 CMES, vol.64, no.1, pp.91-108, 2010

A Simple OpenMP Scheme for Parallel Iteration Solvers in Finite Element Analysis

S.H. Ju[1]

Abstract: This study develops an OpenMP scheme to parallel the preconditioned conjugate gradient methods (PCG) in shared memory computers. The proposed method is simple and systematic, so a minor change in traditional PCG methods may produce effective parallelism. At first, the global stiffness matrix is renumbered in order to produce a parallel three-line form matrix, and a subroutine only needs to be called once in the finite element analysis. Several basic OpenMP commands are then added into the traditional incomplete Cholesky factorization (ILU) and symmetric successive over-relaxation (SSOR) codes to make the procedures of matrix multiplication, decomposition, forward substitution, and backward substitution fully parallel.

Keywords: Finite element method; ILU; OpenMP; Parallel; Preconditioned conjugate gradient methods; Shared memory computers; SSOR.

1 Introduction

For large finite element meshes, most parts of the stiffness matrix are zero, and the preconditioned conjugate gradient methods (PCG) are efficient iteration schemes to handle this sparse matrix. A review of this topic was undertaken by Benzi (2002), who surveyed preconditioning techniques and parallel schemes for the iterative solution of large linear systems. Although PCG methods are efficient for solving sparse matrices, the procedures are not parallel for common preconditioning schemes, such as incomplete Cholesky factorization (ILU) and symmetric successive over-relaxation (SSOR). For parallel PCG schemes, the domain decomposition method is commonly used, and the algorithm can be found in the reference (Smith et al., 1996). Recently, Avery and Farhat (2009) used two domain decomposition methods with Lagrange multipliers for solving iteratively quadratic programming problems with inequality constraints. For ILU and SSOR preconditioning, a good

[1] Department of Civil Engineering, National Cheng-Kung University, Tainan, Taiwan, R.O.C. Phone number: 886-6-2757575-63119; Fax number: 886-6-2358542; Email: juju@mail.ncku.edu.tw

degree of parallelism can be also achieved through graph coloring or disconnected block techniques (Axelsson, 1994). There are several recent papers in these topics. Yun (2003) used a block tridiagonal matrix scheme to generate parallelizable block ILU on distributed-memory computers. Staff (2005) developed a framework for solving the incompressible Navier-Stokes equations in parallel, and an unstructured mesh was decomposed into non-overlapping subdomains corresponding to the number of processors. Gosselet and Cachan (2006) used non-overlapping domain decomposition methods to solve the linear equations. Koulaei and Toutounian (2007) presented the recurrence formulae for computing the approximate inverse factors of tridiagonal and pentadiagonal matrices using a bordering technique. Gordon and Gordon (2009) studied the convergent behavior of parallel-block schemes for PCG, and the efficiency of several block methods was discussed. Currently, computer chips are designed on maximizing the number of cores that access to a common shared memory. This new metric is now taking the place of CPU frequency for characterizing performance. Thus, investigation of parallel finite element schemes in these shared memory computers is more and more important. Several researchers have used OpenMP to develop parallel finite element codes in this shared memory system. For example, Dong and Li (2009) used the OpenMP programming paradigm to parallelize the PCG solver for large-scale ground water flow problems. Huang et al. (2008) developed a parallel Fortran and OpenMP code to solve impact dynamic problems using the array expansion and domain decomposition methods.

This study presents a simple OpenMP scheme to parallel the ILU and SSOR PCG methods. First the global stiffness matrix is re-numbered in order to produce parallel blocks, and this step is required only once in the finite element analysis. Then, several OpenMP commands are added into the traditional ILU and SSOR codes to make the two schemes parallel. The major advantages of this method are that it is simple and systematic, so that the algorithm of parallel PCG methods is almost the same as the traditional one after performing the re-numbering procedure.

2 Pre-conditioned conjugate gradient method

The algorithm for the preconditioned conjugate gradient method is shown in Table 1, which is very suitable for the solution of sparse matrices as well as for parallel computing. However, steps 1 and 6 in Table 1 for the preconditioned matrix require solution procedures that are often difficult to perform parallel computations. We will thus re-order the equation number of the stiffness matrix to obtain parallel procedures for steps 1 and 6 using OpenMP commands. In this study, the global stiffness matrix is stored in three arrays, the diagonal array (D) and the lower and upper off-diagonal arrays of S and T, respectively. If the matrix is symmetric, only

j	1	2	3	4	5	6	7	8	9	10	
L(j)	0	1	2	4	6	8	11	13	17	19	i
LLL =	D1	T1	0	T3	T5	0	0	0	0	0	1
		D2	T2	0	0	0	T9	T12	0	0	2
			D3	T4	0	T7	T10	0	0	0	3
				D4	T6	T8	0	0	T14	0	4
					D5	0	0	0	T15	0	5
						D6	T11	0	T16	T18	6
							D7	T13	0	0	7
								D8	T17	T19	8
									D9	0	9
										D10	10

(a) Arrays LL, D and T

LLL=1, 2, 1,3, 1,4, 3,4, 2,3,6, 2,7, 4,5,6,8, 6,8.

