Dynamic Octree Load Balancing Using Space-Filling Curves*

P. M. Campbell
IBM Microelectronics Division
2070 Route 52
Hopewell Junction, NY 12533

K. D. Devine
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1111

J. E. Flaherty, L. G. Gervasio
Scientific Computation Research Center
Rensselaer Polytechnic Institute
Troy, NY 12180

J. D. Teresco†
Department of Computer Science
Williams College
Williamstown, MA 01267

January 31, 2003

Abstract

The Zoltan dynamic load balancing library provides applications with a reusable object oriented interface to several load balancing techniques, including coordinate bisection, octree/space filling curve methods, and multilevel graph partitioners. We describe enhancements to Zoltan's octree load balancing procedure and its distributed structures that improve performance of the space filling curve (SFC) traversals by

*Williams College Department of Computer Science Technical Report CS-03-01
†Corresponding author. E-mail: terescoj@cs.williams.edu, Tel: (413)597-4251, Fax: (413)597-4250

exploiting similarities between the octree and SFC construction. The SFC implementation includes efficient Morton, Gray code, and Hilbert tree traversals. We present the results of a number of scalability and partition quality studies utilizing the new octree structures and orderings.

1 Introduction

Adaptive computational techniques provide a reliable, robust, and efficient means of solving problems involving partial differential equations (PDEs) by finite difference, finite volume, or finite element technologies [11]. With an adaptive approach, an initial mesh used to discretize the computational domain and numerical method used to discretize the PDEs are enhanced during the course of the solution procedure in order to optimize, *e.g.*, the computational effort for a given level of accuracy. Enhancement typically involves h-refinement [48], where a mesh is refined or coarsened, respectively, in regions of low or high accuracy; r-refinement [1, 2], where a mesh of a fixed topology is moved to follow evolving dynamic phenomena; and p-refinement [3, 46], where the method order is increased or decreased, respectively, in regions of low or high accuracy. Unfortunately, parallelism greatly complicates an adaptive computation. Domain decomposition, data management, and interprocessor communication must be dynamic since adaptive h- and p-refinement alter existing patterns. The dynamic data structures used with adaptive software severely limit automatic parallel optimization.

The *Zoltan* object-oriented, dynamic, load balancing library [12] contains several geometric and graph-theoretical algorithms that isolate load balancing from an application and permit run-time selection of procedures such as coordinate bisection [6, 16, 48], octree [13, 20, 21, 32, 33] and space-filling curve methods [4, 38, 42], refinement tree methods [34], and multilevel graph partitioning [30, 31, 52, 56]. Our interest is the Zoltan octree load balancing procedure, which was originally developed [20, 21, 32, 33, 38] to exploit the hierarchical structures found in both adaptive h-refinement and octree mesh generation [49].

Octree load balancing procedures can provide high-quality and efficient partitions for large-scale scientific applications [20, 21, 32]. They produce small changes to the partition when small imbalances occur [20], which is important since migration is often the most significant performance factor [20]. Gervasio [25] incorporated octree load balancing into the Zoltan library, and utilized space-filling curves [44] with Morton, Gray code, and Hilbert orderings for tree traversals. Unfortunately, the Hilbert ordering was far less efficient than the other two. Herein, we exploit similarities between the octree and space-filling curve construction to create mappings that eliminate any performance differences between the three traversal strategies.

We begin by describing Zoltan's octree distributed octree structures and associated load balancing procedure (§2.1 – §2.4). The various space-filling curve traversals (§2.5) and their efficient implementation (§2.6) follow. A sequence of fixed-mesh and adaptive computations appraise the performance of octree balancing with the various space-filling curve traversals (§3). We conclude with a discussion of our findings and suggested future directions (§4).

2 Octree Partitioning

2.1 Basic Structure

Octree partitioning was motivated by octree-based mesh generation [49], where a problem domain is embedded in a cubic universe that is recursively subdivided into eight *octants* wherever more resolution is required to produce an octree structure [50]. The universe is represented by the *root octant*. Each octant is either a *parent* (or *interior*) *octant*, with exactly eight *children*, or a *leaf* (or *terminal*) *octant* with no children.

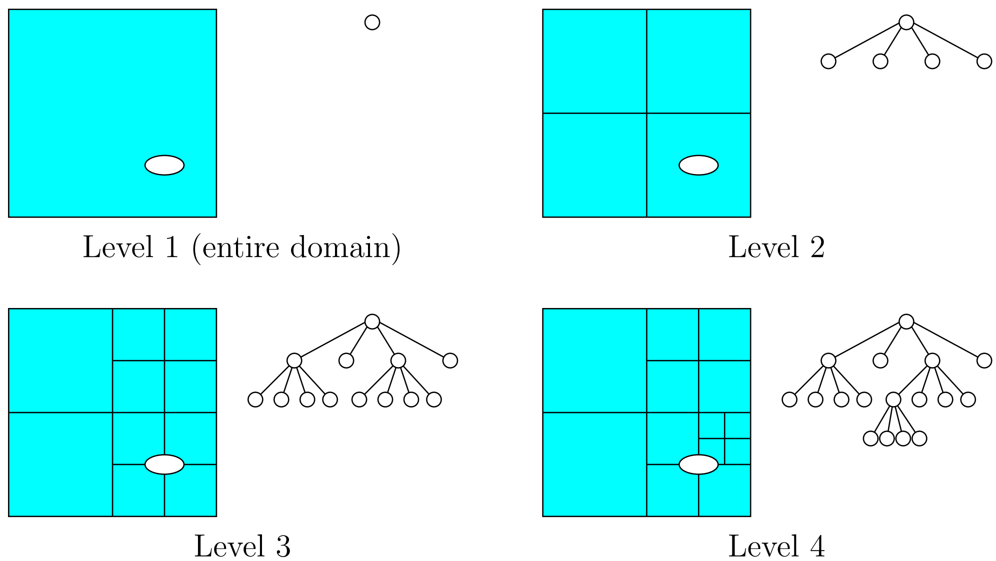


Figure 1: The construction of a quadtree for a square domain with a small hole. At Level 1, only the root quadrant exists. At Level 2, one refinement (bisection) has occurred. Level 3 shows refinement of two of the Level 2 quadrants, and Level 4 shows refinement of one Level 3 quadrant.

Figure 1 shows a quadtree, the two-dimensional analog of an octree, covering a square domain with a small hole. At each level, we show the tree represented as an overlay of the domain and a more traditional tree structure. Octant *refinement* is the replacement of a terminal octant by an interior octant with eight new terminal octants as children, allowing for a greater resolution (a deeper subtree) in parts of the domain. For automatic mesh generation, the amount and location of tree refinement is often controlled by domain features and user-specified tolerances [24]. Similarly, *coarsening* replaces an octant whose children are terminal octants by a single terminal octant.

Meshes of tetrahedral elements may be generated from the octree by using templates that subdivide terminal octants into tetrahedra to provide a direct relation between the elements and octants. Figure 2 shows an example of a 40-element triangular mesh generated from the quadtree of Figure 1.

Octree structures may be constructed for meshes generated by other procedures or created by adaptive refinement by associating an element with the octant containing its centroid.

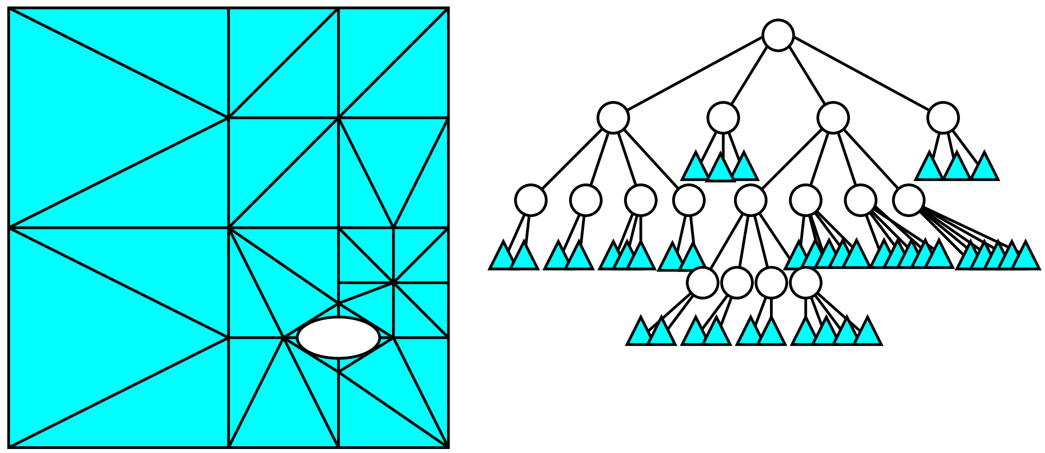
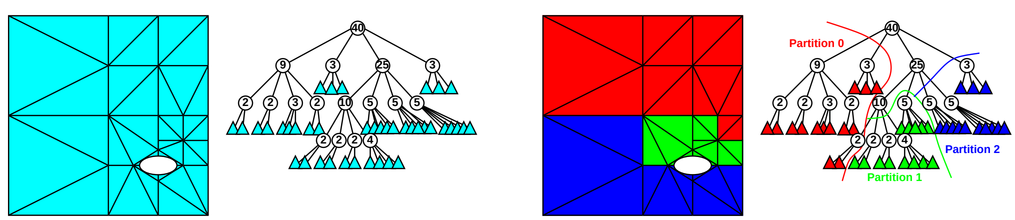


Figure 2: Triangular 40-element mesh generated from the quadtree of Figure 1, and association of mesh entities with leaf quadrants.

