A Parallel and Memory Efficient Algorithm for Constructing the Contour Tree

Aditya Acharya *
Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore

Vijay Natarajan†
Department of Computer Science and Automation
Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore

Abstract

The contour tree is a topological structure associated with a scalar function that tracks the connectivity of the evolving level sets of the function. It supports intuitive and interactive visual exploration and analysis of the scalar function. This paper describes a fast, parallel, and memory efficient algorithm for constructing the contour tree of a scalar function on shared memory systems. Comparisons with existing implementations show significant improvement in both the running time and the memory expended. The proposed algorithm is particularly suited for large datasets that do not fit in memory. For example, the contour tree for a scalar function defined on a 8.6 billion vertex domain ($2048 \times 2048 \times 2048$ volume data) can be efficiently constructed using less than 10GB of memory.

Index Terms: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling

1 Introduction

Scientific data obtained from simulations and measurement devices is often represented as a scalar function over a two, three, or higher dimensional domain. The contour tree tracks topology changes in level sets of a scalar function defined on a simply connected domain, and serves as an abstract representation of the data. It is obtained by mapping each connected component of a level set to a point, see Figure 1. The effectiveness and usefulness of this representation is well established in the literature. In this paper, we propose a parallel algorithm for fast and memory efficient construction of the contour tree to facilitate its application to large data sizes.

1.1 Motivation

The contour tree is one of the most extensively studied and developed topological structures in the visualization literature. In particular, it has been widely applied in the context of volume visualization – for transfer function design [10, 15, 30, 38, 41], efficient computation of isosurfaces [35], and for effective and flexible exploration of isosurfaces [5]. Following its successful application to volume data visualization, several recent efforts have demonstrated the use of the contour tree for visualization of high dimensional data [16, 23, 24]. The power of this abstract representation is clearly demonstrated in its application to volume data analysis – feature extraction and tracking [3, 12, 37], symmetry and similarity detection [27, 32], comparative visualization [28], and volume segmentation [29]. The contour tree has also been applied to solve problems in computer graphics and computer vision such as surface segmentation [17], parametrization [40], model repair [39, 33], and skeletonization [22, 34]. The above list of applications motivates the development of fast algorithms for computing the contour tree.

*e-mail: adiarya1a1@gmail.com
†e-mail:vijayn@csa.iisc.ernet.in

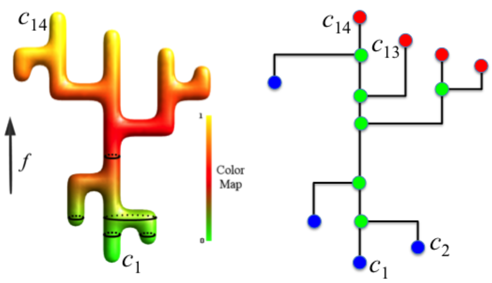

Figure 1: A height function and the corresponding contour tree. The i^{th} critical point in increasing order of function value is labeled c_i.

The rapid growth in compute power has facilitated the generation of higher fidelity simulation data and higher resolution imaging data, which in turn has resulted in a massive increase in the size of the datasets. Topology-based methods were developed with the aim of enabling analysis and visualization of these large datasets by providing abstract representations of the key features in the data. However, the construction of the topological structures is now increasingly becoming a bottleneck. This necessitates the development of efficient algorithms that can additionally handle large data sizes. Further, the auxiliary memory required by these algorithms is often proportional to the size of the input. So, it is imperative to ensure that the algorithm has a reasonably low memory footprint. A related development is that of multicore and manycore CPUs becoming ubiquitous. It is highly desirable that new algorithms for computing the contour tree leverage their power. We address the above-mentioned challenges to design a parallel and memory efficient algorithm for computing the contour tree of a scalar function defined on a volume grid.

1.2 Related Work

The contour tree was first formulated in its current form by de Berg and van Kreveld [9] to answer elevation queries in GIS applications. They describe an algorithm that employs a divide and conquer strategy to compute the contour tree of a scalar function defined on a two-dimensional domain in $O(n \log n)$ time, where n is the number of triangles in the input. van Kreveld et al. [35] developed an algorithm that maintained evolving level sets in order to compute the contour tree in $O(n \log n)$ for two-dimensional input, and in $O(n^2)$ time for three-dimensional input. Tarasov and Vyalyi [31] described an improved algorithm that computes the contour tree of a three-dimensional scalar function in $O(n \log n)$ time. 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 two sweeps.

Carr et al. [4] simplified this approach to develop an algorithm that is arguably the most elegant and widely used algorithm for computing the contour tree. This algorithm computes a join tree and a split tree in two sweeps over the input by tracking the evolution of sub-level and super-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, n is the number of tetrahedra and α is the inverse Ackermann function. Chiang et al. [6] proposed an output sensitive approach that first finds all component critical points representing nodes in the contour tree by querying the local neighborhood. Monotone paths are constructed from these critical points, and later merged to construct the join and split trees containing only the component-critical points. Their 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. [36] 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. The contour tree may be considered as a special case of a Reeb graph [26] when restricted to simply connected domains. The Reeb graph may contain cycles and the above algorithms typically do not apply. Algorithms for computing the Reeb graph can, by definition, be no faster than those for computing the contour tree [2, 11].

Pascucci and Cole-McLaughlin [25] proposed the first known parallel algorithm that computes the contour tree of a piecewise trilinear function defined on a three dimensional structured mesh. The sequential version for a 3D structured grid has a time complexity of $O(n + t \log n)$, where n is the number of vertices and t is the number of critical points. The volume is recursively subdivided into two halves of roughly equal number of vertices, with the common boundary (the separator) consisting of $O(n^{\frac{2}{3}})$ vertices and edges. The algorithm essentially computes the contf tree for each voxel and merges them to eventually obtain the contour tree over the entire domain. While the actual running times are unavailable, the authors report that the algorithm scales well and exhibits linear speedup with increasing number of processors. Although this algorithm is well suited for coarse grained parallelism, computing the contour tree for each voxel individually may result in huge overheads.

