Parallel Linear Octree Meshing with Immersed Surfaces

Jose J. Camata and Alvaro L. G. A. Coutinho
NACAD, High Performance Computer Center
COPPE, Federal University of Rio de Janeiro
{camata,alvaro}@nacad.ufrj.br

Abstract

A parallel octree-based mesh generation method is proposed to create reasonable-quality, geometry-adapted unstructured hexahedral meshes automatically from triangulated surface models. We present algorithms for the construction, 2:1 balancing and meshing large linear octrees on distributed memory machines. Our scheme uses efficient computer graphics algorithms for surface detection, allowing us to represent complex geometries. Isogranular analysis is performed on a variety of input surfaces and demonstrates good scalability. Our implementation is able to execute the 2:1 balancing operations over 4.0 × 10^8 octants on 128 cores in less than 10 seconds per 2 × 10^5 octants/core.

1. Introduction

As the finite element method has become one of the most popular numerical methods to solve various science and engineering problems, mesh generation for complicated geometries has become a major concern [1]. This is particularly important in current petascale supercomputers that are allowing engineers and scientists to solve a wide range of complex, real world problems at scales considered impossible only few years ago. While many solvers have been ported to parallel machines, grid generators have left behind. The need to generate finite element meshes quickly is a common requirement of most fields and it is an inherent requirement of any adaptive process. Several of the current parallel solution methodologies for finite elements are based on some partitioning of a mesh and their mapping onto the target parallel system. The disadvantage of this approach is the fact that the associated partitioning problem is NP-complete. Generating meshes with billions of nodes and elements and to deliver such meshes to processors of such large-scale systems can be unpractical. Thereby, the necessity for development of parallel meshing techniques is well justified.

Parallel meshing algorithms has been an interesting and active research topic; see the annual International Meshing Roundtable series [19]. Several approaches have been developed. Starting in two dimensions, Verhoeven et al. [2] demonstrated the ability to produce parallel unstructured Delaunay meshes across a network of workstations. Topping et al. [3], Laemer et al. [4], Lohner et al. [5], amongst others, have parallelized the advancing front algorithm. Moving to three dimensions, the task becomes more complicated. Chew et al. [6], Chrisochoides et al. [7], Ivanov et al. [10] have parallelized the Delaunay algorithm. Johnson and Tezduyar [8] have applied these ideas to the massive parallel simulation of three-dimensional flows with complex geometries. Lohner [5] has demonstrated the extension of the advancing front algorithm to produce unstructured tetrahedral meshes on parallel platforms.

In recent years, much progress has been made developing scalable parallel algorithms based on octrees [12]. Sundar et al [13] presents the Dendro library that contains algorithms for bottom-up construction and 2:1 balance refinement of large linear octrees on distributed memory machines. It builds upon these algorithms efficient data structures that support trilinear conforming finite element calculations on linear octrees. Dendro stores the leaves octants inside a linear array and has been scaled over 4,000 cores. Tu et al. [14] presents the Octor library. Octor stores the single octree using pointers between parents and child nodes and has been scaled up to 67,000 cores. It has been used for earthquake simulation. A forest-of-octrees approach is presented in [15]. There it is presented the p4est library - a scalable parallel adaptive mesh refinement and coarsening (AMR), partitioning and 2:1 balancing on computational domains composed of multiple connected 2D quadtrees or 3D octrees. The performance studies show that p4est scales well up to 220,320 CPU cores [15].

In this work we present a parallel octree-based meshing generator based on the same concepts of the works presented above. However, our work is capable of handling arbitrary immersed surfaces.

1550-6533/10 $26.00 © 2010 IEEE
DOI 10.1109/SBAC-PAD.2010.26

IEEE computer society

Computational geometry algorithms are used to determine the interception between the octree and the arbitrary surface.

This article is organized as follows. In section 2 we describe the octree structure and the procedures for construction, balancing and meshing the octree. Next, we detail the surface interception algorithm. Performance studies are presented in section 4. Finally, in the last section we draw some conclusions and extensions of the present work.

