High performance level-set based topological data analysis

Charles Gueunet

▶ To cite this version:

Charles Gueunet. High performance level-set based topological data analysis. Image Processing [eess.IV]. Sorbonne Université, 2019. English. NNT : 2019SORUS120 . tel-02141632v2

HAL Id: tel-02141632
https://theses.hal.science/tel-02141632v2

Submitted on 23 Sep 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT DE
SORBONNE UNIVERSITÉ

Spécialité
Informatique

Présentée par
Charles GUEUNET

Pour obtenir le grade de
DOCTEUR de SORBONNE UNIVERSITÉ

Calcul Haute Performance pour l'Analyse Topologique de Données par Ensembles de Niveaux

Soutenue publiquement le 15 février 2019
Devant le jury composé de :

M. Christoph Garth	University of Kaiserslautern	**Rapporteur**
M. Bruno Raffin	INRIA Grenoble	**Rapporteur**
Mme. Raphaëlle Chaine	Université de Lyon	**Examinatrice**
M. Raymond Namyst	Université de Bordeaux	**Examinateur**
M. Pierre Sens	Sorbonne Université	**Examinateur**
M. Pierre Fortin	Sorbonne Université	**Co-encadrant**
M. Julien Jomier	Kitware	**Co-encadrant**
M. Julien Tierny	CNRS, Sorbonne Université	**Directeur de thèse**

Author's publications

Main Publications

International Journals

- **Charles Gueunet**, Pierre Fortin, Julien Jomier and Julien Tierny,
  "Task-based Augmented Contour Trees with Fibonacci Heaps"
  IEEE Transactions on Parallel and Distributed Systems, accepted for publication

International Conferences

- **Charles Gueunet**, Pierre Fortin, Julien Jomier and Julien Tierny,
  "Task-based Augmented Merge Trees with Fibonacci Heaps"
  IEEE Large Data Analysis and Visualization 2017, pp. 6–15

- **Charles Gueunet**, Pierre Fortin, Julien Jomier and Julien Tierny,
  "Contour Forests: Fast Multi-threaded Augmented Contour Trees"
  IEEE Large Data Analysis and Visualization 2016, pp. 85–92

Abstract-Only National Conferences

- **Charles Gueunet**, Pierre Fortin, Julien Jomier and Julien Tierny,
  "Arbres de jointure augmentés par tâches avec les tas de Fibonacci"
  Journées Visu 2018

- **Charles Gueunet**, Pierre Fortin, Julien Jomier and Julien Tierny,
  "Calcul parallèle de l'arbre de contour augmenté via une forêt d'arbres"
  Journées Visu 2017

Other Publications

International Journals

- Julien Tierny, Guillaume Favelier, Joshua A. Levine, **Charles Gueunet**, Michael Michaux
  "The Topology Toolkit"
  IEEE Transactions on Visualization and Computer Graphics 2018

- Jonas Lukasczyk, Garrett Aldrich, Michael Steptoe, Guillaume Favelier, **Charles Gueunet**, Julien Tierny, Ross Maciejewski, Bernd Hamann, Heike Leitte
  "Viscous fingering: A topological visual analytic approach"
  Applied Mechanics and Materials 2017, pp. 9–19

Misc.

- Guillaume Favelier, **Charles Gueunet**, Julien Tierny
  "Visualizing Ensembles of Viscous Fingers"
  IEEE Visualization Contest 2016 [Honorable mention]

- Guillaume Favelier, **Charles Gueunet**, Attila Gyulassy, Julien Jomier, Joshua Levine, Jonas Lukasczyk, Daisuke Sakurai, Maxime Soler, Julien Tierny, Will Usher, Qi Wu
  "Topological Data Analysis Made Easy with the Topology ToolKit"
  IEEE VIS Tutorials 2018

Software

- **Charles Gueunet**, Pierre Fortin, Julien Jomier and Julien Tierny,
  "Task-based Augmented Contour Trees with Fibonacci Heaps"
  IEEE Transactions on Parallel and Distributed Systems
  `https://github.com/CharlesGueunet/Codemit/blob/master/FTC.tgz`

- **Charles Gueunet**, Pierre Fortin, Julien Jomier and Julien Tierny,
  "Task-based Augmented Merge Trees with Fibonacci Heaps"
  IEEE Large Data Analysis and Visualization 2017
  `https://github.com/CharlesGueunet/Codemit/blob/master/FTM.zip`

- **Charles Gueunet**, Pierre Fortin, Julien Jomier and Julien Tierny,
  "Contour Forests: Fast Multi-threaded Augmented Contour Trees"
  IEEE Large Data Analysis and Visualization 2016
  `https://github.com/CharlesGueunet/Codemit/blob/master/ContourForests.zip`

- Julien Tierny, Guillaume Favelier, Joshua A. Levine, **Charles Gueunet**, Michael Michaux
  "The Topology Toolkit"
  IEEE Transactions on Visualization and Computer Graphics
  `https://github.com/topology-tool-kit/ttk`

Contents

Contents vii

1 Introduction 1
1.1 Context and motivations 1
1.2 Motivation and structure of the thesis 4

I Foundations 7

2 Background 9
2.1 Data set 11
2.1.1 Triangulation 11
2.1.2 Manifoldness 14
2.1.3 Connectivity 15
2.1.4 Neighborhood 16
2.2 Scalars 17
2.2.1 Critical points 18
2.3 Topological abstractions 20
2.3.1 Reeb graph 20
2.3.2 Contour tree 21
2.3.3 Merge tree 22
2.4 Data structures 24
2.4.1 Graph and Tree 24
2.4.2 Connectivity problems 25
2.4.3 Ordered traversal 27
2.5 Parallel computing 28
2.5.1 Multi-core parallelism 28
2.5.2 Many-core parallelism 33
2.5.3 Multi-node parallelism 34

3 State of the art 37
3.1 Merge Trees 39
3.1.1 Sequential reference algorithms 39

3.1.2 Parallel algorithms 42
3.2 Contour Trees 46
3.2.1 Sequential reference algorithm 46
3.2.2 Parallel algorithms 47
3.3 Reeb Graphs 48
3.3.1 Cut-based approaches 49
3.3.2 Dynamic connectivity 50

4 Positioning **55**

II Contributions **59**

5 Input sensitive Contour Trees using Contour Forests **61**
5.1 Overview 63
5.2 Scalar value based decomposition for parallel contour tree computations 64
5.2.1 Domain partitioning 64
5.2.2 Local computations 66
5.2.3 Contour forest stitching 67
5.3 Experimental results 68
5.3.1 Detailed performance results 70
5.3.2 Limitations 71
5.4 Conclusion 73

6 Output Sensitive Task-based Merge Trees with Fibonacci Heaps **75**
6.1 Overview 78
6.2 Local propagations for merge tree computations 79
6.2.1 Leaf search 79
6.2.2 Leaf growth 79
6.2.3 Saddle stopping condition 81
6.2.4 Saddle growth 82
6.2.5 Trunk growth 84
6.2.6 Segmentation 85
6.3 Task-based parallel merge trees 85
6.3.1 Taskification 86
6.3.2 Synchronization 87
6.3.3 Parallel trunk growth 88
6.4 Results 89

6.4.1 Performance analysis 90
6.4.2 Limitations 95
6.5 Conclusion 97

7 Output Sensitive Task-based Contour Trees with Fibonacci Heaps 99
7.1 Overview 101
7.2 Task-based contour tree computations 102
7.2.1 Leaf search 102
7.2.2 Task overlapping for merge tree computation 102
7.2.3 Merge tree post-processing 103
7.2.4 Parallel combination 104
7.3 Results 106
7.3.1 Performance analysis 107
7.3.2 Limitations 112
7.4 Conclusion 114

8 Task-based Augmented Reeb Graphs with Dynamic ST-Trees 115
8.1 Overview 118
8.2 Local propagations for Reeb graph computations . . . 119
8.2.1 Leaf search 119
8.2.2 Local growth 119
8.2.3 Critical vertex detection 120
8.2.4 Saddle vertex handling 120
8.2.5 Laziness mechanism for preimage graph 121
8.3 Task-based parallel Reeb graphs 122
8.3.1 Leaf search 122
8.3.2 Local growth 122
8.3.3 Saddle vertex handling 123
8.4 Parallel dual sweep 124
8.4.1 Leaf search 124
8.4.2 Local growth 124
8.4.3 Saddle vertex handling 125
8.4.4 Post-processing for merged arcs 126
8.5 Results 126
8.5.1 Performance analysis 127
8.5.2 Comparisons 128
8.5.3 Limitations 131

8.6 Conclusion 131

III Exploitation 133

9 Applications 135
9.1 Persistence 137
9.2 Merge Trees 138
9.3 Contour Trees 140
9.4 Reeb Graphs 140
9.5 Real-case analysis 142
9.5.1 IEEE Scientific Visualization Contest 2016 142
9.5.2 Input data sets 142
9.5.3 Analysis 144
9.6 Conclusion 151

10 Conclusion 153

Bibliography 157

1 Introduction

0 and 1 are the only informations a computer can manipulate. These 0 and 1 are named bits and can be structured in order to represent characters, which put together form a text like the one you are reading. These characters or the final text are different levels of abstraction of the underlying bits. Other frequently used abstractions in computer science include pictures, musics, videos, . . . or simulation/acquisition results in the case of scientific data sets. In the following, we describe such data sets and detail how modern computers are able to store and process more 0 and 1 than ever, as well as the consequences and new problems this raises, especially in the context of data analysis.

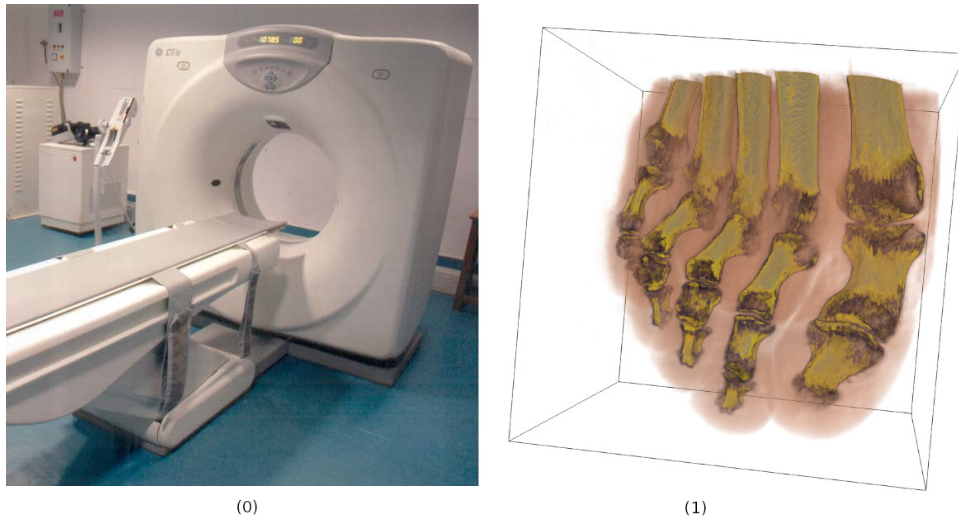
1.1 Context and motivations

In this manuscript, we focus on data sets containing information related to two or three dimensional phenomena. These data sets are from two main origins: they can either be acquired, which means they come from measurement of a real world phenomenon, or simulated when they are resulting from a simulation ran on a (super) computer.

Data acquisition

Data acquisition occurs when measurements of real world phenomena are transformed into numeric values that, in our case, can be manipulated by computers (0 and 1 as previously seen). Phenomena that are to be measured can be as varied as sensors allow. Some examples include: meteorology, medical scans (cf. Figure 1.1) or ground-penetrating radargrams. Acquired data sets usually suffer from noise due to measurement errors.

Through years, measurement techniques and probe accuracy have improved, leading to an increase in the size and details of acquired data sets.



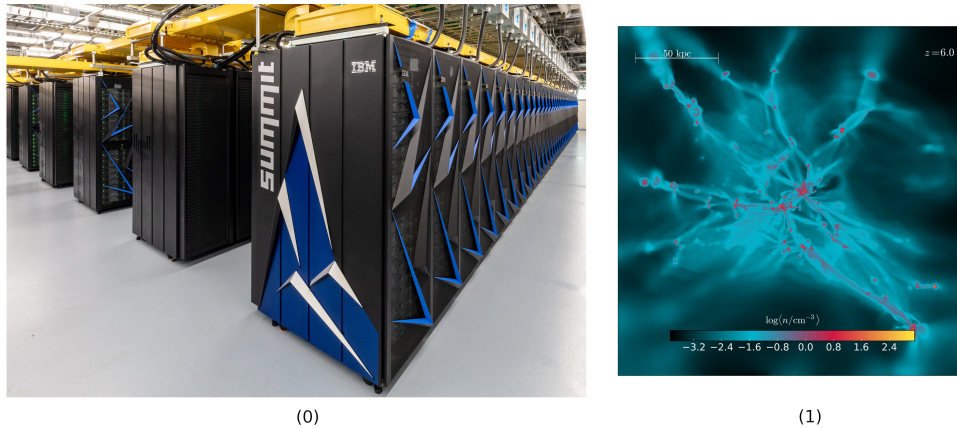
(0) (1)

Figure 1.1 – *(0): A Computerized Tomography (CT) scan used in medical imaging to obtain a tomographic 3D image of a specific area, from Wikimedia (by NithinRao).*
(1): Result of a CT scan on a human foot. This data set has a resolution of 256^3 *samples. Actual scan results are closer to* 2048^3 *or even* 4096^3 *samples.*

Data simulation

As physical and chemical models are presently able to accurately reflect a significant number of real world phenomena, it may be easier to simulate a phenomenon than to reproduce it through experimentation. Additionally, there are situations where experimentation is not possible. It can be for ethical reasons, in the case of virus spreading or nuclear testing for example. Simulations are also used in order to reduce the number of tests in real conditions when these are expensive, like in the case of rocket launches or crash-tests for cars. Finally, simulations are also used to explore phenomenon that can not be directly reproduced like in the case of the cosmological simulation (see Figure 1.2(1)).

The size of these simulations is driven by the performance of the computer on which these are run (from a workstation to a supercomputer like the one shown in Figure 1.2(0)). Furthermore, the compute power is growing through time. A good example of this continuous expansion is the well known Moore's law about the exponential growth of the number of transistors in processors, which has been observed to double every two years. Along with the ever larger use of ever more powerful HPC facilities, this has lead to amounts of data that cannot be interpreted by humans.



(0) (1)

Figure 1.2 – *(0): The Oak Ridge Leadership Computing Facility supercomputer with a computational power of 200 petaflops.*
(1): A cosmological simulation from when the universe was only one billion years old, from Wikimedia (by Vis-sns).

Data analysis

Whether they are acquired or simulated, these data sets must be analyzed. Data analysis is the process of representing, manipulating and exploring data in order to extract relevant information. As we consider two and tree dimensional phenomena, we can rely on scientific visualization, a branch of computer science aimed to help the exploration of such data sets through the use of graphical representations. As data sets get bigger and more complex, it becomes challenging for scientists to glean insight from their data. In the same way the 0 and 1 are not convenient for human beings but a text is, we can create new abstractions that help the analysis of complex data sets: this is the role of topology-based visualization methods. Informally, topology is the study of geometrical properties of spaces, unaltered by continuous deformations such as stretching or bending. In practice, it can be used to create abstractions that serve as maps of the original data set. Data analysis using topological abstractions is part of topological data analysis, often abbreviated TDA. Thanks to its robustness and its ability to extract features of interest at multiple scales of importance, TDA gained in importance over the last few years and was successfully applied to a variety of applications (combustion [15], chemistry [13, 39], astrophysics [68, 71, 78], material science [31, 41, 51], fluid dynamics [19, 42, 46, 69, 77, 85], medical imaging [12, 18], etc). In Figure 1.3 we present several data analysis examples, where topological tools are used to extract and evaluate areas of interests on various data sets. Each time, data sets are colored according to segmentations induced

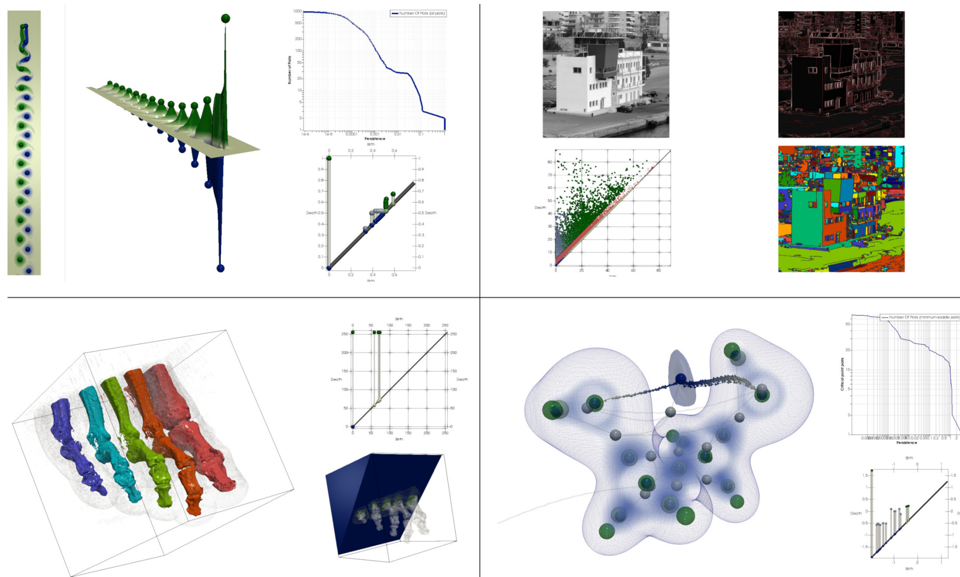


Figure 1.3 – *Four data analysis results on simulated and acquired data sets. In these examples, data exploration is guided by topological tools allowing to extract, count and evaluate the robustness of features on each data set.*

by contour based abstractions, which will be formally defined in the remainder of this manuscript.

Limitations

The increase in size of current data sets represents a challenge for interactivity in the context of topological data analysis. To make matters worse, traditional algorithms used to compute topological abstractions are often sequential and thus do not fully exploit the compute power of modern architectures. These algorithms rely on a global view of the data which makes their parallelization challenging.

1.2 Motivation and structure of the thesis

For ten years, the compute power has grown through parallelism and has significantly increased the size of data sets, without impacting the execution times of topological data analysis algorithms. Indeed, most topological abstractions are still computed using intrinsically sequential algorithms and existing parallel approaches offer limited speedups. Therefore, efficiently parallelizing them would be desirable to exploit at best modern architectures, improving interactivity on workstations and efficiency on supercomputers.

The main topic of this thesis is the design of efficient parallel algorithms

for topological data analysis, focusing on level set based abstractions: merge trees, contour trees and Reeb graphs.

The first part of this manuscript (chapters 2 to 4) is dedicated to prerequisites. All the required definitions are first given in chapter 2. Traditional algorithms used to compute level set based abstractions are presented in chapter 3 and existing parallel approaches are also discussed. In chapter 4, we describe in details the scientific positioning of this thesis: we list the topological abstractions on which we focus, and we justify our choices regarding HPC architectures and algorithms. We also present an overview of our contributions. Afterward, the second part of the manuscript (chapter 5 to 8) details our contributions. First, we present in chapter 5 an approach that efficiently computes contour trees in parallel on shared memory workstations. This algorithm uses thread-based parallelism and relies on a static decomposition of the input mesh by scalar values. Then, an approach using independent local propagations and task-based parallelism to compute merge trees is presented in chapter 6 and refined in chapter 7 to deal with contour trees. Additionally, a task-based algorithm also relying on independent propagations to compute Reeb graphs is detailed in chapter 8. The last part of this manuscript (chapter 9 and 10) is used to emphasize how this work can be exploited through applications and examples in chapter 9. Finally, a conclusion on this thesis is given in chapter 10 and perspectives of this work are also presented.

Part I

Foundations

2 Background

Contents

2.1 Data set 11
2.1.1 Triangulation 11
2.1.2 Manifoldness 14
2.1.3 Connectivity 15
2.1.4 Neighborhood 16
2.2 Scalars 17
2.2.1 Critical points 18
2.3 Topological abstractions 20
2.3.1 Reeb graph 20
2.3.2 Contour tree 21
2.3.3 Merge tree 22
2.4 Data structures 24
2.4.1 Graph and Tree 24
2.4.2 Connectivity problems 25
2.4.3 Ordered traversal 27
2.5 Parallel computing 28
2.5.1 Multi-core parallelism 28
2.5.2 Many-core parallelism 33
2.5.3 Multi-node parallelism 34

This chapter introduces all theoretical notions required for the understanding of this manuscript, as well as parallel computing.

2.1 Data set

In scientific visualization, input data sets are usually geometrical objects (meshes) on which are defined scalar, vector or tensor fields. In the context of this manuscript, we consider manifold triangulations and univariate scalar fields. In the following, we formalize these terms and describe some topological notions required in the remainder of this document.

2.1.1 Triangulation

Computer science is fundamentally a discrete world and so geometrical objects are usually manipulated using meshes. A mesh is a set of polytopes used to represent a surface or a volume, like a CFD simulation model, a mechanical piece, a video game character or any other 2D/3D discrete shape.

The surface or volume on which the analyzed phenomena take place is named the domain. To introduce the notion of domain, we start by defining topological spaces.

Definition 1 (Topological space) *A topological space is an ordered pair* $(\mathbb{X}, \tau)$*, where* $\mathbb{X}$ *is a set and* τ *is a collection of subsets of* $\mathbb{X}$ *having the following properties:*

- $\emptyset$ *and* $\mathbb{X}$ *belong to* τ
- *Any union of members of* τ *belongs to* τ
- *Any finite intersection of members of* τ *belongs to* τ

In order to locate in this space, we use the notion of point.

Definition 2 (Point) *A point in the Euclidean space* $\mathbb{R}^d$ *of dimension* $d > 0$*, is a set of* d *coordinates.*

In the domain, a point is a position in space (not to be confused with a vertex, which is an object of dimension zero as we will see later). In a triangulation with a dimension up to three, the type of cells that can be used are restricted to: vertices, edges, triangles and tetrahedra (only for dimension three). These cells are simplices. To define a simplex, we need the notion of convexity.

Definition 3 (Convex set) *A set* $\mathcal{S}$ *of an Euclidean space* $\mathbb{R}^d$ *of dimension* d *is convex if for any two points* x, y *in* $\mathcal{S}$ *and all* $t \in [0, 1]$ *the point* $(1-t)x + ty$ *also belongs to* $\mathcal{S}$*.*

Intuitively, a set $\mathcal{S}$ is convex if all for all pairs of points $x, y \in \mathcal{S}$ all points on the segment (x, y) are also in $\mathcal{S}$ (cf. Figure 2.1 (0)).

Definition 4 (Convex hull) *The convex hull of a set of points $\mathcal{P}$ in an Euclidean space $\mathbb{R}^d$ is the unique minimal convex set containing all points of $\mathcal{P}$.*

Figure 2.1 (1) shows the minimal convex set of three linearly independent points (yellow). This form a convex hull, in this case a triangle.

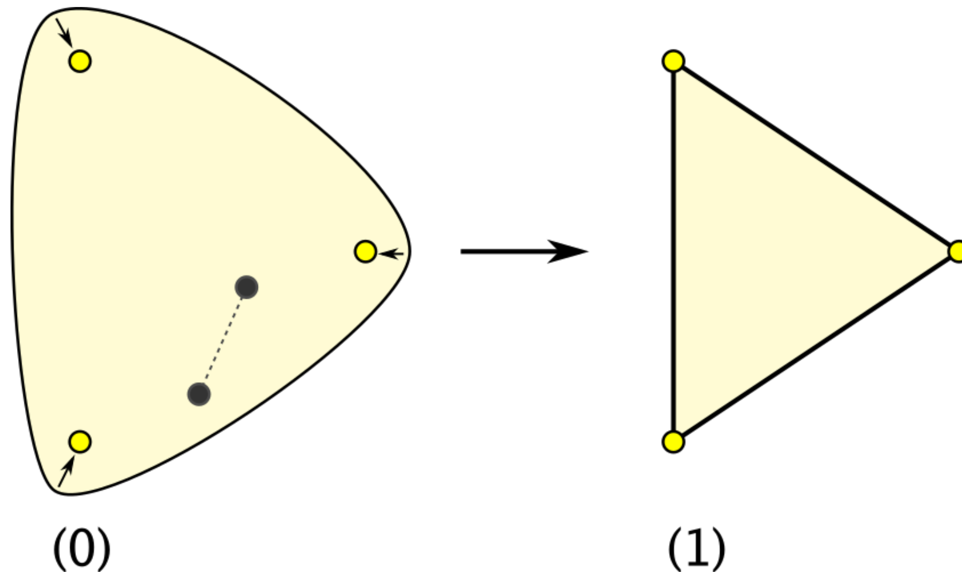


Figure 2.1 – *Example of the convex hull of three yellow points: In (0) the highlighted area is a convex set containing the three yellow points. It is convex since all pairs of points inside it can be joined by a segment entirely inside the set, an example is shown using the two shaded points. In (1) the highlighted area is the minimal convex set containing the three yellow points: the convex hull.*

Definition 5 (Simplex) *A n-simplex is the convex hull of $n + 1$ points linearly independent in an Euclidean space $\mathbb{R}^d$, with $0 \leq n \leq d$.*

In Figure 2.2, simplices up to dimension 3 are illustrated. As we have seen previously, the 0-simplex is a vertex. Additionally, the 1-simplex is an edge, the 2-simplex a triangle and the 3-simplex is a tetrahedron.

Definition 6 (Face) *A face is a simplex containing a sub set of the vertices of another simplex called co-face.*

For example, a tetrahedron has four distinct triangles as face, but also six edges and four vertices. These simplices are the elementary bricks used to represent the geometry of our data sets. Glued together, they form a simplicial complex.

Definition 7 (Simplicial complex) *A simplicial complex $\mathcal{K}$ is a finite collection of simplices σ_i*

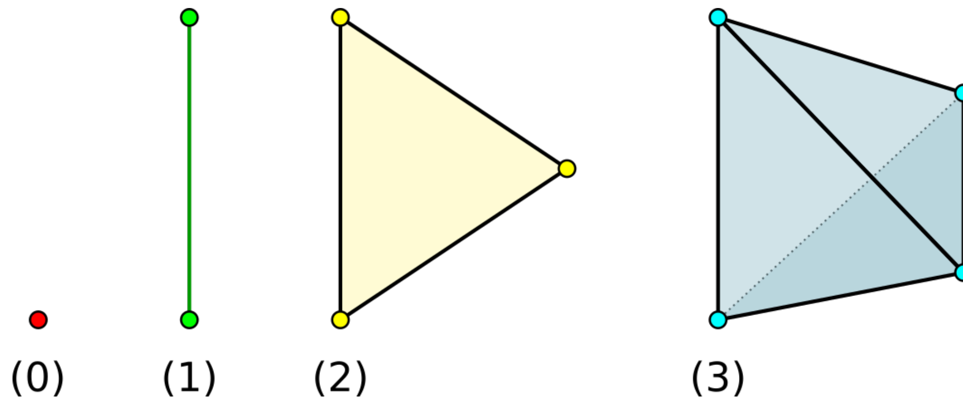


Figure 2.2 – *Illustration of simplices up to dimension three:*
dimension 0*: a vertex containing the red point;*
dimension 1*: an edge containing the two green points;*
dimension 2*: a triangle containing the three yellow points;*
dimension 3*: a tetrahedron containing the four blue points.*

such that every face of a simplex of $\mathcal{K}$ is also in $\mathcal{K}$, and any two simplices intersect in a common face or not at all. The dimension of the simplicial complex is the highest dimension among its simplices.

A simplicial complex of dimension k is noted k-simplicial complex. For example, a 2-simplicial complex may contain vertices, edges and triangles, but not any higher dimensional simplices.

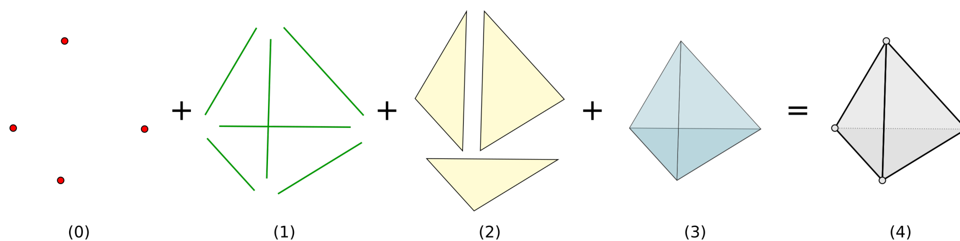


Figure 2.3 – *The set of vertices* (0)*, edges* (1)*, triangles* (2) *and tetrahedra* (3) *composing the simplicial complex* (4)*.*

In Figure 2.3 all simplices contained in the 3-dimensional simplicial complex (4) composed of a single tetrahedron are represented. If we consider vertices, edges and triangles only, omitting the tetrahedron, we obtain a 2-simplicial complex (a surface) in a 3-dimensional domain.

In the context of topological data analysis, a triangulation is a simplicial complex and every mesh in dimension two or three can be easily converted into a triangulation by subdividing its cells into simplices. We will see in the next subsection that the notion of simplicial complex is still too generic for our use cases and requires the definition of the notion of manifoldness.

2.1.2 Manifoldness

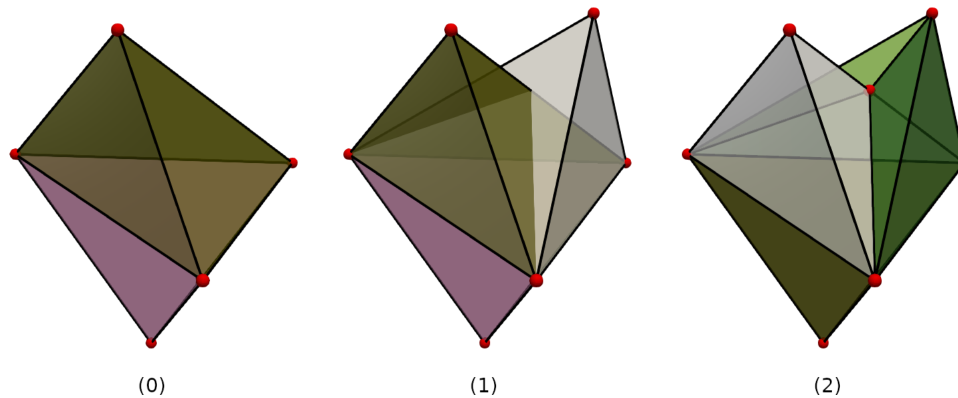


Figure 2.4 – *Three different* 3*-simplicial complexes, each cell having a different color:*
*(*0*): two tetrahedra;*
*(*1*): a mingled tetrahedron is added to (*0*), the mesh is not manifold anymore;*
*(*2*): subdividing simplices allows to obtain a new manifold triangulation*

The notion of simplicial complex alone allows cells to cross each other. Figure 2.4 shows a triangulation having mingled tetrahedra (1). In this case it is possible to subdivide simplices in order to obtain a new mesh without overlapping, having the same shape than the previous one as presented in (2). The term "manifold", described next, requires the notion of homeomorphism.

Definition 8 (Homeomorphisms) *Two topological spaces $\mathbb{A}$ and $\mathbb{B}$ are said to be homeomorphic if and only if there exists a continuous bijection $f : \mathbb{A} \rightarrow \mathbb{B}$ such that the inverse function $f^{-1} : \mathbb{B} \rightarrow \mathbb{A}$ is also continuous.*

Roughly speaking, a homeomorphism is a continuous stretching and bending of a topological space into a new shape. For example, a triangle and a square are homoemorphic to each other, while a sphere and a torus are not. But this description can be misleading as some continuous deformations are not homeomorphisms such as the deformation of a line into a point. Moreover some homeomorphisms cannot be achieved using only continuous deformations, for example a knot and a circle are homeomorphic but the knot needs to be cut and stitched back to be turned into a circle.

Definition 9 (Manifold) *A topological space $\mathbb{X}$ of dimension d is manifold if every point $p \in \mathbb{X}$ has an open neighborhood homeomorphic to an open neighborhood of $\mathbb{R}^d$. More precisely, in dimension d a manifold is referred to as a d-manifold.*

Intuitively, a manifold space locally resembles a Euclidean space near

each point. In Figure 2.4 (1), some points are both in the brown and white tetrahedron, therefore this mesh is not manifold (for this overlap region, there is no bijection from the complex to $\mathbb{R}^3$). In (2), mingled tetrahedra have been subdivided so there is no more overlapping. Another classical example of a non manifold mesh would be two tetrahedra touching only on a single vertex. At this particular vertex the neighborhood is not homeomorphic to a 3-ball and the mesh is not manifold.

2.1.3 Connectivity

Some of the algorithms presented later require the input mesh to be simply connected. This notion is introduced constructively.

Definition 10 (Connected space) *A topological space $\mathbb{X}$ is said to be connected if for every pair of points in $\mathbb{X}$ there is a path in $\mathbb{X}$ between them.*

Definition 11 (Connected component) *A connected component is the maximal subset of a topological space which is connected.*

Definition 12 (Simply connected) *A topological space $\mathbb{X}$ is simply connected if it is connected and for any pair of points in $\mathbb{X}$, any path can be continuously deformed into another.*

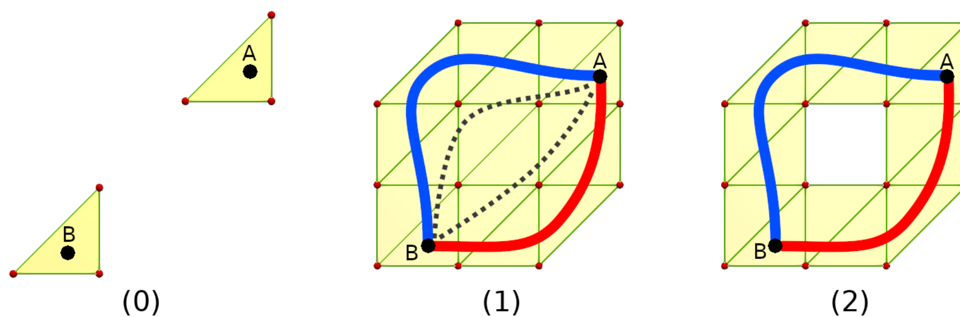


Figure 2.5 – *Let $\mathcal{K}_i$ be the simplicial complex corresponding to the number (i).*
$\mathcal{K}_0$ is not connected as we can find two points (A and B) with no path in $\mathcal{K}_0$ to join them.
$\mathcal{K}_1$ is a connected simplicial complex: there is a path in $\mathcal{K}_1$ between every pair of points.
$\mathcal{K}_2$ is a connected simplicial complex: but contrary to $\mathcal{K}_1$ it is not a simply connected one as we can find two paths that cannot be continuously transformed one into each other.

In Figure 2.5, examples of topological spaces illustrating these various connectivities are presented. The first one (0) is composed of two distinct triangles. Each of these triangles is a connected component and this space is not simply connected. The second one (1) is connected as there is a path between every pair of points on the space. As every path between these points can be continuously deformed into another, this second space is simply connected. On the contrary, the third example (2) shows a hole

which prevents paths to be deformed into some others without stitching. It is not a simply connected topological space, just a connected topological space.

2.1.4 Neighborhood

All domains we consider being simplicial complexes, the notion of neighborhood for a simplex is consistent. For example a vertex is always in the neighborhood of all its neighbors. Several topological notions related to neighborhood are used in this manuscript and we give here their formal definitions.

Definition 13 (Closure) *The closure of a collection of simplices σ of a simplicial complex $\mathcal{K}$ denoted $Cl(\sigma)$ is the minimal sub-simplicial complex of $\mathcal{K}$ that contains each face of σ.*

Definition 14 (Star) *The star of a collection of simplices σ of a simplicial complex $\mathcal{K}$ denoted $St(\sigma)$ is the set of simplices of $\mathcal{K}$ having a simplex of σ as a face.*

Definition 15 (Link) *The link of a collection of simplices σ of a simplicial complex $\mathcal{K}$ denoted $Lk(\sigma)$ is the closure of the star (the closed star) of σ minus the star of σ: $Lk(\sigma) = Cl(St(\sigma)) - St(\sigma)$.*

The link can also be expressed as the set of faces of the simplices in the star of σ that are disjoint from σ.

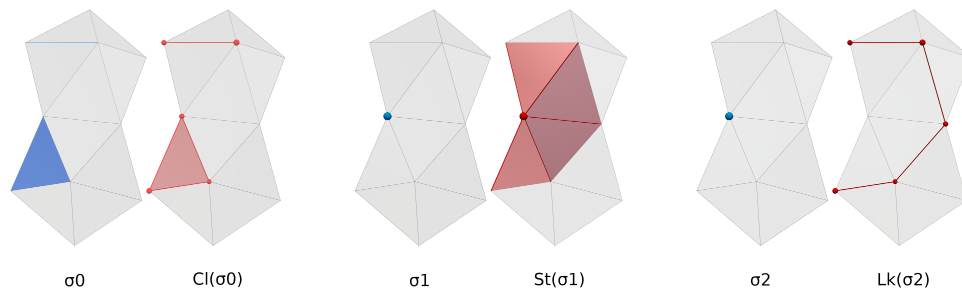


Figure 2.6 – *Three collections of simplices σ (in blue) in a simplicial complex $\mathcal{K}$ with their corresponding closure, star and link (in red).*

In Figure 2.6 the notions of closure, star and link are illustrated. On the left, $\sigma 0$ is composed of an edge and a triangle, so the closure $Cl(\sigma 0)$ is composed of the triangle along with its three edges and three vertices, and the lone edge and its two vertices. The result is a valid simplicial complex. In the middle, $\sigma 1$ is a single vertex. Its star in red is composed by this vertex along with adjacent edges and triangles. This is not a valid simplicial complex as some edges on the triangles are missing. Finally, on

the right $\sigma 2$ is also a vertex. Its link is composed of the simplices which are in the closed star of $\sigma 2$ but not directly attached to the vertex $\sigma 2$.

2.2 Scalars

Data sets in scientific visualization usually contain scalar, vector or tensor fields. In the context of this manuscript, only univariate scalar values, elements of $\mathbb{R}$, are considered. These scalars generally correspond to simulation or acquisition results, can it be a temperature, a density, a pressure or any other physical measure. These values are defined on every vertex of the data set and for the remainder of this manuscript, we consider that each vertex has a distinct scalar value. In practice, this is not a limiting constraint as we can use the simulation of simplicity [30] in order to obtain a consistent disambiguation in an existing data set. These scalar values can be extended to the whole mesh using a linear interpolation with barycentric coordinates.

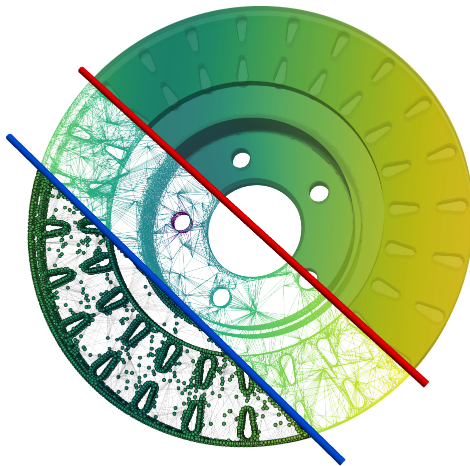


Figure 2.7 – *This data set is a brake disk with the scalar field corresponding to the temperature (blue low, yellow high). Below the blue line the scalar field associated to vertices of the mesh is shown. Then, between the two lines an interpolation extend these scalar values to edges. Finally, above the red line scalar values are interpolated to the whole mesh.*

Scalar values being extended to the whole mesh allows to define the pre-image of a scalar value: the *level set*.

Definition 16 (Level set) *On a simplicial complex $\mathcal{K}$, the level set $f^{-1}(i)$ of an isovalue $i \in \mathbb{R}$ relatively to a scalar field $f : \mathcal{K} \to \mathbb{R}$ is the pre-image of i onto $\mathcal{K}$ through $f : f^{-1}(i) = \{p \in \mathcal{K} | f(p) = i\}$.*

On a d-manifold, a level set is a $(d-1)$-manifold. A notion heavily used in the remainder of this manuscript is the concept of *contour*.

Definition 17 (Contour) *On a simplicial complex $\mathcal{K}$, let $f^{-1}(i)$ be the level set of an isovalue i relatively to a scalar field $f : \mathcal{K} \to \mathbb{R}$. A connected component of $f^{-1}(i)$ is called a contour.*

As a connected component of level set, a contour can also be represented with a simplicial complex. In the following, we denote $f^{-1}(f(p))_p$ the contour containing the point p.
Instead of taking the pre-image of a single scalar, we can also consider the pre-image of all scalars above or below a certain isovalue.

Definition 18 (Sub-level set) *On a simplicial complex $\mathcal{K}$, the sub-level set $f_-^{-1}(i)$ of an isovalue $i \in \mathbb{R}$ relatively to a scalar field $f : \mathcal{K} \to \mathbb{R}$ is the set of points having a scalar value lower than i through f : $f_-^{-1}(i) = \{p \in \mathcal{K} | f(p) \leq i\}$.*

The sur-level set is defined symmetrically as the set of points that have a scalar value above or equal to certain isovalue. This scalar field can be used to refine notions previously seen.

Definition 19 (Lower star) *The lower star $St^-(\sigma_0)$ of a vertex σ_0 is the set of simplices in the star of $\sigma_0 : St(\sigma_0)$ having all their vertices in $f_-^{-1}\big(f(\sigma_0)\big)$.*

And symmetrically, the upper star is the set of simplices in $St(\sigma_0)$ that are in the sur-level set of the scalar value associated with σ_0.