(b) Array LLL

Figure 1: Structure of arrays D, T, LL and LLL for the conjugate gradient-like methods

array S is required to store the lower triangular part. The diagonal array stores the diagonal components of the global stiffness matrix. The off-diagonal array stores the off-diagonal components of the lower and upper parts of the global stiffness matrix, and only the non-zero components are included. Another two integral arrays, LL and LLL, indicate the index of the global stiffness matrix. LL indicates the total number of elements at the last off-diagonal element on each column for the upper part of the global stiffness matrix. LLL indicates the row number of each non-zero element for the upper part of the global stiffness matrix. The structures of LL and LLL are the same for S and T. Figure 1 shows an example of the structure of the T, LL and LLL arrays. The computer memory requirements of this matrix arrangement can be minimized without losing computational efficiency.

Table 1: The solution of **Ax=b** by the preconditioned conjugated gradient method, where **A** is symmtric, **B** is preconditioning and bold characters are a matrix or vector.

Step	Procedures
1	Initialization: m=0, $\mathbf{x}_0=\mathbf{0}$, $\mathbf{r}_0=\mathbf{b}$, $\mathbf{p}_0=\mathbf{z}_0=\mathbf{B}^{-1}\mathbf{r}_0$
2	$\alpha_m = \frac{\mathbf{r}_m^T\mathbf{z}_m}{\mathbf{p}_m^T\mathbf{A}\mathbf{p}_m}$
3	$\mathbf{x}_{m+1}=\mathbf{x}_m+\alpha_m\mathbf{p}_m$
4	$\mathbf{r}_{m+1}=\mathbf{r}_m+\alpha_m\mathbf{A}\mathbf{p}_m$
5	Convergence check $\|\mathbf{r}_{m+1}\|/\|\mathbf{r}_0\| < \text{Eps}$ (Eps from input, usually Eps=1E-6) If OK then return elseif NG then continue
6	$\mathbf{z}_{m+1}=\mathbf{B}^{-1}\mathbf{r}_{m+1}$
7	$\beta_m = \frac{\mathbf{r}_{m+1}^T\mathbf{z}_{m+1}}{\mathbf{r}_m^T\mathbf{z}_m}$
8	$\mathbf{p}_{m+1}=\mathbf{z}_{m+1}+\beta_m\mathbf{p}_m$
9	Let m=m+1; go to 2

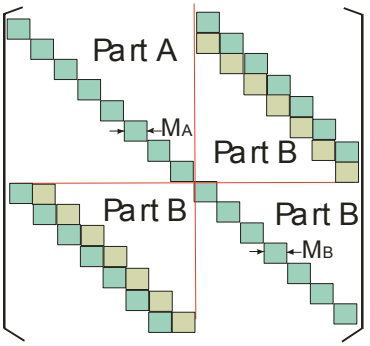

Figure 2: Three-line matrix form for parallel solvers

Table 2: Matrix multiplication ({wrk2}=[T'DT]{wrk1}) using the matrix form in Fig.1

```
1:  wrk2=0.0
2:  do i=2,NDF !Loop from 2 to the number of DOF (NDF)
3:    ii=LL(i-1)+1+NDF !the 1st element number for column i
4:    jj=LL(i)+NDF     !the last element number for column i
5:    aa=0.0           !aa,aa1=working real variables
6:    aa1=wrk1(i)
7:    do j=ii,jj       !Loop all non-zero elements
8:       n=LL(j)       !Column or row number of this element
9:       aa=aa+T(j)*wrk1(n) !for lower triangular part
10:      wrk2(n)=wrk2(n)+T(j)*aa1 !for upper triangular part
11:   enddo
12:   wrk2(i)=wrk2(i)+aa+wrk1(i)*D(I)   !Result of row i
13: enddo
14: wrk2(1)=wrk2(1)+wrk1(1)*D(1)        !Result of row one
```

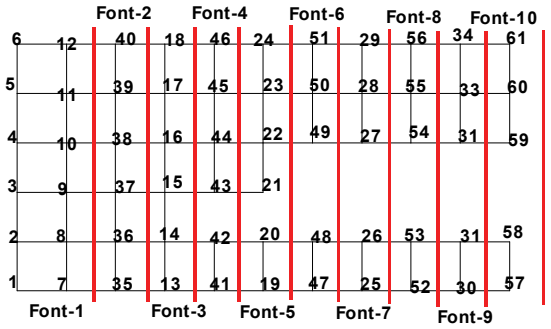

Figure 3: Forming the three-line matrix using wave front methods

3 Disconnection algorithm for SSOR and ILU pre-condition schemes

The most time consuming parts of PCG are the matrix multiplication ($\mathbf{AP}_m$ in step 2 of Table 1) and matrix solution (step 6 of Table 1). It is noted that not only the matrix solution but also the matrix multiplication is difficult to be parallel using the matrix form of Fig.1. Table 2 shows the Fortran codes of the matrix multiplication, where the updating of wrk2(j) in line 10 of Table 2 may be simultaneous in each processor, which causes parallel difficulty. However, it is very easy to modify