3. Parallel linear octrees

3.1. Background

Octrees [12] are hierarchical data structures used in a number of scientific applications that requires an axis-aligned hierarchical partitioning of a volume of 3D region. This region is referred to as the domain of the tree. As the name suggests, each parent node in the octree has eight children. Additionally, each node also has a finite volume associated with it. The root node is generally taken to be the smallest axis-aligned cube fully enclosing the domain. This volume is then subdivided into eight smaller equal-size subcubes (also called cells, octants, or voxels) by simultaneously dividing the cube in half along each of the x, y, and z axes. These voxels form the child nodes of the root node. The child nodes are, in turn, recursively subdivided in the same way. Typical criteria for stopping the recursive creation of the octree include the tree reaching a maximum depth or the cubes getting smaller than some preset minimum size. The depth of a node from the root is referred to as its level.

Figure 1 shows the corresponding 2D scheme, a quadtree domain decomposition and its tree representation. The root of the tree is at level 0 and the children of any node are on level higher than the parent. Nodes that have no children are called leaf nodes.

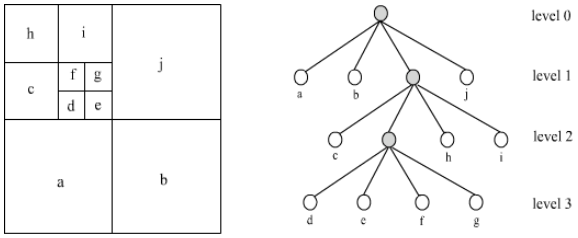


Figure 1. Decomposition of the square domain usign the quadtree and tree representation.

Considering that the maximum permissible depth of an octree is fixed a priori and that each node has exactly eight nodes, the octree can also be represented with a complete list of leaf nodes. Such representation is referred to a linear octree, since a linear array is sufficient for this representation instead of the tree data structure. The major advantage of linear octree is that it does not require storing pointers or interior nodes and reduces the overhead associated with pointers use.

The basic idea of the linear octree is to encode each octant with a scalar key called locational code that uniquely identifies the octants. A locational code is a code that contains information about position and level of the octant in the tree. In this work, we are using Morton's code [12]. An advantage of Morton's code is that when we sort the leaf nodes according to their locational codes (treated as a binary scalar value), the order we obtain is the same as the preorder traversal of the octree.

To build the locational code, we consider the domain as a Cartesian coordinate space that consists of a uniform grid of $2^n \times 2^n \times 2^n$ indivisible voxels where n is the maximum possible. Any octant can be identifying by an integer triplet representing its lower left corner coordinates and its levels in the tree. Thus, Morton's code for any octant is derived by interleaving the binary representation of three coordinates of the octant and then, appending the binary representation of the octant`s level. For instance, Figure 2 illustrates Morton's encoding for a quadtree. We assume that the maximum tree level supported is 3, then the locational code of node d is $(00100100\ 011)_2$.

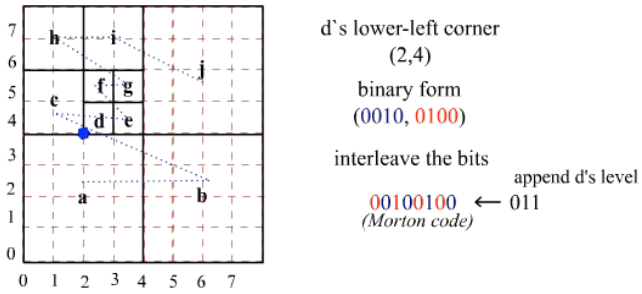


Figure 2. Morton's code

This process is reversible, i.e., given a Morton's id we can extract the integer triplet that identifies the lower corner of an octant. In the remainder of paper the terms lesser and greater are used to compare octants based on their Morton's code.

3.2. Construction and Balancing

In this subsection we describe the algorithms for parallel construction and parallel balancing of a linear octree.

Octree Construction. According to a given number of cores, our algorithm determines the initial global

number of octants and the initial levels of the tree. Then, each core computes its first and last local octants based on Morton's code. Given these two octants, the algorithm generates the minimal number of octants that span the region between these octants according to the Morton's ordering.

