Adaptive Local Refinement with Octree Load-Balancing for the Parallel Solution of Three-Dimensional Conservation Laws

J. E. Flaherty, R. M. Loy, M. S. Shephard,
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. The 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. Processor load imbalances, introduced at adaptive enrichment steps, are corrected by using traversals of an octree representing a spatial decomposition of the domain. To accommodate the variable time steps, octree partitioning is extended to use weights derived from element size. Partition boundary smoothing reduces the communications volume of partitioning procedures for a modest cost. Computational results comparing parallel octree and inertial partitioning procedures are presented for the three-dimensional Euler equations of compressible flow solved on an IBM SP2 computer.

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, 2, 22, 33, 34] 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- or p-refinement.

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 [3, 5, 7, 15] and unstructured [26] meshes. In Section 3, we introduce 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, it leads to a large improvement in overall efficiency.

Poor partitioning of data across the processors of a parallel computer leads to high communication costs. Several static partitioning algorithms have been developed [4, 19, 32]; however, these may be inefficient in an adaptive computational environment. *Parallel Sort Inertial Recursive Bisection* (PSIRB) [35] 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 [6, 9, 14, 15, 17, 27, 35, 43, 44]. These methods provide inexpensive balancings, but may not reduce communication costs.

Octree decomposition is a successful strategy for generating three-dimensional unstructured meshes [36]. We use a hierarchical representation of finite element meshes that is appropriate for h- or p-refinement. A *Parallel Mesh Database* [21, 35] provides operators to create and manipulate distributed mesh data, and a *parallel octree library* supports the creation and distribution of octree structures. We describe a dynamic partitioning technique that exploits the properties of 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 (Section 4) or in parallel (Section 5) [17]. In either case, it is inexpensive; hence, it may be used with adaptive procedures. Partitioning requires tree traversals that (i) calculate the processing costs of subtrees and (ii) form the partitions. Weighting factors proportional to element size may be employed with octree partitioning to help balance the load of LRMs.

Partitions often have uneven boundaries with elements penetrating into or protruding from neighboring partitions, which increase communication costs. In Section 6, we describe a partition boundary smoothing operation which is used to reduce the number of faces lying on partition boundaries. The resulting partitions have approximately the same communications volume as other more expensive strategies [25, 32].

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 7 and are discussed in Section 8.

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 (Section 7), 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) [40, 41] and Essentially Non-Oscillatory (ENO) [37] methods, usually 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, 12, 13]. 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 7 use piecewise constant (p=0) approximations and explicit Euler integration; however, p-refinement may be incorporated [16].

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 [13, 37]; we use van Leer's flux vector splitting [17, 42, 28].

3 The Local Refinement Method

Our LRM selects spatially-dependent time steps based upon the Courant stability condition for explicit time integration. Thus, in a given time period, a smaller number of larger time