Definition 20 (Lower link) *The lower link $Lk^-(\sigma_0)$ of a vertex σ_0 is the set of simplices in the link of $\sigma_0 : Lk(\sigma_0)$ having all their vertices in the sub level set of the isovalue associated with σ_0.*

And the upper link is defined similarly using the sur-level set and is noted $Lk^+(\sigma_0)$. The union of the lower and upper star is not necessarily equal to the complete star as some simplices may be crossing the scalar value of the related vertex. The same remark can be done for the link.

2.2.1 Critical points

The scalar field of a simplicial complex is a piecewise linear function (when a linear interpolation is used). As such, it admits critical points. These points are located on vertices and can only be of two kinds: extrema and saddles.

Definition 21 (Extremum) *On a simplicial complex $\mathcal{K}$ with a scalar field $f : \mathcal{K} \to \mathbb{R}$, a vertex v is a maximum (respectively a minimum) of f iff $Lk^+(v)$ $\big($respectively $Lk^-(v)\big)$ is empty.*

Definition 22 (Saddle) *On a simplicial complex $\mathcal{K}$ with a scalar field $f : \mathcal{K} \to \mathbb{R}$, a vertex v is a saddle of f iff $Lk^-(v)$ or $Lk^+(v)$ have more than one connected components.*

A point which is not a critical point is said to be regular.

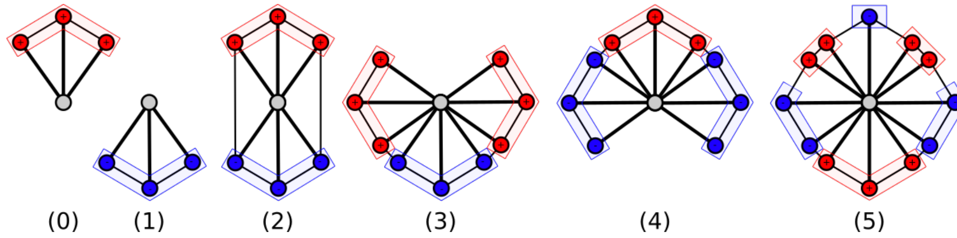


Figure 2.8 – *Neighborhood of a vertex v in grey, with the connected components of the link emphasized:*
(0): $|Lk^-(v)| = 0$, a local minimum;
(1): $|Lk^+(v)| = 0$, a local maximum;
(2): $|Lk^-(v)| = |Lk^+(v)| = 1$, a regular vertex;
(3): $|Lk^-(v)| = 1, |Lk^+(v)| = 2$, a split saddle;
(4): $|Lk^-(v)| = 2, |Lk^+(v)| = 1$, a join saddle;
(5): $|Lk^-(v)| = 3, |Lk^+(v)| = 3$, a degenerate saddle.

In Figure 2.8 examples of vertices neighborhood are given. In (0) and (1) the grey vertex is either the lowest or highest in its neighborhood, hence it is an extrema. In (2), we have an example of a regular vertex with one connected component of both lower and upper link. All other cases are saddles. Using the link to compute critical points of a data set is a classical approach (and is embarrassingly parallel).

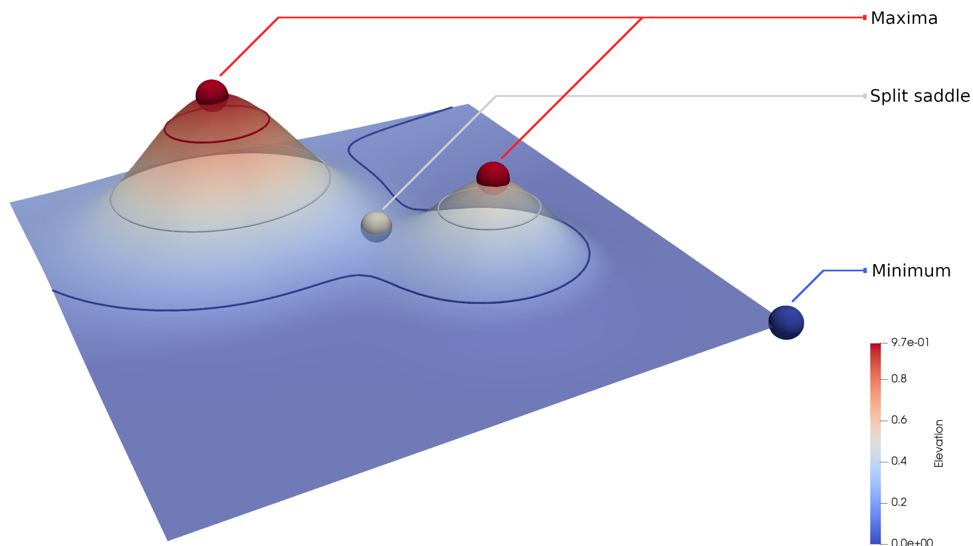


Figure 2.9 – 2-*triangulation where the scalar field is the height, from blue (low) to red (high). Critical points of the mesh are shown using spheres: red for maxima, gray for saddles and blue for minima. Three level sets are shown using colored curves.*

For a macroscopic view of critical points, Figure 2.9 presents a simple data set consisting of two hills with a height scalar field. At the top of each hill we have a local maximum and there is a split saddle at the point

where these two hills become distinct (hence the name of split saddle). The two hills become distinct when the level set change from one connected component to two. On a simplicial complex with a linearly interpolated scalar field, such a change in the number of connected components of level set (or contour) can only occur at the vicinity of a critical point. The opposite is not true, a critical point does not always imply a change in the number of contours.

2.3 Topological abstractions

In this section, we define three topological abstractions which are at the core center of this manuscript. These abstractions track connected components of level sets (or sub-level sets), hence the term *level set based abstractions* used to describe them in this thesis. Other topological abstractions exist, like the Morse-Smale complex [40] which relies on the gradient for example (its definition is out of the scope of this manuscript). Level set based abstractions also rely on the notions of graph and tree formally defined subsection 2.4.1. For now, a graph is a 1-simplicial complex, if it has no loop we can also call it a tree. In this manuscript, the terms arc and node are used to describe a graph structure whereas the terms edge and vertex refer to the mesh.

2.3.1 Reeb graph

The Reeb graph is a topological abstraction reflecting the evolution of the connected components of level sets (contours) on a manifold $\mathcal{M}$. In the context of this manuscript, the input mesh is a manifold triangulation as previously defined.

Let $\sim$ be an equivalence relation such that two points are equivalent through $\sim$ if and only if these two points reside on the same contour. The Reeb graph is defined as the quotient space $\mathcal{M}/\sim$.

Definition 23 (Reeb graph) *On a manifold $\mathcal{M}$, the Reeb graph $\mathcal{R}(f)$ is a one dimensional simplicial complex defined as the quotient space on $\mathcal{M} \times \mathbb{R}$ by the equivalence relation $(p_1, f(p_1)) \sim (p_2, f(p_2))$ which holds iff:*

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

Figure 2.10 presents the Reeb graph of a height scalar field on a hand data set. On the right, two level sets are shown with their contours colored

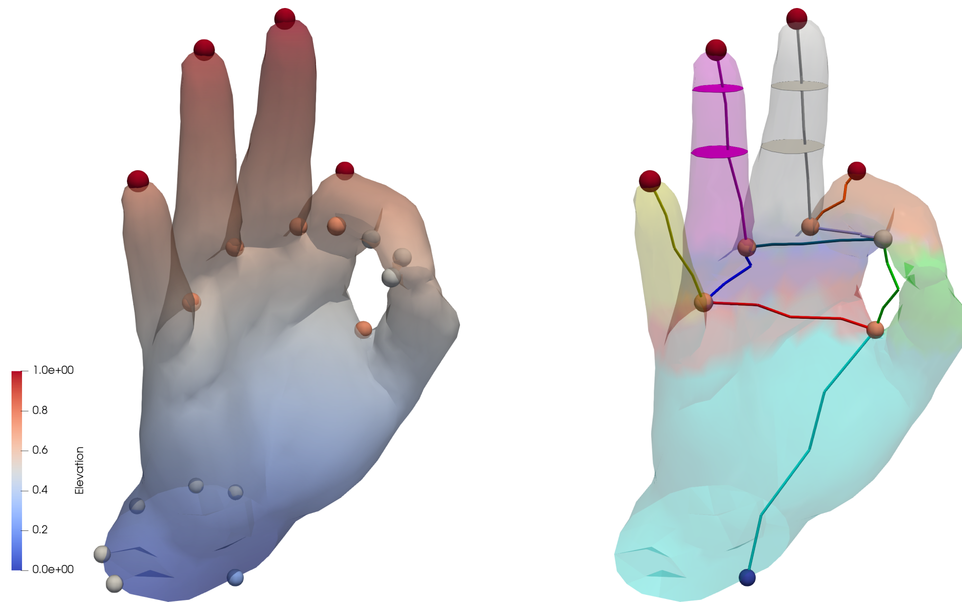


Figure 2.10 – *A 3-triangulation of a hand with the height scalar field. On the left, the scalar field is shown along with all the corresponding critical points. On the right, the Reeb graph of this data set is presented along with the corresponding segmentation. Two level sets are given to emphasize the contour contraction mechanism.*

accordingly to the arc they are related with. Each contour can be replaced by a single point, equivalent to all the points in the contour though $\sim$: this is called a contraction. The Reeb graph of f can also be defined as the continuous contraction of each contour into a point. With this definition, we can see that each arc corresponds to a region where the number of connected components of level sets is equal to one. The corresponding segmentation is used to color the mesh on the right side of Figure 2.10. As seen in subsection 2.2.1, the number of contours can only change at a critical point. This means the arcs of the Reeb graph can only start and end at critical points (but not all critical points are critical nodes on the Reeb graph).

In Figure 2.10, the topological handle created by the fingers leads to a loop in the graph. A loop in the Reeb graph can only occur around a topological handle. The next subsection focus on the case where the input data set has no topological handle.

2.3.2 Contour tree

When the input domain has no topological handle, the output Reeb graph is granted to have no loop. On a simply connected manifold the Reeb graph is called contour tree.

Definition 24 (Contour tree) *The Reeb graph of a scalar field f defined on a simply connected manifold $\mathcal{M}$ is called contour tree and noted $\mathcal{C}(f)$.*

As we will see later, computing the contour tree is several orders of magnitude faster than the Reeb graph computation in practice. This is particularly useful for regular grids which are simply connected by construction. For unstructured meshes, knowledge about the data set is required as a contour tree algorithm may return a wrong output on a non simply connected domain.

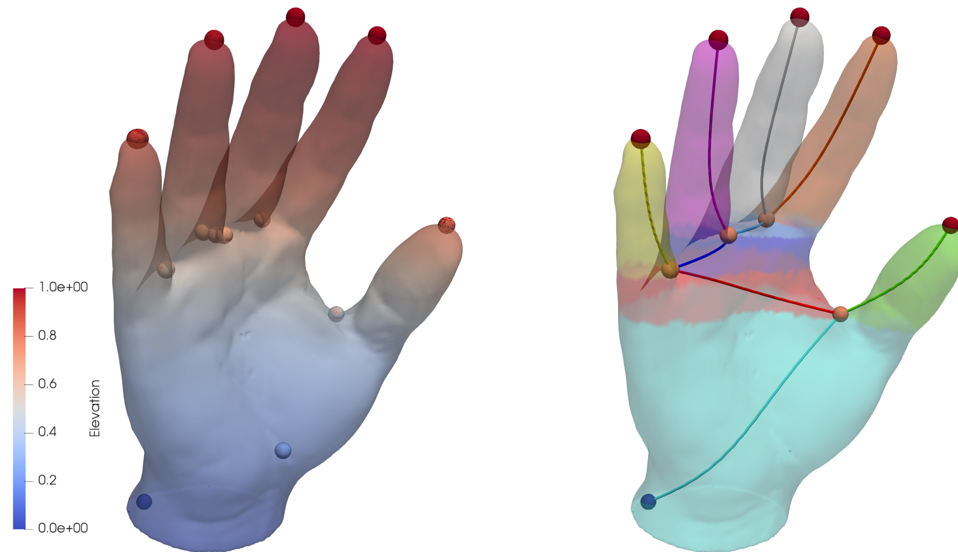


Figure 2.11 – *A simply connected 3-triangulation of a hand with a height scalar field. On the left, the scalar field is shown along with all the corresponding critical points. On the right the contour tree is presented along with the corresponding segmentation.*

Figure 2.11 shows the contour tree of a height scalar field on a simply connected, manifold triangulation of a hand data set. This data set is analogous to the one presented Figure 2.10 but does not contain the handle. As a result, the output is the tree shown on the right.

2.3.3 Merge tree

In the same way the contour tree tracks changes in the number of connected components of level sets, the merge tree tracks changes in the number of connected components of sub/sur-level sets. In this manuscript, we call *join tree* the merge tree tracking changes in the number of sub-level set components as this tree contains all the minima and critical points where the corresponding components join together. We call *split tree* the one containing split saddles and maxima. In the literature, the names of these two trees are sometime interchanged.

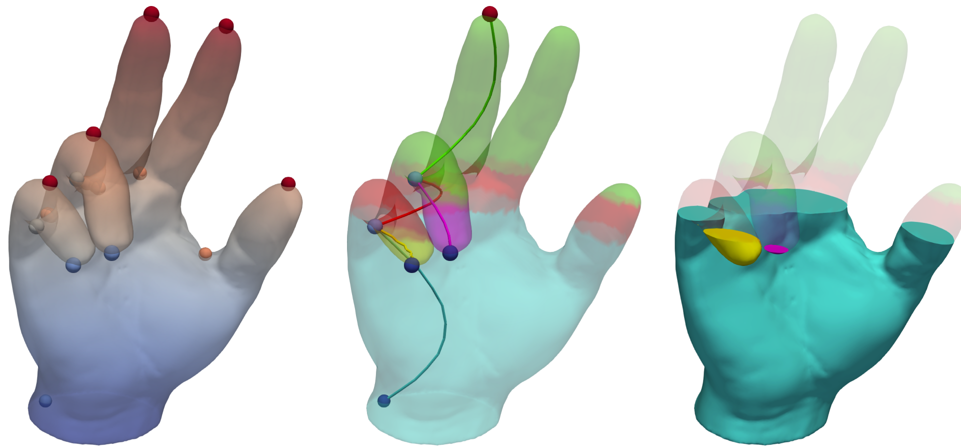


Figure 2.12 – *A simply connected 3-triangulation of a hand with a height scalar field. On the left, the scalar field is shown along with all the corresponding critical points. On the middle, the join tree of this data set is presented along with the corresponding segmentation. There are only three leaves on this tree as this data set has only three minima. On the right, the sub-level set just below the first join saddle is shown, we can see the three connected components. In particular, the light blue and yellow components are about to merge at the saddle.*

Figure 2.12 presents the join tree of a height scalar field on the 3-triangulation of a hand, analogous to those presented previously. There are three minima on this data set, a global one on the wrist and two others on the lowered fingers. The splits and maxima are not tracked by the join tree as they do not change the number of connected components of sub-level sets. The root of the join tree is the global maximum, where the last connected component of sub-level sets ends.

The merge tree is a topological abstraction generally used on data sets where areas of interests are either minima or maxima and their corresponding regions. Additionally, reference algorithms to compute the contour tree also rely on the merge tree computation, as detailed in subsection 3.2.1. Finally, merge trees are used to compute the persistence diagram, which is a powerful tool to measure the number and the robustness of features on a data set.

Segmentation

The hand data sets shown previously are colored according to the segmentation induced by the topological abstraction on the figure. The segmentation is the mapping between all vertices to arcs they belong to in the graph/tree. When the output data structure explicitly models this information, the graph/tree is said to be *augmented*. Otherwise, the output

is only a skeleton and called *non-augmented*. In practice, to enable the full extent of level set based applications (as shown in Figure 1.3), augmented trees are required. Non-augmented trees can only be applied to a specific sub-set of applications. This is challenging as the computation of the augmented trees is more intensive than for non-augmented ones.

2.4 Data structures

The main contributions presented in the second part of this manuscript consist in new algorithms to compute the abstractions presented previously. These computations rely on existing data structures, presented in this section.

2.4.1 Graph and Tree

In computer science, a graph data structure is a set of vertices linked together by edges. As is, an edge can link a vertex to itself. However, in the context of this manuscript, graphs are guaranteed to be 1-simplicial complices and an edge can only link two distinct vertices. More precisely, an arc can only link two distinct nodes as these terms are preferred to describe the graph structure.

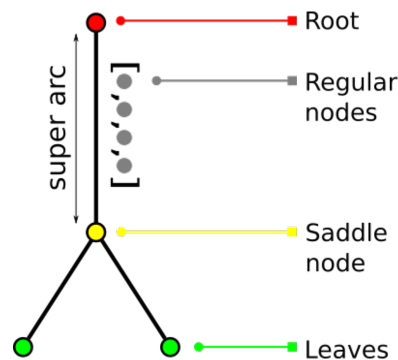


Figure 2.13 – *A simple graph composed of three (super) arcs and four critical nodes. Regular nodes of the root arc are shown as stored separately.*

For Reeb graphs, contour trees and merge trees, the graph structure maps to the mesh. This means our graph structure has to deal with regular nodes. As presented Figure 2.13, we have chosen to store the sorted list of regular nodes of each arc separately. This methods requires less memory than an explicit storage as arcs between these nodes are implicit. As emphasized in the figure, arcs between two critical nodes are called super arcs. As our graph representation only contains super arcs, we adopt the

convention that arcs are always assumed to be super arcs unless otherwise stated.

2.4.2 Connectivity problems

Identifying connected components in a graph (eventually subject to updates) is called the connectivity problem. In the following, we detail data structures addressing this problem, depending on the changes allowed for the graph.

2.4.2.1 Static connectivity

When a graph is static (no arcs are to be added or removed), the only operation required to query connected components is:

- *connected(v,w)*: return true if v and w are connected.

This operation can be implemented using a breadth-first search traversal detailed below.

Breadth First Search. Starting at a n-simplex, it recursively explores its neighborhood using the (closed) star of the current simplex to store the next n-simplices to visit in a queue. It generally stops when there is no more candidate to visit. In other word, a BFS is a walk across simplices, using neighborhood relationship to visit the structure. By construction, it visit all given n-simplices in a connected component, thus this algorithm can check if two vertices are connected. It is used in practice to count the number of connected components in a complex.

2.4.2.2 Incremental connectivity

In the case of the incremental connectivity problem, arcs can be added to the graphs, which means connected components can merge together. The operations required by a data structure addressing this problem are:

- *connected(v,w)*: return true if v and w are connected.
- *insert(v, w)*: add an edge between nodes v and w in the graph.

An efficient data structure addressing this problem is the Union-Find, presented below.

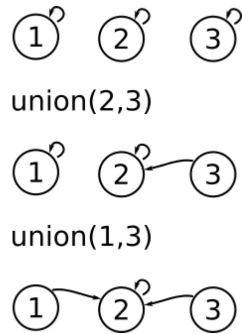


Figure 2.14 – *Example of an Union-Find data structure on a set with three elements. First, all the elements are distinct, each tree has a distinct root. After the first* union *operation, a link is created between nodes* 2 *and* 3*. These two nodes have a common root now (here* 2*). At this point a* findRoot *operation on these two nodes would return the same representative (*2*). Finally, after the second* union*, all nodes are on the same tree and have the same representative (*2*).*

Union-Find. An Union-Find [21] is a set of two operations (*union* and *findRoot*) operating on disjoint data sets to track whether some elements are in the same connected component or not. Internally, it works by maintaining rooted trees. Elements are nodes of the tree and the root is the representative. A *findRoot* operation returns the root of the tree containing the given element. In practice, this operation is typically used to determine the connected component to which belongs a vertex. An *union* operation creates an arc between given distinct trees. This mechanism is illustrated Figure 2.14.

In practice, path compression and tree balancing are used to improve the complexity of these operations [82], leading to an amortized time per operation of $O\big(\alpha(n)\big)$ where n is the number of elements in the structure and α is the extremely slow growing inverse of the Ackerman function. ($\alpha(n) < 5$ for any value that can be written in the physical universe.)

Algorithm 1 connected operation

procedure CONNECTED(v,w)
  return findRoot(v) = findRoot(w)
end procedure

In the context of the incremental connectivity problem, the *connected* operation can be implemented using an Union-Find data structure by checking if the two vertices have the same root as shown in Algorithm 1. The *insert* operation is the same as the *union* operation.

2.4.2.3 Dynamic connectivity

In the case of the dynamic connectivity problem, arcs can either be added or removed from the graphs, which means components can merge together and split. The operations required by a data structure addressing this problem are:

- *connected(v,w)*: return true if v and w are connected.
- *insert(v, w)*: add an edge between nodes v and w in the graph.
- *delete(v, w)*: remove the edge v, w in the graph.

An efficient data structure addressing this problem is the ST-Trees, presented below.

ST-Trees. ST-Trees are dynamic graph data structures described by D. Sleator and R. Tarjan [74], based on vertex-disjoint paths. Each path is represented by an auxiliary data structure like binary search trees or splay trees [75]. Complexities achieved by ST-Trees are shown Table 2.1.

Operation	Amortized complexity
findRoot	$O(\log n)$
insert	$O(\log n)$
delete	$O(\log n)$

Table 2.1 – *Amortized complexities of ST-Trees functions for a graph of size n.*

In Table 2.1 the complexities presented are not exactly those for the dynamic graph connectivity problem. The *findRoot* operation returns the root of the tree containing a node. Similarly to the *findRoot* operation presented for the Union-Find data structure, it can be used to implement the *connected* operation like in Algorithm 1.

2.4.3 Ordered traversal

In computer science, priority queues are containers in which elements are retrieved according to a priority, for example the minimum first (according to some ordering criteria, such as function values). Usually, a priority queue guarantees a constant time lookup of the first element, at the expense of logarithmic insertions and extractions.

Breadth-first search can use priority queues to store simplex candidates. This way, simplices are visited in a sorted fashion, depending on the criterion used by the priority queue. In the following of this manuscript, we use breadth-first search traversals to visit vertices in the order of scalar values, thanks to the efficient priority queue detailed below.

Fibonacci heap The Fibonacci heap is a priority queue described by M. Fredman and R. Tarjan [21, 33] based on a collection of (binomial)

trees. This data structure offers low amortized time complexities as shown Table 2.2.

Operation	Amortized complexity
findMin	$O(1)$
insert	$O(1)$
delete	$O(\log n)$
merge	$O(1)$

Table 2.2 – *Amortized complexities of Fibonacci heap functions for a heap of size n.*

These low complexities are due to the heavy use of lazy operations. For example, the merge of two heaps into a single one is done in constant time by simply concatenating the two lists of internal trees. Inserting an element is equivalent to a merge with a one sized heap. It is only when the current first element is removed that the internal trees are consolidated, hence the logarithmic time of this step.

2.5 Parallel computing

As we have seen in the introduction, the main topic of this thesis is the design of efficient parallel algorithms for topological data analysis. Parallel computing consists in executing multiple operations simultaneously. In terms of hardware, parallel computing encompasses multi-core CPUs, many-core architectures and multi-node parallelism. These types of hardware and existing programming paradigms used to exploit them are presented in the following.

2.5.1 Multi-core parallelism

2.5.1.1 Hardware

Shared memory architectures became particularly developed in the years 2000 with the emergence of multi-core CPUs (Central Processing Units). Before this the computational power relied mainly on to frequency [53]. As shown Figure 2.15, a higher frequency also implies a higher consumption.

This growth in power was not sustainable anymore, so hardware manufacturers have changed their strategy in favor of parallelism. Let us see how this new model has addressed the power issue.

A comparison between two fictive electrical systems is presented Figure 2.16. The first one has a single processor and the second one two processors in parallel running at half the frequency to process the same

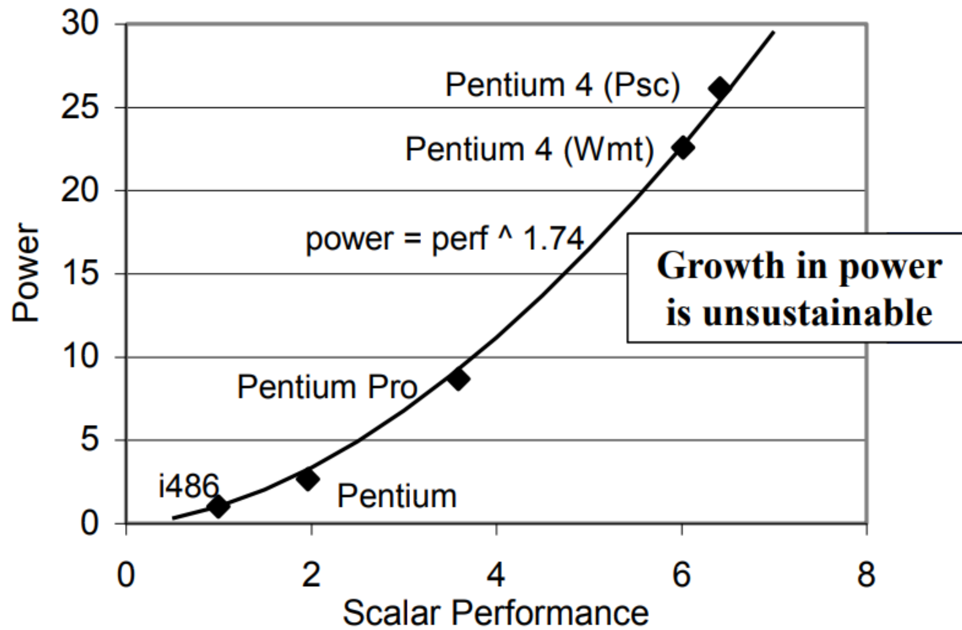


Figure 2.15 – *At the beginning of the years 2000, the* power consumption *used to grow almost quadratically with respect to the* scalar performance *(which reflect the compute power of the processor). This chart comes from* A "Hands-on" Introduction to OpenMP *[53].*

amount of input in the same time. In this scenario, the second system has a bit more than twice the capacitance of the first one as it has two processors. The voltage scales with the frequency so we consider the second voltage being at most 0.6 times the first one. This leads us to a same amount of computation per unit of time for only 40% of the power required for a single processor system: the power issue is addressed. But this scenario is only possible if the processing can be divided between cores: sequential algorithms need to be parallelized in order to exploit the full power of these architectures.

The CPU architectures. CPUs are designed to have a high serial compute power on each core. In 2018, we can target processors with up to 32 cores. Each core can process a stream of instructions called *thread*. Simultaneous Multi-Threading (SMT) is a technique aimed at improving the efficiency of the processor, by allowing two or four threads to be executed simultaneously on a single core. On Intel processors, SMT is named Hyper-threading.

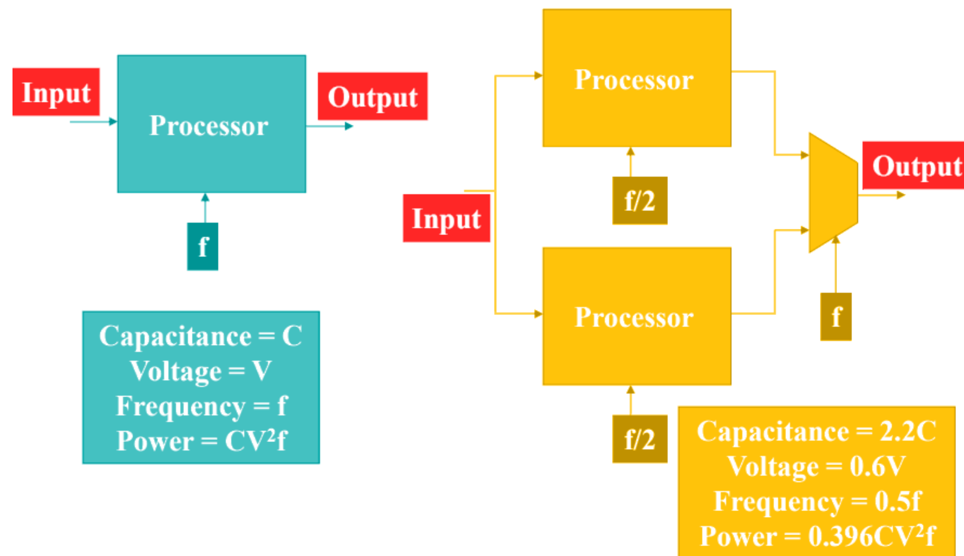


Figure 2.16 – *Two electrical systems able to process the same amount of data. The system with two processors only needs 40% of the power required by a single processor for the same processing power. (Example retieved from* A "Hands-on" Introduction to OpenMP *[53])*

NUMA effect. Present-day shared memory workstations may have several processors and several memory banks. In this case, the memory access time may depend on the memory location relative to the processor. As shown Figure 2.17, a processor may access its local memory bank faster than the memory bank of another processor. This is referred to as a NUMA (Non Uniform Memory Access) architecture. In such a case, data locality needs to be taken into account by parallel algorithms to achieve the best performance.

SIMD CPU vector (or SIMD — Single Instruction, Multiple Data) units can be used when the same operation can be performed on contiguous elements in a vector register. This mechanism relies on specific sets of instruction like SSE (128-bits), AVX (256-bits) or the newer AVX-512 (512-bits).

2.5.1.2 Programming

Thread-based programming. Multi-core parallelism can be achieved using thread-based programming, a paradigm focused on the creation and handling of threads within a single process. Explicit, low-level programming is available with POSIX threads and higher level programming can be done with specific programming interfaces, such as

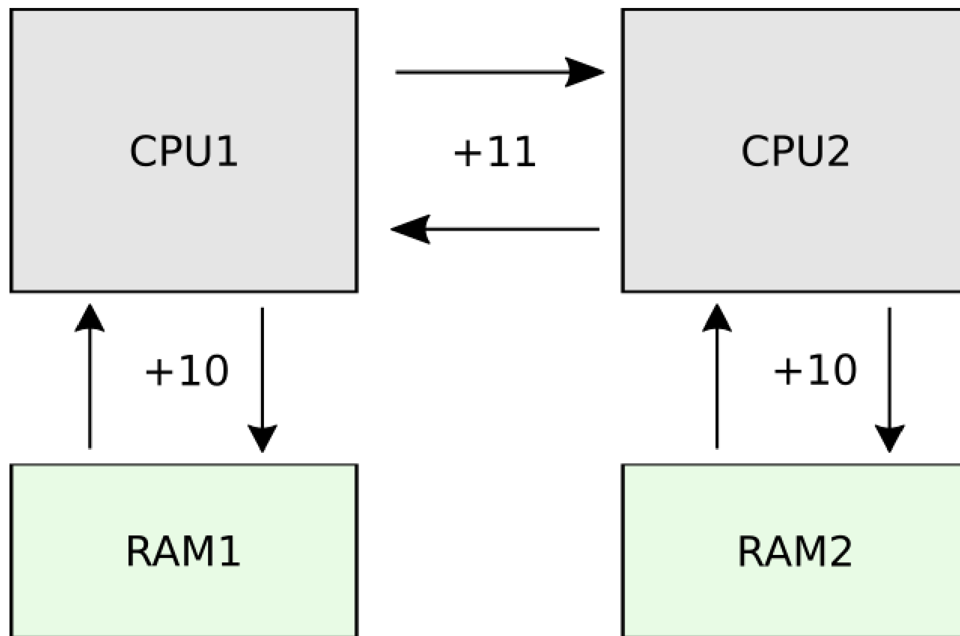


Figure 2.17 – *Simple example of NUMA effects on a dual processor architecture. In this example, CPU*1 *accesses memory in RAM*1 *in only* 10 *cycles, when it needs* 21 *cycles to access RAM*2. *These numbers are taken from one of our personal workstations.*

OpenMP [59], a non intrusive programming paradigm based on *pragmas* (or compiler directives).

Algorithm 2 Thread-based parallelism examples

Model parallel section
  Do in parallel
    ParallelJobA()
    ParallelJobB()
  End
EndModel
Model parallel for loop
  for $i = 0$ to n **do** in parallel
    IndependantProcessing(i)
  end for
EndModel

In Algorithm 2, examples of classical thread-based parallelism constructs are given. In the parallel section, several computations are started simultaneously and the end of the section is reached when the last computation is finished. In the parallel for loop, an independent processing is launched at each iteration of the loop. In practice, several

iterations (successive or not) can be given to each thread. We aim here at balancing the work equally among threads to achieve the best parallel efficiency (i.e. speedup divided by the number of cores). If the amount of work of each iteration is known prior to execution, the loop iterations can be equally distributed among threads using a *static* scheduling. Otherwise, the distribution of work is made at runtime: the loop is divided into small chunks and each available thread processes a chunk until none left. This is called *dynamic* scheduling, an overhead at runtime is induced by the chunk management.

When several threads are working simultaneously, concurrent data accesses are possible. If a thread accesses data being written by another thread, a *data race* occurs and leads to an undefined behavior. *Mutexes* (Mutual Exclusion) and *semaphores* are examples of low level mechanisms which can be used to synchronize threads and to ensure that a memory location is only accessed by a single thread at a time. OpenMP also provides *critical sections* and *atomic operations* via compiler directives. The critical section relies on a global lock, to ensure that a portion of code can be executed by at most one thread at a time. A name can be given to a critical section so that only the sections with the same name are mutually exclusive. Atomic operations are lighter synchronization mechanisms processing a single operation in an uninterruptible way, impacting only the corresponding cache line thanks to the cache coherency protocol of multi-core processors.

Task-based programming is a paradigm for multi-core parallelism introduced by Cilk [11] in 1994 that gained a greater interest in the last ten years. A task is a sequence of instructions within a program that can be processed concurrently with other tasks in the same program [84]. As illustrated Figure 2.18, tasks are stored in a pool of tasks on which available threads pick jobs to process using a dynamic scheduling.

As task-based programming relies on dynamic load balancing, it is well suited for *while* loops. It is also an efficient approach for recursive algorithms and nested parallelism, which is particularly useful to visit or construct hierarchical structures like trees or graphs. Using tasks usually offers better performance than parallel sections for nested parallelism. As a side note, it is interesting to remark that, internally, mutexes are attached to threads and not to tasks. If tasks are not tied to threads, using a mutex may thus lead to a deadlock.

Dependencies between tasks may be expressed to prevent the runtime

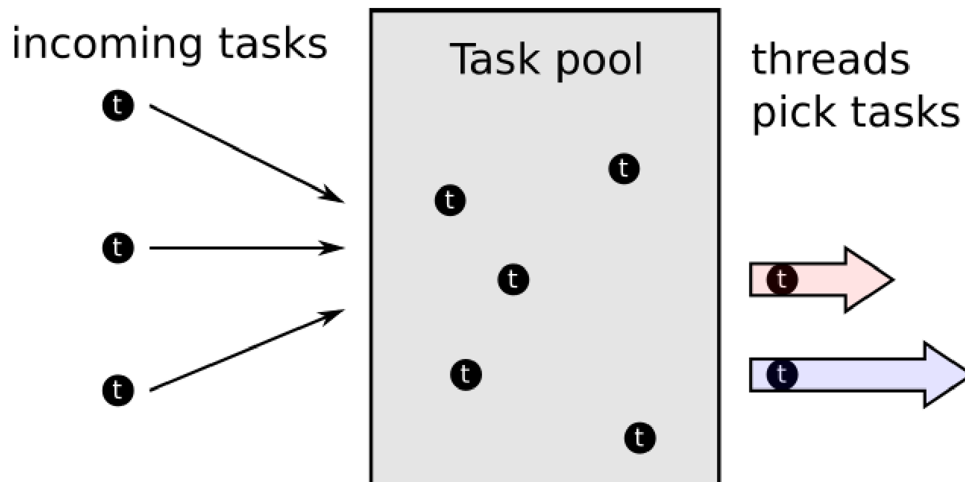


Figure 2.18 – *Example of a pool of tasks with dynamic load balancing. On the left, new tasks are added into the pool. On the right, two threads are picking tasks to execute. The computation loads of these tasks are not necessarily balanced.*

from executing a task before the end of another one. The most advanced runtimes like StarPU [6, 83], or OmpSs [27] make use of a dependency graph. These dependencies can be used to solve data races by preventing tasks operating on a same memory location to be executed simultaneously. Such runtimes can also use the dependency graph to distribute the work cleverly on heterogeneous architectures. A priority mechanism giving the runtime hints on which task to execute first is also potentially available and can be used to improve performance. This task parallelism has been progressively introduced in OpenMP [59], independent tasks first, then dependencies and lately priorities.

2.5.2 Many-core parallelism

Designed for a high degree of parallelism, many-core architectures offer a number of cores greater than CPUs, at the expense of reduced cache and memory sizes and lower single core performance.

2.5.2.1 Hardware

We present here some of the main many-core architectures used for scientific computing.

Graphics Processing Units (GPUs) have a significantly higher number of cores than CPUs [1] and so are able to execute instructions to more

[1]Note however that a GPU core does not match a CPU core, but rather a CPU SIMD lane.

data simultaneously. Additionally, GPUs have a simplified instruction processing. Each GPU core relies on in-order execution, without branch prediction. Also, GPU caches are significantly smaller than the CPU ones. The best performance is obtained for massive, regular and fine-grained data parallelism. Finally, data needs to be transferred from the processor memory to the GPU along with instructions to be executed as the GPU is a separate device: this can undermine the overall GPU performance.

Integrated GPUs. Starting from 2010–2011, Intel has introduced integrated GPUs (iGPUs) and AMD the Accelerated Processing Unit (APU) containing an iGPU. On these two devices, the GPU share the same die as the CPU and so can access its memory directly. This avoids the possible data transfer bottleneck of discrete GPUs. In the same way, they also offer reduced energy consumption compared to the CPU + GPU approach. However, their compute power and memory bandwidth are lower than discrete GPU ones.

Xeon Phi. The last many-core device to be presented here has been introduced by Intel in 2012. Xeon Phi are many-core processors designed to compete with GPUs but using up to 72 x86-compatible cores (288 threads using SMT). The first generation was designed as a PCI device, like discrete GPUs: data and instructions needed to be transferred on the Xeon Phi. The last generation is available as a standalone processor. In November 2017, the last Xeon Phi generation (Xeon Phi 7200, codenamed Knights Landing) has been discontinued by Intel in favor of another architecture built for exascale in the future.

2.5.2.2 Programming

For scientific computing, GPUs can be programmed using OpenCL [79], and CUDA [58] (only for NVIDIA). Xeon Phis support C, C++, Fortran and OpenMP [59] as well as Intel TBB [65] and MPI [32]. For graphic processing on GPU (like shaders or rendering), OpenGL [72] and Vulkan [49] can be used. Finally, some higher level programming tools support many-core architectures like OpenMP [59] or OpenAcc [94].

2.5.3 Multi-node parallelism

A shared memory architecture as previously seen is a single compute node. When higher levels of performance are required, several nodes can

be linked together to form a cluster. A supercomputer is a large cluster of nodes, designed for efficiency and linked together by high speed networks. These architectures allow to reach high compute power that could not be reached with a single machine for decades and to process data distributed among nodes that would not fit in a single workstation.

2.5.3.1 Hardware

These architectures differ from shared memory workstations by the distributed aspect of their memory and computational power. Transferring data between nodes can be slow (accessing the memory of the current node is 100x faster than accessing the one of an external node on the Titan supercomputer for example [9]) and may represent a bottleneck, especially for memory intensive computations having sparse memory accesses.

With such a computational power, saving large results may also represent a major bottleneck (transferring data between nodes is 10x faster than a disk access on the Titan supercomputer for example [9]). For this reason, *in-situ* visualization [67] is aimed to bring scientific visualization algorithms to run within the supercomputer along with the simulation to circumvent the bottleneck associated with saving and retrieving the data.

2.5.3.2 Programming

To exploit distributed architectures, parallel programming needs to be paired with multi-process programming in order to run simultaneously on distinct nodes. For more than 20 years, the HPC standard for multi-process parallel programming has been the Message Passing Interface (MPI) [32], a portable message passing standard. Several implementations are available, as well as bindings for other languages like Python, R or Matlab.

3 State of the art

Contents

3.1 Merge Trees 39
 3.1.1 Sequential reference algorithms 39
 3.1.2 Parallel algorithms 42
3.2 Contour Trees 46
 3.2.1 Sequential reference algorithm 46
 3.2.2 Parallel algorithms 47
3.3 Reeb Graphs 48
 3.3.1 Cut-based approaches 49
 3.3.2 Dynamic connectivity 50

Reference algorithms to compute merge trees, contour trees and Reeb graphs in sequential are detailed here. Corresponding parallel algorithms, when they exist, are introduced and discussed.

In this chapter we present the related work regarding the three contour-based topological abstractions: merge trees, contour trees and Reeb graphs presented in section 2.3. We focus on the augmented version of these abstractions.

3.1 Merge Trees

Merge trees (presented in subsection 2.3.3) are used to track sub/sur-level set components. In this manuscript, *join trees* are merge trees tracking sub-level set components and having minima and join saddles as nodes. *Split trees* are merge trees tracking sur-level set components having maxima and split saddles as nodes.

3.1.1 Sequential reference algorithms

3.1.1.1 Overview

The merge tree of piecewise linear data defined on a manifold simplicial complex can be computed using algorithms similar to the Kruskal's minimum spanning tree algorithm [14, 80, 91]. Carr et al. [16] described an algorithm which became the reference, with optimal time-complexity, good practical performance results and able to deal with data defined in arbitrary dimension. This algorithm relies on a vertex sweep in increasing order of scalar value (for the join tree), while maintaining an Union-Find data structure (see subsubsection 2.4.2.2) to track connected components of sub-level sets. This algorithm starts by a global sort of vertices by scalar value. For completeness, we recall here that vertices with identical scalar value can be distinguished using a consistent artificial noise thanks to a simulation of simplicity [30].

