Distributed Octree Data Structures and Local Refinement Method for the Parallel Solution of Three-Dimensional Conservation Laws

J. E. Flaherty, R. M. Loy, M. S. Shephard,
M. L. Simone, B. K. Szymanski,
J. D. Teresco, and L. H. Ziantz
Scientific Computation Research Center
Rensselaer Polytechnic Institute
Troy, New York 12180-3590, USA

Abstract

Conservation laws are solved by a local Galerkin finite element procedure with adaptive space-time mesh refinement and explicit time integration. A distributed octree structure representing a spatial decomposition of the domain is used for mesh generation, and later may be used to correct for processor load imbalances introduced at adaptive enrichment steps. A Courant stability condition is used to select smaller time steps on smaller elements of the mesh, thereby greatly increasing efficiency relative to methods having a single global time step. To accommodate the variable time steps, octree partitioning is extended to use weights derived from element size. Computational results are presented for the three-dimensional Euler equations of compressible flow solved on an IBM SP2 computer. The problem examined is the flow inside a perforated shock tube.

1 Introduction

Adaptive finite element methods that automatically refine or coarsen meshes (h-refinement) and/or vary the order of accuracy of a method (p-refinement) offer greater reliability, robustness, and efficiency than traditional numerical approaches for solving partial differential equations. Like adaptivity, parallel computation makes it possible to solve previously intractable problems. With problems continuing to increase in complexity through the inclusion of more realistic effects in models, it seems advantageous to unite adaptivity and parallelism to achieve the highest gains in efficiency. Adaptivity on parallel computers, however, introduces complications that do not arise with simpler solution strategies. Adaptive algorithms that utilize unstructured meshes [1, 26, 43, 44] make the task of balancing processor computational load more difficult than with uniform structures. Furthermore, a balanced loading will become unbalanced as degrees of freedom are introduced or removed by adaptive h- and/or p-refinement.

Quadtree and octree decomposition is a successful strategy to localize data in mesh generation procedures on serial computers [2, 28, 37, 42, 46, 55]. Three-dimensional parallel mesh generation also makes use of octree procedures. With duplication of the tree on every processor [15, 16] the process does not scale. Scalable parallel mesh generation requires a distributed octree structure [48]. In a distributed environment, the octree also serves as a means to partition the domain to be discretized. A *parallel octree library* supports the creation and distribution of octree structures (Section 4).

We use a hierarchical representation of finite element meshes that is appropriate for h- or p-refinement [3]. A *Parallel Mesh Database* [22, 45] provides operators to create and manipulate distributed mesh data.

Poor partitioning of data across the processors of a parallel computer leads to high communication costs. Several static partitioning algorithms have been developed [5, 20, 38, 27]; however, these may be inefficient in an adaptive computational environment. *Parallel Sort Inertial Recursive Bisection* (PSIRB) [45] performs recursive bisections of domains in directions normal to their principal axes of inertia. A parallel sort enables its parallel execution; however, it is still costly relative to solution time. This has led to the use of iterative dynamic load balancing techniques that incrementally migrate data from heavily to lightly loaded processors [7, 9, 13, 14, 18, 30, 45, 52, 54]. These methods provide inexpensive balancings, but may not reduce communication costs. Other techniques [36, 41, 53] specifically concentrate on balancing adaptive computations.

Devine and Flaherty [18] employ a quadtree to support a parallel hp-adaptive analysis in two dimensions. We describe a dynamic partitioning technique that exploits the properties of three-dimensional octree-structured meshes. Since such trees are easily constructed from arbitrary meshes, the procedure is independent of octree mesh generation. Partitioning may be done serially to obtain an initial partitioning or in parallel for repartitioning a distributed mesh (Section 4.2) [18]. In either case, it is inexpensive; hence, it may be used with adaptivity.

Adaptive h-refinement introduces variation in element size in order to concentrate computational effort in specific parts of the domain. However, the maximum globally stable time step depends on the size of the smallest element of the mesh. Therefore, an unintended side effect of h-refinement is a reduction of computational efficiency on larger elements. In order to increase efficiency, temporal adaptivity has been applied to overlapping two-dimensional uniform [4, 6, 8, 14] and unstructured [29] meshes. In Section 3.2, we describe an explicit Local Refinement Method (LRM) for the solution of time-dependent conservation laws on three-dimensional unstructured meshes. It permits time steps on elements to be proportional to their size. Larger elements take larger time steps, so work is concentrated on the smaller ones. Although this method complicates load balancing [24], it leads to a large improvement in overall efficiency. Weighting factors proportional to element size may be employed with octree partitioning or other procedures to help balance loading [21, 24].

Using an IBM SP2 computer, we apply the LRM and the parallel octree-based partitioning technique to three-dimensional compressible flow problems involving the Euler equations. Results are presented in Section 5 and discussed in Section 6.

2 The Discontinuous Galerkin Method

We consider three-dimensional conservation laws of the form

$$\mathbf{u}_t(\mathbf{x},t) + \sum_{i=1}^{3} \mathbf{f}_i(\mathbf{x},t,\mathbf{u})_{x_i} = 0, \quad \mathbf{x} \in \Omega, \quad t > 0, \tag{1a}$$

with initial conditions

$$\mathbf{u}(\mathbf{x},0) = \mathbf{u}^0(\mathbf{x}), \quad \mathbf{x} \in \Omega \cup \partial\Omega, \tag{1b}$$

and appropriate well-posed boundary conditions. For the Euler equations, the vector $\mathbf{u}$ specifies the fluid's density, momentum components, and energy. The subscripts t and x_i, $i = 1, 2, 3$, denote partial differentiation with respect to time and the spatial coordinates. Finite difference schemes for (1), such as the Total Variation Diminishing (TVD) [49, 50] and Essentially Non-Oscillatory (ENO) [47] methods, achieve high-order accuracy by using a computational stencil that enlarges with order. A wide stencil makes the methods difficult to implement on unstructured meshes and limits efficient implementation on parallel computers. Finite element methods, however, have stencils that are invariant with method order, allowing them to model problems with complicated geometries more easily and to be efficiently parallelized.