Thus, the entire domain and mesh are embedded in a cubic universe. Each element is inserted into the octant of the tree containing its centroid. If the number of elements assigned to an octant exceeds a prescribed tolerance, the octant is refined and its elements are distributed to the appropriate offspring. The granularity of the tree should be fine enough to allow for a good balance, since all elements in a terminal octant will be assigned to the same partition, yet coarse enough to remain an order of magnitude smaller than the elements being partitioned, for efficiency. In practice, octants are refined when they exceed a prescribed number of elements, which is typically set at 40. Other spatially-distributed objects (*e.g.*, particles) may similarly be associated with octants and, thus, are amenable to octree partitioning.

2.2 Serial Partitioning



Assign weights (DFT), OPS=40/3   Partition (truncated DFT)

Figure 3: Three-way quadtree partitioning of the mesh of Figure 2. Appropriate weights are assigned to subtrees according to cost function (*e.g.*, number of elements) during a depth-first traversal (DFT, left). A second truncated DFT assigns subtrees to the three partitions (right).

A depth-first traversal (DFT) of the octree determines all subtree "costs." In the simplest case, cost can be the number of elements in a sub-tree. With p-refinement, the cost would be a function of the total degrees of freedom associated with a sub-tree. For a local refinement method [20], elemental costs could be the inverse of an element's size to reflect the extra work involved in time stepping smaller elements more frequently than larger ones. Assuming that the costs are the number of elements in a subtree, the upper left portion of Figure 3 shows the result of this process for the mesh of Figure 2. Since the total cost (TC) of the octree and the number of partitions (NP) are known, the optimal partition size (OPS) is

$$OPS = TC/NP. \tag{1}$$

A second (truncated) DFT of the octree adds octants to the current partition if their inclusion does not exceed OPS. If adding an octant's entire subtree exceeds OPS, the traversal 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 under-filled, and start work on the next partition. This decision is based on the relative level of imbalance and the cumulative cost of closed partitions to avoid a very large final partition. Three-way partitioning of the mesh of Figure 2 is shown on the right of Figure 3.

This second DFT defines a one-dimensional ordering of the leaf octants, which is divided into segments corresponding to the partitioning. Members of any given segment tend to be spatially adjacent and, thus, form a good partition. For simplicity, we have assumed that each partition should contain (approximately) the same cost, but the procedure has been generalized to produce weighted partitions, appropriate in the presence of, *e.g.*, heterogeneous processing nodes [18]. We further assume that the number of processes and the number of partitions are equal, but the procedure is being modified to produce k-way partitions with any number of cooperating processes [18].

2.3 Distributed Octree Structures

For scalability, an octree structure used for dynamic repartitioning must be distributed across the cooperating processes [51]. It also must be constructed automatically in parallel.

Each octant maintains information about its region of space (bounding box), process ownership, parent and offspring links, and attached objects and their costs for weighted load balancing. In a distributed tree, links may cross process boundaries, and must include both a process id and a pointer. A distributed tree also increases the complexity and overhead of interprocess communication, octant refinement and pruning, and the insertion of new objects (*e.g.*, elements created or removed by adaptive h-refinement) into the correct octant.

2.3.1 Automatic Tree Construction

Initially, we calculate a bounding box for the entire domain by examining the centroid of each object to be inserted into the tree. This initial object traversal may be avoided by specifying the bounding box through a global bounds function [9]. A root node, representing the entire domain, is created on each process. Each of the NP processes refines this root octant in parallel to the initial refinement level

$$IRL = \lceil \log_8(NP) \rceil \tag{2}$$

to create $8^{(IRL)}$ leaf octants. For a quadtree,

$$IRL = \lceil \log_4(NP) \rceil, \tag{3}$$

which produces one level of refinement for our three-process example (Figure 3).

This produces an identical *global octree* on each process. This global octree, while small relative to the full tree to be generated, simplifies and improves the efficiency of traversal and object insertion. Each terminal octant in the global octree is called a *global octant*. While global octants are replicated on all processes, each is assigned a unique and permanent owner as the process that initially contains its portion of the spatial domain. A *map array* represents the entire global octree as a linear representation of the global octants. Once the map array has been created, the top levels of the tree are no longer needed. The global tree structure and the map array are shown in Figure 4 for the example of Figure 3.

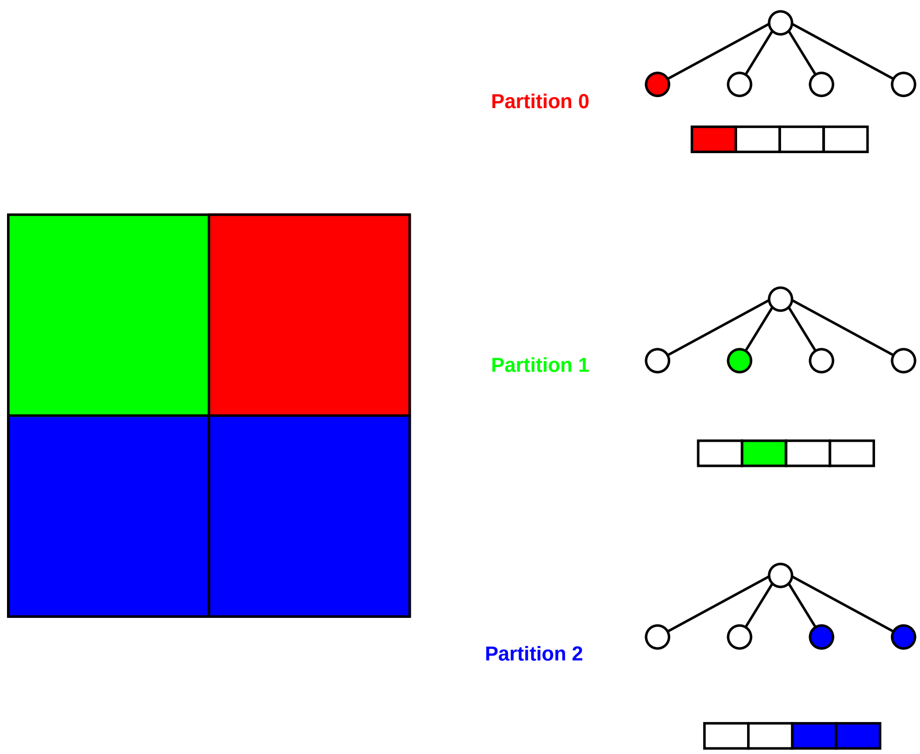


Figure 4: Three-way partitioning of leaf nodes of the global tree for the example of Figure 3. Extra nodes are assigned to higher-numbered processes. Each process has a map array entry for each global quadrant.

Next, each process calculates its range of leaf octants by a DFT. If the number of processes does not evenly divide the number of leaf octants, excess octants are placed in the higher-numbered partitions, with at most one extra octant assigned per partition (Figure 4). This imbalance is countered by placing objects lying on octant boundaries in the lower-numbered octants.

The global octants assigned to each process are initially placed into a *local root list*. The local roots are the roots of the local subtrees that exist on each process. They are maintained in their DFT order.

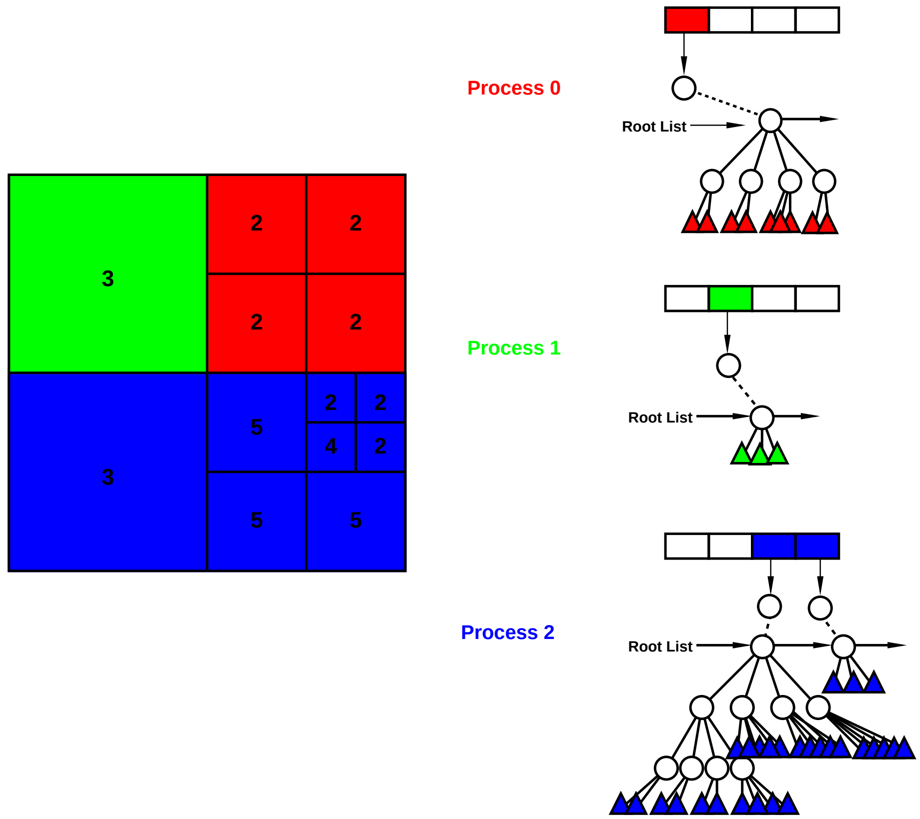


Figure 5: Object insertion and tree refinement. Objects (representing mesh elements) are inserted into the tree. Octants are refined as needed, and are labelled with their final object counts (left). For each process, the map array, its local root list, and local subtrees are shown (right).

