A Hybrid Parallel Algorithm for Computing and Tracking Level Set Topology

Senthilnathan Maadasamy*, Harish Doraiswamy* and Vijay Natarajan*†
*Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, INDIA.
†Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore 560012, INDIA.
{senthil, harishd, vijayn}@csa.iisc.ernet.in

***Abstract*—The contour tree is a topological abstraction of a scalar field that captures evolution in level set connectivity. It is an effective representation for visual exploration and analysis of scientific data. We describe a work-efficient, output sensitive, and scalable parallel algorithm for computing the contour tree of a scalar field defined on a domain that is represented using either an unstructured mesh or a structured grid. A hybrid implementation of the algorithm using the GPU and multi-core CPU can compute the contour tree of an input containing 16 million vertices in less than ten seconds with a speedup factor of upto 13. Experiments based on an implementation in a multi-core CPU environment show near-linear speedup for large data sets.**

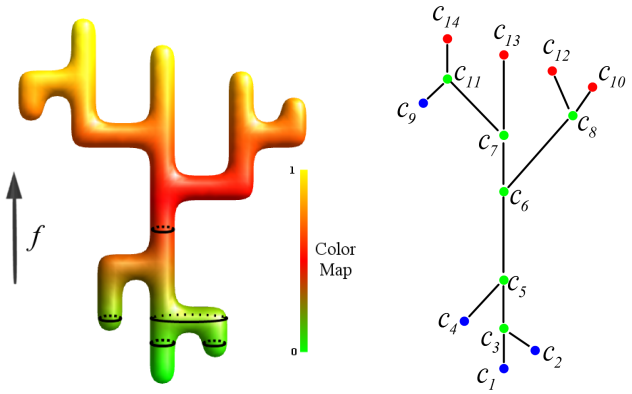
I. Introduction

A level set consists of all points where a given scalar function attains a given real value. Level sets are used extensively to visualize three and higher dimensional scalar functions. The contour tree tracks topology changes in level sets of a scalar function defined on a simply connected domain, and therefore serves as a good abstract representation of the given data. In this paper, we propose a generic parallel algorithm that computes the contour trees of scalar functions defined on unstructured meshes, as well as structured grids.

A. Motivation

We motivate the utility of contour trees with a description of how they are used to efficiently compute and explore level sets of three-dimensional scalar functions [10], [36]. The level set at a given real value can consist of multiple connected components. The contour tree of the input scalar function f defined on a simply connected domain consists of a set of nodes and arcs. The nodes correspond to critical points of the function where the number of components of the level set change, see Figure 1. An arc in the contour tree corresponds to a set of equivalent level set components [15]. An arc (c_i, c_j) *spans* a given function value f_v if $f(c_i) \leq f_v \leq f(c_j)$.

Figure 2(a) shows a volume rendering of the CT-scan of the torso of the Visible Human Male dataset [4]. The scalar value at each sample point in the input denotes the radio-density at that point. This radio-density is mapped to color and transparency to generate the volume rendering. Figure 2(b) shows a level set of this input at function value $f_v = 700$, which consists of more than three thousand components. Volume rendering and level set (isosurface) extraction are classical techniques for scalar data visualization [20], [30].



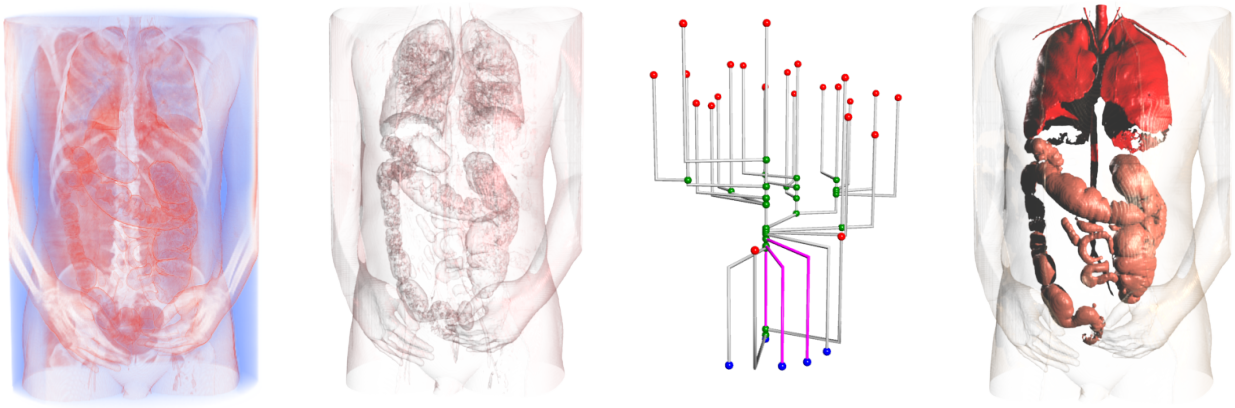
(a) Input mesh and height function (b) Contour tree

Fig. 1. Scalar function f defined on a surface mesh visualized using a color map. A few level sets of f are shown in black. The contour tree tracks the evolution of connected components of level sets of the function f.

Isosurfaces / level sets can be extracted using a variant of the marching cubes algorithm [24], which scans through the entire input to compute the level set. To improve the efficiency of the algorithm, a seed point for each of the level set components can be computed from the arcs of the contour tree that span f_v [36]. The level set is thus efficiently computed by marching through the input starting from the set of seed points. This helps in significantly reducing the overhead of extracting the level set. Additionally, by simplifying the contour tree to remove noise, and selecting seed points from significant arcs, it is possible to extract important components of the level set. Figure 2(d) shows three components of the level set that correspond to the skin, lungs and bowels respectively. The three arcs in the contour tree used to obtain the seed points are highlighted in Figure 2(c).

Contour trees have also been used in various other applications including topography and GIS [31], [34], for surface segmentation and parameterization in computer graphics [23], [27], [39], image processing and analysis of volume data sets [13], [33], designing transfer functions for volume rendering in scientific visualization [19], [32], [38], [40], and exploring high dimensional data in information visualization [21], [28].