Algorithm 1: Construction

```
Output:  R, a partial list octants
1:   level ← 0
2:   n_octants ← 0
3:   {getting minimum required level and total octants}
4:   while level < MAXLEVELS do
5:      if n_octants < numProcs break;
6:      n_octants  ← n_octants * 8
7:      level ← level + 1
8:   end while
9:   // Getting Morton's id for first and last local octant
10:  First ← n_octants*myRank/numProcs
11.  Last ← (n_octants*(myRank+1)/myRank)-1
12.  a ← getOctant(Fisrt,level)
13.  b ← getOctant(Last,level)
15.  ancestor ← NearestCommonAncestor(a,b)
16.  W ← getChildren(ancestor)
17.  for each w ∈ W do
18.       if a < w < b AND w ∉ {AncestorsOf(b)}
19.            R ← R + w
20.      else if w ∉ {AncestorsOf(a), AncestorsOf(b)}
21.            W ← W – w + getChildren(w)
22:      end if
23: end for
24: Sort(R)
```

Octree Refinement. The non-recursive refinement algorithm transverses all leaf octants for each local octree replacing an octant with its eight children. The refinement criterion is determined by an user defined callback function. Note that this procedure modifies the octree structure invalidating the current partition that needs to be recomputed. The pseudo-algorithm below summarizes this process.

Algorithm 2: Refine

```
Input:   R, a partial list of octants
         RefFunc, a callback user defined function
         Ctx, a user data pointer
Output: R, a partial list of octants
1.  for each w ∈ R
2.     if Level(w) < MAXLEVEL AND RefFunc(w,ctx)
3.           R ← R – w + getChildren(w)
4.      end if
5.   end for
6.  Partition(R)
```

Parallel octree partition. After the octree construction, each core has an equi-distributed number of octants. However, that distribution can change during the refinement process. Parallel octree partition refers to redistribution of the octants among processes with the objective to reach load balance (i.e. to equi-partition the computational work uniformly among processes), which is necessary to ensure parallel scalability of an application.

We partition a global octree among all processors with a simple rule that each processor is a host for a contiguous chunk of leaf octants in the pre-order transversal ordering [11]. Our partition algorithm is based on an equi-partition by local octants counts and it can be summarized in two steps. First, we compute the global number of octants and determine the new local number of octants for each process. Next, we determine the new octants boundaries and send the octants to their new owner process. Note that, the partitioning and data redistribution involves leaf octants migrating only between neighbors processors.

2:1 Balance Constraint. In many applications involving octrees, such as octree-based finite element meshing, it is desirable that no leaf octants sharing at level *l* shares an edge or face with another leaf at level greater than *l+1*. To satisfy such constraint, the octants are refined until all their descendents, which are created in the process of subdivision, satisfy the balanced constraint. The algorithm implemented in this work is based on [13]

We refine nodes in a local linear octree, which fail to satisfy the balance constraint. During these subdivisions, new imbalances could be introduced and, consequently, this process needs to be repeated iteratively. The fact that an octant can affect other octants not immediately adjacent to it is known as the ripple effect. Figure 3 illustrates the local balance refinement and ripple effect. We select the finest octants of the tree and check if their insulation layer (all neighbors' octants) satisfies the 2:1 constraint. For instance, in the Figure 3(a), the red octant has an adjacent octant (yellow) that need to be refined. Afterwards, due to the ripple effect, two other non-adjacent octants are refined.

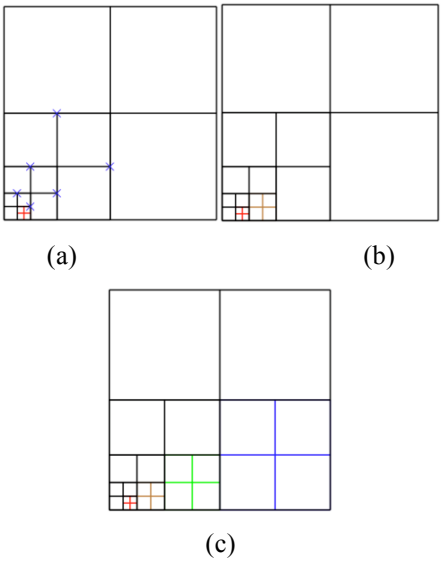


Figure 3. 2:1 Balance constraint and ripple effect

For parallel computations, the balancing scheme is performed in two-stages: first we perform local block balancing on each processor (core), and follow this up balancing across the inter-processor boundaries. In the first stage, the local linear octree is partitioned in coarse blocks. Each block has descendents that are present in the fine octree. This structure works as a linear sub-tree where each block is the root. Using this approach means minimizing the number of octants that needs to be split due to inter-processor violations of the 2:1 balancing rules [13]. The next step after block construction is a local balancing on each block. Then, all intra-processor block boundaries are gathered for validating the 2:1 constraint through the ripple algorithm. More details on the ripple propagation algorithm, see [13]

After intra-processor balancing, the inter-processor boundaries need to be balanced. The process starts with the communication of every local octant in the inter-processor boundary to processors that overlap with their insulation layer. In the second stage of communication, all local inter-processor boundary octants that overlap with the insulation layer of remote octants received from another processor are communicated to that processor. After these communications stages, each processor balances the union of local and remote boundary octants using the ripple algorithm. The parallel balancing is summarized in Algorithm 3.

