Task-based Augmented Merge Trees with Fibonacci Heaps

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, France

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-core computation of augmented merge trees on triangulations. In contrast to most existing parallel algorithms, our technique computes *augmented* trees. This augmentation is required to enable the full extent of merge tree based applications, including data segmentation. Our approach completely revisits the traditional, sequential merge tree algorithm to re-formulate the computation as a set of independent local tasks based on Fibonacci heaps. This results in superior time performance in practice, in sequential as well as in parallel thanks to the OpenMP task runtime. In the context of augmented contour tree computation, we show that a direct usage of our merge tree procedure also results in superior time performance overall, both in sequential and parallel. We report performance numbers that compare our approach to reference sequential and multi-threaded implementations for the computation of augmented merge and contour trees. These experiments demonstrate the runtime efficiency of our approach as well as its scalability on common workstations. We demonstrate the utility of our approach in data segmentation applications. 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 data analysis techniques [16, 27, 39] 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 merge trees [6, 35, 46], contour trees [5, 7, 13, 49], Reeb graphs [3, 38, 40, 44, 52], or Morse-Smale complexes [14, 24, 41, 56]. 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 [8, 53], feature tracking [47], data-summarization [37, 55], transfer function design for volume rendering [54], similarity estimation [28, 50]. Moreover, their ability to capture the features of interest in scalar data in a generic, robust and multi-scale manner has contributed to their popularity in a variety of applications, including turbulent combustion [6, 23, 31], computational fluid dynamics [19, 29], material sciences [25, 26], chemistry [21], or astrophysics [43, 45, 48], etc.

However, as computational resources and acquisition devices improve, the resolution of the geometrical domains on which scalar fields are defined also increases. 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 for the computation of topological abstractions. 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 merge trees – a fundamental topology-based data structure in scalar field visualization – several algorithms have been proposed for its parallel computation [1, 9, 32]. However, these algorithms only compute *non-augmented* merge trees [7], which only represent the connectivity evolution of the sub-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 merge tree, they do not enable them all. For instance, they do not readily support topology based data segmentation. 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 our knowledge.

This paper addresses this problem by presenting a new algorithm for the efficient computation of augmented merge trees of scalar data on triangulations. Such a tree augmentation makes our output data-structures generic application-wise and enables the full extent of merge tree based applications, including data segmentation. Our approach completely revisits the traditional, sequential merge tree algorithm to re-formulate the computation as a set of local tasks that are as independent as possible and that rely on Fibonacci heaps. This results in a computation with superior time performance in practice, in sequential as well as in parallel on multi-core CPUs with shared memory thanks to the OpenMP task runtime. In the context of augmented contour tree computation, we show that a direct usage of our merge tree computation procedure also results in superior time performance overall, both in sequential and parallel. Extensive experiments on a variety of real-life data sets demonstrate the practical superiority of our approach in terms of time performance in comparison to sequential [15] and parallel [22] reference implementations, both for augmented merge and contour tree computations. 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 merge tree, a tree that contracts connected components of *sub-level sets* to points (formally defined in Sect. 2.1), is closely related to the notion of contour tree [5], which contracts connected components of *level sets* to points on simply connected domains. As shown by Tarasov and Vyali [49] and later generalized by Carr et al. [7] in arbitrary dimension, the contour tree can be efficiently computed by combining with a simple linear-time traversal the merge trees of the input function and of its opposite (called the join and split trees, see Sect. 2.1). Due to this tight relation, merge and contour trees have often been investigated jointly in the computer science literature.

A simple sequential algorithm, based on a union-find data-

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

IEEE Symposium on Large Data Analysis and Visualization 2017
October 1–6, Phoenix, Arizona, USA
978-1-5386-0617-9/17/$31.00 ©2017 IEEE

structure [12], is typically used for merge tree computation [7, 49]. It is both simple to implement, relatively efficient in practice and with optimal time complexity. In particular, this algorithm allows for the computation of both augmented and non-augmented merge trees. An open source reference implementation (*libtourtre* [15]) of this algorithm is provided by Scott Dillard as a component of his contour tree implementation. Chiang et al. [10] presented an output-sensitive approach, based on a new algorithm for the computation of non-augmented merge trees using monotone paths, where the arcs of the merge trees were evaluated by considering monotone paths connecting the critical points of the input scalar field.

Applications of merge trees in data analysis and visualization include multi-scale data segmentation [8], feature tracking [47], data-summarization [37, 55], transfer function design for volume rendering [54], similarity estimation [50]. In particular, the segmentation capabilities of the merge trees, combined with simplification mechanisms inspired by persistent homology [8, 17], enable to generate hierarchies of data segmentations for the extraction of features of interest at multiple scales of importance. These capabilities have been successfully applied in a variety of applications, including for instance turbulent combustion [6], computational fluid dynamics [19, 29], chemistry [21], or astrophysics [43], etc. Note that all of the segmentation applications mentioned above require the *augmented* merge tree as they rely on the identification of the sets of regular vertices mapping to each arc of the merge tree to extract these regions for analysis and visualization purposes.

Among the approaches which addressed the time performance improvement of contour tree computation through shared-memory parallelism, only a few of them rely directly on the original merge tree computation algorithm [7, 49]. This algorithm is then used within partitions of the mesh resulting from a static decomposition on the CPU cores, by either dividing the geometrical domain [36] or the range [22]. This leads in both cases to extra work (with respect to the sequential mono-partition computation) at the partition boundaries when joining results from different partitions. This can also lead to load imbalance among the different partitions [22].

In contrast, most approaches addressing shared-memory parallel contour tree computation actually focused on revisiting the merge tree sub-procedure, as it constitutes the main computational bottleneck overall (see Sect. 6.2). Maadasamy et al. [32] 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* [15], see Tab. 1 in [32]). This only yields eventually speedups between 1.6 and 2.8 with regard to libtourtre [15] on a 8-core CPU [32] (20% and 35% parallel efficiency respectively). 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. [7]. Indeed, from our experience, although the extraction of the critical points of the field is a local operation [2], we found in practice that its overall computation time is often larger than that of the merge tree (or even contour tree) itself. Moreover, this algorithm triggers monotone path computations for each saddle point [10], even if it does not yield branching in the join or split trees (which induces unnecessary computations). 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 merge tree-based data segmentation and does not produce an augmented merge tree. Carr et al. [9] presented a data parallel algorithm following a similar approach. However, their implementation only supports non-augmented trees to the best of our knowledge and experiments have only been documented in 2D. Smirnov et al. [46] described a new data-structure for computing the same information as the merge tree. This structure can be computed in parallel by using an algorithm close to Kruskal's algorithm. However, documented experiments report that this algorithm needs a least 4 threads to be more efficient than its sequential version and has a maximum parallel efficiency of 18.4% compared to this sequential version. Acharya and Natarajan [1] specialized and improved monotone-path based computations for the special case of regular grids. Rosen et al. also presented a hybrid CPU-GPU approach for regular grids [42]. In this work, we focus on triangulations because of the genericity of this representation: any mesh can be decomposed into a valid triangulation and regular grids can be implicitly triangulated with no memory overhead [51].

Morozov and Weber [34, 35] and Landge et al. [30] presented three approaches for merge and 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 makes the following new contributions:

1. **A local algorithm based on Fibonacci heaps:** We present a new algorithm for the computation of augmented merge trees. In contrast to the traditional global algorithm, it is based on local sorting traversals, whose results are progressively merged with the help of a Fibonacci heap. In this context, we also introduce a new criterion for the detection of the saddles which generate branching in the output tree, as well as an efficient procedure to process the output arcs in the vicinity of the root of the tree. Our algorithm is simple to implement and it improves practical time performances on average over a reference implementation [15] of the traditional algorithm [7].

2. **Parallel augmented merge trees:** We show how to leverage the task runtime environment of OpenMP to easily implement a shared-memory parallel version of the above algorithm. Instead of introducing extra work with a static decomposition of the mesh among the threads, the local algorithm based on Fibonacci heaps naturally distributes the merge tree arc computations via independent tasks on the CPU cores. We hence avoid any extra work in parallel, while enabling an efficient dynamic load balancing on the CPU cores thanks to the task runtime. This results in superior time and scaling performances compared to previous multi-threaded algorithms for augmented merge trees [22].