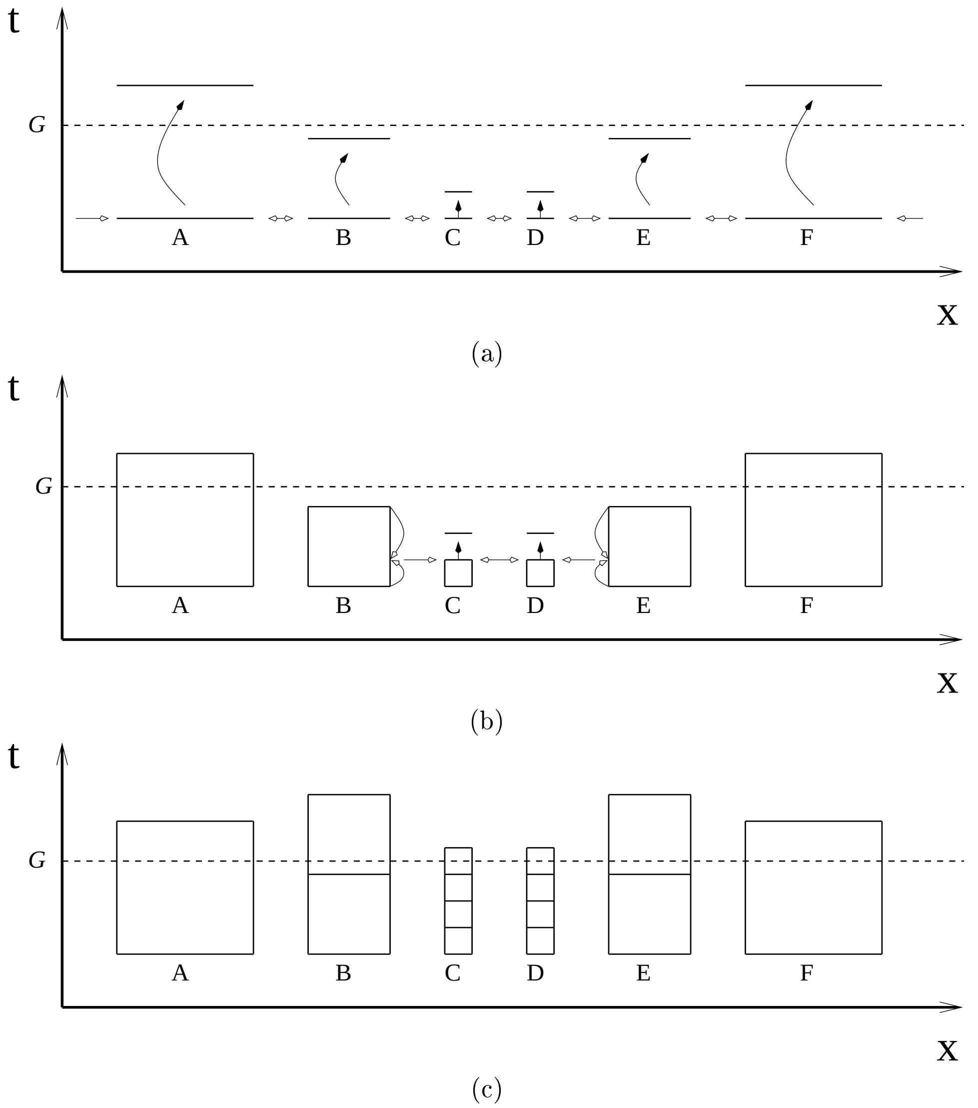


Figure 1: The Local Refinement Method. The set of one-dimensional elements A-F choose time steps according to their stability criteria. (a) The elements exchange information with their neighbors (or evaluate boundary conditions) and advance by a single time step. (b) Elements C and D receive interpolated 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.

steps will be taken on large elements, and the opposite will occur on the 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 the 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.

All elements may advance by their initial time steps (Figure 1a) because spatially adjacent information needed to compute numerical fluxes is available from either neighboring elements or the prescribed boundary conditions.

After this time step (Figure 1b), only elements C and D are able to take another step. The other elements have either reached the goal time (A and F) or lack the necessary boundary fluxes to progress (B and E). Element C may get flux data from its neighbor D directly and from B by using linear interpolation in time. Element D does likewise.

After several time steps, elements B-E have the necessary data and all may step to reach the goal G (Figure 1c). Output or error estimation at G is performed using interpolation to time G.

The six elements shown in Figure 1 have been advanced to time G using four rounds of 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.

Temporal interpolation requires storage for solution data at the previous and current times. Additional space may be required so that the solution may be synchronized and interpolated to a common time for checkpointing or outputing. The interval between synchronization times is referred to as a *major* step. Each major step is composed of several smaller steps, each of which performs a single time step on elements that have the necessary data from their neighbors. These elements are determined by traversing the mesh, but using the octree connectivity might be more efficient. At the beginning of a major step, the software advances G by a multiple of the smallest stable time step on any element of the mesh. An element's time step may straddle the goal time, but it may not take another step after passing G. Thus, a synchronization time is reached.

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 can easily be 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 [26].

3.1 Error Control

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. 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 7. However, discretization error estimates [6, 11, 16] 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.

The LRM complicates error control. Spatial gradients, used as error indicators, are available only when elements are at the same time. Therefore, error evaluation is only done at the end of a major step. If the error is unacceptable, the solution is rolled back to the beginning of the major step, h-refinement is performed, and the process repeated.

3.2 h-Refinement

Mesh refinement and coarsening utilize edge-based error indicators to determine where to perform enrichment [31, 35]. An element may be subdivided isotropically or anisotropically depending on the number of its edges 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.

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 and interior edges of a polyhedron are removed, and the polyhedron 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 may be required if the mesh near an interprocessor boundary is marked for coarsening.

We assign error indicators and solution values to the vertices of the mesh. The solution and error at a given vertex are assigned the volume-weighted average of the piecewise-constant element solutions and errors containing that vertex. Edges are marked for enrichment based on their vertex error values. 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.

4 Octree Partitioning

An octree-based mesh generator [36] 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. For meshes generated by other procedures, an element may be associated with the octant that contains its centroid. Octant subdivision would ensure that octants contain no more than a maximum allowable number of elements.

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 the inverse of element size to reflect the increased cost of time stepping smaller elements more frequently than larger ones. Alternatively, if the octree were sorted according to element size, costs would correspond to tree depth.

The second phase of the algorithm performs another traversal of the octree to accumulate octants into successive partitions. Since the total cost of the octree and the number of partitions (processors) are known, the optimal cost per partition is also known. Beginning at the root, 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.

4.1 Distributed Octree Data Structures

Once the initial tree is partitioned, subtrees are distributed across the processors by message passing. The distributed octree is still 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 [39]. This is queried to determine if an object is local or not. 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 number and remote address for all links, storage for local links is the same as the serial case. However, 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 [39].

A generalized concept of an octree root must be adopted with a distributed octree structure. An octant's parent may not exist on the local processor, and, in this case, we call the octant a *local root*. A parent query still returns a pointer to a structure; however, it contains information about the parent's processor, address, bounding box, and 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, the bounding coordinates of an octant usually require 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.

