Contour Forests: Fast Multi-threaded Augmented Contour Trees

Charles Gueunet*
Kitware SAS
Sorbonne Universites,
UPMC Univ Paris 06,
CNRS, LIP6 UMR 7606,
France.

Pierre Fortin†
Sorbonne Universites,
UPMC Univ Paris 06,
CNRS, LIP6 UMR 7606,
France.

Julien Jomier‡
Kitware SAS

Julien Tierny§
Sorbonne Universites,
UPMC Univ Paris 06,
CNRS, LIP6 UMR 7606,
France.

Abstract

This paper presents a new algorithm for the fast, shared memory multi-threaded computation of contour trees on tetrahedral meshes. In contrast to previous multi-threaded algorithms, our technique computes the *augmented* contour tree. Such an augmentation is required to enable the full extent of contour tree based applications, including for instance data segmentation. Our approach relies on a range-driven domain partitioning. We show how to exploit such a partitioning to rapidly compute *contour forests*. We also show how such forests can be efficiently turned into the output contour tree. We report performance numbers that compare our approach to a reference sequential implementation for the computation of augmented contour trees. These experiments demonstrate the run-time efficiency of our approach. We demonstrate the utility of our approach with several data segmentation tasks. We also provide a lightweight VTK-based C++ implementation of our approach for reproduction purposes.

1 Introduction

As scientific data-sets become more intricate and larger in size, advanced data analysis algorithms are needed for their efficient visualization and interactive exploration. For scalar field visualization, topological analysis techniques have shown to be practical solutions in various contexts by enabling the concise and complete capture of the structure of the input data into high-level topological abstractions such as contour trees [8], Reeb graphs [31, 5, 37], or Morse-Smale complexes [21, 13]. Such topological abstractions are fundamental data-structures that enable the development of advanced data analysis, exploration and visualization techniques, including for instance: small seed set extraction for fast isosurface traversal [39, 9], feature tracking [34], data-summarization [30], transfer function design for volume rendering [40], similarity estimation [36], or application-driven segmentation and analysis tasks [25, 23, 7, 20, 22].

However, with the ongoing development of computational resources on the one hand and of acquisition devices on the other, the resolution of the geometrical domains on which scalar fields are defined is continuously increasing. This resolution increase yields several technical challenges for topological data analysis, including that of computation time efficiency. In particular, to enable truly interactive exploration sessions, highly efficient algorithms are required both for the computation of topological abstractions as well as for their interactive manipulation (i.e. topological simplification for instance). A natural direction towards the improvement of the time efficiency of topological data analysis is parallelism, as all commodity hardware now embeds processors with multiple cores. However, most topological analysis algorithms are originally intrinsically sequential as they often require a global view on the data.

Regarding the contour tree – a fundamental topology-based data structure in scalar field visualization – several algorithms have been proposed for its parallel computation [29, 26, 1]. However, these algorithms only compute *non-augmented* contour trees [8], trees that only represent the connectivity evolution of the level-sets, and not the corresponding data-segmentation (i.e. the arcs are not augmented with regular vertices). While such non-augmented trees enable some of the traditional visualization applications of the contour tree, they do not enable them all. For instance, they do not readily support seed set extraction for fast isosurface extraction or topology based data segmentation. In both examples, additional post-processing will be needed to support these applications. Moreover, fully augmenting in a post-process non-augmented trees is a non trivial task, for which no linear-time algorithm has ever been documented to the best of our knowledge.

This paper addresses this problem by presenting a fast, shared memory multi-threaded algorithm for the computation of augmented contour trees on tetrahedral meshes. Such a tree augmentation makes our output data-structures generic application-wise and enables the full extent of contour tree based applications, including data segmentation. Extensive experiments demonstrate the efficiency of our approach in comparison to a reference sequential implementation (*libtourtre* [14]), despite the strong memory-bound aspect of the graph traversal tasks [2] involved in augmented contour tree computation , and despite other limitations that we detail. We illustrate the utility of our approach with specific use cases for the interactive exploration of hierarchies of topology-based data segmentations that were enabled by our algorithm. We also provide a lightweight VTK-based C++ reference implementation of our approach for reproduction purposes.

1.1 Related work

The notion of contour tree has first been introduced in Computer Science by Boyell and Ruston [6]. Algorithms for their efficient computation have first been investigated for 2D domains [39], then for 3D domains [35] and last for domains of arbitrary dimension [8], with an algorithm that is simple to implement, efficient in practice and with optimal time complexity. In particular, this algorithm allows for the computation of both augmented and non-augmented contour trees. An open source reference implementation (*libtourtre* [14]) of this algorithm is provided by Scott Dillard. Chiang et al. [10] later presented an output-sensitive algorithm for the computation of non-augmented contour trees based on monotone paths, where the arcs of the intermediate trees (called join and split trees, see Sec. 2) were evaluated by considering monotone paths connecting the critical points of the input scalar field. Applications of the contour tree in data analysis and visualization include small seed set extraction for isosurface traversal [39], topological simplification of isosurfaces [9], feature tracking [34], transfer function design for volume rendering [40], similarity estimation [36], or application-driven segmentation and analysis tasks [7]. Note that all of the applications mentioned above require the *augmented* contour tree as they rely on the identification of the sets of regular vertices mapping

*E-mail: charles.gueunet@kitware.com
†E-mail: pierre.fortin@lip6.fr
‡E-mail: julien.jomier@kitware.com
§E-mail: julien.tierny@lip6.fr

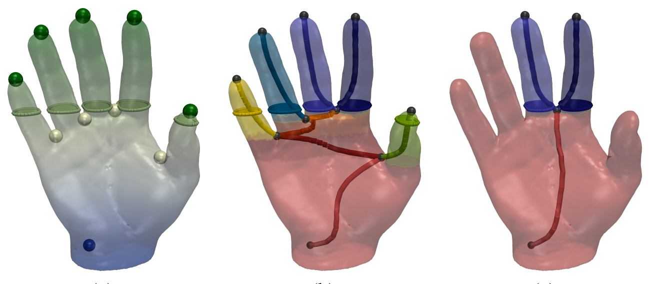


Figure 1: Topology driven hierarchical data segmentation. (a) Input scalar field f (color gradient), level-set (light green) and critical points (blue: minimum, white: saddle, green: maximum). (b) Contour tree of f and its corresponding segmentation (arcs and their pre-images by ϕ are shown with the same color). (c) Contour tree of f and its corresponding segmentation, simplified according to persistence.

to each arc of the contour tree, to find isosurface traversal seeds, to compute geometrical measures on the volume regions corresponding to each arc of the contour tree, to measure overlaps between such regions, to compare geometrical measures on such regions or to simply extract these regions for visualization purpose.