Next, the objects to be partitioned are inserted into the octree. Parts of the tree will be refined when inserted objects exceed the prescribed limit on the number of objects per leaf octant (§2.1). The tree resulting from object insertion for the example of Figure 3 with a refinement limit of five elements per quadrant is shown in Figure 5.

Objects to be inserted may reside on any process, and some objects will likely reside on processes other than those owning their destination octants. Such objects are called *orphans* and must be migrated to the appropriate process, as determined by a $O(\log(NP))$ search using the map array.

2.4 Repartitioning Algorithm

After object insertion, each process computes costs for each locally rooted subtree using traversals within its domain with no interprocess communication. Prefix and global costs are computed from the per-process cost totals, enabling each process to determine its position in the global traversal.

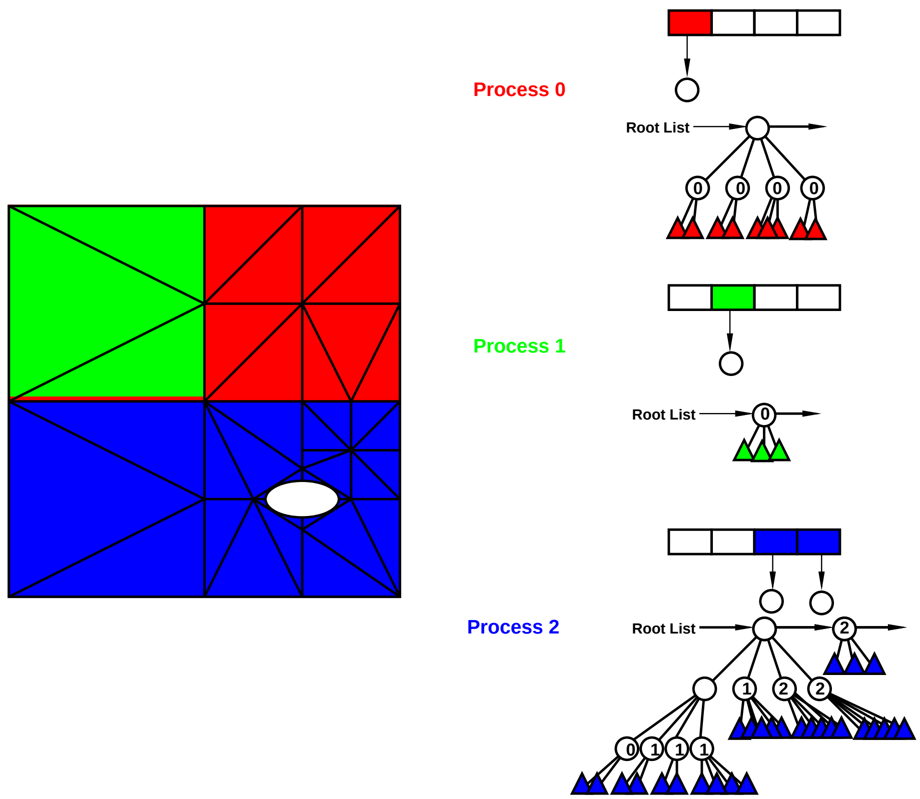


Figure 6: Octants and associated mesh elements after determination of their destination processes for rebalancing. The numbers indicate the destination processes for each octant.

As with the serial procedure, each process traverses its subtrees to create partitions. Each process determines its initial destination partition as

$$IDP = OPS/PC \tag{4}$$

where PC is the prefix cost. Each process traverses its subtrees with no interprocess communication. A load counter computes the sum of the costs of octants assigned to the current destination process (CDP). Once the load counter exceeds the OPS, the CDP is incremented, the load counter is reset, and traversal continues. All remaining octants are assigned to the last partition even if its load exceeds OPS (Figure 6).

When the traversals are complete, subtrees and their associated data are migrated, if necessary, to their assigned destination process. Octant migration occurs in three stages. First, octants migrating to the same process are collected and the destination process is notified. The destination processes allocate space for the arriving data and notify the source processes of their new addresses to update migrating octants' remote parent or offspring links. Finally, the updated octants are sent to their destination and are removed locally. This strategy preserves the octant traversal order. Applications are responsible for object (*e.g.*, mesh element) migration as directed by Zoltan's migration arrays to maintain flexibility.

Local roots are updated and communicated to every process. Each root octant is added to the remote octant list in the map array element corresponding to the octant's global

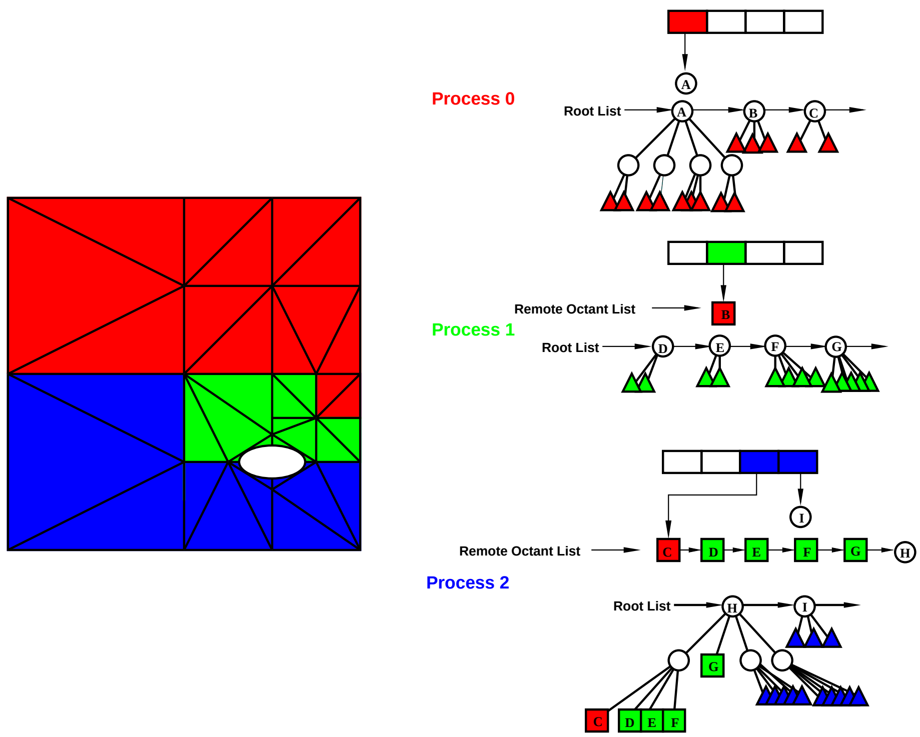


Figure 7: Distributed tree and mesh following quadrant and object migration. The structure has become more complex, with each process having a larger local root list. Octants labeled with the same letter represent the same octant. The map array on processes 1 and 2 includes a remote octant list. Process 0's original tree (A) and Process 2's second child (I) remain local, so these map array entries do not have a remote octant list.

octree ancestor. Figure 7 shows the final partitioning and distributed tree structure for our two-dimensional example.

Remote octant lists are limited in length by the total number of local root octants, which is small relative to the size of the octree. The total number of "cuts" in the global ordering is limited by the total number of partitions. Remote octants are introduced only when one of these cuts separates an octant from its parent or siblings.

2.5 Space-Filling Curve Traversals

We have been using the generic term "traversal" to indicate an ordering, or linearization, of the leaf octants of the octree. Since partitions are formed from contiguous segments of this linearization, its form has a direct effect on the quality of the resulting partitions. Space-filling curves provide a continuous mapping from one-dimensional to d-dimensional space [44] that have been used to linearize spatially-distributed data for partitioning [4, 38, 39, 42], storage and memory management [10, 35], and computational geometry [5]. Herein, we regard the space-filling curves as a way of organizing the octree traversals and, hence, linearizing the leaf octants of the distributed octree.

Space-filling curves have many properties that make them useful for octree traversal [38, 39]. They are self similar and are typically constructed from a single stencil. As constructed, the space-filling curve will visit each terminal quadrant or octant exactly once. Spatial refinement is accompanied by a localized repetition of the stencil, subject to spatial rotations and reflections. Thus, upon refinement, the space-filling curve is modified to visit each octant offspring in place of its parent. By their construction, the space-filling curves preserve a locality of the mapping onto the d-dimensional hyperspace. Thus, points that are close in hyperspace are typically close on the curve.

