DOCTORAL THESIS
SORBONNE UNIVERSITÉ
École Doctorale Informatique, Télécommunication et Électronique

Speciality: **COMPUTER SCIENCE**

Presented by

EVE LE GUILLOU

to obtain the degree of

Ph.D. of Sorbonne Université

Distributed Topological Data Analysis

Tentative date of defense: October 10th, 2025, before the committee:

David CŒURJOLLY	Senior Scientist	CNRS	**Reviewer**
Tom PETERKA	Senior Scientist	Argonne National Laboratory	**Reviewer**
Isabelle BLOCH	Professor	Sorbonne Université	**Examiner**
Bruno RAFFIN	Senior Scientist	Inria	**Examiner**
Federico IURICICH	Assistant Professor	Clemson University	**Examiner**
Christophe CALVIN	Senior Fellow	CEA	**Examiner**
Pierre FORTIN	Professor	University of Lille	**Co-advisor**
Julien TIERNY	Senior Scientist	CNRS	**Advisor**

SORBONNE UNIVERSITÉ
LIP6 – Laboratoire d'Informatique de Paris 6
UMR 7606 Sorbonne Université – CNRS
4 Place Jussieu – 75005 Paris

Mais que diable allait-il faire à cette galère ?
Les Fourberies de Scapin, Molière

Acknowledgments

Remerciements

The order of the names in a same group is generated randomly.
L'ordre des noms dans un même groupe est généré aléatoirement.

Communications

Journal Papers

TTK Is Getting MPI-Ready
Eve Le Guillou, Michael Will, Pierre Guillou, Jonas Lukasczyk, Pierre Fortin, Christoph Garth, Julien Tierny
IEEE Transactions on Visualization and Computer Graphics
Presented at IEEE VIS 2024

Distributed Discrete Morse Sandwich: Efficient Computation of Persistence Diagrams for Massive Scalar Data
Eve Le Guillou, Pierre Fortin, Julien Tierny
Submitted to IEEE Transactions on Parallel and Distributed Systems

Talks

COMPAS 2025
Presenting: Distributed Topological Data Analysis with TTK and MPI

IEEE VIS 2024
Presenting: TTK is Getting MPI-Ready

Journée Visualisation 2024
Presenting: TTK is Getting MPI-Ready

ParaView User Day Europe 2024
Presenting: TTK is Getting MPI-Ready

Contents

Notations

$\mathbb{X}$	Topological space
$\mathbb{M}$	Manifold
$\mathbb{R}^d$	Euclidean space of dimension d
σ, τ	Simplex and face of a simplex
$Lk(v)$, $Lk^-(v)$, $Lk^+(v)$	Link, lower link and upper link of a vertex v
$St(v)$, $St^-(v)$	Star and lower star of a vertex v
w	Isovalue
$f^{-1}(w)$	Level set of f at w
$\mathcal{K}$	Simplicial complex
$\mathcal{T}$	Triangulation
$\mathcal{M}$	Piecewise linear manifold
$f : \mathcal{M} \to \mathbb{R}$	Piecewise linear scalar field
$\mathcal{M}_p$	Piecewise linear manifold of process p
$\mathcal{M}'_p$	Ghosted piecewise linear manifold of process p
$\mathcal{M}^i$	i^{th} step of a filtration on $\mathcal{M}$
$\mathcal{Z}_p(\mathcal{K})$	Group of p-cycles of a simplicial complex $\mathcal{K}$
$\mathcal{B}_p(\mathcal{K})$	Group of p-boundaries of a simplicial complex $\mathcal{K}$
$\mathcal{H}_p(\mathcal{K})$	p^{th} homology group of a simplicial complex $\mathcal{K}$
$\beta_p(\mathcal{K})$	p^{th} Betti number of a simplicial complex $\mathcal{K}$
$\mathcal{L}^+(v)$, $\mathcal{L}^-(v)$	Forward and backward integral line starting in seed v
$\mathcal{V}$	Discrete vector field
$\mathcal{D}_d(f)$	d-dimensional persistence diagram of f
$\phi_d(j)$	Global identifier of the simplex σ_j of dimension d
ϕ_d	Global to local identifier map for d-simplices
ϕ_d^{-1}	Local to global identifier map for d-simplices
$\partial(\sigma)$	Boundary of a simplex
$\mathcal{G}_d$	Graph of sets for dimension d
$\mathcal{G}_{d,p}$, $\mathcal{G}'_{d,p}$	Local graph and ghosted local graph of sets for dimension d on process p

1 Introduction

With the rise of computers, the internet and more recently, AI, we have heard a lot about data: data science, data mining, Big data. Data has made its way into everyone's everyday life. In 2018, it even made its way into european law with the General Data Protection Regulation (GDPR). But what is data? According to the Cambridge Dictionary, data is "acts and numbers giving information about something". Data itself is not information, it holds information. It is useless if there is no way to extract the knowledge it contains. Methods and tools to perform analyses of data are flourishing, each fitting the needs of different problems. Specifically, *topological* methods aim at extracting structural characteristics of data in a concise representation to focus on the underlying information. Instead of concentrating on every single detail, topological methods enable to zoom out and consider a chain of mountains as an ensemble of peaks and valleys rather than an ensemble of rocks.

However, when processing massive datasets, the data may become too large to fit in the memory of a single computer or too large to be processed in a reasonable time-frame. A solution is to turn to high performance environments and supercomputers, i.e. large computers composed of up to thousands of smaller computers, called nodes, connected through a network. Computations can then be made in parallel on several nodes at once. Parallelism can occur both within a single node, where multiple processor cores access a common memory (known as *shared-memory* parallelism), and across multiple nodes, where each node has its own separate memory. In the latter case, the memory is said to be *distributed*. Memory and computing power are no longer a problem and much larger datasets can be analyzed. However, in a distributed-memory setting, each node has its own separate memory. The existing tools and methods need to be reworked to add exchanges between the nodes so the algorithms can per-

form correctly. This thesis builds upon established topological techniques to develop new approaches tailored to distributed-memory computations.

1.1 General Context and Motivations

1.1.1 Data Acquisition, Analysis and Visualization

In scientific computation, data comes from two main sources: data acquisition and data simulation. Data acquisition refers to the act of measuring physical quantities that characterize real world phenomena and translating it to digital values understandable by a computer. These physical quantities may include temperature, pressure, or vibration. The range of phenomena that can be captured is as broad as the variety of available sensors, encompassing everything from medical imaging to sound recording. Sensors technology improves over time to capture more and more details and precision, inducing the growth of the produced datasets.

Data simulation is favored when replicating real-world scenarios is more practical or effective than directly capturing them. It can be due to a number of reasons. For instance, in the study of aerodynamics, simulating the aerodynamics of a model aircraft is significantly more cost-effective than constructing and testing a real one. In cosmology, capturing the entire universe is impossible, making simulation the only viable option. Moreover, simulations are invaluable for predicting future events, such as weather patterns in meteorology. The precision of simulations is heavily dependent on the complexity of the model and the available computation resources. As computational systems have continuously improved, so have the datasets produced by simulations.

As datasets grew, so too did the need for complex and efficient analyses methods to extract the information hidden within the data. Data visualization is at the interface of data analysis and computer graphics and relies on visual representation of the raw data to facilitate its exploration. Specifically, scientific visualization aims at representing scientific phenomena.

1.1.2 Topological Data Analysis

Topological Data Analysis (TDA) [EH09] apprehends the complexity brought by massive data by providing concise encoding of the core patterns in the data, to facilitate its analysis and visualization. It is based on robust, multi-scale algorithms [ELZ02], which capture a variety of structural features [HLH+16]. Examples of applications

include combustion [LBM+06, BWT+11, GBG+14], material sciences [GKL+16, FGT16, SPD+19], nuclear energy [MWR+16], fluid dynamics [KRHH11, NVBB+22], bioimaging [CSvdP04, BDSS18], data science [CGOS13, DTS+20], quantum chemistry [GABCG+14, BGL+18, OGT19, OT23] and astrophysics [Sou11, SPN+16]. In particular, the persistence diagram is a concise and robust encoding of the topological features of a dataset. Several algorithms have been conceived for its computation, the current most efficient method being the Discrete Morse Sandwich [GVT23]. However, the construction of this diagram is quite costly, both in time and memory footprint.

With the above data size increase, it becomes frequent in the applications that the size of a single dataset exceeds the memory capacity of a single computer, hence requiring to consider distributed-memory systems, whose combined memory provides much larger capacities.

1.1.3 The TORI Project

In a world where data is constantly growing, having methods and tools to analyze such volumes of data efficiently is of critical importance. The TORI project (*TOpological Reduction of Information* [1], also referred to as *Insitu Topological Reduction of Scientific 3D Data* [2]) aims at addressing this issue by developing the next generation of data reduction tools using Topological Data Analysis. The new tools and approaches developed during this project are integrated in the Topology ToolKit (TTK) [TFL+17], a library for topological analysis and visualization (see Section 2.3.2). TORI revolves around two main axes: *(i)* developing new methods for the statistical analysis of collections of topological signatures, and *(ii)* designing new approaches capable of carrying out these analyses on large-scale datasets in a high-performance environment. This thesis is focused on the second aspect: building on existing tools and approaches to perform these analyses on large-scale datasets within an acceptable time-frame.

1.2 Problem formulation

In this thesis, we try to tackle several issues relative to distributed topological data analysis. In the following subsections, we specify with more precision the different axes of our work.

[1] https://erc-tori.github.io/

[2] https://cordis.europa.eu/project/id/863464

1.2.1 Distributed-memory Topological Data Analysis

Adding distributed-memory support to an algorithm requires making significant changes to the procedure. Firstly, to ensure that the execution is correct and provides the right results. Secondly, to try and provide the best performance possible, in term of both memory footprint and speed of execution. The changes include exchanges of data or synchronizations between nodes. These modifications to the algorithm induce additional work that can slow down the overall execution, therefore, careful evaluation is essential during redesign to keep these costs to a minimum. TDA algorithms have their own specificities that tend to hinder efficiency in a distributed-memory setting. Indeed, TDA aims at extracting global features, whereas by definition, in a distributed-memory setting, no node has access to the global data. TDA algorithms also tend to require multiple global data traversals and little computation. This combination is difficult to scale efficiently.

1.2.2 Mono-tailored distributed implementations

Existing distributed-memory implementations are mono-tailored for one particular topological representation. This induces several drawbacks that hinder a wider adoption of topological methods. Indeed, it makes it harder to insert in existing workflows. Such implementations often offer support for a limited number of data formats as input. Furthermore, a lot of TDA algorithms do not provide a public implementation. This significantly limits the practical usability of topological methods. TTK aims at providing a unified framework for TDA algorithms with a reusable and efficient data structure [TFL+17], however prior to this work, it was limited to the computation on one node.

1.2.3 Persistence diagram computation

The persistence diagram, see Figure 2.10, is one of the most used topological representation. This can be explained by its mathematical properties that make it a very robust, reliable and simple descriptor of data, with applications such as feature tracking [LGW+19, SPD+19, SPCT18a] and ensemble summarization [VBT20, KVT19, FFST18]. Several algorithms exist to compute its data structure. In a distributed-memory setting, there is only one publicly available implementation: *Dipha* [BKR14b]. Currently, the most efficient algorithm on a single node is the Discrete Morse Sand-

wich (DMS) [GVT23], introduced by Guillou et al. This algorithm, however, is limited to the computation on only one node.

1.3 Contributions

In this thesis, we aim at providing new approaches and tools for distributed Topological Data Analysis computation. This contribution can be broken down in two axes, which we will present in this section.

1.3.1 A unified framework for distributed Topological Data Analysis

We build upon the existing TTK environment to provide a unified framework for the distributed-memory computation of TDA. Specifically, we add distributed support to TTK using the *Message Passing Interface* (MPI), today's most popular solution for distributed-memory computations. We modify TTK's core data structure to make it both usable and practical in a distributed-memory context. Additional low and high level features are added to facilitate developments of future distributed-memory algorithms. Distributed-memory support is added to several existing algorithms. These examples are used to both demonstrate how to use the new features and showcase the performance of our new approaches. In an effort to help future development, a taxonomy of algorithms is provided to categorize the algorithms based on their needs for exchanges between nodes. Performance tests in different scenarios showcased the efficiency of each algorithm as well as the low overhead of the overall software infrastructure. Finally, a real-life use case of topological analysis is applied to two massive datasets to exhibit the proper functioning of our software.

1.3.2 Distributed computation of the persistence diagram

After setting up a helpful software environment for distributed-memory computations of topological methods, we focus our efforts on one particular topological representation: the persistence diagram, and more specifically: the Discrete Morse Sandwich algorithm. This approach is much more complex to modify than the procedures of the previous contribution as parts of DMS rely on a sequential execution. Our new method, the Distributed Discrete Morse Sandwich (DDMS), builds upon DMS and introduces step-specific modifications tailored to the needs of each phase of the algorithm. The multi-core parallelism of the original DMS is preserved and extended, resulting in a hybrid MPI+thread implementation.

Extensive performance tests showcase the efficiency of our approach and demonstrates its gain over the original DMS method as well as *Dipha*, the reference method for persistence diagram computation in a distributed-memory context. Our new algorithm is able to compute the persistence diagram of datasets of up to 6 billion vertices.

1.4 Outline

The rest of this manuscript is organized as follows:

- In Chapter 2, we present the theoretical basis of Topological Data Analysis upon which our work is based. We also provide an introduction to Parallel Computing and present the existing software environments and state-of-the-art tools for high performance Topological Data Analysis.
- In Chapter 3, we present our unified software framework for the distributed-memory computation of topological analysis pipelines.
- In Chapter 4, we describe a new efficient method for computing the persistence diagram in a distributed-memory setting based on the existing Discrete Morse Sandwich algorithm.
- Finally, we summarize our work in Chapter 5 and discuss current limitations as well as open problems.

Foundations

2

Contents

This chapter introduces the basis of Topological Data Analysis and Parallel Computing that our work is based on as well as the existing visualization softwares. First, we formalize the representation of our input data. We then introduce several topological representations useful for our work such as *critical points* and *persistence diagrams*. In a second part, we introduce different types of parallelism, such as shared-memory and

distributed-memory parallelism and we position ourselves within that setting. Finally, we present an overview of the existing visualization software environments. We also present the Topology ToolKit (TTK) [TFL+17], the software framework our work contributes to.

This chapter, particularly its section regarding Topological Data Analysis, contains definitions adapted from [EH09], [Tie18] as well as Jules Vidal's [Vid21], Mathieu Pont's [Pon23] and Charles Gueunet's thesis [Gue19]. We refer the reader to the reference book [EH09] for a more detailed introduction to computational topology.

2.1 Theoretical background on topology

2.1.1 Input Data Representation

In the field of scientific visualization, scalar data is typically defined over a geometric structure, oftentimes called a mesh or a grid. In our applications, the input grid is either two- or three-dimensional. In practice, a computer needs a finite numbers of values and therefore this input is discretized. In this manuscript, we consider piecewise linear (PL) manifolds. It can be intuitively defined as a locally smooth topological space discretized using small building blocks called simplices such as triangles and tetrahedra. The section hereafter formalizes these terms and describes several topological objects fundamental to our work.

2.1.1.1 Domain Representation

The domain is the geometric object on which the input data is defined. The following definitions iteratively formalize the domain to produce the object we will work with for the remainder of the manuscript: the piecewise linear manifold. We start by defining topological spaces and manifolds.

Definition 2.1 *(Topological Space, Topology, Open Sets)* A set $\mathbb{X}$ is called a *topological space* if there exists a collection T of subsets of $\mathbb{X}$ such that:

- The empty set $\emptyset$ and $\mathbb{X}$ itself belong to T.
- Any union of elements of T belongs to T.
- Any finite intersection of elements of T belongs to T.

T is said to be a *topology* of $\mathbb{X}$. Its elements are the *open sets* of $\mathbb{X}$.

Definition 2.2 *(Homeomorphism)* A function $f : \mathbb{X}_1 \to \mathbb{X}_2$ is a homeomorphism if it is a continuous bijection and if its inverse f^{-1} is also continuous. $\mathbb{X}_1$ and $\mathbb{X}_2$ are said to be homeomorphic.

Definition 2.3 *(Manifold)* A topological space $\mathbb{M}$ is a d-manifold if every element $m \in \mathbb{M}$ has an open neighborhood $\mathbb{N}$ homeomorphic to an open neighborhood of $\mathbb{R}^d$

Intuitively, a manifold is a topological space for which, if zoomed in enough, every area resembles an open Euclidean d-ball. The complete geometry of the domain can turn out to be much more complex but when inspected in detail, smaller areas are somewhat smooth and easily described.

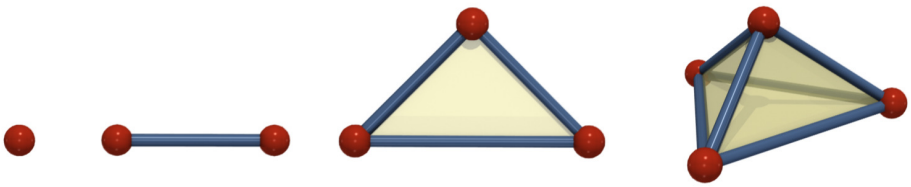


Figure 2.1 – *Illustration of simplices, respectively from left to right:* 0 *(a vertex),* 1 *(an edge),* 2 *(a triangle) and* 3 *(a tetrahedron).*

Definition 2.4 *(Convex set)* A set $\mathbb{C}$ of an Euclidean space $\mathbb{R}^n$ of dimension n is convex if for any two points x, y in $\mathbb{C}$ and all $t \in [0,1]$, the point $(1-t)x + ty$ is also in $\mathbb{C}$.

Definition 2.5 *(Convex hull)* The convex hull of a set of points $\mathbb{P}$ in an Euclidean space $\mathbb{R}^n$ is the unique minimal convex set containing all points of $\mathbb{P}$.

Definition 2.6 *(Simplex)* A d-simplex is the convex hull of $d+1$ affinely independent points of an Euclidean space $\mathbb{R}^n$, with $0 \leq d \leq n$. d is the dimension of the simplex. In our applications, with $d \leq 3$, the simplices are defined as follows (see Figure 2.1):

- A 0-simplex is a vertex.
- A 1-simplex is an edge.
- A 2-simplex is a triangle.
- A 3-simplex is a tetrahedron.

Simplices can be seen as the smallest building blocks that can be used to represent the input domain. The relationship between them can be expressed using the concept of *face*.

Definition 2.7 *(Face)* A face τ of a d-simplex σ is the simplex defined by a non-empty subset of the $d+1$ points of σ, and is noted $\tau \leq \sigma$. σ is called the co-face of τ.

The faces of a tetrahedron are its four triangles, its six edges and four vertices. Now that this building blocks are defined, the natural next step is to define the input domain as a combination of simplices: this is a *simplicial complex*.

Definition 2.8 *(Simplicial Complex)* A simplicial complex $\mathcal{K}$ is a finite collection of non-empty simplices σ_i, such that every face of a simplex in $\mathcal{K}$ is also in $\mathcal{K}$, and any two simplices intersect in a common face or not at all.

Figure 2.2 – *Example of piecewise linear manifold of dimension 3, based on an unstructured (or irregular) grid. On the left figure, the triangles making up the surface of the manifold are visible. On the right figure, the cut in the domain reveals its interior made of tetrahedra.*

Definition 2.9 *(Triangulation)* A triangulation $\mathcal{T}$ of a topological space $\mathbb{X}$ is a simplicial complex $\mathcal{K}$ such that the union of its simplices is homeomorphic to $\mathbb{X}$.

The triangulation is the preferred representation of grids and meshes because all grids can easily be turned into a triangulation by subdividing cells into simplices. However, in practice, the following notion is used as it is a bit more restrictive and illustrated here Figure 2.2.

Definition 2.10 *(Piecewise linear manifold)* A piecewise linear (PL) manifold $\mathbb{M}$ is the triangulation of a manifold $\mathbb{M}$.

2.1.1.2 Scalar Field Representation

The domain of our input has been defined: it is a piecewise linear manifold. However, it is not the subject of interest of our analysis, the scalar data is, in the form of a univariate scalar field. There needs to be a mapping between our domain geometrical structure composed of simplices and the scalar field defined on vertices. This leads to the definition of *barycentric coordinates*, that allow to define any d-simplex σ as a linear combination of 0-simplices.

Definition 2.11 *(Barycentric Coordinates)* Let p be a point of $\mathbb{R}^n$ and σ a d-simplex. Let $\alpha_0, \ldots, \alpha_d$ be a set of real coefficients such that $p = \sum_{i=0}^{d} \alpha_i v_i$, (where $v_0, \ldots, v_d$ are the 0-simplices face of σ) and such that $\sum_{i=0}^{d} \alpha_i = 1$. Such coefficients are called the barycentric coordinates of p relatively to σ.

Figure 2.3 – *The figure on the left shows a piecewise linear manifold in the form of a pegasus on which is defined a scalar field corresponding to the elevation of the vertices. On the middle figure is shown a level set of this manifold (highlighted contour). The figure on the right depicts the sub-level set of this manifold for the same isovalue.*

Definition 2.12 *(Piecewise Linear Scalar Field)* Let a triangulation $\mathcal{T}$ and a function h that maps the vertices of $\mathcal{T}$ to $\mathbb{R}$. A piecewise linear (PL) scalar field f on $\mathcal{T}$ is a function that maps any point p of a d-simplex σ of $\mathcal{T}$ to a value $f(p) = \sum_{i=0}^{d} \alpha_i h(v_i)$ with $\alpha_0, \ldots, \alpha_d$ the barycentric coordinates of p relatively to σ and $v_0, \ldots, v_d$ the 0-simplices of σ. f is linearly interpolated from h on σ.

In the rest of the manuscript, the scalar data will be a piecewise linear scalar field. The input field is typically defined on the vertices of a PL manifold and is interpolated to obtain its value for simplices of higher dimension. Furthermore, we will only consider PL scalar fields that are injective on our domain (i.e. $\forall v_0 \neq v_1 \in \mathcal{M}, f(v_0) \neq f(v_1)$). In pratice, this is easily achieved by substituting the f value of a vertex by its position in the vertex order (by increasing f values), a practice inspired from *Simulation of Simplicity* [EM90]. This will be useful later in subsubsection 2.1.3.1 when defining the notion of *filtration*.

Definition 2.13 *(Level set)* Let $\mathcal{M}$ be a piecewise linear manifold. The level set $f^{-1}(w)$ of an isovalue $w \in \mathbb{R}$ relatively to a scalar field $f : \mathcal{M} \to \mathbb{R}$ is the pre-image of w onto $\mathcal{M}$ through f : $f^{-1}(w) = \{p \in \mathcal{M} | f(p) = w\}$.

Definition 2.14 *(Sub-Level set)* The sub-level set of an isovalue $w \in \mathbb{R}$ relatively to a PL scalar field $f : \mathcal{M} \to \mathbb{R}$ is the set of points: $\{p \in \mathcal{M} | f(p) \leq w\}$.

In other words, the level set of w for the scalar field f is the set of points p of $\mathcal{M}$ for which $f(p) = w$ and the sub-level set of of w for the scalar field f corresponds to the set of points p for which $f(p) \leq w$. Level sets

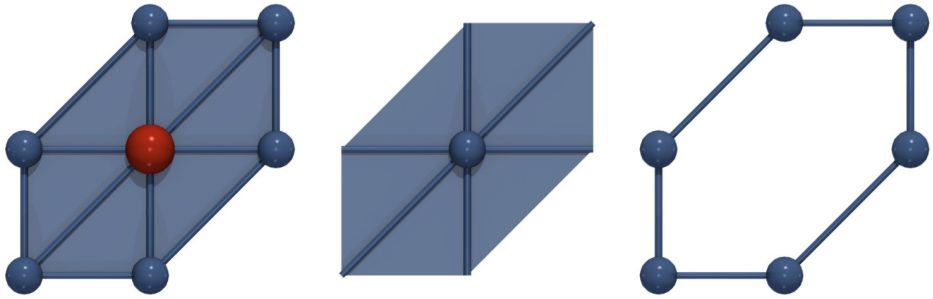


Figure 2.4 – *Illustration of the star and the link of a vertex v (in red) for a two-dimensional PL manifold. The figure on the left shows the simplices in the neighborhood of v. The middle figure represents the star $St(v)$ of v and the figure on the right represents the link $Lk(v)$ of v.*

and sub-level sets are illustrated in Figure 2.3. The notion of sub-level set is instrumental to TDA as it studies the changes in topology of an input data set for various sub-level sets.

2.1.1.3 Related neighborhood definitions

In this subsection, we formally introduce several geometrical constructions to navigate the neighborhoods of vertices within a manifold, namely the *star* and *link* of a simplex, illustrated in Figure 2.4. These notions will be useful for the descriptions of topological representations defined in the next sections.

Definition 2.15 *(Star)* The star of a simplex σ of a simplicial complex $\mathcal{K}$, noted $St(\sigma)$, is the set of simplices of $\mathcal{K}$ that contain σ: $St(\sigma) = \{\sigma' \in \mathcal{K}, \sigma \leq \sigma'\}$.

Definition 2.16 *(Lower Star)* The lower star $St^-(v)$ of a vertex v of a simplicial complex $\mathcal{K}$ relatively to a PL scalar field $f : \mathcal{K} \to \mathbb{R}$ is the subset of the simplices in the star of v whose vertices have lower value than v, defined as $St^-(v) = \{\sigma \in St(v) | \forall u \in \sigma, f(u) \leq f(v)\}$.

Definition 2.17 *(Link)* The link of σ is the set of faces of the simplices of $St(\sigma)$ that are disjoint from σ: $Lk(\sigma) = \{\sigma' \leq \tau, \tau \in St(\sigma), \sigma' \cap \sigma = \emptyset\}$.

Intuitively, the link of a vertex can be viewed as the boundary of its star.

Definition 2.18 *(Lower and Upper Link)* The lower link $Lk^-(v)$ (respectively the upper link $Lk^+(v)$) of a vertex v given a PL scalar field f is the subset of the simplices of the link $Lk(v)$ whose vertices have a strictly lower (respectively higher) function value than v. It is defined for the lower link as $Lk^-(v) = \{\sigma \in$

$Lk(v)|\forall v' \in \sigma, f(v') < f(v)\}$ (respectively $Lk^+(v) = \{\sigma \in Lk(v)|\forall v' \in \sigma, f(v') > f(v)\}$)

2.1.1.4 Topological invariants

Now that our input has been precisely defined, a natural question to ask is how to compare those inputs. Topological invariants are structural descriptors of a space that are preserved under specific types of geometrical deformations (e.g. homeomorphisms). They are useful to compare the topology of two spaces while not focusing on the detailed geometry of the two spaces. An example of topological invariants is the number of connected components.

Definition 2.19 *(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 2.20 *(Connected components)* The largest connected subsets of a topological space are called its connected components.

The notion of connected components allows to describe very succinctly a space and compare it to another space. Naturally, a lot of topological information is lost when relying solely on this descriptor. To go further while still preserving very efficient descriptors, one can look into *homology*.

Homology is a framework used to describe input data by inspecting how a space is connected. It tracks the holes of a domain, which equals, among other things, to its connected components. The homology of a space is described by its *Betti numbers* $\beta_0, \beta_1, \ldots, \beta_d \in \mathbb{N}$. In low dimensions, Betti numbers are accompanied with a very concrete intuition: β_0 is the number of connected components, β_1 is the number of handles and β_2 is the number of voids. Connected components, handles, and voids are all viewed as "holes" in homology. Holes of a domain are robust and efficient topological invariants, useful to compare spaces.

Figure 2.5 illustrates the use of Betti numbers to compare spaces. The ball and the glass have the same number of connected components (one), the same number of handles (none) and the same number of void (none). To understand what it means to say that these two spaces are unaffected by homeomorphism, one can think of the ball as a ball of clay. If it is possible to sculpt this ball of clay into the other shape without creating or deleting connected components, handles or voids, then the two spaces indeed are homeomorphic. It is not, however, possible to sculpt the ball into the mug, as the handle of the mug is a topological handle. These two

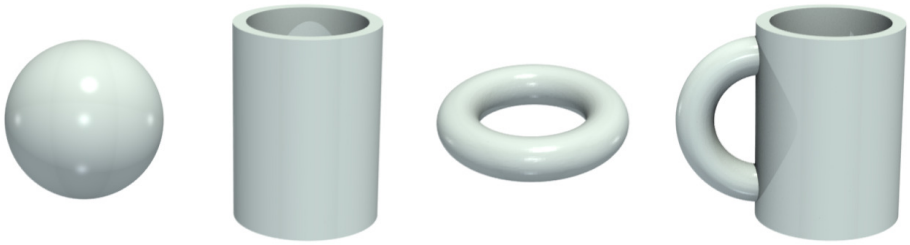


Figure 2.5 – *Manifolds of different Betti numbers. The ball has one connected component ($\beta_0 = 1$), no handle ($\beta_1 = 0$) and no void ($\beta_2 = 0$). It is homeomorphic to the glass, but not to the solid torus ($\beta_0 = 1$, $\beta_1 = 1$ and $\beta_2 = 0$) that possesses a handle. The solid torus is homeomorphic to the mug, that also has a handle.*

spaces therefore have different homology invariants. The mug yields the same Betti numbers as the solid torus, that also has a handle.

Homology is not limited to the study of holes within the whole domain. A very common technique is the study of the evolution of holes on a series of sub-level sets. The domain is characterized by increasing the isovalue of the sub-level set and studying where "holes" appear or disappear. This ensemble of sub-level sets can be formalized into what is called a *filtration*.

2.1.1.5 Filtration

Definition 2.21 *(Filtration)* Let f be an injective scalar field defined on a simplicial complex $\mathcal{M}$, such that $f(\tau) < f(\sigma)$ for each face τ of each $\sigma \in \mathcal{M}$. Let n be the number of simplices of $\mathcal{M}$ and $\mathcal{M}^i$ be the sub-level set of f by the i^{th} value in the sorted set of simplices values. The nested sequence of sub-complexes $\mathcal{M}^0 = \emptyset \subset \mathcal{M}^1 \subset \cdots \subset \mathcal{M}^n = \mathcal{M}$ is called the filtration of f.

Intuitively, a filtration can be seen as a scan of a simplicial complex. All its simplices are sorted in increasing order and added one by one to the current simplicial sub-complex, sweeping through all the simplices. As illustrated in Figure 2.6, the topology of a filtration, and its Betti numbers, will change in specific vertices, called *critical points*. In practice, in this manuscript, the comparison of two simplices is made by listing the vertex orders of both simplices by decreasing values, and by comparing the resulting lists with *lexicographic* comparison. The associated filtration is called the *lexicographic filtration*.

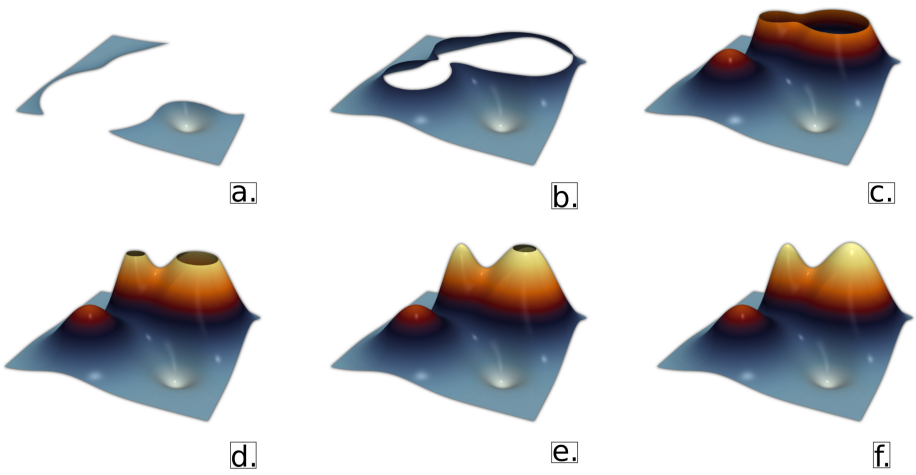


Figure 2.6 – *Example of filtration on a scalar field representing the elevation on a terrain. As the next simplex of lowest scalar value is added to the filtration, the topology of the domain changes: in (a), the terrain is composed of two connected components ($\beta_0 = 2$). In (b), only one connected component remains, but a handle has appeared ($\beta_1 = 1$). This handle continues in (c) ($\beta_1 = 1$). In (d), the handle has been divided in two ($\beta_1 = 2$). In (e), only one handle is left ($\beta_1 = 1$). Finally, in (f), the whole domain has been swept through, no handle is left ($\beta_1 = 0$).*

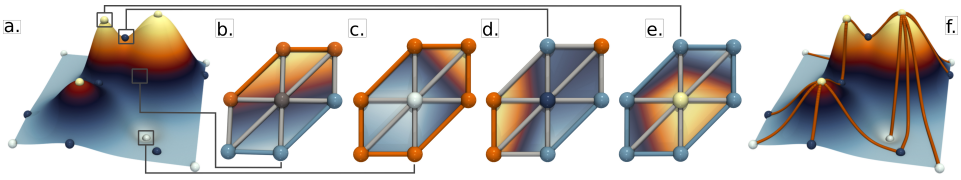


Figure 2.7 – *Example of critical points and integral lines on a toy example (elevation f on a terrain $\mathcal{M}$,* (a)*). The vertices of $\mathcal{M}$ can be classified based on their star into regular vertices* (b)*, local minima* (c)*, saddle points* (d) *or local maxima* (e)*. Integral lines (orange curves,* (f)*) are curves which are tangential to the gradient of f.*

2.1.2 Basic topological abstractions

In this subsection, we will define several topological abstractions useful in the rest of this manuscript using the topological notions defined earlier.

2.1.2.1 Critical Points

Though we first described critical points by relying on the evolution of a filtration, they can also be defined by inspection of their local neighborhoods, as formalized by Banchoff in [Ban67].

