Task-based Augmented Reeb Graphs with Dynamic ST-Trees

Charles Gueunet, Pierre Fortin, Julien Jomier, Julien Tierny

Eurographics Symposium on Parallel Graphics and Visualization (2019)
H. Childs, S. Frey (Editors)

Task-based Augmented Reeb Graphs with Dynamic ST-Trees

C. Gueunet[1], P. Fortin[2], J. Jomier[1], J. Tierny[2]

[1] Kitware SAS, Villeurbane, France
[2] Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, F-75005 Paris, France

Abstract

This paper presents, to the best of our knowledge, the first parallel algorithm for the computation of the augmented Reeb graph of piecewise linear scalar data. Such augmented Reeb graphs have a wide range of applications, including contour seeding and feature based segmentation. Our approach targets shared-memory multi-core workstations. For this, it completely revisits the optimal, but sequential, Reeb graph algorithm, which is capable of handing data in arbitrary dimension and with optimal time complexity. We take advantage of Fibonacci heaps to exploit the ST-Tree data structure through independent local propagations, while maintaining the optimal, linearithmic time complexity of the sequential reference algorithm. These independent propagations can be expressed using OpenMP tasks, hence benefiting in parallel from the dynamic load balancing of the task runtime while enabling us to increase the parallelism degree thanks to a dual sweep. We present performance results on triangulated surfaces and tetrahedral meshes. We provide comparisons to related work and show that our new algorithm results in superior time performance in practice, both in sequential and in parallel. An open-source C++ implementation is provided for reproducibility.

1. Introduction

The current growth in size and complexity of modern scientific data motivates the design of advanced data analysis techniques, in order to support interactive data exploration. For this purpose, topological methods [EH09, PTHT10, HLH*16] have now established themselves as key tools for the concise representation of the features of interest present in the data. In that context, notorious topological constructs include merge trees [BWT*11, SM17], contour trees [BR63, CSA00], Reeb graphs [Ree46, SKK91, PSBM07, BGSF08, TGSP09], or Morse-Smale complexes [DFFIM15]. These fundamental topology-based data structures enable a wide range of data analysis and visualization capabilities (going from feature representation [vKvOB*97, WBP07, TP12, SPCT18a, SPCT18b] to remeshing [VDL*17, TDN*12] or rendering [WDC*07]), which have been used and documented in a variety of scientific applications [BWT*11, COH*13, FGT16, GABCG*14, RWS*17].

Recently, the computational efficiency of topological data analysis techniques started to be challenged by the ever-increasing size and resolution of scientific data, although the individual computational power of CPU cores stagnated since the mid-2000s. This imbalance motivates the design of parallel versions of the existing algorithms of the topological data analysis arsenal. However, such a parallelization is challenging as most existing techniques are sequential in essence as they rely on global manipulations of the input data.

For the merge and contour trees, which are fundamental topology-based data structures in scalar field visualization, efficient algorithms have been proposed for their parallel computation [GFJT16, CWSA16, SM17, GFJT17, GFJT19]. Among those, some algorithms [GFJT16, GFJT17, GFJT19] even support the computation of *augmented* data-structures (i.e. where the arcs of the output trees are augmented with regular vertices). Such an augmentation is required to enable the full extent of applications of these tools, such as data segmentation or level set seeding for instance.

Regarding the Reeb graph [Ree46], which is a generalization of the contour tree to non-simply connected domains, which can potentially contain *loops* and which is, because of this, notoriously more challenging to compute, only one algorithm has been proposed for its parallel computation [HR18] and only for triangulated surfaces. To the best of our knowledge, no parallel algorithm exists for the computation of *augmented* Reeb graphs.

In this paper, we address this problem and introduce a novel algorithm for the fast computation of augmented Reeb graphs of piecewise linear scalar data. Such augmented Reeb graphs are generic and have a wide range of applications. This work shifts to the problem of Reeb graph computation

an overall strategy based on local propagations that we recently introduced for the problems of merge [GFJT17] and contour [GFJT19] tree computation. In particular, given that strategy, we detail how to revisit the optimal, but sequential, Reeb graph algorithm [Par13], which is capable of handling data in arbitrary dimension and with optimal time complexity. We detail in the present paper the modifications of the local propagation strategy that were required to shift to the Reeb graph problem, as well as original contributions specific to the Reeb graph computation. Specifically, our method re-formulates Reeb graph computation as a set of local tasks that are as independent as possible and that rely on Fibonacci heaps. This results in a parallel algorithm with the same optimal time complexity than the sequential reference one. Our implementation provides superior time performance in practice, in sequential as well as in parallel on shared-memory multi-core CPUs thanks to the OpenMP task runtime. We also provide an open-source C++ reference implementation of our approach for reproduction purposes.

1.1. Related work

The Reeb graph, a graph that contracts connected components of *level sets* on manifolds to points (Sec. 2.1), can be computed using several sequential algorithms. The first approach [SKK91] which has been proposed is based on a systematic *cut* of the mesh on all vertices. Since then, new cut-based approaches [PSF08, TGSP09, DN13, DN12] have been introduced, cutting the mesh only at specific vertices. A contour tree algorithm [CSA00] or a local propagation is typically used on the temporarily cut mesh. A final step stitches the mesh back on each cut in order to obtain the final Reeb graph. Because of the cuts, whose number and sizes are both proportional to the number of simplices in the input mesh, these approaches have a quadratic worst case complexity.

Furthermore, in 2007 was introduced an on-line algorithm [PSBM07] for Reeb graphs computations. This approach is able to operate in a streaming way, by processing the simplices of the 2-skeleton of the input mesh (its vertices, edges and triangles) in arbitrary order. A separate graph is used to reflect the neighborhood of the input simplices so when a new simplex is encountered the Reeb graph is updated locally to take this new simplex into account. When all simplices have been visited, the Reeb graph is complete. The final complexity of this algorithm is $O(|\sigma_0| \times |\sigma_1|)$, where $|\sigma_0|$ and $|\sigma_1|$ are respectively the numbers of vertices and edges of the input mesh.

The first algorithm [CMEH*03] to compute the Reeb graph using an ordered sweep of the data (similarly to merge tree algorithms) has been introduced in 2003. Using a sweep on the data set while explicitly maintaining the level set components, this approach only supports 2D data sets (data defined on triangulated surfaces). In 2009 was introduced another method [DN09], using a similar sweep for the mesh traversal as well as a dynamic graph data structure to maintain the level set components. This approach also works with 3D data sets (data defined on tetrahedral meshes). Parsa improved this work in 2013 [Par13] and presented the first algorithm able to compute the Reeb graph in any dimension with an optimal time complexity of $O(m \log m)$ steps where m is the size of the 2-skeleton (see Sec. 2.2). This approach is the basis of the new algorithm introduced in this paper.

Finally, a parallel algorithm [HR18] has been presented to compute Reeb graphs on triangulated surfaces, based on the Cylinder Map approach [DN12], with a scalar partitioning system similar to the one introduced in [GFJT16]. This type of partitioning introduces additional work for each supplementary thread. Moreover, results are only documented for the non-augmented graph, i.e. without the mapping from the mesh vertices to the arcs of the output data structure.

