An Element-Based Concurrent Partitioned for Unstructured Finite Element Meshes

Hong Q. Ding and Robert D. Ferraro
Jet Propulsion Laboratory, MS 168-522, Pasadena, CA 91109

Abstract. *A concurrent partitioner for partitioning Unstructured finite element meshes on distributed memory architectures is developed. The partitioned uses an element-based partitioning strategy Its main advantage over the more conventional node-based partitioning strategy is its modular programing approach to the development of parallel applications. The partitioner first partitions element centroids using a recursive inertial bisection algorithm. Elements and nodes then migrate according to the partitioned centroids, using a data request communication template for unpredictable incoming messages. Our scalable implementation is contrasted to a non-scalable implementation which is a straightforward parallelization of a sequential partitioned. The algorithms adopted in the partitioned scale logarithmically, as confirmed by actual timing measurements on the Intel Delta on up to 512 processors for scaled size problems..*

1. Introduction

Finite element analysis is used in broad and diverse areas, such as structural analysis, fluid dynamics, electromagnetics, etc. Ever-increasingly larger and more complex mesh geometries used in practical applications can only be dealt with by the distributed memory parallel supercomputers because of their ability to scale to large number of processor without losing reasonable performance.

Partitioning a finite element mesh among the processors of a parallel supercomputer sets the stage for the finite element anal ysis problem, The domain partition achieves load balance, preserves proper data locality and reduces communications during the solution of the problem.

Partitioning algorithms, especially for simple grids, have been studied in considerable details (see [1,2] for summarics of recent related works). Most of these works study the grid mesh problem, and the number of edges being cut by the processor subdomain boundary is used as the measure of quality of the partitioner. However, partitioning a finite element mesh involves additional complexities duc to presence of the elements. ”

2. Node-based partitioning strategy

In a node-based partitioning strategy, one simply partitions the nodes (grids on the mesh that forms the elements). Therefore, each node belongs uniquely to a processor. Elements are then assigned to the nodes. Some elements will not be uniquely assigned because they have nodes which reside in different processors. If we simply assign onc such element to one of the relevant processors, that element has to remember that it has some nodes residing on other processors. This is inconvenient, because in finite element analysis, computations are done based on the elements, not the nodes. For these elements which have nodes on other processors, computations have to be carefully designed to get relevant nodal information from other processors. If adaptive refinement is required, nodes on other processors must be brought in so that all elements on the processor have all their nodes locally available before further refinements can proceed. Notice here that the number of edges being cut directly relates to the number of nodes needed to be brought in for element related calculations. This partitioning strategy has been used in [3].

3. Element-based partitioning strategy

Because finite element analysis is fundamentally element based, we prefer an element-based partition where an element in its entirety belongs to a processor uniquely (see Fig, “ 1). This implies that all the nodes of an element must be on the processor, We partition the finite element mesh by associating each element to its center of mass (centroid) and partitioning the resulting collection of centroids via a recursive inertial bisection algorithm. Once the elements are partitioned, nodes are migrated to the processor where their related elements are. Now, processor subdomain boundaries go along the edges, instead of cutting across the edges in a node-based partitioning, A node on processor subdomain boundaries is replicated on all processors which share it. A brief description of the element-based partitioner has been previously published in [4,5].

The most important feature of this partitioning strategy is that the local mesh resulting from the partitioned is a simply

