Eurographics Symposium on Parallel Graphics and Visualization (2014)
M. Amor and M. Hadwiger (Editors)

Finely-Threaded History-Based Topology Computation

Robert Miller[1] Kenneth Moreland[2] Kwan-Liu Ma[1]

[1]University of California, Davis
[2]Sandia National Laboratories

Abstract

Graphics and visualization pipelines often make use of highly parallelized algorithms which transform an input mesh into an output mesh. One example is Marching Cubes, which transforms a voxel grid into a triangle mesh approximation of an isosurface. These techniques often discard the topological connectivity of the output mesh, and instead produce a 'soup' of disconnected geometric elements. Calculations that require local neighborhood, such as surface curvature, cannot be performed on such outputs without first reconstructing its topology. We present a novel method for reconstructing topological information across several kinds of mesh transformations, which we demonstrate with GPU and OpenMP implementations. Our approach makes use of input topological elements for efficient location of coincident elements in the output. We provide performance data for the technique for isosurface generation, tetrahedralization, subdivision, and dual mesh generation, and demonstrate its use in visualization pipelines containing further computations of local curvature and mesh coarsening.

1. Introduction

Computer graphics algorithms rarely produce output from a vacuum. In general, popular algorithms rely on some input mesh. For example, Marching Cubes [Lor87] transforms a voxel grid into an isosurface. Many of these algorithms share the property that output mesh elements may be entirely determined by a small set of input mesh elements, which allows for simple, effective, formulaic parallelization by splitting the algorithms to work separately on each input mesh element. This naive parallelization generates a "soup" of output primitives, with no connectivity information.

Consider the calculation of the curvature of an isosurface. Given only a triangle soup, there is no simple method to determine the neighbors of any given triangle, which is a required step to compute the curvature. Visualization pipelines can be more complex, so that different types of connectivity information may be necessary depending on the pipeline.

This is not a new problem, and techniques exist to determine topological information about a primitive soup [PCC04]. Generally, this approach starts by finding and coalescing duplicate vertices, which may require a bounded-radius search to resolve vertices differing only by floating-point error. All primitives that share two or more vertices are linked as neighbors. Some "duplicate" vertices may not represent desirable connections, and are split in a final pass.

The process just described is computationally intensive, and is specific to the topological connections of triangles. We present a technique which applies equally well to other types of topological connections, such as determining the neighboring facets of tetrahedra. Some examples supported by our technique are demonstrated in Figure 1.

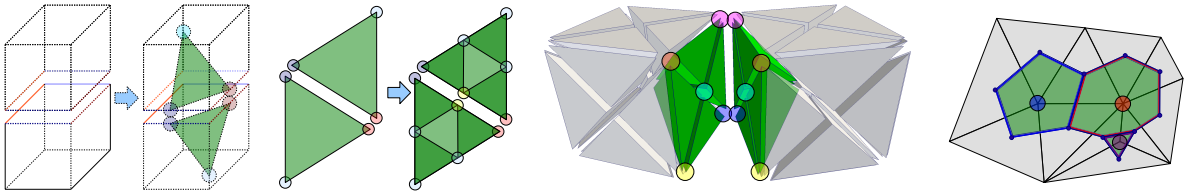
To summarize, this paper makes the following contributions:

- Provide a generalized finely-threaded technique for generation of topological connectivity
- Require no modifications to the algorithms generating geometry save for the storage of information about the input topological features used to generate each output feature.
- Make use of history of disconnected geometry to enhance performance of the topology resolution technique.
- Demonstrate the effectiveness of our technique on several different topology-generating algorithms
- Show how history-aware algorithms can compute mesh coarsening in the same pass as topology resolution.

2. Previous Work

New techniques are required for visualization algorithms to perform efficiently on finely-threaded architectures such as GPUs and expected exascale machines. Several frameworks are under development to tackle these challenges. PISTON [LSA12] approaches this by providing efficient data-parallel operators as building blocks for efficient parallel algorithms.

DOI: 10.2312/pgv.20141083



(a) Marching Cubes (b) Subdivision (c) Face-Centered Tetrahedralization (d) Dual Mesh

Figure 1: Geometry is usually generated from a known input topology. Marching Cubes (a) generates vertices only on the edges of a known voxel grid. We retain connectivity by marking these output vertices with the ID of their generating edges. The same process can be applied to cell subdivision (b). Tetrahedralizations (c) are more complex, but are supportable by keying on input face as well as input edge. Generating a dual mesh (d) exchanges vertex and face connectivity information. In each figure, output topological connections known from input topology are displayed by using identical colors to highlight connected vertices.