This work adapts to the Reeb graph problem an overall strategy based on local propagations with Fibonacci heaps [FT87] that we recently introduced for merge and contour trees [GFJT17, GFJT19]. This adaptation requires to completely revisit the data structures employed at the core of the approach to track connectivity. In particular, the Union-Find data structure (typically used for merge and contour trees [CSA00]) is no longer adapted to the Reeb graph problem (see Sec. 2.2), where more advanced connectivity tracking structures are required (supporting both online addition *and* removal, such as the ST-Tree [ST83]). An additional notable difference is that, in the merge and contour tree setting, the last propagation (monotone sequence of arcs called the *trunk* [GFJT17, GFJT19]) could be processed very efficiently in an embarrassingly parallel way. However, such a specific processing is no longer possible for the Reeb graph problem, where branching (and loops) can still be discovered in the last propagation. This motivated us to introduce a new strategy in the present work, which we call *dual sweep*, that partially compensates the absence of the trunk acceleration. Moreover, as detailed below, we also present further original contributions, such as an improved laziness mechanism for the update of the internal Reeb graph data structures.

1.2. Contributions

This paper makes the following contributions.

1. **A local algorithm based on Fibonacci heaps:** we adapt a recent strategy [GFJT17, GFJT19] based on local propagations with Fibonacci heaps from the contour tree setting to the Reeb graph problem. This results in the reformulation of the optimal sequential algorithm [Par13] into a set of independent, local treatments.
2. **An improved laziness mechanism for ST-Tree updates**: we improve the laziness mechanism presented by Parsa [Par13] by handling one ST-Tree data-structure per local propagation. This implies local hence smaller data-structures, which are independently and efficiently updated by the local propagations when they meet a saddle vertex. This results in a significant performance improvement on most data sets.
3. **Parallel augmented Reeb graphs**: we show how the task runtime environment of OpenMP can be used to implement a shared-memory parallel version of the above al-

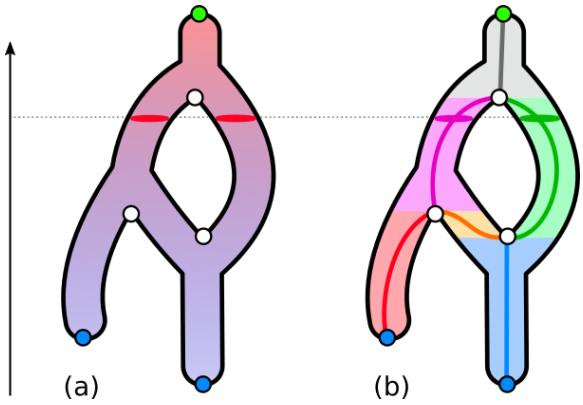

Figure 1: *Topology driven data segmentation. (a) Input scalar field f (color gradient), level-set (red) on the dotted line and critical points (blue: minimum, white: saddle, green: maximum). (b) Reeb graph of f and its corresponding segmentation (arcs and their pre-images by ϕ are shown with the same color).*

gorithm. Our approach benefits from the dynamic load balancing induced by the task runtime, without introducing extra work when new threads are added.

4. **Parallel dual sweep**: we present an improved version of the above parallel algorithm using two series of propagations to increase the parallelism degree. The first series traverses the mesh in increasing order of scalar values while the second one traverses it in decreasing order, until all vertices have been visited by at least one propagation.
5. **Implementation:** we provide an open-source C++ implementation of our approach for reproduction purposes, available as a module of the *Topology ToolKit* [TFL*17].

2. Preliminaries

The theoretical background of our work as well as an overview of our approach are presented in this section. It includes definitions that were adapted from [TFL*17,GFJT19] for self-completeness. We defer the reader to [EH09] for a thorough introduction to computational topology.

2.1. Background

Our algorithm takes as an input a scalar field f defined on a triangulation. Formally, f is a piecewise linear (PL) scalar field $f : \mathcal{M} \to \mathbb{R}$ defined on a PL manifold $\mathcal{M}$ of arbitrary dimension (Sec. 6 presents results on triangulated surfaces and tetrahedral meshes). In practice, f is given at the vertices of $\mathcal{M}$, such that no two vertices share the same f value (which can be obtained easily by symbolic perturbation [EH09]). Linear interpolation is used to extend the data values to any point of $\mathcal{M}$. Two key notions (*star* and *link*) are necessary to define traversals on $\mathcal{M}$. The set of all the simplices of $\mathcal{M}$ which contain a common simplex σ is called the *star* of σ, noted $St(\sigma)$. The set of all the faces of the simplices of $St(\sigma)$ which have an empty intersection with σ is called the *link* of σ, noted $Lk(\sigma)$. The vertices of the link of a vertex v can be classified without ambiguity as being above or below v with regard to f (as f is enforced to be injective on the vertices of $\mathcal{M}$ as mentioned above). This yields the notions of *lower* and *upper links*, respectively defined as $Lk^-(v) = \{\sigma \in Lk(v) \mid \forall u \in \sigma : f(u) < f(v)\}$ and $Lk^+(v) = \{\sigma \in Lk(v) \mid \forall u \in \sigma : f(u) > f(v)\}$. The vertices of $\mathcal{M}$ for which both $Lk^-(v)$ and $Lk^+(v)$ are simply connected are *regular*. The others are *critical*: v is a *minimum* if $Lk^-(v) = \emptyset$ (blue dots, Fig. 1), a *maximum* if $Lk^+(v) = \emptyset$ (green dots, Fig. 1) and a *saddle* otherwise (white, Fig. 1).

For visualization and data segmentation, three key geometrical objects are of particular importance, namely the *level set* and the *sub-* and *sur-level set*. The level set $f^{-1}(i)$ is the set of points of $\mathcal{M}$ which all share the same f value i: $f^{-1}(i) = \{p \in \mathcal{M} \mid f(p) = i\}$ (Fig. 1). The *sub-* and *sur-level sets* are defined similarly, by trading the equality for an inequality, respectively: $f^{-1}_{-\infty}(i) = \{p \in \mathcal{M} \mid f(p) < i\}$ and $f^{-1}_{+\infty}(i) = \{p \in \mathcal{M} \mid f(p) > i\}$.

The Reeb graph is a fundamental topological data structure which tracks the evolution of the connectivity of the level sets of f. It is a simplicial complex of dimension 1 (Fig. 1), noted $\mathcal{R}(f)$, which is defined as the quotient space $\mathcal{R}(f) = \mathcal{M}/\sim$ by the equivalence relation $p_1 \sim p_2$ which holds iff $p_2 \in f^{-1}(f(p_1))_{p_1}$, where $f^{-1}(f(p_1))_{p_1}$ is the connected component of $f^{-1}(f(p_1))$ which contains p_1. Let $\phi : \mathcal{M} \to \mathcal{R}(f)$ be the *segmentation map* of $\mathcal{R}(f)$. It maps each point of $\mathcal{M}$ to its equivalence class in $\mathcal{R}(f)$. As described by Reeb [Ree46], the pre-image of any vertex of $\mathcal{R}(f)$ by ϕ contains a single critical point of f (since f is injective on the vertices of $\mathcal{M}$, it is injective as well by construction on the subset of critical vertices of $\mathcal{M}$). Then valence-1 vertices of $\mathcal{R}(f)$ correspond either to a minimum or a maximum of f, while the remaining vertices, yielding branching in $\mathcal{R}(f)$, correspond to saddles of f, where level set components join or split. In practice, the pre-image ϕ^{-1} is particularly useful for data segmentation purposes (Fig. 1) as the pre-image of each arc of $\mathcal{R}(f)$ is connected by construction. In our data-structures, the pre-image of the segmentation map ϕ is explicitly stored along each arc of $\mathcal{R}(f)$, by storing the list of regular vertices which map to it, hence effectively *augmenting* the arcs of the Reeb graph with the corresponding segmentation (Fig. 1).

2.2. Reference computation with dynamic ST-Trees

The sequential reference algorithm for augmented Reeb graphs [Par13] computes its output incrementally by sweeping the data using a dynamic graph data structure, which represents a level set sweeping continuously the domain. In the following, we consider that an edge of $\mathcal{M}$ *starts* at its vertex of lower scalar value and *ends* at the one with higher value.

