A General Purpose Sparse Matrix Parallel Solvers Package

Hong Q. Ding and Robert D. Ferraro

Jet Propulsion Laboratory, MS 169-315

California Institute of Technolog , Pasadena, California 91109

A parallel solvers package of three solvers with a unified user interface is developed for solving a range of sparse symmetric complex linear systems arising from discretization of partial differential equations based on unstructured meshs using finite element, finite difference and finite volume analysis. Once the data interface is set up, the package constructs the sparse symmetric complex matrix, and solves the linear system by the method chosen by the user, either a preconditioned bi-conjugate gradient solver, or a two-stage Cholesky LDL^T factorization solver, or a hybrid solver combining the above two methods.A unique feature of the solvers package is that the user deals with local matrices on local meshes on each processor. Scaling problem size N with the number of processors P with N/P fixed, test runs on Intel Delta up to 128 processors show that the bi-conjugate gradient method scales linearly with N whereas the two-stage hybrid method scales with $\sqrt{N}$.

1 Introduction

One of the common problems in scientific and engineering computations is the constrcution and solution of sparse linear systems arising from solving partial differential equations based on unstructured grids[1]. Many such examples occur in analysis by finite element, finite difference and finite volume methods. As the scale and complexity of the calculations grow, massively parallel computers are increasingly widely used in these calculations. .

Existing parallel sparse linear system solvers can be classified into to two distinctive classes(see [2] and references therein). One is the inerative conjugate-gradient type of solvers where the key parallel part is the matrix-vector product([2,3,4]); the other is parallel Cholesky factorization solvers using column/row based matrix distributions[5,6,2].

However, these solvers mostly are restricted to positive definite matrices. For a larger classs of symmetric complex matrix systems, few analytical results are known; The solutions are typically obtained by trying several different methods. First, one may try an iterative CG type of solver to see if it converges. If not, one may try a Cholesky (a LDL^T factorization, not the LL^T factorization for positive definite matrices) solver, which is more robust and stable, but requires more calculation and memory. This class of indefinite matrix problem raises the need for a solver package combining very different solvers with a single unified user interface so that user can easily switch to different solvers to try to solve the system.

For this reason, we developed a package to provide three different solvers with a unified user interface. After passing the geometry (mesh) data and edge information (the sparse matrix elements) to the package, the user may choose one of the solver best suited to the problem.

We emphasize that both the solution methods and the user interface make direct use of the underlying geometric mesh of the problem. We use a geometric domain decomposition[7,2], in contrast to most existing solvers which typically take a algebraic decomposition (an exception is described in [8] where the approach is close to ours). After the mesh is partitioned (could be carried out in parallel with a general-purpose partitioner independent of the solver package[9]), the user deals with a local mesh whose boundary points are either real boundary where boundary condition apply, or processor boundary which the user treat as interior points and the solver uses to connect the local patches into a global one.

This unique feature that user only deals with local mesh enables the user to assemble the sparse matrix elements just as on a sequential computer. Thus the construction of the matrix and the solution of the linear system can proceed entirely with the solver package in a parallel fashion. In contrast, all existing solvers deal with readily assembled matrix; there how to assemble those matrix in parallel remains an issue.

Below, we described the package in detail, focuing on the domain decomposition and the corresponding matrix structure. Key points in parallelization of the solution methods are explained in detail. Parallel scaling characteristics of the solvers running on Intel Delta are provided. More systematic analysis of the solvers will be given in a later report.

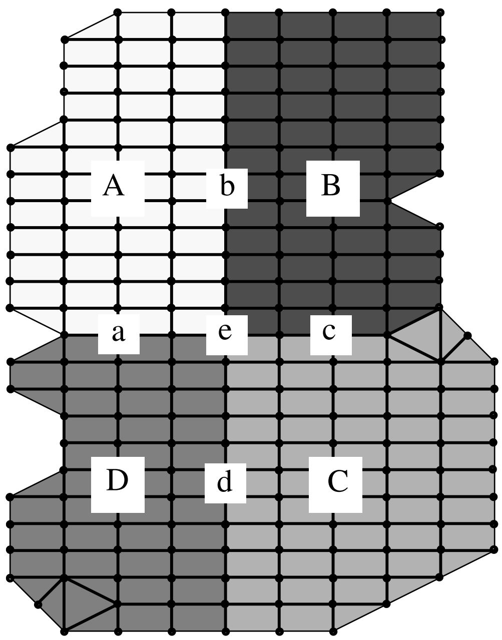


FIGURE 1. A Partitioned Mesh.

