSP(8)	0.05	3.38	57	1.1	1.2	2.3
FIDAP040	MSP(-)	-	-	-	-	-	
	MMSP(8)	0.05	3.46	39	4.3	4.0	8.3
FIDAPM03	MSP(-)	-	-	-	-	-	-
	MMSP(7)	0.05	3.35	62	0.8	1.0	1.8
FIDAPM11	MSP(-)	-	-	-	-	-	-
	MMSP(9)	0.05	6.81	200	16.3	85.1	101.4
FIDAPM13	MSP(-)	-	-	-	-	-	-
	MMSP(8)	0.05	3.58	86	1.1	1.7	2.8
UTM1700A	MSP(-)	-	-	-	-	-	-
	MMSP(7)	0.05	3.45	145	0.7	1.9	2.6
UTM3060	MSP(-)	-	-	-	-	-	-
	MMSP(8)	0.05	4.02	474	1.0	7.9	8.9

Table 4: Comparison of MSP and MMSP for solving a few sparse matrices.

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

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

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

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

[17] J. Zhang. A multilevel dual reordering strategy for robust incomplete LU factorization of indefinite matrices. *SIAM J. Matrix Anal. Appl.*, 22(3):925–947, 2000.

[18] J. Zhang. On preconditioning Schur complement and Schur complement preconditioning. *Electron. Trans. Numer. Anal.*, 10:115–130, 2000.

[19] J. Zhang. Preconditioned Krylov subspace methods for solving nonsymmetric matrices from CFD applications. *Comput. Methods Appl. Mech. Engrg.*, 189(3):825–840, 2000.

[20] J. Zhang. Sparse approximate inverse and multilevel block ILU preconditioning techniques for general sparse matrices. *Appl. Numer. Math.*, 35(1):67–86, 2000.

[21] J. Zhang. A class of multilevel recursive incomplete LU preconditioning techniques. *Korean J. Comput. Appl. Math.*, 8(2):213–234, 2001.

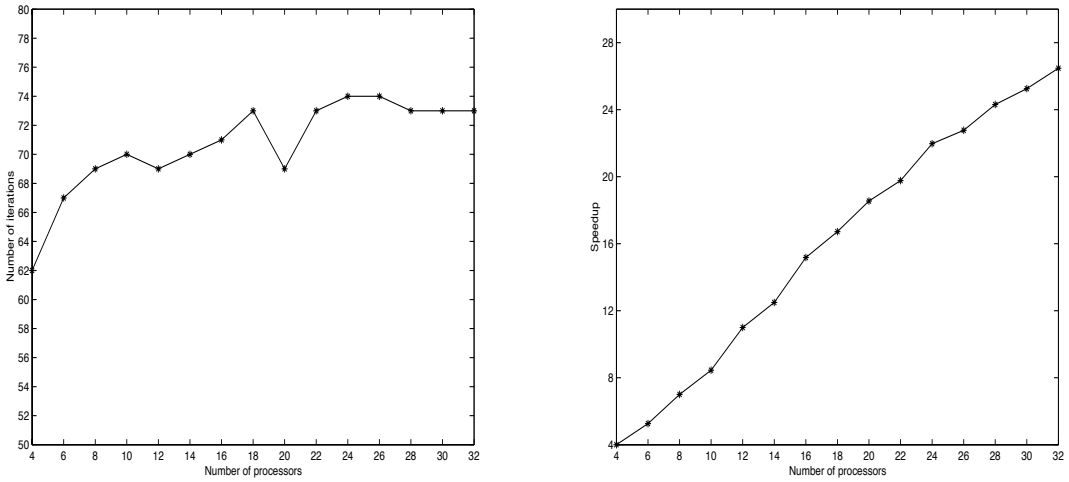

Figure 3: Scalability test of MMSP when solving a 7 point matrix with $n = 100^3$, $nnz = 6940000$ ($\epsilon = 0.1, \mathrm{step} = 2, \mathrm{level} = 10, \mathrm{density} = 2.03$). Left: the number of MMSP iterations versus the number of processors. Right: the speedup of MMSP as a function of the number of processors.

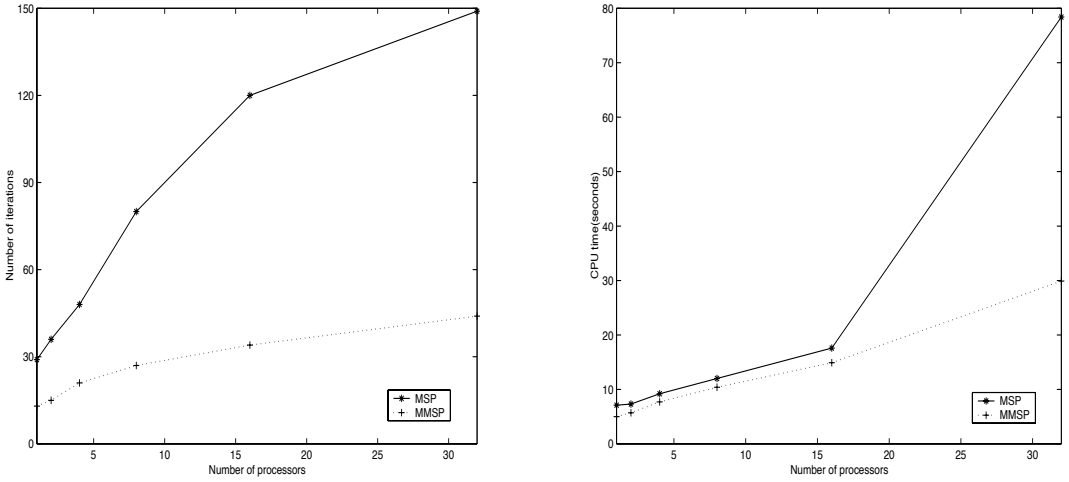

Figure 4: Scaled scalability test of MMSP and MSP for solving a series of 19 point matrices with $n \approx 25^3$ in each processor. Left: the number of iterations versus the number of processors. Right: the total CPU time versus the number of processors.