these codes in Table 2 to become parallel using the proposed method, and we will explain this in next section. For the SSOR preconditioned scheme, the Eisenstat's trick (Eisenstat, 1981) can be used to avoid matrix multiplication. However, the majority of the CPU time is still used to obtain the matrix solution. Thus, even through other parts of PCG can be fully parallel, the efficiency is still low. To overcome this problem, the stiffness matrix is changed into the three-line form, as shown in Fig.2. The matrix is categorized into two parts, where part-A contains the blocks that are disconnected from each others, and part-B contains the blocks that are connected sequentially to two part-A blocks. Since the matrix bands of part-B are considerably large, the number of PCG iterations required is often increased. Thus, we can enlarge the DOF of part-A (M_A in Fig.2) and minimize the DOF in part-B (M_B in Fig.2). To form this type of matrix, a wave front method (George and Liu, 1981) can be used. The nodal numbers are defined sequentially first from the nodes in the region of the odd wave font numbers and then from those of the even wave font numbers. An example is shown in Fig.3. The renumbering of nodes to obtain the minimum band of the stiffness matrix is usually a necessary procedure for finite element analyses, but this is often executed before one obtains the matrix form, such as LL and LLL in Fig.1. To overcome this drawback, the following systematic procedures were developed to obtain the three-line form of the stiffness matrix using only the index arrays LL and LLL, so all the procedures can be completed without going back to the element part.

(1) Renumber the nodes to obtain the minimum band of the stiffness matrix. In this study, the reverse Cuthill-McKee (George and Liu, 1981) method is used.

(2) Obtain the index arrays LL and LLL of the stiffness matrix, as shown in Fig.1.

(3) Let i=1, and select the first block within the degrees of freedom (DOF) from 1 to KL(i), where KL(i) is the maximum degree of freedom of block i, and KL(1) for the first block can be obtained from the input.

(4) Let i=i+1, and find the next block, whose DOF are within KL(i-1)+1 to KL(i), where i is the current block number. KL(i) is obtained from the largest DOF to KL(i) DOF until the LLL(1) of KL(i) DOF is smaller than or equal to KL(i-1).

(5) Go to step (4) for the next block, until no block can be found, and the remaining DOF are the last block (defined as the nKL^{th} block).

(6) Since we need to increase M_A of part-A to obtain a more efficient solution of PCG method, a number of the successful blocks (N) beginning with the odd block number can be combined as a new odd number block. Finally, one obtains the total number of blocks named as nKL_N.

(7) The nodal numbers are re-numbered sequentially, first from the nodes in the region of the odd block numbers and then from those of even block numbers.

(8) Find LL and LLL according to the nodal number arranged in step (7).

For the above procedures, steps 1 and 2 are standard procedures that should be performed in the traditional finite element method. Steps 3 to 8 can be just achieved after completing step 2, and those require only the index arrays LL and LLL. Appendix-A shows a Fortran module to perform this computation, in which only a public subroutine (SPLLOK) needs to be called on to obtain the results of steps 3 to 8. The private subroutine NCPU is used to perform steps 3 to 7, and the private subroutine SPLL is to perform step 8. This module contains only 150 lines without difficult or trick algorithms. After calling subroutine SPLLOK, the three-line form matrix shown in Fig.2 is obtained. This matrix can be fully parallel in the procedures of the matrix multiplication, decomposition, forward substitution, and backward substitution for ILU and SSOR PCG method. The parallel conditions are explained as follows:

For the matrix multiplication and decomposition:

(1) Blocks in part-A can be fully parallel.

(2) Blocks with odd and even numbers in part-B can be fully parallel, after the procedures for part-A blocks are completed. One can first deal with odd number blocks and then even number ones.

For the forward substitution

(1) Blocks in part-A can be fully parallel.

(2) Blocks in part-B can be fully parallel after the procedures for part-A blocks are completed.

For the backward substitution

(1) Blocks with odd and even numbers in part-B can be fully parallel. One can first deal with odd number blocks and then even number ones.

(2) Blocks in part-A can be fully parallel, after the procedures for part-B blocks are completed.

Based on the above explanation, we will use the Open MP codes to perform the parallel procedures. The programming is simple, so the parallel procedures are almost the same as the traditional non-parallel ones.

4 Disconnection algorithm procedures using OpenMP

OpenMP (http://www.openmp.org) is an application program interface that may be used to explicitly direct multithreaded, shared memory parallelism for Fortran, C, or C++. A number of directives can be used for parallel computation in OpenMP. In this study, only the most efficient one, !$OMP do schedule (dynamic), is used. This directive makes the immediately following do-loop be executed in parallel, and schedule (dynamic) means that when one thread finishes its piece of work, it gets a new one immediately. It is simple to change traditional matrix multiplication, decomposition, forward substitution, and backward substitution for the three-line matrix using OpenMP. Moreover, calculation of element matrices and assembling of the global matrix can also be performed without difficulty using OpenMP. These procedures are described in more detail in the following subsections.