Algorithm 3: Balance
Input: L, An unbalanced list of octants Output: R, A balanced list of octants
1: *LocalBlockBalance* 2: *B ← GetIntraProcessorBoundaries(L)* 3: *D ← Ripple(L + B)* 4: *T ← GetInterProcessorBondaries(D)* 5: *R ← Ripple(D+T)* 6: *Sort(R)* 7: *RemoveDuplicatess(R)*

Octree Meshing. Here we describe how we construct the support structure data in order to allow FEM computations. All complexities to handling nodes and mesh entities are encapsulated in this procedure which is performed by the following steps: (1) Exchange the intra-processors boundaries for neighboring processors. (2) Discover hanging and independent nodes (3) Establish sharing relationships between nodes on different processors (4) Assign per-processor local node ids and element ids; (5) Build hanging nodes mapping (6) Get element connectivity. The algorithm for the finite element mesh generation given the linear octree is outlined in Algorithm 4.

Algorithm 4: Meshing
Input : A linear octree Output: Trilinear (hexahedral) finite element mesh
1: Exchange Ghost Nodes 2: Get Nodes 3: Identify hanging nodes 4: Identify Independent nodes 5: Identify communication nodes 6: Get Element connectivity

4. Immersed Surface detection

It is often important that the mesh generators have support to handle arbitrary immersed surfaces boundaries to represent complex solid boundaries. Here we assume that surfaces are given by a triangulation, which typically is obtained from a CAD package via STL files. Finding which octants intersect the surface is an operation often very expensive. To minimize this cost, object bounding volumes are usually tested for overlap before the geometry intersection test is performed.

A bounding volume (BV) is a single simple volume encapsulating one or more objects of more complex nature. The idea is for the simpler volumes (such as boxes and spheres) to have cheaper overlap tests than the complex objects they bound. Using bounding volumes allows for fast overlap rejection tests because

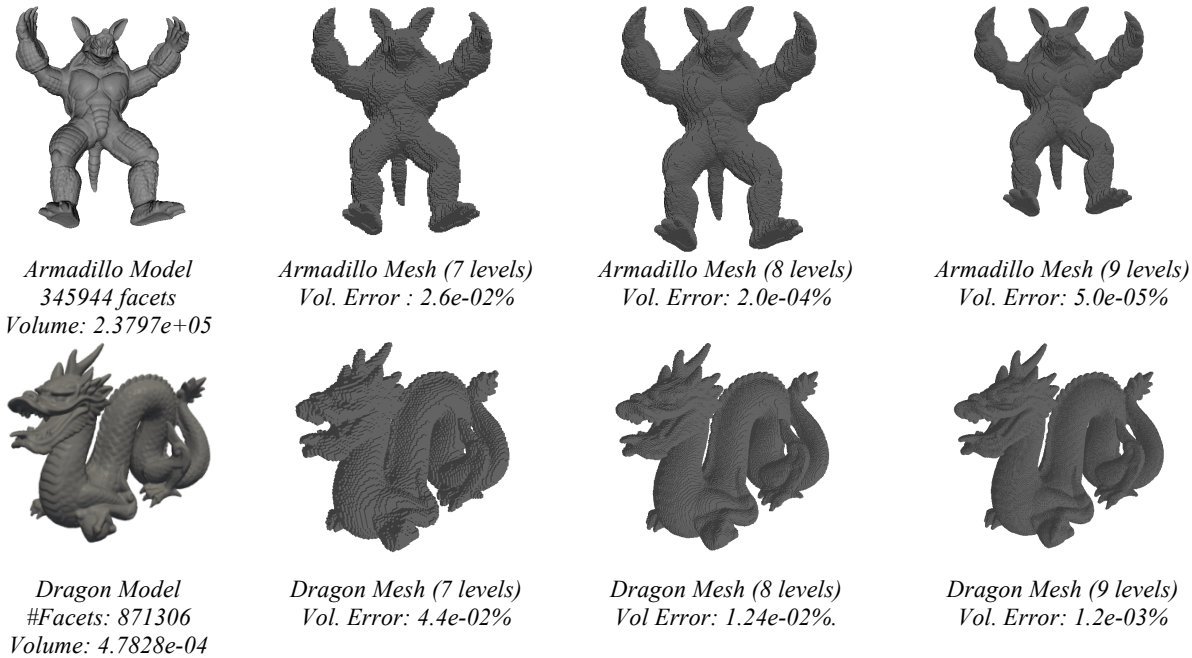
one need only test against the complex bounded geometry when the initial overlap query for the bounding volumes gives a positive result [16]

