Parallel Topological Connection Reconstruction

Robert Miller, Kenneth Moreland, Kwan-Liu Ma

Abstract— Many algorithms applied to mesh-based data structures are easily parallelizable because operations can often be divided by the constituent elements of the mesh and parallelized in large thread blocks. The problem with this technique is that when these threads generate new topological features, they tend to generate redundant information such as coincident vertices. This "soup" of elements requires significant additional memory to store the redundant information and fails to capture the adjacency of neighboring elements. We present a novel approach for the efficient construction of connected topologies on massively parallel systems, which we demonstrate with GPU and OpenMP implementations. In this approach, we use input topological features as a basis for efficient location of coincident features in the output. We provide examples of the technique for isosurface generation and multiple forms of tetrahedralization.

Index Terms—TODO

1 Introduction

Many existing parallel graphics techniques are designed to generate geometry from some form of input mesh. Such algorithms tend to have the attribute that computation of any given output feature depends solely on a small set of input features. This provides a natural partitioning of the input which can then be used to develop parallel algorithms. As an example, the Marching Cubes [18] algorithm can be parallelized per voxel, so that only the data values at the voxel corners are necessary to generate the geometry within the voxel (if any).

The problem with this naive parallelization is that it generates a 'soup' of output primitives, with no information about connectivity. The aforementioned Marching Cubes algorithm simply produces a set of disconnected triangles. This is not a concern so long as all further computation on the output depends exclusively on the local attributes of individual primitives, such as is the case with rendering. Rendering has often been the ultimate goal for the generated output of parallel graphics algorithms and the size of the output is not a concern, so these techniques have been sufficient. When additional processing of the output is desired that requires information about the connectivity between multiple primitives, the situation becomes more difficult. Consider the calculation of the curvature of the isosurface generated by Marching Cubes. 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. A more complex example would be the case of a visualization pipeline, where different types of topological connectivity information may be necessary depending on the structure of the pipeline.

This is not a new problem, and techniques exist to to determine topological information about a primitive soup [19]. Generally this approach starts by finding and coalescing duplicate vertices, which may require a bounded radius nearest neighbor search to resolve vertices that are identical save for floating point error. Next, the algorithm finds all primitives that share two or more vertices, and links them together as neighbors. Finally, some 'duplicate' vertices may not actually represent desirable connections, such as is the case where two cones meet at their apices, so these vertices may need to be split in a final pass. This process is computationally complex, and is specific to the topological connections between triangles.

We would like to support a technique which applies equally well to other types of topological connections, such as determining the neighboring facets of tetrahedra. To this end, we have explored the advantages of making a small modification to the algorithms that generate the geometry, where information is stored about what feature of the input topology was used to generate each output feature. For instance, Marching Cubes generates all output vertices on edges that exist between the voxel coordinates of the input mesh.

To summarize, this paper is intended to make the following contributions:

- Provide a generalized technique for generation of topological connectivity information on highly parallel systems.
- 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 features of input topology to enhance the performance of the topology resolution technique.
- Demonstrate the effectiveness of our technique on the output of the Marching Cubes algorithm and a tetrahedralization algorithm.
- Show how knowledge of the output topology of an algorithm can be used to perform simple mesh coarsening in the same pass as the topology resolution.

We developed our technique for topology resolution so that it could be implemented into the upcoming DAX framework, which will provide data analysis capabilities similar to those in VTK, but targeted toward massively parallel architectures, beginning with the GPU.

2 Previous Work

We use Marching Cubes as an example of parallelized geometry generation [18]. There are myriad existing isosurfacing implementations on the GPU, dating from Rottger's implementation [21]. Dyken provides a fairly detailed overview of the advancements in GPU implementations of Marching Cubes and Marching Tetrahedra [11] .

The basic topology resolution technique described in the introduction section is introduced by Park et al. [19]. Additionally, Kipfer mentions implementing a topology construction technique along with their GPU Marching Cubes algorithm [16]. Kipfer's technique involves the use of the edges that vertices will be placed on as a key for determining which vertices would be duplicated. Kipfer thus avoids the multiple computation of such vertices and avoids the need for coincident point resolution. To have accurate normals at the generated vertices, Kipfer's technique requires precomputation of the gradient

- *Robert Miller is a member of the ViDi group at the University of California, Davis. E-mail: bobmiller@ucdavis.edu*
- *Kenneth Moreland works at Sandia National Laboratories. E-mail: kmorel@sandia.gov*
- *Kwan-Liu Ma leads the ViDi group at the University of California, Davis. E-mail: ma@cs.ucdavis.edu*

Manuscript received 31 March 2011; accepted 1 August 2011; posted online 23 October 2011; mailed on 14 October 2011.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org.

of the volume at the voxel vertices. Any other interpolable attributes to be derived from the volume must also be precomputed at the voxel vertices. Our method for determining dulateplicate vertices extends Kipfer's method in that we may key off of any input feature.

Stream compaction is an important element for efficient geometry generation on the GPU. Horn provides an early method for stream compaction on the GPU [14], which was improved upon by Sengupta [22], who also provides a CUDA implementation. These approaches rely on the data parallelism technique and the application of prefix sums as described by Blelloch [4]. Dyken further optimizes GPU compaction techniques by making use of a data structure called a Histogram Pyramid [11]. 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 could be substituted for the prefix sum technique for increased performance in any case where our method is applied.