Maadasamy et al. [19] describe an output sensitive, work efficient, shared memory, parallel implementation that computes monotone paths similar to Chiang et al. [6] but ensure that the computation is efficient for parallel architectures. They compute the monotone paths in parallel and arbitrary order rather than sequentially to compute the contour tree. Although the method scales well for small unstructured grids, the speedup decreases significantly for large structured grids as the number of available processors increase. Moreover, the required memory to compute the contour tree is huge owing to the additional auxiliary structures needed to compute the join and split tree. We describe an algorithm for shared memory architectures that addresses these weaknesses with a focus on structured grids.

A recent approach towards computing the join and split tree in parallel by Morozov and Weber [20, 21] proceeds by constructing the local join tree or split tree on each processor and merging them across sub-domains. However, each processor stores only the local tree corresponding to the sub-domain and a small subset of the global join / split tree. This method is best suitable for distributed memory systems with the final join and split trees being distributed across nodes. The final computation of the contour tree and even the applications requiring a contour tree have to be significantly modified to work in a distributed memory paradigm. Our proposed shared memory algorithm may be used to compute the local tree within a single node in a multiprocessor environment and hence improve the running time of this distributed algorithm.

A paper simultaneously communicated by Langde et al. [18] describes a distributed algorithm for computing join and split trees. Local trees are built for each sub-domain together with a boundary tree for each sub-domain. The boundary tree aids in reducing communication cost when the local trees are stitched together. A pruning step similar to the one that we propose helps reduce the size of the tree and hence reduces memory requirement. Again, our shared memory algorithm may be used within a single node in conjunction with such a distributed algorithm.

1.3 Contributions

In this paper, we describe a fast and memory efficient parallel algorithm for computing the contour tree of a piecewise trilinear function defined on a large structured grid. The algorithm employs a novel hybrid approach by tracing monotone paths from critical points to compute local join and split trees within different sub-domains, and stitching these trees together using a sequence of union-find operations. While the two approaches have been independently proposed earlier, the hybrid approach is crucial in determining the scalability of the parallel algorithm and its memory efficiency. The algorithm is output sensitive, which essentially means that it computes small contour trees faster and requires more time only for the larger ones. A well engineered pruning step and stitching procedure further reduce the memory footprint of the algorithm and improves scalability, respectively.

Experimental results show significant improvements in terms of time and memory over the existing parallel algorithms. The contour tree for a dataset containing 8.6 billion vertices ($2048 \times 2048 \times 2048$ volume) can be constructed within 3 minutes in a 64-core shared memory environment. In an 8-core environment, the algorithm uses no more than 10GB of memory and computes the tree in approximately 14 minutes.

2 Background

In this section, we introduce the necessary definitions of Morse functions and level set topology [13] that are required to define the contour tree and to describe its construction in the following section. Scalar fields are typically available as a sample together with a mesh representation of the domain. The domain is often represented as a structured grid in many applications.

Let S be a structured grid and f denote a scalar function defined on the domain D represented by S. The function f is available as a sample at vertices of S and is extended via trilinear interpolation to the interior of the grid cells. A *level set* $f^{-1}(a)$ is the set of all points in D having function value equal to a. A *sub-level set* is the set $f^{-1}(-\infty, a]$ consisting of points having function value less than or equal to a. Similarly, $f^{-1}[a, \infty)$ is called a *super-level set*. As we sweep across a range of function values, the connectivity / topology of the corresponding level sets change. Points at which the topology of the level sets change during this evolution are the *critical points* of the function. Points that are not critical are called *regular points*.

A connected component of a level set is called a *contour*. Given two points $x, y \in D$, we say $x \sim y$ iff they belong to the same contour. The *contour tree* is defined as the quotient space $D/\sim$ that glues all points that are equivalent under the binary relation $\sim$. In other words, every contour is represented by a point in the contour tree. Figure 2 shows multiple level sets extracted from a synthetic scalar function defined on a structured grid and Figure 3 shows the corresponding contour tree. Each contour maps to a different arc in the contour tree. The contour tree expresses the evolution of the connected components of the level sets as a graph whose nodes correspond to critical points of the function. A new contour appears at *minimum* (blue), contours merge or split at a *saddle* (green), and a contour disappears at a *maximum* (red). The *join tree* tracks the evolution of sub-level sets and the *split tree* tracks the evolution of super-level sets.