Pascucci and Cole-McLaughlin introduced the first parallel algorithm for contour tree computation [29]. However, this algorithm does not compute contour trees that are augmented with regular vertices[1]. Moreover, the algorithm proceeds with a hierarchical domain decomposition. This means that fewer and fewer threads will be active along the execution, eventually finishing with a non-negligible sequential pass. Also, while this paper describes a parallel algorithm for the computation of intermediate trees (the join and split trees, see Sec. 2), the authors still use a sequential pass to combine these two trees into the final output contour tree.

Maadasamy et al. [26] introduced a multi-threaded variant of the output-sensitive algorithm by Chiang et al. [10], which results in good scaling performances on tetrahedral meshes. However, we note that, in practice, the sequential version of this algorithm is up to three times slower than the reference implementation (*libtourtre* [14]) of the algorithm by Carr et al. [8] (see Tab. 1 in [26]). This only yields eventually speedups between 1.6 and 2.8 with regard to libtourtre [14] on a 8-core CPU [26]. We suspect that these moderate speedups over libtourtre are due to the lack of efficiency of the sequential algorithm based on monotone paths by Chiang et al. [10] in comparison to that of Carr et al. [8]. Indeed, from our experience, although the extraction of the critical points of the field is a local operation [4], we found in practice that its overall computation time is often larger than that of the contour tree itself. Moreover, this algorithm triggers monotone path computations for each saddle point [10], even if it does not yield branching in the contour tree (which induces unnecessary computations). Note also that the approach by Maadasamy et al. [26] only parallelizes the construction of the join and split trees and not their combination into the final contour tree, which is done sequentially. Finally, since it connects critical points through monotone paths, this algorithm does not visit all the vertices of the input mesh. Thus it cannot provide a contour tree-based data segmentation and does not produce an augmented contour tree. Acharya and Natarajan [1] specialized and improved this approach for the special case of regular grids. In this paper, we focus however on tetrahedral meshes because of the genericity of this mesh representation (any mesh can be triangulated).

Morozov and Weber [27, 28] and Landge et al. [24] presented three approaches for contour tree-based visualization in a distributed environment, with minimal inter-node communications. However, these approaches focus more on the reduction of the communication between the processes than on the efficient computation on a single shared memory node as we do here with the target of an efficient interactive exploration in mind.

1.2 Contributions

This paper presents the following new contributions:

1. a fast, shared memory multi-threaded algorithm for the computation of augmented contour trees on tetrahedral meshes;

2. a lightweight VTK-based C++ reference implementation of our approach for reproduction purposes.

2 Preliminaries

This section briefly describes our formal setting and presents an overview of our approach. An introduction to Topological Data Analysis can be found in [16].

2.1 Background

The input to our algorithm is a piecewise linear (PL) scalar field $f : \mathcal{M} \rightarrow \mathbb{R}$ defined on a simply-connected PL d-manifold $\mathcal{M}$. Without loss of generality, we will assume that $d = 3$ (tetrahedral meshes) in most of our discussion, although our algorithm, as it extends Carr's, supports manifolds of arbitrary dimension. The scalar field f is provided on the vertices of $\mathcal{M}$ and is linearly interpolated on the simplices of higher dimension. We will additionally require that the restriction of f to the vertices of $\mathcal{M}$ is injective (which can be easily enforced with a mechanism inspired by simulation of simplicity [18]).

A level-set is defined as the pre-image of an isovalue $i \in \mathbb{R}$ onto $\mathcal{M}$ through f: $f^{-1}(i) = \{p \in \mathcal{M} \mid f(p) = i\}$ (Fig. 1(a)). Each connected component of a level-set is called a *contour*. In Fig. 1(b), each contour of the level-set of Fig. 1(a) is shown with a distinct color. Let $f^{-1}(f(p))_p$ be the contour that contains the point p. The *Reeb graph* [32] is a 1-dimensional simplicial complex (Fig. 1(b)) that is defined as the quotient space $\mathcal{R}(f) = \mathcal{M} / \sim$ by the equivalence relation $p_1 \sim p_2$:

$$\begin{cases} f(p_1) = f(p_2) \\ p_2 \in f^{-1}(f(p_1))_{p_1} \end{cases}$$

Note that f can be decomposed into $f = \psi \circ \phi$ where $\phi : \mathcal{M} \rightarrow \mathcal{R}(f)$ is a *contour retraction* [37] (i.e. a continuous map that retracts each contour to a point such that the restriction of such a map to its image is the identity) and where $\psi : \mathcal{R}(f) \rightarrow \mathbb{R}$ is a continuous function that maps points of $\mathcal{R}(f)$ to their f values. Note that the pre-image by ϕ of $\mathcal{R}(f)$ induces a complete partition of $\mathcal{M}$. In particular, the pre-image $\phi^{-1}(\Sigma_1)$ of a 1-simplex $\Sigma_1 \in \mathcal{R}(f)$ is guaranteed by construction to be connected. This latter property is at the basis of the usage of the Reeb graph in visualization as a data segmentation tool (Fig. 1(b)) for feature extraction or isosurface traversal acceleration. In practice, ϕ^{-1} is represented explicitly by maintaining, for each 1-simplex $\Sigma \in \mathcal{R}(f)$ (i.e. for each arc), the list of regular vertices of $\mathcal{M}$ that retracts to Σ. Moreover, since the Reeb graph is a simplicial complex, persistent homology concepts [17] can be readily applied to it by considering a filtration based on ψ. Intuitively, this progressively simplifies $\mathcal{R}(f)$, by iteratively removing its *shortest* arcs, as described in further details in [31]. This yields hierarchies of Reeb graphs that are accompanied by hierarchies of data segmentations, that the user can interactively explore in practice (see Fig. 1(c)).

[1]Pascucci and Cole-McLaughlin use the term "augmented" with a different meaning than in the current paper. Here, it refers to the augmentation of the arcs with regular vertices (as introduced by Carr et al. [8]) while they refer to an evaluation of the Betti numbers of the level sets.

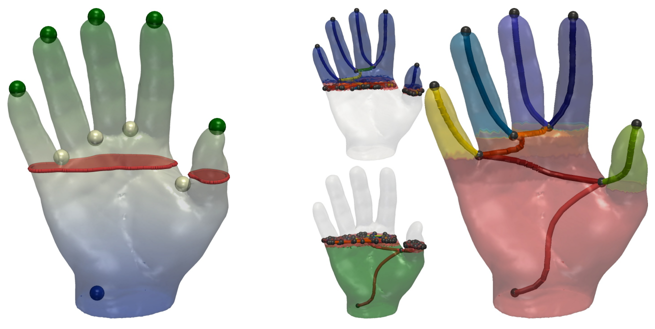


Figure 2: Algorithm overview on the height function f of a volume $\mathscr{M}$ with two threads. (a) Input scalar field f (color gradient) with its critical points (blue: min, white: saddle, green: max). The domain is split into two partitions $\mathscr{P}_i$ and $\mathscr{P}_j$ of roughly equal size corresponding to the pre-images of contiguous intervals $\mathscr{I}_i$ and $\mathscr{I}_j$ of $f(\mathscr{M})$. The *interface level-set* between such two partitions is shown in red. (b) The augmented contour trees $\mathscr{C}(f)_i$ (top) and $\mathscr{C}(f)_j$ (bottom) are constructed in parallel for each partition. These local trees can be easily and efficiently stitched together to form the output augmented contour tree (right).