EAVL [MAPS12] provides a robust data model and a variety of efficient parallel algorithms. DAX [MAGM11] internally uses our technique in its implementation. DAX aims to ease the transition into exascale by providing both a control environment where programmers work in serial stages and an execution environment tailored for massive parallelism.

We rely on efficient GPU stream compaction for geometry generation, derived from the methods of Horn [Hor05] and Sengupta [SHZO07]. These rely on data parallelism techniques described by Blelloch [Ble90]. For our examples, we use geometry generation methods that make use of the prefix-sum method of compaction, but other compaction techniques such as histogram pyramids [DZT08] could be substituted for the prefix sum technique for increased performance in any case where our method is applied.

There are myriad existing GPU isosurfacing implementations, dating from Rottger's implementation [RKE00]. Dyken provides a detailed overview of advancements in GPU implementations of Marching Cubes/Tetrahedra [DZT08]. The basic topology resolution technique in the introduction section is introduced by Park et al. [PCC04]. Additionally, Kipfer and Westermann [KW05] observe that polygon vertices from Marching Tetrahedra all lie on tetrahedral edges to uniquely specify each polygon vertex. We use a similar observation, but our technique is more general and does not require an auxiliary edge structure such as theirs. We also describe a mesh coarsening method which improves mesh quality. There are existing methods to improve triangle quality from contouring methods [MW91], but when our method is used to generate connectivity we can also coarsen or apply other similar operations for negligible additional cost.

Many geometry generating algorithms simply forward disconnected geometry for rendering [DZT08, GF05, JC06]. However, many serial implementations such as VTK and ParaView determine topological connectivity information from discrete geometry, relying on search structures to do so. There exist similar search structures for finely threaded cases such as the GPU [ZG08]. Because operations like tetrahedralization, contour methods, dual mesh generation, and even simple subdivision can radically alter topology, we can not depend on the reuse of large search structures. Instead, we require a lightweight approach that can generate a new representation of topology quickly using known information from a previous representation.

Other researchers [SCMO10, VBS*11] have implemented efficient visualization algorithms directly in MapReduce [DG08]. Although similar in many ways, our work has several distinguishing features from MapReduce. One is that the traditional MapReduce framework is designed for computing large analysis problems as a batch operation. Thus, it is a high bandwidth but also high latency approach. However, in visualization we usually require a low latency solution. We propose doing this by applying the map-reduce in a localized region. Larger scales can be broken into domains using well-worn techniques like ghost regions to interface partitions [ABM*01]. Another distinguishing feature is that we recognize that while key-value pairs are being generated, other values independent to the first operation can also be generated. Such information is hard to capture in a classic MapReduce framework without significant data replication. We are also describing a specific technique in applying MapReduce. We are specifically demonstrating how to use MapReduce to resolve dependent parallel operations on a mesh. Our algorithms could be implemented in a MapReduce framework, but it would require additional collection operations to resolve connectivity.

3. Using Topology History

The essential idea behind our history-enabled topology construction is to use components of the input topology as *keys* in a MapReduce-like framework, but with an additional partition step as demonstrated in Figure 3.

The *map* operation generates key-value pairs where the key is some component of the input topology and the value is