Definition 2.22 *(Critical Point)* Let a PL scalar field $f : \mathcal{M} \in \mathbb{R}$ defined on a PL manifold $\mathcal{M}$. A vertex v is *regular* if and only if both $Lk^-(v)$ and $Lk^+(v)$ are simply connected. Otherwise, v is a *critical vertex* of f.

Definition 2.23 *(Extremum)* Let a PL scalar field $f : \mathcal{M} \in \mathbb{R}$ defined on a PL manifold $\mathcal{M}$. A critical point v is a minimum (respectively a maximum) of f if and only if $Lk^-(v)$ (respectively $Lk^+(v)$) is empty.

Definition 2.24 *(Saddle)* Let a PL scalar field $f : \mathcal{M} \in \mathbb{R}$ defined on a PL manifold $\mathcal{M}$.A critical point v is a saddle if and only if it is neither a minimum nor a maximum of f.

Figure 2.7*(a-e)* illustrates the different types of critical points and their associated lower and upper links on a toy example representing the elevation f of a terrain $\mathcal{M}$. A critical vertex v can be classified by its *index* $\mathcal{I}(v)$, which is 0 for minima (Figure 2.7*(c)*), 1 for 1-saddles (Figure 2.7*(d)*), $(d-1)$ for $(d-1)$-saddles and d for maxima (Figure 2.7*(e)*). Vertices for which the number of connected components of $Lk^-(v)$ or $Lk^+(v)$ are greater than 2 are called *degenerate saddles*.

2.1.2.2 Integral Lines

Integral lines are curves on $\mathcal{M}$ which locally describe the gradient of f (orange curves in Figure 2.7*(f)*). They can be used to capture and visualize adjacency relations between critical points. The starting vertex of an integral line is called a *seed*.

Definition 2.25 *(Forward integral line)* Given a seed v, its *forward* integral line, noted $\mathcal{L}^+(v)$, is a path along the edges of $\mathcal{M}$, initiated in v, such that each edge of $\mathcal{L}^+(v)$ connects a vertex v' to its highest neighbor v''.

When encountering a saddle s, we say that an integral line *forks*: it yields one new integral line per connected component of $Lk^+(s)$. Integral lines can *merge* (and possibly fork later). A *backward* integral line, noted $\mathcal{L}^-(v)$, is defined symmetrically (i.e. integrating downwards).

2.1.2.3 Discrete Gradient

In recent years, an alternative emerged to the PL formalism of critical points described above (subsubsection 2.1.2.1), namely Discrete Morse Theory (DMT) [For98]. This formalism implicitly resolves several challenging configurations (such as degenerate saddles on manifold domains), which has been particularly useful for the development of ro-

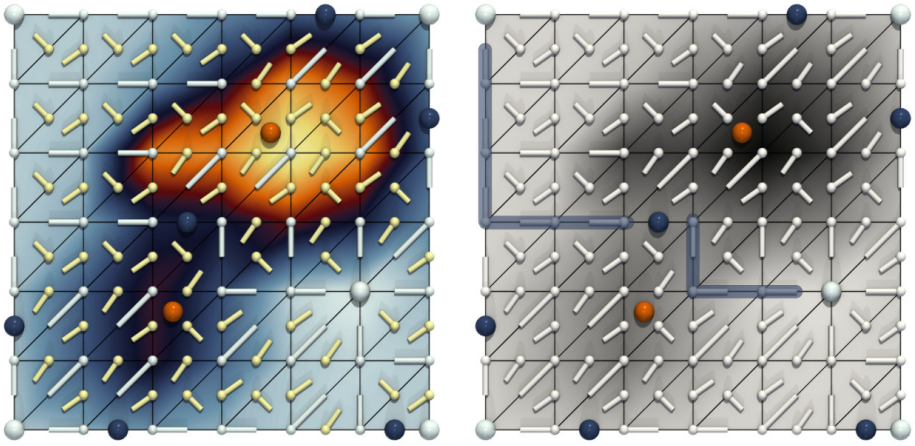


Figure 2.8 – *Example of discrete gradient field for a given scalar field (left). The larger spheres represent critical simplices. The light yellow arrows are edge-triangle pairs. The light blue arrows are vertex-edge pairs. Two examples of v-paths are shown in transparent blue (right), going from a critical edge and ending in critical vertices. These form the unstable set of the critical edge.*

bust algorithms in the context of Morse-Smale complex computation [RWS11, GBP19].

Definition 2.26 *(Discrete Vector)* A *discrete vector* is a pair formed by a simplex $\sigma_i \in \mathcal{M}$ (of dimension i) and one of its co-facets σ_{i+1} (i.e. one of its co-faces of dimension $i+1$), noted $\{\sigma_i < \sigma_{i+1}\}$.

In Figure 2.8 (left), the discrete vectors are the small light blue and light yellow arrows. σ_{i+1} is usually referred to as the *head* of the vector (represented with a small cylinder in Figure 2.8), while σ_i is its *tail* (represented with a small sphere in Figure 2.8). Examples of discrete vectors include a pair between a vertex and one of its incident edges, or a pair between an edge and a triangle containing it.

Definition 2.27 *(A discrete vector field)* A *discrete vector field* on $\mathcal{M}$ is then defined as a collection $\mathcal{V}$ of pairs $\{\sigma_i < \sigma_{i+1}\}$, such that each simplex of $\mathcal{M}$ is involved in at most one pair.

This setting yields a new definition for a critical point, here referred to as a *critical simplex*.

Definition 2.28 *(Critical simplex)* A simplex σ_i which is involved in no discrete vector $\mathcal{V}$ is called a *critical simplex*.

The dimension of the critical simplex corresponds to its index in the

smooth setting [Mil63, Mor34]. A critical 0-simplex (or vertex) is called a minimum, a 1-simplex (or edge) a 1-saddle, a 2-simplex (or triangle) a 2-saddle and a 3-simplex (or tetrahedron) a maximum. In Figure 2.8, critical simplices are represented by larger spheres. Similarly to the critical points defined by Banchoff, critical simplices can be linked by integral lines, called *v-paths* (see Figure 2.8, right).

Definition 2.29 *(V-path)* A *v-path*, or *discrete integral line*, is a sequence of discrete vectors $\{\{\sigma_i^0 < \sigma_{i+1}^0\}, \ldots, \{\sigma_i^k < \sigma_{i+1}^k\}\}$, such that *(i)* $\sigma_i^j \neq \sigma_i^{j+1}$ (i.e. the tails of two consecutive vectors are distinct) and *(ii)* $\sigma_i^{j+1} < \sigma_{i+1}^j$ (i.e. the tail of a vector in the sequence is a face of the head of the previous vector), for any $0 < j < k$.

Definition 2.30 *(Discrete stable and unstable sets)* The collection of all the discrete integral lines terminating in a given critical simplex σ_i is called the discrete stable set of σ_i. Symmetrically, the collection of all the discrete integral lines starting at a given critical simplex σ_i is called the discrete unstable set of σ_i.

Definition 2.31 *(Discrete gradient field)* A *discrete gradient field* is then a discrete vector field such that all its possible *v-paths* are loop-free.

Intuitively, this means that all critical simplices can be connected by following discrete vectors without any loops in the paths. Several algorithms have been proposed to compute such a discrete gradient field from an input PL scalar field. We consider in this work the algorithm by Robins et al. [RWS11], given its proximity to the PL setting: each critical cell identified by this algorithm is guaranteed to be located in the star of a PL critical vertex (subsubsection 2.1.2.1). In practice the computation is performed through inspection of the local neighborhood of each vertex, which makes this step embarrassingly parallel.

2.1.3 Persistent homology

Critical points are a very powerful tool as they can be easily computed using only the data of a neighborhood of a vertex. However, they have their limitations, particularly in a real word setting, where the input data is often very noisy. Every single small function undulation will create a critical point, making the overall result difficult to read as it is impossible to distinguish noise from actual features of the data. A solution to this problem is *persistent homology*: this formalism introduces the notion of *persistence*. It

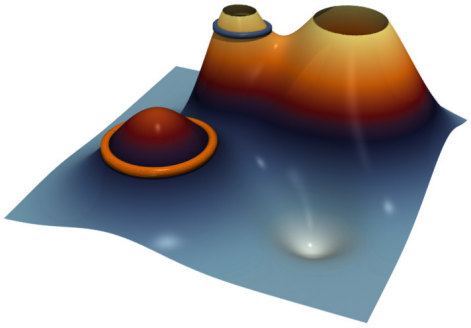


Figure 2.9 – *Examples of* 1*-cycles on a sub-complex of a filtration. The PL scalar field represents the elevation on a terrain. Both the orange and the blue rings are* 1*-cycles. Indeed, the computation of their boundaries results in summing all the vertices of each ring twice (as all vertices are faces of exactly two edges of the ring), yielding a boundary equal to* 0 *for each cycle. However, the two cycles are not homologous: it is not possible to go from one to the other by simply adding a* p*-boundary.*

can be understood as a measure of the importance of a feature detected by a critical point. Using this importance, it is then possible to rank features or even discard the features of lesser importance that correspond to noise. In this subsection, we will formalize the definition of persistent homology and its most common abstraction: the persistence diagram.

2.1.3.1 Homology group

In this subsection, we formalize what was touched upon for low dimensional data in subsubsection 2.1.1.4. Connected components, handles and voids are what is called a *homology class*, that can be generalized to higher dimensions. It relies on the concept of cycles, used to detect "holes", as a sum of simplices.

Definition 2.32 *(*p*-chain)* A p-chain c is a formal sum of modulo 2 coefficients of p-simplices σ_i of $\mathcal{M}$: $\sum \alpha_i \sigma_i$ with $\alpha_i \in \{0, 1\}$. Two p-chains can be summed together component-wise to form a new p-chain.

A p-chain can be modeled with a bit mask where a simplex is present if it has been added an odd-number of times. For example, adding the two 0-chains $a = v_0 + v_1$ and $b = v_1 + v_2$ will yield $a + b = v_0 + v_2$.

Definition 2.33 *(Boundary of a* p*-simplex)* The boundary of a p-simplex σ_i is noted $\partial\sigma_i$ and is defined as the sum of its faces of dimension $(p-1)$.

Definition 2.34 *(Boundary of a* p*-chain)* The boundary of a p-chain c is the sum of the boundaries of its simplices: $\partial c = \sum \alpha_i \partial\sigma_i$.

Definition 2.35 *(p-cycle)* A p-cycle c is a p-chain such that $\partial c = 0$.

Definition 2.36 *(Group of p-cycles)* The group of p-cycles of a simplicial complex $\mathcal{K}$ is the group of all p-cycles of $\mathcal{K}$, noted $\mathcal{Z}_p(\mathcal{K})$.

The concept of a p-cycle is central to the definition of homology group. Intuitively, it is a p-chain that is able to "go around" the domain and end where it started, as illustrated in Figure 2.9 (blue and orange rings). However, it is not enough to detect a "hole" in the domain. For this, one needs to study the relationship between cycles and p-boundaries.

Definition 2.37 *(p-boundary)* A p-boundary is a p-chain that is itself the boundary of a $(p+1)$-chain.

Definition 2.38 *(Group of p-boundaries)* The group of p-boundaries of a simplicial complex $\mathcal{K}$ is the group of all p-boundaries of $\mathcal{K}$, noted $\mathcal{B}_p(\mathcal{K})$.

p-boundaries are always p-cycles however the opposite is not true: p-cycles are not necessarily p-boundaries. For example, in Figure 2.9, the orange cycle is also a boundary (of the rest of the little hill it surrounds). However, the blue cycle surrounds a hole: the boundary of that hole is a 1-cycle, but it is not a 1-boundary, as the hole is devoid of simplices. Cycles that are not p-boundaries are captured by the concept of *homology group*.

Definition 2.39 *(Homology group)* The p^{th} homology group of a simplicial complex $\mathcal{K}$ is the quotient group of its p-cycles modulo its p-boundaries: $\mathcal{H}_p(\mathcal{K}) = \mathcal{Z}_p(\mathcal{K})/\mathcal{B}_p(\mathcal{K})$.

With this definition, we can now define the equivalence between cycles: two p-cycles c and c' are called *homologous* if one can be transformed into the other by adding the boundary of a $(p+1)$-chain c'': $c = c' + \partial c''$. The set of all cycles that are homologous defines a *homology class*. Intuitively, a homology class gathers all the cycles surrounding the same set of "holes". Each class is composed of cycles that are homologous to each other. Any one of them can be chosen as the *representative* of that class. In Figure 2.9, the blue cycle is homologous to the boundary of the hole and therefore belongs to the same class. With these definitions, it is now possible to formally define Betti numbers as the number of homology classes within a homology group.

Definition 2.40 *(Betti Numbers)* The p^{th} Betti number of a simplicial complex $\mathcal{K}$ is the rank of its p^{th} homology group: $\beta_p(\mathcal{K}) = rank(H_p(\mathcal{K}))$.

In other words, it is the number of linearly independent classes

of $\mathcal{H}_p(\mathcal{M})$ (called generators) Detecting the p-dimensional independent *"holes"* in a domain therefore means extracting generators for the p^{th} homology group. It follows that $H_0(\mathcal{K})$ contains the classes representing connected components (with β_0 its number of connected components), $H_1(\mathcal{K})$ the handles and $H_2(\mathcal{K})$ the voids.

2.1.3.2 Persistence Diagram

Persistence homology can now be derived from the previous notions by applying the concept of homology group to the subcomplexes of a filtration. The filtration induces a sequence of mappings of the homology group of the subcomplexes of $\mathcal{K}$:

$$\mathcal{H}_p(\mathcal{K}_0) \rightarrow \mathcal{H}_p(\mathcal{K}_1) \rightarrow \dots \mathcal{H}_p(\mathcal{K}_{n-1}) = \mathcal{H}_p(\mathcal{K}) \tag{2.1}$$

Instead of studying the homology group of one simplicial complex $\mathcal{K}$, the object of interest is the homology group of each subcomplex of its filtration. The goal is not just to extract the generators of homology classes of each subcomplex, but to track their evolution during the filtration: which generator is created at which step? When does it disappear? The sequence of mappings can be used to define the notion of *persistent homology group*. More precisely, these mappings are *homomorphisms*, mappings between groups that commute with the group operation. In the case of homology groups, the group operation is the formal sum used in the definition of p-chains.

Definition 2.41 *(Persistent homology group)* Let a filtration and the mappings induced by inclusion, $f_p^{i,j} : \mathcal{H}_p(\mathcal{K}_i) \rightarrow \mathcal{H}_p(\mathcal{K}_j)$ for $0 \leq i \leq j \leq n-1$, between the corresponding sequence of homology group. The p^{th} persistent homology group corresponds to the i,j images of these homomorphisms, noted $\mathcal{H}_p^{i,j}$.

As the difference between two consecutive subcomplexes of a filtration is typically one simplex in the considered applications, it is possible to identify exactly which simplex induces a change in the homology of the subcomplexes and for which subcomplex of the filtration. Therefore, each generator of $\mathcal{K}_i$ can be associated with a unique pair of critical simplices (c, c'), corresponding to its *birth* (when the generator is created) and *death* (when it disappears). It follows the notion of *persistence*: the persistence of such a pair is equal to $p = f(c') - f(c)$. It is a measure of how long a generator has *persisted* though the filtration. With this, it is easy to distinguish between salient features of the data and noise. Pairs of small persistence

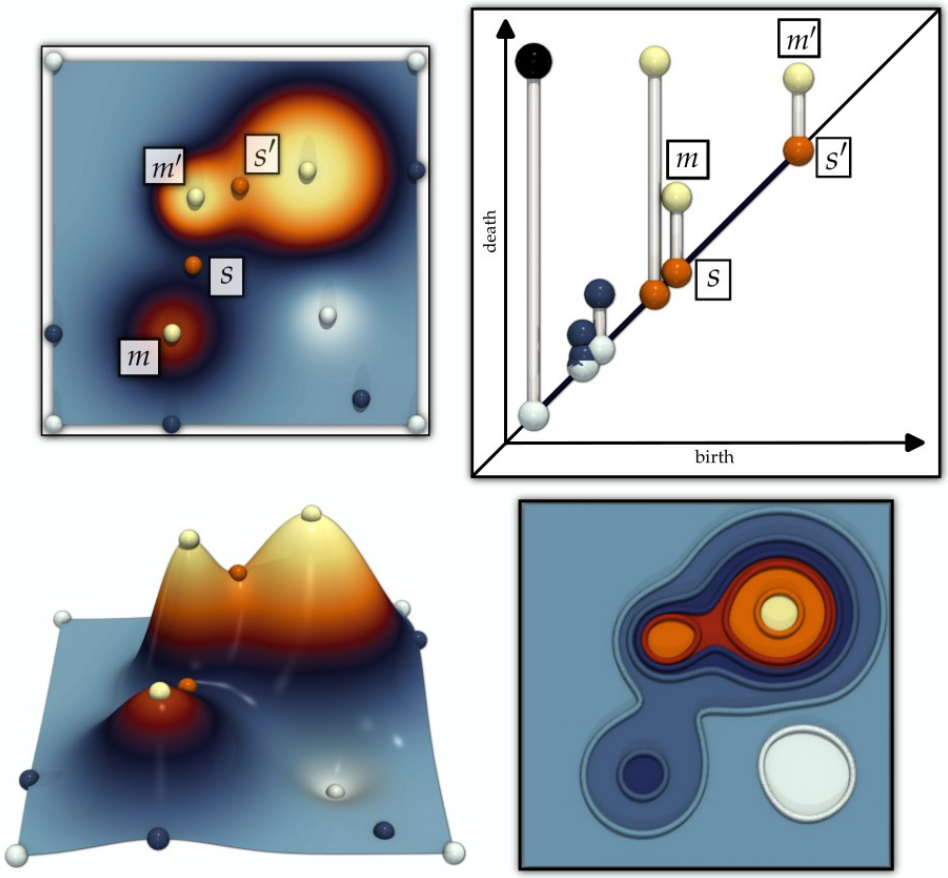


Figure 2.10 – *Example of persistence diagram and its construction. The PL scalar field, noted f, represents the elevation on a terrain (left). The persistence diagram (upper, right) tracks the evolution of homology generators in the filtration of f. Critical simplices of f are represented as spheres, in light blue for minima, dark blue for merging saddles, orange for splitting saddles and light yellow for maxima. The infinite pair is the biggest and first pair of the diagram and ends with a larger black sphere. It represents the fact that the input domain is made of a single connected component. Different sub-level sets of f are shown in the bottom right figure (represented on top of each other). As the isovalue increases, generators of homology classes are created: a new handle is created in s and dies in m, yielding the pair (s, m), visible on the diagram. Another handle is created in s' and dies in m'.*

correspond to noise, whereas important features will yield a high value of persistence.

When two generators merge into each other, the Edler rule is applied [EH09]. It states that the youngest homology class dies in favor of the oldest. The age of a class refers to the difference between the step of the filtration it first appeared in and the current step of the filtration. The older homology class appeared earlier in the filtration, it is therefore born earlier.

The pairs of critical simplices (c, c′) can be efficiently encoded in a topological abstraction called the *Persistence Diagram*.

Definition 2.42

(The Persistence Diagram) Let f be a PL scalar field. The p^{th} persistence diagram of a filtration is a one-dimensional simplicial complex that embeds generators for all homology class of the p^{th} persistent homology group in $\mathbb{R}^2$ by using its birth value as a first component and its birth and death values as second components, noted $\mathcal{D}_p(f)$ (or $\mathcal{D}_p$).

In a persistence diagram, critical simplices are arranged in pairs (c, c′). Each simplex appears in only one pair, with $f(c) < f(c')$. c is a p-simplex and c' a $(p+1)$-simplex. $\mathcal{D}_0(f)$ tracks the birth and death of connected components (and pairs minima and 1-saddles), $\mathcal{D}_1(f)$ of handles (and pairs 1-saddles and 2-saddles) and $\mathcal{D}_2(f)$ of voids (and pairs 2-saddles and maxima).

Each pair of the diagram is embedded as a bar in the birth/death 2D plane at the coordinates $(f(c), f(c))$ and $(f(c), f(c'))$. The persistence of a pair can be read as the height of the bar (i.e. $f(c) - f(c')$). Figure 2.10 shows an example of persistence diagram and illustrates how it is built.

In practice, the Elder rule is applied as follows, for example for $\mathcal{D}_0$: if two connected components, born respectively in critical vertices v and v', with $f(v) < f(v')$, meet at the critical edge e, the younger vertex, here v', dies, in favor of v, the older one. The pair (v', e) is then added to the persistence diagram and the computation for the other pairs continues.

One class never dies in our example during the filtration: the first class to appear in the filtration at the global minimum. It is never merged with another class. It is said to have *infinite persistence*. In practice in the diagram, a pair is still created, with its birth being the birth of the class and its death is the global maxima. It is called an *infinite pair*.

As stated before, salient features are characterized by a high persistence. In the diagram, it corresponds to the points located far from the diagonal. Conversely, the noise of the domain produces points in the

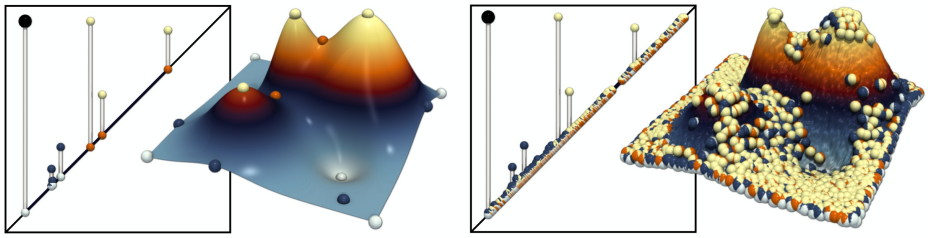


Figure 2.11 – *Persistence diagram for a clean (left) and noisy (right) dataset. Critical simplices are represented by spheres (light blue: minima, light yellow: maxima, black: end of the infinite pair, others: saddles). The persistence of each pair is the height of the bar. Salient features (long pairs) can easily be distinguished from the noise (short pairs).*

plane close to the diagonal, as shown in Figure 2.11. It becomes therefore straightforward to focus on features of a certain importance. One simply can use a threshold to filter out all pairs below a certain value of persistence, producing the diagram on the left of Figure 2.11 from the diagram on the right.

2.1.4 Other Topological Abstractions

Numerous other topological abstractions exist, each with their own characteristics, upsides and limitations. As it is not the focus of our work, we will not describe them in detail. However, we find of interest to sample a few in order to broaden the reader's knowledge of what is possible with TDA. In particular, we will present the Merge Tree, the Reeb Graph and the Morse-Smale Complex, illustrated in Figure 2.12.

2.1.4.1 The Merge Tree

The Merge Tree ([CSA00], Figure 2.12(a)) is very similar to the 0-dimensional persistence diagram but keeps an additional piece of information: the location of merge of the different homology classes. Instead of just having pairs of simplices, the Merge Tree also records their hierarchy by tracking the changes in connected components of sub-level sets. This topological abstraction is popular as it efficiently encodes prominent features of the data while simultaneously describing how they are connected. As the persistence diagram and the Merge Tree are very similar, the diagram of a dataset can be efficiently extracted from a Merge Tree.

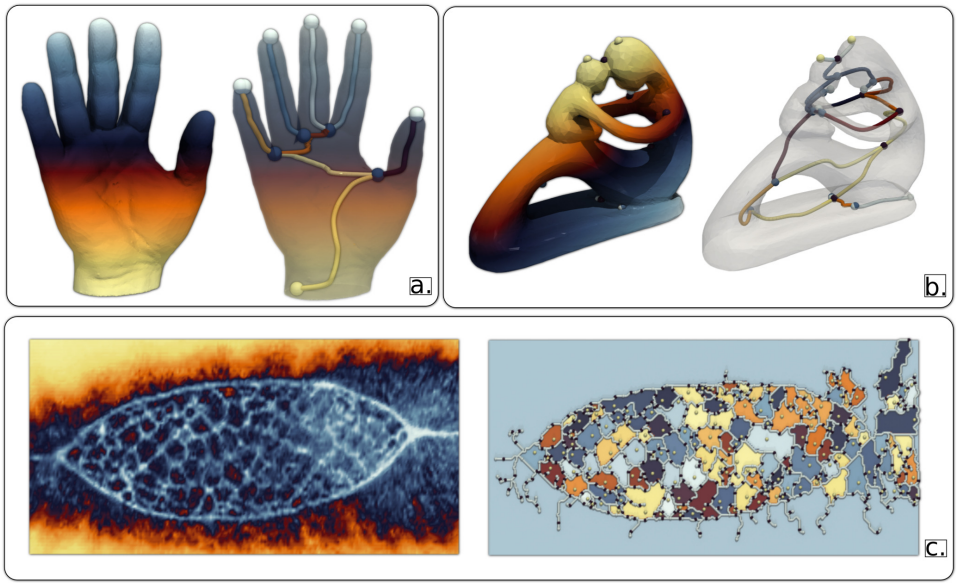


Figure 2.12 – *Examples of topological abstractions. The Merge Tree (a) detects salient features of the data and records their connection to one another. The Reeb Graph (b) enables the extraction of the structure of a shape. The Morse-Smale Complex (c) partitions the data according to the gradient flow. Here it allows to segment the cells of this well-known dataset [EH09].*

2.1.4.2 The Reeb Graph

The Reeb Graph ([Ree46], Figure 2.12(b)) is an advanced topological abstraction that keeps more information than both the persistence diagram and the Reeb Graph. Indeed, where the persistence diagram only keeps track of the birth and death of holes of the data, the Reeb Graph also retains the adjacency relations between the connected components of the level sets, making it an ideal tool to simplify and understand the structure of a complex shape.

2.1.4.3 The Morse-Smale Complex

The Morse-Smale Complex ([EHNP03], Figure 2.12(c)) provides a partition of the domain using integral lines and critical points. Integral lines connect critical points and segment the domain into cells of similar gradient flow. This is particularly useful for applications that see their features align with the gradient (e.g., filament extraction).

2.2 Parallel Computing

We now focus on parallelism and its foundational principles, establishing the core concepts and trends. Since their invention, continuous efforts

have aimed to make computers more powerful. Power came from different sources that varied over time. Initially, manufacturers tried to improve performance by increasing the frequency of processors, so it could perform more computations per second. Eventually, a plateau was reached in the 2000s. Increasing the frequency meant increasing the consumption of electrical power to such a degree that it became too high. A new idea emerged: instead of having one core per processor, what about having several cores that work together within one processor? Such a multi-core computer indeed allowed for more computations, but required significant changes in how computations were programmed, as parallelism was now required to obtain performance gains: this is shared-memory parallelism. There are different types of parallelism. In this section, we will look at several, in particular shared-memory and distributed-memory parallelism. Their specific programming paradigms will also be discussed.

2.2.1 Shared-memory parallelism

2.2.1.1 Shared-memory architectures

Shared-memory parallelism occurs when multiple processors share the same memory on a computer. *Single Instruction Multiple Data* (SIMD) is a type of parallelism that is often used in a shared-memory context. In this context, the same instruction is applied to different data. Here is an example of SIMD instruction: the addition of each element of two vectors of integers of the same size. The instruction is always the same (an addition of two integers) but the data varies (elements of the vectors). The first CPUs that implemented a SIMD paradigm are called *Vector CPUs*. A cluster of Vector CPUs makes up a *Vector Supercomputer*. First examples of vector supercomputers include the ILLIAC-IV1 (1975) and the Cray-1 (1976). Eventually, in the 1990s, interest for Vector CPUs, and their Supercomputers, waned in favor of CPUs capable of processing more complex instructions. However, SIMD operations are still relevant today, modern CPUs allowing for their use through specific sets of instructions like SSE, AVX, AVX2 and AVX-512. Furthermore, this architecture is efficient for repeated single instruction operations and regular memory accesses, two characteristics that TDA algorithms typically lack.

Modern CPUs are composed of multiple cores that share the same physical memory. It is common for a compute node to include more than one processor, often two, with all processors accessing the same physical memory and (see Figure 2.13). Nowadays, commodity processors typi-

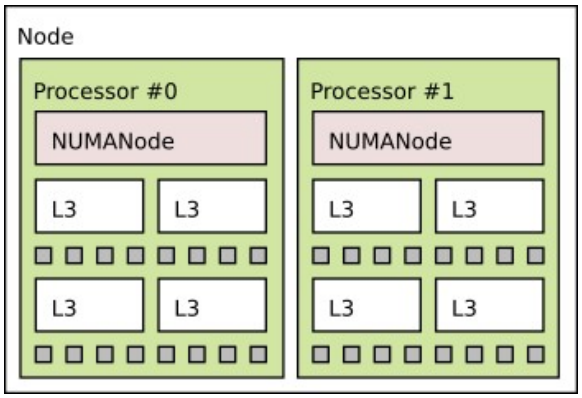


Figure 2.13 – *Architecture of a node of Sorbonne Université's MCMeSU supercomputer. A node is composed of two AMD EPYC 7313 Milan CPUs. Each processor possesses 16 cores and a NUMA node.*

cally have 8 to 32 cores, while supercomputer processors have up to 64 cores and multiple processor per node, as can be seen in the Top 500 [top], a famous ranking of the world's most powerful supercomputers.

Because of the multi-processor architecture of modern compute nodes, it is possible for cores to access some parts of the memory of a node faster than other parts. This leads to a *Non Uniform Memory Access* architecture (NUMA). It is important to take data locality into consideration when programming for such architecture as well as when configuring thread placement for runtime execution.

2.2.1.2 Shared-memory programming

Shared-memory parallelism is often implemented on modern CPUs by relying on *thread-based programming*. *Threads* can be defined as light processes. A process is the instance of a computer program that is being executed. It has its own memory space and code that rules its execution. In a shared-memory parallel setting, a process can spawn several threads. Each thread has access to all the memory of the process. Typically, a thread will be created for each core of the CPU to avoid the overhead of context switch and cache reloading. Simultaneous Multi-Threading (SMT) refers to the execution of instructions of different threads on one core at once. When using SMT, a *physical* core (a core that truly exists on the hardware) can host more than one *logical* core (a software abstraction representing a core). Typically, a physical core will hold two logical cores, allowing for more computations at once without paying the cost of operations such as a context switch. In many instances, such a configuration has proven to

improve overall performance. SMT was not used in this work as it was not available on Sorbonne Université's supercomputers.

Each thread will be given work to perform simultaneously to other threads. As all threads share the same memory, issues that do not arise when running sequentially may occur. These problems are often left to the programmer to solve. For example, some computations may require all threads to have achieved their computation to a certain point before any thread can continue past that point. This problem can be solved by implementing a *barrier* between all threads to ensure all necessary computations have been performed before resuming with the rest. Other problems can also occur: multiple threads may want to access or modify the exact same memory location at the same time. This leads to race conditions and undefined behaviors. Many solutions exist to prevent them from happening. A *Lock* can ensure that only one thread at a time can execute a particular section of code or access a shared resource. *Atomic operations* are lighter way to ensure that the memory is only accessed or written to one thread at a time. For example, an *atomic update* on a variable `i` to perform the operation `i++` will ensure that one thread: *(1)* loads `i` in a register, *(2)* increments it and *(3)* writes it back to the memory shared by all threads. No other threads can perform similar actions on this particular variable after *(1)* started and until *(3)* is finished. These solutions will slow down the overall execution by creating overhead or forcing threads to wait, therefore, it is imperative to manage these mechanisms carefully to optimize performance.

2.2.1.2.1 Thread-based programming Several programming paradigms exist for the creation and management of threads in a process. One of the first general standard to do so was *POSIX Threads*, or *Pthreads* in 1995. Pthreads is still in use today. It is a low level standard. It allows for a very fine-grain control over thread management and a lot of flexibility in what can be done. The drawback is that the programmer must be very careful and explicit about what is being implemented and the code can become quite complex.

Another wildly popular standard for thread management is *OpenMP* [Ope20]. The first version of the standard was published in 1997 and has been iteratively improved and expanded over the years. The latest improvements to OpenMP focus on GPU offloading, first added to OpenMP 4.0 in 2013 and tasks, first added to OpenMP 3.0 in 2011. Though these features were added over a decade ago, recent and significant efforts have

```
#pragma omp directive-name [clause]
{
        // Block
}
```

Listing 2.1 – *OpenMP basic block structure*

```
int i;
#pragma omp parallel for private(i) shared(A, B, C)
{
        for (i = 0; i < N; i++){
                A[i] = B[i] + C[i];
        }
}
```

Listing 2.2 – *Example of OpenMP directive in a C++ code*

greatly improved their performance and capabilities. The latest version, OpenMP 6.0, was published in November 2024. Unlike Pthreads, it is a very high level programming standard. It provides an extension to the C, C++ and Fortran programming languages. OpenMP relies on the use of *directives*, small lines of code that use the `#pragma` mechanism in C and C++, and of comments in Fortran. The directives are high-level indications to the compiler of how it should parallelize the execution of the code. Compilers can easily ignore OpenMP directives if the OpenMP support is not provided or not enabled. This means that code parallelized with OpenMP can also execute sequentially without any changes to the code.

An OpenMP block has the structure shown in Listing 2.1. OpenMP relies on a Fork-Join model, which means that one process executes the program in a master thread until reaching a code section needing parallel execution, signaled using the `parallel` OpenMP directive. The process will then create as many threads as required for the computation, depending on what is explicitly required by the block or by a global configuration variable (such as the environment variable `OMP_NUM_THREADS`).

Listing 2.2 shows a basic parallelization of a `for` loop that computes the element-wise sum of two vectors B and C. The `omp` keyword tells the compiler that this pragma is an OpenMP directive. The `parallel` keyword will induce the creation of threads and the parallel execution of the

code that follows, the `for` keyword will tell the compiler to parallelize the following loop and the `private` and `shared` keyword allow for thread memory management. The work in the `for` loop is parallelized by giving to each thread one or several chunks of iterations of the loop. The number of iterations of a chunk is defined when reaching the directive. Then, the workload of each thread can be managed statically or dynamically, depending on what the programmer requires. A static workload schedule means that chunks are assigned to specific threads once and for all at the start of the `for` loop, whereas a dynamic workload schedule means that chunks are assigned at runtime. This may be more efficient when iterations of the loop are not all equal in terms of workload. In that case, having more chunks than threads will help improving overall performance as the overall work will be more evenly distributed.