4.1 Matrix multiplication and ILU decomposition

For matrix multiplication and ILU decomposition, the traditional method contains an outer loop, as shown in Table 2, where NDF is the total number of DOF. The above loop is simply modified to an OpenMP parallel loop for the three-line matrix, as shown in Table 3, for the matrix multiplication. The only change is that the outer loop of the traditional method is modified to begin from the loop of the part-A blocks, then the part-B odd blocks, and finally part-B even blocks. The codes inside the loop i as shown in Tables 2 and 3 are identical, where the OpenMP commands are optional, since it can still work as a non-parallel loop without them, and with them, it is a parallel loop. Appendix-B shows the OpenMP codes for the matrix decomposition, where the codes inside loop i are identical for non-parallel traditional and parallel three-line matrix methods.

4.2 Forward substitution of the triangular matrix

The forward and backward substitutions of the triangular matrix to obtain a solution vector are the major time-consuming steps for ILU and SSOR PCG, since they need to be executed in each iteration. For the forward substitution, each part-A block can be executed independently. After finishing the forward substitution of the part-A blocks, each part-B block can also be executed independently. Thus, the traditional forward substitution codes can be easily modified to parallel codes for the three-line matrix. Table 4 shows the OpenMP Fortran codes, where only the outer loop of the traditional scheme needs to be modified to the codes from lines 1 to 10. The OpenMP commands are optional to indicate parallel or non-parallel computation.

Table 3: Matrix multiplication ({wrk2}=[T'DT]{wrk1}) using the three-line matrix and OpenMP directives

```
    wrk2=0.0d0
    do kkp=1,3
      if (kkp==3) then !for part-B even blocks
         kk1=nbk2+2; kk2=nbk; inc=2
      elseif (kkp==2) then !for part-B odd blocks
         kk1=nbk2+1; kk2=nbk; inc=2
      elseif (kkp==1) then !for part-A blocks
         kk1=1; kk2=nbk2;  inc=1
      endif
!$OMP PARALLEL private(nb,aa,aa1,i,j,nb1,nb2,ii,jj,n)
!$OMP& shared(kk1,kk2,inc,kLL,ndf,LL,wrk1,wrk2,D,ts)
!$OMP do schedule (dynamic)
      do nb=kk1,kk2,inc
        nb1=kLL(nb-1)+1
        nb2=kLL(nb)
        do i=nb1,nb2
          Exact the same as lines 3 to 12 in Table 3
        enddo
      enddo
!$OMP end do
!$OMP end PARALLEL
    enddo
    wrk2(1)=wrk2(1)+wrk1(1)*D(1)
```

4.3 Backward substitution of the triangular matrix

For the backward substitution, each part-B block with an odd number can first be executed independently, and part-B block with an even number can then also be executed independently. After finishing the procedures for the part-B blocks, all the blocks in part-A can be parallel to perform the backward substitution. Thus, the procedures of backward substitution are similar to those of matrix decomposition, and the only change is that the outer loop of the traditional method is modified to begin from the part-B odd blocks, then the part-B even blocks, and finally the loop of part-A blocks. Table 5 shows the OpenMP Fortran codes, where the codes inside loop i are identical for the non-parallel traditional and parallel three-line matrix methods.

Table 4: Forward substitution of the triangular matrix using the three-line matrix and OpenMP directives (Italic parts are traditional ILU decomposition codes.)

```
  do kk=1,2
   if (kk==1) then        !Part-A blocks
     kk1=1; kk2=nbk2
   else
     kk1=nbk2+1; kk2=nbk !Part-B blocks
   endif
!$OMP PARALLEL private(nb,tmp,i,j,nb1,nb2,j1,j2)
!$OMP& shared(r,T,kk1,kk2,kLL,LL,LLL)
!$OMP do schedule (dynamic)
   do nb=kk1,kk2
    nb1=kLL(nb-1)+1; nb2=kLL(nb)
    do i=nb1,nb2 !Begin. of traditional codes(nb1=2,nb2=ndf)
      j1=LL(i-1)+1; j2=LL(i); tmp=0.0d0
      do j=j1,j2
        tmp=tmp+T(j)*r(LLL(j))
      enddo
      r(i)=r(i)-tmp
    enddo         !End of traditional codes
   enddo
!$OMP end do
!$OMP end PARALLEL
  enddo
```

4.4 Calculation of element matrices and assembling the global matrix

The calculation of element matrices can be fully parallel, but assembling them into the global matrix may not be parallel, since element matrices in two processors can be added into a same DOF of the global matrix simultaneously. One can use a similar disconnected block scheme as the three-line matrix to obtain a parallel assembling procedure. However, the element number should be re-ordered, which requires a complicated procedure. Alternative is to generate element matrices in parallel and to assemble these into the global matrix sequentially. Since the former requires much more CPU time than the latter, this method will not lose much parallel efficiency. Table 6 shows the OpenMP scheme, which changes only a small part of the traditional method.

