Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2015, Article ID 147286, 7 pages
http://dx.doi.org/10.1155/2015/147286

Research Article

A PETSc-Based Parallel Implementation of Finite Element Method for Elasticity Problems

Jianfei Zhang

College of Mechanics and Materials, Hohai University, 1 Xikang Road, Nanjing 210098, China

Correspondence should be addressed to Jianfei Zhang; zhjf77@gmail.com

Received 18 September 2014; Accepted 1 December 2014

Academic Editor: Chenfeng Li

Starting a parallel code from scratch is not a good choice for parallel programming finite element analysis of elasticity problems because we cannot make full use of our existing serial code and the programming work is painful for developers. PETSc provides libraries for various numerical methods that can give us more flexibility in migrating our serial application code to a parallel implementation. We present the approach to parallelize the existing finite element code within the PETSc framework. Our approach permits users to easily implement the formation and solution of linear system arising from finite element discretization of elasticity problem. The main PETSc subroutines are given for the main parallelization step and the corresponding code fragments are listed. Cantilever examples are used to validate the code and test the performance.

1. Introduction

Elasticity is a general problem in solid mechanics and it is fundamental for civil, structural, mechanical, and aeronautical engineering and also other fields of engineering and applied science. In the numerical techniques to solve elasticity problems, finite element method (FEM) [1] is one of the important methods. Throughout the whole finite element analysis procedure, the equations formation and solution are the main time-consuming parts. In the case of large-scale finite element analysis, most of the computation time is spent on the equations solution. The performance of the equations solver determines the overall performance of finite element code. So the finite element equations solver is attracting much more interest than other components in parallelization of finite element computation. Various types of parallel solvers for sparse matrices arising from finite element analysis have been developed. They are classified into two categories: the direct and the iterative solvers. Direct solvers have many weak points in the parallel processing of the finite element analysis for very large-scale problems. Generally, direct solvers require much larger storage and more operation counts than iterative solvers. Furthermore, direct solvers need much more communications among processors and are generally more difficult to parallelize than iterative solvers. Because of these difficulties and the disadvantages of direct solvers, most of researches on parallel finite element analysis focus on iterative methods and iterative solvers have been installed into more and more large-scale parallel finite element code [2–4].

In this paper, parallel finite element computation for elasticity problems is implemented based on the Portable, Extensible Toolkit for Scientific Computation (PETSc) [5] and the parallel code is developed and tested. The remainder of this paper is organized as follows. Section 2 reviews the aspects related to PETSc, FEM, and iterative solution. Sections 3 and 4 present the detailed parallel implementation of finite element method for elasticity problems with PETSc, including finite element equations formation and solution. In Section 5 the performance of the code is comprehensively measured with different test examples. Finally, Section 6 summarizes the main conclusions.

2. Backgrounds

2.1. PETSc. PETSc is a suite of data structures and routines that provide frames to develop large-scale application codes on parallel (and serial) computers. It consists of parallel linear

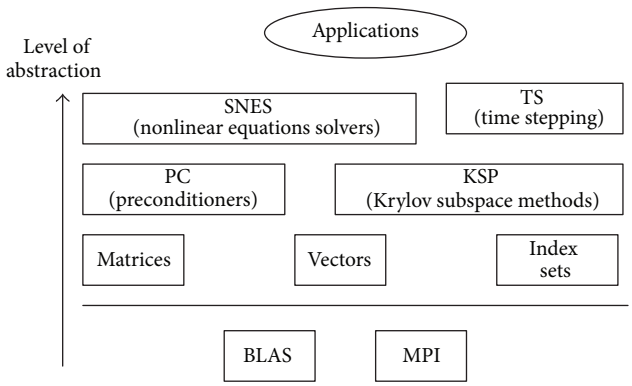

Figure 1: Organization of the PETSc libraries.

and nonlinear equation solvers and time integrators that can be used in application codes written in many languages, such as Fortran, C, and C++. PETSc provides a variety of libraries each of which manipulates a particular family of objects. These libraries are organized hierarchically as in Figure 1, enabling users to employ the most appropriate level of abstraction for a particular problem. The operation performed on the objects has abstract interface which is simply a set of calling sequences, which makes the use of PETSc easy during the development of large-scale scientific application codes. Thus, PETSc provides a powerful set of tools for efficient modeling scientific applications and building large-scale applications on high-performance computers.

