Finite Elements and the Method of Conjugate Gradients on a Concurrent Processor

Gregory A. Lyzenga ✓
Arthur Raefsky
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

Bradford H. Hager
Seismological Laboratory
California Institute of Technology
Pasadena, CA 91125

ABSTRACT

We present an algorithm for the iterative solution of finite element problems on a concurrent processor. The method of conjugate gradients is used to solve the system of matrix equations, which is distributed among the processors of a MIMD computer according to an element-based spatial decomposition. This algorithm is implemented in a two-dimensional elastostatics program on the Caltech Hypercube concurrent processor. The results of tests on up to 32 processors show nearly linear concurrent speedup, with efficiencies over 90% for sufficiently large problems.

INTRODUCTION

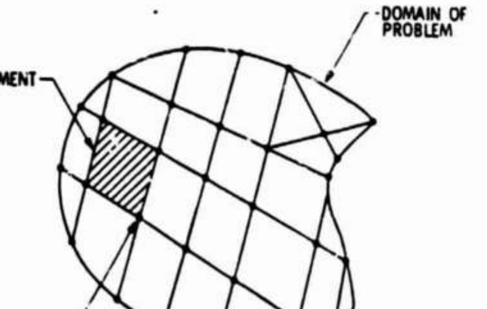
Finite Elements and Elliptic Boundary Value Problems

The need to solve boundary value problems involving elliptic partial differential equations on geometrically complex domains arises in many engineering contexts, and increasingly in a wide variety of scientific fields. The finite element method, which has been treated in a large number of texts (e.g., Zienkewicz,1), provides a flexible and powerful numerical technique for the solution of such problems. The physical problems which have been solved by finite element methods come from such diverse fields as structural and continuum mechanics, fluid dynamics, hydrology, heat flow analysis and many others.

Among the common features of most finite element applications is the need to form and to solve a matrix equation of the form:

$$\mathbf{A}\,\mathbf{x} = \mathbf{b} \qquad (1)$$

Here, $\mathbf{x}$ is a vector of unknown quantities, to be solved for at each of some number of "node" locations within the problem domain. The grid of nodes (usually representing a discretization of some physical spatial domain) also defines a subdivision of the domain into volumes (or areas) called "elements" which jointly comprise the entire problem domain and share nodes along their boundaries (Fig. 1).



$\bar{x}$ - VECTOR OF NODAL DEGREES OF FREEDOM

$\bar{b}$ - VECTOR OF EFFECTIVE "FORCES"

A - ASSEMBLY OF ELEMENT "STIFFNESS"

$$\boxed{A\bar{x} = \bar{b}}$$

Fig.1 Finite element discretization of a boundary value problem

The stiffness matrix $\mathbf{A}$ contains the terms which determine the interaction among the different unknown degrees of freedom in $\mathbf{x}$, and the forcing terms or boundary values are introduced through the right hand side, $\mathbf{b}$. Operationally, entries of $\mathbf{A}$ are computed by performing integrals over the elements which include the node in question. The stiffness matrix therefore consists of an assembly of individual element matrices, which are mapped into the "global" matrix through a master equation bookkeeping scheme.

Common to most such finite element applications is the problem of solving the system given in equation (1). Since the matrix $\mathbf{A}$ may be very large (especially in higher-dimensional problems), considerable thought must go into solving it efficiently. This computation-intensive step is the primary motivation for looking to concurrent computation techniques.

A stiffness matrix $\mathbf{A}$ typically has a number of properties which influence the choice of solution technique. For example, many physical problems give rise to a symmetric positive definite matrix. For the remainder of the present discussion we shall restrict ourselves to this class of finite element problems. This does not mean that more general cases cannot be treated using concurrent techniques similar to those described below, but such problems are not within the scope of this work.

Traditionally, derivatives of Gaussian elimination have been used to solve these systems, giving robust performance under a wide range of matrix ill-conditioning circumstances. Matrix ill-conditioning may occur if a given system is very nearly singular, or exhibits a very large ratio between the largest and smallest eigenvalues, in which case certain matrix operations which are sensitive to round-off error may be inaccurate. While direct techniques such as Gaussian elimination are not particularly sensitive to moderate ill-conditioning, they entail a large amount of computational work. Whether dealing with iterative or direct techniques, the matrix problem nearly always represents the dominant computational cost of a finite element calculation. In the case of Gaussian elimination, this cost increases rapidly with problem size, being proportional to $N_{eq}b^2$, where N_{eq} is the total number of degrees of freedom, and b is the mean diagonal bandwidth of non-zero elements in $\mathbf{A}$.

