Parallel Structures and Dynamic Load Balancing for Adaptive Finite Element Computation

J. E. Flaherty, R. M. Loy, C. Özturan
M. S. Shephard, B. K. Szymanski,
J. D. Teresco and L. H. Ziantz

Scientific Computation Research Center (SCOREC)
and Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180

An adaptive technique for a partial differential system automatically adjusts a computational mesh or varies the order of a numerical procedure to obtain a solution satisfying prescribed accuracy criteria in an optimal fashion. We describe data structures for distributed storage of finite element mesh data as well as software for mesh adaptation, load balancing, and solving compressible flow problems. Processor load imbalances are introduced at adaptive enrichment steps during the course of a parallel computation. To correct this, we have developed three dynamic load balancing procedures based, respectively, on load imbalance trees, moment of inertia, and octree traversal. Computational results on an IBM SP2 computer are presented for steady and transient solutions of the three-dimensional Euler equations of compressible flow.

1 Introduction

The finite element method (FEM) has become a standard analysis tool for solving partial differential equations (PDEs). Adaptive FEMs have gained importance because they provide reliability, robustness, and time and space efficiency. During the solution process, portions of the discretized domain are spatially refined or coarsened (h-refinement), the method order is varied (p-refinement), and/or the mesh is moved to follow evolving phenomena (r-refinement), to concentrate or dilute the computational effort in areas needing more or less resolution [10].

Preprint submitted to Elsevier Preprint *25 March 1997*

Parallel computation is essential for computationally demanding three-dimensional problems; however, it introduces complications such as the need to balance processor loading, coordinate interprocessor communication, and manage distributed data. The standard methodology for optimizing parallel FEM programs relies on a static partitioning of the mesh across the cooperating processors. However, with adaptive software, a good initial partition is not sufficient to assure high performance. Load imbalance caused by adaptive enrichment necessitates a dynamic partitioning and redistribution of data.

Tools developed at the Scientific Computation Research Center (SCOREC) at Rensselaer to facilitate the development and use of parallel adaptive finite element software are described in Section 2. An object-oriented, hierarchical mesh database is used to store and manipulate mesh data [3]. Meshes are created by an automatic finite octree procedure [35]. Parallel extensions to the mesh database allow operations to be performed on distributed data and provide for the dynamic migration of finite elements [7,30]. Parallel mesh enrichment routines are used for spatial refinement and coarsening (h-refinement) [34]. Such reusable software libraries are essential to provide the ability to solve diverse problems, each of which presents its own challenges. These tools allow application software to be written in a uniform way and enable the programmer to concentrate on issues specific to the problem at hand rather than the details of the underlying mesh structures or parallelization concerns.

The quality of the data partitioning is an important efficiency factor (Section 3). One measure of partition quality is the percentage of elements which require access to off-processor data during the computation. On a distributed-memory parallel computer, poor partition quality results in a higher communication cost during the finite element solution phase. Static partitioning methods based on coordinate [4], inertial [20], and spectral [32] bisection are used to reduce communication cost when distributing initial meshes. A parallel version of the inertial partitioning method [34] may also be used for dynamic rebalancing. However, in an adaptive computation, global partitioning strategies can be costly relative to solution time. Thus, a number of iterative dynamic load balancing techniques that incrementally migrate data from heavily to lightly loaded processors have been developed [5,7,14,15,17,26,39,41]. An iterative method based on load imbalance trees [30,34] is available within our system. While these methods provide inexpensive ways to achieve a balanced computation, they can lead to degradation of partition quality. We also use an octree-based partitioning, which takes advantage of an underlying tree structure to achieve balance and to maintain reasonable communication costs [17,28].

Presently, mesh refinement and coarsening precede a balancing step. Were we able to predict imbalance prior to refinement, we could maintain better performance through the enrichment and subsequent computational steps. A strategy for doing this is described in Section 3.4.

Partitions often have jagged boundaries with elements penetrating into or protruding from neighboring partitions. Such features increase communication costs. As described in Section 3.5, interprocessor boundary smoothing may be used as a post-processing step to improve the quality of any load balancing procedure [19,21,28].

We solve compressible steady and transient flow problems on an IBM SP2 computer to demonstrate the capabilities of the parallel adaptive system. A steady conical flow (Example 1, Section 4.2) is used to compare load balancing procedures. The analysis of a transient shock impacting a cone (Example 2, Section 4.3) is used to demonstrate the advantages of predictive load balancing. The solution of a transient flow in a muzzle brake is shown as Example 3 in Section 4.4. In Section 5, we discuss results and present future research directions.

2 SCOREC Mesh Tools

2.1 SCOREC Mesh Database

The *SCOREC Mesh Database* (MDB) [3] provides an object-oriented hierarchical representation of a finite element mesh. It also includes a set of operators to query and update the mesh data structure. The basic mesh entity hierarchy consists of three-dimensional *regions*, and their bounding *faces*, *edges*, and *vertices*, with bidirectional links between mesh entities of consecutive order (Figure 1). In three-dimensional meshes, regions are used as finite elements while faces serve as elements in two-dimensional meshes. Mesh entities are explicitly classified relative to a geometric model of the problem domain to allow for the appropriate representation of the geometry as the mesh is enriched.

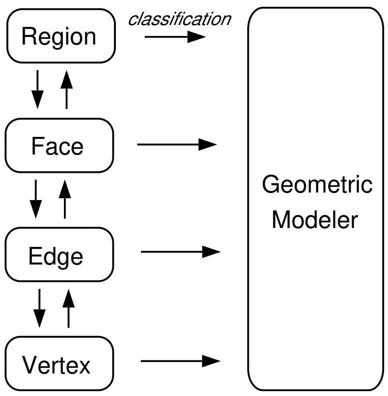


Fig. 1. MDB entity hierarchy, with links to a geometric modeler.

The full entity hierarchy allows the efficient deletion and creation of mesh entities during h-refinement and simplifies attachment of degrees of freedom to the mesh entities during p-refinement [33]. The database allows for the fast retrieval of adjacency information. Examples of available data include the list of faces bounding an element, and the edges sharing a common vertex. All entities can have attached attributes such as solution and error indicator data.

2.2 SCOREC Finite Octree Automatic Mesh Generator

Initial meshes are created using the *SCOREC Finite Octree Automatic Mesh Generator* [35]. Beginning with a geometric model of the domain obtained from CAD software, the mesh generator first discretizes the boundary and then recursively subdivides the domain into cubes called octants to create a variable level octree. The level of local subdivision is consistent with element size on the domain boundary. Octants are classified relative to the problem domain as interior, exterior, or boundary. Exterior octants do not intersect the domain and receive no further consideration. Interior octants are discretized using templates. Boundary octants are discretized by face removal procedures that connect the boundary triangulation to the interior octants.

2.3 Parallel Mesh Database

A *Parallel Mesh Database* (PMDB) [30,34], which provides operators to create and manipulate distributed mesh data, is built on top of MDB. Using PMDB, each processor holds MDB data associated with a subset of the complete mesh. Entities along partition boundaries are shared by more than one processor (Figure 2), and are maintained by a partition boundary data structure. In three-dimensional distributed meshes, each region is assigned to a unique processor, but bounding faces, edges, and vertices of regions along an inter-processor boundary are duplicated on each processor that contains a region using that entity.

Links to off-processor data copies are stored as processor-address pairs called *uses*. Each partition boundary entity keeps a list of its uses. A duplicated partition boundary entity has a unique owning processor that can be determined by finding the minimum ordered pair (p_o,a_o) in the list of uses for that entity using the address as the most significant key. Here p_o is the id of the owning processor and a_o is the address of the entity on p_o. This ownership information can be used to implement an owner-computes rule [9], *e.g.*, during scalar product computation in an iterative linear solver. Since (p_o,a_o) functions as a *global key* for an entity, there is no need to generate and store a separate key

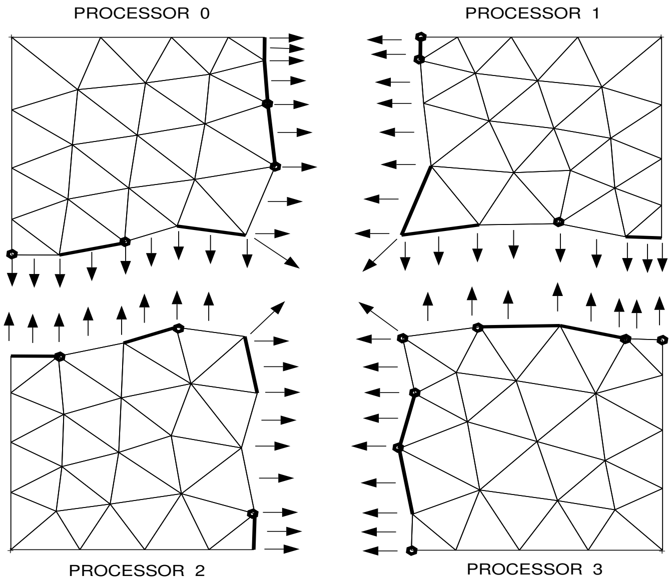


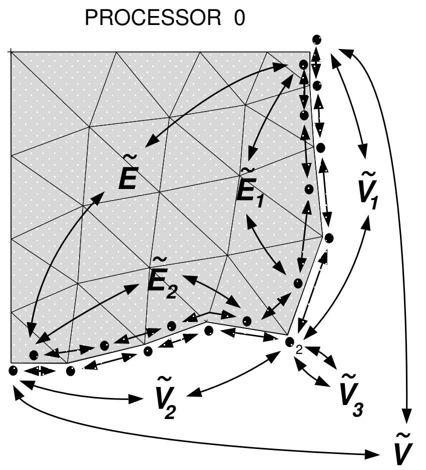
Fig. 2. Two-dimensional example of a distributed mesh. Arrows represent interprocessor pointers between boundary entities. Heavy edges and vertices indicate the unique owner of each shared boundary entity.

by computing the centroid of the entity [42]. Global key generation can, thus, be replaced by the incremental and faster process of ownership regeneration of affected partition boundary entities (Section 2.4).