A space-filling curve may be constructed by using a string-rewriting rule [43], which we do by (*i*) constructing a template curve that represents one unit of refinement in a subdivision of the domain, (*ii*) constructing transformations to project the curve to higher levels of refinement, and (*iii*) determining the final curve by recursive applications of the rewriting rule (§2.6). The traversal ordering and, ultimately, the partitioning are determined by reading the string from beginning to end. Three orderings are considered: Morton (§2.5.1), Gray code (§2.5.2), and Hilbert (§2.5.3)

2.5.1 Morton Ordering

The Morton (Z-code or Peano) ordering [36, 37], originally used in our octree partitioning software [20], is a simple space-filling curve that traverses a quadrant's children in a z-like pattern in the order: I, II, III, IV (Figure 8). The pattern at each refinement is identical to that used by its ancestors, so no rotations or reflections are performed. Without these, there can be large "jumps" in its linearization, particularly as the curve transitions from quadrant II to quadrant III. The jumps result in spatially distant parts of the domain being adjacent in the linearization. Nevertheless, we shall see that the Morton ordering is competitive (§3) and, because of its simplicity, provides a base ordering for all space-filling curves (§2.6).

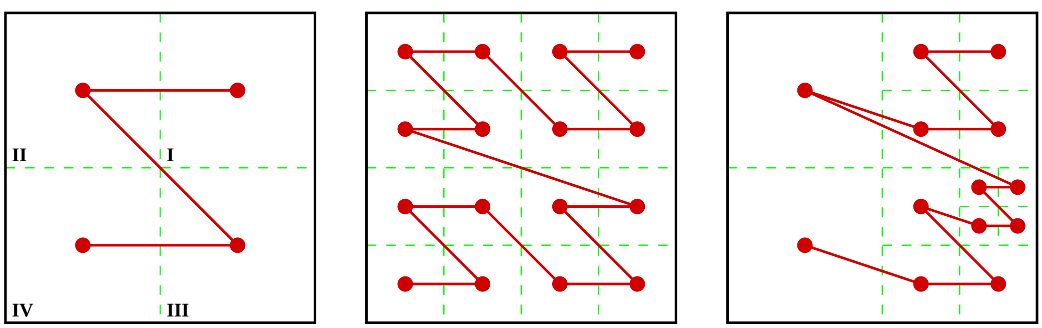


Figure 8: Template curve for the two-dimensional Morton ordering (left), its first level of refinement (center), and an adaptive refinement (right).

The three-dimensional version of the Morton ordering appears in Figure 9. The template consists of two "Z" curves, with the end of the first connected to the start of the second. The "jumps" are even more apparent in three dimensions.

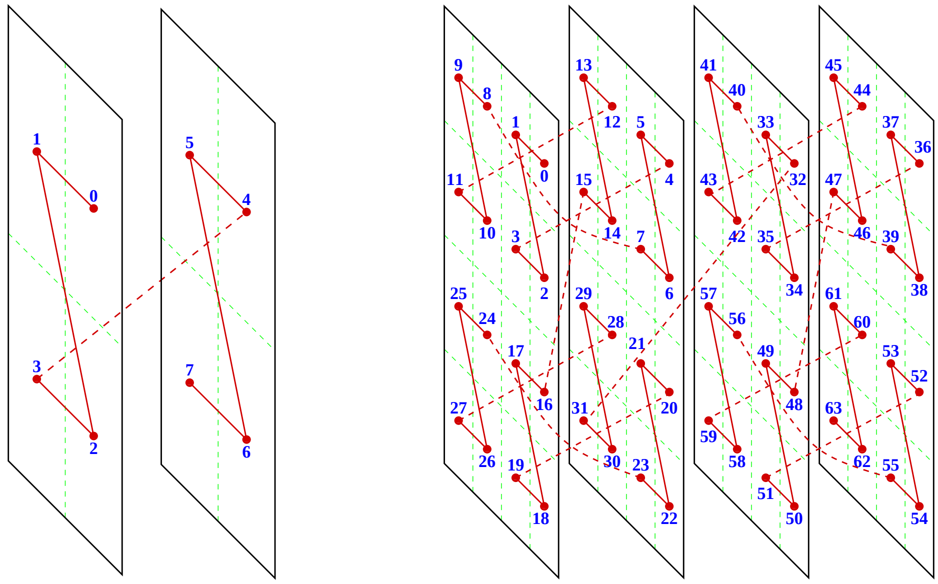


Figure 9: Template curve for the three-dimensional Morton ordering (left), and its first level of refinement (right). The numbers indicate the traversal order of leaf octants.

2.5.2 Gray Code Ordering

With Gray code ordering [15], quadrants are traversed by their "Gray code" sequence [26], where adjacent quadrants differ only by one bit in a binary representation. Letting 00, 01, 10, and 11, correspond to quadrants I, II, III, and IV, respectively, leads to a bracket-like template with quadrants ordered I (00), II (01), IV (11), and III (10). However, the resulting curve is self-intersecting, and not a valid space-filling curve. Adding a reflection to flip the bracket template in two of the quadrants eliminates the self-intersection, producing the modified Gray code ordering shown in Figure 10. While not self-intersecting, the modified gray code ordering still contains jumps that lead to the separation of physically close objects in the space-filling curve.

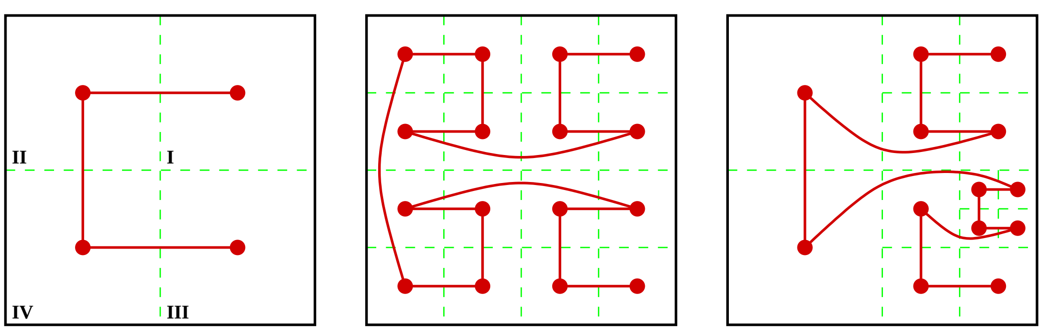


Figure 10: Template curve for the two-dimensional Gray code space filling curve (left), its first level of refinement with rotations to eliminate self intersections (center), and an adaptive refinement (right).

2.5.3 Hilbert Ordering

The Hilbert ordering uses the Peano-Hilbert space-filling curve [7, 40, 41] to order quadrants. It uses the bracket-like template of Gray code ordering, with extra rotations and inversions to keep quadrants closer to their neighbors. It may appear as though this added complexity makes the Hilbert ordering harder to construct, but this will not be the case with our implementation (§2.6). Hilbert orderings appear in Figure 11. The three-dimensional Hilbert template consists of two two-dimensional brackets connected at one endpoint with the ordering of the second bracket being opposite to that of the first (Figure 12).

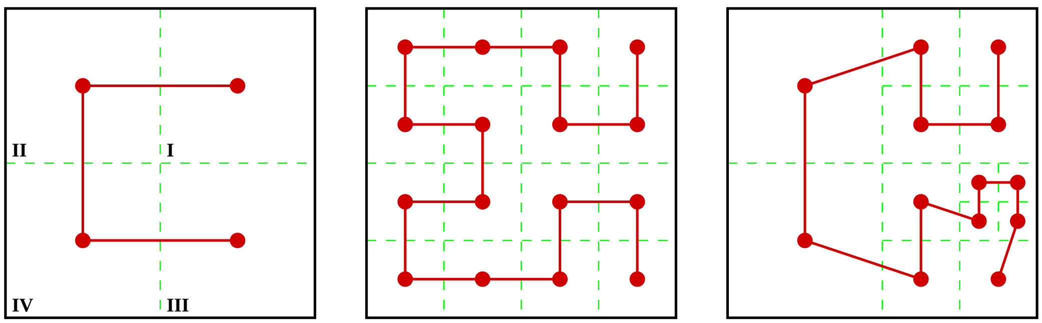


Figure 11: Template curve for the Hilbert ordering (left), its first level of refinement (center), and an adaptive refinement (right).

