Parallel Hybrid Iterative/Direct Solution Methods

Robert D. Ferraro

Jet Propulsion Laboratory

California Institute of Technology

1. Introduction

In solving finite element problems, the differential equation is ultimately reduced to a set of linear equations which must be solved by some method. Various different approaches can be used to solve this linear equation set. In some instances, an explicit method is used to advance the differential equation in time and thus the solution of the linear equation set is accomplished automatically in the process. In most cases, however, a matrix of coefficients must actually be formulated and some numerical method employed to obtain onc or more solutions to this equation set.

There arc generally two broad categories of methods for solving linear equations: direct methods and iterative methods. Direct methods typically involve the factorization of a matrix into some combination of upper and lower triangular matrixes which may be easil y used in backsolving with the right hand side to obtain the solution. Once the factorization is accomplished, little additional effort is required to solve problems which differ from the original only in the right hand side (which usually represents the boundary conditions or driving function). Direct methods, depending on the sparsity of the linear system, involve operation counts which scale approximatel y as order nb^2, where n is the number of degrees of freedom, and b is the average bandwidth of the sparse matrix. Although various numbering techniques may be used to minimize the bandwidth of sparse matrixes, the fact that the problem stems from a finite element formulation implies that general geometric considerations will couple the bandwidth to the number of degrees of freedom and the dimensionality of the problem, so that *b is* on the order of square root of *n* for two dimensional problems, and the two-thirds power of *n* in three dimensional problems.

Iterative methods, on the other hand, typically involve sparse matrix-vector multiplies and the construction of orthogonal vectors to decide on new search directions at each iteration step. The sparse matrix-vector multiply can be computed in order *nf* operations, where *n* is the number of degrees of freedom, and *f is the* average number of non-zero coefficients in each equation. For finite element formulations, *f is* a number which depends on the type and degree of finite elements employed, and is independent of the size of the problem. Obtaining a solution to the. linear equations set involves repeated matrix-vector multiply operations until some convergence criterion is satisfied. 'Ibis typically involves something less than order *n* operations, thus the iterative methods have somewhat of an advantage in terms of operation count compared to direct methods when dealing with very large systems.

More importantly, in current parallel supercomputing environments, locality of memory access is the most important consideration in the design of any method for solving linears ystems. In every parallel supercomputer available, the programmer must pay a price for accessing memory which is not local to a processor. The largest scalable parallel computers today (such as the IBM SP-2, Intel Paragon, Cray T3D, SGI POWER CHALLENGEarray, and Convex Exemplar) have processors with local memory interconnected through some fast network hierarchy. Though some distributed memory systems, like the Cray T3D and Convex Exemplar SPP, allow the user a shared memory programming model, efficiency considerations require the user to manage the locality of data access, since referencing data which is not in the processor's local memory is always a significant cost operation. Thus in designing parallel algorithms. special attention must always be given to arranging data in memory to minimize the number of remote memory accesses across the interconnection network, or at least organizes them into block references which can be fetched efficiently. This usually leads to algorithms which operate on localized blocks of data rather than striding uniformly or randomly through memory. Parallel algorithms for direct and iterative methods have been published in many places in the last fcw years, i.e. Fox, et. al. (1988, 1994), and Barrett, et. al. (1993). Parallel computer manufacturers usuall y provide some type of parallel factorization and parallel iterative method in library form which maybe employed in solving finite clement problems. Although direct methods are typically more stable than iterative methods, for large problem sizes, their operation counts and parallel scaling arc such that they arc not feasible for usc in solving very large problems. As tens of millions of finite elements become used in electromagnetics problems, it will become imperative to apply an iterative method to obtain any solution whatsoever.

A wide variety of iterative methods have been employed to solve finite element problems. The most popular among these are the Krylov subspace orthogonalization methods upon which conjugate gradient and its variations are based. Explicit orthogonal ization methods such as GMRES are also very popular and robust, but require substantially higher operation counts to achieve convergence. This is due to the fact that the latter method explicitly constructs orthogonal search directions based on some number of previously saved iterations. This process becomes more expensive with every new iteration, and thus is typically carried only for a limited number of steps before some restart algorithm is used. An excellent discussion of some of the theory and all of the issues associated with iterative methods maybe found in Barrett, et. al. (1993) and references contained therein, and will not be repeated here.

In this chapter, a novel combination of direct and iterative methods will be considered which can be employed successfull y in solving finite element problems. This combination, referred to here as hybrid methods, originated from the observation that a parallel decomposition of a finite clement mesh resulted in some, number of sub-problems which could be viewed as independent finite element problems and which could be solved in parallel without communication. The bandwidth of each of time problems was substantially smaller than the bandwidth of the total linear system and thus direct methods could be efficientl y employed on these sub-problems when it was not feasible to do soon the global problem.