3. **Parallel augmented contour trees:** We show how to use our merge tree procedure for the computation of augmented contour trees. This direct usage of our algorithm also results in superior time and scaling performances compared to previous multi-threaded algorithms for augmented contour trees [22].

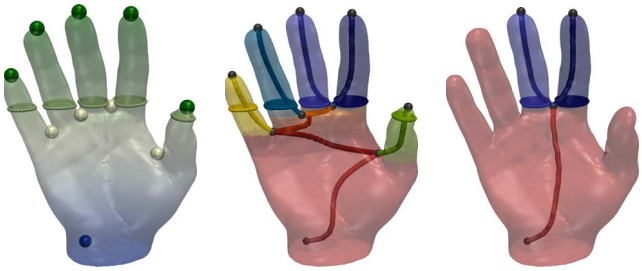
4. **Implementation:** We provide a lightweight VTK-based C++ implementation of our approach for reproduction purposes.

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 $f : \mathcal{M} \to \mathbb{R}$ 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 supports arbitrary dimensions. Adjacency relations on $\mathcal{M}$ can be described in a dimension independent way. The *star* $St(v)$ of a vertex v is the set of



(a) (b) (c)

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) Split tree of f and its corresponding segmentation (arcs and their pre-images by ϕ are shown with the same color). (c) Split tree of f and its corresponding segmentation, simplified according to persistence.

simplices of $\mathscr{M}$ which contain v as a face. The *link* $Lk(v)$ is the set of faces of the simplices of $St(v)$ which do not intersect v.

The scalar field f is provided on the vertices of $\mathscr{M}$ and it is linearly interpolated on the simplices of higher dimension. We will additionally require that the restriction of f to the vertices of $\mathscr{M}$ is injective, which can be easily enforced with a mechanism inspired by simulation of simplicity [18].

The notion of critical point from the smooth setting [33] admits a direct counterpart for PL scalar fields [2]. Let $Lk^-(v)$ be the *lower link* of the vertex v: $Lk^-(v) = \{\sigma \in Lk(v) \mid \forall u \in \sigma : f(u) < f(v)\}$. The *upper link* $Lk^+(v)$ is given by $Lk^+(v) = \{\sigma \in Lk(v) \mid \forall u \in \sigma : f(u) > f(v)\}$. Then, given a vertex v, if its lower (respectively upper) link is empty, v is a local *minimum* (respectively *maximum*). If both $Lk^-(v)$ and $Lk^+(v)$ are simply connected, v is a regular point. Any other configuration is called a *saddle* (white spheres, Fig. 1(a)).

A level-set is defined as the pre-image of an isovalue $i \in \mathbb{R}$ onto $\mathscr{M}$ through f: $f^{-1}(i) = \{p \in \mathscr{M} \mid f(p) = i\}$ (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. Similarly, the notion of *sub-level set*, noted $f^{-1}_{-\infty}(i)$, is defined as the pre-image of the open interval $(-\infty, i)$ onto $\mathscr{M}$ through f: $f^{-1}_{-\infty}(i) = \{p \in \mathscr{M} \mid f(p) < i\}$. Symmetrically, the *sur-level set* $f^{-1}_{+\infty}(i)$ is defined by $f^{-1}_{+\infty}(i) = \{p \in \mathscr{M} \mid f(p) > i\}$. Let $f^{-1}_{-\infty}\big(f(p)\big)_p$ (respectively $f^{-1}_{+\infty}\big(f(p)\big)_p$) be the connected component of sub-level set (respectively sur-level set) of $f(p)$ which contains the point p. The *split tree* $\mathscr{T}^+(f)$ is a 1-dimensional simplicial complex (Fig. 1(b)) defined as the quotient space $\mathscr{T}^+(f) = \mathscr{M}/\sim$ by the equivalence relation $p_1 \sim p_2$:

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

The *join tree*, noted $\mathscr{T}^-(f)$, is defined similarly with regard to an equivalence relation on sub-level set components (instead of sur-level sets). Irrespective of their orientation, the *join* and *split* trees are usually called *merge trees*, and noted $\mathscr{T}(f)$ in the following. The notion of *Reeb graph* [40], noted $\mathscr{R}(f)$, is also defined similarly, with regard to an equivalence relation on level set components (instead of sub-level set components). 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 combining the join and split trees with a linear-time traversal [7,49]. In Fig. 1, since $\mathscr{M}$ is simply connected, the contour tree $\mathscr{C}(f)$ is also the Reeb graph of f. Since f has only one minimum, the split tree $\mathscr{T}^+(f)$ is equivalent to the contour tree $\mathscr{C}(f)$.

Note that f can be decomposed into $f = \psi \circ \phi$ where $\phi : \mathscr{M} \to \mathscr{T}(f)$ maps each point in $\mathscr{M}$ to its equivalence class in $\mathscr{T}(f)$ and where $\psi : \mathscr{T}(f) \to \mathbb{R}$ maps each point in $\mathscr{T}(f)$ to its f value. Since the number of connected components of $f^{-1}_{-\infty}(i)$, $f^{-1}_{+\infty}(i)$ and $f^{-1}(i)$ only changes in the vicinity of a critical point [2, 16, 33], the pre-image by ϕ of any vertex of $\mathscr{T}^-(f)$, $\mathscr{T}^+(f)$ or $\mathscr{R}(f)$ is a critical point of f (blue, white and green spheres in Fig. 1(a)). In particular, the pre-image of vertices of valence 1 necessarily correspond to extrema of f [40]. The pre-image of vertices of higher valence correspond to saddle points which join (respectively split) connected components of sub- (respectively sur-) level sets. Since $f^{-1}_{-\infty}(f(M)) = \mathscr{M}$ for the global maximum M of f, $\phi(M)$ is called the *root* of $\mathscr{T}^-(f)$ and the image by ϕ of any local minimum m is called a *leaf*. Symmetrically, the global minimum of f is the root of $\mathscr{T}^+(f)$ and local maxima of f are its leaves.

Note that the pre-image by ϕ of $\mathscr{T}(f)$ induces a complete partition of $\mathscr{M}$. In particular, the pre-image $\phi^{-1}(\sigma_1)$ of a 1-simplex $\sigma_1 \in \mathscr{T}(f)$ is guaranteed by construction to be connected. This latter property is at the basis of the usage of the merge tree in visualization as a data segmentation tool (Fig. 1(b)) for feature extraction. In practice, ϕ^{-1} is represented explicitly by maintaining, for each 1-simplex $\sigma_1 \in \mathscr{T}(f)$ (i.e. for each arc), the list of regular vertices of $\mathscr{M}$ that map to σ_1. Moreover, since the merge tree 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 $\mathscr{T}(f)$, by iteratively removing its *shortest* arcs connected to leaves. This yields hierarchies of merge trees that are accompanied by hierarchies of data segmentations, that the user can interactively explore in practice (see Fig. 1(c)).

2.2 Overview

An overview of our augmented merge tree computation algorithm is presented in Fig. 2 in the case of the join tree. The purpose of our algorithm, in addition to construct $\mathscr{T}(f)$, is to build the explicit segmentation map ϕ, which maps each vertex $v \in \mathscr{M}$ to $\mathscr{T}(f)$. Our algorithm is expressed as a sequence of procedures, called on each vertex of $\mathscr{M}$. First, given a vertex v, the algorithm checks if v corresponds to a leaf (Fig. 2 left, Sect. 3.1). If this is the case, the second procedure is triggered. For each leaf vertex, the augmented arc connected to it is constructed by a local growth, implemented with a sorted breadth-first search traversal (Fig. 2 middle left, Sect. 3.2). A local growth may continue at a join saddle s, in a third procedure, only if it is the last growth which visited the saddle s (Fig. 2 middle right, Sect. 3.4). To initiate the growth from s efficiently, we rely on the Fibonacci heap data-structure [12,20] in our breadth-first search traversal, which supports constant-time merges of sets of visit candidates. Finally, a fourth procedure is triggered to abbreviate the process if a local growth passing a saddle s happens to be the last active growth. In such a case, all the non-visited vertices above s are guaranteed to map through ϕ to a monotone path from s to the root of the tree (Fig. 2 right, Sect. 3.5). Overall, the time complexity of our algorithm is identical to that of the reference algorithm [7]: $O(|\sigma_0|\ log(|\sigma_0|) + |\sigma_1|\alpha(|\sigma_1|))$, where $|\sigma_i|$ stands for the number of i-simplices in $\mathscr{M}$ and $\alpha()$ is the inverse of the Ackermann function (extremely slow-growing function).