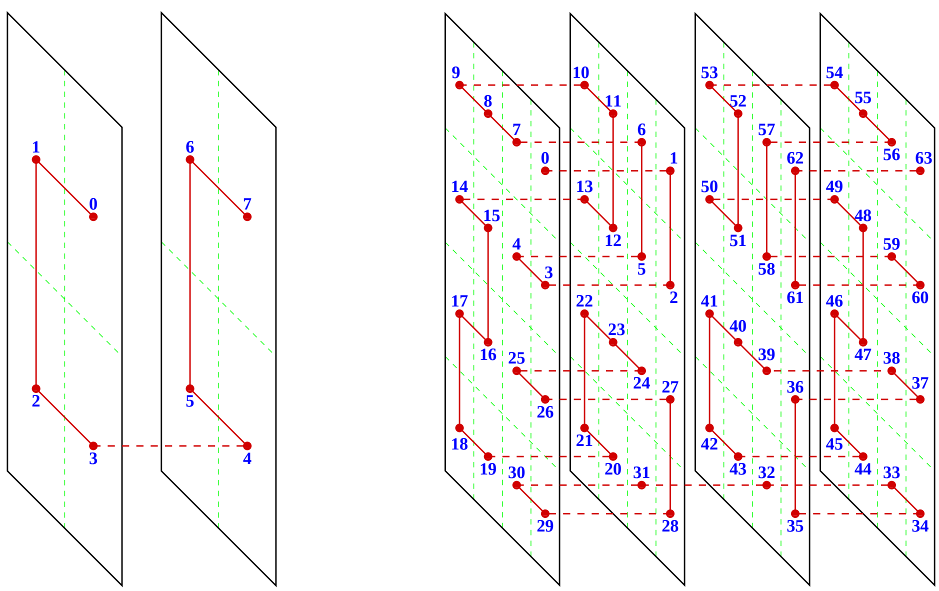


Figure 12: Template curve for the three-dimensional Hilbert ordering (left), and its first level of refinement (right). The numbers indicate the traversal order of leaf octants.

2.6 Octant Ordering Implementation

Fast execution is essential for a successful load balancing algorithm. Since generating the Morton ordering is simple and efficient, we obtain the other orderings from it by providing appropriate mappings. With this technique, there is no appreciable difference in the generation times of the Morton, Gray code, and Hilbert orderings, whereas direct generation of the Hilbert ordering [14] was much slower than the Morton ordering. Our mappings also facilitate the implementation of new orderings, since they only require the specification of "orientation" and "ordering" tables. A similar technique was used to generate the Hilbert curve in the context of spatial databases [29].

The transformations needed for both Gray code and Hilbert ordering are 90° rotations, which allow mappings of fixed size since a series of transformations becomes cyclic. The Gray code ordering requires only two orientations in $\mathbb{R}^2$ and four in $\mathbb{R}^3$. The more complex Hilbert ordering uses four orientations in $\mathbb{R}^2$ and 24 in $\mathbb{R}^3$. The mappings are encoded using ordering and orientation tables of dimension $o_d \times 2^d$, where o_d is the number of unique orientations of the template curve in $d(= 2, 3)$ dimensions. A refining parent with orientation i determines the ordering and orientation of its offspring from row i of each table, $i = 0, 1, \ldots, o_d - 1$. Rows of the ordering table specify the Morton ordering of the offspring and rows of the orientation table specify the orientations to assign to the newly-created offspring. Tables 1, 3, 2, and 4 provide the orderings and orientations of Gray code and Hilbert indexing in two and three dimensions.

Ordering				Orientation			
0	1	3	2	0	1	1	0
3	2	0	1	1	0	0	1

Table 1: Ordering and orientation tables used to map between Gray code and Morton space-filling curve in two dimensions. Row i determines the ordering and orientation of offspring when a parent with orientation i is refined.

Ordering								Orientation							
0	1	3	2	6	7	5	4	0	1	2	3	3	2	1	0
5	4	6	7	3	2	0	1	1	0	3	2	2	3	0	1
3	2	0	1	5	4	6	7	2	3	0	1	1	0	3	2
6	7	5	4	0	1	3	2	3	2	1	0	0	1	2	3

Table 2: Ordering and orientation tables used to map between Gray code and Morton space-filling curve in three dimensions. Row i determines the ordering and orientation of offspring when a parent with orientation i is refined.

For example, consider the initial template and first refinement of the Hilbert curve (Figure 13). The root quadrant has orientation 0, so the offspring at the first level are ordered according to the Morton index sequence given in row 0 of the ordering Table 3: {0 1 3 2}. These offspring are assigned orientations according to row 0 of the orientation Table 3: {1

Ordering				Orientation			
0	1	3	2	1	0	0	2
0	2	3	1	0	1	1	3
3	1	0	2	3	2	2	0
3	2	0	1	2	3	3	1

Table 3: Ordering and orientation tables used to map between Hilbert and Morton space-filling curve in two dimensions. Row i determines the ordering and orientation of offspring when a parent with orientation i is refined.

Ordering								Orientation							
0	1	3	2	6	7	5	4	1	2	0	3	4	0	5	6
0	4	6	2	3	7	5	1	0	7	1	8	5	1	4	9
0	1	5	4	6	7	3	2	15	0	2	22	20	2	19	23
5	1	0	4	6	2	3	7	20	6	3	23	15	3	16	22
3	7	6	2	0	4	5	1	22	13	4	12	11	4	1	20
6	7	3	2	0	1	5	4	11	19	5	20	22	5	0	12
5	1	3	7	6	2	0	4	9	3	6	2	21	6	17	0
0	4	5	1	3	7	6	2	10	1	7	11	12	7	13	14
5	4	0	1	3	2	6	7	12	9	8	14	10	8	18	11
5	4	6	7	3	2	0	1	6	8	9	7	17	9	21	1
0	2	3	1	5	7	6	4	7	15	10	16	13	10	12	17
6	4	0	2	3	1	5	7	5	14	11	9	0	11	22	8
5	7	3	1	0	2	6	4	8	20	12	19	18	12	10	5
3	7	5	1	0	4	6	2	18	4	13	5	8	13	7	19
6	4	5	7	3	1	0	2	17	11	14	1	6	14	23	7
0	2	6	4	5	7	3	1	2	10	15	18	19	15	20	21
6	2	0	4	5	1	3	7	19	17	16	21	2	16	3	18
6	2	3	7	5	1	0	4	14	16	17	15	23	17	6	10
3	2	0	1	5	4	6	7	13	21	18	17	7	18	8	16
6	7	5	4	0	1	3	2	16	5	19	4	3	19	2	13
5	7	6	4	0	2	3	1	3	12	20	13	16	20	15	4
3	2	6	7	5	4	0	1	23	18	21	10	14	21	9	15
3	1	0	2	6	4	5	7	4	23	22	6	1	22	11	3
3	1	5	7	6	4	0	2	21	22	23	0	9	23	14	2

Table 4: Ordering and orientation tables used to map between Hilbert and Morton space-filling curve in three dimensions. Row i determines the ordering and orientation of offspring when a parent with orientation i is refined.

0 0 2}. The next refinement uses this orientation information to determine the order and orientation of their offspring. For example, the orientation in quadrant 0 is 1, so we use row 1 of the tables in Table 3 to determine the offspring order and orientation as {0 2 3 1} and

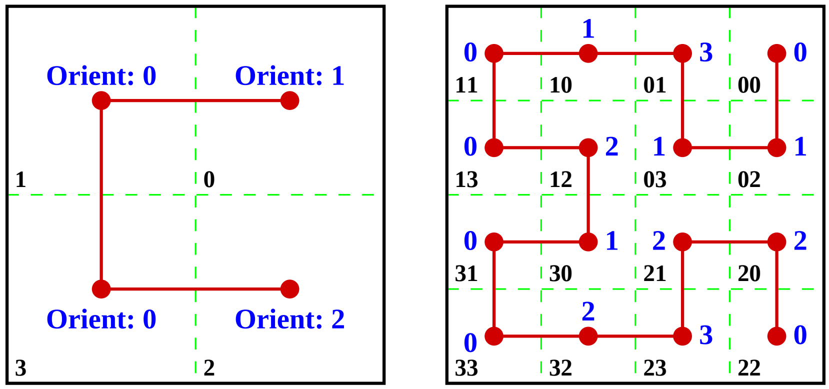


Figure 13: The use of the ordering and orientation tables to generate the first two levels of the two-dimensional Hilbert ordering. The left diagram shows the refinement of the root quadrant, with child ordering {0 1 3 2} and orientations {1 0 0 2} given by row 0 of the tables in Table 3. The right shows the next level of refinement, guided in each quadrant by the row of Tables 3 corresponding to the orientation of the parent in that quadrant.

{0 1 1 3}, respectively.

The ordering and orientation tables define a string rewriting system with the string representing the terminal quadrants or octants that the curve passes through. In the example of Figure 13, the original bracket template is represented by the string {0 1 3 2}. Upon refinement, we "rewrite" the string by replacing each entry by four new entries, each of which consists of the old entry concatenated with an appropriate character from a row of the ordering table. For the refinement shown in Figure 13, the 0 (with orientation 1) is replaced by {00 02 03 01}. Repeating this for the remaining three quadrants gives the string {00 02 03 01 10 11 13 12 30 31 33 32 23 21 20 22} describing the Hilbert curve with one level of refinement. The rewriting applies to adaptively-refined curves as well; thus, the string representing the Hilbert curve on the right of Figure 11 is {00 02 03 01 1 3 23 21 202 201 200 203 22}.

3 Computation

We appraise the performance of octree load balancings using tetrahedral-element meshes arising in applications under investigation. Partition quality (§3.1) and partition execution time are measured for static (fixed-mesh) computation. For successive rebalancings of an adaptive h-refinement computation, we examine partition quality and balancing execution time, as they influence our most important concern: the actual solution time of the computation.