2.2. Finite Element Method for Elasticity. The displacement-based finite element method introduces an approximation for the displacement field in terms of shape functions and uses a weak formulation of the equations of equilibrium, strain-displacement relations, and constitutive relation to arrive at the linear system

$$\mathbf{Ku} = \mathbf{F}, \tag{1}$$

where $\mathbf{u}$ is the vector of unknown nodal displacements, $\mathbf{F}$ is the vector of nodal forces, and $\mathbf{K}$, the structure stiffness matrix, is given by

$$\mathbf{K} = \sum_e \mathbf{C}_e^T \mathbf{k}^e \mathbf{C}_e, \tag{2}$$

$$\mathbf{F} = \sum_e \mathbf{C}_e^T \mathbf{f}^e, \tag{3}$$

where selective matrix $\mathbf{C}_e$ plays a transforming role between the local number and the global number of degrees of freedom (DOF). $\mathbf{k}_e$ is the element stiffness matrix and $\mathbf{f}_e$ is the element nodal forces, which are both computed with integration over each element.

The structure stiffness matrix in (2) is a sparse and symmetric positive definite (SPD) of dimension $n \times n$, where n is the total number of degrees of freedom (DOF).

2.3. Krylov Subspace Methods. Krylov subspace methods [6] are currently the most important iterative techniques for solving large linear systems. These techniques are based on projection processes onto Krylov subspaces. For solving the linear system $\mathbf{Ax} = \mathbf{b}$, a general projection method extracts an approximate solution $\mathbf{x}_m$ from an affine subspace $\mathbf{x}_0 + K_m$ of dimension m by imposing the Petrov-Galerkin condition $\mathbf{b} - \mathbf{Ax}_m \perp L_m$, where L_m is another subspace of dimension m. A Krylov subspace method is a method for which the subspace K_m is the Krylov subspace $K_m(\mathbf{A}, \mathbf{r}_0) = \text{span}(\mathbf{r}_0, \mathbf{Ar}_0, \mathbf{A}^2\mathbf{r}_0, \ldots, \mathbf{A}^{m-1}\mathbf{r}_0)$, where $\mathbf{r}_0 = \mathbf{b} - \mathbf{Ax}_0$. The different versions of Krylov subspace methods arise from different choices of the subspace L_m. The conjugate gradient (CG) algorithm is one of the best known iterative techniques for solving sparse symmetric positive definite linear systems. It is a realization of an orthogonal projection technique onto the Krylov subspace K_m.

Although the methods are well founded in theory, they are likely to suffer from slow convergence for problems from practical applications such as solid mechanics and fluid dynamics. Preconditioning is an important means for improving Krylov subspace methods in these applications. It transforms the original linear system into one with the same solution, but which is easier to solve.

A mathematically equal preconditioned linear system is expressed as follows:

$$\mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b}, \tag{4}$$

where $\mathbf{M}$ is a preconditioner. One simple way to construct preconditioners is to split $\mathbf{A}$ into $\mathbf{A} = \mathbf{M} - \mathbf{N}$. In theory, any splitting with nonsingular $\mathbf{M}$ which is close to $\mathbf{A}$ in some sense can be used.

The Jacobi preconditioner [7] is a commonly used preconditioner with the form of $\mathbf{M} = \text{diag}(\mathbf{A})$. The SOR or SGS preconditioning matrix [6] is of the form $\mathbf{M} = \mathbf{LU}$, where $\mathbf{L}$ and $\mathbf{U}$ are the lower triangular part and the upper triangular part of $\mathbf{A}$, respectively. Another simple way of defining a preconditioner is incomplete factorization of the matrix $\mathbf{A}$ [8]. These incomplete LU factorization (ILU) preconditioners perform decomposition of the form $\mathbf{A} = \mathbf{LU} - \mathbf{R}$, where $\mathbf{L}$ and $\mathbf{U}$ are the lower and upper parts of $\mathbf{A}$ with the same nonzero structure and R is the residual of the factorization. Because classical preconditioners, such as ILU and SSOR, have limited amount of parallelism, a number of alternative techniques have been developed that are specifically targeted at parallel environments, for example, additive Schwarz preconditioners [9] and multigrid preconditioners [10].