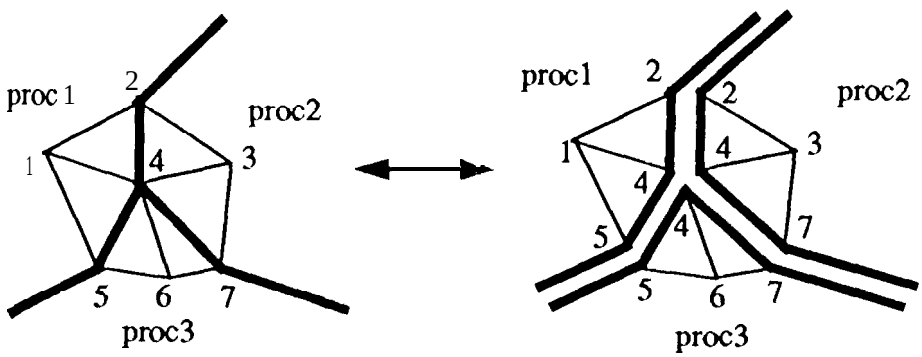
connected mesh, and all element-based calculations proteed as in the sequential case, without reference to any non-local information. As a result, most complicated sequential finite element analysis algorithms can be used without change. In Fig. 1, calculations of triangular elements A 124, A234 etc., and their contribution to stiffness matrix elements proceed exactly as in the sequential case. Further local adaptive refinements and multi-level solution methods could be also applied easily because all relevant information is locally available. Some of the boundary nodes of the local mesh are true boundary nodes subject to boundary conditions. Other boundary nodes are actually interior nodes, but on the processor subdomain boundaries. The finite element analysis treats these processor boundary nodes simply as interior nodes which are no different than other interior nodes, It is the parallel solver which connects the local meshes into a global mesh, constructs the global stiffness equation and solves it (see [7] for more details). This separation allows those in the application area to concentrate on the finite element analysis. This achieves much better modularity, and it is much easier to implement compared with node-based partitioning strategies.

In the following, we describe more details of our element-based concurrent partitioner which contains two major stages. First, the centroids are partitioned. Second, nodes and elements migrate according to centroids. We emphasize that algorithms used in both stages are scalable, i.e.., no worse than a logarithmic scaling. Finally we present several example applications and the timing measurements.

4. Recursive Inertial Bisection

The collection of element centroids form a mesh dual to the original node mesh. Partitioning of the centroids proceeds exactly as partitioning of grids. The edges in the centroid mesh does not correspond directly to anything in the original node mesh, but the cut of an edge in the centroid mesh directly corresponds to an edge in the original element mesh. Therefore, the number of edges being cut during the recursive partitioning of the centroid mesh equals the number of edges on the boundaries of the partitioned element mesh.

Although recursive spectral bisection is generally considered to give the best partitions, its parallel implementation involves solving large eigenvaluc-eigenvector problems which are difficult to implement efficiently on parallel computers. Recursive inertial bisection (RIB)[6,2] usually leads to reasonable partitions with reasonable aspect ratio, because in each recursive step, the remaining mesh subdomain is always cut into two across its current longest extension; this avoids long and thin subdomains often occurring in the standard recursive coordinate bisection. The RIB can be implemented in parallel with high efficiency. Our partitioner uses the RIB algorithm,



proc 1 proc 2 pmc 3

$$'44 = '44' + K_{44}^{\Delta 124} + K_{44}^{\Delta 145} + '44 + K_{44}^{A234} + K_{44}^{A347} + '44 + K_{44}^{\Delta 476} + K_{44}^{A465}$$

$$'24 = '24 + K_{24}^{A124} + K_{24}^{\Delta 234} + 0$$

Fig. 1. Element-based partitioning, Thick lines indicate processor boundaries. Node 4 is replicated on procs 1,2 and 3. Node 2 is replicated on procs 1 and 2, Contributions to stiffness matrix elements K_{44} and K_{24} from triangular elements Al 24, A234 etc. are also indicated.

Theoretically, the RIB algorithm completes in $\log_2(P)$ recursive steps, where P is the desired number of partitions (which is equal to the number of processors). However, $\log_2(P)$ steps does not imply a CPU time proportional to $\log_2(P)$, given the total problem size fixed. First, let us look at the basic steps in the RIB algorithm. A brief description follows. Each centroid has a flag indicating which region it belongs to. In the first step, there is only one region and all centroids belong to this region, We wish to divide this region into two. The inertial tensor is calculated, diagonalized, and the principle axis (which points to the longest extension) is found. All centroids arc projected onto this axis, which forms a one-dimensional array of floating point numbers. The median value of this array of numbers is calculated. Depending on whether its projection is lower or higher than the median, each centroid knows to which of the two regions it belongs. In the second recursive step, this process is repeated on the two regions independently to produce 4 regions. In the third recursive step, the 4 regions are divided into 8 regions. And finally, in the $\log_2(P)$-th recursive step, P/2 regions arc divided into P regions. From this description, wc see that there are $1 + 2 + 4 + \ldots + \frac{P}{2} = P - 1$ regions being calculated during the $\log_2(P)$ recursive steps, although the number of points in each region is reduced by half during each recursive step.

