LA-UR-15-24759

*Approved for public release;
distribution is unlimited.*

Title: Hybrid Data-Parallel Contour Tree Computation

Author(s): Carr, Hamish
Sewell, Christopher Meyer
Lo, Li-Ta
Ahrens, James Paul

Intended for: Los Alamos Technical Report

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the Los Alamos National Security, LLC for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396. By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

Form 836 (7/06)

Hybrid Data-Parallel Contour Tree Computation

Category: Research

Abstract

As data sets increase in size beyond the petabyte, it is increasingly important to have automated methods for data analysis and visualization. While topological analysis tools such as the contour tree and Morse-Smale complex are now well established, there is still a shortage of efficient parallel algorithms for their computation, in particular for massively data-parallel computation on a SIMD model. We report the first data-parallel algorithm for computing the fully augmented contour tree, using a quantized computation model. We then extend this to provide a hybrid data-parallel / distributed algorithm allowing scaling beyond a single GPU or CPU, and provide results for its computation. Our implementation uses the portable data-parallel primitives provided by Nvidia's Thrust library, allowing us to compile our same code for both GPUs and multi-core CPUs.

Index Terms: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations I.6.6 [Simulation and Modeling]: Simulation Output Analysis;

1 Introduction

Modern computational science and engineering depends heavily on ever-larger simulations of physical phenomena. Accommodating the computational demands of these simulations is a major driver for hardware advances, and has led to clusters with petaflops of performance over hundreds of thousands of cores, with petabytes of data storage. For recent hardware, the I/O cost of data storage and movement dominates, and emphasis is increasingly placed on *in situ* analysis and visualization of the data. Moreover, with clusters built around Nvidia's Tesla cards and Intel's Xeon Phi boards, we are seeing a return of SIMD (Single Instruction, Multiple Data) computational models for shared-memory architectures.

In situ analysis and visualization in turn requires more sophisticated analytic tools, as does the recognition that one component of the pipeline remains unchanged: the human perceptual system. This has stimulated research into areas such as computational topology, which constructs models of the mathematical structure of the data for the purposes of analysis and visualization.

One of the principal mathematical tools is the *contour tree* or *Reeb graph*, which summarizes the development of contours in the data set as the isovalue varies. Since contours are a key element of most visualizations, the contour tree and the related *merge tree* are of prime interest in automated analysis of massive data sets.

The value of these computations has been limited by the algorithms available. While there is a well-established algorithm [5] for computing merge trees and contour trees, the picture is patchier for distributed and data-parallel algorithms. In particular, no pure data-parallel algorithm has been described so far for contour tree computation, and the principal result in this paper is to do so for the first time. However, pure data-parallelism is supplemented in practice by hybrid data-parallelism, where individual nodes in a cluster are data-parallel, but the overall computation is distributed between the nodes. We therefore also describe an extension of the data-parallel algorithm to a hybrid data-parallel algorithm.

We therefore begin by describing the relevant background both in data-parallel computation and computational topology in Section 2, before introducing a data-parallel algorithm for contour tree computation in Section 3 and a hybrid distributed algorithm in Section 4. We present some results on the scaling and performance of these algorithms in Section 5, ending by drawing some conclusions in Section 6.

2 Background

Since the goal of this work is to use data-parallel computation to construct an algorithm for contour tree computation, we therefore divide the relevant prior work between data-parallel computation on one hand and contour tree computation on the other. This divide is not strict, since some work has been published on distributed and parallel contour tree computation, but is a convenient division for the sake of clarity.

2.1 Data-Parallel Computation

One effective method for taking advantage of the shared-memory parallelism available on accelerators such as GPUs and multi-core CPUs is to use data-parallelism. Guy Blelloch [3] defined a scan vector model, and demonstrated that a wide variety of algorithms in computational geometry, graph theory, and numerical computation could be implemented using a small set of "primitives". These primitive operators, such as transform, reduce, and scan, can each be implemented in a constant or logarithmic number of parallel steps. Nvidia's open-source Thrust library provides an STL-like interface for such primitive operators, with backends for CUDA, OpenMP, Intel TBB, and serial STL. An algorithm written using this model can utilize this abstraction to run portably across all supported multi-core and many-core backends, with the architecture-specific optimizations isolated to the implementations of the data-parallel primitives in the backends.

We have utilized Thrust in the PISTON and VTK-m projects, implementing algorithms such as isosurfaces, cut surfaces, thresholds, KD-trees, and halo finders [11, 17, 8]. Our halo finding algorithm makes use of a data-parallel union-find algorithm, which most contour tree algorithms depend on. Our Thrust implementation of this algorithm, based on the parallel sparse connected components algorithm presented in [10], is described in detail in [16]. The basic strategy is to create a pseudoforest defined by a function D which maps each vertex to its parent. Initially, each vertex is its own parent. We then iteratively attempt to graft trees onto smaller vertices of other trees, and then perform one level of pointer jumping on each vertex. Once all vertices are in rooted stars (i.e., trees with depth one or less), the algorithm terminates, with D now defining a pseudoforest in which each connected component corresponds to an independent tree. Assuming all edges or vertices can be processed in parallel, each iteration takes constant time.