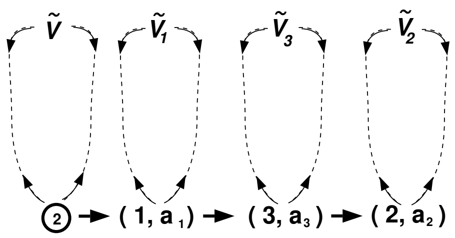
Sets of entities on a partition boundary adjacent to a specific processor are organized as doubly linked lists (Figure 3(a)) allowing constant-time insertion and deletion. These can be used to construct lists of partition boundary entities that are shared among processors. Doubly linked lists are also used to store the information needed to maintain processor adjacencies based on various entity connectivities and the number of entities adjacent to the processor (Figure 3(b)). PMDB provides fast query and update operations on the boundary structure. For example, a finite element procedure can obtain scatter/gather maps of data for use in its communication phase. Fast traversal of entities on interprocessor boundaries is provided by following the interprocessor boundary structure lists. All interprocessor communication is done using the *Message Passing Interface* (MPI) [22].

2.4 Mesh Migration

PMDB handles arbitrary multiple migration of elements between processors (Figure 4) to maintain a balanced computation. Any top-level entity can be marked for migration, although it is frequently the boundary elements that are migrated. The migration procedure uses an owner-updates rule to collect and update any modifications to links on partition boundaries. As illustrated



(a)



(b)

Fig. 3. Doubly linked structures of partition boundary entities: (a) global view and (b) partition boundary view. $\tilde{\mathrm{E}}$ and $\tilde{\mathrm{V}}$ denote lists of all partition boundary edges and vertices, respectively. Symbols with subscripts indicate lists of entities adjacent to a particular processor. The circled number two in (b) corresponds to the small number two in (a). The ordered pairs are processor-address pairs.

in Figure 4, mesh migration is done in three main phases: (*i*) senders migrate mesh entities to receivers, (*ii*) senders and receivers send the migrated boundary to the owners, and (*iii*) owners update the boundary data structures and notify affected processors of the new location of each entity. The processing done in the first stage is proportional to the number of entities being migrated, while the complexity of the second and third stages is proportional to the number of entities in the boundary of the submesh being migrated. As a result, if each processor migrates to a small number of processors, such as its neighbors, the migration will scale with the number of processors [30].

2.5 SCOREC Mesh Enrichment

The SCOREC *mesh enrichment* [34] procedure performs spatial (h-) refinement and coarsening in parallel using error indicator information and enrichment threshold values. From this information, mesh edges are marked to be

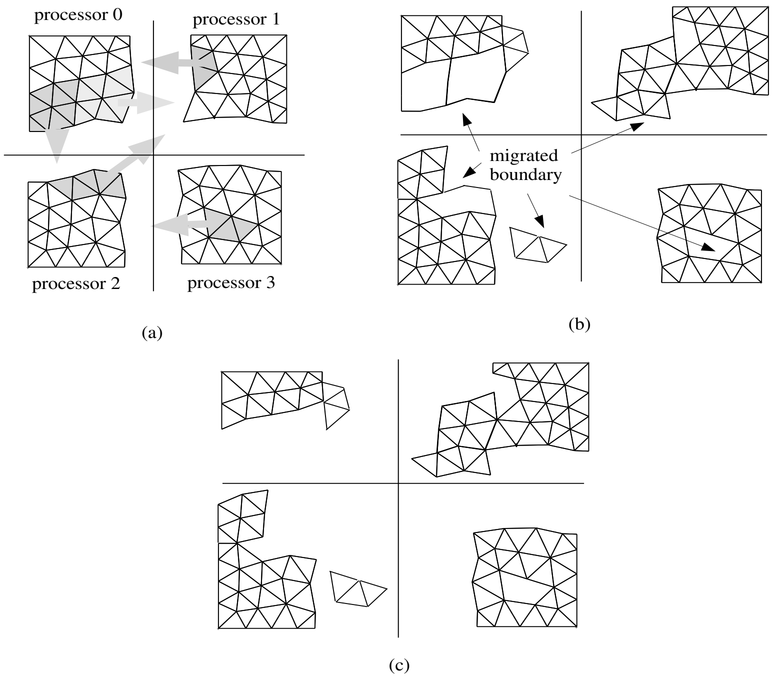


Fig. 4. Example of arbitrary multiple migration illustrating the three-stage process: (a) senders migrate mesh entities to receivers, (b) senders and receivers send the migrated boundary to the owners, and (c) owners update the boundary data structures and notify affected processors.

coarsened, refined, or unchanged. Mesh enrichment is done in stages following the order of (*i*) coarsening, (*ii*) optimization (optional), (*iii*) refinement, (*iv*) optimization (optional), (*v*) refinement vertex snapping, and (*vi*) optimization (optional), as illustrated on the left of Figure 8. Coarsening collapses a marked edge to one of its end vertices. Regions connected to the collapsed vertex that cease to exist are deleted to form a polyhedral cavity, and the faces of the cavity are connected to the target vertex to form new mesh regions. Mesh optimization improves the quality of triangulations with respect to a given criterion (*e.g.*, element shape). Refinement is performed using subdivision patterns. First, faces on partition boundaries with marked edges are triangulated using two-dimensional refinement templates. Each processor then independently applies three-dimensional patterns that have been determined for every configuration of marked edges (Figure 9). The enrichment process has an over-refinement option which reduces element shape degradation at the expense of creating more elements. In the final stage of enrichment, vertices created by the refinement process that are classified as belonging to a curved model boundary must be "snapped" to the appropriate model entity to ensure mesh validity with respect to the geometry of the problem domain.

3 Partitioning and Load Balancing

The distribution of data across the processors of a parallel computer greatly affects performance. A balance of computational load is necessary to avoid idle processors, but does not suffice to ensure efficient parallel computation. Research on mesh partitioning to date has focused on minimizing the number of "cuts" that the subdomains create when the partitioning is viewed as a communication graph whose vertices represent computation and whose edges represent data dependencies [23,24,36]. With finite volume and finite element schemes this closely corresponds to the task of minimizing the number of element faces on interprocessor boundaries. This metric is an excellent indication of the amount of local data that must be communicated to perform a computation, especially with higher-order methods where three-dimensional facial modes increase as the square of the polynomial degree.

We appraise the cost of interprocessor communication by two surface indices of partition quality [8]. The *maximum local surface index* (MLSI) measures the maximum percentage of element faces on the boundary of any processor, and the *global surface index* (GSI) measures the percentage of all faces on interprocessor boundaries. For the discontinuous Galerkin methods used for the computations of Section 4, the GSI is equivalent to the number of edge "cuts" in the communication graph normalized by the total number of these edges. Normalization makes the measure independent of problem size. These surface indices can be thought of as surface-to-volume ratios if the concepts of surface and volume are expanded beyond conventional notions. With current technology, message startup is a significant component of communication cost; therefore, interprocessor connectivity (the number of processors with which each processor must exchange information during the solution phase) is as significant a factor in performance [8] as the number of boundary faces.

Three static partitioning procedures are available in PMDB to distribute mesh data initially. *Orthogonal Recursive Bisection* (ORB) [4], also called *Recursive Coordinate Bisection*, uses the coordinates of element centroids to partition the mesh. At each recursive step, the Cartesian coordinate of the longest dimension of the domain under consideration is bisected, elements are sorted according to the bisecting coordinate, and half of the elements are assigned to each subdomain. *Inertial Recursive Bisection* (IRB) [20], proceeds likewise, but in a direction orthogonal to its principal axis of inertia. *Recursive Spectral Bisection* (RSB) [32] is generally considered to be among the best static mesh partitioning procedures. RSB is costly and may be too expensive for use in a large-scale three-dimensional adaptive computation. ORB and IRB are available as initial partitioning methods in PMDB while RSB is available in Chaco [23] and other packages.