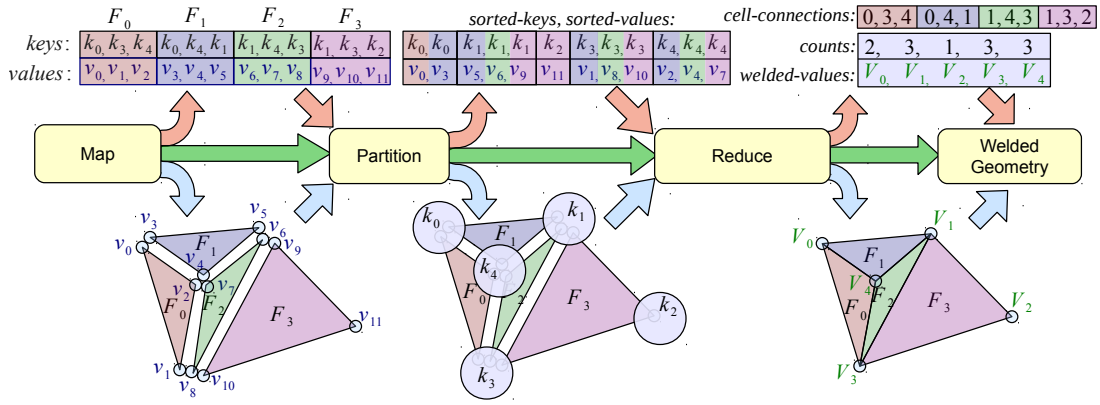


Figure 2: An example of using topology history to find vertex connections. A map operation defines some collection of cells as shown at left. A partition phase groups the keys, then a reduce phase combines these vertices and establishes a connections array to the new indices. We collect the mechanisms of this process in the KEY-REDUCE algorithm.

a generated component of the output topology. In a contouring algorithm, the values are the vertices generated for the new surface mesh, and each vertex is keyed by an identifier for the edge used to interpolate the vertex. The map operation may also generate elements that are known to be unique and therefore do not need keys. For example, when subdividing cells, some vertices come directly from input vertices whereas others are interpolated and must be connected. The map operation must generate key-value pairs independently to be efficiently computed on a very large thread pool.

The second step is a *partition* operation to reorganize the key-value pairs and group duplicate keys. Like most MapReduce implementations, we find parallel sort to be an efficient way to shuffle the data. We can also use domain decompositions when available to shorten partitioning time.

The third step is a *reduce* operation that merges groups of coincident components identified by the partition and generates the connected structures. In contouring algorithms, this reduction consists of averaging of field values on merged vertices and updating the triangle connection indices.

4. Connecting Vertices

History information can be used to identify topology elements of any type, but the most common case we have encountered is the need to find coincident vertices. Many techniques exist for this purpose, as outlined in Section 2, but all operate without prior knowledge of input topology. We show that our history-enabled technique is faster than existing techniques and is less susceptible to some of the inherent limitations of other techniques.

4.1. General Merging Algorithm

Figure 2 provides a simple example of applying the topology history technique, described in Section 3, to find connections among generated vertices. First, a mapping operation generates the connections for a set of cells. Vertices are duplicated to allow independent operation on multiple threads. We assume the map operation is a previously known algorithm such as Marching Cubes with the trivial extension that cell connection lists contain pairs of input index (key) and output vertex (value) rather than just the output vertices.

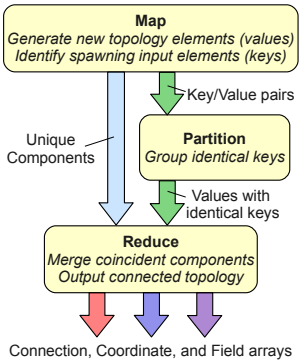


Figure 3: Overview of the flow of an algorithm using our technique.

Next, key-value pairs are partitioned by sorting the pairs based on keys. From the sorted list of keys we can efficiently extract a list of unique keys, which serves to identify the connected vertices to be created. This list of sorted unique keys can also used to look up where each original unsorted key resides in the final list of merged vertices, which is how we generate a *cell-connections* array defining the cell topology.

The final step is to merge values with identical keys in the reduction phase. This reduction operation provides an opportunity to combine neighborhood information such as averaging normals across surface polygons. To facilitate av-

eraging we also generate a *counts* array marking the number of cells incident to each vertex. This array is not necessary to describe the final topology, but it can be leveraged to find vertex incidence lists using the VERTEX-INCIDENCE-LIST method formally described in the supplemental material.

We provide a generalized method that captures these partition and reduce phases named KEY-REDUCE that groups various types of geometric elements, then merges each of these groups into single outputs. The formal definition of KEY-REDUCE is described in the supplemental material.

The KEY-REDUCE algorithm takes the result of a map operation as input. Also passed to KEY-REDUCE are a *merge* operation, which combines two values, and a *transform* operation, which completes a reduction from a fully merged set. Together the merge and transform operations allow reductions to take place iteratively, which can be important for distributed or streaming implementations.

The remainder of this section describes applications of KEY-REDUCE as a typical vertex weld (albeit potentially faster than VERTEX-WELD with a more robust coincident point comparison). Subsequent sections apply the KEY-REDUCE algorithm for other topology generators.