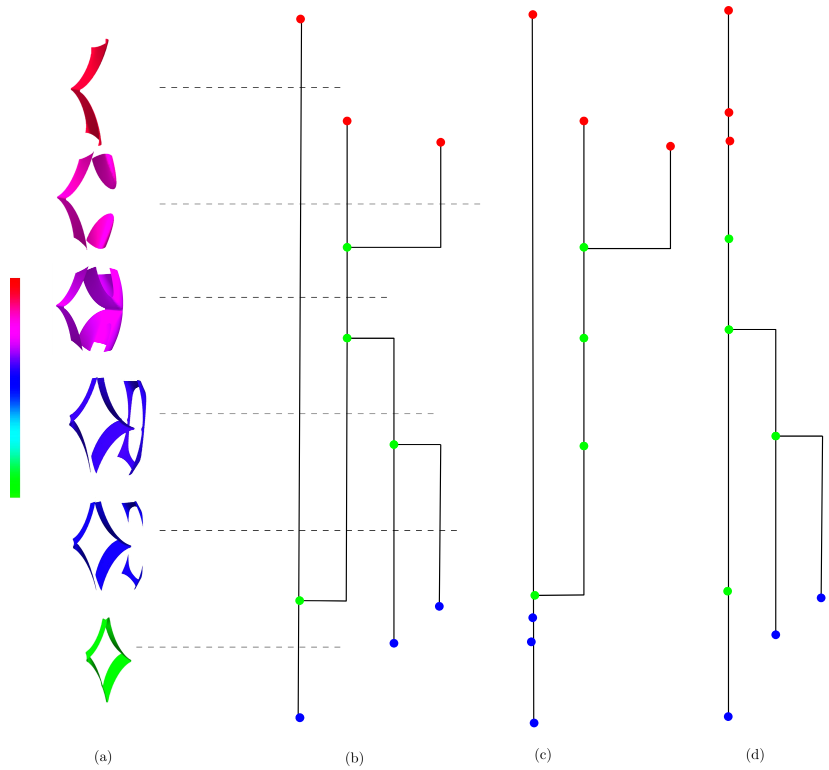

Figure 3: The contour tree for the analytic function shown in Figure 2. (a) Level sets at different function values (b) The contour tree tracks the evolution of connected components of the level sets. (c) The split tree tracks connected components of the super-level sets. (d) The join tree tracks connectivity of sub-level sets.

Figure 2: An analytic function sampled on a structured grid and a visualization that shows multiple level sets of the function. Each level set consists of one or more connected components.

3 Algorithm

We now describe our parallel algorithm for computing the contour tree of a scalar field defined on a volume grid. We assume that the function values at the vertices are unique. This may be achieved via a simulated perturbation using the index of the memory location corresponding to a vertex. Key steps in the algorithms are listed below and described in detail subsequently.

1. Split the domain into sub-domains of appropriate size and assign the sub-domains to different processors.
2. Identify the critical points within each sub-domain.
3. Compute the local join tree split tree for each sub-domain.
4. Prune the representation of the local join and split tree computed in the previous step by identifying and removing nodes that do not correspond to a change in the number of contours.
5. Stitch the local join and split trees across neighboring sub-domains hierarchically to construct the global join and split tree for the entire domain.
6. Merge the global join and split tree to construct the global contour tree.

3.1 Splitting the domain

We decompose the domain into sub-domains following an octree based subdivision. The sub-grid representing a sub-domain is subdivided into two parts at every iteration along the largest dimension. The two resulting sub-grids share a common plane. For example, given a grid S with dimensions (dim_x, dim_y, dim_z), where $dim_x \geq dim_y \geq dim_z$, it is sub-divided in the first iteration into two sub-grids S_1 and S_2 with dimensions $(\lceil dim_x/2 \rceil, dim_y, dim_z)$ and $(dim_x - \lceil dim_x/2 \rceil + 1, dim_y, dim_z)$. The sub-grids share a plane parallel to the YZ plane. S_1 and S_2 are further subdivided and after i iterations, S is subdivided into 2^i sub-grids, which are processed in parallel by different processors.

3.2 Identifying critical points

We classify points as critical or regular based on local behavior of the scalar field. Edelsbrunner et al. [14] consider piecewise linear functions and provide a combinatorial characterization of its critical points, which are always located at the mesh vertices. While maxima and minima of piecewise trilinear functions are always located at vertices, saddles may also be located within a face of a cell or within its body. The presence and number of these *face saddles* and *body saddles* within a cell can be determined using a combinatorial method [25]. Subsequent steps of the algorithm require the list of vertices together with locations of critical points and edges connecting them to their neighborhood. We insert the face and body

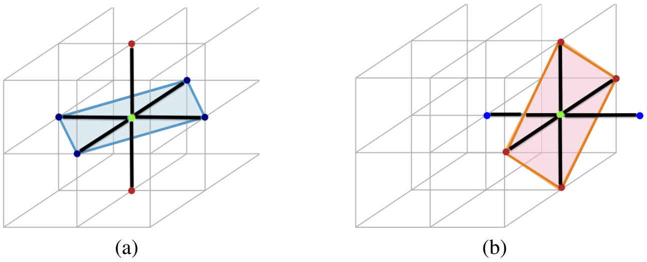

Figure 4: Points classified as saddles in a sub-domain. (a) The green point represents an interior saddle with two upper link components (red) and one lower link component (blue). The lower link component lies on a plane normal to one of the axes and the two upper link component on either side. (b) A point on the boundary with one upper link component (red) and one lower link component (blue) within the sub-domain. It is classified as critical because it is a minimum of the function restricted to the boundary plane.

Algorithm 1: ConstructJoinTree ( )

Input: Set of critical points $C = \cup c_i$
Input: Mesh M
Output: Join tree T_J
Initialize the node set of T_J to C
$UF \leftarrow$ empty union find data structure
Sort C in ascending order
for i $\leftarrow$ 1 to $|C|$ **do**
    Mark vertex c_i as visited
    NewSet (c_i, UF)
    for each Lower Link component L_j of c_i **do**
        Let R_j be a vertex in L_j
        Follow a descending path P from R_j until a visited vertex w is hit
        Insert pointers from every vertex in P to c_i
        Let c_r be the vertex w is pointing to
        $c'_r \leftarrow$ Find(c_r, UF)
        if $c'_r \neq c_i$ **then**
            Add edge (c_i, c'_r) to T_J
            Union(c'_r, c_i, UF)
        end if
    end for
end for
return Join tree T_J

saddles into the vertex list and include edges to vertices in the face or the cell.