In a typical assembled finite element matrix, the entries in a given column or row are non-zero only within a given distance from the diagonal. This is because a given node only contributes interaction terms from those nodes with which it shares an element. Although judicious node numbering can minimize the average column height and thus the mean diagonal bandwidth of the matrix, problems with large numbers of nodes in more than one spatial direction will unavoidably have very large bandwidths. This means that, especially for grids in higher dimensions, the work necessary to solve the system will increase much faster than the number of nodes as we attempt to solve larger and larger systems.

Certain classes of structural analysis problems may have considerably smaller mean bandwidths than the most general three-dimensional problem, and for these, direct techniques remain a viable approach. Research into concurrent methods for these techniques is already well advanced(2,3,4). The present authors have been most interested in continuum mechanics problems, for which direct methods are more limited in their utility.

For the above reasons, we have pursued the implementation of an increasingly popular iterative technique, the method of conjugate gradients. Assuming that it can be made to converge, the conjugate gradient method offers the potential for much improved performance on large three-dimensional grids. This, combined with a rather straightforward concurrent generalization made this an attractive area to explore. In addition, the symmetric positive-definite nature of the matrices considered makes the method of conjugate gradients a good first choice (see for example Jennings,5). In the discussion which follows, we will present the basic concurrent algorithm developed for this application.

Figure 2 shows schematically how a two-dimensional finite element grid might be spread over the processors of an 8-node MIMD machine. Each subset of elements (delineated by the heavy lines) is treated effectively as a separate smaller finite element problem within each processor. Adjoining subdomains need only exchange boundary information with neighboring processors in order to complete calculations for each region concurrently. The next section discusses in more detail how this concurrency is managed.

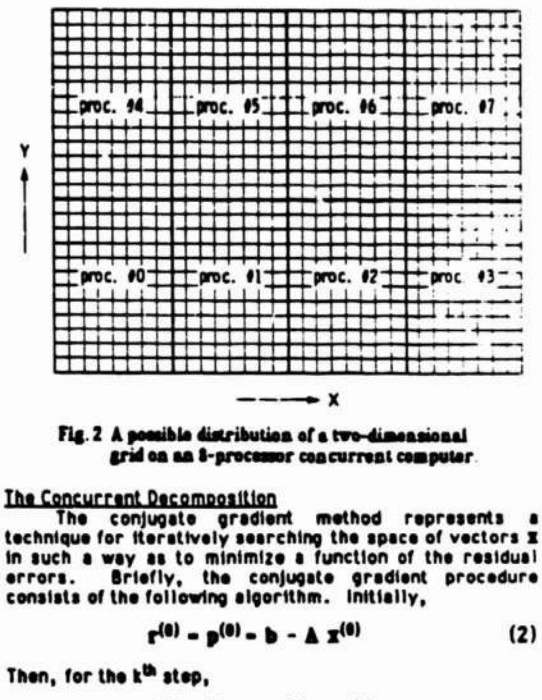


Fig. 2 A possible distribution of a two-dimensional grid on an 8-processor concurrent computer.

<u>The Concurrent Decomposition</u>

The conjugate gradient method represents a technique for iteratively searching the space of vectors $\mathbf{x}$ in such a way as to minimize a function of the residual errors. Briefly, the conjugate gradient procedure consists of the following algorithm. Initially,

$$\mathbf{r}^{(0)} = \mathbf{p}^{(0)} = \mathbf{b} - \mathbf{A}\,\mathbf{x}^{(0)} \qquad (2)$$

Then, for the kth step,

$$\begin{aligned}
&1.\ \alpha_k = (\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)}) / (\mathbf{p}^{(k)} \cdot \mathbf{A}\mathbf{p}^{(k)}) \\
&2.\ \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)} \\
&3.\ \mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k \mathbf{A}\mathbf{p}^{(k)} \qquad (3) \\
&4.\ \beta_k = (\mathbf{r}^{(k+1)} \cdot \mathbf{r}^{(k+1)}) / (\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)}) \\
&5.\ \mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} + \beta_k \mathbf{p}^{(k)} \\
&6.\ k = k+1;\ \textit{go to 1. (continue until converged)}
\end{aligned}$$

As may be seen by inspecting (3) above, this algorithm involves two basic kinds of operations. The first of these is the vector dot product, for example $\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)}$. The second basic operation is the matrix-vector product, $\mathbf{A}\mathbf{p}^{(k)}$. Both of these primitive operations can be done in parallel by decomposing the problem into regions of the physical domain space. A given "global" vector such as $\mathbf{r}$ or $\mathbf{p}$ is spread out among the processors of a concurrent ensemble, with concurrent operations being performed on the various "pieces" of the vectors in each processor. The only need for information from outside a given processor occurs when nodes on the boundary between the "jurisdictions" of two processors are computed.