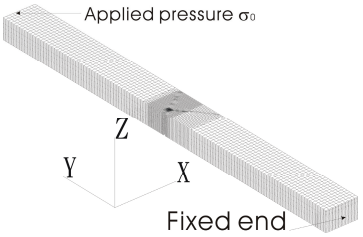

Figure 4: Finite element mesh of a plate with a multi-material interface notch for case 1 (Modeled by 20-node isoparametric elements with 282,612 nodes and 845,853 DOF)

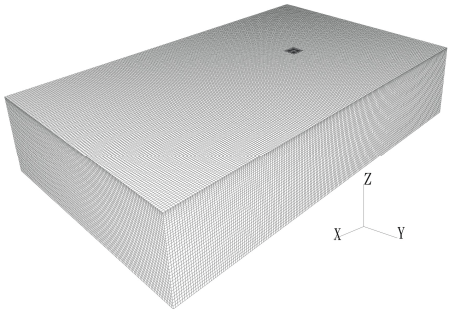

Figure 5: Finite element mesh of the wave propagation in layered soils for case 2 (Modeled by 8-node isoparametric elements with 1,299,594 nodes and 3,895,674 DOF)

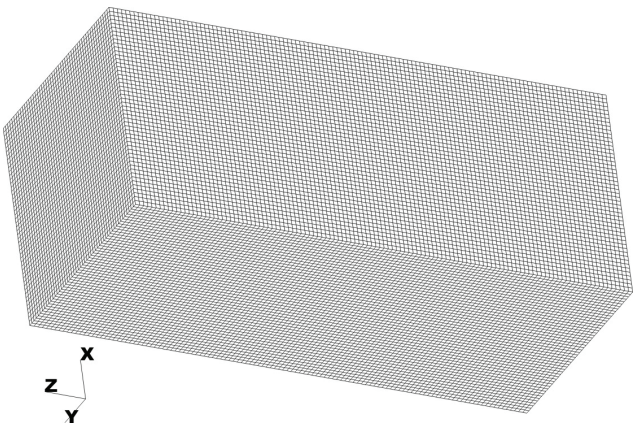

Figure 6: Finite element mesh of a cantilever beam subjected to a tip sine force for case 3 (Modeled by 20-node isoparametric elements with 1,243,941 nodes, and 3,708,720 DOF)

Table 5: Backward substitution of the triangular matrix using the three-line matrix and OpenMP directives (Italic parts are traditional ILU codes.)

```
 do kk=1,3
  if (kk==1) then !Part-B odd blocks
    kk1=nbk2+1; kk2=nbk; inc=2
  elseif (kk==2) then !Part-B even blocks
    kk1=nbk2+1+1; kk2=nbk; inc=2
  else !Part-A blocks
    kk1=1; kk2=nbk2; inc=1
  endif
!$OMP PARALLEL private(nb,tmp,i,j,nb1,nb2,j1,j2)
!$OMP& shared(r,a,kk1,kk2,kLL,ndf,LL,inc)
!$OMP do schedule (dynamic)
  do nb=kk1,kk2,inc
    nb1=kLL(nb-1)+1; nb2=kLL(nb)
    do i=nb2,nb1,-1 !Begin. of traditional codes(nb1=1,nb2=ndf)
      j1=LL(i-1)+1; j2=LL(i); tmp=r(i)
      do j=j1,j2
        jj=LLL(j); r(jj)=r(jj)-T(j)*tmp
      enddo
    enddo !End of traditional codes
  enddo
!$OMP end do
!$OMP end PARALLEL
 enddo
```

Table 6: A simple parallel procedure for the calculation of element matrices and assembling them into the global matrix

```
!$OMP PARALLEL
!$OMP& default (private)
!$OMP& shared(give shared variables)
!$OMP do
   Do n=1,nelm    ! nelm=the total number of elements
        Find element matrices
!$OMP critical
   Assembling element matrices into the global matrix
!$OMP end critical
  Enddo
!$OMP end do
!$OMP end PARALLEL
```

5 Numerical experiments

There are three numerical experiments in this section. Case one is a 3D static finite element analysis of a plate with multi-material notches, as shown in Fig.4. The bottom boundary of the mesh is fixed, and the upper boundary is applied to a pressure in the global X direction, as shown in Fig.4. The details of material properties and dimensions can be found in the reference (Ju, 2010). The total number of nodes is 282,612, and the total number of DOF is 845,853. Case two is a 3D dynamic analysis of the wave propagation in layered soils, as shown in Fig.5, in which there is a foundation with four reinforced concrete piles in the mesh. A shaker generating a sine wave in the Y direction is installed on the top of the concrete foundation, and absorbing boundary conditions are arranged along the mesh boundaries in order to remove boundary reflection waves. The detail of material properties and dimensions can be found in the reference (Ju and Ni, 2007). The total number of nodes is 1,299,594, and the total number of DOF is 3,895,674. Case 3 is a 3D dynamic analysis of a cantilever beam subjected to a tip sine wave force, as shown in Fig.6. The material is steel with a Young's modulus of 2E8 kN/m^2, Poisson's ratio of 0.29, and mass density of 7.9 t/m^3. The total number of nodes is 1,243,941, and the total number of DOF is 3,708,720.

For the three cases, the reverse Cuthill-McKee renumbering procedure (George and Liu, 1981) was first used to minimize the bands of the stiffness matrix. Then, steps (3) to (5) in Section 3 were performed to find the three-line stiffness matrix. Table 7 shows the nKL of the four cases, where nKL is the maximum number of blocks in parts A and B under N=1 (N is defined in step (3) of Section 3). It is noted that N/4 is the maximum number of processors that can be set.