3. Finite Element Equations Assembly

According to the theory of finite element method, the calculation of element stiffness matrix and element nodal load vector only needs the information of the local element. So they can be easily parallelized without any communication. The global stiffness matrix and global nodal load vector are assembled with all element stiffness matrices and element nodal loads according to the relationship between the local number and the global number of DOFs. If nonoverlapping domain

decomposition is used, the computation of the entities of the global stiffness matrix and global nodal load relating to the interface DOFs needs data exchange between adjacent subdomains.

To implement finite element equations assembly in parallel, the first step is to partition the domain into subdomains. The domain partitioning can be done by some graph partition libraries, such as Metis, which makes loads over processes balanced. The Metis subroutine is

$$\begin{aligned} &\text{METIS\_PartMeshDual (int } ne, \text{ int } nn, \text{ int} * elmnt, \\ &\qquad \text{int} * etype, \text{ int} * numflag, \\ &\qquad \text{int } nparts, \text{ int } edgecut, \\ &\qquad \text{int} * epart, \text{ int} * npart), \end{aligned} \tag{5}$$

where ne and nn are numbers of elements and nodes, $elmnt$ is the element node array, $etype$ indicates the element type, $numflag$ indicates the numbering scheme, $nparts$ is the number of the parts, $eldgecut$ stores the number of the cut edges, $epart$ stores the element partition vector, and $npart$ is the node partition vector.

After domain partition, subdomains are assigned to processes and the element stiffness matrices and load vectors of the subdomains are calculated concurrently. These element stiffness matrices and load vectors are then accumulated into the global stiffness matrix. To contain global stiffness matrix **K**, we must use PETSc calls to create a matrix object. Because the stiffness matrix is a sparse symmetric matrix, AIJ format (CSR) is used to store it. There are several ways to create a matrix with PETSc. We can call MatCreateMPIAIJ to create a parallel matrix. The command is

$$\begin{aligned} &\text{MatCreateMPIAIJ (MPI\_Comm } comm, \text{ PetscInt } m, \\ &\qquad \text{PetscInt } n, \text{ PetscInt } M, \text{ PetscInt } N, \\ &\qquad \text{PetscInt } d\_nz, \text{ const PetscInt } d\_nnz[], \\ &\qquad \text{PetscInt } o\_nz, \text{ const PetscInt } o\_nnz[], \\ &\qquad \text{Mat} * A), \end{aligned} \tag{6}$$

where m, M, and N specify the number of local rows and number of global rows and columns, n is the number of columns corresponding to a local parallel vector, d_nz and o_nz are the number of diagonal and off-diagonal nonzeros per row, and d_nnz and o_nnz are optional arrays of nonzeros per row in the diagonal and off-diagonal portions of local matrix.

Because each node has multiple degrees of freedom in the finite element discretization of elasticity problems, we also can create a sparse parallel matrix in block AIJ format (block compressed row) by the command

$$\begin{aligned} &\text{MatCreateMPIBAIJ (MPI\_Comm } comm, \text{ PetscInt } bs, \\ &\qquad \text{PetscInt } m, \text{ PetscInt } n, \text{ PetscInt } M, \\ &\qquad \text{PetscInt } N, \text{ PetscInt } d\_nz, \\ &\qquad \text{const PetscInt } d\_nnz[], \text{ PetscInt } o\_nz, \\ &\qquad \text{const PetscInt } o\_nnz[], \text{ Mat} * A), \end{aligned} \tag{7}$$

where bs is the size of block, d_nz and o_nz are the numbers of diagonal and off-diagonal nonzero blocks per block row, and d_nnz and o_nnz are optional arrays of nonzero blocks per block row in the diagonal and off-diagonal portions of local matrix.

Since dynamic memory allocation and copying between old and new storage are very expensive, it is critical to preallocate the memory needed for the sparse matrix. This preallocation of memory is very important for achieving good performance during matrix assembly of an AIJ matrix or a BAIJ matrix, as this reduces the number of allocations and copies required. For a given finite element mesh, we can loop the neighboring nodes of each node to determine the nonzero structure of each block row. So it is easy to determine the d_nz and o_nz in subroutine MatCreateMPIAIJ or MatCreateMPIBAIJ before computation. The Fortran code to preallocate memory for MPIBAIJ stiffness matrix is listed in Algoritm 1.