We use coincident point resolution (vertex welding) in Marching Cubes with the Thrust C++ template library [3]. Bell, a primary developer of the Thrust library, also presents a vertex welding technique as an example application of Thrust [2]. Bell also notes that GPU sorting techniques can be instrumental in the development of high-performance GPU algorithms. Hoberock uses a lexicographic sort directly on the vertices generated by Marching Cubes, then collapses duplicates to get the final welded surface. Our approach has many similarities, but we differ in the basis for the sort and in the method of collapsing duplicates. We can apply the technique to arbitrary input topologies and also avoid invalid results when floating point errors exist in the vertex coordinates.

Carr et al. [7] describe several ways to subdivide voxel grids into tetrahedral grids. To fairly test performance of our topology generation on unstructured grids, we generate each of these subdivisions and demonstrate that the performance of our approach is similar in all cases.

Observant readers may notice a similarity between our proposed algorithms and those supported by the MapReduce framework [9]. Like MapReduce, our algorithms first map keys to values then collect keys and reduce the values. Although we believe our algorithms could be implemented in a MapReduce framework (an exercise we leave to the reader), it would require additional collection operations to resolve topological connections. Other researchers [24][25] have implemented visualization algorithms directly in MapReduce, but they serve purposes other than those we address.

3 Methods for Merging Algorithm

Without significant modification of the algorithms generating topological features, we present an approach that reconstructs the topological connections between these features. By topological connections, we are referring to the information necessary to determine all topological features that are adjacent to any individual feature of interest. We assume and provide a representation based on point arrays and either implicit or explicit cell connection lists. Generating incidence lists, such as the list of all cells incident to a point, is a trivial extension to our algorithm. Given a point P generated by the Marching Cubes algorithm, the topological connections could be a list of all faces that contain point P as a vertex. Depending on the intended use of the topology, it may be preferable to store the list of points Q where there exists an edge (P,Q). Our approach can be applied to generate any of these forms of connectivity information.

3.1 Goals

In developing our technique for reconstruction of topological connections, we have the following requirements.

- Support at least the following feature types: vertices, edges, faces and facets.
- Require minimal modification to the algorithms used to generate topological features.
- Work as an efficient data-parallel method

3.2 Algorithm

Our technique was developed by generalizing and extending the vertex welding example from the Thrust library [3]. Since many vertices are redundant when all triangles in a mesh are stored discretely, it is often desirable to 'weld' the duplicate vertices to single instances. We base our technique on the Thrust method because it has a similar structure to the approach we take for topology generation, and because it is designed to work with nVidia CUDA platforms as well as platforms supported by OpenMP. The algorithm works as follows:

- Given an array of point coordinates V_1:
  1. Lexicographically sort V_1 so that duplicate vertices are placed together in the array.
  2. Take the first element from each group of duplicates and store it in an output array V_2.
  3. Find the new index in V_2 for each vertex in V_1 and store it in an output array $Map_{1\rightarrow 2}$, which serves as a many-to-one mapping of points in V_1 to V_2.
- Return V_2 and $Map_{1\rightarrow 2}$ as output.

The above algorithm is a good basis for an algorithm to generate topological connectivity because it allows for grouping and processing of duplicate vertices. Since we can store which triangle generated each copy of a duplicate vertex, the above algorithm can be adapted to calculate a neighboring face list for each point. Therefore, using the Thrust algorithm as a basis, we set out to create a technique that could be used for generating topological connectivity information from multiple kinds of disconnected input geometric elements. There are several requirements for this which the Thrust algorithm does not immediately meet.

Unlike the creators of the Thrust algorithm, we are not interested specifically in the elimination of duplicate elements. Instead, we want to form groups of geometric elements and then merge these into an output. These elements may not be simple vertices as in the Thrust example. To use this for the generation of topological connections, we will need to associate attributes with each vertex. To this end, rather than taking as input an array of vertices, we take in an array of geometric elements of some unspecified type. Often, the input will be an array of 'tuples' which may consist of vertices and their associated attributes such as normals or polygon IDs. Hereafter, we will refer to this input array as the 'mergeable' array.

The Thrust algorithm groups bitwise-equal elements together. This would be inappropriate if additional attributes were associated with each sample of a given vertex. For instance, three vertices at the same location that possess different normal vectors would not compare bitwise equal, but we may wish to process them as a group. To resolve this, we take in an array of 'keys' where each key corresponds to an element of the mergeable array. The key for each mergeable element will be compared against the keys for other mergeable elements using a user defined comparison operator *keycomp*. *keycomp* is required to be analagous to the less-than operator, so it takes two keys as input and must return *true* if the first element should be placed earlier in the array than the second element, and $false$ otherwise. If $keycomp(k_1,k_2) = false$ and $keycomp(k_2,k_1) = false$, then the elements are assumed to be part of the same group and will be merged.

The keys can be based off of point coordinates as is done in the Thrust welding example. However, keys based on floating point numbers such as coordinates are inferior because of the large number of bits required to represent them and the high probability of small numerical errors leading to lexicographic differences. Later in this paper we present algorithms with more efficient keys.

Since we no longer require elements of each group to be bitwise equal, we cannot simply use the first element of the group for our output vector. Instead, we will rely on two user-defined operators which together will merge the grouped elements to a single output. The reason for having two operators is that some operations, such as averaging, require two steps; in this case, the first operator would be a sum,

and the second operator would be a divide. In our implementation, we restrict the first operator, hereafter referred to as a merge, to be analagous to a sum: a binary operator that is commutative and associative. We restrict the input and output types of this operator so that it must take in two geometric elements and output a version that merges the elements' properties in some way. Likewise, we restrict the second operator, hereafter referred to as a transform, to be analagous to scalar division: a binary operator taking one geometric element and the number of elements in the group as input. The return type of the transformation operator is unrestricted, and the result of these transformations will be passed as the output of our algorithm.