With the exponential growth in compute power, there is a massive increase in the amount of data generated through



(a) (b) (c) (d)

Fig. 2. Extracting and exploring level sets using contour trees. **(a)** Volume rendering of a CT scan of the torso of the Visible Human Male. **(b)** Level set extracted at the function value 700 and rendered as translucent surfaces. **(c)** Simplified contour tree of the input scalar function. **(d)** Level set components corresponding to the skin, lungs and bowels are extracted using seed points from the highlighted arcs of the contour tree.

simulations and experiments. The above mentioned applications will benefit from a fast algorithm that can compute the contour tree at close to interactive rates for large data sizes. The pervasive availability of multi-core CPUs and programmable GPUs drive the development of parallel algorithms for topological structures such as the contour tree. An efficient parallel algorithm to compute the contour tree will help in greatly reducing the processing time in such scenarios.

B. Related Work

de Berg et al. [14] developed the first algorithm to compute the contour tree and applied it to GIS applications for answering elevation related queries. This algorithm uses a divide and conquer strategy and computes the contour tree of a two-dimensional scalar function in $O(n \log n)$ time, where n is the number of triangles in the input. van Kreveld et al. [36] developed an algorithm that explicitly maintains the evolving level sets in order to compute the contour tree. This algorithm has a running time of $O(n \log n)$ for two-dimensional input, and $O(n^2)$ for three-dimensional input.

Tarasov and Vyalyi [35] described an $O(n \log n)$ algorithm that computes the contour tree of a three-dimensional scalar function. This algorithm performs two sweeps over the input in decreasing and increasing order of function value to identify the joins and splits of the level set components. The contour tree is computed by merging the results of the first two sweeps. Carr et al. [9] simplified and extended this approach, and proposed an elegant algorithm that computes contour trees of scalar functions in all dimensions. In two sweeps over the input, their algorithm computes a join tree and a split tree, which tracks the evolution of super- and sub-level sets respectively. These two trees are then merged to obtain the contour tree. This algorithm has a running time of $O(v \log v + n\alpha(n))$, where v is the number of vertices in the input, and α is the inverse Ackermann function.

Chiang et al. [11] proposed an output sensitive approach that first finds all component critical points that correspond to nodes in the contour tree using local neighborhoods. These critical points are connected using monotone paths in a second step to obtain the join and split trees. This algorithm has a running time of $O(t \log t + n)$, where t is the number of critical points of the input. van Kreveld et al. [37] showed a $\Omega(t \log t)$ lower bound for the construction of contour trees. Since reading the input takes $O(n)$ time, the output sensitive algorithm is optimal in the worst case.

Pascucci et al. [29] proposed the first and only known parallel algorithm that computes the contour tree of a piecewise trilinear function defined on a three-dimensional structured mesh. Join and split trees of each voxel is individually computed and later merged to form the join and split trees of the input. This algorithm has a running time of $O(v + t \log v)$. Dividing the input is non-trivial for triangular meshes and not implicit as in the case of structured meshes. Hence, their algorithm does not scale well for large unstructured meshes. We refer the reader to the following surveys [6], [8], [7] for a detailed discussion of various approaches to compute the contour tree.

Existing serial algorithms are not immediately amenable to parallelization because they sweep the domain in order of increasing function values, which is an inherently sequential operation. We compute the contour tree by first computing monotone paths between critical points and next process these monotone paths directly to compute the join and split trees. Different from Chiang et al. [11], who also compute monotone paths, we allow the computation of monotone paths in any arbitrary order. Essentially, we redesign the computations such that they are parallelizable. The critical points are also located in parallel. An added advantage of this new approach is that the algorithm works for both triangulated meshes and structured grids.

C. Contributions

In this paper, we describe a fast and efficient parallel algorithm for computing the contour tree of a piecewise linear (PL) function in $O(t \log t + n)$ time. Here t is the number of critical points, and n is the number of triangles in the input. The same algorithm also computes the contour tree of a trilinear function defined on a structured grid in $O(t \log t + v)$ time, where v is the number of vertices in the input. The algorithm has the following desirable properties:

- **Output sensitive**. The running time depends on the size of the output.
- **Optimal**. The sequential algorithm is optimal in the worst case.
- **Generic**. The algorithm works without any modification on a piecewise-linear function in any dimension. It also handles trilinear functions defined on structured grids.
- **Easily parallelizable**. The main operations performed by the different steps of the algorithm are independent of each other.
- **Work-efficient**. The parallel implementation does not perform additional work compared to the serial algorithm.

We report experimental results that demonstrate the efficiency of our algorithm. Specifically, we show that a hybrid implementation of the algorithm using both the GPU and a multi-core desktop CPU can be used to compute the contour tree of an input with around 16 million vertices in less than ten seconds. The same parallel implementation using OpenCL gives a speedup of upto a factor of 25 on a 32-core CPU environment. Additionally, the time taken by the sequential implementation is comparable or lower than existing algorithms, demonstrating that the algorithm is efficient even on a single core.

II. Background

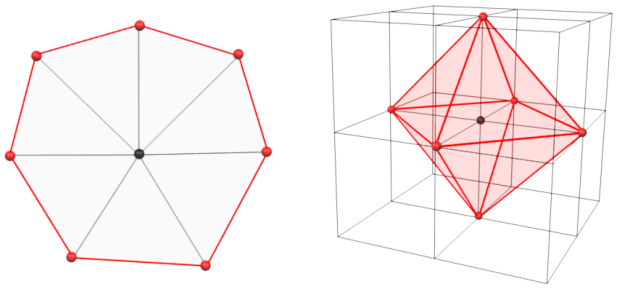
In this section, we briefly introduce the necessary definitions used in this paper. We refer the reader to books on computational topology, algebraic topology, and Morse theory [16], [22], [25], [26] for a detailed discussion of these concepts.

A. Scalar function and domain

A *scalar function* maps points from a spatial domain to real values. The domain is usually represented as a mesh. The scalar function is defined at the vertices of the mesh, and interpolated within each cell of the mesh. In this paper, we consider two types of mesh representations.

Unstructured mesh. A *d-simplex* is the convex hull of $d+1$ affinely independent points. For example, a 0-simplex is a vertex, 1-simplex is an edge, 2-simplex a triangle, and 3-simplex is a tetrahedron. The input domain is represented as a set of non-intersecting simplicies. The scalar function is a piecewise linear (PL) function defined at the vertices of the mesh, and linearly-interpolated within each simplex.