4.2. Marching Cubes

In the case of Marching Cubes/Tetrahedra, we know a good deal about the input topology on which the output is generated. Specifically, the output vertices from Marching Cubes are generated only on the edges of input voxels. To create a history aware Marching Cubes, we start with a standard implementation, but in addition to recording the coordinates of each triangle vertex, we also write out an index identifying the edge on which the vertex lies.

For a structured voxel grid, we assign each unique edge an implicit integer index, then store the corresponding edge index for each vertex generated by Marching Cubes. Because the edge indices are implicit, we need neither to create nor to store an edge list. For unstructured grids, no such implicit edge identifiers necessarily exist. However, each edge is uniquely identified by its two end vertices. We build unique local edge indices by concatenating these two end vertices in a canonical order. This also works for voxel grids, but requires more bits in the indices than necessary and can thus slow down the key sort. The results of this keyed Marching Cubes map are completed using the KEY-REDUCE process described in Section 4.1. We can also merge other field information at the vertices. This is helpful for creating interpolated surface normals from flat triangle normals or input gradients, which are not continuous across cell boundaries.

4.3. Cell Subdivision

Cell subdivision is a simple topology generator which can be used to smooth or improve the representation of a finite element mesh. For example, a triangle can be divided into four sub-triangles as in Figure 1b. As with Marching Cubes, simplex subdivision only adds vertices on edges of the input mesh, so application of KEY-REDUCE for this purpose is similar. One difference is that subdivision reuses all vertices from the initial mesh whose connectivity is known, so only new vertices generated on the edges should be partitioned and reduced, and the others should be passed directly into the output.

4.4. Face-Centered Tetrahedralization

Face-centered tetrahedralization divides each hexahedron into 24 tetrahedra by adding new vertices to each face and the center of the hexahedron. It generates many more tetrahedra than some of its counterpart algorithms, but this tetrahedralization captures the nonlinear field well [CMS06].

Unlike our previous algorithms, face-centered tetrahedralization generates new vertices on faces instead of on edges. To use history information, we merely change how keys are generated. For voxel grids, we can implicitly index the faces is a similar way we indexed edges. For unstructured grids, we create a key consisting of the vertices with the lowest, second lowest, and third lowest indices in that order.

5. Mesh Coarsening

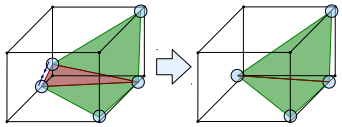


Figure 4: When several Marching Cubes output vertices (blue) fall close to the same input vertex (black), small or skinny triangles may be produced (red).

Marching Cubes generates small or skinny triangles when two adjacent output vertices are generated near the same input mesh vertex as in Figure 4. Skinny triangles lead to a poor sampling of surface normals, shading, and other fields as demonstrated in Figure 5. The juxtaposition of skinny and fat triangles can also cause irregularities in their orientations, which can yield a staircase-like appearance as in Figure 6. The skinny triangles can be eliminated by collapsing nearby vertices. With history-enabled topology generation, this collapsing is simple. Rather than key each output vertex directly on input edge as described in Section 4.2, we key each output vertex by the nearest input vertex connected to the input edge. This simple change partitions all nearby vertices in to one group for the reduce to merge together, thus merging vertex point coordinates and averaging field values.

There exist many surface coarsening algorithms which operate without regard to the mesh's history by either iteratively collapsing features [Pot11] or collectively clustering

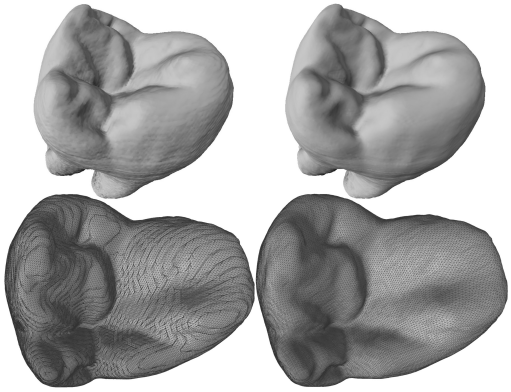


Figure 5: Top: The normal artifacts at left are reduced at right. Bottom: Our coarsening also improves mesh quality.