Table 7: The maximum number of blocks (nKL) in parts A and B under N=1 for the three cases (Maximum number of processors=nKL/4)

Case	1	2	3
nKL	152	132	121

5.1 Comparison of the iteration number between traditional and parallel schemes

The current method changes the original stiffness matrix to the three-line form, which may increase the number of iterations for PCG methods. Thus, this section investigates the PCG efficiency due to this problem. The efficiency ratio (R_N) is defined as follows:

$$R_N = \frac{\text{The number of iterations for the original matrix form}}{\text{The number of iterations for the three - line matrix form under a certain N}}$$

where N defined in step (3) of Section 3 is the number of blocks to be combined for part-A blocks. R_N<1 means that the three-line matrix form requires more iterations than the original one.

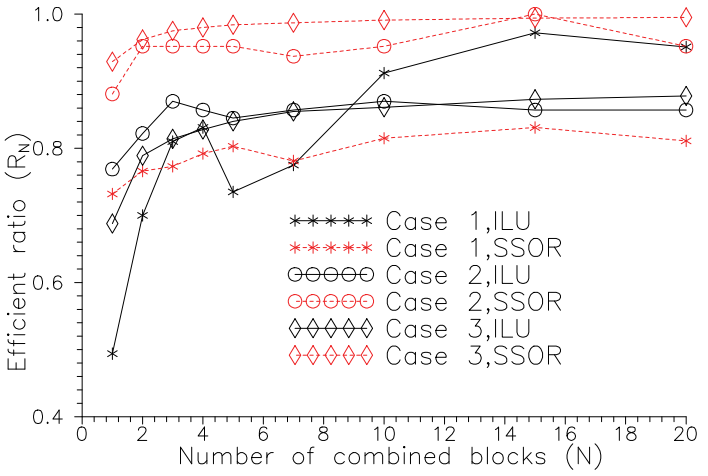

Figure 7: Efficiency ratio (R_N) changing with N for the three numerical experiments.

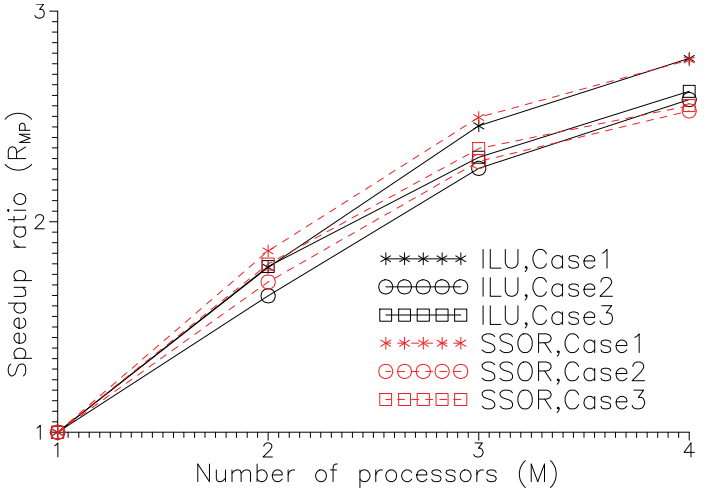

Figure 8: Speedup ratio (R_{MP}) changing with the number of processors (M) for the three numerical experiments.

Figure 7 shows the efficiency ratio (R_N) changing with N for the three numerical experiments. This figure indicates that the efficiency increases when N increases. The number of iteration required for N=1 can be much larger that that of the original matrix form, especially for the ILU scheme. However, when N is larger than or equal to 3, the increase of R_N is much slower, and this efficient ratio is about 0.8 to 1 in this situation. Thus, N equal to 3 is suggested to generate the three-line matrix form, and this N value is also used in the next section. It is noted that using a large N will decrease the total number of three-line matrix blocks, which reduces the maximum number of processors for the parallel solver.

5.2 Comparison of CPU time for multi-processors

This section compares the CPU time for the three numerical experiments using different numbers of processors. An AMD Athlon-II computer with four processor cores and 8 GB RAM was used in the 64-bit Windows-XP operating system. Intel Fortran 10.0 with OpenMP 2.5 was used to compile the finite element program (http://myweb.ncku.edu.tw/∼juju). The results were very similar to these for the 64-bit Windows-7 and 64-bit Ubuntu Linux operating systems. The speedup ratio (R_{MP}) is defined as follows:

$$R_{MP} = \frac{\text{CPU time of the matrix solution using a processor}}{\text{CPU time of the matrix solution using M processors}}$$

Figure 8 shows the speedup ratio (R_{MP}) changing with the number of processors (M) used for the three numerical experiments. This figure indicates that the speedup ratio increases when M increases. However, the efficiency gradually reduces as the number of processors increases. The average speedup using four processors is about 2.7, which is 33% lower than that of the ideal condition. A similar level of efficiency was reported by Dong and Li (2009) for solving groundwater problems, Smelyanskiy et al. (2009) with sparse parallel linear solvers, and González et al. (2009) with three multi-threaded solvers. The parallel efficiency is fairly dependent on program compliers, which has been discussed in these three earlier papers, and is also affected by computer hardware. The speed of data transport between the CPU and memory is important for parallel finite element solvers, since each procedure of the matrix multiplication, forward substitution, and backward substitution requires the transport of the global matrix, which often contains a huge amount of data. This data communication between the CPU and memory is often not parallel for multi-core processors. Thus, this same basic CPU time is required for both sequential and parallel procedures, which causes the speedup efficiency to be smaller than the ideal condition.