The main procedure to compute the merge tree is described in Algorithm 3. Initially, each vertex is associated to its own Union-Find component. For a join tree computation, vertices are visited in increasing order (line 2). For each vertex v_i, distinct Union-Find representatives on its lower link are added into a *cc* set (line 5). If this set is empty, the current vertex has no element in its lower link and is thus a minimum: a new arc is created (line 8). For each representative in *cc*, its corresponding arc is updated (line 11). This update operation consists in adding v_i to the list of regular vertices of this arc. Additionally, v_i is used as the new closing node of this arc. An *union* operation between lower Union-Find representatives and the current vertex is also done (line 12) to propagate

Algorithm 3 Merge tree construction: mesh traversal

```
1: procedure SWEEP(M)
2:     for all vertex v_i ∈ M by increasing scalar order do
3:         cc ← emptySet
4:         for all v_i^- in Lk^-(v_i) do            ▷ representatives in Lk^-(v_i)
5:             add(cc, findRoot(v_i^-))
6:         end for
7:         if |cc| < 1 then                         ▷ new arc on minimum
8:             newArc(v_i, findRoot(v_i))
9:         end if
10:        for all c ∈ cc do                        ▷ update arcs and Union-Find
11:            updateArcs(getArc(c), v_i)
12:            union(c, findRoot(v_i))
13:        end for
14:        if |cc| > 1 then                         ▷ new arc on join saddle
15:            newArc(v_i, findRoot(v_i))
16:        end if
17:    end for
18: end procedure
```

the corresponding sub-level set component. If the number of distinct Union-Find representatives in *cc* is greater than one, the current vertex is a saddle (line 14): a new arc is created, starting at this join saddle.

Figure 3.1 shows a join tree computation on a toy example. (1) The lowest vertex v_1 is visited first, this is the global minimum. Its lower link being empty, v_i is a leaf of the tree and a new arc (blue) is created. (2) The second vertex v_2 is a local minimum leading to the creation of another arc (yellow). (3) For the third vertex v_3, the lower link contains one vertex. v_3 has one Union-Find representative in its lower link so this is a regular node in the join tree. v_3 is added as regular vertex in the tree structure and an *union* is made between v_3 and the representative in its lower neighborhood. (4) The fourth vertex to be visited, v_4, has two distinct representatives in its lower link: blue and yellow. So v_4 is a join saddle. A new arc (green) is created and arcs ending on v_4 are closed. An *union* operation between the Union-Find representative of the yellow and blue arcs is made. The new representative is highlighted in green. (5) On the fifth step, the current vertex v_5 has a yellow and a green vertex in its lower link. Thanks to the previous *union*, both return the same (green) representative after the merge

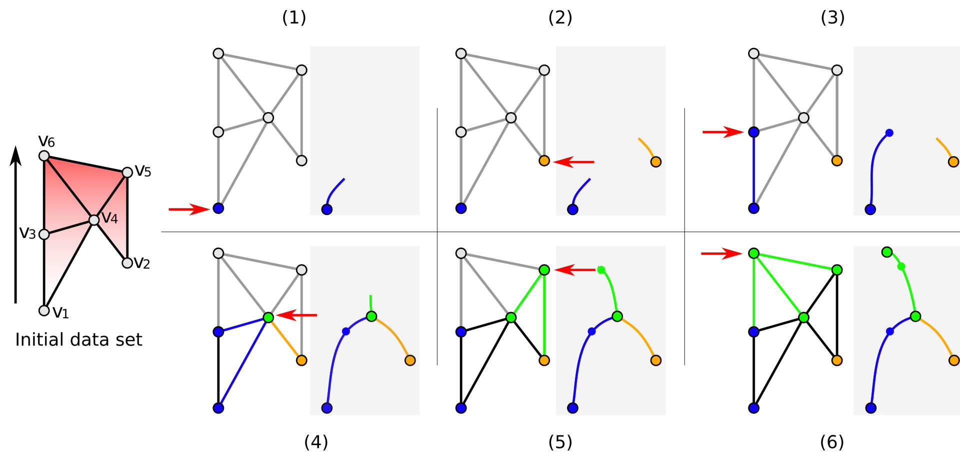


Figure 3.1 – *Join tree computation on a toy data set with a height scalar field. Nodes are colored according to the Union-Find representative (or arc) they correspond to. In this example, the blue and yellow components merge on a join saddle to form a new (green) component.*

so there is only one connected component of sub-level set in the lower link and v_5 is a regular vertex. (6) At the end, the last vertex is the global maximum. The arc reaching this vertex is closed, the tree is complete.

3.1.1.2 Complexity

This algorithm starts by a sort of all vertices. This can be done in $O(\sigma_0 \log(\sigma_0))$ steps, where σ_0 is the number of vertices of the mesh. The Union-Find data structure is used to visit the lower link vertices of each vertex using edges of the mesh. This step takes $O(\sigma_1 \alpha(\sigma_1))$ steps, where σ_1 is the number of edges in the mesh and α is the slow growing inverse of the Ackermann function. See *Worst-case Analysis of the Set Union Algorithms*, by Tarjan and van Leeuwen [81] for a complete explanation. This leads to a total time complexity of $O(\sigma_0 \log(\sigma_0) + \sigma_1 \alpha(\sigma_1))$ for the complete merge tree computation.

3.1.1.3 Non augmented merge tree

If the segmentation information (cf. subsection 2.3.3) is not required, another sequential algorithm can be used to compute the merge tree [20]. The idea is to extract all critical points of the mesh, then to compute monotonously decreasing paths starting at these critical points and ending at local minima. Finally, these paths are stitched together at saddles to form the skeleton of the tree. These steps are shown Figure 3.2. In theory, this algorithm only visits a sub-part of the geometry and should be faster

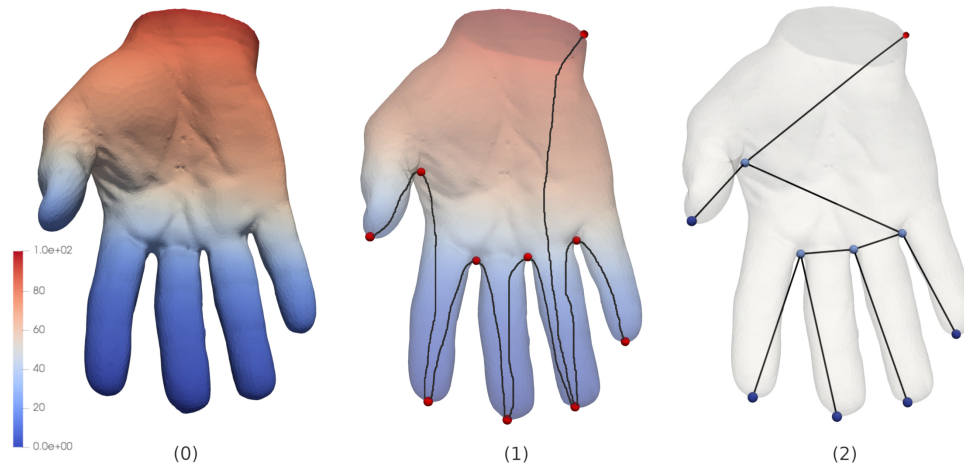


Figure 3.2 – *A toy example of a hand data set with an elevation scalar field (0). Critical points and the monotonously decreasing path are shown in (1). Finally, the skeleton of the tree is obtained by connecting these paths at saddles (remaining regular nodes are removed).*

than the previous algorithm. In practice, the saddle extraction takes almost as much time as the reference algorithm of Carr et al. [16].

3.1.2 Parallel algorithms

Several algorithms to compute merge trees in parallel already exist and are presented next. We focus here on shared memory architectures. Regarding distributed memory architectures, Morozov and Weber [55, 56] have presented two approaches to exploit merge and contour trees in a multi-node environment, minimizing inter nodes communications. Most of the papers presented in this section are aimed at computing the contour tree. But all these approaches only differ on how they compute the two internal merge trees used to obtain the contour tree (see next section). For this reason they are presented in this section.

We divide existing algorithms in two parts: in the *input sensitive* section, the degree of parallelism depends on the input mesh size, whereas in the *output sensitive* section the degree of parallelism depends on the topology of the output tree.

3.1.2.1 Input sensitive

Methods presented here are based on a static decomposition of the input domain.

Spatial decomposition. The first paper to compute the merge tree in parallel has been presented by Pascucci and Cole-McLaughlin [61] and is based on a spatial decomposition of the input data set. Using a *divide and conquer* approach, the domain is split into two halves recursively until only a single cell remains. The merge tree of piecewise linear scalar field defined on a single cell can be deduced directly. Then, cells and their local trees are merged back two by two until the original domain is reconstructed. In practice, it is possible to stop the recursive split when enough independent partitions have been created. The local merge tree of each partition can then be computed using the sequential reference algorithm. Figure 3.3 shows an example of this divide and conquer approach using four partitions.

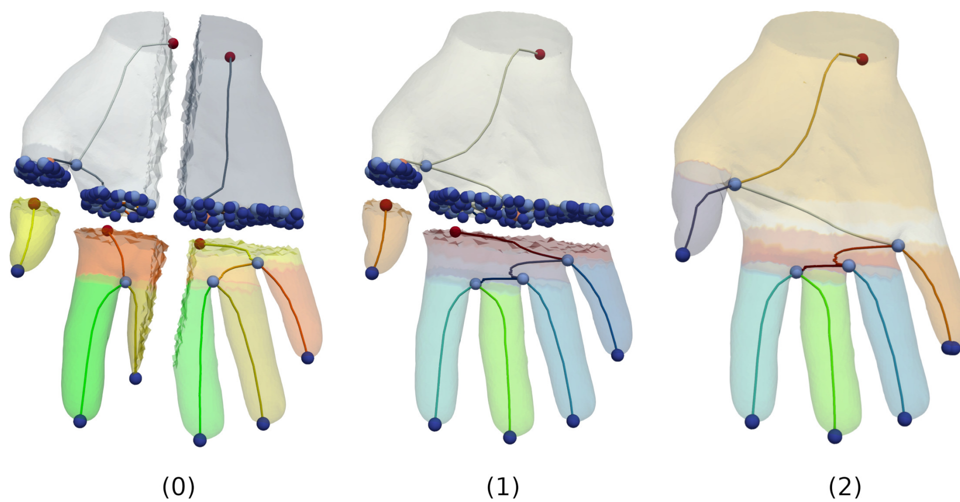


Figure 3.3 – *Divide and conquer algorithm used to compute the join tree of an elevation scalar field defined on the hand data set.*
(0)*: four partitions remain, a lot of noise is visible on the partitions boundaries.*
(1)*: partitions are merged two by two, noise on the merging boundaries is removed.*
(2)*: all partitions have merged, only one remains containing the final tree.*

This algorithm is particularly well suited for regular grids, where splitting the domain evenly is trivial. It can compute the augmented merge tree and is not restricted to barycentric linear interpolation for scalars. This approach is specially adapted to multi-core CPU and can also be exploited in a distributed architecture if the data set is divided spatially between nodes. In that case, the final tree obtained after all local tree merges still has to fit in a single node (Figure 3.3 (2)). A drawback of this algorithm is that, as cells merge, the amount of work increases (the boundary gets larger) but the parallelism degree decreases. At the end, the largest merge between the two last halves of the data set is done in serial. Moreover,

this approach does not guarantee that the work load is balanced among partitions, which can undermine the parallel efficiency. Finally, computing a merge tree requires to cut the mesh recursively into two halves, which is not trivial for unstructured meshes. This operation can be done in $O(n_l)$ time for each split where n_l is the size of the local domain to split.

Scalar value decomposition. Another approach to split the input mesh is to rely on scalar values. This approach is a contribution made in the context of this thesis and is the topic of the chapter 5. The main idea is to divide the input mesh using level sets of the input domain. Each partition thus obtained can be used to compute the merge tree locally. Finally, these merge trees are stitched together on the boundaries using a simple procedure identifying matching arcs. We refer the reader to the corresponding chapter for further details.

3.1.2.2 Output sensitive

Monotone paths. To compute the non-augmented merge tree, parallel versions of the monotone path based approach [20] have been proposed. The first one has been presented by Maadasamy et al. [52]. The initial critical point extraction is easily parallelizable as only local computations are involved (cf. subsection 2.2.1). As shown Figure 3.2, monotone paths grow independently and can be computed in parallel. Finally, these paths are connected hierarchically together in parallel using a few synchronizations. At the end, we obtain the final non-augmented merge tree. As some saddles may not be nodes of the merge tree, regular nodes may appear on the data structure. To the best of our knowledge, this is the first article presenting an algorithm that runs efficiently on GPU (thanks to its massive parallelism) for the merge tree computation.

This algorithm has been refined for the special case of regular grids by Acharya and Natarajan [3]. This new approach is a mix between the parallel monotone path version previously presented and the divide and conquer strategy [61]. On the input regular grid, several partitions are created on which the parallel monotone path approach is used. Then these local trees are merged back recursively to obtain the final non-augmented merge tree. Adapting this approach to run on GPU is left as future work.

These two methods offer good performance results on regular grids. The hybrid CPU-GPU approach offers 13x speedups using GPU compared to the sequential reference algorithm, while the refined algorithm is about

55x faster on 64 CPU cores compared to the reference sequential algorithm. However, the output of this algorithm is a non-augmented tree, which is less versatile application-wise, while the reference algorithm deals with all the segmentation information. Moreover, on unstructured meshes only the hybrid CPU-GPU approach can be used and requires at least four CPU cores to be faster than the sequential algorithm.

Path compression. Another massive data parallel approach on GPU, named Parallel Peak Pruning [17], has been presented in 2016 to compute the augmented merge tree. As the authors wrote themselves "this algorithm is somewhat complex", so the following explanations are just a summary. The core idea is to construct monotone paths from saddles to extrema and then iteratively "prune" peaks, i.e., cuts merge tree branches ending in an extremum. Each prune creates a new extremum-saddle region. In practice, these monotone paths are constructed from each vertex to an extremum. A path compression called pointer jumping [45] is used to label each vertex with its corresponding extremum. Then, all edges are sorted according to the extrema they lead to and saddles in their neighborhood to deduce extrema-saddle pairs and prune the corresponding regions. Every existing path leading to a pruned extremum is redirected to point to the corresponding saddle. At this point, monotone paths are compressed, edges sorted and extrema-saddle pairs pruned once more, until no saddle remain. At the end, unassigned vertices form the last arc of the tree, the root arc.

The sequential version of this algorithm is 40% slower on CPU than the sequential reference algorithm, however reported results show 9.2x speedups with 16 CPU cores over the sequential reference algorithm [16]. The GPU version on their data set is 21x faster than the reference algorithm on one CPU core. However, the algorithm itself is complicated and up to now, no performance results for 3-dimensional meshes have been reported. Moreover, efficiently computing augmented trees with this approach seems to be still an open problem.

Local propagations. Another output sensitive approach is to use local propagations corresponding to the arcs of the merge tree. This approach is a contribution made in the context of this thesis and is the subject of the chapter 6. To the best of our knowledge, this is the most efficient method to compute augmented merge trees.

3.2 Contour Trees

The contour tree is a topological abstraction tracking the connected components of level sets (contours) on a simply connected manifold. See subsection 2.3.2 for more details.

3.2.1 Sequential reference algorithm

3.2.1.1 Overview

The reference algorithm to compute the contour tree [16] is based on a 3-pass method on the data set. Two symmetric merge trees are computed, a join and a split tree. Then these two trees are *combined* together to form the final contour tree. Any method can be used to compute the two merge trees, but critical nodes of each tree need to be transferred in the other one before the combination. The main steps of this algorithm are the following.

1. Sort vertices by scalar value.
2. Construct the join tree by sweeping vertices in increasing order of scalar value.
3. Construct the split tree by sweeping vertices in decreasing order of scalar value.
4. Transfer critical nodes of each merge tree in the other one.
5. Combine the two merge trees into the final contour tree.

All these steps are either self-explanatory or previously described except for the combination of the two merge trees, which is described in the following section.

3.2.1.2 Merge tree combination

The combination algorithm is illustrated Figure 3.4. Initially, each merge tree needs to be augmented with the critical nodes of the other tree (step 0). Then, all leaves of both trees are added onto a queue θ (green arrows in step 1). While θ is not empty, its first leaf is taken out and its parent arc is processed. The node and its parent arc are added into the contour tree (steps 2 to 6). The node is also deleted from the two merge trees. If a new leaf is created in one of the original merge trees (like in steps 2 and 5), this new leaf is pushed into θ. For augmented trees, assuming a super arc representation as introduced in subsection 2.4.1, the list of regular vertices

of each processed arc is traversed and vertices not already in the contour tree are assigned to the newly created arc.

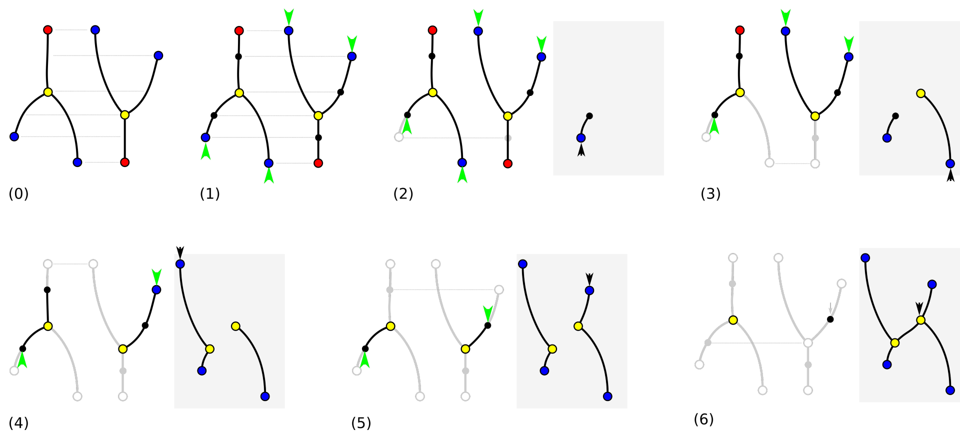


Figure 3.4 – *Example of a combination of a join (left) and a split (right) tree to construct the contour tree (grey background).*
(0) the join and the split tree to be combined.
(1) the join and split trees augmented with nodes of the opposite tree.
(2–6) leaves are removed one by one and added to the contour tree.
(6) the two merge trees have no remaining arcs, the contour tree is complete.

3.2.1.3 Complexity

The combination algorithm is a sweep on the two merge trees. In the case of non augmented trees, it takes $O(a)$ steps where a is the number of arcs of the output tree. In the augmented case, this algorithm traverses the list of regular vertices of each arc, therefore its complexity becomes $O(\sigma_0)$, where σ_0 is the number of vertices in the input mesh. Therefore, the contour tree processing is bounded by the two merge tree computations, each with complexity $O(\sigma_0 \log(\sigma_0) + \sigma_1 \alpha(\sigma_1))$, σ_1 being the number of edges in the input mesh.

3.2.2 Parallel algorithms

Parallel algorithms to compute the contour tree are based on the 3-pass method [16] previously described. All merge tree algorithms presented in section 3.1 can be used to compute the join and split trees in parallel. Additionally, as these two trees are independent, they can be computed simultaneously to add more parallelism. Most of these approaches were documented as using the sequential reference combination [3, 52, 61]. In the following, approaches that differ from the sequential reference algorithm are discussed.

The first article mentioning a parallel combination is the *Hybrid Parallel Algorithm for Computing and Tracking Level Set Topology* [52], in which the idea is to process leaves simultaneously. However, the parallel algorithm is not detailed and performance results are only presented using the reference sequential version.

Using a similar idea, the article *Parallel Peak Pruning* [17] presents a parallel combination on which upper leaves of the split tree and lower leaves of the join tree are processed in parallel alternatively. In this approach, nodes are not deleted in the merge tree during the arc processing but in an intermediate procedure after current leaves have been processed. This way, no data race occurs during the arc processing. Finally, consecutive regular vertices are collapsed in a single (super) arc.

Another parallel combination has been studied in the context of this thesis and is described subsection 7.2.4. Once again, leaves are processed in parallel step by step, but a final parallel procedure is described for the last monotone path.

Finally, even if the original combination algorithm is used it is worth to mention Contour Forest, detailed chapter 5. In this algorithm the combination is automatically computed in parallel as each independent partition can compute the full contour tree.

3.3 Reeb Graphs

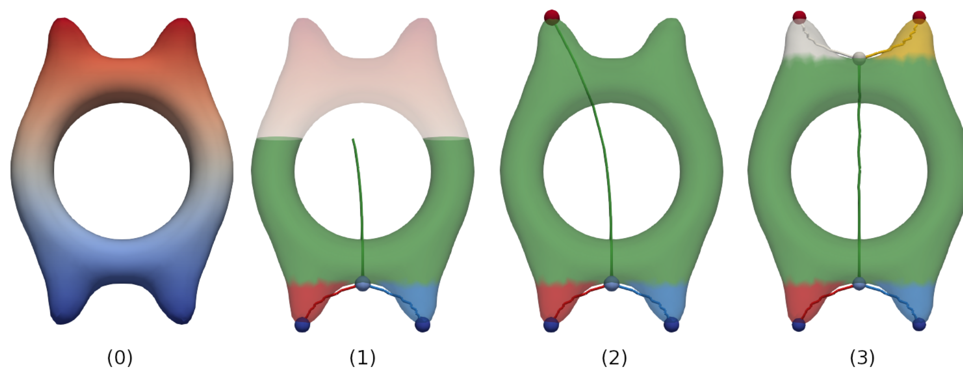


Figure 3.5 – *A toy example with a topological handle (the domain is not simply connected). (0) The heigh scalar field defined on the mesh ranging from blue (low scalar values) to red (high scalar values). (1) During the join tree computation, both sides of the handle corresponds to a same sub-level set components. The final join tree (2) does not contains information about the handle. The split tree is symmetric to the join tree on this simple example, the final contour tree (3) does not contain a loop.*

To compute the Reeb Graph, Union-Find based methods previously

described for the merge and contour tree computations cannot be used anymore. If the mesh is not simply connected, a level set component can split and merge back around a topological handle (see Figure 3.5), leading to a loop in the output graph. Such an event has no impact on the connectivity evolution of the sub/sur-level set: the contour tree algorithm [16] would thus miss the loops in the output data structure. In the following, we introduce existing algorithms to compute Reeb graphs.

3.3.1 Cut-based approaches

The first approach date back to 1991 [47] and is based on a systematic cut of the mesh on all vertices (leading to a quadratic complexity). In the early 2000, several methods based on quantized range contouring were presented [10, 44, 96]. These approaches are *approximated*, their complexity goes from linear to quadratic as we increase the precision of the approximation.

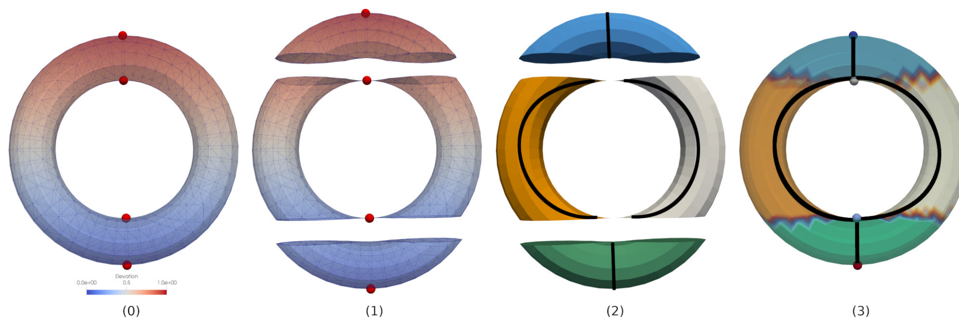


Figure 3.6 – *On a torus with a simple elevation, there are four critical points shown on (0). Using a cut-based approach [24], the mesh is cut at each saddle point as shown on (1). Each connected component consequently obtained (critical point excluded) is homeomorphic to a cylinder. These components are visited by monotone paths in (2) and glued together to obtain the output graph in (3).*

Focusing on exact methods cutting the mesh only on specific points, the first approach [64] has been introduced in 2008. It proceeds to a cut for each level set corresponding to a saddle point (we have seen in subsection 2.2.1 that the topology of level sets only changes at the vicinity of critical points). Arcs of the Reeb graph are then obtained by using an adjacency graph constructed from the regions of the domain delimited by the previously created cuts. A second approach presented in 2009 and named Loop surgery [86] proposes to cut the mesh to guarantee that the corresponding Reeb graph becomes loop-free and hence efficiently computable. Afterward, a contour tree algorithm is used and the Reeb graph is deduced by stitching facing arcs around each cuts. A third

approach has been documented in 2012 [24], which is also based on a cut of the mesh on all saddle vertices. Areas thus obtained are visited using monotone paths propagation, as shown Figure 3.6 and the Reeb graph is deduced. In 2013 was introduced an algorithm [26] on which the cut is only made on a subset of the saddles. A contour tree algorithm is used on the resulting domain similarly to the Loop surgery approach. Finally, a parallel algorithm [43] has been introduced in 2018, based on the monotone paths based approach [24]. In this algorithm, results have only been documented in 2D. To the best of our knowledge, this is the first and only existing parallel algorithm to compute Reeb graphs.

Because of the cut step, these algorithms have theoretically a quadratic worst case complexity. However, these are generally efficient approaches in practice as the quadraticity does not express in most real case data sets.

3.3.2 Dynamic connectivity

In 2007 was introduced an on-line algorithm [63] for Reeb graphs computations. This approach is able to operate in a streaming way, processing simplices of the 2-skeleton of the input mesh 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 $O(|v_0| \times |v_1|)$, where $|v_0|$ is the number of vertices and $|v_1|$ the number of edges of the input mesh. Even if this algorithm is sequential, its ability to process vertices in arbitrary order can be of great interest in an *in-situ* context, where the Reeb graph algorithm is executed alongside the simulation and processes parts of the mesh as they become available. The authors also present an "out-of-core" mode, exploiting the streaming nature of the algorithm to process the input data set by small pieces, without holding it entirely in memory. Additionally, the output graph is written to disk as simplices are visited, allowing to compute the Reeb graph on data sets too large to fit in memory.

3.3.2.1 Sweep approaches

The first algorithm [25] 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 is only available in 2D. In 2009

was introduced another method, 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 and is shown in Figure 3.7. This algorithm has been improved in 2013 [60] by the use of an ST-Trees data structure for the dynamic graph and the introduction of a laziness mechanism for the dynamic graph updates. This results in the algorithm with the best time complexity for the Reeb graph computation. It has a worst-case complexity of $O(m \log m)$, where m is the size of the 2-skeleton, this is certainly optimal if the number of edges and triangles of the complex is in the same order as the number of vertices. It obtains good performance results in practice, however it relies on a global view of the data which makes this approach intrinsically sequential. We address this problem using independent local propagations with the parallel Reeb graph approach presented in chapter 8. In the following, we detail this sequential reference algorithm [60].

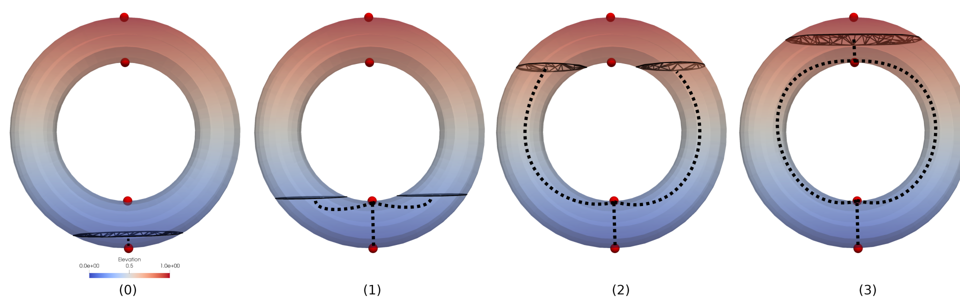


Figure 3.7 – *On a torus with a simple elevation, four level sets are shown. These level sets are computed on the 2-skeleton of the mesh. During the sweep algorithm, the history of the connected component is maintained, illustrated here by the dotted lines. This method outputs the augmented Reeb graph.*

3.3.2.2 Reference sweep algorithm

This algorithm starts by a global sort of the vertices by scalar values. The main procedure of this sweep algorithm is shown Algorithm 4. For each vertex, the preimage graph G_r is queried to deduce the number of contours in its lower star. This is done using the *LowerComponent* procedure detailed Algorithm 5. Then, the preimage graph is updated at the current scalar value to make the level set grow. This step is detailed in the next paragraph using Algorithm 6. The *UpperComponent* procedure used to deduce the number of contours in the upper star of the vertex is similar to the *LowerComponent* procedure. Finally, these numbers of contours are used to deduce the criticality of the current vertex in the Reeb graph $\mathcal{R}(f)$,

Algorithm 4 Reeb graph construction: mesh traversal

```
procedure SWEEP(M)
    for all vertices v_i ∈ M increasing scalar order do
        L_c = LowerComponent(v_i)
        UpdatePreimage(v_i)
        U_c = UpperComponent(v_i)
        if |L_c| ≠ 1 or |U_c| ≠ 1 then
            UpdateReebGraph(v_i)
        end if
    end for
end procedure
```

each contour corresponding to an arc of $\mathcal{R}(f)$. If the number of connected components of level sets in the vicinity of a vertex v goes from 0 to 1, v is a local minimum and a new arc is created in $\mathcal{R}(f)$, starting at v (see Figure 3.7 (0)). Symmetrically, if this number goes from 1 to 0, v is a local maximum and the corresponding arc in $\mathcal{R}(f)$ is closed. Otherwise, if the number of connected components in the vicinity of v is greater than one below or above v, v is a saddle vertex. Corresponding arcs of $\mathcal{R}(f)$ are updated like shown in Figures 3.7 (1) and (3).

Algorithm 5 Gather dynamic graph components below v_i

```
procedure LOWERCOMPONENT(v_i: current vertex)
    L_c ← empty list
    for all edges e ending at vi do
        c ← find(e)                      ▷ dynamic graph representative
        if c is not marked then
            add c to L_c
            mark c
        end if
    end for
end procedure
```

In Algorithm 5 the *LowerComponent* procedure is shown, the *UpperComponent* is symmetric. In the following, we consider that an edge *starts* at its vertex of lower scalar value and *ends* at the one with higher value. In this procedure, distinct representatives in the edges of the lower star of v_i (i.e. edges ending on v_i) are gathered and added into L_c. The size

of L_c, noted $|L_c|$, is the number of connected components of level sets in the lower star of v_i.

Algorithm 6 Impact the dynamic graph to make it growths above $f(v_i)$

1: **for all** triangles $t = v_1, v_2, v_3$ containing v_i **do**
2: assuming $f(v_1) < f(v_2) < f(v_3)$
3: **if** $v_i = v_1$ **then** ▷ new arc
4: insert((v_1, v_2),(v_1, v_3))
5: **end if**
6: **if** $v_i = v_2$ **then** ▷ update existing
7: delete((v_1, v_2),(v_1, v_3))
8: insert((v_1, v_3),(v_2, v_3))
9: **end if**
10: **if** $v_i = v_2$ **then** ▷ remove arc
11: delete((v_1, v_3),(v_2, v_3))
12: **end if**
13: **end for**

Dynamic graph update. The procedure used to make the preimage graph grow with the current level set is the procedure *UpdatePreimage* described in Algorithm 6. As this procedure is the key of the sweep algorithm, an example is shown Figure 3.8. In this example, a single triangle is processed, and each case of the *UpdatePreimage* is emphasized. To update the preimage graph G_r on a vertex v_i, all triangles containing v_i are visited. On each triangle, v_i can be the lowest, the middle or the highest vertex (corresponding to the lowest, medium and highest scalar values). If v_i is the lowest one (Algorithm 6 line 3), the growing level set is entering the triangle. An arc is added into G_r between the two lowest edges as shown Figure 3.8 (1). If v_i is the middle vertex of the triangle (Algorithm 6 line 6), an existing arc in G_r has to be updated as shown Figure 3.8 (2). Finally, if v_i is the highest vertex (Algorithm 6 line 10) the level set is growing outside the triangle and the arc of G_r is removed as shown Figure 3.8 (3).

This sweep algorithm is unrolled step by step on an example with a join saddle Figure 3.9. In this example, the first two vertices to be visited are local minima on (0) and (1). At this point, we can see the two distinct connected components of the dynamic graph in the lower star of the join saddle. This join is detected on step (2), and these two components merge

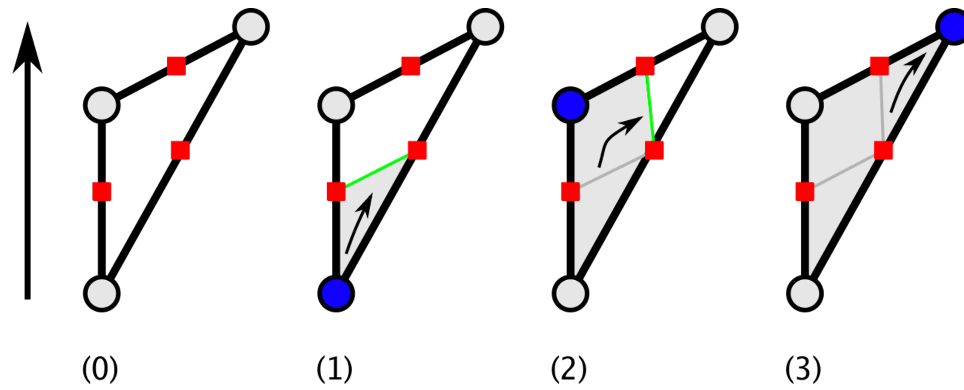


Figure 3.8 – *On a single triangle with an elevation scalar field* (0)*, the initial dynamic graph* G_r *has no arc, only nodes on the edges (red squares). When the lower vertex is first processed* (1)*,* G_r *is updated to correspond to the growing level set: the green arc is added between the two lower edges. The middle vertex is processed in* (2)*, the existing arc being replaced as shown to make the level set grow above the middle vertex. Finally at the vertex with the highest scalar value* (3) *the level set leaves this triangle and the arc of* G_r *is deleted.*

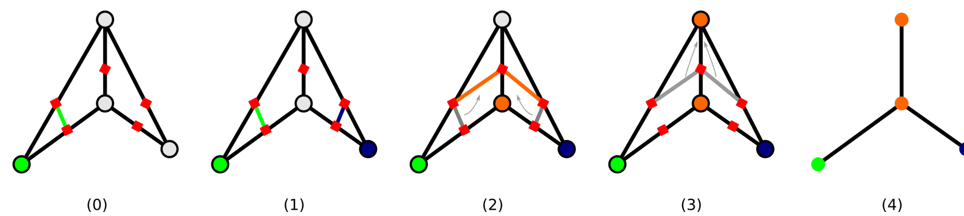


Figure 3.9 – *Two triangles with an elevation scalar field, and one join saddle at the center. The sweep algorithm executed on this data set starts by identifying two local minima in* (0) *and* (1)*. On step* (2)*, the current vertex is a join saddle. The last vertex visited on* (3) *is the global maximum and the output Reeb graph is shown on* (4)*.*

in a single one during the *UpdatePreimage* call. Finally, the maximum value is reached on step (3) and the last component disappears.

To reduce the number of operations on the preimage graph G_r a lazy evaluation mechanism is used. It consists in storing the additions of arcs to process on the preimage graph while only regular vertices are visited. When reaching a critical point, the graph G_r is updated using this list of operations. In practice, a significant number of operations are removed from the insertion list during the visit of regular vertices, without impacting G_r. However, this mechanism requires a critical point extraction in a preprocessing stage to determine vertices on which the preimage graph has to be updated.

The complexity of this algorithm given in the original paper [60] is $O(m \log m)$ where m is the size of the 2-skeleton of the mesh. The quadratic term of previously introduced Reeb graph algorithms is not present.

4 Positioning

For scalar field visualization, level set based topological abstractions are fundamental structures that enable the development of advanced data analysis, exploration and visualization techniques. Reeb graphs, contour trees and merge trees, which are the three level set based abstractions at the center of this manuscript, can be used for example in the context of: small seed sets extraction for fast isosurface traversal [18, 91], feature tracking [50, 76], data-summarization [62], transfer functions design for volume rendering [90, 93], similarity estimation [44] and automatic rigging [7, 66]. Their abilities to capture areas of interest in a robust and multi-scale way has been used in a variety of applications such as combustion [15], astrophysics [68], material science [31, 51], fluid dynamics [19, 42, 69, 77], medical imaging [12, 18], etc. These areas are defined using the mapping between vertices of the mesh and arcs of the graph/tree, only available when the graph/tree is augmented (cf. subsection 2.3.3). Therefore, to enable the full extent of level set based applications, we focus on algorithms computing these augmented abstractions in this manuscript.

The main objective of this work is to speed up the computation of level set based topological abstractions in order to improve the interactivity of data exploration. Indeed, the benefits of ten-fold speedups in terms of interactivity are presented in: "*Power of 10: Time scales in user experience*" [57]. Results presented in this study detail the impact of the computer response time on the user experience: for example a 0.1 second response time creates the illusion of a direct manipulation, a 1 second response time feels like the computer is causing the result but enables the user to stay focused on their current train of thought and a 10 second response time is more than what the short term memory can usually handle to maintain user focus.

In the context of data analysis, post-processing algorithms are usually run on a workstation composed of multi-core CPUs with shared memory

and possibly with GPU. We will thus target such workstations and not consider here distributed memory architectures. Current massively parallel approaches [3, 24, 43, 52] are suitable for GPUs and rely on fine grain parallelism (for example using one thread per input vertex). They are not able to compute augmented abstractions, as they are based on monotone paths starting at critical points and only visit a sub-part of the input mesh. The full mapping between all vertices of the input mesh and arcs of the tree is thus not computed. The only massively parallel approach theoretically able to compute augmented contour trees is the recent Parallel Peak Pruning algorithm [17], introduced in 2016. However, there is currently no implementation of this approach providing augmented trees and results have only been documented for a 2D implementation. Additionally, massively parallel approaches induce more total work than optimal sequential algorithms. For example, the parallel monotone path approach detailed in [52] is three times slower in sequential than the reference implementation for augmented trees (*libtourtre* [23], see Tab.1 in [52]). This only yields eventually speedups between 1.6 and 2.8 with regard to libtourtre on a 8 core CPU (20% and 35% parallel efficiency respectively). We suspect that these moderate speedups over libtourtre are due to the lack of efficiency of the sequential monotone path based algorithm [20] in comparison to the sweep approach [16]. Indeed, from our experience, although the extraction of critical points of the field is a local operation, we found in practice that its overall computation time is often larger than that of the contour tree itself. For these reasons, we have chosen not to rely on massively parallel algorithms, but rather to revisit the efficient sweep algorithm [16]. The sweep being a sorted traversal of the vertices of the mesh by scalar value, this approach relies on a global view of the data and is thus intrinsically sequential. A parallel version of this approach is likely to have a few number of independent sets of instructions and to rely on coarse-grain parallelism (each thread heaving a substantial amount of work). This type of parallelism is hence better suited for multi-core CPUs. Additionally, our algorithms involve a significant number of sparse memory accesses which make them not suitable for vectorization.

For parallel work, we propose a distinction between approaches whose parallelism degree depends on the input mesh size, hereafter named *input sensitive*, and those whose parallelism degree depends on the output graph topology, hereafter named *output sensitive*.

Our first algorithm, **Contour Forests** [38] (cf. chapter 5) is an input sensitive approach based on a static decomposition of the input mesh

by scalar values to compute augmented contour trees. Contrary to spatial based decompositions [3, 61] which involved additional work for unstructured meshes, this method is suited to both regular grids and unstructured meshes. It uses thread-based parallelism (with OpenMP [59]). The scalar decomposition is used to create partitions, each one being the area between two level sets. Contour trees of each partition can be fully computed in parallel and the global contour tree is obtained by stitching matching arcs on each contour at the interfaces between partitions. This approach is the first one using a fully parallel combination.

In order to improve the load balancing and to avoid redundant work, we present an output sensitive, parallel algorithm to compute augmented merge trees named **Fibonacci Task-based Merge trees** (FTM) [36] (cf. chapter 6). This approach is based on local propagations corresponding to arcs of the merge tree. These propagations visit the mesh locally in scalar order, using sorted breadth-first searches based on Fibonacci heaps priority queues (cf. subsection 2.4.1). These independent propagations can be expressed as parallel tasks: we use OpenMP [59], a widely available task runtime providing all the features we require (priorities, task groups, critical sections, atomic operations,...). Notice that we do not use any dependency graph, as we cannot predict our task terminations: hence we cannot exploit advanced task runtimes like StarPU [6, 83] or OmpSs [27].

For augmented contour trees, we present the **Fibonacci Task-based Contour trees** algorithm (FTC) [37] (cf. chapter 7). This output sensitive approach revisits the traditional 3-pass method described in [16] and benefits from the task-based nature of FTM for the concurrent computation of the two merge trees. It combines them using a newly introduced parallel algorithm.

Finally, the **Fibonacci Task-based Reeb graphs** algorithm (FTR) presented chapter 8 is another output sensitive, task-based approach, relying on local propagations to compute augmented Reeb graphs. Our experience with Contour Forests showed us that cut based approaches [24, 64] tend to involve redundant work and memory overhead depending on the number of cuts when parallelized. For these reasons, we choose to revisit a sweep based approach [60] similar to the one used for the merge tree computation. This approach has the best time complexity among Reeb graph algorithms but is intrinsically sequential. We address this problem by using local propagations relying on Fibonacci heaps, similar to the ones introduced for the merge tree computation in FTM. During

these propagations, a dynamic graph data structure (implemented as an ST-Tree) is used to track the connected components of level sets.

For all our algorithms, we have chosen to represent input meshes with simplicial complexes. This choice is made for practical genericity as any mesh (regular or unstructured) can easily be converted into a simplicial complex. Moreover, we restrict our study to manifold domains, as most commonly found in scientific visualization. Additionally, our output data structure is based on a super arc representation, each arc having a sorted array of regular vertices (as opposed to an explicit structure keeping all the small arcs between regular vertices). Our representation is efficient in memory, allows fast traversal of the structure and is a prerequisite of the new parallel combination presented in chapter 7.