Figure 3 illustrates schematically the protocol used for handling shared degrees of freedom between processors in the two-dimensional case. Each processor obeys a convention whereby it accumulates contributions to global vector quantities (**x**, **r**, **p**) from neighboring processors along the "right" and "down" edges of the region. Contributions arising from degrees of freedom along a processor's "left" and "up" edges are sent to the respective neighboring processor for accumulation. The "lower right" and "upper left" corner degrees of freedom are passed twice in reaching their destination.

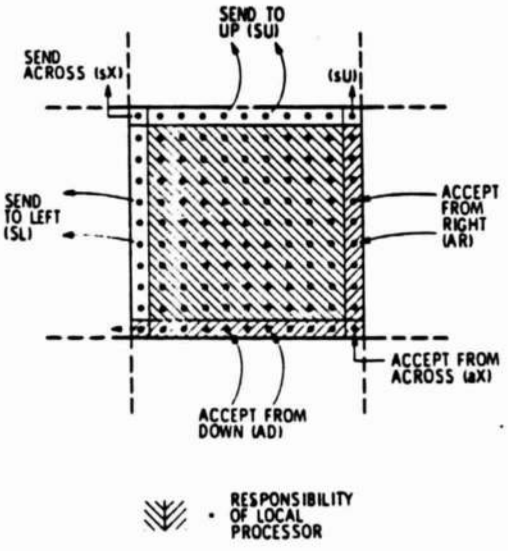


Fig. 3 Two-dimensional protocol for passing shared nodal values

While only one processor has responsibility for accumulating a given shared degree of freedom, each neighboring processor must be provided with a copy of the accumulated result for subsequent calculations. Thus, the process of updating a quantity such as $\mathbf{A}\mathbf{p}^{(k)}$, which involves adding contributions from different elements in different processors, proceeds as follows:

1. The "local" contributions are calculated within each processor independently. This means calculating the contributions to the given matrix or vector product while neglecting the effect of any neighboring processor.

2. A right-to-left pass of edge contributions is done. Vector contributions along the left edge of a processor (i.e., those labeled SL, sL, and sX) are sent to the neighboring processor on the left, at the same time that the corresponding right edge degrees of freedom (AR, aX, and sU) are received from the right. The received contributions are added to the locally calculated contributions already resident in those storage "slots".

3. This is followed by a down-to-up pass. The contributions labeled SU and sU are sent up, while the AD and aX slots receive contributions to accumulate. Note that the corner degrees of freedom labeled sX are transmitted and accumulated twice before reaching their final destination in an aX slot located diagonally from their starting point.

4. Finally, an up-to-down shift, followed by a left-to-right shift which overwrites rather than adds to the current contents of the boundary arrays serves to distribute the the final sum of all contributions to all the involved processors. This ends the communication cycle, and the processors then return to concurrent internal computations.

Aside from this kind of calculation, in which a global vector is updated on the basis of local (element) information, there is one other situation in which processor "responsibility" for global degrees of freedom is significant. Global scalar products are calculated by forming partial dot products within each processor, which are in turn forwarded to a single "control process" (which in this case resides in a separate external processor) which performs the summation and takes action on the result (e.g., terminating the iterative loop). In this case, in order to avoid "double-counting" any entry, each processor calculates its partial dot product only using its internal and accumulated degrees of freedom

The scheme described above is applicable to the hypercube machine architecture specifically, and more generally to any MIMD computer which supports the following communications operations: (1) global broadcast of data from a designated controlling process or processor to the concurrent array, (2) transmission of unique data "messages" between array elements and the control process, and (3) element-to-element data transmission between lattice nearest neighbors.

IMPLEMENTATION

Programming Considerations

The example discussed here was written in the C language, and cross-compiled on a VAX-11/750 system for execution on the 8086 microprocessor-based 32 node Caltech Hypercube (Mark II) machine. Listings and further information on this code may be obtained by contacting one of the authors.