We discretize (1) using a discontinuous Galerkin finite element method [8, 11, 12]. Thus, we partition the domain Ω into tetrahedral elements Ω_j, $j = 1, 2, \ldots, J$, multiply (1a) by a test function $\mathbf{v} \in L^2(\Omega_j)$, integrate the result on Ω_j, and use the Divergence Theorem to obtain

$$\int_{\Omega_j} \mathbf{v}^{\mathrm{T}} \mathbf{u}_t \, d\tau - \sum_{i=1}^{3} \int_{\Omega_j} \mathbf{v}_{x_i}^{\mathrm{T}} \mathbf{f}_i(\mathbf{u}) \, d\tau + \sum_{i=1}^{3} \int_{\partial\Omega_j} \mathbf{v}^{\mathrm{T}} \mathbf{f}_i(\mathbf{u}) n_i \, d\sigma = 0, \quad t > 0, \tag{2}$$

where $\mathbf{n} = [n_1, n_2, n_3]^{\mathrm{T}}$ is the unit outward normal to $\partial\Omega_j$. Approximating $\mathbf{u}(\mathbf{x}, t)$ on Ω_j by a p^{th}-degree polynomial $\mathbf{U}_j(\mathbf{x}, t) \in \mathbf{S}_j \subset L^2(\Omega_j)$, and testing against all functions $\mathbf{V} \in \mathbf{S}_j$ yields the ordinary differential system

$$\begin{aligned}\int_{\Omega_j} \mathbf{V}^{\mathrm{T}} (\mathbf{U}_j)_t \, d\tau \quad &- \quad \sum_{i=1}^{3} \int_{\Omega_j} \mathbf{V}_{x_i}^{\mathrm{T}} \mathbf{f}_i(\mathbf{U}_j) \, d\tau \\ &+ \quad \sum_{i=1}^{3} \int_{\partial\Omega_j} \mathbf{V}^{\mathrm{T}} \mathbf{f}_i(\mathbf{U}_j) n_i \, d\sigma = 0, \quad t > 0, \quad j = 1, 2, \ldots, J. \end{aligned} \tag{3a}$$

Initial conditions are determined by local L^2 projection as

$$\int_{\Omega_j} \mathbf{V}^{\mathrm{T}} (\mathbf{U}^j - \mathbf{u}^0) \, d\tau = 0, \quad t = 0, \quad \forall \mathbf{V} \in \mathbf{S}_j, \quad j = 1, 2, \ldots, J. \tag{3b}$$

Results of Section 5 use piecewise constant (p=0) approximations and explicit Euler integration; however, p-refinement may be incorporated [17].

The normal component of the flux

$$\mathbf{f}_{\mathbf{n}}(\mathbf{u}) = \sum_{i=1}^{3} \mathbf{f}_i(\mathbf{u}) n_i \tag{4}$$

remains unspecified on $\partial\Omega_j$ since the approximate solution is discontinuous there. We specify it using a "numerical flux" function $\mathbf{h}(\mathbf{U}_j^+, \mathbf{U}_j^-)$ dependent on solution states $\mathbf{U}_j^+$ and $\mathbf{U}_j^-$ on the inside and outside, respectively, of $\partial\Omega_j$. Several numerical flux functions are possible [12, 47]; we use van Leer's flux vector splitting [18, 31, 51].

3 Adaptive Techniques

The software uses adaptive h-refinement in space and time to concentrate computational effort in areas of inadequate solution resolution. The computation consists of solution steps with periodic error checking, which leads to adaptive enrichment when necessary.

3.1 h-Refinement

Mesh refinement and coarsening utilize edge-based error indicators to determine where to perform enrichment [34, 45]. Coarsening is performed when a group of elements all have edges that are so marked. Convex polyhedra of such elements containing a central vertex are identified. The interior vertex is removed by collapsing along one of the connected edges, leaving a polyhedron that is discretized without the interior vertex to form fewer elements. Coarsening requires that the entire polyhedron of elements lie on the same processor, so element migration is required when elements that span an interprocessor boundary are involved in a coarsening operation.

Refinement may be isotropic or anisotropic depending on the number of edges of an element selected for refinement. Forty-two templates are employed to accomplish this efficiently. Interprocessor communication is required to update shared vertices, edges, and faces; however, element migration is not necessary.

To propagate the solution during enrichment, error indicators and solution values are assigned to mesh vertices as the volume-weighted average of the piecewise-constant solutions and error indicators of elements containing that vertex. Edges are marked for enrichment based on their vertex error values and user-supplied coarsening and refinement thresholds. During refinement, newly created vertices along bisected edges receive interpolated solution values from the original vertices. During coarsening some vertices may simply be removed, and edges rearranged. After the enrichment procedure, elements average their four vertex solutions to restore the original element-oriented solution. To reduce diffusion, this process is avoided, where possible, by allowing newly created elements to inherit solution values from the previous elements occupying their space.

3.2 The Local Refinement Method

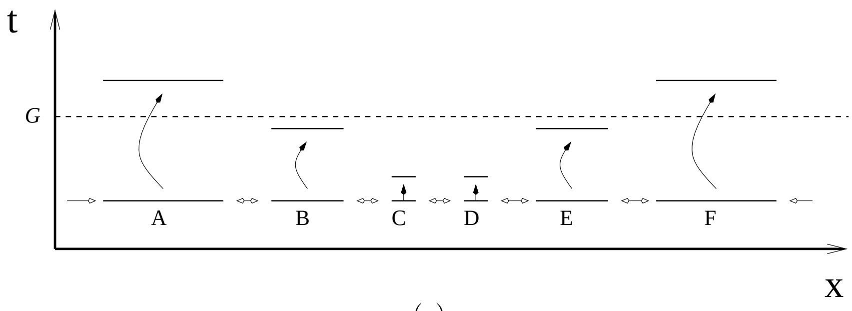
Our LRM selects spatially-dependent time steps based upon a Courant stability condition for explicit time integration. Thus, in a given time period, a smaller number of larger time steps will be taken on large elements, and the opposite will occur on small elements. We illustrate the procedure in Figure 1 for a group of adjacent (one-dimensional) elements (A-F). The solution is periodically synchronized to calculate error estimates or indicators. This "goal time" to which we wish to advance the solution is labeled G and is typically determined to be a small multiple of the smallest time step on any element of the mesh.

The time step for Ω_j is determined from a Courant condition as

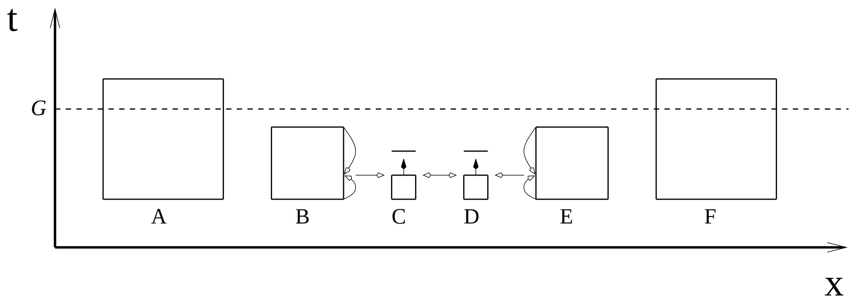
$$\Delta t_j = \alpha \frac{r_j}{v_j}, \quad \alpha \leq 1, \tag{5}$$

where r_j is the radius of Ω_j's inscribed sphere and v_j is the maximum signal speed on Ω_j. For the Euler equations, v_j is the sum of the fluid's speed and the sound speed. The parameter α is introduced to maintain stability in areas of mesh gradation. We empirically chose $\alpha = 0.65$, but a more thorough analysis is necessary.