Regarding technical aspects, for our implementations we rely on C++, which is relevant for high performance software. This choice is also motivated by Paraview [8] and VTK [70], two well-known open-source software packages for scientific visualization. Paraview is a GUI around VTK, both of them are implemented in C++ and can be extended through C++ plugins. All our developments are integrated in TTK [88], an open source platform aimed to help with topological data analysis. It has an integration with Paraview and VTK (using the plugin mechanism). Using TTK allowed us to implement our approaches focusing on the core of the algorithm, the Paraview/VTK wrapping being ensured at minimal cost by the platform. As said earlier in this section, our algorithms are likely to use coarse grain parallelism. For this reason we have chosen not to use VTK-m [54], an emerging toolkit targeting many-core architectures and relying on fine grain data parallelism

Tasks-based approaches presented in chapter 6, chapter 7 and chapter 8 rely on priority queues for the task growths. We have chosen to use Fibonacci heaps [21, 33] (see subsection 2.4.3) instead of Pairing Heaps [34] as these lead to better performance in practice in our implementations.

Part II

Contributions

5 Input sensitive Contour Trees using Contour Forests

Contents

5.1 Overview . . . 63
5.2 Scalar value based decomposition for parallel contour tree computations . . . 64
5.2.1 Domain partitioning . . . 64
5.2.2 Local computations . . . 66
5.2.3 Contour forest stitching . . . 67
5.3 Experimental results . . . 68
5.3.1 Detailed performance results . . . 70
5.3.2 Limitations . . . 71
5.4 Conclusion . . . 73

An input sensitive approach with a scalar value based partitioning strategy is presented for the parallel computation of the augmented contour trees. This work has been published in IEEE LDAV 2016 [38].

(a) (b)

Figure 5.1 – *Algorithm overview on the height function f of a volume $\mathcal{M}$ with two threads. (a) Input scalar field f (color gradient) with its critical points (blue: min, white: saddle, green: max). The domain is split into two partitions $\mathcal{P}_i$ and $\mathcal{P}_j$ of roughly equal size corresponding to the pre-images of contiguous intervals $\mathcal{I}_i$ and $\mathcal{I}_j$ of $f(\mathcal{M})$. The* interface level-set *between such two partitions is shown in red. (b) The augmented contour trees $\mathcal{C}(f)_i$ (top) and $\mathcal{C}(f)_j$ (bottom) are constructed in parallel for each partition. These local trees can be easily and efficiently stitched together to form the output augmented contour tree (right).*

In 2016, the only parallel approach [61] to compute the augmented contour tree relied on a spatial decomposition. In this chapter, we present *Contour Forests*, a new approach which decomposes the input data set based on the scalar value of the mesh vertices. The parallelism degree thus mostly depends on the input mesh, leading to an input sensitive approach. In this chapter, we present the following new contributions:

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

2. the first method with a fully parallel combination algorithm.

5.1 Overview

Our approach is based on a range-driven partitioning strategy, as illustrated in Figure 5.1. First, given n_t threads, the image of the domain $f(\mathcal{M})$ is divided into $n_t/2$ contiguous, non-overlapping intervals $\mathcal{I}_i$ that contain (nearly) the same amount of vertices of $\mathcal{M}$ (subsection 5.2.1):

$$f(\mathcal{M}) = \mathcal{I}_0 \cup \mathcal{I}_1 \cup \cdots \cup \mathcal{I}_{(n_t/2)-1} \tag{5.1}$$
$$|\sigma_0|_i \approx |\sigma_0|_j \quad \forall i \neq j$$

where $|\sigma_0|_i$ refers to the number of vertices of $\mathcal{M}$ mapping to $\mathcal{I}_i$. Next, two threads are assigned to each partition $\mathcal{P}_i$, $\mathcal{P}_i$ being the pre-image of the

corresponding interval, $\mathcal{P}_i = f^{-1}(\mathcal{I}_i)$ (subsection 5.2.1). The two threads then compute the augmented contour tree of the restriction of the function to its partition (subsection 5.2.2), with a variant of the algorithm by Carr et al. [16]: one thread builds the join tree, and the other the split tree[1]. This yields a *forest* of contour trees: $\{\mathcal{C}(f)_0, \mathcal{C}(f)_1, \ldots \mathcal{C}(f)_{n_t-1}\}$. Finally, the output contour tree is retrieved by connecting the trees of the forest along common connected components of partition boundaries (subsection 5.2.3).

Despite its simplicity, our range-driven approach exhibits many advantages. In particular, our strategy enables the computation of augmented contour trees, since it extends Carr et al.'s algorithm. Second, since the input mesh is split into partitions of roughly equal size (in terms of vertices), the work load should be well balanced between the threads. Third, since it is range-driven, our approach allows for a full computation of the local contour tree within each partition (join and split tree computations, plus their combination in the contour tree) while previous approaches systematically delayed the combination to a post-process pass (implemented in serial). Finally, since it is range-based, our approach allows for a simple stitching of the local trees of the forest into the output contour tree, while previous approaches needed to run a special procedure on the common boundary of merged partitions: [52, 61].

5.2 Scalar value based decomposition for parallel contour tree computations

This section details the algorithms for each step of our approach. As a reminder, the term *edge* in this manuscript refers to a 1-simplex of the input mesh, while the term *arc* refers to a 1-simplex of the output tree (or graph).

5.2.1 Domain partitioning

The first step of our approach consists in sorting the vertices of $\mathcal{M}$ by increasing function value, which can be efficiently done in parallel [35, 73, 89]. This step can be done in $O(|\sigma_0| \times \log(|\sigma_0|))$ where $|\sigma_0|$ is the number of vertices in $\mathcal{M}$. Next, the sorted list of vertices is split into $n_t/2$ contiguous sets $\mathcal{P}_i$ of roughly equal size, whose images correspond to the intervals $\mathcal{I}_i$

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

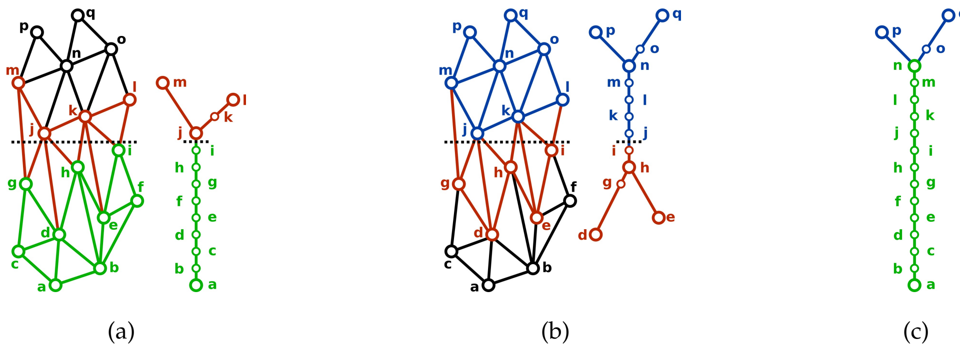


Figure 5.2 – *Domain partitioning for a 2D toy example (height function). (a) Partition $\mathcal{P}_i$ (green) with its overlap simplices (red) and its augmented contour tree $\mathcal{C}(f)_i$. (b) Partition $\mathcal{P}_j$ (blue) with its overlap simplices (red) and its augmented contour tree $\mathcal{C}(f)_j$. The common region between the two partitions is made of all the triangles containing red edges and being crossed by the interface level-set (black dashes). (c) The output, stitched, augmented contour tree $\mathcal{C}(f)$.*

described in Equation 5.1. Next, each vertex set $\mathcal{P}_i$ is extended into a set $\mathcal{P}'_i$ with the following procedure. Let f_{i^-} and f_{i^+} be the two extremities of the interval $\mathcal{I}_i$: $\mathcal{I}_i = (f_{i^-}, f_{i^+})$. The level-sets for the isovalues f_{i^-} and f_{i^+} are called *interface level-sets*. Let $(\sigma_1)_{i^-}$ and $(\sigma_1)_{i^+}$ be the set of edges of $\mathcal{M}$ whose image contains f_{i^-} and f_{i^+} respectively. The vertex set $\mathcal{P}_i$ is extended into $\mathcal{P}'_i$ by adding the vertices of $(\sigma_1)_{i^-}$ and $(\sigma_1)_{i^+}$ (red circles, Figure 5.2). We call such vertices *boundary vertices*. Note that with this approach, two adjacent partitions $\mathcal{P}'_i$ and $\mathcal{P}'_j$ will overlap, precisely along the simplices crossed by f_{i^-} or f_{i^+} (triangles with red edges, Figure 5.2).

This strategy guarantees that each connected component of an interface level-set is captured by the overlaps in between the partitions (triangles with red edges, Figure 5.2). Therefore, all possible contours living in the interval $\mathcal{I}_i$ are completely captured by $\mathcal{P}'_i$. This guarantees that the restriction of the local contour tree $\mathcal{C}(f)_i$ (computed on $\mathcal{P}'_i$) to the interval $\mathcal{I}_i$ (in green in Figure 5.2(a) and blue in Figure 5.2(b)) is equal to the restriction of the output contour tree $\mathcal{C}(f)$ to $\mathcal{I}_i$. This property will be of paramount importance to guarantee an efficient stitching of the contour forest into the output contour tree (subsection 5.2.3).

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

Other methods. The partitioning method presented previously is the most efficient one we have tested. Our first idea was to assign an Union-Find representative corresponding to a virtual extrema on each contour of the interface level sets. This required to process a BFS to extract these contours on each interface level set, which in practice took almost as much time as the full contour tree computation. Another idea was to stop the computation at the boundary level set (without overlap) and to deal with the noise after the local tree computation, during the stitching step. But the procedure to remove the noise, similar to the zipping procedure used in [63] would replace a linear step by a quadratic one.

5.2.2 Local computations

The contour tree $\mathcal{C}(f)_i$ of each of the $n_t/2$ partitions $\mathcal{P}'_i$ is computed by two distinct threads. Note that in practice, the partitions $\mathcal{P}'_i$ are not copied, but represented implicitly. In particular, the list of vertices of the initial partition $\mathcal{P}_i$ is represented by an interval in the global sorted list of vertices. The boundary vertices added in the expansion procedure described in subsection 5.2.1 (red circles, Figure 5.2) are represented by two sorted lists of vertices $\mathcal{B}_{i^-}$ and $\mathcal{B}_{i^+}$, representing the boundary vertices below f_{i^-} and above f_{i^+} respectively.

Given a partition $\mathcal{P}'_i$, its augmented contour tree is computed with a variant of the Carr et al.'s algorithm[16], described subsection 3.2.1. This algorithm has time complexity of $O\big(\sigma_0 \log(\sigma_0) + \sigma_1 \alpha(\sigma_1)\big)$, where σ_0 is the number of vertices in $\mathcal{M}$ and σ_1 the number of edges.

Our approach to the local computation of the augmented contour tree $\mathcal{C}(f)_i$ for each partition $\mathcal{P}'_i$ only requires a slight modification to this algorithm. In particular, when constructing the join tree $\mathcal{T}^-(f)_i$, our algorithm first visits the boundary vertices $\mathcal{B}_{i^-}$ (if any) in increasing order. Next, it visits the vertices of $\mathcal{P}_i$ by traversing the global sorted vertex list within the interval prescribed by the domain partitioning step (subsection 5.2.1). Finally, it completes the traversal by considering the vertices of $\mathcal{B}_{i^+}$ (if any) in increasing order. For each of these three traversals, the join tree construction algorithm [16] is applied as-is by the corresponding thread. The split tree $\mathcal{T}^+(f)_i$ is constructed with a symmetrical pass by the other thread: $\mathcal{B}_{i^+}$, then $\mathcal{P}_i$ and finally $\mathcal{B}_{i^-}$. Once the join and split trees are constructed, they are combined into the augmented contour tree $\mathcal{C}(f)_i$ with the original algorithm ([16]) by one of the two threads. This combination does not require parallelization

within each partition since its computation time is not significant, and since parallelization already applies among partitions.

5.2.3 Contour forest stitching

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

During the local computation of the contour tree $\mathcal{C}(f)_i$ (see subsection 5.2.2), each 1-simplex (each arc) that crosses an interface level set is added to a list of *crossing arcs*, noted $\mathcal{X}_i$. This corresponds in Figure 5.2 to the arcs (a, j) in Figure 5.2(a) and (h, n) in Figure 5.2(b)

Then, the stitching procedure consists in visiting sequentially the list of crossing arcs $\mathcal{X}_i$ for each local contour tree $\mathcal{C}(f)_i$. Given such an arc a_i, its regular vertex v exactly above f_{i^+} (or below f_{i^-}) is identified through a dichotomic search (vertex j in Figure 5.2(a)). Since augmented contour trees store the destination of each vertex into the tree, it is possible to retrieve in constant time the *homologous* arc a_j from the adjacent tree $\mathcal{C}(f)_j$ which contains v. This corresponds to the arc (h, n) in Figure 5.2(b). Finally, a_i is updated to form the union of the arcs a_i and a_j. This operation includes the modification of the higher extremity of a_i (to use a_j's instead) as well as the concatenation of the two sorted lists of regular vertices (see Figure 5.2(c)). Note that the vertex v can belong to multiple partitions. In such a case, a_i will be updated iteratively to form the union of multiple arcs (a_i, a_j, a_k, etc.), by successively applying this pairwise stitching in increasing order of function value (i.e. a_i and a_j will first be stitched, then the result of this stitching will be stitched with a_k and so on). As discussed in section 5.3, this final stitching procedure is extremely fast in practice (hence performed sequentially), since only a small portions of the arcs are visited (only the crossing arcs) and since the merging operation is simple (it simply consists in stitching pairs of arcs across interface level-sets). Note that the simplicity of this stitching procedure is due to our domain partitioning strategy, which guarantees that the restriction of a local tree $\mathcal{C}(f)_i$ to the interval $\mathcal{I}_i$ is equal to the restriction of $\mathcal{C}(f)$ to $\mathcal{I}_i$ (subsection 5.2.1). Note that the zipping procedure employed in the streaming Reeb graph computation algorithm by Pascucci et al. [63] could also be employed for the stitching of the local trees. However, this procedure admits a quadratic time complexity in the number of nodes

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

Table 5.1 – *Running time of the different steps of the algorithm (in seconds).* $\lvert\mathcal{M}\rvert$ *denotes the number of vertices in the data-set, and* $\lvert\mathcal{C}(f)\rvert_A$ *the number of arcs in the output contour tree. Overall corresponds to the complete application, including memory allocations, etc.*

of $\mathcal{C}(f)$, which is prohibitive in our approach, where only sub-quadratic routines have been used.

Segmentation

In terms of implementation, we rely on the notion of super arc introduced subsection 2.4.1. This tree representation has the same information than the explicit one storing all arcs between each regular vertices but allows for a faster computation of the merge tree by marking vertices first during the computation and retrieving the lists or regular vertices of each super arc in parallel in a separate pass. It is also efficient for the combination algorithm as it can deal with the list of regular vertices of an arc directly.

5.3 Experimental results

In this section, we present practical results obtained with a VTK-based C++ implementation of our algorithm (publicly available in TTK [88]). Experiments were performed on a desktop computer with an Intel Xeon CPU E5–2630 v3 (2.4 GHz, 8 cores) with 64GB of RAM. All parallel tests are run with $n_t = 8$ threads for $n_p = 4$ partitions. Other results in this thesis are presented using two CPUs, the choice of using only one for this algorithm is due to both its sensitivity to NUMA effects (see subsection 2.5.1) and mainly to the redundant work induced by the split step (further investigated in subsection 5.3.2) which prevented our algorithm to achieve good speedups on 16 cores.

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

Table 5.2 – *Overall running time comparison (in seconds) between the sequential libTourtre implementation (sTourtre), a naive parallel implementation of libTourtre (pTourtre) and our approach.*

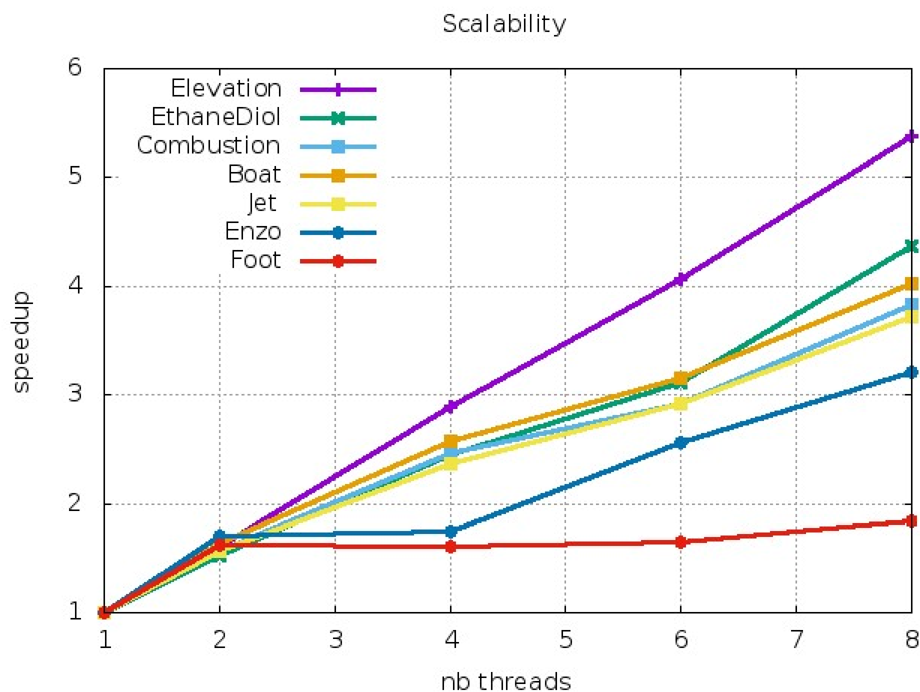


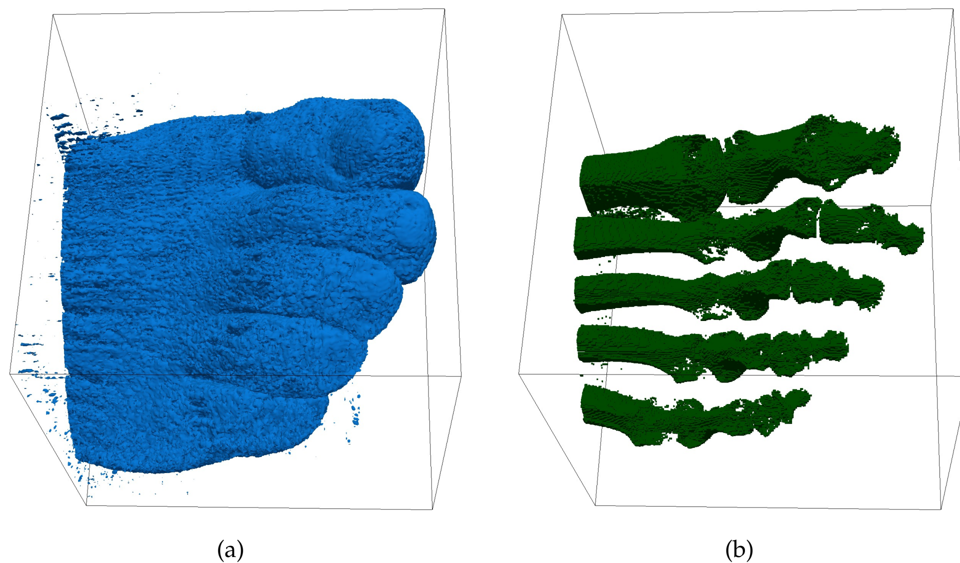
Figure 5.3 – *Speedups obtained by our contour forests implementation as a function of the number of threads (one curve per data set).*

5.3.1 Detailed performance results

Table 5.1 first presents detailed performance results for various data-sets. Plasma, Bucky and SF Earthquake are standard data-sets available at the [4] repository: these are however too small to fully exploit our 8-core CPU. In fact, with our experimental setting, even the sort step is slower in parallel than in sequential for these data-sets (results not shown). This explains the low parallel speedups for such tests.

We thus focus in the following on larger tetrahedral meshes (upper part of Table 5.1) which have been obtained by triangulating regular grids. For the sake of comparison, these have systematically been upsampled to 256^3 vertices. One can first see that the additional Overlap step required to build the $\mathcal{B}_{i^-}$ and $\mathcal{B}_{i^+}$ lists (see subsection 5.2.2) leads to low overheads. Moreover, the stitching step is efficiently performed in sequential which results in small run-times. As far as parallel speedups are concerned, the Elevation data-set is a synthetic and very simple one that shows good speedups, with a parallel efficiency of $5.38/8 = 67\%$. Moving to more complex data-sets (i.e. resulting in larger contour trees), one can see that we obtain good or average speedups (parallel efficiencies ranging between 55% and 40%), except for the Foot data-set which shows a limited speedup. We will detail these limitations and their causes in the next sub-section. The scalability of our approach is evaluated with Figure 5.3, which presents the evolution of the speedup obtained by our algorithm as a function of the number of threads. The slope of these curves shows that the scalability of our approach, similarly to the speedups discussed above, is data-set dependent as well. In particular, our algorithm seems less scalable for the data-sets which result in complex output trees (see the third column of Table 5.1 which shows the number of arcs per tree). Also, the smallest slope (nearly constant) is also observed with the Foot data-set, as further discussed in the next sub-section.

Next, we compare our parallel implementation with the reference sequential Libtourtre implementation in Table 5.2. Our speedups with respect to Libtourtre range between 2.6 and 3.8 (except for Foot), which compares favorably (even if the data-sets differ) with the 1.6–2.8 speedups of the approach based on parallel monotone paths [52], on tetrahedral meshes. Note additionally that in contrast to our approach, the method presented in [52] does not compute the augmented tree. We also added in Table 5.2 performance results of a naive parallel implementation of Libtourtre (library implemented by Dillard [23]), which uses a parallel sort



(a) (b)

Figure 5.4 – *Size difference between two interface level-sets on the Foot data-set. (a) Interface crossing* 1,507,357 *edges (blue). (b) Interface crossing* 606,276 *edges (green). This difference can lead to load imbalance between our partitions.*

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

Table 5.3 – *Partition sizes (in vertices).*

and two OpenMP threads to build independently the join and split trees. Our parallel algorithm outperforms this naive implementation in all our test cases with exception of the foot, which further stresses the efficiency of our approach.

5.3.2 Limitations

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

As presented in Table 5.3, we can see that the actual number of vertices per partition is always greater than the ideal one (obtained by dividing the total number of vertices by the number of partitions). This is due to the

Data-set	8 threads: min	max	1 thread: min	max
Elevation	1,623,500	1,768,720	2,151,250	2,292,390
EthaneDiol	963,962	1,108,170	1,470,960	1,804,410
Combustion	1,029,050	1,190,160	1,688,080	2,006,210
Boat	1,055,410	1,237,030	1,463,880	1,985,720
Jet	1,065,720	1,256,730	1,754,240	2,094,010
Enzo	860,937	933,616	1,166,540	1,366,070
Foot	1,120,560	4,031,030	1,220,800	5,195,250

Table 5.4 – *Computation speeds (in vertices/second) for join or split tree computations with our parallel implementation and with one single thread to perform all computations required by our parallel approach.*

boundary vertices that have to be added to each partition, which imply redundant computations that directly impact the parallel speedups. In particular, there can be important variations in the size of the interface level sets (in terms of crossed edges, in red in Figure 5.2) within a single data-set, as shown in Figure 5.4. Therefore the size of the overlaps between the partitions (expressed as the number of vertices in the lists $\mathcal{B}_{i^-}$ and $\mathcal{B}_{i^+}$, see subsection 5.2.2) can also vary. This induces redundant computations of varying importance within a single data-set. One can also see larger imbalance in the number of vertices per partition for more complex data-sets. This adds load imbalance to the parallel computations which further decreases the speedups.

This load imbalance is worsened by the fact that, depending on its impact on the join and split tree constructions, each vertex of $\mathcal{M}$ does not require the same processing time in practice. This effect is shown in Table 5.4 which shows varying computation speeds among the different partitions of a given data-set. In particular, the more complex is the contour tree, the larger is the gap among the computation speeds. One could choose to use several partitions per thread, with dynamic load balancing, in order to minimize such load imbalance, but this would introduce even more redundant computations (because of the boundary vertices). That is why we choose to use $n_t/2$ partitions for n_t threads: this indeed minimizes the redundant computations, while fully exploiting the complete independency between the join and split tree computations.

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

Finally, a third factor also limits our parallel efficiencies for any data-

set. The contour tree computation requires on average very few operations with respect to the number of memory accesses. Its operational intensity (see [95]) is therefore low which makes such an application memory-bound, like most graph traversal algorithms ([5]). As shown in Table 5.4, giving the parallel computations of the join and split trees to a single thread leads to higher computations speeds. Hence speedups linear in the number of cores cannot be obtained for such memory-bound applications: the memory bandwidth of the processor can not cope with the memory requirement of the height threads at a time. This also justifies our choice not to rely on the 2-way SMT (*Simultaneous Multi-Threading*) capability of our CPU, and to use only one thread (instead of two) per physical CPU core.

5.4 Conclusion

The approach presented here is an efficient algorithm to compute augmented contour trees of scalar fields defined on both unstructured meshes and regular grids. This method relies on a subdivision of the input data set by scalar value, allowing to compute in parallel the full contour tree of each partition thus obtained. In our tests, this algorithm compares favorably to a reference implementation. However, adding more threads leads to redundant computations during the split step and prevents this approach to obtain a good parallel efficiency when using a high number of threads on most data sets. Moreover, computation times of local trees are not balanced among partitions, leading to threads becoming idle.

Using a high number of threads, the mesh may be cut in such a way that the corresponding Reeb graph is loop free (if all topological handles have been cut for example). In that case, after the stitching step the final structure of Contour Forests would be the Reeb graph, loops being reconstructed by the stitch procedure. However, a loop entirely contained in a partition would not be cut and thus would be missing in the final output (the local contour tree does not contain it). As a consequence, the number of partition can be used to control the minimum size (in terms of scalar values) a loop should be to ensure it is present in the output of the algorithm.

6 Output Sensitive Task-based Merge Trees with Fibonacci Heaps

Contents

6.1 Overview 78
6.2 Local propagations for merge tree computations 79
- 6.2.1 Leaf search 79
- 6.2.2 Leaf growth 79
- 6.2.3 Saddle stopping condition 81
- 6.2.4 Saddle growth 82
- 6.2.5 Trunk growth 84
- 6.2.6 Segmentation 85

6.3 Task-based parallel merge trees 85
- 6.3.1 Taskification 86
- 6.3.2 Synchronization 87
- 6.3.3 Parallel trunk growth 88

6.4 Results 89
- 6.4.1 Performance analysis 90
- 6.4.2 Limitations 95

6.5 Conclusion 97

An output sensitive approach with task-based independent local propagations is presented for the parallel computation of the augmented merge trees. This chapter presents the works of two

publications. The first version of this algorithm has been published at IEEE LDAV 2017 [36]. Then, performance results have been improved and the algorithm refined for the case of the augmented contour tree, leading to a paper accepted to the IEEE TPDS [37] journal (to appear).

In this chapter a new parallel algorithm to compute augmented merge trees is presented. It is based on independent local propagations corresponding to the arcs of the output tree. The contour tree related contributions are presented chapter 7. We recall here than the merge tree tracking the join of sub-level sets components as introduced in subsection 2.3.3 is named *join tree* and the one tracking sur-level sets components is named *split tree*. Additionally, the term *arc* is used to describe a 1-simplex belonging to the output tree while the term *edge* is preferred to describe a 1-simplex belonging to the input mesh.

This chapter presents the following contributions:

1. **A new local algorithm based on Fibonacci heap:** We present a new algorithm for the computation of augmented merge trees. Contrary to massively parallel approaches [3, 17, 52], our strategy revisits the optimal sequential algorithm for augmented trees [16]. A major distinction with the latter algorithm is the localized nature of our approach, based on local sorting traversals whose results are progressively merged with the help of a Fibonacci heap. In this context, we also introduce a new criterion for the detection of the saddles which generate branching in the output tree, as well as an efficient procedure to process the output arcs in the vicinity of the root of the tree (hereafter referred to as the *trunk*). Our algorithm is simple to implement and it improves practical time performances over a reference implementation [23] of the traditional algorithm [16].

2. **Parallel augmented merge trees:** We show how to leverage the task runtime environment of OpenMP [59] to easily implement a shared-memory, coarse-grained parallel version of the above algorithm for multi-core architectures. Instead of introducing extra work with a static decomposition of the mesh among the threads (as in chapter 5), the local algorithm based on Fibonacci heaps naturally distributes the merge tree arc computations via independent tasks on the CPU cores. We hence avoid any extra work in parallel, while enabling an efficient dynamic load balancing on the CPU cores thanks to the task runtime. This results in superior time and scaling performances compared to previous multi-threaded algorithms for augmented merge trees [38].

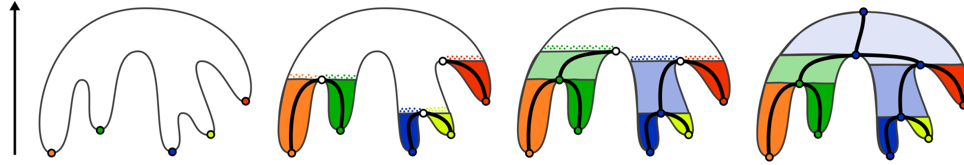


Figure 6.1 – *Overview of our augmented merge tree algorithm based Fibonacci heaps (2D toy elevation example). First, the local extrema of f (corresponding to the leaves of the join tree $\mathcal{T}^-(f)$) are extracted (left, subsection 6.2.1). Second, the arc σ_m of each extremum m is grown independently along with its segmentation (matching colors, center left, subsection 6.2.2). These independent growths are achieved by progressively growing the connected components of level sets created in m, for increasing f values, and by maintaining at each step a priority queue $\mathcal{Q}_m$, implemented with a Fibonacci heap, which stores vertex candidates for the next iteration (illustrated with colored dots). These growths are stopped at merge saddles (white disks, center left, subsection 6.2.3). Only the last growth reaching a saddle s is kept active and allowed to continue to grow the saddle's arc σ_s (matching colors, center right, subsection 6.2.4). The constant time merge operation of the Fibonacci heap (to initialize the growth at s) enables a highly efficient execution for this step in practice. Last, when only one growth remains active, the tree is completed by simply creating its* trunk, *a monotone sequence of arcs to the root of the tree which links the remaining pending saddles (pale blue region, right, subsection 6.2.5). The task-based parallel model allows for a straightforward parallelization of this algorithm, where each arc is grown independently, only requiring local synchronizations on merge saddles.*

6.1 Overview

An overview of our augmented merge tree computation algorithm is presented in Figure 6.1 in the case of the join tree $\mathcal{T}^-(f)$. The purpose of our algorithm, in addition to construct $\mathcal{T}^-(f)$, is to build the explicit segmentation map ϕ, which maps each vertex $v \in \mathcal{M}$ to $\mathcal{T}^-(f)$. Our algorithm is expressed as a sequence of procedures, called on each vertex of $\mathcal{M}$. First, given a vertex v, the algorithm checks if v corresponds to a leaf (Figure 6.1 left, subsection 6.2.1). If this is the case, the second procedure is triggered. For each leaf vertex, the augmented arc connected to it is constructed by a local growth, implemented with a sorted breadth-first search traversal (Figure 6.1 middle left, subsection 6.2.2). A local growth may continue at a join saddle s, in a third procedure, only if it is the last growth which visited the saddle s (Figure 6.1 middle right, subsection 6.2.4). To initiate the growth from s efficiently, we rely on the Fibonacci heap data-structure (described in subsection 2.4.3) in our breadth-first search traversal, which supports constant-time merges of sets of visit candidates. A fourth procedure (the trunk growth) is triggered to abbreviate the process when a local growth happens to be the last active

growth. In this case, all the unvisited vertices above s are guaranteed to map through ϕ to a monotone super-arc-path (a path composed of super arcs) from s to the root (Figure 6.1 right, subsection 6.2.5). Overall, the time complexity of our algorithm is identical to that of the reference algorithm by Carr et al. [16]: $O(|\sigma_0| \log(|\sigma_0|) + |\sigma_1|\alpha(|\sigma_1|))$, where $|\sigma_i|$ stands for the number of i-simplices in $\mathcal{M}$ and $\alpha()$ is the inverse of the Ackermann function (cf. subsubsection 2.4.2.2).

6.2 Local propagations for merge tree computations

In this section, we present our algorithm for the computation of augmented merge trees based on local arc growth. Our algorithm consists in a sequence of procedures applied to each vertex, described in each of the following subsections. In the remainder, we illustrate our discussion with the join tree $\mathcal{T}^{-}(f)$, which tracks connected components of sub-level sets, initiated in local minima.

6.2.1 Leaf search

Algorithm 7 Find minima of the input mesh

```
procedure LeafSearch(Mesh: M)
    for each vertex v ∈ M do                    ▷ in parallel (tasks)
        add v to leaves if |Lk_0^-(v)| = 0
    end for
    return leaves
end procedure
```

The procedure **LeafSearch**, used to find the minima on which local growths will later be initiated, is shown in Algorithm 7. Minima are vertices with an empty lower link: $|Lk_0^{-}(v)| = 0$.

6.2.2 Leaf growth

For each local minimum m, the leaf arc σ_m of the join tree connected to it is constructed with a procedure that we call **ArcGrowth**, presented in Algorithm 8. The purpose of this procedure is to progressively sweep all contiguous equivalence classes (section 2.3) from m to the saddle s located at the extremity of σ_m. We describe how to detect such a saddle s, and therefore where to stop such a growth, in the next subsection

Algorithm 8 Local growth computing one arc of $\mathcal{T}^{-}(f)$

```
procedure ArcGrowth(Q_m: Fibonacci heap, uf: Union-Find)
    Open a new arc in T^-(f) at the first vertex of Q_m
    while not the last active growth do
        Pop the first vertex of Q_m in v
        Process v
        Add Lk_0^+(v) into the Q_m
        Use Lk_0^-(v) to check if v is a merging saddle
        if v is a merging saddle then
            if last growth reaching v then
                SaddleGrowth(v)
            end if
            return
        end if
    end while
end procedure
```

(subsection 6.2.3). In other words, this growth procedure will construct the connected component of sub-level set initiated in m, and will make it progressively grow for increasing values of f.

This is achieved by implementing an ordered breadth-first search traversal of the vertices of $\mathcal{M}$ initiated in m. At each step, the neighbors of v which have not already been visited are added to a priority queue $\mathcal{Q}_m$ (if not already present in it), implemented as a Fibonacci heap [21, 33]. Additionally, v is *processed* by the current growth: the vertex is marked with the identifier of the current arc σ_m for future addition. The purpose of the addition of v to σ_m is to augment this arc with regular vertices, and therefore to store its data segmentation. Next, the following visited vertex v' is chosen as the minimizer of f in $\mathcal{Q}_m$ and the process is iterated until s is reached (subsection 6.2.3). At each step of this local growth, since breadth-first search traversals grow connected components, we have the guarantee, when visiting a vertex v, that the set of vertices visited up to this point (added to σ_m) indeed equals to the set of vertices belonging to the connected component of sub-level set of $f(v)$ which contains v, noted $f^{-1}_{-\infty}\big(f(v)\big)_v$ in section 2.2. Therefore, our local leaf growth indeed constructs σ_m (with its segmentation). Also, note that, at each iteration, the set of edges linking the vertices already visited and the vertices currently in the priority queue $\mathcal{Q}_m$ are all crossed by the level set $f^{-1}\big(f(v)\big)$.

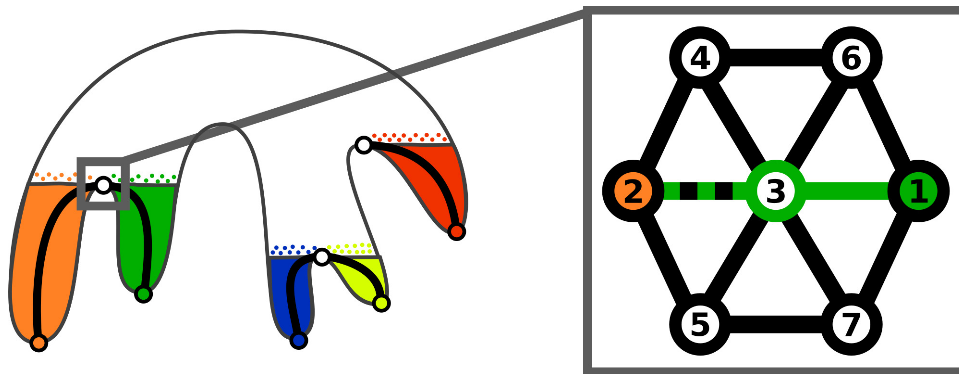


Figure 6.2 – *Local merge saddle detection based on arc growth (2D elevation example from Figure 6.1). The local growth of the arc σ_m (green) will visit the vertex v' at value 3 after visiting the vertex at value 1 (following the priority queue $\mathcal{Q}_m$). At this point, the neighbors of v' which have not been visited yet by σ_m and which are not in $\mathcal{Q}_m$ yet (dashed green edges) will be added to $\mathcal{Q}_m$. The minimizer v of $\mathcal{Q}_m$ (vertex 2) has a scalar value lower than v'. Hence v' is a merge saddle.*

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

6.2.3 Saddle stopping condition

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

In principle, the saddles of f could be extracted by using a critical point extraction procedure based on a local classification of the link of each vertex, as presented in subsection 2.2.1. However, such a strategy has two disadvantages. First not all saddles of f necessarily corresponding to branching in $\mathcal{T}^-(f)$ and/or $\mathcal{T}^+(f)$, thus some unnecessary computation would need to be carried out. Second, we found in practice that even optimized implementations of such a classification [88] tend to be slower than the entire augmented merge tree computation in sequential. Hence, another strategy should be considered for the sake of performance.

The local **ArcGrowth** procedure (subsection 6.2.2) visits the vertices of $\mathcal{M}$ with a breadth-first search traversal initiated in m, for increasing f values. At each step, the minimizer v of $\mathcal{Q}_m$ is selected. Assume that at some point: $f(v) < f(v')$ where v' was the vertex visited immediately before v. This implies that v belongs to the lower link of v', $Lk^-(v')$. Since v was visited after v', this means that v does not project to σ_m through ϕ. In other words, this implies that v does not belong to the

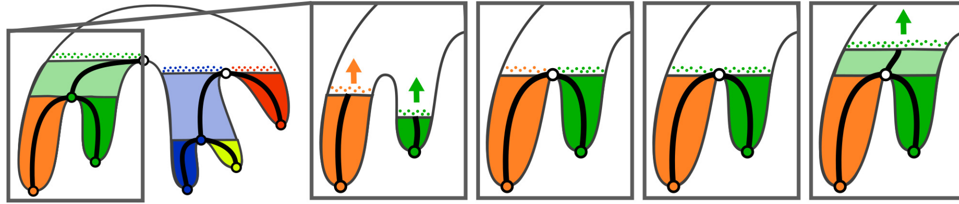


Figure 6.3 – *Union of priority queues at a merge saddle (2D elevation example from Figure 6.1). Initially, each arc growth maintains its own priority queue (illustrated with colored dots, left inset). When reaching a merge saddle s (second inset), the growths which arrived first in s are marked* terminated*. Only the last one (green) will be allowed to resume the growth from s to construct the arc σ_s (last inset). To continue the propagation of the sub-level set component which contains s, the priority queues of all growths arrived at s need to be merged into only one (third inset) prior to resuming the propagation. If done naively, this operation could yield a quadratic runtime complexity for our approach overall. Since Fibonacci heaps support constant time merges, they guarantee the linearithmic complexity of our overall approach.*

connected component of sub-level set containing m. Therefore, v' happens to be the saddle s that correspond to the extremity of σ_m. Locally (Figure 6.2), the local leaf growth entered the star of v' through the connected component of lower link projecting to σ_m and jumped across the saddle v' downwards when selecting the vertex v, which belongs to another connected component of lower link of v'.

Therefore, a sufficient condition to stop an arc growth is when the candidate vertex returned by the priority queue has a lower f value than the vertex visited last. In such a case, the last visited vertex is the saddle s which closes the arc σ_m (Figure 6.2).

6.2.4 Saddle growth

Algorithm 9 Start a local growth at a join saddle

procedure SaddleGrowth(s: join saddle)

Close arcs in $Lk^-(s)$

$\mathcal{Q}_m \leftarrow$ union $\mathcal{Q}_{m_0}, \mathcal{Q}_{m_1}, \ldots \mathcal{Q}_{m_n} \in Lk_0^-(s)$

uf $\leftarrow$ union uf$_0$, uf$_1$,... uf$_n$ $\in Lk_0^-(s)$

ArcGrowth($\mathcal{Q}_m$, uf)

end procedure

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

Given a local minimum m, its leaf growth is stopped at the saddle s which corresponds to the extremity of the arc connected to it, σ_m. When reaching s, if *all* vertices of $Lk^-(s)$ have already been visited by some local leaf growth, we say that the current growth, initiated in m, is the *last* one visiting s. In such a case, the procedure **SaddleGrowth** presented in subsection 6.2.4 is called (see Algorithm 8) and the same breadth-first search traversal can be applied to grow the arc of $\mathcal{T}^-(f)$ initiated in s, noted σ_s. However, in order to represent all the connected components of sub-level set merging in s, such a traversal needs to be initiated with the *union* of the priority queues $\mathcal{Q}_{m_0}, \mathcal{Q}_{m_1}, \ldots \mathcal{Q}_{m_n}$ of *all* the arcs merging in s. Such a union models the entire set of candidate vertices for absorption in the sub-level component of s (Figure 6.3). Since both the number of minima of f and the size of each priority queue can be linear with the number of vertices in $\mathcal{M}$, if done naively, the union of all priority queues could require $O(|\sigma_0|^2)$ operations overall. To address this issue, we model each priority queue with a Fibonacci heap (described in [21, 33]), which supports the removal of the minimizer of f from $\mathcal{Q}_m$ in $\log(|\sigma_0|)$ steps, and performs both the insertion of a new vertex and the merge of two queues in constant time.