Figure 6: At left artifacts are visible due to sampling error caused by purely local sampling. At center averaging helps but does not remove the artifacts. At right our coarsening removes the artifacts by providing a better local sampling without additional performance costs.

vertices [DT07]. Our approach is similar except that we piggyback the clustering with the existing partitioning stage of our method and therefore we can coarsen without requiring a bounded-radius nearest-neighbor search or auxiliary search structures. We define triangle quality $Q = \frac{4a\sqrt{3}}{h_1^2+h_2^2+h_3^2}$, where h_x are side lengths and a is the triangle area, and triangles where $Q > 0.6$ are of acceptable quality [BH].

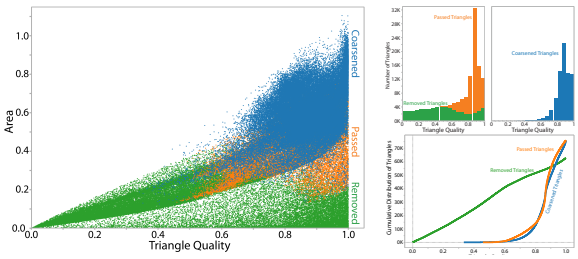


Figure 7: Left: Scatterplot of triangle area and quality for the MRI head dataset. Top Right: Quality histograms of passed, removed, and coarsened triangles. Bottom Right: Cumulative distributions of triangles before and after coarsening.

Figure 7 demonstrates our coarsening on the isosurface from Figure 6. Most triangles that are removed by our coarsening (green) are small or poor quality as compared with other triangles in the distribution. Triangles that are not removed (orange) are altered into the output triangles (blue) when point coordinates are averaged. After coarsening, most triangles are large and high quality. The cumulative distributions show that our coarsening reduces the size of the output by approximately half. The median quality of the removed triangles is approximately $Q = 0.5$ whereas the median quality of passed and output triangles is approximately $Q = 0.85$.

6. Grouping Cell Connections

Thus far, we have discussed algorithms that are able to locally determine the structure of cells but require collective operations to resolve the vertices. However, our technique can also be applied to the converse problem where vertices can be locally determined but the cell structure can only be constructed by tracing through the input topology. The process uses the same strategy discussed in Section 3 and the same KEY-REDUCE process described in Section 4.1. Some input feature is uniquely assigned to each cell in the output, and this feature's index is used for the key. The reduce phase then generates cell connectivity lists.

We demonstrate the concept of using history-enabled topology construction for cell grouping with an algorithm for finding the dual of a surface mesh.The map phase of the dual mesh algorithm (Figure 1d) first finds the centroid of each input polygon, which becomes a vertex in the output dual mesh. Each centroid is known to be unique and can be written directly to a vertex list. The map also writes a key-value pair for each vertex of each cell of the input mesh. The key is the index of the input vertex and the value is a 3-tuple containing the index to the output vertex (generated from the polygon centroid) and the two input vertices adjacent to the keyed vertex and incident to the polygon. These latter vertices describe the edges through which the dual polygon will connect around the keyed vertex.

These key-value pairs cause the reduce phase to merge together the centroids of all cells incident to the keyed vertex. Such a group of centroids corresponds to a polygon in the output mesh. Thus, our merge operation will output a polygon connecting these centroids. However, the partitioning generally will not produce the centroids in the proper order around the polygon, so we use the vertex key and the two incident input vertices stored in the value to sort the generated vertices based on angle from the input edge, thus yielding the correct order.

7. Results

Each test dataset, listed in Table 1, originates as a voxel grid. The 3D unstructured grids are minimal tetrahedralizations

of these voxel grids. The unstructured surfaces are generated by Marching Cubes with isovalues chosen to show relevant features. We attach vertex normals and a scalar color value as associated attributes to each input vertex for surface contour data. Memory constraints prevent tetrahedral tests on datasets larger than 256^3. Our measurement system has an Intel Core i7 975 Processor, which has a clock speed of 3.33GHz. It also has 12GB of RAM, and contains 2 nVidia Tesla C2070 cards. For OpenMP tests, we use 8 threads.