As discussed by Cole-McLaughlin et al. [11], the construction of the Reeb graph can lead to the removal of 1-cycles, but not to the creation of new ones. This means that the Reeb graphs of PL scalar fields defined on simply-connected domains are loop-free. Such a Reeb graph is called a *contour tree* and we will note it $\mathscr{C}(f)$. Contour trees can be computed efficiently by considering the combinations of two trees called *join* and *split trees* [8]. Given an isovalue $i \in \mathbb{R}$, the *sub-level set* $L^-(i)$ is defined as the pre-image of the open interval $(-\infty, i]$ onto $\mathscr{M}$ through f: $L^-(i) = \{p \in \mathscr{M} \mid f(p) \leq i\}$. The *join tree*, noted $\mathscr{J}(f)$, is a 1-dimensional simplicial complex obtained by contracting each connected component of a sub-level set to a point (similarly to the Reeb graph). The 0-simplices of $\mathscr{J}(f)$ that are attached to only one 1-simplex are either the root of this tree (corresponding to the global maximum of f) or its leaves (corresponding to the local minima of f). All other 0-simplicies (which induce branching in the tree) correspond to saddles of f that join distinct connected components of $L^-(i)$. The split tree $\mathscr{S}(f)$ of f is defined symmetrically, by considering the *sur-level sets* $L^+(i)$: $L^+(i) = \{p \in \mathscr{M} \mid f(p) \geq i\}$.

2.2 Overview

Our approach is based on a range-driven partitioning strategy, as illustrated in Fig. 2. First, given n_t threads, the image of the domain $f(\mathscr{M})$ is divided into $n_t/2$ contiguous, non-overlapping intervals $\mathscr{I}_i$ that contain (nearly) the same amount of vertices of $\mathscr{M}$ (Sec. 3.1):

$$f(\mathscr{M}) = \mathscr{I}_0 \cup \mathscr{I}_1 \cup \dots \cup \mathscr{I}_{n_t-1} \qquad (1)$$
$$|\sigma_0|_i \approx |\sigma_0|_j \quad \forall i \neq j$$

where $|\sigma_0|_i$ refers to the number of vertices of $\mathscr{M}$ mapping to $\mathscr{I}_i$. Next, two threads are assigned to each partition $\mathscr{P}_i$, $\mathscr{P}_i$ being the pre-image of the corresponding interval, $\mathscr{P}_i = f^{-1}(\mathscr{I}_i)$ (Sec. 3.1). The two threads then compute the augmented contour tree of the restriction of the function to its partition (Sec. 3.2), with a variant of the algorithm by Carr et al. [8]: one thread builds the join tree, and the other the split tree[2]. This yields a *forest* of contour trees: $\{\mathscr{C}(f)_0, \mathscr{C}(f)_1, \dots \mathscr{C}(f)_{n_t-1}\}$. Finally, the output contour tree is retrieved by connecting the trees of the forest along common connected components of partition boundaries (Sec. 3.3).

[2]Note that n_t threads could have been used (one thread per partition), by building these trees sequentially. However, our experiments showed that it was less efficient than using two threads per partition.

Despite its simplicity, our range-driven approach exhibits many advantages. In particular, our strategy enables the computation of augmented contour trees, since it extends Carr's algorithm [8]. Second, since the input mesh is split into partitions of roughly equal size (in terms of vertices), the work load should be well balanced between the threads. Third, since it is range-driven, our approach allows for a full computation of the local contour tree within each partition (join and split tree computations, plus their combination in the contour tree) while previous approaches systematically delayed the combination to a post-process pass (implemented in serial). Finally , since it is range-based, our approach allows for a simple stitching of the local trees of the forest into the output contour tree, while previous approaches needed to run a special procedure on the common boundary of merged partitions [29, 26].

3 Algorithm

This section details the algorithms for each step of our approach.

3.1 Domain partitioning

The first step of our approach consists in sorting the vertices of $\mathscr{M}$ by increasing function value, which can be efficiently done in parallel [38, 33, 19]. This step can be done in $O(|\sigma_0| \times log(|\sigma_0|))$ where $|\sigma_0|$ is the number of vertices in $\mathscr{M}$. Nex, the sorted list of vertices is split into $n_t/2$ contiguous sets $\mathscr{P}_i$ of roughly equal size, whose images correspond to the intervals $\mathscr{I}_i$ described in Eq. 1. Next, each vertex set $\mathscr{P}_i$ is extended into a set $\mathscr{P}'_i$ with the following procedure. Let f_{i^-} and f_{i^+} be the two extremities of the interval $\mathscr{I}_i$: $\mathscr{I}_i = (f_{i^-}, f_{i^+})$. The level-sets for the isovalues f_{i^-} and f_{i^+} are called *interface level-sets*. Let $(\sigma_1)_{i^-}$ and $(\sigma_1)_{i^+}$ be the the set of edges of $\mathscr{M}$ whose image contains f_{i^-} and f_{i^+} respectively. The vertex set $\mathscr{P}_i$ is extended into $\mathscr{P}'_i$ by adding the vertices of $(\sigma_1)_{i^-}$ and $(\sigma_1)_{i^+}$ (red circles, Fig. 3). We call such vertices *boundary vertices*. Note that with this approach, two adjacent partitions $\mathscr{P}'_i$ and $\mathscr{P}'_j$ will overlap, precisely along the simplices crossed by f_{i^-} or f_{i^+} (triangles with red edges, Fig. 3).

This strategy guarantees that each connected component of an interface level-set is captured by the overlaps in between the partitions (triangles with red edges, Fig. 3). Therefore, all possible contours living in the interval $\mathscr{I}_i$ are completely captured by $\mathscr{P}'_i$. This guarantees that the restriction of the local contour tree $\mathscr{C}(f)_i$ (computed on $\mathscr{P}'_i$) to the interval $\mathscr{I}_i$ (in green in Fig. 3(a) and blue in Fig. 3(b)) is equal the restriction of the output contour tree $\mathscr{C}(f)$ to $\mathscr{I}_i$. This property will be of paramount importance to guarantee an efficient stitching of the contour forest into the output contour tree (Sec. 3.3).

In practice, this expansion procedure is performed efficiently by visiting in parallel all the edges (σ_1) of $\mathscr{M}$ and tracking the vertices of the edges crossing f_{i^-} and f_{i^+} for a given interval $\mathscr{I}_i$, in $O(|\sigma_1|)$ steps. In particular, each of the n_t threads maintains its own list of boundary vertices, which are merged globally (and sequentially in practice since this merge implies minor computation times).

