An efficient algorithm integrating a discretized mesh and a profile solver

TODD RASCOE and SHAUL SOREK

TAHAL Water Planning for Israel, P.O. Box 11170, Tel Aviv, Israel 61111

A procedure that links output from a discretised mesh to the input needed by a column profile solver is described. The procedure, based on a heapsort algorithm, takes arbitrarily ordered mesh output and prepares it for the highly structured input of the solver. The composition of the heapsort and the column profile solver were tested for the solution of a set of algebraic equations as assembled by the finite element method. The above procedure proved highly efficient in its use of storage and c-pu when compared with an accepted profile solver of a different method.

INTRODUCTION

Storing information from a discretised mesh within a global coefficient matrix with minimal memory allocation creates a high degree of applicability while being cost effective. Winget and Hughes[1] propose an efficient algorithm of a column profile solver for a system of linear equations. The Harwell report[2] exhibits a different variation of the profile solution method. Additional space saving methods were introduced by Puttonen[3] involving bandwidth reduction schemes. Williams' heapsort[4] with Floyd's approach,[5] will be utilised as a reliable medium to transform mesh output into column profile format input. A technique shall be presented in which mesh information, as presented by Neuman,[6] is made compatible with the column profile solver via the heapsort, to create a solution set with minimal storage.

Storage structure of stiffness matrix resulting from grid

The finite element approach is being widely used to solve systems of partial differential equations in the major engineering disciplines. By this method, one discretises a domain into finite cells and then creates a mesh structure which assembles information to be stored in arrays. Figure 1 displays the local nodal numbering method with the ordering of node connections.[6]

The matrix that results from the nodes of the finite mesh needs to be represented in a manner where all the information may be retrieved easily upon request during the solution section. This may be accomplished in a three-vector system, the row and column indices ('Rowind' and 'Colind') and the appropriate element of the global coefficient matrix ('Astiff'). These three vectors, regardless of the nodal input order, adequately present a low cost storage and retrieval system needed for solutions involing grid-like techniques.

The elegance of the column profile solver is a combination structuring of the input vectors and the modified Cholesky decomposition algorithm. The solver evolves from a three-vector framework. Two vectors ('Uptri' and 'Lowtri') contain the the values of the upper and lower triangles of the global stiffness matrix in profile form. The third vector ('Kdiag') stores the addresses of the elements of the main diagonal.[1] Notice that the first two corresponding vectors store one-half of the global stiffness matrix, while the third vector maintains a list equal in length to the number of equations.

Conversion from mesh structure to solver structure

How can we produce 'Uptri', 'Lowtri', and 'Kdiag' from 'Rowind', 'Colind', and 'Astiff'? Together 'Uptri' and 'Lowtri' are reorderings of 'Astiff'. This permutation is the column profile format. The input format is based on an ascending sort of the row or column index of the elements stored in 'Uptri' and 'Lowtri'. Therefore, two phases are made apparent. First, it is necessary to separate the stiffness matrix image ('Astiff' vector) into upper and lower triangles and second, sort the upper and lower triangles appropriately into a column profile storage scheme. The triangular separation is achieved by directly comparing 'Rowind' and 'Colind'. The reordering of the two triangles is accomplished by a sorting algorithm on the indices, known as heapsort.

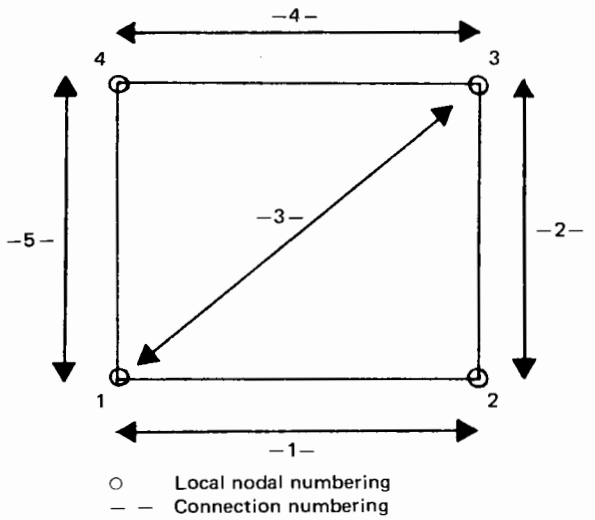

Figure 1. An element with local designation

Accepted June 1984. Discussion closes March 1985.