Finally, for this technique to be useful it must be possible to determine where each element of the mergeable array is mapped to in the output array. Our algorithm will therefore return an array of the new indices for each of the input elements.

With these modifications, we provide a generalized method to group various types of geometric elements and then merge each group into single outputs. A high-level overview of the algorithm follows:

- Given:
  - An array of mergeable elements M
  - An array of keys K of the same size as M
  - A key comparator $keycomp(k_1, k_2) \rightarrow bool$
  - A merging operator $merge(m_1, m_2) \rightarrow m_3$
  - A transformation operator $transform(m_1, GroupSize) \rightarrow Output$

  1. Sort M using the key array K and the key comparator $keycomp$ to divide M into groups.
  2. Determine the new indices for each element and store them in an output array $Map_{1\rightarrow 2}$.
  3. Merge the elements of each group in M into single elements using the *merge* operator and store in an array of mergeables N. During this process, maintain a count of the number of elements merged to form each group, storing these in an array C
  4. Transform each element in N into outputs using the $transform$ operator with the sizes from C and store in an array of outputs O.

- Return the output array O, the new index array $Map_{1\rightarrow 2}$, and the group sizes C as output.

To provide a more concrete example, our templated CUDA implementation of this technique using the Thrust library is available in the supplemental material.

4 Methods for Mesh Generation

In order to test our method for generation of topological connectivities on a range of data types, we made use of several techniques to generate geometry. To test with data calculated along a structured grid, we used the Marching Cubes algorithm to generate contour surfaces from input volume data. Next, to test with data from unstructured grids, we tetrahedralized the volume in each of the manners as presented by Carr, et al. [7]. We generated contour surfaces from these tetrahedralizations to get similar data as from the Marching Cubes results but from an unstructured grid. Additionally, we use the tetrahedralizations directly as the input to test our technique to find the topological connections of a 3D structure.

4.1 Performance Measurement

We measure performance on a test system containing an Intel Core i7 975 Processor, which has a clock speed of 3.33GHz. Our test system has 12GB of RAM, and contains 2 nVidia Tesla C2070 cards. The datasets we use for testing are shown in Table 1. For all OpenMP tests, we set the number of threads to 8.

Name	Dimensions	Tris (MC)
Spherical Distance A	64x64x64	35,996
Spherical Distance B	128x128x128	149,420
Spherical Distance C	256x256x256	607,388
Spherical Distance D	512x512x512	2,451,212
MRI Head A	64x64x64	138,510
MRI Head B	128x128x128	789,440
MRI Head C	256x256x256	4,947,294

Table 1. Test Datasets

The isovalues used for contour generation are consistent for all tests within each dataset, and are chosen to show relevant features of the dataset. We perform Marching Cubes, Marching Tetrahedra and several tetrahedralizations on each dataset to produce input geometry. All reported times in all tables are in milliseconds. The Thrust library does not currently support spanning operations over multiple GPUs. We make use of two GPUs for our measurements by partitioning the input and performing the algorithm separately on each, then merging the results.

4.2 Method of Topological Connectivity Construction

Our technique is an extension of existing vertex welding methods. We will first analyze the original vertex welding approach and the manner in which we extended it for topological connectivity construction. We will then discuss methods of performance improvement of the original vertex welding method that we can achieve due to our extensions. Finally, we will discuss the usage of our extensions for the generation of topological connectivity information for various types of geometric inputs.

4.2.1 Vertex Welding

Because we made several modifications and extensions to the original vertex welding technique, we measure the additional performance cost of our implementation of the technique as compared to the original. To perform a fair comparison, it is necessary to apply our technique to the same task. Vertex welding with our generic group-merging algorithm can be accomplished by specifying the following inputs:

- The array of mergeables M should be an input array of vertices.
- The array of keys K should be the same array as M.
- The key comparitor should compare vertices lexicographically.
- The merge operator should return the first element.
- The reduction operator should simply return the given element.

For the purposes of vertex welding, we ignore the C output of the group merging algorithm, and treat the outputs O and $Map_{1\rightarrow 2}$ as the original algorithm's outputs V_2 and $Map_{1\rightarrow 2}$, respectively.

Although we were able to measure the performance differences using this technique, it provides no benefit over the original implementation. Also, our generated contour surfaces contain per-vertex normals and several scalar attributes, and this approach loses all information for these associated attributes except for the first instance of each vertex. To resolve this, we substitute a merge operator that sums these attributes, and a reduce operator that divides by the group count (and renormalizes the normal vector) so that our output vertices contain an averaged sample of the attributes from each merged instance. We then measure the performance impact from this change.

To improve performance over the initial technique, we make use of our ability to use input keys other than the original vertices. Initially we considered the case of the output of our structured grid approach, Marching Cubes. In this case, we know a good deal about the input topology on which the output is generated that was not previously being used to help generate the topological connections. Specifically, the

output vertices from Marching Cubes are generated only on the edges of input voxels. Each of these edges is shared by at most four voxels (ignoring degenerate vertex intersections), so there can be at most four instances of any one vertex. We assign each unique edge in the voxel grid an implicit integer ID, then store the corresponding edge ID for each vertex generated by Marching Cubes. Other than the additional storage requirement, this modification causes no measurable change in the performance of geometry generation, but significantly improves performance during vertex welding. Because it is known that these keys are identical if and only if the vertices are identical, we hypothesize that by using these integer IDs as the keys in the group-merging algorithm we noticeably improve the performance of the technique due to the simpler comparison operations for the sorts.