3.2 Local computations

The contour tree $\mathscr{C}(f)_i$ of each of the $n_t/2$ partitions $\mathscr{P}'_i$ is computed by two distinct threads. Note that in practice, the partitions $\mathscr{P}'_i$ are not copied, but represented implicitly. In particular, the list of vertices of the initial partition $\mathscr{P}_i$ is represented by an interval in the global sorted list of vertices. The boundary vertices added in the expansion procedure described in Sec. 3.1 (red circles, Fig. 3) are represented by two sorted lists of vertices $\mathscr{B}_{i^-}$ and $\mathscr{B}_{i^+}$, representing the boundary vertices below f_{i^-} and above f_{i^+} respectively.

Given a partition $\mathscr{P}'_i$, its augmented contour tree is computed with a variant of the algorithm by Carr et al. [8], for which we

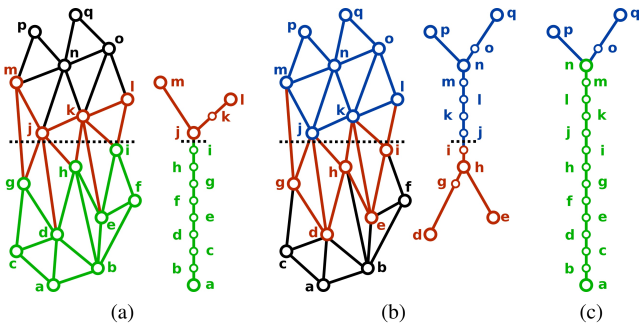


Figure 3: Domain partitioning for a 2D toy example (height function). (a) Partition $\mathscr{P}_i$ (green) with its overlap simplices (red) and its augmented contour tree $\mathscr{C}(f)_i$. (b) Partition $\mathscr{P}_j$ (blue) with its overlap simplices (red) and its augmented contour tree $\mathscr{C}(f)_j$. The common region between the two partitions is made of all the triangles containing red edges and being crossed by the interface level-set (black dashes). (c) The output, stitched, augmented contour tree $\mathscr{C}(f)$.

briefly sketch the main steps here for completeness. This algorithm constructs the join tree $\mathscr{J}(f)$ (and symmetrically the split tree $\mathscr{S}(f)$) by visiting the vertices of the domain in increasing function value and by keeping track of the connectivity evolution of the sub-level sets $L^{-}(i)$ with a Union-Find data-structure [12]. Given an input sorted list of vertices, this steps requires $O(|\sigma^i| \times \alpha(|\sigma^i|))$ steps, where $|\sigma^i|$ is the number of simplices in the partition $\mathscr{P}_i'$ and where α $\alpha(.)$ is an extremely slow-growing function (inverse of the Ackermann function). Next, the join tree $\mathscr{J}(f)$ and the split tree $\mathscr{S}(f)$ are combined into the output augmented contour tree $\mathscr{C}(f)$ by iteratively removing arcs connected to leaves either in $\mathscr{J}(f)$ or $\mathscr{S}(f)$ and adding them to $\mathscr{C}(f)$ [8]. This final steps takes a linear time with the number of arcs in $\mathscr{C}(f)$ (which is of the same order of magnitude than $|\sigma^i|$ in practice).

Our approach to the local computation of the augmented contour tree $\mathscr{C}(f)_i$ for each partition $\mathscr{P}_i'$ only requires a slight modification to this algorithm. In particular, when constructing the join tree $\mathscr{J}(f)_i$, our algorithm first visits the boundary vertices $\mathscr{B}_{i^-}$ (if any) in increasing order. Next, it visits the vertices of $\mathscr{P}_i$ by traversing the global sorted vertex list within the interval prescribed by the domain partitioning step (Sec. 3.1). Finally, it completes the traversal by considering the vertices of $\mathscr{B}_{i^+}$ (if any) in increasing order. For each of these three traversals, the join tree construction algorithm by Carr et al. [8] is applied as-is by the corresponding thread. The split tree $\mathscr{S}(f)_i$ is constructed with a symmetrical pass by the other thread : $\mathscr{B}_{i^+}$, then $\mathscr{P}_i$ and finally $\mathscr{B}_{i^-}$. Once the join and split trees are constructed, they are combined into the augmented contour tree $\mathscr{C}(f)_i$ with the original algorithm [8] by one of the two threads. This combination does not require parallelization within each partition since its computation time is not significant, and since parallelization already applies among partitions.

3.3 Contour forest stitching

Once the n_t threads have finished the computation of each local augmented contour tree $\mathscr{C}(f)_i$, the resulting forest is stitched into the final augmented contour tree $\mathscr{C}(f)$ with the following procedure.

During the local computation of the contour tree $\mathscr{C}(f)_i$ (Sec. 3.2), each 1-simplex (each arc) that crosses an interface level set is added to a list of *crossing arcs*, noted $\mathscr{X}_i$. This corresponds in Fig. 3 to the arcs (a, j) in Fig. 3(a) and $(h-n)$ in Fig. 3(b)

Then, the stitching procedure consists in visiting sequentially the list of crossing arcs $\mathscr{X}_i$ for each local contour tree $\mathscr{C}(f)_i$. Given such an arc a_i, its regular vertex v exactly above f_{i^+} (or below f_{i^-}) is identified through dichotomic search (vertex j in Fig. 3(a)). Since augmented contour trees store the destination of each vertex into the tree, it is possible to retrieve in constant time the *homologous* arc a_j from the adjacent tree $\mathscr{C}(f)_j$ which contains v. This corresponds to the arc $(h-n)$ in Fig. 3(b). Finally, a_i is updated to form the union of the arcs a_i and a_j. This operation includes the modification of the higher extremity of a_i (to use a_j's instead) as well as the concatenation of the two sorted lists of regular vertices (see Fig. 3(c)). Note that the vertex v can belong to multiple partitions. In such a case, a_i will be updated iteratively to form the union of multiple arcs (a_i, a_j, a_k, etc.), by successively applying this pairwise stitching in increasing order of function value (i.e. a_i and a_j will first be stitched, then the result of this stitching will be stitched with a_k and so on). As discussed in Sec. 4, this final stitching procedure is extremely fast in practice (hence performed sequantially), since only a small portions of the arcs are visited (only the crossing arcs) and since the merging operation is simple (it simply consists in stitching pairs of arcs across interface level-sets). Note that the simplicity of this stitching procedure is due to our domain partitioning strategy, which guarantees that the restriction of a local tree $\mathscr{C}(f)_i$ to the interval $\mathscr{I}_i$ is equal to the restriction of $\mathscr{C}(f)$ to $\mathscr{I}_i$ (Sec. 3.1). Note that the zipping procedure employed in the streaming Reeb graph computation algorithm [31] could also be employed for the stitching of the local trees. However, this procedure admits a quadratic time complexity in the number of nodes of $\mathscr{C}(f)$, which is prohibitive in our approach, where only sub-quadratic routines have been used.