A dynamic load balancing scheme that operates on distributed mesh data is essential for adaptive computation. *Multilevel Recursive Spectral Bisection* (MRSB) [2] has improved the efficiency of RSB, but its parallelization relies heavily on a shared memory architecture and is unlikely to be efficient in a true message passing environment [1]. Other enhancements to RSB [37,38,40] may make it more useful as a dynamic repartitioner, but serious doubts remain. Three dynamic load balancing schemes are available for use with PMDB data structures and mesh migration operators. *Iterative Tree Balancing* [30,34] (ITB) performs repeated local migrations to achieve balance. *Parallel Sort Inertial Recursive Bisection* [34] (PSIRB) uses IRB with a parallel sort. *Octree Partitioning* [17] (OCTPART) uses the octree structure underlying the mesh to achieve load balance.

The measure of imbalance or "cost function" that reflects the computational load on each processor is generally chosen as the number of elements on a processor with h-refinement. However, heterogeneous costs are necessary when (i) using p-refinement or spatially-dependent solution methods, (ii) using spatially-dependent time steps, (iii) enforcing boundary conditions, and (iv) using predictive load balancing (Section 3.4). PMDB provides an element weighting scheme that can be used to address each of these needs.

3.1 Iterative Tree Balancing

ITB follows Leiss and Reddy [26], Wheat [41], and Devine and Flaherty [17] in that lightly loaded processors request load from their most heavily loaded neighbors. However, instead of considering an immediate neighborhood of processors, the algorithm views the requests as forming a forest of trees (Figure 5b). Each tree is then linearized, and a logarithmic-time scan operation is used to compute load flows [30] to determine the amount of data to be migrated (Figure 5c). Layers of elements on interprocessor boundaries are moved from heavily loaded to lightly loaded processors to achieve balance within each tree (Figure 6) [30]. ITB is "diffusive", and a heavily loaded processor will distribute load to several lightly loaded neighbors. ITB may be iterated to achieve a global balance within a specified tolerance or be set to terminate after a fixed number of iterations. With low per-iteration costs, ITB can be executed for a few iterations between substages of operations like mesh enrichment [7,34] without the large time penalty of a global repartitioning.

3.2 Parallel Sort Inertial Recursive Bisection

Parallel sorting of elemental coordinates in the inertial frame enables IRB to be used as a dynamic repartitioning procedure called PSIRB. Thus, mesh data

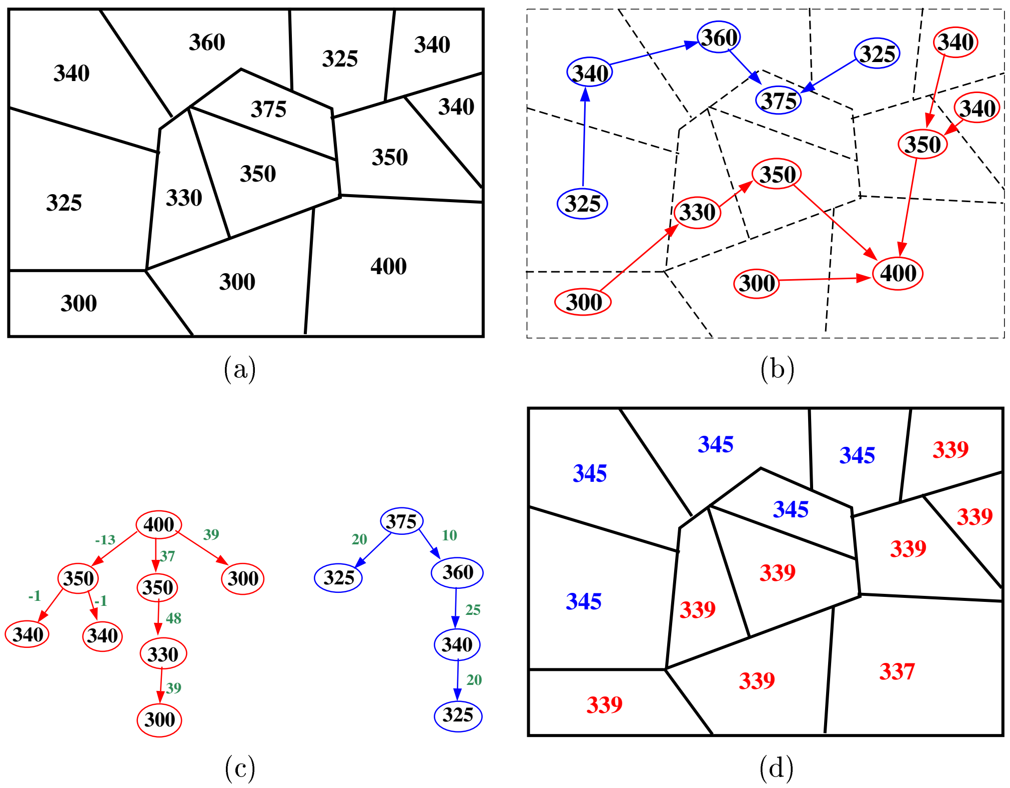


Fig. 5. An iteration of ITB: (a) original, unbalanced loads, (b) load requests, (c) forest of trees induced by load requests, and (d) loads after one iteration.

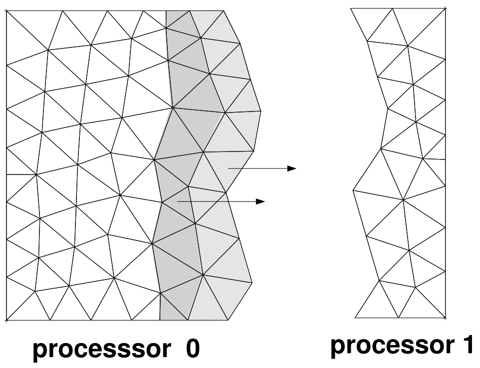


Fig. 6. ITB “slice-by-slice” element selection

can be distributed before the initial PSIRB invocation. A partitioning of the initial mesh used in Example 3 (Section 4.4) distributed with PSIRB is shown on the left of Figure 15. This partition has an MLSI of 12.9% and a GSI of 3.2%.

3.3 Octree Partitioning

Octree-based partitioning [17] employs automatic octree generation procedures [35] and uses tree topology to create a one-dimensional ordering of the octree nodes. The ordered list of nodes is divided into segments corresponding to nearly equal load. Members of any given segment tend to be spatially adjacent and, thus, form a good partition. Minyard *et al.* [28] also present an octree-based partitioning procedure that uses orthogonal coordinate bisection. The use of space-filling curves [31] is an alternative that also keeps neighboring elements of the ordering in close spatial proximity.

Initially, cost metrics of all subtrees of the octree are determined to indicate loading in specific spatial regions. The cost metrics, usually the number of elements in an octant, may be weighted by polynomial degree or other factors to indicate heterogeneity. The second phase of the balancing uses the cost information to construct partitions. Since the total cost is known, the optimal partition size is also known. Each partition consists of a set of subtrees determined by a truncated depth-first traversal beginning at the root (Figure 7a). Subtrees are visited during the traversal and added to a current partition if they fit (Figure 7b). If a subtree exceeds the optimal size of the current partition (Figure 7c), a decision must be made as to whether the subtree should be added, or whether the traversal should examine it further. In the latter case, the traversal continues with the offspring of the node, and the subtree may be divided among two or more partitions (Figure 7d). When the imbalance at a node is too large to justify inclusion in the current partition and the node is either terminal or sufficiently deep in the tree, the partition is closed (Figure 7e), and subsequent nodes are added to the next partition. The process continues until the traversal is complete (Figure 7f).

The decision on whether to add a subtree to a partition or to examine it further is based on the amount by which the optimal partition size is exceeded. A small excess may not justify an extensive search and may be used to compensate for some other partition which is slightly undersized.

After initial mesh partitioning, the tree and mesh data are distributed to the processors. Tree links may be local or off-processor, allowing each processor to store part of the octree and its associated mesh. There is no replication of the octree. Let N_{max} be the maximum number of elements on a processor, P be the number of processors, and N be the number of mesh elements. Then dynamic rebalancing may be performed in parallel in $O(N_{max})$ time, assuming $N_{max} > \log P$. For a nearly balanced mesh, this approaches $O(N/P)$. Partitioning time does not grow with P as it would for a recursive algorithm. In terms of scalability, our algorithm is more advantageous than that of Minyard *et al.* [28] who store the global octree on each processor. This effectively pro-