2 Domain decomposition

Domain decomposition defines the structure of the sparse coefficient matrix of the linear system.The requirement that matrix assembly involves only local data leads to a element-based decomposition, i.e., each finite element of the mesh should entirely belongs to one processor. The decomposition bases on the geometric mesh, not the graph associated with the sparse matrix. All points defining a finite element reside on one processor, thus no communication is required in calculating contributions to the matrix elements. The subdomain boundaries always go along edges on the mesh, and grid points sitting on these boundary edges are replicated (shared) among the processors who share the bounderies. This implies that the local mesh is a complete simply connected one. Thus the user can do all the usual sequential work on this local mesh without any communications. Grid points on processor boundaries serve as separators in standard sparse matrix techniques: they partition the mesh (and thus the sparse matrix) into disjoint regions (independent blocks). We will refer to them simple as ``boundary points" in contrast to these interior points. Note that a boundary point has ownership: only one processor "owns'' it, other processors share it.

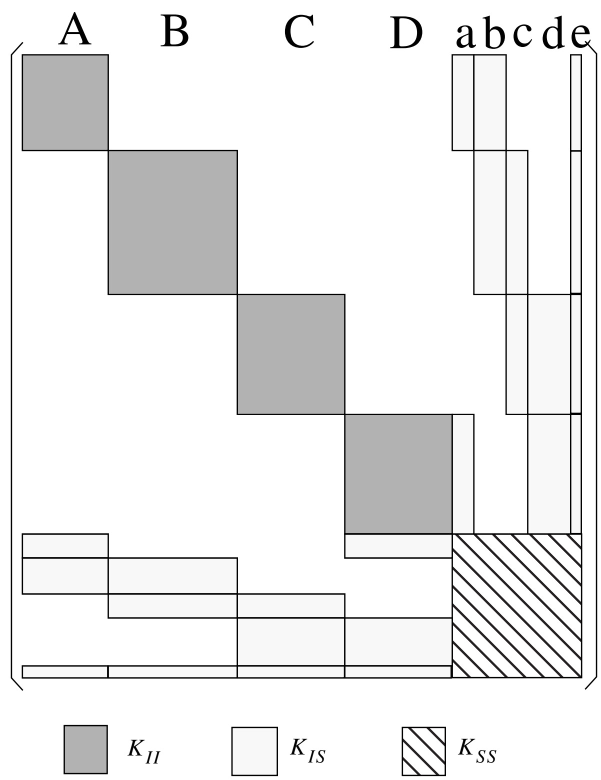


FIGURE 2. Sparse Matrix Structure.

An added benifit of this decomposition is that the adaptive or multilevel refinement of the mesh as a simple local sequential process, since all information with regard to the local mesh is locally available. The only constraint is that the edges/faces on the processor boundary should be refined consistently throught out all the allocated processors, e.g., creating new points only at mid-edge on those boundary edges. An algorithm exists to match the newly created nodes along the subdomain boundary, thus connecting local meshes into a global one.

3 Matrix Structure

The sparse matrix based on geometric decomposition (Fig.1) is shown in Fig.2. Only shaded area are possible to have nonzero matrix elements (actual storage schemes will be discussed later); the domain decomposition provides a natural ordering much like the one resulted from nested dissection based on the graph of the matrix. We can formerly write the global matrix as

$$\begin{bmatrix} K_{II} & K_{IS} \\ K_{SI} & K_{SS} \end{bmatrix} \begin{bmatrix} x_I \\ x_S \end{bmatrix} = \begin{bmatrix} f_I \\ f_S \end{bmatrix} \quad \text{(EQ 1)}$$