4 Experimental results

In this section, we present practical results obtained with a VTK-based C++ implementation of our algorithm (provided as additional material). Experiments were performed on a desktop computer with an Intel Xeon CPU E5-2630 v3 (2.4 GHz, 8 cores) with 64GB of RAM. All parallel tests are run with $n_t = 8$ threads for $n_p = 4$ partitions.

4.1 Detailed performance results

Table 1 first presents detailed performance results for various data-sets. Plasma, Bucky and SF Earthquake are standard data-sets available at the AIM@SHAPE repository [15]: these are however too small to fully exploit our 8-core CPU. In fact, with our experimental setting, even the sort step is slower in parallel than in sequential for these data-sets (results not shown). This explains the low parallel speedups for such tests.

We thus focus in the following on larger tetrahedral meshes (upper part of Table 1) which have been obtained by triangulating regular grids. For the sake of comparison, these have systematically been upsampled to 256^3 vertices. One can first see that the additionnal Overlap step required to build the $\mathscr{B}_{i^-}$ and $\mathscr{B}_{i^+}$ lists (see Sec. 3.2) leads to low overheads. Moreover, the stitching step is efficiently performed in sequential which results in small run-times. As far as parallel speedups are concerned, the Elevation data-set is a synthetic and very simple one that shows good speedups, with a parallel efficiency of $5.38/8 = 67\%$. Moving to more complex data-sets (i.e. resulting in larger contour trees), one can see that we obtain good or average speedups (parallel efficiencies ranging between 55% and 40%), except for the Foot data-set which shows a limited speedup. We will detail these limitations and their causes in the next sub-section. The scalability of our approach is evaluated with Fig. 4, which presents the evolution of the speedup obtained by our algorithm as a function of the number of threads. The slope of these curves shows that the scalability of our approach, similarly to the speedups discussed above, is data-set dependent as well. In particular, our algorithm seems less scalable for the data-sets which result in complex output trees (see the third column of Table 1 which shows the number of arcs per tree). Also, the smallest slope (nearly constant) is also observed with the Foot data-set,

Table 1: Running time of the different steps of the algorithm (in seconds). $|\mathcal{M}|$ denotes the number of vertices in the data-set, and $|\mathcal{C}(f)|_A$ the number of arcs in the output contour tree. Overall corresponds to the complete application, including memory allocations, etc.

Data-set	$\|\mathcal{M}\|$	$\|\mathcal{C}(f)\|_A$	Sequential	Sort	Overlap	Local trees	Stitching	Overall	Speedup
Elevation	82,906,875	1	29.18	0.91	0.18	4.18	0.14	5.42	**5.38**
EthaneDiol	82,906,875	29	33.09	0.67	0.33	6.64	0.14	7.81	**4.37**
Combustion	82,906,875	3649	28.04	0.61	0.34	6.19	0.15	7.31	**3.83**
Boat	82,906,875	3235	29.94	0.69	0.41	6.17	0.14	7.44	**4.02**
Jet	82,906,875	4171	26.82	0.65	0.36	6.03	0.15	7.21	**3.72**
Enzo	82,906,875	282800	39.63	0.74	1.50	9.48	0.66	12.40	**3.20**
Foot	82,906,875	844463	18.09	0.49	0.99	7.12	1.10	9.72	**1.86**
Plasma	1,310,720	2851	0.18	0.01	0.01	0.06	0.01	0.09	**2**
Bucky	1,250,235	4377	0.11	0.01	0.01	0.05	0.01	0.08	**1.38**
SF Earthquake	2,067,739	11887	0.19	0.01	0.02	0.09	0.02	0.13	**1.46**

Table 2: Overall running time comparison (in seconds) between the sequential libTourtre implementation (sTourtre), a naive parallel implementation of libTourtre (pTourtre) and our approach.

Data-set	sTourtre	pTourtre	Speedup wrt.	Ours	Speedup wrt.	
			sTourtre		sTourtre	pTourtre
Elevation	20.63	10.07	2.04	5.42	3.81	2.64
EthaneDiol	23.47	13.96	1.68	7.81	3.00	1.79
Combustion	21.26	12.39	1.72	7.31	2.91	1.70
Boat	23.26	12.52	1.85	7.44	3.13	1.68
Jet	20.60	11.50	1.79	7.21	2.86	1.60
Enzo	32.51	18.07	1.80	12.40	2.62	1.46
Foot	13.52	8.40	1.60	9.72	1.39	0.86
Plasma	0.08	0.08	1.00	0.09	0.89	0.89
Bucky	0.07	0.06	1.16	0.08	0.88	0.75
SF Earthquake	0.12	0.10	1.20	0.13	0.92	0.77

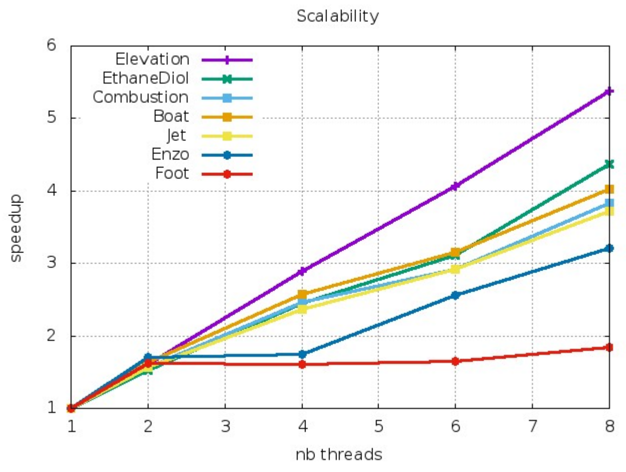


Figure 4: Speedups obtained by our algorithm as a function of the number of threads (one curve per data set).

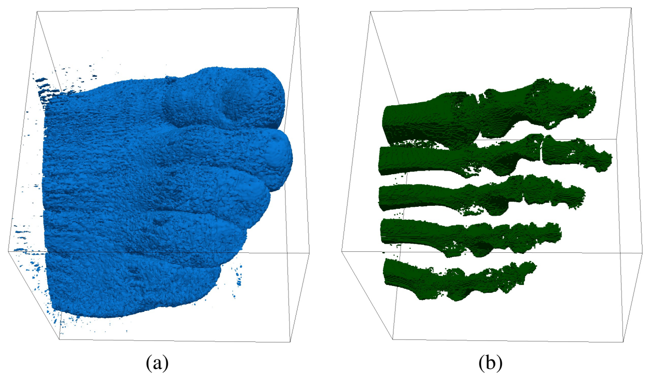


Figure 5: Size difference between two interface level-sets on the Foot data-set. (a) Interface crossing 1,507,357 edges (blue). (b) Interface crossing 606,276 edges (green). This difference can lead to load imbalance between our partitions.

as further discussed in the next sub-section.