In parallelizing a finite element method, consideration must be given not only to the parallel linear equations solution method, but also to the parallel assembly of the coefficient matrix of the linear system, and the boundary conditions or excitation terms which constitute the right hand side of the linear system. Since finite elements arc geometric objects with geometric connectivity and spatial relationships, the sparsity pattern of the coefficient matrix mimics the spatial pattern and connectivity of the finite elements from which the matrix is constructed. The parallel partitioning of the finite element mesh maybe used directly to construct a decomposed sparse coefficient matrix for use by iterative methods, since the matrix-vector multiplies can be done in parallel without regard to the order in which the operations are carried out.

performed at all times in order to achieve the highest efficiency. Thus besides data decomposition, load balance is a critical consideration in any parallel algorithm. For conjugate gradient algorithms, the majority of work is in the matrix-vector product and since that work is directly proportional to the number of nonzero entries in the matrix, a load balanced implementation corresponds to a uniform distribution of the nonzero matrix elements across processors. Attention should also be paid to the work involved in computing the vector-vector inner products, but this work is usually small compared to the matrix-vector multiply.

The basic conjugate gradient algorithm for linear systems which result in symmetric positive definite matrices is given in Fig, 1. This algorithm is intended to solve the linear system

$$\mathbf{Ax} = \mathbf{b} \tag{1}$$

Here, and in Fig. 1, A is a symmetric positive definite matrix corresponding to the coefficients of the linear system and b is the right-hand side vector. In the conjugate gradient algorithm of Fig. 1,x(i) is the solution at each iteration step, $\mathbf{r}^{(i)}$ is the residual vector at each iteration step, and a and β arc scaling coefficients which are used to determine the ncw conjugate search directions $\mathbf{p}^{(i)}$ and q(i). his algorithm involves one matrix-vector product, two vector inner products, and three vector scale & add operations (S AXPYs) in the inner loop. The theory behind conjugate gradient algorithms can be found in many places, and will not be discussed here. Instead this section focuses on the issues of designing an efficient parallel implementation for conjugate gradient algorithms and their variations.

Parallelism in conjugate gradient algorithms essentially comes fı om parallelism in the matrix-vector multiply, the inner products, and the SAXPYS. The remainder of the operations involved in the conjugate gradient algorithm in Fig. 1 are trivial compared to these. In the main loop of the algorithm the sequence of operations to be performed is fixed by the algorithm itself. However, certain operations can proceed in parallel since there arc no dependencies. In particular, the update of the. residual vector and the solution vector do not depend on each other, and could be pei formed in parallel. Neither of these parallel operations can proceed before the nlatrix-vector product is completed and the scaling coefficient a is computed from its result. Likewise, the matrix-vector product of the next iteration may not proceed until the residual *vector* has been updated. Thus there are two points at which all processors must synchronize before they may proceed further with the algorithm. It is important that the work load remain balanced between synchronization points so that all processors participate in all of the computation that needs to take place and that none go idle during that period. Therefore the parallel implementation of the matrix-vector product should be tailored to the details of the matrix itself so that it achieves its maximum parallel efficiency and the work for the vector inner products and SAXPYS be equally divided among processors. Note that this requirement does not necessarily impose a rigid constraint on how the matrix and vectors are decomposed in parallel. Any decomposition which achieves these goals will allow a parallel conjugate gradient algorithm to perform well.

In exact arithmetic, the conjugate gradient algorithm applied to a set of n linear equations converges to the solution uniformly (in some norm) and in n iterations. The residual error is generally reduced at each iteration so that the solution comes closer and closer to the exact solution with each iteration step. Thus when only finite precision is required of the answer, conjugate gradient may arrive at a sufficiently good solution much earlier than the n iterations required for exact convergence. This means that in practice, conjugate gradient algorithms converge in less than order n^2 operations. When problem sizes arc large, conjugate gradient is very attractive compared to direct factorization methods. (The Reader is reminded that the uniform convergence property of the conjugate gradient algorithm is applicable only to positive definite symmetric linear systems. The variants of conjugate gradient for other types of linear systems do not, in gcncrat, have such convergence properties. However, the reduction of the residual error to some stopping criteria generall y produces a uniform error solution, so that less than order n^2 operations are the norm for conjugate gradient and all of its variations.)

The basic conjugate gradient algorithm, however, is almost never used because large linear systems often have poor condition numbers. Finite precision arithmetic in combination with poor conditioning causes convergence to be slow, or can prevent convergence altogether. Methods have been developed to improve the conditioning of the problem, and speed the rate at which the conjugate grad icnt iterations achieve the required precision. These methods, known as preconditioners, essentially transform the linear equation set into a ncw set of equations which are better suited to solution by conjugate gradient methods. Written formally, a preconditioned $\mathbf{M} = \mathbf{M}_l\mathbf{M}_r$ takes the linear system in eqn. (1) and transforms it by

$$(\mathbf{M}_l^{-1}\mathbf{A}\mathbf{M}_r^{-1})(\mathbf{M}_r\mathbf{x}) = (\mathbf{M}_l^{-1}\mathbf{b}) \qquad (2)$$

into

$$\mathbf{A}'\mathbf{x}' = \mathbf{b}' \qquad (3)$$

Writing M as the product of a left preconditioner Ml and a right preconditioner $\mathbf{M}_r$

allows the properties of eqn. (1) to be preserved in eqn. (3), provided that Ml, M_r and thus M, arc appropriately chosen. For example, if A in eqn. (1) is symmetric and positive definite, and $M_l = M_r^T$, then A' will also be symmetric and positive definite.

