SANDIA REPORT

SAND2010-6400
Unlimited Release
Printed September 2010

Parallel Octree-Based Hexahedral Mesh Generation for Eulerian to Lagrangian Conversion
LDRD Project #149521

Steven J. Owen, Matthew L. Staten

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2010-6400
Unlimited Release
Printed September 2010

Parallel Octree-Based Hexahedral Mesh Generation for Eulerian to Lagrangian Conversion LDRD Project #149521

Steven J. Owen and Matthew L. Staten
Computational Simulation Sciences Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0382
sjowen@sandia.gov, mlstate@sandia.gov

Abstract

Computational simulation must often be performed on domains where materials are represented as scalar quantities or volume fractions at cell centers of an octree-based grid. Common examples include bio-medical, geotechnical or shock physics calculations where interface boundaries are represented only as discrete statistical approximations.

In this work, we introduce new methods for generating Lagrangian computational meshes from Eulerian-based data. We focus specifically on shock physics problems that are relevant to ASC codes such as CTH and Alegra.

New procedures for generating all-hexahedral finite element meshes from volume fraction data are introduced. A new primal-contouring approach is introduced for defining a geometric domain. New methods for refinement, node smoothing, resolving non-manifold conditions and defining geometry are also introduced as well as an extension of the algorithm to handle tetrahedral meshes. We also describe new scalable MPI-based implementations of these procedures.

We describe a new software module, *Sculptor*, which has been developed for use as an embedded component of CTH. We also describe its interface and its use within the mesh generation code, CUBIT. Several examples are shown to illustrate the capabilities of Sculptor.

Acknowledgment

Special thanks to David Hensinger for his help in integrating Sculptor into CTH. Also to Roshan Quadros and Greg Whitford for their contributions to the Sculptor algorithms.

Contents

Appendix

Figures

Tables

Nomenclature

Alegra Arbitrary Lagrangian Eulerian General Research Application developed at Sandia.

CTH Eulerian-based code for simulation of high speed impact and penetration problems. Developed and maintained at Sandia National Laboratories.

CAMAL Set of meshing algorithm components developed for use in Cubit and for integration in other FEA based tools. Includes tet, hex, quad and tri meshing as well as refinement and smoothing.

CUBIT Computational simulation pre-processing software tool for mesh generation and geometry manipulation. Developed and maintained at Sandia National Laboratories.

Exodus Mesh data format developed at Sandia.

Eulerian Grid Three-dimensional axis-aligned grid used as the computational domain for Eulerian-based codes such as CTH.

Lagrangian Grid Finite element mesh typically used as the computational domain for Langrangian based codes such as Presto.

Presto Sandias in-house explicit, transient, dynamic finite-element software package.

Sculptor New CAMAL software component developed as part of this work to generate hexahedral and tetrahedral meshes from volume fraction data.

Spyplot Data manipulation and visualization module used by CTH. Volume fraction data can be dumped from Spyplot module for use in Sculptor.

1 Introduction

1.1 Motivation

Computational simulation must often be performed on domains where materials are represented as scalar quantities or volume fractions at cell centers of an octree-based grid. Common examples include bio-medical, geotechnical or shock physics calculations where interface boundaries are represented only as discrete statistical approximations. CTH is an example of a code that utilizes an Eulerian grid as its computational domain. The results of a CTH calculation are represented as volume fractions in the individual cells of the domain. In practice, this is represented as a 3-dimensional array of scalar values ranging from 0.0 to 1.0, where 1.0 represents material that completely fills the volume of the cell, and zero represents the absence of material. Values that fall between represent the percentage of material, by volume, that is filling the volume of the cell.

There are situations where it is desirable that the results of the calculation of an Eulerian-based code is used as input to a Langrangian, or finite element based code. For our application we focus on the specific problem of generating a hexahedral finite element mesh from the volume fraction data generated from CTH to be used as input to the finite element code, Presto. To accomplish this, the scalar volume fraction data array must be interpreted and converted into a boundary aligned hexahedral mesh that is of sufficient quality to be used in a finite element calculation.

Methods for generating Lagrangian meshes from Eulerian grids have been presented in the literature (see Background section below). The CTH example provides unique and challenging aspects that existing methods have not yet addressed. In particular, the problem of generating an all-hex mesh in parallel, is of importance. In this work we introduce new approaches to solving the all-hex meshing problem from volume fraction data that specifically address the problem in the context of distributed memory parallel processing. We also introduce improved methods applicable for both serial and parallel processing. For example a new primal-contouring approach is introduced for defining the fragment domains computed in CTH. New methods for refinement, node smoothing, resolving non-manifold conditions and defining geometry are also introduced as well as an extension of the algorithm to handle tetrahedral meshes.

We describe a new software module, *Sculptor*, which has been developed for use as an embedded component of CTH. We also describe its interface and its use within the mesh generation code, CUBIT. Several example meshes generated with Sculptor including shock physics examples are shown.

1.2 Background

The development of general-purpose unstructured hexahedral mesh generation procedures for an arbitrary domain have been a major challenge for the research community. A wide variety of techniques and strategies have been proposed for this problem. It is convenient to classify these methods into two categories: *geometry-first* and *mesh-first*. In the former case, a topology and

geometry foundation is used upon which a set of nodes and elements is developed. Historically significant methods such as plastering [1], whisker weaving [17] and the more the recent unconstrained plastering [15] can be considered *geometry-first* methods. These methods begin with a well defined boundary representation and progressively build a mesh. Most of these methods define some form of advancing front procedure that requires resolution of an interior void and have the advantage of conforming to a prescribed boundary mesh. Although work in the area is ongoing, the ability to generalize these techniques for a comprehensive set of B-Rep configurations has proven a major challenge and has yet to prove successful for a broad range of models.

In contrast, the mesh-first methods start with a base mesh configuration. Procedures are then employed to extract a topology and geometry from the base mesh. These methods include grid-overlay or octree methods. In most cases these methods employ a Cartesian or octree refined grid as the base mesh. Because a complete mesh is used as a starting point, the interior mesh quality is high, however the boundary mesh produced cannot be controlled as easily as in geometry-first approaches. As a result the mesh may suffer from reduced quality at the boundary and can be highly sensitive to model orientation. In addition, grid-overlay methods may not accurately represent the topology and geometric features as defined in the geometric model. In spite of these inherent deficiencies, mesh-first methods have proven a valuable contribution to mesh generation tools for modeling and simulation. In contrast to geometry-first techniques, fully automatic mesh-first methods have been developed for some applications where boundary topology is simple or is not critical to the simulation. In particular, bio-medical models [23] [24] [4], metal forming applications [12] [8], and viscous flow [18] methods have utilized these techniques with some success. Automating and extending mesh-first methods for use with general B-Rep topologies would provide an important advance in hexahedral meshing technology.

As one of the first to propose an automatic overlay-grid method, Schneiders [12] developed techniques for refining the grid to better capture geometry. He utilized template-based refinement operations, later extended by Ito [4] and H. Zhang [22] to adapt the grid so that geometric features such as curvature, proximity and local mesh size could be incorporated. Y. Zhang [23] [24] and Yin [21] independently propose an alternate approach known as the Dual-contouring method that discovers and builds sharp features into the model as the procedure progresses. This is especially effective for meshing volumetric data where a predefined topology is unknown and must be extracted as part of the meshing procedure.

The dual contouring method for generating a hexahedral mesh described by Y. Zhang [23] begins by computing intersections of the geometry with edges in the grid. Intersection locations are used to approximate normal and tangent information for the geometry. One point per intersected grid cell is then computed using a minimization procedure that is based upon Hermite approximations from the tangents computed at the grid edges. The base mesh in this case is defined as the dual of the Cartesian grid, using the cell centroids and interpolated node locations at the boundary. While attractive as a method for extracting features from volumetric data, it does not guarantee capture of a pre-existing topology such as that contained in a CAD solid model.

Recent work on mesh-first approaches have focused more on the capturing of features of the geometry. A common thread among many of these methods [23] [4] [13] is the introduction of a buffer layer of hex elements to improve element quality near the boundary. Shepherd [14] also

describes an approach to mesh-first hexahedral mesh generation utilizing geometric capture procedures. This work utilizes theory and assertions developed in [6] [7] . Owen [10] expands on this work by introducing specific geometric algorithms for capturing topology for mesh first methods.

Methods that employ level sets to identify interfaces in a finite element mesh also employ techniques that build discrete boundaries from scalar data. Noble, et. al. [9] is one such example. They propose a method for imposing an interface through a fixed tetrahedral mesh based on the results of a volume of fluids calculation. This work introduces parallel methods for defining the interface surfaces, however, further smoothing and mesh improvement are not employed, which can result in poor quality or sliver tetrahedra. It is also not clear how these methods would extend to generation of hexahedra in parallel. Other interface reconstruction techniques, such as that proposed by Garimella et. al. [3] will construct very accurate interfaces that preserve individual cell volume fraction characteristics on an Eulerian grid. This work does not however address the problem of Lagrangian mesh construction, nor does it provide smooth surfaces from which to define a boundary aligned mesh. It is however, worthwhile to note these interface construction methods and their similarities to the current problem.

The proposed procedure in this work is a mesh-first method that uses the Eulerian grid defined in CTH as the base mesh. Since the application we are targeting does not require the definition of sharp features, we will neglect the topology capture issues necessary for CAD-based modeling. The dual contouring approach used by Zhang [23] appears to be attractive for our application, particularly as it has direct extension to the generation of hexahedral elements. We have chosen to further explore this technique and to extend its application for our purposes.

Several important contributions to this field are introduced in this work. Existing literature does not currently address parallel meshing problems and the unique issues it entails for mesh-first mesh generation methods. In addition, no literature is currently available on the appplication of octree-type methods to shock consolidation or transient dynamics Eulerian codes. This work develops further some of the concepts introduced by others in the field and applies them specifically to address the unique problems of parallel meshing for building a hexahedral mesh from an Eulerian grid.

2 Algorithm

2.1 Overview of Method

The following is a brief description of the procedure used for generating a hexahedral mesh from volume fraction data in parallel.

CTH provides a decomposition of the domain based upon 3-dimensional $L \times M \times N$ subdomains of the global grid as shown in Figure 1(a) . The global domain is assumed to be an axis aligned Cartesian grid with any number of subdomains with scalar volume fraction data assigned to the cell-centers of each cell of the grid. Figures 1(b), 1(c) and 1(d), illustrate the cells in the domain that contain scalar volume fraction data of 1.0, 0.0 and values between 0.0 and 1.0 respectively. The objective is for each subdomain to be processed independently with minimal communication, generating its portion of the global hexahedral mesh. The following procedure outlines the algorithm for a given $L \times M \times N$ subdomain.

1. Establish Cartesian Grid: A light-weight grid data structure is established to store and work on the data.

2. Estimate Gradients: Based upon the cell-centered data field, gradient vectors are approximated.

3. Compute Virtual Edge Crossings: Assuming an iso-value of 0.5, virtual edges (connecting adjacent cell centers) that have endpoints that bound the iso-value are identified.

4. Compute surface points: Interpolating the virtual edge crossing data, cell centers are projected to an interpolated iso-surface.

5. Establish Interior Hexes: Cells that lie within the surface points are established as the basis for the hexahedral mesh.