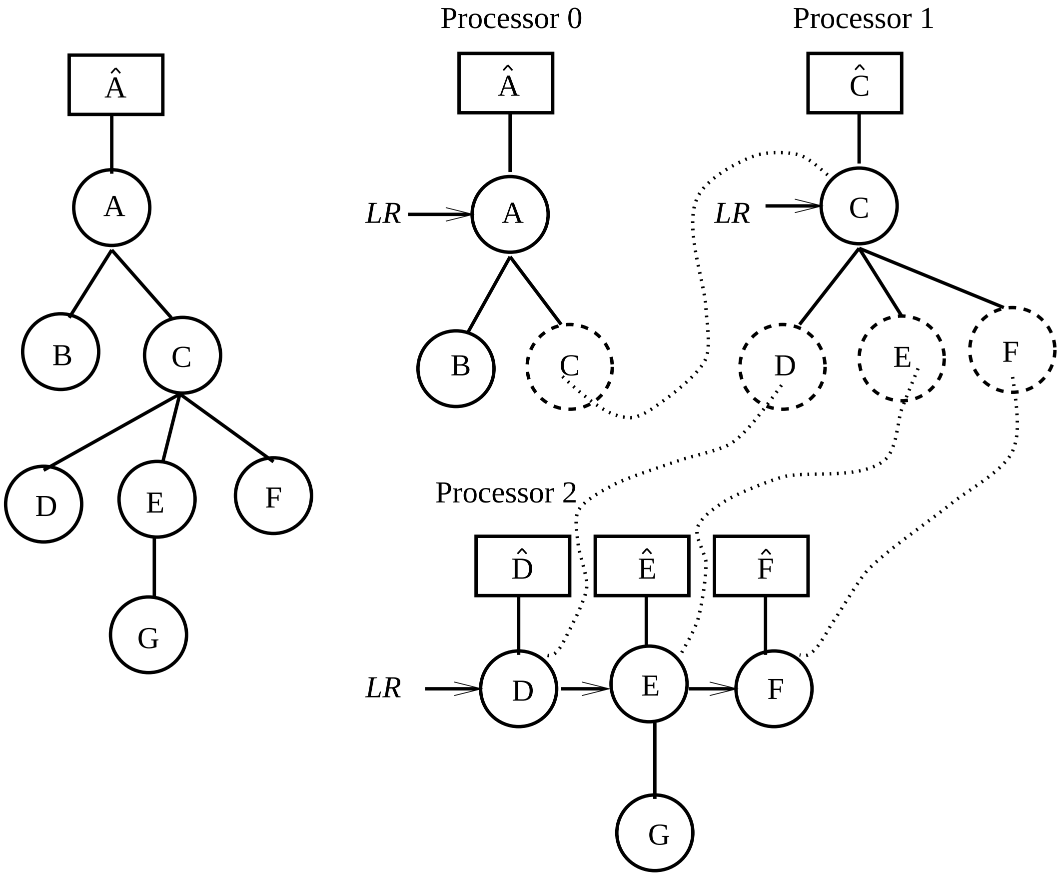
A simple tree having root A appears in Figure 2a. Its data structure including bounding box information is stored in $\hat{\mathrm{A}}$. In Figure 2b, the tree has been distributed across three processors. The dotted circles indicate remote references. Only the remote location is stored in these cases. 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.

4.2 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 the elements are created or deleted; 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 balancing.

We assume elements created in the mesh refinement stage lack an octree association. Since a newly created element lies within its parent, 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.

Elements resulting from mesh coarsening must also be assigned 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



(a) (b)

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

connectivity.

After mesh refinement, a traversal of the octree is made to determine if the octree needs to be extended or pruned. Crowded leaf octants are subdivided by distributing elements to offspring of the octant according to their positions. If necessary, this may be done recursively so that all octants have less than a proscribed number of elements.

Conversely, subtrees having too few elements are pruned to coarsen the octree. Elements are accumulated to their parent octant's parent, and the leaf octants are deleted. This may also be repeated. If an octant is a candidate for coarsening, but has one or more off-processor sub-octants, then the octant is left untouched. Pruning the octree reduces storage use but has no effect on the efficiency of the partitioning algorithm. Empty subtrees will always be skipped in the truncated partitioning traversal, and sparse subtrees will be be skipped with high probability. Therefore, there is no incentive to incur interprocessor communication to accomplish the pruning.

Comprised of local operations only, the octree refinement and pruning traversal takes $O(N_{max})$, where N_{max} is the maximum number of octants on a processor.

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.

5 Parallel Octree Partitioning

The serial octree partitioning algorithm may be extended to operate in parallel, and we refer to this algorithm as OCTPART.

When dynamic partitioning is needed, 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. This 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 is used to determine where to end partitions. In contrast to the serial algorithm, a processor must begin its traversal with a specified partition. In serial, there is the option to include a small additional load in a partition rather than beginning precisely at a multiple of C/P. In practice, this difference is negligible.

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. Migration may be done using global communication; however, on some computers, it is more efficient to move data via simultaneous processor shift operations. This linear communication pattern is possible due to the unidimensionality of the partitioning traversal.

6 Partition Smoothing

Application of the octree-based partitioning method to octree-generated meshes yields communications volumes similar to recursive spectral bisection [17]. However, since mesh refinement and coarsening are independent of the octree, elements are not necessarily aligned with octant boundaries. Thus, choosing partition boundaries based on octants yields partitions with bumpy surfaces, which increase communication costs. This effect may be reduced by smoothing the partition boundaries [20, 29]. To do this, each processor traverses 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 processor, reducing both its boundary and that of the receiver 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 case 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.* [29] perform processor boundary smoothing by a similar iterative method. They identify elements on interprocessor boundaries whose vertices 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 the smoothing requires computational time proportional to the number of elements on a processor's boundary. It requires communication time proportional to the number of elements selected for smoothing migration. The actual amount varies with the initial partition quality. In practice, the number of boundary faces on a processor may be reduced from 0-25%.

7 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 [18]. We match flow conditions to those of shock tube studies of Dillon [18] and Nagamatsu *et al.* [30]. 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 initial mesh contains 80,659 tetrahedral elements. 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.