The *link* of a vertex in the original structured grid is the triangulation of its neighboring six vertices that consists of a triangle within each of the eight cells incident on the vertex. The link of a face / body saddle is the set of its neighboring mesh vertices together with the induced edges and triangles. Link vertices with lower function values together with the induced edges and triangles form the *lower link*. Similarly, link vertices with higher function values together with the induced edges and triangles form the *upper link*. A vertex is *regular* if its upper link and one lower link have exactly one component. A vertex is a *maximum* if its upper link is empty and a *minimum* if its lower link is empty. All other points are classified as *saddle*.

A vertex in the input structured grid is a saddle only when its lower (upper) link lies on a plane normal to one of the axes and its upper (lower) link consists of two isolated vertices. Figure 4(a) show such a saddle and the separating plane. The critical points are identified within each sub-domain. While processing boundary points, their neighborhood within the entire domain may have to be considered to ensure correct classification. However, this requires access to neighboring sub-domains. We avoid the associated communication or memory costs by reporting boundary extrema as potential critical points.

A critical point may not be classified correctly if the link is restricted to the sub-domain only when the separating plane lies on the boundary. Such a boundary point is an extremum on the boundary plane. We insert all such boundary extrema to the list of critical points. Some of these points may not correspond to nodes of final contour tree and are pruned away. Figure 4(b) represents one such boundary extremum.

3.3 Computing local join and split trees

We compute the local join tree by tracing monotone paths from critical points and hence identifying connected components of sub-level sets. This approach is similar to the one proposed by Chiang et al. [6]. However, we optimize the method for structured grids and further apply it only to construct the local join tree. Algorithm 1 describes the procedure to compute the local join tree.

The list of critical points available from the previous step is first sorted according to their function values. The critical points form the ground set for the procedure and are processed in increasing order of their function values. The join tree corresponding to $f^{-1}(-\infty, f(c_i)]$ is constructed when c_i is processed. Therefore, after processing the critical point with highest function value, the local join tree corresponding to the sub-domain is fully constructed. A union-find data structure is used to maintain connected components of sub-level sets during the construction. The highest valued vertex is chosen as its representative. Descending paths from the critical point c_i are constructed from each lower link component of c_i until a previously visited vertex w is encountered. All vertices on the paths are provided a pointer to c_i. Next the pointer from w is followed to find the critical point c_r that already had a descending path to w. Finally, we compute the union of the sets containing c_i and c_r and insert an edge from c_i to the representative of the component containing c_r.

For optimal performance and low memory utilization, we store the join tree as a parent array. For example if Jt is the array corresponding to the join tree then $Jt[i]$ represents the parent of the i^{th} critical point. We also store the corresponding children array to enable faster access. This array is particularly useful in the subsequent steps of the algorithm. The construction of the local split tree is analogous to the construction of the local join tree, and proceeds by processing the critical points in decreasing order of their function values and constructing ascending paths.

3.4 Pruning local join and split trees

In this step, we prune the local join tree and split tree for each sub-domain in parallel. Pruning consists of identifying and removing nodes that do not represent a change in the number of connected components. If c_i is a degree-2 node in the local join tree then the number of connected components of the sub-level set does not change when it crosses $f(c_i)$. Similarly, a degree-2 node in the split tree contributes no additional information regarding the number of super-level set components. Hence, a node that neither corresponds to a join vertex nor to a split vertex can be safely pruned away. We note that such nodes might represent other topological changes such as a change in genus. Algorithm 2 describes the procedure to prune the local join and split trees.

We do not prune the join and split trees independently. In other words, we remove only those nodes that are degree-2 in both the local join and split trees. We retain other degree-2 nodes because, in the final step when two join and split trees are merged to construct the contour tree, we require the location of such nodes from

Algorithm 2: PruneTrees ( )

Input: List of critical points C, Join Tree T_J and Split tree T_S
Output: Pruned trees t_j and t_s
for every vertex $c_i \in C$ **do**
  if c_i is a degree-2 node in both T_J and T_S and does not lie on the boundary **then**
    Remove c_i from C
  end if
end for
$\{$*Prune Join Tree*$\}$
for every vertex $c_i \in C$ **do**
  if c_i is the not the root of T_J **then**
    $p_i \leftarrow c_i.JoinParent$
    while $p_i \notin C$ **do**
      $p_i \leftarrow p_i.JoinParent$
    end while
    Add edge (p_i, c_i) to t_j
  end if
end for
Delete T_J
$\{$*Prune Split Tree*$\}$
for every vertex $c_i \in C$ **do**
  if c_i is not the root of T_S **then**
    $p_i \leftarrow c_i.SplitParent$
    while $p_i \notin C$ **do**
      $p_i \leftarrow p_i.SplitParent$
    end while
    Add edge (p_i, c_i) to t_s
  end if
end for
Delete T_S
return t_j and t_s

Algorithm 3: StitchJoinTrees (t_{j1}, t_{j2})

Input: Sorted list of nodes T_1 and T_2
Output: Join tree t_j
Initialize $t_j \leftarrow t_{j1} \cup t_{j2}$
$UF \leftarrow$ empty union find data structure
$T \leftarrow Merge(T_1, T_2)$
for i $\leftarrow 1$ to $|T| - 1$ **do**
  if v_i and v_{i+1} are boundary duplicate points **then**
    NewSet(v_i, UF)
    NewSet(v_{i+1}, UF)
    Union(v_i, v_{i+1}, UF) making v_{i+1} as the head
    v_i.JoinParent $\leftarrow v_{i+1}$
    Add v_i to v_{i+1}.JoinChildrenList
  end if
  for each child c_j of v_i **do**
    if c_j is present in UF **then**
      if v_i is not present in UF **then**
        NewSet(v_i,UF)
      end if
      Delete c_j from v_i.JoinChildrenList
      $c' \leftarrow$ FIND(c_j,UF)
      if $v_i \neq c'$ **then**
        c'.JoinParent $\leftarrow v_i$
        Add c' to v_i.JoinChildrenList
        Union(c',v_i,UF) ensuring v_i as the head
      end if
    end if
  end for
end for
return Join tree t_j

both trees. We also preserve the boundary points because they are required for correct stitching of the trees across the sub-domains.