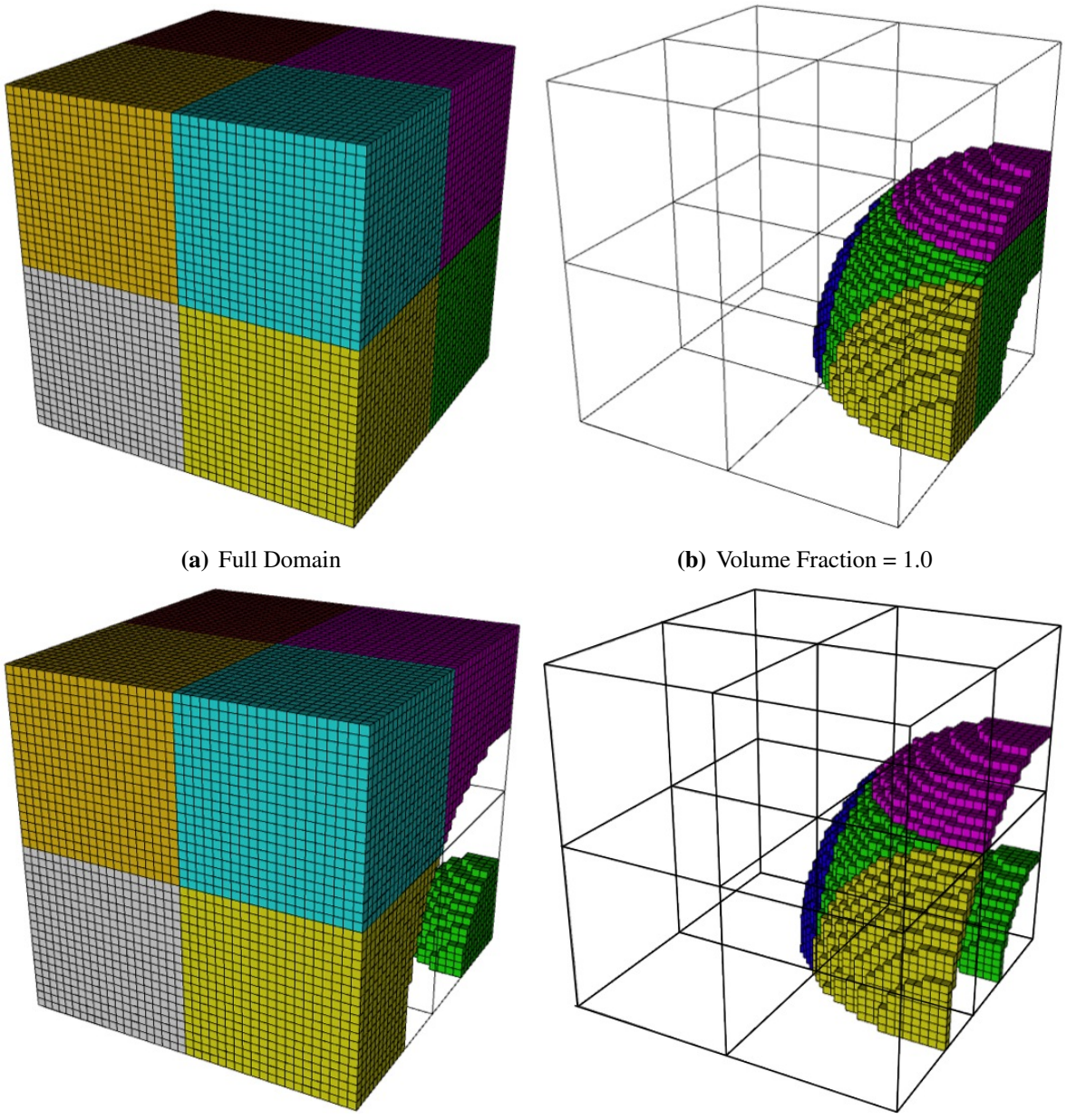
6. Resolve Non-Manifold Cases: Cells are added or deleted from the base set of hexes to resolve cases that would result in non-manifold connections in the final mesh.

7. Define Boundary Nodes: Grid nodes that lie on the iso-surface and those that define the boundary of the domain are established as the boundary of the Lagrangian grid.

8. Create Geometry Definition: Based upon the node and hex definition, a geometry description comprised of volumes, surfaces, curves and vertices is established with associativity to grid entities.

9. Insert Hex Buffer Layer: A layer of hexes is inserted at the iso-surface boundary. Care is taken to ensure the layer extends through surfaces defined at domain interfaces.

10. Generate Hex Mesh: The final connectivity for the hex mesh is established.



(a) Full Domain

(b) Volume Fraction = 1.0

(c) Volume Fraction = 0.0

(d) Volume Fraction between 0.0 and 1.0

Figure 1. An example of volume fraction data defined on a global domain. Colors represent separate processor subdomains. Cells are shown to illustrate different cell values for volume fractions

11. Smooth: Smoothing procedures are employed for curves, surfaces and volume mesh entities to improve mesh quality.

The following sections contain a discussion of some of the details of this procedure.

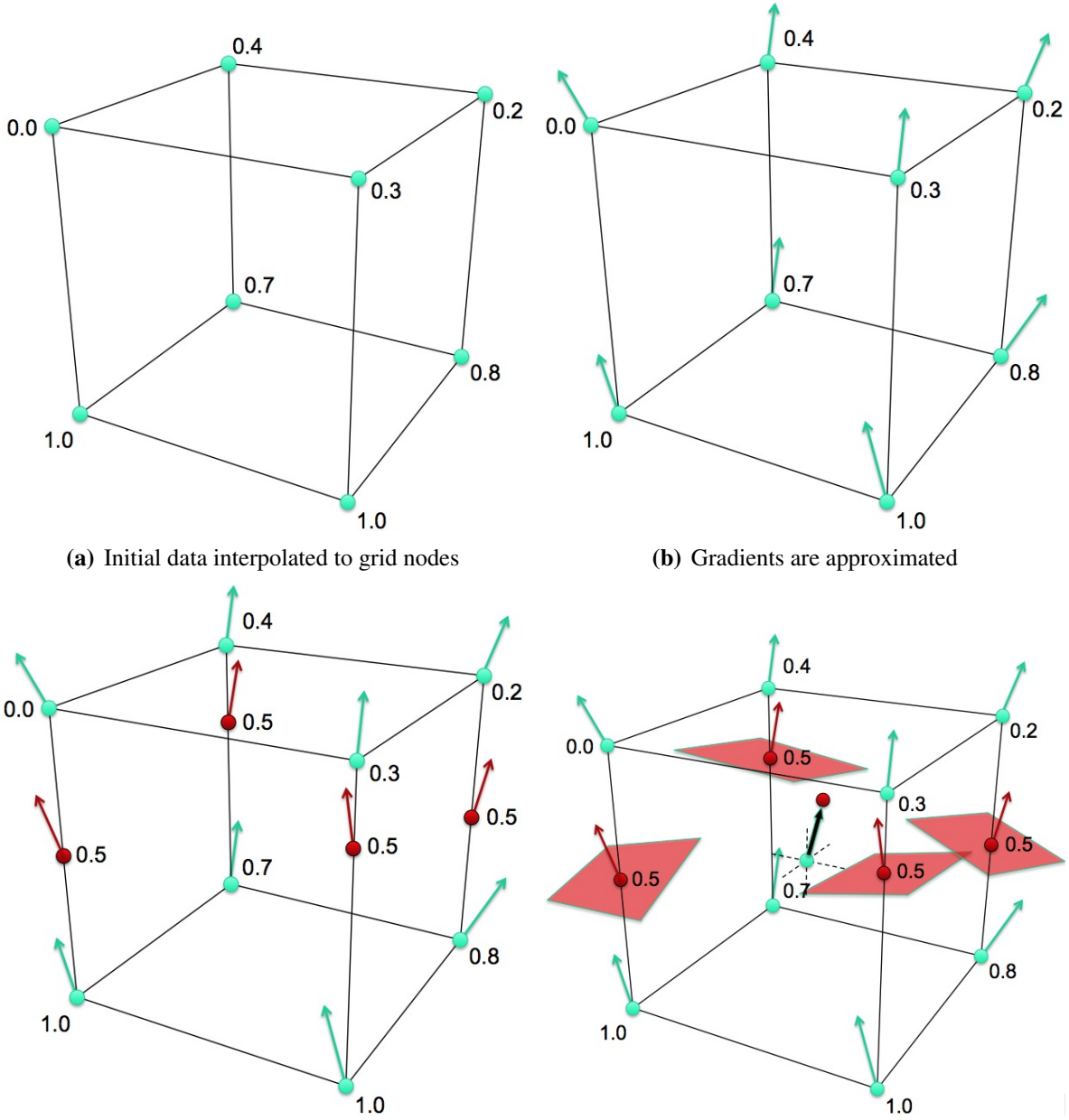
2.2 Dual Contouring in the Primal

In recent literature [23] [24], the principal method for generating meshes from scalar field data defined on a Cartesian grid is based upon the dual contouring procedure [16]. This procedure is illustrated in Figure 2. The dual contouring method is briefly described here for clarity, although modifications to the published procedure have been implemented for simplicity and for purposes of working in parallel. The assumption for this method, is that data is provided at the grid nodes as shown in Figure 2(a). Cell centered data would first be interpolated to the nodes to utilize this procedure. Next, the gradient vectors of the scalar field are interpolated at the grid nodes. This can be done using a least squares fit of the differences of the values at neighboring nodes. Figure 2(b) illustrates the approximated gradient vectors of the scalar field at the nodes.

Interpolation of a single gradient requires the volume fraction data at neighboring cells. This has implications for parallelism, as grid nodes at the processor boundaries will need to retreive data from neighboring processors. To reduce the overhead of communication, the neighboring cell data is retrieved only once at the start of the procedure and made available to the current processor for interpolation as *ghost cells*. Figure 3 shows the domains from the eight processors in the example in Figure 1. The cells from one of the processors are colored based upon their volume fraction value. Here we illustrate how an additional 2 layers of volume fraction data from neighboring domains are used for processor subdomain. This ensures that interpolation of gradient vectors on either side of a domain boundary will be consistent resulting in continuity of the resulting surface across processor boundaries. Although additional ghost cell layers are included in the Cartesian grid definition, hexahedra are only generated in the non-ghost cells.

The identification of edge crossings is then performed as illustrated in 2(c). In this case, edges in the domain whose end points bound the iso-value, 0.5 are established. The iso-value of 0.5 defined the statistical volume fraction value where the surface definition is most likely to exist. Since no real surface data is present, this is obviously an approximation, and at best, a guess at where the surface actually exists. The objective therefore, is to define an iso-surface everywhere in the domain where the scalar field is interpolated to be a value of 0.5. By identifying the edges and interpolating where the iso-value lies on the edge, we provide discrete data for the surface definition. For our purposes we use a simple linear interpolation of the edges. The example in Figure 2(c) illustrates 4 edges where edge crossings have been defined.

The objective of the dual contouring procedure is to relocate the center point of a cell based upon the edge crossing data of its edges. Reference [16] utilizes a hermite interpolation or minimization procedure to determine the cell center perturbation. We have found that a simple gradient-based interpolation of the edge crossing data is sufficient. Figure 2(d) show the gradient planes as they would be interpolated at the 4 edge crossing points for a cell. Using an inverse distance



(a) Initial data interpolated to grid nodes

(b) Gradients are approximated

(c) Edge Crossings Identified

(d) Cell Center Location Interpolated

Figure 2. Basic dual contouring procedure illustrated on a single cell of the grid

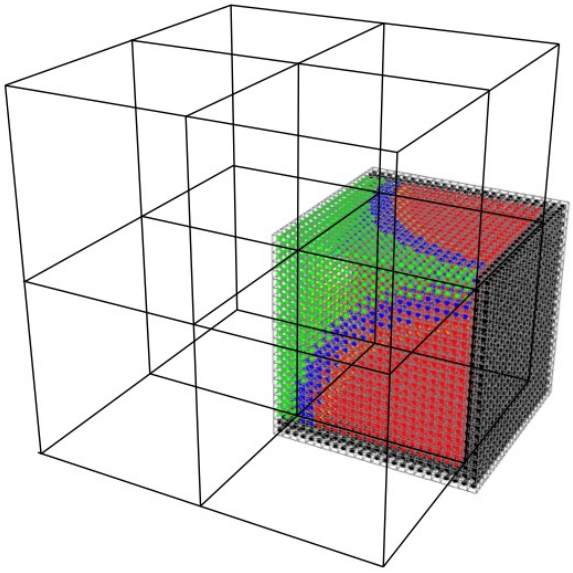


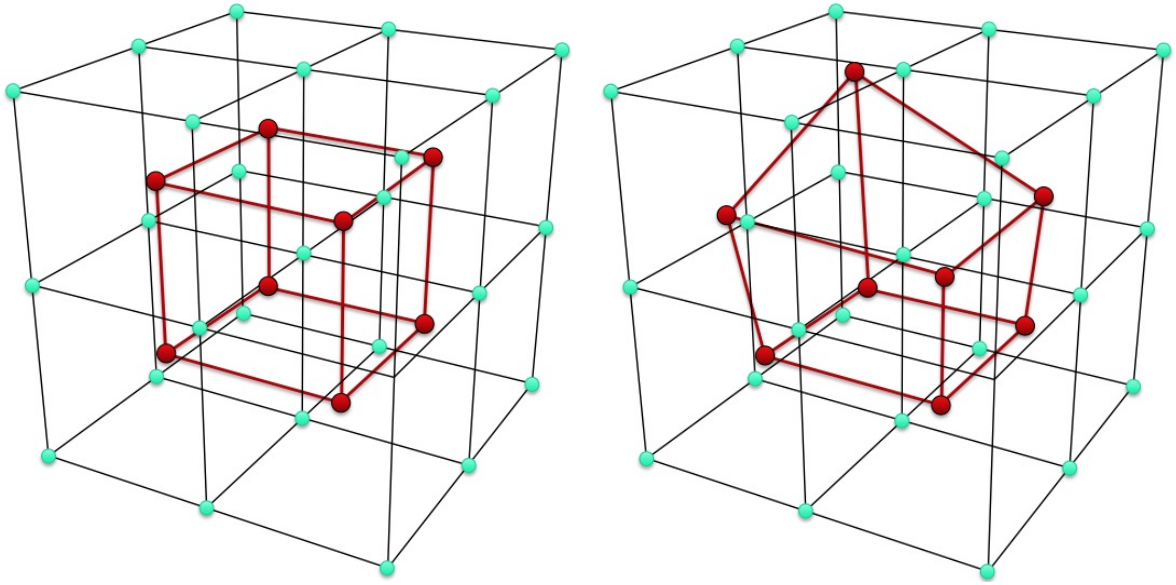
Figure 3. Representation of 8 processor domains where an additional 2 layers of volume fraction data is included on each processor.

weighted interpolation procedure of the gradient planes at its edge crossing, the perturbation of the center node can be approximated.