Structured grid. Three dimensional imaging data is often available as scalar values sampled on a three dimensional rectilinear grid. For such an input, the scalar values in the interior of a cell is computed using trilinear interpolation.



(a) Vertex in a 2D mesh (b) Vertex in a 3D grid

Fig. 3. Neighborhood of a vertex. The link of a vertex, which is used to classify critical points, is highlighted in red.

B. Level sets and critical points

Consider a scalar function f defined on a domain D. The *level set* corresponding to a real value f_i is defined as the set $\{x \in D | f(x) = f_i\}$. A *sub-level* set is the set $\{x \in D | f(x) \leq f_i\}$, while a *super-level set* is the set $\{x \in D | f(x) \geq f_i\}$. Consider the evolution of the level sets with increasing function value. Points at which the topology of the level sets change during this evolution are known as *critical points*. Points that are not critical are called *regular points*.

Morse theory studies the relationship between critical points of a scalar function and the topology of its level sets. If all critical points of f are isolated and non-degenerate, then f is a *Morse function* [25], [26]. Critical points of a Morse function can be classified based on the behavior of the function within a local neighborhood. The above condition typically does not apply for piecewise linear and piecewise trilinear functions. However, a simulated perturbation of the function [18, Section 1.4] ensures that no two critical values are equal. The simulated perturbation imposes a total order on the vertices and helps in consistently identifying the vertex with the higher function value between a pair of vertices. Consequently, critical points can be identified and classified based on local behavior of the function. We now describe this classification for the two kinds of meshes that we consider in this work.

Unstructured mesh. Consider a vertex u in the input mesh shown in Figure 3(a). The *link* of u is the set of all vertices adjacent to u together with the induced edges, triangles, and higher-order simplices. Adjacent vertices with lower function value and their induced simplices constitute the *lower link*, whereas adjacent vertices with higher function value and their induced simplices constitute the *upper link*. Figure 3(a) highlights the link of u within a two-dimensional triangular mesh in red.

Banchoff [5] and Edelsbrunner et al. [17] describe a combinatorial characterization for critical points of a PL function, which are always located at vertices of the mesh. Critical

points are characterized by the number of connected components of the lower and upper links. The vertex is regular if it has exactly one lower link component and one upper link component. All other vertices are critical. A critical point is a *maximum* if the upper link is empty and a *minimum* if the lower link is empty. Else, it is classified as a *saddle*.

Structured grid. In case of a structured grid, the local neighborhood of a vertex u is described by its neighboring six vertices. Define the link of u as the triangulation of these six vertices, see Figure 3(b). Critical points are again classified based on the number of components in the lower and upper links. In addition to critical points that are present at the vertices of the mesh, saddles may be located within a face or a cell of the grid. These *face saddles* and *body saddles* can be computed explicitly or their number can be inferred from a count of the number of local minima and maxima within a face and a cell as shown by Pascucci and Cole-McLaughlin [29].

C. Contour tree

Given a scalar function f defined on a simply connected domain, the *contour tree* of f is obtained by contracting each connected component of a level set to a point. It expresses the evolution of connected components of level sets as a graph whose nodes correspond to critical points of the function. Figure 1(b) shows the contour tree of the height function defined on the model shown in Figure 1(a). The blue, red, and green nodes denote the set of minima, maxima, and saddles respectively. Note that not all saddles correspond to degree-3 nodes in the contour tree. In particular, the topology of the level sets, say the genus, does change when it sweeps past these critical points but the number of connected components of the level set remains unchanged.

III. Contour tree algorithm

This section describes our contour tree computation algorithm. We assume that the input domain is simply connected. Given a mesh together with scalar values defined at its vertices, the algorithm computes the contour tree of this input in four steps:

1) Identify critical points.
2) Compute the *join tree*, which tracks the connectivity of super-level sets of the input.
3) Compute the *split tree*, which tracks the connectivity of the sub-level sets of the input.
4) Merge the join and split tree to construct the contour tree.

The key difference between the proposed method and that of Chiang et al. [11] and Carr et al. [9] is the computation of the join tree (split tree) using ascending (descending) paths between a critical point and a maximum (minimum). This approach enables a parallel implementation of the algorithm. The rest of this section is organized as follows. We first give a detailed description of the different steps of the algorithm in Sections III-A–III-C. For ease of explanation, we first describe the algorithm as a sequential procedure. We analyze the time complexity of the algorithm in Section III-D.

Procedure FindAscPaths

Input: Critical points C
Output: Path lists L_m for each maximum c_m, maximum lists M_i for each critical point c_i

```
1: for each critical point c_i ∈ C do
2:   for each component in the upper link of c_i do
3:     Trace an ascending path from c_i to a maximum c_m
4:     Add c_i to the path list L_m corresponding to the
       maximum c_m
5:     Add the maximum c_m to M_i
6:   end for
7: end for
```

A. Identify critical points

The algorithm uses the classification described in Section II-B to identify the set of critical points of the input. It counts the number of components of the upper and lower links of every vertex via a breadth first search in the graph formed by vertices and edges in the upper and lower links respectively. In addition to classifying vertices, the algorithm also stores one representative vertex from each component of the upper and / or lower links of critical vertices.

B. Join tree computation

The join tree of the input is computed in two phases. In the first phase, a set of monotonically increasing paths are created, which are merged in the second phase to obtain the join tree. The same procedure is used to compute the split tree after reversing the order on vertices defined by the scalar function.

Find ascending paths. Beginning from each component of the upper link of every critical point, the algorithm traces a monotonically increasing path from the critical point to reach a maximum. This is accomplished by a simple traversal starting from the representative vertex of each upper link component, and moving to a vertex with a higher function value until the traversal reaches a maximum. Each maximum c_m maintains a list L_m consisting of all critical points whose ascending paths terminate at that maximum. Procedure FindAscPaths outlines this computation. Figure 4 shows the output of this procedure for the input shown in Figure 1(a). The set of all maxima reached from a critical point c_j is stored in a list M_j.

Construct join tree. Procedure ConstructJoinTree uses the following property of the join tree to construct it in a bottom-up manner.

Property 1: Each node in the join tree has at most one neighbor with a lower function value.

