Title

ExTreeM: Scalable Augmented Merge Tree Computation via Extremum Graphs

Journal

IEEE Transactions on Visualization and Computer Graphics, 30(1)

ISSN

1077-2626

Authors

Lukasczyk, Jonas
Will, Michael
Wetzels, Florian
et al.

Publication Date

2024

DOI

10.1109/tvcg.2023.3326526

Peer reviewed

ExTreeM: Scalable Augmented Merge Tree Computation via Extremum Graphs

Jonas Lukasczyk, Michael Will, Florian Wetzels, Gunther H. Weber, and Christoph Garth

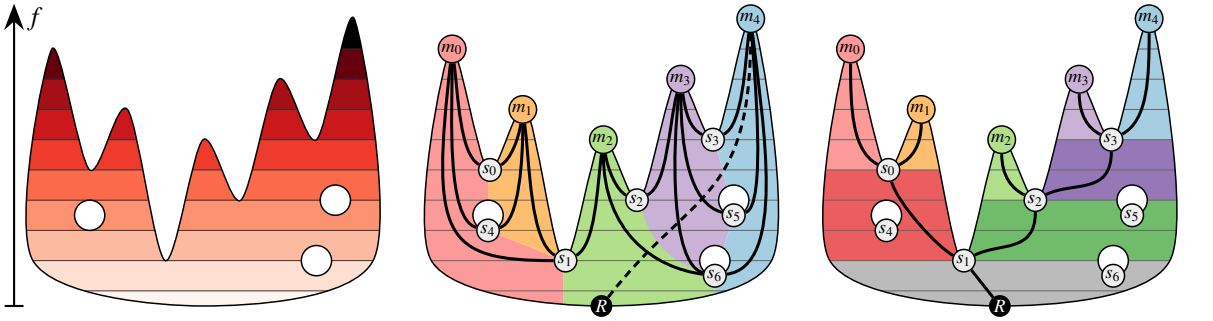


Fig. 1: The input of ExTreeM is a piecewise-linear scalar field $f : \mathcal{K} \to \mathbb{R}$ defined on a connected simplicial complex $\mathcal{K}$ (left). First, ExTreeM derives the descending manifold $d : \mathcal{V}(\mathcal{K}) \to \mathcal{M}(\mathcal{K})$ that assigns to each vertex of $\mathcal{K}$ the maximum that would be reached by following the gradient of f along the steepest ascent on $\mathcal{K}$ (middle; background color) via path compression. Then the algorithm uses d to efficiently identify split saddles of f (middle; gray nodes). Next, ExTreeM derives the extremum graph $\mathcal{G}$ of f by connecting saddles and maxima according to the descending manifold of the local saddle neighborhood (middle; black lines) and the global minimum and maximum (middle; dashed line). Then ExTreeM derives the unaugmented merge tree $\mathcal{T}$ of f on $\mathcal{G}$ instead of $\mathcal{K}$ (right; black edges). The key aspect of the proposed approach is that the merge trees of f on $\mathcal{K}$ and $\mathcal{G}$ are equivalent, but the computation on $\mathcal{G}$ is much faster than on $\mathcal{K}$ since $\mathcal{G}$ contains, in general, several orders of magnitude fewer simplices than $\mathcal{K}$. Moreover, we describe a specialized merge tree algorithm for extremum graphs that further accelerates computation. Finally, in a post-processing procedure, ExTreeM derives a merge tree augmentation that assigns each vertex of $\mathcal{K}$ to an edge of $\mathcal{T}$ (right; background).

Abstract—Over the last decade merge trees have been proven to support a plethora of visualization and analysis tasks since they effectively abstract complex datasets. This paper describes the ExTreeM-Algorithm: a scalable algorithm for the computation of merge trees via extremum graphs. The core idea of ExTreeM is to first derive the extremum graph $\mathcal{G}$ of an input scalar field f defined on a cell complex $\mathcal{K}$, and subsequently compute the unaugmented merge tree of f on $\mathcal{G}$ instead of $\mathcal{K}$; which are equivalent. Any merge tree algorithm can be carried out significantly faster on $\mathcal{G}$, since $\mathcal{K}$ in general contains substantially more cells than $\mathcal{G}$. To further speed up computation, ExTreeM includes a tailored procedure to derive merge trees of extremum graphs. The computation of the fully augmented merge tree, i.e., a merge tree domain segmentation of $\mathcal{K}$, can then be performed in an optional post-processing step. All steps of ExTreeM consist of procedures with high parallel efficiency, and we provide a formal proof of its correctness. Our experiments, performed on publicly available datasets, report a speedup of up to one order of magnitude over the state-of-the-art algorithms included in the TTK and VTK-m software libraries, while also requiring significantly less memory and exhibiting excellent scaling behavior.

Index Terms—Scalar field topology, merge trees, persistence pairs, high performance computing.

1 Introduction

Merge trees are fundamental data abstractions of scalar field topology that record at which scalar values superlevel set components appear and merge (Fig. 1; right). Due to their ability to capture the inherent structure of scalar fields, and to effectively characterize features, they are the basis for many advanced data analysis and visualization approaches [21] (Fig. 2). Several efficient algorithms have been developed for their computation (Sec. 3), yet for large datasets these approaches still require a significant amount of time. To enable interactive visualization and analysis for ever-growing data sizes, the merge tree computation needs to be further and further accelerated.

- *Jonas Lukasczyk, Michael Will, Florian Wetzels, and Christoph Garth are with RPTU Kaiserslautern-Landau.*
  E-mail: {lukasczyk,mswill,wetzels,garth}@rptu.de
- *Gunther H. Weber is with the Lawrence Berkeley National Lab.*
  E-mail: ghweber@lbl.gov

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

In this paper, we describe ExTreeM: a scalable merge tree algorithm with high parallel efficiency and low memory footprint (Sec. 4). The core idea of ExTreeM is to first derive the extremum graph $\mathcal{G}$ of the input scalar field f defined on an input complex $\mathcal{K}$, and then to compute the merge tree of f on $\mathcal{G}$ instead of $\mathcal{K}$. The key aspect of this concept is that the merge tree of f on $\mathcal{G}$ is equivalent to the one of f on $\mathcal{K}$; which we prove formally (Sec. 5). Since $\mathcal{G}$ can be up to multiple orders of magnitude smaller than $\mathcal{K}$, the merge tree computation can be performed much faster on $\mathcal{G}$ than on $\mathcal{K}$. In this generic approach of ExTreeM, any algorithm can be used to compute the merge tree of f on $\mathcal{G}$. To further boost performance, we also describe a specialized algorithm to compute merge trees on extremum graphs. The last step of ExTreeM then uses the derived merge tree to compute the corresponding segmentation of $\mathcal{K}$.

We compare ExTreeM to two state-of-the-art merge tree algorithms available in the *Topology ToolKit (TTK)* [19, 28, 40] and the *Visualization Toolkit (VTK-m)* [7, 29, 37]. In all our experiments ExTreeM outperformed the competitors; in the best case even by up to an order of magnitude (Sec. 6).

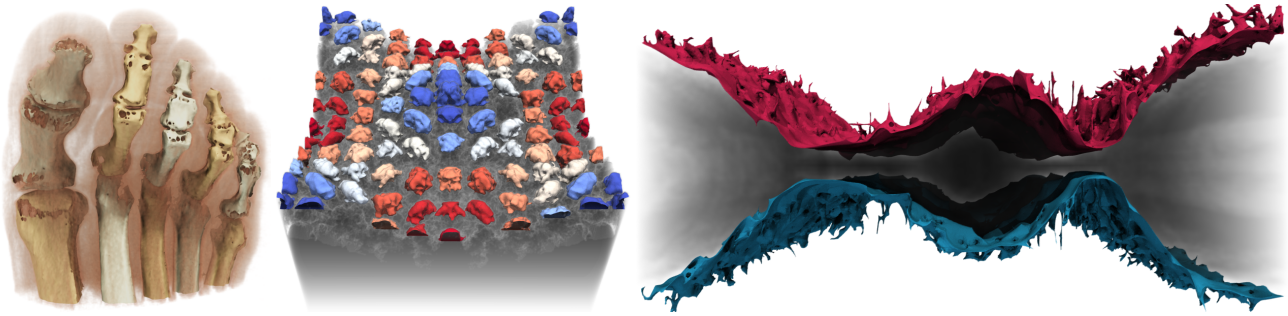


Fig. 2: Merge tree leaf segmentations for the ctBones [41], Richtmyer-Meshkov instability [10], and the magnetic reconnection [20] datasets, which characterize bones of the foot, hills of higher entropy, and high-density boundaries, respectively. Generating such visualizations requires the computation of the merge trees, which with respect to each dataset takes $1.74s$, $196.00s$, and $159.71s$ with *Parallel Peak Pruning* (VTK-m), whereas ExTreeM only requires $0.50s$, $39.15s$, and $14.26s$ (which is 3.5, 5.0, and 11.2 times faster while also requiring only half as much memory). Thus, ExTreeM is one step towards performing interactive topological data analysis and visualization of large datasets on common workstations.

The main aspects of this paper are:

1. an overview of common merge tree algorithms (Sec. 3);
2. our main contribution: a novel merge tree algorithm with high parallel efficiency (Sec. 4);
3. a lookup-table-based approach for critical point classification on regular grids that can be utilized by existing algorithms (Sec. 4.2);
4. a formal proof of correctness of the proposed approach (Sec. 5),
5. a comparison between three merge tree algorithms (Sec. 6); and
6. an implementation of the proposed algorithm in TTK.

2 Background

This section provides the theoretical background of the proposed approach and introduces the notations used throughout the manuscript. For a comprehensive introduction to computational topology, we refer the reader to the textbook of Edelsbrunner and Harer [13].

2.1 Scalar Fields

The input of our approach is a piecewise-linear (PL) *scalar field* $f : \mathcal{K} \to \mathbb{R}$, where real-valued data is given at the vertices of a connected simplicial complex $\mathcal{K}$, and values on edges are linearly interpolated. We denote the vertices (0-simplices) and edges (1-simplices) of the complex $\mathcal{K}$ with $\mathcal{V}(\mathcal{K})$ and $\mathcal{E}(\mathcal{K})$, respectively. Neighbor vertices of a vertex v are denoted by $\mathcal{N}(v,\mathcal{K}) = \{u \in \mathcal{V}(\mathcal{K}) \ : \ \langle v,u\rangle \in \mathcal{E}(\mathcal{K})\}$. $\mathcal{K}$ does not need to be simply connected, but we require that f is injective on the vertices of $\mathcal{K}$, which can always be enforced by applying a variant of *Simulation of Simplicity* [15].

2.2 Critical Points

The *superlevel set* $f^{-1}_{+\infty}(\ell,\mathcal{K})$ for a *level* $\ell \in \mathbb{R}$ consists of all points of the underlying space of $\mathcal{K}$ whose scalar values are greater or equal to ℓ, i.e., $f^{-1}_{+\infty}(\ell,\mathcal{K}) = \{p \in |\mathcal{K}| \ : \ f(p) \geq \ell\}$. As ℓ continuously decreases, the topology of $f^{-1}_{+\infty}(\ell,\mathcal{K})$ changes at certain vertices of $\mathcal{K}$, called the *critical points* of $\mathcal{K}$ for f. Specifically, new components appear at maxima and merge at saddles. The same can be observed symmetrically for *sublevel sets* $f^{-1}_{-\infty}(\ell,\mathcal{K}) = \{p \in |\mathcal{K}| \ : \ f(p) \leq \ell\}$ while continuously increasing the level ℓ.