Parsa's algorithm is based on a global view of the data and starts by a sort of the vertices of the mesh by scalar value. Then, vertices are visited in increasing order of scalar value. At each vertex v visited by the growth procedure, the

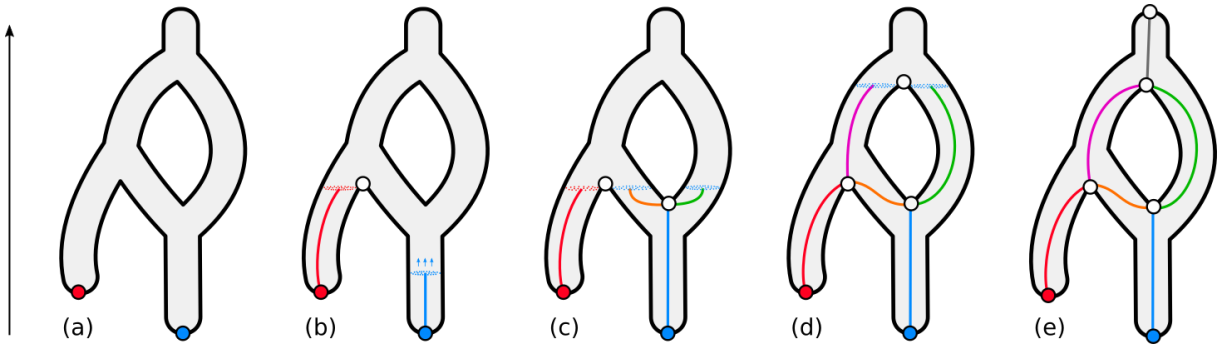

Figure 2: *Overview of our augmented Reeb graph algorithm based on Fibonacci heaps and dynamic ST-Trees on a toy elevation example. (a) The local minima of f (corresponding to leaves of $\mathcal{R}(f)$) are extracted (red and blue points). (b) The arc σ_m of each minimum is grown independently along with its segmentation. These independents growths are achieved by progressively growing the connected components of sub-level sets created at m, for increasing f values, and by maintaining at each step a priority queue θ_m, implemented with a Fibonacci heap, which stores vertex candidates for the next iteration (disks colored according to their starting minimum). These growths stop at join saddles as shown with the red one in (b). (c) The blue growth on the right has visited a split saddle and is now processing two arcs (orange and green) thanks to the dynamic graph implemented with a ST-Tree data structure. (d) The blue growth is the last one to reach the left saddle and is thus kept active. Here, the red propagation merges with the blue one. The corresponding priority queues are merged in constant time thanks to the Fibonacci heap. (d) The last growth processes two arcs around the topological handle. (e) The augmented Reeb graph is complete.*

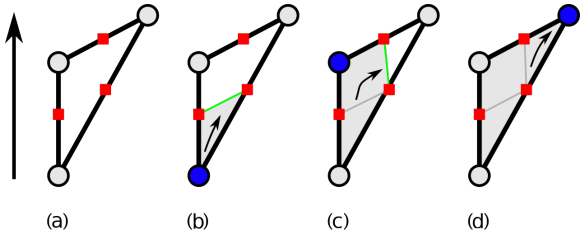

Figure 3: *Evolution of the dynamic graph (red nodes and green arcs) while a sweep is performed on a single triangle with an elevation scalar field. The vertex being currently processed is shown in blue.*

preimage $f^{-1}(v)$ is updated to make the level set grow from the scalar value just below $f(v)$ to the one just above $f(v)$. This preimage can be abstracted into a graph G_r, named the *preimage graph* and used to identify the connected component of level set to which each vertex belongs. This preimage graph is implemented as an ST-Tree data structure [ST83]: nodes of G_r are edges of $\mathcal{M}$ intersecting the preimage, and arcs of G_r are triangles of $\mathcal{M}$ contributing to the preimage (connecting edges, hence nodes together).

The update of the preimage graph is done using triangles incident to v. When v is the lowest vertex of the triangle t_v (Fig. 3 (b)), the two edges of t_v starting at v are linked by an arc in G_r to reflect the level set entering t_v. When v is the middle vertex in the triangle t_v (Fig. 3 (c)), the arc in G_r between the two lowest edges is removed and a new arc is added between the two highest edges. Finally, when v is the highest vertex of the triangle t_v (Fig. 3 (d)), the level set is growing out of t_v and the arc of G_r remaining between the two edges of the triangle ending at v is removed. Unlike the Union-Find data-structure, which can dynamically track connected components in a graph upon arc insertions (and which is used at the basis of most merge tree algorithms, to model sub-level set components), the ST-Trees can dynamically track connected components upon both arc insertions *and* removals. Thus, ST-Trees can efficiently track the connected components of G_r (which models the current level set component) at each iteration of the propagation. In particular, the operations on the ST-tree (connected component query, arc insertion, arc removal) are performed in at most logarithmic time with regard to its size, resulting to an overall time complexity of $O(m\ log\ m)$ steps, where m is the size of the 2-skeleton of $\mathcal{M}$ (vertices, edges and triangles).

The output Reeb graph $\mathcal{R}(f)$ is updated at each vertex using the preimage components on its neighborhood. Before the dynamic graph is updated at $f(v)$, the connected components are retrieved using edges ending at v. If more than one component is retrieved, v is a join saddle and the corresponding arcs of $\mathcal{R}(f)$ are closed. After the dynamic graph update, the connected components are retrieved once again, using edges starting at v. If more than one component is retrieved, v is a split saddle and each component leads to the creation of a new arc in $\mathcal{R}(f)$. If no edge starts at v, the vertex is a local maximum and the corresponding arc is closed. Finally, if the vertex v is regular in $\mathcal{R}(f)$ (both lower and upper components have one connected component), v is simply added to its corresponding arc in $\mathcal{R}(f)$.

2.3. Overview

Fig. 2 presents an overview of our approach for the computation of augmented Reeb graphs. Our algorithm revisits the sequential sweep approach of Parsa [Par13], described in the previous section, but performs independent local growths for the mesh traversal. The vertices of $\mathcal{M}$ are first visited to extract the list of minima of f (Fig. 2 (a), Sec. 3.1). Then, a second procedure is launched: for each local minimum at vertex v, a local growth in charge of constructing the augmented arc attached to v is executed, based on a sorted breadth-first search traversal implemented with a Fibonacci heap [FT87] (Fig. 2 (b), Sec. 3.2). A dynamic graph data structure corresponding to the growing level set components and implemented as an ST-Tree data structure [ST83] is maintained during the growth. As described in Sec. 2.2, this dynamic graph allows to track both join and split saddles and to update the Reeb graph data structure accordingly on the fly (Fig. 2 (b) to (e)). To ensure that the lower link of any processed vertex has always been visited, only the last growth reaching a join saddle can continue the processing, after having processed the saddle as described in Sec. 3.3.

Each iteration of the local propagations performs only log-time operations on the ST-trees [ST83], as well as on the Fibonacci heaps [FT87]. Since these heaps can merge in constant time, this results in an overall time complexity of $O(m\ log\ m)$ steps, where m is the size of the 2-skeleton of $\mathcal{M}$ (vertices, edges and triangles), which is identical to the optimal but sequential reference algorithm [Par13].

3. Local propagations for Reeb graph computations

We present here our new algorithm for the computation of augmented Reeb graphs using local growths. The procedures corresponding to the different steps of the algorithm are described, along with specific treatments and optimizations. In particular, we describe how an overall strategy based on local propagations needs to be adapted from the contour tree setting [GFJT17, GFJT19] to the Reeb graph problem.