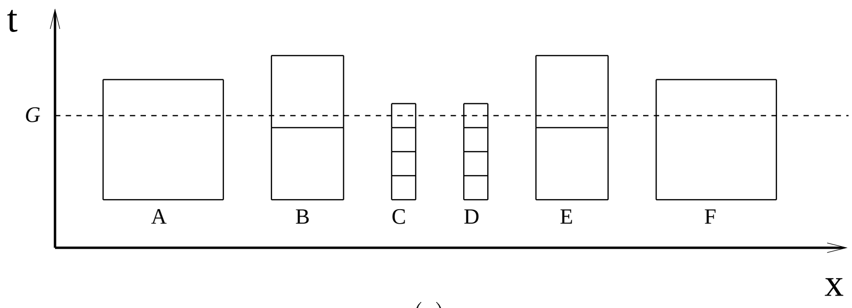
The six elements shown in Figure 1 have been advanced to time G using four rounds of time stepping. Only 14 element time steps are necessary as compared to the 24 steps that would have been required had all elements taken the largest globally stable time step of elements C and D. Greater improvements in efficiency can be seen for production runs (Section 5.2).



(a)



(b)



(c)

Figure 1: The Local Refinement Method. The set of one-dimensional elements A-F choose time steps according to their stability criteria. (a) The elements exchange information with their neighbors (or evaluate boundary conditions) and advance by a single time step. (b) Elements C and D receive interpolated data from B and E, respectively, and advance a second time step. (c) The process is repeated until all elements have reached the goal time G.

In principle, elements may take any stable time step; however, allowing arbitrary time steps is not efficient. Neighboring elements tend to be similar in size and, hence, use similar time steps. However, small differences in element sizes and shapes could lead to minor differences in time steps. This, in turn, leads to time stepping of isolated elements, causing additional flux evaluations and complex interpolations. This problem is easily solved by rounding time steps down to the next lower (fractional) power of two. Direct bitwise manipulation is used for efficiency. Thus, neighboring similar-sized elements advance together as a group. Fluxes computed on faces interior to the group are used twice, once for each element, halving the work relative to computation with isolated elements. Since flux calculations are typically the most expensive part of the integration, this savings outweighs any possible losses due to using reduced time steps. Choosing time steps that are fractional powers of two also helps to organize the computation [29].

3.3 Error Control

Error control is accomplished through backtracking. Time steps are 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. At $t = 0$, the initial conditions are used rather than solution interpolation to reduce diffusion.

Error indicators based on jumps or gradients of the density, energy, pressure, or Mach number across a face are used to control adaptive h-refinement for the Euler equations. These face-based indicators may be used directly or scaled by 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 was used for the problem presented in Section 5. However, discretization error estimates [7, 10, 17] must be developed for compressible flow applications.

The rejection threshold is selected so that accepted steps provide acceptable solution resolution. Refinement and coarsening thresholds, respectively, are the error indicator values above and below which an element will be scheduled for refinement or coarsening. The coarsening threshold should be set well below the rejection threshold. The refinement threshold should also be set below the rejection threshold to allow refinement of elements that have indicators near the rejection threshold, thereby decreasing the likelihood of subsequent rejected time steps.

Without an error estimate, threshold selection 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. Once refinement thresholds are set at the beginning of the simulation, refinement and coarsening are automatic.

4 Distributed Octree

Quadtrees and octrees have been successfully used as spatial data structures to localize data in mesh generation procedures on serial computers [2, 28, 37, 42, 46, 55]. The localization allows finite elements to be generated efficiently by focusing on specific neighborhoods rather than the entire domain. An octree-based mesh generator [46] recursively subdivides an embedding of the problem domain in a cubic universe into eight octants wherever more resolution is required. Octant subdivision is initially based on geometric features of the domain, but solution-based criteria are introduced during adaptive h-refinement. Finite element meshes of tetrahedral elements are generated from the octree by subdividing terminal octants. Octants containing too many or too few elements following mesh enrichment are subdivided or pruned to ensure that octants contain no more than a maximum or no fewer than a minimum allowable number of elements. For meshes generated by other procedures, or for meshes adapted independently of the octree, each element may be associated with the octant that contains its centroid. Scalable parallel mesh generation by the octree technique requires distribution of the tree structure across the network of processors [48]. A distributed octree is defined by octants with parent and child links; however, some links are off-processor. In the design of the parallel octree library, all parent and child queries return a pointer to a structure in the local processor's memory [48], which contains information about object locality. If it is local, it is processed in the normal fashion. If not, the processor number and remote address are available. By using this design instead of directly storing processor numbers and remote addresses for all links, storage for local links is the same as in the serial case. Remote links require one level of indirection and storage of the intermediate structure. Since most links will be local, there is an overall space savings [48].

An octant whose parent is off-processor is called a *local root*. A parent query still returns a pointer to a structure; however, it contains information about the parent's processor, address, bounding coordinates, and its level in the global octree. Storing this information locally enables complex queries on octants in the subtree to be performed via local tree traversals. For example, in the serial case, finding the bounding coordinates of an octant

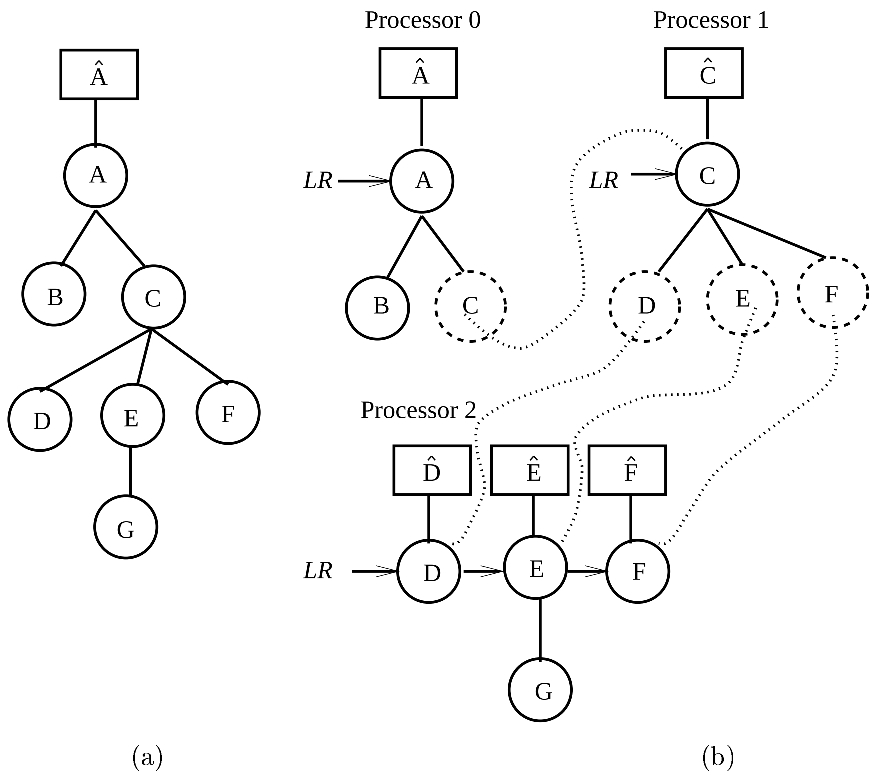


Figure 2: Parallel tree structures: (a) entire tree and (b) tree distributed across three processors.

usually requires a traversal to the root. The bounding coordinates of the root and the path from it to the octant uniquely determine the octant's coordinates. Inter-processor communication needed for the traversal to the root of a distributed tree is avoided by storing bounding information with each local root, thus, truncating the search on the local processor. Each processor maintains a list of local roots. All octants on a processor may be reached by traversing the subtrees rooted by the octants in the local root list.