Figure 5 illustrates the first three iterations of this procedure for the input shown in Figure 1(a). The join tree initially consists of the set of maxima, which also forms the initial set of "growing" nodes. In each iteration, the algorithm connects a growing node with the highest vertex in the associated path list and removes the latter from the path list of the former.

Maximum List	
$M_1 = \{c_{12}\}$	$M_6 = \{c_{10}, c_{13}\}$
$M_2 = \{c_{13}\}$	$M_7 = \{c_{13}, c_{14}\}$
$M_3 = \{c_{13}\}$	$M_8 = \{c_{10}, c_{12}\}$
$M_4 = \{c_{10}\}$	$M_9 = \{c_{14}\}$
$M_5 = \{c_{13}\}$	$M_{11} = \{c_{14}\}$

Path List	
$L_{10} = \{c_4, c_6, c_8\}$	$L_{14} = \{c_7, c_9, c_{11}\}$
$L_{12} = \{c_1, c_8\}$	$L_{13} = \{c_2, c_3, c_5, c_6, c_7\}$

Fig. 4. Output of Procedure FINDASCPATHS. M_j stores all maxima that are reached from critical point c_j via an ascending path. L_m stores all critical points whose ascending paths terminate at maximum c_m

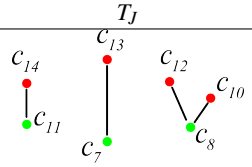
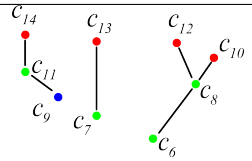
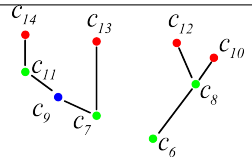
#	Growing Nodes	L_m	Join Tree T_J
1	c_{10} c_{12} c_{13} c_{14}	$\{c_4, c_6, c_8\}$ $\{c_1, c_8\}$ $\{c_2, c_3, c_5, c_6, c_7\}$ $\{c_7, c_9, c_{11}\}$	
2	c_8 c_{11}	$\{c_1, c_4, c_6\}$ $\{c_7, c_9\}$	
3	c_9	$\{c_7\}$	

Fig. 5. Constructing the join tree. The growing nodes and their path lists are updated at each iteration.

Procedure ConstructJoinTree

Input: Set of critical points C
Input: Path lists L_m of every maximum
Input: Maximum lists M_i of every critical point
Output: Join Tree T_J

1: **for** each critical point c_i that is not a maximum **do**
2: Initialize $L_i = \emptyset$
3: **end for**
4: Initialize the set *GrowingNodes* to be the set of maxima
5: Initialize the set $NewNodes = \emptyset$
6: **while** $GrowingNodes \neq \emptyset$ **do**
7: **for** each critical point $c_i \in GrowingNodes$ **do**
8: Let c_k be the critical point with maximum function value in L_i
9: Add arc (c_k, c_i) to the join tree T_J
10: Remove c_k from L_i
11: **if** number of join tree arcs incident on c_k = $|M_k|$ **then**
12: Add c_k to *NewNodes*
13: **end if**
14: **end for**
15: **for** each critical point c_k in *NewNodes* **do**
16: **for** each join tree arc (c_k, c_i) **do**
17: Set $L_k = L_k \cup L_i$
18: Replace c_i with c_k in all maximum lists M_j that contain c_i
19: **end for**
20: **end for**
21: Set $GrowingNodes = NewNodes$
22: Set $NewNodes = \emptyset$
23: **end while**
24: **return** Join tree T_J

When a node is connected to all critical points in its list M_j, it is inserted into the set of growing nodes for the next iteration. This operation essentially removes a leaf node after it is processed, thereby creating a new leaf node, a growing node. All ascending paths that terminate at the removed leaf will now terminate at the newly created growing node. The associated path list of a new growing node is computed as the union of path lists of maxima that it is connected to. The second iteration in Figure 5 illustrates this operation, in which c_8 and c_{11} become the growing nodes. The above procedure is repeated until there are no more growing nodes. Figure 6 shows the resulting join and split trees.

Correctness. We now prove the correctness of the join tree construction algorithm. In order to show that the tree computed by the above method is indeed the join tree, it is sufficient to show that, for each critical point, the algorithm correctly identifies the unique neighbor having a lower function value.

Consider the first iteration, when the set of growing nodes is equal to the set of maxima. Let c_m be a maximum under consideration. Let (c_k, c_m) be an arc of the join tree, that is, c_k is the unique neighbor of c_m in the join tree. This implies that there exists a monotone ascending path from c_k to c_m starting from one of the components of the upper link of c_k [11]. Let u_i be the representative vertex of such an upper link component. Also, since c_k is a neighbor of c_m in the join tree, it follows that there is no other critical point in the path between u_i and c_m. Therefore, the algorithm traces this path, and adds c_k to the path list L_m of the maximum c_m. To show that c_k has the maximum function value among all critical points in L_m, let us assume, for sake of contradiction, that there exists $c_l \in L_m$ such that $f(c_l) > f(c_k)$. So, there exists an ascending path from c_l to c_m. This implies that, when tracking super-level set connectivity during a downward sweep in function value, the super-level set component that is created at c_m, reaches c_l before c_k. Hence c_l is the neighbor of c_m, a contradiction.

Let c_n be one of the growing nodes at the end of the first iteration. This implies that c_n is connected to all the maxima in its set M_n. The set L_n essentially stores all paths that ended in one of the maxima present in M_n. Removing these maxima from the input results in c_n becoming a leaf node similar to a maximum. Repeating the above argument for c_n, we prove that the unique neighbor of the new leaf

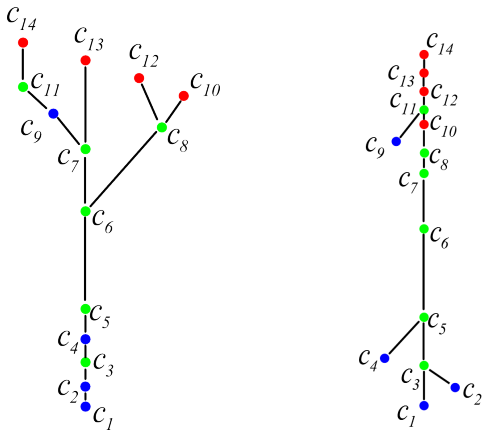


Fig. 6. Join tree (left) and split tree (right) of the input shown in Figure 1(a).

node is correctly identified during the next iteration. Hence, the algorithm correctly computes all arcs of the join tree.