Next, we compare our parallel implementation with the reference sequential libtourtre implementation in Table 2. Our speedups with respect to libtourtre range between 2.6 and 3.8 (except for Foot), which compares favorably (even if the data-sets differ) with the 1.6-2.8 speedups of Maadasamy et al. [26] on tetrahedral meshes (see Sec. 1.1). Note additionally that in contrast to Maadasamy et al. [26], our approach computes the *augmented* contour tree, which constitutes a more generic and versatile version of the contour tree. We also added in Table 2 performance results of a naive parallel implementation of libtourtre [14] which uses a parallel sort and two OpenMP threads to build independently the join and split trees. Our parallel algorithm outperforms this naive implementation in all our test cases with exception of the foot, which further stresses the efficiency of our approach.

Table 3: Partition sizes (in vertices).

Data-set	ideal	min	max
Elevation	4,194,304	4,259,840	4,325,376
EthaneDiol	4,194,304	4,362,086	4,616,938
Combustion	4,194,304	4,353,986	4,635,078
Boat	4,194,304	4,418,409	4,791,092
Jet	4,194,304	4,358,176	4,701,586
Enzo	4,194,304	5,234,144	6,474,322
Foot	4,194,304	4,499,572	6,044,708

4.2 Limitations

In this section, we detail the three factors that limit our parallel speedups in practice.

As presented in Table 3, we can see that the actual number of vertices per partition is always greater than the ideal one (obtained by dividing the total number of vertices by the number of partitions). This is due to the boundary vertices that have to be added to each partition, which imply redundant computations that directly impact the parallel speedups. In particular, there can be important variations in the size of the interface level sets (in terms of crossed edges, in red in Fig. 3) within a single data-set, as shown in Fig. 5. Therefore the size of the overlaps between the partitions (expressed as the number of vertices in the lists $\mathscr{B}_{i^-}$ and $\mathscr{B}_{i^+}$, see Sec. 3.2) can also vary. This induces redundant computations of varying importance within a single data-set. One can also see larger imbalance in the number of vertices per partition for more complex data-sets. This adds load imbalance to the parallel computations which further decreases the speedups.

This load imbalance is worsened by the fact that, depending on its impact on the join and split tree constructions, each vertex of $\mathscr{M}$ does not require the same processing time in practice. This effect is shown in Table 4 which shows varying computation speeds among the different partitions of a given data-set. In particular, the more complex is the contour tree, the larger is the gap among the computation speeds. One could choose to use several partitions per thread, with dynamic load balancing, in order to minimize such load imbalance, but this would introduce even more redundant computations (because of the boundary vertices). That is why we choose to use $n_t/2$ partitions for n_t threads: this indeed minimizes the redundant computations, while fully exploiting the complete independency between the join and split tree computations.

These two factors (redundant computations and load imbalance) jointly explain our lower speedups with more complex data-sets (especially for Foot).

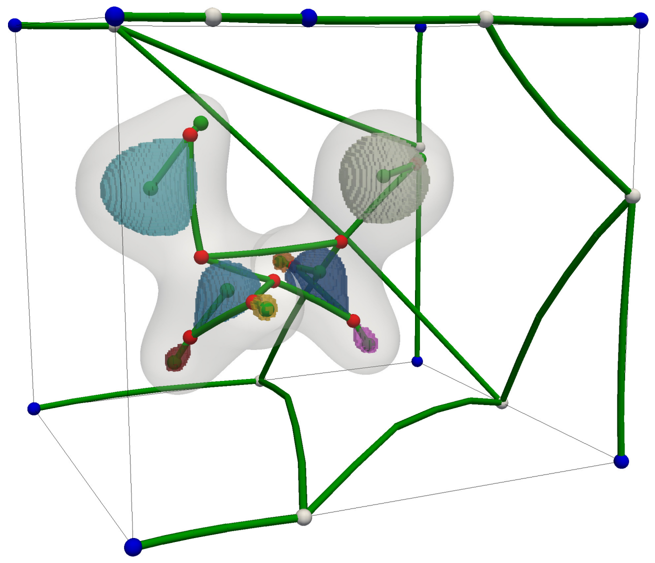
Finally, a third factor also limits our parallel efficiencies for any data-set. The contour tree computation requires on average very few operations with respect to the number of memory accesses. Its operational intensity [41] is therefore low which makes such an application memory-bound, like most graph traversal algorithms [2]. As shown in Table 4, giving the parallel computations of the join and split trees to a single thread leads to higher computations speeds. Hence speedups linear in the number of cores cannot be obtained for such memory-bound applications: the memory bandwidth of the processor can not cope with the memory requirement of the 8 threads at a time. This also justifies our choice not to rely on the 2-way SMT (*Simultaneous MultiThreading*) capability of our CPU, and to use only 1 thread (instead of 2) per physical CPU core.

5 Application: Data Segmentation

The purpose of this work is to improve interactivity in contour-tree based data analysis. Our algorithm enables the computation of augmented contour trees within interactive run-times (up to a dozen seconds, Table 1), even for large and complex data-sets. Our aug-

Table 4: Computation speeds (in vertices/second) for join or split tree computations with our parallel implementation, and with one single thread to perform all computations required by our parallel approach.

Data-set	Parallel: min	max	1 thread: min	max
Elevation	1,623,500	1,768,720	2,151,250	2,292,390
EthaneDiol	963,962	1,108,170	1,470,960	1,804,410
Combustion	1,029,050	1,190,160	1,688,080	2,006,210
Boat	1,055,410	1,237,030	1,463,880	1,985,720
Jet	1,065,720	1,256,730	1,754,240	2,094,010
Enzo	860,937	933,616	1,166,540	1,366,070
Foot	1,120,560	4,031,030	1,220,800	5,195,250


(a)

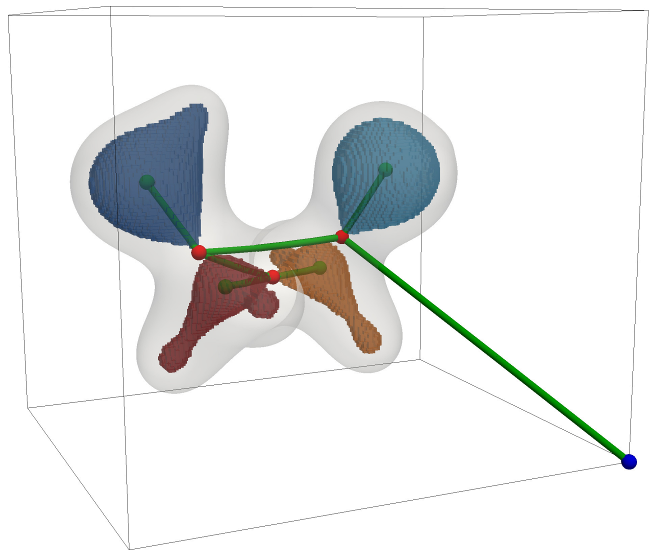

(b)

Figure 6: Augmented contour-trees for the EthaneDiol data-set (electron density) without (a) and (b) with topological simplification. An isosurface of electron density is shown in transparent grey in both cases. Pre-images through ϕ of arcs attached to maxima of electron density (i.e. atoms) are shown in color. This segmentation extracts the regions of influence for each atom (a) or for the main atom groups in the molecule (b).