The following four meshes were chosen for our studies to represent a variety of mesh sizes and geometric complexity.

- **Cone**: This is a 42,786-element mesh, used to model a shock wave in a compressible flow impacting the side of the cone. Only half of the cone is modeled [19].

- **Muzzle**: This is a 169,733-element mesh used in the simulation of unsteady compressible flow in perforated shock tube. This mesh is a quarter cylinder with half of a cylindrical venting hole [17].
- **Onera wing**: This is a 85,567-element mesh of the flow about an Onera M6 wing, used in transonic flow simulations [8].
- **Artery**: This is a 1,103,018-element mesh generated from magnetic resonance imaging data of human arteries, used in blood flow simulations [53].

3.1 Partition Quality

The quality of the partitions used to distribute a computation has a significant effect on solution time for many applications [8]. While many factors may be important when considering the "quality" of a partitioning [27], the most common are computational balance and partition boundary size. We will quantify partition quality using three metrics: (*i*) surface index, (*ii*) interprocess connectivity, and (*iii*) intraprocess connectivity. Our examples assign exactly one partition per process.

3.1.1 Surface Indices

Surface indices measure interprocess communication volume. In our examples, they are analogous to surface to volume ratios, with the number of element faces on partition boundaries being viewed as "surface area" relative to the "volume" of element faces in a partition. A large ratio of faces on partition boundaries to total faces indicates a large inter-processor communication volume.

Thus, for NP partitions $P = P_1, P_2, ..., P_{NP}$, let b_i denote the number of partition-boundary faces and f_i denote the total number of faces of P_i. The *maximum local surface index* is

$$r_M = \max_{i=1,...,NP} \frac{b_i}{f_i}. \tag{5}$$

Let b_t denote the total number of boundary faces in all partitions and f_t the total number of faces in all partitions. The *global surface index* is

$$r_G = \frac{b_t}{f_t}, \tag{6}$$

and the *average local surface index* is

$$r_A = \frac{1}{NP} \sum_{i=1}^{NP} \frac{b_i}{f_i}. \tag{7}$$

The average local surface index counts boundary entities once for each partition sharing that entity, while the global surface index counts each boundary entity exactly once. When considering only mesh faces as boundary entities, as in our examples, $r_A = 2r_G$, since a partition boundary face exists in exactly two partitions. Small differences are introduced when considering mesh edges or vertices, which may be shared among three or more partitions.

3.1.2 Interprocess Connectivity

Interprocess connectivity, or interprocess connection density, is the percentage of other processes with which each process must communicate. This correlates to the number of message startups needed during a solution process. This is especially important for an interconnection network with high message latency. Exchanging information with a number of neighboring processes often requires a serialization of message setup, making the maximum interprocess connectivity an important measure of scalability. These measures can also be significant when network topology allows nearest-neighbor communication to be significantly faster than more general communication.

3.1.3 Intraprocess Connectivity

Intraprocess connectivity measures the number of disjoint regions assigned to a given process. Such disconnection, also known as sub-domain splitting [28], can result from repeated repartitioning using some algorithms [8], or from unfortunate cutting planes in an algorithm such as orthogonal coordinate bisection [6] or octree partitioning. With octree partitioning, disjoint regions may also be the result of partitions spanning "jumps" in the traversal ordering (§2.5).

Sub-domain splitting makes the interprocess boundary larger than necessary and can adversely effect the performance of some linear solution procedures [55]. Two regions are face-connected if they share a common face, edge-connected if they share a common edge, and vertex-connected if they share a common vertex. The intraprocess connectivity is computed as the number of disjoint connected components per process for each degree of connectivity. Unlike the other metrics, intraprocess connectivity can be expensive to compute since all mesh regions must be traversed to form the connectivity graphs.

3.1.4 Partition Quality

We present surface index metrics (Figure 14) and interprocess adjacencies (Figure 15) for each sample mesh, using the Morton, Gray code, and Hilbert traversals of the octree. These studies use from 4 (8 for the Artery mesh) to 56 processors of an IBM SP computer at Rensselaer.

The Hilbert ordering generally achieves the best average and maximum surface indices for three of the meshes (Cone, Onera, and Artery), followed in each case by Morton ordering and Gray code ordering (Figure 14). The ordering has little apparent effect on the Muzzle mesh. We will see (§3.2) that, following refinement and repartitioning, Hilbert orderings produce superior surface indices for the Muzzle mesh as well. As expected, surface index measurements are generally ranked according to the discontinuities found in the space-filling curves.

Interprocess adjacencies for the Artery, Cone, and Muzzle meshes are generally lowest for the Hilbert ordering, next for the Morton ordering, and highest for the Gray code ordering (Figure 15). The Artery mesh shows only slight differences between the three orderings as a result of its complex structure and large size that make it difficult for any ordering to show a clear advantage. Interprocess adjacencies for the Onera mesh are ranked in the opposite order (Figure 15). The mesh structure, a flat, compact concentration of fine mesh

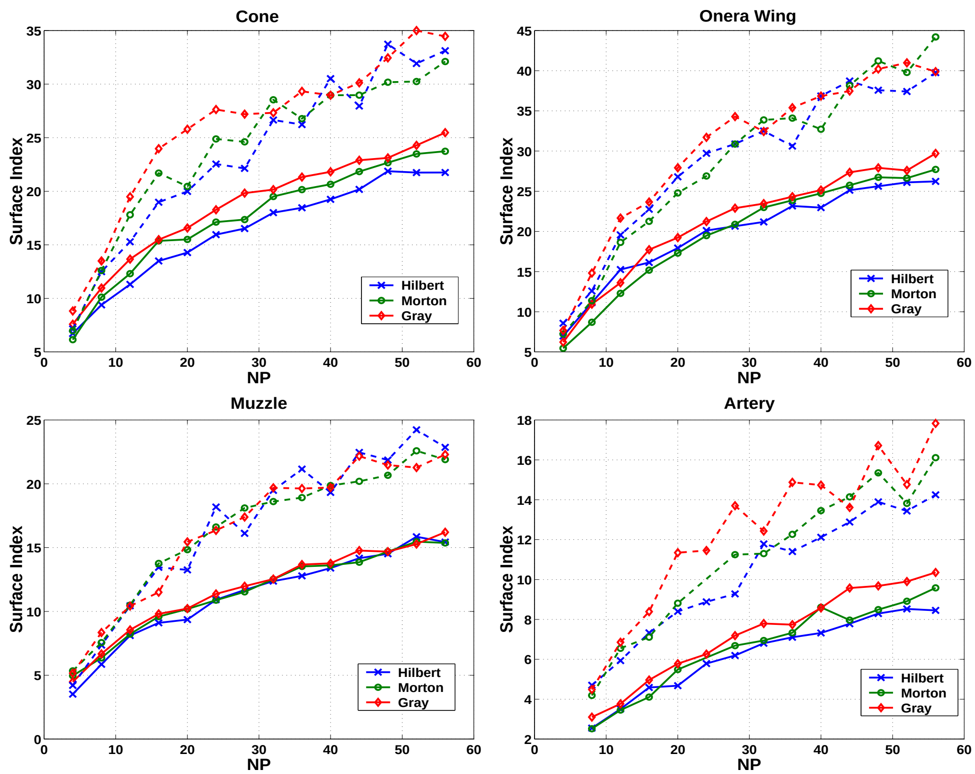


Figure 14: Surface indices vs. number of processes (NP) for Cone (top left), Onera (top right), Muzzle (lower left), and Artery (lower right) meshes. Solid lines show the average surface index while the dashed lines show the maximum surface index.

elements near the wing surface, appears to favor the Gray code and Morton orderings. This underscores the importance of mesh structure on load balancing performance, and shows how no traversal is optimal in all situations.

3.2 Dynamic Load Balancing for an Adaptive Analysis

To illustrate dynamic load balancing performance, we solve a finite element problem for the unsteady compressible flow through a vented cylinder using six adaptive h-refinement steps (and rebalancings) of the Muzzle mesh [17].

Load balancing uses element weights specified as the inverse of the radius of a sphere inscribed in each element to account for the computational imbalance introduced by the LRM.