Block matrix K_{II}, standing for interior grids coupled to interior grids, is itself a block diagonal matrix (EQ 2)

$$K_{II} = \begin{bmatrix} K_{AA} & & & \\ & K_{BB} & & \\ & & K_{CC} & \\ & & & K_{DD} \end{bmatrix}$$

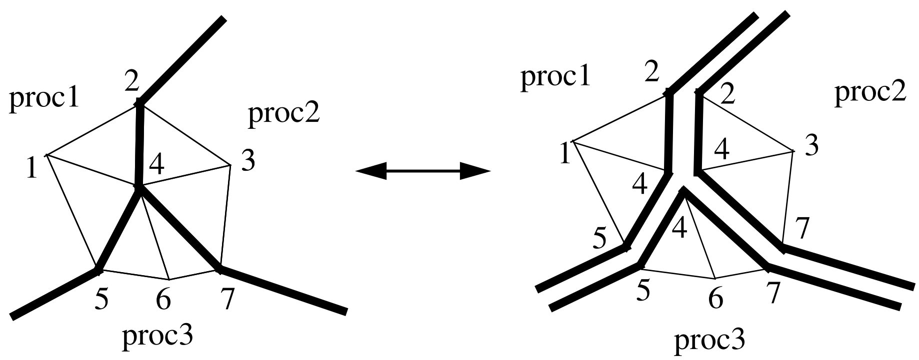
where K_{AA} stands for interior grids coupled to themselves in region A , etc; Each of the square block belongs to a unique processor. It is important to note that all block matrices such as here and below are themselves sparse matrices in nature; their sparse pattern will change from one application problem to another, and their storage may vary depending upon the solution method used. The matrix structures in Eqs(3.2-3.4) are the structures due to the domain decomposition used in the package, and it only indicates that there are vast amount of block matrix elements of the global matrix which are identically zero, and thus are never represented in the solver package.

Block matrix K_{IS} represents interior grids coupled to shared boundary grids, and has the following block structure,(EQ 3)

$$K_{IS} = \begin{bmatrix} K_{Aa} & K_{Ab} & & & K_{Ae} \\ & K_{Bb} & K_{Bc} & & K_{Be} \\ & & K_{Cc} & K_{Cd} & K_{Ce} \\ K_{Da} & & & K_{Dd} & K_{De} \end{bmatrix}$$

where, e.g., K_{Aa} stands for interior A coupled to shared boundary a . Each of the block elements belongs to a unique processor.

Block matrix K_{SS} represents shared boundary grids coupled to itself; each of its matrix elements are usualy split into several processors, as shown in Fig.3. The reason for the element split is twofold. Consider the global matrix element associated with the edge 2-4 in Fig.3. It has two contributions from mesh element 124 on processor p1 and mesh element 234 on processor p2. If we assign to only one processor, the other processor has to passing its contribution to



$$K_{44} = \overbrace{K_{44}^{\Delta124}+K_{44}^{\Delta145}}^{\text{proc 1}} + \overbrace{K_{44}^{\Delta234}+K_{44}^{\Delta347}}^{\text{proc 2}} + \overbrace{K_{44}^{\Delta476}+K_{44}^{\Delta465}}^{\text{proc 3}}$$

$$K_{24} = K_{24}^{\Delta124} + K_{24}^{\Delta234} + 0$$

Figure 3. Matrix Element Split. Thick lines indicate processor boundaries.

this processor, creating a special situation the user of the package has to handle. We instead let both processors have storage for (as shown in Fig.3) so that matrix construction is local with no communication required..