This pruning step contributes to the huge memory savings achieved by our algorithm. In our experiments, we observe that the size of the local tree reduces by a factor of 5-10, depending upon the dataset, after pruning. If the domain is divided into d sub-domains and processed using p processors, then the memory requirement of our algorithm is p/d times the maximum memory utilized by an algorithm that does not partition the domain. We choose d such that it is significantly larger than p and hence require only a fraction of the maximum memory utilized by other algorithms. The memory required for subsequent steps of the algorithm further reduces due to the pruning.

3.5 Stitching local join and split trees

Local join and split trees of sub-domains that share a common boundary are stitched together in parallel. A union-find data structure is again used to maintain connectivity. However, only the portions of the trees affected by the boundary nodes are processed and updated. This crucially determines the run time efficiency of the algorithm. Algorithm 3 describes the stitching procedure for the join tree. Split trees are stitched together using a similar procedure.

Let t_{j1} and t_{j2} denote the local join tree of the adjoining sub-domains D_1 and D_2. Let T_1 and T_2 be the list of nodes in t_{j1} and t_{j2}. The nodes in T_1 and T_2 are already sorted. We merge these sorted lists in linear time to obtain a sorted list of nodes from $T_1 \cup T_2$. Duplicate nodes are retained to avoid reorganizing the data structures. The duplicate nodes would appear next to each other in the sorted list. We insert an edge between these duplicate nodes essentially creating a new mesh M_{12} whose vertex set equals the nodes in $T_1 \cup T_2$ and whose edge sets are the union of the arc sets of t_{j1} and t_{j2} together with the newly inserted edges.

The join tree of the scalar function restricted to $D_1 \cup D_2$ is computed as the join tree of M_{12} by maintaining a union-find data structure UF. First, the nodes and arc sets of t_{j1} and t_{j2} are merged. The vertices of M_{12} are processed in sorted order. The first set is created in UF when the first boundary node is processed. Subsequently, a new set is created only when another boundary node is processed or when a child of the node being processed belongs to UF. Union operations are triggered in both cases. Note that the several nodes of t_{j1} and t_{j2} are not inserted into UF because they remain unaffected after stitching. We observe in our experiments that the time required for stitching is indeed roughly proportional to the number of boundary nodes on the sub-domains.

Trees across adjoining sub-domains are stitched hierarchically traveling up the domain decomposition octree. Independent stitching processes are scheduled in parallel. As the stitching proceeds to move towards the root of the octree, the number of parallel jobs naturally continues to decrease. The stitching process is top heavy and does not scale well with the number of processors. In the final iteration, we merge the trees across two halves of the domain resulting in the global join and split trees.

Split trees can be processed similarly and stitched across the sub-domains. In fact, when processors are available, we schedule the stitching of the split trees in parallel with the stitching of the join trees. We do not prune the duplicate vertices at the end of the step and defer it to the final step instead. In practice, the final pruning reduces the number of points in split and join tree by only about 5%. It is expensive to scan the entire tree to find degree-2 nodes and therefore we avoid pruning the trees after every stitch operation.

3.6 Merging global join and split trees

The final contour tree is constructed from the global join and split tree using a procedure similar to that described by Carr et al. [4].

Algorithm 4: MergeTrees ( )

```
Input: Global Join tree T_j and Split tree T_s
Output: Contour tree T_c
G = Set of leaves in T_j and T_s
while G ≠ φ do
    if c_i is a leaf in T_j or T_s then
        Process c_i and remove it from G
        T = tree in which c_i is a leaf
        while c_i ≠ T.root and c_i is not processed do
            n_i = c_i
            c_i = parent vertex of c_i in T
        end while
        Remove n_i from T and G
        Add arc(n_i,c_i) to T_c
        if c_i is either a leaf in T_j or T_s then
            Add c_i to G
        end if
    end if
end while
```

We present it here in a form that is amenable to a parallel implementation, see Algorithm 4. Each iteration of 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. The procedure terminates when all arcs of the join and split trees are processed. The running time of the sequential version is linear in the number of critical points. The set of leaves are removed in parallel in our implementation.

3.7 Analysis

We assume there are v vertices in the structured grid, t critical points in the domain, and d number of sub-domains. Let t_i, $i = 1..d$, denote the number of critical points present in the i^{th} sub-domain. Let b_i represent the number of boundary nodes classified as critical in the i^{th} sub-domain.

Locating the critical points takes $O(v)$ time as it takes constant amount of time for every vertex to find the number of upper link and lower link components. Chiang et al. [7] show it takes $O(v + t \log t)$ time for constructing the join and split tree where t is the number of critical points. Since the maximum number of critical points processed within each sub-domain in Step 2 and Step 3 is $t_i + b_i$, these two steps take $O(v/d + (t_i + b_i)\log(t_i + b_i))$ time. Pruning the local join and split trees again takes $O(t_i + b_i)$ time. Stitching sub-domains D_i and D_j requires a maximum of $\left(t_i + t_j + b_i + b_j\right)$ unions and find operations, which can be performed in $\left(t_i + t_j + b_i + b_j\right) \alpha \left(t_i + t_j + b_i + b_j\right)$ time [8]. This is a very conservative estimate since the majority of the nodes remain unaffected and hence are not processed by the union and find operations. The merging of the two sorted lists takes $O\left(t_i + t_j + b_i + b_j\right)$ time. The final cleanup and the merging of the global join and split tree to form the contour tree takes O(t) time.