2.2 Contour Tree Computation

Given a function of the form $f : \mathbb{R}^d \to \mathbb{R}$, a level set is defined as an inverse image $f^{-1}(h)$ for an isovalue h, and a contour is a single connected component of a level set. The Reeb graph can then be defined to be the result of contracting each contour to a single point [15], and is well defined for Euclidean spaces or for general manifolds. For simple domains, the graph is guaranteed to be a tree, and is called the contour tree.

For data analysis, we normally assume that the domain is a mesh - i.e. a tessellated subvolume of $\mathbb{R}^d$, such as is used for numerical simulation. For simplicial meshes in particular, all critical points of the function are guaranteed to be at vertices of the mesh [2], massively simplifying topological computations.

For simplicial meshes over simple domains, the standard algorithm [5] for computing contour trees performs a sorted sweep through the data, incrementally adding all vertices to a union-find data structure [18]. As components are created or merged in the union-find, critical points are identified, and a partial contour tree is created, called a merge tree. After performing both ascending and descending sweeps through the data, the two resultant merge trees are combined to produce the contour tree.

While this algorithm is simple and efficient, it is based on a metaphor of a sweep through the contours which is inherently sequential, and this has hindered development of parallel algorithms. Pascucci & Cole-McLaughlin [14] described a distributed computation in which the data is divided into spatial blocks. The contour tree was computed separately for each block, then a fan-in process combined the contour trees of individual blocks until a single master node computed the entire contour tree.

In practice, contour trees have a significant memory footprint, and, for noisy or complex data set, their size is nearly linear in input size, which forces the contour tree for the entire data set to reside on the master node, defeating one of the purposes of parallelisation: distribution of cost both in computation and in storage.

More recently, Morozov & Weber [12] have proposed a method for distributing a merge tree computation by observing that each vertex in the mesh belongs to a unique component based at a single root maximum, and to a corresponding component at a minimum. Thus, by storing the location of each vertex in a merge tree, the merge tree is held implicitly, distributed across the nodes of the computation. They then generalized this further [13] and stored unique maximal and minimal roots for each vertex. Since this combination is unique for each edge of the contour tree, this implicitly stores the contour tree across the nodes of the computation. These algorithms, however, exploit distributed computing but not data-parallelism, and do not extract arcs and nodes of the tree explicitly.

Notably, one of the advantages of this work is that instead of relying on transferring all of the topology computed per block during the fan-in, it only needs to transfer information relating to boundaries between blocks - i.e. its communication cost can be bounded by $O(n^{2/3})$ for a data set of size n.

Related to this, Widanagamaachchi et al [19] described a data-parallel model for computing the merge tree, breaking the computation into a finite number of fan-in stages. This in effect quantized the merge tree, an effect that was acceptable for the task in hand.

In addition to work on contour tree computation, some of the work on Reeb graph and higher-dimensional topological computation is also relevant. In particular, Hilaga et al [9] quantized the range of the function, explicitly dividing an input mesh into slabs - i.e. the inverse image of intervals rather than of single isovalues. They then identified the neighborhood relationships between these slabs to approximate the Reeb graph of a 2-manifold. Since these slabs are known in $3D$ as interval volumes [7], we will use *interval regions* to describe them in any dimension, and *interval contours* to describe their connected components. More recently, Carr & Duke [4] generalized this with the Joint Contour Net, which approximates the Reeb space [6] for higher dimensional cases by quantizing all variables in the range.

3 Pure Data-Parallel Algorithm

We can see from the foregoing discussion that, while a data-parallel contour tree algorithm has not previously been described, many of the pieces required for such an algorithm are now in place, such as hybrid distributed/data-parallel structures and a reliable union-find algorithm. The principal missing element, however, is a replacement for the queue-based combination of merge trees on which the standard algorithm [5] relies. Since this stage is essentially sequential, replacing it is the principal concern in constructing a data-parallel contour tree algorithm.

Rather than assume that a good data-parallel algorithm is necessarily based on the corresponding serial algorithm, however, we shall instead ask where the inherent parallelism in the problem is. Clearly, this is not in the idea of an incremental sweep through the data, or in a serialized queue for combining two merge trees.

Ideally, we would apply the mathematical definition of contracting contours to single points [15]. Suppose that we have computed a single level set with multiple contours for a given isovalue h in a triangulation in $2D$. To contract these contours, we note that each is made up of a finite number of linear segments which we can represent as nodes in a graph. Moreover, since we are guaranteed a continuous sequence of fragments, we can represent the connectivity between them as edges in the same graph. This transforms the question of contour contraction to a simple application of the union-find algorithm [18], which now exists in data-parallel form.

In practice, however, we cannot perform this computation for every possible contour, as this would require not just infinite parallelism, but uncountably infinite parallelism, which even Blelloch's scan-vector model cannot accommodate. Instead, we observe that, as with algorithms for quantized Reeb graphs [9] and Joint Contour Nets [4], quantizing the range into intervals allows us to approximate the result with any degree of fidelity desired, while keeping the computation bounded in practice.