0141-1195/85/010042-03$2.00

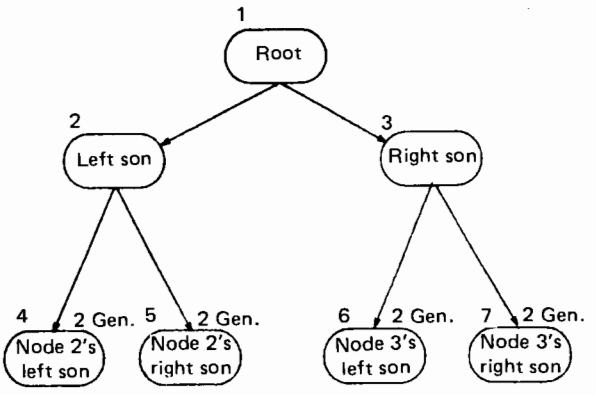

Figure 2. The heapsort binary tree with vector indices representing a father-son relationship

The heapsort has a binary tree structure (see Fig. 2) that maintains the root as its lowest (or highest if designated) element. Thus, the heapsort retains the property that:[7]

$$\text{Node}(\lfloor j/2 \rfloor) \Leftarrow \text{Node}(j)$$

$$\text{For } 1 \Leftarrow \lfloor j/2 \rfloor < j \Leftarrow \text{Num Nodes}$$

Such that:

$$\text{Node}(1) = \text{Min}[\text{Node}(1), \text{Node}(2), \ldots, \text{Node}(\text{Num Nodes})]$$

Now one must 'heapify'[8] the upper and lower triangles of the stiffness matrix. This may be perceived as a two key sort. The first key being the row or the column. The second being the rows within the column for the upper triangle or the columns within the row for the lower triangle. The outcome is 'Uptri' and 'Lowtri' in correct form for their respective triangles.

The heapsort algorithm is generally encountered in environments that are capable of dynamic storage allocation involving recursive routines for mobility within the tree. But, that environment is not necessary nor optimal in this application of the heapsort. The storage requirement for the heap need only concern itself with, at most, a triangle plus the main diagonal of the global stiffness matrix, as only one vector is sorted at a time. Therefore, the memory limitations, as declared for the vectors, are always known in advance. In addition, one need not worry that the size of the allocated dimension is the same as the amount dictated by the binary tree. The heapsort is not sensitive to this issue. The binary tree structure may be imitated in vector format according to the following relationships (see numbering of the nodes in Fig. 2):

$$\forall j : j \in \{2 \ldots \text{number of nodes}\}$$

$$\text{Father} = \text{Node}([j/2])$$

$$\text{Left son} = \text{Node}(2*[j/2])$$

$$\text{Right son} = \text{Node}((2*[j/2]) + 1)$$

The ordering of the input has very little effect upon the actual running time of the algorithm. As a result, total freedom is given to the creator of the discretised mesh when ordering the nodes within vectors 'Rowind' and 'Colind'. This solves the problems of dynamic storage allocation and recursion.

Analysis of algorithms

Two sets of profile solver code have been compared for purposes of time and space evaluations of the solutions. The first code is based on Gaussian elimination whereas the second code (a combination of heapsort and column profile solver) implements Cholesky's decomposition scheme. Both programs were tested in the IBM 370/VM environment on a FORTRAN IV compiler.

The test cases were standardised in all possible ways so as to avoid comparing unlike circumstances. The same rectangular discretised mesh was used in every case varying only the number of nodes. Length and width of the rectangular mesh, and input physical data remained unchanged. Eleven cases were executed according to the above restrictions. Three factors were estimated to assess the reliability and efficiency of the programs:

1. deviations in solutions
2. measured c-pu, and
3. amount of storage allocation.

The analysis of the results,for both codes, proved to be identical to one another, with their respective solutions. Thus c-pu and storage remained to be analysed.

In each case the second approach was faster in c-pu and required less storage than the first (see Fig. 3). The code in case two tended to be a constant time gap faster than the code in case 1. This indicates that the overhead to create the buffer space within case one is a fixed cost greater than that of case two. The difference occurs in the amount of buffer space, which is only created once.

The memory requirement displays much more significant findings. The second code requires approximately one-half the storage space of the first code (see Fig. 3b). The reason becomes apparent when measuring the large amount of scratch space (working buffers) needed in case one as compared with that of case two. The controlling factor of scratch storage in the first code was directly proportional to the number of nodes in the grid, whereas, this is not so in the second code. The storage limitations in the second code are dependent upon the number of divisions in the shorter direction of the rectangular mesh. It can be seen that the difference in storage per execution was much greater than the difference in time as illustrated in Fig. 4.