C. Constructing the contour tree

The contour tree is constructed from the join and split trees using the merge procedure described by Carr et al. [9]. For completeness, we outline this algorithm in Procedure MERGETREES. Here, up-neighbors and down-neighbors correspond to neighbors having higher and lower function values respectively. Up-degree and down-degree correspond to the number of up-neighbors and down-neighbors respectively. Each iteration in this procedure identifies an arc of the contour tree that is incident on a leaf, removes the arc from the join and split tree and inserts it into the contour tree. This process is repeated until all arcs of the join and split tree are processed.

Procedure MergeTrees

Input: Join tree T_J, Split tree T_S
Output: Contour tree T_C

Set $Queue = \emptyset$
for each critical point c_i **do**
  if up-degree(c_i) in T_J + down-degree(c_j) in T_S = 1 **then**
    Add c_i to $Queue$
  end if
end for
Initialize the set $NextSet = \emptyset$
while $Queue \neq \emptyset$ **do**
  for each critical point $c_i \in Queue$ **do**
    if c_i is a leaf in T_J **then**
      set $c_k \leftarrow$ down-neighbor(c_i) in T_J
    else
      set $c_k \leftarrow$ up-neighbor(c_i) in T_S
    end if
    Add arc (c_i, c_k) to the contour tree T_C
    Remove c_i from T_J and T_S
    if up-degree(c_k) in T_J + down-degree(c_k) in T_S = 1 **then**
      Add c_k to $NextSet$
    end if
  end for
  Set $Queue = NextSet$
  Set $NextSet = \emptyset$
end while
return Contour tree T_C

D. Running time

Let v be the number of vertices and t be the number of critical points of the input scalar function. Let n be the number of triangles when the input is a triangular mesh. For a structured grid input, n denotes the number of cells and $n = \Theta(v)$. Identifying critical points requires computing the link of a vertex, which is accomplished by traversing all triangles / cells incident on the vertex. Each triangle / cell is therefore accessed a constant number of times in total and hence identifying critical points can be accomplished in $O(n)$ time. When tracing a monotone ascending path, if the traversal reaches a vertex that was already reached by another path, then this information can be used to immediately identify the maximum that ends the path. Hence, each vertex is accessed a constant number of times and tracing all monotone paths takes $O(v)$ time.

While constructing the join tree from the set of monotone paths, the processing performed at each critical point requires identifying the highest vertex in its path list, and computing the union of the path lists of its neighbors. Using an efficient data structure such as a Fibonacci heap, or a skew heap [12], each of these operations require $O(\log t)$ time. Each critical point contributes to one highest vertex computation, and one union operation. Summing over all critical points, a total of $2t$ operations are performed on the data structure used to maintain path lists. Maximum lists are updated in response to the removal of growing nodes using the union-find data structure. This operation requires $O(t\alpha(t))$ time, where α is the inverse Ackermann function. Combining the above steps, we obtain an $O(t \log t)$ bound on the time taken to construct the join tree. Note that the same algorithm is used to compute the split tree. So, the split tree can also be computed in $O(t \log t)$ time.

Merging the join and split tree takes $O(t)$ time. Combining all the steps, we obtain a $O(t \log t + n)$ time algorithm to compute the contour tree. Note that this is the optimal bound for the sequential computation of contour trees [37].

IV. PARALLEL IMPLEMENTATION

The algorithm described in the previous section can be immediately parallelized. We now describe such an implementation on a shared memory environment.

A. Critical point identification

The classification procedure described in Section III-A requires the connectivity of the local neighborhood of each vertex and can therefore be computed independently for each vertex. This procedure is embarrassingly parallel.

TABLE I
CONTOUR TREE COMPUTATION TIME FOR UNSTRUCTURED MESHES. LIBTOURTRE PERFORMS BETTER THAN PARALLELCT ON A SINGLE CORE FOR SMALL UNSTRUCTURED MESHES.

Model	Input size		Time taken (sec)	
	# Vertices	# Tetrahedra	PARALLELCT (1 core)	LIBTOURTRE
Torso	168,930	1,082,723	0.72	0.19
Bucky Ball	262,144	1,250,235	0.49	0.24
Plasma	274,626	1,310,720	0.51	0.28
SF Earthquake	378,747	2,067,739	0.95	0.36

TABLE II
CONTOUR TREE COMPUTATION TIME FOR STRUCTURED GRIDS. PARALLELCT ON A SINGLE CORE PERFORMS BETTER THAN LIBTOURTRE.

Model	Input size		Time taken (sec)	
	# Vertices	# Cells	PARALLELCT (1 core)	LIBTOURTRE
Head MRT	$256 \times 320 \times 128$	$255 \times 319 \times 127$	27.7	74.3
Boston Teapot	$256 \times 256 \times 178$	$255 \times 255 \times 177$	10.1	11.4
Vis. Human Torso	$256 \times 256 \times 226$	$255 \times 255 \times 225$	36.0	152.3
Aneurism	$256 \times 256 \times 256$	$255 \times 255 \times 255$	10.5	12.4
Bonsai	$256 \times 256 \times 256$	$255 \times 255 \times 255$	20.2	19.0
Foot	$256 \times 256 \times 256$	$255 \times 255 \times 255$	21.5	31.5

B. Join tree construction

The ascending path from each critical point can be computed independent of each other. Thus, computation of the set of ascending paths is embarrassingly parallel.

Each iteration of the Procedure CONSTRUCTJOINTREE, from Line 7 to Line 14, processes a set of maxima. Each maximum maintains its own path list. Thus, the arc in the join tree incident on a maximum can be computed independent of other maxima. We parallelize this arc identification process. Similarly, updating the path lists of the growing nodes and the maximum lists (Line 15 - Line 20) can also be implemented in parallel for each growing node.