A straightforward conversion of the above sequential RIB algorithm to a parallel partitioner is not scalable. In that implementation[8], nodes and elements are read in from disk and arc distributed evenly among processors in some fashion. The basic RIB steps are performed without moving any data around. At the end, nodes and elements migrate to their final destination processor (or processors) according to the region flag, All the calculations of clement centroids, the region inertial tensor, eigenvector and median tinding arc carried out in a synchronous way, with every processor participated in all the calculations. The net effect is that each processor does work proportional to P.

A scalable implementation uses a processor group concept, a feature nicely supported by the Message Passing Interface (MPI) standard (although we have written a library [9] to implement partial operations on groups of processors in the Intel Paragon NX environment). Here, once the entire centroid mesh is divided in two regions, the centroids are physically moved to the relevant processors. For example, on 64 processors, all centroids with projection smaller than median go to processors O-31, and all other centroids to processors 32-63. In the next recursive step, the two partitioning processes proceed independently on the two processor groups to produce 4 subdomains on 4 processor groups. This process repeats until we have 64 subdomains on 64 processors. In this implementation, each processor does $\log_2(P)$ calculations of region inertial tensor, eigenvector, and median finding calculations. Although this is still more than the theoretical limit of $(P-1)/f = 1$, it grows much slower than the linear scaling in the above non-scalable implementation.

5. Migration and Load-Balance

In the node-based partition strategy, once the nodes arc partitioned, elements need to be migrated according to the partitioned nodes. When the relevant nodes of an element are distributed on different processors, a decision has to be made as to which processor to assign the element.

In the element-based partition strategy, once the elements arc partitioned, only nodes have to migrate accordingly. In our clement-bawl partition, nodes on subdomain boundaries are identified and replicated on relevant processors. A list containing these relevant processors' ids is replicated together with the node itself.

Our implementation of the element-based partitioned involves an extra stage, which simplifies the programming efforts. In principle we can let the elements go together with the centroids during the recursive bisection process, so that when recursive bisection finishes, elements are in the right processors. However, elements are "heavy" --- they contain additional information beyond the simple coordinates, and thus add an extra burden during the centroid redistribution following each recursive bisection. We prefer to move the elements only once at the end of the process. Another reason for migrating elements after centroids are partitioned is that an element has to inform its nodes to which processor they must migrate. If the element leaves the processor where its nodes reside, it has to have a mechanism to know in which processor these nodes are and send relevant information to this processor. These extra complexities are all absent if the element remains in the processor during the recursive partitioning of the element centroids and then migrate after it has informed its nodes about their destination processors.

Among the identically replicated nodes, only one is considered the original node owned by a processor, and others are considered copies of the original node (not owned by the processor), This ownership may be important to the later solution of the stiffness equation, For example, in our conjugate gradient salver implementation, node ownership is used to load balance computation, and arbitrate contributions to dot products[7].

6. Template for Unpredictable Incoming Messages

A data request protocol frequently occurs in the migration of elements and nodes. For example, the partitioned centroids request that the element structures migrate to centroids' processors. The requesting processor knows to which processor to send requests, but the receiving processor does not know how many messages it should expect and how long each message will be, This is the problem of unpredictable incoming messages.

We have designed a scalable (no worse than the logarithm of' number of processors) communication template to resolve this problem. It proceeds as follows:

(a) sort data requests on sending processor by destination processor (this information is stored in two arrays);
(b) call a global-sum() on one array to obtain the # of messages each processor should receive, and call a global-max() on the other array to obtain the maximum length of each message;
(c) make the correct number of calls to receive the requests with the maximum message length it expects.