3.1. Leaf search

First, we construct the lower link $Lk^-(v)$ of each vertex $v \in \mathcal{M}$. This detects the minima (empty lower link $Lk^-(v)$), upon which the growth procedure described in the next subsection is started. This very first step is identical to the leaf search procedure in the contour tree setting [GFJT19].

3.2. Local growth

Given a local minimum m, a local growth procedure, named *local growth* starting at m is called in order to progressively sweep all contiguous equivalence classes (Sec. 2.1) between m and the next join saddle s. In other words, this growth procedure will sweep the connected components of sub-level set initiated in m while maintaining a growing level set to construct the corresponding arcs of $\mathcal{R}(f)$ on the fly.

The sweep on the connected components of sub-level set is achieved thanks to an ordered breadth-first search traversal of the vertices of $\mathcal{M}$ started in m. During this sweep, for each new vertex v, the neighbors of v (not already visited) are added to a priority queue $\mathcal{Q}_m$ (unless already present in it). Then, the next vertex v' to process is chosen as the minimizer of f in $\mathcal{Q}_m$. We iterate the process until reaching a join saddle s (Sec. 3.3). Breadth-first search traversals grow connected components: this ensures that, for each vertex v, all the edges of $\mathcal{M}$ connecting visited vertices to visited candidates (stored in $\mathcal{Q}_m$) are indeed crossed by the component of $f^{-1}(f(v))$ which contains v. Hence, this sorted traversal indeed maintains connected components of level sets at each iteration of the local sweep. In practice the priority queues are implemented as Fibonacci heaps.

During the sweep, the preimage graph G_r is maintained on each vertex using the same procedure as the reference algorithm described in Sec. 2.2. In practice this preimage graph is implemented as a ST-Tree data structure [ST83]. This is a notable difference with the contour tree setting which only requires to maintain a simpler Union-Find data structure, as further detailed in the next two sub-sections.

3.3. Saddle vertex handling

Join saddles. If the number of connected components of dynamic graph in edges ending at v is greater than 1 before v has been processed, v is a join saddle and the current growth stops (without updating the preimage graph). Only the last local growth reaching the join saddle can process it and continue. The last growth detection can be done by looking at edges in the lower star of a join saddle s: if all these edges have already been visited, the current growth is the last one visiting s and is in charge of carrying on the computation. This situation is illustrated in Fig. 4. The arcs of the Reeb graph in the lower star of s are retrieved using the dynamic graph G_r and closed at s like in the reference algorithm (red and orange arcs in Fig. 4 (a)). Then, the dynamic graph is updated on s. Priority queues of local growths stopped at s are merged with the current one before a new growth, initiated with the resulting priority queue, is run. This merge is done in constant time thanks to the Fibonacci heap. In Fig. 4, we can see the red priority queue merging with the blue one at the join saddle.

Split saddles. If the number of connected components of dynamic graph in edges starting at v is greater than 1 after v has been processed, v is a split saddle. Like in the reference algorithm, the arc ending here is closed (if v is not also a join saddle) and a new arc is created for each component of dynamic graph in the upper star of v. The current local growth continues the processing, handling both arcs. Fig. 4 (a) shows an example of a local growth that encountered a split saddle (right white circle): the orange and green arcs have been created at the split saddle and the same growth (blue) handles both arcs.

3.4. Laziness mechanism for preimage graph

In the reference algorithm [Par13], a "lazy insertion" optimization is described. In order to make the implementation

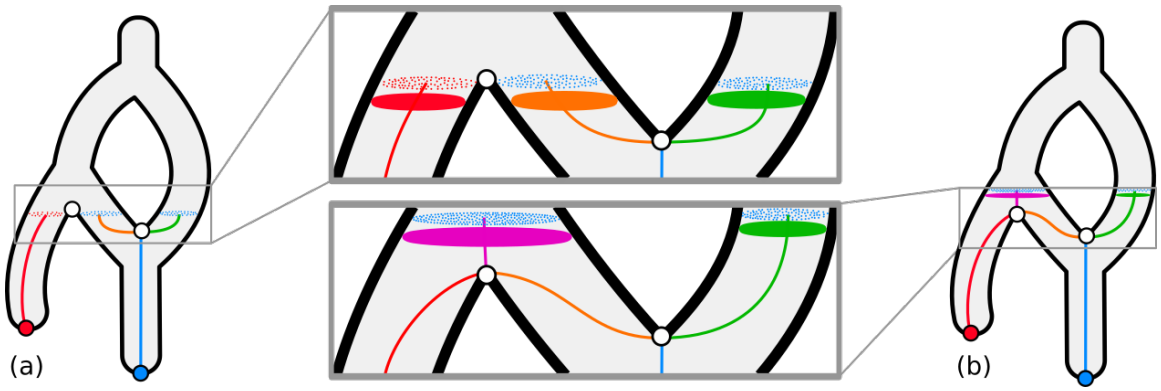

Figure 4: *On a 2D toy elevation example, priority queues (colored dots) and dynamic graphs (plain disks) in the proximity of critical points are highlighted. First, on the left and on the top, the right growth (in blue) has passed a split saddle. The blue priority queue contains candidates vertices from both sides of the split (blue dots) and handles two connected components of the preimage graph (orange and green disks). Second, on the right and on the bottom, the left join saddle has been processed. The red and blue priority queues have merged in constant time and a single growth is remaining, handling two arcs (purple and green). The red and orange components of preimage graphs have also merged at the join saddle.*

faster, additions and deletions of arcs in the dynamic graph G_r are stored in a *history list*, which serves as a record of operations. When a critical vertex v is encountered, each arc which has been both marked as added and deleted from the history list is discarded and only the remaining operations are applied to G_r. This allows to grow at once the level set modeled by G_r up to the value $f(v)$, without having to perform the in-between operations which do not change the connectivity of G_r. This optimization requires to extract all critical vertices in a pre-processing step, which can be done efficiently by counting the number of connected components of lower and upper links of each vertex (cf. Sec. 2.1).

This optimization is further improved in our work by breaking this global history list of operations into local ones. A naive way would be to have one history list per local growth. This way, when a saddle vertex s is encountered, instead of updating the preimage graph on the whole level set $f(s)$ only the sub-level set component containing s is updated. However, we found out that we can improve this mechanism by subdividing the list of operations further, having one history list per arc of the output graph $\mathcal{R}(f)$. This way, when a local growth encounters a saddle vertex s, only the connected component of level set containing s is updated, which corresponds to the minimal amount of operations to maintain a valid preimage graph.

4. Task-based parallel Reeb graphs

In order to implement our new algorithm for the construction of augmented Reeb graphs, we rely on the task parallel programming paradigm, available e.g. in OpenMP [Ope15], Intel Threading Building Blocks [Phe08], Intel Cilk Plus [Int], etc. The programmer only handles tasks, not threads. These tasks are then executed concurrently and asynchronously by the runtime on the available threads, whose number is fixed at execution time by the user. Our new algorithm being based on local growths, and the growths starting from minima being independent, the task-based parallelization is straightforward. However, local synchronisations are required on join saddles as the task corresponding to a join saddle growth can only start its execution after all of the lower link of its saddle has been visited: this will require task synchronizations. Also,we emphasize that our new algorithm based on local growths does not introduce any supplementary computation in parallel, and that the dynamic load balancing of the task runtime will help improving parallel speedups in practice. We rely here on OpenMP tasks [Ope15], but other task environments could also be used with only minor adjustments.

In the following, we describe how we have parallelized the different steps of our algorithms. It can be first noticed that our implementation starts with a parallel global sort of all the vertices according to their scalar value (using the Standard Template Library). Once this pre-sort is finished, all vertex comparisons can be done by only comparing their position index in the sorted order: this is faster than having to access the scalar values, and this also makes this comparison independent of the data types used to represent scalar values in the input.