OpenMP also enables to control the memory management of the threads. Though they can access all the memory, each thread has its own private memory. Directives allow the programmer to indicate which data element should be only accessible within the private memory of a thread and which is shared by all threads. The *critical* directive restricts the execution of a code section to one thread at a time. The *atomic* directives ensure an atomic access or modification of a variable.

OpenMP's biggest strength is also the source of its weakness: this is a very high level programming paradigm. With very little additional code, a programmer is capable of parallelizing code and reducing drastically its execution time. However, because OpenMP handles many tasks autonomously, it is sometimes complicated to understand precisely what is hindering performance and how to fix the problem. Furthermore, the programmer is limited to the scope of the directives. For example, some directives can only be applied to parallelize `for` loops. If the program is structured differently, such directives are unusable. Nevertheless, its ease of use and generally good performance make it a wildly used standard for shared-memory parallelization in C/C++ and Fortran.

2.2.1.2.2 Task-based programming Another programming paradigm for shared-memory parallelism is the *task-based* programming. Threads are also used to perform the work; however, the way the work is divided and assigned to threads is handled a bit differently. The work is divided in *tasks* that correspond to small units of work. When a task is created, it is placed in a *task pool*, where it waits until it is picked up by a thread for computation. Mechanisms of dependencies and priorities organize tasks within the task

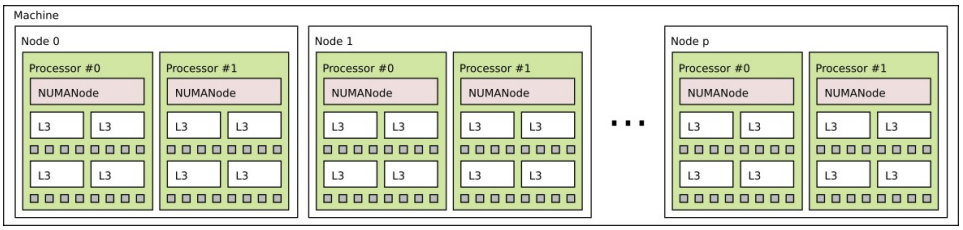


Figure 2.14 – *Architecture of $p + 1$ nodes of Sorbonne Université's MCMeSU supercomputer. Each node is composed of two CPUs and is connected with others through a network.*

pool to allow for correct and efficient execution. Task-based programming was first introduced in Cilk [BJK+95] in 1994. It has gained traction and is now wildly used in several softwares and standards, as it allows to focus on the tasks themselves rather than on the thread management. Frameworks and standards that implement task-based programming include StarPU [AAF+12], oneTBB [Inta, Intb] or Charm++ [KK93]. OpenMP also implements this paradigm, progressively adding tasks, dependencies and then priorities. For OpenMP in particular, task-based programming enables to dynamically balance a workload often more efficiently than nested parallelism of blocks of code as described in the previous subsection.

While shared-memory parallelism allows for great improvements of the execution time of a computation, one is still limited to the resources of a single computer. If a computation needs more memory than a computer can have then shared-memory parallelism is not a viable solution. TDA problems often have a significant memory footprint, one too big to fit in the memory of one computer. Additionally, execution time may become prohibitively long, making computations impractical. When these problems arise, one can look into distributed-memory parallelism.

2.2.2 Distributed-memory parallelism

When the capabilities of a single computer are insufficient, a common solution is to use multiple computers, called *computing nodes*, into a cluster of nodes. Nodes are interconnected using a high-speed network and form a supercomputer. The difficulty of programming in a distributed-memory context is that each node has its own memory and does not have direct access to the memory of other nodes. Additional measures have to be taken in order to exchange data between nodes and ensure the correct and efficient execution of the computation. Similarly to the shared-memory context, synchronization steps also may be required to ensure correct ex-

ecution. Nowadays, the most popular solution for distributed-memory computations is the *Message Passing Interface*.

2.2.2.1 The Message Passing Interface

The *Message Passing Interface* (MPI) is a standard created in 1994 for distributed-memory programming [Mes23]. It provides language bindings for C and Fortran. Additional libraries, like mpi4py [DF21] in Python, are built on top of the standard and follow closely C++ bindings. Similarly to OpenMP, this standard has been progressively improved and expanded over the years. Some of the latest major feature additions to MPI include large counts in communications (limited up until now to a count described by an integer), persistent collectives (collectives that will be reused frequently) and an Application Binary Interface (a specification of the memory layout of the set of types and constants defined in MPI's API). This last feature was introduced in the latest version of MPI, version 5.0, published in 2025.

Most MPI communications need to be explicitly defined both on the sending and receiving end. When a program executes with MPI, several processes will spawn, each with a distinct rank (i.e. its identifier), its own memory, etc ... It is possible to have more than one process on one node. In that case, some implementation may take advantage of the proximity of the processes located on one node to speed-up the exchanges in practice. For example, *Open MPI*, a well-known open-source implementation of MPI, uses a mechanism called the *sm BTL* (or *Shared-Memory Byte Transfer Layer*) that relies on shared-memory for transferring data between two processes.

There are different types of communications. The most common one is point-to-point exchange: one process will communicate with a single other process. An example of such a communication is shown in Listing 2.3. In this code, process 0 will send the value of variable *a* to process 1. Both the sending and receiving action need to be specified using separate functions (`MPI_Send` and `MPI_Recv`). Other actions related to MPI featured in this code include initializing the MPI context (with `MPI_Init`), retrieving the total number of processes (with `MPI_Comm_size`), retrieving the rank of the current process (with `MPI_Comm_rank`) and finalizing the MPI context (with `MPI_Finalize`). This makes for quite a verbose implementation.

Collectives make for a second type of communications. They are defined as communications involving a group of processes. Examples of col-

```
#include <iostream>
#include <mpi.h>

int main(int argc, char * argv[]){
        int npes, myrank;
        int a;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &npes);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        MPI_Status status;
        if ( myrank == 0 ){
                a = 15;
                MPI_Send(&a, 1, MPI_INT, 1, 1,
                        MPI_COMM_WORLD);
        }
        if ( myrank == 1){
                MPI_Recv(&a, 1, MPI_INT, 0, 1,
                        MPI_COMM_WORLD, &status);
                std::cout << "a␣:" << a << std::endl;
        }
        MPI_Finalize();
        return 0;
}
```

Listing 2.3 – *Example of MPI code.* MPI_COMM_WORLD *describes the group containing all the processes.*

lectives include `MPI_Reduce`, that performs global reduction operations such as sum, maximum or the logical "and", `MPI_Gather`, that gathers elements of an array scattered on multiple processes to the memory of one process and `MPI_Scatter`, that scatters elements of an array present on one process across multiple processes. Finally, MPI also permits the use of one-sided communications through the use of dedicated "windows" that expose part of the memory of a process to direct access from other processes. In practice, the use of one-sided communications is limited.

Both collectives and point-to-point communications can be executed in a blocking or non-blocking manner. A non-blocking communication will not stop the execution of the program: the process will exit the function call and continue the rest of the computation even if the communication itself is not over. Non-blocking communications induce less idle time between processes and therefore less time spent waiting. It also enables to overlap communications and computations. However, it requires particular care to ensure the communications have occurred before using their results. The overhead of non-blocking communications may sometimes not be worth the gain.

This balance between cost and gain is true for all MPI actions. Communications and synchronizations between processes have a cost. Typically, the execution time of a parallel program eventually stops decreasing even when adding new processes to the execution. A plateau can be reached due to the cost of running the computation in a distributed-memory context. There can be too many communications, too many synchronizations to enable a speedup of the execution time of the program. It is therefore interesting to try and limit the number of MPI processes. A common solution is to use a hybrid MPI-X parallelization, with X being a programming paradigm for shared-memory parallelism. Indeed, the nodes of most supercomputers have multiple cores. Instead of having one MPI process per available core, one can have one process per node and use shared-memory parallel programming on each node. In particular, we will look into hybrid MPI+thread programming and more specifically, hybrid MPI+OpenMP programming.

2.2.3 Hybrid MPI+thread programming

Hybrid MPI+thread programming can improve scalability over pure MPI programming by reducing synchronizations and communications while harnessing the power of multiple nodes. Furthermore, it can easily en-

able dynamic load balancing between threads of the same node. There are fewer processes and possibly fewer communication overhead overall while not being limited to the use of a single computer as in pure shared-memory parallelism. If the program requires some data to be duplicated over all processes, hybrid programming also permits to reduce memory overhead. The main downside of this hybrid setting is that the program needs to be parallelized for both programming paradigms. Development can become overly complex and difficult to maintain. Furthermore, usage gets more complicated as one must optimize the placement of both threads and processes. The threads can be bound to the CPU cores using the OpenMP thread affinity features [Ope20]. Processes can also be bound to specific nodes or hardware elements of a node, such as as a NUMA node or a core. Process placement tools are often specific to each MPI implementation. Such placement can be useful to better exploit compute nodes with multiple NUMA nodes, by spawning and binding one process for each NUMA node. This configuration ensures that the processes and their threads will only access memory that is close and does not require an extra cost.

The MPI standard supports the use of threads and provides different levels of thread support depending on the communication needs of the threads. The level of support is activated at the beginning of the program, by initializing the MPI context using the function `MPI_Init_thread` and specifying the required level of support. The lowest level is `MPI_THREAD_SINGLE`: only one thread will execute. This is the default (the execution is not multithreaded). The next level is `MPI_THREAD_FUNNELED`: only the thread that called `MPI_Init_thread` will make MPI calls. The third level of support is: `MPI_THREAD_SERIALIZED`: only one thread will make MPI library calls at one time. Finally, the last level of support is `MPI_THREAD_MULTIPLE`: multiple threads may call MPI at once with no restrictions. In practice, only the last level of support creates a significant overhead at runtime, as additional synchronizations need to be made for each MPI calls to ensure calls made at the same time by multiple threads do not collide. Lower thread supports influence more the development of the program than the runtime execution. Additionally, some implementations of MPI allow to request a given thread support through environment variables (such as Open MPI with `OMPI_MPI_THREAD_LEVEL`).

A typical MPI+thread application that needs to perform communications and computations at the same time may want to use all threads

for its computation needs. The master thread then performs all the MPI communications. This potentially creates an imbalance of work, with the Master thread needing to perform both his computation and communication work. Having all threads participate equally in communications is not necessarily an effective solution. Indeed, using the highest level of thread support may induce an overhead not worth the gain [VKP+15].

The *communication thread* is an alternative approach for applications that perform communications and computations concurrently. It is a thread dedicated to communications, both sending and receiving calls. It is the only one to perform communications and in turn it performs little to no computation work. This model has been implemented in numerous projects like DAGuE [BBD+12], starPU [AAF+12] and others [SFLC18, VKP+15]. This model also allows for true overlap of communication and computation. Indeed, even though in the standard non-blocking communications allow for overlap, in practice communications often do not progress in the background by default [DT16, HSSW11]. The communication may effectively occur only when waiting for its completion, after the computation or when the process enters the MPI layer for any other MPI function call. The communication thread dedicates a thread to communications. In this setting, the thread is regularly waiting for communication completion and therefore ensures the progress of the communications. It also allows for more reactivity as messages can be sent as soon as ready, without using the highest thread support.

An immediate drawback from using a communication thread is that there is one fewer thread to perform computation, therefore the gain must be significant in order to justify its use. One could choose to spawn one more thread than there are cores available to maintain the number of computation cores, however this may decrease the reactivity of the communication thread and add additional costs with context switching and cache loading, therefore this is not necessarily an efficient solution. Another disadvantage is the complexity of implementing and maintaining such a model. Debugging becomes significantly more complex to handle as threads have different roles. The data structures for sending and receiving messages also needs to be thread safe, meaning that all data accessed and modified by threads relative to communications need to be designed for safe concurrent accesses and modifications.

2.2.4 Alternative distributed-memory paradigms

As stated before, programming with MPI can be complex and time-consuming. In this section, we look into several solutions that have been implemented to try and mitigate this difficulty while maintaining good performance.

2.2.4.1 Shared Virtual Memory Systems

Shared Virtual Memory Systems aim at simplifying programming in a distributed-memory setting by considering the distributed memory of the nodes as one global memory. This mechanism can be implemented at the operating system level. The program is then run "as if" all nodes have access to all the memory. No changes need to be implemented within the program for a distributed-memory execution. However, the convenience of use of such a system often collides with its cost.

2.2.4.2 The PGAS model

The *Partitioned Global Address Space* (PGAS) model [YBC+07] aims at simplifying distributed-memory programming by providing a global address space that can be accessed by any process through one-sided communications. Examples of languages implementing the PGAS model include Coarray Fortran [NR98] and UPC [CDC+99]. The global address space allows for a global memory access while maintaining awareness of distant data access. The model relies on one-sided communications: only one node is aware the communication is happening, the remote node that possess the data is not. This induces a simpler programming structure as fewer calls are necessary to perform most communications. However, the memory management can be quite complicated as it creates problematic situations. For example, one must ensure with appropriate function calls that only one process at a time will perform one-sided communication of the same memory location. This model and its languages is high level, in particular compared to MPI, and may be a bit too restrictive for complex data traversal and computations.

2.2.4.3 Apache Spark

Another popular framework that seeks to abstract distributed-memory management is Apache Spark [ZCD+12]. It is an in-memory distributed computing engine that can distribute computations across multiple nodes.

Apache Spark is built on an advanced distributed SQL engine for large-scale data. It is specialized in data engineering, data science and machine learning. It is very high-level and provides API in Python, SQL, Java, R and Scala. The engine is self-managing and fault-tolerant and hides the distributed-memory problems as much as possible from the user, with most distributed-memory management being done implicitly. Its in-memory execution allows it to be much faster than its counterpart, Hadoop. Indeed, at each processing step, Hadoop will read and write the data to disk, whereas Spark relies on RAM to store the data during processing. Its data structures relies on *Resilient Distributed Datasets*, which are distributed memory abstractions for fault-tolerance and in-memory computations.

However, most Spark computations are built on the MapReduce model or SQL queries. This is efficient for structured grid datasets or computations that can be performed on DataFrames-like data. In the field of TDA, performance results for such computations are promising [QLIF24] but it has been limited to embarrassingly parallel algorithms such as computations of the critical points or of the Forman gradient. When more complex data traversals and computations are needed, SQL-like and MapReduce-like queries are not always adapted.

2.2.4.4 DIY

DIY [MP16a] is a block-parallel software library that enables developers to write a single implementation of a given algorithm, while supporting at the same time multiple runtime configurations (out-of-core, shared-memory parallelism or distributed-memory parallelism). The input data is partitioned into blocks processed by threads, with either one or multiple threads per process, seamlessly combining distributed-memory message passing with shared-memory thread parallelism. The underlying abstraction that facilitates these capabilities is block parallelism [MP16a]. In this model, computational blocks and their associated message queues are assigned to processing elements (such as MPI processes or threads) and are dynamically migrated between memory and storage by the DIY runtime. DIY implements complex communication patterns, such as neighbor exchange, merge reduction, swap reduction, and all-to-all exchange.

In practice, though it is not specifically designed for topological algorithms, this library is mostly used in TDA applications [NM20, MP16b]. We made the choice not to rely on this library in our work because we be-

lieved that a custom MPI+thread implementation can provide more flexibility than DIY. In particular, the block-parallel model prevents the implementation of a communication thread, which we believe could improve the performance of topological algorithms in a hybrid MPI+thread setting. Furthermore, the block-parallel model hinders dynamic load-balancing within each node. With DIY, the workload within a single data block cannot be distributed among multiple threads if needed, unlike a custom MPI+thread implementation would allow.

2.2.5 GPU Computing

There is another architecture for parallelism that is left to mention: *Graphics Processing Units*, or GPUs, and their many-core parallelism. In 2006, NVIDIA released the CUDA architecture and its associated language, CUDA C [Nvi25]. GPUs were originally designed to perform graphics computation but their use has broaden over the years. Their architecture is characterized by their very high number of computing cores (often reaching thousands). Each core is less powerful than a CPU core but their number makes for an overall excellent performance, in particular for embarrassingly parallel applications with repeated single instruction operations and regular memory accesses such as the training of neural networks or computational fluid dynamics calculations. GPUs are not autonomous, they need a CPU to function and trigger their computation. This setting generates specific problems and solutions, such as data transfers between the GPU and CPU or separate memory allocations. The rising popularity of this execution model led to the development of standards such as OpenCL [Khr25], which is similar to the CUDA C language but is not limited to NVIDIA GPUs. On top being a commodity-based component crucial to two industries (the gaming industry and professional graphics), GPUs are also considered a better alternative to CPUs for scientific computations (in terms of GFlop/s for energetic cost). All this contributes to their current popularity.

However, we will not target GPUs in our work. This is due to two main reasons. First, most TDA algorithms rarely require only regular memory accesses. The cost of topological algorithms often rely on multiple irregular traversals of the data. The computations themselves are limited and therefore would likely not be optimized for the GPU infrastructure. This is why the past efforts to parallelize TTK in a shared-memory setting has been focused on CPUs. Second, the current limitation pushing us towards

distributed-memory computation is the memory. TDA algorithms often have a significant memory footprint. This is currently the limitation preventing the use of TTK on larger datasets. Switching from executing on a CPU to executing on a GPU will not provide more memory. In fact, GPUs often have less memory than CPUs.

2.2.6 Performance metrics for parallelism on CPUs

To evaluate the performance and scalability of algorithms and their implementations, one needs appropriate metrics and benchmarks. Different methods and measurements of performance evaluation have been used over the years. A good scaling study shows how effectively additional cores are leveraged by an algorithm. A common analysis for assessing scalability is the *strong scaling analysis*, where, for a particular dataset, the algorithm is run several times with an increasing number of cores. The main limitation of this setting is that there will always be a number of cores beyond which performance will plateau or worsen. Another limitation is that it requires that the algorithm be run on one node, which limits the size of the input dataset. To solve these problems, one can turn to a *weak scaling analysis*. In this setting, the size of the dataset is increased proportionally with the number of cores in order to increase the workload. However, increasing the size of the dataset does not necessarily increase the workload proportionally, in particular in the case of TDA algorithms, where the workload is often more dependent on the number of topological structures than the size of the grid. For a more comprehensive study of performance, in this manuscript, we will perform both strong and weak scaling analyses when assessing performance of distributed-memory algorithm.

Another question of performance evaluation is the representation of the results. Though the execution time is the quantity that is measured in the benchmarks, it may not be the best metric to understand the scalability of an algorithm. We use the parallel efficiency, a commonly used metric defined as follows for strong and weak scaling analyses.

Definition 2.43 *(Speedup)* The speedup s_p for p cores is defined as $s_p = \frac{t_1}{t_p}$, with t_p and t_1 being the execution times for p and 1 cores.

Definition 2.44 *(Strong scaling efficiency)* The *strong scaling efficiency* for p cores is defined as $\frac{s_p}{p} \times 100$, with s_p the speedup on p cores.

Intuitively, a strong scaling efficiency of 100% corresponds to the case

where, when multiplying by two the number of cores, the execution time is divided by two. Such an efficiency is often unattainable but is an ideal goal.

Definition 2.45 *(Weak scaling efficiency)* The *weak scaling efficiency* for p cores is defined as $\frac{t_1}{t_p} \times 100$, with t_1 and t_p being the execution times on 1 and p nodes.

Intuitively, a weak scaling efficiency of 100% corresponds to the case where, when proportionally increasing the number of cores and the workload, the execution time is constant. Again, such an efficiency is often unattainable but is an ideal goal.

This metric facilitates the representation of different datasets within a figure by unifying the range. Indeed, the efficiency will always range from 0% to 100%, whereas execution time can vastly differ from one dataset to the next. It is also a less forgiving representation than the speedup, as it highlights the gradual drop-off from the unattainable perfect efficiency, where the speedup usually shows an upward slop. This makes differences of performance between datasets or implementations more visible as the number of cores increases.

2.3 Software environment

In this section, we examine the existing software tools upon which our work is built. First, we will present several front end visualization softwares, in particular the Visualization ToolKit (VTK) and ParaView. Second, we will present the Topology ToolKit (TTK) and compare it to other existing TDA softwares. Finally, we will discuss existing shared-memory parallel algorithms and implementations.

2.3.1 Existing front end visualization software frameworks

There are numerous existing softwares for large scale visualization that allow for distributed-memory computations. Some come in the form of tool kits and libraries, such as the Visualization ToolKit (VTK) [Kit03], a library for manipulating and displaying scientific data, and VTK-m [MAB+24], a library for visualization and analysis optimized to perform well on many-core platforms such as GPUs. Others offer more complete interfaces. Examples of such softwares include SCIRun [SCI23], a problem solving environment developed by the NIH Center for Integrative Biomedical Computing, VisIt [CBW+12], a scientific visualization and analysis tool that operates on mesh-based field data, Ascent [LBCH22], a lightweight, in-

situ visualization and analysis library designed for running multi-physics simulations on HPC systems, and ParaView [AGL05], an open-source, multi-platform scientific data analysis and visualization tool that enables analysis and visualization of extremely large datasets.

In particular, ParaView has become the de-facto standard for the visualization and analysis of large-scale data, by combining raw power, scalability, extensibility, and usability. It has been developed and maintained by the company Kitware for over 20 years and uses VTK to provide the visualization and data processing model. It is backed by a large community of contributors and researchers that drive its evolution to meet their needs, making it a very complete tool for a wide-ranging number of use cases.

2.3.2 The Topology ToolKit

The *Topology ToolKit* (TTK) [TFL+17] is an open-source library for topological data analysis and visualization, written in C++, which implements a substantial collection of algorithms [BMBF+19] for scalar data, bivariate data, ensemble data or even point cloud data. Over 40 contributors have participated in the development of TTK, totaling now 165 000 lines of code in its core base. In contrast to pre-existing, tailored, mono-algorithm implementations, TTK ***(1)*** supports multiple algorithms, ***(2)*** it is versatile (it provides time and memory efficient supports for multiple, typical data representations found in scientific computing and imaging, such as triangulated domains or regular grids) and ***(3)*** it consistently supports the combination of multiple algorithms into a *topological analysis pipeline* (see the *TTK Online Example Database* [TTK22], a database of real-life data analysis use cases, implementing advanced topological analysis pipelines, combining multiple algorithms). This section provides some background regarding TTK, and it details its pre-existing support (i.e. prior to this work) for triangulation traversal.

2.3.2.1 Scope and interfaces

While TTK can be used directly via its raw, low-level C++ interface, TTK also provides an interface of higher-level, for VTK. In particular, as described in its companion paper [TFL+17], each TTK algorithm is *wrapped* into a VTK *filter* (i.e. an elementary data processing unit in the VTK terminology). Specifically, each topological algorithm implemented in TTK inherits from the generic class named `ttkAlgorithm`, itself inherit-

ing from the generic VTK data processing class named `vtkAlgorithm`. Then, when reaching a TTK algorithm within a distributed pipeline, ParaView will call the function `ProcessRequest` (from the `vtkAlgorithm` interface, see Figure 3.5). The re-implementation of this function in the `ttkAlgorithm` class will trigger all the necessary preconditioning before calling the actual topological algorithm (see Section 3.5 for examples), implemented in the generic function `RequestData` (from the `vtkAlgorithm` interface). Thanks to this *wrapping*, a developer can use TTK features with the same syntax as VTK features. TTK also provides a plugin for ParaView. Then, ParaView users can interactively call TTK filters via its graphical user interface. Finally, TTK also provides two Python interfaces (a low-level one, matching its VTK interface, and a high-level one, matching its ParaView interface). In 2022, TTK was officially added to ParaView, increasing its accessibility to non-programmer end-users.

2.3.2.2 Pre-existing triangulation

This section briefly summarizes the pre-existing implementation of TTK's triangulation data structure [TFL+17]. Internally, each topological algorithm implemented in TTK is exploiting this data-structure. In the following, the triangulation $\mathcal{M}$ is assumed to be of uniform top dimension, i.e. any d'-simplex (with $d' \in \{0, 1, \ldots, d-1\}$) admits at least one d-dimensional co-face. In the explicit case (the input is a simplicial mesh), this data structure takes as an input a pointer to an array of 3D points (modeling the vertices of $\mathcal{M}$), as well as a pointer to an array of indices (modeling the d-simplices of $\mathcal{M}$). In the implicit case (the input is a regular grid), it takes as an input the origin of the grid as well as its resolution and spacing across each dimension. These can be provided by any IO library (in our experiments, these are provided by VTK).

Based on this input, the triangulation supports a variety of traversal routines, to address the needs of the algorithms.

1. **Simplex enumeration:** for any $d' \in \{0, \ldots, d\}$, the data structure can enumerate all the d'-simplices of $\mathcal{M}$.

2. **Stars and links:** for any $d' \in \{0, \ldots, d\}$, the data structure can enumerate all the simplices of the star and the link of any d'-simplex σ.

3. **Face / co-face:** for any $d' \in \{0, \ldots, d\}$, the data structure can enumer-

ate all the d''-simplices τ which are faces or co-faces of a d'-simplex σ, for any dimension d'' (i.e. $d'' \neq d'$ and $d'' \in \{0, \ldots, d\}$).

4. **Boundary tests:** $d' \in \{0, \ldots, d-1\}$, the data structure can be queried to determine if a d'-simplex σ is on the boundary of $\mathcal{M}$ or not.

As discussed in the original paper [TFL+17], such traversals are rather typical of topological algorithms, which may need to inspect extensively the local neighborhoods of simplices. All traversal queries (e.g. getting the i^{th} d''-dimensional co-face of a given d'-simplex σ) are addressed by the data structure in constant time, which is of paramount importance to guarantee the runtime performance of the calling topological algorithms. This is supported by the data structure via a *preconditioning* mechanism. Specifically, in a pre-processing phase, each calling topological algorithm needs to explicitly declare the list of the types of traversal queries it is going to use during its main routine. This declaration will trigger a pre-conditioning of the triangulation, which will pre-compute and cache all the specified queries, whose results will later be addressed in constant time at query time. This design philosophy is particularly relevant in the context of analysis pipelines, where multiple algorithms are typically combined together. There, the preconditioning phase only pre-computes the information once (i.e. if it is not already available in cache). Thus, multiple algorithms can benefit from a common preconditioning of the data structure. Moreover, another benefit of this strategy is that it adapts the memory footprint of the data structure, based on the types of traversals required by the calling algorithm.

In the specific case of regular grids, adjacency relations can be easily inferred, given the regular pattern of the grid sampling (considering the Freudenthal triangulation [Fre42, Kuh60] of the grid). Then, TTK's triangulation supports an *implicit* mode for regular grids: for such inputs, the preconditioning does not store any information and the results of all the queries are computed on-the-fly at runtime [TFL+17]. An extension to periodic grids (i.e. with periodic boundary conditions, for all dimensions) is also implemented. The switch from one implementation to the other (explicit mode for meshes or implicit mode for grids) is automatically handled by TTK and developers of topological algorithms only need to produce one implementation, interacting with TTK's generic triangulation data structure.

2.3.3 Existing TDA software frameworks

Numerous open-source software implementations exist to perform TDA computations, however they often focus on a particular topological abstraction or narrowly defined set of methods, such as the computation and study of persistent homology. The input may differ from one implementation to the next.

Some implementations focus on low-dimensional manifolds, studying topological representations such as critical points, persistence diagrams or Morse-Smale Complexes. For example, Dillard's library *libtourtre* computes the contour tree [Dil07], and Doraiswamy et al.'s *libRG* library [DN12a] and *Recon* [DN12b] as well as Tierny's *vtkReebGraph* [TGSP09] compute the Reeb graph. On the side of the Morse-Smale Complex computation, Shivashankar and Natarajan developed a scriptable implementation for it [SN17] and Sousbie created *DisPerSE*, an implementation tailored for cosmological data analysis [Sou11].

Other TDA softwares focus on the persistent homology of high-dimensional point clouds. One of the earliest is Mapper [SMC07]. Dionysus2 [Mor17] and JavaPlex [TVJA14] implement the standard algorithm nitroduced by Zomorodian and Carlsson [ZC05]. Perseus [Nan21] implements a topology-preserving preprocessing procedure to reduce the number of filtered input cells [MN12]. Bauer's *Ripser* focuses on the fast computation of persistent homology for the Vietoris-Rips filtration [Bau19]. Gudhi [MBGY14] supports persistent homology on simplicial complexes using the *Simplex tree* data structure [BM14]. Additional operations, such as statistical tools on the persistence, and data structures, such as the Toplex Map, were later added to this project. PHAT [BKRW17] computes efficiently persistent homology through the reduction of boundary matrices.

Many of those TDA tools rely on custom file formats [Dil07, DN12b, DN12a, Sou11, Nan21]. Though significant efforts were conducted to improve the accessibility of several of the mentioned softwares by making them accessible through Python (with *Ripser.py* [TSBO18] and *PHAT.py* [Kel17]) or R packages [Fou] (with Gudhi, Dionysus2, and PHAT in Fasy et al.'s TDA package [FKLM14]), most also require command-line usage or programming knowledge, limiting accessibility for users without technical backgrounds. Additionally, they often lack versatility in handling different data types and dimensions. For example, the Morse-Smale complex implementations by Shivashankar and Natarajan [SN17] are restricted to 2D

triangulations or 3D regular grids. In contrast, TTK is built for seamless integration, supporting dependency-free C++ code and accommodating various data formats and dimensions. Its close integration with ParaView also enables intuitive use by non-programmers. Furthermore, of the mentioned softwares, only Gudhi, PHAT and Dionysus2 are currently actively maintained. In contrast to these three packages, TTK specifically targets low dimensional (2D or 3D) domains for applications in scientific data analysis and visualization and does not solely focus on the computation of persistent homology.

2.3.4 Shared-memory parallelism for TDA

To improve the time efficiency of the algorithms computing the topological representations presented in Section 2.1, a significant effort has been carried out to re-visit TDA algorithms for shared-memory parallelism. Several authors focused on the shared-memory computation of the persistence diagram [BKRW17, GVT23], others focused on the merge and contour trees [MDN12, AN15, GFJT16, SM17, CWS+21, GFJT19a] or the Reeb graph [GFJT19b], while several other approaches have been proposed for the Morse-Smale complex [RWS11, SN12, GBP19]. Recently, a localized approach based on shared-memory parallelism has been introduced for the on-the-fly triangulation connectivity computation [LI24]. In terms of TDA packages, many implement shared-memory parallelism on some part of their code. Of the three packages still active, Gudhi and PHAT support parallelism, either by implementing the parallelism within the project or by relying on libraries that implement it. Dionysus2 does not seem to offer parallelism.

In line with these developments, TTK has adopted shared-memory parallelism, using OpenMP. It provides shared-memory parallel computations for various objects, including the following, non-exhaustive list: continuous scatterplots [BW08], data or geometry smoothing, dimensionality reduction [DTS+20], fiber surfaces [KTCG17], Jacobi sets [EH04], mandatory critical points [GST14], marching tetrahedra, merge and contour trees [GFJT19a, LWW+23], merge tree distances and encoding [PVDT22, WPTG23, PVT23, PT24], Morse-Smale complexes [TFL+17], path compression [MLT+23], persistence diagrams [GVT23], persistence diagram encoding [SDT24], Reeb graphs [GFJT19b], Reeb spaces [TC16], Rips complexes, scalar field normalizer, topological compression [SPCT18b], topo-

logical simplification [TP12, LGMT20]. Some of those parallel implementations use OpenMP tasks, such as [GFJT19a, PVDT22, PVT23].

While the above parallel approaches succeed in improving computation times, they still require a shared-memory system, capable of storing the entire input dataset into memory. Thus, when the size of the input dataset exceeds the capacity of the main memory of a single computer, distributed-memory approaches need to be considered. Moreover, provided that the performance of these distributed approaches scales with the number of nodes, they also contribute to reducing computation times.

3 A Software Framework for Distributed Topological Analysis Pipelines

Contents

This chapter presents the technical foundations for the extension of the *Topology ToolKit* (TTK) to distributed-memory parallelism with MPI. Most of TTK's algorithms support shared-memory parallelism using multiple threads with OpenMP, however, TTK did not support, up to now, distributed-memory parallelism and thus, was restricted to datasets of limited size, fitting in the memory of a single computer. We address this limitation in this chapter by extending TTK to distributed-memory parallelism, an addition that also enables more performance and a faster execution. Furthermore, while several recent papers introduced topology-based approaches for distributed-memory environments, these were reporting experiments obtained with tailored, mono-algorithm implementations. In contrast, we describe in this chapter a versatile approach (supporting both triangulated domains and regular grids) for the support of topological analysis *pipelines*, i.e., a sequence of topological algorithms interacting together, possibly on distinct numbers of processes. While developing this extension, we faced several algorithmic and software engineering challenges, which we document in this chapter. Specifically, we describe an MPI extension of TTK's data structure for triangulation representation and traversal, a central component to the global performance and generality of TTK's topological implementations. We also introduce an intermediate interface between TTK and MPI, both at the global pipeline level, and at the fine-grain algorithmic level. We provide a taxonomy for the distributed-memory topological algorithms supported by TTK, depending on their communication needs and provide examples of hybrid MPI+thread parallelizations. Detailed performance analyses show that parallel efficiencies range from 20% to 80% (depending on the algorithms), and that the MPI-specific preconditioning introduced by our framework induces a negligible computation time overhead. We illustrate the new distributed-memory capabilities of TTK with an example of advanced analysis pipeline, combining multiple algorithms, run on the largest publicly available dataset we have found (120 billion vertices) on a standard cluster with 64 nodes (for a total of 1536 cores).