After the matrix has been created, it is time to insert values. When implemented with PETSc, each process loops the elements in its local domain, computes the element stiffness matrices, and assembles them into global matrix without regard to which process eventually stores them. This can be done in two ways with PETSc, by either inserting a single value or inserting an array of values. In order to accumulate element stiffness matrices into global matrix, we can use the below subroutine to insert or add a dense subblock of dimension $m \times n$ into the stiffness matrix:

$$\begin{aligned} &\text{MatSetValues (Mat } mat, \text{ PetscInt } m, \\ &\qquad \text{const PetscInt } idxm[], \text{ PetscInt } n, \\ &\qquad \text{const PetscInt } idxn[], \\ &\qquad \text{const PetscScalar } v[], \text{ InsertMode addv}), \end{aligned} \tag{8}$$

where v is a logically two-dimensional array of values, m and $idxm$ are the number of rows and their global indices, n and $idxn$ are the number of columns and their global indices, and $addv$ is the operation of either ADD_VALUES or INSERT_VALUES, where ADD_VALUES means adding values to any existing entries and INSERT_VALUES means replacing existing entries by new values. For stiffness matrix assembly, the contributions from related elements are accumulated into global entities and ADD_VALUES is used.

Also, there are similar procedures to create vectors and insert values into those vectors to store global nodal load

```
! compute the neighbouring nodes of nodes and store them in array ndcon( ) and ndptr( )
do j=mlow+1,mhigh
   jj=j−mlow
   dnn(jj)=0
   ist=ndptr(j)
   ied=ndptr(j+1)−1
   nnd=ied-ist+1
   do i=ist,ied
      icon=ndcon(i)
      if((icon>=mlow+1).and.(icon<=mhigh)) then
          dnn(jj)=dnn(jj)+1
      endif
   enddo
   onn(jj)=nnd-dnn(jj)
enddo
```

Algorithm 1

```
      ! create matrix and vectors
      call MatCreateBAIJ(PETSC_COMM_WORLD,nodof,mmdof,          &
                         mmdof,nndof,nndof,0,dnn,0,onn, K,ierr)
      call MatSetOption(AK,MAT_SYMMETRY_ETERNAL,PETSC_TRUE,ierr)
      call VecCreateMPI(PETSC_COMM_WORLD, mmdof, nndof,F,ierr)
      call VecDuplicate(F,x,ierr)
      call VecSetOption(F,VEC_IGNORE_NEGATIVE_INDICES,PETSC_TRUE,ierr)
      ! Insert values into matrix and vector
    elements_loop: DO iel=1,nels
       call MatSetValues(AK,ndof,g_ele,ndof,g_ele,ke,ADD_VALUES,ierr)
       call VecSetValues(F,ndof,g_ele,fe,ADD_VALUES,ierr)
    END DO elements_loop
    ! Assembling
    call MatAssemblyBegin(AK,MAT_FINAL_ASSEMBLY,ierr)
    call MatAssemblyEnd(AK,MAT_FINAL_ASSEMBLY,ierr)
    call VecAssemblyBegin(F,ierr)
    call VecAssemblyEnd(F,ierr)
```

Algorithm 2

F and nodal displacement. PETSc currently provides two basic vector types: sequential vector and parallel (MPI based) vector. The created vector is distributed over all processes. Any process can set any components of the vector and PETSc insures that they are automatically stored in the appropriate locations.

The Fortran code to create parallel matrix and vector to store stiffness matrix and nodal vectors is listed in Algorithm 2.

Note that the valuation of the element stiffness and nodal load is ignored in the above code for simplicity.

4. Solution of Assembled System

After the final assembly of the stiffness matrix and nodal load vector, the system is now ready to be solved. PETSc provides easy and efficient access to all of the package's linear system solvers with the object KSP, that is, the heart of PETSc. We here combine CG methods and preconditioners to solve the linear system (1) from finite element discretization. Because KSP provides a simplified interface to the lower-level KSP and PC modules within the PETSc package, we can easily implement this preconditioned CG solver.