A simple tree having root A appears in Figure 2a. Its data structure including bounding box information is stored in Â. In Figure 2b, the tree has been distributed across three processors. The dotted circles indicate remote references. Only the remote location is stored in these cases. All data associated with a node is stored on its assigned processor. Each processor has a local root list denoted by LR, and each local root has a data structure storing its bounding box and tree level information.

4.1 Octree Updating

After mesh enrichment, the octree and its element relationships must be updated. It might be most efficient to update the element-octant associations at the time of creation and deletion of elements; however, performing the operations in a post-processing stage does not introduce a large overhead and is more general since it allows mesh refinement to be independent of the octree.

Elements resulting from mesh coarsening must be assigned to an octant. When the convex polyhedron used to coarsen is completely internal to the local processor's spatial domain, octree insertion is straightforward. However, the situation is more complex when the coarsening procedure has to import elements from other processors to form a convex polyhedron on one processor. The polyhedron occupies space corresponding to terminal octants on at least one other processor; therefore, the coarsened mesh occupies this space as well. Some of the new elements do not belong to any octant on the local processor, and must be migrated to the processor containing their octant. Since some of these elements lie on the processor's spatial boundary, the destination processor may be determined by mesh connectivity.

Each element created in the mesh refinement stage lies within its parent, so it may simply inherit the octant association of the parent. If this information is not available from the refinement procedure, the element may be inserted into the octree in time proportional to the depth of the local octree, which is $O(\log n)$, where n is the number of elements on the processor.

After mesh enrichment, a traversal of the octree determines if the octree needs to be extended or pruned. Crowded leaf octants are subdivided, and elements are distributed among the offspring of the octant according to their positions. If necessary, this may be done recursively so that all octants have less than a prescribed number of elements. Subtrees having too few elements are pruned, coarsening the octree. To coarsen, children of an octant are accumulated to the octant's parent, and the leaf octants are deleted. This may also be repeated. If an octant that is a candidate for coarsening has one or more off-processor sub-octants, the octant is left untouched. The pruning reduces storage but has no effect on the efficiency of the partitioning algorithm. Empty subtrees are always skipped in the truncated partitioning traversal, and sparse subtrees are skipped with high probability. Therefore, there is no incentive to incur interprocessor communication to accomplish the pruning. Octree refinement and pruning traversal takes $O(N_{max})$, where N_{max} is the maximum number of octants on a processor, and requires no communication.

The thresholds for octant subdivision and coarsening determine the granularity of additions and deletions to a partition when using octree partitioning, and should be chosen accordingly. Currently, no more than

40 and no fewer than 10 elements are allowed per octant. Depending on the number and position of elements within an octant, the criterion for coarsening may be met after refinement. In this case, a tie is broken in favor of refinement.

4.2 Octree Partitioning

To partition a mesh initially using the octree, the initial mesh and associated octree are loaded onto one processor. A depth-first traversal of the octree is made to determine all subtree costs. For simple partitioning, the cost is the number of elements in the subtree. With p-refinement, this can be generalized to a function of the total number of degrees of freedom associated with a subtree. For a LRM, elemental costs are weighted by the inverse of element size to reflect the increased cost of time stepping smaller elements more frequently than larger ones.

Next, the octree is traversed a second time to accumulate octants into successive partitions. The total cost of the octree and the number of partitions (processors) are known, so the optimal cost per partition is also known. Tree nodes are visited in depth-first order and are added to the current partition if the cost of the subtree it roots does not exceed the optimal amount. If the subtree cost exceeds the partition size, the traversal recursively descends the tree and continues. Terminal octants are not split; thus, if a terminal octant overfills a partition, a decision must be made whether to add it or to close the current partition, leaving it slightly unfilled, and start work on the next partition. This decision is based on the relative level of imbalance and the cumulative cost of previously closed partitions to avoid a very large final partition. Once the initial tree is partitioned, subtrees are distributed across the processors by message passing.

OCTPART is an extension of the octree partitioning algorithm which operates in parallel to allow it to be used for dynamic load balancing [18]. To rebalance using OCTPART, each processor computes costs for each locally rooted subtree using traversals within its domain. The subtrees are sorted to be in depth-first order in a global traversal. The traversal step requires no interprocessor communication. An inexpensive parallel prefix operation is performed on the processor cost totals to obtain a global cost structure. This information enables a processor to determine its local tree traversal position in the global traversal.

As with the serial procedure, each processor traverses its subtrees to create partitions. A processor determines its initial partition index using the total cost of processors preceding it. Starting with this prefix cost, each processor traverses its subtrees accumulating the cost of visited nodes. Partitions end near cost multiples of C/P, where C is the total cost and P is the number of processors. Exceeding a multiple of C/P during the traversal is analogous to exceeding the optimal partition size in the serial case, and the same criteria are used to determine where to end partitions.

In contrast to the serial algorithm, a processor must begin its traversal with a specified partition.

When all processors finish their traversals, each subtree and its associated data is assigned to a partition and is migrated to that location, if necessary.

When used on non-octree generated meshes, elements may not be well-aligned with octant boundaries. This can lead to jagged partition boundaries, and a larger than necessary interprocessor boundary. An inexpensive boundary smoothing procedure [24] has been developed to correct for this. The smoothing procedure is independent of the octree and may be used as a post-processing step for any load balancer.

Typically, processor load balancing follows an h-refinement step. The ability to predict and correct for imbalance prior to enrichment can improve performance during refinement and coarsening while maintaining a balanced computation during the successive solution phase. Such predictive balancing strategies may be used with OCTPART or other partitioners [21, 23, 25].

5 Results

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 [19]. We match flow conditions to those of shock tube studies of Dillon [19] and Nagamatsu *et al.* [32]. Our focus is on the quasi-steady flow that exists behind the contact surface for a short time; thus, we initiate the problem by rupturing a hypothetical diaphragm between the two cylinders. Using symmetry, the flow may be solved in one half of the domain bounded by a plane through the vent. The larger cylinder (the shock tube) initially contains air 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. All results were obtained using 16 processors of an IBM SP2 computer.

5.1 Vent Tube Aspect Ratio Comparisons

Following umerical results of Nagamatsu *et al.* [32], we examine vent aspect ratios where the length L of the vent to its diameter D are 1, 2, and 3. The initial meshes contain 94,395, 69,572, and 61,648 tetrahedral elements, respectively, for L/D=1, 2, and 3.