Once data requests arc received, each processor sends the requested data back to the requesting processors. Element and node migration is implemented using this communication template. Minor modifications to the template codes are made to handle the complications duc to the variable number of nodes each finite element could have and due to the variable number of processors among which a node is shared .

7. Connection to a Sparse Matrix Solvers Package

The linear equations arising from finite element analysis is usually very large and sparse; its solution on a parallel architecture is also a main consideration. Fortunately, as mentioned above, constructing the local sparse coefficient matrix from local mesh partitions is a straightforward sequential process, which can be done by the user with their existing sequential algorithms.

The task of integrating local sparse matrices into the global sparse matrix (in fact, interpreting them as appropriate matrix blocks in the global matrix) and solving the global equation can be carried out by invoking an existing sparse matrix parallel solvers package[7] that wc have developed in connection with the partitioned. The solver suite deals with symmetric complex matrix problems. A preconditioned hi-conjugate gradient method, a two-stage Cholesky factorization method, and a hybrid method combining both methods have been implemented. All three solvers use a unified data interface so that users can switch to anyone of thcm at link time. This is quite convenient for those problems which are not positive definite. Furthermore, the local sparse matrix construction based on the local mesh partitions produced in our partitioned is well defined and is therefore standardized into subroutine calls in the solvers package. The user does not need to worry about the global sparse matrix organization at all; instead he/she concentrates on the ph ysics problem itself. We emphasize that this modular programing approach to parallel computing is made possible by our element-bawd mesh partitioning strategy,

8. Performance Characteristics

Wc measured two performance characteristics of the concurrent partitioner on the Intel Delta withupto512 processors. The data is either a *32,768* hexagonal element mesh (squares in Fig. 2) or a 24,264 tetrahedral element sphere-cylinder (circles in Fig. 2). The fixed problem size performance (speedup) is shown in Fig. *2*. In the region from small to medium number of processors (up to 128 processors), the total time reduces as the number of partitions increases. However, as the number of processors becomes larger than 128 (i.e., the resulting number of partitions becomes larger than 128), the total time hit the plateau. This is expected, since various overheads in the parallel algorithm remain fixed or increase slightly with the number of processors and thus eventually become dominant.

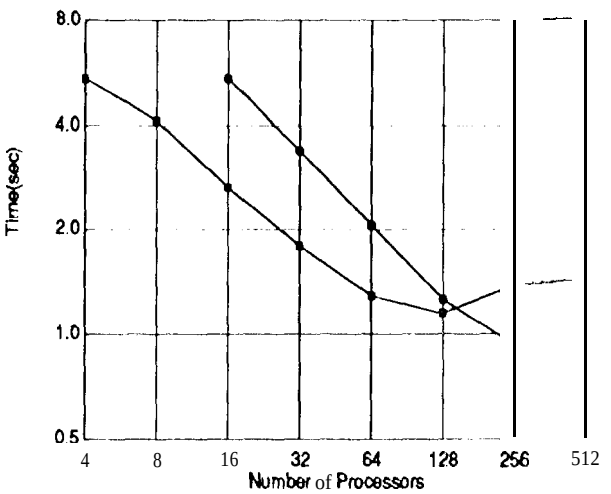


Fig.2. Execution time for two problems with fixed sixes

The scaled problem behavior was also studied (see Fig.3). On 4-processors, the partitioned takes 0.21 sec to partition the 512 element problem (each element is 8-node hexagon). The 4096-element problem on 32-processor takes 0.51 sec. while the 32,768-element problem on 256-processor takes 0.93 sec. If we take 4-processors as the minimum processor size where a partitioning algorithm make senses and normalize all timing accordingly, a logarithmic scaling is clearly followed for this scaled problem set:

$$T(P)/T(4) = 0.8 \log_2(P/4).$$

This indicates the scalable nature of the algorithms implemented in this partitioned.