3 Merge tree computation by local arc growth

In this section, we present our algorithm for the computation of augmented merge trees based on local arc growth. Our algorithm consists of a sequence of procedures applied to each vertex, described in each of the following sub-sections. In the remainder,

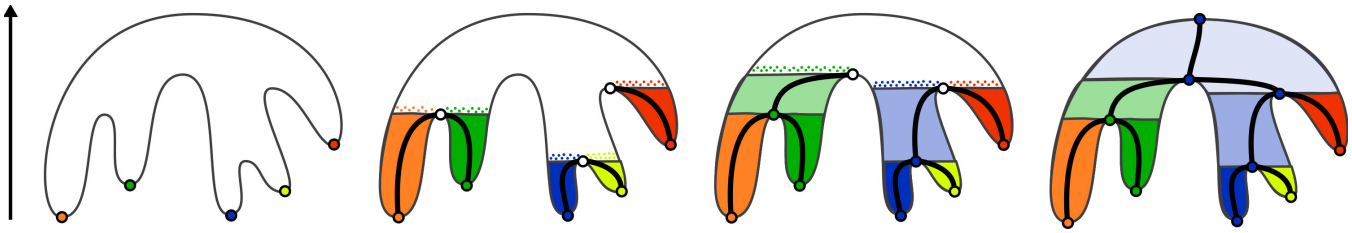


Figure 2: Overview of our augmented merge tree algorithm based Fibonacci heaps (2D toy elevation example). First, the local extrema of f (corresponding to the leaves of $\mathscr{T}(f)$) are extracted (left, Sect. 3.1). Second, the arc σ_m of each extremum m is grown independently along with its segmentation (matching colors, center left, Sect. 3.2). These independent growths are achieved by progressively growing the connected components of level sets created in m, for increasing f values, and by maintaining at each step a priority queue $\mathscr{Q}_m$, implemented with a Fibonacci heap, which stores vertex candidates for the next iteration (illustrated with colored dots). These growths are stopped at merge saddles (white disks, center left, Sect. 3.3). Only the last growth reaching a saddle s is kept active and allowed to continue to grow the saddle's arc σ_s (matching colors, center right, Sect. 3.4). The constant time merge operation of the Fibonacci heap (to initialize the growth at s) enables a highly efficient execution for this step in practice. Last, when only one growth remains active, the tree is completed by simply creating its *trunk*, a monotone sequence of arcs to the root of the tree which links the remaining pending saddles (pale blue region, right, Sect. 3.5). The task-based parallel model allows for a trivial parallelization of this algorithm, where each arc is grown independently, only requiring local synchronizations on merge saddles.

we will illustrate our discussion with the join tree, which tracks connected components of sub-level sets, initiated in local minima.

3.1 Leaf search

First, given a vertex $v \in \mathscr{M}$, its lower link $Lk^-(v)$ is constructed. If it is non-empty, v is not a local minimum and the procedure stops. Otherwise, if it is empty, v is a local minimum, a leaf, and the leaf growth procedure, described in the next sub-section, is called.

3.2 Leaf growth

Given a local minimum m, the arc σ_m of the join tree connected to it is constructed with a procedure that we call *local leaf growth*. The purpose of this procedure is to progressively sweep all contiguous equivalence classes (Sect. 2.1) from m to the saddle s located at the extremity of σ_m. We describe precisely how to detect such a saddle s, and therefore where to stop such a growth, in the next subsection (Sect. 3.3). In other words, this growth procedure will construct the connected component of sub-level set initiated in m, and will make it progressively grow for increasing values of f.

This is achieved by implementing an ordered breadth-first search traversal of the vertices of $\mathscr{M}$ initiated in m. At each step, the neighbors of v which have not already been visited are added to a priority queue $\mathscr{Q}_m$ (if not already present in it) and v is added to σ_m. The purpose of the addition of v to σ_m is to augment this arc with regular vertices, and therefore to store its data segmentation. Next, the following visited vertex v' is chosen as the minimizer of f in $\mathscr{Q}_m$ and the process it iterated until s is reached (Sect. 3.3). At each step of this local growth, since breadth-first search traversals grow connected components, we have the guarantee, when visiting a vertex v, that the set of vertices visited up to this point (added to σ_m) indeed equals to the set of vertices belonging to the connected component of sub-level set of $f(v)$ which contains v, noted $f^{-1}_{+\infty}\big(f(v)\big)_v$ in Sect. 2.1. Therefore, our local leaf growth indeed constructs σ_m (with its segmentation). Also, note that, at each iteration, the set of edges linking the vertices already visited and the vertices currently in the priority queue $\mathscr{Q}_m$ are all crossed by the level set $f^{-1}\big(f(v)\big)$.

The time complexity of this procedure is $O(|\sigma_0|\, log(|\sigma_0|) + |\sigma_1|)$, where $|\sigma_i|$ stands for the number of i-simplices in $\mathscr{M}$.

3.3 Saddle stopping condition

Given a local minimum m, the leaf growth procedure is stopped when reaching the saddle s corresponding to the other extremity of σ_m. We describe in this sub-section how to detect s.

In principle, the saddles of f could be extracted by using the critical point extraction procedure presented in Sect. 2.1, based on a local classification of the link of each vertex. However, such a strategy has two disadvantages. First not all saddles of f necessarily corresponding to branching in $\mathscr{T}^-(f)$ and/or $\mathscr{T}^+(f)$. Thus some unnecessary computation would need to be carried out. Second, we found in practice that even optimized implementations of such a classification [51] tend to be slower than the entire augmented merge tree computation in sequential. Therefore, another strategy should be considered for the sake of performance.

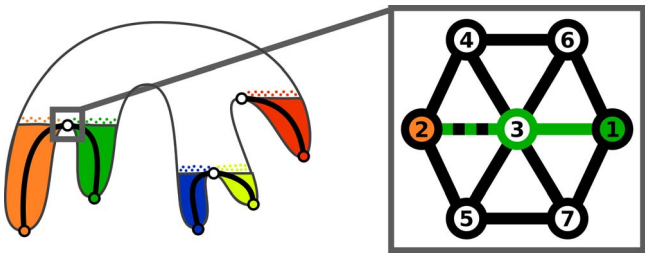


Figure 3: Local merge saddle detection based on arc growth (2D elevation example from Fig. 2). The local growth of the arc σ_m (green) will visit the vertex v' at value 3 after visiting the vertex at value 1 (following the priority queue $\mathscr{Q}_m$). At this point, the neighbors of v' which have not been visited yet by σ_m and which are not in $\mathscr{Q}_m$ yet (dashed green edges) will be added to $\mathscr{Q}_m$. The minimizer v of $\mathscr{Q}_m$ (vertex 2) has a scalar value lower than v'. Hence v' is a merge saddle.

The local leaf growth procedure (Sect. 3.2) visits the vertices of $\mathscr{M}$ with a breadth-first search traversal initiated in m, for increasing f values. At each step, the minimizer v of $\mathscr{Q}_m$ is selected. Assume that $f(v) < f(v')$ where v' was the vertex visited immediately before v. This implies that v belongs to the lower link of v', $Lk^-(v')$. Since v was visited after v', this means that v does not project to σ_m through ϕ. In other words, this implies that v does not belong to the connected component of sub-level set containing m. Therefore, v' happens to be the saddle s that correspond to the extremity of σ_m. Locally (Fig. 3), the local leaf growth entered the star of v' through the connected component of lower link projecting to σ_m and jumped across the saddle v' downwards when selecting the vertex v, which belongs to another connected component of lower link of v'.

Therefore, a sufficient condition to stop a local leaf growth is when the candidate vertex returned by the priority queue has a lower f value than the vertex visited last. In such a case, the last visited vertex is the saddle s which closes the arc σ_m (Fig. 3).