Similarly to the traditional merge tree algorithm [16, 80], we maintain a Union-Find data structure [21] (introduced subsubsection 2.4.2.2) to precisely keep track of the arcs which need to be merged at a given saddle s. Each local minimum m is associated with a unique Union-Find element, which is also associated to all regular vertices mapped to σ_m (subsection 6.2.2). Also, each Union-Find element is associated to the arc it currently grows. When an arc σ reaches a join saddle s last, the find operation of the Union-Find is called on each vertex of $Lk^-(s)$ to retrieve the set of arcs which merge there and the union operation is called on all Union-Find associated to these arcs to keep track of the merge event. Thus, overall, the time complexity of our augmented merge tree computation is $O\big(|\sigma_0| \log(|\sigma_0|) + |\sigma_1|\alpha(|\sigma_1|)\big)$, where $\alpha()$ is an extremely slow-growing function (inverse of the Ackermann function). The $|\sigma_1|\alpha(|\sigma_1|)$ term yields from the usage of the Union-Find data structure, while the Fibonacci heap, thanks to its constant time merge support, enables to grow the arcs of the tree in logarithmic time. The time complexity of our algorithm is then exactly equivalent to the traditional algorithm [16, 80]. However, comparisons to a reference implementation by Dillard [23] (section 6.4) show that our approach provides superior performance in practice.

6.2.5 Trunk growth

Algorithm 10 Compute the last monotone super-arc-path

procedure TRUNK
  Close arcs on pending saddles
  Create a monotone super-arc-path from the last visited vertex to the global maximum
  for each unvisited vertex v_u **do** ▷ in parallel (tasks)
    Project v_u into its arc on the monotone super-arc-path
  end for
end procedure

Time performance can be further improved by abbreviating the process when only one arc growth is remaining. Initially, if f admits N local minima, N arcs (and N arc growths) need to be created. When the growth of an arc σ reaches a saddle s, if σ is not the last arc reaching s, the growth of σ is switched to the *terminated* state. Thus, the number of remaining arc growths will decrease from N to 1 along the execution of the algorithm. In particular, the last arc growth will visit all the remaining, unvisited, vertices of $\mathcal{M}$ upwards until the global maximum of f is reached, possibly reaching on the way an arbitrary number of *pending* join saddles, where other arc growths have been stopped and marked terminated (white disks, Figure 6.1, third column). Thus, when an arc growth reaches a saddle s, if it is the last active one, we have the guarantee that it will construct in the remaining steps of the algorithm a sequence of arcs which constitutes a monotone super-arc-path from s up to the root of $\mathcal{T}^-(f)$. We call this sequence the *trunk* of $\mathcal{T}^-(f)$ (Figure 6.1) and we present the corresponding procedure in Algorithm 10. The trunk of the join tree can be computed faster than through the breadth-first search traversals described in subsection 6.2.2 and subsection 6.2.4. Let s be the join saddle where the trunk starts. Let $S = \{s_0, s_1, \ldots s_n\}$ be the sorted set of join saddles that are still pending in the computation (which still have unvisited vertices in their lower link). The trunk is constructed by simply creating arcs that connect two consecutive entries in S. Next, these arcs are augmented by simply traversing the vertices of $\mathcal{M}$ with higher scalar value than $f(s)$ and projecting each unvisited vertex v_u to the trunk arc that spans it scalar value $f(v_u)$.

Thus, our algorithm for the construction of the trunk does not use any breadth-first search traversal, as it does not depend on any mesh

traversal operation, and it is performed in $O(|\sigma_0| \log(|\sigma_0|))$ steps (to maintain regular vertices sorted along the arcs of the trunk). To the best of our knowledge, this algorithmic step is another important novelty of our approach.

Finally, the overall merge tree computation is presented in Algorithm 11.

Algorithm 11 Overall merge tree computation for a mesh $\mathcal{M}$

```
leaves ← LeafSearch(M)
for each v ∈ leaves do
    Q_m ← new Fibonacci heap containing v
    uf  ← new Union-Find
    ArcGrowth(Q_m, uf)                              ▷ task
end for
Trunk()
```

6.2.6 Segmentation

Our output tree is based on a super arc representation (introduced subsection 2.4.1), each arc having a list of regular vertices. This representation allows to compute efficiently the segmentation in a 2-pass manner. Vertices are marked with the identifier of the arc they correspond to during the merge tree construction, then the lists of regular vertices of each arc are pre-allocated and filled in parallel. This way, memory is allocated once. Maintaining the list of vertex on the fly would either leads to scattered memory access for vertex retrieval or re-allocation during the computation as the number of regular vertex on each arc cannot be foreseen.

6.3 Task-based parallel merge trees

The previous section introduced a new algorithm based on local arc growths with Fibonacci heaps for the construction of augmented join trees (split trees being constructed with a symmetric procedure). Note that this algorithm enables to process the minima of f concurrently. The same remark goes for the join saddles; however, a join saddle growth can only be started after all of its lower link vertices have been visited. Such an independence and synchronization among the numerous arc growths can be straightforwardly parallelized thanks to

the task parallel programming paradigm. Also, note that such a split of the work load does not introduce any supplementary computation or memory overhead. Task-based runtime environments also naturally support dynamic load balancing, each available thread picking its next task among the unprocessed ones. We rely here on OpenMP tasks [59], but other task runtimes (e.g. Intel Threading Building [65] Blocks, Intel Cilk Plus [1], etc.) could be used as well with a few modifications. In practice, users only need to specify a number of threads among which the tasks will be scheduled. In the remainder, we will present our taskification process for the merge tree computation, as well as the required task synchronizations.

At a technical level, our implementation starts with a global sort of all the vertices according to their scalar value in parallel (using the GNU parallel sort [35]). This reduces further vertex comparisons to comparisons of indices, which is faster in practice than accessing the actual scalar values and which is also independent of the scalar data representation. Our experiments have shown that this sort benefits from a better data locality, and is thus more efficient, when using an array of structures (AoS) rather than a structure of arrays (SoA) for the vertex data structures (id, scalar value, offset).

6.3.1 Taskification

Parallel leaf search: For each vertex $v \in \mathcal{M}$, the extraction of its lower link $Lk^-(v)$ is a local operation. This makes this step embarrassingly parallel and enables a straightforward parallelization of the corresponding loop using OpenMP [59] tasks: see Algorithm 7. Once done, we have the list of extrema from which the leaf growth should be started. This list is sorted so that the leaf growths are launched in the order of the scalar value of their extremum, starting with the "deepest" leaves. With minor changes, it is also possible to launch the growth on the fly during the leaf search, but we found in practice that the scheduling induced by the "deepest" leaf first strategy gives better performance results.

Arc growth tasks: Each arc is independent from the others, spreading locally until it finds a saddle. Each leaf growth is thus simply implemented as a task, starting at its previously extracted leaf as shown in Algorithm 11. All tasks but the last one stop at the next saddle: this last task then proceeds with this saddle growth.

6.3.2 Synchronization

In the following, we present the task synchronizations required for a parallel execution of our algorithm.

Saddle stopping condition: The saddle stopping condition presented in subsection 6.2.3 can be safely implemented in parallel with tasks. When a vertex v, unvisited so far by the current arc growth, is visited immediately after a vertex v' with $f(v) < f(v')$, then v' is a saddle. To decide if v was indeed not visited by an arc growth associated to the sub-tree of the current arc growth, we use the Union-Find data structure [21] described in subsubsection 2.4.2.2 (one Union-Find node per leaf). In particular, we store for each visited vertex the Union-Find representative of its current growth (which was originally created on a minimum). Our Union-Find implementation supports concurrent *find* operations from parallel arc growths (executed simultaneously by distinct tasks). A *find* operation on a Union-Find currently involved in a *union* operation is also possible but safely handled in parallel in our implementation. Since the *find* and *union* operations are local to each Union-Find sub-tree [21], these operations generate only few concurrent accesses. Moreover, these concurrent accesses are efficiently handled since only atomic operations are involved.

Detection of the last growth reaching a saddle: When a saddle s is detected, we also have to check if the current growth is the last to reach s as described in subsection 6.2.4. For this, we rely on the size of $Lk_0^-(s)$, noted $|Lk_0^-(s)|$ (number of vertices in the lower link of s). This computation being restricted to vertices where it is necessary, we address synchronization issues as follows. Initially, a *lower link counter* associated with s is set to -1. Each task t reaching s will atomically decrement this counter by n_t, the number of vertices in $Lk^-(s)$ visited by t. Using here an OpenMP capture atomic operation, only the first task reaching s will retrieve -1 as the initial value of s (before the decrement). This first task will then compute $|Lk_0^-(s)|$ and will (atomically) increment the counter by $|Lk_0^-(s)| + 1$. Since the sum over n_t for all tasks reaching s equals $|Lk_0^-(s)|$, the task eventually setting the counter to 0 will be considered as the "last" one reaching s (note that it can also be the one which computed $|Lk_0^-(s)|$). We thus rely here only on lightweight synchronizations, and avoid using a critical section.

Growth merging at a saddle: Once the lower link of a saddle has been completely visited, the "last" task which reached it merges the priority queues (implemented as Fibonacci heaps), and the corresponding Union-Find data structures, of all tasks *terminated* at this saddle. Such an operation is performed sequentially at each saddle, without any concurrency issue both for the merge of the Fibonacci heaps and for the *union* operations on the Union-Find. The saddle growth starting from this saddle is performed by this last task, with no new task creation. This continuation of tasks is illustrated with shades of the same color in Figure 6.1 (in particular for the green and blue tasks). As the number of tasks can only decrease, the detection of the trunk start is straightforward. Each time a task terminates at a saddle, it decrements atomically an integer counter, which tracks the number of remaining tasks. The trunk starts when this number reaches one.

Early trunk detection: In parallel, an early trunk detection procedure can be considered in order for the last active task to realize earlier, before reaching its upward saddle, that it is indeed the last active task and therefore to trigger the efficient (and parallel) trunk processing procedure even earlier. This detection consists in regularly checking, within each local growth, if a task is the last active one or not. In practice, we check the number of remaining tasks every $10,000$ vertices on our experimental setup to avoid slowing down significantly the computation. This improvement is particularly beneficial on data sets composed of large arcs. In this case, a significant section of the arc that would have been processed by only one active task is efficiently processed in parallel during the trunk growth procedure.

6.3.3 Parallel trunk growth

During the arc growth step, we keep track of the *pending* saddles (saddles reached by some tasks but for which the lower link has not been completely visited yet). The list of pending saddles enables us to compute the trunk in parallel as described in Algorithm 10. Once the trunk growth has started, we only focus on the vertices whose scalar value is strictly greater than the lowest starting node of arcs ending at the lowest pending saddle, as all other vertices have already been processed during the regular arc growth procedure. Next, we create the sequence of arcs connecting pairs of pending saddles in ascending order. At this point, each vertex can

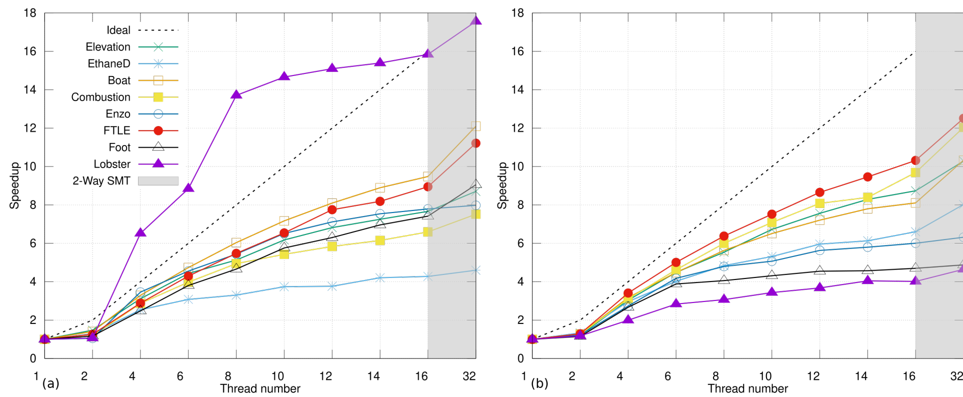


Figure 6.4 – *FTM scalability on our* 512^3 *regular grid data sets for (a) the join tree and (b) the split tree computation. The gray area represents the usage of two threads per core with SMT (simultaneous multithreading).*

be projected independently of the others along one of these arcs. Using the sorted nature of the list of pending saddles, we can use dichotomy for a fast projection. Moreover when we process vertices in the sorted order of their index, a vertex can use the arc of the previous one as a lower bound for its own projection: we just have to check if the current vertex still projects in this arc or in an arc with a higher scalar value. We parallelize this vertex projection procedure using tasks: each task processes chunks of contiguous vertex indices out of the globally sorted vertex list (see e.g. the OpenMP taskloop construct [59]). For each chunk, the first vertex is projected on the corresponding arc of the trunk using dichotomy. Each new vertex processed next relies on its predecessor for its own projection. Note that this procedure can visit (and ignore) vertices already processed by the arc growth step.

6.4 Results

In this section we present performance results obtained on a workstation with two Intel Xeon E5–2630 v3 CPUs (2.4 GHz, 8 CPU cores and 16 hardware threads each) and 64 GB of RAM. By default, parallel executions will thus rely on 32 threads. These results were performed with our VTK/OpenMP based C++ implementation (available publicly in TTK [88]) using g++ version 6.4.0 and OpenMP 4.5 [59]. This implementation (called *Fibonacci Task-based Merge tree*, or FTM) was built as a TTK module. FTM uses TTK's triangulation data structure which supports both tetrahedral meshes and regular grids by performing an implicit triangulation with

		Sequential	Parallel (32 threads on 16 cores)						
				Leaf	Arc	Trunk		
Data set	$\lvert\mathcal{T}(f)\rvert$	Overall	Sort	search	growth	growth	Overall	Speedup
Elevation	1	11.44	0.84	0.14	0	0.20	1.19	9.57
	1	18.71	0.84	0.65	0	0.20	1.71	10.89
Ethane Diol	17	35.13	1.31	0.28	5.16	0.62	7.38	4.75
	19	30.79	1.31	0.30	2.58	0.62	4.82	6.38
Boat	5,426	29.72	1.24	0.24	0.07	0.64	2.21	13.41
	1,715	29.59	1.24	0.40	0.59	0.63	2.88	10.27
Combustion	26,981	37.20	1.23	0.37	3.04	0.61	5.27	7.04
	23,606	32.38	1.23	0.29	0.53	0.63	2.69	12.03
Enzo	96,061	129.62	1.35	0.36	12.79	0.69	15.20	8.52
	115,287	43.23	1.35	0.36	4.06	0.77	6.55	6.59
Ftle	147,748	31.21	1.28	0.37	0.42	0.70	2.78	11.19
	202,865	35.85	1.28	0.31	0.60	0.70	2.91	12.31
Foot	241,841	25.06	1.04	0.26	0.80	0.55	2.67	9.38
	286,654	48.59	1.06	0.55	7.82	0.53	9.97	4.87
Lobster	472,862	96.34	1.07	0.30	3.59	0.73	5.71	16.86
	490,236	36.64	1.05	0.62	5.45	0.78	7.91	4.62

Table 6.1 – *Running times (in seconds) of the different steps of FTM on a* 512^3 *grid for the* join *and* split trees *(white and gray backgrounds respectively).* $\lvert\mathcal{T}(f)\rvert$ *is the number of arcs in the tree.*

no memory overhead for the latter. For the Fibonacci heap, we used the implementation available in Boost.

Our tests have been performed using eight data sets from various domains. The first one, Elevation, is a synthetic data set where the scalar field corresponds to the z coordinate, with only one connected component of level set: the output is thus composed of only one arc. Five data sets (Ethane Diol, Boat, Combustion, Enzo and Ftle) result from simulations and two (Foot and Lobster) from acquisition, containing large sections of noise. For the sake of comparison, these data sets have been re-sampled, using single floating-point precision, on the same regular grid and have therefore the same number of vertices.

6.4.1 Performance analysis

Table 6.1 details the execution times and speedups of FTM for the join and the split trees on a 512^3 grid. One can first see that the FTM sequential execution time varies greatly between data sets despite their equal input size. This denotes a sensitivity on the output tree, which is common to most merge tree algorithms. Moving to parallel executions the embarrassingly parallel leaf search step offers very good speedups close to 14. The key step for parallel performance is the arc growth. On most of

Data set	Sequential		Parallel	
	Arc growth	Trunk	Arc growth	Trunk
Elevation	0	113,217,189	0	468,537,720
Ethane Diol	472,861	13,862,083	1,003,125	202,175,593
Boat	446,981	13,941,128	933,281	193,274,082
Combustion	453,784	14,104,274	1,416,082	196,810,503
Enzo	344,129	11,170,128	2,514,479	138,666,543
Ftle	594,694	14,007,046	3,198,233	154,453,693
Foot	447,270	27,073,541	2,257,674	182,413,262
Lobster	734,705	19,884,438	2,534,264	135,125,845

Table 6.2 – *Process speed in vert/sec for the arc growth and trunk procedure in sequential and in parallel (join tree, grid:* 512^3*).*

our data sets this step is indeed the most-time consuming in parallel, but its time varies in a large range: this will be investigated in subsection 6.4.2. The last step is the trunk computation, which takes less than one second. Overall, with a minimum speedup of 4.62x, a maximum one of 16.86x and an average speedup of 9.29x on 16-cores, our FTM implementation achieves an average parallel efficiency greater than 58%. These speedups are detailed on the scaling curves of the join and split tree computation in Figure 6.4a and Figure 6.4b. The first thing one can notice is the monotonous growth of all curves. This means that more threads always implies faster computations, which enables us to focus on the 32-thread executions.

Another interesting point is the Lobster data set presenting speedups greater than the ideal one when using four threads and more. This unexpected but welcome supra-linearity is due to the trunk processing of our algorithm. As highlighted in Table 6.2, in sequential mode, the trunk step is indeed able to process vertices much faster than the arc growth step, since no breadth-first search traversal is performed in the trunk step (see subsection 6.2.5). In parallel, the performance gap is even larger thanks to the better parallel speedups obtained in the trunk step than in the arc growth step. The trunk processing step is 30x faster than the arc growth in sequential execution, and 110x faster in parallel. The arc growth is indeed 3x faster in parallel than in sequential while the trunk is 10x faster in parallel than in sequential. This enforces the benefits from maximizing the trunk step in our algorithm to achieve both good performances and good speedups. However, for a given data set, the size of the trunk highly depends on the order in which arc growths (leaves and saddles) have been processed. Since the trunk is detected when only one growth remains

Data set	Min	Max	Range	Average	Std. dev
Elevation	1.17	1.19	0.02	1.18	0.01
Ethane Diol	7.37	8.67	1.29	8.00	0.42
Boat	2.11	2.21	0.09	2.14	0.02
Combustion	4.61	5.27	0.65	4.89	0.17
Enzo	14.44	15.82	1.38	15.29	0.53
Ftle	2.75	2.82	0.07	2.78	0.02
Foot	2.63	2.70	0.07	2.67	0.02
Lobster	5.36	5.71	0.34	5.53	0.13

Table 6.3 – *Stability of the execution time of FTM in parallel (join tree, 10 runs,* 512^3 *grid).*

active, distinct orders in leaf and saddle processing will yield distinct trunks of different sizes, for the same data set. Hence maximizing the size of this trunk minimizes the required amount of computation, especially for data sets like Lobster where the trunk encompasses a large proportion of the domain. That is why we launch the leaf growth tasks in the order of the scalar value of their extremum (subsection 6.3.1). Note however, that the arc growth ordering which would maximize the size of the trunk cannot be known in advance. In a sequential execution, it is unlikely that the runtime will schedule the tasks on the single thread so that the last task will be the one that corresponds to the greatest possible trunk. Instead, the runtime will likely process each available arc one at a time, leading to a trunk detection at the vicinity of the root. On the contrary, in parallel, it is more likely that the runtime environment will run out of leaves sooner, hence yielding a larger trunk than in sequential and thus leading to increased (possibly supra-linear) speedups. For example, on the Lobster data set the number of vertices processed by the trunk step is about 70 millions (57% of the data set) during sequential executions while this number grows up to 124 millions (92% of the data set) during parallel executions.

As the dynamic scheduling of the tasks on the CPU cores may vary from one parallel execution to the next, it follows that the trunk size may also vary across different executions, hence possibly impacting noticeably runtime performances. As shown in Table 6.3, the range within which the execution times vary is clearly small compared to the average time and the standard deviation shows a very good stability of our approach in practice.

Finally, in order to better evaluate the FTM performance, we compare our approach to two reference implementations, which are, to the best of

Data set	LT	CF	FTM	LT / FTM	CF / FTM
Elevation	5.81	7.70	1.44	4.01	5.31
Ethane Diol	11.59	17.75	3.61	3.20	4.91
Boat	11.84	17.11	3.06	3.86	5.57
Combustion	11.65	16.87	4.05	2.87	4.15
Enzo	14.33	17.99	13.62	1.05	1.32
Ftle	11.32	15.62	3.55	3.18	4.39
Foot	9.45	12.72	3.20	2.95	3.97
Lobster	11.65	14.80	10.05	1.15	1.47

Table 6.4 – Sequential *join tree computation times (in seconds) and ratios between libtourtre (LT [23 - Dillard]), Contour Forests (CF [38 - Gueunet et al.]) and our Fibonacci Task-based Merge tree (FTM), on a* 256^3 *grid.*

Data set	LT	CF	FTM	LT / FTM	CF / FTM
Elevation	5.00	2.33	0.18	27.19	12.67
Ethane Diol	8.95	4.54	0.85	10.52	5.33
Boat	8.24	4.40	0.29	28.02	14.96
Combustion	7.96	5.82	0.54	14.62	10.69
Enzo	12.18	8.92	1.60	7.60	5.56
Ftle	8.19	4.98	0.54	15.12	9.19
Foot	7.60	6.94	0.86	8.78	8.02
Lobster	8.40	9.02	0.92	9.03	9.70

Table 6.5 – Parallel *join tree computation times (in seconds) and ratios between libtourtre (LT [23 - Dillard]), Contour Forests (CF [38 - Gueunet et al.]), and our Fibonacci Task-based Merge tree (FTM) on a* 256^3 *grid.*

our knowledge, the only public implementations supporting augmented trees:

- *Libtourtre* (LT) [23], an open source sequential reference implementation of the traditional algorithm [16];
- the open source implementation [88] of the parallel Contour Forests (CF) algorithm [38].

In each implementation, TTK's triangulation data structure [88] is used for mesh traversal. Due to its important memory consumption, we were unable to run CF on the 512^3 data sets on our workstation. Thus, we have created a smaller grid (256^3 vertices) with down-sampled versions of the scalar fields used previously. For the first step of this comparison we are interested in the sequential execution. The corresponding results are reported in Table 6.4 We note that in sequential, Contour Forests and Libtourtre implements the same algorithm. Our sequential implementation is about 3.90x faster than Contour Forests and more than 2.70x faster than Libtourtre for most data sets. This is due to the faster processing speed of our trunk step. The parallel results for the merge tree implementation are presented in Table 6.5. The sequential Libtourtre implementation starts by sorting all the vertices, then computes the tree. Using a parallel sort instead of the serial one is straightforward. Thus, we used this naive parallelization of LT in the results reported in Table 6.5 with 32 threads. As for Contour Forests we report the best time obtained on the workstation, which is not necessarily with 32 threads. Indeed, as detailed in chapter 5 increasing the number of threads in CF can result in extra work due to additional redundant computations. This can lead to greater computation times, especially on noisy data sets. The optimal number of threads for CF has thus to be chosen carefully. On the contrary, FTM always benefit from the maximum number of hardware threads. In the end, FTM largely outperforms the other implementations for all data sets: Libtourtre by a factor 15.11x (in average) and Contour Forests by a factor 9.51x (in average). We emphasize that the two main performance bottlenecks of CF in parallel, namely extra work and load imbalance, do not apply to FTM thanks to the arc growth algorithm and to the dynamic task scheduling.

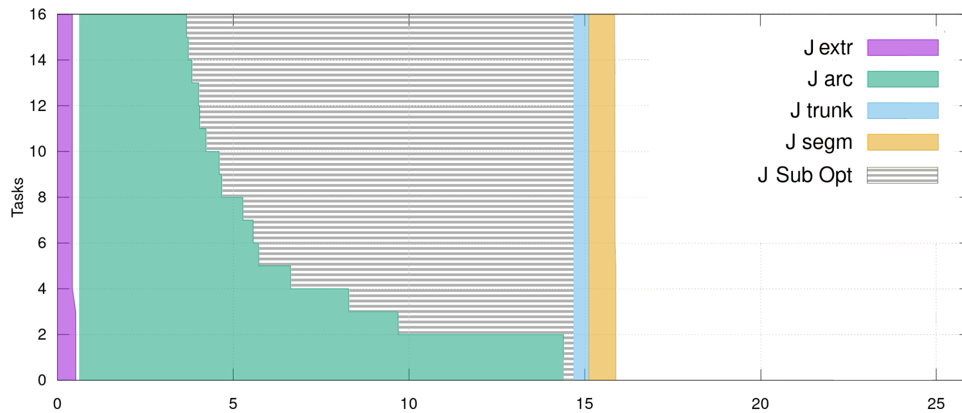


Figure 6.5 – *Number of remaining tasks through time for a parallel execution on the Enzo data set. Each step of the algorithm is shown with a distinct color. The suboptimal section is shown in the area stripped in gray.*

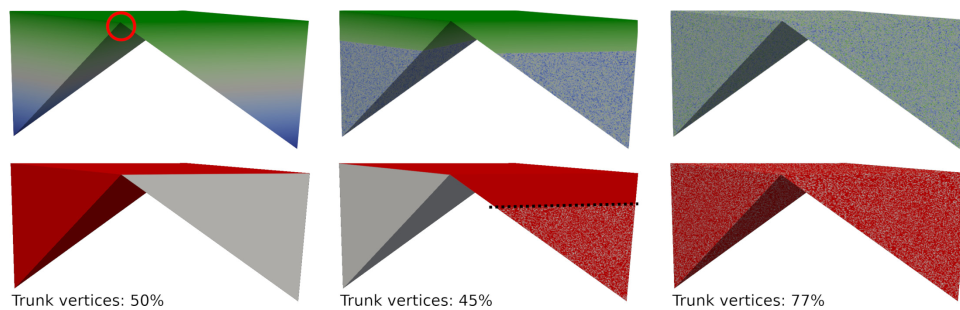


Figure 6.6 – *Worst case data set with the initial scalar field (top left, blue to green), with* 50% *(top middle), and with* 100% *of randomness (top right). The red circle indicates a saddle point induced by the Elevation scalar field, called hereafter "natural saddle". Vertices processed by the trunk procedure are shown in red (bottom).*

6.4.2 Limitations

In order to understand the limitations of our approach, in Figure 6.5 is presented the number of remaining tasks through time, focusing on the part where this number of tasks becomes lower than the number of cores (16). During the arc growth step (shown in green on the figure), the number of active tasks is decreasing as the propagations merge at saddles. When this number becomes lower than 16, the algorithm enters what we call a suboptimal section (stripped area on the figure). During this time, there is less active tasks than available cores, so we do not fully exploit the parallel compute power of our CPUs. This suboptimal section stops when there is only one propagation (task) remaining and the highly parallel trunk procedure is triggered.

The main limitation of this algorithm is the presence of this so-called suboptimal section. In order to further evaluate the impact of this

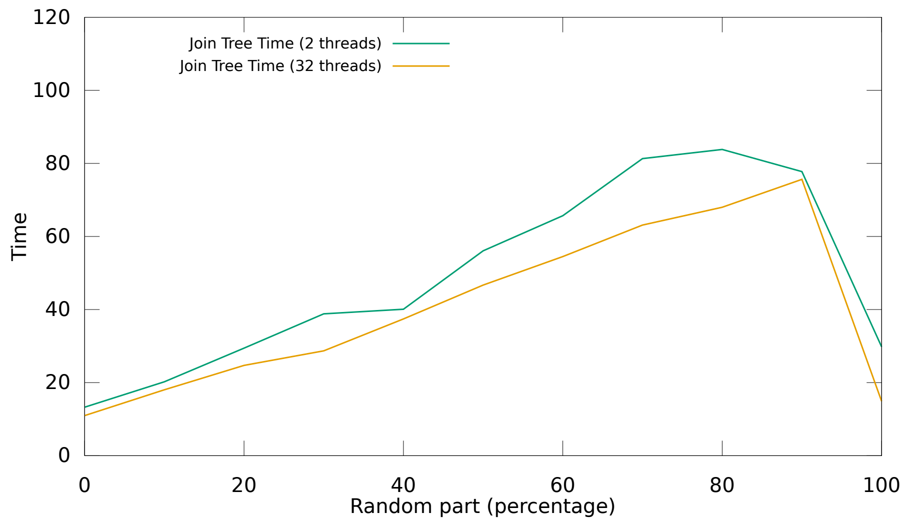


Figure 6.7 – *FTM join tree computation times for* 2 *and* 32 *threads on our worst case data set as the random part progresses from* 0 *to* 100%.

suboptimal section, Figure 6.6 presents a data set used for a worst case analysis. On the initial state (leftmost version), the scalar field is a simple elevation and the join tree is only composed of two large arcs merging in a small root. These two arcs correspond to the two preponderant peaks of the mesh and merge on the natural saddle emphasized with the red circle. Then, we progressively introduce randomness at the bottom of the data set (at the leaves of the tree) and make it grow until it covers the whole mesh.

In Figure 6.7, merge tree computation times are presented for this worst case data set with the randomness ranging from 0 to 100% by step of 10%. During the join tree computation on the initial scalar field (simple elevation), having two or more threads available only have a low impact on the execution times as the predominant arc growth step can only spawn two tasks. As the random progresses, more and more work can be done in parallel using more than two tasks and so the computation with 32 threads becomes shorter than the one with 2 threads. Above 90% of randomness, the random area has reached the natural saddle. At this point, the two peaks have no more impact on the tree topology and the data set becomes similar to a completely random one. Interestingly, such a random data set is no longer the worst case for our algorithm (see the execution time drop at 100%, Figure 6.7), as the set of vertices processed by the efficient trunk procedure remains sufficiently large (Figure 6.6, right).

6.5 Conclusion

We have presented here a new approach to compute augmented merge trees based on local propagations. These propagations correspond to arcs of the tree and can be expressed as parallel tasks, benefiting from the dynamic load balancing of the task runtime for parallel computations. Additionally, this method does not include redundant work in parallel and, contrary to Contour Forests (chapter 5), leads to greater speedups for an increasing number of threads. Results obtained using our implementation show that this algorithm is efficient in sequential, offering 2.70x speedups over a reference implementation thanks to an optimized processing of the last monotone super-arc-path (named *trunk*). In addition to this efficiency, this approach leads to significant speedups, with an average of 9.29x on our 16 cores setup.

7 Output Sensitive Task-based Contour Trees with Fibonacci Heaps

Contents

7.1 Overview . . . 101
7.2 Task-based contour tree computations . . . 102
7.2.1 Leaf search . . . 102
7.2.2 Task overlapping for merge tree computation . . . 102
7.2.3 Merge tree post-processing . . . 103
7.2.4 Parallel combination . . . 104
7.3 Results . . . 106
7.3.1 Performance analysis . . . 107
7.3.2 Limitations . . . 112
7.4 Conclusion . . . 114

An output sensitive approach, extending the strategy with task-based independent local propagations described in chapter 6, is presented for the parallel computation of augmented contour trees. This work is described in a paper accepted to the IEEE TPDS [37] journal (to appear).

This chapter focus on the contour tree computation algorithm. The Fibonacci Task-based Contour Tree algorithm described here will be abbreviated FTC.

In this chapter, we introduce the following contributions:

- **Task overlapping:** Every parallel work for our entire approach has been expressed using tasks and nested parallelism. This complete taskification enables us to overlap tasks arising from the concurrent computations of the join and split trees. In practice this allows the runtime to pick tasks from one of the two trees if the other is running out of work, thus improving the parallel efficiency.
- **Parallel combination of the join and split trees:** We present a new parallel algorithm to combine the join and split trees into the output contour tree. First, we describe a procedure to combine arcs in parallel which exploits nested parallelism. Second, to further speedup this step, we introduce a new original method for the fast parallel processing of the arcs on the trunk of the tree. Detailed performance results concerning this parallel combination are given and analyzed.

7.1 Overview

The reference sequential algorithm to compute the contour tree described subsection 3.2.1 is based on the combination leaf by leaf of two symmetric merge trees. The algorithm presented here uses FTM [36] described in the previous chapter, to compute these two merge trees in parallel. For completeness, we recall here the main steps of the join tree computation with FTM. This method is based on local propagations initiated at the minima and merging together at join saddles until one monotone super-arc-path (a path composed of super arcs) remains. At this point, an efficient trunk procedure processes all remaining vertices independently by projecting them into the monotone super-arc-path.

In the case of the contour tree, the join and the split trees can be computed simultaneously, which enables tasks from both tress to overlap. Once they are both computed, a short post-processing step computing the list of regular vertices for each arc is done. Finally, the two trees are combined together with an algorithm inspired from the reference algorithm [16], processing each block of leaves in parallel until only

one monotone super-arc-path remains. This last monotone super-arc-path is processed by the same trunk procedure than described for FTM subsection 6.2.5.

7.2 Task-based contour tree computations

Algorithm 12 Overall contour tree computation for a mesh $\mathcal{M}$

LeafSearch($\mathcal{M}$)
Compute JT }
Compute ST } ▷ using 2 concurrent tasks
Post-processing of the two merge trees
ArcsCombine()
TrunkCombine()

Our task-based merge tree algorithm (described chapter 6) can be used to compute augmented contour trees efficiently in parallel. First, as shown in Algorithm 12 the **LeafSearch** procedure (detailed subsection 7.2.1) is used to extract all the leaves of both merge trees in a single traversal. Then, these two merge trees are computed concurrently (see subsection 7.2.2) taking advantage of the task-based nature of the FTM algorithm. A post-processing step (described subsection 7.2.3) is required before the new efficient parallel combination algorithm (introduced subsection 7.2.4).

7.2.1 Leaf search

In the FTM algorithm, the merge tree computation starts by extracting the extrema corresponding to the leaves of the tree: minima for the join tree and maxima for the split tree. When computing a contour tree, both can be extracted in a single sweep. This allows to traverse the data set only once, reducing the total amount of data accesses. In terms of implementation, we rely on the task mechanism to perform this leaf search in parallel, giving each task a chunk of $400,000$ vertices to mitigate the cost of creating and managing them.

7.2.2 Task overlapping for merge tree computation

When FTM was presented chapter 6, every parallel step has been expressed using tasks. This also applies to the merge tree post-processing step (see subsection 7.2.3) The task mechanism can be exploited when computing the two merge trees at the same time by overlapping tasks from both

trees. This increases the number of available tasks during the computation and thus improves the parallel efficiency. More precisely, as discussed in subsection 6.4.2, during the arc growth step, the number of active tasks decreases monotonically and is driven by the topology of the tree. When the number of remaining tasks to process becomes smaller than the number of available threads, the computation enters a *suboptimal section*, where the parallel efficiency of our algorithm is undermined as some threads are idle. During the contour tree computation the two merge trees are computed simultaneously and the task overlapping enables us to lower the performance impact of the suboptimal sections. Indeed, when the computation of one of the two merge trees enters a suboptimal section, the runtime can pick tasks from the other tree computation (from its arc growth step, or from subsequent steps). By overlapping the two merge tree computations, we can thus rely on more tasks to exploit at best the available CPU cores.

In order to introduce such task overlap only when required, and thus to benefit from it as long as possible, we also impose a higher priority on all tasks from one of the two trees. We tried another simple heuristic to make the best choice here: the highest priority for the tree with the highest number of leaves. The purpose is to cover the largest suboptimal section with tasks form the other tree, which means having a higher priority on tasks of the tree with this largest suboptimal section. In practice, this heuristic did not give better results over all data sets than just choosing randomly on of the two trees. As the suboptimal section size cannot easily be determined a priori, there is no simple heuristic to make this choice with the limited amount of information we have. Having one tree with a higher priority still helps cover its suboptimal section at best.

7.2.3 Merge tree post-processing

Our merge tree procedure segments $\mathcal{M}$ by marking each vertex with the identifier of the arc it projects to through ϕ. In order to produce such a segmentation for the output contour tree (subsection 7.2.4), each arc of $\mathcal{T}(f)$ needs to be equipped at this point with the explicit sorted list of vertices which project to it. We reconstruct these explicit sorted lists in parallel. For vertices processed by the arc growth step, we save during each arc growth the visit order local to this growth. During the parallel post-processing of all these vertices, we can safely build (with a linear operation count) the ordered list of regular vertices of each arc in parallel thanks to

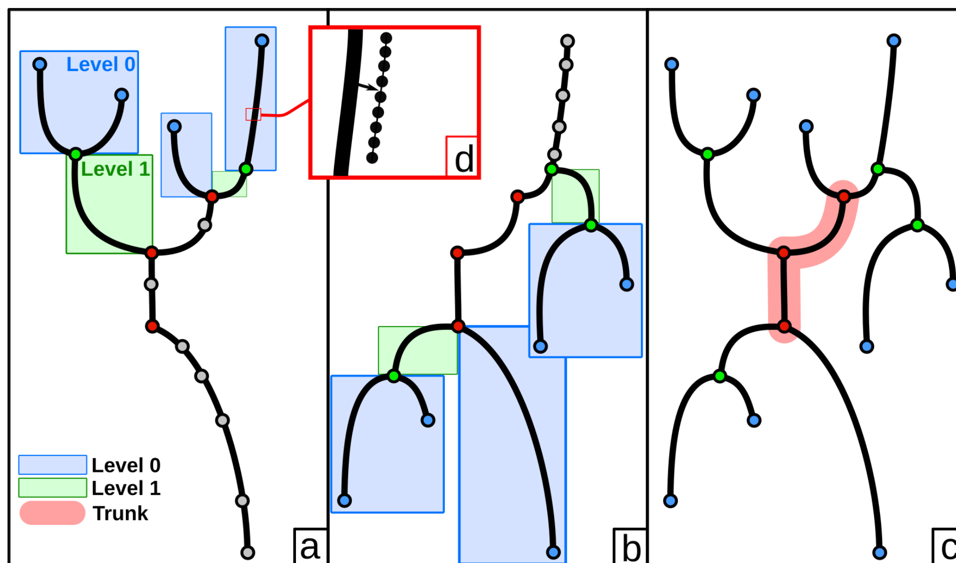


Figure 7.1 – *A join (a) and a split (b) tree augmented with the critical nodes of the final tree. The combination of these two trees results in the final contour tree (c). The notion of* level *(length of the shortest monotone super-arc-path to the closest leaf) is emphasized using the blue and green boxes, corresponding respectively to the levels* 0 *and* 1*. The last monotone super-arc-path can be filled using our highly parallel trunk procedure and is highlighted in red. In (d), we illustrate the list of regular vertices corresponding to the arc segmentation.*

this local ordering. Regarding the vertices processed by the trunk step, we cannot rely on such a local ordering of the arc. Instead each thread concatenates these vertices within bundles (one bundle per arc for each thread). The bundles of a given arc are then sorted according to their first vertex and concatenated in order to obtain the ordered list of regular vertices for this arc. Hence, the $O(n \log n)$ operation count of the sort only applies to the number of bundles, which is much lower than the number of vertices in practice. At this point, to use the combination pass the join tree needs to be augmented with the nodes of the split tree and vice-versa. This step is straightforward since each vertex stores the identifier of the arc it maps to, for both trees. This short step can be done in parallel, using one task for each tree.

7.2.4 Parallel combination

For completeness we sketch here the main steps of the reference algorithm [16] used to combine the join and split trees into the final contour tree. According to this algorithm, the contour tree is created from the two

merge trees by processing their leaves one by one, adding newly created leaves in a queue until it is empty:

1. Add leaf nodes of $\mathcal{T}^-(f)$ and $\mathcal{T}^+(f)$ to a queue $\mathcal{Q}$.
2. Pop the first node of $\mathcal{Q}$ and add its adjacent arc in the final contour tree $\mathcal{C}(f)$ with its segmentation.
3. Remove the processed node from the two trees. If this creates a new leaf node in the original merge tree, add this node into $\mathcal{Q}$
4. If $\mathcal{Q}$ is not empty, repeat from 2.

During phase 2, the arc and its list of regular vertices (shown in Figure 7.1d) are processed. The list of regular vertices is visited and all vertices not already marked are marked with the new arc identifier in the final tree. As a vertex is both in the join and split trees, each vertex will be visited twice. In phase 3, when a node is deleted from a merge tree, three situations may occur. First, if the node has one adjacent arc: remove the node along with this adjacent arc. Second, if the node has one arc up and one down: remove the node to create a new arc which is the concatenation of the two previous ones. Finally in all other situations, the node is not deleted yet: a future deletion will remove it in a future iteration.