4.1. Leaf search and saddle extraction

The extraction of the lower link $Lk^-(v)$ of each vertex $v \in \mathcal{M}$ being a local operation, this step is embarrassingly parallel and the correponding loop can be straightforwardly parallelized with OpenMP. This step is identical to the contour tree setting [GFJT19].

When the optimization described Sec. 3.4 is enabled, both the lower and upper links of v are extracted in order to also detect saddle vertices. We recall that some vertices may be locally saddles, but do not imply changes in the number of connected components of level set and so end up being regular nodes in the output Reeb graph.

4.2. Local growth

All local growths initiated at a leaf (minimum) are implemented as tasks, starting at their previously extracted leaf. Each growth (task) spreads locally and independently, until it finds a join saddle, by managing its own Fibonacci heap, as well as its own connected components of dynamic graph so that the update on each vertex does not involve any data race. Similarly, the list of edge deletions and insertions used for the laziness optimization (Sec. 3.4) only impacts the preimage graph on components local to the current growth and so no data race among concurrent growths may occur.

4.3. Saddle vertex handling

The saddle vertex processing presented in Sec. 3.3 can be implemented in parallel with tasks. As discussed in Sec. 4.1, saddles of f are first extracted in a parallel pre-processing step. However, not all of these saddles will yield some branching in the output graph. Therefore, *join* and *split* saddles (which respectively yield downward and upward forks in the output graph, Sec. 2.2) must be distinguished among this initial set of saddles.

Split saddles can be identified on-the-fly during the growth, exactly as in the original reference algorithm (Sec. 2.2). Join saddles, however, require more attention. When a local growth reaches a saddle s (red growth in Fig. 4(a)), to determine if s is a join saddle or not, we use the *saddle stopping condition* described by Gueunet et al. [GFJT17, GFJT19]. In particular, this condition states that if a local growth g reaches a saddle s for which some of the vertices of $Lk^{-}(s)$ have not been visited before by the same growth g, then s is a join saddle. Indeed, since each growth reconstructs a connected component of sub-level set, such a configuration corresponds to points where several components of sub-level set merge with each other (hence the appearance of a join saddle).

As described in Sec. 3.3, we have to detect the last task reaching a join saddle. For this, we rely only on lightweight synchronizations (OpenMP atomic operations) as detailed in [GFJT17, GFJT19]. The processing done by the last task reaching the join saddle (Sec. 3.3) only involves already computed information. Arcs are closed, the preimage graph updated and the Fibonacci heaps merged sequentially by the last task: no task synchronization is required here.

5. Parallel dual sweep

In the parallel algorithm described Sec. 4, the number of independent growths (i.e. the number of tasks) corresponds initially to the number of minima and strictly decreases as join saddles are encountered, eventually reaching one. As a consequence, a substantial part of the data set (at least all the region above the highest join saddle) may be processed sequentially, using a single task and undermining parallel performance. In order to reduce this effect, we propose a parallel dual sweep algorithm traversing the data set simultaneously from minima (in increasing order of scalar value) and from maxima (in decreasing order of scalar value). These

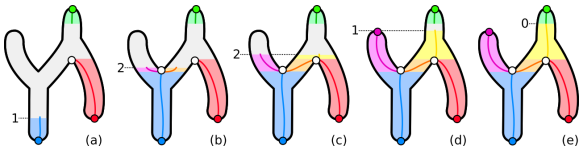

Figure 5: *Evolution of the number of active arcs for the local propagation initiated at the blue minimum. The green arc is computed by a decreasing growth and is only here to show an example of arcs merging. (a): initially, there is one active arc (blue). (b): after the join, there are two arcs managed by this growth (purple and orange). (c): at the join, one arc is closed (orange) and one opened (yellow); the number of active arcs remains two (purple and yellow arcs). (d): an arc (purple) is closed at a maximum and only one arc (yellow) remains active. (e): the last arc (yellow) of the growth merges in an incoming arc (green), the growth has no more active arc and stops.*

two sweeps use local growths as described previously and stop when they meet each other. Sweeping the data set using both minima and maxima leads to the creation of a higher number of independent growths and allows to process with a higher parallelism degree areas of the mesh that would have been processed by a few number of task otherwise. We describe in the following how to adapt the algorithm presented in Sec. 4 to this dual sweep strategy. Note that this dual sweep is a completely original procedure which is another notable difference with the contour tree setting [GFJT19].

5.1. Leaf search

In order to launch growths from minima and maxima, both are extracted in a single pass using the lower and upper links of each vertex. Local growths initiated at maxima are symmetric to those starting at minima and traverse the data set in decreasing order of scalar value. In practice, this step is also in charge of extracting all saddles, as required by the laziness mechanism described in Sec. 3.4.

5.2. Dual growth meeting points

The growths initiated at minima and those initiated at maxima will eventually encounter each other. In the following, we describe how to detect when two growths are crossing and how to merge the corresponding arcs.

Growths mark vertices they visit in two arrays: one for growths sweeping in increasing order of scalar value and one for growths sweeping in decreasing order. This information is used by a local growth g to check if its current vertex has not already been visited by an opposite one g'. If so, the current arc is marked as merged with the incoming arc from the opposite growth g' (see Fig. 5 (e)), and the current growth g stops processing this arc. A post-processing step described in Sec. 5.4 is in charge of computing the final arc, resulting from this merge. The candidate vertices in $\mathcal{Q}_m$ corresponding to a merged arc can be discarded from the remainder of the propagation g (which may itself continue to propagate

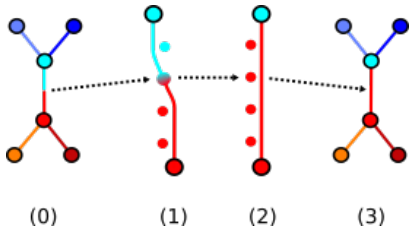

Figure 6: *Two halves of an arc computed by opposite growths are merged into a single arc. Regular vertices are updated accordingly. The blue colors are used for arcs computed by downward growths initiated at maxima, red colors for upward growths initiated at minima.*

other arcs in other parts of $\mathcal{M}$). Atomic operations are used to visit (and check) vertices in order to avoid data races.

During the traversal, each growth keeps a local counter of the number of arcs it handles (see Fig. 5). This counter is increased when new arcs are created (Fig. 5 (b)) and decreased when arcs are closed or merged (Fig. 5 (d) and (e)). For the last growth continuing at a join saddle s, its counter is incremented by the sum of remaining active arcs associated with all the growths which merged at s. If this counter reaches 0 during the computation, the current growth has no more arc to process and can stop (Fig. 5 (e)).

5.3. Saddle vertex handling

At critical vertices, nodes of the Reeb graph are created using a global lock (implemented as a critical section in OpenMP) so that a given node cannot be created simultaneously by an upward growth and a downward one. As detailed in Sec. 6, this global lock does not have a significant impact on execution times in practice. If a growth g tries to create an already existing node n, this means g is crossing an opposite growth g' (which created the node first). Thus, the current growth g does not propagate further its arc(s) ending in n.

5.4. Post-processing for merged arcs

When the dual sweep is performed in parallel, it is possible for two arcs to merge in the middle of their construction (like in Fig. 5). A post processing step is in charge of computing the final arc from these two parts and to update the map ϕ of regular vertices accordingly (see Fig. 6). In practice, this step takes a negligible time in our computations (less than 5% of the total time).