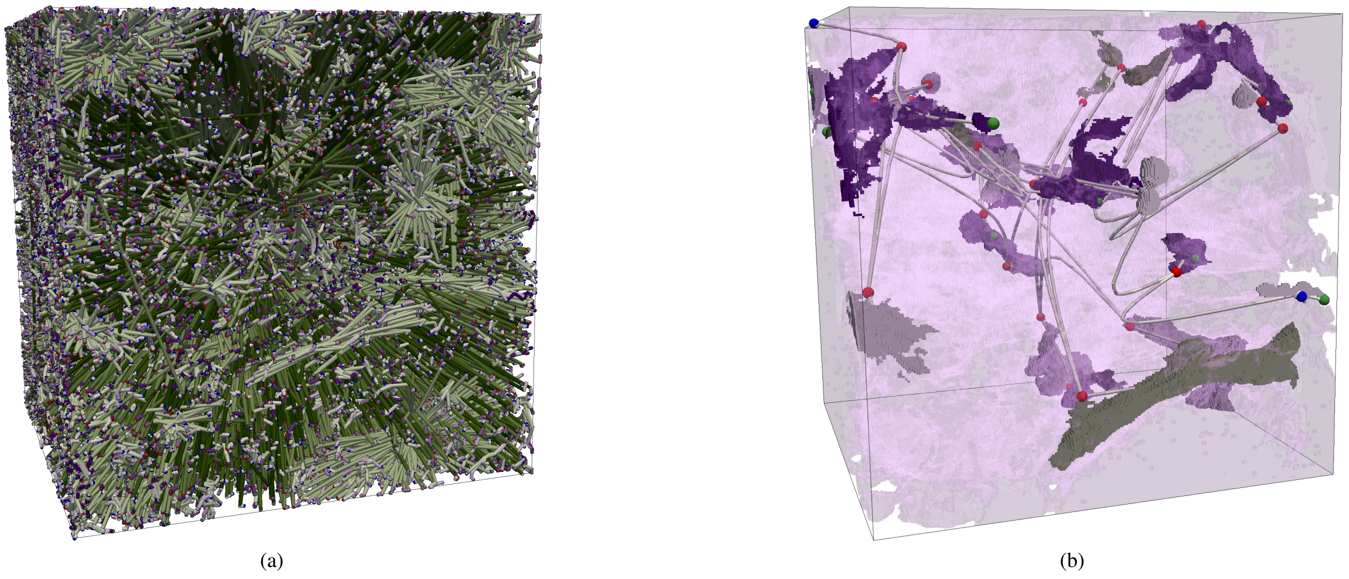


Figure 7: Augmented contour-trees for the Enzo data-set without (left) and with (right) topological simplification. The interactive topological simplification of our contour tree based data-segmentation progressively reveals the core structures of the cosmic web (region of high matter density, opaque surfaces), surrounded by voids of low density (transparent surfaces).

mented output data-structure is readily usable for data segmentation tasks, especially when features of interest coincide with connected components of level-sets. Combined with persistent homology based topological simplification (see Sec. 2.1), this enables the interactive exploration of hierarchies of data segmentations. Such a segmentation capability is an important feature for quantitative analysis [7].

Fig. 6 shows an example of such a contour tree based data segmentation on the EthaneDiol data-set (electron density). In particular, the contour tree is shown with a 3D embedding (colored cylinders). In this example, the pre-images of the arcs attached to maxima (which correspond to atom centers in the electron density field) reveal the influence regions for each atom (Fig. 6(a)) or for each atom group if topological simplification is employed (Fig. 6(b)). Such a segmentation enables the quantitative analysis of these features (for instance volume measurement). Saddles of the electron density (red spheres, Fig. 6) are located in configurations at the boundary between multiple atom influence zones. These correspond to covalent bonds. Last, the minima (blue spheres, Fig. 6) correspond to noise induced by the boundary effects. This noise is easily removed with topological simplification (Fig. 6(b)).

A second use case is shown with the Enzo data-set (Fig. 7) which represents matter density in universe expansion simulations. Regions of high density correspond to the galaxies forming the core structure of the data (often called *cosmic web*). These are surrounded by large zones, called *voids*, of low matter density. Originally the augmented contour tree is composed of hundreds of thousands of arcs (Fig. 7(a)), which challenges its interpretation. Here, the interactive topological simplification of our contour tree based data segmentations progressively reveals the core structures of the cosmic web (Fig. 7(b)). In particular, regions of high matter density correspond to pre-images of arcs attached to maxima (opaque surfaces) while voids correspond to pre-images of arcs attached to minima (transparent surfaces).

In both use cases, such data segmentations were provided by the augmented contour tree, for which our approach enables a computation within interactive times (at most a dozen seconds, Table 1). In comparison, the topological simplification takes typically less than a second (1.25 for the simplification of 436,726 persistence pairs on the Foot data-set) while the update of the segmentation is instantaneous.

6 Conclusion

In this paper, we have presented a fast, shared memory multi-threaded algorithm, based on a range-driven partitioning strategy, for the complete parallel computation of augmented contour trees on tetrahedral meshes. We have also provided a lightweight VTK-based C++ reference implementation of our approach based on OpenMP multi-threading.

In practice, especially for complex and noisy contour trees arising from acquired data-sets, the parallel speedups of our implementation are limited by redundant computations and load imbalance among the partitions, as well as by the memory-bound nature of the contour tree computation, which we have demonstrated experimentally (Table 4). Nevertheless, on simulated data-sets we managed to obtain good speedups with respect to the reference sequential libtourtre [14] implementation: these speedups exceed the ones of Maadasamy et al. [26] (although the data-sets differ). Also, our experiments demonstrate the superiority of our approach over a naive parallelization of libtourtre. Moreover, our parallel approach allows for the augmented contour tree computation which enables data-segmentation.