The number of growing nodes that are processed decreases with each iteration. It is possible that this number quickly becomes one, after which the join tree construction is essentially sequential. Figure 5 shows one such example where the number of growing nodes decreases to one after the second iteration. However, note that once the number of growing nodes reduces to one, the remaining critical points are processed in decreasing order of function value. To improve the performance of the algorithm in such scenarios, we modify the algorithm to stop the CONSTRUCTJOINTREE procedure when there is a single growing node. Instead, the unprocessed critical points are sorted in decreasing order of function value. Arcs are then added between consecutive critical points in this list. The advantage of this modification is two-fold: (1) the set of critical points can be sorted in parallel, and (2) the time complexity of the sequential step (adding the remaining arcs) is reduced from $O(t \log t)$ to $O(t)$.

C. Merge procedure

We chose to use the sequential implementation of the merge procedure, since the time taken to perform this operation was negligible compared to the overall time required to construct the contour tree. However, note that different iterations of this step (Lines 9-20), as outlined in Procedure MERGETREES, can be parallelized.

V. EXPERIMENTAL RESULTS

We implement the steps corresponding to the identification of critical points, and computing the ascending and descending paths using OpenCL [2], a parallel programming framework designed for heterogeneous architectures such as multi-core CPUs and GPUs. Construction of the join and split trees using the ascending and descending paths is implemented using OpenMP [3]. This is because the current OpenCL specification does not support global barriers, which is required for an efficient implementation of the join / split tree construction algorithm. We evaluate the performance of the algorithm on an 8-core Intel Xeon workstation with 16 GB memory and each core running at 2.0 GHz. The workstation includes a NVIDIA GeForce GTX 460 GPU having 336 cores and 1GB memory. In Section V-A, we report the performance of our algorithm on a single processor. We evaluate the performance of the hybrid implementation in Section V-B. Finally, in Section V-C we report the performance of our implementation on a larger multi-core CPU environment.

A. Single core environment

We first compare the performance of our algorithm with LIBTOURTRE [1], a publicly available and widely used implementation of the sweep algorithm by Carr et al. [9]. We restrict our implementation to execute on a single processor for this comparison. Table I shows the contour tree computation time of the above two algorithms for 3D unstructured meshes. Table II compares the performance of our algorithm with LIBTOURTRE for 3D structured grids. The optimized LIBTOURTRE library performs better than our algorithm for unstructured meshes. However, note that for such a structured mesh input, PARALLELCT running on a single core is faster

TABLE III
SPEEDUP ACHIEVED USING THE HYBRID AND MULTI-CORE CPU IMPLEMENTATION FOR STRUCTURED GRIDS.

Model	Hybrid		8-core CPU	
	Time (sec)	Speedup	Time (sec)	Speedup
Head MRT	7.20	3.8	7.60	3.6
Boston Teapot	0.95	10.6	1.50	6.7
Vis. Human Torso	9.40	3.8	11.43	3.2
Aneurism	0.82	12.8	1.40	7.5
Bonsai	1.50	13.5	3.20	6.3
Foot	3.40	6.3	4.25	5.1

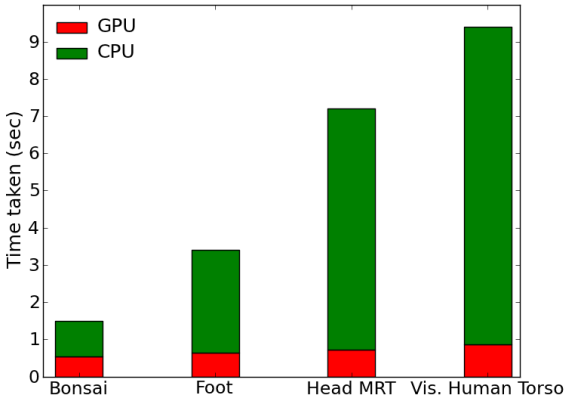


Fig. 7. Time taken by the steps executed on the GPU (red) and CPU (green) for the hybrid implementation.

than LIBTOURTRE. This can be attributed to the output-sensitive nature of our algorithm – it essentially processes only the critical points of the input. The savings in time become significant especially when the size of the input is large.

We do not report comparison of our algorithm with the output-sensitive algorithm by Chiang et al. [11] since its implementation is not available. However, we note that the output-sensitive algorithm was shown to have running times comparable with the sweep algorithm in [11].

B. Hybrid environment

In this section, we discuss the scaling behavior of the hybrid CPU-GPU implementation. The critical point classification and ascending / descending path computation, which was implemented using OpenCL, is executed on the GPU. These steps, which are embarrassingly parallel, are able to utilize the large number of cores available on current GPUs. The join / split tree computation is implemented using OpenMP, and is therefore executed on the 8-core CPU. Table III shows the time taken to compute the contour tree of structured grids. We observe a speedup of up to 13 times on the overall contour tree computation time. For the steps executed on the GPU, the speedup is at least a factor of 25. Figure 7 shows the split in the time taken by the GPU and CPU for a few data sets. Due to the memory limitation imposed by the GPU, we currently show results obtained using moderately large datasets, having approximately 16 million input vertices.

TABLE IV
SPEEDUP ACHIEVED USING THE HYBRID AND MULTI-CORE CPU IMPLEMENTATION FOR UNSTRUCTURED MESHES.

Model	Hybrid		8-core CPU	
	Time (sec)	Speedup	Time (sec)	Speedup
Torso	0.32	2.3	0.12	6.0
Bucky Ball	0.37	1.3	0.10	4.9
Plasma	0.40	1.3	0.10	5.1
SF Earthquake	0.60	1.6	0.19	5.0

Table IV depicts the scaling obtained for unstructured meshes. The benefits are not as pronounced as seen for the larger structured grids. This is because, for the smaller datasets, the total computation time itself is less than a second. The data transfer time between the CPU and GPU, which is of the order of a few hundred milliseconds, takes up a significant fraction of the overall computation time. For example, in case of the SF Earthquake dataset, the time taken to transfer the input to the GPU takes around 350 ms, which is more than half the time taken to compute the contour tree. We do, however, expect a significant benefit when the size of the input is large.

C. Multi-core CPU environment

In this section, we show the scaling obtained by our algorithm on a multi-core CPU environment. Tables III and IV show the speedup obtained when the parallel implementation was used to compute the contour tree using 8 cores. Note that, for the smaller datasets (unstructured grids), a multi-core environment provides a better speedup than a hybrid setup.