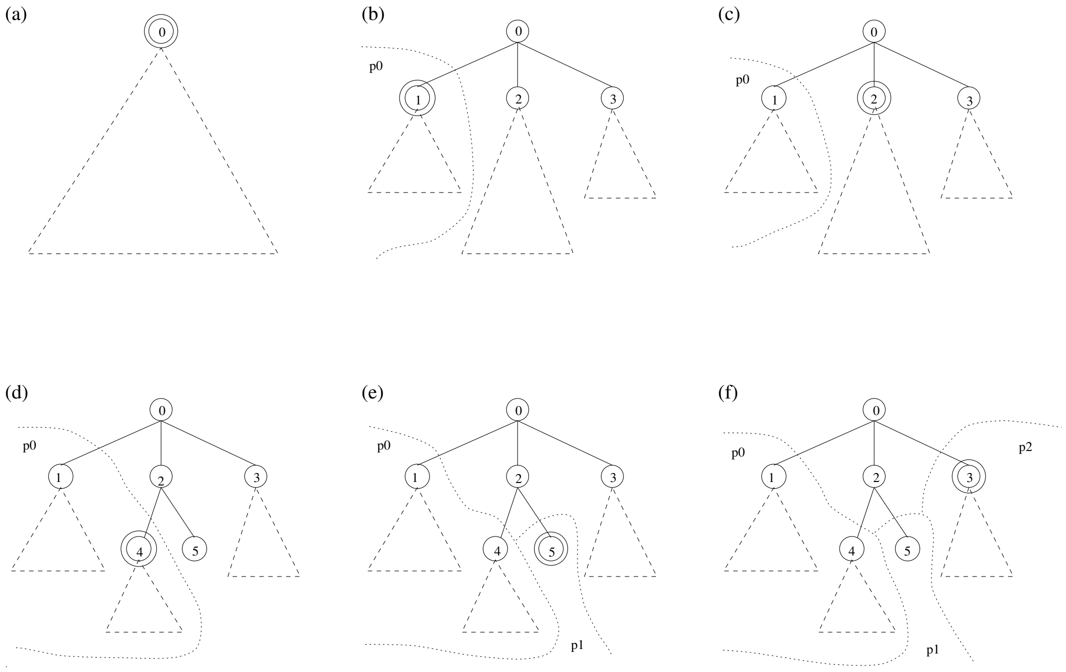


Fig. 7. Depth-first traversal for partitioning using OCTPART.

duces a procedure having serial complexity since the partitioning is duplicated on all processors. However, global information located on a single processor makes possible more informed decisions and may lead to higher quality partitions.

A partitioning of the mesh of Example 3 (Section 4.4) onto eight processors using OCTPART is shown on the right of Figure 15. This initial partition has an MLSI of 9.6% and a GSI of 3.8%. Thus, OCTPART and PSIRB have comparable GSIs, but OCTPART's lower MLSI indicates a more uniform partitioning in this instance.

3.4 Predictive Load Balancing

At present, enrichment precedes a full load balancing, although a few ITB iterations are performed between substages of the enrichment to maintain some balance (left of Figure 8). However, the work done during the enrichment procedure is not necessarily proportional to the number of elements on a processor at the start of refinement. Thus, balancing using a unit load per element may not be appropriate. We hope to improve this by using the error indication data to select element weightings and perform load balancing before the refinement stage of the enrichment process (right of Figure 8). This predictive load balancing should reduce imbalance during both the refinement stage of enrichment and the subsequent solution phase and result in fewer elements being migrated. At present, predictive load balancing is performed using weighted ITB (WITB); however, any load balancing procedure that recognizes elemental weights may be substituted, *e.g.* , OCTPART.

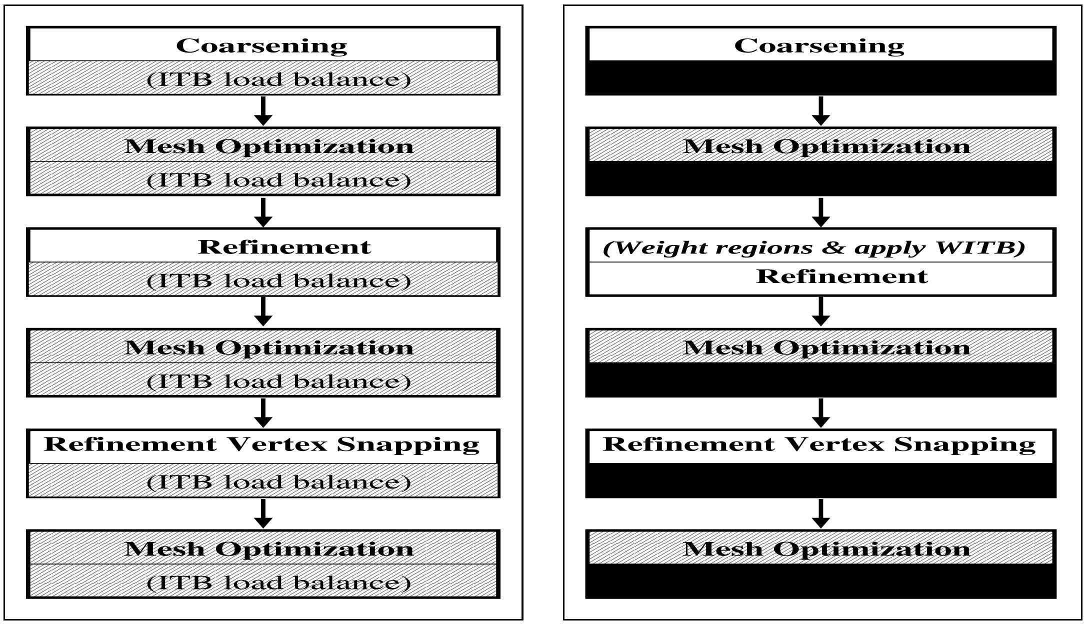


Fig. 8. Current nonpredictive (left) and predictive (right) enrichment procedures. Shaded portions indicate optional steps.

Weight assignments for h-refinement are made after edges have been marked for splitting based on error indicator information and a refinement threshold but before the refinement is actually performed. The weighting is based on the three-dimensional subdivision patterns used by the refinement algorithm (Section 2.5) as indicated by the parenthetical numbers in Figure 9. An element is assigned a unit weight if none of its edges are marked. If p-refinement were being performed, the number of degrees of freedom associated with each element would also have to be considered in the weighting.

Using the predictive balancing, a small imbalance may occur in both the refinement and the subsequent numerical computation. Imbalance seen in the solution phase is a result of the migration of mesh elements that occurs during refinement vertex snapping and mesh optimization. It can also be caused by the deletion and creation of elements in the optimization stage. A greater imbalance can occur when over-refinement (Section 2.5) is used since more subdivisions will be performed than predicted by the original edge marking. Imbalance after predictive balancing may also occur during refinement if many elements with large weights come to reside on a few processors while the remaining processors have mostly unit weight elements.

Table 1 compares the nonpredictive and predictive enrichment procedures illustrated in Figure 8 for Example 2 (Section 4.3). Tests were run on 16 processors of an IBM SP2 computer. The table shows timings for a sequence of ten meshes generated during a transient adaptive analysis. The times shown for

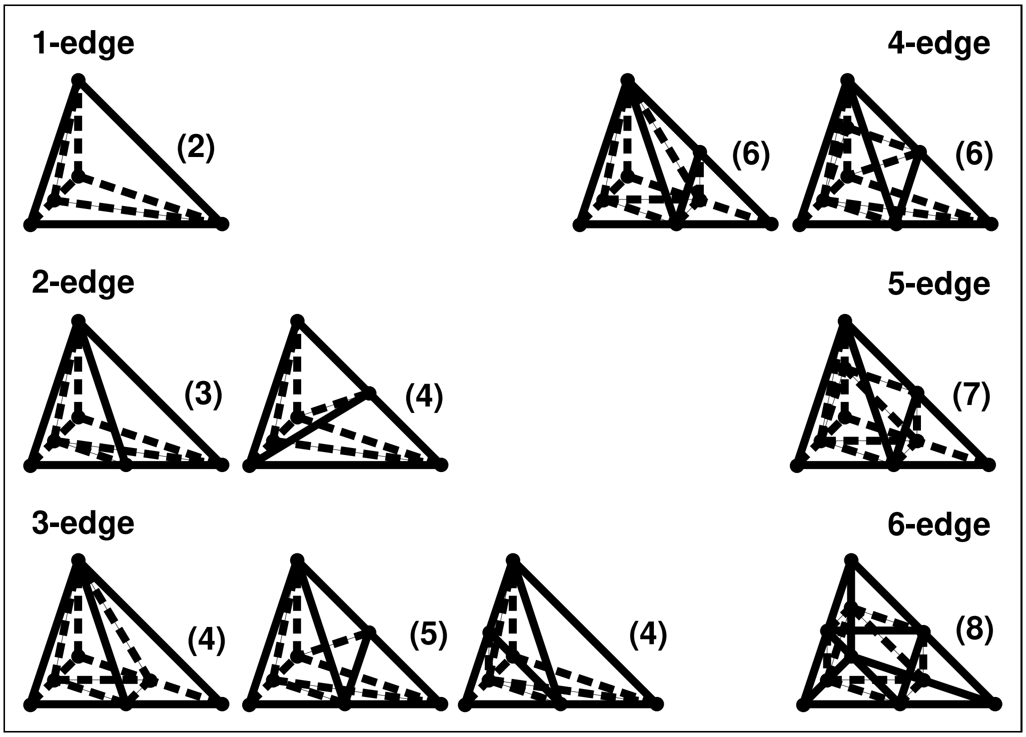


Fig. 9. Weights for subdivision patterns and predictive load balancing.

each mesh were averaged over ten runs. The "Size" column gives the number of elements in each mesh for both balancings, the "Enrich+Bal Time" column shows the sums of enrichment and balancing times, and the last column contains sums of enrichment, balancing, and solution times. The last row of the table gives an average across the ten meshes generated adaptively. There is approximately a 35 percent improvement when comparing sums of balancing and enrichment times, and a 20 percent improvement in the overall times when using predictive enrichment. In this transient example, the solution and combined balancing-enrichment times were comparable; when solution time dominates adaptivity, *e.g.* , in a steady-state problem, gains due to predictive balancing are smaller.