The images on the left of Figure 3 illustrate the Mach number with velocity vectors at solution time $t = 1.0$ using the LRM for each L/D ratio. In each, a strong shock has formed near the downwind vent-shock tube

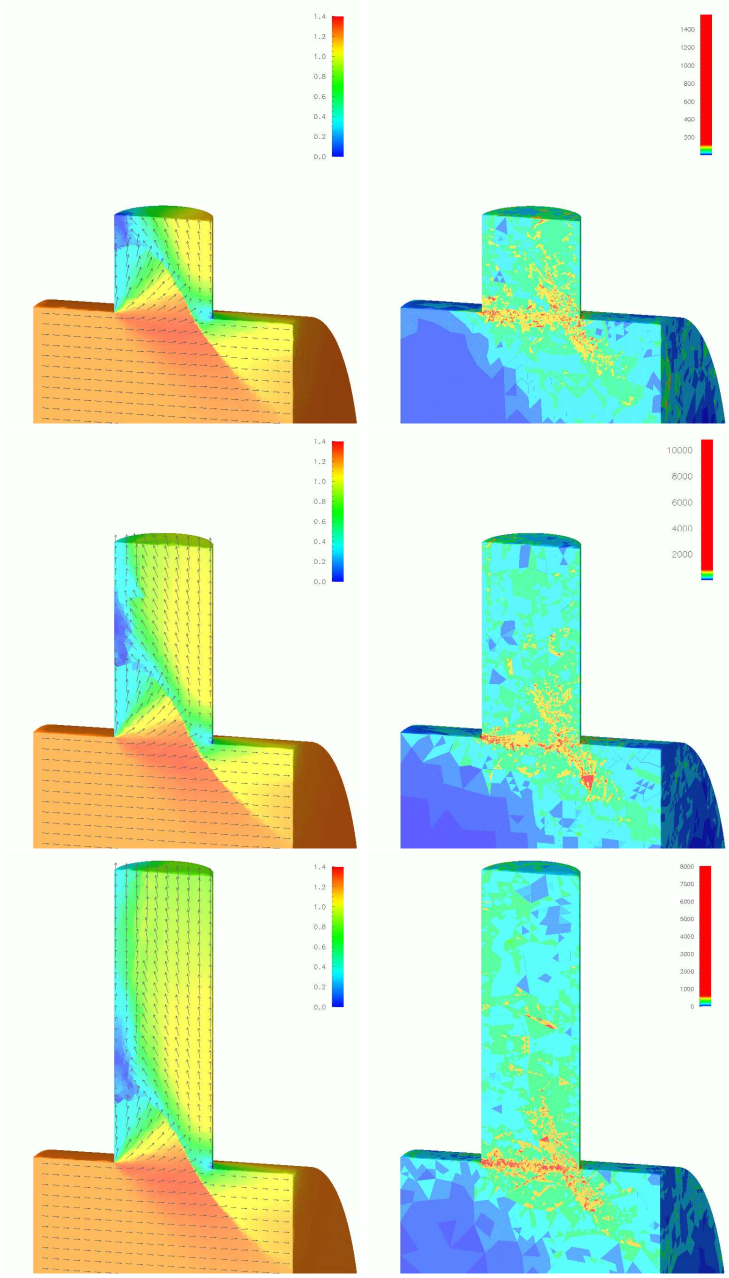


Figure 3: Projections of the Mach number and velocity vectors onto the surfaces of a perforated cylinder (left) and number of LRM time steps per element (right) at $t = 1.0$ for the L/D=1 (top), L/D=2 (center), and L/D=3 (bottom) cases. Mach number projections range from 0.0 (blue) to 1.4 (red), and time step counts range from 0 (blue) to approximately 10,000, with all values over approximately 750 shown in red.

interface, and a portion of the flow in the vent has accelerated 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. In the L/D=1 case, a supersonic flow has reached the vent exit, while the flow at the vent exit for L/D=2 and L/D=3 has not yet become sonic. These flow features compare favorably with experimental and numerical results of Nagamatsu *et al.* [32].

Figure 4 shows partitions of the initial meshes (left column) for this problem obtained using OCTPART. The meshes at $t = 1.0$ (right column) show extensive refinement near discontinuities and none in areas with little solution variation.

5.2 Global vs. Local Refinement

To advance the L/D=2 perforated tube solution from $t = 0$ to 0.1, the LRM takes 8.1-9.1$\times 10^7$ element time steps requiring the computation of 3.2-3.6$\times 10^8$ fluxes. To advance the computation with global time-stepping at the smallest acceptable time step would have required an estimated 1.3-2.4$\times 10^9$ element time steps and greater than 2.6-4.8 $\times$ 10^9 flux computations. This is a factor of 14-30 more element time steps and 7-15 more flux evaluations. The estimate assumes that the same spatial meshes would be used for the two methods, and the same solution would have been generated.

Figure 3 also illustrates the computational gain by using the LRM for the perforated tubes at $t = 1.0$ for each L/D ratio. Colors indicate the total number of local time steps taken by each element since the last mesh enrichment. For example, in the L/D=2 case, more than 10,000 time steps are taken on the smallest elements, while the largest elements take only 22. Small time steps are concentrated in the shock and expansion regions near the intersection of the two cylinders. The largest time steps occur in the interior of the shock tube.

The LRM rounds time steps down to the nearest power of two, which encourages adjacent elements to step by the same amount, allowing the computed flux between them to be shared. Employing this strategy reduced the number of fluxes computed per element from 3.97 to 2.47 with L/D=2 from $t = 0.0$ to $t = 1.0$, as compared to computation using unrounded time steps. The number of faces visited per element time step, a measure of the overhead involved with finding candidate elements to step, was reduced from 5.88-7.05 to 3.73-4.25, in the L/D=2 case.

5.3 Size-Weighted Balancing

Let the time-step imbalance be the maximum number of elements time stepped on a processor relative to the average number stepped on all processors [18]. Likewise, let the flux imbalance be the maximum number of