We next consider the case of unstructured grid inputs. In this case edges cannot in general be implicitly assigned IDs, so they must be built from vertex IDs, which are implicit. We pair vertex IDs (least to greatest) to form a unique integral ID for each edge of the unstructured grid and store this as an attribute of the output vertex.

Using the integer ID approach described above carries an additional advantage. From our Marching Cubes and Marching Tetrahedra implementations, we find that the Thrust approach often fails to weld meshes as expected. We trace this failure to the fact that some vertices that should be identical actually differ slightly due to floating point error, and thus are not bitwise equal as required by the Thrust approach. Such errors are exceptionally easy to encounter: since floating-point operations are not associative, our errors are generated simply by failing to ensure that we always use the same vertex order when generating interpolants. By assigning integral IDs to the output vertices, our approach is agnostic to such floating-point error and does not require this change to the geometry-generating implementation.

4.2.2 Vertex Welding Results

We measure performance of the Thrust library example for vertex welding on each set of generated geometry. Additionally, we measure our version of vertex welding with and without merging of vertex properties, and with and without integral IDs assigned to each vertex. The difference in recorded times for geometry generation are negligible for both structured and unstructured grids when generating integral IDs for each vertex and face, as opposed to the timings when these IDs are not generated. Because of this, and because geometry generation is independent of our technique, we do not report timings for geometry generation here. We report the CUDA and OpenMP timings for each approach for vertex welding on the largest spherical dataset in Table 2.

Test	CUDA	OpenMP
Thrust vertex welding (no attribute merging)	281	1557
Our vertex welding (no attribute merging)	281	1557
Our vertex welding (with attribute merging)	527	2862
Our vertex welding (integral IDs, no merging)	204	1620
Our vertex welding (integral IDs, merging)	350	2446

Table 2. Performance comparison for different approaches to Vertex Welding

Because the timings are best for merging when using integral edge IDs as keys for vertex welding when merging is necessary, all further timings in this paper use that approach for the vertex welding component. Also note that using integral IDs for the non-merging case yields better performance in CUDA than the thrust approach, due to the less complex comparison operator within the sort. The OpenMP non-merging case does not appear to benefit from the use of integral IDs. Timings for our vertex welding technique for all input datasets are shown in Table 3

4.2.3 Incidence and adjacency list construction

When only considering vertex welding, we make no use of the output C of our algorithm, which represents the sizes of the merged groups.

	Spherical Distance				MRI Head		
Arch.	64^3	128^3	256^3	512^3	64^3	128^3	256^3
CUDA	14	40	139	555	39	188	1177
OpenMP	26	116	512	2446	115	771	5502

Table 3. Timings for vertex welding of contours on structured grids.

This output is more useful when we wish to determine the topological connections between the merged elements. We demonstrate the use of our technique to determine the following kinds of topological connectivity information:

- Lists of adjacent vertices and incident edges, faces and tetrahedra for each input vertex
- Lists of faces incident to each edge or adjacent to each face, even those connected only by a single vertex
- Lists of faces adjacent to each face, sharing exactly one edge
- Lists of tetrahedra adjacent to each tetrahedron, including those connected only by an edge or point
- Lists of tetrahedra adjacent to each tetrahedron, sharing exactly one face

We begin with the generation of incident faces for each input vertex on a generated surface. The procedure is as follows:

- Generate an array F containing the ID of the face represented by each instance of each vertex. For a triangular mesh, such an array would normally have the form $\{0,0,0,1,1,1,2,2,2...\}$, and thus can be generated implicitly. More complex meshes may require the IDs to be generated during geometry generation, which requires extra storage but is computationally trivial.
- Perform Vertex Welding via our previously described method, using M to merge any associated attributes and with K being the integral IDs assigned per-vertex described in 4.2.1. F is not passed into this method.
- Use the output array $Map_{1\to 2}$ from the group merging operation in the previous step as a key to sort both F and $Map_{1\to 2}$ by the elements in $Map_{1\to 2}$.
- Perform an exclusive scan on the output array C from the group merging operation, storing the result in I.
- The output incident face list is represented by our outputs F, I, and C. Elements in F represent faces incident to specific vertices. Elements in I represent the indices in F where face lists for individual vertices begin. The elements in C represent the sizes of the face lists in F pointed to by I.

This process can be trivially extended to tetrahedral inputs by modifying the array F to represent tetrahedra or any other cell type rather than faces. To determine edges surrounding a point, the process could be similarly modified to have each item in F label the represented edge. Note that when surrounding tetrahedra are generated for each vertex, surrounding faces and are also generated as a subset. Similarly, when surrounding faces are generated surrounding edges are generated as a subset, which can be used to quickly determine the nearest neighboring vertices for each input vertices where an edge exists between the two.

Marching Cubes and Marching Tetrahedra both have the property that all generated surface geometry forms a manifold, with the exception of degenerate and boundary cases. Because of this, we can determine more information about the neighboring geometric features. To determine all faces touched by an edge, simply combine the incident

face lists of the two endpoints. If only the two faces separated by the edge are desired, these can be detected because they will occur twice in the list. All other faces will occur only once.