3.2.1 Partition Quality

Figures 16 and 17 show the partition quality metrics after each rebalancing step for the perforated shock tube problem on 28 and 56 processes, using Morton and Hilbert orderings. (Gray code traversals produced poorer decompositions and were not included.) Surface indices for the 28 (Figure 16) and 56 (Figure 17) process cases are, respectively, marginally

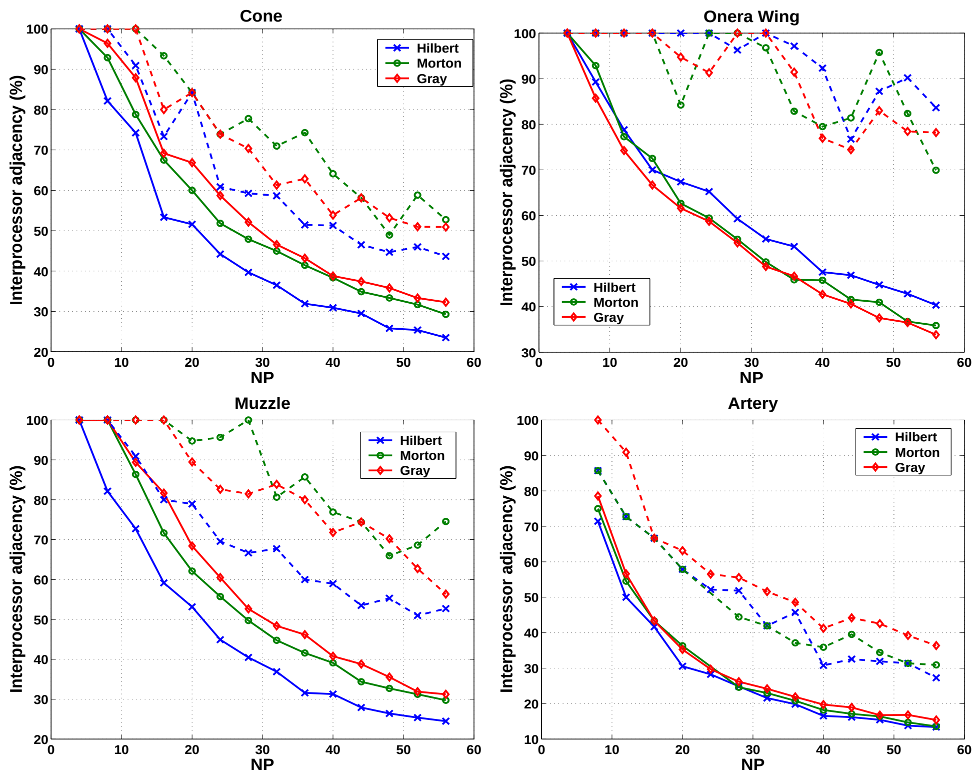


Figure 15: Interprocess adjacency *vs.* number of processes (NP) for Cone (top left), Onera (top right), Muzzle (lower left), and Artery (lower right) meshes. Solid lines indicate average interprocess adjacency and dashed lines the maximum interprocess adjacency.

and approximately 10% better with the Hilbert ordering than with the Morton ordering. In both cases, the Hilbert ordering achieves better interprocess connectivity, with a 25% difference in the 56 process case. Figure 16 shows average and maximum face connectivity measurements and the number of face connected components. No significant difference is seen in face connectivity, and the number of connected components does not change during the computation despite the fact the mesh size increases.

3.2.2 Solution Time

Ultimately, total solution time is the only important measure. Solution times when the perforated shock tube problem was run on 56 processors of an IBM SP computer for eight refinement steps using octree balancing with Hilbert and Morton orderings appear in Figure 18. Times have been averaged over several runs to avoid anomalous effects. The results indicate a 4% decrease in execution time with the Hilbert ordering. The effect is less noticeable with tests on fewer processes. The improved solution time with Hilbert ordering is primarily due to the lower interprocess connectivity and, to a lesser degree, the improved surface index. While both represent a measure of communication cost, interprocess connectivity corresponds more closely to message latency cost, which is a significant factor on the IBM SP computer. With more processes and refinement steps the difference in solution time

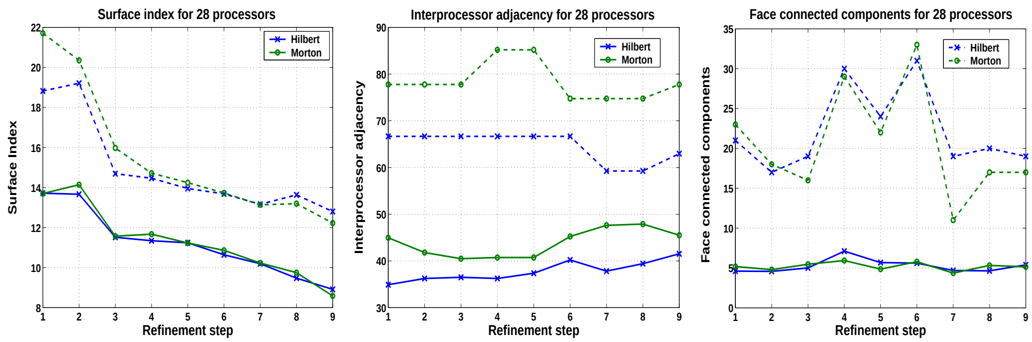


Figure 16: Surface index, interprocess adjacency, and face connectivity for 9 rebalancing steps on 28 processes. Solid lines show the average local and dashed lines show maximum surface index (left), interprocess adjacency (center), and face connectivity (right).

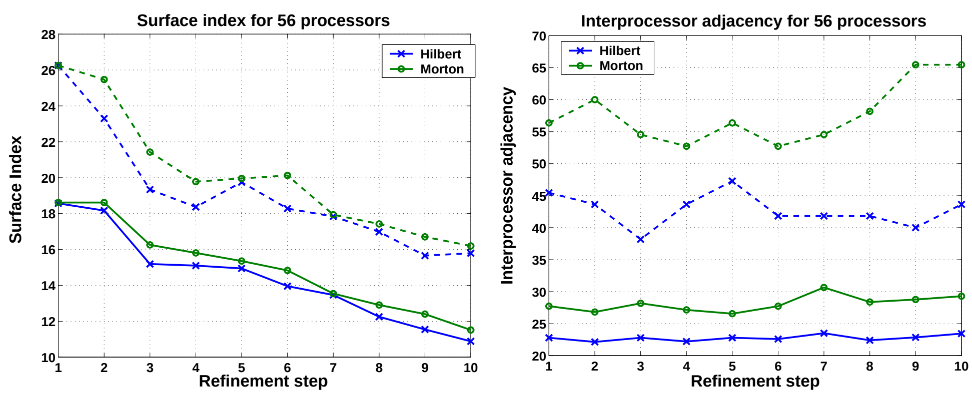


Figure 17: Surface index and interprocess adjacency for 10 rebalancing steps on 56 processes. Solid lines show the average local and dashed lines show maximum local surface index (left) and interprocess adjacency (right).

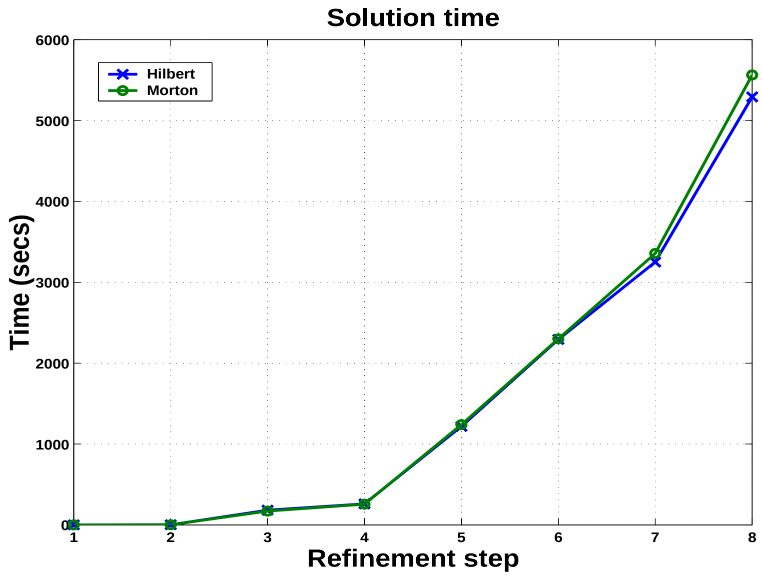


Figure 18: Wall clock solution time for Hilbert and Morton orderings for 56 processes of Rensselaer's IBM SP computer for 8 refinement steps of the perforated shock tube problem.

would be a more significant factor.

4 Discussion

The new space-filling curve traversals in the octree load balancer, in particular the Hilbert ordering, provide better partition quality, measured by surface index and interprocess connectivity. Our new implementation of the traversals makes Hilbert ordering efficient, and a viable option for use in real problems, as both Hilbert and Gray code ordering are as inexpensive as Morton. Hilbert ordering generally achieves the best resulting partitions, likely due to the continuous nature of the Hilbert curve. In general, the quality metrics for the traversals are ranked by the discontinuities evident in the orderings, with Hilbert being the best, followed by Morton and Gray code. The extent of these differences are highly dependent on the mesh structure, and the rankings were different in some cases. It is not clear that one ordering should be used in all circumstances but as a general rule, Hilbert is a good choice. This is consistent with studies that found the Hilbert ordering beneficial in other contexts [5, 29, 35, 42]. Similar studies for other types of problems are necessary to make a stronger statement comparing the space-filling curves.

The octree structures described are designed to allow dynamic updating between successive load balancing steps. Early experiments with these structures showed them to be quite useful in reducing balancer startup costs and the total amount of communication needed. However, recent enhancements to Zoltan allow a more efficient implementation of the "recreated tree" version of the octree load balancer, which minimizes or even eliminates the benefit of the maintained structures. We believe that maintaining load-balancing related structures in the octree load balancer as well as other Zoltan load balancers must be investigated, especially when they are being used for applications where adaptive steps are frequent but produce only incremental changes to the mesh, and hence, the partitioning. Maintaining the octree has the additional advantage that it can be used for other purposes, *e.g.*, multi-level preconditionings of iterative linear algebraic procedures [55], adaptive h-refinement [47], localized spatial searches [45], and hierarchical visualization [22, 23].