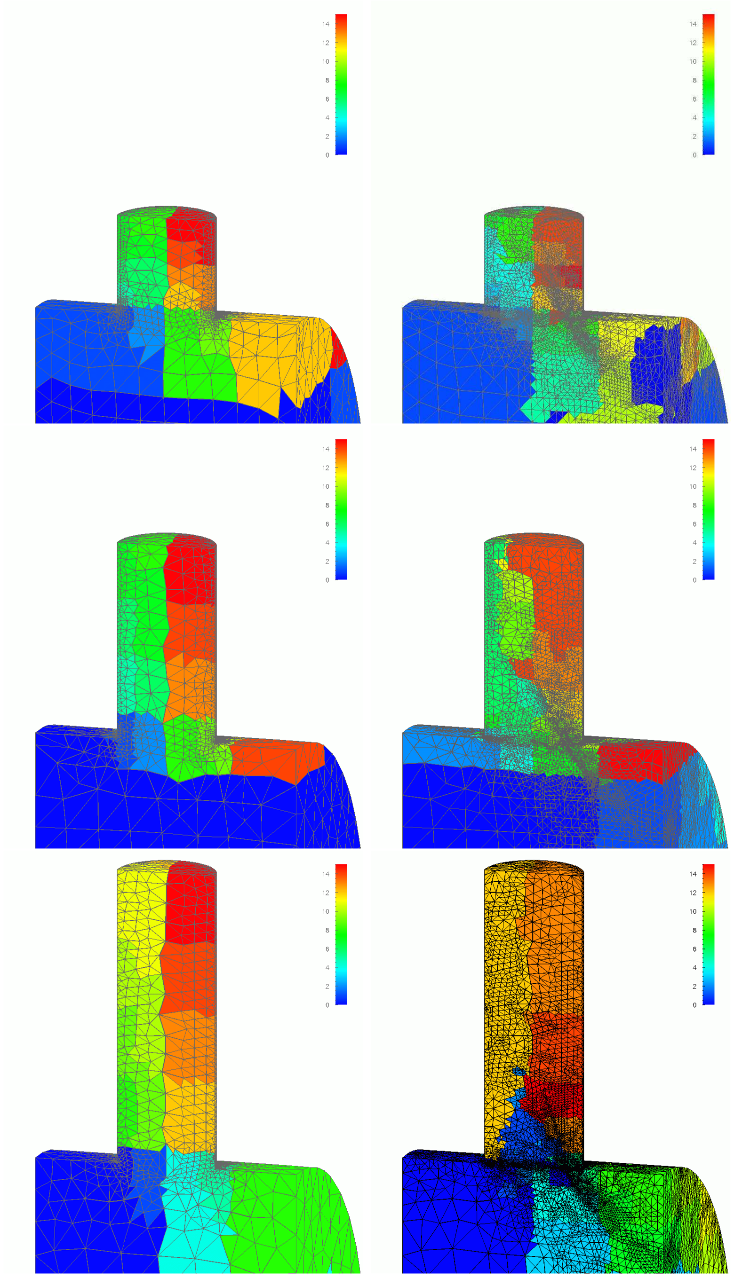


Figure 4: Partitioning for the runs of Figure 3 using OCTPART. The left column shows partitionings of the initial meshes onto 16 processors, and the right shows partitionings of the meshes at t = 1.0. The L/D=1, 2, and 3 cases are shown from top to bottom. Colors indicate processor assignments.

fluxes computed on a processor relative to the average number computed on all processors. In either case, let the average imbalance at simulation time t be a weighted average of all imbalances to time t. The weighting is the wall-clock duration of an imbalance relative to the total wall-clock time of the computation.

To illustrate the advantages of size-weighted balancing, we solved the L/D=2 problem from $t = 0.0$ to $t = 0.1$ with and without size-weighting. As shown in Figure 5, balancing based solely on the number of elements per processor produces average time-step imbalances of 1.38-1.52 while size-weighted balancing reduced this to 1.21-1.28. Likewise, average flux imbalances of 1.37-1.50 were reduced to 1.19-1.25 by the size-weighted balancing. One of the runs of Figure 5 is shown in more detail in Figure 6. Size-weighted balancing has a greater variation than the un-weighted balancing, but its overall performance is better.

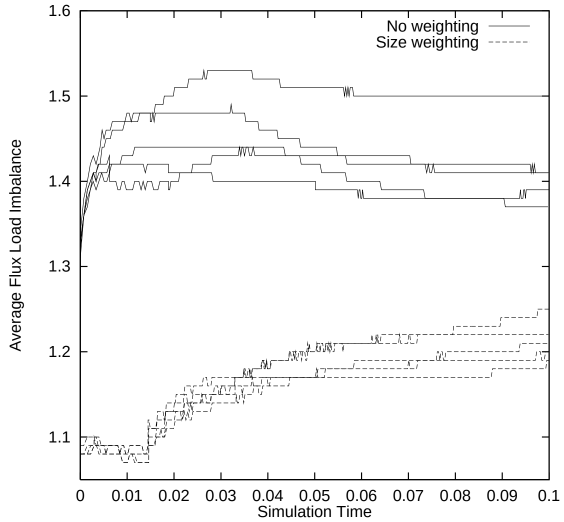
Element weights are determined strictly by their size (Section 4.2), but for efficiency the LRM rounds element time steps down to the next lower power of two (Section 3.2). This may lead to a variation up to a factor of 2 in imbalance when using size-weighting, as seen in Figure 6. For more accurate size-weighted balancing, the rounding should be applied to the weight as well.

When a given mesh is partitioned and migrated, the resulting distributed mesh is not stored in a unique way. Mesh elements, faces, and vertices migrated to a processor are added to the local data structure's linked lists in the order that they arrive. Order variations do not directly affect the solution process but they do affect mesh enrichment. Enrichment is performed in the order that elements are encountered in the linked structures. Additionally, different enrichment operations are performed on partitions in order to reduce unnecessary data migration. For example, procedures might decline to coarsen a region on an interprocessor boundary. Thus, runs with identical input will lead to slightly different meshes and, hence, slightly different solutions. To ameliorate such variations, the runs of Figures 5 and 6 are performed five times to observe trends in behavior. However, these variations make comparisons by CPU times difficult or impossible.

6 Discussion

Distributed octree structures are useful at several stages of a parallel adaptive finite element analysis. Mesh generation makes extensive use of the octree, and a successful dynamic load balancing algorithm is based on the octree. OCTPART is an inexpensive parallel repartitioner which controls partition shapes well [18, 24].

The local refinement method greatly reduces computational costs of transient solutions with no loss of accuracy relative to global time step



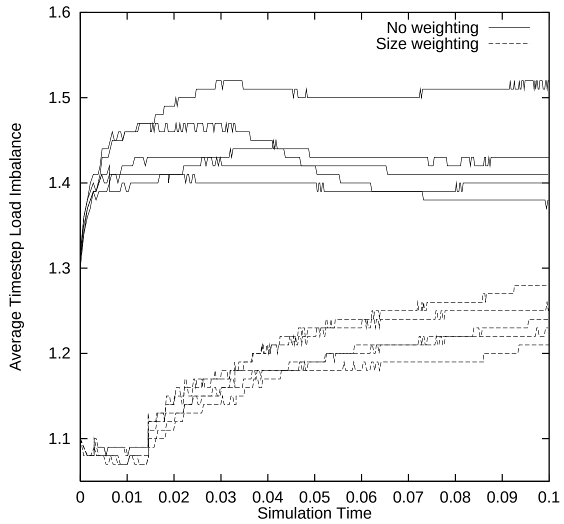


Figure 5: Average flux (top) and time-step (bottom) load imbalances for a sequence of five runs with and without size weighting.

approaches. While it does introduce additional storage and synchronization requirements, the demonstrated factors of at least 7 in flux computation and at least 14 in time step computation more than justify these costs.