6. Results

For the following performance results, we rely on a workstation equipped with two Intel Xeon E5-2630 v3 CPUs (2.4 GHz, 8 CPU cores and 16 hardware threads per CPU), 64 GB of RAM, `g++` version 7.3.0 and OpenMP 4.5. Unless stated otherwise, we will use 32 threads on 16 cores. The implementation of our new algorithm (called *Fibonacci Task-based Reeb graph*, or FTR) is built as a C++ TTK [TFL*17] module (provided as additional material). We have used the Boost implementation of the Fibonacci heap [FT87], and our own ST-Tree [ST83] implementation for the dynamic graph.

Our tests have been performed using ten data sets, five triangulated surfaces (Fig. 7) and five tetrahedral meshes. For all of them, the considered scalar field f is a height function, except for *Mechanical*, where the considered scalar data is the norm of a flow velocity field. The first surface, *Spring*, is the boundary surface of the first volume, *Spring3D*. It is made of four connected components and its output Reeb graph has 24 leaves, each leading to a large arc. The *Eiffel* data set is a synthetic, open surface produced by a graphical designer and counting many disconnected components. Most of these data sets have been subdivided in order to obtain significant execution times on our setup.

6.1. Performance analysis

Tab. 1 details the execution times and speedups of FTR on our data sets. First, the sequential execution times approximately follow a linearithmic evolution. The observed variations from the theoretical complexity are common to most Reeb graph algorithms, which tend to be output sensitive. This behavior is greatly accentuated by our ST-tree lazy update mechanism, which only triggers updates at critical points. Regarding parallel executions, the embarrassingly parallel critical point search offers very good speedups (averaging at 18.4x). The key step for parallel performance is the Sweep step performing the independents local growths. On all our data sets this step is indeed the most time-consuming in parallel and offers an average speedup of 5.3x. The almost ideal speedups of the spring data set (13.4x and 14.3x with 32 threads on 16 cores) can be used as an evidence that neither the critical section on node creation nor the atomic updates on visited vertices prevent good speedups.

Fig. 8 presents the parallel scaling curves of our FTR implementation. First, these curves are monotonically increasing. This means that increasing the number of threads does not imply an increase in execution time, and hence illustrates that our algorithm does not yield extra work when run in parallel. This also justifies our default choice of using 2 threads per core (instead of 1) since this results in greater speedups for all data sets but one (gray band, Fig. 8). We emphasize that the maximum number of tasks created for the local growths is equal to the number of leaves in the output graph, which implies that the speedups of the sweep step is bounded by this number of leaves. In practice, tasks merge together at saddles and the number of available tasks quickly decreases. This translates in reduced parallel efficiencies: our speedups quickly reach 2 but seems to reach a plateau around 4 for most data sets (further details in Sec. 6.3).

In parallel, the dynamic load balancing of the task runtime can lead to different schedulings between multiple executions over a given data set. However, as already demonstrated in the case of the merge tree in [GFJT17], this kind of task-based approaches offers consistent computation times between executions. In our experiments, the average stan-

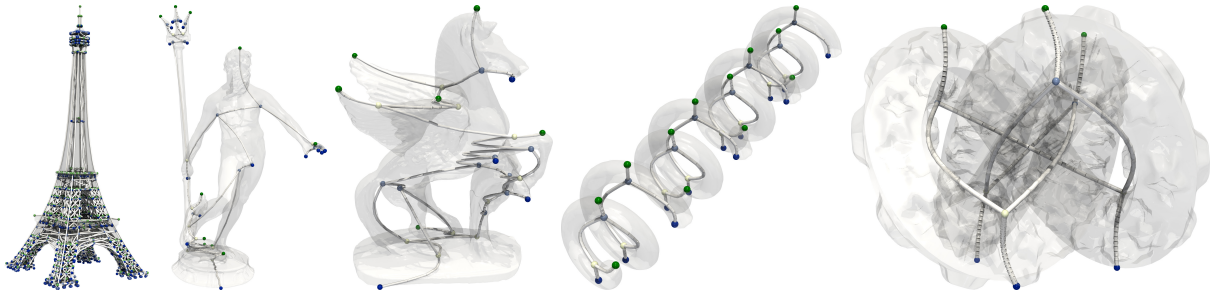

Figure 7: *Gallery of Reeb graphs computed with our algorithm. From left to right: Eiffel, Neptune, Pegasus, Spring, Starknot.*

Table 1: *Memory footprint and running times (in seconds) of the different steps of FTR on our data sets.* $|\sigma_2|$ *is the number of triangles in* $\mathcal{M}$ *and* $|\mathcal{R}(f)|$ *the number of arcs in the output Reeb graph. These executions use the dual sweep strategy. File sizes are used to estimate the input and output footprints.* Peak *reports an estimation of the maximum runtime memory footprint.*

Dimension	$\|\sigma_2\|$	Data set	$\|\mathcal{R}(f)\|$	Memory footprint (MB) Input	Output	Peak	Sequential Overall	Parallel (32 threads on 16 cores) Sort	Crit. search	Sweep	Post. proc	Overall	Speedup
2	1,728k	Spring	44	189	18	1,795	7.40	0.05	0.12	0.37	0.01	0.55	13.45
2	6,936k	Eiffel	372,944	752	79	7,316	24.00	0.29	0.59	2.88	0.32	4.08	5.88
2	8,009k	Pegasus	571	876	83	8,333	35.67	0.36	0.70	12.79	0.02	13.87	2.57
2	9,552k	Neptune	486	1,045	100	9,951	42.66	0.31	0.77	14.68	0.03	15.79	2.70
2	9,600k	StarKnot	2,835	1,050	100	10,014	44.96	0.36	0.62	6.65	0.02	7.65	5.88
3	23,098k	Spring3D	44	608	26	9,465	58.87	0.42	0.75	2.94	0.01	4.12	14.29
3	32,002k	Elephant	48	897	30	12,618	73.24	0.39	0.78	19.14	0.02	20.33	3.60
3	48,413k	Hand	185	1,362	45	18,935	113.51	0.57	1.37	28.10	0.02	30.06	3.78
3	60,532k	Skull	30	1.700	56	23,763	169.87	0.79	1.69	52.58	0.84	55.90	3.04
3	71,486k	Mechanical	180	2.091	69	28,605	211.14	0.84	1.76	38.16	0.02	40.78	5.18

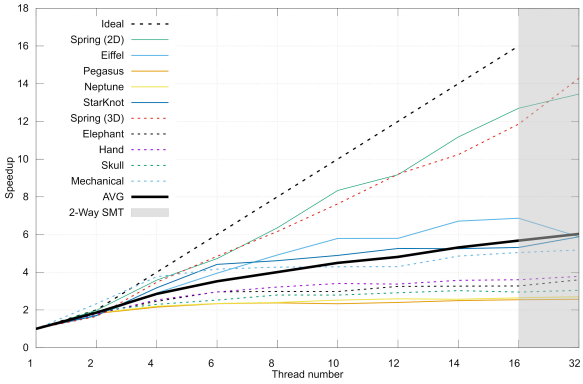

Figure 8: *FTR scalability on our various data sets. The gray area denotes using 2 threads per core.*

Table 2: *Left: execution* times *(in seconds) of the sweep procedure using no laziness, compared to our approach (one list per arc). Right: number of* rotations *made by the ST-Trees using the reference algorithm [Par13], compared to our approach. These runs use single-sweep sequential executions.*