Moreover, we note that this connectivity computation is independent for any two contours or for the interval contours of any two interval regions. We can therefore compute not just one set of interval contours in parallel, but of all interval contours simultaneously.

This, however, merely performs the contraction, leaving a set of points representing individual interval contours, without representing the vertical adjacencies between them. However, we note that these adjacencies are the summation of local adjacencies between individual fragments. Thus, if we take the set of all local adjacencies between fragments and convert them to edges between their corresponding union-find components, we can then suppress duplicate edges to extract the contour tree desired.

We illustrate this process in Figure 1, showing detailed pseudo-parallel code in Algorithm 1. In this algorithm, each significant step is shown as a for loop with the understanding that these represent data-parallel transformations of the input.

In Stage I (lines 1-4), we create fragments of each edge, divided at integer multiples of a basic slab quantization parameter q. Here, the name fragment is deliberately chosen to evoke rasterization, as we are in fact performing 1-D rasterization of the intervals spanned by each edge. These fragments are shown in darker colors in Figure 1 along the boundaries of the triangles. These fragments will form the vertices for our union-find computation.

For Stage II (lines 5-15), any fragment at a slab value i on one edge of a triangle must connect to all fragments at the same slab value i on the other edges of the triangle. We can represent this by pairing the longest (by value) edge with both other edges.

In Figure 1, we pair edge $e1$ with edges $e0$ and $e5$. Since $e0, e5$ are the shorter edges, we will end up with as many pairs as both of them together have fragments - i.e. 2 and 1 in this instance. We end up with a list of paired fragments in a horizontal array H representing all horizontal connections between edge fragments.

Again considering the triangle with edges $e0, e1, e5$, these pairs will be: $(f0, f2), (f1, f3), (f11, f3)$, representing the connectivity across the middle of the cell by the contour interval. These will form the edges for our union-find computation.

Since we have now defined both vertices and edges of a graph, Stage III (lines 16-21) performs the Union-Find reduction to compute connected components, and each component in the Union-Find array UF represents one interval contour - i.e. one colored band in Figure 1, with one of the fragments being used as the union-find representative, as shown in red in the figure.

Stage IV (lines 23-28) copies the representatives to a new array

Sample computation for non-trivial data

This is a base-level sub-block, so we generate edges directly

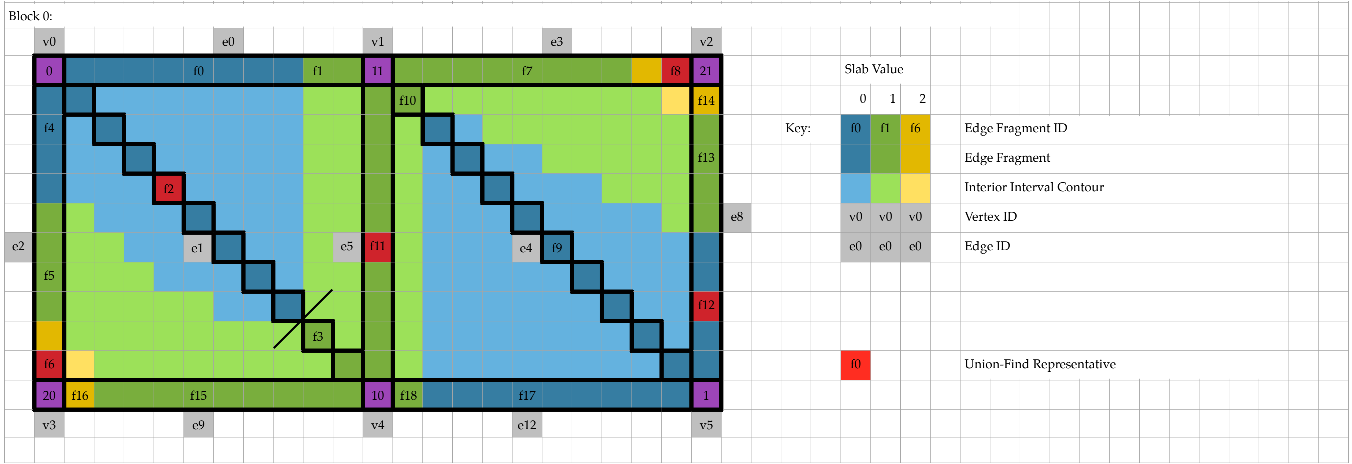

Figure 1: Pure Data-Parallel Example. In this figure, the interval contours and edge fragments are color-coded by slab index (i.e. by quantization interval). In the first stage of the algorithm, one vertex for each colored block is chosen as a union-find representative (in red). In the second stage, vertical edges are computed. See text for details of worked computation.

and suppresses duplicates to get a single unique node for each entire interval contour - i.e. representing nodes in the contour tree.

Finally, since all fragments are indexed, with each edge in ascending order, any pair f_i, f_{i+1} of fragments are adjacent vertically iff they belong to the same edge. Stage V (lines 29-36) therefore takes all such pairs and finds their union-find representatives to describe an arc in the contour tree, with lines 35-36 suppressing duplicates to get unique representation.