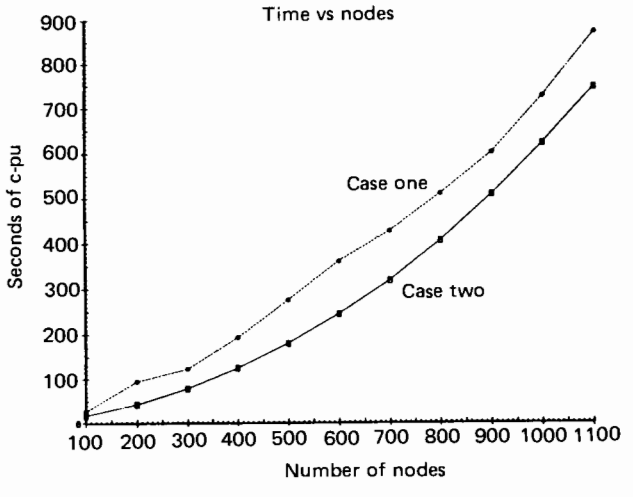

Figure 3a. Execution time as a function of the number of grid nodes

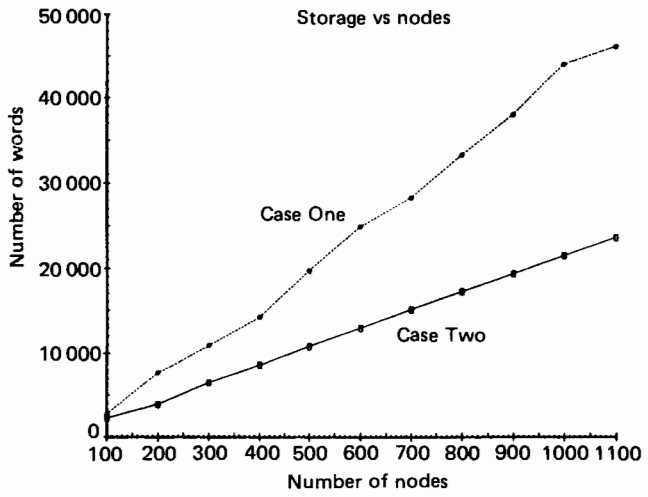

Figure 3b. Required buffer storage as a function of grid nodes

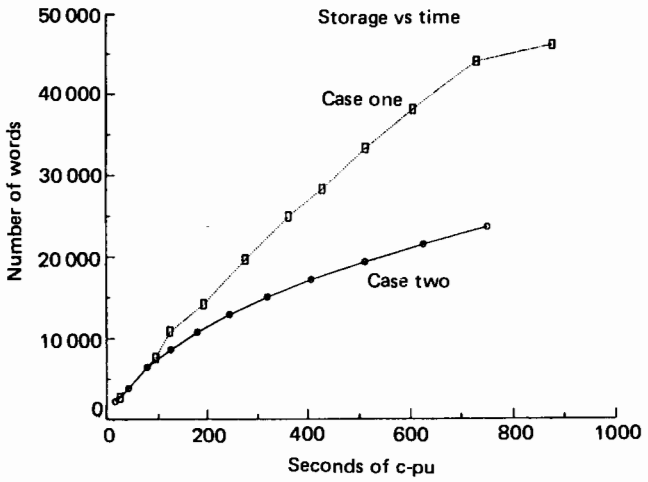

Figure 4. Comparison of the algorithm efficiency based on computer resources

One should notice that the number of divisions in the shortest direction of the grid dictates the bandwidth of the global matrix. With this in mind, it is necessary to number the mesh to receive the minimal number of cells in a designated direction.

CONCLUSION

The goal was to accomplish an integration between finite element methods and efficient solutions of the algebraic set of equations resulting from them. A technique was established taking mesh output, as associated with the global set of equations, to fulfil input requirements, as dictated by a column profile solver. Within this algorithm, it is possible to number the nodes of the mesh in any order so that the user is not restricted in either set of code. Results are shown relating two different profile solvers in 11 cases. The executions demonstrate that the combination of the integrative technique and the column profile solver yield higher efficiency of computer resources. The symmetric case, when applied, eliminates more storage and c-pu in the heapsort as well as the solver. Future modifications of the storage allocation are imminent. In the column profile solver, zero diagonals within the banded matrix are stored as are non-zero diagonals. Research to eliminate zero diagonals would significantly decrease storage allocation providing a more efficient algorithm.

REFERENCES

1 Winget, M. W. and Hughes, T. J. R. A profile solver for specially structured symmetric-unsymmetric equation systems, *Adv. Eng. Software* 1982, **4** (2), 64-67
2 Harwell Subroutine Library on Profile Solver, Computer Science and Systems Division, Atomic Research Establishment, Harwell, Oxfordshire, England
3 Puttonen, J. Simple and effective bandwidth reduction algorithm, *Int. J. Numer. Meth. Engng* 1983, **19**, 1139-1152
4 Williama, J. W. J. *Communications of the ACM* 1964, **7**, 347-348
5 Floyd, R. W. *Communications of the ACM* 1964, **7**, 701
6 Neuman, S. P. Finite element computer programs for flow in saturated unsaturated porous media, *Second Annual Report, Part 3*, Technion, Israel, 1972a
7 Knuth, D. E. *The Art of Computer Programming*, Vol. III: Fundamental Algorithms, Addison-Wesley, Reading, MA, 1973
8 Bentley, J. L. Computer Science 15-451, Applied Algorithm Design, Fall Semester, Carnegie-Mellon University, 1982