An effective preconditioner essentially transforms the matrix into *another* matrix which is closer to the identity than the original. The ideal preconditioned is in fact the matrix inverse itself! However, the cost of computing the inverse is the cost of solving the problem in the first place. A detailed discussion of preconditioning is also beyond the scope of this chapter. Preconditioning is an art rather than a science, and the made.r is referred again to Barrett, et. al. (1993) for a variety of methods in common USC.

In Fig. 2, the conjugate gradient algorithm has been modified to include the application of a symmetric positive definite pre.conditioner, which is represented by solving the linear system M z = r. The preconditioner is almost never applied directly to the matrix, because that would destroy the sparsity characteristics of the matrix, and thus the advantages of sparse vector matrix multiply. Rather, the algorithm is modified to include a linear system solve at each iteration. The preconditioner has presumably been chosen so that the linear system solve can be accomplished quickly! However, the application of a preconditioner to a decomposed linear system on a parallel processor proves to be a serious complication to the efficiency of parallel conjugate gradient. The linear equation solve that represents the application of the preconditioner can be as complicated to implement in parallel as any of the direct factorization methods themselves or it may require a high setup cost to construct in the first place. Unless the preconditioner chosen has data decomposition characteristics which match the matrix decomposition characteristics, the efficiency of the conjugate gradient multiply can be ruined by the inefficiency of applying the preconditioner. A

1

inefficient preconditioner, or onc which is expensive to construct in a parallel environment can more than cancel the benefit of a reduced iteration count that it was meant to provide. So particular care must be taken in choosing a preconditioner in a parallel implementation.

The basic conjugate gradient algorithm discussed above is useful only for positive-definite linear systems, but there area variety of extensions and variations of this algorithm which can be applied to linear systems with other problems. Bi-conjugate gradient, for example, can be used for complex systems, quasi-minimum residual (QMR) method, for complex symmetric systems, and conjugate gradient squared, and bi-conjugate gradient stabilized for complex indefinite systems. Each of these variations involves a different method of computing the search directions, but the basic operations remain the same: matrix-vector products, vector inner products, SAXPYS, and some preconditioning scheme which is applied at each iteration step. Thus parallel efficiency for any of these methods will be achieved by efficient parallel implementation of these core operations. Ignoring the preconditioner, the parallel scaling for the conjugate gradient algorithm is the same as the scaling for parallel matrix multiply. Very high efficiency can be achieved by careful attention to minimizing the communications involved in these operations.

The introduction of a preconditioner can destroy this scaling property, Thus even though the number of iterations to achieving the solution is substantially reduced by the usc of a preconditioner, the total execution time may actually rises if the parallel efficiency for the application of the preconditioner itself is poor. In some situations, it is in fact better to usc a very simple preconditioner like diagonal scaling (Jacobi preconditioning) which has excellent parallelization properties than it is to usc a ‘sophisticate preconditioner like incomplete Cholesky factorization, Even though the iteration count is higher with diagonal scaling, the execution time to solution may be as good as or better than the more sophisticated method,

3. Hybrid direct-conjugate gradient algorithms

The implementation of finite element methods for parallel computers provide an opportunity to employ a unique method of combining direct and iterative linear equation solvers which, when taken together, enjoy better parallel scaling and convergence than either alone. The typical implementation of a finite clement method uses a domain decomposition which splits the finite element mesh into compact submeshes so that each processor has (approximately) an equal portion of the entire problem. From the individual processor's viewpoint, it has a complete finite element problem. Thus one can employ a direct method to solve the finite element subproblem without regard to the meshes which arc being processed on other processors. Consider the example two-dimensional finite element mesh shown in Fig. 3. In this example, the mesh has been partitioned among four processors, such that elements in the mesh reside in onc and only onc processor. The boundaries between partitions lie along element edges or faces, so that nodal points (and degrees of freedom corresponding to those points) serve as a dividing line between processors. Taken individually, each processor has a partition which is a finite element problem unto itself with boundary conditions that arc dcpcndcnt on the results obtained in other processors. In Fig. 3 the *nodal points* interior to a partition arc represented by filled *circles* and nodal points lying on partition boundaries arc denoted by open circles. Were it not for the fact that the values on the partition boundaries are coupled to results in neighboring processors, a standard sequential finite elements method could be used without modification to solve the interior problem.