Determination of adjacent face lists for each triangular face works similarly to edges. Combine the incident face lists of all three endpoints. The face that appears three times in the combined list is the current face. Faces that occur twice are separated from this face by an edge, and those that appear only once are separated by a point. Likewise for tetrahedra: The tetrahedron that appears four times in the resultant list is the current tetrahedron, those that appear three times are adjacent to it through a face, those that appear twice are adjacent to it through an edge, and those that appear once are adjacent to it through a point. Similar behavior should occur for other types of polyhedra and polygons.

Although the optimizations used to perform the Vertex Welding change depending on whether the input is a structured or unstructured grid, the output is processed into topological connectivity information in the same manner for either case.

4.2.4 Incidence and adjacency list construction results

The time necessary to generate topological connectivity information for contours of structured and unstructured grids is shown in Table 4.

	Spherical Distance				MRI Head		
Arch.	64^3	128^3	256^3	512^3	64^3	128^3	256^3
CUDA	14	40	139	555	39	188	1177
OpenMP	9	43	186	844	44	283	1814

Table 4. Timings for topological connectivity generation of contours on structured (SG) and unstructured (UG) grids.

The time necessary to generate topological connectivity information for tetrahedral volumes is shown in Table 5:

	Spherical Distance				MRI Head		
Arch.	64^3	128^3	256^3	512^3	64^3	128^3	256^3
CUDA	?	?	?	?	?	?	?
OpenMP	?	?	?	?	?	?	?

Table 5. Timings for topological connectivity generation of tetrahedral grids.

5 Applications

To test that the generated topological connectivity information is useful on parallel systems once generated, we implement several calculations that make use of the generated topological connectivity information. Specifically, we have implemented the following:

- Local averaging of vertex attributes
- Two forms of mesh coarsening, one of which can be implemented as an extension to vertex welding without incurring additional performance cost
- Generation of a dual mesh

The local averaging technique is an example that does not alter the topological connectivity information, and thus does not require an additional application of our technique after its completion. Mesh coarsening is an example that alters the input topology, and thus requires a regeneration of the topological connectivity information after its completion. We also use mesh coarsening to demonstrate that certain types of mesh processing can occur during the vertex welding phase of topological connectivity construction. Generation of a dual mesh is an example that generates a new topology without modifying the original topology.

5.1 Local Averaging

We perform a simple averaging of properties of the nearest neighboring vertices surrounding each processed vertex. This is accomplished by first obtaining the list of indices to neighboring points as described in 4.2.3. Duplicates in this list are removed, then the attributes of each are stored. We perform a segmented scan [5] to sum the properties of the neighboring vertices using a merge operator that is appropriate for the property type, then divide the summed properties by the number of neighbors for the group. Some vertex properties, such as normals, need an alternative operator to replace the division operation. Since we implemented our approach as a C++ template, this operator can be specified as a template parameter.

5.2 Local Averaging Results

Table 6 shows the timings for calculating the local averages of vector properties on the generated surfaces.

		Spherical Distance				MRI Head		
Type	Arch.	64^3	128^3	256^3	512^3	64^3	128^3	256^3
V	CUDA	2	10	85	681	21	169	1365
V	OpenMP	42	180	887	3882	184	1182	7888
S	CUDA	2	10	85	681	21	169	1365
S	OpenMP	42	181	898	3892	191	1194	7882

Table 6. Timings for local averaging of vector (V) and scalar (S) properties on contour surfaces

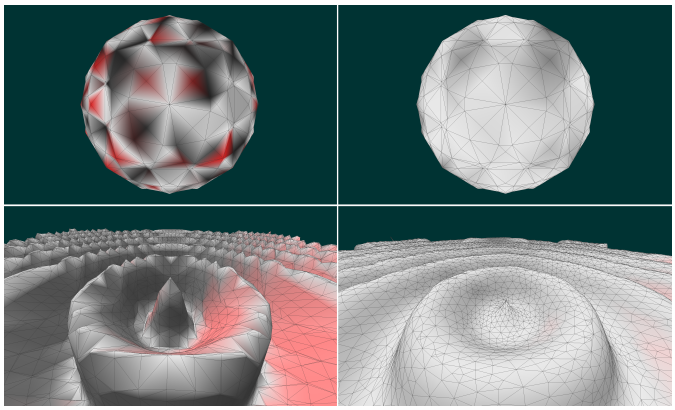

Fig. 1. Top left: Before neighbor scalar averaging function. Top right: After scalar averaging function. Bottom left: Before vertex averaging (simple mesh smoothing). Bottom Right: After vertex averaging.

Figure 1 shows a before and after comparison of averaging used for scalar averaging and vertex averaging.

5.3 Mesh coarsening

Many algorithms that make use of an input topology must modify that topology to produce their output. To demonstrate how such an algorithm can be constructed with our technique, we implement a form of simple mesh coarsening. Traditional implementations of mesh coarsening operate without regard to the mesh's provenance by either iteratively collapsing features [20] or collectively clustering vertices [10]. However, by combining edge collapsing with vertex welding we can provide a first-level coarsening that is fast, has low error, provides well-formed cells, and adapts to the structure of the original mesh.

Rather than computing the topological connectivity of the original generated data and then coarsening it thereafter, we instead alter the input key array to our Vertex Welding technique to perform a type of coarsening in the process of topological connectivity generation. Furthermore, this technique is intended to be more practically useful than the first implementation in that it would improve mesh quality.