To illustrate this, we show each stage in a data-parallel form in Figure 2, using the data in Figure 1 as our running example.

In Stage I, we compute the fragments along each edge by finding the minimum and maximum vertex values, then dividing the minimum value by the quantization q to obtain the *offset* - i.e. the slab index of the lowest fragment. This allows us to find fragments on the edge efficiently later on. We then perform modular arithmetic to determine how many fragments per edge in nEdgeFrags, and use a prefix-sum to find the base index for fragments on each edge.

Note that our edge IDs are not contiguous - this is because we will later need an easy way of doing reverse lookup from the edge ID, and this is trivial if we have a systematic numbering system. Here, edge $0 mod 3$ is always horizontal, edge $1 mod 3$ is diagonal, and edge $2 mod 3$ is vertical.

In Stage II, we again compute the number of pairs first, then generate them for use as arcs in the Union-Find computation. For example, the first pair $(0,2)$ indicates that fragments 0 and 2 are connected in the interval contour, and so on. We have omitted the detailed calculations for clarity, and shown only the result.

In Stage III, we invoke the Union-Find computation using the fragments from Stage I as nodes and the pairs from Stage II as edges. We will see in the discussion of the hybrid algorithm that which fragment we choose as Union-Find Representative is significant, but in the pure data-parallel algorithm we can choose any fragment, and we have therefore chosen arbitrary fragments. At the end of this stage, we can see that there is exactly one unique UF representative for each interval contour in Figure 1.

Finally, in Stage IV, we use the Union-Find representatives again to compute which interval contours are connected to each other by finding vertical pairs along edges, converting these to Union-Find representatives, sorting and suppressing duplicates to get the list of arcs between nodes in the contour tree.

At the bottom of the figure, we build the contour tree from this set of nodes and arcs to confirm that it is indeed the correct quantized contour tree for the input data.

Algorithm 1 Pure Data Parallel Algorithm. All for statements are executed data-parallel.

Require: Triangulation T, vertex values, slab quantization q

1: **for all** Edges e in I **do**
2: Divide e at isovalues nq for integer n
3: Store fragments in fragment array F
4: **end for**
5: **for all** Triangles t in I **do**
6: Find the longest edge e_1 in t by value
7: Pair the longest edge with both other edges e_2, e_3
8: Store pairs $(e_1, e_2), (e_1, e_3)$ in pair array P
9: **end for**
10: **for all** Pairs $p = (e_1, e_2)$ in P **do**
11: **for all** Integer n such that nq is a value on e_2 **do**
12: Divide (e_1, e_2) at nq
13: **end for**
14: Store all fragment pairs (f_1, f_2) in horizontal array H
15: **end for**
16: **for all** Fragments f in F **do**
17: Initialise Union-Find array $UF(f) = f$
18: **end for**
19: **for all** Pairs $h = (f_1, f_2)$ in H **do**
20: Add edge f_1, f_2 to Union-Find array UF
21: **end for**
22: Perform data-parallel Union-Find on UF
23: **for all** Fragments f in UF **do**
24: Find UF representative $u = UF(f)$ 1
25: Store u in node array N
26: **end for**
27: Parallel Sort N
28: Remove duplicates in N
29: **for all** Fragments f in F **do**
30: **if** Fragment f on same edge e as next fragment g **then**
31: Find UF representatives $u = UF(f), g = UF(g)$
32: Store arc $a = (u, v)$ in arc array A
33: **end if**
34: **end for**
35: Parallel Sort A lexicographically on (u, v)
36: Remove duplicates in A
37: N, A now contain nodes & arcs of contour tree

Stage Ia: Fragment Counting

edgeID	0	1	2	3	4	5	6	7	8	9	10	11	12
offset	0	0	0	1	0	1	.	.	0	1	.	.	0
nEdgeFrags	2	2	3	2	2	1	.	.	3	2	.	.	2
bEdgeFrags	0	2	4	7	9	11	12	12	12	15	17	17	17

Stage Ib: Fragment Generation

fragID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
fragEdge	0	0	1	1	2	2	2	3	3	4	4	5	8	8	8	9	9	12	12
fragSlab	0	1	0	1	0	1	2	1	2	0	1	1	0	1	2	1	2	0	1

Stage IIa: Edge Pair Counting

pairID	0	1	2	3	4	5	6	7
pairEdgeLong	0	0	2	2	8	8	4	4
pairEdgeShort	1	5	1	9	4	3	12	5
nPairFrags	2	1	2	2	2	2	2	1
bPairFrags	0	2	3	5	7	9	11	13

Stage IIb: Fragment Pair (Horizontal Edge) Generation

horizID	0	1	2	3	4	5	6	7	8	9	10	11	12	13
horizFrom	0	1	1	4	5	5	6	12	13	13	14	9	10	10
horizTo	2	3	11	2	3	15	16	9	10	7	8	17	18	11