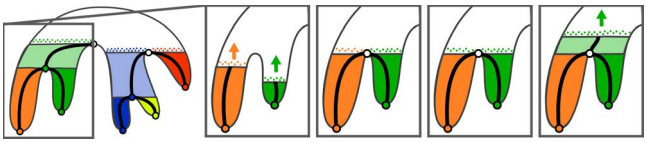


Figure 4: Union of priority queues at a merge saddle (2D elevation example from Fig. 2). Initially, each arc growth maintains its own priority queue (illustrated with colored dots, left inset). When reaching a merge saddle s (second inset), the growths which arrived first in s are marked *terminated*. Only the last one (green) will be allowed to resume the growth from s to construct the arc σ_s (last inset). To continue the propagation of the sub-level set component which contains s, the priority queues of all growths arrived at s need to be merged into only one (third inset) prior to resuming the propagation. If done naively, this operation could yield a quadratic runtime complexity for our approach overall. Since Fibonacci heaps support constant time merges, they guarantee the linearithmic complexity of our overall approach.

3.4 Saddle growth

Up to this point, we described how to construct each arc σ_m connected to a local minimum m, along with its corresponding data segmentation. The remaining arcs can be constructed similarly.

Given a local minimum m, its leaf growth is stopped at the saddle s which corresponds to the extremity of the arc connected to it, σ_m. When reaching s, if *all* vertices of $Lk^-(s)$ have already been visited by some local leaf growth (initiated in m and other minima), we say that the current growth, initiated in m, is the *last* one visiting s.

In such a case, the same breadth-first search traversal can be applied to grow the arc of $\mathscr{T}^-(f)$ initiated in s, noted σ_s. However, in order to represent all the connected components of sub-level set merging in s, such a traversal needs to be initiated with the *union* of the priority queues $\mathscr{Q}_{m_0}, \mathscr{Q}_{m_1}, \dots \mathscr{Q}_{m_n}$ of *all* the arcs merging in s. Such a union models the entire set of candidate vertices for absorption in the sub-level component of s (Fig. 4). Since both the number of minima of f and the size of each priority queue can be linear with the number of vertices in $\mathscr{M}$, if done naively, the union of all priority queues could require $O(|\sigma_0|^2)$ operations overall.

To address this issue, we model each priority queue with a Fibonacci heap [12, 20], which supports the removal of the minimizer of f from $\mathscr{Q}_m$ in $log(|\sigma_0|)$ steps, and performs both the insertion of a new vertex and the merge of two queues in constant time.

Similarly to the traditional merge tree algorithm [7, 49], we maintain a Union-Find data structure [12] to precisely keep track of the arcs which need to be merged at a given saddle s. Each local minimum m is associated with a unique Union-Find element, which is also associated to all regular vertices mapped to σ_m (Sect. 3.2). Also, each Union-Find element is associated to the arc it currently grows. When an arc σ reaches a join saddle s last, the find operation of the Union-Find is called on each vertex of $Lk^-(s)$ to retrieve the set of arcs which merge there and the union operation of the Union-Find is called on each of these to keep track of the merge event.

Therefore, overall, the time complexity of our augmented merge tree computation is $O\big(|\sigma_0|log(|\sigma_0|) + |\sigma_1|\alpha(|\sigma_1|)\big)$, where $\alpha()$ is an extremely slow-growing function (inverse of the Ackermann function). The $|\sigma_1|\alpha(|\sigma_1|)$ term yields from the usage of the Union-Find data structure, while the Fibonacci heap, thanks to its constant time merge support, enables to grow the arcs of the tree in logarithmic time. Thus, the time complexity of our algorithm is exactly equivalent to the traditional, sequential algorithm [7, 49]. However, comparisons to a reference implementation [15] (Sect. 6) show that our approach provides superior performances in practice.

3.5 Trunk growth

Time performance can be further improved by abbreviating the process when only one arc growth is remaining. Initially, if f admits N local minima, N arcs (and N arc growths) need to be created. When the growth of an arc σ reaches a saddle s, if σ is not the last arc reaching s, the growth of σ is switched to the *terminated* state. Therefore, the number of remaining arc growths will decrease from N to 1 along the execution of the algorithm. In particular, the last arc growth will visit all the remaining, unvisited, vertices of $\mathscr{M}$ upwards until the global maximum of f is reached, possibly reaching on the way an arbitrary number of *pending* join saddles, where other arc growths have been stopped and marked terminated (white disks in Fig. 2, third column). Thus, when an arc growth reaches a saddle s, if it is the last active one, we have the guarantee that it will construct in the remaining steps of the algorithm a sequence of arcs which constitutes a monotone path from s up to the root of $\mathscr{T}^-(f)$. We call this sequence the *trunk* of $\mathscr{T}^-(f)$ (Fig. 2).

The trunk of the join tree can be computed faster than through the breadth-first search traversals described in Secs. 3.2 and 3.4. Let s be the join saddle where the trunk starts. Let $S = \{s_0, s_1, \dots s_n\}$ be the sorted set of join saddles that are still pending in the computation (which still have unvisited vertices in their lower link). The trunk is constructed by simply creating arcs that connect two consecutive entries in S. Next, these arcs are augmented by simply traversing the vertices of $\mathscr{M}$ with higher scalar value than $f(s)$ and projecting each unvisited vertex v to the trunk arc that spans it scalar value $f(v)$.

Thus, our algorithm for the construction of the trunk does not use any breadth-first search traversal, as it does not depend on any mesh traversal operation, and it is performed in $O(|\sigma_0|log(|\sigma_0|))$ steps (to maintain regular vertices sorted along the arcs of the trunk). To the best of our knowledge, this algorithmic step is another important novelty of our approach.

4 Task-based parallel merge trees

The previous section introduced a new algorithm based on local arc growths with Fibonacci heaps for the construction of augmented join trees (split trees being constructed with a symmetric procedure). Note that this algorithm enables to process the minima of f concurrently. The same remark goes for the join saddles; however, a join saddle growth can only be started after all of its lower link vertices have been visited. Such an independence and synchronization among the numerous arc growths can be straightforwardly parallelized thanks to the task parallel programming paradigm. Also, note that such a split of the work load does not introduce any supplementary computation. Task-based runtime environments also naturally support dynamic load balancing, each available thread picking its next task among the unprocessed ones. We rely here on OpenMP tasks [4], but other task runtimes (e.g. Intel Threading Building Blocks, Intel Cilk Plus, etc.) could be used as well with a few modifications. In practice, users only need to specify a number of threads among which the tasks will be scheduled. In the remainder, we will detail our task-based implementation for the arc growth step, and also present how we have parallelized the other steps.

At a technical level, our implementation starts with a global sort of all the vertices according to their scalar value in parallel (using the STL parallel sort). This allows all vertex comparisons to be done only by comparing two indices, which is faster in practice than accessing the scalar values, and which does not depend on the scalar type of the input data set.

4.1 Parallel leaf search

For each vertex $v \in \mathscr{M}$, the extraction of its lower link $Lk^-(v)$ is a local operation. This makes this step embarrassingly parallel and enables a straightforward parallelization of the corresponding loop using OpenMP. The size of $Lk^-(v)$ for each vertex is required in the saddle detection step. For this reason, we have to perform the complete leaf search first. Once done, we have the list of extrema from which the leaf growth should be started.

4.2 Leaf growth tasks

Each leaf growth is independent from the others, spreading locally until it finds a saddle. Each leaf growth is thus simply implemented as a task, starting at its previously extracted leaf.

4.3 Saddle stopping condition

The saddle stopping condition presented in Sect. 3.3 can be safely implemented in parallel with tasks. When a vertex v, unvisited so far by the current arc growth, is visited immediately after a vertex v' with $f(v) < f(v')$, then v' is a saddle. To decide if v was indeed not visited by an arc growth associated to the sub-tree of the current arc growth, we use a Union-Find data structure [12] (one Union-Find node per leaf). In particular, we store for each visited vertex the Union-Find representative of its current growth (which was originally created on a minimum). Our Union-Find implementation supports concurrent *find* operations from parallel arc growths (executed simultaneously by distinct tasks). A *find* operation on a Union-Find currently involved in a *union* operation is also possible but safely handled in parallel in our implementation. Since the *find* and *union* operations are local to each Union-Find sub-tree [12], these operations generate only few concurrent accesses. Moreover, these concurrent accesses are efficiently handled since only atomic operations are involved.

When a saddle s is detected, we also have to check if the current growth is the last to reach s as described in Sect. 3.4. In this purpose, each task detecting a saddle s atomically decrements an integer counter, initialized at the size of $Lk^-(s)$ during the leaf search step, by the number of vertices below s coming from the current growth. The task setting this counter to zero is the last reaching this saddle.