There are no basic algorithmic differences between the concurrent conjugate gradient algorithm discussed here and its equivalent counterpart on a sequential machine. Thus the concurrent program should, assuming infinite precision of calculation, yield results identical to the sequential version.

In actual practice however, the exact result obtained by the conjugate gradient method, and the number of iterations required for it to converge, are rather sensitive to the finite precision of the numerical calculations. The accumulation steps which occur during interprocessor communications represent additional arithmetic operations of finite precision, which introduce some additional round-off error not present in the sequential equivalent. This causes the concurrent code to produce slightly different (but numerically equivalent) results.

The computation/communication structure of the concurrent algorithm is quite regular, in that each processor is simultaneously doing the same type of task as every other, and the only load imbalance or processor waiting is caused by giving processors responsibility for different numbers of elements or degrees of freedom. The problem of how to best decompose an arbitrarily shaped finite element grid presents a problem in the preparation of the input data, but this preprocessing step has nothing to do directly with the actual operation of the concurrent algorithm(6). Highly irregular and/or non-rectilinear grids are handled transparently, just as long as all shared boundary nodes are properly identified (as per Fig. 3) in the input data stream.

Among the special considerations to note for this application are restrictions on available memory. In the present example, we have chosen to calculate and store each element stiffness entry. In such a case, it becomes possible to exhaust the relatively modest memory available in the present generation of machines. This problem is particularly acute in moving from two- to three-dimensional problems. One approach to overcome this limitation is not to store, but to recalculate stiffnesses at each iteration. In such a case, the extra arithmetic operations may be minimized by utilizing one-point quadrature with mesh stabilization techniques (7).

Results of Initial Tests

The above described finite element/conjugate gradient program was tested on a series of two-dimensional plane strain elastostatics problems, intended to a) verify the proper execution of the program and b) provide benchmark measurements of the program's concurrent efficiency as a function of problem size and number of processors.

The Mark II hypercube was used in these tests in configurations from one node (0-dimensional cube) through the full 32 nodes (5-dimensional hypercube). The actual problems solved were rectangular arrays of elements with boundary displacements imposed to produce a simple shear field solution. Concurrent efficiency is a measure of how nearly a concurrent processor with *m* processors approaches speeding a given calculation by a factor of *m*. Thus we obtain experimental efficiencies by dividing the time required to solve a given problem in a single processor by the concurrent time and the number of processors. As long as the sequential and concurrent algorithms are truly equivalent, this efficiency ϵ has an upper bound of unity.

Table 1 - Two-dimensional conjugate gradient run times and efficiencies

number of elements	number of processors	times per iteration (sec.)	efficiency
64	1	0.68	-
144	1	1.64	-
256	1	3.02	-
400	1	4.83	-
576	1	7.07	-
576	2	3.60	0.98
576	4	1.82	0.97
576	8	0.99	0.89
576	16	0.52	0.85
576	32	0.29	0.76
1152	32	0.53	0.85
4608	32	2.01	0.91
18432	32	7.65	0.96

In Table 1, we summarize the results of these test runs on a variety of problem sizes and numbers of processors. Since the maximum number of elements we could accomodate within a single processor was on the order of 600, the efficiencies listed for the larger 32-node runs are based upon extrapolations of the smaller problems' execution times. In this extrapolation, we have assumed a strictly linear relation between number of equations and execution time per iteration. The single-processor proportionality constant is assumed here to be 6.41 millisecond / iteration per degree of freedom. The quoted experimental efficiencies are valid to the degree that this scaling assumption holds.

In a problem which is perfectly load balanced, the concurrent efficiency will be governed by the ratio of time spent communicating to time spent calculating concurrently. Since in a two-dimensional array, the number of communicated edge values varies as $n^{1/2}$, where n is the total number of equations per processor, this ratio should vary as $n^{-1/2}$ and the calculation should obey an efficiency relation,

$$\epsilon = 1 - Cn^{-1/2}, \tag{4}$$

where C is a constant. A log-log plot such as Figure 4 may be used to compare the above efficiencies with this theoretical prediction.