The first step to solve a linear system with KSP is to create a solver context with the command

$$\text{KSPCreate}\left(\text{MPI\_Comm } comm, \text{KSP} * ksp\right), \tag{9}$$

where *comm* is the MPI communicator and *ksp* is the new solver context. Before solving a linear system with KSP, we must call the following routine to make the matrices associated with the linear system:

$$\text{KSPSetOperators}\left(\text{KSP } ksp, \text{Mat } Amat, \text{Mat } Pmat, \text{MatStructure } flag\right), \tag{10}$$

where the matrix *Amat* defines the linear system and *Pmat* represents the matrix from which the preconditioner is to be constructed. It can be the same as the matrix that defines

```
call KSPCreate(PETSC_COMM_WORLD,ksp,ierr)
call KSPSetOperators(ksp,AK,AK,DIFFERENT_NONZERO_PATTERN,ierr)
call KSPSetType(Ksp,KSPCG,ierr)
call KSPCGSetType(Ksp,KSP_CG_SYMMETRIC,ierr)
call KSPGetPC(ksp,pc,ierr)
call PCSetType(pc,PCJACOBI,ierr)
tol = 1.0d-7
call KSPSetTolerances(ksp,tol,PETSC_DEFAULT_DOUBLE_PRECISION,        &
     PETSC_DEFAULT_DOUBLE_PRECISION,PETSC_DEFAULT_INTEGER,ierr)
call KSPSolve(ksp,F,x,ierr)
```

Algorithm 3

the linear system. The argument $flag$ indicates information about the structure of preconditioner matrix during successive solutions.

To solve a linear system, we set the right-hand side and solution vectors by calling the routine

$$\text{KSPSolve}\left(\text{KSP } ksp, \text{Vec } b, \text{Vec } x\right), \tag{11}$$

where b and x, respectively, denote the rhs and solution vectors.

When solving by Krylov subspace methods with PETSc, a number of options are needed to set. First of all, we need set the Krylov subspace method to be used by calling the command

$$\text{KSPSetType}\left(\text{KSP } ksp, \text{KspType } method\right). \tag{12}$$

Due to the slow convergence of Krylov subspace methods for the linear system arising from practical elasticity applications, preconditioning is usually combined to accelerate the convergence rate of the methods. To employ a particular preconditioning method, we can set the method with the subroutine

$$\text{PCSetType}\left(\text{PC } pc, \text{PCType } method\right). \tag{13}$$

Each preconditioner may have a number of options to be set. We can set them with different routines [11]. During solution of preconditioned Krylov method, the default convergence test is based on the l_2-norm of the residual. Convergence is decided by three values: the relative decrease of the residual norm to that of the right-hand side, $rtol$, the absolute value of the residual norm, $atol$, and the relative increase of the residual, $dtol$. These parameters and the maximum number of iterations can be set with the command

$$\text{KSPSetTolerances}\left(\text{KSP } ksp, \text{double } rtol, \text{double } atol, \text{double } dtol, \text{int } matrix\right). \tag{14}$$

Since the linear system derived from finite element discretization of elasticity problems is sparse and symmetric positive definite (SPD), the conjugate gradient (CG) algorithm is chosen here to solve it. For the conjugate gradient method with complex numbers, there are two slightly different algorithms subject to whether the matrix is Hermitian symmetric or truly symmetric. The default option is Hermitian symmetric. Because the solution of finite element equations uses symmetric version, we need indicate that it is symmetric with the command

$$\text{KSPCGSetType}\left(\text{KSP } ksp, \text{KSCGType KSP\_CG\_SYMMETRIC}\right). \tag{15}$$

The Fortran code to create the KSP context and perform the solution is as in Algorithm 3.

In this portion of code, the KSP method being used is the CG with JACOBI preconditioner. The convergence tolerance is set to $1.0e-7$.

5. Experimental Results

5.1. Test Platform and Examples. Our numerical experiments were conducted on a platform composed of 4 Intel Core i5-2450M CPUs @ 2.50 GHz and 4 GB RAM. The operation system is 64-bit CentOS 6. In this section we use a beam problem in bending to validate the PETSc-based finite element code and measure the performance of the linear system solver with different preconditioners. It is known that the number of iterations for convergent solutions to finite element equations from elasticity problems often differs, especially for the cases with different materials. This is because stiffness matrices are very singular when materials are very different. Figure 2 shows this cantilever problem. The size of the beam is 1 m × 1 m × 5 m. The beam is fixed at its left end and loaded by uniform pressure on its top surface. There are two cases with different material composition to be tested. One is of single uniform material and the other is of bimaterial with a strip of much lower Young's modulus than other portions. The base material's Young's modulus and Poisson ratio are of $2.0e8$ and 0.3. The strip material's Young's modulus and Poisson ratio are of $2.0e3$ and 0.3. Two three-dimensional hexahedral meshes with different numbers of elements have been generated and used in computation. The fine mesh has 10000 elements and 12221 nodes, and the coarse one has 80000 elements and 88641 nodes.