The elements in this simple example belong uniquely to a single partition. In a standard sequential finite element formulation, each element contributes additively to the global *stiffness* matrix K and *force* vector **f** [see Hughes (1987), e.g.] as follows:

$$\mathbf{K} = \sum_{e} K^{(e)} \qquad (4a)$$

$$\mathbf{f} = \sum_{e} f^{(e)} \qquad (4b)$$

Here $K^{(e)}$ and $f^{(e)}$ are the contributions of a single element to the global stiffness matrix and global force vector respectively, and the. summation is over all elements in the mesh. In a partitioned mesh such as shown in Fig. 3, the contributions of all the elements may be computed in an *embarrassingly parallel* fashion (i.e., without interprocessor communication), since each partition is assigned to its own processor. The stiffness matrix $K^{(p)}$ which corresponds to a partition contribution is computed on its processor using eqn. (4a) for the elements contained in the partition, and the global stiffness matrix may be recovered (if necessary) by summing over processors *p*:

$$\mathbf{K} = \sum_{p} \mathbf{K}^{(p)} \qquad (5)$$

The force vector **f** is computed similarly using eqn. (4b) on each processor, but the. results must be summed across all processors p so that the values at the shared nodes correctly include contributions from elements in different partitions,

For iterative methods which require only matrix-vector products and inner products, it is not necessary to recover the global stiffness matrix. These methods only require the result of the matrix-vector product, which can be computed in parallel directly from the individual partition stiffness matrices K‘)

$$\mathbf{Kx} = \sum_{P} (\mathbf{K}^{(p)} \mathbf{x}^{(p)}) \qquad (6)$$

Here $X^{(p)}$ is the portion of a global vector which corresponds to the unknowns in partition *p*. The vector $X^{(p)}$ will contain entries for shared nodes on partition boundaries which arc duplicated in more than one processor. In partition A in Fig. 4, for example, $X^{(A)}$ would consist of the unknowns associated with the interior nodes of the partition and the shared nodes labeled *a*, *b*, and *e*, but no others. The matrix-vector multiply of eqn. (6) will automatically produce the correct results in parallel for interior nodes. The summation of eqn. (6) must be performed only for the shared nodes, i.e., the shared nodes have contributions from elements in multiple partitions which must be summed together to obtain the correct result.

This observation leads to the possibility of using standard sequential direct factorization methods to remove the degrees of freedom interior to the partitions entirely from the problem, leaving only the shared nodal points on partition boundaries to be solved in parallel. This idea can be represented formally by writing out the linear equation set which corresponds to the finite element problem in the following manner, First, the degrees of freedom attached to partition interior nodal points arc numbered in order by partition, followed by the degrees of freedom which arc attached to the partition boundary nodes. For edge clement problems, degrees of freedom arc associated with edges instead of nodes, but the same numbering methodology applies. The global matrix which corresponds to the partitioned finite element problem can now be seen to consist of a set of matrix blocks which correspond to the coupling of interior points of each partition, the coupling terms between the interior points and the boundary points in each partition, and finally the entries corresponding to coupling among the boundary points themselves as represented pictorially in Fig. 4 for the partitioned mesh shown in Fig. 3. Returning to the usual linear algebra notation of eqn. (l), this linear system can be written as

$$\begin{bmatrix} A_{ii} & A_{is} \\ A_{si} & A_{ss} \end{bmatrix} \begin{bmatrix} x_i \\ x_s \end{bmatrix} = \begin{bmatrix} b_i \\ b_s \end{bmatrix} \tag{7}$$

where A_{ii} arc the blocks which result from pairs of interior nodes, A_{is} arc the blocks which result from interior node/shared node pairs, and A_{ss} is the block which results from pairs of shared nodes. Note that the large blocks of partition interior nodes in this matrix arc coupled to each other only through shared nodes and thus maybe formal] y removed from the problem by simple algebra, The upper equation in eqn. (7) maybe solved for xi to obtain

$$x_i = A_{ii}^{-1} (b_i - A_{is} x_s) \tag{8}$$

Here it is understood that A_{ii}^{-1} is a shorthand notation for a factorization of A_{ii}. Introducing this expression for xi into the lower equation of eqn. (7) results in a reduced linear equation set consisting only of shared partition boundary points:

$$(A_{ss} - A_{si} A_{ii}^{-1} A_{is}) x_s = (b_s - A_{si} A_{ii}^{-1} b_i) \tag{9}$$

or

$$A_r x_s = b_r \tag{lo}$$

where

$$A_r = (A_{ss} - A_{si} A_{ii}^{-1} A_{is}) \tag{ha}$$

$$\mathbf{b}_r = (\mathbf{b}_s - \mathbf{A}_{si}\mathbf{A}_{ii}^{-1}\mathbf{b}_i) \tag{11b}$$

The process of eliminating the interior points couples every shared point on every partitioned boundary with every other shared point.