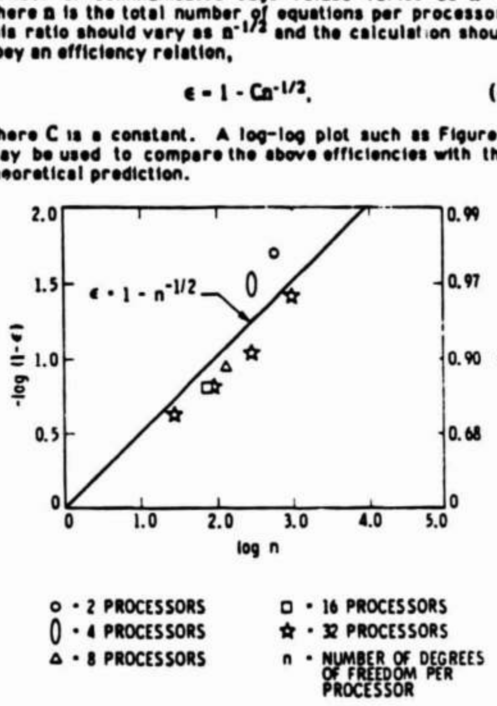


Fig. 4 Concurrent speedup efficiency as a function of problem size and number of processors

On this plot, star symbols indicate the 32-node efficiencies at various problem sizes. The results seem to agree well with the predicted *slope = 1/2* line. The points representing those cases with fewer than 16 processors plot above the 32-node trend because they do not involve processors communicating on all four boundaries. In summary, the concurrent conjugate gradient algorithm exhibits performance characteristics which are theoretically understood, and well suited for application to large ensembles of processors.

It is interesting to note that the experimental value for the constant C in (4) is apparently approximately 1. This result, which agrees with the results of Meier(8) for a two-dimensional finite difference scheme on the same machine, is a reflection of the fact that the times for calculation and communication of a single degree of freedom on this hardware are roughly equal.

CONCLUSIONS

We find that the concurrent conjugate gradient algorithm outlined here meets our expectations in providing large, nearly linear speedup in finite element system solutions. Results from actual runs on the 32-node Mark II hypercube system yield net efficiencies upwards of 90%. From this we can conclude that future major use of this and related algorithms on large finite element applications will hinge upon the applicability of iterative techniques, and not upon any issue of concurrency or efficiency.

Several areas are now indicated for future research in this area. Among the most immediate needs are: (1) effuctive preconditioners to speed convergence of the CG algorithm, (2) extension to three dimensions, (3) investigation of methods to reduce or eliminate stiffness storage requirements, and perhaps most importantly, (4) automated procedures for breaking up and balancing an arbitrary finite element problem among processors. We anticipate addressing these problems in the near future.

ACKNOWLEDGEMENTS

The research in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The authors also gratefully acknowledge the valuable advice and assistance of Prof. T. J. R. Hughes (Stanford Univ.). and that of S. W. Otto and M. A. Johnson (Caltech).

REFERENCES

1. Zienkiewicz, O.C., The Finite Element Method, 3rd ed., McGraw-Hill, London, 1977.

2. Heller, D., "Some Aspects of the Cyclic Reduction Algorithm for Block Tridiagonal Linear Systems," SIAM Journal of Numerical Analysis, vol.13, 1976, pp. 484-496.

3. Salama, M., Utku, S., and Melosh, R., "A Family of Permutations for Concurrent Factorization of Block Tridiagonal Matrices," 2nd Conference on Vector and Parallel Processors in Computational Science, Oxford, August 1984 (proceedings in press).

4. Sameh, A. and Kuck, D., "Parallel Direct Linear System Solvers - A Survey," Parallel Computers - Parallel Mathematics, Int'l. Assoc. for Mathematics and Computers in Simulation, 1977.

5. Jennings, A., Matrix Computation for Engineers and Scientists, John Wiley and Sons, Chichester, 1977, pp. 212-221.

6. Fox, G. C. and Otto, S. W., "Algorithms for Concurrent Processors," Physics Today, vol. 37, No. 5, May 1984.

7. Belytschko, T., Ong, J. S., Liu, W. K., Kennedy, J. M., "Hourglass Control in Linear and Nonlinear Problems," Computational Methods and Applications in Mechanical Engineering, vol. 43, 1984, pp. 251-276.

8. Meier, D.L., "Two-Dimensional, One-Fluid Hydrodynamics: An Astrophysical Test Problem for the Nearest Neighbor Concurrent Processor," Hm-90, July 1984, Caltech Concurrent Computation Project, Pasadena, Calif.