4.4 Saddle growth tasks

Once the lower link of a saddle has been completely visited, the last task which reached it merges the priority queues (implemented as Fibonacci heaps), and the corresponding Union-Find data structures, of all tasks *terminated* at this saddle. Such an operation is performed sequentially at each saddle, without any concurrency issue both for the merge of the Fibonacci heaps and for the *union* operations on the Union-Find. The saddle growth starting from this saddle is performed by this last task, with no new task creation. This continuation of tasks is illustrated with shades of the same color in Fig. 2 (in particular for the green and blue tasks). As the number of tasks can only decrease, the detection of the trunk start is straightforward. Each time a task terminates at a saddle, it decrements atomically an integer counter, which tracks the number of remaining tasks. The trunk starts when this number reaches one.

4.5 Parallel trunk growth

During the arc growth step, we keep track of the *pending* saddles (saddles reached by some tasks but for which the lower link has not been completely visited yet). The list of pending saddles enables us to compute the trunk. Once the trunk growth has started, we only focus on the vertices whose scalar value is strictly greater than the lowest pending saddle, as all other vertices have already been processed during the regular arc growth procedure. Next, we create the sequence of arcs connecting pairs of pending saddles in ascending order. At this point, each vertex can be projected independently of the others along one of these arcs. Using the sorted nature of the list of pending saddles, we can use dichotomy for a fast projection. Moreover when we process vertices in the sorted order of their index, a vertex can use the arc of the previous one as a lower bound for its own projection: we just have to check if the current vertex still projects in this arc or in an arc with a higher scalar value. We parallelize this vertex projection procedure using chunks of contiguous vertex indices out of the globally sorted vertex list (chunks are dynamically distributed among the threads). For each chunk, the first vertex is projected on the corresponding arc of the trunk using dichotomy. Each new vertex processed next relies on its predecessor for its own projection. Note that this procedure can visit (and ignore) vertices already processed by the arc growth step.

5 Task based parallel contour trees

As described in Sect. 1.1, an important use case for the merge tree is the computation of the contour tree. Our task-based merge tree procedure can be used quite directly for this purpose. First, it is used for the computation of the join and split trees. Next these two trees can then be efficiently combined into the output contour tree using the linear combination pass of the reference algorithm [7]. We describe in the following the adjustments that are necessary for this.

5.1 Post-processing for contour tree augmentation

Our merge tree procedure segments $\mathscr{M}$ by marking each vertex with the identifier of the arc it projects to through ϕ. In order to produce such a segmentation for the output contour tree (Sect. 5.2), each arc of $\mathscr{T}(f)$ needs to be equipped at this point with the explicit sorted list of vertices which project to it. We reconstruct these explicit sorted lists in parallel. For vertices processed by the arc growth step, we save during each arc growth the visit order local to this growth. During the parallel post-processing of all these vertices, we can safely build (with a linear operation count) the ordered list of regular vertices of each arc in parallel thanks to this local ordering. Regarding the vertices processed by the trunk step, we cannot rely on such a local ordering of the arc. Instead each thread concatenates these vertices within bundles (one bundle per arc for each thread). The bundles of a given arc are then sorted according to their first vertex and concatenated in order to obtain the ordered list of regular vertices for this arc. Hence, the $O(n \log n)$ operation count of the sort only applies to the number of bundles, which is much lower than the number of vertices in practice.

At this point, to use the linear combination pass from the reference algorithm [7], the join tree needs to be augmented with the nodes of the split tree and vice-versa. This step is straightforward since each vertex stores the identifier of the arc it maps to, for both trees.

5.2 Combination

Next, we combine the join and split trees into the output contour tree by adding arcs from both trees leaf after leaf, according to the reference algorithm [7]. Each time we add an arc of one of the two trees, we have to remove the list of regular vertices of this arc from the other tree. As this algorithm is not straightforward to parallelize, we execute it sequentially in our current implementation. As we detail next, this sequential procedure is very fast in practice and it has only a limited impact on the parallel performances.

6 Results

In this section we present performance results obtained on a workstation with two Intel Xeon E5-2630 v3 CPUs (2.4 GHz, 8 CPU cores and 16 hardware threads each) and 64 GB of RAM. By default, parallel executions will thus rely on 32 threads. These results were performed with our VTK/OpenMP based C++ implementation (provided as additional material) using g++ version 5.4.0 and OpenMP 4.0. This implementation (called *Fibonacci Task-based Merge tree*, or FTM) was built as a TTK [51] module. FTM uses TTK's triangulation data structure which supports both tetrahedral meshes and regular grids by performing an implicit triangulation with no memory overhead for the latter. For the Fibonacci heap, we used the implementation available in Boost.

Our tests have been performed using eight data sets from various domains. The first one, Elevation, is a synthetic data set with only one arc in the output tree. Five data sets (Ethane Diol, Boat, Combustion, Enzo and Ftle) result from simulations and two (Foot and Lobster) from acquisition, containing large sections of noise. For the sake of comparison, these data sets have been re-sampled on the same regular grid and have therefore the same number of vertices.

Table 1: Running times (in seconds) of the different FTM steps on a 512^3 grid for the *join tree* (white background) and *split tree* (gray background) computations. $|\mathscr{T}(f)|$ is the number of arcs in the tree.

Data set	$\lvert\mathscr{T}(f)\rvert$	Sequential	Parallel (32 threads on 16 cores)					
		Overall	Sort	Leaf search	Arc growth	Trunk growth	Overall	Speedup
Elevation	1	29.67	1.39	1.17	0	0.20	2.97	9.99
	1	29.65	1.42	1.17	0	0.21	2.99	9.92
Ethane Diol	17	85.83	3.41	1.17	7.52	0.50	12.81	6.70
	19	81.15	3.43	1.16	2.41	0.51	7.74	10.48
Boat	1,596	75.36	3.69	1.17	0.09	0.82	5.99	12.58
	1,673	77.11	3.84	1.17	0.74	0.81	6.77	11.39
Combustion	26,981	87.29	3.46	1.17	3.33	0.78	8.97	9.73
	23,606	84.85	3.51	1.17	0.57	0.79	6.24	13.60
Enzo	96,061	190.27	3.67	1.17	14.05	0.79	19.91	9.56
	115,287	98.28	3.72	1.17	4.14	0.96	10.19	9.64
Ftle	147,748	83.34	3.21	1.17	1.58	0.90	7.07	11.79
	202,865	86.51	3.21	1.18	0.69	0.91	6.20	13.95
Foot	241,841	57.57	2.34	1.18	0.90	0.68	5.33	10.80
	286,654	75.23	2.39	1.18	7.65	0.72	12.17	6.18
Lobster	472,862	122.12	2.49	1.18	3.99	0.89	8.77	13.92
	490,236	61.38	2.51	1.19	6.37	0.76	11.03	5.56

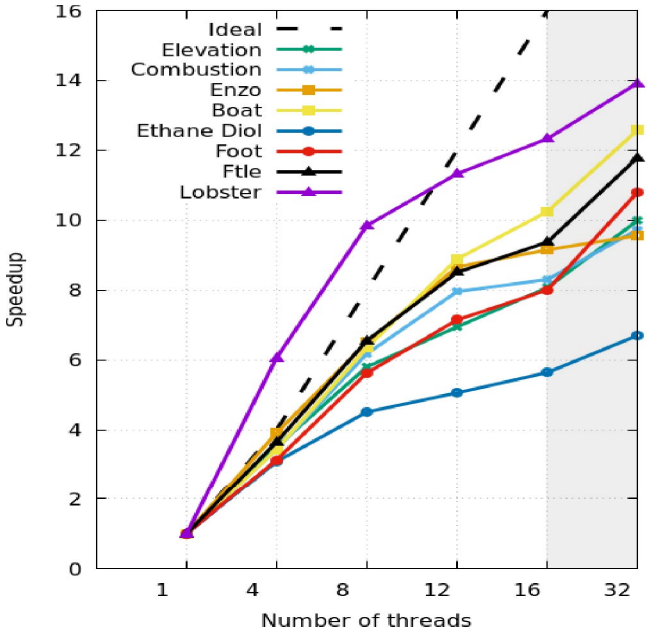


Figure 5: Merge tree scalability on various 512^3 data sets. The gray area denotes using 2 threads per core.