Considering that the surface does not change during octree refinement, a bounding box tree (BBT) is built after reading the STL file and used for the searching algorithm. The first step to build a BBT is encapsulating each surface triangle in a bounding box and inserts it in a list. Each bounding box has a pointer to the bounded object. Our algorithm builds a new hierarchy of bounding boxes from the bounding boxes list. At each level, the node data field contains a bounding box of all children. The tree is binary and is built by repeatedly cutting in two approximately equal halves the bounding boxes at each level until a leaf node (i.e. a triangle bounding box) is reached. In order to minimize the depth of the tree, the cutting direction is always chosen as perpendicular to the longest dimension of the bounding box.

Finding which octants intercept the surface consists in performing a binary search over the tree. Given an octant (which can be seen like a bounding box), the algorithm return NULL, if the octant does not intercept any bounding box, or a node tree. In the last case, the node data field has a pointer to a list of surface triangles. If the octant intercepts some triangle in the list, the algorithm returns true, otherwise, false. This algorithm has a complexity of $O(nlogn+k)$ where n is number of surface triangles and k depends on the list size returned by the searching algorithm.

6. Performance Results

In this section we present computational results for the tree construction, balancing, and meshing for a number of different case studies. These algorithms are implemented in C/C++ using message passing interface through OPENMPI library. The TAU library [17] is used for profiling the code. We test the code performance on the SGI Altix ICES 8200 system equipped with 16 compute nodes (two Intel Quad-core 2.8 GHz and 16GB per node) at the High Performance Computer Center (NACAD). We consider mesh generation for three different immersed surfaces. In all cases, the initial refinement is created by the construction algorithm followed by an iterative invocation of Refine algorithm using the surface detection algorithm to identify octants that must be refined. Then Balance and Partition algorithms are executed to ensure the 2:1 constraint and load-balancing among processors. Finally, the meshing procedure is used to build nodes and elements (hexahedra) connectivity.



Armadillo Model
345944 facets
Volume: 2.3797e+05

Armadillo Mesh (7 levels)
Vol. Error : 2.6e-02%

Armadillo Mesh (8 levels)
Vol. Error: 2.0e-04%

Armadillo Mesh (9 levels)
Vol. Error: 5.0e-05%

Dragon Model
#Facets: 871306
Volume: 4.7828e-04

Dragon Mesh (7 levels)
Vol. Error: 4.4e-02%

Dragon Mesh (8 levels)
Vol Error: 1.24e-02%.