Data set	Time (s) no laziness	Ours	Gain	Rotations [Par13]	Ours	Gain
Spring	40.14	4.40	9.12	8.52e6	3.34e5	25.52
Eiffel	51.51	15.58	3.31	3.95e7	6.82e7	0.58
Pegasus	2,998.73	25.70	116.68	5.08e7	8.98e6	5.65
Neptune	4,744.04	29.48	160.92	6.19e7	1.51e7	4.10
StarKnot	1,945.30	32.35	60.13	1.70e8	3.10e7	5.49
Spring3D	36.30	36.97	0.98	1.68e8	5.81e6	28.98
Elephant	51.21	51.22	1.00	6.75e8	1.83e7	36.82
Hand	82.67	82.54	1.00	1.50e9	6.67e7	22.43
Skull	233.71	152.85	1.53	2.88e9	5.96e7	48.35
Mechanical	189.18	162.52	1.16	2.95e9	6.24e7	47.30

Table 3: *Comparison of execution times (in seconds) between the single and dual sweep strategies (presented respectively in sections 4 and 5) during parallel executions.*

Data set	Single sweep	Dual sweep	Speedup
Spring	0.90	0.37	2.43
Eiffel	2.45	2.88	0.85
Pegasus	26.74	12.79	2.09
Neptune	29.88	14.68	2.04
StarKnot	18.44	6.65	2.77
Spring3D	6.79	2.94	2.31
Elephant	43.89	19.14	2.29
Hand	53.76	28.10	1.91
Skull	103.46	52.58	1.97
Mechanical	90.51	38.16	2.37

dard deviation obtained using 10 runs on our data sets is 0.6 second for a global average time of 19.3 seconds.

In order to evaluate the performance gains obtained by our improved laziness mechanism for the preimage graph update (Sec. 3.4), we present in Tab. 2 execution times with and without this optimization, using single-sweep sequential executions. This optimization is especially efficient on 2D data sets, improving execution times by a factor up to 160.92x. On our 3D data sets however, this optimization seems to have less impact with an average gain of 1.13x. Tab. 2 also provides the number of internal rotations performed by the ST-Tree data structures upon their updates, for both the reference algorithm [Par13] and our approach. These internal rotations are performed in practice to decrease the depth of the ST-Trees and thus improve their efficiency. Overall, this number of operations is a relevant indicator of the amount of work performed by the ST-Trees. Our improved laziness (using one history list per arc) yields in average 22 times less rotations than the reference algorithm [Par13].

Table 4: *Reeb graph computation times (in seconds) and ratios between the reference algorithm [Par13] and our approach (FTR) using 1 and 32 threads (on 16 cores).*

	Times			FTR Speedups	
Data set	[Par13]	FTR (1)	FTR (32)	(1)	(32)
Spring	20.80	7.40	0.55	2.81	37.82
Eiffel	59.98	24.00	4.08	2.50	14.70
Pegasus	65.18	35.67	13.87	1.83	4.70
Neptune	71.96	42.66	15.79	1.69	4.56
StarKnot	75.41	44.96	7.65	1.68	9.86
Spring3D	84.73	58.87	4.12	1.44	20.56
Elephant	97.23	73.24	20.33	1.33	4.78
Hand	156.48	113.51	30.06	1.38	5.21
Skull	221.96	169.87	55.9	1.31	3.97
Mechanical	217.53	211.14	40.78	1.03	5.33

Our dual sweep strategy (Sec. 5) aims at increasing the parallelism degree, and so at improving the parallel efficiencies. The gains obtained by this dual sweep strategy for a parallel execution are presented in Tab. 3. Complete execution times are reported, as the dual sweep method impacts both the critical point extraction and the sweep steps. Starting from both minima and maxima leads to a significantly higher number of tasks and allows to process efficiently in parallel regions of the mesh that would have been processed by a low number of tasks using the single sweep method. The dual sweep mechanism hence leads to an average speedup of 2.1x over the single sweep version.

Note that the dual sweep approach implies that upward and downward growths can cross each others, visiting some vertices of the mesh twice (along connected components of level sets). This situation only occurs on a fraction of the output arcs and in practice the work overhead is negligible: the average number of vertices visited twice is about 0.4% of the total number of vertices in average in our test cases.

6.2. Comparisons

Tab. 4 provides a run time comparison between our approach and the sequential reference algorithm [Par13], with an implementation kindly provided by its author. Note that, the latter implementation only produces a non-augmented graph on its output, without the segmentation information. Moreover, the memory footprint of this implementation is larger than ours. Internally, it pre-sorts the simplices of the 2-skeleton in arrays (vertices of each edge, edges of each triangle, adjacent triangles and edges of each vertex). These arrays are used during the sweep to efficiently retrieve pre-sorted vertices upon adjacency queries (hence speeding up the computation). We decided not to implement such a speedup mechanism as we wanted our implementation to maintain a reasonable memory footprint (Tab. 1), to improve its practical usability. Despite this, FTR is in average 1.56 time faster in sequential, thanks to our improved laziness mechanism. Using 32 threads on 16 cores, our implementation leads to substantial improvements, speeding up the computation by a factor of 11.14x in average.

6.3. Limitations

During the sweep procedure, the number of available tasks monotonically decreases, as local propagations merge at join saddles. When the number of remaining tasks eventually becomes lower than the number of available cores (Fig. 9), we say that the computation enters a *suboptimal section*, where the computational power of our multi-core CPU is not fully exploited, hence undermining the parallel efficiency of the approach. This drawback was mitigated in the merge/contour tree setting [GFJT17, GFJT19] thanks to the *trunk* procedure (see Sec. 1.1), which however is inapplicable to the Reeb graph problem. This motivated us to design our dual sweep strategy, which partially addresses this issue, by maintaining the number of active tasks above 2 for the vast majority of the computation. This is the reason why our approach achieves almost ideal speedups when using only two threads (Fig. 8). For larger numbers of threads, depending on the topological complexity of the data, the suboptimal sections start at various points in the computation times, resulting in a high variability in parallel efficiency overall (from 16% to 90%, average: 38%).

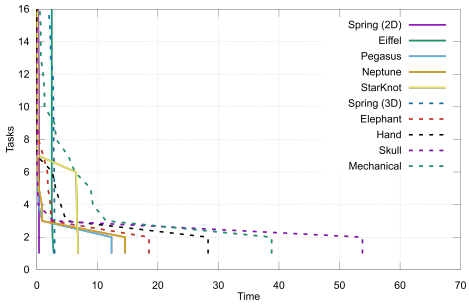

Figure 9: *Number of tasks through time (cropped at* 16 *to emphasize the suboptimal section on our* 16*-core setup).*

7. Conclusion

We have presented the first parallel algorithm to compute the augmented Reeb graph on shared-memory multi-core architectures. For this, we have rewritten the optimal, but sequential, algorithm [Par13] to design a new algorithm based on independent local growths with Fibonacci heaps. The local nature of our approach enabled us to improve the lazy update mechanism of the ST-Trees (used to track level set components), which results in less work than the reference algorithm and improved sequential performances in practice. The design of our algorithm is conducive to parallelism and we have presented an efficient task-based parallel version. We have also presented a "dual sweep" strategy, which guarantees good speedups for low numbers of threads. Finally, we also provide an open-source OpenMP/C++ reference implementation of our approach (available in the *Topology ToolKit* [TFL*17]), which is, to our knowledge, the only documented parallel implementation to compute the augmented Reeb graph. In the future, we plan to consider extensions of our approach to address distributed computations.

Acknowledgments

This work is partially supported by the European Commission grant H2020-FETHPC-2017 "VESTEC" (ref. 800904) and by the ANRT, in the framework of the CIFRE partnership between Sorbonne Université and Kitware SAS CIFRE (reference #2015/1039).