6.1 Merge tree performance results

Table 1 details the execution times and speedups of FTM for the join and the split tree on a 512^3 grid. One can first see that the FTM sequential execution time varies greatly between data sets despite their equal input size. This denotes a sensitivity on the output tree, which is common to most merge tree algorithms. Moving to parallel executions, the leaf search times are the same for all data sets since this step merely depends on the number of input vertices. Compared to sequential leaf search times (not shown here), this step, being embarrassingly parallel, offers very good speedups greater than 14. The key step for parallel performance is the arc growth. On most of our data sets this step is indeed the most-time consuming in parallel, but its times vary in a large range: this will be investigated in Sect. 6.3. The last step is the trunk computation, which takes less than one second. Overall, with a minimum speedup of 5.56, a maximum one of 13.95 and an average speedup of 10.4 on 16 cores, our FTM implementation achieves an average parallel efficiency greater than 64%. These speedups are detailed on the scaling curves of the join tree computation in Fig. 5. The first thing one can notice is the monotonous growth of all curves. This means that more threads always implies faster computations, which enables us to focus on the 32-thread executions. Another interesting point is the Lobster data set presenting speedups greater than the ideal one for 4 and 8 threads. This unexpected but welcome supra-linearity is due to the trunk processing of our algorithm.

Table 2: Process speeds in verts/sec of the different FTM steps on a 512^3 grid for the *join tree* computation in sequential.

Dataset	Sort	Leaf Search	Arcs growth	Trunk growth	Overall
Elevation	12,959,274	7,942,846	0	102,983,615	4,607,907
Ethane Diol	2,518,028	7,943,999	470,459	20,763,762	1,554,494
Boat	2,783,166	7,939,169	419,286	11,083,335	1,720,265
Combustion	2,554,022	7,943,867	431,910	13,420,788	1,496,260
Enzo	2,299,084	7,935,284	328,118	9,882,932	709,224
Ftle	2,563,862	7,925,717	606,633	12,638,528	1,576,565
Foot	4,745,053	7,930,530	416,211	23,287,623	2,331,692
Lobster	5,118,903	7,029,316	709,345	23,578,774	1,076,218

Table 3: FTM execution stability on our 512^3 grid for 10 executions

Data set	Min	Max	Range	Average	Std. dev
Elevation	5.24	5.45	0.21	5.32	0.05
Ethane Diol	20.18	21.29	1.10	20.74	0.33
Boat	12.99	13.77	0.77	13.40	0.25
Combustion	14.91	16.07	1.16	15.74	0.35
Enzo	27.71	29.45	1.73	28.73	0.67
Ftle	15.35	16.20	0.85	15.72	0.26
Foot	19.42	21.08	1.65	20.21	0.50
Lobster	22.37	25.59	3.21	24.36	0.83

As highlighted in Table 2, in sequential mode, the trunk step is able to process vertices much faster than the arc growth step, since no breadth-first search traversal is performed in the trunk step (see Sect. 3.5). However, for a given data set, the size of the trunk highly depends on the order in which leaves have been processed. Since the trunk is detected when only one growth remains active, distinct orders in leaf processing will yield distinct trunks of different sizes, for a given data set. Hence maximizing the size of this trunk minimizes the required amount of computation, especially for data sets like Lobster where the trunk encompasses a large proportion of the domain. Note however, that the leaf ordering which would maximize the size of the trunk cannot be known in advance. In a sequential execution, it is unlikely that the runtime will schedule the tasks on the single thread so that the last task will be the one that corresponds to the greatest possible trunk. Instead, the runtime will likely process each available arc one at a time, leading to a trunk detection at the vicinity of the root. On the contrary, in parallel, it is more likely that the runtime environment will run out of leaves sooner, hence yielding a larger trunk than in sequential, hence leading to increased (possibly supra-linear) speedups.

As the dynamic scheduling of the tasks on the CPU cores may vary from one parallel execution to the next, it follows that the trunk size may also vary across different executions, hence possibly impacting noticeably runtime performances. As shown in Table 3, the range within which the execution times vary is clearly small compared to the average time and the standard deviation shows a very good stability of our approach in practice.

Finally, in order to better evaluate the FTM performance, we compare our approach to two reference implementations, which are, to the best of our knowledge, the only two public implementations supporting augmented trees:

- *libtourtre* (LT) [15], an open source sequential reference implementation of the traditional algorithm [7];
- the open source implementation [51] of the parallel Contour Forest (CF) algorithm [22].

In both implementations, TTK's triangulation data structure [51] is used for mesh traversal. Due to its important memory consumption, we were unable to run CF on the 512^3 data sets on our workstation.

Table 4: *Sequential* join tree computation times (in seconds) and ratios between libtourtre (LT), Contour Forest (CF) and our Fibonacci Task-based Merge tree (FTM), on a 256^3 grid (bold: FTM speedups).

Dataset	LT	CF	FTM	LT / FTM	CF / FTM
Elevation	5.81	7.70	3.57	**1.63**	**2.15**
Ethane Diol	11.59	17.75	7.14	**1.62**	**2.48**
Boat	11.84	17.11	6.93	**1.70**	**2.46**
Combustion	11.65	16.87	8.06	**1.44**	**2.09**
Enzo	14.33	17.99	17.94	0.79	**1.00**
Ftle	11.32	15.62	7.15	**1.58**	**2.18**
Foot	9.45	12.72	5.94	**1.59**	**2.14**
Lobster	11.65	14.80	13.99	0.83	**1.05**

Table 5: *Parallel* join tree computation times (in seconds) and ratios between libtourtre (LT), Contour Forest (CF) and our Fibonacci Task-based Merge tree (FTM), on a 256^3 grid.

Dataset	LT	CF	FTM	LT / FTM	CF / FTM
Elevation	5.00	2.33	0.43	11.63	5.42
Ethane Diol	8.95	4.54	1.33	6.73	3.41
Boat	8.24	4.40	0.69	11.94	6.38
Combustion	7.96	5.82	0.94	8.47	6.19
Enzo	12.18	8.92	1.98	6.15	4.51
Ftle	8.19	4.98	1.04	7.88	4.79
Foot	7.60	6.94	1.27	5.98	5.46
Lobster	8.40	9.02	2.40	3.50	3.76

Table 6: Contour tree computation times (in seconds) with FTM-CT on the 512^3 grid. The construction of the merge trees (join plus split) is reported under the MT column. The post-processing step needed for the contour tree is in the Post-MT column and the sequential combination of these trees in the Combine column.

		Sequential	Parallel (32 threads on 16 cores)					
Dataset	$\lvert\mathscr{T}(f)\rvert$	Overall	Sort	MT	Post-MT	Combine	Overall	Speedup
Elevation	1	52.54	1.37	2.92	0.98	0	5.87	8.96
Ethane Diol	36	139.05	3.45	13.51	1.72	3.26	22.52	6.17
Boat	3,269	140.06	3.60	5.16	1.40	4.13	14.90	9.40
Combustion	50,587	154.6	3.31	8.10	1.15	3.73	16.88	9.16
Enzo	211,347	263.60	3.56	23.17	1.36	4.42	33.10	7.96
Ftle	350,603	141.17	3.24	7.14	1.45	4.45	16.85	8.38
Foot	528,495	129.17	2.38	13.21	1.51	4.75	21.51	6.01
Lobster	963,069	189.93	2.04	15.64	2.68	5.49	26.62	7.13

Table 7: *Sequential* contour tree computation times (in seconds) and ratios between libtourtre (LT), Contour Forest (CF) and our Fibonacci Task-based Merge tree adapted for Contour Tree (FTM-CT), on a 256^3 grid (bold: FTM-CT speedups).