The work presented in this chapter has been published in the journal IEEE Transactions on Visualization and Computer Graphics [LWG+24]

and presented at the IEEE VIS 2024 conference. It was certified replicable by the Graphics Replicability Stamp Initiative (http://www.replicabilitystamp.org/). An example of use is available online (https://github.com/eve-le-guillou/TTK-MPI-at-example). Our implementation is integrated in TTK (starting version 1.2.0).

3.1 Outline

This chapter documents the technical foundations which are required for the extension of TTK to distributed-memory parallelism using multiple processes with MPI, hence enabling the design of topological pipelines for the analysis of large-scale datasets on supercomputers. Specifically, after formalizing our conceptual model for the distributed representation of the input and output data (Section 3.2), we present the extension of TTK's internal triangulation data-structure (a central component of its performance and versatility) to the distributed setting (Section 3.3). We also document an interface between TTK and MPI (Section 3.4) enabling the consistent combination of multiple topological algorithms within a single, distributed pipeline. Unlike previous work (Section 3.1.1), this chapter does not focus on the distributed computation of a specific topological object (such as merge trees or persistence diagrams). Instead, it documents the necessary building blocks for the extension to the distributed setting of a diverse collection of topological algorithms such as TTK. To evaluate the efficiency of our extension, we document several examples (Section 3.5), extending to the distributed setting a selection of topological algorithms. We also provide a taxonomy of TTK's topological algorithms (Section 3.5.1), depending on their communication needs and provide examples of hybrid MPI+thread parallelizations for each category (Section 3.5.3), with detailed performance analyses (Section 3.6.1). We illustrate the new distributed capabilities of TTK with an example of advanced analysis pipeline (Section 3.5.4), combining multiple algorithms, run on a dataset of 120 billion vertices distributed on 64 nodes (Section 3.6.2) of 24 cores each. This work has been integrated in the main source code of TTK and is available in open-source.

3.1.1 Related work for distributed-memory TDA methods

Though numerous topological analysis methods have been adapted for shared-memory parallelism (see Section 2.3.4), fewer approaches have been documented for the computation of topological data representations in a distributed-memory environment. This is partly because the algorithmic advances in parallelism described for shared-memory approaches do not directly translate to a distributed environment. Indeed, a key to the performance of the shared-memory approaches discussed above is the ability of a thread to access any arbitrary element in the input dataset. It

also allows for easily implementable and efficient dynamic load balancing across threads.

In contrast, in a distributed setup, the initial per-process decomposition of the input dataset is often a *given*, which the topological algorithm cannot modify easily and which is likely to be unfavorable to its performances. Moreover, since most topological algorithms need to consider the input dataset in its globality, communications and synchronizations between processes have to be considered. However, these can be highly unfavorable to the performances of the algorithm and should be minimized. Then, existing efforts for distributing TDA approaches typically consist in first computing a *local* topological representation (i.e. persistence diagram, contour tree, etc.) given the local block of input dataset accessible to the process and then, in a second stage, to aggregate the local representations into a common *global* representation while attempting to minimize communications between processes (which are much more costly than synchronizations in shared-memory parallelism). Note that in several approaches [MW13, MW14, HKP+21], the final *global* representation may not be strictly equivalent to the output obtained by a traditional sequential algorithm, but more to a distributed representation, capable of supporting access queries by post-processing algorithms in a distributed fashion. Following the above general strategy, approaches have been documented for the distributed computation of the persistence diagram [BKR14b] as well as the merge and contour trees [PC04, MW13, MW14, NM19, WG21, HKP+21, CRW22]. However, these efforts focused on tailored implementations (i.e. supporting a single algorithm, typically restricted to regular grids), which neither needed to interact with other algorithms within a single analysis pipeline, nor to support compatibility with outputs computed sequentially. For instance, DIPHA [BKR14b] focuses on persistence diagram computation. For that, it relies on a data representation based on the boundary matrix of the input filtration, which is versatile, but at the expense of a potentially high memory footprint. Moreover, this representation is not accompanied by any mesh traversal functionality. Reeber [NM20, NM19] focuses on merge tree computation. It is tailored for regular grids (with optional support of adaptive mesh refinement via AMReX [ZAB+19]) and its data structure only models vertex adjacency relations (which is the only traversal functionality required for merge tree computation).

In contrast to mono-tailored implementations [NM20, NM19, BKR14b], our work provides a data-structure (Section 3.3) which is versatile, com-

pact and time-efficient, flexible, and conducive to pipeline re-use (it consistently maintains global indices for each simplex, irrespective of the number of processes). A necessary building block for distributing TDA algorithms is an infrastructure supporting a distributed access to the input dataset. To support topological algorithms, a data structure must be available to efficiently traverse the input dataset, with possibly advanced traversal queries. TTK [TFL+17, BMBF+19] implements such a triangulation data structure, providing advanced, constant-time, traversal queries, supporting both explicit meshes as well as the implicit triangulation of regular grids (with no memory overhead). While several data structures have been proposed for the distributed support of meshes [EWS+10, ISSS16, ZAB+19] (with a focus on simulation driven remeshing), we consider in this work the distribution of TTK's triangulation data structure (Section 3.3), with a strong focus on traversal time efficiency and compatibility with a non-distributed usage, to support post-processing interactive sessions on a workstation (c.f. Section 3.2).

3.1.2 Contributions

This chapter makes the following new contributions.

1. *An efficient, distributed triangulation data structure* (Section 3.3): We introduce an extension of TTK's triangulation data structure for the support of distributed datasets.

2. *A software infrastructure for distributed topological pipelines* (Section 3.4): We document a software infrastructure consistently supporting advanced, distributed topological pipelines, consisting of multiple algorithms, possibly run on a distinct number of processes.

3. *Examples of distributed topological algorithms* (Section 3.5): We provide a taxonomy of the algorithms supported by TTK, depending on their communication needs, and document examples of distributed parallelizations, with detailed performance analyses, following an MPI+thread strategy. This includes an advanced pipeline consisting of multiple algorithms, run on a dataset of 120 billion vertices on a compute cluster with 64 nodes (1536 cores, total).

4. *An open-source implementation*: Our implementation is integrated in TTK 1.2.0, to enable others to reproduce our results or extend TTK's distributed capabilities.

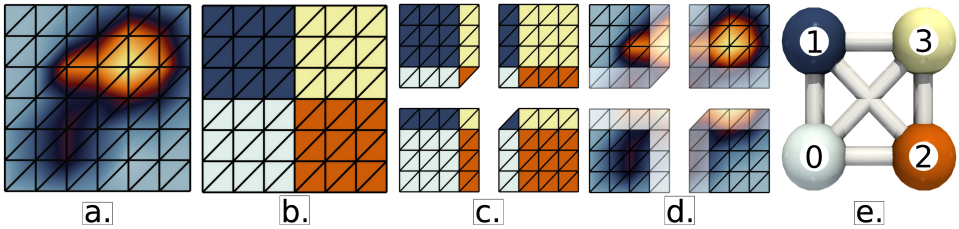


Figure 3.1 – *The input data* (a) *is assumed to be loaded in the memory of n_p independent processes in the form of n_p disjoint* blocks *of data* ((b), *one color per block, $n_p = 4$ in this example). A layer of* ghost *simplices* ((c), *coming from adjacent blocks, matching colors) is added to each block. This local data duplication* ((d), *transparent) eases subsequent processing on block boundaries. A local adjacency graph is constructed to encode local neighbor relations between blocks* (e).

5. *A reproducible example:* We provide a reference Python script of one of our advanced pipelines for replicating our results with a dataset size that can be adjusted to fit the capacities of any system (publicly available at: https://github.com/eve-le-guillou/TTK-MPI-at-example).

3.2 Distributed Model

We now formalize our distributed model, which will eventually be used as a blueprint to port the algorithms described above (Section 2.1) to distributed computations (Section 3.5).

3.2.1 Input distribution formalization

3.2.1.1 Decomposition

Our distributed-memory model is based the following convention. f is assumed to be loaded in the memory of n_p processes in the form of n_p disjoints *blocks* of data (Figure 3.1*(a-b)*). Specifically, each process $p \in \{0, \ldots, n_p - 1\}$ is associated with a local block $f_p : \mathcal{M}_p \to \mathbb{R}$, such that:

- $\mathcal{M}_p \subset \mathcal{M}$: each block $\mathcal{M}_p$ is a d-dimensional simplicial complex, being a subset of the global input $\mathcal{M}$.
- Any simplex σ present in multiple blocks (e.g. at the boundary between adjacent blocks) is said to be *exclusively owned*, by convention, by the process with the lowest identifier (among the processes containing σ).
- A simplex $\sigma \in \mathcal{M}_p$ which is not exclusively owned by the process p is called a *ghost simplex* (subsubsection 3.2.1.2).

- $\cup_{\mathcal{M}_p} = \mathcal{M}$: the union of the blocks is equal to the input.

3.2.1.2 Ghost layer

In such a distributed setting, *ghost* simplices are typically considered, in order to save communications between processes for local tasks. Ghost simplices are typically simplices inside the block of a process that are copies of the interfacing simplices of an adjacent block (see the lighter simplices in Figure 3.1, (d)). We note $\mathcal{M}'_p$ the d-dimensional simplicial complex obtained by considering a layer of ghost simplices, i.e. by adding to $\mathcal{M}_p$ the d-simplices of $\mathcal{M}$ which share a face with a d-simplex of $\mathcal{M}_p$, along with all their d'-dimensional faces (with $d' \in \{0, \dots, d-1\}$). Overall, all the simplices added in this way to the block $\mathcal{M}_p$ to form the *ghosted block* $\mathcal{M}'_p$ are *ghost simplices* (Figure 3.1*(c-d)*).

The usage of such a ghost layer is typically motivated in practice by algorithms which perform local traversals (e.g. PL critical point extraction, subsubsection 2.1.2.1). Then, when such algorithms reach the boundary of a block, they can still perform their task without any communication, thanks to the ghost layer. Also, the usage of a ghost layer facilitates the identification of boundary simplices (i.e. located on the boundary of the *global* domain $\mathcal{M}$, see subsubsection 3.3.1.1).

The blocks are also positioned in relation to one another. Processes p and p will be considered adjacent (Figure 3.1*(e)*) if $\mathcal{M}'_p$ contains d-simplices that are exclusively owned by q and if $\mathcal{M}'_q$ contains d-simplices that are exclusively owned by p.

3.2.1.3 Global simplex identifiers

For any $d' \in \{0, \dots, d\}$, each d'-simplex σ_j of each block $\mathcal{M}'_p$ is associated with a *local* identifier $j \in [0, |\mathcal{M}'^{d'}_p| - 1]$. This integer uniquely identifies σ_j within the local block $\mathcal{M}'_p$.

The simplex σ_j is also associated with a *global* identifier $\phi_{d'}(j) \in [0, |\mathcal{M}^{d'}| - 1]$, which uniquely identifies σ_j within the *global* dataset $\mathcal{M}$. Such a global identification is motivated by the need to support varying numbers of processes. In particular, assume that a first analysis pipeline P_1 (for instance extracting critical vertices, subsubsection 2.1.2.1) uses $n_p(P_1)$ processes to generate an output (e.g. the list of critical vertices). Let us consider now a second analysis pipeline P_2 using $n_p(P_2)$ processes (possibly on a different machine) to post-process the output of P_1 (for instance, seeding integral lines, subsubsection 3.5.3.5, at the previously extracted critical

vertices). Since $n_p(P_1)$ and $n_p(P_2)$ differ between the two sub-pipelines, their input decompositions into local blocks will also differ. Then the local identifiers of the critical vertices employed in P_1 may no longer be usable in P_2. For instance, if $n_p(P_1) < n_p(P_2)$, the local blocks of P_2 may be much smaller than those of P_1 and the local identifiers of P_1 can become out of range in P_2. Thus, a common ground between the two pipelines need to be found to reliably exchange information, hence the global, unique identifiers.

Note that the support for a varying number of processes is a necessary feature for practical distributed topological algorithms. While it is a challenging constraint (c.f. Section 3.3), it is beneficial to various application use cases. For instance, P_2 can be a post-processing pipeline run on a workstation. P_2 can also be executed on a different (possibly larger) distributed-memory system than P_1. Last, P_1 and P_2 can be part of a single, large pipeline, which would include an aggregation step of the outputs of P_1 to a different number of processes ($n_p(P_2)$).

3.2.1.4 Simplex-to-process maps

Each block $\mathcal{M}'_p$ is associated with *simplex-to-process maps*, which map each simplex to the identifier of the process which exclusively owns it.

3.2.2 Output distribution formalization

Topological algorithms typically consume an input (possibly complex), to produce a (usually) simpler output (such as the topological representations described in Section 2.1). Moreover, multiple topological algorithms can be combined sequentially to form an analysis pipeline. For instance, a first algorithm A_1 may compute integral lines (subsubsection 3.5.3.5) for a first field f, while a second algorithm A_2 may extract the critical vertices (subsubsection 2.1.2.1) for a second field g, defined on the integral lines generated by the first algorithm A_1. Thus, the output produced by a distributed topological algorithm A_1 *must* be readily usable by another distributed algorithm A_2.

This implies that the output computed by a topological algorithm must also strictly comply to the input specification (Section 3.2.1) and should contain: *(i)* a ghost layer, *(ii)* global simplex identifiers, and *(iii)* simplex-to-process maps.

Note that, according to this formalism, the output of a topological algorithm *is* distributed among several processes. Depending on the complex-

ity of this output, specialized manipulation algorithms (handling communication between processes) may need to be later developed to exploit them appropriately in a post-process.

3.2.3 Implementation specification

We now review the building blocks which are necessary to support the distributed model specified in Secs. 3.2.1 and 3.2.2.

The pipeline combining the different topological algorithms can be encoded in the form of a Python script (c.f. contribution 5, Section 3.1.2). The initial decomposition of the global domain $\mathcal{M}$ and the ghost layer (specifically, the ghost vertices and the ghost d-simplices) are computed by ParaView [AGL05]. Then, the TTK algorithms present in the pipeline will be instantiated by ParaView on each process and from this point on, they will be able to access their own local block of *ghosted* data and communicate with other processes.

While ParaView offers in principle the possibility to compute vertex-to-process maps, we have observed several inconsistencies (in particular when using ghost layers), which prevented us to use it reliably. This required us to develop our own process identification strategy (Section 3.4.2).

Moreover, while ParaView also offers in principle the possibility to generate global identifiers for vertices and cells (i.e. d-simplices), we have experienced technical difficulties with it (such as a dependence of the resulting identifiers on the number of processes), as well as issues which made it unusable for large-scale datasets (such as an excessively large memory footprint). This required us to develop our own strategy for the global identification of vertices and cells (i.e. d-simplices), documented in Secs. 3.3.1 and 3.3.2.

The input PL scalar field f is required to be injective on the vertices. This can be easily obtained via lexicographic vertex comparison, by considering for each vertex v the tuple $\big(f(v), \phi_0(v)\big)$, i.e. the tuple formed by its scalar value and its global identifier. In practice, to accelerate these comparisons for *local* vertices (i.e. vertices present in a common block $\mathcal{M}'_p$), the process p will first sort all its local vertices (in lexicographic order) in a preconditioning step, and local vertex comparisons will later be based on their order in the sorted list.

Section 3.3 documents the extension of TTK's triangulation data struc-

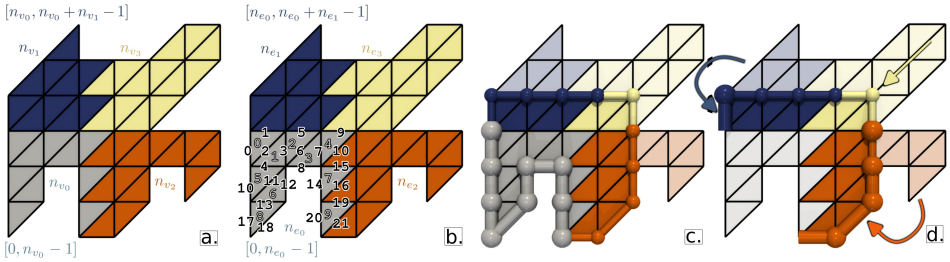


Figure 3.2 – *Preconditioning of our distributed explicit triangulation.* (a) *Each process p enumerates its number n_{v_p} of exclusively owned vertices and d-simplices. Next, an MPI prefix sum provides a local offset for each process to generate global identifiers.* (b) *For each process p, simplices of intermediate dimensions (edges (n_{e_p}), triangles) are locally enumerated for contiguous intervals of global identifiers of d-simplices (white numbers). Next, all the intervals are sent to the process 0 which sorts them first by simplex-to-process identifier, then by interval start, yielding a per-interval offset that each process can use to generate its global identifiers (black numbers).* (c) *Within a given block, the vertices at the boundary of the domain $\mathcal{M}$ are identified as non-ghost boundary vertices (large spheres). Next, a simplex which only contains boundary vertices is considered to be a boundary simplex (larger cylinders).* (d) *The global identifiers and boundary information of the ghost simplices are retrieved through MPI communications with the neighbor processes. The ghost simplices on the global boundary are flagged as boundary simplices (larger spheres and cylinders).*

ture to support our model of distributed input and output (Secs. 3.2.1 and 3.2.2).

Additional procedures easing the combination of multiple algorithms into a single pipeline (adjacency graph computation, ghost data exchange) are documented in Section 3.4.2.

3.3 Distributed Triangulation

This section describes the distributed extension of TTK's triangulation data structure (see subsubsection 2.3.2.2 for the initial triangulation design), later used by each topological algorithm. In the following, we assume that the input block is loaded in the memory of the local process p and ghosted (i.e. we consider the ghosted block $\mathcal{M}'_p$, subsubsection 3.2.1.2). Moreover, we consider that, for each process p, a list of *neighbor processes* is available (Figure 3.1*(e)*).

3.3.1 Distributed explicit triangulation

This section describes our distributed implementation of the TTK triangulation in explicit mode, i.e. when an explicit simplicial complex is provided as a global input.

3.3.1.1 Distributed explicit preconditioning

The preconditioning of explicit triangulations in the distributed setting involves the computation of four main pieces of information: ***(1)*** global identifiers, ***(2)*** ghost global identifiers, ***(3)*** boundary, and ***(4)*** ghost boundary.

***(1)* Global identifiers:** The first step consists in determining global identifiers for the vertices (i.e., the map ϕ_0, Section 3.2.1, its inverse, ϕ_0^{-1}). This step is not optional and is triggered automatically. For each ghosted block $\mathcal{M}'_p$, the number n_{v_p} of *non-ghost* vertices that the block exclusively owns is computed (Figure 3.2*(a)*). Next, an MPI prefix sum is performed to determine the offset that each block p should add to its local vertex identifiers to obtain its global vertex identifiers. The map ϕ_d and its inverse ϕ_d^{-1} are computed similarly.

Next, global identifiers need to be computed for the d'-simplices of intermediate dimension (i.e. $d' \in \{1, \ldots, d-1\}$, Figure 3.2*(b)*)). This step is optional and is only triggered if the calling algorithm pre-declared the usage of these simplices in the preconditioning phase.

For this, each process p first identifies, among its list of exclusively owned d-simplices, intervals of contiguous global identifiers. These are typically interleaved with global identifiers of ghost d-simplices. Then, intervals are processed independently via shared-memory parallelism, and for each interval x, the d'-simplices are provided with a local identifier (with the same procedure as used in the non-distributed setting). Given a d'-simplex σ at the interface between two blocks (i.e. σ is a face of a ghost d-simplex), a tie break strategy needs to be established, to guarantee that only one process tries to generate an identifier for σ. Specifically, the process p will generate an identifier for σ only if p is the lowest simplex-to-process identifier among the exclusive owners of the d-simplices in $St(\sigma)$ (Section 3.2.1). Next, all the intervals (along with their simplex-to-process identifier and number of d'-simplices) are sent to the process 0 which, after ordering the intervals of d-simplices first by simplex-to-process identifier then by local identifier, determines the offset that each interval x should add to its local d'-simplex identifiers to obtain its global identifiers.

***(2)* Ghost global identifiers:** The second step of the preconditioning consists in retrieving for a given block $\mathcal{M}'_p$ the global identifiers of its ghost $0-$ and $d-$simplices. This step is not optional and is always triggered. This feature can be particularly useful when performing local computations on the boundary of the block (e.g. discrete gradient, Section 3.5).
Once all the processes have established their vertex global identifiers, each process p queries each of its neighbor processes j, to obtain the global identifiers of its ghost vertices (a KD-tree data-structure is employed to establish, with shared-memory parallelism, the correspondence between vertices coming from different blocks). Once global vertex identifiers are available for the ghost vertices of $\mathcal{M}'_p$, a simpler exchange procedure is used to collect the global identifiers of the ghost d'-simplices with $d' \in \{1, \ldots, d\}$ (the correspondence between d'-simplices coming from different blocks is established, with shared-memory parallelism, based on the global identifiers of their vertices).
***(3)* Boundary:** The third step consists in determining the simplices which are on the boundary of the global domain $\mathcal{M}$. This step is optional and is only triggered (on a per simplex dimension basis) if the calling algorithm pre-declared the usage of boundary simplices in the preconditioning phase. This feature is particularly useful for algorithms which process as special cases the simplices which are on the boundary of $\mathcal{M}$ (e.g. critical point extraction, Section 3.5).
Each process p identifies the boundary vertices of its *ghosted block* $\mathcal{M}'_p$ (See Figure 3.2*(c)*), with exactly the same procedure as the one used in the non-distributed setting [TFL$^+$17]. Then, thanks to the ghost layer, it is guaranteed that among the set of boundary vertices identified above, the non-ghost vertices are indeed on the boundary of the global domain $\mathcal{M}$. Finally, a d'-simplex will be marked as a boundary simplex if all its vertices are on the boundary of $\mathcal{M}$.
***(4)* Ghost boundary:** Similarly to step *(2)*, a final step of data exchange between the process p and its neighbors enables the retrieval of the ghost simplices of $\mathcal{M}'_p$ which are also on the boundary (Figure 3.2*(d)*). This step is optional and is only triggered if the calling algorithm pre-declared the usage of boundary simplices in the preconditioning phase.

Finally, the preconditioning of any other traversal routine is identical to the non-distributed setting.

3.3.1.2 Distributed explicit queries

In this section, we describe the implementation of the traversals of the triangulation, as queried by a calling algorithm. This assumes that the calling algorithm first called the appropriate preconditioning functions in a pre-process.

The traversal of a local ghosted block $\mathcal{M}'_p$ by an algorithm instantiated on the process p is performed identically to the non-distributed setting, with local simplex identifiers. This requires the calling algorithm to locally translate input (and output) *global* simplex identifiers into *local* ones (i.e. with the maps introduced in subsubsection 3.2.1.3).

3.3.2 Distributed implicit triangulation

This section describes our distributed implementation of the TTK triangulation in implicit mode, i.e. when a regular grid is provided as a global input. Then, as described below, most traversal information can be computed on-the-fly at runtime, given the regular sampling pattern of the Freudenthal triangulation [Fre42, Kuh60] of the input grid.

3.3.2.1 Distributed implicit preconditioning

In implicit mode, the preconditioning of the triangulation identifies the position of the local ghosted grid $\mathcal{M}'_p$ within the global grid $\mathcal{M}$, as detailed in Figure 3.3. This step is not optional and is triggered automatically. The preconditioning of any traversal routine returns immediately without any processing (all queries are computed on-the-fly).

3.3.2.2 Distributed implicit queries

In this section, we describe the implementation of the traversals of the triangulation, as queried by a calling algorithm.

The traversal of a local ghosted block $\mathcal{M}'_p$ by an algorithm instantiated on the process p is performed identically to the non-distributed setting, with local simplex identifiers.

Similarly to the explicit case (subsubsection 3.3.1.2), the calling algorithm must now translate input (and output) *global* simplex identifiers into *local* ones (i.e. with the maps from subsubsection 3.2.1.3).

The important difference with the explicit mode is that all the information computed in explicit preconditioning (i.e. ***(1)*** global identifiers, ***(2)*** ghost global identifiers, ***(3)*** boundary, and ***(4)*** ghost boundary, see subsub-

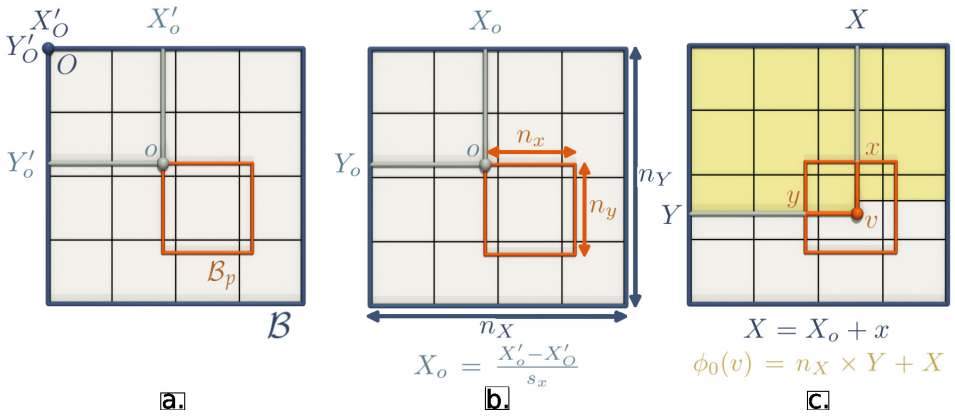


Figure 3.3 – *Preconditioning of our distributed implicit triangulation.* (a) *Each process p computes (with shared-memory parallelism) the bounding box $\mathcal{B}_p$ of its ghosted block $\mathcal{M}'_p$. The vertex o, respectively O, is the origin of $\mathcal{M}'_p$, respectively $\mathcal{M}$, with (X'_o, Y'_o, Z'_o), respectively (X'_O, Y'_O, Z'_O), its floating-point coordinates. The bounding box $\mathcal{B}$ of $\mathcal{M}$ is computed (via MPI parallel reductions) from all the local $\mathcal{B}_p$.* (b) *Two key pieces of information are computed at this step: the dimensions of the global grid (n_X, n_Y, n_Z) (the number of vertices of $\mathcal{M}$ in each direction) and the local grid offset (X_o, Y_o, Z_o) (the global discrete coordinates of o). It is computed from (X'_O, Y'_O, Z'_O), (X'_o, Y'_o, Z'_o) and the floating-point spacing of the grid (s_x, s_y, s_z). Following that, each process locally instantiates a global implicit triangulation model of $\mathcal{M}$.* (c) *Given a local vertex identifier, its global discrete coordinates (X, Y, Z) in $\mathcal{M}$ are inferred from its local discrete point coordinates (x, y, z) (with $x \in [0, n_x - 1]$, $y \in [0, n_y - 1]$, and $z \in [0, n_z - 1]$, n_x, n_y and n_z being the number of vertices of the grid $\mathcal{M}'_p$ in each direction), and its local grid offsets. Next, its global identifier, $\phi_0(v)$, is determined on-the-fly by global row-major indexing.* $\mathcal{B}_p$

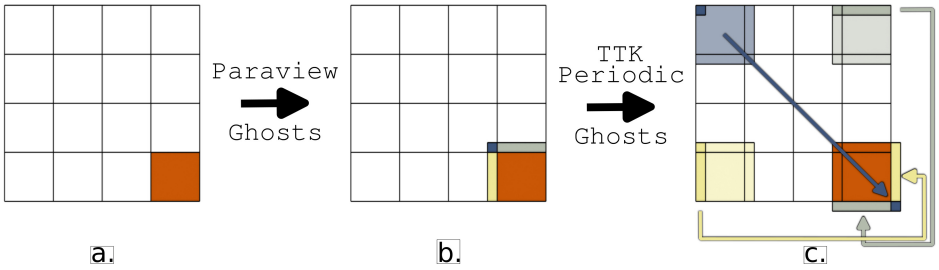


Figure 3.4 – *Preconditioning of our distributed periodic implicit triangulation. This triangulation type is handled similarly to the implicit case, but additional ghost simplices need to be computed. Given a data block $\mathcal{M}_p$ ((a), orange), ParaView generates a first layer of ghost d-simplices ((b), blue, grey, yellow). If $\mathcal{M}_p$ was located on the boundary of the global grid $\mathcal{M}$, periodic boundary conditions must be considered by adding an extra layer of ghost d-simplices (arrows) for each periodic face of $\mathcal{M}$ (c).*

section 3.3.1.1) now needs to be computed on-the-fly at runtime (i.e. upon the query of this information by the calling algorithm).

***(1)* Global identifiers:** As detailed in Figure 3.3, given a local vertex v, its global discrete coordinates (X, Y, Z) in the global grid $\mathcal{M}$ are inferred from its local discrete point coordinates (x, y, z) in $\mathcal{M}'_p$ (Figure 3.3*(c)*), and the local grid offset (X_o, Y_o, Z_o). From the coordinates (X, Y, Z), the global identifier of v is computed on-the-fly with the procedure used in the non-distributed setting [TFL+17] (global row-major indexing). The same procedure is used for d-simplices.

The global identifier of any d'-simplex ($d' \in \{1, \ldots, d-1\}$) is computed by identifying the d'-simplex in $\mathcal{M}$ which has the same global vertex identifiers (via vertex star inspection).

***(2)* Ghost global identifiers:** The global identifier of a ghost simplex is also computed with the above procedure.

***(3)* Boundary:** To decide if a given d'-simplex is on the boundary of $\mathcal{M}$, its global identifier is first retrieved (above) and the local copy of the global grid $\mathcal{M}$ is queried for boundary check based on this global identifier (with the exact procedure used in the non-distributed setting [TFL+17]).

***(4)* Ghost boundary:** The boundary check for ghost simplices is also computed with the above procedure.

3.3.3 Distributed implicit periodic triangulation

Periodic grids (with periodicity in all dimensions) are supported via implicit Freudenthal triangulation [Fre42, Kuh60] like in the previous section. However, the periodic boundaries require specific adjustments in terms of preconditioning.

Since ParaView's ghost cell generator only produces ghosts at the interface of the domain of processes, an extra layer of ghost simplices needs to be computed, as illustrated in Figure 3.4. Specifically, each process p checks if its block $\mathcal{M}'_p$ is located on the boundary of the global grid $\mathcal{M}$ (via bounding box comparison). If so, the list of *periodic faces* of the bounding box $\mathcal{B}$ of $\mathcal{M}$ along which $\mathcal{M}'_p$ is located is identified (i.e. left, right, bottom, top, front, back). This information is used to trigger exchanges of data chunks, as illustrated in Figure 3.4*(c)*, whose extent depends on the periodic face type (corner, edge, face). Additionally, the local adjacency graph is updated to account for blocks which are adjacent via the periodic boundaries.

Similarly to Section 3.3.2, runtime queries are performed on each process by querying the local copy of the global periodic triangulation $\mathcal{M}$ (with the necessary local-to-global identifier translations).

3.4 Distributed Pipeline

This section provides an overview of the overall processing by TTK of a distributed dataset. It documents the preconditioning steps handled by the core infrastructure of TTK (beyond the triangulation handling, Section 3.3) in order to complete the support of the distributed model specified in Section 3.2.

3.4.1 Overview

The input data is provided in the form of a distributed dataset (see Section 3.2.1) loaded from a filesystem (e.g. *PVTI* file format) or provided in-situ (e.g. with *Catalyst*). As shown in Figure 3.5, ParaView's execution flow enters TTK via the function `ProcessRequest`, which triggers TTK's preconditioning, including the *Distributed Pipeline Preconditioning* (specific to the distributed mode, top yellow frame) prior to the traditional, local preconditioning (middle yellow frame) and finally the implementation of the topological algorithm (bottom yellow frame). In the following, we describe the *Distributed Pipeline Preconditioning*.

***(1)* Ghost layer generation:** if the local data block does not include any ghost cells, the ghost layer generation algorithm (implemented by ParaView) is automatically triggered. This step is omitted if a valid ghost layer is already present.

***(2)* Local adjacency graph (LAG) initialization:** An estimation of the local

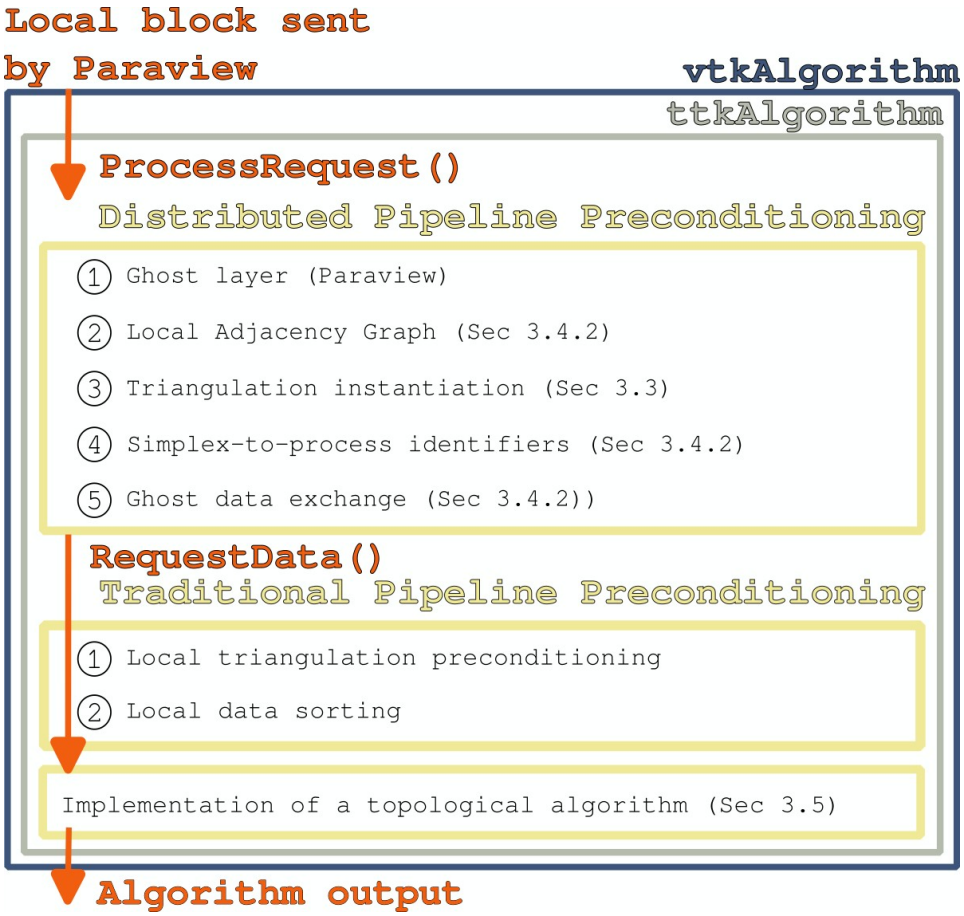


Figure 3.5 – *Overview of the overall pipeline upon the the delivery of a data block $\mathcal{M}_p$ by ParaView (top). A step of pipeline preconditioning specialized for the distributed setting (top yellow frame) is automatically triggered before calling the actual implementation of the topological algorithm. Note that each preconditioning phase is only triggered if the corresponding information has not been cached yet. Then, for practical pipelines, the preconditioning typically only occurs before the first algorithm of the pipeline.*

adjacency graph (i.e. connecting the data block to its neighbors) is constructed. This step (described in Section 3.4.2) is omitted if a valid LAG is already present.