Figure 10 illustrates the Mach number with velocity vectors at solution time $t = 0.3$ using global time stepping. A strong shock has formed near the downwind vent-shock tube 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. These flow features compare favorably with experimental and numerical results of Nagamatsu *et al.* [30]. The superior numerical properties of the LRM [26] allowed the same solution to be computed with greater accuracy using only 8 mesh enrichments, as opposed to the 61 needed with a global time step. A time sequence showing the solution obtained with the LRM on the left and the corresponding partitionings using OCTPART on the right is given in Figure 12. The center image may be compared to the global time stepping solution of Figure 10.

When a given mesh is partitioned and migrated, the resulting distributed mesh will not be 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 they arrive. The result will always describe the same mesh, but the order of the elements and other entities may vary. This does not directly affect the solution process but does affect mesh enrichment. Because enrichment is performed in the order that elements are encountered, different meshes can result from the same input mesh and error indicators. Additionally, the enrichment process performs different mesh operations depending on the partitioning so as to avoid unnecessary element migration. For example, it might decline to coarsen a region on an interprocessor boundary. Thus, runs with identical input will lead to slightly different meshes and, hence, solutions. To ameliorate this variation, we ran each example five times and noted trends in behavior. However, these variations make comparisons by CPU times difficult or impossible.

7.1 Global vs. Local Refinement

To advance the 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 meshes would be used for the two methods, and the same solution would have been generated.

Figure 11 illustrates the computational gain achieved by using the LRM for the mesh at $t = 1.05$ in the run of Figure 12. Shading indicates the total number of local time steps taken by each element since the last mesh enrichment. Over 5000 time steps are taken on the smallest elements for every one on the largest elements. 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 main tube.

The LRM rounds time steps down to the nearest power of two. As described, a greater number of adjacent elements 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. 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.

7.2 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 [17]. Likewise, let the flux imbalance be the maximum number of 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.

As shown in Figure 3, 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 3 is shown in more detail in Figure 4. Size-weighted balancing has a greater variation than the un-weighted balancing, but its overall performance is better.

7.3 Partition Performance

We compare the performance of PSIRB and OCTPART using the percentage of elements moved during migration as well as balancing time. Comparison of partition quality follows in Section 7.4.

The percentages of elements moved by PSIRB and OCTPART during balancing are shown in Figure 5 for five runs of the perforated tube problem. Rebalance zero corresponds to the initial partitioning; thus, both techniques show a high percentage of data movement. Rebalancings 1, 4, 7, ..., 13 are performed after mesh coarsening while 2, 5, 8, ..., 14 follow mesh refinement. The other rebalancing indices follow "snapping," which is an operation to assure mesh validity with respect to geometry that generally involves element migration [35]. OCTPART tends to move fewer elements than PSIRB after the coarsening and snapping phases, and is generally comparable to or slightly better than PSIRB after refinement. Figure 6 underscores these trends by plotting relative percent differences in data migration between the two techniques. If m_{PSIRB} and m_{OCTPART} represent the percentages of elements moved by PSIRB and OCTPART during a given rebalancing, the relative percent difference between the two methods is

$$m_{\text{diff}} = \frac{m_{\text{PSIRB}} - m_{\text{OCTPART}}}{m_{\text{PSIRB}}} \times 100. \tag{6}$$

Thus, a positive value indicates that OCTPART is outperforming PSIRB. The mean relative percent improvement in data movement of OCTPART compared to PSIRB is approximately 24% for each of the five runs.

Figure 9 shows a worst-case behavior of PSIRB. The mesh on the left was refined adaptively to produce the mesh on the right. The refinement has shifted the principal axis of inertia in such a way that 100% of the elements are moved from their prior processor location by PSIRB. This indicates the dangers in using a static partitioning procedure for transient problems.

In Figure 7, we show the relative percent differences in balancing time between PSIRB and OCTPART computed as in (6) for a sequence of rebalancings. While PSIRB has an advantage relative to OCTPART at the beginning of the computation, OCTPART begins to consistently outperform PSIRB as the simulation progresses. The mean relative percent improvement of OCTPART compared to PSIRB for each run is 17-26%.

7.4 Partition Smoothing

We evaluate the performance of partition smoothing by its effect on the cost of interprocessor communication. We appraise this cost using a *global surface index* (GSI) [10] which is the percentage of all element faces on interprocessor boundaries. For the discontinuous Galerkin method used herein, the GSI is equivalent to the number of edge "cuts" in the communication graph induced by a partitioning [23, 24, 38] normalized by the total number of these edges. Normalization makes the metric independent of problem size.

The effect of partition boundary smoothing on the GSI is shown for a run of 16 rebalancings for OCTPART and PSIRB in Figure 8. One iteration of smoothing reduces the global surface index by 1.2-1.8 percentage points for OCTPART and 0.7-1.4 points for PSIRB. Repeating the smoothing yields an additional improvement of 0.1-0.3 points for OCTPART and 0.0-0.2 points for PSIRB. Total relative improvement of GSI after two smoothing iterations was 22-27% for OCTPART and 18-26% for PSIRB. More than two smoothing iterations did not provide a significant improvement, and, in most cases, one smoothing appears to be sufficient.

The relative partition quality of OCTPART and PSIRB can be seen in the lower curves of the Figure 8. While OCTPART outperforms PSIRB in data movement and run time (Section 7.3), PSIRB tends to produce lower GSIs.

8 Discussion

The local refinement method greatly reduces computational costs of transient solutions with no loss of accuracy relative to global time step approaches. While it does introduce additional storage and synchronization requirements, the demonstrated factor of at least 7 in flux computation and at least 14 in time step computation more than justifies these costs.