6 Conclusions

An OpenMP scheme was developed to parallel the ILU and SSOR PCG methods in shared memory computers. First the global stiffness matrix is renumbered in order to produce the three-line form matrix, and this step requires only the index arrays of the global stiffness matrix. A 150-line Fortran module was included to perform this procedure, in which a subroutine only needs to be called once in the finite element analysis. Additionally, this procedure can be easily added to any traditional linear or nonlinear finite element codes. Several OpenMP commands are than added into the traditional ILU and SSOR codes to make the procedures of the matrix multiplication, decomposition, forward substitution, and backward substitution fully parallel. The major advantages of this method are that it is simple and systematic, so that the algorithm of parallel PCG methods is almost the same as the traditional one after performing the renumbering procedure.

References

Axelsson, O. (1994): *Iterative Solution Methods*, Cambridge University Press, NewYork.

Avery, P. ; Farhat, P. (2009): The FETI family of domain decomposition methods for inequality-constrained quadratic programming: Application to contact problems with conforming and nonconforming interfaces. *Computer Methods in Applied Mechanics and Engineering*, vol.198, pp.1673-1683.

Benzi, M. (2002): Preconditioning techniques for large linear systems: a survey. *Journal of Computational Physics*, vol.182, pp.418–477.

Dong, Y.; Li, G. (2008): A Parallel PCG Solver for MODFLOW. *Ground Water*, vol.47, pp.35-44.

Eisenstat, S.C. (1981): Efficient implementation of a class of preconditioned conjugate gradient methods. *SIAM J. ScI. Stat. Comput.*, vol. 2, pp.1-4.

George, A.; Liu, J.W. (1981): *Computer Solution of Large Sparse Positive Definite Systems.* Prentice–Hall, Englewood Cliffs, NJ.

González, F.; Luaces, A.; Dopico, D.; González, M. (2009): Parallel linear equation solvers and OpenMP in the context of multibody system dynamics. Proceedings of the ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2009, San Diego, California, USA, pp.1-11.

Gordon, D.; Gordon, R. (2009): Solution methods for nonsymmetric linear systems with large off-diagonal elements and discontinuous coefficients. *CMES: Computer Modeling in Engineering & Sciences*, vol.53, pp.23-45.

Gosselet, P. (2006): Archives of computational methods in engineering. *LMT Cachan State of the Art Reviews Non-overlapping Domain Decomposition Methods in Structural Mechanics*, vol.13, pp.515-572.

Huang, P.; Zhang, X.; Ma, S.; Wang, H.K. (2008): Shared memory OpenMP parallelization of explicit MPM and its application to hypervelocity impact. *CMES: Computer Modeling in Engineering & Sciences*, vol.38, pp.119-147.

Ju, S.H. (2010): Finite element calculation of stress intensity factors for interface notches. *Computer Methods in Applied Mechanics and Engineering,* vol.199, pp.2273-2280.

Ju, S.H.; Ni S.-H. (2007): Determining Rayleigh damping parameters of soils for finite element analysis. *International Journal for Numerical and Analytical Methods in Geomechanics*, vol.31, pp.1239-1255.

Koulaei, M.H.; Toutounian, F. (2007): On computing of block ILU preconditioner for block tridiagonal systems. *Journal of Computational and Applied Mathematics*, vol.202, pp.248-257.

Smelyanskiy M.; Skedxielewski, S.; Dulong, C. (2005): Parallel computing for large-scale optimization problems: Challenges and solutions. *Intel Technology Journal*, vol.9, pp.151-164.

Smith, B.F.; Bjørstad, P.E.; Gropp, W.D. (1996): Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*, Cambridge Univ. Press, Cambridge/New York/Melbourne.

Staff, O.; Wille, S.O. (2005): Parallel ILU preconditioning and parallel mesh adaptation with load balancing for general domain decompositions for the Navier–Stokes equations. *International Journal for Numerical Methods in Fluids*, vol.47, pp.1301-1306.

Yun, J.H. (2003): Parallel performance of block ILU preconditioners for a block-tridiagonal matrix. *The Journal of Supercomputing*, vol.24, pp.69-89.