Table 1: Test Datasets

Name	Dims	Triangles	Tetrahedrons
Spherical Distance A	128^3	149,420	10,241,915
Spherical Distance B	256^3	607,388	82,906,875
Spherical Distance C	512^3	2,451,212	N/A
MRI Head A	128^3	789,440	10,241,915
MRI Head B	256^3	4,947,294	82,906,875
Supernova	432^3	7,036,776	N/A

The supernova dataset shown in Figure 8 is from a supernova simulation made available by John Blondin at the North Carolina State University and Anthony Mezzacappa of Oak Ridge National Laboratory [BD03]. We calculate a per-vertex estimate of mean curvature [KM03] during the marching cubes algorithm and attach this as a vertex attribute through all additional computational passes.

7.1. Marching Cubes

Bell, a primary developer of the Thrust library, presents VERTEX-WELD as an example application of Thrust [Bel10]. Bell suggests a lexicographic sort where first the x, then y, then z coordinates are compared and compacted, which is sensitive to floating-point error because small differences may lead to nonadjacent placement of otherwise identical vertices. VERTEX-WELD merges duplicate vertices, allowing for simple determination of all facets that contain a particular vertex, which allows computation of incidence and adjacency lists. Our approach has many similarities, but VERTEX-WELD was not designed to accommodate topological reconstruction, and because we sort based on input mesh features, our method is robust against floating-point error.

One of the features of our implementation is the ability to merge vertex properties. We report the time of the algorithm without any attribute merge and with the averaging of surface normals. We also provide timings for VERTEX-WELD. Note, however, that the VERTEX-WELD algorithm does not feature the ability to merge vertex attributes, so the timing is only comparable to the non-merging version of our algorithm. These timings are listed in Table 2

Our CUDA implementation achieves better performance than the spatial search in VERTEX-WELD, but the OpenMP implementation does not appear to have the same benefit. Our method allows for attribute merging with a small overhead in both cases, and performance scales approximately with the number of input vertices.

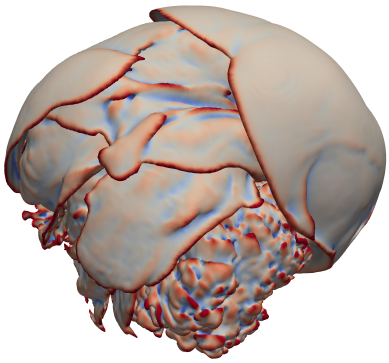


Figure 8: Supernova dataset, displayed with a single subdivision step. Our coarsening technique has also been applied to reduce shading artifacts. Red and blue areas in the image highlight regions of high local mean curvature.

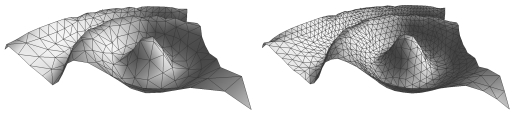


Figure 9: Bezier patch cell subdivision on a rippled surface.

The tetrahedral case achieves approximately twice the performance per vertex of the contour case because of the absence of the normal vector as an attribute. The OpenMP version scales similarly to the CUDA version until system memory is exceeded in the largest tetrahedral dataset, at which point disk thrashing occurs during the sorts.

7.2. Cell Subdivision

We compare our history-enabled cell subdivision against one using VERTEX-WELD in Table 3, with an example shown in Figure 9. Tetrahedral grids performed similarly.

7.3. Face-Centered Tetrahedralization

We compare our history-enabled face-centered tetrahedralization against a similar version that uses VERTEX-WELD. See Figure 1c for a diagram of face-centered tetrahedralization of two voxels. Timings are listed in Table 4.

7.4. Mesh Coarsening

Table 5 shows timings for mesh coarsening during vertex welding. Compare with Table 2 to observe that the coarsening has no significant effect on performance.

Table 2: Time to perform Marching Cubes complete with vertex merging for a manifold surface.