Octree-based partitioning is an effective and efficient partitioning strategy that may either be used in conjunction with octree mesh generation [36] or on its own. It provides a suitable means of controlling communication volumes based solely on a geometric decomposition of space. The amount of data movement performed by OCTPART is superior to to that of PSIRB by approximately 24%. OCTPART is also faster than PSIRB by 17-26%.

Balancing the load of a LRM is a difficult problem. The introduction of 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. Further study on balancing LRMs is warranted, especially when p-refinement is introduced.

Partition boundary smoothing reduced the GSI of a partitioned mesh by at least 22% for OCTPART and 18% for PSIRB. Additional smoothing opportunities may provide even greater improvements.

Incremental migration strategies for use with adaptivity are being developed [15, 17]. 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.

9 Acknowledgements

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; 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] C. G. Armstrong, ed. *Advances in Engng. Software*, vol. 13:5/6. Computational Mechanics Publ., Southhampton, 1991.

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

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

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

[6] 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.

[7] R. Biswas. *Parallel and Adaptive Methods for Hyperbolic Partial Differential Systems.* PhD thesis, Computer Science Dept., Rensselaer Polytechnic Institutue, Troy, 1991.

[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 Applied Aerodynamics Conference*, no. 94-1888, Colorado Springs, 1994.

[10] 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 B. K. Szymanski and B. Sinharoy, eds., *Proc. Third Workshop on Languages, Compilers, and Runtime Systems*, Troy, pp. 265–277, 1996.

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

[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.*, 84:90–113, 1989.

[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.*, 52:411–435, 1989.

[14] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.

[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.*, 16:157–182, 1994.

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

[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*, vol. 75, pp. 215–242, Springer-Verlag, Berlin-Heidelberg, 1995.

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

[19] 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.

[20] 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.*, 38:989–1000, 1995.

[21] 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.

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

[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] Z. Johan, K. Mathur, and S. L. Johnsson. An efficient communication strategy for finite element methods on the Connection Machine CM-5 system. Technical Report 256, Thinking Machines Corporation, 1993.

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

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

[28] 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.

[29] T. Minyard, Y. Kallinderis, and K. Schulz. Parallel load balancing for dynamic execution environments. In *Proc. 34th Aerospace Sciences Meeting and Exhibit*, Reno, no. 96-0295, 1996.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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, pp. 6.1–6.49. Agard, Neuilly-Sur-Seine, 1995.

[36] 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.

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

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

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

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

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

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

[43] 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.

[44] 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*, Kihei, pp. 463–472, 1994.

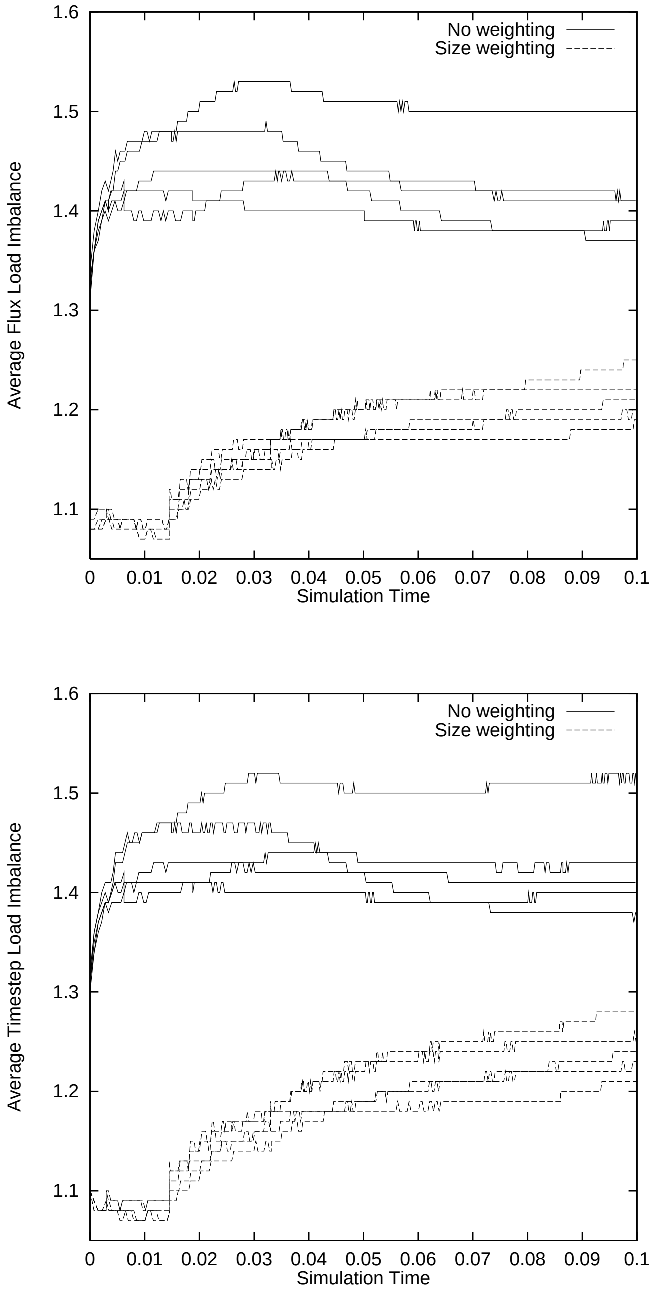
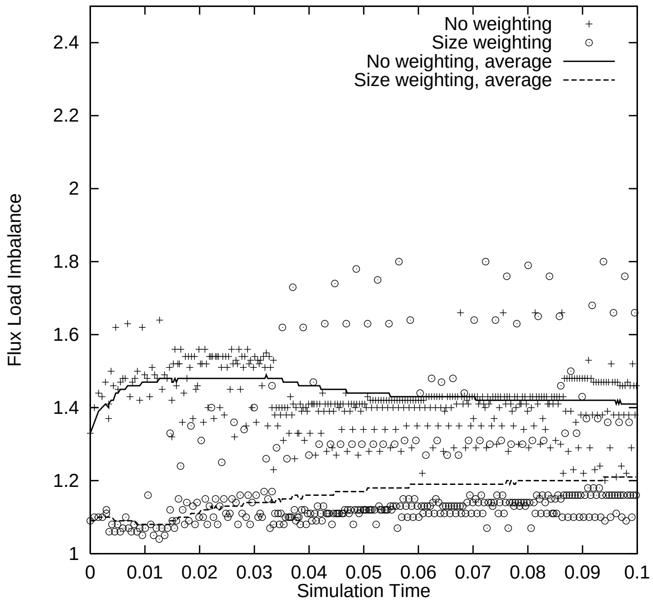