Stage III: Union-Find Computation

fragID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
UF (initial)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
UF (final)	2	11	2	11	2	11	6	11	8	12	11	11	12	11	8	11	6	12	11
UF (sorted)	2	2	2	6	6	8	8	11	11	11	11	11	11	11	11	11	12	12	12
UF (unique)	2	6	8	11	12														

Stage IV: Arc Computation

vertID	0	1	2	3	4	5	6	7	8	9
loFragment	0	2	4	5	7	9	12	13	15	17
hiFragment	1	3	5	6	8	10	13	14	16	18
loUFRep	2	2	2	11	11	12	12	11	11	12
hiUFRep	11	11	11	6	8	11	11	8	16	11
loUF (sorted)	2	2	2	11	11	11	11	12	12	12
hiUF (sorted)	11	11	11	6	6	8	8	11	11	11
loUF (unique)	2	11	11	12						
hiUF (unique)	11	6	8	11						

Computed Contour Tree:

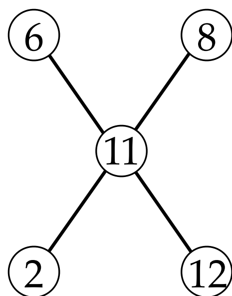

Figure 2: Worked Example of Data-Parallel Contour Tree Computation. See text for discussion

3.1 Algorithmic Complexity

As always with algorithmic development, it is necessary to analyze the performance of the new algorithm. For data-parallel algorithms, this is measured by considering the number of data parallel steps required under infinite parallelism [3]. In general, if an operation can be performed without an expansion or reduction, it takes $O(1)$ steps, but if expansion or reduction is needed, it takes $O(\log n)$ steps.

Thus, for Stage I (lines 1-4) is essentially a sequence of algebraic computations followed by constructing fragment array F. Figure 2 shows some more details, and we can see that the algebraic computation takes $O(1)$ steps, but construction of the array F to store the fragments requires a prefix-sum followed by an expansion, and takes $O(\log n_f)$ steps, where there are n_f fragments overall.

Stage II (lines 5-15) follows a similar pattern, except for lines 11-13. Naïve implementation of this stage would use a nested loop, but can be replaced by reusing the set of fragment slab values for the short edge to generated fragment pairs, resulting in an overall $O(\log n_f)$ cost for the stage. Note that we use n_f as the parameter here, since each fragment can only occur in at most 2 pairs - one for each incident triangle.

Stage III (lines 16-28) performs the Union-Find operation in $O(\log n_f)$ steps, followed by parallel sort & duplicate suppression in $O(\log n_f)$ further steps.

Stage IV (lines 29-36) then performs the arc extraction in $O(\log n_f)$ steps, leading to an overall cost of $(O(log n_f))$ parallel steps. Provided the number of fragments per edge is small, this is about as efficient as we are likely to achieve. Further studies on this parameter could be performed, but previous work [4] indicates that the number will be related to the gradient of the field, and will (on average) be sub-linear. Even in the worst case, $n_f = O(N^2)$, where N is the number of input variables, and this still leads to $O(\log(N^2))$ parallel steps overall.

4 Hybrid Data-Parallel Algorithm

Once we have computed a data-parallel contour tree, the next task is to build a hybrid distributed version for larger data sets. For this, we observe that each of the two principal stages of the pure data-parallel algorithm described above can be parallelized by fanning in computations to progressively larger blocks. For this to work, however, we need to limit data communication to a size proportional to the boundaries of the data as with previous computations [12, 13].

We start with the interval contour contraction, and observe that each interval contour is either contained entirely within the block or intersects the boundary. If it is contained entirely within the block, it cannot merge with interval contours in any other block, and therefore does not need to passed between blocks.

Interval contours that cross the boundary, however, will merge with interval contours in other blocks, so we prepare for the fan-in by selecting only those fragments that intersect the boundaries. Unfortunately, this means that our array of fragments becomes non-contiguous, which means we will have to renumber the fragments in each parent block. To do so, we establish a unique identifier for each fragment, composed of the global ID number of the edge to which it belongs, combined with the index of the interval to which it belongs. As with the serial version, our first step in the fan-in will be to construct a single contiguous thrust-vector that lists all of these fragments for union-find.

This in turn means that we need to guarantee that the representative of each union-find component is in the set passed to the parent. We therefore perform the union-find so that fragments on the boundary between blocks are used as the representative if possible, tie-breaking with the global edge ID. This is applied separately at each level of the fan-in, since a boundary fragment at a lower level of the fan-in will normally become an interior element of a higher block in the hierarchy.

We now observe that we are incrementally gluing together contours in larger and larger blocks, at each stage discarding all interior contours, and limiting data communication to the size of the boundary between blocks. At each level of the hierarchy, the principal work is converting fragment and edge IDs from global to local IDs, performing the union-find contraction, then converting back to global IDs.

Once the process has completed at the root of the hierarchy, the root node will have the correct union-find representatives for all of the contours that cross boundaries of its immediate child nodes. To ensure that all blocks have the correct union-find representatives, we then fan this information back out, ensuring that each child node has the correct global representative for each of its local representatives, and so on.