***(3)* Triangulation instantiation:** this step instantiates a new TTK triangulation data structure (Section 3.3). This step is omitted if a valid triangulation is already present.

***(4)* Simplex-to-process map generation:** this step computes the simplex-to-process identifier for each simplex (as specified in Section 3.2.1). This step (described in Section 3.4.2) is omitted if valid simplex-to-process maps are already present.

***(5)* Ghost data exchange:** this step computes for each neighbor process q the list of vertices or cells exclusively owned by it, and which are ghosts in the process p. This step (described in Section 3.4.2) is optional and is only triggered if the calling algorithm pre-declared its usage at preconditioning.

After these steps, the traditional TTK preconditioning is executed (middle yellow frame, Figure 3.5).

3.4.2 Infrastructure details

This section describes the implementation of the pipeline preconditioning mentioned in the above overview (Section 3.4.1), specifically, the routines which are not directly related to the distributed triangulation (which has been covered in Section 3.3).

Local adjacency graph (LAG) initialization: Given a ghosted block $\mathcal{M}'_p$, the goal of this step is to store a list of processes, which are responsible for the blocks adjacent to $\mathcal{M}'_p$ (Figure 3.1*(e)*). First, each process p computes the bounding box $\mathcal{B}_p$ of its ghosted block $\mathcal{M}'_p$. Next, all processes exchange their bounding boxes. Finally, each process p can initialize a list of neighbor processes by collecting the processes whose bounding box intersects with $\mathcal{B}_p$. This first estimation of the LAG will be refined (next paragraph) after the generation of the simplex-to-process identifiers (which is relevant in the case of explicitly triangulated domains).

Simplex-to-process map generation: As specified in Section 3.2.1, each simplex is associated to the identifier of the process which exclusively owns it. This convenience feature can be particularly useful to quickly identify where to continue a local processing when reaching the boundary of a block (e.g. integral lines, subsubsection 3.5.3.5).

Each vertex $v \in \mathcal{M}'_p$ is classified by ParaView as ghost or non-ghost. For

each non-ghost vertex v, we set its simplex-to-process identifier to p. Then, the *ghost global identifier list* is computed (it contains the global identifiers of all the ghost vertices of $\mathcal{M}'_p$). Next, this list is sent to each process q marked as being adjacent in the LAG (previous paragraph). Then, the process q will return its identifier (q) and the subset of the *ghost global identifier list*, corresponding to non-ghost vertices in $\mathcal{M}'_q$. Finally, the process p will set the simplex-to-process identifier of v to q, for each vertex v returned by q. The procedure for the d-simplices is identical. The simplex-to-process maps for the simplices of intermediate dimensions are inferred from these of the d-simplices, as specified in Section 3.2.1. Following the generation of the simplex-to-process maps, the LAG is updated, by only considering the block p and q as neighbors if p contains ghost vertices which are exclusively owned by q and reciprocally.

In implicit mode, the preconditioning of the simplex-to-process map generation is limited to the computation of discrete bounding boxes (i.e. expressed in terms of global discrete coordinates) for the non-ghosted block $\mathcal{M}_p$. The bounding boxes are then exchanged between neighboring processes. Then, the simplex-to-process maps are inferred on-the-fly, at query-time, from the discrete bounding boxes.

Ghost data exchange: In many scenarios, it may be desirable to update the data attached to the ghost simplices of a given block $\mathcal{M}'_p$. For instance, when considering smoothing (subsubsection 3.5.3.4), at each iteration, the process p needs to retrieve the new, smoothed f data values for its ghost vertices, prior to the next smoothing iteration. We implement this task in TTK as a simple convenience function. First, using the list of neighbors (collected from the LAG), the process p will, for each neighbor process q, send the global identifiers of the simplices which are ghost for p and owned by q (using their simplex-to-process maps). This is computed once, in an optional preconditioning step (step 5, Section 3.4.1). This list of ghost vertex identifiers is cached in q and used at runtime, when necessary, to send to p the updated values (exchange data buffers are updated with shared-memory parallelism). A similar procedure is available for d-simplices.

3.5 Examples

Secs. 3.3 and 3.4 documented the implementation of the distributed model specified in Section 3.2. In this section, we now describe how to make use of this model to extend topological algorithms to the distributed set-

ting. Specifically, we will mostly focus on the algorithms described in Section 2.1.

3.5.1 Algorithm taxonomy

In this section, we present a taxonomy of the topological algorithms implemented in TTK, based on their needs of communications on distributed-memory architectures.

***(1)* No Communication (NC):** This category includes algorithms for which processes do not need to communicate with each other to complete their computation. This is the simplest form of algorithms and the easiest to extend to a distributed setting. Such algorithms are often referred to as *embarrassingly parallel*. In TTK, this includes algorithms performing local operations and generating a local output, e.g.: critical point classification subsubsection 2.1.2.1, discrete gradient computation subsubsection 2.1.2.3, Jacobi set extraction [EH04], Fiber surface computation [KTCG17] and marching tetrahedra.

***(2)* Data-Independent Communications (DIC):** This category includes algorithms for which processes do need to communicate with each other, but at predictable stages of the algorithm, with a predictable set of processes and communication volume, independently of the data values. This typically corresponds to algorithms performing a local operation on their block that need intermediate results from adjacent blocks to finalize their computation. In TTK, this includes for instance: data normalization, data or geometry smoothing (Section 3.5.3), or continuous scatter plots [BW08].

***(3)* Data-Dependent Communications (DDC):** This category includes algorithms which do not fall within the previous categories, i.e. for which communications can occur at unpredictable stages of the algorithm, with an unpredictable set of processes or communication volume, depending on the data values. This is the most difficult category of algorithms to extend to the distributed setting, since an efficient port would require a complete re-design of the algorithm. Unfortunately, we conjecture that most topological algorithms fall into that category. In TTK, this includes for instance: integral lines (subsubsection 2.1.2.2), persistence diagrams [GVT23], merge and contour trees [GFJT19a], path compression [MLT+23], Reeb graphs [GFJT19b], Morse-Smale complexes [TFL+17], Rips complexes, topological simplification [TP12, LGMT20], Reeb spaces [TC16], etc.

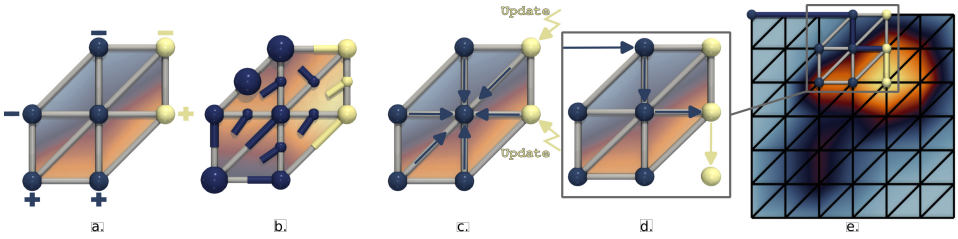


Figure 3.6 – *Examples of topological algorithm modifications for the support of distributed memory computation.* (a) *Scalar Field Critical Points (NC): Critical points are generated similarly to the sequential mode. Upper and lower links (+ and − signs in the figure) of non-ghost vertices on the boundary of $\mathcal{M}_p$ are computed using ghost vertices (here in yellow).* (b) *Discrete Gradient (NC): Similarly to (a), this algorithm processes each vertex of the domain independently. For each non-ghost vertex on the boundary of $\mathcal{M}_p$, the lower link computation can rely on ghost vertices. Critical simplices are represented by bigger spheres.* (c) *Scalar Field Smoother (DIC): This procedure smooths a scalar field f by local averaging for a user-defined number of iterations. The values of ghost vertices (in yellow) will need to be updated after each iteration.* (d) *and* (e) *Integral Lines (DDC):* (e) *each process will compute the integral lines whose seeds lie within its block $\mathcal{M}_p$. Then either the integral line reaches its final vertex within $\mathcal{M}_p$,* completing *the computation, or the integral line reaches a vertex outside of $\mathcal{M}_p$ (here in yellow in* (d)*). In the latter case, the integral line data is stored to be sent later to the yellow process. Once all the work is done on all processes, they exchange the data of incomplete integral lines and resume the computation of the integral lines on their block. The computation stops when all integral lines have completed.*

3.5.2 Hybrid MPI+thread strategy

As mentioned in Section 2.2.3, using an MPI+thread strategy can improve performance compared to a pure MPI configuration thanks to fewer MPI communications (due to fewer MPI processes) and to a (better) dynamic load balancing among threads within each MPI process. The overall memory footprint is also lower with the hybrid strategy, since using fewer MPI processes implies fewer ghost simplices and less data duplication.

Regarding the MPI+thread strategy and the port examples described in Section 3.5.3, we rely in TTK on the `MPI_THREAD_FUNNELED` thread support level in MPI [Mes23]. According to this level of thread support, only the master (i.e. original) thread can issue calls to MPI routines. In each port example, within each MPI process, the communication steps (if any) are thus performed in serial whereas the computation steps are multi-threaded, using the OpenMP implementations already available in TTK.

3.5.3 Distributed algorithm examples

We now illustrate the taxonomy of Section 3.5.1 by describing the distributed-memory parallelization of algorithms belonging to each of the categories, while exploiting the distributed model we introduced (Section 3.2).

3.5.3.1 NC: Scalar Field Critical Points

This algorithm processes each vertex v of the domain independently and performs the classification presented in subsubsection 2.1.2.1. Since it processes a local piece of data (the lower and upper links $Lk^-(v)$ and $Lk^+(v)$) and that it generates a localized output (a list of critical points for the local block), it does not require any communication (Figure 3.6*(a)*). Thus, it is classified in the category NC of the above taxonomy. To port this embarrassingly parallel algorithm to the distributed setting, two modifications are required. First, the algorithm does not classify ghost vertices (which will be classified by other processes). Second, to fulfill the distributed output specification (Section 3.2.2), each output critical point is associated with its *global* vertex identifier (instead of its local one).

3.5.3.2 NC: Discrete Gradient

Similarly to the previous case, this algorithm processes each vertex v of the domain independently. Specifically, it generates discrete vectors for the lower star $St^-(v)$ and the simplices which are assigned to no discrete vectors are stored as critical simplices (subsubsection 2.1.2.3). Similarly to the previous case, this algorithm only requires local data and only produces local outputs, without needing communications (hence its NC classification) (Figure 3.6 *(b)*). The port of this embarrassingly parallel algorithm requires two modifications. First, only the vertices which are exclusively owned by the current process (Section 3.2.1) are processed. The gradient for ghost vertices, and the simplices in their lower links, is not computed. Second, similarly to the previous case, the simplex identifiers associated with the discrete vectors and critical simplices are expressed with *global* identifiers (instead of local ones).

3.5.3.3 DIC: Scalar Field Normalizer

This procedure normalizes an input scalar field f to the range $[0, 1]$. It is divided into two steps. First, each process computes its local extreme values and all processes exchange their extreme values to determine the values f_{min} and f_{max} for the entire domain $\mathcal{M}$ using MPI collective communications. Second, all data values are normalized independently, based on f_{min} and f_{max}. The first step of the algorithm requires inter-process communications in a way which is predictable and independent of the actual data values (hence its DIC classification in the taxonomy).

3.5.3.4 DIC: Scalar Field Smoother

This procedure smooths a scalar field f by local averaging (i.e. by replacing $f(v)$ with the average data values on the vertices of $St(v)$). This averaging procedure is typically iterated for a user-defined number of iterations. However, at a given iteration, in order to guarantee a correct result for each vertex v located on the boundary of the local block (i.e. v is a non-ghost vertex adjacent to ghost-vertices), the updated f values from the previous iteration need to be retrieved for each of its ghost neighbors (Figure 3.6*(c)*). Thus, at the end of each iteration, each process p needs to communicate with its neighbors to retrieve the smoothed values for its ghost vertices, which is achieved by using the generic ghost data exchange procedure described in Section 3.4.2 (hence the DIC classification for this algorithm).

3.5.3.5 DDC: Integral lines

Unlike the previous cases, the port of this algorithm requires quite extensive modifications. The first step is similar to its sequential version (Sec 2.1.2.2): each process p will compute the integral lines whose seeds lie within its block $\mathcal{M}_p$ (each seed is processed independently via shared-memory parallelism with OpenMP). Moreover, the process p will be marked as the exclusive owner of the part of the integral line (i.e. the vertices and edges of the sub-geometry) created on its block. From there, two possibilities arise: either the integral line reaches its final vertex within $\mathcal{M}_p$, *completing* the computation, or the integral line reaches a ghost vertex owned by another process q and is *incomplete*. In the latter case, some of the integral line data (such as global identifier, the distance from the seed or the global identifier of the seed) is stored in a vector to be sent later to the process q (Figure 3.6*(d) and* (e)). Once all integral lines on all processes are marked as either complete or incomplete, all processes exchange the data of their incomplete integral lines and use that data to resume computation of the integral lines on their block.

These computation and communication steps are run until all integral lines on all processes are completed. Consequently, depending on the dataset, and the process, there may be very little communication, e.g. if all the integral lines lie within the bounds of a block, or a lot of communications, e.g. if some integral lines are defined across the blocks of multiple processes (hence its DDC classification).

3.5.4 Integrated pipeline

In this section, we describe an integrated pipeline that produces a real-life use case combining all the the port examples presented in Sec.3.5.3. All of the algorithms, their order as well as their input are described in Table 3.1. The input dataset is a three-dimensional regular grid with two scalar fields f, the electronic density in the *Adenine Thymine* complex (AT) and its gradient magnitude $|\nabla f|$. First, f and $|\nabla f|$ are smoothed and f is normalized. Critical points of f are computed and used as seeds to compute integral lines of f. The extracted integral lines capture the covalent and hydrogen bonds within the molecule complex (Figure 3.7). Then, critical points are computed for $|\nabla f|$ on the integral lines. The extracted critical points indicate locations of covalent bonds where the electronic density experiences rapid changes, indicating transition points occurring within the bond (Figure 3.7).

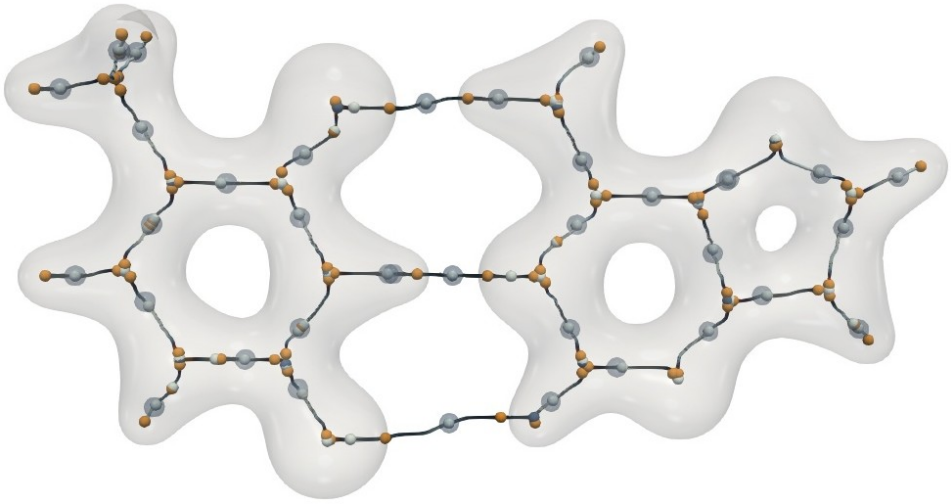


Figure 3.7 – *Output of the integrated pipeline on the AT dataset, a three-dimensional regular grid of the electronic density (and its gradient magnitude) in the Adenine Thymine complex (AT). The extracted integral lines capture the covalent and hydrogen bonds within the molecule complex. The transparent spheres are the critical points used as seeds of the integral lines while the full spheres are the critical points of* $|\nabla f|$ *and show where the electronic density experiences rapid changes, indicating transition points occurring within the bond. This image was obtained by resampling the original dataset to* 2048^3 *and executing the integrated pipeline on 64 nodes of 24 cores each (1536 cores) on MeSU-beta.*

Abbreviation	Algorithm	Input
1. SFS1	ScalarFieldSmoother	f
2. SFS2	ScalarFieldSmoother	$\|\nabla f\|$
3. SFN1	ScalarFieldNormalizer	f_{SFS1}
4. AP	ArrayPreconditioning	f_{SFN1}
5. SFCP1	ScalarFieldCriticalPoints	f_{AP}
6. IL	IntegralLines	f_{AP} (domain), f_{SFCP1} (seeds)
7. GS	GeometrySmoother	f_{IL}
8. SFCP2	ScalarFieldCriticalPoints	$\|\nabla f\|_{SFS2}$ on $\mathcal{M}_{GS}$

Table 3.1 – *Composition of the integrated pipeline. Each line denotes an algorithm in the pipeline, by order of appearance (top to bottom), as well as its input.* f *is the input scalar field. Each algorithm modifies the scalar field:* f_A *is the modified scalar field* f*, output of algorithm* A*.* $\mathcal{M}_{GS}$ *is the output domain of GeometrySmoother.*

The local order of f is required by two algorithms: the first critical points (SFCP1) and the integral lines (IL). Since these two algorithms are separate leaves of the pipeline, each of them would trigger the automatic local order computation. Instead, to avoid this duplicated computation, we manually call the local order computation in a preprocess (i.e. by calling the *ArrayPreconditioning* algorithm).

The chosen dataset is intentionally quite small ($177 \times 95 \times 48$) to ensure reproducibility. It is resampled before the pipeline to create a more sizable example, using ParaView's *ResampleToImage* feature (i.e. grid resampling via trilinear interpolation). Anyone can execute this pipeline to the best of their resources, by choosing the appropriate resampling dimensions. In our case, the new dataset is of dimensions (2048^3), encompassing roughly 8.5 billion vertices.

The pipeline was also run on a second, larger, dataset (*Turbulent Channel Flow*), to show TTK's capability to handle massive datasets (specifically, the largest publicly available dataset we have found). This dataset represents a three dimensional pressure field of a direct numerical simulation of a fully developed flow at different Reynolds numbers in a plane channel (obtained from the Open Scientific Visualization Datasets [Kla20]). Its dimensions are ($10240 \times 7680 \times 1536$), which is approximately 120 billion vertices. Before applying the pipeline, the gradient magnitude is computed and added to the dataset, and the result is converted using single-precision floating-point numbers (thereby reducing memory consumption at runtime).

3.6 Results

For the following results, we rely on Sorbonne Université's supercomputer, MeSU-beta. MeSU-beta is a compute cluster with 144 nodes of 24 cores each (totaling 3456 cores). Its nodes are composed of 2 Intel Xeon E5-2670v3 (2.7 GHz, 12 cores), with SMT (simultaneous multithreading) disabled (i.e. running 1 thread per core), and with 128GB of memory each. The nodes are interconnect with Mellanox Infiniband.

When measuring the performance of a specific algorithm, only the execution of the algorithm itself is timed. None of the preconditioning or input and output formatting is timed unless explicitly stated. The preconditioning steps are an investment in time: they can be used again by other algorithms later on in the pipeline, thus, including the cost of these steps in the execution time of a single algorithm would not provide an accurate

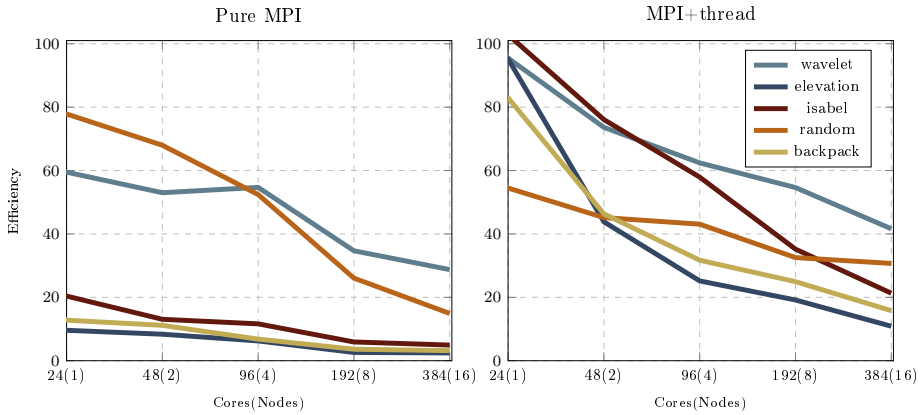


Figure 3.8 – *Strong scaling efficiencies for the Integral line computation algorithm with* 500,000 *seeds, randomly distributed on all processes, using the pure MPI strategy (left) and the MPI+thread one (right) with 1 MPI process and 24 threads per node. The MPI+thread strategy is significantly more efficient than the pure MPI one.*

representation of performance in a more complicated pipeline. They are therefore excluded from the individual benchmarks (Section 3.6.1) but included in the study of the global, integrated pipeline Section 3.6.2 (which is timed using ParaView's internal timer).

The benchmark is performed on five different datasets: *Wavelet*, *Elevation*, *Isabel*, *Random* and *Backpack*. The datasets all originate from publicly available repositories [Kla20, TTK20]. See Appendix A for more details on the datasets used in this chapter.

3.6.1 Distributed algorithms performance

This section evaluates the practical performance of the extension to the distributed setting (Section 3.5.3) of the algorithms presented in Section 2.1, by considering strong and weak scaling.

3.6.1.1 Strong scaling

For a given problem size, we first evaluate the runtime performance of our novel framework for distributed computations in TTK, as more computational resources are available. For this, we conduct a strong scaling analysis with results shown in terms of parallel efficiency (see Section 2.2.6). Each dataset is resampled to 512^3 via trilinear interpolation.

We first compare the pure MPI and the MPI+thread strategies (Section 3.5.2). Regarding the MPI+thread strategy, we rely on one MPI process per node (and 24 threads within) instead of one MPI process per processor (and 12 threads each). According to performance tests (not

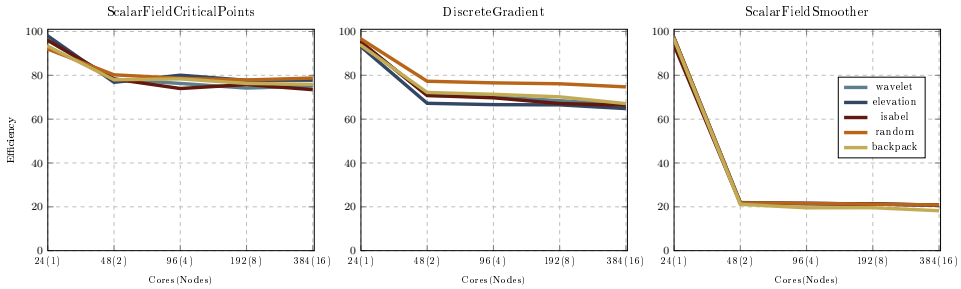


Figure 3.9 – *Strong scaling efficiencies for various algorithms (MPI+thread: 1 MPI process and 24 threads per node).*

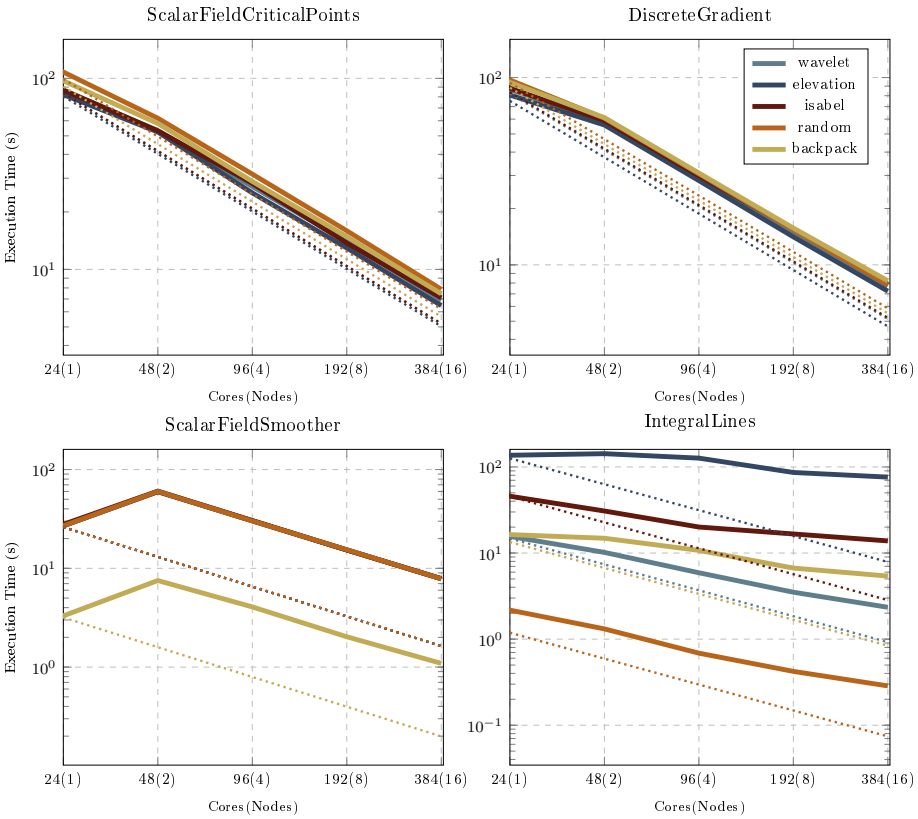


Figure 3.10 – *Strong scaling (execution times) for various algorithms (MPI+thread: 1 MPI process and 24 threads per node). The dotted lines indicate ideal performances.*

shown here), both options lead indeed to similar performance results, except when using one single node: in this case, having one single MPI process (no communication and no ghost simplices required) is more efficient than two. Having one MPI process per node also leads to a lower memory usage.

As shown in Figure 3.8, using MPI+thread (with one MPI process and 24 threads per node) is then substantially more efficient than using a pure MPI design for the integral line algorithm, for all datasets except the *Random* dataset. More precisely, even for MPI+thread, the efficiency decreases with the number of cores and depends significantly on the dataset. This is due to a strong workload imbalance between the processes: the integral lines are not evenly distributed on the MPI processes which can lead to long idle periods for some processes (waiting for the other to process their integral lines). This applies to the *Backpack* dataset for example. Regarding the *Elevation* dataset (very smooth, with only one maximum and one minimum) or the *Isabel* one (very smooth too), the generated integral lines are here especially lengthy and span several (but not all) processes, leading to low efficiencies. On the contrary the *Random* dataset is very balanced, but is also very noisy, leading to very short integral lines: for the same number of integral lines, the computation times are much shorter than for the other datasets which makes the communication cost more detrimental to performance. Finally, the *Wavelet* dataset is the most balanced one, with long enough integral lines, and thus shows the best performance results. Compared to the pure MPI strategy, the MPI+thread one benefits from fewer MPI processes and therefore from a lower load imbalance. See Appendix B for an additional comparison of different MPI+thread strategies, namely: 2×12, and 1×24 as well as a pure MPI one.

The performance results for the other distributed algorithms can be found in Figs 3.9 (efficiency) and 3.10 (execution time). For the *ScalarFieldCriticalPoints*, a very good efficiency (80%) is achieved (which is comparable to its shared-memory parallel implementation on one node, 90%), with little dependence on the dataset. The *DiscreteGradient* likewise performs very well in terms of efficiency, albeit slightly less, due to the parallelization method of the algorithm, for which adding ghost simplices will add a small amount of extra work in parallel. These two algorithms strongly benefit from parallel computing, even when using hundreds of cores. The *ScalarFieldSmoother* exhibits lower efficiency. This can be explained by the need for communications at each iteration, as well as by the low cost of the smoothing process (which is a simple averaging operation). Indeed,

the faster a computation, the stronger the impact of communications on the overall performance.

Finally we emphasize that, at the exception of *IntegralLines* (for which we derived a new implementation, subsubsection 3.5.3.5), shared-memory parallel implementations of these algorithms (using OpenMP threads) pre-existed in TTK prior to this work. In our MPI+thread strategy, we leverage these same shared-memory parallel implementations regarding multi-thread parallelism. Moreover when using only one MPI process, MPI communications are not triggered and processing specific to the distributed setting (e.g. on ghost simplices) is not carried out. Thus, when running our novel MPI+thread extension of these algorithms on only one MPI process, performances are identical to these of the pre-existing, shared-memory-only implementations.

3.6.1.2 Communication thread trial

We have tried to improve the parallel efficiencies of the integral line algorithm, by dedicating a thread to MPI communications (See Section 2.2.3). Thanks to this thread, an incomplete integral line is sent right away, without waiting for all integral lines on the process to be computed. Each process also continuously receives integral lines and adds them immediately to the pool of integral lines to be computed.

Performance results showed an overall similar efficiency for both implementations (with and without a communication thread). Further testing provided clarification regarding the underlying factors contributing to this outcome. Without communication thread, processes spend time waiting at synchronization steps. This idle time should be reduced by adding a communication thread. However, processes often wait for the same process that almost always performs the most work of a computation step. That particular process does not spend a lot of time waiting at synchronization steps as it most often arrives last. Therefore, adding a communication thread does not significantly help speed up the computation of that process. In fact, the time gained with the added reactivity just only compensates the loss of a computation thread (that was turned into a communication thread). The real problem here is the workload imbalance between processes, resulting in equivalent efficiency with or without a communication thread. Furthermore, this design based on a communication thread adds a significant amount of complexity to the implementation (due to the required thread synchronizations). As a result,

we do not rely on this communication thread design in our distributed integral lines implementation.

3.6.1.3 Weak scaling

Next, we evaluate, the ability of our framework to process datasets of increasing sizes. For this, we conduct a weak scaling analysis, with results shown in terms of parallel efficiency (see Section 2.2.6). The datasets have been resampled to 512^3 on one node. For *ScalarFieldCriticalPoints*, *DiscreteGradient*, and *ScalarFieldSmoother*, the input size is increased by doubling the number of samples, one dimension at a time. For *IntegralLines*, the workload is increased by doubling the number of seeds at each iteration. This choice was made as doubling the size of the input does not double the workload. Indeed, increasing the sampling rate creates artifacts that produce critical points that cut short integral lines.

As shown in Figs. 3.11 (efficiency) and 3.12 (execution time), for the *ScalarFieldCriticalPoints* and the *DiscreteGradient*, the efficiency remains quite high as the amount of work and the number of cores double: this is close to the ideal performance. Therefore, the conclusions are the same as for the strong scaling study: the performance is very good on all data sets, slightly less for the *DiscreteGradient* than the *ScalarFieldCriticalPoints*. For the *ScalarFieldSmoother*, the weak scaling shows that after the first drop of performance from one to two processes, due to synchronizations and communications that do not occur on one node, the computation actually scales really well, with a nearly constant efficiency on more than one node.

For the *IntegralLines*, the datasets *Backpack*, *Elevation* and *Isabel* show degraded performance similarly to the strong scaling. However, the results for the *Wavelet* and *Random* stay much closer to the ideal than for the strong scaling study. This can be explained by two factors. First, unlike the case of the strong scaling study, the number of seeds per node in the weak study is constant and does not decrease. Hence, the workload imbalance has a smaller impact and does not deteriorate the performance as much. Second, it is likely that the workload for the strong scaling study becomes too small as the number of cores increases. This makes the relative cost of communications and synchronizations very important.