The d sub-domains are stitched together in $\log d$ iterations . In the worst case, within each iteration, we process $z = t + 3d^{\frac{1}{3}} \cdot v^{\frac{2}{3}}$ points in $O(z\alpha(z))$ time. If the algorithm is executed sequentially, the net run time complexity is $O(v + \sum_{i=1}^{d}(t_i + b_i)\log(t_i + b_i) + \log d \cdot z\alpha(z)) = O(v + z\log z + \log d \cdot z\alpha(z))$. If d is much smaller than v and t, the sequential running time is $O\left(v + \left(t + v^{\frac{2}{3}}\right)\log\left(t + v^{\frac{2}{3}}\right)\right)$.

4 Experimental Results

We evaluate our implementation, called DivCT, on a shared memory system with 64 cores. All experiments were conducted on an AMD Opteron 6274 processor with 64 cores running at 2.2GHz. Data used for the experiments is from `http://volvis.org` and available on a structured grid. We first report run times for the sequential version of the algorithm where the entire data is processed by a single processor. Next, we report run times for increasing number of processors. Finally, we compare the running times with an existing parallel algorithm ParallelCT[19], which computes the contour tree in a shared memory system without partitioning the domain.

4.1 Single core environment

On a single core environment DivCT clearly performs better than existing implementations for data that fits in memory. We compare running times with libtourtre [1], a publicly available and widely used serial implementation of the algorithm due to Carr et al. libtourtre offers an implementation for structured grids. We also report running times for ParallelCT which also contains an implementation for structured grids, see Table 1. Both DivCT and ParallelCT are faster than libtourtre for structured grids. This is expected because both DivCT and ParallelCT are output sensitive. We also observe that running times of DivCT are comparable or better than ParallelCT. This improvement over ParallelCT may be attributed to the additional computations in ParallelCT for constructing the auxiliary data structures.

4.2 Multi-core environment

For processing a dataset on p cores, we divide our domain into at least $8p$ sub-domains. If the sub-domains are still large and do not fit in memory, they are further partitioned. Ensuring a minimum of $8p$ sub-domains results in a reasonable load balance among the cores while computing the local join and split trees. In practice, we observe that this step scales almost linearly with increasing number of processors with an additional, but small, expense of handling greater number of boundary vertices. We observe in our experiments that the total number of boundary vertices including the duplicate points that are misclassified as critical points is roughly equal to only 5% of the final size of the join and split trees. Therefore, the increase in number of sub-domains does not adversely affect the computation time. Note that the domain is not partitioned for the sequential execution.

Scaling. Figure 5 shows the scaling behavior of DivCT with respect to increasing number of processors on large data sets. The graph plots indicate that we achieve close to the ideal speedup (blue). The exact speedup factors are listed in Table 2. Graph plots showing the scaling behavior of the key steps, local join / split tree computation and the stitching step, are included in the supplementary material. As expected, the local join and split tree computation for sub-domains scales linearly and very close to the ideal speedup. This is primarily responsible for the overall near-linear speedup. On the other hand, the stitching step scales poorly. It is the primary contributor to the deviation from the ideal linear speedup seen in Figure 5. For experiments on 64 cores, the time taken for stitching together with the final merge to compute the contour tree is comparable to the time taken to compute the local join and split trees for all the sub-domains.

Comparison. We compare the performance of DivCT with ParallelCT in Table 2. We observe significant improvements both in terms of running time and speedup over ParallelCT, the best known parallel implementation for constructing the contour tree. We observe a saturation in ParallelCT with increasing number of processors whereas DivCT exhibits good scaling. We also observe significant improvements in terms of memory consumption. For example, ParallelCT requires approximately 12GB of memory to compute the contour tree for the Vertebra ($512 \times 512 \times 512$) dataset. However, DivCT requires only one-fifth as much memory because the data is partitioned into sub-domains. In the case of larger data sizes, it is infeasible to use ParallelCT

Model	#Vertices	LIBTOURTRE	PARALLELCT	DIVCT
Aneurysm	$256 \times 256 \times 256$	15.2	9.4	7.7
Bonsai	$256 \times 256 \times 256$	21.9	18.9	16.1
Foot	$256 \times 256 \times 256$	31.5	19.8	17.2

Table 1: Time taken (in seconds) to compute the contour tree on a single core. DIVCT outperforms both LIBTOURTRE and PARALLELCT.

Model	#Vertices	PARALLELCT			DIVCT		
		1 core	8 cores	64 cores	1 core	8 cores	64 cores
Vertebra	$512 \times 512 \times 512$	91.4	19.8 (4.6×)	8.8 (10.4×)	76.7	12.1 (6.3×)	2.7 (28.4×)
ColonPhantom	$512 \times 512 \times 442$	182.6	52.9 (3.5×)	15.4 (11.9×)	156.5	25.2 (6.2×)	5.5 (27.5×)
Vertebra1024	$1024 \times 1024 \times 1024$	-	-	-	532.5	74.0 (7.2×)	14.3 (37.2×)
ColonPhantom1024	$1024 \times 1024 \times 884$	-	-	-	1142.5	156.4 (7.3×)	32.5 (35.2×)
Vertebra2048	$2048 \times 2048 \times 2048$	-	-	-	6513.8	868.5 (7.5 ×)	173.7 (37.5×)
ColonPhantom2048	$2048 \times 2048 \times 2048$				12890.7	2113.2 (6.1×)	493.9 (26.1×)

Table 2: Time taken (in seconds) for computing the contour tree in a multicore system by PARALLELCT and DIVCT. The speedup factor is shown within parenthesis. DIVCT exhibits better scaling and is also faster in terms of total running time. PARALLELCT is unable to process larger datasets because it requires more memory than available.