4 Local Matrices

The element split of thus allows the user to construct/ assemble the stifness matrix in a entirely local manner (Fig.4). On processor p0, the user has a local mesh A , and he construct the matrix based on this local mesh; the only consideration due to the fact that mesh A is a subdomain of a global mesh is that the user has to separate interior points on A from boundary points ∂A . Thus the local linear system is (EQ 4)

$$\begin{bmatrix} K_{AA} & K_{A\partial A} \\ K_{\partial AA} & K_{\partial A\partial A} \end{bmatrix} \begin{bmatrix} x_A \\ x_{\partial A} \end{bmatrix} = \begin{bmatrix} f_A \\ f_{\partial A} \end{bmatrix}$$

where

$$K_{A\partial A} = \begin{bmatrix} K_{Aa} \; K_{Ab} \; K_{Ae} \end{bmatrix}$$

because ∂A consists of a, b, e . These K_{Aa}, K_{Ab}, K_{Ae} appear in K_{IS} in Eq.3.2. On processor p1, user construct a similar equation based on subdomain B . Note that ∂B consists of b, c, e and has an overlap with ∂A . These overlapped matrix elements are added up indirectly in the PBCG; they are added up explicitly in the two-stage factorization solver.

These description of the global matrix leads to the local matrix stored on each processor as shown in Fig.4 for processors p0 and p1. Note the duplication of K_{bb} on two processors: $K_{bb}^{[p0]}$ contains contributions from elements on p0 and $K_{bb}^{[p1]}$ contains those on p1. Similarly, K_{ee} is duplicated on all 4 processors. Only in the two-stage Cholesky factorization solver these pieces are explicitly summed into one piece by the solver; in both the PBCG and the two-stage hybrid solvers, they are summed up indirectly.

The labeling of processor boundaries a, b, c, d, e here is for illustration purpose. In actual implementation, all of them are treated same. Facts such as points in e are replicated on all 4 processors, or points in a are replicated on processors p0 and p1, etc, are recorded in a proc-list associated with each point: a processor-id is stored in the list if the point resides on the processor.

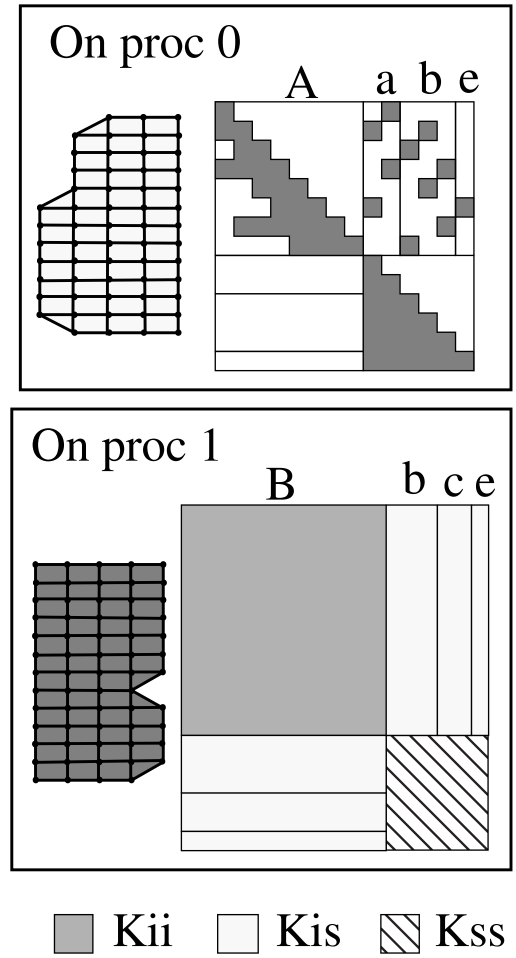


Figure 4. Local matrices. On proc 0, we also indicates the storage schemes used in two-stage Cholesky factorization solver.

It is important to note that the matrix structure on processor p0 as showned in Fig.7 is the same matrix structure one would get as if he is dealing with the local mesh on a sequential computer; the only difference is that the grids on processor boundary are not real boundary grids and thus a variable (unknown) is assigned to it. This is the central idea throughout the designe of the package: user deals with the local mesh the same way as he deals with a global mesh on a sequential computer. This provides complete freedom for user to handle and modify the problem at hands.