Banchoff [3] introduced a PL characterization of critical points based on their local neighborhood, which is represented by the *upper* and *lower link* of a vertex. The *upper link* $Lk^+(v)$ of a vertex $v \in \mathcal{V}(\mathcal{K})$ is defined as all simplices of its link which exceed the scalar value of v: $Lk^+(v) = \{\sigma \in Lk(v) \mid \forall u \in |\sigma| : f(u) > f(v)\}$ (blue vertices and edges in Fig. 3). Symmetrically, the *lower link* is defined as $Lk^-(v) = \{\sigma \in Lk(v) \mid \forall u \in |\sigma| : f(u) < f(v)\}$ (red vertices and edges in Fig. 3). If both the lower and upper link of a vertex v are simply connected, then v is called a *regular* point, i.e., not critical (Fig. 3a). Otherwise, if the lower or upper link is empty, then v is called a *minimum* or *maximum*, respectively (Fig. 3b-c). If the lower or upper link consists of more than one connected component, then v is called a *join saddle* or *split saddle*, respectively (Fig. 3d). In the context of merge trees, we further need to distinguish between saddles that actually merge distinct superlevel or sublevel set components, called *merge saddles* (Fig. 3e), and saddles that only change the genus of the components, further referred to as *genus saddles* (Fig. 3f). It is important to note that only merge saddles appear in the merge tree, but just based on the local neighborhood of a saddle it is not possible to distinguish between merge and genus saddles.

2.3 Ascending and Descending Manifolds

Let $\mathcal{M}(\mathcal{K}) \subset \mathcal{V}(\mathcal{K})$ be the set of maxima of $\mathcal{K}$ for f. Then the *descending manifold* is a map $d : \mathcal{V}(\mathcal{K}) \to \mathcal{M}(\mathcal{K})$ that assigns to each vertex $v \in \mathcal{V}(\mathcal{K})$ the maximum $m \in \mathcal{M}(\mathcal{K})$ that would be reached by following the path starting at v along the steepest ascent on $\mathcal{K}$. This path is a sequence of n vertices $(v = v_1, v_2, ..., v_n = m)$ where $v_{i+1} = \text{argmax}_{u \in \mathcal{N}(v_i,\mathcal{K})} f(u)$. The *ascending manifold* is defined symmetrically for minima reached by following the path starting at v along the steepest descent. These paths are unambiguous since f is injective.

2.4 Extremum Graphs

We define the extremum graph as a one-dimensional simplicial complex $\mathcal{G}$, whose vertices correspond to the union of the maxima and split saddles of f on $\mathcal{K}$. If and only if the descending manifold maps a larger neighbor of a saddle s to a maximum m, then there exists an edge between them in the extremum graph, i.e.,

$$\langle s,m\rangle \in \mathcal{G} \iff \exists u \in \mathcal{N}(s,\mathcal{K}) : f(u) > f(s) \wedge d(u) = m.$$

Symmetrically, we can define an extremum graph for minima, join saddles, and the ascending manifold, but the remainder of the manuscript focuses on extremum graphs induced by maxima and split saddles.

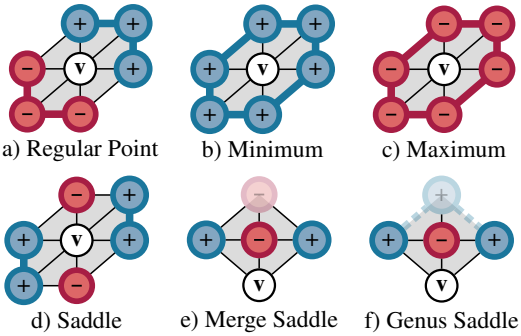


Fig. 3: Critical point classification in cell complexes based on the connectivity of upper (blue) and lower link (red) of a vertex v. v is regular if both are simply connected (a). If either is empty, then v is a minimum or maximum, respectively (b-c). If lower or upper link consist of more than one component, then v is a join or split saddle (d). A merge saddle (e) v merges distinct components; otherwise, a component changes its genus and v is a genus saddle (f). As shown in (e-f), it is not possible to distinguish between merge and genus saddles solely based on neighborhood.

Since the vertices $\mathcal{V}(\mathcal{G})$ are a subset of the vertices $\mathcal{V}(\mathcal{K})$, we extend the definition of the scalar field f to also cover $\mathcal{G}$, i.e., f assigns the same scalar value to the vertices of $\mathcal{G}$ and $\mathcal{K}$, and values of edges of $\mathcal{G}$ are again linearly interpolated. The key aspect of ExTreeM is that the merge tree of f on $\mathcal{G}$ is equivalent to the merge tree of f on $\mathcal{K}$.

2.5 Merge Trees

The *merge tree* $\mathcal{T}$ is a one-dimensional simplicial complex that either records the connectivity changes of superlevel or sublevel set components. In the first case, the merge tree is also called a *split tree*, and in the latter case a *join tree*. The join and split tree can be combined to form the *contour tree* that encodes the evolution of level set components (contours) [6]. The remainder of this paper focuses on split trees, but all arguments apply symmetrically for join trees.

The construction of the merge tree based on Kruskal's serial algorithm [6, 25] for 1D and 2D scalar fields can be illustrated through the metaphor of flooding a landscape whose elevation corresponds to the scalar field. Imagine that this landscape is initially completely submerged. As the water level now continuously decreases, islands (superlevel set components) appear at maxima and grow until they merge at merge saddles. When a component appears, the corresponding maximum is added to $\mathcal{T}$ and becomes the representative of that component. Then, when two or more distinct components merge at a saddle, then this merge saddle is added to $\mathcal{T}$, as well as an edge between each component's representative and the saddle; after which the saddle becomes the representative of the merged components. By convention, when the water level is below the global minimum, then that minimum is added to $\mathcal{T}$ and connected to the smallest vertex of $\mathcal{T}$.

ExTreeM derives a persistence-based branch decomposition $\mathcal{B}$ of $\mathcal{T}$ [34], where each branch $B \in \mathcal{B}$ is a monotone path $m = v_1, v_2, \ldots, v_n$ starting at a maximum $m \in \mathcal{M}(\mathcal{K})$ towards the root until the path reaches a vertex at which the branch merges with another branch that originated at a larger maximum. The termination of the branch originating at the smaller (younger) maximum and the continuation of the larger (older) maximum is referred to as the *elder rule* [13]. We denote the first and last vertex of a branch B by $\alpha(B)$ and $\omega(B)$, respectively. Then, the persistence p (significance) of each branch $B \in \mathcal{B}$ is measured by the function value difference of its first and last vertex: $p(B) = f(\alpha(B)) - f(\omega(B))$. Thus, the first and last vertex of each branch form a maximum-saddle persistence pair.

Every branch is uniquely identified through its starting point, i.e., the maximum at which the branch originates. We refer to a branch originating at maximum m by $\mathcal{B}[m]$. Furthermore, the last vertex of every branch $B \in \mathcal{B}$ is contained in at least one other branch $B' \in \mathcal{B}$. From those branches, the one originating at the largest maximum is called the *parent branch* of B, i.e., $P(B) = \operatorname{argmax}_{\{B' \in \mathcal{B} \mid \omega(B) \in B'\}} f(\alpha(B'))$. Note, this implies that the branch originating at the global maximum is its own parent branch. We also define *leaf branches* as those branches that are not a parent of any other branch.

3 Related Work

Since merge trees are fundamental abstractions for topological data analysis, several approaches have been developed for their computation. In this section, we briefly describe five of these algorithms and discuss similarities and differences to ExTreeM. There also exists a different family of merge tree algorithms based on localized data structures [23, 30, 31], which first create a forest of merge trees for individual parts of an input domain, and then derive the global merge tree in a post processing step. Such algorithms naturally leverage themselves for distributed computation. However, we focus on directly computing the global merge tree on single shared memory machines.

3.1 Serial Merge Tree Computation via Union-Find

The most common merge tree algorithm is based on Kruskal's algorithm for the computation of the minimum spanning tree [25]. We already described this algorithm when introducing merge trees in Sec. 2.5. This algorithm can be used to serially compute the merge tree on an input complex with n vertices and m edges via a union-find data structure in $\mathcal{O}(m \log n)$ time [6, 39].

3.2 FTM-Tree (TTK)

Gueunet et al. [19] proposed a task-parallel version of Kruskal's algorithm, called *FTM-Tree*, which currently is the fastest merge tree algorithm available in the Topology ToolKit [28, 40]. In short, *FTM-Tree* starts to grow superlevel set components at all maxima, where each growth operation (called propagation) is a task than can be executed in parallel. Propagations terminate as soon as they reach a merge saddle, except for the last propagation that reaches the saddle, which then merges all propagations that reached that saddle and continues the propagation. Therefore, this procedure requires some light synchronization at the saddles. The major drawback of this approach is that the entire sweep front of the propagations—i.e., all vertices that are adjacent to the corresponding superlevel set components—need to be maintained in a Fibonacci heap. Despite their theoretical advantages, these heaps require a lot of memory—especially for huge sweep fronts that are common in large datasets—and are therefore slow in practice when they contain many elements (as shown in Table 1).

3.3 Parallel Peak Pruning (VTK-m)

In contrast to the task-parallel approach of *FTM-Tree*, Carr et al. [8] described a fundamentally data-parallel merge (contour) tree algorithm, called *Parallel Peak Pruning (PPP)*, which is implemented in the *VTK-m library* [29]. Instead of performing a sweep, *PPP* makes heavy use of data-parallel primitives, such as sorting. *PPP* and ExTreeM follow the same global strategy: identify leaf branches of the current input, remove (prune) them from the current input, and then repeat this procedure until all branches of the original input have been identified. Furthermore, similar to the extremum graph used in ExTreeM, *PPP* also first reduces the size of the input complex to accelerate computation [5].

Although *PPP* and ExTreeM share many concepts (such as using path compression to determine the descending manifold) there are significant differences. For one, *PPP* does not explicitly compute saddles, just saddle candidates. Those are vertices whose upper link is part of at least two different regions in the descending manifold. However, this is true for many vertices, hence *PPP* has to process significantly more vertices than ExTreeM, which in turn can efficiently compute the actual saddles (Sec. 4.2). Conversely, this means *PPP* can probably be improved in the future to only process the critical points identified by ExTreeM. To identify the leaf branches, these candidates also have to be sorted in each iteration of *PPP* according to multiple criteria, whereas ExTreeM utilizes the special structure of extremum graphs to replace the sorting with a simpler and more efficient maximum polling procedure (Sec. 4.4). Another difference is that ExTreeM only derives merge tree augmentations, whereas *PPP* immediately derives contour tree augmentations, by merging both trees and then deriving a hyperstructure [4], which is queried for each vertex of the domain.

3.4 Merge Tree Computation via Integral Lines

The concept of accelerating the merge tree computation by utilizing an extremum-graph-like structure has also been employed in the approach of Maadasamy et al. [27], which is in turn a multi-threaded version of the approach of Chiang et al. [9]. Their approach first determines all critical points of the domain, then explicitly computes integral lines from saddles towards maxima, and incrementally constructs the merge tree from the leaves towards the root based on these integral lines.

The major difference to ExTreeM is the design of the iterations that yield the merge tree. In their approach, each iteration adds critical points toward the root, whereas iterations in our approach add multiple leaf branches. This has an important implication in practice since most datasets exhibit significantly more genus saddles than merge saddles. Currently (also in our approach) there is no way to predetermine if a saddle is a merge or genus saddle (that is the whole point of the merge tree computation). Thus, according to their iteration scheme, these saddles are added sequentially to the merge tree; effectively serializing the algorithm and therefore reducing parallel performance. In contrast, the number of iterations in our approach does not dependent on the number of critical points, but instead depends on the nesting depth of branches; which is several orders of magnitude smaller.

Ande et al. [2] improved the preliminary steps of computing critical points and integral lines for shared-memory machines. A significant difference to ExTreeM is that their approach explicitly computes integral lines seeded at the saddles, whereas we first apply path compression on the entire domain to determine the reachable maxima from the saddles. Although path compression has been shown to have excellent parallel scaling [8, 26], a comparative study between path compression and explicit integral line traversal is needed, but not within the scope of this paper.