Data	VERTEX WELD (no merge)	KEY REDUCE (no merge)	KEY REDUCE (averaging)
Spherical Distance, Structured CUDA			
128^3	40ms	31ms	42ms
256^3	166ms	141ms	177ms
512^3	847ms	752ms	894ms
Spherical Distance, Unstructured CUDA			
128^3	242ms	226ms	251ms
256^3	1871ms	1791ms	2014ms
MRI Head, Structured CUDA			
128^3	168ms	138ms	181ms
256^3	1092ms	903ms	1193ms
MRI Head, Unstructured CUDA			
128^3	545ms	476ms	583ms
256^3	4193ms	4011ms	4347ms
Supernova, Structured CUDA			
432^3	842ms	723ms	885ms
Spherical Distance, Structured OpenMP			
128^3	184ms	168ms	191ms
256^3	652ms	640ms	721ms
512^3	3266ms	3254ms	3582ms
Spherical Distance, Unstructured OpenMP			
128^3	816ms	771ms	901ms
256^3	9623ms	9159ms	10515ms
MRI Head, Structured OpenMP			
128^3	654ms	634ms	721ms
256^3	4022ms	3090ms	4851ms
MRI Head, Unstructured OpenMP			
128^3	2123ms	2117ms	2514ms
256^3	29425ms	29224ms	31429ms
Supernova, Structured OpenMP			
432^3	3145ms	3108ms	3427ms

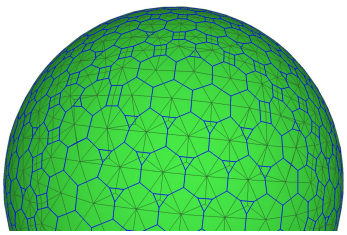


Figure 10: Dual mesh (blue) overlaid as as a skeleton upon a contour mesh (green) on the spherical distance dataset

7.5. Dual Mesh Generation

Figure 10 shows a mesh and its dual overlaid as a skeleton. Table 6 shows the timings for generating dual meshes.

Table 3: Time to perform one level of cell subdivision

Data	VERTEX-WELD	KEY-REDUCE
Spherical Distance CUDA		
128^3	91ms	84ms
256^3	362ms	340ms
512^3	1443ms	1367ms
MRI Head CUDA		
128^3	479ms	409ms
256^3	3768ms	3231ms
Supernova CUDA		
432^3	5798ms	5514ms
Spherical Distance OpenMP		
128^3	360ms	337ms
256^3	1410ms	1304ms
512^3	5566ms	5334ms
MRI Head OpenMP		
128^3	1532ms	1376ms
256^3	8905ms	7939ms
Supernova OpenMP		
432^3	13797ms	13548ms

Table 4: Time to perform face-centered tetrahedralization.

Data	VERTEX-WELD	KEY-REDUCE
CUDA Spherical Distance		
128^3	79ms	75ms
256^3	624ms	595ms
CUDA MRI Head		
128^3	152ms	125ms
256^3	1219ms	997ms
OpenMP Spherical Distance		
128^3	95ms	77ms
256^3	766ms	721ms
OpenMP MRI Head		
128^3	215ms	194ms
256^3	1994ms	1781ms

8. Conclusions and Future Work

We provide an efficient, generalized method for parallel generation of topological connectivity information. We require little to no alteration to the algorithms generating geometry, although we leverage small modifications to allow for knowledge of input topological features for better performance. We demonstrate how to use such modifications to gain performance on structured grids, and to perform a simple mesh coarsening on either structured or unstructured grids. Future work includes determination of bottlenecks for scalability on larger architectures.

9. Acknowledgements

This work was supported in full by the DOE Office of Science, Advanced Scientific Computing Research, award

Table 5: Performance of Marching Cubes with coarsening.

Data	CUDA	OpenMP
Spherical Distance, Structured		
128^3	41ms	199ms
256^3	175ms	725ms
512^3	885ms	3571ms
Spherical Distance, Unstructured		
128^3	245ms	899ms
256^3	1997ms	10529ms
MRI Head, Structured		
128^3	179ms	729ms
256^3	1150ms	4875ms
MRI Head, Unstructured		
128^3	589ms	2519ms
256^3	4332ms	31444ms
Supernova, Structured		
432^3	717ms	2958ms

Table 6: Time to create a dual mesh from a contour surface.

Data	CUDA	OpenMP
Spherical Distance, Structured		
128^3	38ms	100ms
256^3	138ms	465ms
512^3	575ms	1992ms
Spherical Distance, Unstructured		
128^3	52ms	124ms
256^3	198ms	657ms
MRI Head, Structured		
128^3	203ms	662ms
256^3	1348ms	4414ms
MRI Head, Unstructured		
128^3	297ms	904ms
256^3	1321ms	4259ms
Supernova, Structured		
432^3	1653ms	5779ms

number 10-014707 and DE-CS0005334, program manager Lucy Nowell, and DE-FC02-12ER26072, program manager Ceren Susut-Bennett. Part of this work was performed by Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration.