3.5 Partition Smoothing

Application of the PSIRB or OCTPART partitioning methods to distribute mesh data yields reasonable surface indices [8,17]. However, when a general mesh enrichment procedure is used on an unstructured mesh, the resulting finite elements are not usually aligned with partition boundaries. When elements are assigned to a processor based on their centroid locations, a choice of partition boundaries based on octants, cut planes, or slices (Figure 6) may yield partitions with jagged edges. Such jagged edges increase communication costs; however, this may be reduced by smoothing the partition bound-

Mesh Num.	Size		Enrich+Bal Time			Total Time		
	Non (Tets)	Pred (Tets)	Non (s)	Pred (s)	% Diff	Non (s)	Pred (s)	% Diff
10	160303	160303	65.41	51.19	21.74	170.57	145.58	14.65
11	159820	159820	74.76	51.14	31.59	169.70	137.61	18.91
12	160712	160710	72.17	45.47	37.00	181.98	146.25	19.63
13	160496	160521	74.97	45.25	39.64	185.83	143.97	22.54
14	160665	160687	76.34	45.08	40.95	165.68	126.00	23.95
15	161412	161441	86.95	52.70	39.39	197.71	151.36	23.44
16	160871	160889	93.53	58.36	37.61	208.31	160.32	23.04
17	162107	162087	89.43	57.20	36.04	187.51	142.42	24.05
18	163337	163431	88.50	55.43	37.37	193.20	146.89	23.97
19	164894	165367	94.15	60.52	35.72	209.36	166.27	20.58
(av)	161462	161526	81.62	52.23	36.01	186.98	146.67	21.56

Table 1. Timings of nonpredictive vs. predictive enrichment procedures.

aries [19,21,28]. To do this, we let each processor traverse its boundary looking for elements that satisfy the following criteria:

(i) *Four faces adjacent to four other processors.* This is an isolated element that is migrated to any processor sharing a face. The donating processor's boundary is reduced by four faces, and the receiving processor has a net gain of three faces. This case may occur where several processor domains meet.

(ii) *Four faces adjacent to one other processor.* Typically, this case occurs when the element's centroid lies on the local processor, but its faces touch only elements on adjacent processors. The element is migrated to the other processor to eliminate four faces from the donating processor and four faces from the receiving processor's boundary.

(iii) *Three faces adjacent to one other processor.* The element forms a spike into a pocket on the other processor. The element is migrated to the other processor, reducing the boundary of each processor by two faces.

(iv) *Two faces adjacent to one other processor for each of a pair of elements with a common face.* The pair forms a spike into a pocket on the other processor. The pair is migrated to the other processor, reducing both the donating and receiving processors' boundaries by two faces.

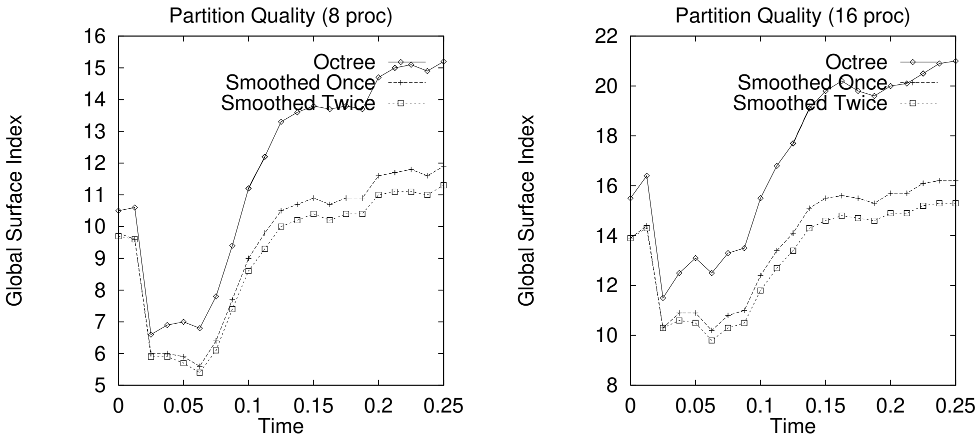
(v) *Three faces adjacent to two other processors.* The element is migrated to the processor sharing the highest number of faces. The donating processor's boundary is reduced by two faces, and the receiving processor's boundary size is unchanged.

Pattern detection and migration for each of these cases must be performed in a separate phase to avoid conflicting migrations which would degrade boundary smoothness. Furthermore, within a given case, operations between two or more processors must be colored to avoid simultaneous exchanges resulting in diminished gain or even loss. For example, spikes on one side of a boundary may exchange sides with spikes on the other side, resulting in a larger boundary than before the exchange. The coloring may be done using subphases where a processor first sends elements to higher-numbered processors and then sends them to lower-numbered ones. When three processors are involved, three subphases are necessary based on their relative order.

Minyard *et al.* [28] perform processor boundary smoothing by a similar iterative method. They identify elements on interprocessor boundaries whose nodes are all shared by two processors. These correspond to Cases (*ii*), (*iii*), and (*v*). Patterns involving more than two processors (Case (*i*)) are not considered. After all elements are marked, half of the marked elements along a boundary are migrated to one side, and half to the other. While this strategy will maintain a better load balance, it misses some opportunities to reduce interprocessor communication. For example, if case (*iv*) were encountered in the mesh, this strategy could result in no net improvement of the interprocessor boundaries.

Pattern recognition for each phase of our smoothing algorithm requires time proportional to the number of elements on a processor's boundary. Communication is also bounded by the number of elements on the processor's boundary; however, in practice, it takes much less time since typically less than 1% of the boundary qualifies for smoothing migration. In return, a significant drop in the number of boundary faces may be achieved.

Performance data for this smoothing algorithm is shown in Figure 10. The data were collected from a shock tube problem solved in three dimensions on 8 and 16 processors using OCTPART for load balancing. At the start of both runs, the GSI is high because distribution of the small initial mesh results in each processor having few elements. After time 0.03, however, adaptive mesh refinement has increased the mesh size substantially and the GSI improves. Over the remainder of the run, one iteration of smoothing reduced the GSI by 1-3 points on 8 processors and 2-4 points on 16 processors. In both cases repeating the smoothing operation yielded an additional improvement of up to 1 point. Total relative improvement after two smoothing iterations on both the 8 and 16 processor cases was 14-30%. More than two smoothing iterations did not provide a significant improvement. In general, partition qualities are better for the 8 processor case than the 16 because having twice the number of elements per processor simplifies the partitioning problem. Processor imbalance immediately after partitioning is 0.5% and is increased by a maximum of 2% after one iteration of smoothing and a maximum of 3.5% after two iterations.



(a) (b)

Fig. 10. Mesh improvements for boundary smoothing.

4 Applications

We solve compressible steady (Example 1, Section 4.2) and transient (Examples 2 and 3, Sections 4.3 and 4.4) flow problems to demonstrate the capabilities of the parallel adaptive system. Although these are demanding problems, applications with more complex geometries and loading can be addressed with the same technique.

4.1 Parallel Euler Solver

Three-dimensional solutions of the Euler equations were obtained using a spatially discontinuous Galerkin finite element method [6,12,13] and explicit time integration. Steady problems are solved by local time stepping with all elements advancing at their maximum stable timestep as determined by the Courant condition. Solution residuals are monitored to sense an approach to steady state. Transient problems are solved with a global step obtained as the maximum acceptable timestep over all elements according to the Courant condition.

Explicit, *reflected*, and *far field* conditions can be prescribed for all mesh faces classified as faces of the problem domain. For boundaries where density, pressure, and velocity are specified, a virtual element across the face supplies the specified data. Reflected conditions are imposed by hypothesizing a virtual element across the boundary face with a mirror image of the element's data. At far field boundaries, the virtual element has a copy of the element's data.

Error control is accomplished through backtracking. Time steps are either accepted or rejected based on whether or not elemental error indicators exceed a prescribed tolerance. Rejected time steps are repeated subsequent to adaptive space-time h-refinement and rebalancing. Coarsening is essential to keep mesh sizes manageable as fine-scaled structures move through the domain. Upon h-refinement, the solution is interpolated to the new mesh, and a new time step is attempted.

Error indicators based on jumps or gradients of the density, energy, pressure, or Mach number across a face control adaptive h-refinement. These face-based indicators may be used directly or scaled by either face area or inter-element distance. If desired, they may be combined to form element-based indicators. Experience suggests that a density gradient scaled by element volume is most informative, and this indicator has been used for the problems presented here. However, true error estimates [5,11,16] must be developed for compressible flow applications.