Figure 3: Average flux (top) and time-step (bottom) load imbalances for a sequence of five runs with and without size weighting.



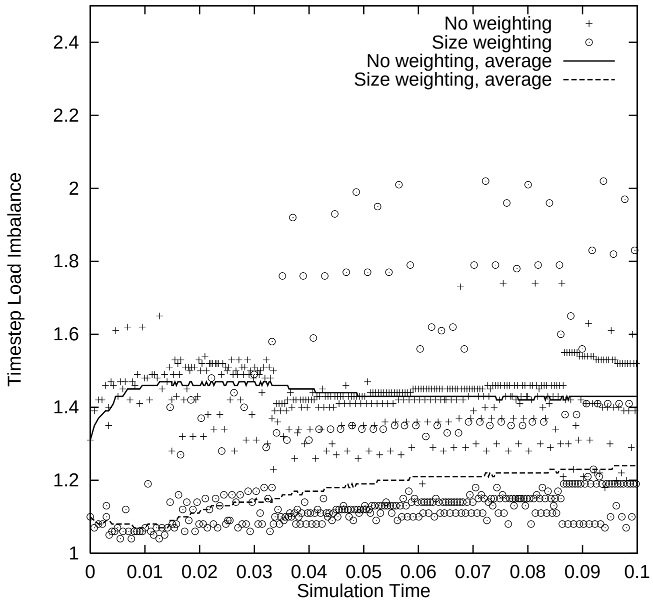


Figure 4: Flux (top) and time-step (bottom) load imbalances during a representative run as shown in Figure 3. Scatter data show the actual load imbalance as a function of simulation time. The curves show average load imbalance.

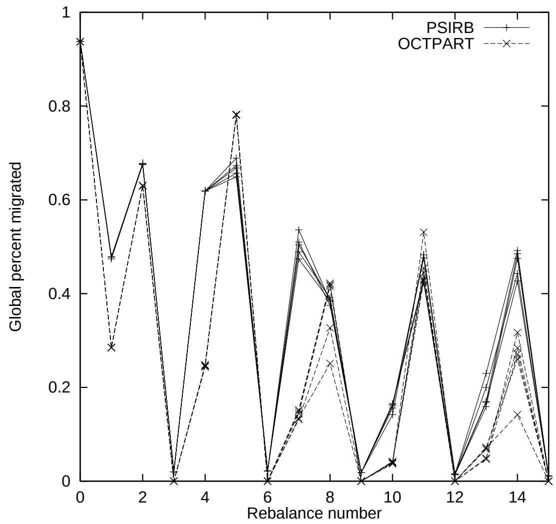


Figure 5: Percentages of elements moved during balancing by PSIRB and OCTPART for a sequence of five runs.

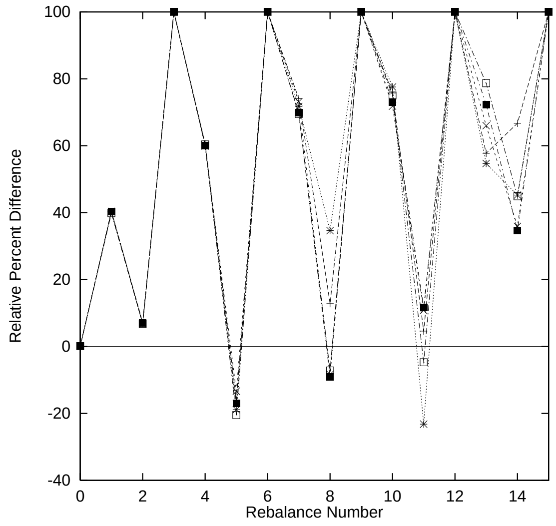


Figure 6: Relative percent differences m_{diff} in data migration of OCTPART compared to PSIRB for a sequence of five runs.