Balancing the load of a LRM is a difficult problem. The introduction of

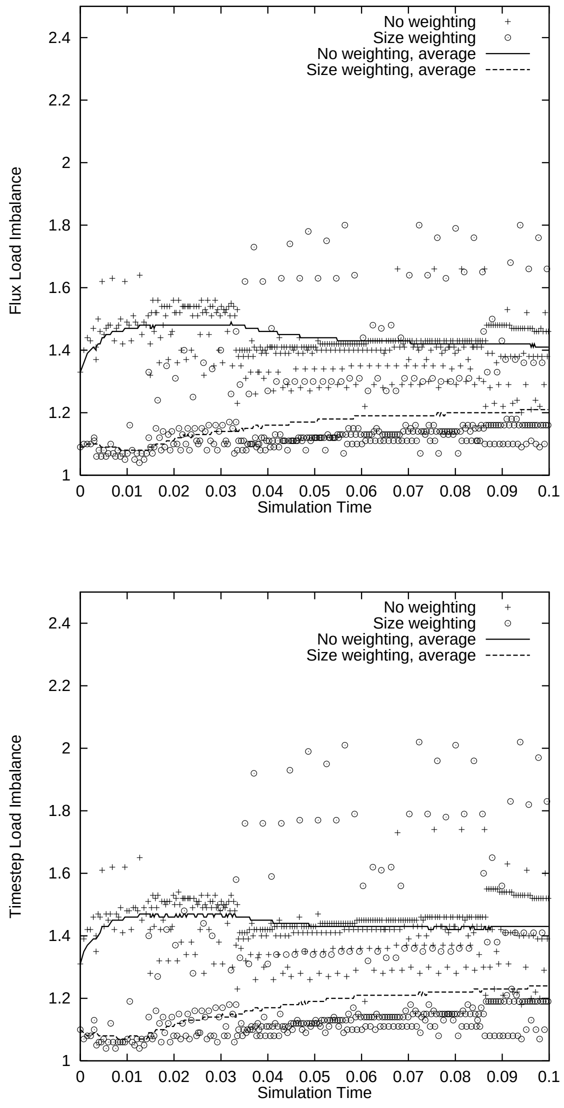


Figure 6: Flux (top) and time-step (bottom) load imbalances during a representative run as shown in Figure 5. Scatter data show the actual load imbalance as a function of simulation time. The curves show average load imbalance.

sized-weighted octant partitioning reduces imbalance significantly relative to simple element weighting. In our example, the time step imbalance improved from approximately 1.45 to 1.24 and flux imbalance improved from approximately 1.44 to 1.22 with size weighting for L/D=2. Further study on balancing LRMs is warranted, especially when p-refinement is introduced.

An efficient adaptive and parallel procedure should spend most of its time performing computation as opposed to performing mesh enrichment and rebalancing. We provide estimates, based on an instrumentation of the code, of the portion of time spent on each phase of the computation for the runs used to obtain the L/D=3 results. Our findings indicate that 3.6% of the time is spent on mesh enrichment, 2.8% of the time is spent on rebalancing and partition smoothing, and 81.0% of the time is spent doing computation. The remaining 12.7% is spent on tasks such as loading meshes, writing checkpoint and solution files, and monitoring performance.

All load balancing, migration, enrichment, and solution procedures have been designed to scale. Nevertheless, a detailed study to verify this property must be conducted. Barring such an investigation, we note that results of a two-dimensional system similar to our three-dimensional system produced excellent scalability [8]. Scalability studies of three-dimensional steady flows using PSIRB with enrichment and migration strategies similar to those reported here also indicated very good scalabilty [45]. Mesh generation using the distributed octree structures rather than replicated trees enhances the scalability of that process. Finally, preliminary runs of the present system on a larger number of processors produced encouraging results.

Incremental migration strategies for use with adaptivity are being developed [14, 18]. If cost or locality of data movement is more important than global load balance, another approach with OCTPART may be taken. The processors may shift partition boundaries, thus, migrating subtrees from a processor p_i to its neighbors p_{i-1} and p_{i+1}. If, for example, processor p_i seeks to transfer load r to p_{i-1}, it may simply traverse its subtrees accumulating their loads until it reaches r. The nodes visited comprise a subtree which may be transferred to p_{i-1} and which is contiguous in the traversal with the subtrees in p_{i-1}. Likewise, if p_i desires to transfer work to p_{i+1}, the reverse traversal could remove a subtree from the trailing part of p_i.

Multilevel partitioning algorithms allow a more expensive partitioning algorithm to be used on a coarser structure to achieve good partitions at a smaller cost [33]. The distributed octree may be useful in this context as well. Partitioning the terminal octants as the coarser mesh with a graph partitioner is a potential way to take advantage of this.

The Discontinuous Galerkin Method lends itself to a parallel, higher order spatial solution. The spatial accuracy can be extended to an arbitrarily high order while still maintaining a nearest neighbor communication pattern. We are currently implementing a locally adaptive p-refinement method to complement the existing h-refinement.

We currently use Van Leer flux vector splitting. Other methods such as Roe [39, 40] and Osher [35] introduce substantially less artificial dissipation, producing more accurate solutions than Van Leer on the same mesh, but at the expense of more computationally demanding iterations.

7 Acknowledgements

We wish to thank our colleagues at Rensselaer, particularly Stephen Cosgrove, for their assistance.

This research was partially supported by the U.S. Army Research Office through Contract Number DAAH04-95-1-0091; the National Science Foundation though grant number CCR-9527151 and grant number DMS-9318184; the U.S. Office of Naval Research through grant number N00014-95-1-0892; a DARPA Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland; and a fellowship from the Northrop Grumman Corporate Research Center.

References

[1] S. Adjerid, J. E. Flaherty, P. Moore, and Y. Wang. High-order adaptive methods for parabolic systems. *Physica-D*, 60:94–111, 1992.

[2] P. L. Baehmann, S. L. Wittchen, M. S. Shephard, K. R. Grice, and M. A. Yerry. Robust, geometrically based, automatic two-dimensional mesh generation. *Int. J. Numer. Meth. Engng.*, 24:1043–1078, 1987.

[3] M. W. Beall and M. S. Shephard. A general topology-based mesh data structure. *Int. J. Numer. Meth. Engng.*, 40(9):1573–1596, 1997.

[4] M. J. Berger. On conservation at grid interfaces. *SIAM J. Numer. Anal*, 24(5):967–984, 1987.

[5] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Computers*, 36(5):570–580, 1987.

[6] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.*, 53:484–512, 1984.

[7] 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(22):3889–3907, 1995.

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

[9] 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 Appl. Aero. Conf.*, number 94-1888, Colorado Springs, 1994.

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

[11] 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.*, 84:90–113, 1989.

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

[13] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel and Dist. Comput.*, 7:279–301, 1989.

[14] 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.*, 16:157–182, 1994.