Time steps are rejected whenever error indicators exceed a rejection threshold. This threshold should be selected so that accepted steps provide an adequate solution resolution. Refinement and coarsening thresholds, respectively, are the error indicator values above and below which an element will request to be refined or coarsened. Coarsening thresholds for h-adaptivity should be set such that elements whose error indicator values are well below the rejection threshold are marked for coarsening. The refinement threshold should also be set below the rejection threshold to allow refinement of elements which are near the rejection threshold, thereby increasing the likelihood that a large number of time steps will be accepted before the next rejection.

Without an error estimate, the threshold selection process cannot be fully automatic and problem independent. An error histogram can aid in the selection of refinement and coarsening thresholds. Using the histogram, the system can monitor the percentages of elements whose error values fall into prescribed ranges and which are marked for refinement or coarsening. This information is used to select appropriate thresholds. In addition, to avoid overflowing available memory, the refinement threshold may be automatically adjusted based on an estimate of the number of elements that would be created during refinement.

4.2 Example 1: Steady Conical Flow

Consider the steady flow at Mach 5 past a cone having a half-angle of 10 degrees. This problem has a known analytical solution [27] that may be used to appraise accuracy. Using symmetry, we solve this problem in a box surrounding one quarter of the cone. The initial mesh contains 41,842 tetrahedral elements. The entire domain is initialized to a Mach 5 parallel flow toward the cone base. Reflected boundary conditions are applied on symmetry planes and on the cone's surface. A Mach 5 flow is prescribed at the inlet, and far field supersonic conditions are applied at the outlet.

This problem was run on 16 processors of an IBM SP2 computer. The initial mesh was partitioned using IRB, and the partitioning was rebalanced with PSIRB after each of three adaptive steps taken to reach steady state.

This example was used to compare the performance of the OCTPART and PSIRB partitioning procedures. Tests were done on 8 processors of an IBM SP2 computer. Surface indices are averaged over 6 to 8 runs. Figure 11 shows that PSIRB produces partitions which are superior as measured by MLSI and GSI. However, times to partition and migrate mesh data to achieve load balance for PSIRB are typically twice that of OCTPART. For problems in which solution time dominates rebalancing time, the additional cost of PSIRB may be worthwhile since the improved mesh quality can reduce total solution time (Section 5).

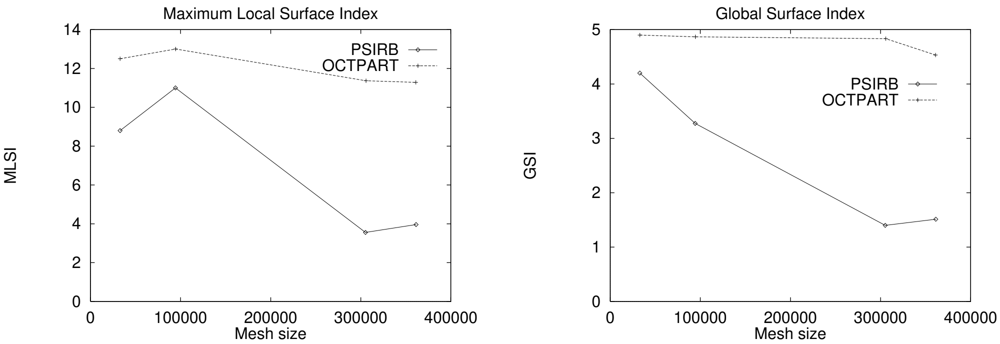


Fig. 11. Maximum Local (left) and Global (right) Surface Indices for 8-processor steady conical flow runs using the PSIRB and OCTPART partitioning procedures.

4.3 Example 2: Transient Shock Impacting a Cone

Consider a Mach 2 shock impacting a cone with a 10-degree half angle from its side. The domain is a box surrounding the cone with one end at the base of the cone and the opposite one beyond the tip of the cone. Using symmetry, we solve for the flow about one half of the cone with an initial mesh containing

28,437 elements. The domain is quiescent initially, and a Mach 1.25 flow behind the Mach 2 shock enters the top of the flow domain. Reflected conditions are applied on solid surfaces, planes of symmetry, and on the sides of the problem domain. Far field conditions apply at the bottom face.

This transient flow problem was solved on 16 processors of an IBM SP2 computer. Figure 13 shows the density at global time $t = 0.3$. Away from the cone, the shock travels without appreciable disturbance, but the density increases significantly as the shock diffracts in regions near the cone. The partitioning selected by PSIRB at $t = 0.3$ is shown in Figure 14.

4.4 Example 3: Transient Flow in a Muzzle Brake

Consider the three-dimensional unsteady compressible flow in a cylinder containing a cylindrical vent. This problem was motivated by flow studies in perforated muzzle brakes for large calibre guns [18]. We match flow conditions to those of shock tube studies of Dillon [18] and Nagamatsu *et al.* [29]. Our focus is on the quasisteady flow that exists behind the contact surface for a short time. Using symmetry, the flow may be solved in one half of the domain bounded by a plane through the vent. The initial mesh (Figure 12) contains 45,093 tetrahedral elements. The mesh contains 80,659 elements after a prerefinement stage which forces refinement near the interface between the shock tube and vent. The larger cylinder (the shock tube) initially contains helium gas moving at Mach 1.23 while the smaller cylinder (the vent) is quiet. A Mach 1.23 flow is prescribed at the tube's inlet and outlet. The walls of the cylinders are given reflected boundary conditions, and a far field condition is applied at the vent exit. Flow begins as if a diaphragm between the two cylinders were ruptured.

This problem was run on 16 processors of an IBM SP2 computer. The initial mesh was partitioned with IRB, and the partitioning was rebalanced with PSIRB after each adaptive step. Partition quality remains good using PSIRB. As an example, the partitioning of a 485,345 element mesh after 141 adaptive refinement steps has a GSI of 2.48% and a MLSI of 7.63%. Figure 16 shows the Mach number with velocity vectors in the vent region. Flow features compare favorably with experimental and numerical results of Nagamatsu *et al.* [29].

The flow accelerates as it enters the vent. A strong shock forms near the downwind vent-shock tube interface. A portion of the flow in the vent accelerates to supersonic conditions. The reflection of the flow from the downwind vent face produces a component of the flow at the vent exit in a direction opposite to the principal flow direction. In a cannon, this helps to reduce recoil.

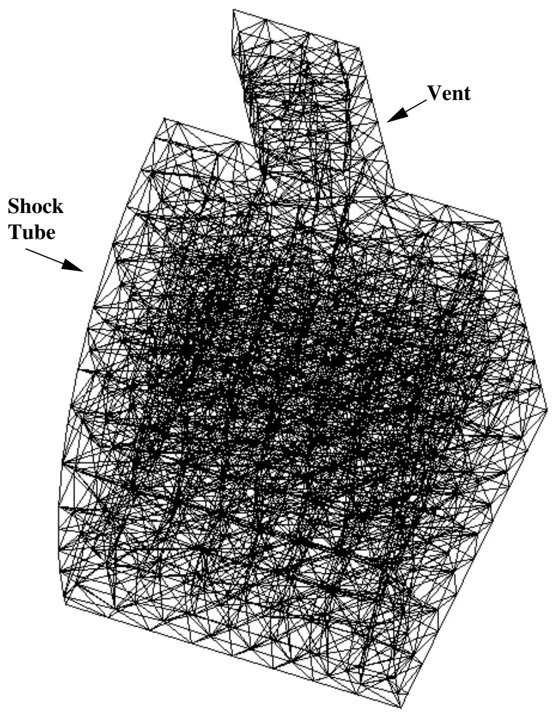


Fig. 12. Muzzle brake (Example 3) initial mesh.

5 Discussion

Many of the methods and software libraries used to solve these problems can be applied in other areas. Those areas under investigation include materials processing, crystal growth, and biomechanics.

In this paper, we have focused on describing and comparing several load balancing schemes. Comparisons by timing are difficult, since times vary between runs having the same parameters. The high-speed switch of the IBM SP2 computer is a shared resource that affects run times. More subtle effects can result from differences in the order in which messages used for migration are processed. Changes in the order in which those messages are received and integrated into the local MDB result in different traversal orders of the mesh entities. These differences cause small changes in load balancings and coarsenings. While such differences in meshes and partitionings do not affect the solution accuracy, they can cause sufficient changes in efficiency to make precise timings difficult. Qualitatively, PSIRB produced the best partitions (measured as a function of total analysis time). Octree-generated partitions were comparable but resulted in slightly longer solution times. In both cases, one or two iterations of partition boundary smoothing led to a quality improvement. ITB by itself resulted in poorer partition quality, but is useful when mesh changes are small between computational stages. Predictive enrichment provided superior performance to our current enrichment process with transient problems where there are frequent enrichment and balancing steps.