It is a consequence of eqn. (5), which is a property of the finite element method, that eqn. (1 la) partitions completely and independently among processors. Each of the matrix blocks in eqn. (1 la) turn out to be themselves partitioned among processors so that

$$\mathbf{A}_r^{(p)} = (\mathbf{A}_{ss}^{(p)} - \mathbf{A}_{si}^{(p)}(\mathbf{A}_{ii}^{(p)})^{-1}\mathbf{A}_{is}^{(p)}) \tag{12}$$

and

$$\mathbf{A}_r = \sum_P \mathbf{A}_r^{(p)} \tag{13}$$

Each of these reduced matrices $\mathbf{A}_r^{(p)}$ arc dense matrices. It should bc noted that eqn. (8) may also bc computed independently in each partition for that partition's interior nodes, and thus may be done in parallel without communication, The reduced matrices could be recombined across all processors, and redistributed for a parallel factorization step, but the. decomposition method allows these matrices to be used directly by an iterative schcmc, e.g., a conjugate gradient method, to complete the solution of eqn. (10) for the shared nodes. Note that the solution obtained for the reduced equations is exactly the desired values for the degrees of freedom on the shared nodes. To obtain the solution values on the interior nodes is a simple matter of applying eqn. (8) to the solution for the shared nodes. The factorization allow for forward and back substitution in parallel on each processor without communication. Additionally, for simple changes in global boundary conditions, the factorizations of the interior node matrix blocks may be retained so that multiple right-hand sides may bc done successively in an efficient manner.

This method produces reduced matrices whose characteristics are the same as those of the global matrix from which it was derived, i.e., if the global matrix is symmetric, the reduced matrix is also symmetric; if the global matrix is Hermitian, then the reduced matrix is Hermitian. Thus, any of the standard iterative **methods** can be used to solve the reduced equation set since the matrix vector multiplies arc the fundamental operations and each conjugate gradient method differs only in how the results arc combined to form ncw search directions. Addi tionally, for poorly conditioned problems, the application of a direct factorization method with pivoting can improve the conditioning of the system and allow the conjugate gradient algorithm to converge on the reduced equations when it would not have converged on the global system.

In constructing the software to implement this method, the essential considerations arc the numbering of the unknowns locally within each processor, so that the matrix structure naturally falls into the form shown in Fig. 4. The algorithm for this hybrid method is presented in Fig. 5. The segregation into the interior and shared node blocks allows sequential algorithms for the computation of $\mathbf{A}_{ii}^{-1}$ (factorization) to be applied unchanged, Many are available, see for example, Press, ct. al. The application of cqn. (12) maybe done using standard library routines like the BLAS of Lawson, ct. al. (1 979) or, if special storage methods are used, can bc written simply as matrix vector multiplies. The final reduced matrix, which is essentially dense, may be used with a BLAS SEGMV routine which is typically available on all parallel platforms, and optimized for its architecture. Thus the only communications required is a global summation of vectors at the end of each call to SEGMV.

Since the iterative part of this method is applied only to the unknowns which reside on partition boundaries, this method has scaling properties which arc better than an iterative method applied to the original problem. The operation count for conjugate gradient on the global finite element problem scales as order n^2 where n is the number of degrees of freedom. In this method, the number of unknowns on the partition boundary goes as $n^{1/2}$ for two-D problems or $n^{2/3}$ for 3-I> problems, so that the iterative portion of the solver converges in either order n operations in 2-D or order $n^{4/3}$ in 3-D. The remainder of the operations involved in the method scale as the number of interior points. As problem size grows with the number of processors, this work load is a constant independent of problem size. Thus if wc compare the implementation of a conjugate gradient method with diagonal preconditioning to this hybrid method, we see that the hybrid method outperforms the conjugate gradient method after only a small number of processors.

In Fig. 6 the scaling behavior of each of these methods is plotted versus number of processors for a series of scaled problems with 1600 grid points per processor. That is to say, a base problem consisting of 1600 unknowns was solved on a single processor, and scaled problems consisting of $1600p$ unknowns were solved for p = 2, 4,8, 16, .,.The data shown is the ratio of execution time for solution of each of these problems normalized to the execution time for the 1600 grid problem on a single processor. The bi-conjugate gradient method scales as p where p is the number of processors while the hybrid method applied to the same problems scales approximatel y as the square root of p. It is clear from the graph that for large problem sizes on large numbers of processors, the hybrid method is the clear winner.

Although the results presented here are for a Cholesk y factorization of the interior degrees of freedom followed by a conjugate gradient solution of the shared degrees of freedom, it is expected that this scaling behavior will carry over to any appropriate combination of direct and iterative method. The direct method is a constant cost, while the iterative method operates on a problem size which is reduced fı om the original by the surface to volume ratio (since the shared nodes arc the surfaces of a partitioned volume). Although this method is most easily understood in terms of and applied to parallel finite element problems, it relics only on the assumption that the linear equations set being solved represents some local coupling among unknowns (like that typically arising from the solution of partial differential equations via finite difference, finite element, or finite volume methods) and that a geometric compact partitioning of the unknowns is possible.