We present here a new parallel algorithm to combine the join and the split trees, which improves the reference algorithm [16]. First, we define the notion of *level* of a node in a merge tree as the length of the shortest monotone super-arc-path to its closest leaf. For example, in Figure 7.1 the blue nodes are the leaves and correspond to the level 0, while the green ones at a distance of one arc correspond to the level 1. During the combination, all the nodes and arcs at a common level can be processed in arbitrary order. This corresponds to the **ArcsCombine** procedure in Algorithm 12. We use this for parallelism, by allowing each node (and its corresponding arc) to be processed in parallel. Moreover, processing an arc consist of marking unvisited vertices with an identifier. This can be done in parallel, using tasks, by processing contiguous chunks of regular vertices. In summary, we have two nested levels of task-parallelism available during the arc combination. First we can create tasks to process each arc, then we can create tasks to process regular vertices of an arc in parallel. We use this to create tasks with a large enough computation grain size, and to avoid being constrained by the (possible) low number of arcs to process. In our experimental setup, we choose $10,000$ vertices per task.These two

levels of parallelism are a novelty of our approach, improving both the load balancing and the task computation grain size while also increasing the parallelism degree. However, we note that two synchronizations are required. First, the procedure needs to wait for all nodes of a given level to be processed before going to the following level. Second, data races may occur if the node deletion is not protected in the merge trees as several nodes can be deleted along a same arc simultaneously. A critical section is added around the corresponding deletions. In practice, since most of the time is spent processing arcs and their segmentations, this lock does not represent a performance bottleneck.

Finally, similarly to the merge tree, there is a point where all the remaining work is a monotone super-arc-path tracing, when the contribution of the join and split trees is reduced to one node each. We can interrupt the combination and use the same trunk procedure than described in subsection 6.2.5 for the merge tree to process the remaining nodes, arcs and vertices in parallel. This trunk procedure (corresponding to the **TrunkCombine** procedure in Algorithm 12) will indeed offer a higher parallelism degree at the end of our combination algorithm. This procedure ignores already processed vertices and project the unvisited ones in the arcs of the remaining monotone super-arc-path. Note that the size of this trunk does not depend on the task scheduling (as it is the case for the merge tree), but is fixed by the topology of the join and split trees.

7.3 Results

In this section we present performance results obtained on a workstation with two Intel Xeon E5–2630 v3 CPUs (2.4 GHz, 8 CPU cores and 16 hardware threads each) and 64 GB of RAM. By default, parallel executions will thus rely on 32 threads. For the sake of comparison, this setup is the same than the one used for the FTM algorithm described in the previous chapter. These results were performed with our VTK/OpenMP based C++ implementation (publicly available in TTK [88]) using g++ version 6.4.0 and OpenMP 4.5. This implementation (called *Fibonacci Task-based Contour tree*, or FTC) was built as a TTK module. FTC uses TTK's triangulation data structure which supports both tetrahedral meshes and regular grids by performing an implicit triangulation with no memory overhead for the latter. For the Fibonacci heap, we used the implementation available in Boost.

Data set	$\|\mathcal{T}(f)\|$	Sequential	Parallel (32 threads on 16 cores)					Speedup
		Overall	Sort	Leaf search	MT	Combine	Overall	
Elevation	1	20.92	1.07	0.61	1.08	0	2.77	7.54
Ethane Diol	35	70.63	1.48	0.44	9.29	0.61	11.84	5.96
Boat	7,140	59.33	1.39	0.48	2.55	2.78	7.21	8.22
Combustion	50,586	76.00	1.37	0.49	5.22	1.57	8.66	8.76
Enzo	211,346	215.08	1.47	0.58	15.63	1.99	19.68	10.92
Ftle	350,602	73.42	1.46	0.56	3.32	1.73	7.08	10.36
Foot	528,494	83.44	1.15	0.77	10.06	3.01	14.99	5.56
Lobster	963,068	143.15	1.21	0.89	9.80	6.77	18.68	7.66

Table 7.1 – *Contour tree computation times (in seconds) with FTC on the* 512^3 *grid. Extremum detection is reported under the Leaf Search column. The concurrent computation of the two merge trees is reported under the MT column. The parallel combination of these trees is in the Combine column.*

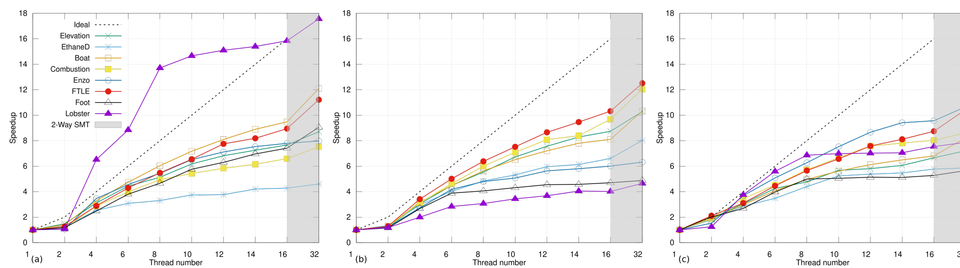


Figure 7.2 – *FTC scalability on our* 512^3 *regular grid data sets for (a) the join tree (from FTM), (b) the split tree (from FTM), (c) the contour tree computation. The gray area represents the usage of two threads per core with SMT (simultaneous multithreading).*

Our tests have been performed using the same eight data sets than we used for FTM (cf. section 6.4).

7.3.1 Performance analysis

Table 7.1 details execution times for our contour tree computation. As for the merge tree, the sequential times vary across data sets due to the output sensitivity of the algorithm. A single leaf search is performed for both merge trees (corresponding to a 25% performance improvement for this step over two separate executions, both in sequential and in parallel). Regarding parallel executions, most of the time is spent computing the join and the split trees as reported under the MT column. We further investigate this step later with Table 7.2 and Figure 7.3. As for the combination, it takes longer to compute for larger trees, with the exception of the Boat data set having a particularly small trunk. This illustrates the output sensitivity of our combination algorithm, as detailed in Table 7.3. Our contour tree computation algorithm results in speedups

Data set	JT then ST	Task overlapping	Overlap speedups
Elevation	2.25	1.73	1.30
Ethane Diol	12.80	10.14	1.26
Boat	3.90	3.11	1.25
Combustion	6.49	5.55	1.17
Enzo	21.34	17.69	1.21
Ftle	4.74	3.86	1.23
Foot	12.14	10.48	1.16
Lobster	14.45	10.81	1.34

Table 7.2 – *Merge tree processing time during the parallel contour tree computation (*512^3 *grid).* JT then ST *reports results obtained by separately computing first the join tree then the split tree, leading to the successive execution of two distinct suboptimal sections. In* Task overlapping*, the two trees are concurrently computed and overlap occurs in their task scheduling.*

varying between 5.56 and 10.92 in our test cases, with an average of 8.12 corresponding to an average parallel efficiency of 50.75%.

The evolution of these speedups as a function of the number of threads is shown in Figure 7.2c. These speedups are consistent with those of the merge tree (recalled Figure 7.2a and Figure 7.2b). Our algorithm benefits from the dynamic task scheduling and its workload does not increase with the number of threads. This also applies to our combination algorithm. Therefore in theory, the more threads are available, the faster FTC should compute the contour tree. In practice, this translates into monotonically growing curves as shown in Figure 7.2. For the contour tree computation, curves shown Figure 7.2c have lower slopes than those of the merge trees (Figure 7.2a and Figure 7.2b). This is mainly due to the combination procedure which has a smaller speedup than our merge tree procedure as detailed below in Table 7.3.

Task overlapping. Table 7.2 presents speedups obtained by computing both trees concurrently, allowing tasks to overlap in their scheduling during the merge tree parallel computation, thanks to the complete taskification of our implementation. This overlap reduces the size of the suboptimal section, as shown in Figure 7.3. This strategy results in speedups up to 1.34x (1.24x in average) compared to a successive computation of the two trees.

Indeed, as mentioned in subsection 7.2.2, during the arc growth computation, the number of remaining tasks becomes smaller than the number of threads. As illustrated Figure 7.3 this leads to a suboptimal

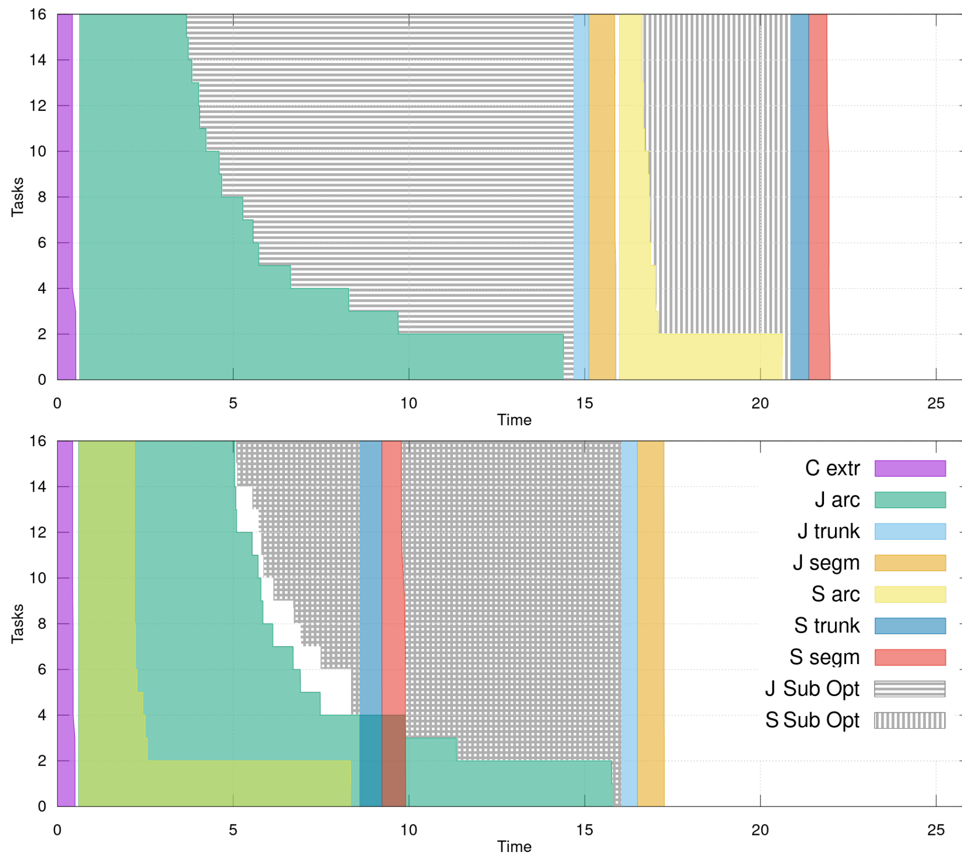


Figure 7.3 – *Number of remaining tasks over time for a parallel execution on the Enzo data set. Each step of the algorithm is shown with a distinct color. The suboptimal sections are shown with areas stripped in gray. At the top, the join and split trees are computed separately (join tree first). At the bottom, they are computed concurrently (hence, at a given time, the number of remaining tasks is the sum of the overlapping curves).*

Data set	Ref	Sequential no trunk	Sequential trunk	Parallel no trunk	Parallel trunk	Seq / parallel FTC with trunk
Elevation	0	0	0	0	0	N.A.
Ethane Diol	3.23	3.82	6.40	2.51	0.54	5.98
Boat	3.11	3.99	3.60	2.63	2.64	1.17
Combustion	3.29	3.63	5.62	3.30	1.49	2.20
Enzo	4.72	4.52	7.03	4.18	1.90	2.48
Ftle	4.79	5.13	7.62	5.01	1.70	2.81
Foot	4.63	4.46	5.15	5.04	3.14	1.47
Lobster	7.11	7.22	7.46	8.33	6.72	1.05

Table 7.3 – *Task-based combination procedure times for sequential and parallel executions with and without the trunk processing, compared to the sequential combination procedure* (Ref, *without tasks), on a* 512^3 *grid. The 0 values for the Elevation data-set are due to the filiform nature of its merge trees (which implies instantaneous combintations).*

section, where some available threads are left idle. On this chart, the suboptimal section is shown using the stripped gray area. If the join and split trees are computed one after the other, (Figure 7.3, top chart) we observe two distinct suboptimal sections: one for the join tree and one for the split tree. In contrast, when the join and split trees are computed simultaneously (Figure 7.3, bottom chart) the OpenMP runtime can pick tasks among either trees, hence reducing the area of the stripped section. Moreover, at the bottom chart of Figure 7.3, when the arc growth procedure of the split tree finishes, that of the join tree is still processing. The remaining steps of the split tree computation (trunk processing and regular vertex segmentation) continue in the meantime, which contributes to reducing the suboptimal section (blue and red columns in Figure 7.3). At the end, this task overlapping strategy results in a smaller stripped area and so in an improved parallel efficiency. In the same manner, the total time of the leaf search plus merge tree computation reaches 21.34 seconds when merge trees are computed one after the other and 17.69 seconds in an overlapped merge trees execution (cf. Table 7.2).

Parallel Combination. For the combination step, we report in Table 7.3 comparisons between various versions of our task-based combination and the reference sequential combination algorithm (without tasks, cf. subsection 7.2.4) Note that our parallel algorithm executed sequentially, without triggering the fast trunk procedure, lead to execution times similar to those of the reference sequential algorithm [16]. According to this table, enabling the trunk on a sequential execution of our new algorithm is slower by 33% in average. We believe this is due to two reasons. First,

Data set	LT	CF	FTC	LT / FTC	CF / FTC
Elevation	10.84	8.15	2.82	3.83	2.88
Ethane Diol	21.54	17.73	6.61	3.25	2.67
Boat	21.10	16.63	5.68	3.71	2.92
Combustion	21.52	16.92	7.38	2.91	2.29
Enzo	27.79	19.71	19.33	1.43	1.01
Ftle	23.05	15.89	7.33	3.14	2.16
Foot	19.24	13.41	9.77	1.96	1.37
Lobster	23.39	51.32	17.04	1.37	3.01

Table 7.4 – Sequential *contour tree computation times (in seconds) and ratios between libtourtre (LT [23]), Contour Forests (CF [38]) and our current Fibonacci Task-based Contour tree FTC, on a* 256^3 *grid.*

each regular vertex additionally checks if it should be added to the current arc (subsection 7.2.4). Second, the trunk procedure may re-visit some vertices already visited by the arc combination procedure, which results in redundant visits (subsection 7.2.4). In our test cases, this redundant work affects less than 1% of the total number of vertices. In contrast, enabling the trunk procedure in a parallel execution is necessary to achieve significant speedups, by an average factor of 1.98x in Table 7.3, with respect to the sequential reference algorithm implemented in FTM. Indeed, in the parallel combination algorithm the number of arcs at each level decreases, inducing a decreasing trend in the number of vertices processed (and tasks created) at each level, and leading to another suboptimal section. The trunk procedure occurs at a point where the arcs combination is likely to use a small number of tasks and replace it by a highly parallel processing, thus improving parallel efficiency. Finally, according to these observations, we choose to trigger the trunk processing only for parallel executions.

Comparison. For the contour tree computation we compare our approach with the two public reference implementations computing the augmented contour tree. Results are shown in Table 7.4. Due to the important memory consumption of Contour Forests [38], we were unable to run these tests on our 512^3 regular grid. Results are reported using a down-sampled 256^3 grid. Our implementation in sequential mode outperforms the two others for every data set. FTC is in average 2.70x faster than libtourtre [23] and 2.29x faster than Contour Forests. In sequential, these two implementations correspond to the reference algorithm [16]. As shown with the merge tree in section 6.4, our algorithm

Data set	LT	CF	FTC	LT / FTC	CF / FTC
Elevation	5.00	2.33	0.40	12.31	5.73
Ethane Diol	8.95	4.54	1.23	7.24	3.67
Boat	8.24	4.40	0.92	8.93	4.77
Combustion	7.96	5.82	1.15	6.86	5.01
Enzo	12.18	8.92	2.87	4.23	3.09
Ftle	8.19	4.98	1.35	6.03	3.66
Foot	7.60	6.94	3.10	2.44	2.23
Lobster	8.40	9.02	4.66	1.80	1.93

Table 7.5 – Parallel *contour tree computation times (in seconds) and ratios between libtourtre (LT [23]), Contour Forests (CF [38]) and our current Fibonacci Task-based Contour trees (FTC), on a* 256^3 *grid.*

is able in practice to process vertices faster thanks to the trunk step, hence the observed improvement.

For the comparison in parallel, results are presented in Table 7.5. For libtourtre, a naive parallelization is achieved by using the GNU parallel sort and by computing the two merge trees concurrently. For contour forests, we present the best time using the optimal number of threads (not necessarily 32). Again, FTC is the fastest for all our test cases. It outperforms libtourtre by an average factor of 6.23x (up to 8.93x for real-life data sets), our naive parallelization of libtourtre having a maximum speedup of 2.81x on 16 cores. FTC is also faster than Contour Forests by a factor 3.76x, taking benefits from the dynamic task scheduling and from the absence of additional work in parallel.

7.3.2 Limitations

As seen subsection 6.4.2, a limitation of the merge tree approach is the presence of the suboptimal section. By launching the tasks of the two merge trees concurrently (allowing them to overlap), the suboptimal section is reduced (see Figure 7.3).

We have also considered using task priorities to maximize the task overlapping, or to minimize the suboptimal sections. We have first studied simple heuristics (based e.g. on the higher number of extrema) to choose which tree will be computed with the high task priority (subsection 7.2.2). However no simple heuristic led to the best choice for all our data sets. We thus arbitrarily assign the high priority to the split tree tasks. Second, we have also considered using task priorities to maximize the number of active tasks at the end of the arc growth step. However this would likely reduce the trunk size, which would lead to lower overall performance results since

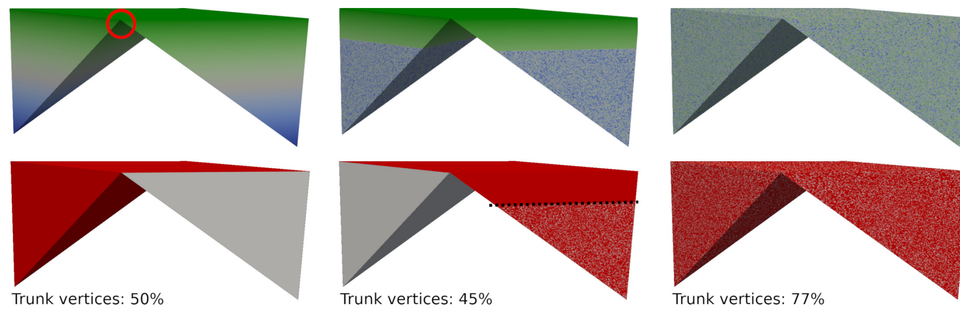


Figure 7.4 – *Worst case data set with the initial scalar field (top left, blue to green), with 50% (top middle), and with 100% of randomness (top right). The red circle indicates a saddle point induced by the Elevation scalar field, called hereafter "natural saddle". Vertices processed by the trunk procedure are shown in red (bottom).*

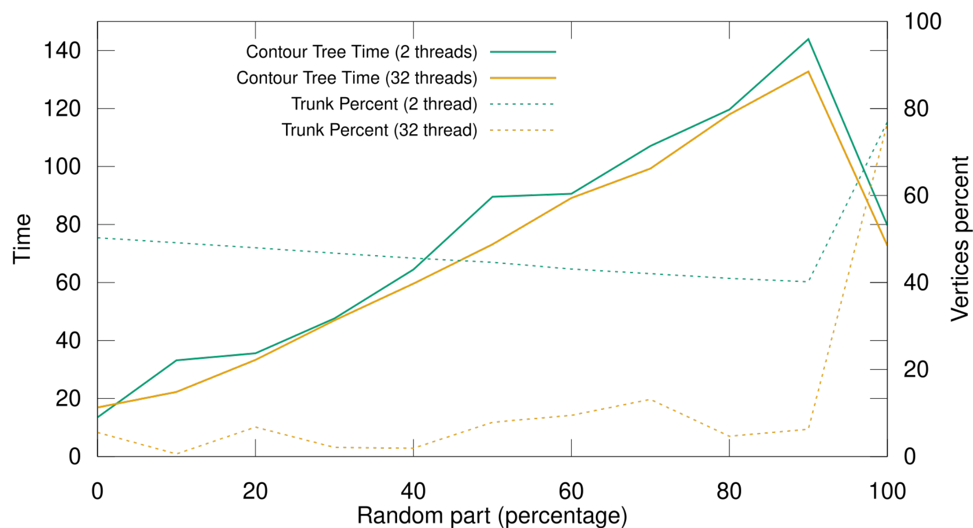


Figure 7.5 – *FTC contour tree computation time for 2 and 32 threads on our worst case data set as the random part progresses form 0 to 100% (plain lines, left axis) and percentage of vertices processed by the trunk procedure (dashed lines, right axis).*

the trunk processing is two orders of magnitude faster than the arc growth one (section 6.4). Finally, we have also tried using distinct task priorities for the successive steps of our algorithm (and still for the two merge trees), but to no avail.

As for FTM (subsection 6.4.2), we have created a worst case data set in order to illustrate the performance impact of these suboptimal sections. This data set is composed of only two large arcs as illustrated on the left of Figure 7.4. As expected, the speedup of the join tree arc growth step on this data set does not exceed 2, even when using 32 threads (results not shown). Then we randomize this worst case data set gradually, starting by the leaf side as illustrated in Figure 7.4 and report the corresponding contour tree computation times with 2 and 32 threads in Figure 7.5. As

the random part progresses (from 0 to 90%) the execution time increases. This is due to the output sensitive nature of contour tree algorithms, but also to the smaller trunk size when the percentage of random vertices increases. Figure 7.4 shows the vertices processed by the trunk procedure (in red, bottom) for different percentages of randomness. Increases in the level of randomness (from left to right) decrease the number of vertices processed by the efficient trunk procedure. When the level of randomness goes beyond the *natural saddle* of the data set (red circle, Figure 7.4), the specifically designed 2-arc worst-case structure disappears and the data set becomes similar to a fully random data set. This translates into better performance results in Figure 7.5, this phenomena already emphasized in FTM (subsection 6.4.2) shows once again that a random data set is not the worst case scenario for our algorithm.

7.4 Conclusion

The approach presented here allows to efficiently compute the augmented contour tree using the FTM algorithm presented chapter 6. This method takes advantage of the task-based nature of FTM, overlapping tasks of the two merge trees and thus reducing the suboptimal section size and improving the parallel efficiencies. A new parallel algorithm for the combination of the two merge trees is also presented, relying on tasks for nested parallelism and using the highly parallel *trunk* procedure for improved parallelism. The corresponding implementation offers 2.70x speedups over a reference implementation in sequential and results in an average speedup of 8.12x on our 16-core setup. To the best of our knowledge, this is the fastest implementation to compute the augmented merge and contour trees.

8 Output Sensitive Task-based Augmented Reeb Graphs with Dynamic ST-Trees

Contents

8.1 Overview 118
8.2 Local propagations for Reeb graph computations 119
8.2.1 Leaf search 119
8.2.2 Local growth 119
8.2.3 Critical vertex detection 120
8.2.4 Saddle vertex handling 120
8.2.5 Laziness mechanism for preimage graph 121
8.3 Task-based parallel Reeb graphs 122
8.3.1 Leaf search 122
8.3.2 Local growth 122
8.3.3 Saddle vertex handling 123
8.4 Parallel dual sweep 124
8.4.1 Leaf search 124
8.4.2 Local growth 124
8.4.3 Saddle vertex handling 125
8.4.4 Post-processing for merged arcs 126
8.5 Results 126
8.5.1 Performance analysis 127
8.5.2 Comparisons 128
8.5.3 Limitations 131
8.6 Conclusion 131

An output sensitive approach with task-based independent local propagations is presented for the parallel computation of the augmented Reeb graphs. This chapter presents a work in progress, which has not been submitted yet.

In this chapter a new parallel algorithm to compute augmented Reeb graphs is presented. We recall here that the input domain is not required to be simply connected contrary to the case of the contour tree computation. For this reason, Reeb graphs may contain loops (see Figure 8.1) and cannot be computed using the 3-pass method [16]. The algorithm presented here is based on independent local propagations maintaining a dynamic graph data structure corresponding to the connected components of level sets, similarly to the Parsa's algorithm [60] presented subsubsection 3.3.2.2. Results presented here are preliminary results.

This chapter presents the following contributions.

1. **A local algorithm based on Fibonacci heaps:** we present a new algorithm for the computation of augmented Reeb graphs. This approach revisits the sequential sweep algorithm presented by Parsa [60] which offers the best time complexity among Reeb graph algorithms. Our method is based on local sorting traversals, whose results are progressively merged with the help of Fibonacci heaps.

2. **An improved laziness mechanism for ST-Trees updates**: we improve the laziness mechanism presented in [60] by further reducing the number of operations impacting the dynamic graph. We update this graph only locally when a saddle vertex is encountered. 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 algorithm. 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 parallel algorithm using two sweeps to increase the parallelism degree while processing the data set. The first one uses a mesh traversal in increasing order of scalar value while the second one relies on a decreasing order. These sweeps stop when they cross each other.

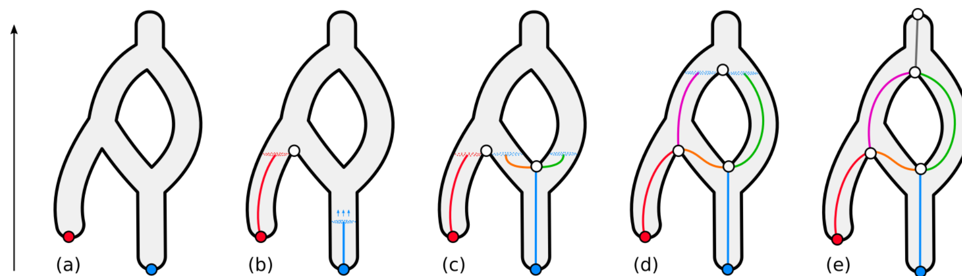


Figure 8.1 – *Overview of our augmented Reeb graph algorithm based on Fibonacci heaps on a 2D toy elevation example. (a) The local minima of f (corresponding to leaves of $\mathcal{R}(f)$) are extracted. (b) The arc σ_m of each minimum is grown independently along with its segmentation. These independents 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 (illustrated with 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 handling two arcs (orange and green) thanks to the dynamic graph implemented with a ST-Tree data structure. (d) When this local growth reaches the left saddle, only the last growth reaching a saddle is kept active. Here, the red one merges in the blue one. (d) The last growth manages two arcs around the topological handle. (e) The augmented Reeb graph of this toy example is complete.*

8.1 Overview

An overview of our augmented Reeb graph computation algorithm is presented Figure 8.1. The purpose of our algorithm, in addition to construct the Reeb graph $\mathcal{R}(f)$, is to build the explicit segmentation map ϕ, which maps each vertex $v \in \mathcal{M}$ to $\mathcal{R}(f)$. Our algorithm is based on the sequential sweep approach of Parsa [60], described subsubsection 3.3.2.2 but uses independent local growths for the mesh traversal. First, given a vertex v, the algorithm checks if v corresponds to a local minimum (Figure 8.1a, subsection 8.2.1). Then, a second procedure is triggered: for each local minimum 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 (Figure 8.1b, subsection 8.2.2). A dynamic graph data structure corresponding to the growing level set components is maintained during the growth. As described in subsubsection 3.3.2.2, this dynamic graph allows to track both join and split saddles and to update the Reeb graph data structure accordingly on the fly (Figure 8.1 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 with a third procedure described in subsection 8.2.4.

8.2 Local propagations for Reeb graph computations

In this section, we present a new algorithm for the computation of augmented Reeb graphs based on local growths. Our algorithm consists in a sequence of procedures applied to each vertex, described in each of the following sub-sections.

8.2.1 Leaf search

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

8.2.2 Local growth

Given a local minimum m, a local growth procedure, named *local growth* starting at m is called. The purpose of this procedure is to progressively sweep all contiguous equivalence classes (section 2.3) from m to 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 by implementing an ordered breadth-first search traversal of the vertices of $\mathcal{M}$ initiated in m. At each step when a vertex v is processed, the neighbors of v which have not already been visited are added to a priority queue $\mathcal{Q}_m$ (implemented as a Fibonacci heap, presented subsection 2.4.3) if not already present in it. Next, the following visited vertex v' is chosen as the minimizer of f in $\mathcal{Q}_m$ and the process is iterated until a join saddle s is reached (subsection 8.2.4). At each step of this local growth, since breadth-first search traversals grow connected components, we have the guarantee, when visiting a vertex v, that all the edges of $\mathcal{M}$ connecting visited vertices to visit candidates (stored in $\mathcal{Q}_m$) are indeed crossed by the connected component, the contour, 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.

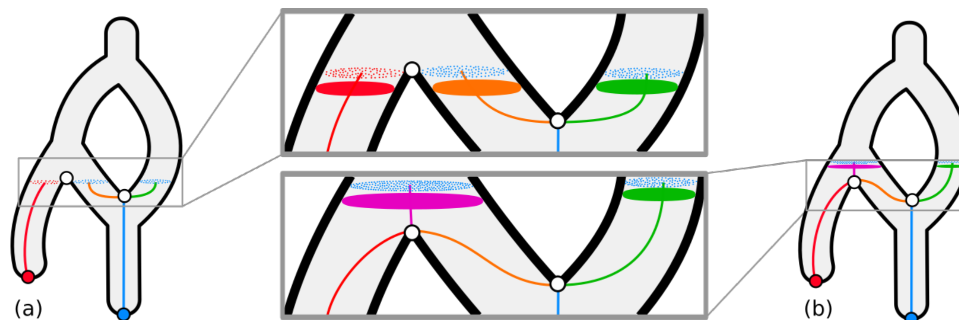


Figure 8.2 – *On a 2D toy elevation example, priority queues (colored dots) and dynamic graphs (plain circles) in the proximity of critical points are highlighted. First, on the left and the top, the right growth has passed a split saddle. The blue priority queue contains candidates vertices of both sides and handles two connected components of the preimage graph shown, below the priority queue, in orange and green with its corresponding arcs. Second, on the right and bottom, the left join saddle has been processed. The red and blue priority queues have merged 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.*

During the sweep, the preimage graph G_r is maintained on each vertex using the same procedure as the reference algorithm described in subsubsection 3.3.2.2 (cf. Algorithm 6). In practice this preimage graph is implemented as a ST-Tree data structure.

8.2.3 Critical vertex detection

Critical vertices are detected using the preimage graph as done in the reference algorithm. This detection is described in subsubsection 3.3.2.2 (cf. Algorithm 5).

8.2.4 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 (similarly to the FTM algorithm in chapter 6). 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 Figure 8.2. 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 Figure 8.2a). 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 Figure 8.2, we can see the red priority queue merging into 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 (and so G_r updated), 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 at a time. Figure 8.2a 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 a same growth (blue) handles both.

8.2.5 Laziness mechanism for preimage graph

In the reference algorithm, a "lazy insertion" optimization is described. In order to make the implementation faster, additions of arcs in the dynamic graph G_r are stored into a list. When a critical vertex v is encountered, the stored operations are applied to G_r making it grow to the level set at the value $f(v)$. This way, additions and deletions of a same arc of the preimage graph are discarded, without impacting G_r. This optimization however requires to extract all saddles in a previous step, which can be done by counting the number of connected components in the lower and upper star of each vertex as described in subsection 2.2.1.

This optimization can be improved by breaking this global list of operations into local ones. A naive way would be to have one insertion 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 insertion 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.

8.3 Task-based parallel Reeb graphs

The previous section introduced a new algorithm based on local growths with Fibonacci heaps for the construction of augmented Reeb graphs. Note that this algorithm enables to process the growths starting at the minima and at the split saddles of f concurrently. The same remark goes for the join saddles; however, a join saddle growth can only be started after all of its lower link vertices have been visited. Such an independence and synchronization among the numerous growths can be straightforwardly parallelized thanks to the task parallel programming paradigm. Also, note that such a split of the work load does not introduce any supplementary computation. In the remainder, we will detail our task-based implementation for the arc growth step, and also present how we have parallelized the other steps.

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

8.3.1 Leaf search

For each vertex $v \in \mathcal{M}$, the extraction of its lower link $Lk^{-}(v)$ is a local operation. This makes this step embarrassingly parallel and enables a straightforward parallelization of the corresponding loop using OpenMP. When the optimization described subsection 8.2.5 is enabled, both the lower and the 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 sets and so end up being regular nodes in the output Reeb graph.

8.3.2 Local growth

Each local growth initiated at a leaf is independent from the others, spreading locally until it finds a join saddle. Each local growth is thus simply implemented as a task, starting at its previously extracted leaf. Each growth manages its own connected components of dynamic graph so the update on each vertex does not involve any data race. Similarly, the list of edge deletions and insertions used for the laziness

optimization described in subsection 8.2.5 only impacts the preimage graph on components local to the current growth and so no data race may occur.

8.3.3 Saddle vertex handling

The saddle vertex detection presented in subsection 8.2.4 can be implemented in parallel with tasks. For regular vertices, split saddles and maxima, only preimage graph components local to the growth are involved. In case of join saddles, a growth can make connectivity queries on preimage graph components local to another growth. The only relevant information required in such a case is the presence of edges ending in v which are not in the current preimage graph component. Such an edge can be either unvisited (its corresponding growth is yet to come), or already visited by another growth. In both case, the join saddle is detected when the number of preimage graph components in the lower star is greater than one. Such concurrent query is safely handled by our implementation.

When a join saddle s is detected, we also have to check if the current growth is the last to reach s as described in subsection 8.2.4. For this, we rely on the size of $Lk_0^-(s)$, noted $|Lk_0^-(s)|$ (number of vertices in the lower link of s). We restrict this computation to vertices where it is necessary and address synchronization issues as follows. Initially, a *lower link counter* associated with s is set to -1. Each task t reaching s will atomically decrement this counter by n_t, the number of vertices in $Lk^-(s)$ visited by t. Using here an OpenMP capture atomic operation, only the first task reaching s will retrieve -1 as the initial value of s (before the decrement). This first task will then retrieve $|Lk_0^-(s)|$ and will (atomically) increment the counter by $|Lk_0^-(s)| + 1$. Since the sum over n_t for all tasks reaching s equals $|Lk_0^-(s)|$, the task eventually setting the counter to 0 will be considered as the "last" one reaching s (note that it can also be the one which retrieved $|Lk_0^-(s)|$). We thus rely here only on lightweight synchronizations, and avoid using a critical section.

The processing done by the last task reaching the join saddle, described in subsection 8.2.4 only involves finished work. Arcs are closed, the preimage graph updated and the priority queues merged without data race.

8.4 Parallel dual sweep

In the parallel algorithm described section 8.3, 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 two sweeps use local growths as described previously and stop when they cross 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 low number of tasks otherwise.

8.4.1 Leaf search

In order to launch growths from minima and maxima, both are extracted in a single pass using the lower and upper link 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 subsection 8.2.5.

8.4.2 Local growth

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 to check if its current vertex has not already been visited by an opposite one. If so, the current arc is marked as merged with the incoming arc from the opposite growth (see Figure 8.3 (e)), and the current growth stops processing this arc. A post-processing step described in subsection 8.4.4 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. Atomic operations are used to visit (and check) vertices in order to avoid data races.

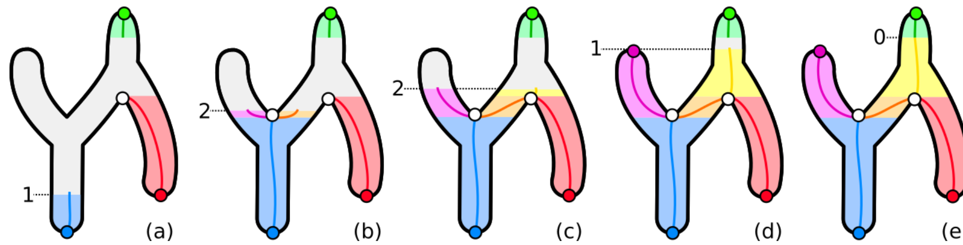


Figure 8.3 – *Evolution of the number of active arcs for the local propagation initiated at the blue minimum. The green arc is computed by a decreasing growth and is only here to show an example of arcs merging.*
(a): initially, there is one active arc (blue).
(b): after the join, there is two arcs managed by this growth (purple and orange).
(c): at the split, one arc is closed (orange) and one opened (yellow), the number of active arcs remains two.
(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.

During the traversal, each growth keeps a local counter of the number of arcs it handles (see Figure 8.3). This counter is increased when new arcs are created (Figure 8.3 (b)) and decreased when arcs are closed or merged (Figures 8.3 (d) and (e)). For the last growth continuing at a join saddle, its counter is incremented by the number of arcs each merged growth was handling. If this counter reaches 0 during the computation, the current growth has no more arc to manage and can stop (Figure 8.3 (e)).

8.4.3 Saddle vertex handling

At critical vertices, the 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 increasing growth and by a decreasing one. As detailed in section 8.5, this global lock does not have a significant impact on execution times in practice.

If a growth tries to create an already existing node, this growth is crossing an opposite one (that created the node). Therefore, the current growth does not propagate after the node as the corresponding region has already been visited by an incoming growth.

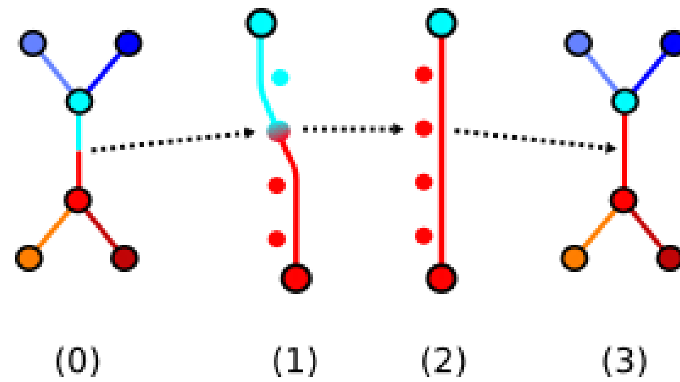


Figure 8.4 – *Two halves of an arc computed by opposite growths are merged into a single arc. Regular vertices are updated accordingly. Blue-ish colors are used for arcs computed by decreasing growth initiated at maxima, red-ish colors for increasing growth initiated at minima.*

8.4.4 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 Figure 8.3). A post processing step is in charge of computing the final arc from these two parts and to update the regular vertices accordingly (see Figure 8.4). In practice, this step takes a negligible time in our computations (less than 5% of the total time).

8.5 Results

In this section we present performance results obtained on a workstation with two Intel Xeon E5-2630 v3 CPUs (2.4 GHz, 8 CPU cores and 16 hardware threads each) and 64 GB of RAM. By default, parallel executions will thus rely on 32 threads. These results were performed with our VTK/OpenMP based C++ implementation using g++ version 7.3.0 and OpenMP 4.5. This implementation (called *Fibonacci Task-based Reeb graph*, or FTR) was built as a TTK [88] module. For the Fibonacci heap [33], we used the implementation available in Boost and for the dynamic graph we have implemented our own ST-Tree [74] data structure (cf. subsection 2.4.1).

Our tests have been performed using eight data sets from various domains. The first one, Spring, is a synthetic closed surface data set composed of four distinct springs with a radial elevation corresponding to the x coordinate. The corresponding Reeb graph has 16 leaves each leading to a large arc. Three other data sets (Dragon, BrakeDisc and Happy) are

$\|\sigma_0\|$	Data set	$\|\mathcal{R}(f)\|$	Sequential	Parallel (32 threads on 16 cores)				
			Overall	Sort	Leaf search	Sweep	Overall	Speedup
1728k	2D_Spring	44	8.86	0.06	0.19	0.41	0.66	13.42
1798k	2D_Dragon	1,681	14.66	0.06	0.19	2.45	2.70	5.43
8249k	2D_BrakeDisc	419	119.84	0.35	0.85	28.85	30.05	3.99
1303k	2D_Happy	15,599	7.15	0.04	0.15	2.64	2.83	2.53
4271k	3D_Hand	2,238	122.46	0.79	2.04	22.74	25.57	4.79
5387k	3D_Skull	27	230.64	1.04	2.50	64.68	68.22	3.38
2793k	3D_Post	131	111.38	0.48	1.22	39.73	41.43	2.69
6596k	3D_Mechanic	180	220.24	1.13	2.87	33.97	37.97	5.80

Table 8.1 – *Running times (in seconds) of the different steps of FTR on our data sets.* $|\sigma_0|$ *is the number of vertices in the mesh and* $|\mathcal{R}(f)|$ *the number of arcs in the output Reeb graph. These executions use the dual sweep strategy.*

also closed 2D surfaces and the last four (Hand, Skull, Post, Mechanic) are 3-manifolds. Most of these data sets have been subdivided in order to obtain significant execution times on our setup.

8.5.1 Performance analysis

Table 8.1 details the execution times and speedups of FTR on our data sets. One can first see that the FTR sequential execution time does not vary logarithmically with the size of the input mesh, as predicted by the complexity of the algorithm. This denotes a sensitivity on the output graph, which is common to most Reeb graph algorithms and which is further accentuated by our lazy update mechanism. Moving to parallel executions, the embarrassingly parallel leaf 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 5.2x. The almost ideal speedup (13.4x on 16 cores) of the spring data set can be used as an evidence that neither the critical section on node creation nor the atomic update on visited vertices prevent good speedups.