References

[ABM*01] AHRENS J., BRISLAWN K., MARTIN K., GEVECI B., LAW C. C., PAPKA M.: Large-scale data visualization using parallel data streaming. *IEEE Computer Graphics and Applications 21*, 4 (July/August 2001), 34–41. 2

[BD03] BLONDIN J. M. M. A., DEMARINO C.: Stability of Standing Accretion Shocks, with an Eye toward Core-Collapse Supernovae. *The Astrophysical Journal* (2003). 6

[Bel10] BELL N.: High-Productivity CUDA Development with the Thrust Template Library, 2010. 6

[BH] BANK R. E., HOLST M.: A New Paradigm for Parallel Adaptive Meshing Algorithms. *SIAM 2003*. 5

[Ble90] BLELLOCH G.: Prefix sums and their applications. *Synthesis of Parallel Algorithms* (1990). 2

[CMS06] CARR H., MÖLLER T., SNOEYINK J.: Artifacts caused by simplicial subdivision. *TVCG* (2006). 4

[DG08] DEAN J., GHEMAWAT S.: MapReduce: Simplified data processing on large clusters. *ACM Communications* (2008). 2

[DT07] DECORO C., TATARCHUK N.: Real-time mesh simplification using the GPU. In *I3D* (2007). 5

[DZT08] DYKEN C., ZIEGLER G., THEOBALT C.: High-speed Marching Cubes using HistoPyramids. *CGF* (2008). 2

[GF05] GOETZ F. JUNKLEWITZ T.: Real-Time Marching Cubes on the Vertex Shader. *EUROGRAPHICS* (2005). 2

[Hor05] HORN D.: Stream Reduction Operations for GPGPU Applications. In *GPU Gems 2*. 2005. 2

[JC06] JOHANSSON G., CARR H.: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research. *SIGGRAPH* (2006). 2

[KM03] KINDLMANN G. W. R. T. T., MOLLER T.: Curvature-based transfer functions for direct volume rendering: Methods and applications. *Vis* (2003). 6

[KW05] KIPFER P., WESTERMANN R.: GPU construction and transparent rendering of iso-surfaces. In *VMV* (2005). 2

[Lor87] LORENSEN W.: Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH* (1987). 1

[LSA12] LO L.-T., SEWELL C., AHRENS J. P.: Piston: A portable cross-platform framework for data-parallel visualization operators. In *EGPGV* (2012), pp. 11–20. 1

[MAGM11] MORELAND K., AYACHIT U., GEVECI B., MA K.-L.: Dax Toolkit: A proposed framework for data analysis and visualization at Extreme Scale, 2011. 2

[MAPS12] MEREDITH J. S., AHERN S., PUGMIRE D., SISNEROS R.: Eavl: the extreme-scale analysis and visualization library. In *Eurographics Symposium on Parallel Graphics and Visualization* (2012), The Eurographics Association, pp. 21–30. 2

[MW91] MOORE D., WARREN J.: Mesh displacement: An improved contouring for trivariate data. *Computer* (1991). 2

[PCC04] PARK J., CHOI B., CHUNG Y.: Efficient topology construction from triangle soup. In *GMP* (2004). 1, 2

[Pot11] POTTER M.: *Anisotropic mesh coarsening and refinement on GPU architecture*. PhD thesis, Imperial College London, Department of Computing, June 2011. 4

[RKE00] RÖTTGER S., KRAUS M., ERTL T.: Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Visualization* (2000). 2

[SCMO10] STUART J., CHEN C., MA K.-L., OWENS J.: Multi-GPU volume rendering using MapReduce. In *1st International Workshop on MapReduce and its Applications* (June 2010). 2

[SHZO07] SENGUPTA S., HARRIS M., ZHANG Y., OWENS J.: Scan Primitives for GPU Computing. *Computing* (2007). 2

[VBS*11] VO H., BRONSON J., SUMMA B., COMBA J., FREIRE J., HOWE B., PASCUCCI V., SILVA C.: Parallel visualization on large clusters using MapReduce. In *LDAV* (2011). 2

[ZG08] ZHOU K. H. Q. W. R., GUO B.: Real-time KD-tree construction on graphics hardware. *ACM Graphics* (2008). 2