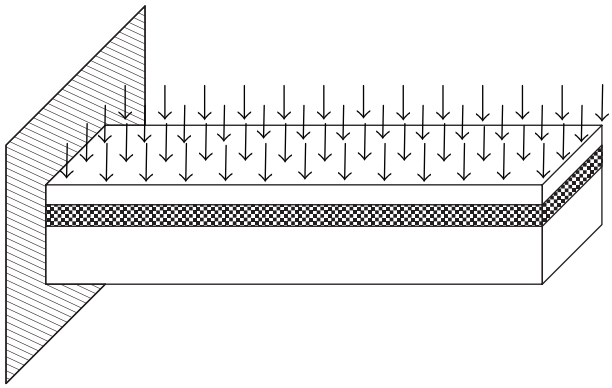

Figure 2: The sketch of the testing cantilever.

Table 1: Serial performance of the PCG (coarse mesh).

Preconditioner	Single material		Bimaterial	
	Iterations	Solution time (unit: sec.)	Iterations	Solution time (unit: sec.)
None	563	5.932	1645	17.215
Jacobi	507	5.407	1466	15.523
SOR	330	6.357	977	18.586
AMG	182	7.542	535	21.459

Table 2: Serial performance of the PCG (fine mesh).

Preconditioner	Single material		Bimaterial	
	Iterations	Solution time (unit: sec.)	Iterations	Solution time (unit: sec.)
None	1065	100.335	2958	276.267
Jacobi	1013	97.147	2763	261.196
SOR	649	106.462	1749	294.09
AMG	355	125.183	961	363.442

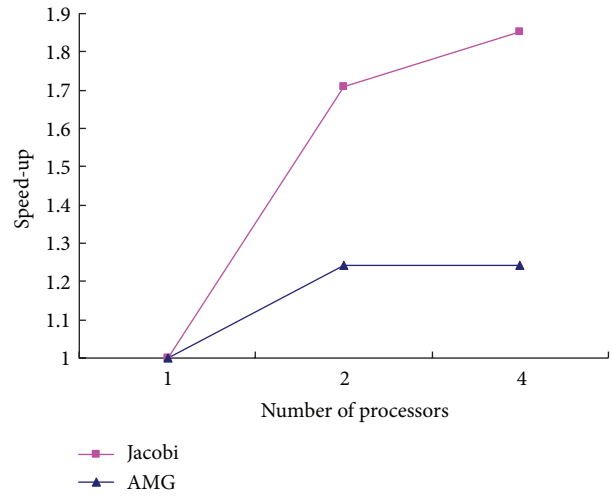

Figure 3: Speed-up of PCG solutions on coarse grid.

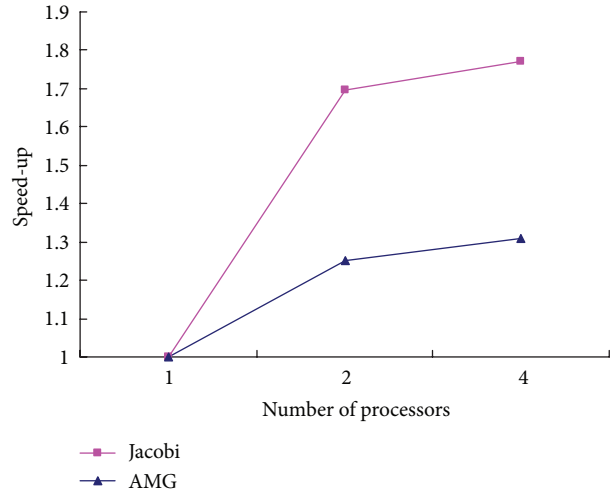

Figure 4: Speed-up of PCG solutions on fine grid.