In order to further investigate these speedups, we present in Figure 8.5 the scaling curves of our FTR implementation on our various data sets. The first thing one can notice is the monotonous growth of these curves. This means that more threads imply shorter (or similar) execution times. We highlight 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

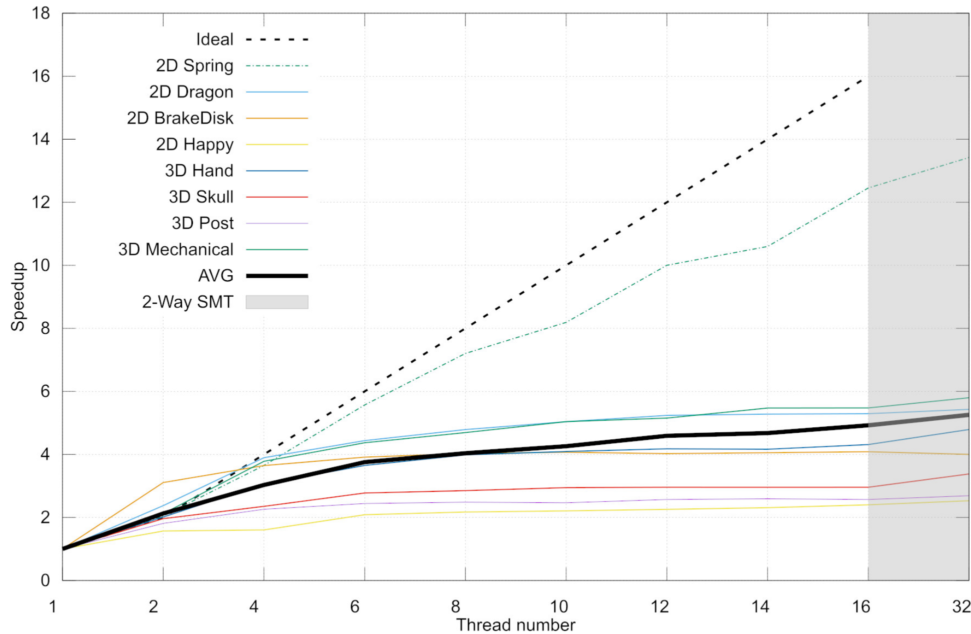


Figure 8.5 – *FTR scalability for our data sets. The gray area denotes using 2 threads per core.*

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 seem to come to a plateau around 4 for most data sets. This will be investigated further in subsection 8.5.3.

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 section 6.4, this kind of task-based approaches offers consistent computation times between executions. In our experiments, the average standard deviation obtained using 10 runs on our data sets is 0.8 second for an average time of 28.0 seconds.

8.5.2 Comparisons

In order to evaluate the performance gains obtained by our improved laziness mechanism for the preimage graph update (introduced subsection 8.2.5), we present in Table 8.2 execution times with various degrees of laziness, using the single sweep strategy and sequential executions. In the *initial* column, results are reported when no laziness mechanism is used. The *naive* column presents results when one list of insertion per local growth is used. Finally the *improved* column reports results obtained with our improved laziness mechanism, having one list

	Times			Gain/initial	
Data set	inital	naive	improved	naive	improved
2D_Spring	38.63	8.58	5.81	4.50	6.65
2D_Dragon	217.99	27.76	11.29	7.85	19.31
2D_BrakeDisc	3630.76	586.49	118.15	6.19	30.73
2D_Happy	70.00	7.34	4.33	9.54	16.17
3D_Hand	78.23	93.08	76.23	0.84	1.03
3D_Skull	248.14	209.67	159.49	1.18	1.56
3D_Post	121.77	149.40	77.39	0.81	1.57
3D_Mechanic	177.43	167.22	144.16	1.06	1.23

Table 8.2 – *Execution times (in seconds) of the sweep procedure using no lazyness (*initial*), a* naive *version with one list per propagation or our* improved *version using one list per arc. These tests are run using sequential executions of the single sweep approach.*

Data set	Single sweep	Dual sweep	Speedup
Spring	1.21	0.66	1.83
Dragon	9.43	2.70	3.49
BrakeDisc	119.10	30.05	3.96
Happy	5.14	2.83	1.82
Hand	57.71	25.57	2.26
Skull	146.64	68.22	2.15
Post	70.95	41.43	1.71
Mechanic	97.62	37.97	2.57

Table 8.3 – *Comparison of execution times (in seconds) between the single sweep and the dual sweep strategies (presented respectively in sections 8.3 and 8.4) during parallel executions.*

per arc of the output graph. These optimizations are especially efficient on 2D data sets, improving execution times by an average factor of 7.02x for the *naive* version and an average factor of 18.22x for our *improved* mechanism. On our 3D data sets, the naive version failed to expedite the computation, leading to an average speedup of 0.97x, however our improved mechanism still manages to improve the computation time by an average factor of 1.35x.

The dual sweep method introduced in section 8.4 is aimed at improving the parallel efficiency of our approach. The gains obtained by this dual sweep over a single one for a parallel execution are presented in Table 8.3. In this array, complete execution times are reported as the dual sweep method impacts both the leaf 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 double sweep mechanism hence leads to an average speedup of 2.5x over the single sweep version.

	Times				FTR Speedups		
Data set	Sweep	FTR (1)	FTR (4)	FTR (32)	(1)	(4)	(32)
2D_Spring	23.08	8.86	2.56	0.66	2.60	9.01	34.97
2D_Dragon	16.20	14.66	3.78	2.70	1.11	4.29	6.0
2D_BrakeDisc	69.71	119.84	33.80	30.05	0.58	2.06	2.32
2D_Happy	12.63	7.15	4.13	2.83	1.77	3.06	4.46
3D_Hand	147.17	122.46	40.97	25.57	1.20	3.59	5.76
3D_Skull	236.51	230.64	94.88	68.22	1.03	2.49	3.47
3D_Post	160.00	111.38	47.25	41.43	1.44	3.39	3.86
3D_Mechanic	224.93	220.24	59.50	37.97	1.02	3.78	5.92

Table 8.4 – *Reeb graph computation times (in seconds) and ratios between the original Parsa's* Sweep *algorithm (cf. subsubsection 3.3.2.2) and our Fibonnaci Task-based Reeb graph (*FTR*) implementation using 1, 4 and 32 threads.*

Additionally, the dual sweep approach implies that growths initiated at minima and those initiated at maxima can cross each others, visiting some vertices of the mesh twice (along connected components of level sets). Such a situation occurs on crossing 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.

Finally, in order to better evaluate the FTR performance, we compare our approach to the sequential reference implementation of the sweep algorithm by Parsa [60] in Table 8.2. For 3D data sets, the implementation of this algorithm (kindly provided by the authors) requires the explicit construction of the 2-skeleton of the mesh as a pre-process, whose computation times have not been reported in Table 8.2. Additionally, even if the implementation visits all the vertices of the mesh, it results in a non augmented graph, without the segmentation information. This implementation offers similar performance in sequential than our FTR algorithm. However, even with 4 threads, as we can find in any present-day setup, our implementation offers substantial performance gains (3.96x faster in average). Using 16 cores leads to slightly better performance, speeding up the computation by a factor of 8.35x in average (4.54x without Spring). In terms of memory, the footprint of the implementation of the reference sequential algorithm is higher than the one of our implementation. Internally, it pre-sorts some simplices of the 2-skeleton in arrays: vertices of each edge, edges of each triangle, adjacent triangles and edges in the neighborhood of each vertex. These arrays are used during the sweep to retrieve already sorted vertices (speeding up the computation).

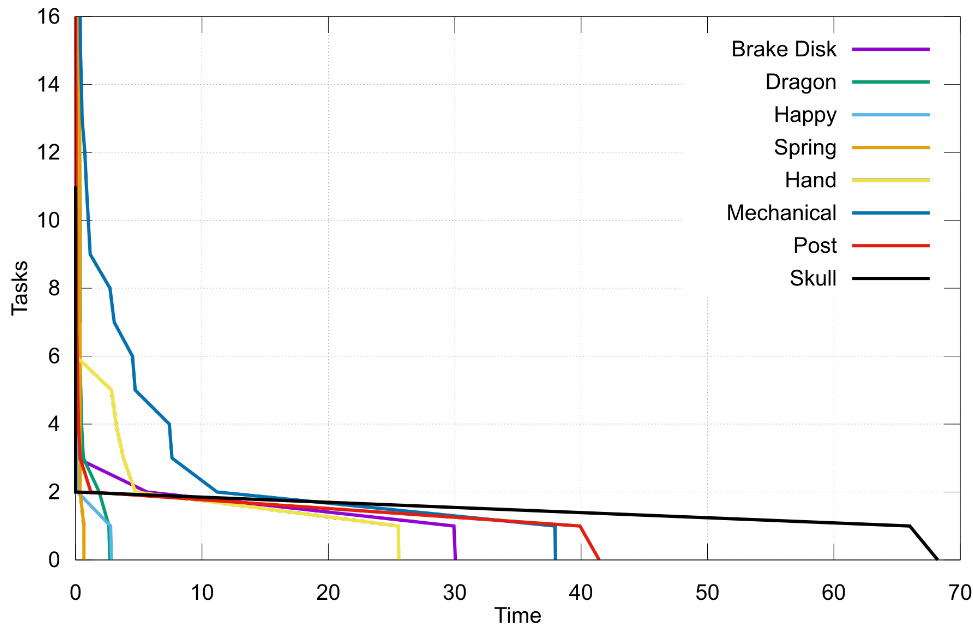


Figure 8.6 – *Number of remaining tasks throughout time. This chart is cropped at* 16 *to highlight the suboptimal section on our* 16*-core setup.*

8.5.3 Limitations

During the sweep procedure, the number of available tasks decreases through time, as local propagations merge at join saddles. There is a time during the execution where the number of available tasks eventually becomes lower than the number of threads (cf. Figure 8.6). During this *suboptimal section*, the computational power of our multi-core CPU is not fully exploited, undermining the parallel efficiency of the approach. Using the dual sweep approach and depending on the data set, there is a substantial amount of time when the number of available tasks is 2. This is the reason why our approach seems to achieve almost ideal speedups when using two threads on Figure 8.5, but fails to deliver good parallel efficiencies (except for spring) when more threads are used.

8.6 Conclusion

The method presented here is a parallel approach based on the sequential algorithm with the best time complexity. The resulting implementation is the fastest to compute augmented Reeb graphs using only four threads. However, speedups are bounded by around 5 for most data sets. This is partly due to the presence of large suboptimal sections where we do not fully exploit all available cores. Contrary to our merge tree algorithm, there is also no *trunk* step to expedite these suboptimal sections.

Part III

Exploitation

9 Applications

Contents

9.1 Persistence 137
9.2 Merge Trees 138
9.3 Contour Trees 140
9.4 Reeb Graphs 140
9.5 Real-case analysis 142
9.5.1 IEEE Scientific Visualization Contest 2016 142
9.5.2 Input data sets 142
9.5.3 Analysis 144
9.6 Conclusion 151

We present here some applications aimed to illustrate the utility of level-set based topological abstractions for data analysis and exploration. A real case analysis presented at the IEEE Scientific Visualization contest [31] is also detailed.

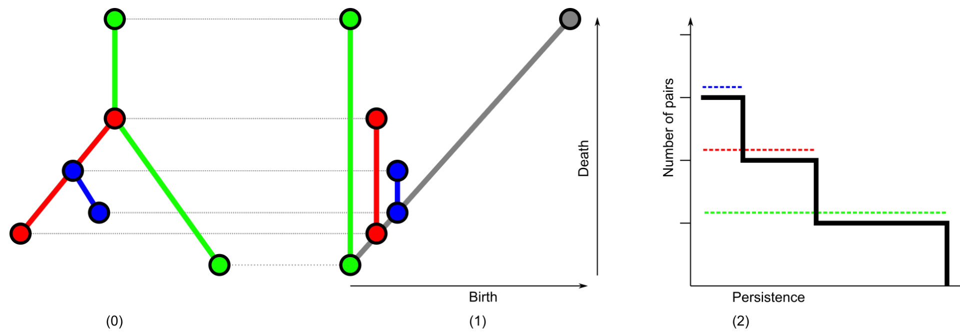


Figure 9.1 – *Persistence diagrams can be computed from the contour tree by pairing leaves with saddles hierarchically using the Elder's rule [29].*
(0): a join tree with arcs colored according to the persistence pairs.
(1): the persistence diagram corresponding to this join tree.
(2): the persistence curve corresponding to this join tree. The three persistences are shown with dotted lines.

The analyses presented in the following are aimed at illustrating the utility of level set based topological abstractions. These results have been obtained using TTK [88] and can be reproduced using data sets available on the TTK website [88]. Merge and contour tree examples follow a pipeline analogous to one used in the flexible isosurface [18] and TopoAngler [12] frameworks.

9.1 Persistence

Persistence diagrams and persistence curves [28] are powerful tools to measure the number and robustness of features on a data set. For low dimensions, these diagrams can be computed by using the hierarchy induced by merge trees (cf. Figure 9.1). Here, given a simple join tree (0) the persistence diagram (1) is obtained by sweeping the tree structure and tracking component birth and death. When arcs merge together only the oldest component is kept alive, as stipulated by the Elder's rule [29]. The scalar range of pairs thus defined is named *persistence*. The number of remaining pairs depending on a persistence threshold results in the persistence curve (shown Figure 9.1 (2)). The persistence diagram and persistence curve filters in TTK are based on FTM (chapter 6) since 2017.

In Figure 9.2, we present a classical example in fluid dynamics, the von Kármán vortex street. In this data set, vortices are created by a body disrupting a stream of liquid and studied using the rotational of the z-coordinate. The corresponding persistence curve, shown at the top right, can be subdivided in three parts. First, for low persistence

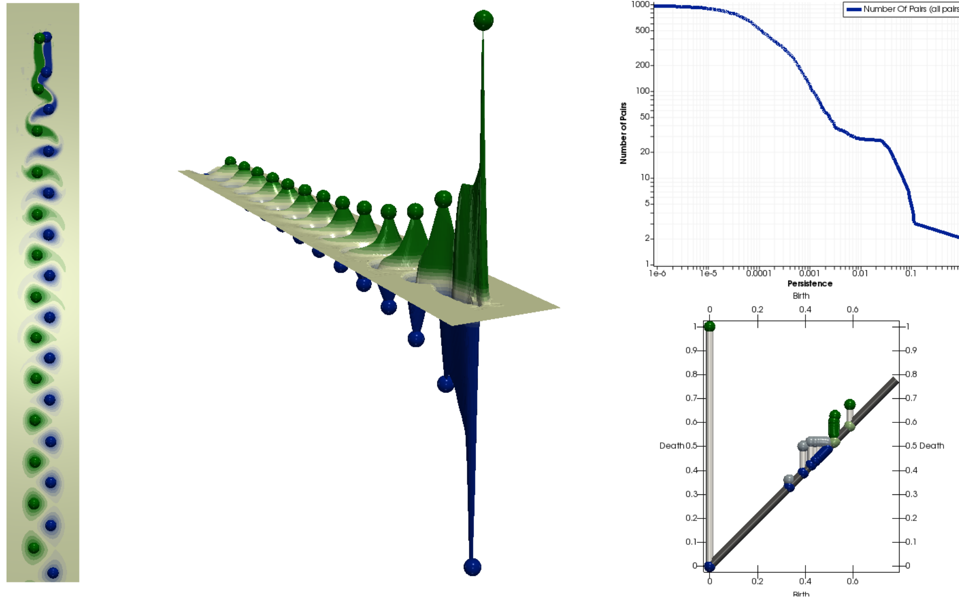


Figure 9.2 – *A two dimensional data set representing a von Kármán vortex street: a fluid stream disrupted by a blunt body. The scalar field is the rotational of the z-coordinate (normal to the plane of the data set), commonly used to study vortices. On the left, spheres are used to emphasize the main vortices. In the center, the data set is wrapped using the scalar value to show the vortices. On the right, the persistence curve and diagram of this data set are shown.*

values, the number of pairs strictly decreases. These pairs have a small persistence, thus correspond to the noise. Secondly, we can see a plateau (for persistence values ranging from 0.002 to 0.02) which corresponds to the smallest vortices emphasized with the spheres. Finally, for higher persistence values even the pairs corresponding to these vortices are removed and the curve shows a new decrease to 0. The presence of a plateau between the noise and the features is a common phenomenon, which is commonly used to drive topological simplification [87].

9.2 Merge Trees

In the following, we will see how the merge tree segmentation can be applied on medical data sets, using Figure 9.3 (0) which shows a 3D scan of a human foot. The considered scalar field is the matter density, different densities corresponding to different tissues (high density indicating bones, medium density skin and lower density air). In this case, we are interested in extracting and identifying the bones of the foot, areas of high density. Using a split tree, we are able to extract regions attached to local maxima, corresponding to the segmentation of the leaf arcs. However, the split tree

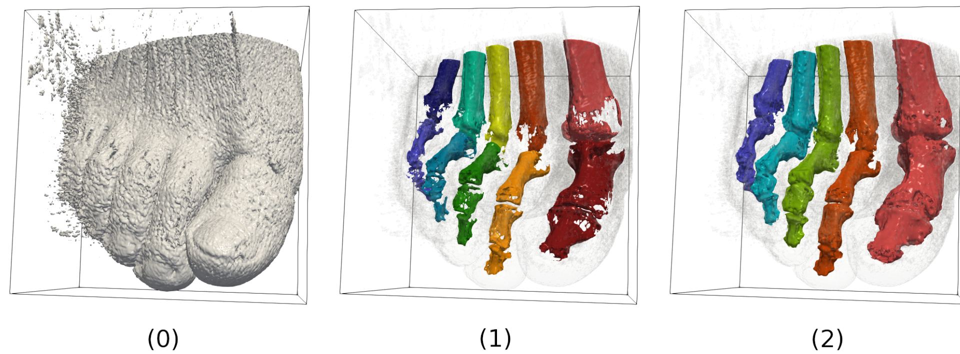


Figure 9.3 – *Medical scan of a human foot on which the scalar field is the density. We use the split tree segmentation to extract areas of high density corresponding to bones. (0) One contour corresponding to the skin of the foot. (1) The different bones highlighted using the segmentation of the deepest arcs of the tree. (2) Using topological simplification enables us to identify bones belonging to a common toe.*

of the initial data set contains 192, 375 leaves as small "bumps" due to noise in the data set lead to many local maxima. In order to reduce this noise, we pre-process the data using a topological simplification [87] based on persistent homology [29]. The persistence diagram and persistence curves allow to control this simplification in order to keep only the desired number of features. In Figure 9.3 (1), we present the segmentation obtained by the leaf arcs of the split tree when only the 10 most important leaves are kept. With this level of simplification, the resulting segmentation extracts the bone area successfully. In Figure 9.3 (2), only the five more robust leaves are kept and the resulting segmentation corresponds to the five toes of the foot. Thanks to the merge tree algorithms and implementations presented in this thesis, this exploration can be done in a handful of seconds on our setup, even for 512^3 grids, which greatly improves interactivity in visual exploration tasks.

Another example emphasizing the interest of merge tree segmentation is shown in Figure 9.4. This data set is a chemical one, representing the electronic density of an Ethylene Glycol molecule. In (0) the initial contour tree of the data set is shown. Opaque regions are areas corresponding to leaf arc segmentations. In this data set, each maximum is an atom, the smallest ones being hydrogen. Saddles of the electron density (red spheres in the figure) are located in configurations at the boundary between multiple atom influence zones. These correspond to covalent bounds. Here, we are not interested in areas attached to minima so in (1), only the split tree is considered. Topological simplification as previously described is also used, so each region corresponds to an atom group: hydrogen atoms

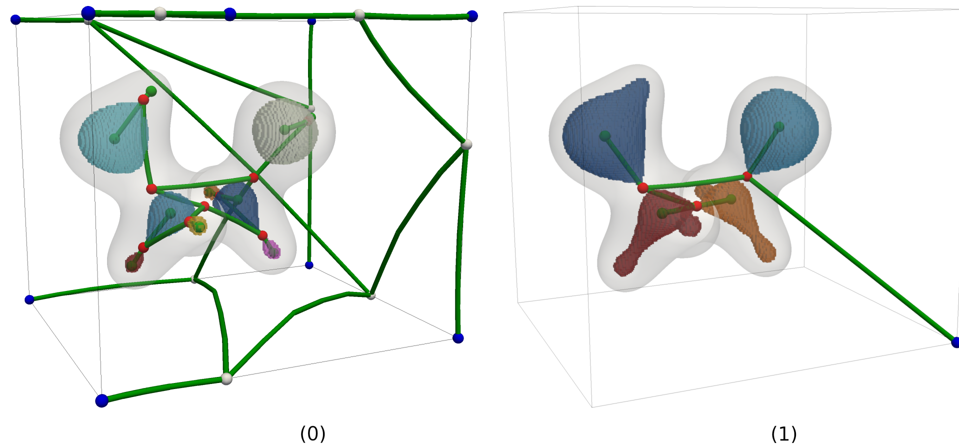


Figure 9.4 – *Molecular data set (representing an Ethylene Glycol molecule) on which the scalar value is the electronic density. In (0), the full contour tree is shown and in (1), the simplified split tree allows to extract regions containing carbon and oxygen atoms (linked to hydrogen ones).*

are merged with the atom they are linked to. Such a segmentation enables quantitative analysis of these features (for instance volume measurement).

9.3 Contour Trees

For the contour tree algorithm, we present the topological analysis of the result of an Enzo simulation in Figure 9.5. This data set contains cosmology simulation results, the scalar field being the density of matter on each vertex. Such a simulation is used in order to better understand the growth of the universe, as well as the dark matter distribution. Once again, the initial data set (0) is too noisy for a human exploration so topological simplification is used. This result in (1), where regions attached to maxima form the *cosmic web*, with areas of high density linked together by long filaments and surrounded by large zones of lower density named *voids*. These two types of regions correspond to leaf arcs of the contour tree, the core structure of the cosmic web is shown using opaque areas in (1).

9.4 Reeb Graphs

To emphasize another use of level set based abstractions, we will move away from scientific data analysis and use the Reeb graph to extract the skeleton of a 3D mesh. Its ability to track shapes has already been used in automatic rigging [7, 66], in the context of 3D animation. In Figure 9.6, we present a mesh of a dancer (0). Using a distance field from the center of

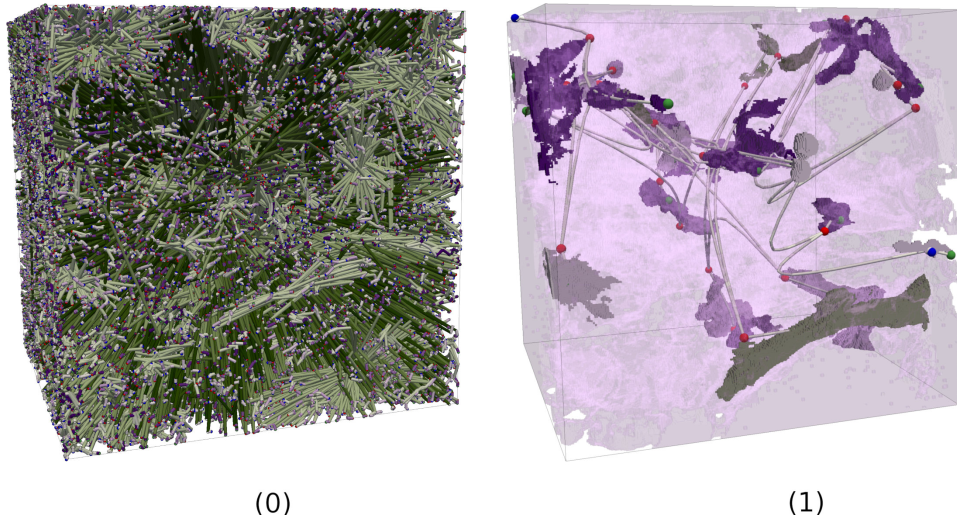


Figure 9.5 – *The Enzo data set is a regular grid on which the defined scalar field is the matter density, obtained from a universe simulation. This type of data set is used to analyze the cosmic web. In (0), the full contour tree of the data set is shown, leading to challenging exploration. In (1) topological simplification is used and the core structure of the cosmic web is shown using opaque areas.*

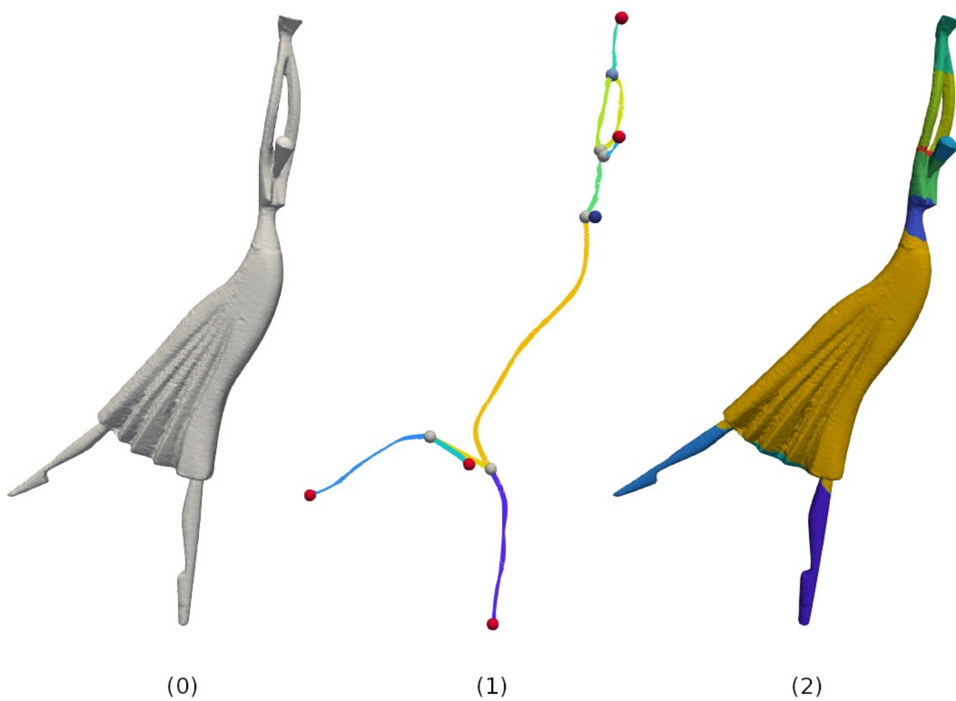


Figure 9.6 – *(1) Skeleton and (2) segmentation of the mesh (0) of a dancer, using a distance field from the center of the mesh.*

the mesh, the corresponding Reeb graph is able to accurately describe the shape of the dancer, each limb being represented by a distinct arc of the graph (1). The corresponding segmentation is shown in (2).

9.5 Real-case analysis

In the following, we present a real-case analysis where level set based abstractions (and more precisely persistence diagrams) have been used. The amount of data to explore in this use case was large (several terabytes) and so efficient approaches were required in order to process all the data in the given time frame. In particular, the pipeline presented in the following makes use of the persistence diagram, which relies on our Fibonacci Task-based Merge tree presented in chapter 6. This section is also used to show a full analysis pipeline.

9.5.1 IEEE Scientific Visualization Contest 2016

In 2016, the IEEE "Sci Viz" contest [2] focused on a phenomenon studied in material science, called *viscous fingering*. Our participation led to two publications: the first one is our submission for the contest [31] for which we received an honorable mention. The second one [51] combine our work with the results of the winning team. Their submission uses the Reeb graph segmentation to identify fingers and to track them, along with an interactive tracking graph to represent their evolutions through time. The approach presented in the following only relies on our submission.

During the contest, several tasks were given to the participants. We had to create a framework allowing a (near-)interactive visualization and browsing of the data. Fingers had to be identified at each time step and we had to be able to track them through time in order to make statistics about their evolutions (as individuals and all together).

9.5.2 Input data sets

Viscous fingering is an instability phenomenon which occurs in porous media at the interface between two fluids of distinct viscosity. In particular, it appears when a less viscous fluid is injected within a more viscous one. It intervenes in many fields of science and engineering, including geology, hydrology as well as in oil industry where it plays a key role in the extraction process.

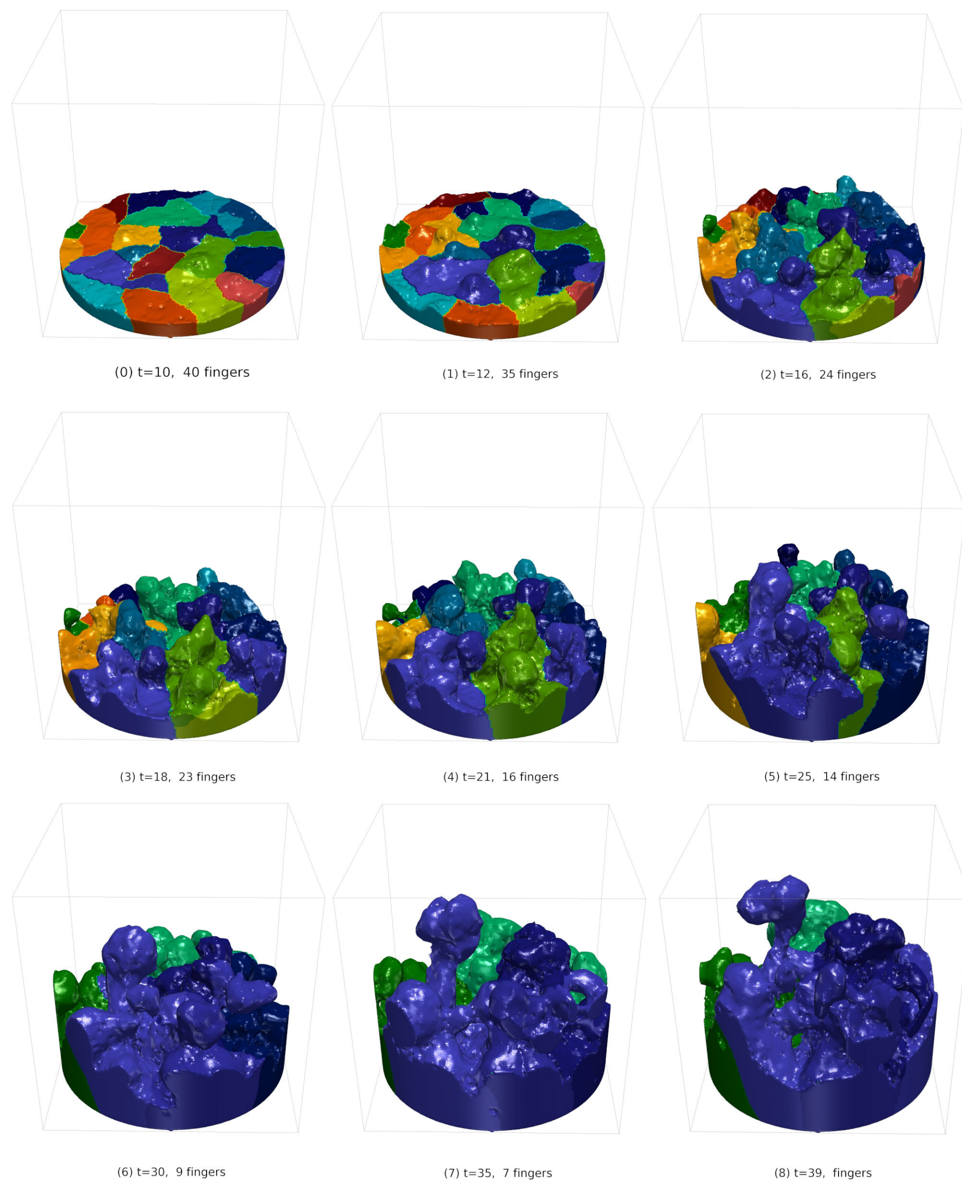


Figure 9.7 – *Nine time-steps of a same run on the simulation of continuously dissolving salt and water after we have extracted and identified fingers created by the viscous fingering phenomenon. Data sets are shown upside down to reduce occlusion.*

This phenomenon is characterized by the formation of a characteristic pattern, called *viscous fingers* (Figure 9.7). In particular, the geometrical evolution of these patterns provides good indications about the evolution of the penetration of the less viscous fluid. Thus, capturing, tracking and analyzing the geometry of viscous fingers is of first importance for the understanding of the penetration process.

Viscous fingering can be decomposed into three major regimes. First, the *launch*: initially, the interface between the two fluids is approximately planar. Then, the less viscous fluid starts to penetrate the more viscous one when sufficient injection force is applied to it. Viscous fingers of low and uniform amplitude start to appear in this phase as shown Figures 9.7 (0) and 9.7 (1). Then, the *expansion* phase occurs: once the reaction is launched, the difference of pressure between the two fluids tends to favor an acceleration of the penetration speed for the areas where the less viscous fluid penetrates the most the more viscous fluid. In other words, larger fingers will tend accelerate and grow faster than smaller ones, cf. Figures 9.7 (3) to (8). Optionally, a *termination* phase: depending on the characteristics of the media and of the fluids, the two fluids can eventually mix together in a termination state, where the finger pattern has completely disappeared, after the merge of the large fingers.

In our case, the simulation studies viscous fingering in the context of the mix of continuously dissolving salt and water. As detailed in [2], the simulation runs are given as time-varying particle data-sets representing salt concentration. Due to the stochastic nature of the simulation algorithm, several runs are considered, at distinct resolutions. This yields the following challenges:

1. The overall volume of generated data prevents a fast and easy analysis, visualization and interpretation of the phenomenon.

2. The stochastic nature of the simulation code raises the question of the stability of the fingering process across several runs, which needs to be analyzed.

3. The multi-resolution nature of the data raises the question of the convergence of the simulation code, which also needs to be analyzed.

9.5.3 Analysis

In this section, we present an interactive framework for the analysis and visualization of ensembles of viscous fingers. In particular, we show

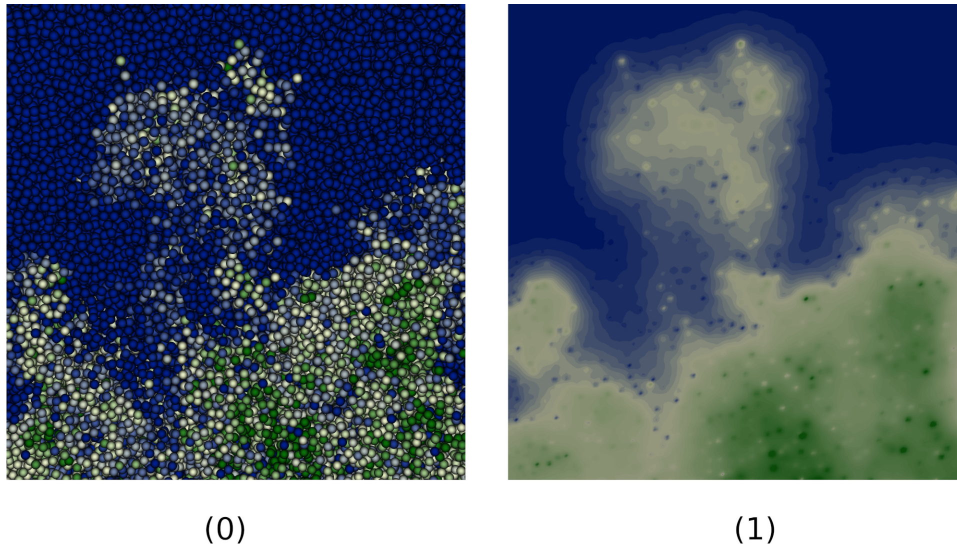


Figure 9.8 – *On a single time-frame, (0) the initial data sets composed of independent vertices and (1) the result after our Shepard's method has been used.*

how to extend and adapt to viscous fingering the data analysis pipeline proposed by Laney et al. [48] in the context of the study of the Rayleigh Taylor instability. Finally we report the findings we made using our data analysis framework.

9.5.3.1 Data Pre-processing

The input data is given by three sets of simulation runs (one set per resolution). Each run is represented by time-varying particles carrying salt concentration. We pre-process each time-step by computing a volumetric interpolation of the particle data onto a 128^3 regular grid, as shown in Figure 9.8. This results in a tetrahedral mesh that will be the input to our data analysis pipeline (cf. subsubsection 9.5.3.2).

9.5.3.2 Data Analysis Pipeline

Our data analysis pipeline adapts the approach of Laney et al. [48] to viscous fingering. It is composed of 5 steps, illustrated in Figure 9.9, which are described in the following.

1. **Fluid discrimination per time-step:** For each time step we separate the dissolving salt from the ambient water. Given the piecewise linear scalar field f_c representing salt concentration and defined on the tetrahedral mesh $\mathcal{M}$ representing our input domain, we identify as dissolving salt the sur-level set $\mathcal{L}^+$ of f_c at the isovalue i_{salt} : $f_+^{-1}(i_{salt})$ (cf. section 2.2). From

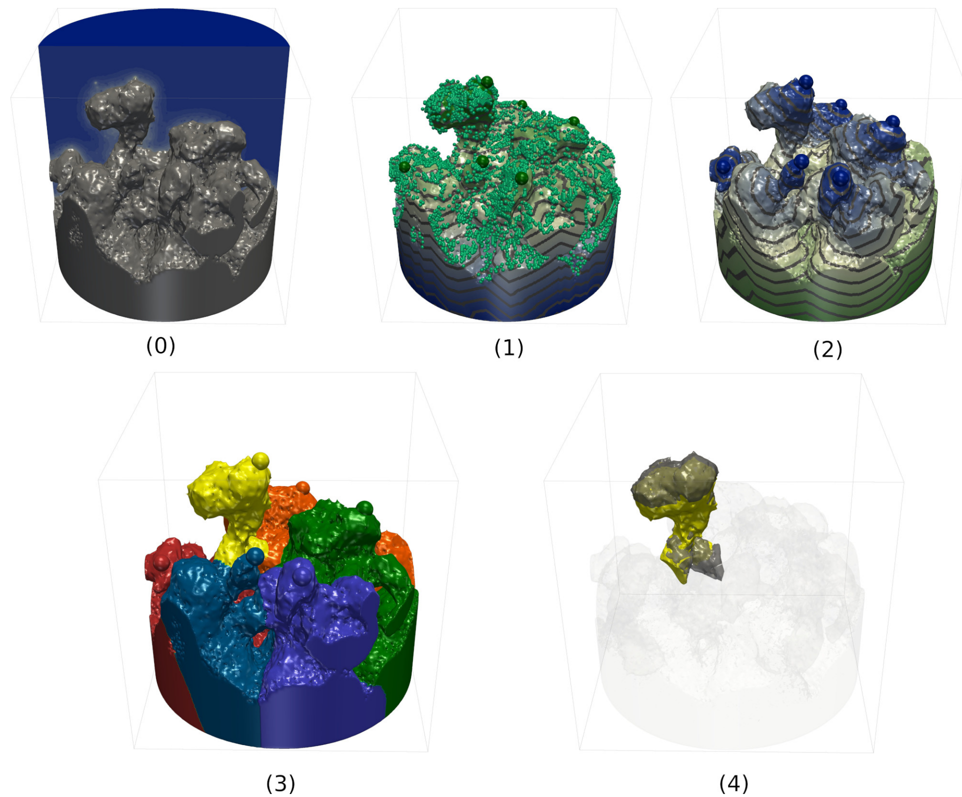


Figure 9.9 – *Overview of our topological data analysis pipeline. (0) The dissolving salt is first isolated from the ambient water by considering the largest connected component (noted $\mathcal{S}$, in gray) of the sur-level set of salt concentration. (1) Finger tips are identified as local maxima (small light green spheres) of the geodesic distance $f_d : \mathcal{S} \rightarrow \mathbb{R}$ (color gradient and level lines) from the top of the cylinder. Restricting the identification to the most persistent maxima (larger dark green spheres) enables the identification of the most prominent fingers. (2) Geodesic distance field from the most persistent maxima $f_t : \mathcal{S} \rightarrow \mathbb{R}$. (d) The Morse complex of f_t decomposes $\mathcal{S}$ into fingers. (3) Each finger at time step t is connected to the finger at time step $t+1$ which maximizes the volume of their intersection. (4) An example finger is shown in yellow while the corresponding maximizer at time step $t+1$ is shown in transparent black. Data-sets are shown upside down to reduce occlusion.*

our experience, we found that an isovalue i_{salt} of 10 gave consistent results along time steps and across runs. Finally, to ignore spurious bubbles, we only consider in the remainder the largest connected component of $\mathcal{L}^+(i_{salt})$, that we note $\mathcal{S}$ (Figure 9.9(0)).

2. **Finger identification per time-step:** Given the geometry of the dissolving salt $\mathcal{S}$, we aim at extracting in a robust manner the tips of the viscous fingers. To achieve this, we consider as finger tips the points which are locally the furthest away from the top of the domain, where salt is continuously added. Intuitively, this corresponds to salt particles which traveled the furthest from their origin. As shown in Figure 9.9(1) (small light green spheres), this strategy identifies as finger tip even slight bumps in the geometry. Thus, we employ persistent homology [29] (using the persistence diagram introduced section 9.1) to filter the maxima. In particular, we found in practice that preserving maxima whose persistence is higher than 10% of the function span provides consistent results along time-steps and across runs (Figure 9.9(1), large dark green spheres). Next, given the list of finger tips $\mathcal{T}$ previously identified, we compute for each vertex of $\mathcal{S}$, the geodesic distance to its closest finger tip, noted $f_t : \mathcal{S} \rightarrow \mathbb{R}^+$ (Figure 9.9(2)). We finally identify as *viscous fingers* each cell of the Morse complex of f_t (a topological abstraction based on the gradient, see [22] and [40]). The result is shown Figure 9.9(3).

3. **Finger tracking per run:** Once fingers have been extracted on a per time-step basis, we proceed to their tracking through time with an approach similar to topology based techniques [15, 76]. In particular, for each finger at a time-step t, we connect it to the finger at time-step $t+1$ which maximizes the volume of their intersection (Figure 9.9(4)).

4. **Quantitative analysis per run:** Once the fingers have been tracked through time, various time-varying statistics are computed on a per finger basis, such as the evolution of its volume for instance (see subsubsection 9.5.3.3 for a comprehensive list of measures). Each of these statistics is shown to the users as a $1D$ plot over time with a color code matching that of the segmentation in the 3D view (cf. Figure 9.10).