Dragon Mesh (9 levels)
Vol. Error: 1.2e-03%

Figure 4: Meshes for Dragon and Armadillo models for several refinement levels

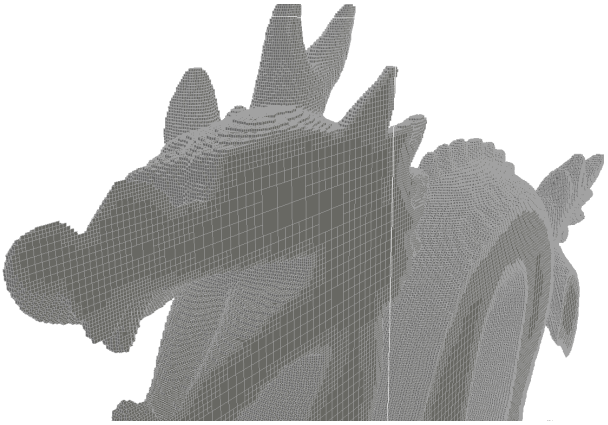


Figure 5: Dragon finite element mesh detail

We begin our case study of mesh generation from octants located inside surfaces. In this case, the solid model is kept which is typical in meshes employed in computational solid mechanics. We choose two surfaces with complex geometry to test our surface detection algorithm. Figure 4 illustrates the surfaces used for these tests and the finite element mesh obtained for several refinement levels. For each model we present the number of facets in the surface followed by the enclosed volume and the relative error between the computed octree volume and the surface enclosed volume. We note that for 9 levels the volume error is negligible for both models. In Figure 5 we show for the Dragon model a zoom of the resulting finite element mesh, where we can appreciate the quality of the generated mesh. In addition, Table 1 reports some metrics related to octree refinement and 2:1 balancing. For each mesh, we show the total number of unbalanced octants obtained after the refinement algorithm, the total number of octants from a balanced octree and the total number of octants intercepted by the surface. Note that, an increase in the refinement level yields around 8 times as many octants.

Table 1. Problem size

Armadillo Model			
Levels	*#unbalanced octants*	*#balanced octants*	*Mesh Element*
7	374,186	470,492	262,971
8	6,054,763	7,731,088	4,002,262
9	24,253,825	31,086,595	15,823,612
10	388,478,651	499,164,968	250,727,430
Dragon Model			
Levels	*#unbalanced octants*	*#balanced octants*	*Mesh Element*
7	94,179	11,6803	71,348
8	6,254,459	7,983,864	4,126,210
9	25,050,551	32,110,891	16,316,371
10	40,1146,316	515,502,793	258,876,070

An isogranular mesh generation scalability analysis is performed for both models listed above and core counts ranging from 8 to 128. Isogranular scalability is performed by tracking the execution time while increasing the problem size (refinement levels) and number of processors. Table 2 presents the exclusive execution time (*in secs*) taken by TAU profiler utility for the algorithms: surface detection, 2:1 balancing and octree meshing. Our scheme demonstrated reasonable scalability up to 128 cores. In both cases, with 128 cores, the balancing and meshing runtime was almost 4 and 5 times the runtime for 64 cores, respectively. However, the problem size increases around 16 times. We execute the 2:1 balancing operations over 4.0×10^8 octants on 128 cores in less than 10 seconds per 2×10^5 octants per core.

Table 2. Isogranular scalability

Armadillo Model				
Levels	*CPU*	*Surface Detection*	*Balancing*	*Meshing*
7	8	6.393	19.433	14.649
8	16	13.796	46.701	26.154
	32	9.104	42.370	16.154
9	64	13.268	59.828	52.564
10	128	70.139	256.695	303.451
Dragon Model				
Levels	*CPU*	*Surface Detection*	*Balancing*	*Meshing*
7	8	5.799	15.952	9.787
8	16	18.027	51.393	26.686
	32	10.751	52.427	26.406
9	64	18.069	87.364	53.231
10	128	117.344	312.072	270.800

In Figure 6, we display for the Armadillo model the runtime percentages of the main balancing components. We may note that local block partition and local balancing requires about from 20% to 30% of the balancing runtime while the ripple cost is kept constant in around 10%. The communication of boundary octants among neighboring processors takes only 5% of balancing runtime. After the 2:1 balancing, the octree is not balanced, and needs to be repartitioned. It takes up 15% of runtime, except in 8 cores where there is no inter-node communications and a small number of octants per core.