References

[BGSF08] BIASOTTI S., GIORGIO D., SPAGNUOLO M., FALCIDIENO B.: Reeb graphs for shape analysis and applications. *TCS* (2008). 1

[BR63] BOYELL R. L., RUSTON H.: Hybrid techniques for real-time radar simulation. In *Proc. of the IEEE F.J.C.C.* (1963). 1

[BWT*11] BREMER P., WEBER G., TIERNY J., PASCUCCI V., DAY M., BELL J.: Interactive exploration and analysis of large scale simulations using topology-based data segmentation. *IEEE TVCG* (2011). 1

[CMEH*03] COLE-MCLAUGHLIN K., EDELSBRUNNER H., HAREER J., NATARAJAN V., PASCUCCI V.: Loops in Reeb graphs of 2-manifolds. In *SoCG* (2003). 2

[COH*13] CHEN F., OBERMAIER H., HAGEN H., HAMANN B., TIERNY J., PASCUCCI V.: Topology analysis of time-dependent multi-fluid data using the reeb graph. *CAGD* (2013). 1

[CSA00] CARR H., SNOEYINK J., AXEN U.: Computing contour trees in all dimensions. In *Proc. of SoDA* (2000). 1, 2

[CWSA16] CARR H., WEBER G. H., SEWELL C. M., AHRENS J. P.: Parallel peak pruning for scalable smp contour tree computation. In *Proc. of IEEE LDAV* (2016). 1

[DFFIM15] DE FLORIANI L., FUGACCI U., IURICICH F., MAGILLO P.: Morse complexes for shape segmentation and homological analysis: discrete models and algorithms. *CGF* (2015). 1

[DN09] DORAISWAMY H., NATARAJAN V.: Efficient algorithms for computing reeb graphs. *CG* (2009). 2

[DN12] DORAISWAMY H., NATARAJAN V.: Output-sensitive construction of reeb graphs. *IEEE TVCG* (2012). 2

[DN13] DORAISWAMY H., NATARAJAN V.: Computing reeb graphs as a union of contour trees. *IEEE TVCG* (2013). 2

[EH09] EDELSBRUNNER H., HARER J.: *Computational Topology: An Introduction*. AMS, 2009. 1, 3

[FGT16] FAVELIER G., GUEUNET C., TIERNY J.: Visualizing ensembles of viscous fingers. In *IEEE SciVis Contest* (2016). 1

[FT87] FREDMAN M., TARJAN R.: Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. ACM* (1987). 2, 5, 8

[GABCG*14] GUENTHER D., ALVAREZ-BOTO R., CONTRERAS-GARCIA J., PIQUEMAL J.-P., TIERNY J.: Characterizing molecular interactions in chemical systems. *IEEE TVCG* (2014). 1

[GFJT16] GUEUNET C., FORTIN P., JOMIER J., TIERNY J.: Contour Forests: Fast Multi-threaded Augmented Contour Trees. In *Proc. of IEEE LDAV* (2016). 1, 2

[GFJT17] GUEUNET C., FORTIN P., JOMIER J., TIERNY J.: Task-based Augmented Merge Trees with Fibonacci heaps. In *LDAV* (2017). 1, 2, 5, 7, 8, 10

[GFJT19] GUEUNET C., FORTIN P., JOMIER J., TIERNY J.: Task-based Augmented Contour Trees with Fibonacci heaps. *IEEE TPDS* (2019). 1, 2, 3, 5, 6, 7, 10

[HLH*16] HEINE C., LEITTE H., HLAWITSCHKA M., IURICICH F., DE FLORIANI L., SCHEUERMANN G., HAGEN H., GARTH C.: A survey of topology-based methods in visualization. *CGF* (2016). 1

[HR18] HAJIJ M., ROSEN P.: An efficient data retrieval parallel reeb graph algorithm. *CoRR abs/1810.08310* (2018). 1, 2

[Int] Intel Cilk Plus. URL: `www.cilkplus.org`. 6

[Ope15] OPENMP ARCHITECTURE REVIEW BOARD: OpenMP Application Program Interface, V 4.5, 2015. 6

[Par13] PARSA S.: A deterministic $o(m \log m)$ time algorithm for the reeb graph. *DCG* (2013). 2, 3, 4, 5, 9, 10

[Phe08] PHEATT C.: Intel Threading Building Blocks. *J. Comput. Sci. Coll. 23*, 4 (Apr. 2008), 298–298. 6

[PSBM07] PASCUCCI V., SCORZELLI G., BREMER P.-T., MASCARENHAS A.: Robust on-line computation of reeb graphs: simplicity and speed. In *ACM ToG* (2007). 1, 2

[PSF08] PATANE G., SPAGNUOLO M., FALCIDIENO B.: Reeb graph computation based on a minimal contouring. In *SMI* (2008). 2

[PTHT10] PASCUCCI V., TRICOCHE X., HAGEN H., TIERNY J.: *Topological Data Analysis and Visualization: Theory, Algorithms and Applications*. Springer, 2010. 1

[Ree46] REEB G.: Sur les points singuliers d'une forme de Pfaff complètement intégrable ou d'une fonction numérique. *Comptes-rendus de l'Académie des Sciences 222* (1946), 847–849. 1, 3

[RWS*17] ROSEN P., WANG B., SETH A., MILLS B., GINSBURG A., KAMENETZKY J., KERN J., JOHNSON C. R.: *Using Contour Trees in the Analysis and Visualization of Radio Astronomy Data Cubes*. Tech. rep., U. of South Florida, 2017. 1

[SKK91] SHINAGAWA Y., KUNII T., KERGOSIEN Y. L.: Surface coding based on morse theory. *IEEE CGA* (1991). 1, 2

[SM17] SMIRNOV D., MOROZOV D.: Triplet Merge Trees. In *TopoInVis* (2017). 1

[SPCT18a] SOLER M., PLAINCHAULT M., CONCHE B., TIERNY J.: Lifted wasserstein matcher for fast and robust topology tracking. In *Proc. of IEEE LDAV* (2018). 1

[SPCT18b] SOLER M., PLAINCHAULT M., CONCHE B., TIERNY J.: Topologically controlled lossy compression. In *Proc. of PV* (2018). 1

[ST83] SLEATOR D. D., TARJAN R. E.: A data structure for dynamic trees. *JCSS* (1983). 2, 4, 5, 8

[TDN*12] TIERNY J., DANIELS J., NONATO L. G., PASCUCCI V., SILVA C.: Interactive quadrangulation with Reeb atlases and connectivity textures. *IEEE TVCG* (2012). 1

[TFL*17] TIERNY J., FAVELIER G., LEVINE J. A., GUEUNET C., MICHAUX M.: The Topology ToolKit. *IEEE TVCG* (2017). `https://topology-tool-kit.github.io/`. 3, 8, 10

[TGSP09] TIERNY J., GYULASSY A., SIMON E., PASCUCCI V.: Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees. *IEEE TVCG* (2009). 1, 2

[TP12] TIERNY J., PASCUCCI V.: Generalized topological simplification of scalar fields on surfaces. *IEEE TVCG* (2012). 1

[VDL*17] VINTESCU A., DUPONT F., LAVOUÉ G., MEMARI P., TIERNY J.: Conformal factor persistence for fast hierarchical cone extraction. In *Eurographics (short papers)* (2017). 1

[vKvOB*97] VAN KREVELD M., VAN OOSTRUM R., BAJAJ C., PASUCCI V., SCHIKORE D.: Contour trees and small seed sets for isosurface traversal. In *SoCG* (1997). 1

[WBP07] WEBER G. H., BREMER P., PASCUCCI V.: Topological Landscapes: A Terrain Metaphor for Scientific Data. *IEEE TVCG* (2007). 1

[WDC*07] WEBER G., DILLARD S. E., CARR H., PASCUCCI V., HAMANN B.: Topology-controlled volume rendering. *IEEE TVCG* (2007). 1