Rather than using the Cartesian grid, which we will refer to as the ”primal” grid, the dual contouring procedure represents the hexahedra as the dual of the Cartesian grid. Figure 4(a) shows a local 8-cell configuration in the primal grid, with its corresponding dual cell shown in red. The vertices of the dual cell are defined by the centers of the primal cells. The dual cell vertices, which will become the nodes of the hexhadral mesh, are modified using the preceding procedure. Figure 4(b) shows an example of a single dual cell with some of its vertices modified at the 0.5 iso-value. Dual cell vertices where no edge crossings have been identified in their corresponding primal cell edges, remain unmodified.

For the purposes of this work, we have proposed a modification to the traditional dual contouring approach. Rather than utilizing the dual cells as the basis for the final hexahedral mesh, we modify the primal cells themselves. Figure 5 shows an example grouping of 8 primal cells. Rather than identifying edge crossing data on the primal grid, we instead compute it on the dual edges. Dual edges are illustrated in Figure 5 as red dotted lines that connect primal cell centers. The same dual contouring calculations can then be computed on the basis of dual edge-crossings rather than primal edge-crossings. As a result, the interpolation to define the surface nodes is based on the dual edges. For practical purposes, this modification to the dual contouring algorithm can be thought of as shifting the definition of the grid globally by 1/2 cell.

Two main reasons motivated the dual contouring in the primal approach:



(a) Initial dual cell (red) defined on the Cartesian grid **(b)** Vertices of a dual cell are shown perturbed

Figure 4. The dual cell is illustrated surrounded by its eight primal cells in the Cartesian grid

1. Parallelism: Because the original dual contouring approach defines the hexahedral data at the primal cell centers, there would be a need to share data for a single hexahedron between neighboring processors. The extension to dual contouring in the primal allows data for every hex to be contained on a single processor.

2. Cell centered data: The data provided by the CTH code is cell-centered. Using the original dual contouring approach first requires interpolation of the data to the primal grid nodes. The proposed approach avoids this additional interpolation step.

Once this procedure is complete, the basic hexahedral mesh can be represented. Figure 9 shows a hexahedral mesh in one of the processor domains where the surface has been described using the *dual contouring in the primal* approach described here. It is clear that element quality is not sufficient at this point, so additional steps are taken to improve the shape of the elements.

2.3 Non-Manifold Resolution

Choosing which cells will become hexahedra by examination of the volume fraction at the cell center alone can result in non-manifold connections between the resulting hexahedra. Given a set of cells C, which forms the set of all cells in the Eulerian grid, a set of cells H, $H \subset C$, such that every $h_i \in H$ was chosen to be converted into hexahedra, a non-manifold connection exists when H contains non-face-connected hexahedra. For example, Figure 6(a) illustrates two non-

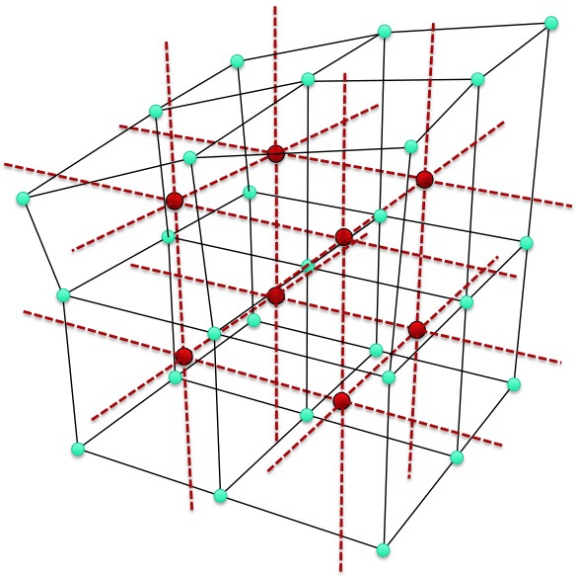
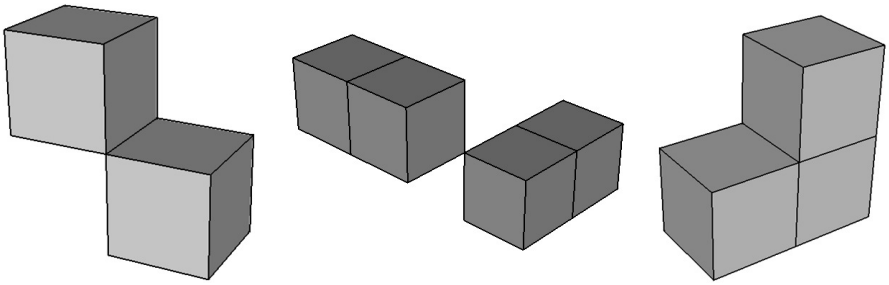


Figure 5. Representation of dual contouring in the primal. Red dotted lines show the dual of the Cartesian (primal) grid. Primal grid nodes are shown modified at the iso-value.

manifold connection cases, while Figure 6(b) illustrates a complete face connected manifold set of hexahedra.



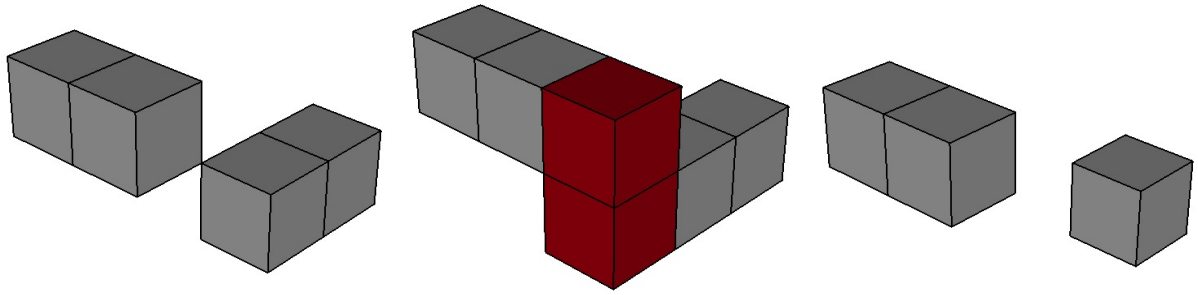
(a) Two cases of non-manifold connections between hexes

(b) Example of manifold connections between hexes

Figure 6. Illustration of non-manifold vs manifold connections between grouping of hexahedra

Non-manifold resolution involves modifying the set of hexahedra, H, to eliminate any non-manifold connections. If a non-manifold connections are not removed, the resulting hexahedral mesh will have element topology which will only admit inverted elements when the buffer layer is inserted.

Non-manifold connections are removed by judiciously adding or subtracting elements from H. For example, Figures 7(b) and 7(c) illustrate two possible resolutions of the non-manifold connection in Figure 7(a). In Figure 7(b), 2 hexahedra are added to H. In Figure 7(c), one hex is removed from H. For a given non-manifold connection, there are several possible ways to resolve a manifold connection through addition or subtraction. The option that will result in the least amount of volume changed, based on the volume fraction data is selected. Additionally, resolution of one non-manifold connections often creates a new non-manifold connection, thus a recursive resolution process is required, which terminates once all non-manifold connections are eliminated.



(a) Initial non-manifold connection that must be resolved

(b) Connection in (a) resolved by adding hexahedra

(c) Connection in (a) resolved by removing hexahedra

Figure 7. Two different approaches shown for resolving non-manifold connections

For parallel execution of our implementation, care is taken with non-manifold connections on or near a processor boundary. Adjacent processor must resolve non-manifold connections on the processor boundary and in the corresponding ghost cells the same way in order to ensure proper mesh connection across the processor boundary. Currently, our implementation uses the following procedure:

1. Each processor resolves all non-manifold connection on its boundaries (i.e. involves elements owned by more than one processor). The in/out status in H of all hexahedra on processor boundaries is then fixed.
2. Each processor resolves all non-manifold connections in ghost cells from other processors, and cells that it owns which are ghosted on adjacent processors. Then the in/out status in H of all ghost cells is fixed.
3. Each processor resolves all non-manifold connections of all interior cells.

The above procedure ensures that consistent resolution of non-manifold connections at processor boundaries is maintained. However, it does yield a different (although still topologically valid) result if the domain decomposition is different where different size processor subdomains are defined. Modifying the non-manifold connection resolution to be independent of processor boundaries is a tractable problem that we leave for future development.

2.4 Geometry Definition

Once the final set of hexahedra are identified, we choose to generate a geometry definition comprised of volumes, surfaces, curves and vertices. The geometry serves as a convenient grouping mechanism for the mesh entities, for subsequent operations on the mesh. In particular, the buffer layer insertion procedure can operate specifically at faces associated with surfaces. The smoothing procedures can also be applied based on nodal surface and volume association. Additionally, the geometric information can be used to define distinct fragment volume information.

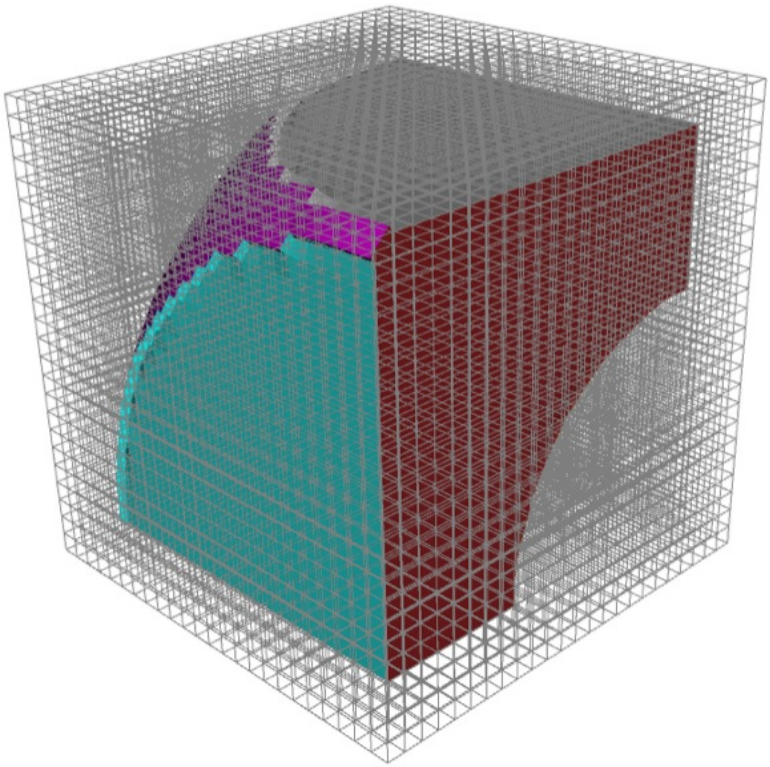


Figure 8. A single processor domain is shown with geometry defined from the hexahedra. Colors represent distinct surface definitions. Curves and vertices are also defined between surfaces.

In our implementation a boundary representation model is generated by first separating groups of contiguous hexahedra into distinct volumes. Surfaces are then defined by skinning the hexahedra in each volume. Sharp features in the surfaces are currently not detected, instead a smooth surface definition is assumed. Distinct surface definitions are however defined at domain boundaries. Figure 8 is an example of surface geometry generated from the hexahedra in a single processor domain. Different colors represent distinct surfaces. Note that surfaces at domain boundaries are also distinguished. In practice, surfaces are defined by collecting groups of grid faces that have a common *face type*. A face type is classified based on its association with one of the six domain boundaries or as an interior face that contains a perturbed primal grid node at the 0.5 iso-value.

Once surfaces are defined, curves and vertices can then be generated by finding the boundary of each surface and curve respectively.