Once this fan-out is complete, the nodes have been correctly identified, and we proceed to the vertical arcs between nodes. Again, we perform this at the child blocks first, then fan-in to get global information.

Here, we observe that these arcs can be of three types: interior-interior, interior-boundary, or boundary-boundary. In the first case, interior-interior, the arc connects two interval contours interior to the block, and does not need to be passed to the parent. This is also true in the second case: since one of the interval contours is interior, it has no impact on connectivity in other blocks. This leaves only the arcs connecting pairs of nodes on the block boundaries, which we transmit to the parent. At each level, we suppress duplicates and pass only boundary pairs to the parent, until the computation is complete, at which point all children and parent blocks have identified all arc pairs in their interior and their boundaries.

5 Results

An initial implementation of the algorithm described above has been written for regular DEM (Digital Elevation Model) data (i.e. for two-dimensional data).

For the on-node data-parallel algorithm described in Section 3, we use Nvidia's Thrust library. The transform, for_each, reduce, scan, sort, scatter, gather, and unique operators are used throughout the algorithm, along with our custom functors. Union-find representatives are found as described in Section 2.1, followed by an additional step in which the requirement that the chosen representative be a fragment on the boundary if possible (with ties broken by global edge ID) is satisfied by sorting the edge slabs by their initial union-find representative, and then using a segmented scan (inclusive_scan_by_key) with a customized functor to find the correct representative for each group, followed by a segmented reverse max-scan to propagate that representative ID back to all members of the group, and finally a scatter to restore the array to its original ordering. An example is shown in Figure 3 for how Thrust operators are used to generate a vector of fragments in Stage I of Figure 2. One significant advantage of using a data-parallel algorithm implemented using a portable library such as Thrust is that the exact same code can run on all supported architectures, including GPUs (with the CUDA backend) and multi-core CPUs (with the OpenMP and TBB backends), while the serial backend can help make debugging easier.

The hybrid algorithm described in Section 4 is implemented using MPI. Data that must be transferred during the fan-in and fan-out stages is copied from Thrust device vectors (which may reside in the memory of accelerators such as GPUs) to Thrust host vectors on the CPU, which can then be passed to MPI using raw pointers. The input data is partitioned among the ranks according to a domain decomposition with a specified block size, with the data along boundaries shared between each pair of adjacent blocks.

Results of the data-parallel and hybrid algorithms were verified against a quantized version of the standard serial algorithm [5]. Here, the serial algorithm was used to extract the contour tree, of

```
// The number of fragments to be generated for each edge, taken as input for this example
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
nEdgeFrags	2	2	3	2	2	1	0	0	3	2	0	0	2						

```
// Use a prefix sum to get the starting index for each edgeID in the new fragment vector
thrust::exclusive_scan(nEdgeFrags.begin(), nEdgeFrags.end(), bEdgeFrags.begin(), 0, thrust::plus<signed long>());
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
bEdgeFrags	0	2	4	7	9	11	12	12	12	15	17	17	17						

```
// The total number of fragments is the starting index for the last edgeID, plus the number of fragments for that last edge
signed long numFrags = bEdgeFrags.back() + nEdgeFrags.back();
thrust::device_vector<signed long> fragEdge(numFrags, 0);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
numFrags	19																		
fragEdge	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