Enhancements to the existing load balancing procedures and the implementation of new ones are under investigation. Improvements in the slice-by-slice technique used by ITB for migration are necessary. Experiments with geometrical methods that use the spatial location of elements relative to the centroids of sending and receiving processors showed promise at reducing the number of processor interconnections. Vidwans *et al.* [39] presented divide-and-conquer load balancing methods that take advantage of the geometric information in a similar framework. Using inertial techniques in conjunction with the iterative methods should give results similar to geometrical methods while potentially costing less. Balancing methods must optimize the total of partitioning, redistribution, and computational costs. However, realizing the difficulty of this task, methods that select elements to maintain "compactness" of partitions, those that move elements to improve interprocessor adjacency, and those that control the volume of data migrated [25,37,38] are being considered. We are also seeking more efficient and effective load balancing techniques for use with time-dependent problems and adaptive p-refinement. The weightings described here will be useful with p-refinement.

We are implementing a general-purpose, object-oriented, parallel adaptive framework. Parallel structures will be implemented at the lowest level of the framework to allow some operations to be done in a more natural and efficient way than with PMDB, which resides on top of the sequential MDB. The additional information which will be available in the object-oriented framework will allow more sophisticated load balancing procedures to be implemented.

A recent improvement to our flow solver adds spatially-dependent local time stepping. Elements that take larger steps wait for elements using smaller steps to catch up. Preliminary testing indicates a significant reduction in solution times. This extends local temporal refinement from clusters of uniform meshes [4,16] to unstructured meshes. We are extending load balancing to account for this local time stepping.

Acknowledgement

We would like to thank Mark Beall, Carlo Bottasso, Hugues de Cougny, Hema Murty, and Wesley Turner for the generous use of their software and many valuable suggestions. Computer systems used in the development and analysis runs include the 36-node IBM SP2 computer at Rensselaer, the 400-node SP2 at the Maui High Performance Computing Center, and the 512-node SP2 at the Cornell Theory Center. Authors were supported by AFOSR Grant F49620-

95-1-0407, ARO grant DAAH04-95-1-0091, and NSF Grant CCR-9527151.

References

[1] S. T. Barnard, PMRSB: parallel multilevel recursive spectral bisection, in: *Proc. Supercomputing 95*, San Diego (1995).

[2] S. T. Barnard and H. D. Simon, Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems, *Concurrency: Practice and Experience*, **6** (1994) 101–117.

[3] M. W. Beall and M. S. Shephard, A general topology-based mesh data structure, to appear *Int. J. Numer. Meth. Engng.* (1997).

[4] M. J. Berger and S. H. Bokhari, A partitioning strategy for nonuniform problems on multiprocessors, *IEEE Trans. Computers*, **36** (1987) 570–580.

[5] K. S. Bey, A. Patra, and J. T. Oden, *hp*-version discontinuous Galerkin methods for hyperbolic conservation laws: a parallel adaptive strategy, *Int. J. Numer. Meth. Engng.*, **38** (1995) 3889–3907.

[6] R. Biswas, K. D. Devine, and J. E. Flaherty, Parallel, adaptive finite element methods for conservation laws, *Appl. Numer. Math.*, **14** (1994) 255–283.

[7] C. L. Bottasso, H. L. de Cougny, M. Dindar, J. E. Flaherty, C. Özturan, Z. Rusak, and M. S. Shephard, Compressible aerodynamics using a parallel adaptive time-discontinuous Galerkin least-squares finite element method, in: *Proc. 12th AIAA Applied Aerodynamics Conference*, Colorado Springs, AIAA-94-1888 (1994).

[8] C. L. Bottasso, J. E. Flaherty, C. Özturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, The quality of partitions produced by an iterative load balancer, in: *Proc. Third Workshop on Languages, Compilers, and Runtime Systems*, Troy (1996) 265–277.

[9] D. Callahan and K. Kennedy, Compiling programs for distributed-memory multiprocessors, *J. Supercomputing*, **2** (1988) 151–169.

[10] K. Clark, J. E. Flaherty, and M. S. Shephard, *Appl. Numer. Math., special ed. on Adaptive Methods for Partial Differential Equations*, **14** (1994).

[11] B. Cockburn and P.-A. Gremaud, Error estimates for finite element methods for scalar conservation laws, *SIAM J. Numer. Anal*, **33** (1996) 522–554.

[12] B. Cockburn, S.-Y. Lin, and C.-W. Shu, TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws III: One-Dimensional systems, *J. Comput. Phys.*, **84** (1989) 90–113.

[13] B. Cockburn and C.-W. Shu, TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws II: General framework, *Math. Comp.*, **52** (1989) 411–435.

[14] G. Cybenko, Dynamic load balancing for distributed memory multiprocessors, *J. Par. Dist. Comput.*, **7** (1989) 279–301.

[15] H. L. de Cougny, K. D. Devine, J. E. Flaherty, R. M. Loy, C. Özturan, and M. S. Shephard, Load balancing for the parallel adaptive solution of partial differential equations, *Appl. Numer. Math.*, **16** (1994) 157–182.

[16] K. D. Devine and J. E. Flaherty. Parallel adaptive *hp*-refinement techniques for conservation laws. *Appl. Numer. Math.*, **20** (1996) 367–386.

[17] K. D. Devine, J. E. Flaherty, R. Loy, and S. Wheat, Parallel partitioning strategies for the adaptive solution of conservation laws, in: I. Babuška, J. E. Flaherty, W. D. Henshaw, J. E. Hopcroft, J. E. Oliger, and T. Tezduyar, eds., *Modeling, Mesh Generation, and Adaptive Numerical Methods for Partial Differential Equations*, No. 75, (Springer-Verlag, Berlin-Heidelberg, 1995) 215–242.

[18] R. E. Dillon Jr., A parametric study of perforated muzzle brakes, ARDC Technical Report ARLCB-TR-84015, Benet Weapons Laboratory, Watervliet, 1984.

[19] P. Diniz, S. Plimpton, B. Hendrickson, and R. Leland, Parallel algorithms for dynamically partitioning unstructured grids, in: *Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco (1995) 615–620.

[20] C. Farhat and M. Lesoinne, Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics, *Int. J. Numer. Meth. Engng.*, **36** (1993) 745–764.

[21] C. Farhat, N. Maman, and G. W. Brown, Mesh partitioning for implicit computations via iterative domain decomposition: impact and optimization of the subdomain aspect ratio, *Int. J. Numer. Meth. Engng.*, **38** (1995) 989–1000.

[22] MPI Forum, *MPI: A Message Passing Interface Standard*, University of Tennessee, Knoxville, first edition, 1994.

[23] B. Hendrickson and R. Leland, The Chaco user's guide, version 1.0, Technical Report SAND93-2339, Sandia National Laboratories, Albuquerque, 1993.

[24] B. Hendrickson and R. Leland, Multidimensional spectral load balancing, Technical Report SAND93-0074, Sandia National Laboratories, Albuquerque, 1993.

[25] Y. F. Hu and R. J. Blake, An optimal dynamic load balancing algorithm, Preprint DL-P-95-011, Daresbury Laboratory, Warrington, 1995.

[26] E. Leiss and H. Reddy, Distributed load balancing: design and performance analysis, *W. M. Kuck Research Computation Laboratory*, **5** (1989) 205–270.

[27] J. Maccoll, The conical shock wave formed by a cone moving at a high speed, *Proc. Royal Society of London*, Series A, **CLIX** (1937) 459–472.

[28] T. Minyard, Y. Kallinderis, and K. Schulz, Parallel load balancing for dynamic execution environments, in: *Proc. 34th Aerospace Sciences Meeting and Exhibit*, Reno, AIAA-96-0295 (1996).

[29] H. T. Nagamatsu, K. Y. Choi, R. E. Duffy, and G. C. Carofano, An experimental and numerical study of the flow through a vent hole in a perforated muzzle brake, ARDEC Technical Report ARCCB-TR-87016, Benet Weapons Laboratory, Watervliet, 1987.

[30] C. Özturan, *Distributed Environment and Load Balancing for Adaptive Unstructured Meshes*, PhD thesis, Computer Science Dept., Rensselaer Polytechnic Institute, Troy, 1995.

[31] A. Patra and J. T. Oden, Problem decomposition for adaptive *hp* finite element methods, *Comp. Sys. Engng.*, **6** (1995) 97.

[32] A. Pothen, H. Simon, and K.-P. Liou, Partitioning sparse matrices with eigenvectors of graphs, *SIAM J. Mat. Anal. Appl.*, **11** (1990) 430–452.

[33] M. S. Shephard, S. Dey, and J. E. Flaherty, A straight forward structure to construct shape functions for variable p-order meshes, SCOREC Report # 6-1996, Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, 1996. To appear *Comp. Meth. in Appl. Mech. and Engng.*.

[34] M. S. Shephard, J. E. Flaherty, H. L. de Cougny, C. Özturan, C. L. Bottasso, and M. W. Beall, Parallel automated adaptive procedures for unstructured meshes, in: *Parallel Computing in CFD*, No. R-807 (Agard, Neuilly-Sur-Seine, 1995) 6.1–6.49.

[35] M. S. Shephard and M. K. Georges, Automatic three-dimensional mesh generation by the Finite Octree technique, *Int. J. Numer. Meth. Engng.*, **32** (1991) 709–749.