2.5 Buffer Layer Insertion

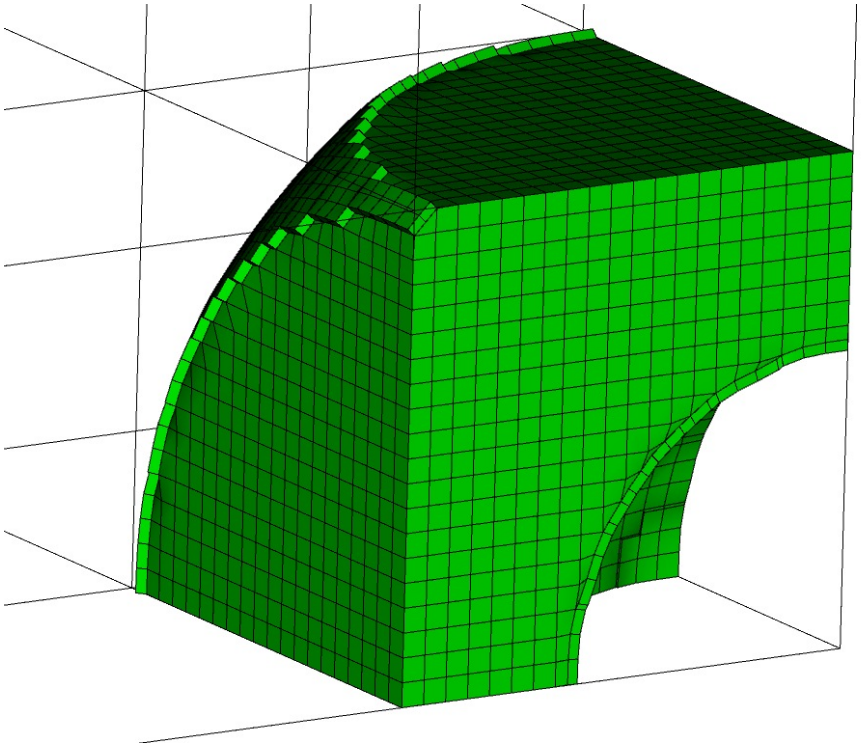


Figure 9. A single processor domain is shown with its hexes defined using dual contouring in the primal. In addition, one layer of hexes have been inserted at the iso-surface boundary. Note that the buffer layer continues through the processor boundary.

Perturbing the primal grid nodes at the iso-value, can result in poor quality hexes near the surface. To help mitigate the problem, a single layer of hex elements is inserted at the iso-surface. To accomplish this, the faces that are associated with the interior iso-surface are used. Each of the nodes on the surface is duplicated and projected to a location along the gradient vector towards the interior of the volume a distance of 1/4 of the grid cell edge length. Connectivity of individual buffer layer hexes can be generated using a surface quad and its duplicated/projected nodes. It should be noted that this procedure is only performed at interior domain surfaces and not at the domain boundaries. This will result in a continuous buffer layer definition across processors. Figure 9 shows an example of a single processor domain with the buffer layer defined. It can also be seen from this figure that the buffer layer insertion alone will result in inverted or poor quality elements. The addition of the buffer layer will permit subsequent smoothing operations to improve the element quality near the surface.

2.6 Smoothing

As noted previously, mesh quality can be poor based only on the base hexes and the inserted buffer layer. Smoothing is a key procedure that can make the mesh acceptable as a computational domain. With mesh entities classified according to their geometry associativity, existing smoothing techniques may be employed to smooth surface and volume nodes. Although the smoothing procedures are still under development at the printing of this report, two main approaches have been explored for surface smoothing:

1. Geometry Based: This procedure utilizes the discrete surface definition we defined in section 2.4 as the basis for smoothing. In this case an iterative Length-weighted Laplacian smoothing procedure is employed. The resulting node location is then projected to the underlying surface facets of the surface. This approach follows the traditional smoothing procedures for CAD-based models and is relatively successful, although currently less efficient than the quadric surface method.

2. Quadric surface approximation: In this method a local quadric surface patch is defined based on the local neighborhood of the current node being smoothed. While this method also utilizes a length-weighted Laplacian operation, its distinguishing feature is the smooth surface fitting. Rather than using the discrete surface representation, the surface is implicitly defined at each node by using a least squares minimization to compute the coefficients of a quadric surface patch. This method appears to provide a smoother surface representation and is currently more efficient than using the direct discrete surface evaluations. References [5] and [20] describe the basic equations for defining the local least squares quadric approximation of the surface.

Figure 10 shows an example of smoothing based on the quadric surface approximation method for one processor domain. Note that smoothing is also performed on the surfaces defined at the processor boundaries so that nodes are projected to its planar surface definition. It is clear that future work will need to be done in order to remove the artificial constraints of the processor boundaries so that nodes can float between processor domains to achieve optimal mesh quality and be independent of the chosen domain decomposition.

Also illustrated in Figure 11 is the smooth and continuous surface definition defined across processor boundaries. This is a combined result of the dual contouring node positioning procedure and the smoothing procedures described above.

Smoothing of nodes inside the volume is also currently performed using a length-weighted Laplacian procedure. As is well-known for this type of smoothing, although it can be very efficient, there is no guarantee that mesh quality will indeed improve. For many of the example models that have been tried to date, the interior mesh quality is still not sufficient for analysis purposes. Future work will need to focus on improving the volumetric smoothing capabilities beyond a simple Laplacian smoothing technique.

Another smoothing issue yet to be addressed is the smoothing of nodes on curves. Curve definitions defined at processor boundaries, like surfaces, can also impose artificial constraints to the mesh. Future research will also need to address positions of nodes that are associated with curves.

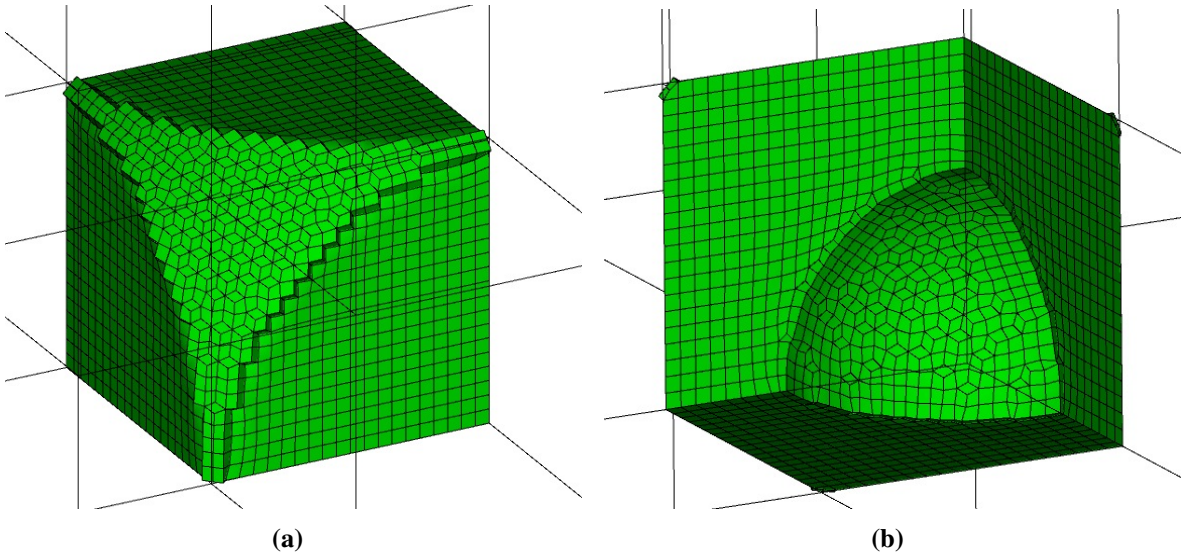
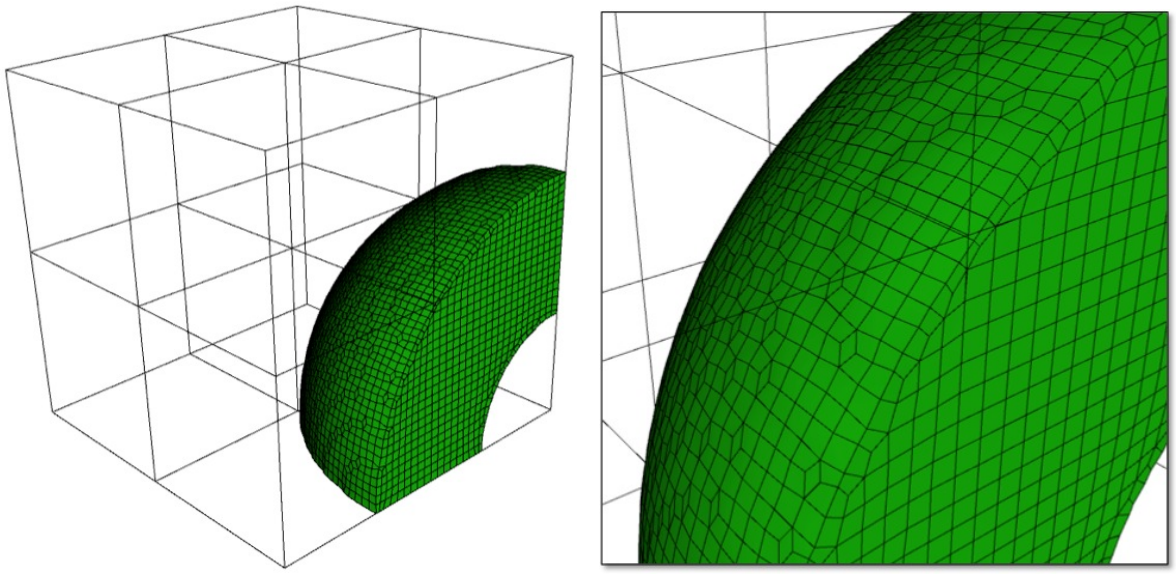


Figure 10. Two views of a processor domain illustrating that surfaces meshes have been smoothed

2.7 Refinement

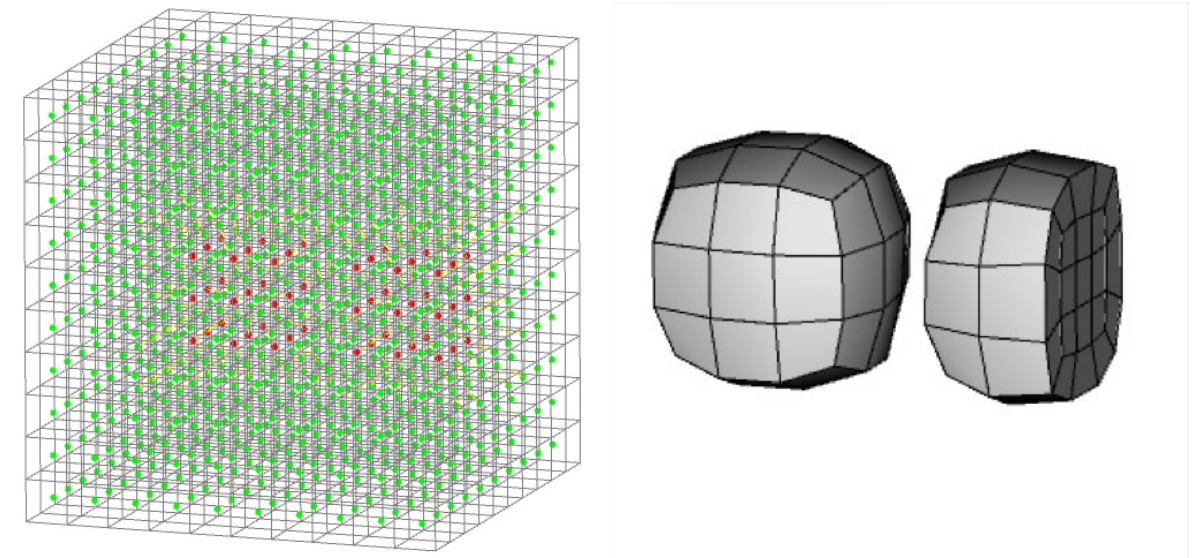
In many instances, the volume fraction data provided by the Eulerian code is not sufficient to adequately describe the shape of a fragment. In addition, Eulerian codes, such as CTH will provide automatically refined (AMR) data at multiple resolutions. For the CTH case, blocks or subdomains are uniformly refined. This will result in hanging nodes or discontinuities in the grid at the boundaries of the subdomains. The dual contouring approach we are using, however requires a conformal base mesh in order to operate. This required development of a refinement procedure that would maintain conforming cells throughout the base mesh prior to applying the dual contouring algorithm.