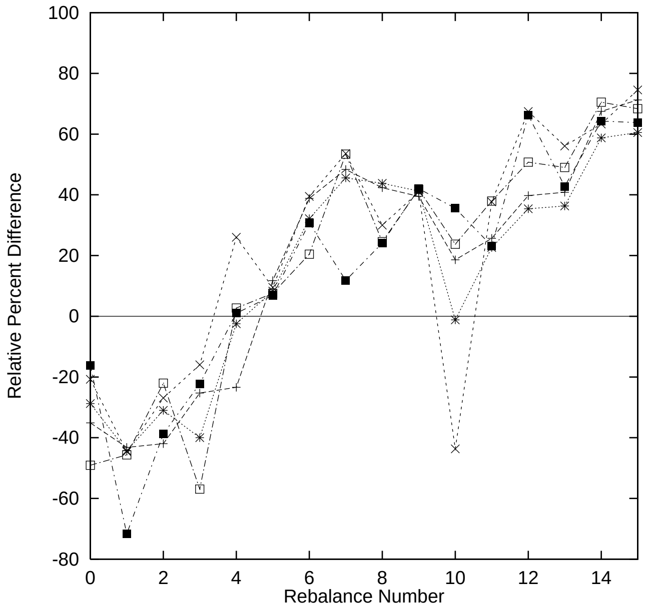


Figure 7: Relative percent differences in PSIRB and OCTPART repartitioning times for a sequence of five runs.

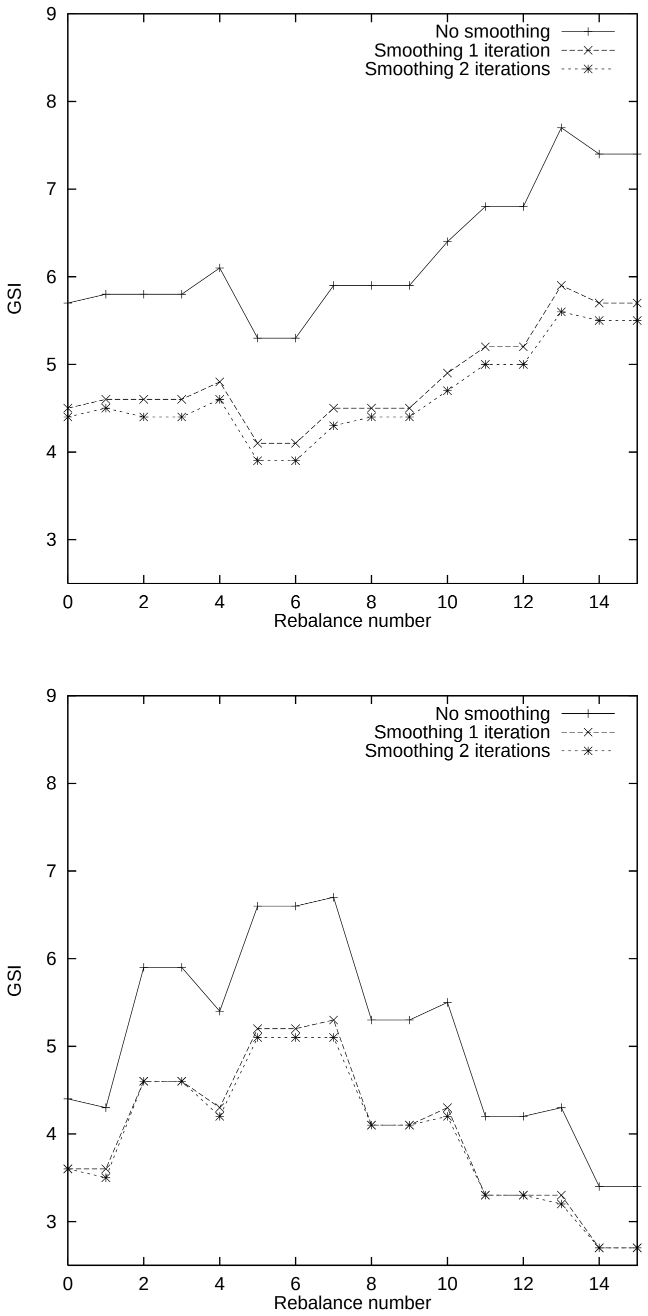
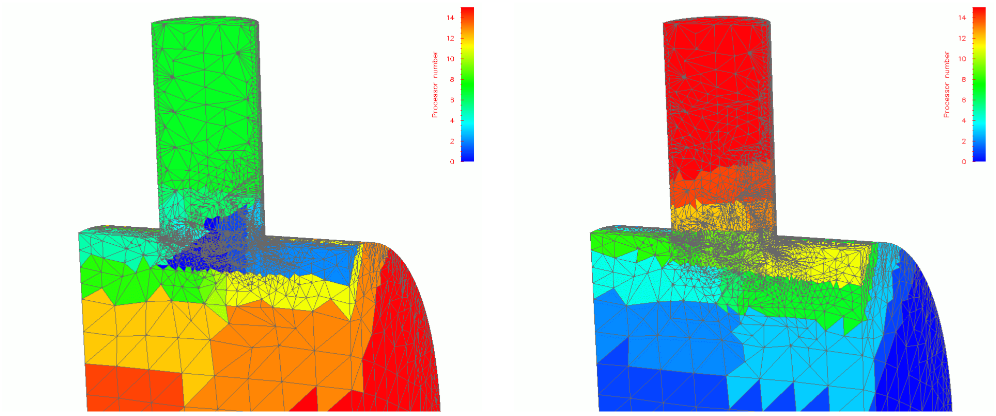


Figure 8: Global surface indices (GSIs) before and after smoothing for OCTPART (top) and PSIRB (bottom).



(a) (b)

Figure 9: Partitioning of consecutive meshes with PSIRB. Minor mesh modifications to (a) have shifted the principal axis of inertia, resulting in nearly a total redistribution of data (b).