Dataset	LT	CF	FTM-CT	LT / FTM-CT	CF / FTM-CT
Elevation	10.84	8.15	6.53	**1.66**	**1.25**
Ethane Diol	21.54	17.73	12.37	**1.74**	**1.43**
Boat	21.10	16.63	11.99	**1.76**	**1.39**
Combustion	21.52	16.92	13.57	**1.59**	**1.25**
Enzo	27.79	19.71	25.83	**1.08**	0.76
Ftle	23.05	15.89	14.38	**1.60**	**1.11**
Foot	19.24	13.41	14.30	**1.35**	0.94
Lobster	23.39	51.32	22.75	**1.03**	**2.26**

Thus, we have created a smaller grid (256^3 vertices) with downsampled versions of the scalar fields used previously. For the first step of this comparison we are interested in the sequential execution. The corresponding results are reported in Table 4. Our sequential implementation is about twice faster than Contour Forests and more than one and half time faster than libtourtre for most data sets. This is due to the faster processing speed of our trunk step. The parallel results for the merge tree implementation are presented in Table 5. The sequential libtourtre implementation starts by sorting all the vertices, then computes the tree. Using a parallel sort instead of the serial one is straightforward. Thus, we used this naive parallelization of LT in the results reported in Table 5 with 32 threads. As for Contour Forest we report the best time obtained on the workstation, which is not necessarily with 32 threads. Indeed, as detailed in [22] increasing the number of threads in CF can result in extra work due to additional redundant computations. This can lead to greater computation times, especially on noisy data sets. The optimal number of threads for CF has thus to be chosen carefully. On the contrary, FTM always benefits from the maximum number of hardware threads. In the end, FTM largely outperforms the two other implementations for all data sets: libtourtre by a factor 7.8 (in average) and Contour Forest by a factor 5.0 (in average). We here emphasize that the two main performance bottlenecks of CF in parallel, namely extra work and load imbalance among the threads, do not apply to FTM thanks to the arc growth algorithm and to the dynamic task scheduling.

6.2 Contour tree performance results

The performance results obtained using our merge tree implementation adapted for Contour Tree (Sect. 5, named hereafter FTM-CT) on the 512^3 grid are presented Table 6. We recall that the combination of the two merge trees is performed sequentially. These results still show good speedups varying between 6.01 and 9.40. The average speedup is 7.9 leading to an average parallel efficiency of 49%.

As the merge tree in Sect. 6.1, we compare our implementation with libtourtre (LT) and Contour Forest (CF) using the smaller (256^3 vertices) grid. The sequential comparison is shown Table 7. As previously, our sequential implementation is faster for most data sets, improving in average libtourtre by a factor 1.5 and Contour Forest by a factor 1.3. For the parallel comparison (see Table 8), we have created a naive parallel implementation of libtourtre by sorting in parallel using 32 threads and by computing the join and split trees in parallel (with 2 threads). Here again for Contour Forest, we report the best time on the workstation, which can be obtained with less than 32 threads (due to possible extra work in parallel). Despite our straightforward contour tree computation based on FTM, which includes a sequential combination, we clearly outperform the two other implementations for all data sets: libtourtre with a factor of 4.7 (on average) and Contour Forest with a factor of 2.7 (on average).

6.3 Limitations

In order to understand the limitations of the our approach, we first detail the arc growth step in Fig. 6 which presents the number of remaining tasks through time, focusing on the part where this number of tasks becomes lower than the number of threads (32). From the time the curve is lower than 32, until it reaches 1, our arc growth step has less remaining tasks than available threads and does not thus fully exploit the parallel compute power of our CPUs. Depending on the data set, this suboptimal section can have different proportions: 30% for Foot, but 100% for Ethane Diol. As we launch one task by leaf and this latter data set has only 8 leaves, we can not expect this arc growth step to have a speedup greater than 8. More generally, depending on the data set and the number of cores, this suboptimal

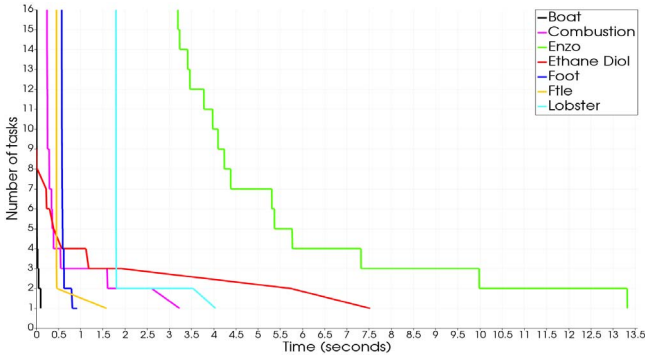


Figure 6: Number of remaining tasks throughout time. This chart is cropped at 32 to highlight the suboptimal section.

Table 8: *Parallel* contour tree computation times (in seconds) and ratios between libtourtre (LT), Contour Forest (CF) and our Fibonacci Task-based Merge tree for Contour Tree (FTM-CT), on a 256^3 grid.

Dataset	LT	CF	FTM-CT	LT / FTM-CT	CF / FTM-CT
Elevation	5.15	2.53	0.83	6.20	3.04
Ethane Diol	12.65	5.39	2.27	5.57	2.37
Boat	12.95	4.97	1.46	8.87	3.40
Combustion	10.59	6.89	1.95	5.43	3.53
Enzo	14.82	11.72	4.37	3.39	2.68
Ftle	12.51	6.28	2.80	4.47	2.24
Foot	10.85	8.30	4.52	2.40	1.84
Lobster	13.05	17.69	8.14	1.60	2.17

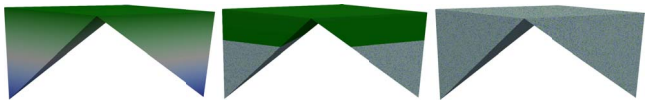


Figure 7: Worst case data set with the initial scalar field (left), with 50% of added randomness (middle), and with 100% of added randomness (right). The color map goes from blue (low) to green (high).

section can limit our parallel speedups for the arc growth step. The larger the number of cores, the sooner this step becomes suboptimal.

In order to study this effect further, we have created a worst case data set composed of only two large arcs as illustrated on the left of Fig. 7. As expected, the speedup of the arc growth step on this data set does not exceed 2, even when using 32 threads (results not shown). Then we randomize this worst case data set gradually, starting by the leaf side as illustrated in Fig. 7 and report the corresponding computation times with 2 and 32 threads in Fig. 8. As the random part progresses (from 10 to 40%), the 32-thread execution takes less time while the 2-thread execution takes more. This is due to the faster processing of the leaf growths associated with random noise with 32 threads. When the random part becomes too large, the execution time increases for both 2 and 32 threads, probably since most of our tasks are now too small. Fortunately, in practice such a worst case and such a large random section are unlikely to appear in real life acquired or simulated data sets.

Finally, a last limitation of our current approach lies in the sequential combination of the contour tree computation. This combination takes a minor part in the overall computation in sequential (about 2% of the total time). However when using a high number of cores in parallel, this sequential step can limit the overall parallel speedups according to Amdahl's law. As a future work, an efficient parallel combination could lead to even better speedups than the ones presented in Table 6.

7 Application

The merge tree is a well known tool for data segmentation used in various applications. It is especially used in the medical domain [8]

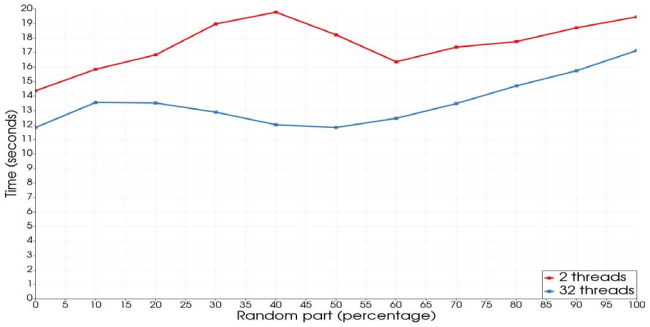


Figure 8: FTM computation time for 2 and 32 threads on our worst case data set as the random part progresses form 0 to 100%.

(a) (b) (c)

Figure 9: The Foot data set is a 3D scan of a human foot on which the scalar field is the density. We use the split tree segmentation to extract bones. (a) One contour corresponding to the skin of the foot. (b) The different bones highlighted using the segmentation of the deepest arcs of the tree. (c) Using topological simplification enables us to identify bones belonging to a same finger.

as illustrated by Fig. 9 which shows a 3D scan of a human foot. The scalar field is the matter density, different densities corresponding to different tissues. The skeleton is easy to detect as it corresponds to the highest density. We can extract the corresponding regions using the segmentation of deepest arcs of the split tree (the arcs adjacent to the leaves) as shown in Fig. 9(b). By using topological simplification we can merge regions of interest to identify bones belonging to the same toe as illustrated by Fig. 9(c). Thanks to our approach this processing can be done in a handful of seconds, even for 512^3 grids, which greatly improves interactivity in visual exploration tasks.