5.2. Performance Tests. First, a comparative analysis of the serial performance of the CG method with different preconditioners has been carried out. Jacobi, SOR, and algebraic multigrid (AMG) preconditioners are tested. The options for these preconditioners are set to the default values. Tables 1 and 2 report the different serial performance results of preconditioned CG (PCG) solver for single material and bimaterial cases, respectively. From these tables, we can see that the AMG PCG converges the fastest, SOR PCG is the second fast one, Jacobi PCG follows them, and the convergence of the none preconditioned CG is the slowest. But from the view of running time, AMG takes the longest time, SOR is the second one, none preconditioner is the third one, and Jacobi one consumes the shortest running time. This is because the AMG and SOR need more preconditioning operations in each CG iteration. Though the number of CG iterations reduces, the overall running time increases.

Second, the parallel performance of the parallel finite element code has been measured. Figures 3 and 4 show parallel performance of the finite element linear system solution stage on coarse and fine meshes, respectively. As shown, the solution stage scales unsatisfactorily on the multicore computer for the communication bottleneck of the computing platform. The AMG preconditioner does not work better than the Jacobi one. This is because there are many options in AMG to tune for optimal performance [12].

6. Conclusions

We have integrated PETSc into the parallel finite element method for elasticity problems and developed the parallel code. Because PETSc includes libraries of numerical methods that can be applied directly to applications, the process of porting PETSc into the existing application codes becomes easier than developing with low-level parallel interfaces. In this work, we implement the main steps, formation and solution, of the parallel finite element computation of elasticity problems with PETSc subroutines. In the formation stage,

memory preallocation is conducted before computation to enhance the performance by avoiding the memory dynamic allocation and copying. For the solution, preconditioned CG method is used as a solver to the linear system derived from finite element discretization. PETSc provides various preconditioners for this solver. Numerical tests show that the formation stage can achieve good performance for its high parallelism and the solution stage scales not very well on the multicore computer for the communication problem.

In future work, this code will be migrated to distributed memory parallel systems and applied to practical problems. Furthermore, more preconditioners will be implemented with PETSc in the code to provide users with more options.

Conflict of Interests

The author declares that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grants nos. 51109072 and 11132003).

References

[1] O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu, *The Finite Element Method: Its Basis and Fundamentals*, Butterworth-Heinemann, Oxford, UK, 6th edition, 2005.

[2] M. A. Heroux, P. Vu, and C. Yang, "A parallel preconditioned conjugate gradient package for solving sparse linear systems on a Cray Y-MP," *Applied Numerical Mathematics*, vol. 8, no. 2, pp. 93–115, 1991.

[3] A. R. M. Rao, "MPI-based parallel finite element approaches for implicit nonlinear dynamic analysis employing sparse PCG solvers," *Advances in Engineering Software*, vol. 36, no. 3, pp. 181–198, 2005.

[4] Y. Liu, W. Zhou, and Q. Yang, "A distributed memory parallel element-by-element scheme based on Jacobi-conditioned conjugate gradient for 3D finite element analysis," *Finite Elements in Analysis and Design*, vol. 43, no. 6-7, pp. 494–503, 2007.

[5] S. Balay, S. Abhyankar, M. F. Adams et al., PETSc Web page, 2014, http://www.mcs.anl.gov/petsc.

[6] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, Pa, USA, 2nd edition, 2003.

[7] F. H. Lee, K. K. Phoon, K. C. Lim, and S. H. Chan, "Performance of Jacobi preconditioning in Krylov subspace solution of finite element equations," *International Journal for Numerical and Analytical Methods in Geomechanics*, vol. 26, no. 4, pp. 341–372, 2002.

[8] G. Bencheva and S. Margenov, "Parallel incomplete factorization preconditioning of rotated linear FEM system," *Journal of Computational and Applied Mechanics*, vol. 4, no. 2, pp. 105–117, 2003.

[9] S. C. Brenner, "Two-level additive Schwarz preconditioners for nonconforming finite element methods," *Mathematics of Computation*, vol. 65, no. 215, pp. 897–921, 1996.

[10] P. T. Lin, J. N. Shadid, M. Sala, R. S. Tuminaro, G. L. Hennigan, and R. J. Hoekstra, "Performance of a parallel algebraic multilevel preconditioner for stabilized finite element semiconductor device modeling," *Journal of Computational Physics*, vol. 228, no. 17, pp. 6250–6267, 2009.

[11] S. Balay, S. Abhyankar, M. F. Adams et al., "PETSc users manual," Tech. Rep. ANL-95/11, Revision 3.5, Argonne National Laboratory, 2014.

[12] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial*, SIAM, 2nd edition, 2000.