Specifically, it is our goal to eliminate poor quality triangles in the mesh while performing minimal modifications to the others. We define triangle quality as follows:

$$Q = \frac{4a\sqrt{3}}{h_1^2 + h_2^2 + h_3^2} \quad (1)$$

This measure of triangle quality considers only aspect ratio, and is such that triangles where $Q > 0.6$ are considered to be of acceptable quality, and $Q = 1$ when the triangle is equilateral [1]. This is equivalent to the pdetriq function in MATLAB.

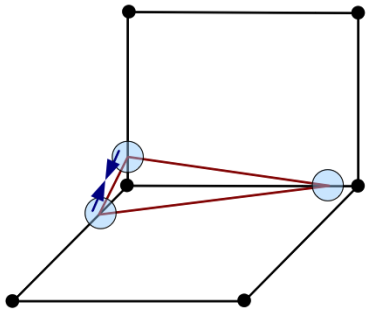

Fig. 2. When Marching Cubes and Marching Tetrahedra generate contour surfaces, the produced vertices can only occur on voxel edges (black). When the output vertices (blue) fall close to the voxel vertices (black), small or skinny triangles are produced (red). We eliminate these by merging all vertices that fall close to a voxel vertex into a single output. This collapses the skinny triangles into degenerate triangles, which are then removed.

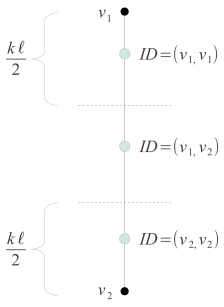

Fig. 3. We use a control parameter k to determine the size of the regions along edges of length ℓ. Our coarsening technique can eliminate these during vertex welding without additional performance cost.

The basic idea for our coarsening is described in Figure 2. Marching Cubes generates small or skinny triangles when two adjacent output vertices are generated near the same input mesh vertex. To eliminate this possibility, we consider all vertices close to any particular endpoint as identical so that during Vertex Welding, these skinny triangles become degenerate and can be removed. Essentially we divide the edges on which geometry is generated into three regions. We reserve a parameter k to determine the size of these regions as a fraction of the edge length ℓ. For unstructured grids, we modify the ID assignment we used previously in vertex welding. As before, we initially assign each vertex in the input topology an ID. During geometry generation, we pair these vertex IDs to form edge IDs, which are attached as attributes to the output vertex. However, when a vertex falls within $\frac{k\ell}{2}$ of an endpoint of this edge, the ID of that endpoint is repeated as shown in Figure 3.

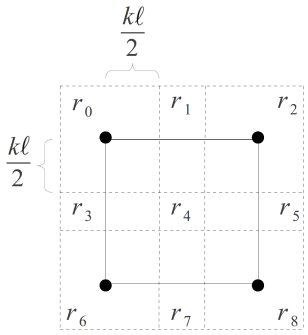

Fig. 4. 2D representation of implicit spatial subdivision into regions for the assignment of IDs to vertices generated on structured grids. Each region surrounding a voxel edge is shared by between 4 and 8 voxels. Because these regions can be determined directly from the output vertex, they need not be stored. As such, no modification to the geometry generating function is required. Furthermore, determination of the integral region ID allows for faster sorting, as no lexicographic comparison of the vertices is required.

As with the original ID assignment for structured grids, we can use our knowledge of the input topology to assign the IDs implicitly so that there is no performance or storage cost. This is accomplished by subdividing space as shown in Figure 4 and assigning an integral ID to each cell. All vertices within these cells will be considered identical and merged during Vertex Welding. The value of k is restricted such that $0 <= k <= 1$, and k specifies the width of the cells that will group vertices. By placing the center of these cells over the intersections of the voxel grid, the 'skinny' triangles generated by Marching Cubes can be reduced or eliminated. In the structured grid case, when $k = 1$ we can guarantee a minimum possible triangle quality. For cubic voxel grids, this minimum is 0.445. We cannot make such a guarantee about the output of the coarsening in the unstructured grid case because the triangle quality depends on the aspect ratios of the input cells. When $k = 0$ this technique is equivalent to the previous form of vertex welding.

After the coarsening has been applied, we expect many degenerate triangles in the output. These triangles may be quickly removed in parallel by removing all instances of faces containing two or more identical indices. This invalidates the output array of group sizes C from the Vertex Welding array, but a corrected group sizes C can be generated by sorting the output array $Map_{1\to 2}$ and performing a segmented scan to determine the number of instances of each index.

A larger problem with the coarsening technique is that in some cases, such as the case shown in Figure 5, non-manifold surface elements may be generated. If such surface elements are undesirable, they may be eliminated by calculating the maximum edge curvature of each triangle, and removing all triangles containing an edge curvature of 180°.

5.4 Mesh Coarsening Results

Table 7 shows the timings for coarsening the mesh as part of the vertex welding algorithm. Compare with Table 3 to observe that simultaneously performing the coarsening operation does not significantly affect the timing of the topology generation.