Overall, this weak scaling analysis shows that, for *ScalarFieldCriticalPoints* and *DiscreteGradient*, the weak scaling is close to ideal (i.e. a problem of growing size can be processed in constant time when increasing accordingly the number of cores). For *ScalarFieldSmoother*, after a first

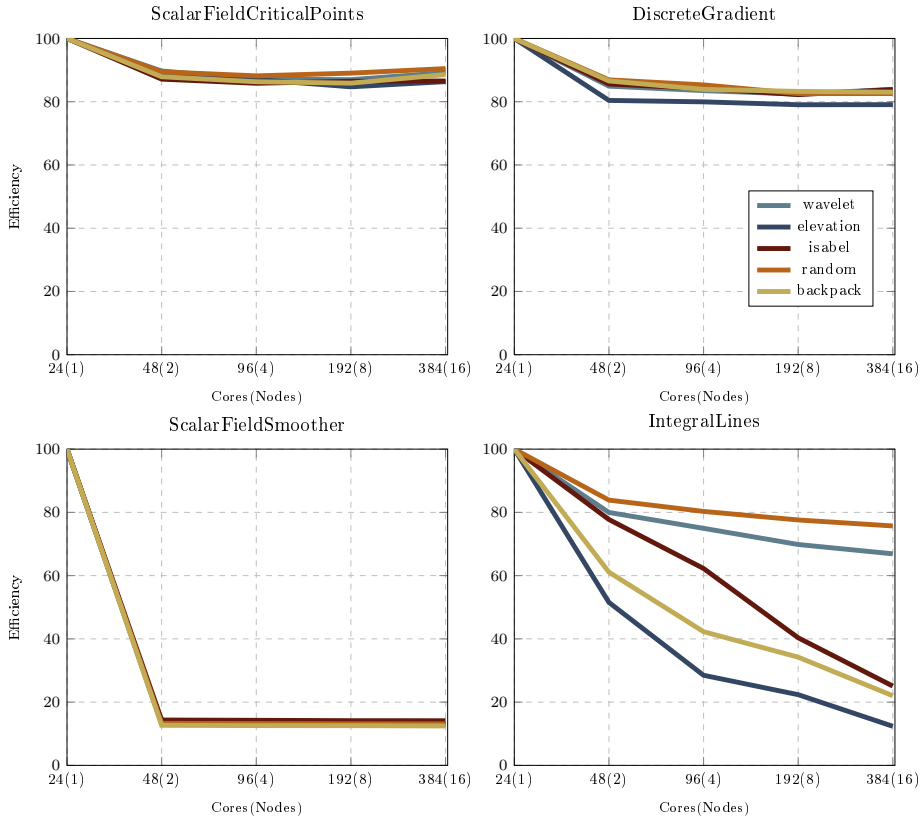


Figure 3.11 – *Weak scaling efficiencies for various algorithms (MPI+thread: 1 MPI process and 24 threads per node)*

degradation due to inter-process synchronization and communication, the efficiency is nearly constant. Finally, weak scaling performances are degraded overall for *IntegralLines*, at the exception of well balanced datasets that show much better performance than in the strong scaling study.

3.6.2 Integrated pipeline performance

We now present experimental results for the integrated pipeline (Section 3.5.4), which exemplifies a real-life use case combining all of the port examples described in Section 3.5.3, on datasets which were too large (8.5 and 120 billion vertices, Section 3.5.4) to be handled by TTK prior to this work.

The results for the integrated pipeline are twofold: an output image (Figure 3.7 and Figure 3.14) and the time profiling of the pipeline (Figure 3.13). The image is produced using offscreen rendering with OSMesa on our supercomputer. Profiling is done using both Paraview's timer (average, minimum and maximum computation times across processes, for an overall algorithm, preconditioning included) and the TTK timer de-

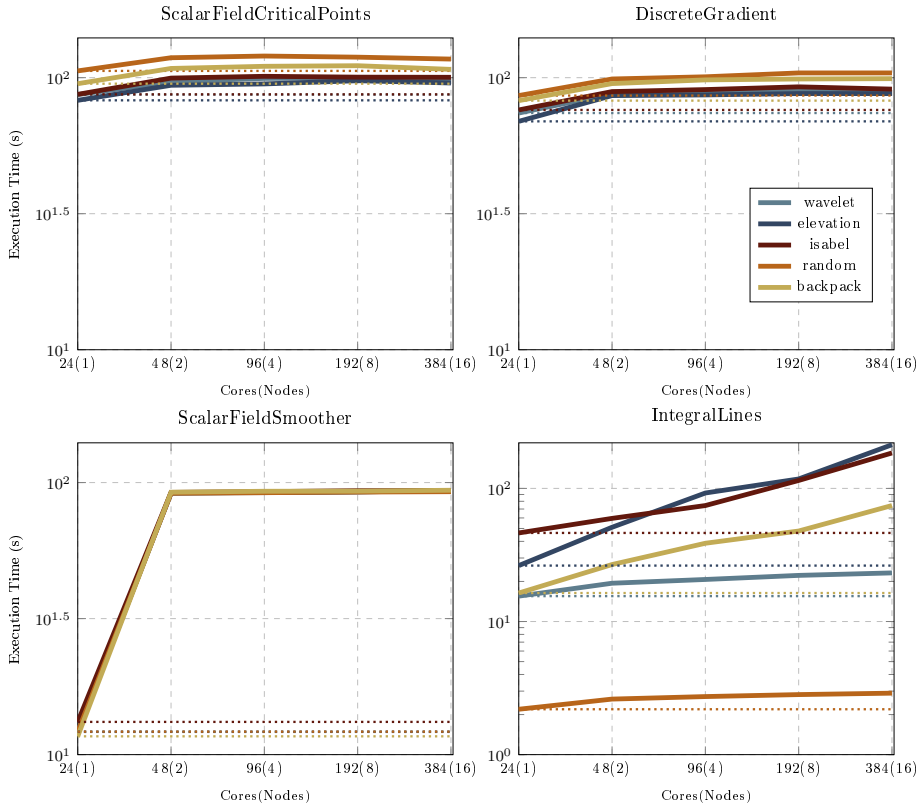


Figure 3.12 – *Weak scaling (execution times) for various algorithms (MPI+thread: 1 MPI process and 24 threads per node). The dotted lines indicate ideal performances.*

fined in Section 3.4.2 (for a fine-grain account of the execution time within an algorithm and its preconditioning).

3.6.2.1 The Adenine Thymine complex (AT) dataset

For the experiments of Figs. 3.7 and 3.13 (left), the selected resampling dimensions for the input regular grid are 2048^3, a choice explained in Section 3.5.4. The overall computation takes 241.2 seconds. Preconditioning is triggered once, before executing the first TTK algorithm. The longest preconditioning step is Paraview's ghost cells generation (24.2% of the total pipeline time), a step commonly used in a distributed-memory setting, regardless of TTK. The preconditioning specific to TTK's use of MPI (i.e. *Local Adjacency Graph*, *Simplex-To-Process Maps*, *Ghost Data Exchange*) is significantly faster and takes only 1.2% of the overall pipeline computation time, which can be considered as negligible next to the rest of the pipeline. TTK computations (preconditioning included) make up 70.1% of the total pipeline computation, which can be considered as a satisfactory efficiency.

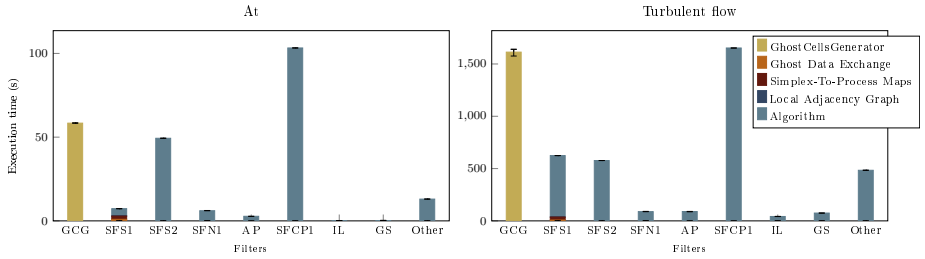


Figure 3.13 – *Time profiling for the integrated pipeline for the AT dataset resampled to roughly* 8.5 *billion vertices (left) and the* Turbulent Channel Flow *dataset (right) of 120 billion vertices. The execution was conducted using 64 nodes of 24 cores each (*1536 *cores in total) on MeSU-beta. Each bar corresponds to the execution time of one algorithm. SFS1 is computed for 1 iteration for the AT dataset and 10 iterations for the turbulent flow dataset (which is more irregular). The* Other *step consists in steps that are not part of an algorithm, such as loading the TTK plugin in Paraview, Paraview overhead and I/O operations. Only algorithms that take up a significant amount of time are shown in the profiling (see Table 3.1 for a description of the abbreviations). In both cases, the MPI preconditioning computed by our framework (*Local Adjacency Graph, Simplex-To-Process Maps, Ghost Data Exchange*) is negligible within the overall pipeline execution time (at most* 1.2%*).*

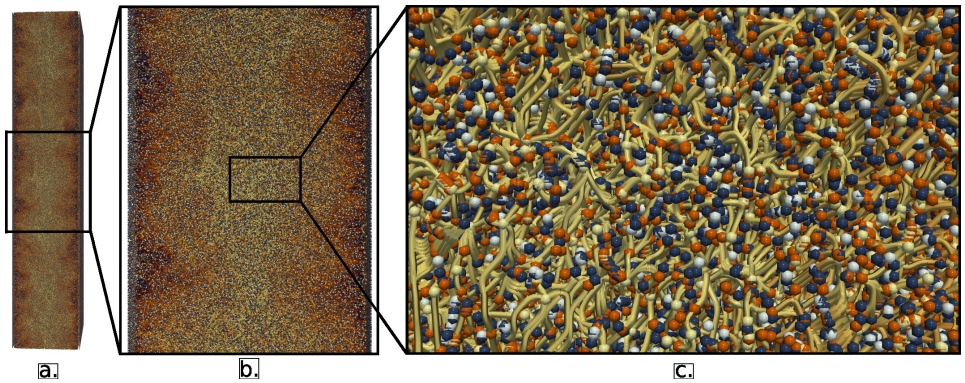


Figure 3.14 – *Output of the integrated pipeline on the* Turbulent Channel Flow *dataset (120 billion vertices), a three-dimensional regular grid with two scalar fields, the pressure of the fluid and its gradient magnitude. The pipeline was executed up to the Geometry Smoother algorithm. The spheres correspond to the pressure critical points and the tubes are the integral lines starting at saddle points. Figure (a) shows all of the produced geometry, while (b) and (c) show parts of the output zoomed in. These images were produced on a quarter of the total dataset due to rendering related issues (see subsubsection 3.6.2.2), while Figure 3.13 was produced on the full dataset.*

3.6.2.2 The Turbulent Channel Flow dataset

The computation shown in Figure 3.13 (right) was performed on the complete dataset (120 billion vertices, single-precision, Section 3.5.4). The overall computation takes 5257.5 seconds.

The execution time of this pipeline includes the algorithms listed in Table 3.1. Note that the rendering time is not included in the time profiling reported in Figure 3.13 (for both datasets). For the turbulent flow dataset, explicit glyphs were used for the rendering of the critical points (spheres) and integral lines (cylinders), as the screen-space glyph rendering features of ParaView did not produce satisfactory results in a distributed setting. However, the generation of glyph geometry required a lot of memory, therefore the rendering in Figure 3.14 was performed on only a quarter of the dataset. The pipeline profiled in Figure 3.13, however, was indeed executed on the whole dataset.

Similarly to the AT dataset, the longest preconditioning step is Paraview's ghost cells generation (30.7% of the total pipeline time). Again, TTK's specific MPI-preconditioning is marginal and takes up only 0.7% of the overall pipeline computation time. Computations of TTK algorithms (preconditioning included) make up for 59.2% of the total execution time. When compared to the AT dataset, the execution time of SFCP1 is multiplied by a factor of roughly 15, which is comparable to the increase in data size between datasets, indicating good scalability.

Overall, this experiment shows that, thanks to our MPI-based framework, TTK can now run advanced analysis pipelines on massive datasets (up to 120 billion vertices on our supercomputer), which were too large to be handled by TTK prior to this work. We showed that this could be achieved in an acceptable amount of time, while requiring a TTK-MPI specific preconditioning of negligible computation time overhead (0.7% of the total computation).

3.6.3 Limitations

Section 3.3 presented our strategy to provide consistent global simplex identifiers, irrespective of the number of processes. This guarantees a per-bit compatibility of the input data representation with the sequential mode of TTK, and consequently a per-bit compatibility of the pipeline outputs. However, the usage of threads can challenge the determinism of certain algorithms, given the non-deterministic nature of the thread scheduler. Then, an additional effort may need to be made by the developers to ad-

dress this non-determinism within their implementation of a topological algorithm (to ensure per-bit compatibility). In our experiments, we opted not to enforce determinism for integral lines, given the lack of control over the thread scheduler.

A significant difficulty occurring when processing massive datasets with ParaView is the substantial memory footprint induced by ParaView's interactive pipeline management. Data flows through the pipeline, being transformed at each step by algorithms. Rather than modifying data in-place, algorithms generate copies before implementing changes. This methodology offers several advantages, such as preventing redundant computation of inputs when multiple branches share the same input, resulting in better efficiency, especially when adjusting interactively the algorithm parameters. However, this copy-before-computation approach leads to a rapid increase in memory usage during computations, which can become problematic in practice for pipelines counting a large number of algorithms.

Finally, several specialized domain representations which are popular in scientific computing – such as grids with periodic conditions along a restricted set of dimensions or adaptive mesh refinement (AMR) – are not natively supported by TTK and these currently need to be explicitly triangulated in a pre-process.

3.7 Summary

In this chapter, we presented a software framework for the support of topological analysis pipelines in a distributed-memory model. Specifically, we instantiated our framework with the MPI model, within TTK. An extension of TTK's efficient triangulation data structure to a distributed-memory context was presented, as well as a software infrastructure supporting advanced and distributed topological pipelines. A taxonomy of algorithms supported by TTK was provided, depending on their communication requirements. The ports of several algorithms were described, with detailed performance analyses, following a MPI+thread strategy. We also provided a real-life use case consisting of an advanced pipeline of multiple algorithms, run on a dataset of 120 billion vertices on a compute cluster with 64 nodes (1536 cores), showing that the cost of TTK's MPI preconditioning is marginal next to the execution time of the pipeline. TTK is now able to compute complex pipelines involving several algorithms on datasets too large to be processed on a commodity computer.

As a perspective, the entire stack of TDA algorithms can now be revisited to be adapted to the distributed setting, which we initiate in the next chapter with persistent homology computation.

4 Distributed Discrete Morse Sandwich: Efficient Computation of Persistence Diagrams for Massive Scalar Data

Contents

This chapter presents the extension of the *"Discrete Morse Sandwich"* (DMS) to distributed-memory parallelism with MPI. The persistence diagram which describes the topological features of a dataset, is a key descriptor in Topological Data Analysis. The DMS method is reported to be the most efficient algorithm for computing persistence diagrams of 3D scalar fields on a single node, using shared-memory parallelism. In this work, we extend DMS to distributed-memory parallelism for the efficient and scalable computation of persistence diagrams for massive datasets across multiple compute nodes. On the one hand, we can leverage the embarrassingly parallel procedure of the first and most time-consuming step of DMS (namely the discrete gradient computation). On the other hand, the efficient distributed computations of the subsequent DMS steps are much more challenging. Like most TDA algorithms, they indeed provide a global view point on the data, which requires multiple global and irregular data traversals with little computation, a combination of factors that challenges scalability due to the additional communications, synchronizations and computations required to ensure the correctness of the algorithm. To address this, we have extensively revised the DMS routines by contributing a new self-correcting distributed pairing algorithm, redesigning key data structures and introducing computation tokens to coordinate distributed computations. We have also introduced a dedicated communication thread to overlap communication and computation. Besides, DDMS relies on a hybrid MPI+thread design combining the advantages of both shared and distributed memories: *(i)* larger total memory and *(ii)* reduced communication and synchronization overheads, along with improved intra-node dynamic load balancing. Detailed performance analyses show the scalability of our hybrid MPI+thread implementation for strong and weak scaling using up to 16 nodes of 32 cores each (512 cores total). Our implementation outperforms DIPHA, a reference implementation for the distributed computation of persistence diagrams, with an average speedup of $\times 8$ on 512 cores. Finally, we show the capabilities of

our algorithm by computing the persistence diagram of a 3D scalar field of 6 billion vertices in 174 seconds on 512 cores.

The work presented in this chapter has been submitted to the journal IEEE Transactions on Parallel and Distributed Systems. An example of use is available online (https://github.com/eve-le-guillou/DDMS-example). Our implementation is currently being integrated in TTK.

4.1 Outline

This chapter introduces the Distributed Discrete Morse Sandwich (DDMS), an efficient algorithm for persistence diagram (see subsubsection 2.1.3.2) computation exploiting distributed-memory parallelism. The original DMS algorithm is summarized in Section 4.2. Section 4.3 offers a high-level overview of DDMS. In Section 4.4, we provide a detailed explanation of the distributed computation of $\mathcal{D}_0$ and $\mathcal{D}_2$, including our novel self-correcting procedure at the core of this computation. In Section 4.5, we present the distributed computation of $\mathcal{D}_1$ which has been designed by revisiting the DMS procedure "PairCriticalSimplices". Our experiments are presented in Section 4.6, both in strong and weak scaling. We also demonstrate significant gain over DIPHA [BKR14b] (Section 4.6.5), to our knowledge the only publicly available implementation for this problem in a distributed context. Finally, we illustrate the new distributed capabilities of DDMS by computing the persistence diagram (Section 4.6.6) of a 3D scalar field of 6 billion vertices distributed on 16 nodes of 32 cores each.

4.1.1 Related work

Persistent homology: Multiple research groups independently introduced persistent homology [Bar94, ELZ02, FL99, Rob99]. In numerous data analysis applications, topological persistence rapidly emerged as a compelling measure of importance, helping in the identification of salient topological features within the data. The most common method for persistence homology computation relies on the reduction of the boundary matrix (that describes the facet/co-facet relations between the simplices of the input domain). DMS relies on a different strategy, based on discrete Morse Theory [For98, MN12, RWS11], but there are several conceptual similarities to existing documented accelerations. Bauer et al. [Bau19] introduced the idea of *apparent pairs*, which is similar to the *zero-persistence skip* pre-computation step of DMS. Furthermore, the stratification strategy employed in DMS can be linked with Bauer's et al. [BKR14a] stratification through "*Clearing*" and "*Compression*" that enables to discard simplices already involved in persistence pairs. Edelsbrunner et al. [ELZ02] observe that the persistence diagram for dimension 0 can be computed through a Union-Find data structure. They also observe that the 2-dimensional persistence diagram can be obtained, by symmetry, with a Union-Find

structure. This was aggressively exploited by Guillou et al. [GVT23] as they restricted the Union-Find computation to stable and unstable sets of 1 and 2 saddles.

Other methods have investigated Morse Theory to accelerate the computation of persistent homology [Mil63, Mor34], specifically in a discrete setting [For98]. In particular, Robins et al [RWS11] introduced the discrete gradient employed in DMS and used it to accelerate the computation of persistence. Other approaches improved on this idea [GRWH12, Iur21, Wag23] or extended it to a more general setting not limited to regular grids [MN12]. For instance, Wagner introduced an out-of-core approach [Wag23] capable of processing massive scalar datasets on commodity hardware, but at the expense of significantly long computations (typically hours of computation for several billions of data points). In contrast, our approach targets high-performance hardware in a distributed setting, with much faster computations (Section 4.6.6). Also, note that this out-of-core approach [Wag23] is not included in the performance benchmark by Guillou et al. [GVT23] (which was published before) but our preliminary experiments report a typical $\times 2$ speedup in favor of DMS [GVT23] on our hardware. Overall, in contrast to previous approaches based on discrete Morse theory, DMS significantly accelerates the process thanks to an aggressive stratification strategy, taking advantage of the specificities of the diagrams $\mathcal{D}_0$ and $\mathcal{D}_2$ and avoiding the computation of the full Morse complex.

In practice, there are numerous software packages available to produce persistence diagrams, such as *PHAT* [BKRW17], DIPHA [BKR14b], *Gudhi* [MBGY14], *Ripser* [Bau19] or *Eirene* [HP18]. Each implementation however focuses on particular data structures, such as generic filtrations of cell complexes (for *PHAT*, DIPHA and *Gudhi*) or Rips filtration of high-dimensional point clouds (for *Eirene* and *Ripser*). Some of the listed software for persistence diagram computation are purely sequential (*Eirene*, *Ripser*), while others implement shared-memory parallelism (*PHAT*, *DMS*). DMS has been reported to be the fastest implementation using shared-memory parallelism, according to the benchmark provided with its introductory paper [GVT23].

Distributed-memory algorithms: There are few existing approaches relative to distributed-memory parallelism and persistence diagrams. The most well-known method is DIPHA, introduced by Bauer et al. in [BKR14b]. It computes the boundary matrix of the domain and partitions the matrix into blocks of contiguous columns. The boundary matrix

represents the relations between the simplices and their faces. The matrix is then reduced using a variant of Gaussian elimination. In particular, it is very similar to the spectral sequence algorithm for persistent homology [EH09], with several adaptations to make it correct and efficient in a distributed-memory setting. Each block is first reduced locally on a process. When the blocks have been reduced to the best of a process capabilities, the unreduced columns are sent to the next process to the left. These communication and computation steps are performed until all columns are completely reduced. The persistence pairs of the diagram can then be extracted from the columns and rows of the reduced matrix. DIPHA offers good parallel speedups, using only multi-process (MPI) parallelism (no multi-threading), and allows for the analysis of larger datasets than anterior work.

Another distributed method was introduced by Ceccaroni et al. in [CDRFPB24]. However, their parallelism is limited to the concurrent processing of multiple, smaller datasets (i.e., at least one full dataset per process). On the contrary, we aim in this chapter at processing one voluminous dataset on multiple compute nodes. An approach recently introduced by Nigmetov et al. [NM24] also uses distributed-memory parallelization to produce a persistence diagram. The algorithm combines spatial and range partitioning by computing first a local reduction of the data and then switching to a global reduction. However, this algorithm relies on persistent co-homology while our algorithm uses homology. As stated by Nigmetov et al., experiments suggest that persistent homology may be more efficient for computing the persistence diagram. Experiments that we performed with DIPHA using both homology and co-homology confirm this. At the time of writing this manuscript, no public implementation has been found for the approach of Nigmetov et al. To our knowledge, DIPHA is thus the only public implementation for distributed-memory computation of persistent homology.

4.1.2 Contributions

This chapter makes the following new contributions:

1. *A hybrid distributed, shared-memory algorithm for the computation of persistence diagrams for 1D, 2D and 3D data:* Our work extends the fastest shared-memory approach (DMS) to the distributed setting, enabling an efficient computation of persistence diagrams for scalar

fields of significant sizes. This is achieved thanks to the following novel procedures:

(a) A novel self-correcting distributed algorithm for computing $\mathcal{D}_0$ and $\mathcal{D}_2$, extending the related procedure of DMS [GVT23] to the distributed setting.

(b) A novel procedure for computing $\mathcal{D}_1$, based on an efficient extension of the "PairCriticalSimplices" procedure [GVT23] with specific data structures, computation tokens, and a dedicated communication thread to overlap communication and computation.

The algorithm is output sensitive and provides substantial gains over the original DMS approach as well as DIPHA, the reference method for computing persistence diagrams in a distributed setting.

2. *An open-source implementation:* For reproduction purposes, we provide a C++ implementation of our approach using MPI+OpenMP and based on the Topology ToolKit (TTK) [BMBF+19, TFL+17, LWG+24] integrated in ParaView.

3. *A reproducible example*: We provide a reference Python script for computing the persistence diagram with a dataset size that can be adjusted to fit the capacities of any system (publicly available at: https://github.com/eve-le-guillou/DDMS-example).

4.2 The original Discrete Morse Sandwich algorithm

Here is an overview of the original Discrete Morse Sandwich algorithm. For a more detailed description of the algorithm, we refer the reader to its introduction paper [GVT23]. We consider here 3D datasets, where $\mathcal{D}_0$, $\mathcal{D}_1$ and $\mathcal{D}_2$ have to be computed. First, the discrete gradient is computed in parallel using multi-threading. The critical simplices are then deduced from the gradient. This step is called the *zero-persistence skip*: each remaining, non-critical simplex forms a zero-persistence pair with the other simplex involved in its discrete vector (subsubsection 2.1.2.3). The rest of the algorithm will focus on pairing the obtained critical simplices, following a stratification strategy where $\mathcal{D}_0$ and $\mathcal{D}_2$ (special cases) are computed before $\mathcal{D}_1$.

The diagram $\mathcal{D}_0$ is computed first. An overview of this algorithm is shown in Figure 4.1. We start by building an extremum graph noted $\mathcal{G}_0$ by

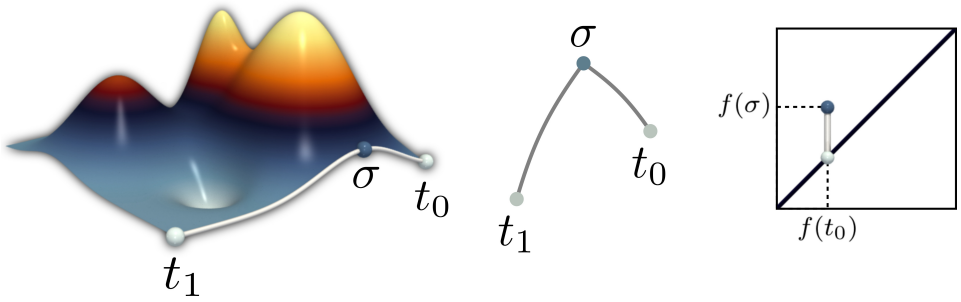


Figure 4.1 – *Overview of the DMS algorithm for the computation of $\mathcal{D}_0$. First, the unstable set is computed from the vertices of the critical* 1*-simplex σ by tracing v-paths (white curves, left). The set is then collapsed into an* extremum graph *$\mathcal{G}_0$ (middle). Each critical simplex is represented by a node in the graph, with the arcs representing the v-paths. Finally, the graph is processed with a Union-Find structure to produce the persistence pair in $\mathcal{D}_0$ (right).*

following, for the two vertices v_0 and v_1 of a critical edge σ, the gradient until a critical 0-simplex (or extremum) is reached (t_0 and t_1). Each critical edge is processed in parallel (multi-threading). If t_0 and t_1 are distinct, the triplet (σ, t_0, t_1) represents new elements of the graph $\mathcal{G}_0$, adding the three nodes σ, t_0 and t_1 (one per element of the triplet) and two arcs representing the v-paths between σ and the two extrema. When all critical 1-simplices are processed, $\mathcal{G}_0$ is complete. $\mathcal{D}_0$ is then computed by visiting the edges of $\mathcal{G}_0$ in increasing order following the PairExtremaSaddles algorithm presented in Algorithm 1. This step is intrinsically sequential, and relies on a Union-Find data structure for each node of $\mathcal{G}_0$. A Union-Find efficiently models connectivity in data through two primitives: *find()*, that returns the representative of the connected component containing the node, and *union()*, that merges together two components by unifying their representatives. Initially, each 0-simplex is its own representative. For each triplet (σ, t_0, t_1) of $\mathcal{G}_0$, the following procedure is applied: the representatives r_0 and r_1 of t_0 and t_1 are retrieved (using *find()*). r_0 is ensured to be the highest representative (l. 5-6 in Algorithm 1), by swapping if necessary its value with r_1. The highest representative, r_0, is then paired with σ and is assigned r_1 as representative. In Figure 4.1, this step creates the pairing (σ, t_0), with t_1 the new representative of t_0. To speed up the computation, the triplets of $\mathcal{G}_0$ are collapsed as they are visited. This means that the representative of t_0 is also set to r_1 (l. 12), which is equivalent to a *path compression* in a Union-Find data structure. This concludes the computation of $\mathcal{D}_0$.

The diagram $\mathcal{D}_2$ is computed similarly on critical 2- and 3-simplices using a dual discrete gradient field, obtained by reversing every discrete

vector of the discrete gradient on the domain's dual complex (see [GVT23] for more details).

Algorithm 1 PairExtremaSaddles

Input: An ordered set C_1 of triplets (σ_j, t_0, t_1) of $\mathcal{G}_\mathbf{0}$
Output: Persistence diagram $\mathcal{D}_0$

```
1: for j ∈ C_1 do                                  // Process the 1-simplex σ_j
2:    r_0 ← findRepresentative(t_0)
3:    r_1 ← findRepresentative(t_1)
4:    if r_0 ≠ r_1 then
5:       if r_0 < r_1 then
6:          swap(r_0, r_1)
7:       end if
8:       addPair(σ_j, r_0)
9:       Representative[r_0] ← r_1
10:      Representative[t_0] ← r_1                 // The arc is collapsed
11:   end if
12: end for
```

For 3D datasets, $\mathcal{D}_1$ is computed last. The first step of the construction of $\mathcal{D}_1$ consists in restricting its input to the unpaired critical 1- and 2-simplices: the critical 1- and 2-simplices already paired in $\mathcal{D}_0$ or $\mathcal{D}_2$ are hence not considered here. This stratification strategy greatly reduces the number of input simplices for $\mathcal{D}_1$. We then apply Algorithm 2 using the PairCriticalSimplex procedure defined in Algorithm 3. For each unpaired critical 2-simplex σ, a *homologous propagation* (See Figure 4.2) is performed. It expands a boundary, initially equal to $\partial\sigma$, by selecting the highest 1-simplex τ of the current boundary and adding to it the boundary of the 2-chain associated to τ. For simplicity, we will refer in the remainder of this chapter to the boundary initiated this way in σ as the *boundary of* σ. The propagation stops at the first unpaired 1-simplex. The pair (τ, σ) can then be created. σ is the death and τ the birth of the pair. Note that the propagation is expended in reverse relative to the filtration order: given a triangle σ, this process identifies the edge τ which created the latest 1-cycle γ in the filtration which is *homologous* to $\partial\sigma$ (subsubsection 2.1.3.2). All this assumes that the input set of simplices is ordered. However, as introduced by Nigmetov et al. in [MN20], it is possible to process the simplices in a random manner, hence in parallel (multi-threading). An extra case has then to be considered: when performing the homologous propagation for a simplex σ, it is possible to reach a 1-simplex τ that has already

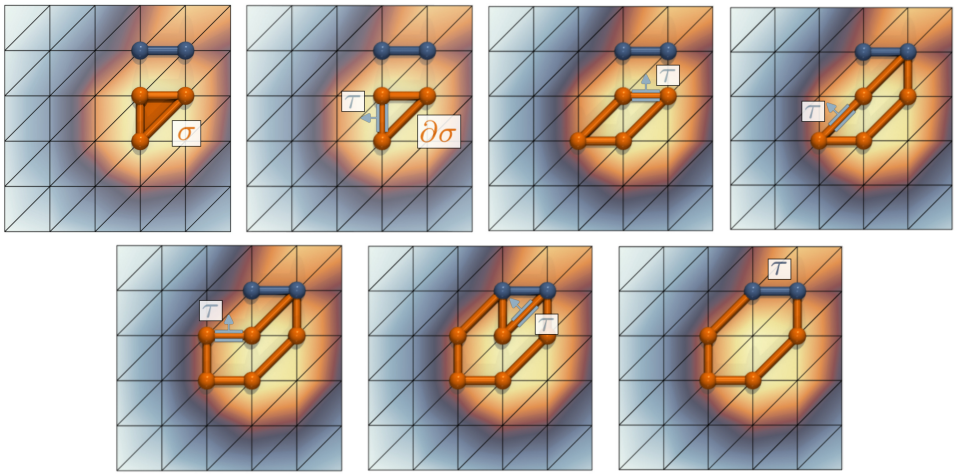


Figure 4.2 – *A homologous propagation given a simplex σ. The boundary of σ, $\partial\sigma$, is iteratively expanded by selecting its highest 1-simplex τ and adding to the boundary of σ the boundary of the 2-chain associated to τ. The boundary is expanded until reaching a critical 1-simplex (here in dark blue). The pair (σ, τ) is then added to $\mathcal{D}_1$.*

been paired to a 2-simplex σ_τ through homologous propagation. In that case, there are two possibilities: either (i) σ_τ is lower than σ and the propagation carries on by merging $\partial\sigma_\tau$ with $\partial\sigma$, or (ii) σ_τ is higher than σ. In that case, the pair (τ, σ_τ) is removed, (τ, σ) is added and the homologous propagation of σ_τ is resumed. Compare-And-Swap operations are used for thread-safe memory accesses as described by Nigmetov et al.[MN20]. $\mathcal{D}_1$ is finally created from the temporary pairs once all 2-simplices have been completely processed.

Algorithm 2 PairCriticalSimplices

Input: Set C_2 of unpaired critical 2-simplices
Output: Persistence diagram $\mathcal{D}_1$

1: **for** $j \in C_2$ *in parallel (multi-threading)* **do**
2: PairCriticalSimplex(σ_j)
3: **end for**
4: **for** $j \in C_2$ **do**
5: $\mathcal{D}_1 \leftarrow \mathcal{D}_1 \cup (Pair(\sigma_j), \sigma_j)$
6: **end for**

Regarding 2D datasets, only the diagrams $\mathcal{D}_0$ and $\mathcal{D}_1$ have to be computed. $\mathcal{D}_1$ is then computed in 2D on critical 1- and 2-simplices like $\mathcal{D}_2$ in 3D.

Algorithm 3 PairCriticalSimplex *(homologous propagation)*

Input: An unpaired critical 2-simplex σ

Output: A temporary pair of $\mathcal{D}_1$

```
1: if Boundary(σ) == 0 then
2:   Boundary(σ) ← ∂σ
3: end if
4: while Boundary(σ) ≠ 0 do
5:   τ ← max(Boundary(σ))
6:   if τ is not a critical simplex then                    // Expand boundary
7:     Boundary(σ) ←
8:       Boundary(σ) + Boundary(Pair(τ))
9:   else                                                  // τ is critical
10:    if Pair(τ) == ∅ then                                // τ is unpaired
11:      addPair(σ, τ)
12:      break
13:    else                                  // τ has already been paired to σ_τ
14:      σ_τ ← Pair(τ)
15:      if σ_τ < σ then                                   // Merge the boundaries
16:        Boundary(σ) ←
17:          Boundary(σ) + Boundary(σ_τ)
18:      else                                // σ is older and the true death of τ
19:        addPair(σ, τ)
20:        Pair(σ_τ) ← ∅
21:        PairCriticalSimplex(σ_τ)                        // Resume for σ_τ
22:      end if
23:    end if
24:  end if
25: end while
```

4.3 Overview

This section provides an overview of our approach. First, the global order of vertices is computed. This step is called *Array Preconditioning*. In a distributed-memory setting, a global order is necessary for comparing vertices owned by different processes. The computation is done in parallel on multiple processes in three steps: we start by creating locally a vector for all vertices of the elements to sort (comprised of the scalar value of the vertex, its global identifier and the rank of the process that owns it). The vector is then sorted in distributed using *psort* [CSGE07]. This particular implementation was chosen because it is lightweight, modifiable and relatively efficient. DIPHA also uses this implementation to construct its boundary matrix. Each process can then compute the global order of the elements that are present locally following the distributed sort. Finally, the global orders are sent back to the owner of the corresponding vertices. For simplices of higher dimension, comparisons are performed using the lexicographic comparison on their global vertex orders (subsubsection 2.1.1.5).