In future work, we plan to investigate processing of larger data-sets that do not fit in memory by designing efficient out-of-core algorithms. Also, our range-driven partitioning strategy seems particularly conducive to a parallelization of the persistence-driven simplification of the trees. We will explore this direction in the future to further improve the interactivity of the user-guided exploration of persistence thresholds for interactive data segmentation. Moreover, we believe our range-driven partitioning scheme could be further improved, in particular by trying to minimize the number of boundary vertices, to reduce computation redundancy. In that perspective, the contour spectrum [3] could be a good candidate for the selection of optimal cutting isovalues, which would cross only few edges. However, the research of the optimal trade-off between the balancing of the partition size and the computation redundancy remains an open question. Last, we would like to investigate in the future the extension of our approach to distributed systems. However, such use cases are particularly appealing in simulation contexts where the data is already distributed at the time of the simulation. This means that the data partitioning scheme is likely to be imposed by the simulation code, which would challenge our approach, based on a range-driven partitioning.

Acknowledgements

This work is supported by the French National Association for Research and Technology (ANRT), in the framework of the LIP6 - Kitware SAS CIFRE partnership reference 2015/1039.

References

[1] A. Acharya and V. Natarajan. A parallel and memory efficient algorithm for constructing the contour tree. In *PacificVis*, 2015.

[2] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The landscape of parallel computing research: a view from berkeley. Technical report, University of California at Berkeley, 2006.

[3] C. Bajaj, V. Pascucci, and D. Schikore. The contour spectrum. In *IEEE VIS*, 1997.

[4] T. F. Banchoff. Critical points and curvature for embedded polyhedral surfaces. *The American Mathematical Monthly*, 1970.

[5] S. Biasotti, D. Giorgio, M. Spagnuolo, and B. Falcidieno. Reeb graphs for shape analysis and applications. *Theoretical Computer Science*, 2008.

[6] R. L. Boyell and H. Ruston. Hybrid techniques for real-time radar simulation. In *Proc. of the IEEE Fall Joint Computer Conference*, 1963.

[7] P. 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 Transactions on Visualization and Computer Graphics*, 2011.

[8] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. In *Proc. of Symposium on Discrete Algorithms*, pages 918–926, 2000.

[9] H. Carr, J. Snoeyink, and M. van de Panne. Simplifying flexible isosurfaces using local geometric measures. In *Proc. of IEEE VIS*, pages 497–504, 2004.

[10] Y. Chiang, T. Lenz, X. Lu, and G. Rote. Simple and optimal output-sensitive construction of contour trees using monotone paths. *Computational Geometry Theory and Applications*, 2005.

[11] K. Cole-McLaughlin, H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci. Loops in Reeb graphs of 2-manifolds. In *Proc. of ACM Symposium on Computational Geometry*, pages 344–350, 2003.

[12] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[13] L. De Floriani, U. Fugacci, F. Iuricich, and P. Magillo. Morse complexes for shape segmentation and homological analysis: discrete models and algorithms. *Computer Graphics Forum*, 2015.

[14] S. Dillard. libtourtre: A contour tree library. `http://graphics.cs.ucdavis.edu/~sdillard/libtourtre/doc/html/`, 2007.

[15] A. I. M. A. T. S. H. A. P. E. AIM@SHAPE Shape Repository. `http://shapes.aim-at-shape.net/`, 2006.

[16] H. Edelsbrunner and J. Harer. *Computational Topology: An Introduction*. American Mathematical Society, 2009.

[17] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. *Discrete & Computational Geometry*, 28:511–533, 2002.

[18] H. Edelsbrunner and E. P. Mucke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9:66–104, 1990.

[19] GNU. C++ Standard Library, Parallel Mode. `https://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html`.

[20] A. Gyulassy, P. Bremer, R. Grout, H. Kolla, J. Chen, and V. Pascucci. Stability of dissipation elements: A case study in combustion. *Computer Graphics Forum (Proc. of EuroVis)*, 2014.

[21] A. Gyulassy, P.-T. Bremer, B. Hamann, and P. Pascucci. A practical approach to Morse-Smale complex computation: scalabity and generality. *IEEE Transactions on Visualization and Computer Graphics (Proc. of IEEE VIS)*, pages 1619–1626, 2008.

[22] A. Gyulassy, A. Knoll, K. Lau, B. Wang, P. Bremer, M. Papka, L. A. Curtiss, and V. Pascucci. Interstitial and interlayer ion diffusion geometry extraction in graphitic nanosphere battery materials. *IEEE Transactions on Visualization and Computer Graphics (Proc. of IEEE VIS)*, 2015.

[23] A. Gyulassy, V. Natarajan, M. Duchaineau, V. Pascucci, E. Bringa, A. Higginbotham, and B. Hamann. Topologically Clean Distance Fields. *IEEE Transactions on Visualization and Computer Graphics (Proc. of IEEE VIS)*, 13:1432–1439, 2007.

[24] A. Landge, V. Pascucci, A. Gyulassy, J. Bennett, H. Kolla, J. Chen, and T. Bremer. In-situ feature extraction of large scale combustion simulations using segmented merge trees. In *SuperComputing*, 2014.

[25] D. E. Laney, P. Bremer, A. Mascarenhas, P. Miller, and V. Pascucci. Understanding the structure of the turbulent mixing layer in hydrodynamic instabilities. *IEEE Transactions on Visualization and Computer Graphics (Proc. of IEEE VIS)*, 2006.

[26] S. Maadasamy, H. Doraiswamy, and V. Natarajan. A hybrid parallel algorithm for computing and tracking level set topology. In *International Conference on High Performance Computing*, 2012.

[27] D. Morozov and G. Weber. Distributed merge trees. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2013.

[28] D. Morozov and G. Weber. Distributed contour trees. In *Topological Methods in Data Analysis and Visualization III*, 2014.

[29] V. Pascucci and K. Cole-McLaughlin. Parallel computation of the topology of level sets. *Algorithmica*, 2003.

[30] V. Pascucci, K. Cole-McLaughlin, and G. Scorzelli. Multi-resolution computation and presentation of contour trees.

[31] V. Pascucci, G. Scorzelli, P. T. Bremer, and A. Mascarenhas. Robust on-line computation of Reeb graphs: simplicity and speed. *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH)*, 26:58.1–58.9, 2007.

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

[33] J. Singler, P. Sanders, and F. Putze. The Multi-Core Standard Template Library. In *Euro-Par*, 2007.

[34] B. S. Sohn and C. L. Bajaj. Time varying contour topology. *IEEE Transactions on Visualization and Computer Graphics*, 2006.

[35] S. Tarasov and M. Vyali. Construction of contour trees in 3d in o(n log n) steps. In *Proc. of ACM Symposium on Computational Geometry*, 1998.

[36] D. M. Thomas and V. Natarajan. Multiscale symmetry detection in scalar fields by clustering contours. *IEEE Transactions on Visualization and Computer Graphics (Proc. of IEEE VIS)*, 2014.

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

[38] P. Tsigas and Y. Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. In *Conference on Parallel, Distributed and Network-based Processing*, 2003.

[39] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pasucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proc. of ACM Symposium on Computational Geometry*, 1997.

[40] G. Weber, S. E. Dillard, H. Carr, V. Pascucci, and B. Hamann. Topology-controlled volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 2007.

[41] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.