In order to demonstrate the scalability on more than 8 processors, the contour tree computation algorithm was evaluated on a 32-core AMD Opteron workstation with 64 GB memory and each core running at 2.4 GHz. The run times are reported for different number of processors. The initial setup and memory allocation time together with the time taken by the sequential steps of the algorithm, which is in the order of milliseconds, contributes to a significant fraction of the contour tree computation time for small data sets. This is because, for small data sets, the total running time is less than a second. Therefore, we only report scaling results for large data sets.

In order to show the effectiveness of our method for large data, we triangulated existing structured grids to create an unstructured mesh. Figure 8 shows the speedup obtained for such data. Note that the speedup is close to the ideal speedup, which is denoted by the blue curve.

We would like to note that the parallel algorithm by Pascucci and Cole-McLaughlin shows a similar speedup for structured grids [29]. However, they do not report results when computing contour trees of unstructured meshes. An extension of their method to unstructured meshes is non-trivial because determining a good mesh partition is costly and hence affects the speedup.

Figure 9 shows the speedup obtained with increasing number of processors for large structured grids. In our experiments, we observed that when the number of critical points increases, a significant fraction of the running time is spent in the sequential processing during the computation of the join and

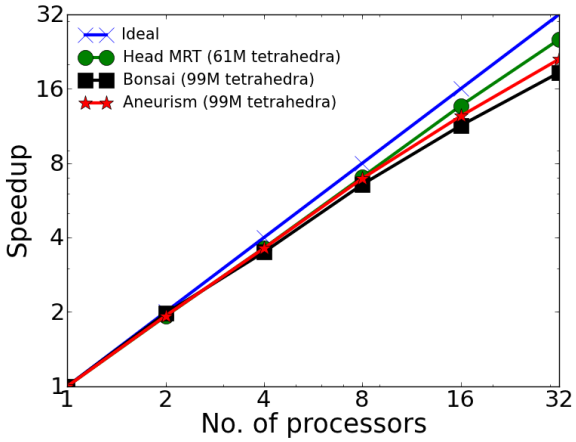


Fig. 8. Speedup for unstructured mesh input. The parallel algorithm scales almost linearly with the number of processors.

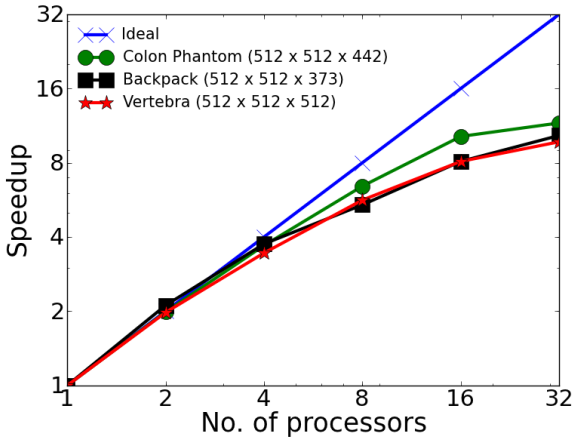


Fig. 9. Speedup for structured grid input.

split trees. This causes the dip in the curve as observed in Figure 9.

VI. Conclusions

We have presented a simple, output-sensitive and work-efficient parallel algorithm to compute the contour tree of a scalar function defined over either an unstructured mesh or a structured grid. A hybrid CPU-GPU implementation of the algorithm executed on a common desktop like environment can be used to compute the contour tree of an input with around 16 million vertices in a few seconds. The near-linear speedup obtained on various data sets also indicate that our algorithm scales well with the number of processors.

The memory limitation posed by the GPU restricts the size of input that can be processed in a hybrid environment to $256 \times 256 \times 256$ sized grids. A possible solution is to subdivide a larger grid into smaller grids and process the grids in sequence. However, it is non-trivial to directly extend this method for large unstructured meshes.

The time taken by the sequential steps of the algorithm contributes to a significant fraction of the total running time, especially when the number of critical points is large. A majority of these critical points correspond to noisy features which are removed when simplifying the contour tree. This simplification is usually performed off-line after computing the contour tree. A parallel procedure to remove such noisy critical points on the fly, would help reduce the time taken by the sequential steps of the algorithm.

Our current implementation requires the entire data to be in memory even in a multi-core CPU environment. In future, we intend to use a simple divide and conquer strategy to extend our algorithm to handle data sets that do not fit in memory. Specifically, the data can be split into blocks that fit in memory and processed to compute the contour tree of the subdomain. These contour trees could be merged together in a second step. It would also be interesting to identify methods to automatically choose the runtime environment depending on the input size.

Acknowledgments

Harish Doraiswamy was supported by Microsoft Corporation and Microsoft Research India under the Microsoft Research India PhD Fellowship Award. This work was supported by a grant from Intel India.