	Spherical Distance				MRI Head		
Arch.	64^3	128^3	256^3	512^3	64^3	128^3	256^3
CUDA	16	38	137	557	37	190	1174
OpenMP	26	135	524	2439	109	753	5332

Table 7. Timings for mesh coarsening during vertex welding

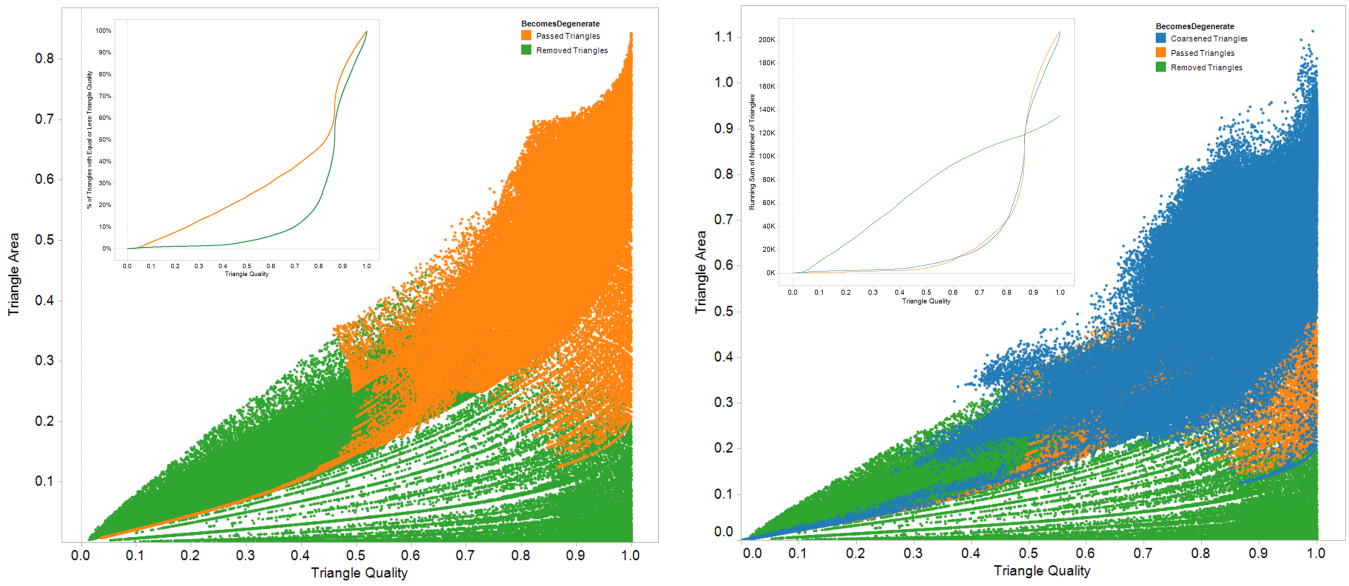

Fig. 7. Top left: Orange represents the triangles that the coarsening chose to keep. Green represents the triangles that the coarsening removed. Top right: Green and Orange are as in the top left. Blue represents the quality of the triangles that the orange triangles were modified into. Top left inset: distribution of triangle quality as compared between the original triangles, and the coarsened triangles. Top right inset: Distribution of triangle quality for coarsened, passed and removed triangles.

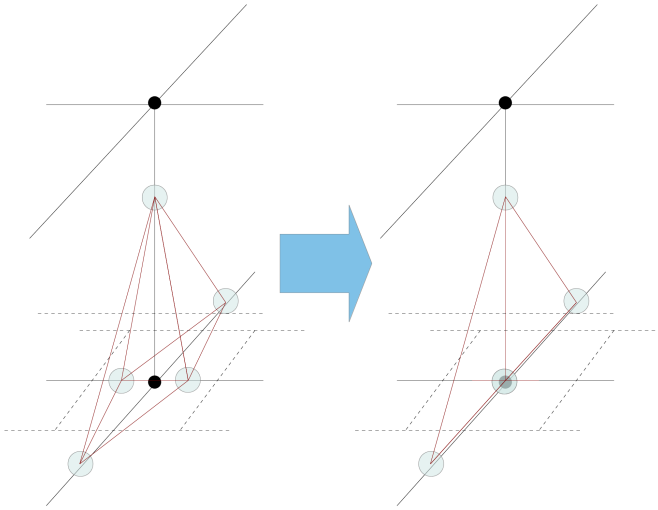

Fig. 5. A problem with our coarsening method is that in some cases nonmanifold faces are produced. In the example above, two vertices from different faces are merged and a 3D volume is collapsed into a single plane. To recreate a manifold surface, we detect and remove these cases in a separate step.

Figure 6 shows a before and after comparison of our coarsening technique with $k = 1$. Figure 7 shows the change in triangle quality before and after coarsening.

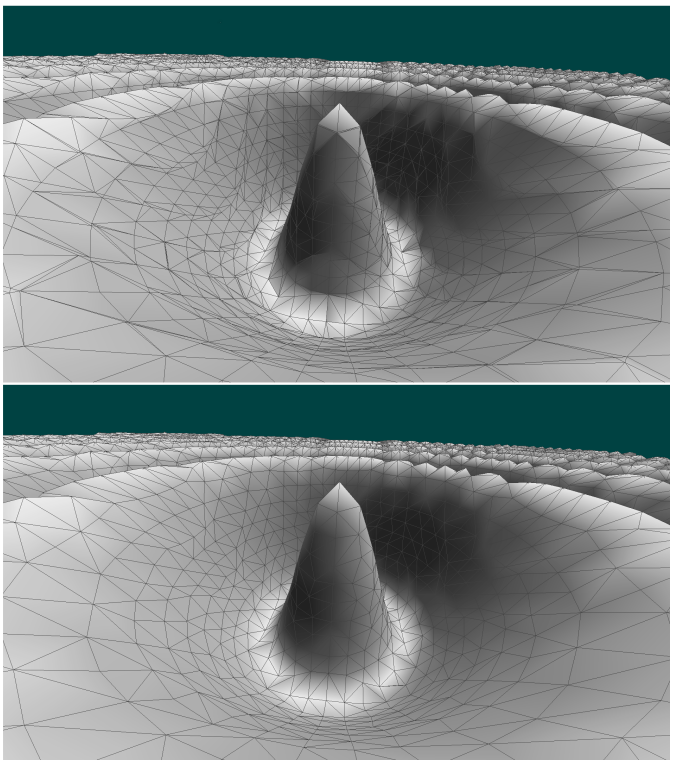

Fig. 6. Top: Before coarsening. Bottom: After coarsening.