4. Acknowledgments

Many people have contributed to the development of this idea (over several years and projects) into a working software package. In particular, the author wishes to acknowledge G.A. Lyzenga as the originator of the idea a number of years ago at the Jet Propulsion Laboratory, and thanks H.Ding for his contributions in extending the software to finite difference problems and in measuring the methods sealing properties.

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology under a contract with the National Aeronautics and Space Administration. Rcfcrcncc herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

This research was performed in part using the Intel Touchstone Delta and Paragon Systems operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by the Jet Propulsion Laboratory, and NASA Office of Aeronautics.

5. Bibliography

Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and van der Vorst, H. (1993). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Philidelphia:SIAM. iv+92 pp.

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.

H.Q. Ding and R.D. Ferraro, Slices: A Scalable Partitioner for Finite Element Meshes, in Proceedings of 7th SIAM Conference on Parallel Processing for Scientific Computing, 1995.

1.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices,* Oxford University Press, London, 1986.

R.D.Ferraro, T.Cwik, N.Jacobi, P.C.Liewer, T.G.Lockhart, G.A.Lyzenga, J.Parker, and J.E.Patterson, *Parallel Finite Elements Applied to the Electromagnetic Scattering Problem* in Proceedings of the 5th Distributed Memory Computing Conference, Edited by D.W.Walker and Q.F.Stout, IEEE Computer Society Press, Los Alamitos, CA. (1990) p.417.

G.C. Fox, M.A. Johnson, G.A. Lyzcnga, S.W. Otto, J.K. Salmon and D.W. Walker, Solving *Problems on Concurrent Processors,* Vol. 1, Prentice Han, Englcwood Cliffs, Ncw Jersey, 1988. Chap.7.

Fox, G.C, Williams, R, D., and Mcssina, P.C. (1994). Parallel Computing Works! Morgan Kaufmann Publishers, Inc., San Francisco

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 Socicty Press, Los Alamitos, CA.

Hughes, T, J. R. (1987). *The Finite Element Method.* Prentice-liall, Inc., Englcwood Cliffs, Ncw Jersey.

Jacobs, D.A.H. (181) 'The exploitation of Sparsity of Iterative Methods'. In: *Sparse Matrices and their Uses,* (I.S.Duff, editor). London: Academic Press

V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing*, Benjamin/Cummirlgs, Redwood City, CA, 1994. Chap.11.

Lawson, C., Hanson, R., Kincaid, D., andKroch,F.(1979). *Basic Linear Algebra Subprograms for FORTRAN usage,* ACM Trans. Math. Soft., 5, pp. 308-325.

G.A. Lyzcnga, A. Racfsky and B. Nour-Omid, *Implement Finite Element Software on the Hypercube*, Proceedings of 3rd Hypercube Conference, ACM Press, Ncw York (1988), p.1755.