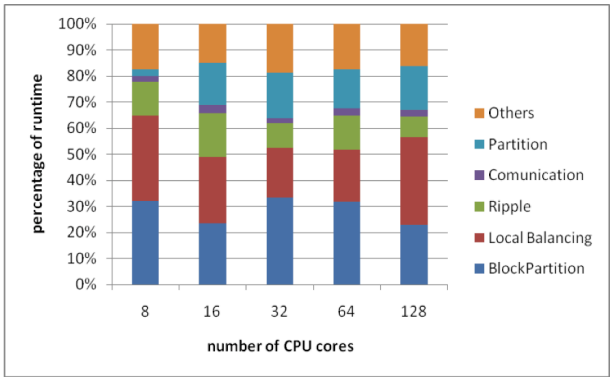
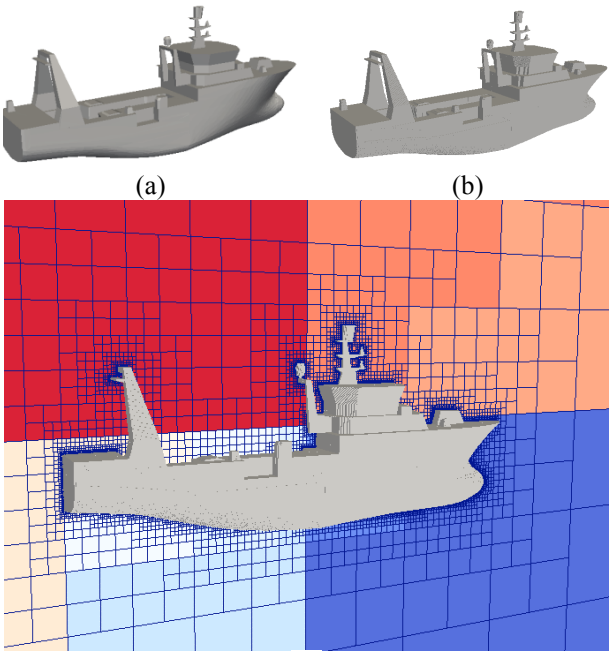


Figure 6. Armadillo model - Percentage runtime for 2:1 balancing

In our second case study, we are interested in mesh generation from octants located outside the surface, i. e. we remove the solid body and keep only the cubic domain as typically required in computational fluid dynamics. In this case, we consider the Tangaroa model from [18], as presented in Figure 7. Figure 6(a) and 6(b) illustrate the Tangaroa model with 2,668 facets and internal octants, respectively. Figure 6(c) shows the finite element mesh generated in 16 processors.



(a) (b)

(c)

Figure 7. Tangaroa model and finite element mesh

In order to perform the isogranular scalability analysis in this test case, we fixed the maximum number of octants subdivisions during the surface inteception stage in 9, 10 and 11 levels. Table 3 reports the number of balanced octants, hexahedral elements, number of independents nodes and number of hanging nodes for these refinement levels.

Table 3: Tangaroa model – mesh size

	Refinement Levels		
	Level 9	*Level 10*	*Level 11*
Bal. Octants	3,230,662	12,922,407	20,7519,379
Hex. Elem.	1,516,756	6,421,987	104,502,895
Indep. Nodes	2,527,744	10,434,275	168,573,848
Hang. Nodes	719,904	2,899,980	47,634,334

From Table 4, we can observe the same behavior reported in the last two cases. The 2:1 balancing is the more expensive procedure. The number of finite elements created in 128 cores correspond about 17 times the number of elements created in 64 cores and the runtime increased in the same proportion. The surface detection procedure in the Tangaroa case presents better computational cost than in the other cases. In fact, this gain can be related to the number of facets in the surface models. Tangaroa's surface has 2,668 facets while Armadillo and Dragon models have 345,944 and 871,306 facets, respectively.

Table 4. Tangaroa model – Isogranular scalability

Tangaroa Model				
CPU	***Level***	***Surface Detection***	***Balancing***	***Meshing***
32	9	6.626	43.135	18.353
64	10	0.732	66.195	36.191
128	11	1.708	799.886	532.260

Figure 8 shows the percentage runtime of the main subroutines related to the mesh generation. We can observe that the costs of *GetElement* routine increases when the number of elements increases. It corresponds to around 80% of mesh generation runtime in 128 cores. In *GetElement* routine the element connectivity generation demands higher costs. Connectivity is computed searching locally in hash tables for eight element nodes. To do this operation *GetGhost* gathers all ghost octants information needed to generate independently in each core the element connectivity.

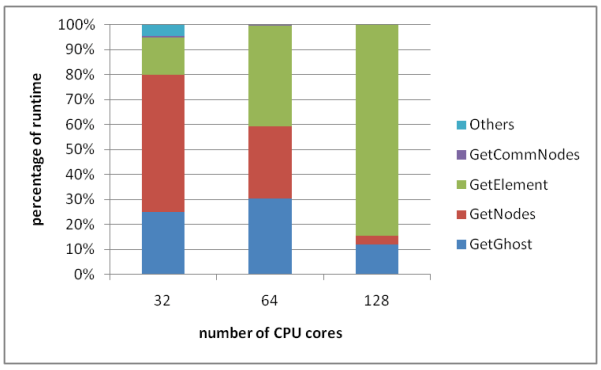


Figure 8. Tangaroa model – percentage runtime of finite element mesh generation

7. Conclusion and future work

In this work, we have presented a parallel hexahedral mesh generation method based on octrees with support to immersed surfaces. We describe a set of algorithms for construction, balancing and meshing of linear octrees. Our target applications include the mesh generation for arbitrary geometries applied to solid and fluid mechanics. To reach this, we use efficient computer graphics algorithms for surface detection based on boundaries box tree. We presented results that verify the scalability of our code. It executes the 2:1 balancing operations over 4.0×10^8 octants on 128 cores in less than 10 seconds per 2×10^5 octants/core.