5 Preconditioned Bi-conjugate Gradient Solver

For complex symmetric matrix, we implemented the bi-conjugate gradient method[10]. This popular iterative solver is fast and preserves the sparsity of the matrix, thus saving computer memory. However, the iteration does not always converges, depending on the properties of the matrix. The solver involves two kinds of communications. One is the dot product of two global vectors, which are implemented by using global_sum() type of communication subroutines. The other is a global matrix-vector multiplication, which is implemented as a local matrix multiplication plus an inter-processor communication called globalize(). Suppose the matrix-vector multiplication is $\tilde{x} = Kx$. Let's look at the component $\tilde{x}_2$ (see Fig.3). Noting that K_{24} is split on processors p1 and p2, We have

$$\tilde{x}_2 = K_{21}x_1 + (K_{24}^{[p1]} + K_{24}^{[p2]})x_4 + K_{23}x_3$$
$$= (K_{21}x_1 + K_{24}^{[p1]}x_4) + (K_{24}^{[p2]}x_4 + K_{23}x_3)$$

i.e, this becomes local matrix multiplications on relevant processors,

$$\tilde{x}^{[p1]} = K^{[p1]}x^{[p1]}, \quad \tilde{x}^{[p2]} = K^{[p2]}x^{[p2]}$$

plus the summation of the resulting local vectors on relevant processors. Because point 2 is replicated on processors p1 and p2, the variable associated with point 2 will be the sum of the appropriate entries on processors p1 and p2; the sum is then replicated on the two processors. Similarly, The variable associated with point 4 will be the sum of entries on p1, p2 and p3. Etc. This can be implemented as the non-owned boundary points sending value to the owned boundary points; the sum is done at the owner processor and then broadcasted back to those non-owned processors. This summation and replication process on shared boundary points is called globalization of vector .

The storage in the CG type solution is non-zero only, i.e., only non-zero matrix elements are stored. This minimum storage scheme preserves the sparsity of the matrix. At present, only diagonal preconditioning is implemented. Incomplete Cholesky preconditioners are not implemented, because their usefulness for the symmetric complex matrices has not been documented.

6 Two-stage Cholesky factorization solver

The Cholesky factorization method proceeds in two stages, using the domain decomposition described above. The separation of interior girds from boundary grids leads to the sparsity pattern in the global matrix (see Fig.2), which greatly facilitates this process. Starting with the global equation Eq.3.1 Since each block matrix in K_{II} reside entirely on one processor, we can do a complete local factorization to obtain its inverse, thus obtaining K_{II}^{-1} without any inter-processor communication. Solving the equation with respect to x_I and substitute into the equationwithrespecttoxS,weobtainthereducedequation(EQ 5)

$$(K_{SS} - K_{SI}K_{II}^{-1}K_{IS})x_S = f_S - K_{S1}K_{II}^{-1}f_I$$

or simply $\tilde{K}_{SS}x_S = \tilde{f}_S$ Since $f_I, f_S, K_{SI}, K_{II}^{-1}$ are known, $\tilde{K}_{SS}, \tilde{f}_S$ can be calculated. This reduced equation deals with only the shared boundary grids, as schematically shown in Fig.5.; all the interior grids have been eliminated. Thus one we have reduced the original larger system to a relatively much smaller system. The reduced matrix is still symmetric and preserves the positive-definiteness, i.e., if the original matrix is positive-definite, the reduced matrix is also positive-definite. This property is important both for the two-stage Cholesky solver described in this section, and for thetwo-stagehybridsolverdescribedinthenextsection.(EQ 6)

The first stage of the solver is to define and calculate local reduced equations on each processors independently: for example,we hav

$$K_{\partial A\partial A} - K_{\partial AA}K_{AA}^{-1}K_{A\partial A})x_{\partial A} = f_{\partial A} - K_{\partial AA}K_{AA}^{-1}f_A \quad \text{(EQ 7)}$$

or simply

$$\tilde{K}_{\partial A\partial A} \cdot x_{\partial A} = \tilde{f}_{\partial A}$$

on processor p0 for the boundary of subdomain A. That this local version is possible is due to the important fact that every entry in uniquely belongs to one processor (unlike the element-splited entries in), and so are every block matrices in. One can view this equation as directly obtained from Eq.4.1. on a sequential architecture. Local reduced equations on other processors are similarly calculated.