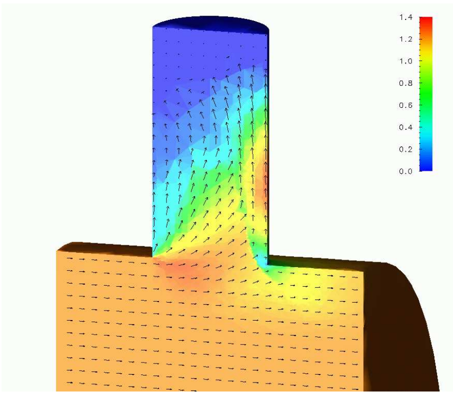


Figure 10: Projection of the Mach number and velocity vectors onto the surface of a perforated cylinder using global time stepping at $t = 0.3$.

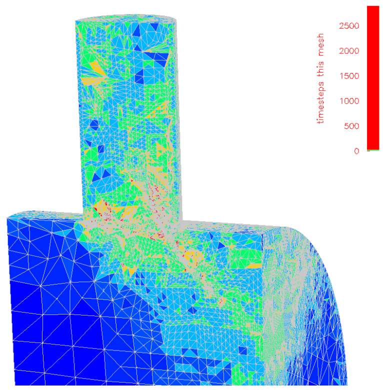


Figure 11: Number of local time steps taken on the mesh at $t = 1.05$ of the run in Figure 12.

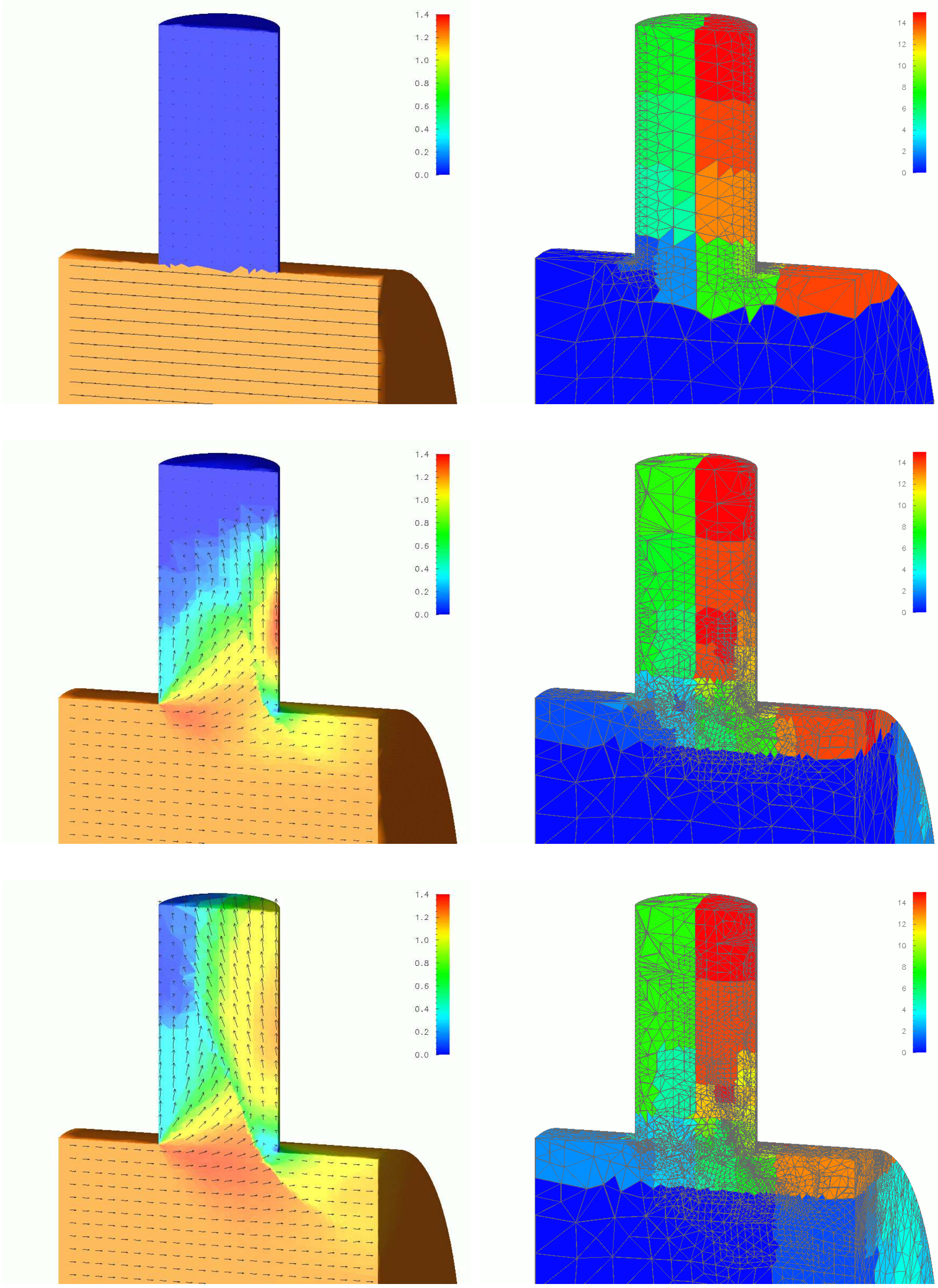


Figure 12: Projections of the Mach number and velocity vectors (left) and mesh and partitioning (right) onto the surfaces of a perforated cylinder at times 0 (top), 0.3 (center), and 0.6 (bottom).