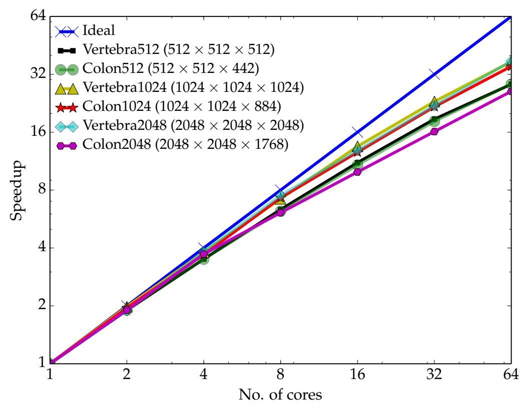

Figure 5: Speedup for large datasets with increasing number of cores. DIVCT exhibits close to ideal scaling behavior. The exact speedup factors are available in Table 2.

to compute the contour tree. For example, it requires more than 60GB of memory for a $1024 \times 1024 \times 1024$ dataset. DIVCT consumes at most 11GB of memory for the same dataset.

DIVCT may be used to improve the running time of the distributed contour tree algorithm [21]. In particular, DIVCT may be employed within a single node in a multi-processor environment and hence supplement the benefits of the distributed algorithm. For example, the distributed contour tree algorithm requires approximately 21 seconds to compute the contour tree for the Vertebra dataset on 64 cores. In contrast, DIVCT requires only 2.7 seconds on 64 cores to compute the contour tree. However, it is applicable only within a node of the cluster as compared to the distributed algorithm, which is shown to scale up to 256 processors. So, we propose the application of DIVCT within a node for computing the local tree stored at the node followed by the distributed algorithm across nodes in order to achieve improved performance.

Memory consumption. The maximum memory required by DIVCT to construct the trees can be reduced by further subdividing the sub-domains. In fact, the minimum available memory required by DIVCT is comparable to the size of the final contour tree. Assuming that the scalar values are stored in single precision, DIVCT requires $112 \cdot t$ bytes. The final contour tree can be computed for the Vertebra1024 ($1024 \times 1024 \times 1024$) dataset using 2.5GB of memory on an 8-core machine by partitioning it into 512 sub-domains of size $128 \times 128 \times 128$ each. For datasets even larger in size, say a $2048 \times 2048 \times 2048$ containing about 8.6 billion points, we similarly divide the domain into 512 sub-domains of size $256 \times 256 \times 256$. We compute the final contour tree in less than 14 minutes consuming roughly 10GB of memory on 8 cores. With 64 cores, the computation time drops to approximately 3 minutes. The number of nodes in the pruned final contour tree is listed for all datasets in a table in the supplementary material.

Discussion. Contour tree based methods for visualization, analysis, and interactive exploration of data typically compute the contour tree in a preprocessing step. Following this computation, the methods support fast feature extraction, measurement, comparative and visual analysis, and interaction often with real-time response. It is important to ensure that the preprocessing step does not become a performance bottleneck. For example, given the contour tree, a level set component can be computed for the Vertebra dataset in approximately 5 seconds using 64 cores [21]. Topology controlled transfer function may be automatically designed within 1-2 seconds for data sizes up to $400 \times 400 \times 400$ [41]. Repeating patterns within a scalar field can be computed by identifying similar subtrees of the contour tree within 1 second for data sizes up to $500 \times 500 \times 500$ [32]. Table 2 shows that DIVCT can compute the contour tree for the Vertebra dataset within 3 seconds, three times faster than PARALLELCT. This improvement is significant for all three application scenarios listed above.

5 CONCLUSIONS

We have presented a simple and memory efficient algorithm for parallel construction of the contour tree of a scalar function defined on a 3D structured grid. We compute the contour tree for extremely large datasets of size up to $2048 \times 2048 \times 2048$ that do not fit in memory. The near-linear speedup obtained for various datasets indicates that our implementation scales well with increasing number of processors. We also report significant improvements in memory usage over an existing shared memory based parallel algorithm for computing the contour tree. In future, it would be interesting to see if we can utilize GPUs or a CPU-GPU hybrid environment for faster computation of the contour tree.

ACKNOWLEDGEMENTS

This work was partially supported by the Department of Science and Technology, India, under Grant SR/S3/EECE/0086/2012 and by the Robert Bosch Centre for Cyber Physical Systems, Indian Institute of Science. Vijay Natarajan was supported by a fellowship for experienced researchers from the Alexander von Humboldt Foundation. We thank Dilip M. Thomas and Vidya Narayanan for feedback on an early draft of the paper and members of the Visualization and Graphics Lab at IISc for various discussions.

References

[1] libtourtre: A contour tree library. `http://graphics.cs.ucdavis.edu/~sdillard/libtourtre/doc/html/`.

[2] S. Biasotti, D. Giorgi, M. Spagnuolo, and B. Falcidieno. Reeb graphs for shape analysis and applications. *Theor. Comput. Sci.*, 392:5–22, February 2008.

[3] P.-T. Bremer, G. Weber, J. Tierny, V. Pascucci, M. Day, and J. Bell. Interactive exploration and analysis of large-scale simulations using topology-based data segmentation. *IEEE Trans. Visualization and Computer Graphics*, 17(9):1307–1324, 2011.

[4] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. *Comput. Geom. Theory Appl.*, 24(2):75–94, 2003.

[5] H. Carr, J. Snoeyink, and M. van de Panne. Flexible isosurfaces: Simplifying and displaying scalar topology using the contour tree. *Computational Geometry*, 43(1):42–58, 2010.