References

[1] libtourtre: A contour tree library. [Online]. Available: http://graphics.cs.ucdavis.edu/ sdillard/libtourtre/doc/html/
[2] Open Computing Language (OpenCL). [Online]. Available: http://www.khronos.org/opencl/
[3] Open Multiprocessing (OpenMP). [Online]. Available: http://www.openmp.org
[4] The visible human project. [Online]. Available: http://www.nlm.nih.gov/research/visible/
[5] T. F. Banchoff, "Critical points and curvature for embedded polyhedral surfaces," *Am. Math. Monthly*, vol. 77, pp. 475–485, 1970.
[6] S. Biasotti, L. De Floriani, B. Falcidieno, P. Frosini, D. Giorgi, C. Landi, L. Papaleo, and M. Spagnuolo, "Describing shapes by geometrical-topological properties of real functions," *ACM Comput. Surv.*, vol. 40, pp. 12:1–12:87, October 2008.
[7] S. Biasotti, D. Attali, J.-D. Boissonnat, H. Edelsbrunner, G. Elber, M. Mortara, G. S. Baja, M. Spagnuolo, M. Tanase, and R. Veltkamp, "Skeletal structures," in *Shape Analysis and Structuring*, ser. Mathematics and Visualization, L. Floriani, M. Spagnuolo, G. Farin, H.-C. Hege, D. Hoffman, C. R. Johnson, and K. Polthier, Eds. Springer Berlin Heidelberg, 2008, pp. 145–183.
[8] S. Biasotti, L. Floriani, B. Falcidieno, and L. Papaleo, "Morphological representations of scalar fields," in *Shape Analysis and Structuring*, ser. Mathematics and Visualization, L. Floriani, M. Spagnuolo, G. Farin, H.-C. Hege, D. Hoffman, C. R. Johnson, and K. Polthier, Eds. Springer Berlin Heidelberg, 2008, pp. 185–213.
[9] H. Carr, J. Snoeyink, and U. Axen, "Computing contour trees in all dimensions," *Comput. Geom. Theory Appl.*, vol. 24, no. 2, pp. 75–94, 2003.
[10] H. Carr and J. Snoeyink, "Path seeds and flexible isosurfaces using topology for exploratory visualization," in *Proceedings of the symposium on Data visualisation 2003*, ser. VISSYM '03. Eurographics Association, 2003, pp. 49–58.
[11] Y.-J. Chiang, T. Lenz, X. Lu, and G. Rote, "Simple and optimal output-sensitive construction of contour trees using monotone paths," *Comput. Geom. Theory Appl.*, vol. 30, no. 2, pp. 165–195, 2005.
[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, 2001.
[13] J. Cox, D. B. Karron, and N. Ferdous, "Topological zone organization of scalar volume data," *J. Math. Imaging Vis.*, vol. 18, pp. 95–117, March 2003.

[14] M. de Berg and M. J. van Kreveld, "Trekking in the alps without freezing or getting tired," *Algorithmica*, vol. 18, no. 3, pp. 306–323, 1997.

[15] H. Doraiswamy and V. Natarajan, "Output-sensitive construction of Reeb graphs," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, pp. 146–159, 2012.

[16] H. Edelsbrunner and J. Harer, *Computational Topology: An Introduction*. Amer. Math. Soc., Providence, Rhode Island, 2009.

[17] H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci, "Morse-Smale complexes for piecewise linear 3-manifolds," in *Proc. Symp. Comput. Geom.*, 2003, pp. 361–370.

[18] H. Edelsbrunner, *Geometry and Topology for Mesh Generation*. England: Cambridge Univ. Press, 2001.

[19] I. Fujishiro, Y. Takeshima, T. Azuma, and S. Takahashi, "Volume data mining using 3d field topology analysis," *IEEE Computer Graphics and Applications*, vol. 20, pp. 46–51, 2000.

[20] C. D. Hansen and C. R. Johnson, *Visualization Handbook*. Academic Press, 2004.

[21] W. Harvey and Y. Wang, "Topological landscape ensembles for visualization of scalar-valued functions," *Computer Graphics Forum*, vol. 29, pp. 993–1002, 2010.

[22] A. Hatcher, *Algebraic Topology*. New York: Cambridge U. Press, 2002.

[23] F. Hétroy and D. Attali, "Topological quadrangulations of closed triangulated surfaces using the Reeb graph," *Graph. Models*, vol. 65, no. 1-3, pp. 131–148, 2003.

[24] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 163–169, Aug. 1987.

[25] Y. Matsumoto, *An Introduction to Morse Theory*. Amer. Math. Soc., 2002, translated from Japanese by K. Hudson and M. Saito.

[26] J. Milnor, *Morse Theory*. New Jersey: Princeton Univ. Press, 1963.

[27] M. Mortara and G. Patané, "Affine-invariant skeleton of 3d shapes," in *SMI '02: Proceedings of the Shape Modeling International 2002 (SMI'02)*, 2002, p. 245.

[28] P. Oesterling, C. Heine, H. Jänicke, G. Scheuermann, and G. Heyer, "Visualization of high dimensional point clouds using their density distribution's topology," *IEEE Transactions on Visualization and Computer Graphics*, vol. 99, no. PrePrints, 2011.

[29] V. Pascucci and K. Cole-McLaughlin, "Parallel computation of the topology of level sets," *Algorithmica*, vol. 38, no. 1, pp. 249–268, 2003.

[30] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit, Third Edition*. Kitware Inc., 2007.

[31] S. Takahashi, "Algorithms for extracting surface topology from digital elevation models," in *Topological Data Structures for Surfaces: An Introduction to Geographical Information Science*. John Wiley & Sons, 2006, pp. 31–51.

[32] S. Takahashi, Y. Takeshima, and I. Fujishiro, "Topological volume skeletonization and its application to transfer function design," *Graphical Models*, vol. 66, no. 1, pp. 24–49, 2004.

[33] S. Takahashi, I. Fujishiro, and Y. Takeshima, "Interval volume decomposer: a topological approach to volume traversal," in *Proc. SPIE*, 2005, pp. 103–114.

[34] S. Takahashi, T. Ikeda, Y. Shinagawa, T. L. Kunii, and M. Ueda, "Algorithms for extracting correct critical points and constructing topological graphs from discrete geographical elevation data," *Computer Graphics Forum*, vol. 14, no. 3, pp. 181–192, 1995.

[35] S. P. Tarasov and M. N. Vyalyi, "Construction of contour trees in 3d in o(n log n) steps," in *Proceedings of the fourteenth annual symposium on Computational geometry*, ser. SCG '98. New York, NY, USA: ACM, 1998, pp. 68–75.

[36] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. R. Schikore, "Contour trees and small seed sets for isosurface traversal," in *Proc. Symp. Comput. Geom.*, 1997, pp. 212–220.

[37] ——, "Contour trees and small seed sets for isosurface traversal," Department of Computer Science, Utrecht University, Tech. Rep. UU-CS-1998-25, 1998.

[38] G. H. Weber, S. E. Dillard, H. Carr, V. Pascucci, and B. Hamann, "Topology-controlled volume rendering," *IEEE Trans. Vis. Comput. Graph.*, vol. 13, no. 2, pp. 330–341, 2007.

[39] E. Zhang, K. Mischaikow, and G. Turk, "Feature-based surface parameterization and texture mapping," *ACM Trans. Graph.*, vol. 24, no. 1, pp. 1–27, 2005.

[40] J. Zhou and M. Takatsuka, "Automatic transfer function generation using contour tree controlled residue flow model and color harmonics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1481–1488, 2009.