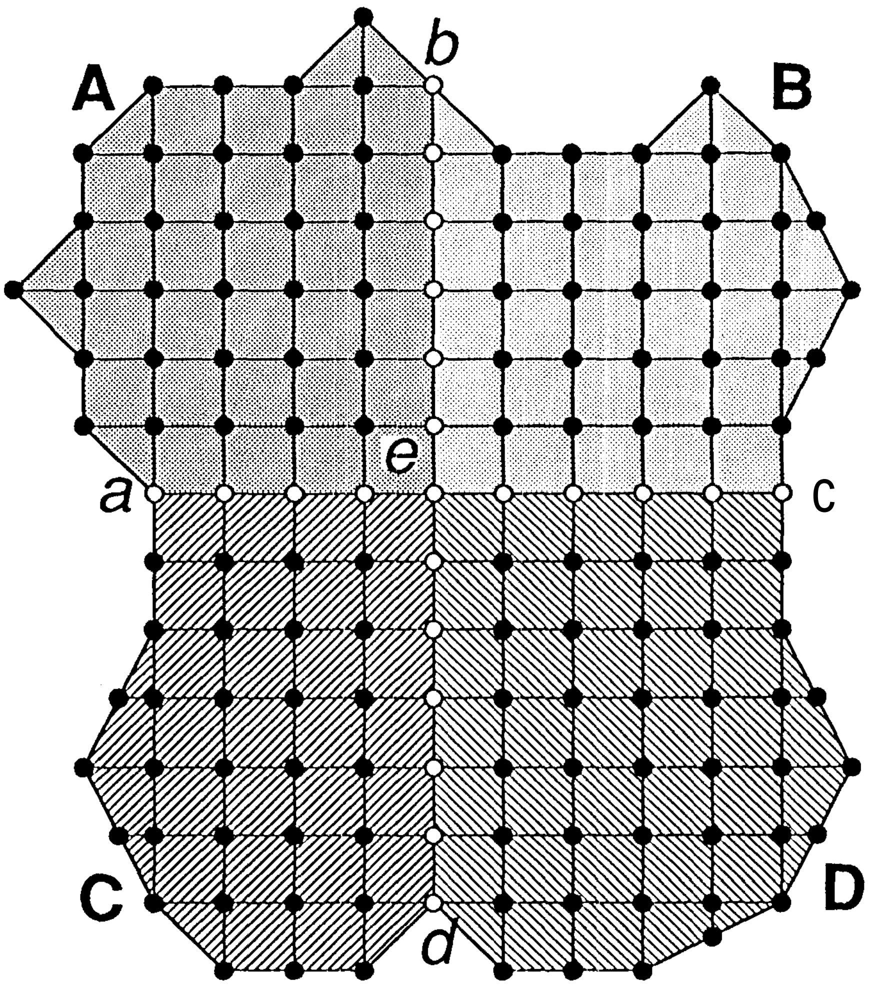
Press, W. H., Flannery, B.P., Teukolsky, S. A., and Vctterling, W.T. (1986), *Numerical Recipes. Ncw* York: Cambridge Universit y Press. pp. 31-37

```
$p^{(0)} = r^{(0)} = b - Ax^{(0)}$
$\rho_0 = r^{(0)} \cdot r^{(0)}$
while $\| r^{(i)} \|$ not converged
    $q^{(i)} = Ap^{(i)}$
    $\alpha_i = \frac{\rho_{i-1}}{p^{(i)} \cdot q^{(i)}}$
    $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
    $r^{(i)} = r^{(i-1)} - \alpha_i p^{(i)}$
    $\rho_i := r^{(i)} \cdot r^{(i)}$
    $\beta_i := \frac{\rho_i}{\rho_{i-1}}$
    $p^{(i)} = r^{(i)} + \beta_i p^{(i-1)}$
    end while
```

Figure 1. The conjugate gradient algorithm for symmetric, positive definite linear systems,

```
$\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
*solve* $\mathbf{M}\mathbf{z}^{(0)} = \mathbf{r}^{(0)}$
$\rho_0 = \mathbf{r}^{(0)} \cdot \mathbf{z}^{(0)}$
$\mathbf{p}^{(0)} = \mathbf{z}^{(0)}$
*while* $\|\mathbf{r}^{(i)}\|$ *not converged*
    $\mathbf{q}^{(i)} = \mathbf{A}\mathbf{p}^{(i)}$
    $\alpha_i = \dfrac{\rho_{i-1}}{\mathbf{p}^{(i)} \cdot \mathbf{q}^{(i)}}$
    $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha_i \mathbf{p}^{(i)}$
    $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha_i \mathbf{p}^{(i)}$
    *solve* $\mathbf{M}\mathbf{z}^{(i)} = \mathbf{r}^{(i)}$
    $\rho_i = \mathbf{r}^{(i)} \cdot \mathbf{z}^{(i)}$
    $\beta_i = \dfrac{\rho_i}{\rho_{i-1}}$
    $\mathbf{p}^{(i)} = \mathbf{z}^{(i)} + \beta_i \mathbf{p}^{(i-1)}$
*end while*
```

Figure 2. The preconditioned conjugate gradient algorithm for symmetric, **positive** definite linear systems.



● Interior Node
O Shared Node

Figure 3. A simple finite element mesh partitioned among four processors Elements are uniquely assigned to partitions A, B, C, and D. Nodes on partition boundaries **a, b, c, d,** and the center node *e are* assigned to multiple partitions.