[36] H. D. Simon, Partitioning of unstructured problems for parallel processing, *Comp. Sys. Engng.*, **2** (1991) 135–148.

[37] A. Sohn, R. Biswas, and H. D. Simon, Impact of load balancing on unstructured adaptive computations for distributed-memory multiprocessors, in: *Proc. Eighth IEEE Symposium on Parallel and Distributed Processing*, New Orleans (1996) 26–33.

[38] R. Van Driessche and D. Roose, An improved spectral bisection algorithm and its application to dynamic load balancing, *Parallel Computing*, **21** (1995) 29–48.

[39] V. Vidwans, Y. Kallinderis, and V. Venkatakrishnan, Parallel dynamic load-balancing algorithm for three-dimensional adaptive unstructured grids, *AIAA J.*, **32** (1994) 497–505.

[40] C. H. Walshaw and M. Berzins, Dynamic load balancing for PDE solvers on adaptive unstructured meshes, *Concurrency: Practice and Experience*, **7** (1995) 17–28.

[41] S. R. Wheat, K. D. Devine, and A. B. MacCabe, Experience with automatic, dynamic load balancing and adaptive finite element computation, in: *Proc. 27th Hawaii International Conference on System Sciences*, Kihei (1994) 463–472.

[42] R. D. Williams, Voxel databases: A paradigm for parallelism with spatial structure, *Concurrency: Practice and Experience*, **4** (1992) 619–636.

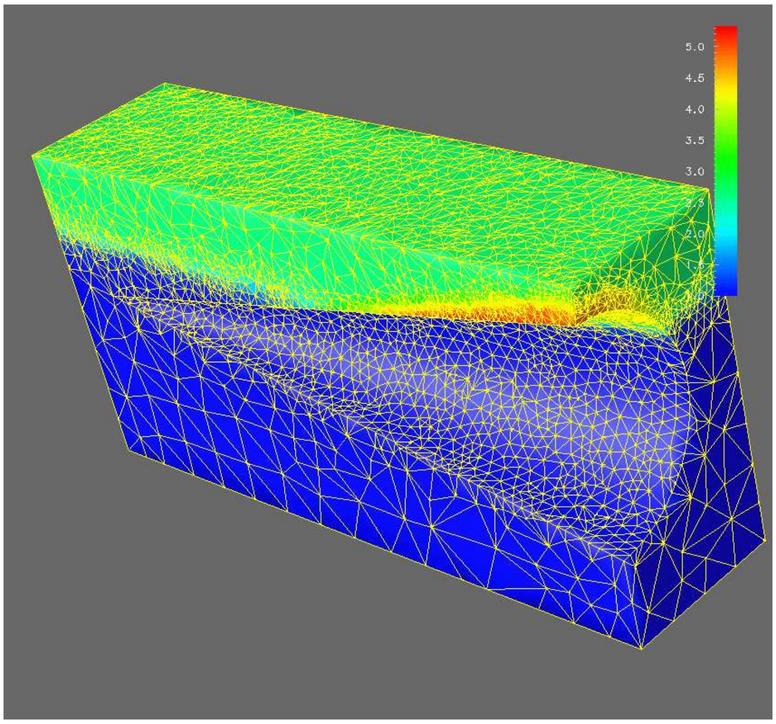


Fig. 13. Density values for Example 2 at time $t = 0.3$. Shading indicates values of the density on each element.

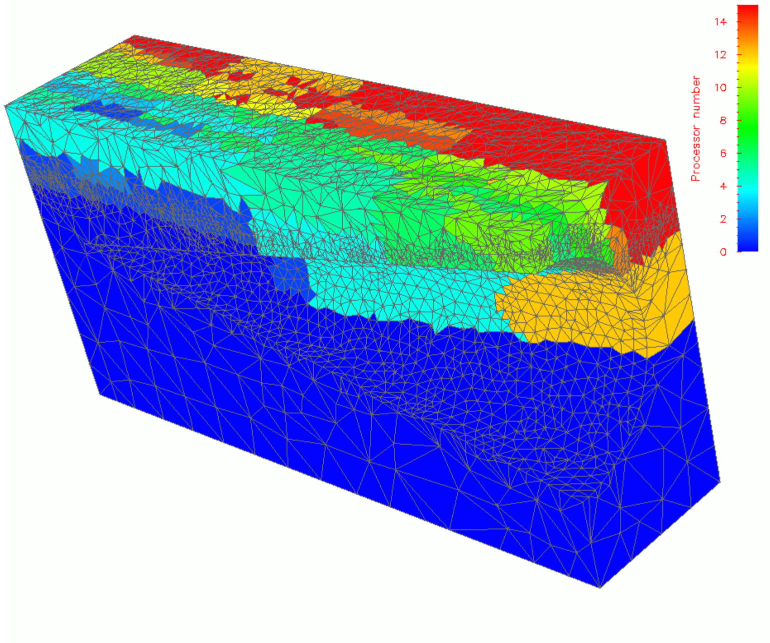


Fig. 14. Partitioning of the mesh for Example 2 using PSIRB at time $t = 0.3$. Shading indicates processor assignments.

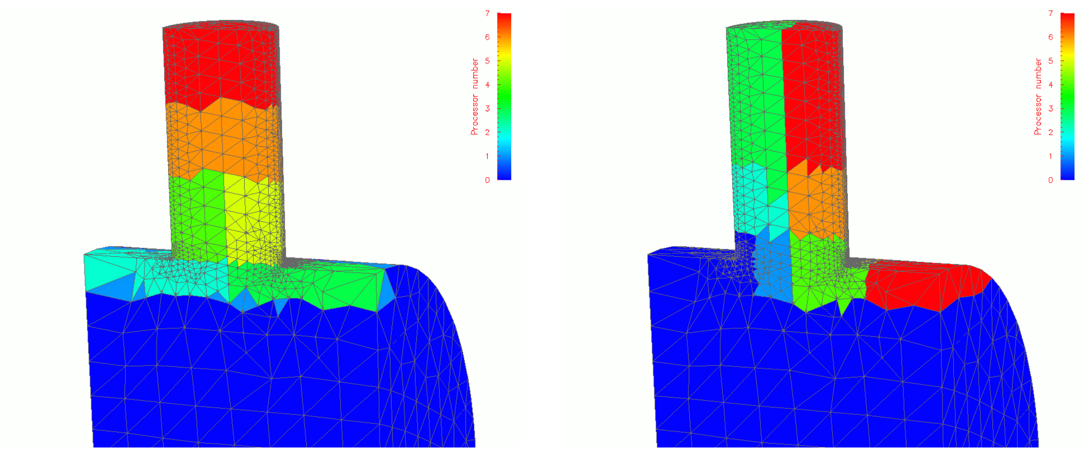


Fig. 15. Initial meshes for Example 3 distributed onto 8 processors by PSIRB (left) and OCTPART (right). Shading indicates processor assignments.

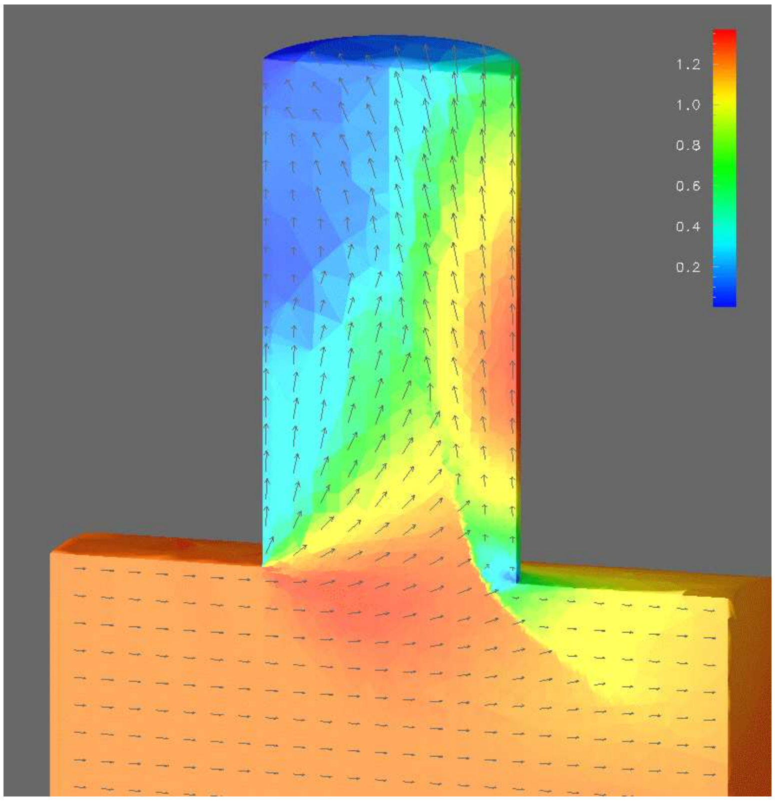


Fig. 16. Mach number with velocity vectors on the symmetry plane for Example 3.