3.5 Triplet Merge Trees

Smirnov and Morozov [38] proposed an alternative approach to merge tree computation, called *triplet merge trees (TMT)*, which is also data-parallel and only requires the compare-and-swap parallel primitive. The core idea of *TMT* is not to directly compute merge tree edges, but instead derive its branches by representing them as vertex triplets. The first two entries of a triplet correspond to the endpoints of branches, and the third entry corresponds to the starting point of its parent branch. The concept behind the main procedure of *TMT* is to initialize its output as if the complex would not contain any edges—i.e., every vertex forms its own triplet—and then add edges of the input complex one by one and "merge" the triplets of the edge vertices. Thus, at every iteration, the triplets correspond to the branch decomposition for the current connectivity, and all branches have been found as soon as all edges have been processed. Moreover, the iteration over all input edges can be parallelized by protecting the merge operation with a compare-and-swap lock. Recently, it was shown that a shared-memory parallel implementation of *TMT* can outperform *Parallel Peak Pruning* [32].

Although ExTreeM also derives a branch decomposition, it uses a fundamentally different strategy to do so. Since *TMT* scales with the number of input edges and for many datasets the extremum graph can have multiple orders of magnitude fewer edges than the original complex (Table 1), it seems promising to first apply ExTreeM to derive the extremum graph, and then to use *TMT* to derive its merge tree.

4 The ExTreeM-Algorithm

The ExTreeM-algorithm outlined in Alg. 1 consists of five high-level procedures: the computation of the descending manifold, critical points, extremum graph, merge tree, and optionally the merge tree augmentation. This section details every procedure in its own subsection.

Algorithm 1: ExTreeM

Inputs: • simplicial complex $\mathcal{K}$
• scalar field $f : \mathcal{K} \to \mathbb{R}$
Outputs: ○ merge tree branch decomposition $\mathcal{B}$
○ merge tree augmentation $a : \mathcal{V}(\mathcal{K}) \to \mathcal{V}(\mathcal{B})$

1 $d \leftarrow$ ComputeDescendingManifold($\mathcal{K}$, f)
2 $\mathcal{M}, \mathcal{S} \leftarrow$ ComputeMaximaAndSaddles($\mathcal{K}$, f, d)
3 $\mathcal{G} \leftarrow$ ComputeExtremumGraph($\mathcal{K}$, f, d, $\mathcal{M}$, $\mathcal{S}$)
4 $\mathcal{B} \leftarrow$ ComputeMergeTree($\mathcal{G}$, f)
5 $a \leftarrow$ ComputeMergeTreeAugmentation($\mathcal{K}$, f, d, $\mathcal{B}$)

4.1 Descending Manifold Computation

We compute the descending manifold $d : \mathcal{V}(\mathcal{K}) \to \mathcal{M}(\mathcal{K})$ via path compression, also known as pointer doubling. This procedure scales well [26] and is also used in other merge tree algorithms such as parallel peak pruning [7]. Path compression is an improvement over the naïve approach of explicitly computing the integral path from each vertex towards a maximum along the steepest ascent. Instead, path compression operates in global iterations, wherein in the first iteration every vertex points to its largest neighbor, and then in subsequent iterations, each vertex points to the vertex its current pointer points to. This procedure is repeated until all pointers converged to the maxima $\mathcal{M}$ of f on $\mathcal{K}$.

Alg. 2 outlines this procedure, where line 1 initializes the output. Note, all vertices can be uniquely identified by their index, so the output is a simple integer array that will record at position i the index of the assigned maximum of vertex i. At first, each vertex is assigned to its largest neighbor (line 2-3). Since this can be done for each vertex independently, this task is trivial to parallelize. To perform the actual path compression in parallel, we distribute all vertices to the available threads (line 4-5). Thus, each thread has to only process its own list of active vertices A, where a vertex is called active as long as it is not pointing to a maximum. For each active vertex v, the thread first retrieves the current pointer u of v (line 8), and then retrieves the vertex w pointed to by u (line 9-10). Note, only the current thread will update the pointer of v, but some other thread might update the pointer of u while the current thread resolves the pointer. This is why the first lookup does not need to be synchronized, but the second lookup needs an atomic read lock (line 9). Only if both pointers u and w are equal, i.e., v now points to a maximum, then v can be removed from the list of active vertices (line 12). Otherwise, the pointer of v is updated to point to w (line 14). Since only the current thread updates the pointer of v, this write operation does not require synchronization.

It is possible to perform path compression without any synchronization by maintaining two arrays, and only reading from one array and writing into the other during block synchronous iterations. However, in our experiments the described approach using light synchronization outperformed the lock free alternative; probably due to memory overhead and cache locality. Alg. 2 requires $\mathcal{O}(2n)$ memory, where n is the number of vertices of the input complex, where half of the memory is required for the output, and the other half to maintain the lists of active vertices. In the worst case, Alg. 2 requires as many iterations in line 6 as the number of vertices in the longest monotone path. However, these paths are relatively short compared to the extent of the input domain, and the parallization effectively doubles in each iteration the lengths of the computed monotone paths; hence Alg. 2 only required few iterations for all our experiments.

Algorithm 2: ComputeDescendingManifold

Inputs: • simplicial complex $\mathcal{K}$
• scalar field $f : \mathcal{K} \to \mathbb{R}$
Outputs: ○ descending manifold $d : \mathcal{V}(\mathcal{K}) \to \mathcal{M}$ where $\mathcal{M}$ are the maxima of f on $\mathcal{K}$

1 $d \leftarrow array(\ |\mathcal{V}(\mathcal{K})|\ )$ *// create integer array with $|\mathcal{V}(\mathcal{K})|$ entries*
2 **parallel foreach** vertex $v \in \mathcal{V}(\mathcal{K})$ **do**
3 $\quad d[v] \leftarrow \text{argmax}_{u \in \mathcal{N}(v,\mathcal{K})} f(u)$ *// assign v to largest neighbor*
4 **parallel foreach** thread t **do**
5 $\quad A \leftarrow$ AssignVerticesToThread(t, $\mathcal{V}(\mathcal{K})$)
6 $\quad$ **while** $|A| > 0$ **do**
7 $\quad\quad$ **foreach** vertex $v \in A$ **do**
8 $\quad\quad\quad u \leftarrow d[v]$ *// current pointer of v*
9 $\quad\quad\quad$ # atomic read
10 $\quad\quad\quad w \leftarrow d[u]$ *// current pointer of u*
11 $\quad\quad\quad$ **if** $u = w$ **then**
12 $\quad\quad\quad\quad A \leftarrow A \setminus \{v\}$ *// delete v from active vertices*
13 $\quad\quad\quad$ **else**
14 $\quad\quad\quad\quad d[v] \leftarrow w$ *// assign w to v*

4.2 Critical Point Computation

As described in Sec. 2.2, critical points are characterized by the connectivity of their upper and lower link. For the computation of the split tree we only need to determine maxima and split saddles, i.e., examine the upper link of a vertex. Alg. 3 outlines this procedure that iterates over each vertex in parallel (line 2-11). For each vertex v, the procedure first retrieves the vertices U of the upper link of v (line 3). If that set is empty, then v must be a maximum (line 5-6). Otherwise, we additionally need to retrieve the edges between the vertices of U (line 8), and only add v to the saddles if these edges are all connected (line 9-11). However, fetching these edges and checking their connectivity via union-find is relatively expensive. We can accelerate the overall procedure by only performing this check if it is actually necessary. To this end, we can exploit the fact that a vertex v can only be a split saddle if its upper link

Algorithm 3: ComputeMaximaAndSaddles

Inputs: • simplicial complex $\mathcal{K}$
• scalar field $f : \mathcal{K} \to \mathbb{R}$
• descending manifold $d : \mathcal{V}(\mathcal{K}) \to \mathcal{M}$
Outputs: ○ maxima $\mathcal{M}$ of f on $\mathcal{K}$
○ saddles $\mathcal{S}$ of f on $\mathcal{K}$

1 $\mathcal{M}, \mathcal{S} \leftarrow \emptyset$ *// initialize output*
2 **parallel foreach** vertex $v \in \mathcal{V}(\mathcal{K})$ **do**
3 $U \leftarrow \{ u \in \mathcal{N}(v,\mathcal{K}) \mid f(u) > f(v) \}$ *// upper link vertices of v*
4 $R \leftarrow \{ \ d(u) \ \mid u \in U \}$ *// reachable maxima of v*
5 **if** $|U| < 1$ **then**
6 $\mathcal{M} \leftarrow \mathcal{M} \cup \{v\}$ *// add v to maxima*
7 **else if** $|R| > 1$ **then**
8 $U' \leftarrow \{ \langle u,v \rangle \in \mathcal{E}(\mathcal{K}) \mid u,v \in U \}$ *// upper link edges*
9 $n \leftarrow$ ComputeNumberOfConnectedComponents(U')
10 *// if U' has more than one component add v to saddles*
11 **if** $n > 1$ **then** $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$

vertices U lead to at least two distinct maxima through an ascending monotone path. Since this necessary condition will not be true for a majority of the regular vertices, this additional check significantly accelerates the overall procedure. Moreover, this information is already recorded in the descending manifold, so the number of reachable maxima can be efficiently retrieved (line 4 and 7).

ExTreeM also includes an optimization of Alg. 3 for regular grids. As stated before, fetching the edges of the upper link and checking their connectivity is relatively expensive, but for regular grids this subprocedure (line 8-9) can be completely replaced with a lookup table. Note, for regular grids the number of neighbors of interior vertices is always constant: 6 for 2D and 14 for 3D grids according to a Freudenthal triangulation [14, 17] (the default triangulation scheme of TTK and PPP). We can also consistently enumerate these neighbors and build a binary number based on their presence in the upper link (Fig. 4). This binary number is then interpreted as an address index in a lookup table that records if for this setting the upper link is connected or not. To generate this lookup table one has to iterate over every possible setting and explicitly perform a connectivity check via union-find. This includes $2^6 = 64$ different settings for 2D, and $2^{14} = 16384$ for 3D. Since the lookup table only needs to store one byte per setting, the lookup table for 3D grids only requires $16kb$ of memory, and the table can even be computed at compile time. This lookup table approach further significantly accelerates the critical point computation, since only boundary vertices need to perform runtime connectivity checks. Obviously, this optimization only supports regular grids, but such grids are frequent use cases in scientific computing.

4.3 Extremum Graph Computation

After the computation of the descending manifold and the critical points, it is straightforward to derive the extremum graph. As outlined in Alg. 4, first the maxima and saddles are added to the output (line 1). Then the algorithm iterates over the saddles in parallel (line 2) and adds an edge between the saddle and every reachable maximum of its upper link (line 3-6). This entire procedure can be performed lock-free by only storing the outgoing edges at each saddle.

4.4 Merge Tree Computation

Once the extremum graph $\mathcal{G}$ is computed, one could apply any merge tree algorithm to derive $\mathcal{T}$ of f on $\mathcal{G}$. However, we propose an approach that utilizes the special structure of extremum graphs, called *Extremum Graph Pairing (EGP)*. The execution of EGP for the running example is shown in Fig. 5. The algorithm is based on the core observation that some edges of $\mathcal{G}$ correspond to persistence pairs of f on $\mathcal{G}$, and therefore to branch endpoints of $\mathcal{T}$. Specifically, those edges $\langle m,s \rangle \in \mathcal{E}(\mathcal{G})$ where (i) s is the largest saddle directly connected to m, and (ii) m is the smallest maximum directly connected to s. Fig. 5 highlights these edges in red for the running example.