[6] 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.*, 30(2):165–195, 2005.

[7] Y.-J. Chiang, T. Lenz, X. Lu, and G. Rote. Simple and optimal output-sensitive construction of contour trees using monotone paths. *Computational Geometry*, 30(2):165–195, 2005.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 2001.

[9] M. de Berg and M. J. van Kreveld. Trekking in the alps without freezing or getting tired. *Algorithmica*, 18(3):306–323, 1997.

[10] H. Doraiswamy and V. Natarajan. Output-sensitive construction of Reeb graphs. *IEEE Trans. Visualization and Computer Graphics*, 18:146–159, 2012.

[11] H. Doraiswamy and V. Natarajan. Computing reeb graphs as a union of contour trees. *IEEE Trans. Visualization and Computer Graphics*, 19(2):249–262, 2013.

[12] H. Doraiswamy, V. Natarajan, and R. S. Nanjundiah. An exploration framework to identify and track movement of cloud systems. *IEEE Trans. Visualization and Computer Graphics*, 19(12):2896–2905, 2013.

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

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

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

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

[17] F. Hétroy and D. Attali. Topological quadrangulations of closed triangulated surfaces using the Reeb graph. *Graph. Models*, 65(1-3):131–148, 2003.

[18] A. G. Landge, V. Pascucci, A. Gyulassy, J. C. Bennett, H. Kolla, J. Chen, and P.-T. Bremer. In-situ feature extraction of large scale combustion simulations using segmented merge trees. In *Proc. ACM/IEEE Conf. on Supercomputing (SC14)*, volume 14, 2014.

[19] S. Maadasamy, H. Doraiswamy, and V. Natarajan. A hybrid parallel algorithm for computing and tracking level set topology. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–10. IEEE, 2012.

[20] D. Morozov and G. Weber. Distributed merge trees. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 93–102. ACM, 2013.

[21] D. Morozov and G. H. Weber. Distributed contour trees. In *Topological Methods in Data Analysis and Visualization III*, pages 89–102. Springer, 2014.

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

[23] P. Oesterling, C. Heine, H. Jänicke, G. Scheuermann, and G. Heyer. Visualization of high dimensional point clouds using their density distribution's topology. *IEEE Trans. Visualization and Computer Graphics*, 99(PrePrints), 2011.

[24] P. Oesterling, C. Heine, G. H. Weber, and G. Scheuermann. Visualizing nd point clouds as topological landscape profiles to guide local data analysis. *IEEE Trans. Visualization and Computer Graphics*, 19(3):514–526, 2013.

[25] V. Pascucci and K. Cole-McLaughlin. Parallel computation of the topology of level sets. *Algorithmica*, 38(1):249–268, 2003.

[26] G. Reeb. Sur les points singuliers d'une forme de pfaff complètement intégrable ou d'une fonction numérique. *Comptes Rendus de L'Académie ses Séances, Paris*, 222:847–849, 1946.

[27] H. Saikia, H.-P. Seidel, and T. Weinkauf. Extended branch decomposition graphs: Structural comparison of scalar data. *Computer Graphics Forum (Proc. EuroVis)*, 33(3):41–50, June 2014.

[28] D. Schneider, A. Wiebel, H. Carr, M. Hlawitschka, and G. Scheuermann. Interactive comparison of scalar fields based on largest contours with applications to flow visualization. *IEEE Trans. Visualization and Computer Graphics*, 14(6):1475–1482, 2008.

[29] S. Takahashi, I. Fujishiro, and Y. Takeshima. Interval volume decomposer: a topological approach to volume traversal. In *Proc. SPIE*, pages 103–114, 2005.

[30] S. Takahashi, Y. Takeshima, and I. Fujishiro. Topological volume skeletonization and its application to transfer function design. *Graphical Models*, 66(1):24–49, 2004.

[31] 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*, SCG '98, pages 68–75, New York, NY, USA, 1998. ACM.

[32] D. M. Thomas and V. Natarajan. Symmetry in scalar field topology. *IEEE Trans. Visualization and Computer Graphics*, 17(12):2035–2044, 2011.

[33] J. Tierny, A. Gyulassy, E. Simon, and V. Pascucci. Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees. *IEEE Trans. Visualization and Computer Graphics*, 15(6):1177–1184, 2009.

[34] J. Tierny, J.-P. Vandeborre, M. Daoudi, et al. 3d mesh skeleton extraction using topological and geometrical analyses. In *Proc. Pacific Graphics*, 2006.

[35] 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.*, pages 212–220, 1997.

[36] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. R. Schikore. Contour trees and small seed sets for isosurface traversal. Technical Report UU-CS-1998-25, Department of Computer Science, Utrecht University, 1998.

[37] G. Weber, P.-T. Bremer, M. Day, J. Bell, and V. Pascucci. Feature tracking using reeb graphs. In *Topological Methods in Data Analysis and Visualization*, pages 241–253. Springer, 2011.

[38] G. H. Weber, S. E. Dillard, H. Carr, V. Pascucci, and B. Hamann. Topology-controlled volume rendering. *IEEE Trans. Visualization and Computer Graphics*, 13(2):330–341, 2007.

[39] Z. Wood, H. Hoppe, M. Desbrun, and P. Schröder. Removing excess topology from isosurfaces. *ACM Trans. Graph.*, 23(2):190–208, 2004.

[40] E. Zhang, K. Mischaikow, and G. Turk. Feature-based surface parameterization and texture mapping. *ACM Trans. Graph.*, 24(1):1–27, 2005.

[41] J. Zhou and M. Takatsuka. Automatic transfer function generation using contour tree controlled residue flow model and color harmonics. *IEEE Trans. Visualization and Computer Graphics*, 15(6):1481–1488, 2009.