Previous work on conformal hexahedral refinement [11] was used as the basis for refinement of the base Cartesian grid. The main issue however, was the definition of the data structures needed for general unstructured refinement and the regular Cartesian grid data structure. The Cartesian grid has the advantage that it can be represented as a lightweight set of arrays where all grid entities such as vertices, edges, faces and cells can be represented implicitly. For example, an individual face does not have an allocated memory footprint, rather it is referred to only by an ID that is computed as it is needed. The advantage to this implicit data definition is that huge data sets can be represented with a very small memory footprint. The unstructured data structures required by



(a) **(b)**

Figure 11. View of multiple processor subdomains illustrating continuity of the surface definition following smoothing

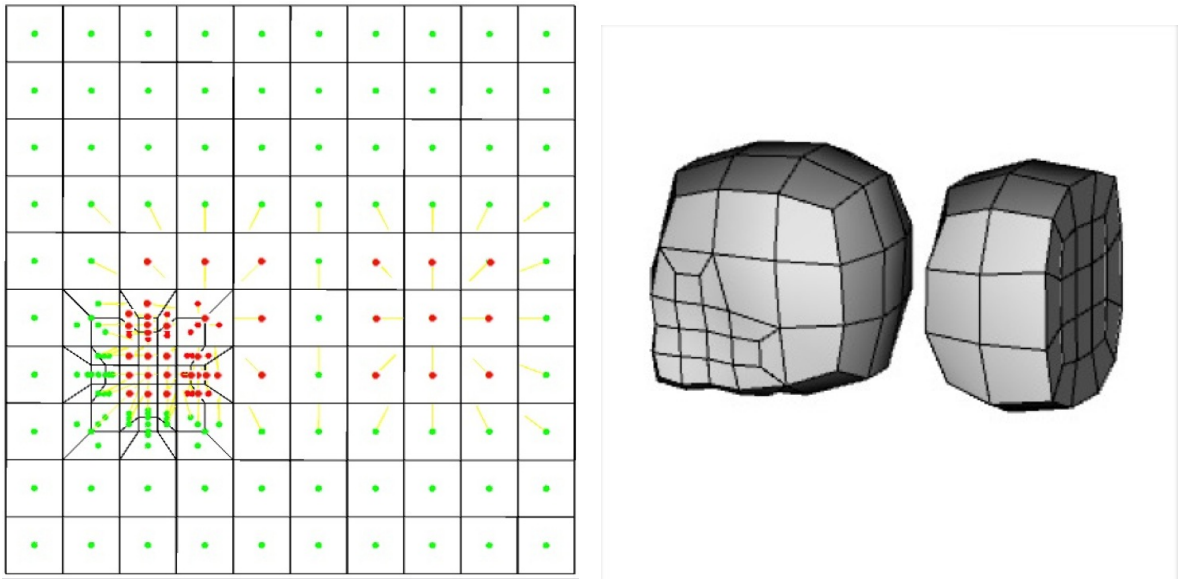


(a) Cartesian grid showing cell-centered volume fractions (red = 1.0, green = 0.0)

(b) Resulting hexahedral mesh from volume fraction data at left

Figure 12. Simple example of a regular Cartesian grid and its volume fraction data. The resulting hexahedral mesh is shown

refinement however require a full data description in order to fully represent the connectivity. The challenge we had was to integrate these two data definitions so that the dual contouring procedure could operate without the need to define special case operations for refined data and another for regular Cartesian data. It is clear that we could have converted the Cartesian data into a fully unstructured data when refinement was necessary, however, we make the assumption that only small portions of the grid will need to be refined for any given problem. This avoids ballooning the data footprint for the entire grid and restricts the additional data only to the localities where it is required.



(a) Side view of refined Cartesian grid with interpolated cell-centered volume fractions

(b) Resulting hexahedral mesh from the volume fraction data at left

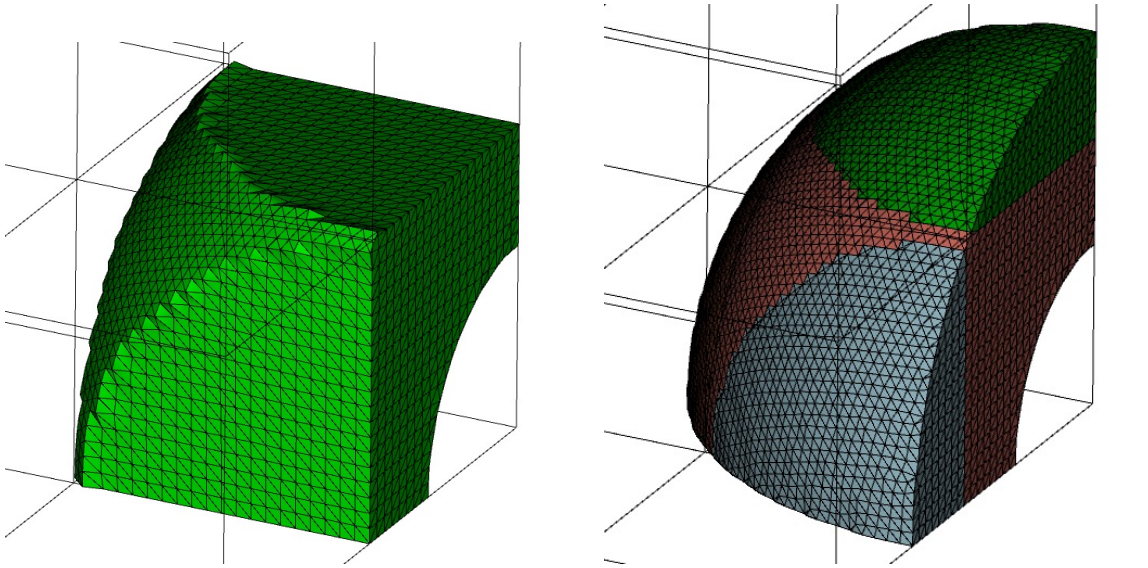
Figure 13. Cartesian grid from Figure 12 with selected cells refined and its resulting hexahedral mesh

Figure 12(a) shows a representation of a regular Cartesian grid where cell-centered data is color coded based on its value. In this simple example, we have defined two local sets of cells where the volume fraction is 1.0. The remaining cells are defined as 0.0. The resulting hexahedral grid from this simple configuration is shown in 12(b). Note that since no attempt is made to capture sharp features from the grid that the gradient approximations result in rounding and bowing of the otherwise brick shapes.

To illustrate the refinement procedure, Figure 13(a) shows a side view of the same Cartesian grid where a few cells have been selected for refinement. Note that the resulting base grid now has a refined region with conformal transition hexes to the courser regular Cartesian definition. We also note that Figure 13(a) shows the additional cell-centered data that can now be used to enrich the final hexahedral grid definition. The values used for this new volume fraction data must be interpolated from its parent grid cells. It is not clear at this point how this data should be interpolated, however several methods have been tried including an inverse distance weighted

interpolation of nearby cells. For the example in Figure 13, we simply use the volume fraction data at the parent cell as the data for the refined child cell without interpolating, which seemed to achieve the desired results. Figure 13(b) shows the resulting hexahedral mesh resulting from the refined Cartesian grid. Note that the resulting hexahedral shapes in the refined region conform better to the original data definition.

To date, the ability to identify specific cells for refinement prior to the dual contouring procedure has been implemented and integrated within Sculptor. Future work will need to apply the refinement procedure adaptively based on the interpretation of local volume fraction data. Ideally we would like to minimize the use of refinement, instead preferring to resolve features using the non-manifold resolution techniques described in section 2.3, however it is unlikely that all cases will be handled adequately by adding and subtracting cells alone from the hex set. Ideally an adaptive procedure that can determine when to use non-manifold and when to use a refinement solution should be investigated. In addition, in order to manage AMR data from CTH, adaptive refinement will be required to capture varying resolution of the grid.



(a) Tetrahedra generated on a single processor **(b)** Resulting tetrahedra on multiple processors

Figure 14. Illustration of tetrahedral mesh generated in parallel

2.8 Tetrahedral Meshing

The ability to generate an unstructured tetrahedral mesh was also included in this work. The general procedures defined in section 2.1 are also applicable to generating a tetrahedral mesh with a few minor exceptions. Items 1 through 8 in this list are also necessary for tet mesh generation. Once a geometry definition has been defined, the surface facets can be used as the basis for a tet mesh. In our implementation, we use the Tetmesh-GHS3D [19] tetrahedral meshing algorithm

which is part of the CUBIT, CAMAL software. Once a watertight set of facets have been extracted from the geometry definition, surface smoothing procedures described in section 2.6 can be employed to improve surface element quality. The smoothed facets can then be sent to the tet mesher to define a boundary conforming mesh. Figure 14 shows an example of a tetrahedral mesh defined from the dual contouring procedures.

2.9 Parallel Processing

It is clear that some of the algorithms described in this report are a work in progress, requiring additional research and development. Enhancing the Sculptor's results when run in parallel is an area where significant additional research is required. The contributions to date in this area, as we will describe, include the identification of the full neighborhood of all subdomains, (not just face neighbors), and the communication of ghost cell information between neighbors.

The primary source of input for the Sculptor is CTH, which is a massively parallel Eulerian code that computes the volume fraction input to the Sculptor module. Figure 15 illustrates a 2D example of the data that comes from CTH. In this example, each processor owns a 6x6 grid of cells. Each processor also shares one layer of ghost cells, represented as grey cells in Figure 15, with all of its' neighboring processors. In addition, the green arrows indicate that each processor is aware of the processors that are at its edge boundaries. However, information provided by CTH does not provide information on diagonal or vertex neighbors. For example, in Figure 15, processor 1 is aware of processor 2 and 3, but not aware of processor 4. For 3 dimensional problems, this means that each processor is aware of any neighboring processor that it shares a face with, however, it is initially unaware of any processor that it shares only an edge or a vertex with.

Although CTH provides one layer of ghost cells, Sculptor requires a full two layers of ghost cells from all of its neighboring processors. The two layers of ghost cells are required for accurate interpolation of derivatives as described in section 2.2 of this report. As a result, the Sculptor must discover its edge and vertex neighbors. The Sculptor code discovers its edge and vertex neighbors using the following procedure:

Given a CTH data model on P processors:

1. Each processor, $p_i, i \in [1,P]$, builds a list, $N_i = \{n_1, ...n_m\}$, of its M face neighbors

2. Each processor sends (MPI_Send) its list, N_i, to every neighbor in N_i.

3. Each processor receives (MPI_Irecv) from each of its face neighbors, $n_j, j \in [1,M]$, that neighbors face neighbor list, N_j.

4. Each processor builds a new complete neighbor list, C_i, by merging its list, N_i, with each of its neighbors' lists $N_j, j \in [1,M]$. Duplicates are removed. A bounding box check is made to reject any processors in C_i which are not immediately adjacent to p_i.

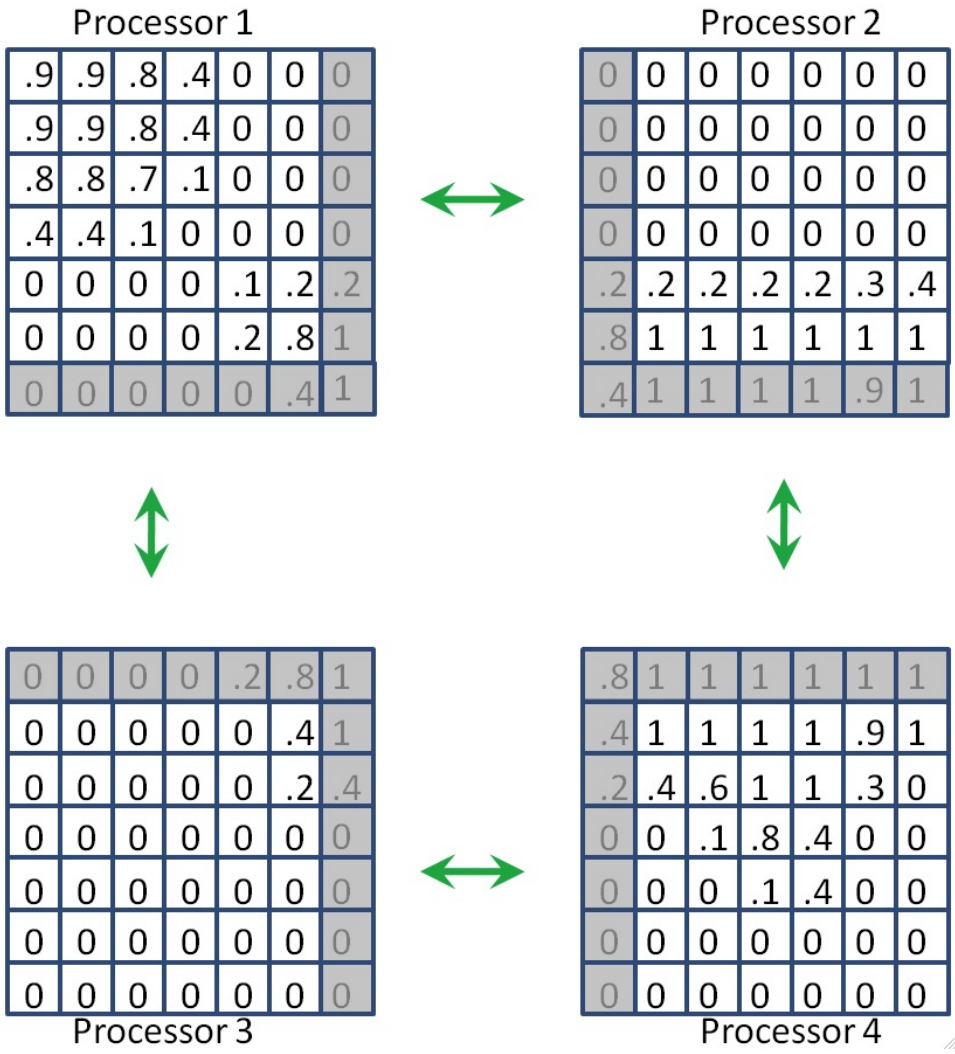


Figure 15. A 2D representation of volume fraction data on four subdomains as CTH (splyplot) provides the data to Sculptor. One layer of ghost cells is provided along with face neighbors (No edge and vertex neighbors provided)

Once each processor knows about all of its neighboring processors, it communicates with each neighbor to extract the full two-layers of ghost cells as illustrated in Figure 16.