Algorithm 4: ComputeExtremumGraph

Inputs: • simplicial complex $\mathcal{K}$
• scalar field $f : \mathcal{K} \to \mathbb{R}$
• descending manifold $d : \mathcal{V}(\mathcal{K}) \to \mathcal{M}$
• maxima $\mathcal{M}$ of f on $\mathcal{K}$
• saddles $\mathcal{S}$ of f on $\mathcal{K}$
Outputs: ○ extremum graph $\mathcal{G}$ of f on $\mathcal{K}$

1 $\mathcal{G} \leftarrow \mathcal{M} \cup \mathcal{S}$ *// initialize output*
2 **parallel foreach** saddle $s \in \mathcal{S}$ **do**
3 $U \leftarrow \{ u \in \mathcal{N}(s,\mathcal{K}) \mid f(u) > f(s) \}$ *// upper link vertices of s*
4 $R \leftarrow \{ \ d(u) \ \mid u \in U \}$ *// reachable maxima of s*
5 **for** maximum $m \in R$ **do**
6 $\mathcal{G} \leftarrow \mathcal{G} \cup \{\langle m,s \rangle\}$ *// add edge between s and each m*

The strategy of EGP is to first identify these edges and add the corresponding persistence pairs/branch endpoints to the output. Afterwards, EGP performs a specialized topological simplification of the extremum graph such that it no longer exhibits these pairs. Then, the core observation holds again for the simplified extremum graph $\mathcal{G}'$ (Fig. 5 right; red edges), so the procedure is repeated on $\mathcal{G}'$ until it only contains the global maximum. The remainder of this section describes the technical details of our EGP implementation, and the appendix contains a formal proof that the outlined procedure actually derives the merge tree of f on $\mathcal{G}$. EGP can be efficiently parallelized using four arrays:

- $\mathcal{B}$: the output array that records at index i the branch of the merge tree that originates at the i-th maximum;
- R: the replacement map that records the maximum that replaced another maximum during the simplification;
- L: the array that records for each iteration the largest (saddle) neighbor of each maximum; and
- N: the array that records for each iteration the current neighbors of the saddles.

Our implementation, outlined in Alg. 5, first initializes these arrays (line 1-8) such that each maximum points to itself in the replacement map R, and the neighbors recorded in N correspond to the saddle neighbors of the input extremum graph. Then the root branch between the global maximum and minimum is already added to the output branches (line 9-11). To perform the iterations, EGP maintains a list of unpaired maxima $\mathcal{M}'$ and saddles $\mathcal{S}'$, i.e., the vertices of the simplified extremum graph $\mathcal{G}'$, which initially correspond to the maxima and saddles of the input extremum graph $\mathcal{G}$ (line 13).

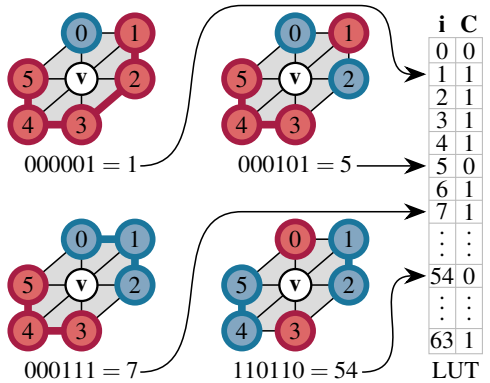


Fig. 4: Lookup-table-based critical point computation on regular grids: Neighbors of interior vertices of such grids can be consistently enumerated (vertex numbers), where the i-th bit encodes whether the i-th neighbor is part of the upper link (blue vertices). For 2D grids with a Freudenthal triangulation [14, 17], 6 bits are needed, and 14 bits for 3D grids. Interpreted as an integer, this number is an address into a lookup table (LUT) that records whether the upper link is connected or not.

As long as there is more than one unpaired maximum, EGP performs the following iteration. First, EGP determines the largest saddle currently connected to each maximum (line 15-21). This is done by first initializing L for each maximum with the global minimum (line 16-17), and then iterating in parallel over every unpaired saddle. Here, EGP iterates over each saddle neighbor (i.e., connected maximum) and checks if the current saddle is larger than the saddle currently recorded in L for that neighbor (line 19-21). If yes, then EGP needs to replace that entry in L with the current saddle. This operation needs to be synchronized with an *atomic compare and swap* lock (line 20).

Next, EGP iterates in parallel over every unpaired maximum to identify all pairable maxima $\mathcal{M}''$ of the current iteration (line 22-31). First, EGP retrieves for each maximum the largest connected saddle via L (line 25). If the current maximum is also the smallest neighbor of that saddle, then EGP found an edge that corresponds to a persistence pair (line 27). In this case, the current maximum is removed from $\mathcal{M}'$ (line 28) and added to $\mathcal{M}''$ (line 29), and R records that the current maximum is replaced by the largest saddle neighbor (line 30). Finally, EGP adds an output branch between the current maximum and the saddle (line 31).

Since one iteration can pair multiple maxima, it is possible that paired maxima point to also paired maxima in R (see the dashed blue arrow in Fig. 5. To resolve this issue, EGP needs to essentially perform a path compression on R (line 32-37). Thus, as long as the current pointer of a maximum is not pointing to itself—i.e., as long as it is pointing to a paired maximum—EGP needs to recursively update the pointer until it does. Note, similarly to the path compression described in Alg. 2, this step requires an *atomic read* lock (line 35).

In the last step of each iteration, EGP iterates in parallel over all unpaired saddles and replaces their neighbors according to R (line 38-40). After this replacement, some saddles will now have only one neighbor and can therefore never be a merge saddle in the simplified extremum graph. Thus, these saddles can safely be removed from the list of unpaired saddles (line 42).

After the extremum graph has been simplified until it only contains the global maximum, all necessary branches have been added to the branch decomposition $\mathcal{B}$. However, each branch currently consists only of its start and end point, i.e., the maximum-saddle persistence pair. For the merge tree, the branches also need to be connected, i.e., EGP needs to determine the parent branch of each branch (line 43-49). Note, this information is already available in R (for a formal argument see the proof in the appendix). To shorten the following description, we say that a branch X was replaced by the branch Y if $\alpha(Y) = R[\alpha(X)]$. So to retrieve the parent branch of B, EGP first has to retrieve the branch B' that replaced B, i.e., find the branch that originates at the maximum that replaced $\alpha(B)$ (line 45). Then, EGP recursively searches for the branch that replaced B' until the endpoint of B' is smaller than the endpoint of the original branch B (line 47). The last B' of this search operation is the parent branch of B, so EGP adds the endpoint of B to B' (line 49). Since multiple threads might want to simultaneously add vertices to a branch, this write operation needs to be synchronized with a lock (line 48).

If more than two branches merge at a saddle, then each child branch will add that saddle to the parent branch; hence the saddle duplication on the parent branch and the need for their removal (line 50-52). Afterwards, $\mathcal{B}$ corresponds to the branch decomposition of the merge tree of f on $\mathcal{G}$ as defined in Sec. 2.5.

4.5 Augmented Merge Tree Computation

The last step of ExTreeM is to compute the merge tree augmentation $a : \mathcal{V}(\mathcal{K}) \to \mathcal{V}(\mathcal{B})$ that assigns each domain vertex to a vertex of the branch decomposition $\mathcal{B}$. This assignment trivially parallelizes over vertices, as outlined in Alg. 6 (line 2). For each vertex v, we first have to find the correct merge tree branch that would contain v. For this, consider the superlevel set component $C \subseteq f^{-1}_{+\infty}(f(v), \mathcal{K})$ that contains v, i.e., $v \in C$. Every maximum of C is the starting point of a branch, and by the definition of the merge tree, all these branches are connected by vertices that are larger than v (otherwise they would not be part of the same component). Since the endpoints of these branches are persistence

Algorithm 5: ComputeMergeTree / ExtremumGraphPairing

Inputs: • extremum graph $\mathcal{G}$
• scalar field $f : \mathcal{G} \to \mathbb{R}$
• descending manifold $d : \mathcal{V}(\mathcal{G}) \to \mathcal{M}$
• maxima $\mathcal{M}$ of f on $\mathcal{G}$
• saddles $\mathcal{S}$ of f on $\mathcal{G}$
Outputs: ○ merge tree branch decomposition $\mathcal{B}$ of f on $\mathcal{G}$

```
1  B ← array( |M| )  // branch array with |M| entries
2  R ← array( |M| )  // replacement map
3  L ← array( |M| )  // records only largest neighbor of each max
4  N ← array( |S| )  // records current neighbors of saddles
5  parallel foreach maximum m ∈ M do
6      R[m] ← m   // initially all maxima point to themselves
7  parallel foreach saddle s ∈ S do
8      N[s] ← {m | ⟨m,s⟩ ∈ E(G)}
9  // add root branch between global max and min
10 (m̂, m̌) ← (argmax_{v∈V(G)} f(v), argmin_{v∈V(G)} f(v))
11 B[m̂] ← (m̂, m̌)
12 // compute remaining branches
13 M', S' ← M, S  // initialize unpaired maxima and saddles
14 while |M'| > 1 do
15     // find largest saddle for each maximum (update L)
16     parallel foreach maximum m ∈ M' do
17         L[m] ← m̌  // reset to global minimum
18     parallel foreach saddle s ∈ S' do
19         foreach maximum m ∈ N[s] do
20             # atomic compare and swap
21             if f(L[m]) < f(s) then L[m] ← s
22     // find all pairable maxima
23     M'' ← ∅  // set of maxima paired in the current iteration
24     parallel foreach maximum m ∈ M' do
25         s ← L[m]  // retrieve largest saddle connected to m
26         // if m is also the smallest neighbor of s then pair
27         if m = argmin_{v∈N[s]} f(v) then
28             M' ← M' \ {m}  // remove from unpaired maxima
29             M'' ← M'' ∪ {m}  // add to paired maxima
30             R[m] ← argmax_{v∈N[s]} f(v)  // update replacement map
31             B[m] ← (m, s) // add branch endpoints / persistence pair
32     // path compress replacement map
33     parallel foreach maximum m ∈ M'' do
34         m' ← m
35         # atomic read
36         while m' ≠ R[m'] do m' ← R[m']
37         R[m] ← m'
38     // update saddle neighbors (i.e., replace edges)
39     parallel foreach saddle s ∈ S' do
40         N[s] ← { R[m] | m ∈ N[s] }
41         // if s has only one neighbor remove from unpaired saddles
42         if |N[s]| = 1 then S' ← S' \ {s}
43 // add intermediate saddles to each branch
44 parallel foreach branch B ∈ B do
45     B' ← B[R[α(B)]]  // retrieve branch originating at R[α(B)]
46     // while ω(B') is larger than ω(B) get next branch
47     while B ≠ B' ∧ f(ω(B')) ≥ f(ω(B)) do B' ← B[R[α(B')]]
48     # lock and then unlock B at α(B')
49     B[α(B')] ← B' ∪ {ω(B)}  // add saddle ω(B) to parent
50 // finally format branches
51 parallel foreach branch B ∈ B do
52     B ← sortAndRemoveDuplicateVertices(B)
```

Algorithm 6: ComputeMergeTreeAugmentation

Inputs: • simplicial complex $\mathcal{K}$
• scalar field $f : \mathcal{K} \to \mathbb{R}$
• descending manifold $d : \mathcal{V}(\mathcal{K}) \to \mathcal{M}$
• merge tree branch decomposition $\mathcal{B}$

Outputs: ○ merge tree augmentation $a : \mathcal{V}(\mathcal{K}) \to \mathcal{V}(\mathcal{B})$

1 $a \leftarrow array(\,|\mathcal{V}(\mathcal{K})|\,)$ *// create integer array with $|\mathcal{V}(\mathcal{K})|$ entries*
2 **parallel foreach** $v \in \mathcal{V}(\mathcal{K})$ **do**
3 *// get some maximum of the superlevel set*
4 *// component $C \subseteq f^{-1}_{+\infty}(f(v),\mathcal{K})$ that contains v*
5 $m \leftarrow d[v]$
6 *// find the branch originating in C that contains v*
7 $B \leftarrow \mathcal{B}[m]$ *// start with the branch originating at m*
8 *// while last vertex of B is larger than v get parent branch*
9 **while** $f(\omega(B)) > f(v)$ **do** $B \leftarrow P(B)$
10 *// perform binary search on branch for upper bound of v*
11 $a[v] \leftarrow$ UpperBound(B, v, f)

pairs, there can only be one branch whose last vertex is smaller than v. This branch belongs to the largest maximum of C, and therefore also has the highest persistence. The strategy of Alg. 6 is to first find some maximum of C, by using the the descending manifold (line 5). Then retrieve the corresponding branch (line 7), and follow the chain of parent branches until finding the target branch whose last vertex is smaller than v (line 9). Once the target branch has been identified, find the smallest vertex of that branch that is not smaller than v, the *upper bound* of v on B (line 11). Since the vertices of B are sorted, the upper bound can be found using binary search and then assigned to v in the augmentation. An example of the augmentation is illustrated in Fig. 6.