Second, the discrete gradient of the input data is computed by using the algorithm described by Robins et al. [RWS11], which is embarrassingly parallel for both shared- and distributed-memory contexts [LWG+24].

Third, the critical simplices are extracted from the gradient and sorted in the step *Extract & sort*.

Fourth, the diagrams $\mathcal{D}_0$ and $\mathcal{D}_2$ are computed by processing the unstable and stable sets of the 1-saddles and 2-saddles of f and applying a self-correcting pairing algorithm to extract the pairs of the diagram (Section 4.4).

Next, the diagram $\mathcal{D}_1$ is computed from the unpaired 1- and 2-saddles using our novel algorithm "DistributedPairCriticalSimplices" (Section 4.5).

Finally, the classes of infinite persistence are extracted by collecting the remaining, unpaired critical simplices.

4.4 Extremum-Saddle Persistence Pairs

In this section, we will describe the different modifications to PairExtremaSaddles (see Algorithm 1) that we contributed to obtain a distributed-memory version. This algorithm is applied to compute both $\mathcal{D}_0$ and $\mathcal{D}_2$. An overview of the algorithm is provided in Figure 4.3.

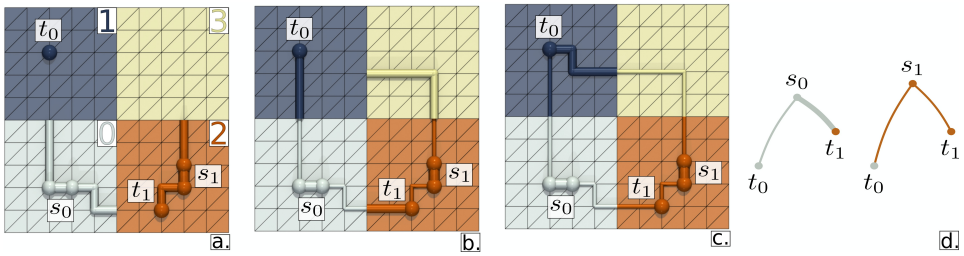


Figure 4.3 – *Overview of our algorithm for extremum-saddle pairing for $\mathcal{D}_0$. First, the unstable set is computed from the vertices of the critical* 1*-simplices s_0 and s_1 (sub-figure a, the two thick edges with their vertices) by following the gradient. When the computations of sets reach ghost simplices, a message is sent to the relevant process to notify it to resume computation on its domain (sub-figure b, the newly computed part of the set is thicker). Rounds of computations and communications are performed until all sets are computed (sub-figure c). The set is then collapsed into the distributed graph $\mathcal{G}_0$ following the rules stated in Section 4.4.2 (sub-figure d). The ownership of extremum t_0 is given to process 0 as it is the process with the lowest rank identifier that has a set that ends in t_0. The ownership of extremum t_1 is given to process 2 as it has an unstable set started in $\mathcal{M}_2$, at s_1, that ends in t_1. The graph is then processed with a distributed Union-Find structure to produce the persistence pair in $\mathcal{D}_0$. The thicker arc (s_0, t_1) corresponds to the computed persistence pair.*

4.4.1 Stable and unstable sets computation

The unstable sets of all critical 1-simplex are computed as follows: a v-path is extracted from each vertex of each critical 1-simplex. Then, two possibilities arise: a critical vertex is encountered (i.e. a local minimum), ending the computation there for this unstable set, or a ghost vertex is encountered, in which case a message is stored to be later sent to the process owning the ghost vertex so that the computation can resume there later. Once all computations either are completed or have generated a message, all processes exchange their stored messages and resume their computations on their block. These successive computation and communication steps run until no messages are sent on any process during a communication round. For stable sets, the computation is similar but is applied on 2-simplices as start of the set and 3-simplices as final simplex. The gradient is followed in reverse to emulate the dual gradient without explicitly computing it.

4.4.2 Distributed extremum graph construction

The previous step computes the stable and unstable sets. Now we need to collapse those sets into the distributed extremum graphs $\mathcal{G}_i$ with $i \in \{0, 2\}$. In the remainder, we focus on the case $i = 0$, the case $i = 2$ being symmet-

ric. The nodes of the graph are the saddles and extrema of the previously computed sets (Section 4.4.1). The arcs of the graph represent the v-paths connecting these critical simplices. A triplet of the graph refers to the three nodes (σ, t_0, t_1), where σ is a critical saddle and t_0 and t_1 extrema linked to σ by v-paths. Nodes of this graph may be located on different processes. We therefore establish a few additional rules to fit the definitions of $\mathcal{G}_0$ to a distributed-memory setting. Saddles are present on only one process. A saddle node in $\mathcal{G}_0$ is owned by a process p if its associated critical simplex is exclusively owned by p (Section 3.2). Extrema, however, can be present on multiple processes but they are owned by only one and are ghost on other processes. The ownership of an extremum node is determined as follows. If an extremum simplex e is exclusively owned by the process p (Section 3.2) and if there exists a saddle simplex s also exclusively owned by p such that one of its unstable sets terminates in e, then the extremum node of $\mathcal{G}_0$ associated to e is *owned* by p. Otherwise, the node associated to e is *owned* by the process with the lowest rank identifier that owns a saddle node whose unstable set ends in e. Figure 4.3 shows an example of a distributed extremum graph and applied ownership rules. The local graph on a process p is noted $\mathcal{G}_{0,p}$. The ghosted local graph of a process p is noted $\mathcal{G}'_{0,p}$. An extremum node is called *at the interface* of two processes p and q if it is owned by either p or q and is a ghost on the other process.

The computation of the collapse of the sets to build a graph is fairly straightforward: once all the sets are computed, the processes possess lists of all sets ending on a local extremum they own with regard to their domain. Each process will then determine which process is the owner of the extremum in the graph and send back to the owner of the originating saddle the extremum and its new ownership. It will also send to the new owner of the node a list of all the processes on which the extremum is a ghost with regard to the graph. Each process will then receive and build the parts of its local graph.

4.4.3 Self-correcting distributed pairing

We now have to build the $\mathcal{D}_0$ pairs from the local graph $\mathcal{G}'_{0,p}$ (and similarly for $\mathcal{D}_2$). This step was originally performed sequentially in DMS since its execution time was negligible compared to others. However, in a distributed-memory setting, a sequential execution is not viable as it would prevent any speedup on multiple nodes.

We thus have to design a distributed pairing algorithm. We use the

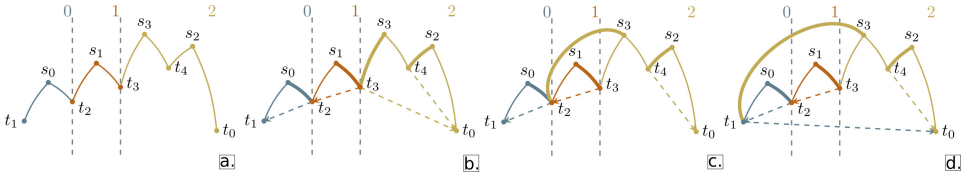


Figure 4.4 – *Illustration of our self-correcting pairing algorithm on an example of* $\mathcal{G}_0$. *Sub-figure a represents a distributed extremum graph on 3 processes, 0 (in blue), 1 (in orange) and 2 (in yellow). Extrema at the interface of two processes intersect with the dashed grey lines separating the domain of the processes. Sub-figure b represents the first computation iteration. Each process computes on its local graph without taking other processes into consideration. On process 2,* s_2 *is paired with* t_4 *(shown here by a thicker arc) and* t_0 *is made the new representative of* t_4 *(shown here by a dashed arrow). Then the pairing of* s_3 *is performed:* s_3 *can be paired with either* t_3 *or the representative of* t_4, t_0. *Since* t_3 *is higher than* t_0, s_3 *is paired with* t_3. *On process 1,* s_1 *is also paired with* t_3. *After the computation, processes exchange data regarding relevant, shared pairings. Process 2 will tell process 1 it created the pairing* (s_3, t_3) *with* t_0 *as the new representative, because* t_3 *is owned by process 1. Process 1 will tell process 2 about its owned pairing* $((s_1, t_3)$, *with* t_2 *as the new representative) because it knows that* t_3 *is present as a ghost on the graph of process 2. Process 2, upon receiving this message, will assess that the message is correct and its pairing* (s_3, t_3) *is wrong. It will compute a new pairing:* (s_3, t_2) *(as* t_2 *is the new representative of* t_3*) and tell process 1 about the pairing (sub-figure c). Process 1 will update* t_2 *in the pairing to its representative,* t_1. *As* t_1 *is owned by process 0, process 1 will tell it about the pairing. Process 0, knowing that* t_1 *is unpaired, will accept the pairing* (s_3, t_1), *update the representative of* t_1 *to* t_0 *and send back the information to process 2 that the actual correct pairing is* (s_3, t_1) *as shown in sub-figure d.*

idea of comparing saddles to see which is the oldest (or the youngest), but restrict the algorithm to the distributed extremum graph created at the previous step. Our algorithm design is also inspired by the self-correcting mechanism of the parallel (multi-threaded) procedure *PairCriticalSimplices*, computing $\mathcal{D}_1$ in the original DMS algorithm. This introduces incorrect pairings resulting in extra computations. Such overhead is however handled in parallel, ultimately benefiting overall performance gains (see Section 4.6). Comparisons to the original DMS algorithm in Figure 4.10 also show that our method incurs limited overhead. Moreover, we expected other strategies relying on a more direct approach (i.e. without incorrect pairings) to induce too much synchronizations and idle time to perform well.

A key difference from the original DMS algorithm is that the representative of an extremum will now store two pieces of information: the representative and the identifier of the saddle that assigned the representative to the extremum. This enables, when computing the pairing of a saddle σ, to stop the computation of the representatives of its extrema nodes t_0 and t_1 when the loop reaches representatives assigned by saddles older than σ (as this would not occur in sequential). The path compression mechanism of DMS mentioned in Section 4.2 (see Algorithm 1) is no longer applied as the resulting representatives may be false, leading to potentially incorrect computations for other pairings during the computation of other representatives. When a wrong pairing is detected by saddle comparison, the computation of the representatives of its original triplet is re-started from the beginning using Algorithm 4.

Here is a description of the overall self-correcting distributed algorithm. A practical example is shown in Figure 4.4. First, each process executes *DistributedProcessTriplet* (See Algorithm 4) for all its triplets (in a sorted manner, as it proved to be more efficient even though it is not required). The messages to be sent are stored until all computations are performed then all processes exchange their messages. For each message (σ, m_0, m_1) received, the process will first detect if it is a recomputation (encoded by $(\sigma, -1, -1)$) and will trigger it. If it is not a recomputation, the process will update the representatives of the message to $(\sigma, \mathit{Representative}(m_0, \sigma), \mathit{Representative}(m_1, \sigma))$. If the newly computed representatives belong to another process, then the message is passed along to that other process. Otherwise, the process detects if the message should trigger a correction through saddle comparison. If so, the pairs and the representatives are updated and a recomputation is triggered for

Algorithm 4 DistributedProcessTriplet

Input: A triplet (σ, t_0, t_1) of $\mathcal{G}_{0,p}$ on process p
Output: A temporary pair of $\mathcal{D}_0$

for i in $\{0, 1\}$ **do** // Compute the representatives
  $r_i \leftarrow findRepresentative(t_i, \sigma)$
  $r_i Paired \leftarrow Pair(r_i) \neq \emptyset$
  $r_i Invalid \leftarrow r_i Paired$ and $\sigma < Pair(r_i)$
  if $r_i Invalid$ **then**
    $r_i Paired \leftarrow false$
  end if
end for
if $r_0 \notin \mathcal{G}'_{0,p}$ or $r_1 \notin \mathcal{G}'_{0,p}$ **then**
  Send (σ, r_0, r_1) to owner
end if
if $r_0 == r_1$ **then**
  if $Pair(\sigma) \neq \emptyset$ **then** // σ should not be paired
    $r \leftarrow Pair(\sigma)$
    $Representative[r] \leftarrow r$ // re-initialize the representative
    $Pair(\sigma), Pair(r) \leftarrow \emptyset, \emptyset$ // remove the invalid pair
  end if
else
  if ($r_0 < r_1$ or $r_0 Paired$) and $!(r_1 Paired)$ **then**
    swap r_0 and r_1 data
  end if
  if $!(r_0 Paired)$ **then**
    if $r_0 Invalid$ **then** // remove the invalid pair
      $\sigma_k \leftarrow Pair(r_0)$
      $Pair(\sigma_k) \leftarrow \emptyset$
    end if
    $Pair(\sigma), Pair(r_0) \leftarrow r_0, \sigma$ // Add the new pair
    $Representative[r_0] \leftarrow (r_1, \sigma)$
    if r_0 is at the interface of p and process q **then**
      Send (σ, r_0, r_1) to process q
    end if
    if $r_0 Invalid$ **then** // Recompute the invalid σ_k
      if $\sigma_k \in \mathcal{G}_{0,p}$ **then**
        DistributedProcessTriplet(σ_k)
      else
        Send recomputation signal to the owner of σ_k
      end if
    end if
  end if
end if

the invalid saddle. This cycle of communications and computations is repeated until no messages are sent on any process during a communication round.

The original DMS algorithm used vectors to store several variables relative to critical simplices in its *PairCriticalSimplices* algorithm. These vectors were defined for all simplices of the triangulation, even though they were used only for critical simplices. This allowed for fast memory accesses, as the index of the simplex in the triangulation was equal its index in the vectors. In a distributed-memory setting, this is no longer possible: the global number of simplices will most likely prevent the memory allocation of such large vectors. We therefore reduced the size of the vectors to the number of local critical simplices and used maps to convert a global simplex index to its index in these vectors.

4.4.4 Shared-memory parallelism

The computations of stable and unstable sets are straightforwardly processed with multiple threads in each MPI process as all set computations are independent. The communications are however performed only by the OpenMP primary thread. The construction of the distributed graph is similarly parallelized.

The self-correcting pairing algorithm is not multi-threaded. As shown by Guillou et al. [GVT23] (Table 3, Appendix C), the computation of $\mathcal{D}_0$ and $\mathcal{D}_2$ with DMS is often negligible in terms of computation time with regard to the other procedures. This indicates that a parallelization of this step (even efficient) would result in very modest gains overall, if any. Notice also that a multi-threaded implementation of a similar algorithm by Smirnov et al. [SM17] leads indeed to limited speedups.

$\mathcal{D}_0$ and $\mathcal{D}_2$ being completely independent, we assigned each diagram computation to an OpenMP task that can itself generate threads in a nested manner.

4.5 Saddle-Saddle Persistence Pairs

In this section, we will describe the different modifications to the original *PairCriticalSimplex* (Algorithm 3) and *PairCriticalSimplices* (Algorithm 2) algorithms necessary for the efficient distributed computation of $\mathcal{D}_1$.

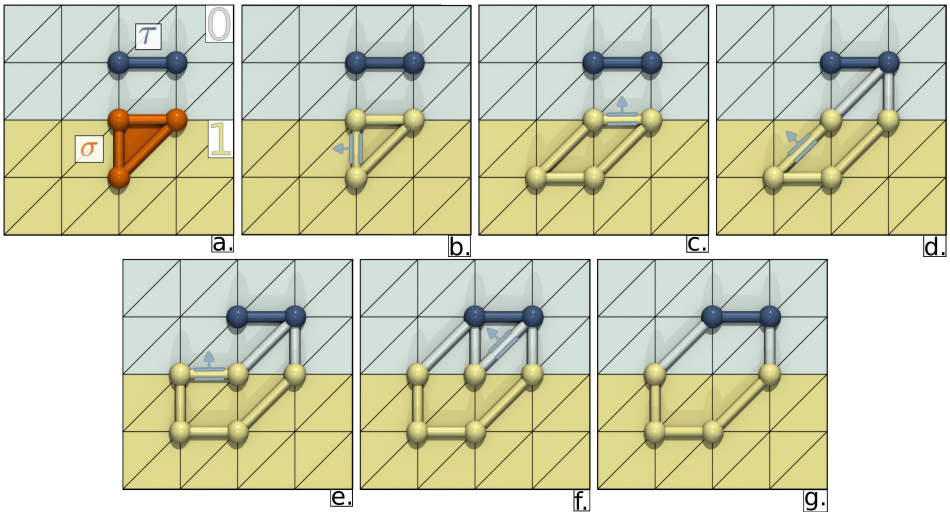


Figure 4.5 – *Distributed homologous propagation. Sub-figure a shows two critical simplices: σ, the death saddle, and τ, the birth saddle. σ is located on process 1 (light yellow) and τ is located on process 0 (light blue). Initially, the boundary of σ is only located on process 1 (Sub-figure b). New edges are added to the boundary that are owned by process 1 (Sub-figure c). Then new edges are added to the boundary that are ghost for process 1 (Sub-figure d) and a message is sent to process 0 so that it adds the edges to its local boundary of σ. On process 1, a new piece of information is stored regarding the boundary: the highest edge for process 0 (equal to the highest edge of the two ghost edges). As the global highest edge is still located on process 1 (gray arrow), the computation continues on process 1 (Sub-figure d) and additional ghost edges are added (Sub-figure e). After adding two new ghost edges (Sub-figure f), the highest edge is now located on process 0. Process 1 will send the computation token to process 0. Upon receiving it, process 0 will resume the computation and propagate the boundary (Sub-figure f). Finally, the propagation ends in τ and the pair (σ,τ) is created on process 0.*

4.5.1 Distributed-memory parallel algorithm

Our algorithm relies on a new data structure: the global-local boundary. For a 2-simplex σ, this structure is composed of two elements: the set of edges of the boundary initiated in σ (called in the remainder, for simplicity, the *boundary* of σ), which are owned by the current process p (called *local boundary*), and the highest boundary edges of all processes containing a part of the boundary of σ (called the *global boundary*). The local boundary is identical to the boundary of the original DMS algorithm. The global boundary is updated by other processes all through the computation. At any given time, on a given process p, the highest edge σ of its local boundary is always lower or equal to the highest edges reported by the global boundaries of the other processes.

A central idea in our distributed-memory algorithm is the notion of *computation token*. Each 2-simplex σ for which a homologous propagation needs to be computed is associated with a token. At any time, the token of each propagation is present in only one process. Only the process owning the token is allowed to propagate the boundary of σ. This means that, taken individually, each propagation is carried out sequentially. However, all the propagations are computed in parallel similarly to the shared-memory context.

Here is a description of our algorithm *DistributedPairCriticalSimplex* as defined in Algorithm 5 and illustrated in Figure 4.5. This algorithm revisits *PairCriticalSimplex* Algorithm 3 in order to compute a distributed homologous propagation for an unpaired critical 2-simplex σ. For an unpaired critical 2-simplex $\sigma \in \mathcal{M}_p$, with p a process, its local boundary is propagated by following the same rules as *PairCriticalSimplex*. However, when a ghost edge, owned by process q, needs to be added to the global-local boundary of σ, a message will be sent to q so that q adds this particular edge to its local boundary of σ. The propagation can trigger a merge between two boundaries. Local boundaries are merged similarly to *PairCriticalSimplex* (Algorithm 3, l.15-17). Global boundaries are merged by keeping the highest of the two edges for each process. A message will then be sent to notify all the relevant processes that a merge has occurred and should be performed by them as well. As soon as the highest edge of the local boundary is no longer the highest edge in the global boundary, the propagation on p is stopped. Then, the computation token will be sent to the process owning the highest edge in the global boundary, to resume the propagation on its block.

At the end of the computation, the pair (τ, σ) is stored on the process that owns τ, as the boundary of another propagation may reach τ and a comparison between the two originating 2-simplices may be required. The process that owns σ does not have to be aware of which simplex completed the pair.

DistributedPairCriticalSimplex hence generates two types of messages: computation tokens and boundary updates. Boundary updates correspond to either a merge order between two global-local boundaries, an addition of an edge to a local boundary or an update of the highest edge in a global boundary. The received boundary updates have to be performed in a particular order, to ensure two properties: (i) updates from potentially multiple processes with regard to one particular propagation need to be received and processed in the same order they were created in, (ii) updates from one process with regard to potentially multiple propagations need also to be received and processed in the same order they were created in. Other orders may result in an incorrect outcome. For (i), the property can be ensured by following a round-by-round design with alternating communication and computation steps. Processing the boundary updates sequentially for each process will ensure property (ii). However, messages sent by different processes can be processed in any order as long as they involve different propagations.

The overall algorithm *DistributedPairCriticalSimplices* (Algorithm 6) revisits *PairCriticalSimplices* (Algorithm 2) with distributed processes, and following this round-by-round design. All the propagations are first computed locally using multi-thread parallelism. Once all propagations are either completed or their computation token needs to be sent to another process, the communications start. First, the boundary updates are exchanged and processed. Then the computation tokens are exchanged and the new propagations are computed in parallel using threads. These rounds of communications and computations are performed until all critical 2-simplices are paired.

Similarly to Section 4.4.3, our implementation reduced the size of data structures and vectors to the local number of critical simplices and used maps to convert a global simplex index to its index in vectors.

4.5.2 Anticipation of propagation computation

The algorithm presented in the previous section presents a critical flaw: in the worst case scenario, for boundaries stretched out on multiple pro-

Algorithm 5 DistributedPairCriticalSimplex

Input: An unpaired critical 2-simplex σ
Output: A temporary pair of $\mathcal{D}_1$

if *GlobalLocalBoundary*$(\sigma) == 0$ **then**
  addEdge(*GlobalLocalBoundary*$(\sigma), \partial\sigma$)
end if
while *GlobalLocalBoundary*$(\sigma) \neq 0$ **do**
  $\tau \leftarrow \max($*LocalBoundary*$(\sigma))$
  $\tau_p \leftarrow \max($*GlobalBoundary*$(\sigma))$
  if $\max(\tau, \tau_p) == \tau_p$ **then** // τ_p is the highest edge
    UpdateMaxGlobal$(\sigma, \tau,$ *GlobalBoundary*$(\sigma))$
    Mark computation token of σ for sending to p
    return
  end if // τ is the highest edge
  if τ is not a critical simplex **then**
    addEdge(*GlobalLocalBoundary*$(\sigma), \partial($*Pair*$(\tau))$
  else
    if *Pair*$(\tau) == \emptyset$ **then**
      addPair(σ, τ) // τ is unpaired
      UpdateMaxGlobal$(\sigma, \tau,$ *GlobalBoundary*$(\sigma))$
      break
    else
      $\sigma_\tau \leftarrow$ *Pair*(τ) // τ has already been paired to σ_τ
      if $\sigma_\tau < \sigma$ **then**
        MergeGlobalLocalBoundaries(σ, σ_τ)
      else
        addPair(σ, τ) // σ is older and the true death of τ
        UpdateMaxGlobal$(\sigma, \tau,$ *GlobalBoundary*$(\sigma))$
        Pair$(\sigma_\tau) \leftarrow \emptyset$
        PairCriticalSimplex(σ_τ) // Resume for σ_τ
      end if
    end if
  end if
end while

Algorithm 6 DistributedPairCriticalSimplices

Input: Set C_2 of unpaired critical 2-simplices
Input: Set C_1 of unpaired critical 1-simplices
Output: Persistence diagrams $\mathcal{D}_1$

for $j \in C_2$ *in parallel (multi-threading)* **do**
  DistributedPairCriticalSimplex(σ_j)
end for
while Global number of terminated propagations $< |C_2|$ **do**
  // Perform global boundary updates
  Send boundary updates to other processes
  Receive boundary updates from other processes
  Update received boundaries
  // Resume computations with tokens
  Send computation tokens to other processes
  Receive computation tokens from other processes
  for all received tokens σ *in parallel (multi-threading)* **do**
    DistributedPairCriticalSimplex(σ)
  end for
end while
for $j \in C_1$ **do** // Extract pairs from boundary computation
  $\mathcal{D}_1 \leftarrow \mathcal{D}_1 \cup (\sigma_j, Pair(\sigma_j))$
end for

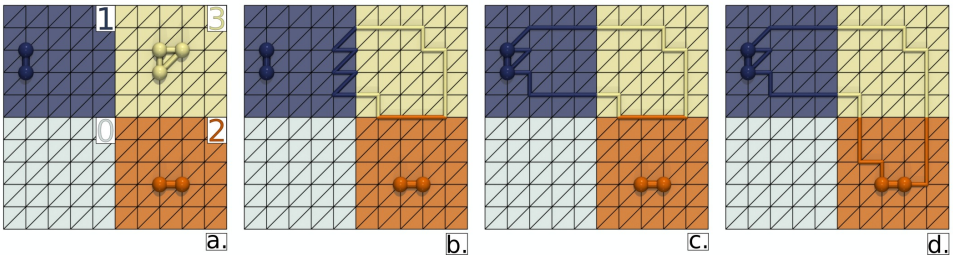


Figure 4.6 – *Example of anticipation of propagation computation for 4 processes (process 0 in light blue, process 1 in dark blue, process 2 in orange and process 3 in light yellow). A critical triangle (in yellow) is the starting point of the distributed homologous propagation. The boundary is propagated as described in Algorithm 5, however the computation token is not sent when the global highest edge of the boundary becomes located on process 1 (Sub-figure b). Instead, the propagation is continued on process 3 until the number of propagation iterations (i.e. while loop iterations in Algorithm 5) reaches a predefined counter. Then, the computation token is sent to the process owning the global highest edge (here, process 1). The boundary is propagated on process 1 (Sub-figure c) until an unpaired critical edge is reached (thick, dark, blue edge). At this point, the global highest edge is located on process 2 (orange). The blue critical edge is therefore not paired with the critical triangle and the computation token is sent to process 2 that resumes the propagation on its domain. When reaching its own critical edge, as the global highest edge is still located on the domain of process 2, the propagation ends here and the pair is created on process 2 (Sub-figure d). Instead of having to send the computation token back and forth between process 1, 2 and 3 to produce the final pairing, the anticipation of propagation enables to exchange the token only twice.*

cesses, the maximum may change process every time a simplex is added to the propagation, resulting in repeated exchanges of the computation token. This may generate an extremely high number of communications. Anticipating this back and forth is possible by changing slightly the original algorithm as shown in Figure 4.6. Instead of stopping the computation and sending the computation token to another process as soon as the highest edge of the boundary is located on another process, we further the computation regardless on the current process, until either the number of propagation iterations (i.e., while loop iterations in Algorithm 5) reaches a predefined counter (arbitrarily equal to 0.01% of the number of triangles of $\mathcal{M}'_p$) or until an unpaired critical simplex c is reached. Only then is the computation token sent to the process that owns the highest edge. Not pairing the potential simplex c ensures that the propagation never expands too far, leading to potentially incorrect pairs, which would have been difficult to detect and correct afterwards.

4.5.3 Overlap of communication and computation

There are two limitations to the previous algorithm that we want to address in this section: thread idle time and cost of communication. Indeed, at the end of each computation round, when waiting for all work to complete, there is often just a few propagations being computed, resulting in significant idle time when using many threads. We aim at reducing this idle time by triggering a communication round before all computations are finished. On the other hand, the cost of communication can be reduced by effectively overlapping communications with computations at the MPI level.

A dedicated *communication thread* can solve both these problems. We preserve the round-by-round structure for the communications to ensure that the update of global-local boundaries is processed in the right order (see Section 4.5.1), but the communication rounds are now triggered by the communication threads. In each MPI process, the communication thread sends and receives messages, updates boundary data and creates one OpenMP task for each propagation, while the other *compute threads* process the propagations.

Even if we aim at starting earlier each communication round, these are not triggered as soon as one message can be sent. Making the communication thread wait a little and sending multiple messages in one MPI communication at once limits indeed the number of communications. It

ensures that the MPI layer is not overloaded with numerous messages, and limits the number of OpenMP atomic operations performed by the communication thread: these atomic operations are required for a correct synchronization with the compute threads. Messages will only be sent if there are no tasks left to be computed within the current process or if the number of messages waiting to be sent by the current process is above a certain threshold. This threshold is set dynamically to increase reactivity as the computation progresses. At first, it is equal to 0.01% of the local number of unpaired 2-simplices. Then, at every round of communications, it is updated using the remaining global number of unpaired critical 2-simplices to add reactivity to the communications. The communication thread also performs the update of global-local boundaries. This can be done in parallel of propagation computation as updating global-local boundaries from the current round will not interfere with the computation of propagations from previous rounds. This is because the updates are not directly related to the current computation tokens.

The compute threads can now continuously process the local and incoming propagations through a task pool filled by the communication thread, harnessing more efficiently the intra-node multi-core parallelism. The idle time of the threads is therefore significantly reduced as compute threads no longer have to wait at the end of each computation round and the cost of communication is effectively hidden with the overlap of communication with computation as usual with a communication thread [DT16, HSSW11].

4.6 Results

For the following results, we rely on Sorbonne Université's supercomputer, MCMeSU, which has replaced MeSU-beta (Section 3.6) in 2024. MCMeSU contains 48 nodes of 32 cores each. Each node is composed of 2 AMD EPYC 7313 Milan CPUs with 256GB of RAM. The nodes are interconnected with Mellanox Infiniband. In our tests, we use up to 16 nodes (512 cores total), with one MPI process and 32 threads per node to minimize MPI communications and synchronizations as well as the memory footprint. When using a communication thread (see Section 4.5.3), we rely on 31 compute threads only. Our algorithm is implemented in C++ with MPI+OpenMP within TTK [TFL+17, BMBF+19]. The correctness of our implementation was checked for all test datasets, by comparing our outputs against those generated by DMS (which were already compared

to DIPHA's for triangulated voxel data, see [GVT23] for more details). Tests of strong and weak scaling are conducted to study the performance of our algorithm. The preconditioning time for TTK's distributed triangulation (Section 3.2) is not accounted for in this work (it is negligible for regular grids [LWG+24]). Specifically, when reporting execution times, we consider that the data is already distributed among the nodes, in the form of ghosted blocks (Section 3.2), which is a standard input for analysis pipelines in distributed environments. Only the execution time of our algorithm and its direct preconditioning are measured. Our algorithm is also compared to the original DMS algorithm as well as to DIPHA.

4.6.1 Datasets

The performance of our software has been evaluated using multiple datasets, selected to demonstrate a broad spectrum of cases. These datasets are sourced from publicly available repositories [Kla20, TTK20]: Backpack, Isabel, Wavelet, Isotropic pressure, Magnetic reconnection, Synthetic truss, Elevation (pathological case with a persistence diagram of one class of infinite persistence in $\mathcal{D}_0$), Random (pathological case with a high number of spatially evenly distributed persistence pairs). See Appendix A for more details on the datasets used in this chapter.

For the strong scaling benchmarks, all datasets were resampled to 512^3 via trilinear interpolation, except for Random, that was resampled to $512^2 \times 256$ as the execution time for this dataset is particularly long for 512^3 with all tested softwares, making it unpractical to manage. This smaller size still makes Random the dataset with the longest execution time, as it is our worst case scenario.

For the weak scaling benchmarks, the size of the input (number of vertices) doubles each times the number of nodes doubles (by doubling the number of vertices along one, alternating, dimension). The initial size on one node is the same as the strong scaling one ($512^2 \times 256$ for Random, 512^3 for the others). The datasets were re-sampled in different ways depending on the size of the original data: Isabel, Backpack, Magnetic Reconnection have been up-sampled, whereas Synthetic truss and Isotropic Pressure have been down-sampled. Random was generated for its biggest weak scaling case and then down-sampled to smaller datasets. Elevation was generated for each size, so that it always has only one pair in its diagram. Due to its symmetry, Wavelet was generated for the largest weak

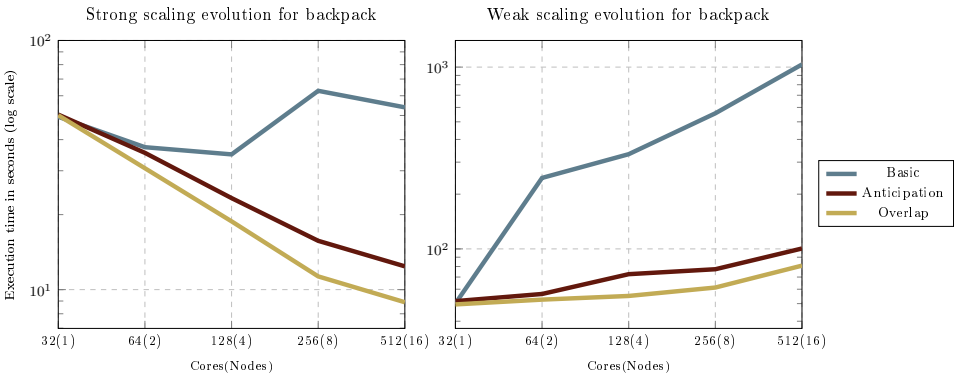


Figure 4.7 – *Performance impact of the different $\mathcal{D}_1$ versions on the overall DDMS execution time for Backpack.* Basic *corresponds to the first version for $\mathcal{D}_1$ described in Section 4.5.1,* Anticipation *to the second one described in Section 4.5.2, and* Overlap *to the final one described in Section 4.5.3.*

scaling case and then resized by being cut in two along each dimension alternatively.

4.6.2 Performance improvements

We start by assessing the performance improvements of our different versions for computing $\mathcal{D}_1$, namely: *Basic*, the initial version (see Section 4.5.1); *Anticipation*, that implements the anticipation of computation for $\mathcal{D}_1$ (see Section 4.5.2) and *Overlap* that iterates on *Anticipation* and adds the overlap of communication and computations thanks to the communication thread (see Section 4.5.3). As shown in Figure 4.7, the anticipation of computation dramatically improves the overall DDMS performance, making *Anticipation* 6 times faster than *Basic* on 16 nodes in strong scaling and over 12 times in weak scaling. *Overlap* also improves the performance, by adding reactivity to the execution. On 16 nodes, it reduces the overall execution time by 20% in weak scaling and 28% in strong scaling. These results validate and justify our modifications, which are hence necessary to efficiently deploy such TDA algorithms on multiple nodes.