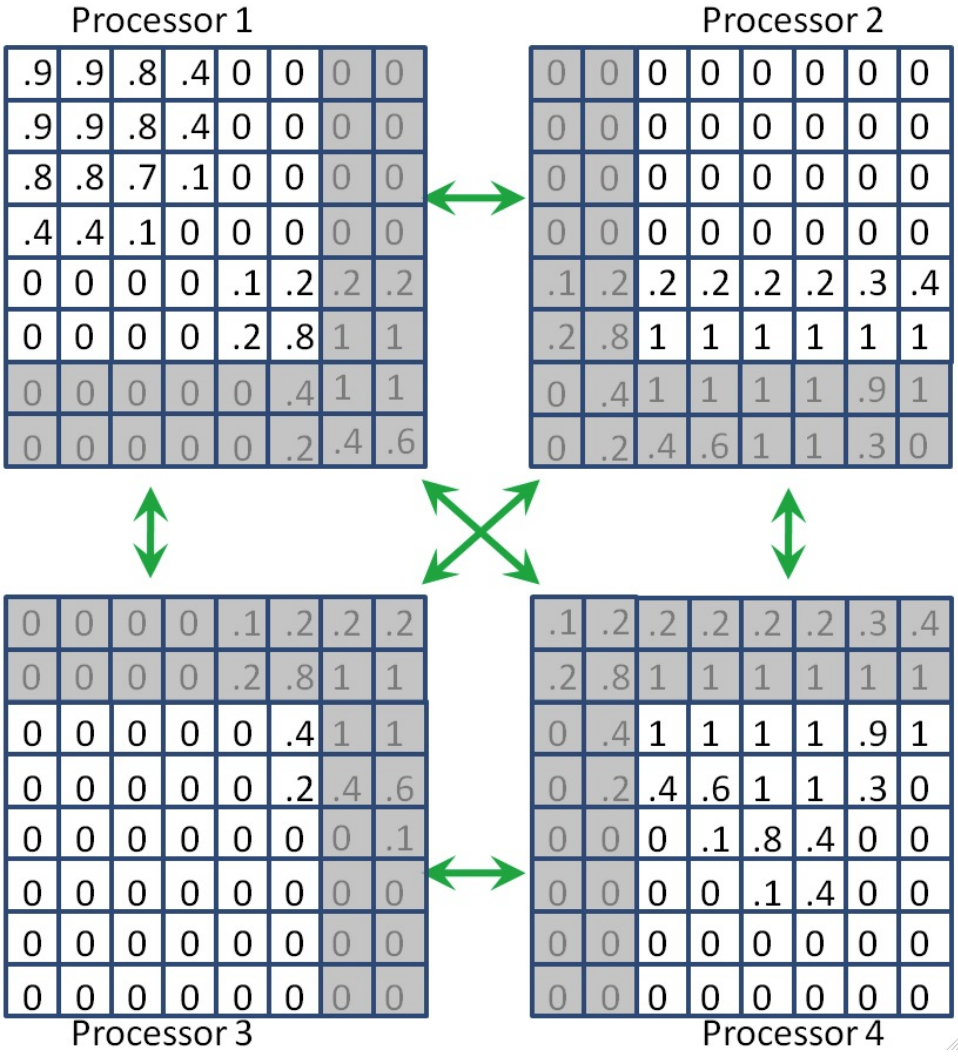
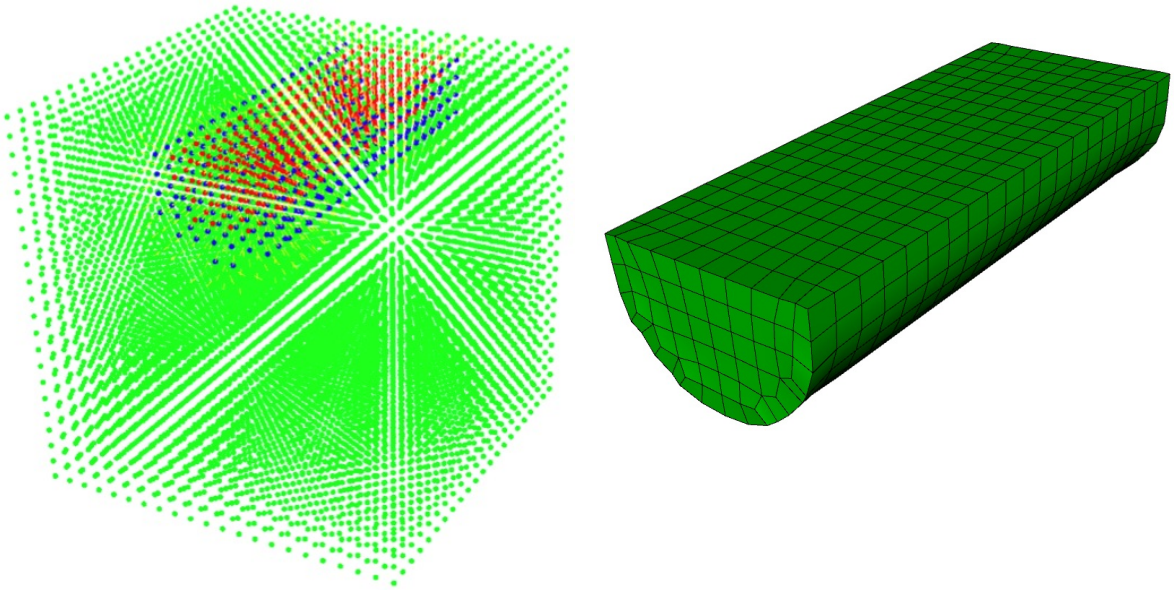


Figure 16. A 2D representation of volume fraction data on four subdomains illustrating data communicated between processors. Two layers of ghost cells as well as its full set of neighboring processor IDs are communicated

3 Examples

In this section a few examples are provided to illustrate the capabilities of the Sculptor module. The first shows a simple half-spherical shape that was initially represented as volume fraction data shown in Figure 17(a). Note the buffer layer insertion is present, but exits the volume at the processor boundary



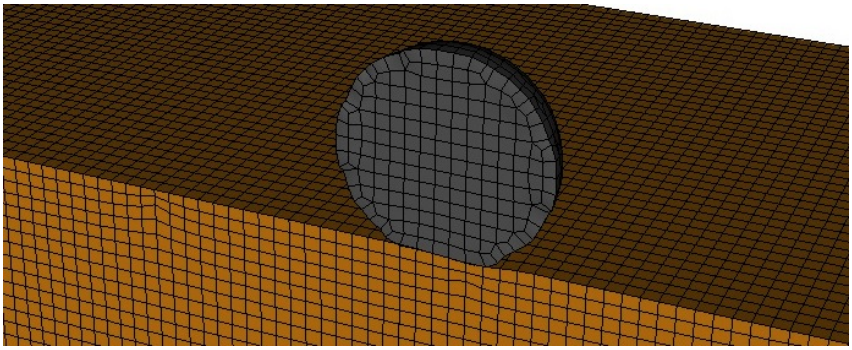
(a) Volume fraction data represented as colored points at cell centers on a Cartesian grid

(b) Hexahedral mesh generated from volume fraction data at left

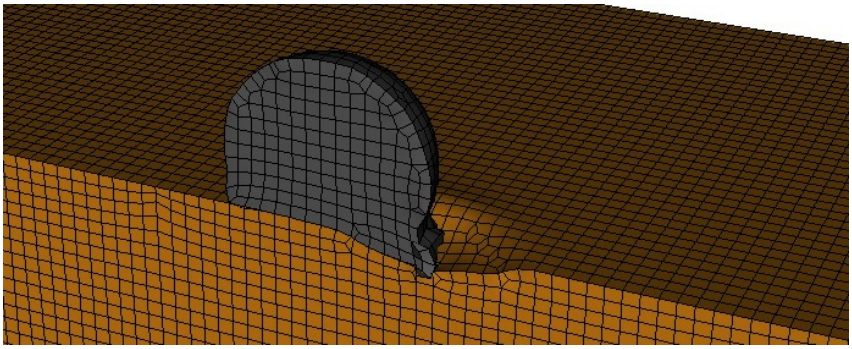
Figure 17. Illustration of volume fraction data describing a half cylinder and its resulting hexahedral mesh

Transient CTH volume fraction data representing the impact of a ball with a plate at an oblique angle was generated on a two-processor system. The Sculptor module was called on two processors to generate the corresponding hexahedral meshes at critical time-steps. Figure 18 illustrates the impact at four timesteps. The grey and orange colors represent the two material zones of the ball and plate. In Figure 18(d), the ball breaks into multiple fragments, each of which is represented by a separate set of hexahedral elements. The impact also leaves a depression in the plate as illustrated in Figures 18(b), 18(c) and 18(d).

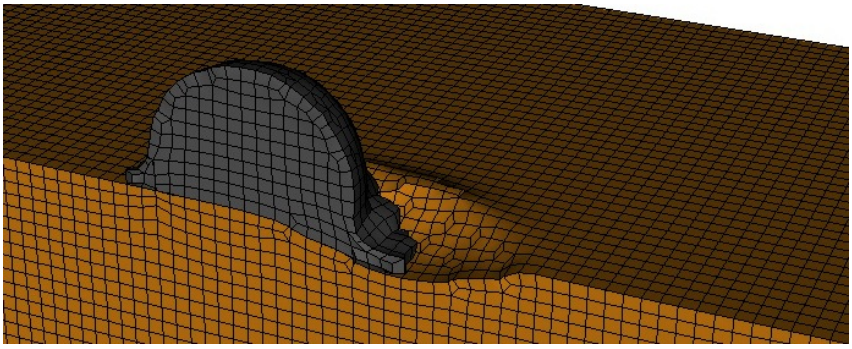
Figure 19 represent a static volume fraction data set representing a variety of different shapes generated with the diatom capability in CTH. The hexahedra were generated on a single processor and illustrate the ability of the Sculptor module to capture relatively complex features. Each color represents a different material as defined by the spyplot data. Figure 20 illustrates the same CTH diatom shapes data meshed with tetrahedra. In this example approximately 259,000 tetrahedra were generated with all positive jacobian shape metrics.