5 Proof of Correctness

Let f be a scalar field defined on a simplicial complex $\mathcal{K}$, and let $\mathcal{G}$ be the extremum graph of f. To prove the correctness of the generic concept of ExTreeM (Alg. 1), we show that the merge tree of f on $\mathcal{K}$ is equivalent to the merge tree of f on $\mathcal{G}$. The proof that EGP computes the correct merge tree can be found in the appendix.

We show the equivalence of the two merge trees of $\mathcal{G}, f$ and $\mathcal{K}, f$ in two directions. First, we show that the superlevel set components of the extremum graph are covered by superlevel set components of the domain. Then we show that the superlevel set components of the domain are covered by superlevel set components of the extremum graph. This means that at each level the partition of maxima induced by connected components of the superlevel set are identical and thereby also the nesting of merge tree nodes. Hence, we can conclude that the two merge trees are isomorphic.

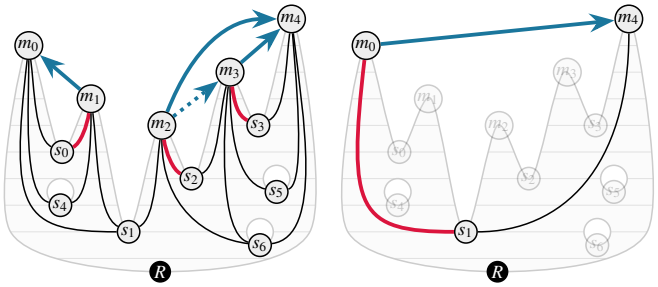
Lemma. *For each connected component $\hat{C}$ of a superlevel set $f^{-1}_{+\infty}(\ell,\mathcal{G})$ exists a connected component C of $f^{-1}_{+\infty}(\ell,\mathcal{K})$ with $\mathcal{M}(\hat{C}) \subseteq \mathcal{M}(C)$.*

Proof. Let $u, w \in \mathcal{M}(\hat{C})$. Then, $f(u) \geq \ell$ and $f(w) \geq \ell$. Also, there is a path $(u = v_1, v_2, \ldots, v_n = w)$ with $v_i \in \mathcal{V}(\hat{C}) \subseteq \mathcal{V}(\mathcal{G})$, $\langle v_i, v_{i+1}\rangle \in \mathcal{E}(\mathcal{G})$, and $f(v_i) \geq \ell$ for all $i \in [1,n]$. Since $\langle v_i, v_{i+1}\rangle \in \mathcal{E}(\mathcal{G})$, there is a monotone path p from v_i to v_{i+1} in $\mathcal{K}$ with $f(x) \geq \ell$ for each $x \in p$. Appending these paths, we get a path p' from u to w in $\mathcal{K}$ with $f(x) \geq \ell$ for each $x \in p'$. Thus, u and w are in the same connected component C of $f^{-1}_{+\infty}(\ell,\mathcal{K})$. In total, we can conclude that $\mathcal{M}(\hat{C}) \subseteq \mathcal{M}(C)$. □

Lemma. *For each connected component C of a superlevel set $f^{-1}_{+\infty}(\ell,\mathcal{K})$ exists a connected component $\hat{C}$ of $f^{-1}_{+\infty}(\ell,\mathcal{G})$ with $\mathcal{M}(C) \subseteq \mathcal{M}(\hat{C})$.*

Proof. We show this through an induction over the size of C. If $|C| = 1$, then C contains exactly one maximum. This maximum has to appear in some component of $f^{-1}_{+\infty}(\ell,\mathcal{G})$. Now assume that the claim holds for any C with $|C| \leq i$ and all $\ell \in \mathbb{R}$.

Consider a component C of $f^{-1}_{+\infty}(\ell,\mathcal{K})$ with $|C| = i+1$. Let $v = \text{argmin}_{u \in \mathcal{V}(C)} f(v)$ be the vertex in C with minimal scalar value. Then v is either a regular vertex or a saddle.



$\mathcal{S}$	N_0	N_1	N_2
s_0	$\{m_0, m_1\}$	$\{m_0\}$	
s_1	$\{m_0, m_1, m_2\}$	$\{m_0, m_4\}$	$\{m_4\}$
s_2	$\{m_2, m_3\}$	$\{m_4\}$	
s_3	$\{m_3, m_4\}$	$\{m_4\}$	
s_4	$\{m_0, m_1\}$	$\{m_0\}$	
s_5	$\{m_3, m_4\}$	$\{m_4\}$	
s_6	$\{m_2, m_3, m_4\}$	$\{m_4\}$	

$\mathcal{M}$	R_0	R_1	R_2	$\mathcal{B}$
m_0			m_4	(m_0, s_1)
m_1		m_0		(m_1, s_0)
m_2		m_4		(m_2, s_2)
m_3		m_4		(m_3, s_3)
m_4				(m_4, R)

Fig. 5: *Extremum Graph Pairing (EGP)* procedure (Alg. 5). EGP maintains connectivity of the current extremum graph $\mathcal{G}'$ through the neighborhood array N, which records the saddle neighbors for each iteration (left table). Per iteration, EGP first detects all edges $\langle m, s\rangle \in \mathcal{G}'$ where the maximum m is the smallest neighbor of the saddle s, and s is also the largest neighbor of m (red edges). These edges correspond to persistence pairs and are added to the output (right table; column $\mathcal{B}$). Then, EGP performs a topological simplification of $\mathcal{G}'$ using the replacement map R. At the beginning of each iteration, R records if an unpaired maximum replaced an already paired maximum (right table; blue entries). So if a pair $\langle m, s\rangle$ is found, then $R[m]$ at first points to the largest neighbor of s (blue edges). Since one iteration can identify multiple pairs, R needs to be path compressed to guarantee that ultimately every paired maximum points to an unpaired maximum. In the example above, m_2 is initially replaced by m_3 (dotted blue line), but will point to m_4 after the path compression. Next, EGP replaces all saddle neighbors according to R, and then all saddles with one neighbor (left table; red entries) are removed from $\mathcal{G}'$. This procedure is repeated until all pairs are found, except between the global maximum and minimum which is added explicitly.

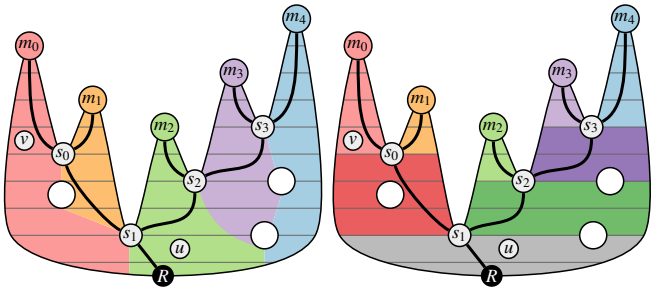


Fig. 6: Illustration of the merge tree augmentation procedure outlined in Alg. 6. Consider the example vertex v of $\mathcal{K}$. Alg. 6 first retrieves the maximum that is assigned to v by the descending manifold (left; background), here m_0. Next, Alg. 6 retrieves the corresponding branch $B = (m_0, s_0, s_1) \in \mathcal{B}$ of the branch decomposition (black edges). Since $f(\omega(B)) = f(s_1) < f(v)$, v belongs to B. Now Alg. 6 determines the smallest vertex of B that is still larger than v, in this case m_0, and assigns this vertex to the augmentation (right; background). For the example vertex u, Alg. 6 first retrieves the branch $B' = (m_2, s_2) \in \mathcal{B}$, but since $f(\omega(B')) = f(s_2) > f(u)$, Alg. 6 must follow the chain of parent branches until the last vertex is no longer larger than u. For u, this is the parent branch $P(B') = (m_4, s_3, s_2, s_1, R)$, and the smallest vertex of $P(B')$ that is still larger than u is s_1.

If v is regular, then $C' := C \setminus \{v\}$ is a connected component of $f^{-1}_{+\infty}(f(v)+\varepsilon, \mathcal{K})$ and $\mathcal{M}(C') = \mathcal{M}(C)$. By induction hypothesis there is a connected component $\hat{C}'$ of $f^{-1}_{+\infty}(f(v)+\varepsilon, \mathcal{G})$ with $\mathcal{M}(C') \subseteq \mathcal{M}(\hat{C}')$. Furthermore, $\hat{C}'$ is included in some component $\hat{C}$ of $f^{-1}_{+\infty}(\ell, \mathcal{G})$. Thus, $\mathcal{M}(C) = \mathcal{M}(C') \subseteq \mathcal{M}(\hat{C}') \subseteq \mathcal{M}(\hat{C})$.

If v is a saddle, then $C' := C \setminus \{v\} = C_1 \cup C_2 \cup \cdots \cup C_n$ where each C_i with $i \in [1,n]$ is a connected component of $f^{-1}_{+\infty}(f(v)+\varepsilon, \mathcal{K})$. Since v is not a maximum, we again have $\mathcal{M}(C) = \mathcal{M}(C') = \bigcup_{i\in[1,n]} \mathcal{M}(C_i)$. If $n = 1$ (i.e., if v is a genus saddle) then this case is analogous to the case where v is regular. If $n > 1$ (i.e., if v merges n connected components) then for each component C_i exists by induction hypothesis a component $\hat{C}_i$ of $f^{-1}_{+\infty}(f(v)+\varepsilon, \mathcal{G})$ with $\mathcal{M}(C_i) \subseteq \mathcal{M}(\hat{C}_i)$. Note that the following fact always holds: the upper link $Lk^+(v)$ of v has $m \geq n$ connected components $U_1, U_2, \ldots, U_m$, and for each component C_i exists at least one U_j such that $U_j \subseteq C_i$. Since each C_i contains at least one U_j, we know that by construction of $\mathcal{G}$ that v is connected to at least one maximum in each $\hat{C}_i$ in $\mathcal{G}$. Thus, with $f(v) \geq \ell$ we can conclude that $\hat{C} := \hat{C}_1 \cup \cdots \cup \hat{C}_n \cup \{v\}$ is a connected component in $f^{-1}_{+\infty}(\ell, \mathcal{G})$. We then get $\mathcal{M}(C) = \bigcup_{i\in[1,n]} \mathcal{M}(C_i) \subseteq \bigcup_{i\in[1,n]} \mathcal{M}(\hat{C}_i) = \mathcal{M}(\hat{C})$. □

6 Experiments

This section describes our experiments in which we compare ExTreeM with two state-of-the-art merge tree algorithms. ExTreeM is implemented in the *Topology ToolKit (TTK)* [28, 40] and utilizes its general data structures. We also added the optimization for lookup-table-based critical point classification on regular grids.

6.1 Setup

We compare ExTreeM with the *FTM-Tree* algorithm [19] of TTK, and the *Parallel Peak Pruning (PPP)* algorithm [4, 8] of *VTK-m* [29, 37]. We described these algorithms operate Sec. 3. All implementations have been parallelized via OpenMP [12, 33], and they were configured to compute the same output: the join tree, the split tree, and the augmentation of both trees. However, *PPP* does not compute the merge tree augmentations separately, it instead immediately derives a contour tree augmentation. To this end, *PPP* combines the split and join tree into the contour tree, and then derives a hyperstructure to perform the contour tree augmentation. So one has to be aware that this can lead to imprecise timing comparisons with the other algorithms regarding the augmentation output. However, as discussed later in detail, our experiments still show significant speedups of ExTreeM over *PPP* without the augmentation step.