5.4.1 Dual Mesh Generation

To generate the dual of our triangular input meshes, we perform the following procedure: For each face, average the vertices to get the centroid, and store it in a new array at the same ID as the original face. Now the surrounding face lists for each of the vertices in the original mesh form a representation of each face of the dual mesh.

5.4.2 Dual Mesh Generation Results

Table 8 shows the timings for generating dual meshes.

TODO: There's a bug in my dual mesh generation code that is giving me trouble. I need to fix it so I can make the figure and update the timings. !!!!Figure!!!! shows a mesh and its dual overlaid as a skeleton.

Architecture	V1	V2	V3	V4	V5	V6	V7
CUDA	18	3	< 1	< 1	39	5	< 1
OpenMP	?	?	?	?	?	?	?

Table 8. Timings for dual mesh generation

6 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 small modifications to allow for knowledge of input topological features can be leveraged 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. We demonstrate that our algorithm is effective on structured grids and on several different tetrahedral grids.

Future work includes measuring how this technique scales up approaching petascale and exascale and determining what bottlenecks, if any, need to be addressed for this transition.

Acknowledgments

This work was supported in full by the DOE Office of Science, Advanced Scientific Computing Research, under award number 10-014707, program manager Lucy Nowell.

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

[1] R. E. Bank and M. Holst. A New Paradigm for Parallel Adaptive Meshing Algorithms. *SIAM Review*, 45(2):291, 2003.

[2] N. Bell. High-Productivity CUDA Development with the Thrust Template Library, 2010.

[3] N. Bell and J. Hoberock. Thrust: A Productivity-Oriented Library for CUDA. *thrust.googlecode.com*, pages 359–373, 2012.

[4] G. Blelloch. Prefix sums and their applications. *Synthesis of Parallel Algorithms*, pages 35–60, 1990.

[5] G. E. Blelloch. Vector Models for Data-Parallel Computing. *Computing Control Engineering Journal*, 2(5):238, 1991.

[6] T. Boubekeur and C. Schlick. Generic mesh refinement on GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on graphics hardware*, number July, pages 99–104. ACM, 2005.

[7] H. Carr, T. Möller, and J. Snoeyink. Artifacts caused by simplicial subdivision. *IEEE transactions on visualization and computer graphics*, 12(2):231–42, 2006.

[8] H. P. Computing. Scientific Visualization with VTK, The Visualization Toolkit. *Fluid Dynamics*, 12(1):1–9, 1992.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[10] C. DeCoro and N. Tatarchuk. Real-time mesh simplification using the GPU. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games (I3D '07)*, pages 161–166, 2007. DOI 10.1145/1230100.1230128.

[11] C. Dyken, G. Ziegler, C. Theobalt, and H.-P. Seidel. High-speed Marching Cubes using HistoPyramids. *Computer Graphics Forum*, 27(8):2028–2039, 2008.

[12] M. Garland, A. Willmott, and P. S. Heckbert. Hierarchical face clustering on polygonal surfaces. *Proceedings of the 2001 symposium on Interactive 3D graphics SI3D 01*, pages 49–58, 2001.

[13] H. Hoppe. Progressive meshes. *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques SIGGRAPH 96*, 30(3):99–108, 1996.

[14] D. Horn. Stream Reduction Operations for GPGPU Applications. In M. Pharr, editor, *GPU Gems 2*, volume 2, chapter 36, pages 573–589. Addison Wesley, 2005.

[15] H. Hsieh and W. Tai. A simple GPU-based approach for 3D Voronoi diagram construction and visualization. *Simulation Modelling Practice and Theory*, 13(8):681–692, Nov. 2005.

[16] P. Kipfer and R. Westermann. GPU construction and transparent rendering of iso-surfaces. In *Proceedings Vision, Modeling and Visualization*, volume 5, 2005.

[17] P. Lindstrom. Out-of-core simplification of large polygonal models. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 259–262. ACM Press/Addison-Wesley Publishing Co., 2000.

[18] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer*, 21(4):163–169, 1987.

[19] J. Park, B. Choi, and Y. Chung. Efficient topology construction from triangle soup. In *Geometric Modeling and Processing, Proceedings*, volume 1, pages 359–364. IEEE, 2004.

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

[21] S. Röttger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings of the conference on Visualization'00*, pages 109–116. IEEE Computer Society Press, 2000.

[22] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU Computing. *Computing*, 21(3):106, 2007.

[23] E. Sifakis. Arbitrary Cutting of Deformable Tetrahedralized Objects. *Computing*, 1, 2007.

[24] J. A. Stuart, C.-K. Chen, K.-L. Ma, and J. D. Owens. Multi-GPU volume rendering using MapReduce. In *1st International Workshop on MapReduce and its Applications*, June 2010.

[25] H. T. Vo, J. Bronson, B. Summa, J. L. Comba, J. Freire, B. Howe, V. Pascucci, and C. T. Silva. Parallel visualization on large clusters using MapReduce. In *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization*, pages 81–88, October 2011. DOI 10.1109/LDAV.2011.6092321.