8 Conclusion

In this paper, we have presented a new algorithm to compute the augmented merge tree on shared memory multi-core architectures. This new approach makes use of the Fibonacci heaps to completely revisit the traditional algorithm and compute the merge tree using independent local growths which can be expressed using tasks. This work is well suited for both regular grids and unstructured meshes. We also provided a lightweight generic VTK-based C++ reference implementation of our approach, based on the OpenMP task runtime. This implementation is the fastest to our knowledge to compute this topological data structure in augmented mode, both sequentially and in parallel. Moreover, using our implementation to compute the contour tree gives competitive results, clearly outperforming libtourtre and Contour Forest in all our test cases.

As future work, we plan to improve our parallel performance in four different ways. First, we would like to investigate improvements to our arc growth step, whose scalability is currently bounded by the number of leaves in the output tree (Sect. 6.3). Such improvements could be beneficial for machines with a large number of cores, such as the Intel Xeon Phi. Second, we plan to improve our parallel contour tree performances on multi-core architectures thanks to a parallel combination algorithm. Third, while our efforts focused so far on time efficiency, we would also like to further improve the memory complexity of our implementation, to be able to address larger data-sets; for 1024^3 grids, only topologically simple scalar fields are currently handled by our implementation on our test hardware (64 GB of RAM, Sect. 6). A fourth interesting research direction would be to study the relevance of our approach for *in-situ* visualization, where the analysis code is executed in parallel and in synergy with the simulation code generating the data.

Acknowledgments

This work is partially supported by the Bpifrance grant "AVIDO" (Programme d'Investissements d'Avenir, reference P112017-2661376/DOS0021427) and by the French National Association for Research and Technology (ANRT), in the framework of the LIP6 - Kitware SAS CIFRE partnership reference 2015/1039. The authors would like to thank the anonymous reviewers for their thoughtful remarks and suggestions.

REFERENCES

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

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

[3] S. Biasotti, D. Giorgio, M. Spagnuolo, and B. Falcidieno. Reeb graphs for shape analysis and applications. *TCS*, 2008.

[4] O. A. R. Board. OpenMP Application Program Interface, V 4.0, 2013.

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

[6] 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 TVCG*, 2011.

[7] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. In *Symposium on Discrete Algorithms*, 2000.

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

[9] H. Carr, G. H. Weber, C. M. Sewell, and J. P. Ahrens. Parallel peak pruning for scalable smp contour tree computation. In *Proc. of IEEE Large Data Analysis and Visualization*, 2016.

[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 *SoCG*, 2003.

[12] T. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.

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

[14] 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.

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

[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*, 2002.

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

[19] G. Favelier, C. Gueunet, and J. Tierny. Visualizing ensembles of viscous fingers. In *IEEE SciVis Contest*, 2016.

[20] M. Fredman and R. Tarjan. Fibonacci Heaps and their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 1987.

[21] D. Guenther, R. Alvarez-Boto, J. Contreras-Garcia, J.-P. Piquemal, and J. Tierny. Characterizing molecular interactions in chemical systems. *IEEE Trans. on Vis. and Comp. Graph.*, 2014.

[22] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Contour Forests: Fast Multi-threaded Augmented Contour Trees. In *Proc. of IEEE Large Data Analysis and Visualization*, 2016.

[23] A. Gyulassy, P. Bremer, R. Grout, H. Kolla, J. Chen, and V. Pascucci. Stability of dissipation elements: A case study in combustion. *Comp. Graph. For.*, 2014.

[24] A. Gyulassy, P.-T. Bremer, B. Hamann, and P. Pascucci. A practical approach to Morse-Smale complex computation: scalabity and generality. *IEEE Trans. on Vis. and Comp. Graph.*, pp. 1619–1626, 2008.

[25] 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 Trans. on Vis. and Comp. Graph.*, 2015.

[26] A. Gyulassy, V. Natarajan, M. Duchaineau, V. Pascucci, E. Bringa, A. Higginbotham, and B. Hamann. Topologically Clean Distance Fields. *IEEE Trans. on Vis. and Comp. Graph.*, 13:1432–1439, 2007.

[27] C. Heine, H. Leitte, M. Hlawitschka, F. Iuricich, L. De Floriani, G. Scheuermann, H. Hagen, and C. Garth. A survey of topology-based methods in visualization. *Comp. Graph. For.*, 2016.

[28] M. Hilaga, Y. Shinagawa, T. Kohmura, and T. L. Kunii. Topology matching for fully automatic similarity estimation of 3D shapes. In *Proc. of ACM SIGGRAPH*, 2001.

[29] J. Kasten, J. Reininghaus, I. Hotz, and H. Hege. Two-dimensional time-dependent vortex regions based on the acceleration magnitude. *IEEE Trans. on Vis. and Comp. Graph.*, 2011.

[30] 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.

[31] D. E. Laney, P. Bremer, A. Mascarenhas, P. Miller, and V. Pascucci. Understanding the structure of the turbulent mixing layer in hydrodynamic instabilities. *IEEE Trans. on Vis. and Comp. Graph.*, 2006.

[32] 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.

[33] J. Milnor. *Morse Theory*. Princeton U. Press, 1963.

[34] D. Morozov and G. Weber. Distributed contour trees. In *Topological Methods in Data Analysis and Visualization III*, 2013.

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

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

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

[38] V. Pascucci, G. Scorzelli, P. T. Bremer, and A. Mascarenhas. Robust on-line computation of Reeb graphs: simplicity and speed. *ACM Trans. on Graph.*, 2007.

[39] V. Pascucci, X. Tricoche, H. Hagen, and J. Tierny. *Topological Data Analysis and Visualization: Theory, Algorithms and Applications*. Springer, 2010.

[40] 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.

[41] V. Robins, P. Wood, and A. Sheppard. Theory and algorithms for constructing discrete morse complexes from grayscale digital images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2011.

[42] P. Rosen, J. Tu, and L. Piegl. A hybrid solution to calculating augmented join trees of 2d scalar fields in parallel. In *CAD Conference and Exhibition (accepted)*, 2017.

[43] P. Rosen, B. Wang, A. Seth, B. Mills, A. Ginsburg, J. Kamenetzky, J. Kern, and C. R. Johnson. Using contour trees in the analysis and visualization of radio astronomy data cubes. Technical report, University of South Florida, 2017.

[44] Y. Shinagawa, T. Kunii, and Y. L. Kergosien. Surface coding based on morse theory. *IEEE Computer Graphics and Applications*, 1991.

[45] N. Shivashankar, P. Pranav, V. Natarajan, R. van de Weygaert, E. P. Bos, and S. Rieder. Felix: A topology based framework for visual exploration of cosmic filaments. *IEEE TVCG*, 2016.

[46] D. Smirnov and D. Morozov. Triplet Merge Trees. In *TopoInVis*, 2017.

[47] B. S. Sohn and C. L. Bajaj. Time varying contour topology. *IEEE Trans. on Vis. and Comp. Graph.*, 2006.

[48] T. Sousbie. The persistent cosmic web and its filamentary structure: Theory and implementations. *Royal Astronomical Society*, 2011.

[49] S. Tarasov and M. Vyali. Construction of contour trees in 3d in o(n log n) steps. In *SoCG*, 1998.

[50] D. M. Thomas and V. Natarajan. Multiscale symmetry detection in scalar fields by clustering contours. *IEEE TVCG*, 2014.

[51] J. Tierny, G. Favelier, J. A. Levine, C. Gueunet, and M. Michaux. The Topology ToolKit. *IEEE TVCG (Proc. of IEEE VIS)*, 2017. `https://topology-tool-kit.github.io/`.

[52] J. Tierny, A. Gyulassy, E. Simon, and V. Pascucci. Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees. *IEEE Trans. on Vis. and Comp. Graph.*, 15:1177–1184, 2009.

[53] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pasucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *SoCG*.

[54] G. Weber, S. E. Dillard, H. Carr, V. Pascucci, and B. Hamann. Topology-controlled volume rendering. *IEEE TVCG*, 2007.

[55] G. H. Weber, P. Bremer, and V. Pascucci. Topological Landscapes: A Terrain Metaphor for Scientific Data. *IEEE TVCG*, 2007.

[56] K. Weiss, F. Iuricich, R. Fellegara, and L. D. Floriani. A primal/dual representation for discrete morse complexes on tetrahedral meshes. *Comp. Graph. For.*, 2013.