During the local factorization we use a simple variable-banded (also called profiling or skyline) scheme in which on each row, we store every element between the first nonzero element and the diagonal element (see Fig.4). With this storage scheme, the local factorization of proceeds easily. The rectangle matrices coupling between the interior and boundary, such as $K_{A\partial A}$ (see Fig.4) are stored simply as nonzero only (same as in the iterative method cases), because they are only involved in matrix multiplications. The boundary block matrix $K_{\partial A\partial A}$ (see Fig.4) is stored as a lower triangular dense matrix. Because these matrix elements are typically split into several processors as explained previously, a dense matrix representation makes it simple to keep track of indices across many processors in later reshuffle of data.

The second stage is to solve the reduced equation via a parallel Cholesky factorization. First the global reduced equation is explicitly summed up via inter-processor communications as explained in Sec.4, and so is the global vector on the shared boundaries. At the same time, the resulting and are redistributed with each processor holds several rows of in a cyclical fashion, much like a dense matrix factorization. The symmetric complex matrix is stored as a variable banded matrix, i.e., on each row storage is allocated from the first nonzero element to the diagonal. Numerical factorization and back substitution are performed in parallel.

In summary, the reduction of global matrix are calculated through local matrices based on local meshes and then followed by the inter-processor summation which is structured enough to be handled by the solver. The simple correspondence in matrix blocks between the global and local reduced equations are due to the particular matrix distribution method we choose.

7 Two-stage hybrid solver

We can also solve the reduced global equation (boundary grids), Eq.(6.3), by using the Preconditioned Bi-Conjugate Gradient method in Sec.5. Identical routines are used, except that here we are dealing with only boundary points and the local matrix-vector is , instead of in Sec.5. Explicit construction of Eq.6.8 is not needed. Since the dimension of is far more smaller than , we expect a better convergence property for PBGC on this reduced system. Also, this hybrid method has the best scaling property; it scales as when the problem size increases in proportion to the total number of processor .

8 Usage

The solver is designed for minimal requirements on the user side. The user prepares for one data interface to the package, and solve the system by invoking one of the three solvers.The interface between the user and the package has two parts. First, the user passes to the package the list of local grid points, including usual entries like grid-id and coordinates etc., and a proc-list which contains the proc-id's which share the grid. These information come with the partitioning of the unstructured mesh in a separated procedure[9] outside the solver package.

Second, based on the list of local sets of grid points (the elements in finite element methods or the stencils in finite difference methods), the user calculates the problem-dependent matrix elements associated with each local sets, as if he is in a single processor environment. The user passes the local sets information and the contributions to the global sparse matrix elements by two rounds of subroutine calls. (The user does not need to know how to construct the sparse matrix). The first round of calls to subroutine allocK() with the local sets information is made to set up the local and global indices and to allocate the storage needed. A second round of calls to subroutine addKf() is made to pass to the solver package the local sets information and the user-calculated sparse matrix element entries and right-hand-side entries. Typical boundary conditions are specified by the user through subroutine calls to addKf(). These calls completes the assemble of the sparse matrix elements.

The linear system is then solved by one of the three methods chosen by the user. However, since the storage scheme depends on the solution method, user has to specify the solution at the beginning, by linking in one of three different subroutine sets. Note, however, the interface the user specified remains identical.

9 Performance

We have completed the solver package. The solver package is applied both to a finite element solution of electromagnetic wave scattering from conducting sphere, and to a finite difference solution to a static heat distribution problem governed by the Poisson equation.

In Fig.5, we plotted the scaling behavior of the solver for the heat problem (data for two-stage Cholesky solver will be presented in a later report). We let the problem size scale with number of processors while fixing 1600 grid points per processor. The data shown is the ratio of the timing for the entire problem on processors vs. solving a 1600-grid problem on one processor. The preconditioned bi-conjugate gradient methods scales as , where one comes from the matrix-vector product (sparsity reduced to) and another comes the order iterations. Since , we expect the scaling be linear in , and our data supported this scaling clearly. The two-stage hybrid solver first does a sequential factorization on the interior points and then does a parallel bi-conjugate gradient method for those boundary points. The first part remains a constant for the fixed size per processor, and the second part deals with those boundary points which increase as square root of the total points. Once the first part saturates, the total time for the hybrid method should increase as , as our data indicated.