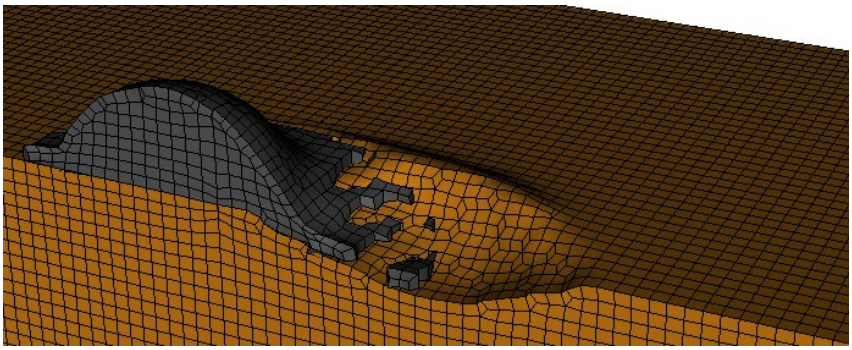
(a) time = 00060



(b) time = 00097



(c) time = 00122



(d) time = 00172

Figure 18. Hexahedral meshes generated at multiple time steps from transient CTH volume fraction data representing the impact of a ball with a plate at an oblique angle

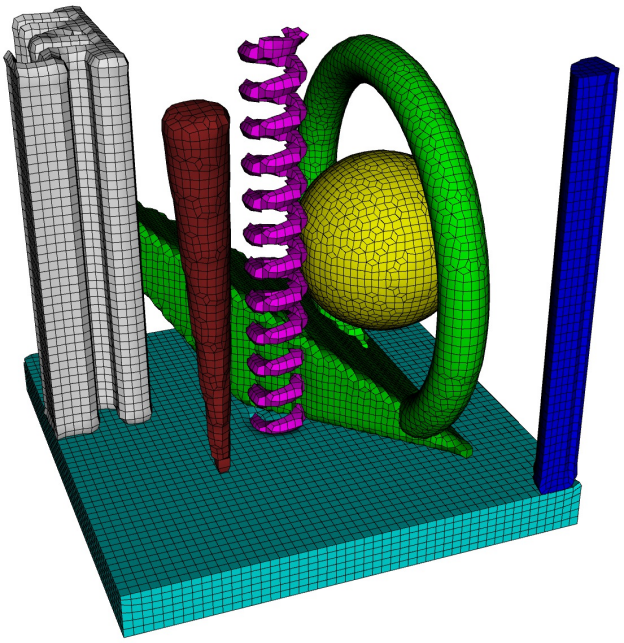


Figure 19. Hexahedral meshes generated from CTH diatom shape data

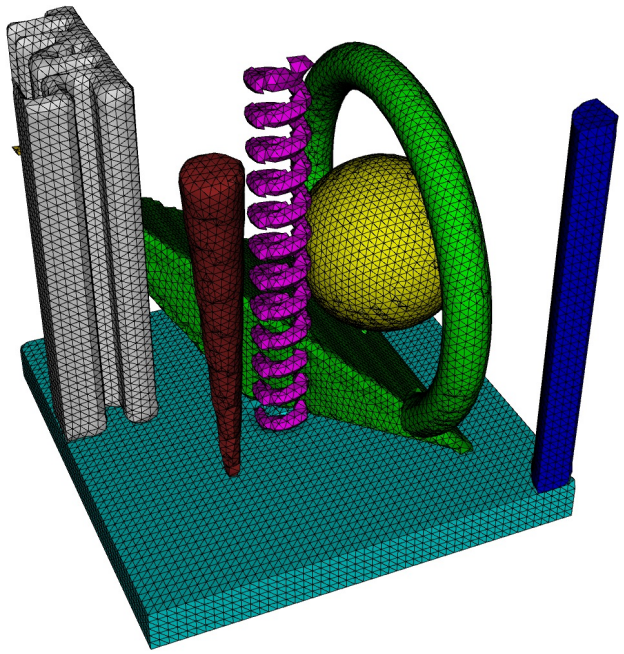
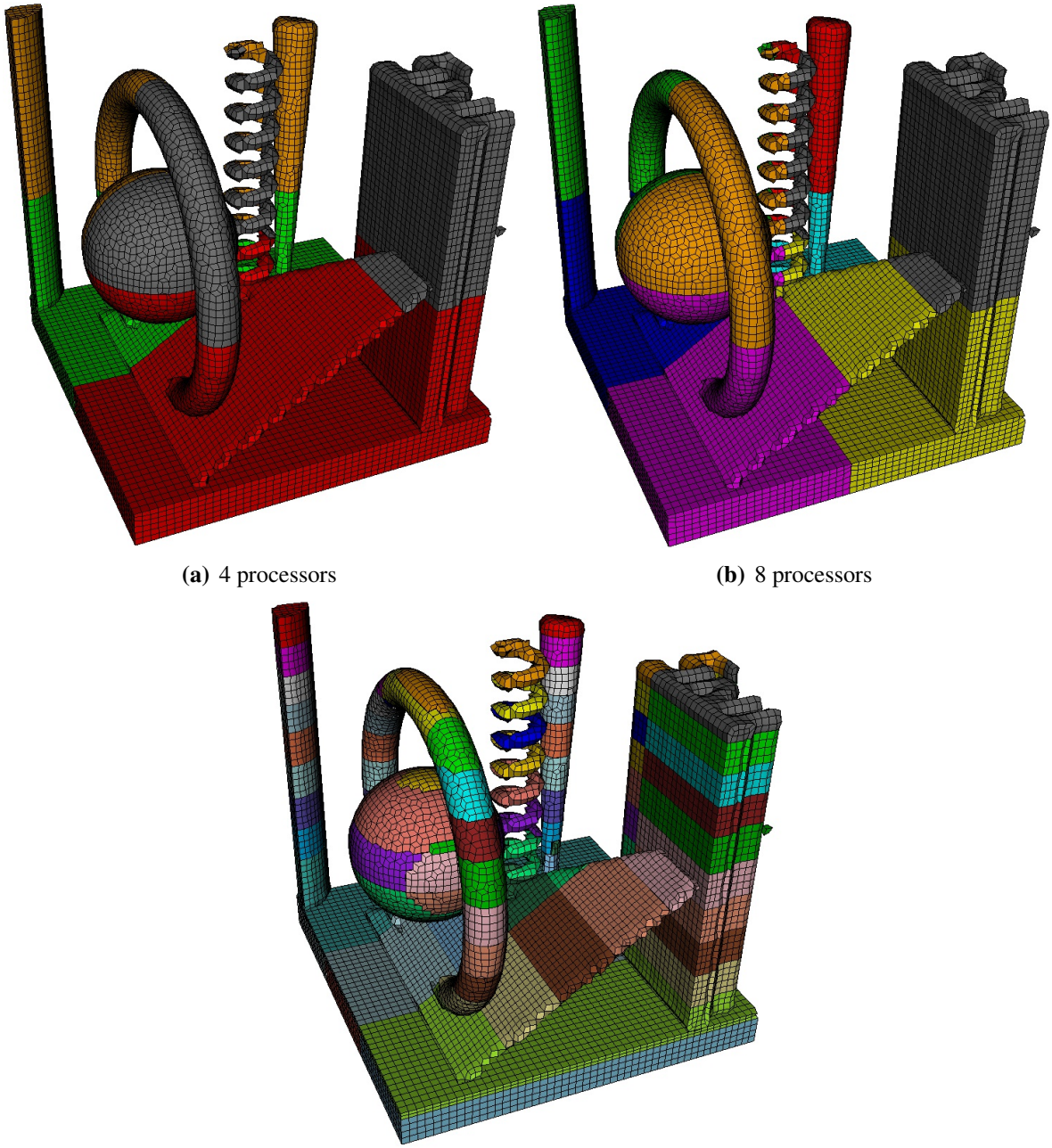


Figure 20. Tetrahedral meshes generated from CTH diatom shape data

Figure 21 illustrates the ability of the Sculptor code to run on multiple processors. The colors represent the domain decomposition created by CTH. The data and hexahedral meshes in Figure 21 were generated by running CTH and the Sculptor on 4, 8, and 33 processors respectively.



(a) 4 processors

(b) 8 processors

(c) 33 processors

Figure 21. Hexahedral meshes generated from CTH diatom shape data at multiple processor resolutions

4 Conclusion and Future Work

This work has introduced new procedures and methods for generating both hexahedral and tetrahedral meshes from volume fraction data defined on a Cartesian grid. We recognize that with the limited resources for this project, that there are many areas left to explore. We would anticipate that the results of such research would move this technology towards a tool that can be robustly used for coupling Eulerian and Lagrangian codes. We do however offer that these methods improve on existing techniques proposed in the literature particular as they apply to parallel mesh generation using overlay grid methods.

4.1 Future Work

The algorithms and methods discussed here are clearly only a start to the process of building a robust system for generating both hex and tet meshes from volume fraction data. In this report we have identified specific areas that will need additional study and development. Here we enumerate the proposed areas of future R&D.

1. *Smoothing*: Improved methods for smoothing will need to be addressed. Four specific areas have been identified: (a) Curve smoothing has not yet been addressed. Curves at boundaries remain fixed in the current implementation. Work will need to be done to better represent curves on processor boundaries. (b) Progress has been made on surface smoothing, however a solution that involves a combined geometry based solution with local quadric interpolation would be beneficial. (c) Current volume smoothing utilizes a simple Laplacian-based smoothing scheme. Since this does not yet address the mesh quality issues we are seeing in the mesh, further work will need to be done to identify and implement improved volume meshing methods. (d) Smoothing the mesh in parallel is a significant concern. Subdomain boundaries currently impose artificial constraints on the mesh. This will require new methods for processor communication that will focus on improved iterative smoothing techniques across processor boundaries.

2. *Non-Manifold Resolution*: The current solution implemented for non-manifold resolution in some may result in different solutions based on where processor boundaries are imposed. New processor-independent methods for non-manifold resolution will need to be developed.

3. *Refinement*: In this work we have developed a capability for refining the Cartesian grid for use in the dual contouring procedure. This capability now needs to be incorporated into an adaptive process where regions requiring additional resolution are automatically identified and refinement operations applied. The current work is also incorporating the 3-refinement scheme developed by Parrish [11]. Current work on 2-refinement [2] should be incorporated to allow for less drastic local mesh size changes common with 3-refinement. Another driver for automatic refinement will be the need to incorporate AMR data from CTH where regions of the mesh will have higher resolution than others.

4. *Buffer-Layer Improvement*: Current procedures for hex meshing include a procedure for adding a single buffer layer at the iso-surface boundary. The ability to adaptively insert additional layers or to only improve local mesh quality through topology operations should be explored.

5. *Tetrahedral Meshing*: A tetrahedral mesh generation capability has been provided, however mesh quality improvement for the boubdary mesh will need to be addressed further.

6. *Multiple Materials*: The current implementation of Sculptor allows for a single material definition. Although multiple runs of Sculptor can generate a model with many materials, there is no guarantee of mesh conformity. Managing multiple materials implies that multiple scalar values will be provided for each cell of the Eulerian grid which will be processed simultaneously. New methods will need to be explored to determine how to best interpret the data and how to accurately model interfaces between multiple materials.

7. *Generalized Hexahedral Meshing:* This work has shown that all hex methods for fragment-type data generated from volume fractions is a tractable method for generating an all-hex mesh. Further work should be done to explore how these techniques might apply to a more general hexahedral meshing capability for CAD-based or facet-based models. Integration of the methods proposed by Owen [10] is one avenue to explore.

References

[1] Blacker TD, Meyers RJ (1993) Seams and Wedges in Plastering:A 3D Hexahedral Mesh Generation Algorithm, Engineering with Computers, 2(9):83–93

[2] Edgel, J (2010), An Adaptive Grid-Based All-Hexahedral Meshing Algorithm Based on 2-Refinement, Master's Thesis, Brigham Young University

[3] Garimella R, Dyadechko V, Swartz B, Shashkov M, (2005) Interface Reconstruction in Multi-fluid, Multi-phase Flow Simulations, Proceedings of 14th International Meshing Roundtable 19–32

[4] Ito Y, Shih AM, Soni BK, (2009) Octree-based reasonable-quality hexahedral mesh generation using a new set of refinement templates, International Journal for Numerical Methods in Engineering, 77(13):1809–1833

[5] Jones NL (1990) Solid Modeling of Earth Masses for Applications in Geotechnical Engineering, Dissertation, University of Texas, Austin

[6] Ledoux F, Shepherd JF (2009) Topological and geometrical properties of hexahedral meshes. Submitted to Engineering with Computers

[7] Ledoux F, Shepherd JF (2009) Topological modifications of hexahedral meshes via sheet operations: A theoretical study. Submitted to Engineering with Computers

[8] Kwak DY, Im YT (2002) Remeshing for metal forming simulations - Part II: Three-dimensional hexahedral mesh generation, International Journal for Numerical Methods in Engineering, 53:2501–2528

[9] Noble DR, Newren EP, Lechman JB, (2010) A conformal decomposition nite element method for modeling stationary uid interface problems, International Journal for Numerical Methods in Engineering, 63:725-742

[10] Owen SJ, Shepherd JF (2009) Embedding Features in a Cartesian Grid, Proceedings, 18th International Meshing Roundtable 117–138

[11] Parrish, M, Borden M, Staten L, Benzley SE (2007), A Selective Approach to Conformal Refinement of Unstructured Hexahedral Finite Element Meshes, Proceedings, 16th International Meshing Roundtable 251–268

[12] Schneiders R, Schindler F, Weiler F (1996) Octree-based Generation of Hexahedral Element Meshes, In: Proceedings of the 5th International Meshing Roundtable, 205–216

[13] Shepherd JF (2007) Topologic and Geometric Constraint-based Hexahedral Mesh Generation, PhD Thesis, University of Utah, Utah

[14] Shepherd JF (2009) Conforming Multi-Volume Hexahedral Mesh Generation via Geometric Capture Methods, Proceedings, 18th International Meshing Roundtable 85–102

[15] Staten ML, Kerr RA, Kerr, Owen SJ, Blacker TD (2006) Unconstrained Paving and Plastering: Progress Update, In: Proceedings, 15th International Meshing Roundtable 469–486

[16] Ju, Tao, Lasasso F, Schaefer S., Warren J, (2002) "Dual Contouring of Hermite Data," Proceedings of the 29th annual conference on Computer graphics and interactive techniques 339–346

[17] Tautges TJ, Blacker TD, Mitchell SA (1996) The Whisker Weaving Algorithm: A Connectivity-Based Method for Constructing All-Hexahedral Finite Element Meshes, International Journal for Numerical Methods in Engineering, 39:3327–3349

[18] Tchon KF, Hirsch C, Schneiders R (1997) Octree-based Hexahedral Mesh Generation for Viscous Flow Simulations, American Institute of Aeronautics and Astronautics A97-32470 781–789

[19] TetMesh-GHS3D, Distine LLC, http://www.distene.com/build/meshing.html

[20] Wang J, Zeyun Yu (2009) A Novel Method for Surface Mesh Smoothing: Applications in Biomedical Modeling, Proceedings 18th International Meshing Roundtable 195–210

[21] Yin J, Teodosiu (2008) Constrained mesh optimization on boundary, Engineering with Computers 24:231-240

[22] Zhang H, Zhao G (2007) Adaptive hexahedral mesh generation based on local domain curvature and thickness using a modified grid-based method, Finite Elements in Analysis and Design, 43:691–704

[23] Zhang Y, Bajaj CL (2006) Adaptive and Quality Quadrilateral/Hexahedral Meshing from Volumetric Data. Computer Methods in Applied Mechanics and Engineering 195: 942–960

[24] Zhang Y, Hughes TJR, Bajaj CL (2007) Automatic 3D Mesh Generation for a Domain with Multiple Materials. Proceedings of the 16th International Meshing Roundtable 367–386

A Sculptor Interface

Sculptor has been developed as an independent module that is callable from an external application. To date, two applications use the Sculptor module to link with: CUBIT and CTH. A C-style API has been developed for these applications to interface with Sculptor. This appendix documents the current Sculptor API.