All experiments were run on a workstation containing 500GB of RAM and two AMD EPYC 7453 28-Core processors, leading to 56 physical cores. We ran the experiments with at most 56 threads, to exclude effects from hyperthreading, However, since the mainboard has two processors with their own caches, performance gains might deteriorate as we scale to more than 28 threads; depending on the data locality of the algorithms.

Our experiments were performed on regular grid datasets downloaded from the *Open Scivis (OSV) Data* [22] repository. In addition, we derived an interval volume—i.e., an unstructured grid—of the Richtmyer-Meskov instability [10] dataset. As *PPP* currently only supports regular grids, we only compare ExTreeM to *FTM-Tree* for this unstructured grid, where we derived the largest grid that could still be processed by *FTM-Tree* on our machine.

Additionally, we generated synthetic datasets of various resolutions consisting of one layer of *Perlin Noise* [35] with a frequency of one in every dimension, and an amplitude of one. These datasets are in the class of worst-case inputs for ExTreeM, as almost every second vertex is critical, and therefore part of a huge extremum graph.

6.2 Results

Table 1 provides an overview of the timing results of our experiments, where rows are first grouped by dataset, and then by algorithm. We report the total runtime speedup of ExTreeM over the other algorithms for 56 threads in the third column. As the table shows, *PPP* always outperforms *FTM-Tree*, and ExTreeM always outperforms *PPP*. It becomes evident that the performance of ExTreeM depends on the ratio between the number of simplices of the extremum graph and the original complex, i.e., $\frac{|\mathcal{V}(\mathcal{G})|+|\mathcal{E}(\mathcal{G})|}{|\mathcal{V}(\mathcal{K})|+|\mathcal{E}(\mathcal{K})|}$, which we report next to the datasets as percentages. Hence, ExTreeM achieves its best result for the *Jet in Crossflow* dataset with a speedup of around one order of magnitude over *PPP*, since the extremum graph corresponds to only 0.03% of the input complex. Conversely, ExTreeM exhibits its weakest speedups for the Perlin noise datasets due to the huge sizes of the extremum graphs. Our results also highlight the performance issues of *FTM-Tree* for large datasets and scaling issues for unstructured grids. The core procedure of *FTM-Tree* relies on Fibonacci heaps, which require a lot of memory and become slow if they contain a lot of elements. For some datasets, it was not possible to report timings for *FTM-Tree* as the machine ran out of memory. In contrast, ExTreeM only required roughly half as much memory than *PPP* for all experiments.

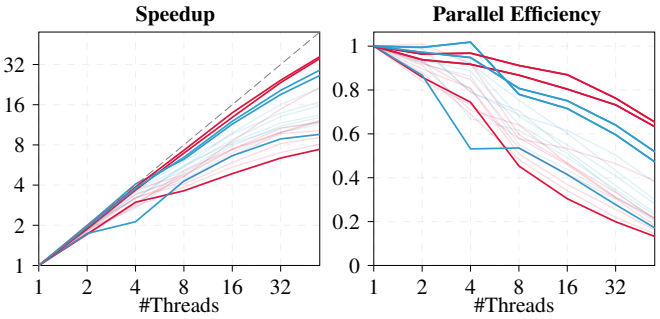


Fig. 7: Comparison of parallel speedup and efficiency between ExTreeM (red) and *PPP* (blue) on the tested datasets. The bold lines correspond, from top to bottom, to the *Jet in Crossflow* [18], *Rayleigh-Taylor instability* [11], and *Perlin Noise* (256^3) datasets, respectively. The left plot shows the parallel speedup (y-axis) defined as the runtime of 1 thread divided by the runtime of n threads (x-axis). The right plot shows the parallel efficiency (y-axis) as the speedup divided by the number of threads (x-axis). The plots show that ExTreeM scales similarly to *PPP*, where for small and large extremum graphs ExTreeM's scaling is slightly better or worse, respectively. Even for the worst-case inputs, i.e., the Perlin noise datasets, ExTreeM does not perform much worse than *PPP*.

We compare the timings of *PPP* and ExTreeM in Table 2 in more detail. Subroutines of both algorithms can roughly be attributed to solving the same task: (i) the computation of the path compression; (ii) the extremum graph (the *topology graph* [5] for *PPP*); (iii) both merge trees; and (iv) both merge tree augmentations (the contour tree computation and augmentation for *PPP*). We concede that the computation of the contour tree itself requires a significant amount of time, but as shown in Table 2, even excluding the time for the augmentations, ExTreeM can still report strong speedups over *PPP*. It is important to note that if *PPP* would use the path compression and extremum graph subroutines of ExTreeM, then the total speedup of ExTreeM would still outperform *PPP* due to the significant speedup of the merge tree subroutine. Specifically, the merge tree subroutine of ExTreeM exhibits the largest speedup, while also having the weakest scaling behaviour as its total time is already relatively low compared to the other tasks. The other tasks have excellent scaling behaviour, with the augmentation step almost scaling perfectly as it can process each vertex independently.

To further evaluate the scaling of ExTreeM, we show in Fig. 7 plots of the total parallel speedup and efficiency of *PPP* and ExTreeM. Overall, the plots show that ExTreeM and *PPP* scale similarly, and that ExTreeM scales with the relative size of the extremum graphs. However, as already shown in Table 2, ExTreeM's merge tree subroutine (EGP) is short in terms of absolute time, but that time is almost constant for all thread counts; which weakens the overall scaling behaviour of ExTreeM. In future work, we may investigate if ExTreeM would benefit from another merge tree subroutine which might take longer than EGP at first, but scales better for large thread counts.

Table 1: Timing results of our experiments first grouped by dataset, and then by algorithm. The third column reports the total runtime speedup (SU) of ExTreeM over the current algorithm for 56 threads. The remaining columns show, per thread count, the total runtime of the two merge tree computations and augmentations in seconds. Datasets were downloaded from the *OSV Data* [22] repository, and the last two datasets correspond to generated Perlin noise. For each dataset we provide the original size, as well as the relative size of the extremum graph (percentages). The results show that *PPP* is always faster than *FTM-Tree* (at scale), and that ExTreeM is always faster than *PPP*. In the best case, ExTreeM is up to an order of magnitude faster than *PPP* (blue). Even for the Perlin noise dataset, which corresponds to the class of worst-case inputs, ExTreeM still outperforms *PPP*, but not by much (red). For some datasets the *FTM-Tree* computations ran out of memory and are therefore missing from the table.

Dataset	Algorithm	SU	56T	32T	16T	8T	4T	2T	1T
ctBones [41]	ExTreeM	-	0.50	0.56	0.80	1.17	2.11	3.27	5.98
128^3 (3.26%)	PPP	**3.48**	1.74	1.90	2.50	3.96	5.74	10.86	20.36
	FTM-Tree	**14.34**	7.17	5.13	4.37	5.08	5.74	6.47	9.66
Backpack [24]	ExTreeM	-	4.42	4.67	6.81	10.58	14.01	25.25	45.23
$512 \times 512 \times 373$ (4.79%)	PPP	**3.03**	13.41	15.59	21.89	34.97	64.44	91.50	172.45
	FTM-Tree	**14.32**	63.33	45.28	40.27	44.69	41.73	51.31	79.02
Magnetic Reconnection [20]	ExTreeM	-	21.58	23.92	33.15	48.13	60.79	105.52	195.81
512^3 (8.84%)	PPP	**2.95**	63.72	73.39	103.75	164.92	210.57	395.35	726.40
	FTM-Tree	**42.35**	914.22	787.64	865.48	830.96	820.62	924.67	931.89
Rayleigh-Taylor instability [11]	ExTreeM	-	14.21	21.50	39.16	72.56	137.31	268.47	503.67
1024^3 (0.30%)	PPP	**8.01**	113.76	157.82	262.92	482.16	738.87	1513.28	3009.64
	FTM-Tree	**10.41**	147.88	147.78	167.88	207.60	376.77	725.51	1284.83
Neurons in Marmoset [16]	ExTreeM	-	32.34	37.39	48.04	67.86	95.01	149.20	262.18
$1024 \times 1024 \times 314$ (15.21%)	PPP	**2.05**	66.18	76.14	105.82	157.29	280.86	445.43	887.27
Kingsnake [36]	ExTreeM	-	36.34	43.99	59.17	93.39	128.17	235.03	434.40
$1024 \times 1024 \times 795$ (4.71%)	PPP	**2.79**	97.97	118.47	170.53	272.38	445.99	823.12	1641.87
Jet in Crossflow [18]	ExTreeM	-	14.26	21.39	37.51	71.55	134.80	270.67	521.87
$1408 \times 1080 \times 1100$ (0.03%)	PPP	**11.20**	159.71	226.46	386.06	718.04	1223.16	2386.21	4609.00
Richtmyer-Meshkov instability [10]	ExTreeM	-	39.15	56.85	98.95	182.57	287.42	502.10	844.03
$1536 \times 1536 \times 1408$ (0.31%)	PPP	**5.01**	196.00	258.66	422.94	756.77	1316.27	2110.14	4163.87
Unstructured Richtmyer-Meshkov [10]	ExTreeM	-	1.48	1.76	2.57	4.12	6.97	11.60	21.68
$\sim 7 \times 10^6$ vertices, $\sim 42 \times 10^6$ edges (12.82%)	FTM-Tree	**67.85**	100.42	93.76	87.38	92.12	94.69	94.48	100.61
Perlin Noise	ExTreeM	-	2.24	2.60	3.41	4.58	5.58	9.67	16.60
256^3 (23.28%)	PPP	**1.83**	4.10	4.43	5.92	9.17	18.48	22.65	39.29
	FTM-Tree	**139.85**	313.26	281.10	273.31	207.78	272.25	204.69	273.59
Perlin Noise	ExTreeM	-	148.81	181.72	243.14	379.63	490.85	897.46	1811.55
1024^3 (23.20%)	PPP	**1.41**	209.42	251.81	373.76	604.46	892.84	1766.66	3285.86

Table 2: Detailed timing comparison between ExTreeM and PPP based on their subroutines for the *Jet in Crossflow* dataset [18] (top group), and the Richtmyer-Meshkov instability dataset [10] (bottom group). Subroutines of both algorithms do not match perfectly, but rough comparisons are possible. In the *Path Compression* subroutine both algorithms derive the ascending and descending manifold (for ExTreeM this requires executing 2×Alg. 2). Then, in the *Extremum Graph* step, ExTreeM derives the extremum graph (2×Alg. 3+2×Alg. 4), and *PPP* derives a similar (yet much larger) data structure called the topology graph [5]. Then, ExTreeM performs the actual merge tree computations (2×Alg. 5), and *PPP* performs its main procedure. In the final step ExTreeM derives both merge tree augmentations (2×Alg. 6), while *PPP* computes the contour tree, the hyperstructure, and the contour tree augmentation. The table shows that even if we exclude the augmentation step, ExTreeM still exhibits strong speedups over *PPP*; in particular for the merge tree computation (blue). However, the merge tree computation also exhibits the worst scaling behavior of all subroutines (red).