```
// For edges with at least one fragment, scatter the edge indices (from a counting iterator) to the indices in fragEdge specified by bEdgeFrags
thrust::scatter_if(thrust::make_counting_iterator(0),                    // Beginning of input indices (counting iterator not actually stored in memory)
                   thrust::make_counting_iterator(0)+nEdgeFrags.size(),  // End of input indices
                   bEdgeFrags.begin(),                                   // Indices to which to scatter the input
                   nEdgeFrags.begin(),                                   // Only scatter input for which this stencil evaluates to true
                   fragEdge.begin(),                                     // Beginning of output vector
                   threshold(0));                                        // Predicate applied to stencil vector
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
counting_iterator	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
fragEdge	0	0	1	0	2	0	0	3	0	4	0	5	8	0	0	9	0	12	0

```
// Use a max-scan to propagate the edge id to the other fragments corresponding to each edge
thrust::inclusive_scan(fragEdge.begin(), fragEdge.end(), fragEdge.begin(), thrust::maximum<signed long>());
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
fragEdge	0	0	1	1	2	2	2	3	3	4	4	5	8	8	8	9	9	12	12

Figure 3: Code used to generate fragments from a vector containing the number of fragments to generate for each edge. This example illustrates how parts of Stage I from Figure 2 are implemented using Thrust.

which the arcs were quantized with respect to the same slab size. Any edge with exactly one fragment was discarded, as all of the corresponding contours belonged to the same interval contour, and would therefore be collapsed into a node rather than an arc. Any edge with at least two fragments then generated one arc for each interval boundary crossed. Although this does the quantization after the contour tree computation, not before, it is not hard to show that the result is the same as the algorithm described above.

Since slab, edge, and fragment ids are specific to individual implementations and domain decompositions, results were compared based on the number of edges in the contour tree proceeding out of each slab. Results were verified in this way for a simple 3x3 test case using 4 ranks (2x2 blocks, each of size 2x2), and for an 18x21 elevation data set for Vancouver, Canada, at quantization 5m (for data in the range of 0 to 91m), using 1 rank, 9 ranks (3x3 blocks of size 8x8), and 16 ranks (4x4 blocks of size 6x6). In cases where the data size is not evenly divisible by the block size, the right-most and/or bottom-most blocks were left undersized as needed.

We have also run scaling studies for our algorithms using a 4800x4800 chunk of data from the GTOPO30 database, which contains elevation maps for the Earth at a horizontal grid spacing of 30 arc seconds (roughly one-half to one kilometer). The chunk we used spans a topologically interesting region covering India and the Himalayas. Tests were run on the Moonlight supercomputer at Los Alamos National Laboratory. Each node has a 16-core 2.6 GHz Intel Xeon E5-2670 CPU, 64 GB of RAM, and two Nvidia Tesla M2090 GPUs (although we only used one per node in our tests). A large quantization level (1000 m) was used to meet the memory constraints for tests run on a single node, and the contour trees produced for this data set have not yet been independently vetted. Our code is currently limited to on the order of 100,000,000 edge fragments in the 64 GB of memory available per node, although significant opportunities for optimizations in memory usage exist (see Section 7). Scaling with the number of OpenMP threads, using Thrust's OpenMP backend (along with our custom OpenMP parallel backend for the scan operator, since Thrust provides only a serial scan for OpenMP), is shown in Figure 4. Figure 5 shows the scaling with the number of MPI ranks, with one rank per node, up to 16 nodes. The same test was also run on the GPUs of 16 nodes, using the same code, by compiling to Thrust's CUDA backend, with performance comparable in this case to the 16-node, 16-thread OpenMP test (1.87 and 1.69 seconds, respectively). While the OpenMP scaling tails off after around 4 threads on these 16-core machines, the scaling with the number of nodes does well up to at least 16 nodes, as the amount of communication necessary in the hybrid algorithm is relatively small (only data at the boundaries between blocks).

We were also able to compute the contour tree for this data set at a quantization of just 10m by running it across 64 nodes on Moonlight with OpenMP, with one rank per node and block sizes of 601x601, in 29.2 seconds. Using the GPUs, in about 10.6 seconds, we were able to compute the contour tree at a quantization of 50m (smaller quantizations exceeded the memory constraints of the GPUs).

6 Conclusions

We have presented the first algorithm for computing contour trees using a quantized approach that exploits both shared-memory data-parallelism and distributed-memory domain decomposition parallelism. Our initial implementation of this method, using Nvidia's Thrust library and MPI, has been used to verify the correctness of the algorithm for small data sets, and to demonstrate its parallel scaling with larger data sets. This work lays the foundation for using contour trees as a tool for automatically finding contours of interest in massive data sets.