There are two important extensions: building conforming meshes and adaptive mesh refinement/coarsening simulations. For the former, we need to design new algorithms to eliminate hanging nodes and, if necessary, make the mesh boundary-fitted. Adaptive mesh simulations require additional restriction and prolongation operators to allow refinement and coarsening schemes.

Acknowledgments

This work is partially supported by CNPq, Petrobras, and ANP. Computer time on the SGI Altix ICE 8200 is provided by the High Performance Computer Center at COPPE/UFRJ.

8. References

[1] P. L. George, Automatic Mesh Generation: Application to Finite Element Methods, Wiley, New York, 1991

[2] N. A. Verhoeven, N. P. Weatherill, K. Morgan, Dynamic load balancing in a 2D parallel Delaunay mesh generator, Proc. Parallel CFD Conference, 1995.

[3] B. H. V. Topping, B. Cheng, Parallel and distributed adaptive quadrilateral mesh generation, Computers and Structures, 73 (1999), pp. 519–536.

[4] L. Laemer, M. Burghardt, Parallel generation of triangular and quadrilateral meshes, Advances in Engineering Software, 31 (2000), pp. 929–936.

[5] R. Lohner, J. Camberos and M. Merriam, Parallel Unstructured Grid Generation, Comp. Meth. Appl. Mech. Eng., 95 (1992), pp. 343–357.

[6] L. P. Chew, N. Chrisochoides and F. Sukup, Parallel Constrained Delaunay Meshing, Proc. Workshop on Trends in Unstructured Mesh Generation, 1997.

[7] N. Chrisochoides, D. Nave, Simultaneous mesh generation and partitioning for Delaunay meshes, Mathematics and Computers in Simulation, 54 (2000), pp. 321–339.

[8] A. A. Johnson, T. E. Tezduyar, Parallel computation of incompressible flows with complex geometries, International Journal for Numerical Methods in Fluids, 24(12): 1321-1340, 1997.

[9] A. Shostko and R. Lohner, Three-Dimensional Parallel Unstructured Grid Generation, Int. J. Num. Meth. Eng., 38 (1995), pp. 905–925.

[10] E. G. Ivanov, H. Andrae and A. N. Kudryavtsev, Domain Decomposition Approach for Automatic Parallel Generation of Tetrahedral Grids, International Mathematical Journal Computational Methods in Applied Mathematics, Vol.6(2), 2006, pp. 178--193.

[11] S. Aluru and F. E. Sevilgen, Parallel domain decomposition and load balancing using space-filling curves, In Proceedings on the 4th IEEE conference on High Performance Computing, 1997.

[12] H. Samet, Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS, Addison-Wesley, Publishing Company, 1990.

[13] H. Sundar, R. S. Sampath, S. S. Adavani, C. Davatzikos and G. Biros, Low constant parallel algorithms for finite element simulations using linear octrees. In Becky Verastegui (Ed.): Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, November 10-16, 2007, Reno, Nevada, USA. ACM Press 2007.

[14] T. Tu, D. R. O'Hallaron and O. Ghattas, Scalable parallel octree meshing for terascale applications. In Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA.

[15] C. Burstedde, L. C. Wilcox, and O. Ghattas, p4est: Scalable algorithms for parallel adaptive mesh refinement on forest of octrees. Submitted to SIAM Journal on Scientific Computing.

[16] C. Ericson, Real collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology), Morgan Kaufmann, January 2005

[17] S. Shende and A. D. Malony. The TAU Parallel Performance System. International Journal of High Performance Computing Applications, 20(2): 287-311, 2006.

[18] S. Popinet, Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries, J. Comput. Phys.190 (2):572-600, 2003.

[19] International Meshing Roundtable, http://www.imr.sandia.gov/, last visited, May 20, 2010.