Subroutine	Algorithm	56T	32T	16T	8T	4T	2T	1T
Path Compression	ExTreeM	7.6	10.5	17.3	32.1	59.9	119.8	231.1
	PPP	27.0	38.4	61.1	113.0	176.9	328.2	669.6
Extremum Graph	ExTreeM	1.4	2.5	4.4	8.7	15.6	30.6	56.9
	PPP	66.6	107.4	204.3	394.7	660.0	1315.0	2537.8
Merge Tree	ExTreeM	**0.2**	**0.2**	**0.3**	**0.3**	**0.2**	**0.3**	**0.3**
	PPP	**34.0**	44.9	67.8	120.4	224.2	432.9	819.6
Augmentation	ExTreeM	4.8	8.0	15.3	30.2	58.9	119.8	233.3
	PPP	25.0	28.6	45.5	80.8	153.9	300.5	572.2
Path Compression	ExTreeM	18.2	24.9	41.3	74.2	128.6	243.8	410.5
	PPP	24.7	32.7	55.2	96.8	156.0	200.2	427.6
Extremum Graph	ExTreeM	6.9	11.8	22.3	42.3	67.5	112.7	194.1
	PPP	61.5	95.6	170.3	327.8	569.3	979.9	1958.9
Merge Tree	ExTreeM	4.2	3.8	3.7	4.2	3.7	5.5	7.6
	PPP	56.7	65.5	104.9	175.9	289.0	534.3	1037.8
Augmentation	ExTreeM	9.0	15.3	30.9	60.9	86.8	138.8	231.1
	PPP	42.4	53.8	79.9	140.1	284.5	367.8	701.9

7 Conclusion

We described a generic merge tree computation scheme, called ExTreeM, that first derives the extremum graph $\mathcal{G}$ of a scalar field f defined on a complex $\mathcal{K}$, and then uses any merge tree algorithm to derive the merge tree of f on $\mathcal{G}$ instead of $\mathcal{K}$. We proved that this generic scheme is correct by formally showing that the merge tree of f on $\mathcal{G}$ is equivalent to the merge tree of f on $\mathcal{K}$. Additionally, we described a merge tree algorithm specialized for extremum graphs, called *extremum graph pairing (EGP)*. Our experiments showed that ExTreeM with the *EGP* subroutine outperforms two state-of-the-art merge tree algorithms: *FTM-Tree* [19] of the Topology ToolKit (TTK) [28, 40], and *Parallel Peak Pruning* (*PPP*) [4, 8] of the Visualization Toolkit (VTK-m). In the best case, ExTreeM achieves a speedup of up to one order of magnitude over *PPP*, and two orders of magnitude over *FTM-Tree*.

Furthermore, we provide an implementation of ExTreeM in TTK, which is open source and integrated in the widely-used ParaView [1] frontend. However, as confirmed in our experiments, with the recent advances in merge tree algorithms, *FTM-Tree* is no longer competitive at scale. The integration of ExTreeM in TTK therefore makes TTK's merge tree algorithm again suitable for benchmarks, as well as benefits a huge number of practitioners.

We consider the generic concept of ExTreeM to be its strongest contribution. Specifically, in future work, we intent to investigate the impact of different merge tree subroutines on the overall performance of ExTreeM. In particular, the *Triplet Merge Tree* algorithm (*TMT*) [32, 38] is of special interest here as it scales with the number of input edges, and the extremum graph can be up to multiple orders of magnitude smaller than the original complex. We also see potential improvements of *PPP* by further reducing the size of processed saddle candidates via ExTreeM's lookup-table-based critical point classification. *PPP* and *TMT* can be executed on the GPU, but TTK—and therefore ExTreeM—currently lacks a GPU backend. Since in this work we focused on CPU architectures, the port of ExTreeM to the GPU is left for future work. It also seems promising to adapt ExTreeM for distributed computing, as the computation of the extremum graph, via descending manifolds and critical points, can easily be distributed and is expected to scale well.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration under Contract No. DE-AC02-05CH11231 to the Lawrence Berkeley National Laboratory.

References

[1] J. Ahrens, B. Geveci, and C. Law. ParaView: An End-User Tool for Large-Data Visualization. *The Visualization Handbook*, pp. 717–731, 2005. 9

[2] A. Ande, V. Subhash, and V. Natarajan. TACHYON: Efficient Shared Memory Parallel Computation of Extremum Graphs. In *Computer Graphics Forum*. Wiley Online Library, 2023. 4

[3] T. F. Banchoff. Critical Points and Curvature for Embedded Polyhedral Surfaces. *The American Mathematical Monthly*, 77(5):475–485, 1970. 2

[4] H. A. Carr, O. Rübel, G. H. Weber, and J. P. Ahrens. Optimization and Augmentation for Data Parallel Contour Trees. *IEEE Transactions on Visualization and Computer Graphics*, 28(10):3471–3485, 2021. 3, 8, 9

[5] H. A. Carr and J. Snoeyink. Representing Interpolant Topology for Contour Tree Computation. *Topology-Based Methods in Visualization II*, pp. 59–73, 2009. 3, 8, 9

[6] H. A. Carr, J. Snoeyink, and U. Axen. Computing Contour Trees in all Dimensions. *Computational Geometry*, 24(2):75 – 94, 2003. 3

[7] H. A. Carr, G. Weber, C. Sewell, and J. Ahrens. Parallel Peak Pruning for Scalable SMP Contour Tree Computation. In *IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 75–84, 2016. 1, 4

[8] H. A. Carr, G. H. Weber, C. M. Sewell, O. Rübel, P. Fasel, and J. P. Ahrens. Scalable Contour Tree Computation by Data Parallel Peak Pruning. *IEEE Transactions on Visualization and Computer Graphics*, 27(4):2437–2454, 2019. 3, 4, 8, 9

[9] Y.-J. Chiang, T. Lenz, X. Lu, and G. Rote. Simple and Optimal Output-Sensitive Construction of Contour Trees using Monotone Paths. *Computational Geometry*, 30(2):165–195, 2005. 3

[10] R. H. Cohen, W. P. Dannevik, A. M. Dimits, D. E. Eliason, A. A. Mirin, Y. Zhou, D. H. Porter, and P. R. Woodward. Three-Dimensional Simulation of a Richtmyer–Meshkov Instability with a Two-Scale Initial Perturbation. *Physics of Fluids*, 14(10):3692–3709, 2002. 2, 8, 9

[11] A. W. Cook, W. Cabot, and P. L. Miller. The mixing transition in Rayleigh-Taylor instability. *Journal of Fluid Mechanics*, 511:333–362, 2004. doi: 10.1017/S0022112004009681 8, 9

[12] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. 8

[13] H. Edelsbrunner and J. Harer. *Computational Topology: An Introduction*. American Mathematical Society, 2009. 2, 3

[14] H. Edelsbrunner and M. Kerber. Dual Complexes of Cubical Subdivisions of $\mathbb{R}^n$. *Discrete & Computational Geometry*, 47:393–414, 05 2012. doi: 10.1007/s00454-011-9382-4 5

[15] H. Edelsbrunner and E. P. Mücke. Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms. *ACM Transactions on Graphics (tog)*, 9(1):66–104, 1990. 2

[16] F. Federer. Pyramidal neurons in the marmoset primary visual cortex. https://klacansky.com/open-scivis-datasets/. [Accessed 29-Mar-2023]. 9

[17] H. Freudenthal. Simplizialzerlegungen von Beschrankter Flachheit. *Annals of Mathematics*, 43:580, 1942. 5

[18] R. W. Grout, A. Gruber, H. Kolla, P.-T. Bremer, J. C. Bennett, A. Gyulassy, and J. H. Chen. A direct numerical simulation study of turbulence and flame structure in transverse jets analysed in jet-trajectory based coordinates. *Journal of Fluid Mechanics*, 706:351–383, 2012. doi: 10.1017/jfm.2012.257 8, 9

[19] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Task-Based Augmented Merge Trees with Fibonacci Heaps. In *IEEE Symposium on Large Data Analysis and Visualization*, 2017. 1, 3, 8, 9

[20] F. Guo, H. Li, W. Daughton, and Y.-H. Liu. Formation of hard power laws in the energetic particle spectra resulting from relativistic magnetic reconnection. *Phys. Rev. Lett.*, 113:155005, Oct. 2014. doi: 10.1103/PhysRevLett.113.155005 2, 9

[21] 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. In *Computer Graphics Forum*, vol. 35, pp. 643–667. Wiley Online Library, 2016. 1

[22] P. Klacansky. Open Scientific Visualization Datasets, 2023. https://klacansky.com/open-scivis-datasets. 8, 9

[23] P. Klacansky, A. Gyulassy, P.-T. Bremer, and V. Pascucci. Toward Localized Topological Data Structures: Querying the Forest for the Tree. *IEEE Transactions on Visualization and Computer Graphics*, 2019. doi: 10.1109/TVCG.2019.2934257 3

[24] K. Kreeger. Ct scan of a backpack filled with items. https://klacansky.com/open-scivis-datasets/. [Accessed 29-Mar-2023]. 9

[25] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956. 3

[26] R. G. Maack, J. Lukasczyk, J. Tierny, H. Hagen, R. Maciejewski, and C. Garth. Parallel Computation of Piecewise Linear Morse-Smale Segmentations. *IEEE Transactions on Visualization and Computer Graphics*, 2023. 4

[27] S. Maadasamy, H. Doraiswamy, and V. Natarajan. A Hybrid Parallel Algorithm for Computing and Tracking Level Set Topology. In *2012 19th International Conference on High Performance Computing*, pp. 1–10. IEEE, 2012. 3

[28] T. B. Masood, J. Budin, M. Falk, G. Favelier, C. Garth, C. Gueunet, P. Guillou, L. Hofmann, P. Hristov, A. Kamakshidasan, C. Kappe, P. Klacansky, P. Laurin, J. A. Levine, J. Lukasczyk, D. Sakurai, M. Soler, P. Steneteg, J. Tierny, W. Usher, J. Vidal, and M. Wozniak. An Overview of the Topology ToolKit. In *TopoInVis 2019-Topological Methods in Data Analysis and Visualization*, 2019. 1, 3, 8, 9

[29] K. Moreland, C. Sewell, W. Usher, L.-t. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, et al. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE computer graphics and applications*, 36(3):48–58, 2016. 1, 3, 8

[30] D. Morozov and G. Weber. Distributed Merge Trees. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 93–102, 2013. 3

[31] A. Nigmetov and D. Morozov. Local-Global Merge Tree Computation with Local Exchanges. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13, 2019. 3

[32] A. Nigmetov and D. Morozov. Fast Merge Tree Computation via SYCL. In *2022 Topological Data Analysis and Visualization (TopoInVis)*, pp. 1–8. IEEE, 2022. 4, 9

[33] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.0, May 2008. 8

[34] V. Pascucci, K. Cole-McLaughlin, and G. Scorzelli. Multi-Resolution Computation and Presentation of Contour Trees. In *Proc. IASTED Conference on Visualization, Imaging, and Image Processing*, pp. 452–290, 2004. 3

[35] K. Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985. 8

[36] T. Rowe. Digimorph - Lampropeltis getula (common kingsnake) — digimorph.org. http://www.digimorph.org/specimens/Lampropeltis_getula/egg/whole/. [Accessed 29-Mar-2023]. 9

[37] W. J. Schroeder, K. Martin, and W. E. Lorensen. *The Visualization Toolkit: An Object Oriented Approach to 3D Graphics*. Kitware, Inc., 2004. 1, 8

[38] D. Smirnov and D. Morozov. Triplet Merge Trees. In *Topological Methods in Data Analysis and Visualization V: Theory, Algorithms, and Applications 7*, pp. 19–36. Springer, 2020. 4, 9

[39] S. P. Tarasov and M. N. Vyalyi. Construction of Contour Trees in 3D in O(n log n) Steps. In *Proceedings of the fourteenth annual symposium on Computational geometry*, pp. 68–75, 1998. 3

[40] J. Tierny, G. Favelier, J. A. Levine, C. Gueunet, and M. Michaux. The Topology ToolKit. *IEEE Transactions on Visualization and Computer Graphics*, 2017. https://topology-tool-kit.github.io/. 1, 3, 8, 9

[41] TTK Contributers. TTK data examples. https://github.com/topology-tool-kit/ttk-data. [Accessed 29-Mar-2023]. 2, 9

APPENDIX

EGP derives the merge tree of f on $\mathcal{G}$

In the following, we argue the correctness of Alg. 5. We start by introducing some notation. Let $\mathcal{M}^i, \mathcal{S}^i$ be the sets $\mathcal{M}', \mathcal{S}'$ before the i-th iteration of the while-loop in line 14. Let N^i be the neighbor array N at iteration i. Let G^i be the graph defined by the neighborhood N^i on the vertex set $\mathcal{M}^i \cup \mathcal{S}^i$.