The Sculptor module has been implemented inside of the Cubit CAMAL library as CMLPSculptor. The header file to include is CMLPSculptor.hpp for a C++ interface, or CMLPSculptor_c.h for a C interface.

A.1 Initialization of the Sculptor

The CMLPSculptor must be initialized with the following function:

```
  // construct a psculptor object.  Return a handle to it.
int c_create_cml_psculptor( void );
```

Once the mesh is generated and extracted, the psculptor must be deleted with the function:

```
  // delete the psculptor object
  // Arg  handle  The handle to the psculptor to delete
void c_delete_cml_psculptor( int handle );
```

A.2 Mesh Generation by Directly Linking with CMLPSculptor

Once created, there are two methods to generate meshes with the sculptor. The first method is by linking CMLPSculptor directly into another program. This has been implemented in CTH. CTH passes the data directly to CMLPSculptor for the volume fraction data owned by the current processor, along with face-neighbor information to allow communication to extract ghost cells. Currently this method allows for meshing only a single material at a time. A single function call passes in the volume fraction data and generates the mesh:

```
  // create the mesh with the given inputs.
  // ARG  handle  The handle to the psculptor to mesh in.
  // ARG  xmin, ymin, zmin, xmax, ymax, zmax - the bounding box of the domain
  //                                           owned by this processor.
  // ARG  xint  The number of grid cells in the x direction
  // ARG  yint  The number of grid cells in the y direction
  // ARG  zint  The number of grid cells in the z direction
  // ARG  vfrac  Array of volume fraction data for each cell in the grid.
  //               length = xint*yint*zint
  // ARG  neighbor_plus_i      Rank of processor on the +i side of this processor.
  // ARG  neighbor_negative_i  Rank of processor on the -i side of this processor.
  // ARG  neighbor_plus_j      Rank of processor on the +j side of this processor.
```

```
  // ARG  neighbor_negative_j  Rank of processor on the -j side of this processor.
  // ARG  neighbor_plus_k      Rank of processor on the +k side of this processor.
  // ARG  neighbor_negative_k  Rank of processor on the -k side of this processor.
  // ARG  auto_refine  1 to allow local refinements
  // ARG  stair_step  1 to force simple conversion of cells to hexahedra without
  //                   projecting nodes to iso-surface.
  // ARG  smooth  1 to perform node relocation aposteriori to improve element
  //               quality.
  // ARG  tet_mesh  0 to create a hex mesh, 1 to create a tetmesh.
int c_mesh_from_data( int handle,
                      double xmin, double ymin, double zmin,
                      double xmax, double ymax, double zmax,
                      int xint, int yint, int zint,
                      double *vfrac,
                      int neighbor_plus_i,
                      int neighbor_negative_i,
                      int neighbor_plus_j,
                      int neighbor_negative_j,
                      int neighbor_plus_k,
                      int neighbor_negative_k,
                      int auto_refine,
                      int stair_step,
                      int smooth,
                      int tet_mesh );
```

The SpyPlot files are output from CTH one per processor containing the volume fraction data for the cells owned by that particular processor. The SpyPlot file also contains data about processor face neighbors to allow communication to extract the required two layers of ghost cells.

The use of SpyPlot file input to the Sculptor has been implemented by writing a stand alone MPI program, which calls the CMLPSculptor once on each processor with the SpyPlot file output from CTH on that processor. This makes debugging of CMLPSculptor internal algorithms easier than when linked directly into CTH.

A.3 Retrieving the Generated Mesh

The mesh generated by CMLPSculptor can be extracted either in bulk, or fragment by fragment. Extracting in bulk returns the entire mesh regardless of the number of distinct fragments the mesh might represent. The functions to extract the mesh in bulk are:

```
  // retrieve the size of the mesh generated from the volume fraction meshing
  // ARG  handle  the handle to the sculptor to query
  // ARG  num_nodes  the number of nodes created by the sculptor
  // ARG  num_hexes  the number of hexes created by the sculptor
  // ARG  num_tets  the number of tets created by the sculptor
void c_get_free_totals( int handle,
                        int *num_nodes,
                        int *num_hexes,
                        int *num_tets );

  // retrieve the coordinate and connectivity array for the volume fraction hex mesh
  // ARG  handle  the handle to the sculptor to query
  // ARG - coords  node coordinates (array size=3*num_nodes)
  // ARG - hexes  - connectivity of hexes (array size=8*num_hexes)
int c_get_free_hexes( int handle,
                      double *coords,
```

```
                      int *hexes );

  // retrieve the coordinate and connectivity array for the volume fraction tet mesh
  // ARG  handle  the handle to the sculptor to query
  // ARG - coords  node coordinates (array size=3*num_nodes)
  // ARG - tets   - connectivity of tets (array size=4*num_tets)
int c_get_free_tets( int handle,
                     double *coords,
                     int *tets );
```

Alternatively, the resulting mesh elements might represent several separate pieces, such as the break up of the ball in the oblique impact example in the Examples Section 3. These pieces can be extracted one at a time with the functions:

```
  // return the number of distinct fragments generated
  // ARG  handle  the handle to the sculptor to query
int c_get_num_fragments( int handle );

  // return the number of nodes, hexes, and tets generated for a given fragment
  // ARG  handle  the handle to the sculptor to query
  // ARG - ifrag - integer from 1 to number of fragments returned from
  //               get_num_fragments
int c_get_fragment_totals( int handle,
                           int ifrag,
                           int *num_nodes,
                           int *num_hexes,
                           int *num_tets );

  // return the coordinate and connectivity array for a given fragment
  // ARG  handle  the handle to the sculptor to query
  // ARG - ifrag - integer from 1 to number of fragments returned from
  //               get_num_fragments
  // ARG - coords  node coordinates (array size=3*num_nodes)
  // ARG - hexes  - connectivity of hexes (array size=8*num_hexes)
int c_get_fragment_hexes( int handle,
                          int ifrag,
                          double *coords,
                          int *hexes );

  // return the coordinate and connectivity array for a given fragment
  // ARG  handle  the handle to the sculptor to query
  // ARG - ifrag - integer from 1 to number of fragments returned from
  //               get_num_fragments
  // ARG - coords  node coordinates (array size=3*num_nodes)
  // ARG  tets   - connectivity of tets (array size=4*num_tets)
int c_get_fragment_tets( int handle,
                         int ifrag,
                         double *coords,
                         int *tets );
```

B CUBIT Sculptor Users Manual

The CUBIT Geometry and Meshing Toolkit provides a convenient graphical user interface to the Sculptor module. CUBIT provides the ability to read both Exodus and Spyplot data files. The Exodus file must define a basic hexahedral mesh with element-based scalar data provided. For our purposes we limit the hex mesh to a simple Cartesian grid structure. In addition, a custom dump of data from a CTH run can be exported from the Spyplot module. Although not a standardized format, it provides a convenient way to import data into CUBIT so that the data can be meshed and visualized. This section documents the CUBIT commands relevant for the Sculptor and provides a few examples for generating meshes in cubit using the Sculptor module.

B.1 CUBIT Commands

Note that these commands are currently under development so it is necessary to issue the following command in Cubit to gain access to the Sculptor module:

set dev on

The command for reading data from a spyplot data file:

import volume fraction "$\langle filename.txt \rangle$**"**

where $\langle filename.txt \rangle$ is the name of a splyplot data file. All materials and data will be read from the file and stored in cubit. A listing of the material IDs in the data file will be displayed on import.

The command for reading data from an exodus file:

import volume fraction "$\langle filename.txt \rangle$**" variable "**$\langle varname \rangle$**" [time** $\langle timeval \rangle$**]**

where $\langle filename.txt \rangle$" is the name of an exodus file that contains blocks of hexahedra containing element-based scalar data. $\langle varname \rangle$ defines a material identifier that is used in the exodus file and the $\langle timeval \rangle$ is an optional argument to specify a specific timestep present in the exodus file.

To generate a mesh once the data has been read in to CUBIT, the following command is used:

mesh volume fraction [material $\langle matid \rangle$**] [stairstep] [no_smooth] [smooth** $\langle smoothid \rangle$**] [tetmesh]**

The basic command without optional arguments will generate a hexahedral mesh of the first material that was read from the data file. The optional arguments are defined as follows:

material $\langle matid \rangle$ The mesh will be generated for the specified material ID

stairstep A stairstep hex mesh will be generated with no attempt to define a smooth surface.

no_smooth Turn off all smoothing in the mesh.

smooth ⟨*smoothid*⟩ Define the smoothing algorithm to be used by Sculptor. 0=no_smooth, 1=quadric surface, 2=geometry-based. (See section 2.6)

tetmesh Generate a tetrahedral mesh.

B.2 CUBIT Examples

A hardcoded test function has been embedded in CUBIT to validate the Sculptor capability. The following command should generate the simple mesh shown in Figure 12(b)

mesh volume fraction test

The optional arguments can also be used with this test. or example:

mesh volume fraction test tetmesh
mesh volume fraction test no_smooth

In addition the simple refinement case can be generated by setting the sculptor refine flag

set sculpt refine on
mesh volume fraction test
set sculpt refine off

Some examples shown in section 3 of this document can be generated with the following commands. See the authors to get a copy of the data files. They are also available as part of the CUBIT test suite (cubit_test).

To generate the example in Figure 19

cd "cubit_test/sculpt"
reset
set dev on
import vol frac 'vfblock_data_00001.txt'
mesh vol frac mat 100001
mesh vol frac mat 100002
mesh vol frac mat 100003
mesh vol frac mat 100004
mesh vol frac mat 100005
mesh vol frac mat 100006
mesh vol frac mat 100007
mesh vol frac mat 100008
create mesh geom hex all feature 0

To generate the example shown in Figure 14 use the following commands. Note that this example uses an exodus file as input and generates a tet mesh.

```
cd ”cubit_test/sculpt”
reset
set dev on
import vol frac ’cube_mesh.exo.1.0’ variable ’MAT_10’
mesh vol frac tetmesh
create mesh geom tet all feature 0
```

DISTRIBUTION:

1 Steven Benzley
350-U CB, Brigham Young University
Provo, UT, 84602

1 Ray Meyers
Elemental Technologies
17 Merchant Street, American Fork, UT 84003

1 MS 0387 Steven Owen, 1543

1 MS 0387 Matthew Staten, 1543

1 MS 0387 Ted Blacker, 1543

1 MS 0387 Brett Clark, 1543

1 MS 1321 Roshan Quadros, 1543

1 MS 0387 Byron Hanks, 1543

1 MS 0387 Greg Whitford, 1543

1 MS 1318 Patrick Knupp, 1414

1 MS 1323 Jason Shepherd, 1424

1 MS 0378 David Hensinger, 1431

1 MS 0836 David Crawford, 1541

1 MS 0847 Steve Attaway, 1525

1 MS 0378 Allen Robinson, 1431

1 MS 0822 W. Alan Scott, 9326

1 MS 0380 David Womble, 1540

1 MS 0824 Joel Lash, 1510

1 MS 0899 Technical Library, 9536 (electronic copy)

1 MS 0123 D. Chavez, LDRD Office, 1011

Sandia National Laboratories