Additional areas of future work related to octree balancing include predictive load balancing [17] and fast octant neighbor finding [51]. These have been implemented and tested in other applications. Performance of the octree orphan insertion phase could be improved with the addition of neighbor finding links in the octants, which could be used to speed the search for the correct owning octant. Work is underway to handle multiple partitions per process. Multiple partitions can be made to utilize hierarchical partition models such as Rensselaer Partition Model [54]. This may provide better cache utilization and a framework for multi-threading. It could also improve performance on heterogeneous and hierarchical computers. A machine model to allow architecture-dependent load balancing is under development.

Acknowledgments

The authors were supported by contract 15162 with Sandia National Laboratories, a multi-program laboratory operation by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

References

[1] S. Adjerid and J. E. Flaherty. A moving finite element method for time dependent partial with error estimation and refinement. *SIAM J. Numer. Anal*, 23:778–796, 1986.

[2] S. Adjerid and J. E. Flaherty. A moving mesh finite element method with local refinement for parabolic partial differential equations. *Comput. Methods Appl. Mech. Engrg.*, 55:3–26, 1986.

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

[4] S. Aluru and F. Sevilgen. Parallel domain decomposition and load balancing using space-filling curves. In *Proc. International Conference on High-Performance Computing*, pages 230–235, 1997.

[5] J. J. Bartholdi and L. K. Platzman. An $O(n \log n)$ travelling salesman heuristic based on spacefilling curves. *Operation Research Letters*, 1(4):121–125, September 1982.

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

[7] T. Bially. Space-filling curves: their generation and their application to band reduction. *IEEE Trans. Inform. Theory*, IT-15:658–664, Nov. 1969.

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

[9] P. M. Campbell. The performance of an octree load balancer for parallel adaptive finite element computation. Master's thesis, Computer Science Dept., Rensselaer Polytechnic Institute, Troy, NY, 2001.

[10] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive array layouts and fast parallel matrix multiplication. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, 1999.

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

[12] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[13] 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.

[14] H. C. Edwards. *A Parallel Infrastructure for Scalable Adaptive Finite Element Methods and its Application to Least Squares C^∞ Collocation*. PhD thesis, The University of Texas at Austin, May 1997.

[15] C. Faloutsos. Gray codes for partial match and range queries. In *IEEE Trans. Software Eng.*, volume 14, pages 1381–1393, October 1988.

[16] 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.

[17] J. E. Flaherty, M. Dindar, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. An adaptive and parallel framework for partial differential equations. In D. F. Griffiths, D. J. Higham, and G. A. Watson, editors, *Numerical Analysis 1997 (Proc. 17th Dundee Biennial Conf.)*, number 380 in Pitman Research Notes in Mathematics Series, pages 74–90. Addison Wesley Longman, 1998.

[18] J. E. Flaherty, L. G. Gervasio, and J. D. Teresco. A generalized octree load balancer. In preparation, 2002.

[19] 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. *Appl. Numer. Math.*, 26:241–263, 1998.

[20] 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. *J. Parallel Distrib. Comput.*, 47:139–152, 1997.

[21] J. E. Flaherty, R. M. Loy, M. S. Shephard, and J. D. Teresco. Software for the parallel adaptive solution of conservation laws by discontinuous Galerkin methods. In B. Cockburn, G. Karniadakis, and S.-W. Shu, editors, *Discontinous Galerkin Methods Theory, Computation and Applications*, volume 11 of *Lecture Notes in Compuational Science and Engineering*, pages 113–124, Berlin, 2000. Springer.

[22] L. A. Freitag and R. M. Loy. Adaptive, multiresolution visualization of large data sets using a distributed memory octree. In *Proc. Supercomputing '99*, Portland, 1999.

[23] L. A. Freitag and R. M. Loy. Comparison of remove visualization strategies for interactive exploration of large data sets. In *Proc. 2nd Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications, IPDPS2001*, San Francisco, 2001.

[24] R. Garimella, S. Dey, R. Ramamoorthy, M. K. Georges, and M. S. Shephard. Finite octree mesh generation and output options. SCOREC Report #6-1994, Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, 1994.

[25] L. G. Gervasio. Octree load balancing techniques for the dynamic load balancing library. Master's thesis, Computer Science Dept., Rensselaer Polytechnic Institute, Troy, 1998.

[26] E. N. Gilbert. Gray codes and paths on the n-cube. *Bell Systems Technical Journal*, 37:815–826, 1958.

[27] B. Hendrickson. Load balancing fictions, falsehoods and fallacies. *Appl. Math. Modelling*, 25:99–108, 2000.

[28] S.-H. Hsieh, G. H. Paulino, and J. F. Abel. Evaluation of automatic domain partitioning algorithms for parallel finite element analysis. Structural Engineering Report 94-2, School of Civil and Environmental Engineering, Cornell University, Ithaca, 1994.

[29] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proc. ACM SIGMOD*, pages 332–342, 1990.

[30] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scien. Comput.*, 20(1), 1999.

[31] G. Karypis and V. Kumar. Parallel multivelel k-way partitioning scheme for irregular graphs. *SIAM Review*, 41(2):278–300, 1999.

[32] R. M. Loy. *Adaptive Local Refinement with Octree Load-Balancing for the Parallel Solution of Three-Dimensional Conservation Laws.* PhD thesis, Computer Science Dept., Rensselaer Polytechnic Institute, Troy, 1998.

[33] T. Minyard, Y. Kallinderis, and K. Schulz. Parallel load balancing for dynamic execution environments. In *Proc. 34th Aerospace Sciences Meeting and Exhibit*, number 96-0295, Reno, 1996.

[34] W. F. Mitchell. The refinement-tree partition for parallel solution of partial differential equations. *NIST Journal of Research*, 103(4):405–414, 1998.

[35] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the Hilbert space-filling curve. *IEEE Trans. Knowledge and Data Engng.*, 13(1):124–141, January/February 2001.

[36] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., March 1966.

[37] J. A. Orenstein. Spatial query processing in an object-oriented database system. In *Proc. ACM SIGMOD*, pages 326–336, May 1986.

[38] M. Parashar and J. C. Browne. On partitioning dynamic adaptive grid hierarchies. In *Proc. 29th Annual Hawaii International Conference on System Sciences*, volume 1, pages 604–613, Jan. 1996.

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

[40] E. A. Patrick, D. R. Anderson, and F. K. Brechtel. Mapping multidimensional space to one dimension for computer output display. *IEEE Trans. Computers*, C-17(10):949–953, October 1968.

[41] G. Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36:157–160, 1890.

[42] J. R. Pilkington and S. B. Baden. Dynamic partitioning of non-uniform structured workloads with space filling curves. *IEEE Trans. on Parallel and Distributed Systems*, 7(3):288–300, 1996.

[43] G. Rozenberg and A. Salomaa. *The mathematical theory of L systems.* Academic Press, New York, 1980.

[44] H. Sagan. *Space-Filling Curves.* Springer-Verlag, 1994.

[45] H. Samet. *Applications of Spatial Data Structures, Computer Graphics, Image Processing, and GIS.* Addison-Wesley, 1989.

[46] M. S. Shephard, S. Dey, and J. E. Flaherty. A straightforward structure to construct shape functions for variable p-order meshes. *Comp. Meth. in Appl. Mech. and Engng.*, 147:209–233, 1997.

[47] M. S. Shephard, J. E. Flaherty, C. L. Bottasso, H. L. de Cougny, C. Özturan, and M. L. Simone. Parallel automatic adaptive analysis. *Parallel Comput.*, 23:1327–1347, 1997.

[48] 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.

[49] 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.

[50] M. L. Simone. *A distributed octree structure and algorithms for parallel mesh generation.* PhD thesis, Mechanical Engineering Dept., Rensselaer Polytechnic Institute, Troy, 1998.

[51] 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.

[52] A. J. Soper, C. Walshaw, and M. Cross. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph Partitioning. Tech. Rep. 00/IM/58, Comp. Math. Sci., Univ. Greenwich, London SE10 9LS, UK, April 2000.

[53] C. A. Taylor, T. J. R. Hugues, and C. K. Zarins. Finite element modeling of blood flow in arteries. *Comput. Methods Appl. Mech. Engrg.*, 158(1–2):155–196, 1998.

[54] J. D. Teresco. *A Hierarchical Partition Model for Parallel Adaptive Finite Element Computation.* PhD thesis, Computer Science Dept., Rensselaer Polytechnic Institute, Troy, 2000.

[55] W. D. Turner, J. E. Flaherty, S. Dey, and M. S. Shephard. Multilevel preconditioned QMR methods for unstructured mesh computation. *Comput. Methods Appl. Mech. Engrg.*, 149:339–357, 1997.

[56] C. Walshaw and M. Cross. Parallel Optimisation Algorithms for Multilevel Mesh Partitioning. *Parallel Comput.*, 26(12):1635–1660, 2000.