7 Future Work

The current initial implementation of the algorithm uses many intermediate vectors for the sake of simplicity and clarity, but the

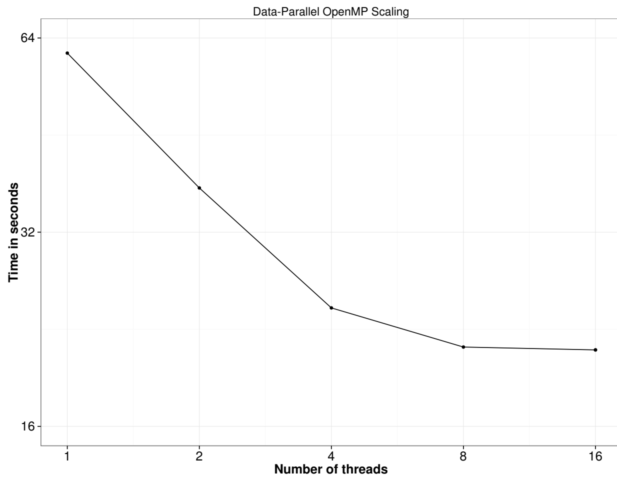

Figure 4: Data-parallel scaling on a single node (log-log plot), for a 4800x4800 chunk of elevation data in India and the Himalayas from GTOPO30 at quantization 1000m.

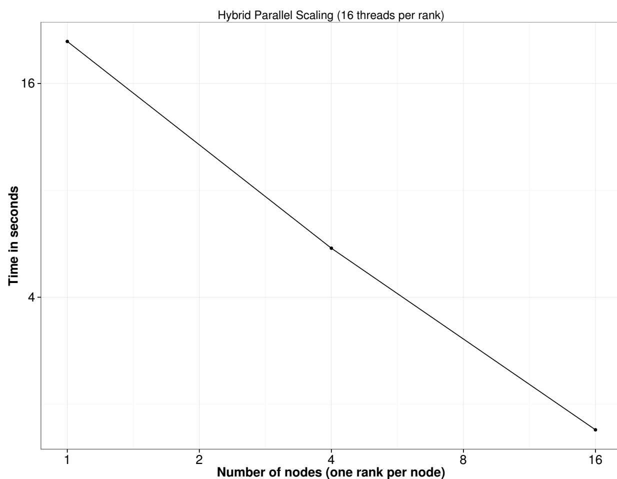

Figure 5: Hybrid scaling across nodes (log-log plot), for a 4800x4800 chunk of elevation data in India and the Himalayas from GTOPO30 at quantization 1000m.

memory footprint could likely be significantly decreased by optimally reusing allocated vectors. More work is also needed to verify the results for very large data sets, such as those from GTOPO30. We believe that it is possible to extend the *post facto* quantization of the serially computed contour tree to obtain the exact fragment representatives, &c. so that validation of the computation may be performed automatically.

Thereafter, the secondary computations such as geometric simplification of the contour tree, bounding hierarchy extraction, single contour extraction, and so forth, still need to be implemented. These steps are of particular interest within in-situ frameworks such as the Cinema image database [1], as they are necessary for automated feature selection.

Moreover, while the current version is written on the assumption of a 2D DEM (the easiest case to implement), all that would need changing for an arbitrary tessellation would be to use hashing or other reverse lookup techniques in all of the places where we use modular arithmetic to compute edge & fragment IDs. Equally, since tetrahedra can be quantized into planar interval contours in the same way as triangles, the algorithm will naturally extend to tetrahedral meshes at the cost of some additional book-keeping. Further development for non-simplicial meshes should also be possible.

We also note that, unlike the original contour tree algorithm, this version naturally handles Reeb graph computation as well, and we would like to implement and test this for $2-manifold$ surfaces and non-simple meshes as well. Variations on the approach described may also allow Morse-Smale Complex extraction as well, but this is purely speculative at present.

References

[1] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen. An image-based approach to extreme scale in situ visualization and analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 424–434, 2014.

[2] T. F. Banchoff. Critical Points and Curvature for Embedded Polyhedra. *Journal of Differential Geometry*, 1:245–256, 1967.

[3] G. Blelloch. *Vector Models for Data-Parallel Computing*. PhD thesis, MIT, 1990.

[4] H. Carr and D. Duke. Joint Contour Nets. *IEEE Transactions on Visualization and Computer Graphics*, 20(8):1100–1113, 2014.

[5] H. Carr, J. Snoeyink, and U. Axen. Computing Contour Trees in All Dimensions. *Computational Geometry: Theory and Applications*, 24(2):75–94, 2003.

[6] H. Edelsbrunner, J. Harer, and A. K. Patel. Reeb Spaces of Piecewise Linear Mappings. In *Proceedings of ACM Symposium on Computational Geometry*, pages 242–250., 2008.

[7] B. Guo. Interval Set: A Volume Rendering Technique Generalizing Isosurface Extraction. *Proceedings of IEEE Visualization*, 1995.

[8] K. Heitmann, N. Frontiere, C. Sewell, S. Habib, A. Pope, H. Finkel, S. Rizzi, J. Insley, and S. Bhattacharya. The Q Continuum Simulation: Harnessing the Power of GPU Accelerated Supercomputers. To appear in the Astrophysical Journal Supplement., 2015.

[9] M. Hilaga, Y. Shinagawa, T. Kohmura, and T. L. Kunii. Topology Matching for Fully Automatic Similarity Estimation of 3D Shapes. In *SIGGRAPH 2001*, pages 203–212, 2001.

[10] J. JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[11] L.-T. Lo, C. Sewell, and J. Ahrens. PISTON: A Portable Cross-Platform Framework for Data-Parallel Visualization Operators. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, pages 11–20, 2012.

[12] D. Morozov and G. Weber. Distributed Merge Trees. *ACM SIGPLAN Notices*, 48(8):93–102, 2013.

[13] D. Morozov and G. Weber. *Distributed Contour Trees*. Topological Methods in Data Analysis and Visualization II. Springer, 2014.

[14] V. Pascucci and K. Cole-McLaughlin. Parallel Computation of the Topology of Level Sets. *Algorithmica*, 38(2):249–268, 2003.

[15] G. Reeb. Sur les points singuliers d'une forme de Pfaff complètement intégrable ou d'une fonction numérique. *Comptes Rendus de l'Acadèmie des Sciences de Paris*, 222:847–849, 1946.

[16] C. Sewell, K. Heitmann, L.-T. Lo, S. Habib, and J. Ahrens. Utilizing Many-Core Accelerators for Halo and Center Finding within a Cosmology Simulation. In submission., 2015.

[17] C. Sewell, L.-T. Lo, and J. Ahrens. Portable Data-Parallel Visualization and Analysis in Distributed Memory Environments. In *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, pages 25–33, 2013.

[18] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

[19] W. Widanagamaachchi, C. Christensen, P.-T. Bremer, and V. Pascucci. Interactive Exploration of Large-Scale Time-Varying Data Using Dynamic Tracking Graphs. In *Proceedings of Large-Scale Data Analysis and Visualization (LDAV)*, pages 9–17, 2012.