5. **Comparative analysis across runs:** Our user interface also offers comparative analysis capabilities by supporting the side by side display

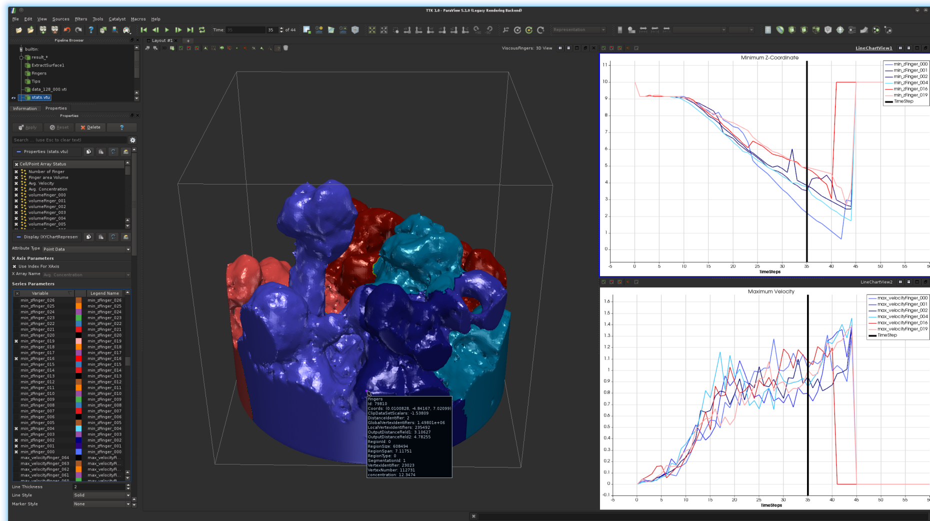


Figure 9.10 – *Screen-shot of the user interface to our data-analysis framework (intra-run mode). Users can select from the bottom-left list the* per-finger, *time-varying statistics to visualize on the right side of the screen (white background). There, the vertical black line indicates the time-step being currently visualized in the linked 3D view (center). Users can navigate through time steps with the time navigation buttons (top, center). Various per-finger statistics can be displayed in the 3D view by pointing on a finger with the cursor (dark rectangle).*

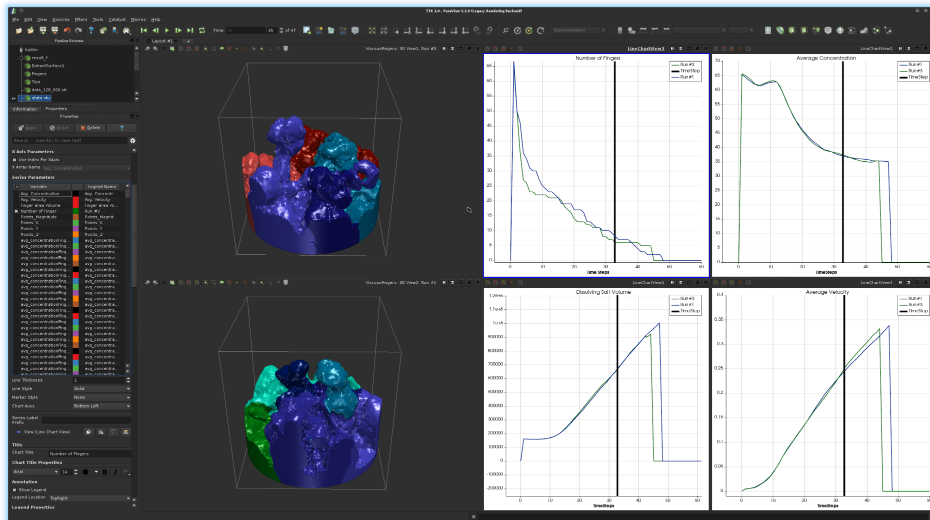


Figure 9.11 – *Screen-shot of the user interface to our data-analysis framework (comparative inter-run mode). Users can select from the bottom-left list the* global *time-varying statistics to visualize on the right side of the screen. In this case, the evolution of the number of fingers, of the fingers' volume, of average salt concentration and average velocity are shown for the runs #1 and #3 of the lowest particle resolution. There, the vertical black line indicates the time-step being currently visualized in the linked 3D views (center top: run #3, center bottom: run #1). Users can navigate through time steps with the time navigation buttons (top, center).*

of multiple time-varying *global* statistics (one per run), as well as 3D views that are linked to each of the time-varying statistics windows (Figure 9.11).

9.5.3.3 Results

This section reports the findings we made using our data analysis framework. In particular, we first focus our analysis on the 22 runs at high resolution (1.7M particles, green curves in Figure 9.12). The comparative analysis across resolutions (544k and 194k particles) is discussed in the **Inter-resolution analysis**

Regime identification: In this paragraph, we first try to corroborate the decomposition of the fingering process in three regimes (subsection 9.5.2). To do so, we will first inspect summary views of global statistics (cf. the **Comparative analysis across runs** step in subsubsection 9.5.3.2) for all runs. We first analyze the evolution of the descent of the dissolving salt, by looking at the *Minimum Z-Coordinate* (in Figure 9.12) of $\mathcal{S}$ through time: high resolution are shown using green curves. The three regimes identified in subsection 9.5.2 are clearly visible: the *launch* phase for time steps 0 to 10, the dissolving salt remains at the top of the domain; then the *expansion* phase from time steps 10 to 50, the dissolving salt traverses the domain almost linearly; finally the *termination* step occurs between time steps 50 and 60.

Characteristics of the expansion regime: In this paragraph, we try to corroborate the description of the fingering process in the expansion regime (subsection 9.5.2), where larger fingers are supposed to grow faster, at the expense of smaller ones, which they eventually absorb. To do so, we first inspect the evolution of the *Finger Number* through time (in Figure 9.12). This plot confirms that the number of fingers globally decreases for all runs, with a consistent decrease rate across runs, to eventually tend to a small number (typically five or less) towards the end of the expansion regime.

Inter-resolution analysis: In this paragraph, we study the impact of the input data particle resolution on the characterization of the viscous fingering process. To do so, we visualize global summarizes provided by global time-varying statistics for all runs in Figure 9.12. In particular, each of the three resolutions is represented with a distinct color. These three sets of curves exhibit similar global behaviors: a salt descent

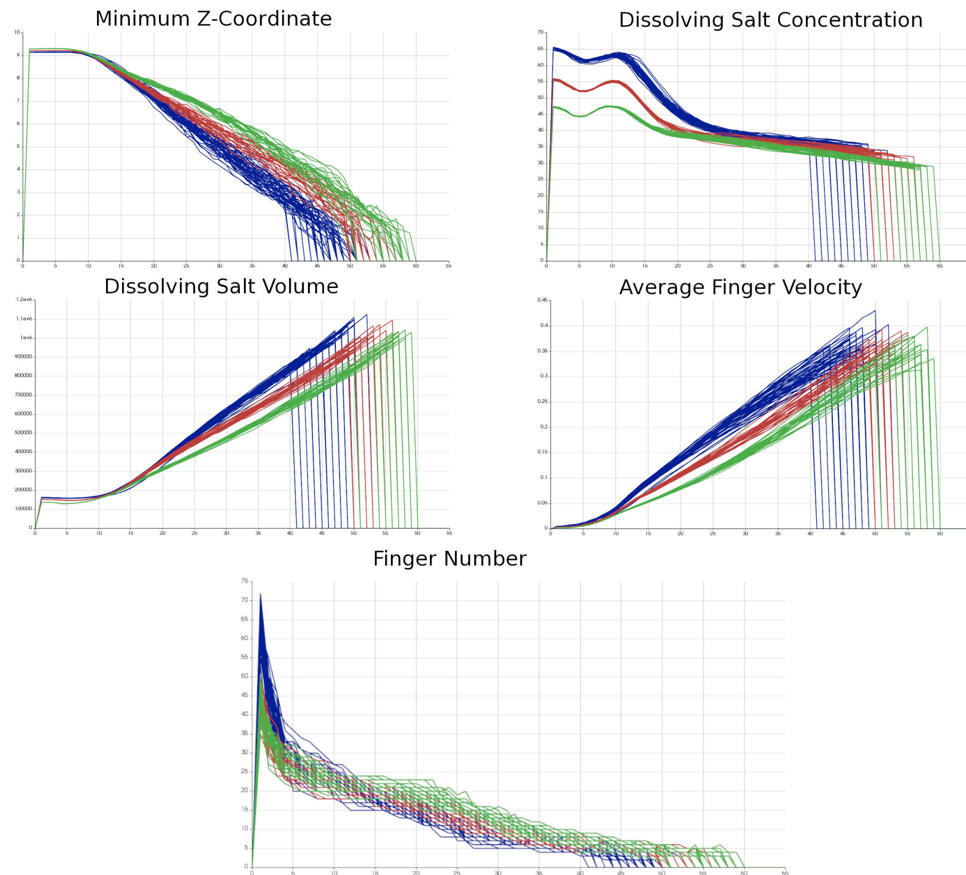


Figure 9.12 – *Global statistics as a function of time for the* 22 *high-res (green curves),* 23 *medium-res (red curves) and* 48 *low-res (blue curves) runs. These three sets of curves exhibit similar global behaviors: a salt descent according to a linear slope (**Minimum Z-Coordinate**), a linear decrease of salt concentration in the expansion regime (**Dissolving Salt Concentration**) and a linear increase in both volume (**Dissolving Salt Volume**) and velocity (**Average Finger Velocity**). However, each resolution can be easily distinguished from the others as curves of the same color tend to cluster, with only few overlap with the other colors. This indicates clear distinctions between resolutions. The **Minimum Z-Coordinate** of the dissolving salt $\mathcal{S}$ indicates clearly distinct descent rates for the deepest fingers (different slopes). Here lower resolutions hit the bottom of the domain faster. The **Average Salt Concentration** of the dissolving salt $\mathcal{S}$ indicates clearly distinct initial salt concentration levels, with a slight delay for the start of the expansion regime (local maximum of concentration) as the resolution decreases. The **Dissolving Salt Volume** $\mathcal{S}$ also indicates different growth rates (lower resolutions grow faster), as suggested by the **Minimum Z-Coordinate**. The **Average Finger Velocity** within the dissolving salt $\mathcal{S}$ also indicates different average speeds, confirming the increase in speed for lower resolutions. The evolution of the **Finger Number** indicates a clear distinction for the lowest resolution in the early time-steps. However, this increase is not confirmed in the expansion regime (typically around time-step* 25*), where fewer fingers are extracted as the resolution decreases. In conclusion, runs of distinct resolutions exhibit similar global behavior, however with later expansion starts, faster penetration rates and fewer viscous fingers as the resolution decreases. This comparative study shows that the similarity to the highest resolution runs (according to the above criteria) decreases with the resolution, suggesting a convergence of the simulation code for increasing resolutions.*

according to a linear slope in *Minimum Z-Coordinate*, a linear decrease of salt concentration in the expansion regime in *Dissolving Salt Concentration* and a linear increase in both volume and velocity in *Dissolving Salt Volume* and *Average Finger Velocity*. However, each resolution can be easily distinguished from the others as curves of the same color tend to cluster, with only few overlap with the other colors. This indicates clear distinctions between resolutions.

9.6 Conclusion

For all the applications presented in this chapter, the time to wait for a result to be available harms the user experience. We have already seen how a 10-fold speedup can improve the interactivity in chapter 4 and we have presented here some examples where our efficient parallel algorithms can be used to transform a tedious wait into an interactive exploration. This motivation for interactivity has also been illustrated by the "Sci Viz" contest, for which interactivity was the first challenge participants were asked to address.

Conclusion 10

In this manuscript, we have presented a framework for the efficient parallel computation of level set based topological abstractions on multi-core shared memory workstations. The approaches presented here are generic, both in terms of input and output. Input meshes can be any dimensional triangulations. In practice, any mesh can be easily converted into a triangular one. The output of our algorithms are augmented abstractions, containing all the segmentation information and allowing the full extent of level set based analysis.

The major contribution of this manuscript resides in the use of the task mechanism to compute, in parallel, augmented level set based topological abstractions. We have revisited efficient sequential algorithms [16, 60] relying on a global view of the data and we have designed new approaches based on independent local propagations expressed as parallel tasks. These propagations, based on sorted breadth-first searches, rely on the Fibonacci heap data structure in order to traverse the mesh with the same complexity as the original sweep traversal. Resulting algorithms can be executed in parallel with no overhead, and benefit from the dynamic load balancing induced by the task runtime. Task-based approaches presented for the augmented merge and contour tree computations are, in practice, more than twice as fast in sequential as the traditional algorithm [16] introduced in 2000. This speedup is due to an optimization (named the *trunk*), where the last growth processing arcs forming a monotone super-arc-path ending at the root of the tree is replaced by a procedure having a linear time complexity. Additionally, these approaches offer significant speedups on our 16-core setup, with an average speedup of 9.29x for our merge tree algorithm and of 8.12x for our contour tree algorithm.

From a user perspective, this framework requires no knowledge of the underlying algorithms as more threads always imply faster computations. In practice, the level set based abstractions presented in this manuscript

are of great interest for scientists and their parallelization allows for new interactive applications [12, 90, 92].

Finally, this work is part of the TTK [88] open-source library, a software collection for topological data analysis providing generic, efficient and robust implementations of key algorithms in this domain. As this framework is open-source, it is freely available to any end user (students, engineers, researchers) and also to any developer, whom contributions are welcome. The free availability of this work is also a way to help researchers reproduce our results for comparison. Thanks to TTK, this work is also available in VTK [70] and ParaView [8], which are two well established scientific visualization software packages, providing all the tools required (I/O, rendering, user interaction, filters,. . .) to load, represent, manipulate and explore scientific data sets.

Perspectives

We think the locality of the task-based merge tree approach presented in chapter 6 can be exploited for an on-the-fly simplification of the output data structure, by maintaining the persistence of the local growths during the arc computations. The hierarchical traversal made by the growths during the local propagation step is similar to how the persistence pairs are constructed from the final merge tree. We believe this property could be exploited to simplify the merge tree during the computation, without having to rely on a preprocessing stage like this is actually the case. This would greatly simplify the pipeline for most merge and contour tree based analysis and improve the interactivity of the exploration. For the Reeb graph algorithm presented in chapter 8, we believe the sequential efficiency can be pushed further. Using advanced profiling may help us locate the hotspots in our implementation and guide our future optimizations. Additionally, we would like to check the execution time of our implementation on new data sets to bring more diversity, especially in 3D where most of our actual data sets only have a low number of arcs.

In terms of performance, out-of-core algorithms could be considered. These approaches are used to process large data sets that do not fit in memory by visiting the data set piece by piece without holding it entirely in RAM. The memory footprint of approaches developed in this manuscript is large as we focused our efforts on speed optimization. With 64 GB of RAM, we could not compute the merge tree of most of our 1024^3 data sets. Furthermore, the memory consumption of the Reeb graph

algorithm is even higher, due to more adjacency information required in memory. Improving the memory consumption of these approaches is definitely a future work, but adapting these algorithms to make them out-of-core would have a greater impact, especially for the command line version of our plugins, suited for streaming processing.

The main limitation of our approaches is the presence of a suboptimal section as introduced in chapters 6, 7 and 8. On these task-based approaches the number of tasks strictly decreases during the computation and at some point becomes lower than the number of core, which undermines their parallel efficiencies. This problem is worse when a high number of cores is used as the suboptimal section intervenes sooner and the number of idle threads is higher. We believe that task parallelism will not enable us to overcome these performance bottlenecks, as these are intrinsic to our approaches. We rather think that only a complete revisit of our algorithms would allow to circumvent these limitations: this is currently a real challenge.

At the pipeline level, we would like to study the benefits of our task-based approaches when computing several abstractions simultaneously. Presently, each abstraction is computed in a separate process, and each process has several threads. We believe it could be beneficial to use a single task pool for the tasks of all abstractions. This would allow the runtime to reduce the suboptimal sections using a task overlap mechanism similar to the one introduced in chapter 7, by efficiently overlapping independent steps of the pipeline. Additionally, a single task pool would reduce the number of threads created and the memory contention between them. Furthermore, we would like to emphasize the suitability of our task-based approaches for *in-situ* visualization. In this context, topological data analysis algorithms are run alongside the simulation and resources are shared between them. The local nature of our algorithm could be exploited in order to only process parts on which the simulation is already completed. Additionally, the resources allocated to the topological data analysis algorithm may vary during the computation and the dynamic load balancing induced by the task runtime is better suited to a variable number of threads. Finally, if a single process is used for both a task-based simulation and our task-based abstraction computation, these can share a single task pool so that task priorities can be used to drive the computation.

Overall, we believe task-based parallelism as presented here has a great potential to parallelize algorithms relying on a sorted traversal of the input domain. Therefore, we would like to exploit it for other types

of topological abstractions, that could lead to new contributions to the TTK library. In the longer term, the TTK library is meant to be tied more closely with Paraview. The idea is to bring new features to people already using Paraview, but also to potentially open new markets for the Kitware company. TTK may help topological data analysis to become more accessible to end users.

Bibliography

[1] Intel Cilk Plus homepage. URL http://www.cilkplus.org/. Accessed Sep. 16th, 2014. (Cited page 86.)

[2] IEEEVIS. Scientific visualization contest. 2016. http://www.uni-kl.de/scivizcontest/. (Cited pages 142 and 144.)

[3] A. Acharya and V. Natarajan. A parallel and memory efficient algorithm for constructing the contour tree. In *pacificVis*, 2015. (Cited pages 44, 47, 56, 57, and 77.)

[4] AIM@SHAPE. AIM@SHAPE Shape Repository. http://shapes.aim-at-shape.net/, 2006. (Cited page 70.)

[5] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The landscape of parallel computing research: a view from Berkeley. Technical report, University of California at Berkeley, 2006. (Cited page 73.)

[6] Cedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-Andre Wacrenier. StarPU: A Unified Platform For Task Scheduling On Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198. doi: 10.1002/cpe.1631. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1631. (Cited pages 33 and 57.)

[7] Grégoire Aujay, Franck Hétroy, Francis Lazarus, and Christine Depraz. Harmonic Skeleton for Realistic Character Animation. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '07, pages 151–160, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN 978-1-59593-624-0. URL http://dl.acm.org/citation.cfm?id=1272690.1272711. (Cited pages 55 and 140.)

[8] Utkarsh Ayachit. *The ParaView Guide: A Parallel Visualization*

Application. Kitware, Inc., USA, 2015. ISBN 1930934300, 9781930934306. (Cited pages 58 and 154.)

[9] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock, and E. W. Bethel. In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms. *Computer Graphics Forum*, 35(3): 577–597. doi: 10.1111/cgf.12930. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12930`. (Cited page 35.)

[10] Silvia Biasotti, Michela Mortara, and Michela Spagnuolo. Surface compression and reconstruction using reeb graphs and shape analysis. In *Spring Conference on Computer Graphics*, pages 174–185, 2000. (Cited page 49.)

[11] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.*, 30(8):207–216, August 1995. ISSN 0362-1340. doi: 10.1145/209937.209958. URL `http://doi.acm.org/10.1145/209937.209958`. (Cited page 32.)

[12] Alexander Bock, Harish Doraiswamy, Adam Summers, and Claudio Silva. Topoangler: Interactive topology-based extraction of fishes. *IEEE transactions on visualization and computer graphics*, 24(1):812–821, 2018. (Cited pages 3, 55, 137, and 154.)

[13] Roberto A Boto, Julia Contreras-García, Julien Tierny, and Jean-Philip Piquemal. Interpretation of the reduced density gradient. *Molecular Physics*, 114(7-8):1406–1414, 2016. (Cited page 3.)

[14] Roger L Boyell and Henry Ruston. Hybrid techniques for real-time radar simulation. In *Proceedings of the November 12-14, 1963, fall joint computer conference*, pages 445–458. ACM, 1963. (Cited page 39.)

[15] Peer-Timo Bremer, Gunther Weber, Julien Tierny, Valerio Pascucci, Marc Day, and John Bell. Interactive Exploration and Analysis of Large-Scale Simulations Using Topology-Based Data Segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 17(9):1307–1324, September 2011. ISSN 1077-2626. doi: 10.1109/TVCG.2010.253. URL `http://dx.doi.org/10.1109/TVCG.2010.253`. (Cited pages 3, 55, and 147.)

[16] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. In *Proc. of Symposium on Discrete Algorithms*, pages 918–926, 2000. (Cited pages 39, 42, 45, 46, 47, 49, 56, 57, 64, 66, 77, 79, 83, 94, 101, 104, 105, 110, 111, 117, and 153.)

[17] H. Carr, G. H. Weber, C. M. Sewell, and J. P. Ahrens. Parallel peak pruning for scalable SMP contour tree computation. In *ldav*, 2016. (Cited pages 45, 48, 56, and 77.)

[18] Hamish Carr, Jack Snoeyink, and Michiel van de Panne. Simplifying Flexible Isosurfaces Using Local Geometric Measures. In *Proceedings of the Conference on Visualization '04*, VIS '04, pages 497–504, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8788-0. doi: 10.1109/VISUAL.2004.96. URL `http://dx.doi.org/10.1109/VISUAL.2004.96`. (Cited pages 3, 55, and 137.)

[19] Fang Chen, Harald Obermaier, Hans Hagen, Bernd Hamann, Julien Tierny, and Valerio Pascucci. Topology analysis of time-dependent multi-fluid data using the Reeb graph. *Computer Aided Geometric Design*, 30:557–566, 07 2013. doi: 10.1016/j.cagd.2012.03.019. (Cited pages 3 and 55.)

[20] Y. Chiang, T. Lenz, X. Lu, and G. Rote. Simple and optimal output-sensitive construction of contour trees using monotone paths. *Computational Geometry Theory and Applications*, 2005. (Cited pages 41, 44, and 56.)

[21] T.H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009. (Cited pages 26, 27, 58, 80, 83, and 87.)

[22] Leila De Floriani, Ulderico Fugacci, Federico Iuricich, and Paola Magillo. Morse complexes for shape segmentation and homological analysis: discrete models and algorithms. In *Computer Graphics Forum*, volume 34, pages 761–785. Wiley Online Library, 2015. (Cited page 147.)

[23] S. Dillard. libtourtre: A Contour Tree Library. `http://graphics.cs.ucdavis.edu/~sdillard/libtourtre/doc/html/`, 2007. (Cited pages 56, 70, 77, 83, 93, 94, 111, and 112.)

[24] H. Doraiswamy and V. Natarajan. Output-Sensitive Construction of Reeb Graphs. *IEEE Transactions on Visualization and Computer Graphics*,

18(1):146–159, Jan 2012. ISSN 1077-2626. doi: 10.1109/TVCG.2011.37. (Cited pages 49, 50, 56, and 57.)

[25] Harish Doraiswamy and Vijay Natarajan. Efficient algorithms for computing Reeb graphs. *Computational Geometry*, 42(6-7):606–616, 2009. (Cited page 50.)

[26] Harish Doraiswamy and Vijay Natarajan. Computing Reeb Graphs as a Union of Contour Trees. *IEEE Trans. Vis. Comput. Graph.*, 19(2): 249–262, 2013. (Cited page 50.)

[27] Alejandro Duran, Eduard Ayguade, Rosa M. Badia, Jesus Labarta, Luis MArtinell, Xavier Martorell, and Judit Planas. OmpSs: A Proposal For Programming Heterogeneous Multi-core Architectures. *Parallel Processing Letters*, 21(02):173–193, 2011. doi: 10.1142/S0129626411000151. URL https://doi.org/10.1142/S0129626411000151. (Cited pages 33 and 57.)

[28] Edelsbrunner, Letscher, and Zomorodian. Topological persistence and simplification. *Discrete & Computational Geometry*, 28(4):511–533, Nov 2002. ISSN 1432-0444. doi: 10.1007/s00454-002-2885-2. URL https://doi.org/10.1007/s00454-002-2885-2. (Cited page 137.)

[29] H Edelsbrunner and J Harer. Computational topology: An Introduction. *American Mathematical Society*, 2009. (Cited pages 137, 139, and 147.)

[30] Herbert Edelsbrunner and Ernst P Mucke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM ToG*, 1990. (Cited pages 17 and 39.)

[31] Guillaume Favelier, Charles Gueunet, and Julien Tierny. Visualizing ensembles of viscous fingers. In *IEEE Scientific Visualization Contest*, 2016. (Cited pages 3, 55, 135, and 142.)

[32] Message Passing Interface Forum. *MPI: a Message-passing Interface Standard: Version 3.1*. High-Performance Computing Center, 2015. URL https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. (Cited pages 34 and 35.)

[33] Michael Fredman and Robert Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 1987. (Cited pages 27, 58, 80, 83, and 126.)

[34] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, Nov 1986. ISSN 1432-0541. doi: 10.1007/BF01840439. URL https://doi.org/10.1007/BF01840439. (Cited page 58.)

[35] GNU. C++ Standard Library, Parallel Mode. https://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html. (Cited pages 64 and 86.)

[36] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Task-based Augmented Merge Trees with Fibonacci heaps. In *LDAV*, 2017. (Cited pages 57, 76, and 101.)

[37] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Task-based Augmented Contour Trees with Fibonacci heaps. In *pending*, pending. (Cited pages 57, 76, and 99.)

[38] Charles Gueunet, Pierre Fortin, Julien Jomier, and Julien Tierny. Contour Forests: Fast Multi-threaded Augmented Contour Trees. In *IEEE Large Data Analysis and Visualization*, 2016. (Cited pages 56, 61, 77, 93, 94, 111, and 112.)

[39] David Gunther, Roberto A Boto, Julia Contreras-Garcia, Jean-Philip Piquemal, and Julien Tierny. Characterizing Molecular Interactions in Chemical Systems. 20:2476, 12 2014. (Cited page 3.)

[40] A. Gyulassy, V. Natarajan, V. Pascucci, and B. Hamann. Efficient Computation of Morse-Smale Complexes for Three-dimensional Scalar Functions. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1440–1447, Nov 2007. ISSN 1077-2626. doi: 10.1109/TVCG.2007.70552. (Cited pages 20 and 147.)

[41] A. Gyulassy, A. Knoll, K. C. Lau, B. Wang, P. Bremer, M. E. Papka, L. A. Curtiss, and V. Pascucci. Interstitial and Interlayer Ion Diffusion Geometry Extraction in Graphitic Nanosphere Battery Materials. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):916–925, Jan 2016. ISSN 1077-2626. doi: 10.1109/TVCG.2015.2467432. (Cited page 3.)

[42] David Günther, Joseph Salmon, and Julien Tierny. Mandatory Critical Points of 2D Uncertain Scalar Fields. *Computer Graphics Forum*, 33(3):

31–40. doi: 10.1111/cgf.12359. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12359. (Cited pages 3 and 55.)

[43] Mustafa Hajij and Paul Rosen. An Efficient Data Retrieval Parallel Reeb Graph Algorithm. 08 2018. (Cited pages 50 and 56.)

[44] Masaki Hilaga, Yoshihisa Shinagawa, Taku Kohmura, and Tosiyasu L. Kunii. Topology Matching for Fully Automatic Similarity Estimation of 3D Shapes. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 203–212, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X. doi: 10.1145/383259.383282. URL http://doi.acm.org/10.1145/383259.383282. (Cited pages 49 and 55.)

[45] Joseph F. JaJa. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992/// 1992. ISBN 0-201-54856-9. (Cited page 45.)

[46] J. Kasten, J. Reininghaus, I. Hotz, and H. Hege. Two-Dimensional Time-Dependent Vortex Regions Based on the Acceleration Magnitude. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2080–2087, Dec 2011. ISSN 1077-2626. doi: 10.1109/TVCG.2011.249. (Cited page 3.)

[47] T. L. Kunii and Y. Shinagawa. Constructing a Reeb graph automatically from cross sections. *IEEE Computer Graphics and Applications*, 11:44–51, 11 1991. ISSN 0272-1716. doi: 10.1109/38.103393. URL doi.ieeecomputersociety.org/10.1109/38.103393. (Cited page 49.)

[48] Daniel Laney, P-T Bremer, Ajith Mascarenhas, Paul Miller, and Valerio Pascucci. Understanding the structure of the turbulent mixing layer in hydrodynamic instabilities. *IEEE Transactions on Visualization & Computer Graphics*, (5):1053–1060, 2006. (Cited page 145.)

[49] Pawel Lapinski. *Vulkan Cookbook*. Packt Publishing, 2017. ISBN 1786468158, 9781786468154. (Cited page 34.)

[50] Jonas Lukasczyk, Ross Maciejewski, Christoph Garth, and Hans Hagen. Understanding hotspots: A topological visual analytics approach. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 36. ACM, 2015. (Cited page 55.)

[51] Jonas Lukasczyk, Garrett Aldrich, Michael Steptoe, Guillaume Favelier, Charles Gueunet, Julien Tierny, Ross Maciejewski, Bernd Hamann, and Heike Leitte. Viscous Fingering: A Topological Visual Analytic Approach. In *Physical Modeling for Virtual Manufacturing Systems and Processes*, volume 869 of *Applied Mechanics and Materials*, pages 9–19. Trans Tech Publications, 9 2017. doi: 10.4028/www.scientific.net/AMM.869.9. (Cited pages 3, 55, and 142.)

[52] Senthilnathan Maadasamy, Harish Doraiswamy, and Vijay Natarajan. A hybrid parallel algorithm for computing and tracking level set topology. pages 1–10, 12 2012. (Cited pages 44, 47, 48, 56, 64, 70, and 77.)

[53] Tim Mattson. A 'Hands-on' Introduction to OpenMP. `https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf`. (Cited pages 28, 29, and 30.)

[54] Kenneth Moreland, Christopher Sewell, William Usher, Li-Ta Lo, Jeremy Meredith, David Pugmire, James Kress, Hendrik Schroots, Kwan-Liu Ma, Hank Childs, Matthew Larsen, Chun-Ming Chen, Robert Maynard, and Berk Maynard. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. 36:48–58, 05 2016. (Cited page 58.)

[55] D. Morozov and G. Weber. Distributed Contour Trees. In *TopoInVis*, 2013. (Cited page 42.)

[56] D. Morozov and G. Weber. Distributed Merge Trees. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2013. (Cited page 42.)

[57] J. Nielsen. Power of 10: Time scales in user experience. 2009. `https://www.nngroup.com/articles/powers-of-10-time-scales-in-ux/`. (Cited page 55.)

[58] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2010. Version 3.2. (Cited page 34.)

[59] OpenMP Architecture Review Board. OpenMP Application Program Interface, V 4.5, 2015. (Cited pages 31, 33, 34, 57, 77, 86, and 89.)

[60] Salman Parsa. A deterministic $o(m \log m)$ Time Algorithm for the Reeb graph. *Discrete Comput. Geom.*, 49(4):864–878, June 2013. ISSN

0179-5376. doi: 10.1007/s00454-013-9511-3. URL https://doi.org/10.1007/s00454-013-9511-3. (Cited pages 51, 54, 57, 117, 118, 130, and 153.)

[61] V. Pascucci and K. Cole-McLaughlin. Parallel Computation of the Topology of Level Sets. *Algorithmica*, 2003. (Cited pages 43, 44, 47, 57, 63, and 64.)

[62] Valerio Pascucci, Kree Cole-McLaughlin, and Giorgio Scorzelli. Multi-Resolution Computation and Presentation of Contour Trees. 01 2004. (Cited page 55.)

[63] Valerio Pascucci, Giorgio Scorzelli, Peer-Timo Bremer, and Ajith Mascarenhas. Robust on-line computation of Reeb graphs: simplicity and speed. In *Acm transactions on graphics (tog)*, volume 26, page 58. ACM, 2007. (Cited pages 50, 66, and 67.)

[64] G. Patane, M. Spagnuolo, and B. Falcidieno. Reeb graph computation based on a minimal contouring. In *2008 IEEE International Conference on Shape Modeling and Applications*, pages 73–82, June 2008. doi: 10.1109/SMI.2008.4547953. (Cited pages 49 and 57.)

[65] Chuck Pheatt. Intel® Threading Building Blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, April 2008. ISSN 1937-4771. URL http://dl.acm.org/citation.cfm?id=1352079.1352134. (Cited pages 34 and 86.)

[66] Martin Poirier and Eric Paquette. Rig Retargeting for 3D Animation. In *Proceedings of Graphics Interface 2009*, GI '09, pages 103–110, Toronto, Ont., Canada, Canada, 2009. Canadian Information Processing Society. ISBN 978-1-56881-470-4. URL http://dl.acm.org/citation.cfm?id=1555880.1555907. (Cited pages 55 and 140.)

[67] Marzia Rivi, Luigi Calori, Giuseppa Muscianisi, and Vladimir Slavnic. In-situ visualization: State-of-the-art and some use cases. *PRACE White Paper*, pages 1–18, 2012. (Cited page 35.)

[68] Paul Rosen, Bei Wang, Anil Seth, Betsy Mills, Adam Ginsburg, Julia Kamenetzky, Jeff Kern, and Chris R. Johnson. Using Contour Trees in the Analysis and Visualization of Radio Astronomy Data Cubes. 04 2017. (Cited pages 3 and 55.)

[69] Dominic Schneider, Alexander Wiebel, Hamish Carr, Mario Hlawitschka, and Gerik Scheuermann. Interactive comparison of

scalar fields based on largest contours with applications to flow visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6), 2008. (Cited pages 3 and 55.)

[70] W. J. Schroeder, K. Martin, L. S. Avila, and C. C. Law. *The VTK user's guide*. Kitware, 2001. (Cited pages 58 and 154.)

[71] Nithin Shivashankar, Pratyush Pranav, Vijay Natarajan, Rien van de Weygaert, EG Patrick Bos, and Steven Rieder. Felix: A topology based framework for visual exploration of cosmic filaments. *IEEE Transactions on Visualization and Computer Graphics*, 22(6):1745–1759, 2016. (Cited page 3.)

[72] Dave Shreiner. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999. ISBN 0201657651. (Cited page 34.)

[73] J. Singler, P. Sanders, and F. Putze. The Multi-Core Standard Template Library. In *Euro-Par*, 2007. merged in STL since GCC 4.3. (Cited page 64.)

[74] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362 – 391, 1983. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(83)90006-5. URL `http://www.sciencedirect.com/science/article/pii/0022000083900065`. (Cited pages 27 and 126.)

[75] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting Binary Search Trees. *J. ACM*, 32(3):652–686, July 1985. ISSN 0004-5411. doi: 10.1145/3828.3835. URL `http://doi.acm.org/10.1145/3828.3835`. (Cited page 27.)

[76] B. . Sohn and Chandrajit Bajaj. Time-varying contour topology. *IEEE Transactions on Visualization and Computer Graphics*, 12(1):14–25, Jan 2006. ISSN 1077-2626. doi: 10.1109/TVCG.2006.16. (Cited pages 55 and 147.)

[77] Maxime Soler, Mélanie Plainchault, Bruno Conche, and Juilen Tierny. Lifted Wasserstein Matcher for Fast and Robust Topology Tracking. In *IEEE Symposium on Large Data Analysis and Visualization*, Berlin, Germany, October 2018. URL `https://hal.archives-ouvertes.fr/hal-01857913`. (Cited pages 3 and 55.)

[78] Thierry Sousbie. The persistent cosmic web and its filamentary structure–I. Theory and implementation. *Monthly Notices of the Royal Astronomical Society*, 414(1):350–383, 2011. (Cited page 3.)

[79] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12(3):66–73, May 2010. ISSN 0740-7475. doi: 10.1109/MCSE.2010.69. URL `http://dx.doi.org/10.1109/MCSE.2010.69`. (Cited page 34.)

[80] S. Tarasov and M. Vyali. Construction of contour trees in 3D in O(n log n) steps. In *SoCG*, 1998. (Cited pages 39 and 83.)

[81] Robert E. Tarjan and Jan van Leeuwen. Worst-case Analysis of Set Union Algorithms. *J. ACM*, 31(2):245–281, March 1984. ISSN 0004-5411. doi: 10.1145/62.2160. URL `http://doi.acm.org/10.1145/62.2160`. (Cited page 41.)

[82] Robert Endre Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM*, 22(2):215–225, April 1975. ISSN 0004-5411. doi: 10.1145/321879.321884. URL `http://doi.acm.org/10.1145/321879.321884`. (Cited page 26.)

[83] StarPU Doc Team. *StarPU 1.3 Reference Manual*. Samurai Media Limited, United Kingdom, 2017. ISBN 9789888407149, 9888407147. (Cited pages 33 and 57.)

[84] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, 74(4):1422–1434, Apr 2018. ISSN 1573-0484. doi: 10.1007/s11227-018-2238-4. URL `https://doi.org/10.1007/s11227-018-2238-4`. (Cited page 32.)

[85] J. Tierny and H. Carr. Jacobi Fiber Surfaces for Bivariate Reeb Space Computation. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):960–969, Jan 2017. ISSN 1077-2626. doi: 10.1109/TVCG.2016.2599017. (Cited page 3.)

[86] J. Tierny, A. Gyulassy, E. Simon, and V. Pascucci. Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees. *IEEE*

Transactions on Visualization and Computer Graphics, 15(6):1177–1184, Nov 2009. ISSN 1077-2626. doi: 10.1109/TVCG.2009.163. (Cited page 49.)

[87] Julien Tierny and Valerio Pascucci. Generalized topological simplification of scalar fields on surfaces. *IEEE Transactions on Visualization & Computer Graphics*, (12):2005–2013, 2012. (Cited pages 138 and 139.)

[88] Julien Tierny, Guillaume Favelier, Joshua A. Levine, Charles Gueunet, and Michael Michaux. The Topology ToolKit. *IEEE TVCG (Proc. of IEEE VIS)*, 2017. `https://topology-tool-kit.github.io/`. (Cited pages 58, 68, 81, 89, 94, 106, 126, 137, and 154.)

[89] P. Tsigas and Y. Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000. In *Conference on Parallel, Distributed and Network-based Processing*, 2003. (Cited page 64.)

[90] Will Usher and Qi Wu. Topology Guided Volume Exploration. Technical report, 2017. Accessed: 2018-07-13. (Cited pages 55 and 154.)

[91] Marc van Kreveld, René van Oostrum, Chandrajit Bajaj, Valerio Pascucci, and Dan Schikore. Contour Trees and Small Seed Sets for Isosurface Traversal. In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, SCG '97, pages 212–220, New York, NY, USA, 1997. ACM. ISBN 0-89791-878-9. doi: 10.1145/262839.269238. URL `http://doi.acm.org/10.1145/262839.269238`. (Cited pages 39 and 55.)

[92] Lei Wang, Quan Guo, Jianqiao Zhao, Shengnan Zhang, and Lisu Yang. The Fast Contour Tree-Based Medical Volume Rendering Method. *Journal of Medical Imaging and Health Informatics*, 8(7):1451–1455, 2018. (Cited page 154.)

[93] G. H. Weber, S. E. Dillard, H. Carr, V. Pascucci, and B. Hamann. Topology-Controlled Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):330–341, March 2007. ISSN 1077-2626. doi: 10.1109/TVCG.2007.47. (Cited page 55.)

[94] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC: First Experiences with Real-world Applications. In

Proceedings of the 18th International Conference on Parallel Processing, Euro-Par'12, pages 859–870, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-32819-0. doi: 10.1007/978-3-642-32820-6_85. URL `http://dx.doi.org/10.1007/978-3-642-32820-6_85`. (Cited page 34.)

[95] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, 2009. (Cited page 73.)

[96] Zoë J Wood, Mathieu Desbrun, Peter Schroder, and David Breen. Semi-regular mesh extraction from volumes. In *Visualization 2000. Proceedings*, pages 275–282. IEEE, 2000. (Cited page 49.)

Calcul Haute Performance pour l'Analyse Topologique de Données par Ensembles de Niveaux

L'analyse de données topologique nécessite des algorithmes de plus en plus efficaces pour être capable de traiter des jeux de données dont la taille et le niveau de détail augmente continûment. Dans cette thèse, nous nous concentrons sur trois abstractions topologiques fondamentales dérivées des ensembles de niveaux : l'arbre de jointure, l'arbre de contour et le graphe de Reeb. Nous proposons trois nouveaux algorithmes parallèles efficaces pour leur calcul sur des stations de travail composées de processeurs multi-cœur en mémoire partagée. Le premier algorithme élaboré durant cette thèse se base sur du parallélisme multi-thread pour le calcul de l'arbre de contour. Une seconde approche revisite l'algorithme séquentiel de référence pour le calcul de cette structure et se base sur des propagations locales exprimables en tâches parallèles. Ce nouvel algorithme est en pratique deux fois plus rapide en séquentiel que l'algorithme de référence élaboré en 2000 et offre une accélération d'un ordre de grandeur en parallèle. Un dernier algorithme basé sur une approche locale par tâches est également présenté pour une abstraction plus générique : le graphe de Reeb. Contrairement aux approches concurrentes, nos algorithmes construisent les versions augmentées de ces structures, permettant de supporter l'ensemble des applications pour l'analyse de données par ensembles de niveaux. Les méthodes présentées dans ce manuscrit ont donné lieu à des implémentations qui sont les plus rapides parmi celles disponibles pour le calcul de ces abstractions. Ce travail a été intégré à la bibliothèque libre : Topology Toolkit (TTK).

High Performance Level-set based Topological Data Analysis

Topological Data Analysis requires efficient algorithms to deal with the continuously increasing size and level of details of data sets. In this manuscript, we focus on three fundamental topological abstractions based on level sets: merge trees, contour trees and Reeb graphs. We propose three new efficient parallel algorithms for the computation of these abstractions on multi-core shared memory workstations. The first algorithm developed in the context of this thesis is based on multi-thread parallelism for the contour tree computation. A second algorithm revisits the reference sequential algorithm to compute this abstraction and is based on local propagations expressible as parallel tasks. This new algorithm is in practice twice faster in sequential than the reference algorithm designed in 2000 and offers one order of magnitude speedups in parallel. A last algorithm also relying on task-based local propagations is presented, computing a more generic abstraction: the Reeb graph. Contrary to concurrent approaches, these methods provide the augmented version of these structures, hence enabling the full extend of level-set based analysis. Algorithms presented in this manuscript result today in the fastest implementations available to compute these abstractions. This work has been integrated into the open-source platform: the Topology Toolkit (TTK).