The overall idea of our arguments is as follows. In each iteration of the while-loop, we simplify the graph G^i (starting with $\mathcal{G} = G^0$) by simplifying leaf branches (i.e. those branches that are not a parent of any other branch). In essence, we perform an iterated topological simplification and remember the simplified branches, until the whole set of persistence pairs is found.

Formally, we first show that the while-loop in lines 14-42 correctly computes the persistence pairs $\mathcal{P}$ of f on $\mathcal{G}$. Towards this, we show the core invariants and termination of the loop, which implies correctness. Afterwards, we show that lines 44-49 compute the correct nesting of the persistence pairs and thereby the correct merge tree.

Lemma. *In the while-loop in lines 14-42, the following invariants hold at each iteration i:*

(i) At line 32, for all paired maxima $m \in \mathcal{M}''$ with $\mathcal{B}[m] = (m,s)$, it holds that $R[m]$ and m are in the same component of $f^{-1}_{+\infty}(f(s), G^i)$.

(ii) At line 32, for all paired maxima $m \in \mathcal{M}''$ with $\mathcal{B}[m] = (m,s)$ it holds that $(m,s) \in \mathcal{P}(G^i)$.

(iii) The persistence pairs of the current graph $\mathcal{P}(G^i)$ are the persistence pairs of the original graph $\mathcal{P}(\mathcal{G})$ restricted to the unpaired vertices $\mathcal{M}^i \cup \mathcal{S}^i$, i.e. $\mathcal{P}(G^i) = \mathcal{P}(\mathcal{G}) \cap (\mathcal{M}^i \times \mathcal{S}^i)$.

(iv) For any saddle $s \in S^i$ and any component C of $f^{-1}_{+\infty}(f(s), G^i)$, there is a component C' of $f^{-1}_{+\infty}(f(s), \mathcal{G})$ with $C \subseteq C'$.

Proof. The invariants obviously hold initially with $\mathcal{M}^i = \mathcal{M}$, $\mathcal{S}^i = \mathcal{S}$ and $\mathcal{B} = \emptyset$. Now consider a later iteration $i+1$. We first assume (for the sake of simplicity) that iteration $i+1$ only pairs (line 31) and removes (lines 28 and 42) one maximum-saddle pair (m,s). Then, the loop in lines 33 to 37 does not have any effect and we can ignore it for now.

Invariant (i). We now argue that invariant (i) is preserved. Let $m \in \mathcal{M}''$ in line 32. This means that $m = \operatorname{argmin}_{v \in N[s]} f(v)$ (line 27) and $R[m] = \operatorname{argmax}_{v \in N[s]} f(v)$ (line 30). Thus, m and $R[m]$ are both neighbors of s in G^i and therefore in the same connected component of $f^{-1}_{+\infty}(f(s), G^i)$. Hence, invariant (i) is preserved.

Invariant (ii). We now argue that if (m,s) is added to $\mathcal{B}$ (i.e. $s = L[m]$ and $m = \operatorname{argmin}_{v \in N[s]} f(v)$ in line 27), then (m,s) is a pair in $\mathcal{P}(G^i)$. Since $s = L[m]$, we know that $\langle m, s \rangle \in G^i$ and $f(s) > f(s')$ for each $\langle m, s' \rangle \in G^i$. Thus, for any path $(m = v_1, v_2, \ldots, v_k)$ from m to some other vertex, we know that $f(v_2) \leq f(s)$. This implies that there is a connected component C with $\mathcal{M}(C) = \{m\}$ in $f^{-1}_{+\infty}(\ell, G^i)$ for any $f(s) < \ell \leq f(m)$.

Let $C_1, C_2, \ldots, C_k$ be the components of $f^{-1}_{+\infty}(f(s) + \varepsilon, G^i)$ that are merged by s. W.l.o.g. we can assume that $\mathcal{M}(C_1) = \{m\}$. Since every other neighbor of s is larger than m (as $m = \operatorname{argmin}_{v \in N[s]} f(v)$), we can conclude that each C_j with $j > 1$ contains at least one maximum larger than m and therefore has greater persistence than C_1. Thus, (m,s) is a persistent pair. Since (m,s) is added to $\mathcal{B}$, it follows that (ii) is preserved.

Invariant (iii). By induction hypothesis, we know that invariant (iii) holds for G^i. We now show that simplifying (m,s) in G^i to obtain G^{i+1} preserves this property.

We have $\mathcal{M}^{i+1} = \mathcal{M}^i - \{m\}$, and either $\mathcal{S}^{i+1} = \mathcal{S}^i - \{s\}$ or $\mathcal{S}^{i+1} = \mathcal{S}^i$ (either s appears in exactly one persistence pair and is therefore also removed in line 42 or s appears in multiple persistence pairs that have not been found yet). We now argue that $\mathcal{P}(G^{i+1}) = \mathcal{P}(G^i) - \{(m,s)\}$. Consider any other persistence pair $(s', m') \in \mathcal{P}(G^i)$. Then there is a path $p = (s' = v_1, v_2, \ldots, v_k = m')$ in G^i connecting s' to m' with $f(v) \geq f(s')$ for all $v \in p$. We now argue that there is a path p' in G^{i+1} connecting s' to m' with $f(v) \geq f(s')$ for all $v \in p'$. This implies that $(m', s') \in \mathcal{P}(G^{i+1})$ as well, since persistence (or scalar values at all) of other branches are not changed by simplifying (m,s).

If $m \notin p$ and $s \notin p$, then p is still a path in G^{i+1} and $(m', s') \in \mathcal{P}(G^{i+1})$. Otherwise, if there is a j with $v_j = s$, we do the following case distinction: either $s \in \mathcal{S}^{i+1}$ or $s \notin \mathcal{S}^{i+1}$. If $s \in \mathcal{S}^{i+1}$, then s can remain in the path. Otherwise, s has been simplified, and thus m and $R[m]$ are the only neighbors of s in G^i: for $v_j = s$ we get $\{v_{j-1}, v_{j+1}\} = \{m, R[m]\}$. We can replace (v_{j-1}, s, v_{j+1}) by $R[m]$ to obtain a path between s' and m'. We now have to consider the only remaining case that $m \in p$ but $s \notin p$. Let $v_j = m$, then v_{j+1} and v_{j-1} are connected to $R[m]$ in G^{i+1}, thus we can remove m from p by replacing v_j through $R[m]$. We obtain a valid path between s' and m' in G^{i+1}.

Next, we argue that removing any other saddle (i.e. those that have not been paired in this iteration) in line 42 does not change the persistence diagram $\mathcal{P}(G^{i+1})$. Since s only has one neighbor in N^{i+1}, it is not a merge saddle in G^{i+1} and thus not in the persistence diagram. The connectivity of superlevel sets is also not changed, since s can not be part of any connecting path, as it only has one neighbor.

Invariant (iv). To show invariant (iv) for G^{i+1}, let s' be some saddle with a component C of $f^{-1}_{+\infty}(f(s), G^i)$. Let $m' \neq m, m'' \neq m$ be two maxima in C, i.e. there is a path p between them with $f(v) \geq f(s')$ for all $v \in p$. With the same arguments as for invariant (iii), we can conclude that there is such a path in G^i if and only if there is one in G^{i+1}. Thus, if m', m'' are connected via a path above $f(s')$ in G^{i+1}, they are so in G^i and by induction hypothesis in $\mathcal{G}$.

It remains to show that the invariants are also preserved if, instead of one pair (m,s), multiple pairs are simplified. Note that the arguments for invariants (i) and (ii) are not changed in this case. For invariant (iii) and (iv), we still need to argue that paths for all non-simplified pairs exist. First, note that $R[m]$ points to an unpaired maximum for each $m \in \mathcal{M}''$ after line 37. This follows directly from R not containing any loops even after recursive replacement ($f(R[m]) > f(m)$ prohibits the existence of loops). Then, all arguments about the paths between m' and s' can be applied analogously, as we simply have to replace multiple simplified maxima recursively. The connectivity arguments still hold, as for each $m \in \mathcal{M}''$ the maximum $R[m]$ is a vertex in G^{i+1}. □

Next, we want to show that the loop terminates, which directly implies the computation of the full set of persistence pairs.

Lemma. *Each iteration of the while-loop in lines 14-42 simplifies at least one pair. Thus, after line 42, the set $\mathcal{B}$ contains exactly the persistence pairs of $\mathcal{G}$, i.e. $\mathcal{B} = \mathcal{P}(\mathcal{G})$.*

Proof. To see that each iteration finds a pair, note that the following holds:

Fact. For every non-trivial extremum graph G, f, there is at least one merge saddle s such that $f^{-1}_{+\infty}(f(s), G)$ has at least two components $C_1, \ldots C_k$ with $C_i = \{m_i\}$ for all $1 \leq i \leq k$ and $m_i \in \mathcal{M}(G)$.

For such a saddle s, $\langle s, m_i \rangle \in G$ for each $1 \leq i \leq k$ by definition. Let m be the minimum within all m_i. Then, $s = L[m]$ and $m = \operatorname{argmin}_{v \in N[s]} f(v)$. Thus, (m,s) is found and pruned.

Since clearly for each iteration i, G^i is an extremum graph, we know that always one pair is pruned. If no such saddle exists, then the extremum graph is trivial and the global maximum is paired.

Then, since a maximum m is removed from $\mathcal{M}'$ if and only if a pair (m,s) is added to $\mathcal{B}$, lines 14 to 42 compute the persistence diagram of $\mathcal{G}$, in particular, $\mathcal{B} = \mathcal{P}(\mathcal{G})$ after line 42. □

As a last step, we now show that lines 44-49 compute the correct nesting of branches.

Lemma. *For each branch $(m,s) = B \in \mathcal{B}$, the branch B' determined by lines 45-47 is the parent branch of B.*

Proof. Let $\hat{B} = (\hat{m}, \hat{s}) = P(B)$ be the parent branch of B. Consider the superlevel set component $C = f^{-1}_{+\infty}(f(s))$ containing m. By definition,

$\hat{m}$ is the largest maximum in C. Also, there is exactly one maximum in C that is paired with a vertex that is smaller than s, in particular $\hat{s}$ (to simplify our argument we assume that the global minimum is not a saddle). Otherwise $\mathcal{B}$ would not be a valid persistence-based branch decomposition.

Now consider the first branch $B' = (m', s')$ retrieved for the input branch $B = (m, s)$ in line 45. Initially, we have $m' = R[m]$. By invariants (i) and (iv) from the lemma above, we know that $m' \in C$. Furthermore, we know that $f(m') > f(m)$.

If $f(s') < f(s)$, then m' is exactly the unique maximum in C as discussed above and therefore has to be the origin of the parent branch $\hat{B}$. Otherwise, we continue the while-loop. So m' is set to $R[m']$ and R maintains that $m' \in C$ and $f(m') > f(m)$. Thus, the two properties are invariant for the whole loop. The first property ensures that m' always gets strictly larger, thus prohibiting repeating m'. Since there are only finitely many maxima in C, we can conclude that eventually $m' = \hat{m}$ holds.

For completeness, consider the case where the saddle s is also the global minimum. In this case, there is *no* other pair m', s' with $m \in C$ and $f(s') < f(s)$. In particular $f(\hat{s}) = f(s)$. Nonetheless, the loop in line 47 continues for any $s' \neq \hat{s}$, but if $s' = \hat{s}$, we also have $B' = \hat{B}$ and the loop stops. Therefore, the behavior is still identical and correct. □