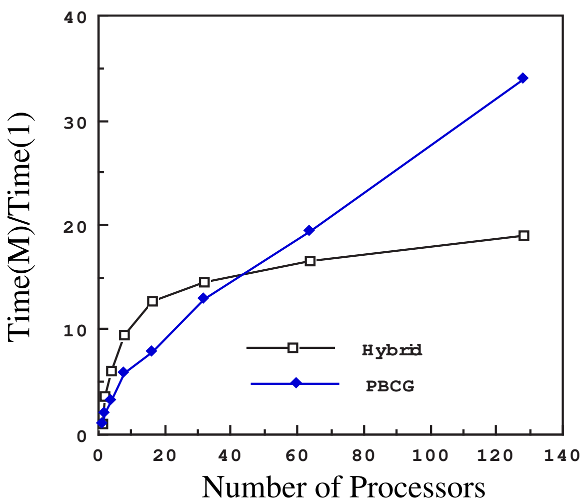


Figure 5. Scaling Characteristics of the solvers on Intel Delta.

10 Conclusions

We have implemented a software package for constructing and solving the sparse coefficient matrix linear systems arising from solving partial differential equations based on unstructured grids. The sparse symmetric complex linear system can be solved by either a preconditioned bi-conjugate gradient solver, or by a two-stage Cholesky factorization solver, or by a hybrid solver combining both. The interface of the solver is designed so that one interface fits to all three different solvers. Scaling test runs for fixed problem-size/processor indicate good scaling behavior of the two-stage hybrid method.

Acknowledgments. This work is funded under the NASA HPCC ESS Project. We thank Caltech Concurrent Supercomputing Facility for the use of the Intel Delta. We thank G.A. Lyzenga, J. Parker, N. Jacobi, T.G. Lockhart, J. Patterson and others who contributed to this project at earlier stage.

References

1. I.S. Duff, A.M. Erisman, and J.K. Reid. Direct Methods for Sparse Matrices, Oxford University Press, London, 1986.
2. V. Kumar, A. Grama, A. Gupta and G. Karypis, Introduction to Parallel Computing, Benjamin/Cummings, Redwood City, CA, 1994. Chap.11.
3. Solving Problems on Concurrent Processors, G.C. Fox, M.A. Johnson, G.A. Lyzenga, S.W. Otto, J.K. Salmon and D.W. Walker, Chap.8, Vol.1, Prentice Hall, Englewood Cliffs, New Jersey, 1988. Chap.7.
4. R.Cook, and J. Sadecki, Sparse Matrix Vector Product on Distributed Memory MIMD architectures, in Proceedings of 6th SIAM Conference on Parallel Processing for Scientific Computing, 1993, p.429.
5. C. Ashcraft, S.C.Eisenstat, and J. W-H. Liu, A fan-in algorithm for distributed sparse numerical factorization, in SIAM J. Scient. Statis. Comput. p.593, 1990.
6. E. Rosenberg and A. Gupta, An efficient block-oriented approach to parallel sparse Cholesky factorization, in Proceedings of Supercomputing 93, pp.503-512.
7. R.D. Ferraro, et al., Parallel Finite Elements Applied to the Electromagnetic Scattering Problem, in Proceedings of 5th Distributed Memory Computing Conference, 1990, IEEE Computer Society Press, Los Alamitos, CA. p.417.
8. M. T. Heath, and P. Raghavan, Performance of a Fully Parallel Sparse Solver, in Proceedings of Scalable High Performance Computing Conference 1994, p.334, IEEE Computer Society Press, Los Alamitos, CA.
9. H.Q. Ding and R.D. Ferraro, Slices: A Scalable Partitioner for Finite Element Meshes, will appear in Proceedings of 7th SIAM Conference on Parallel Processing for Scientific Computing, 1995.
10. D.A.H. Jacobs, The exploitation of Sparsity of Iterative Methods, in Sparse Matrices and their Uses, edit. by I.S.Duff, Academic Press, London, (1981)