Appendix: A Fortran module to find the three-line matrix for parallel solvers

```
!*******************************************************
    module nbkLL
       integer*4, allocatable:: kLL(:)
       integer*4 nbk,nbk2; public nbk,nbk2,kLL,SpLLOK
    contains
!*******************************************************
!**  Find parallel data and re-arrange LL, LLL and NOSC
!**  Output (in module nbkLL)
!**  kLL(i)=the last DOF of block i; nbk=number of blocks
!**  nbk2=the last block number of the first step
!**  NOSC(node No., DOF of the node)=Global DOF, (=0)fixed
!**  node=number of nodes, kdf=max.DOF per node
!**  NDF=number of global DOF, LL & LLL in Fig.1
!**
     subroutine SpLLOK(LL,LLL,NOSC,node,ndf,kdf)
     dimension LL(ndf),LLL(*),NOSC(node,kdf)
     allocatable id(:); allocate (id(ndf))
!**
!**  Find Id(old DOF)=new DOF
!**
     NLL=ndf/1000+5
     call nCPU(LL,LLL,id,NLL,ndf)
!**
!**  Change LL,LLL, and Nosc due to Id(old-DOF)=new-DOF
!**
     call SpLL(LL,LLL,NOSC,NODE,NDF,KDF,id,LL(ndf))
     end subroutine
!*******************************************************
!**  Find Id(old-DOF)=new-DOF
!**  kLL(1-nbk)=the last DOF of block i (in module nbkLL)
!**  nbk=number of blocks; nbk2=the last PART-A block number
!**
     subroutine nCPU(LL,LLL,Id,NLL,ndf)
     dimension LL(ndf),LLL(*),id(ndf)
     allocatable kL1(:); allocate (kL1(0:ndf-1))
!**
!**  NLL=the number of DOF for the first block
!**  kLL(i)=the last DOF of block i; nbk=number of blocks
!**
     nbk=1; kL1(nbk)=NLL; i=NLL; nkLL=i
     do
       do j=ndf,nkLL+1,-1
          if (LLL(LL(j-1)+1)<=nkLL) exit
       enddo
       nbk=nbk+1; i=j; if (i>ndf) i=ndf
       kL1(nbk)=i; nkLL=i; if (i==ndf) exit
     enddo
     allocate (kLL(0:nbk))
!**
!**  kL1(i)=the last DOF of block i, nbk blocks
!**  inc is the number of part-A blocks to be combined
!**
     ii=1; inc=5; kk=0
     do i=inc,nbk-1,inc
        kLL(ii)=kL1(i); ii=ii+1; kLL(ii)=kL1(i+1); kk=i+1
     enddo
!**
     if (kk<nbk) then
        ii=ii+1; kLL(ii)=kL1(nbk)
     endif
!**
     nbk=ii
     do i=1,nbk
        kL1(i)=kLL(i)
     enddo
!**
!**  Find id(old DOF)=new DOF
!**
     kLL(0)=0; kL1(0)=0; ndof=0; nbk1=0
     do i=1,nbk,2 !Begin from the odd blocks
        do j=kL1(i-1)+1,kL1(i)
           ndof=ndof+1; id(j)=ndof
        enddo
        nbk1=nbk1+1; kLL(nbk1)=ndof
     enddo
     nbk2=nbk1
!**
     do i=2,nbk,2 !Continuous for the even blocks
        do j=kL1(i-1)+1,kL1(i)
           ndof=ndof+1; id(j)=ndof
        enddo
        nbk1=nbk1+1; kLL(nbk1)=ndof
     enddo
     end subroutine
!**
!*******************************************************
!**  Input id(old)=new, change LL,LLL, and Nosc
!**
     subroutine SpLL(LL,LLL,NOSC,NODE,NDF,KDF,Id,nnn)
     dimension LL(ndf),LLL(nnn),Nosc(node,kdf),id(ndf)
     allocatable L1(:),LL1(:),nosc1(:,:)
!**
!**  change Nosc according id(old)=new
!**
     allocate(nosc1(node,kdf))
     do i=1,node
       do j=1,kdf
         ii=nosc(i,j)
         if (ii<=0) then
           nosc1(i,j)=nosc(i,j)
         elseif (ii>0) then
           nosc1(i,j)=id(ii)
         endif
       enddo
     enddo
     nosc=nosc1; deallocate (nosc1)
!**
!**  change LL and LLL according id(old)=new
!**
     allocate (L1(ndf),LL1(ndf)); L1=0
     do i=2,ndf
        I1=LL(i-1)+1; I2=LL(i); ii=id(i)
        do j=I1,I2
           jj=id(LLL(j))
           if (ii>jj) then;       L1(ii)=L1(ii)+1
           elseif (ii<jj) then ; L1(jj)=L1(jj)+1
           endif
        enddo
     enddo
!**
     L1(1)=0
     DO I=NDF,2,-1
        L1(I)=L1(I-1)
     ENDDO
!**
     DO i=2,ndf
       L1(I)=L1(I)+L1(I-1)
     ENDDO
!**
!**  change LLL according id(old)=new
!**
     do i=2,ndf
        I1=LL(i-1)+1; I2=LL(i); ii=id(i)
        do j=I1,I2
           jj=id(LLL(j))
           if (ii>jj) then
             L1(ii)=L1(ii)+1; LL1(L1(ii))=jj
           elseif (ii<jj) then
             L1(jj)=L1(jj)+1;LL1(L1(jj))=ii
           endif
        enddo
     enddo
!**
!**  Sort LLL
!**
     L1(1)=0
     do i=2,ndf
       j=L1(i-1)+1; JJ=L1(i)-L1(i-1)
       call sort(LL1(j),JJ) ! a Quick sort program
     enddo
!**
!**  Give back to LL and LLL
!**
     LL=L1; LLL=LL1; deallocate (L1,LL1)
!**
     end subroutine
   end module nbkLL
```