[15] H. L. de Cougny, M. S. Shephard, and C. Özturan. Parallel three-dimensional mesh generation. *Comp. Sys. Engng.*, 5:311–323, 1994.

[16] H. L. de Cougny, M. S. Shephard, and C. Özturan. Parallel three-dimensional mesh generation on distributed memory MIMD computers. *Engng. with Computers*, 12(2):94–106, 1996.

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

[18] 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, editors, *Modeling, Mesh Generation, and Adaptive Numerical Methods for Partial Differential Equations*, volume 75, pages 215–242, Berlin-Heidelberg, 1995. Springer-Verlag.

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

[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.*, 36:745–764, 1993.

[21] J. E. Flaherty, M. Dindar, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Uniform and variable predictive load balancing for parallel adaptive finite element computation. To appear, *Proc. 17th Biennial Conf. on Numer. Analysis*, 1997.

[22] J. E. Flaherty, R. M. Loy, C. Özturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Parallel structures and dynamic load balancing for adaptive finite element computation. SCOREC Report 22-1996, Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, 1996. To appear *Appl. Num. Math.*

[23] J. E. Flaherty, R. M. Loy, P. C. Scully, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Load balancing and communication optimization for parallel adaptive finite element computation. To appear, *Proc. SCCC '97*, 1997.

[24] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws. IMA Preprint Series 1483, Institute for Mathematics and its Applications, University of Minnesota, 1997. To appear, *J. Parallel and Dist. Comput.*

[25] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Predictive load balancing for parallel adaptive finite element computation. In H. R. Arabnia, editor, *Proc. PDPTA '97*, volume I, pages 460–469, 1997.

[26] P. L. George. *Automatic Mesh Generation*. John Wiley and Sons, Ltd., Chichester, 1991.

[27] G. Karypis and V. Kumar. Metis: Unstructured graph partitioning and sparse matrix ordering system. Tech. Report, University of Minnesota, Department of Computer Science, Minneapolis, MN, 1995.

[28] A. Kela. Hierarchical octree approximations for boundary representation-based geometric models. *Computer Aided Design*, 21:355–362, 1989.

[29] W. L. Kleb and J. T. Batina. Temporal adaptive Euler/Navier-Stokes algorithm involving unstructured dynamic meshes. *AIAA J.*, 30(8):1980–1985, 1992.

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

[31] R. A. Ludwig, J. E. Flaherty, F. Guerinoni, P. L. Baehmann, and M. S. Shephard. Adaptive solutions of the Euler equations using finite quadtree and octree grids. *Computers and Structures*, 30:327–336, 1988.

[32] 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 Tech. Report ARCCB-TR-87016, Benet Weapons Laboratory, Watervliet, 1987.

[33] L. Oliker and R. Biswas. Efficient load balancing and data remapping for adaptive grid calculations. In *Proc. 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 33–42, Newport, 1997.

[34] L. Oliker, R. Biswas, and R. C. Strawn. Parallel implementaion of an adaptive scheme for 3D unstructured grids on the SP2. In *Proc. 3rd International Workshop on Parallel Algorithms for Irregularly Structured Problems*, Santa Barbara, 1996.

[35] S. Osher and S. Chakravarthy. Upwind schemes and boundary conditions with applications to the euler equations in general coordinates. *J. Comput. Phys.*, 50, 1983.

[36] A. Patra and J. T. Oden. Problem decomposition for adaptive *hp* finite element methods. *Comp. Sys. Engng.*, 6(2):97, 1995.

[37] R. Perucchio, M. Saxena, and A. Kela. Automatic mesh generation from solid models based on recursive spatial decompositions. *Int. J. Numer. Meth. Engng.*, 28:2469–2501, 1989.

[38] A. Pothen, H. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Mat. Anal. Appl.*, 11(3):430–452, 1990.

[39] P. L. Roe. Approximate riemann solvers, parametric vectors and difference schemes. *J. Comput. Phys.*, 43, 1981.

[40] P. L. Roe. Characteristic based schemes for the euler equations. *Annual Review of Fluid Mechanics*, 18, 1986.

[41] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel diffusion algorithms for repartitioning of adaptive meshes. Tech. Report 97-014, University of Minnesota, Department of Computer Science and Army HPC Center, Minneapolis, MN, 1997.

[42] W. J. Schroeder and M. S. Shephard. Geometry-based fully automatic mesh generation and the delaunay triangulation. *Int. J. Numer. Meth. Engng.*, 26:2503–2515, 1988.

[43] M. S. Shephard. Approaches to the automatic generation and control of finite element meshes. *Applied Mechanics Review*, 41(4):169–185, 1988.

[44] M. S. Shephard. Update to: Approaches to the automatic generation and control of finite element meshes. *Applied Mechanics Reviews*, 49(10, part 2):S5–S14, 1996.

[45] 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 Comput. in CFD*, number R-807, pages 6.1–6.49. Agard, Neuilly-Sur-Seine, 1995.

[46] M. S. Shephard and M. K. Georges. Automatic three-dimensional mesh generation by the Finite Octree technique. *Int. J. Numer. Meth. Engng.*, 32(4):709–749, 1991.

[47] C.-W. Shu and S. Osher. Efficient implementation of essentially non-oscillatory shock-capturing schemes, II. *J. Comput. Phys.*, 27:1–31, 1978.

[48] M. L. Simone, M. S. Shephard, J. E. Flaherty, and R. M. Loy. A distributed octree and neighbor-finding algorithms for parallel mesh generation. Tech. Report 23-1996, Rensselaer Polytechnic Institute, Scientific Computation Research Center, Troy, 1996.

[49] P. K. Sweby. High resolution schemes using flux limiters for hyperbolic conservation laws. *SIAM J. Numer. Anal*, 21:995–1011, 1984.

[50] B. Van Leer. Towards the ultimate conservative difference scheme. IV. A new approach to numerical convection. *J. Comput. Phys.*, 23:276–299, 1977.

[51] B. Van Leer. Flux vector splitting for the Euler equations. ICASE Report 82-30, ICASE, NASA Langley Research Center, Hampton, 1982.

[52] V. Vidwans, Y. Kallinderis, and V. Venkatakrishnan. Parallel dynamic load-balancing algorithm for three-dimensional adaptive unstructured grids. *AIAA J.*, 32(3):497–505, 1994.

[53] C. H. Walshaw, M. Cross, and M. Everett. Mesh partitioning and load-balancing for distributed memory parallel systems. In *Proc. Par. Dist. Comput. for Comput. Mech.*, Lochinver, Scotland, 1997.

[54] S. Wheat, K. Devine, and A. MacCabe. Experience with automatic, dynamic load balancing and adaptive finite element computation. In H. El-Rewini and B. Shriver, editors, *Proc. 27th Hawaii International Conference on System Sciences*, pages 463–472, Kihei, 1994.

[55] M. A. Yerry and M. S. Shephard. Automatic three-dimensional mesh generation by the modified octree technique. *Int. J. Numer. Meth. Engng.*, 20:1965–1990, 1984.