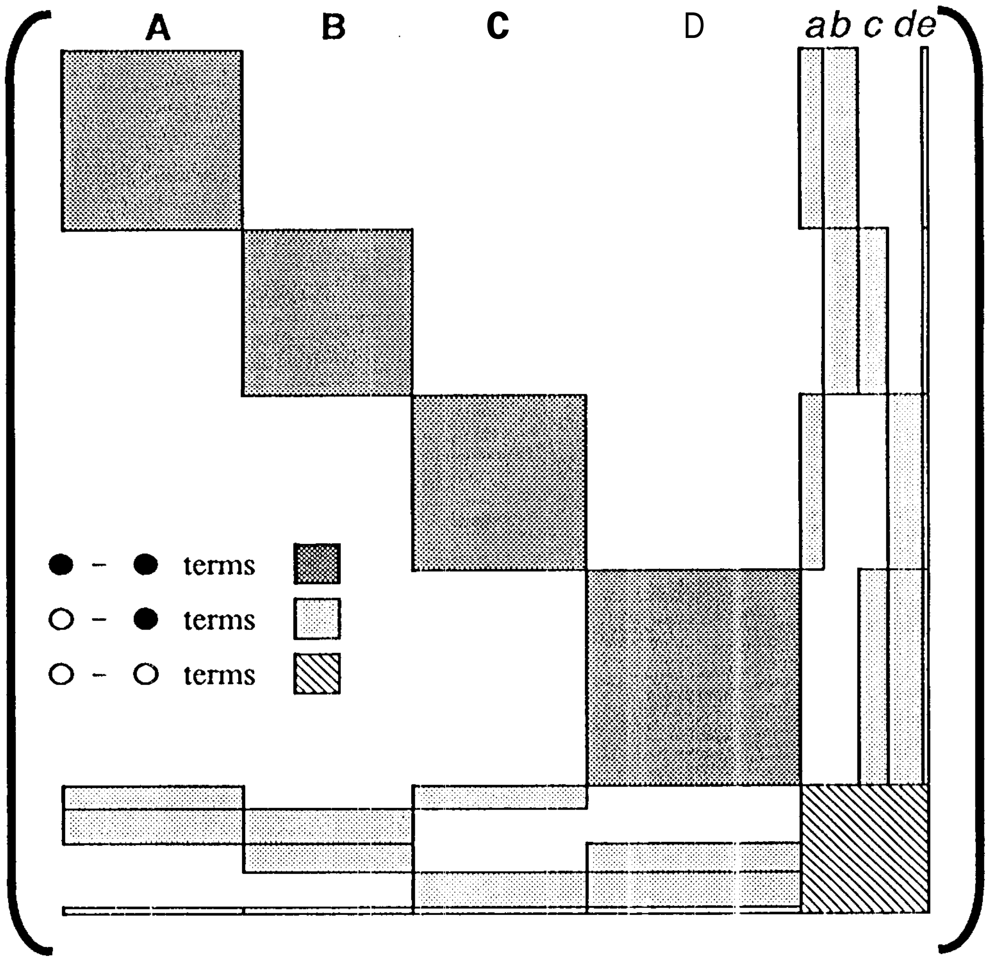


Figure 4. The structure of matrix of coefficients of the linear system which results from the finite element problem of Fig, 3. Unshaded areas arc zero values.

```
For each partition in parallel:
    Factor $\mathbf{A}_{ii}^{(p)}$
    Compute    $\mathbf{A}_r^{(p)} = \mathbf{A}_{ss}^{(p)} - \mathbf{A}_{si}^{(p)} (\mathbf{A}_{ii}^{(p)})^{-1} \mathbf{A}_{is}^{(p)})$
    Compute $\mathbf{b}_r^{(P)} = \mathbf{A}_{si}^{(p)} (\mathbf{A}_{ii}^{(p)})^{-1} \mathbf{b}_i^{(p)}$
Compute and distribute $\mathbf{b}_r = \mathbf{b}_s - \sum_P \mathbf{b}_r^{(p)}$
Solve $(\sum_p \mathbf{A}_r^{(p)}) \mathbf{x}_s = \mathbf{b}_r$ using a parallel iterative solver
For each partition in parallel:
    Compute $\mathbf{x}_i^{(p)} = (\mathbf{A}_{ii}^{(p)})^{-1} (\mathbf{b}_i^{(p)} - \mathbf{A}_{is}^{(p)} \mathbf{x}_s)$
```

Figure 5. The parallel hybrid algorithm. The factor and iterative solve steps must be tailored to match the properties of the underlying linear system. For symmetric systems, some additional algebraic recombiniations are possible.

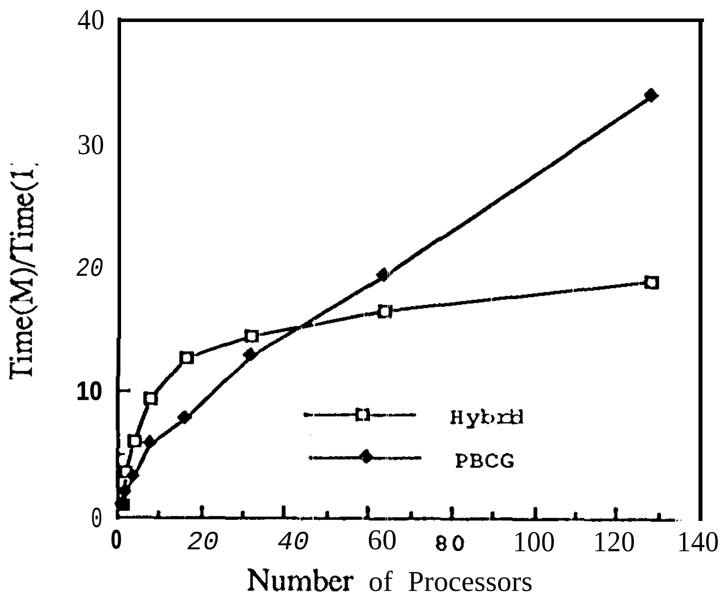


Figure 6. Scaling characteristics of the hybrid method compared to a diagonally preconditioned conjugate gradient method for the same problems.