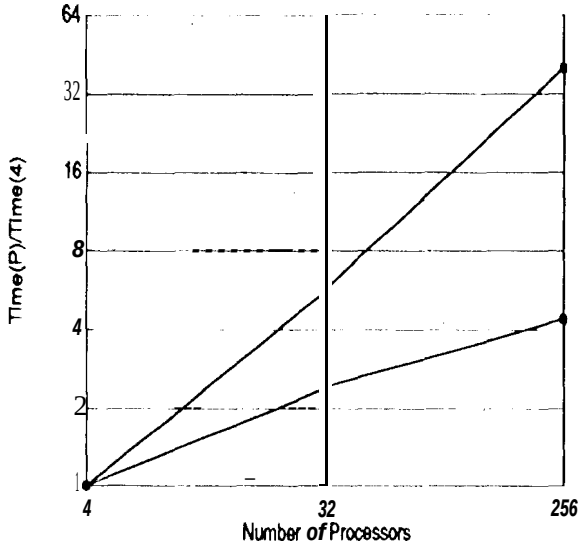


Fig.3 Execution time for a scaled size problem. 128 hexagon elements per processor

In comparison, an earlier non-scalable implementation[8] results are also shown in Fig.3 as the top curve.

9. Summary

Wc have developed a concurrent partitioned for partitioning unstructured finite element meshes on distributed memory architectures using an element-based partitioning strategy. We examined the scalable implementation of the recursive inertial bisection algorithm and discussed issues related to migrating nodes and elements. Test runs of our partitioned on large meshes indicate a logarithmic scaling with problem size for fixed element/processor ratio, thus demonstrating the Statability of the algorithms implemented in this partitioned, Finally, we have emphasized a modular programing approach to separate the application specific parts from the parallelization, so that users can concentrate on their own applications.

Acknowledgment

We thank Jim McComb for explaining the details of an old implementation [8]. We thank the Concurrent Supercomputing Consortium for the use of Intel Delta. This work was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with National Aeronautics and Space Administration.

References

1. H, D. Simon, Partitioning for Unstructured Problems for Parallel Processing, *Computing Systems in Engineering, v.2,* p.135, 1991.
2. R. Leland and B. Hendrickson, An Empirical Study of Static Load Balancing Algorithms, *Proceedings of Scalable High Performance Computing Conference,* 1994, IEEE Computer Society Press, Los Alamitos, CA, p 682.
3. M.T. Jones and P.E. Plassmann, Parallel Algorithms for Adaptive Refinement and Partitioning of Unstructured Meshes, *Proceedings of Scalable High Performance Computing Conference,* 1994, IEEE Computer Society Press, Los Alamitos, CA. p 478.
4. H.Q. Ding and R.D. Ferraro, Slices: A Scalable Partitioner for Finite Element Meshes, *Proceedings of 7th SIAM Conference on Parallel Processing for Scientific Computing, 1995,* SIAM Press, Philadelphia, PA. p.633.
5. The basic idea of element-based partitioning dates back to early 80's. A good description along with early reference is in G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Processors I,* Prentice Hall, Englewood Cliffs, NJ, 1988, chap. 8. See also [6,7].
6. B. Nour-Omid, A. Racfsky and G. Lyzenga, Solving Finite Element Equations on Concurrent Computers, in *Proceedings of the Symposium on Parallel Computations and their Impact on Mechanics,* Ed. A. K.NoOr, ASME, New York, 1988, p.209.
7. H.Q. Ding and R.D. Ferraro, A General Purpose Sparse Matrix Parallel Solvers Package, *Proceedings of 9th Internal Parallel Processing Symposium,* 1995, IEEE Computer Society Press, Los Alamitos, CA. p 70.
8. J.C. McComb, et. al., Parallel Implementation of the Recursive Inertial Partitioning Algorithm, unpublished JPL report, 1993.
9. H.Q. Ding, PGLIB: A Library of Partial Global Operations on Parallel Architectures, See web page at http://olympic.jpl.nasa.gov.