From now on, we will only consider the *Overlap* version, whose detailed execution profile is presented in Figure 4.8. For both strong and weak scalings, the *Array Preconditioning* step, which corresponds to the computation of global order of vertices (see Section 4.3), is quite short and minority. The discrete gradient step scales very well as expected, which is an important source of overall performance gains. The computations of $\mathcal{D}_0$ and $\mathcal{D}_2$ scale also well, in both weak and strong scaling. This also applies to the *Extract & Sort* step which corresponds to the extraction and local sort of critical simplices for all dimensions (see Section 4.3), and which

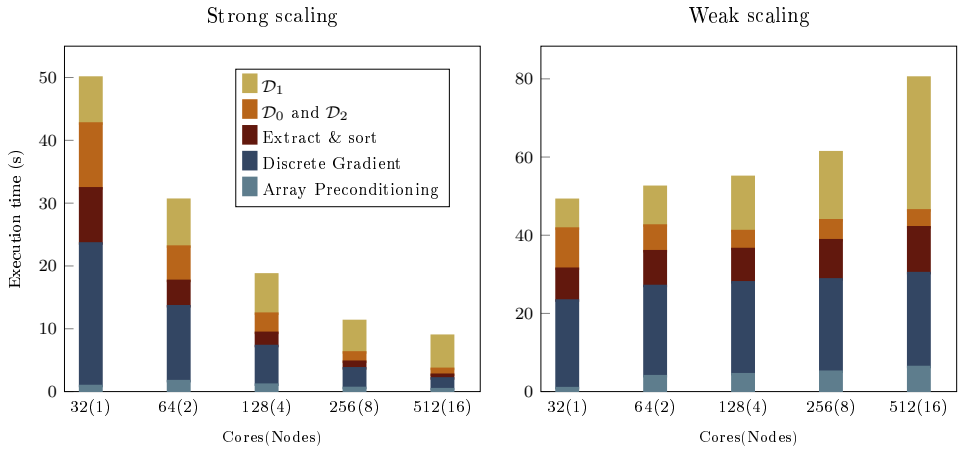


Figure 4.8 – *Execution time of each step of DDMS, for strong (left) and weak (right) scalings for Backpack.*

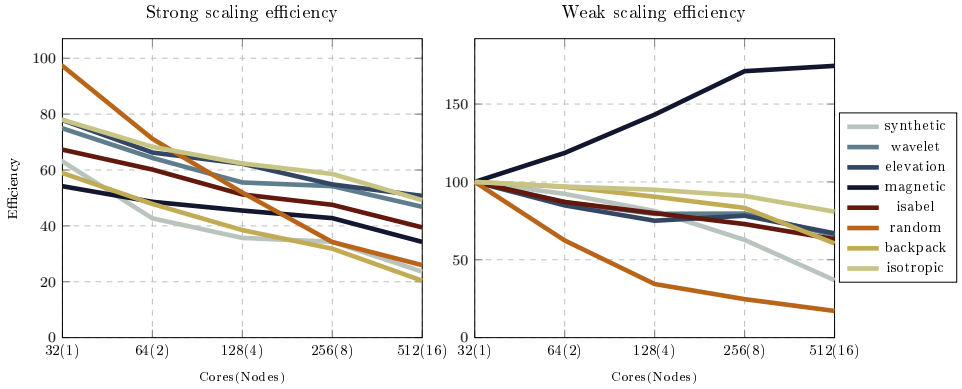


Figure 4.9 – *Parallel efficiency of DDMS for strong (left) and weak (right) scaling.*

is performed independently on each process. Only the computation of $\mathcal{D}_1$ (which is the most intensive in terms of time complexity) scales unfavorably in both strong and weak scaling. But, thanks to our successive improvements presented in Section 4.5, there is no strong performance loss and we still manage to obtain overall significant performance gains when increasing the number of nodes.

4.6.3 Strong scaling

The results for all datasets in strong scaling are shown in Figure 4.9 (left) in terms of parallel efficiency with respect to the execution on one core (see Section 2.2.6). The execution times are also available in Figure 4.10. With the exception of Random, the efficiencies of all datasets fit within the range of 55% to 80% on one node and of 20% to 50% on 512 cores. This shows the scalability of our approach. Though the efficiencies decrease as the number of cores increases, Figure 4.10 shows that the execution times

continue to decrease even on 512 cores, with most datasets eventually requiring less than 20 seconds on 512 cores.

Random behaves a bit worse than the other datasets, presenting the biggest drop in efficiency: from the best efficiency on one node (close to 100%) to one of the worse one on 512 cores (close to 26%). This is explained by the output-sensitivity of our algorithm. The more pairs are present in the output persistence diagram, the greater the workload. Another factor is the spatial placement of the birth and death of a pair within the dataset. The further apart they are, the longer the computation will be. Random is one of the noisiest of our datasets (with Magnetic Reconnection and Synthetic Truss), however, unlike those two datasets its pairs are evenly distributed. Consequently, the birth and death tend to be further apart spatially, requiring more work and more communications. This leads to very good efficiency on one node, but this quickly becomes a performance issue as the number of nodes increases, leading to such an efficiency drop.

4.6.4 Weak scaling

The weak scaling results for all datasets are shown in Figure 4.9 (right) in terms of parallel efficiency (see Section 2.2.6). The weak scaling efficiency is better than the strong scaling one for most datasets. This is partly due to the fact that doubling the dataset size through re-sampling often results in less than a twofold increase in the number of critical simplices, and hence in computational workload (given the output sensitivity of our algorithm). For most datasets, the efficiency is in the range of 35% to 80% on 16 nodes, which again shows the scalability of our approach. There are however two exceptions: Random and Magnetic Reconnection. For Random, the efficiency eventually drops lower to 17 % for the same reasons as in strong scaling: its pairs are numerous and spatially stretched out. For Magnetic Reconnection, the efficiency largely exceeds 100%. This is due to the up-sampling of the original dataset that barely multiplies the number of pairs by a factor of 1.6 between 1 and 16 nodes. This is most likely because the topological features are already numerous and unevenly distributed across the dataset. While this also applies to other datasets, such as Isabel, the other specificity of Magnetic Reconnection is that it produces the most pairs out of all the datasets on 512^3 (35 millions). The execution time for $\mathcal{D}_0$, $\mathcal{D}_1$ and $\mathcal{D}_2$ is therefore substantial compared to the computation of other steps, such as the gradient. As the number of nodes increases, even though the size of the dataset increases, each process actually computes

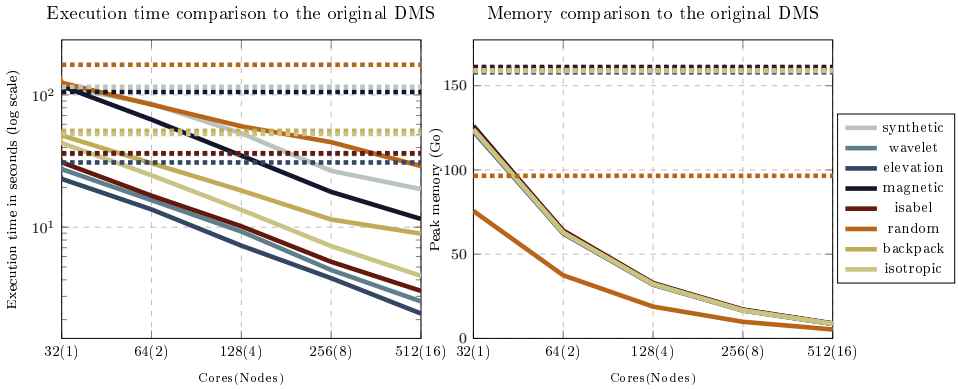


Figure 4.10 – *Comparison between DMS (dotted lines) and DDMS (full lines) in terms of execution time (left) and per-node peak memory footprint (right).*

less and less pairs leading to lower execution times. The same applies for Isabel, but has no impact on the overall execution time, as the computation of $\mathcal{D}_0$, $\mathcal{D}_1$ and $\mathcal{D}_2$ is originally negligible compared to the gradient computation.

4.6.5 Performance comparison

Comparison with DMS: In Figure 4.10 are shown the execution time and per-node peak memory footprint of our DDMS algorithm compared to the original DMS algorithm on one node. For all datasets, the execution times of both algorithms are comparable on one node, the distributed DDMS algorithm executing slightly faster than its shared-memory counterpart for all datasets, except Synthetic Truss and Magnetic Reconnection. The DDMS extra cost for these two datasets is due to changes in the algorithm (for example, the removal of arc collapse in the $\mathcal{D}_0$ and $\mathcal{D}_2$ computations in Section 4.4.3) or to more costly data structures (Section 4.4.2), enabling MPI execution. The overhead is however very limited and executing DDMS on two nodes already outperforms DMS for all datasets.

In terms of peak memory footprint, DDMS uses significantly less memory for all datasets. This is due to our reduction of vector size to the number of critical simplices as mentioned in Section 4.4.3 and Section 4.5.1. This allowed DDMS to produce an overall smaller footprint on one node.

Comparison with DIPHA: We now compare our algorithm to DIPHA, to our knowledge the only publicly available MPI implementation (without multithreading) for persistence diagram computation. The DIPHA execution time is measured using the built-in *benchmark* mode and corresponds to its total execution without the I/O time. We start by comparing the execution times in strong scaling on Figure 4.11. On one core, DDMS

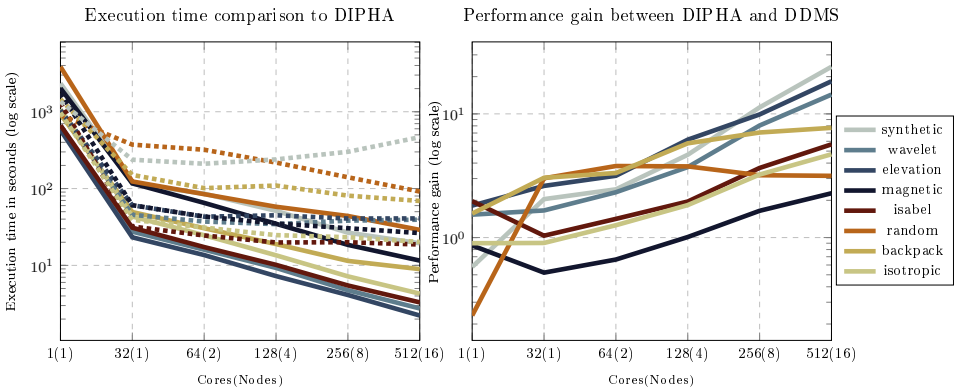


Figure 4.11 – *Comparison based on execution time (left) between DIPHA (dotted lines) and DDMS (full lines) and performance gain (right) for a strong scaling setting. The performance gain on a given number of cores is defined as* t_{DIPHA}/t_{DDMS}*, with* t_{DIPHA} *and* t_{DDMS} *the execution times of DIPHA and DDMS respectively. A performance gain higher than 1 means that DDMS is faster than DIPHA.*

outperforms DIPHA only on the smoother datasets (Elevation, Wavelet, Isabel and Backpack), as these can really harness the preconditioning of the discrete gradient to speed up the rest of the computation. On multiple nodes, DDMS scales much better and hence outperforms DIPHA for all datasets starting from 4 nodes. Notice than on one and two nodes, only one dataset out of eight (Magnetic Reconnection) is more efficiently processed by DIPHA. Moreover, considering both execution times and scaling, the worst case dataset for DIPHA (Synthetic Truss) scales relatively well for DDMS, whereas the worst case for DDMS (Random) is always processed faster (up to $\times 3$) by DDMS on more than one core. Finally, the average speedup for all datasets is around $\times 8$ on 512 cores, showing a substantial performance gain of DDMS over DIPHA.

Figure 4.12 compares the memory consumption of both approaches. On the fewest number of nodes, DIPHA requires a lower footprint than DDMS for all datasets but the memory scalability is much better for DDMS. Hence, on the largest number of nodes the peak memory footprint of DDMS ends up being smaller than that of DIPHA. DDMS divides indeed by almost two its memory usage (with a small overhead due to the ghost cells) every time the number of cores is multiplied by two (Section 3.2).

For DIPHA, its data distribution is more conducive to memory imbalance as the final reduced columns are stored on the process that completed their reduction, meaning that some process may store a significantly higher number of reduced columns than other processes. This im-

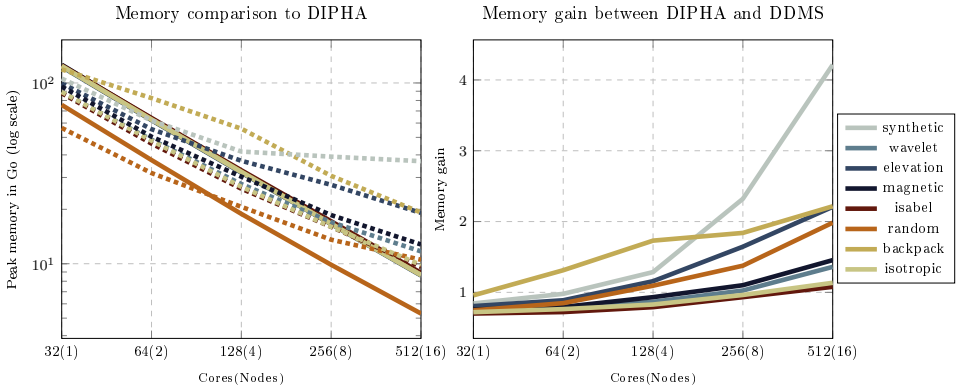


Figure 4.12 – *Maximum over all nodes of the per-node peak memory footprint (left) for DIPHA (dotted lines) and DDMS (full lines) and memory gain (right) for a strong scaling setting. The memory gain on a given number of nodes is defined as* m_{DIPHA}/m_{DDMS}*, with* m_{DIPHA} *and* m_{DDMS} *the maximum over all node of the per-node peak memory footprint for DIPHA and DDMS respectively. A memory gain higher than 1 means that DIPHA uses more memory than DDMS.*

balance is more likely as the number of processes increases, leading to DIPHA having a larger per-node peak memory footprint than DDMS.

4.6.6 Example

We ran DDMS on a larger dataset (Turbulent Channel Flow [Kla20]) to show our algorithm's capability to handle massive datasets. This dataset is a direct simulation of a fully developed flow at different Reynolds numbers in a plane channel. The scalar field is the three dimensional pressure field and has been converted to single-precision floating-point numbers for a lower memory consumption. The computation was run on a subset of the original dataset of size 2048x1920x1536 which is approximately 6 billion vertices. The memory bottleneck of our implementation mainly lies in the computation of the discrete gradient, which is the most memory-consuming step. For this example, we used TTK's compile option `TTK_ENABLE_DCG_OPTIMIZE_MEMORY` which optimizes the gradient memory footprint (by trading on its computation time). The execution was performed on 16 nodes of 32 cores and 256GB of RAM each (512 cores and 4096GB of RAM total). The persistence diagram shown in Figure 4.13 was computed in 174 seconds and contained a little under 19 million pairs. For comparison, out-of-core techniques [Wag23] require several hours of computation for datasets of similar size, which further illustrates the practical interest of our work for high-performance computing contexts.

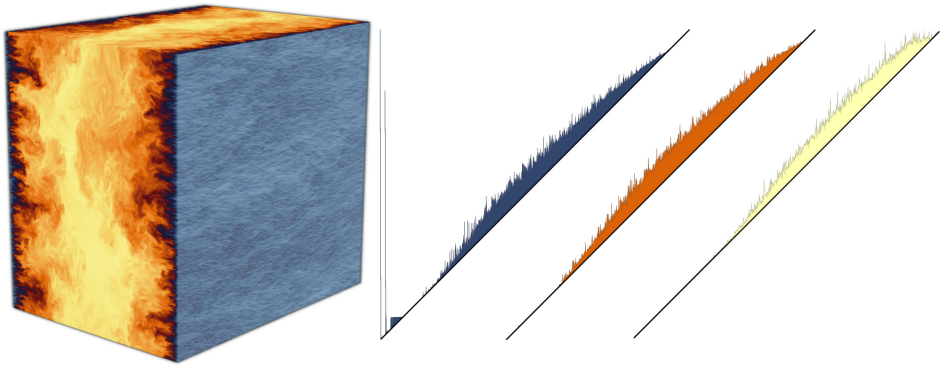


Figure 4.13 – *Persistence diagram on a subset of Turbulent Channel Flow (2048x1920x1536, 6 billion vertices). The dataset (left) is the pressure field of a direct simulation of a turbulent flow in a plane channel. The execution was performed on 16 nodes of 32 cores and 256GB of RAM each (512 cores and 4096GB of RAM total). The persistence diagram (right: $\mathcal{D}_0$ in blue, $\mathcal{D}_1$ in orange, $\mathcal{D}_2$ in yellow) was computed in 174 seconds (19 million pairs).*

4.6.7 Limitations

For completeness, we discuss here some limitations of our work. Our implementation of the DDMS algorithm only supports structured grids at the time of writing this manuscript, even though TTK support both structured and unstructured grids. Although the original DMS implementation has been validated on both, additional tests would be required to validate our DDMS implementation on unstructured grids. Additionally, several specialized domain representations popular in scientific computing such as adaptive mesh refinement (AMR) are not supported by TTK and therefore by our work.

The primary memory bottleneck of our implementation lies in the computation of the gradient. Indeed, this computation is not only costly in terms of execution time but also in terms of memory footprint, as discrete vectors are computed for all simplices of all dimensions. This generates a significant memory footprint, which may be addressed in the future by an improved compact encoding of the discrete vectors, exploiting the regular structures of structured grids.

Finally, similarly to DMS, since our work also exploits Robins' discrete gradient [RWS11], it suffers from the same limitation regarding an extension to higher dimensions. In particular, strong guarantees on the restriction of critical cells upon homology changes of the sub-complexes are provided in up to three dimensions. Beyond, additional, spurious critical cells may appear, limiting the effectiveness of our approach.

4.7 Summary

This chapter introduced a new algorithm for the efficient computation of the persistence diagram on scalar data in a distributed-memory setting using a hybrid MPI+thread parallelization. The performance of our algorithm was tested on a set of 8 datasets representing various use cases. Thanks to our algorithmic improvements for $\mathcal{D}_0$, $\mathcal{D}_1$ and $\mathcal{D}_2$, and to the scalability of the discrete gradient computation, we have shown that our approach delivers significant speedups on up to 512 cores, with parallel efficiencies up to 50% in strong scaling and up to 80% in weak scaling. DDMS also outperforms DIPHA, the only publicly available implementation for computing the persistence diagram in distributed-memory, by a factor of $\times 8$ on average. It produces a slightly larger memory footprint than DIPHA on few nodes, but a smaller one on 16 nodes.

Additionally, we showed that our algorithm is capable of processing massive datasets by running DDMS on a larger dataset (Turbulent Channel Flow [Kla20]) of 6 billion vertices in under 3 minutes on 512 cores.

Conclusion

5

In this thesis, we focused on high-performance approaches for the distributed-memory computation of topological representations of very large datasets. The aim is to provide efficient and robust methods and tools to compute topological representations of larger than ever before datasets in a high-performance context (i.e. on a supercomputer). An effort has also been made to ensure accessibility and re-usability of our contributions. Our work aims at being a foundation stone in building a wide-ranging unified ensemble of methods and implementations for computing topological representations in a distributed-memory setting.

5.1 Summary of Contributions

The contributions of this thesis are twofold: we first added distributed-memory support to the core of the existing library the *Topology ToolKit* and then we used these new features to provide a new efficient distributed-memory approach for the computation of persistence diagrams. This work has been integrated into TTK and significantly enhances its applicability to large-scale data analysis tasks, making it a valuable tool for researchers and practitioners dealing with complex topological computations.

5.1.1 A Software Framework for Distributed Topological Analysis Pipelines

In this first effort towards distributed-memory computation for Topological Data Analysis, we introduced in Chapter 3 a software framework designed to support topological analysis pipelines within a distributed-memory environment. Specifically, we integrated this new support into TTK, using the *Message Passing Interface* (MPI) while preserving its shared-memory parallelism. Modifications were applied to TTK's data structure,

the triangulation, to efficiently adapt to a distributed-memory setting and to provide key features for future implementations of topological algorithms in such a setting. An additional software infrastructure was developed, at both a fine grain and high level, to facilitate the construction of advanced topological pipelines. The resulting software offers a unified framework supporting both triangulated domains and regular grids. Several existing algorithms were adapted to a distributed-memory context, resulting in hybrid MPI+thread parallel implementations. A taxonomy was proposed to categorize these algorithms by their communication requirements. Our performance evaluations demonstrated parallel efficiencies ranging from 20% to 80%, depending on the algorithm, with minimal overhead introduced by our MPI-specific preconditioning. Finally, to showcase TTK's new large scale capabilities, we presented an advanced analysis pipeline that integrates multiple algorithms and processes the largest publicly available dataset we have found, comprising 120 billion vertices, on 64 nodes of a supercomputer (for 1,536 cores in total).

5.1.2 Efficient Computation of Persistence Diagrams for Massive Scalar Data

The previous section focused on a general unified framework. Distributed-memory support was added to several topological algorithms, however these algorithms were relatively simple to parallelize. In this second stage, described in Chapter 4, we used our framework and focused on providing distributed-memory support to a much more complex algorithm: the Discrete Morse Sandwich, currently the most efficient shared-memory parallel approach to compute persistence diagrams. We provided a new method, the Distributed Discrete Morse Sandwich, a hybrid MPI+thread algorithm that enables efficient computation of persistence diagrams for scalar fields of significant sizes. This method is composed of two different algorithms: (i) the first is an extension of DMS's unstable and stable set compression procedure, and provides a self-correcting distributed pairing for $(0,1)$ and $(d-1,d)$ critical simplices and (ii) the second is an extension of DMS's *PairCriticalSimplices* procedure and exploits a communication thread to improve reactivity and performance for the pairing of $(1,2)$ critical simplices. Extensive testings showed that the overall procedure is output sensitive and provides substantial gains over the original DMS approach as well as *Dipha* (8 times faster on average on 512 cores), the reference method for computing persistence diagrams in a distributed

setting. To showcase TTK's new capabilities, the persistence diagram of a dataset of 6 billion vertices was computed, using 512 cores, in 174 seconds.

5.2 Discussion

The limitations and discussions regarding our various contributions have been addressed in their respective chapters. Nonetheless, we would like to underscore a few additional noteworthy aspects related to distributed Topological Data Analysis and the Topology ToolKit.

TTK algorithms can be accessed through dependency-free standalone C++ executables. Though it is technically possible to execute these standalone executables in a distributed-memory setting, we have not done so. We have accessed TTK algorithms using ParaView instead (both its GUI and Python3 APIs), as it can easily perform operations such as distributing the data across nodes or generating the ghost simplices. In their current state, standalone executables are not very practical in a distributed-memory setting from a user's point of view, as they would require programming a significant amount of code to perform the operations we have relied on ParaView to execute.

TTK's MPI extension is heavily dependent on ParaView. It facilitated numerous computation steps, such as data distribution, distributed file writing and reading or ghost generation. It was a welcome help as these operations can be quite complex and time-consuming to implement. However, it also means that we have to adapt to the decisions made by ParaView's developers. For example, ParaView's MPI extension technically does not support thread. Only the basic thread support level (`MPI_THREAD_SINGLE`) is possible. In practice, there is a workaround using environment variables at runtime but it is MPI-implementation dependent and not conform to the MPI standard.

AMR (Adaptive Mesh Refinement) grids are not implemented in TTK. Though it was somewhat limiting when TTK was restricted to a shared-memory setting, its addition may become a bit more pressing. Indeed, this type of grid tends to be used a lot in applications that process voluminous datasets, such as astrophysics and cosmology, to reduce their memory footprint and computation time. AMR therefore becomes a more important feature when increasing the scale of the computation. Now that TTK can be used a distributed-memory setting, more applications for which TDA algorithms would be useful may require AMR grids. This is

not a straightforward or simple addition as it would require to implement a new triangulation data structure.

An additional limitation of our work is that not all of TTK algorithms have been adapted to distributed-memory execution. In fact, for now, though such support has been added to several algorithms, a minority of TTK algorithms are parallelized for a distributed-memory setting. We will discuss this in more detail in the next section.

5.3 Perspectives

There are four axes of future work: improving and extending the core of the framework, adding distributed-memory support to more algorithms, adding support for a different type of parallelism and building on the distributed-memory algorithm to implement out-of-core computation.

5.3.1 Investigating the cost of ghost simplices generation

The generation of ghost simplices is the costliest MPI-related preconditioning step, in terms of execution time, as seen in the results of Chapter 3. ParaView's algorithm focuses on robustness and genericity to provide a reliable solution for its many data formats. This makes for a very costly step in our preconditioning. Designing our own algorithm could allow us to implement different algorithms for each of our triangulation types (i.e. implicit and explicit) and exploit properties specific to each of our triangulation data structures, such as the regularity of the implicit triangulation. This would most likely allow to speed up the computation for certain types of triangulation.

5.3.2 Adding distributed-memory support to NC and DIC algorithms

The port of No Communication (NC) and Data-Independent Communications (DIC) algorithms (such as ContinuousScatterPlot, ManifoldCheck, DistanceField, JacobiSet or FiberSurface) is relatively straightforward (see Section 3.5.1 for a definition of the categories). For DIC algorithms, the initial step entails identifying the data to be exchanged, the processes involved in the exchange, and the appropriate timing for performing these communications. For NC algorithms, no exchange between processes take place. Then, the implementation can be done in TTK, using TTK's MPI-API as well as low-level MPI directives (for specific communications). The necessary modifications would likely demand only a limited understand-

ing of distributed-memory computing and MPI. This could for example be done during a hackathon.

5.3.2.1 Adding distributed-memory support to DDC algorithms

For Data-Dependent Communications (DDC) algorithms (such as Discrete Morse Sandwich, topological simplification, contour tree or Rips complex computations), the port may be much more complicated. For each of these DDC algorithms, their distributed-memory parallelization may be a substantial research problem. For example, the Discrete Morse Sandwich algorithm required significant changes and time-investment to provide the results shown in Chapter 4. In continuity of our work, one could explore the use of the discrete gradient as an accelerating preconditioning step for other algorithms. Lukasczyk et al. have used a descending manifold to accelerate the computation of an augmented merge tree, a representation very similar to the persistence diagram, in [LWW+23]. Therefore, the computation of the merge tree may be a good candidate. Another algorithm that could see its distributed support relying on similar methodology to DMS is TTK's topological simplification, that uses mechanisms similar to DMS's homologous propagation.

5.3.2.2 Porting on many-core architectures

As said in Section 2.2.5, in this work, we did not produce algorithms nor implementations adapted to GPUs. Our first goal was to address the memory bottleneck of current TDA algorithms by adding support to a distributed-memory setting. However, efforts to adapt TDA algorithms to many-core architectures exist (such as VTK-m [MAB+24]) and the widespread availability of GPU nodes drive us to consider such direction of research for future work. A hybrid MPI+X setting, where X is a paradigm for many-core architecture, would solve the memory footprint issue while allowing the use of GPUs. NC algorithms, such as Critical Points or the discrete gradient, will most likely result in significant performance gains in such a setting. It may also be the case of compute-intensive DIC algorithms. However, it will be very complex and time-consuming to adapt DDC algorithms to many-core parallelism. For example, in the case of DDMS, though the discrete gradient will likely induce a significant performance gain on a GPU, the rest of the algorithm that pairs the critical simplices would need to be significantly re-designed as it is currently not

well suited for GPUs. Indeed, the approach requires multiple irregular data accesses and uses irregular data structures such as maps.

An avenue for further research is deploying DDMS on heterogenous architectures. Given the gradient algorithm's embarrassingly parallel nature, CPU architectures with integrated GPUs, such as the AMD Instinct MI300A APU, may further accelerate DDMS by leveraging GPU cores for the gradient computation and CPU cores for the computation of $\mathcal{D}_0$, $\mathcal{D}_1$ and $\mathcal{D}_2$.

5.3.2.3 Porting to out-of-core computation

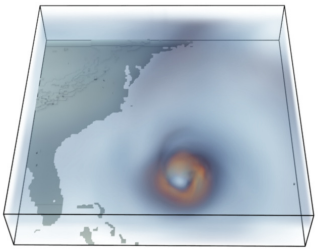
Another research direction is out-of-core computation, which enables processing large datasets on a single compute server by offloading some of the data present in the RAM to some other storage such as disk. Specifically, extensions of our self-correcting pairing (for $\mathcal{D}_0$ and $\mathcal{D}_2$, Section 4.4.3) as well as our anticipation strategy for homologous propagation (for $\mathcal{D}_1$, Section 4.5.2) could be considered for out-of-core contexts, but significant adaptations would be required to maintain the time performance of our approach.

Appendix: Data Specification

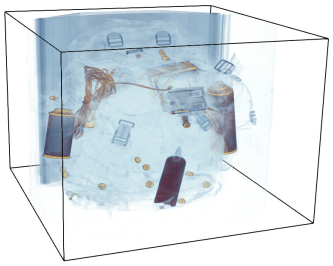
A

This appendix provides a list of the datasets used in this thesis. In particular, we document the data provenance and its representation. All of these ensemble datasets were extracted from public repositories or generated using open-source software. The dimensions given here are the dimensions of the original dataset. In our experiments, datasets have been resized depending on the need of the test.

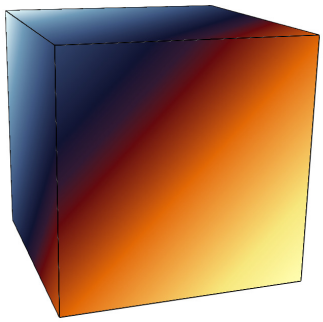
Isabel: Magnitude of the wind velocity in a simulation of hurricane Isabel that hit the east coast of the USA in 2003. This dataset is very smooth and possesses few but significant topological features.
Regular grid of dimensions $250 \times 250 \times 50$.
From [Cono4].



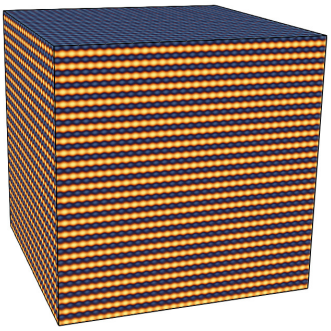
Backpack: Density in the CT scan of a backpack with items. This dataset is spatially imbalanced with regard to its topological features and a good test case for workload balancing.
Regular grid of dimensions $512 \times 512 \times 373$.
From [Kla20].



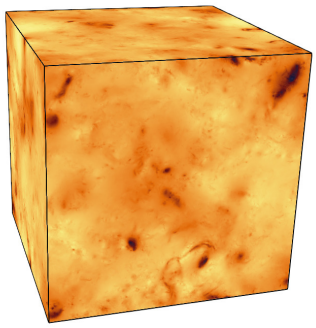
Elevation: Synthetic dataset of the altitude within a cube, with a unique maximum at one corner of the cube and a unique minimum at the opposite corner. This dataset is a pathological case with almost no topological feature.
Generated through ParaView's *Elevation* filter.



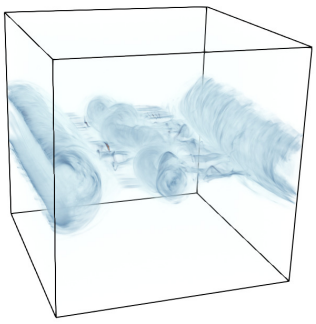
Wavelet: Synthetic dataset of wavelets, following a parametrized sinusoid in 3 dimensions. This dataset is quite smooth and symmetric with small topological features and results in good workload balance.
Generated through ParaView's *Wavelet* filter.



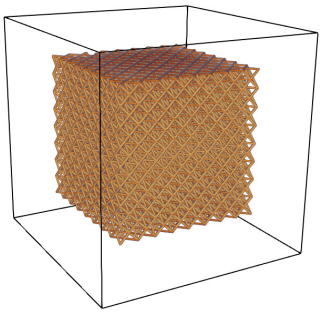
Isotropic pressure: Pressure field of a simulation of forced isotropic turbulence. This dataset is neither smooth nor very noisy with relatively evenly spread out topological features.
Regular grid of dimensions $4096 \times 4096 \times 4096$.
From [Kla20].



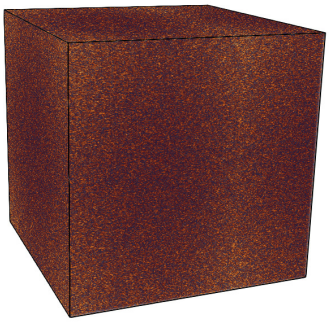
Magnetic reconnection: Simulation of magnetic reconnection, showing interaction between magnetic fields. This dataset is extremely noisy and holds a very large number of topological features.
Regular grid of dimensions $512 \times 512 \times 512$.
From [Kla20].



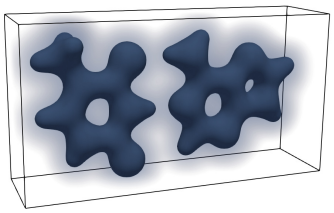
Synthetic truss: Simulated CT scan of a truss with defects. This dataset possesses very rich and symmetric topological features.
Regular grid of dimensions $1200 \times 1200 \times 1200$.
From [Kla20].



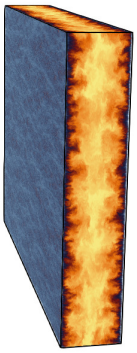
Random: Synthetic dataset of a random field. This dataset is a pathological case where all of its topological features are noise and therefore numerous, small and evenly distributed.
Generated through ParaView's *RandomAttributes* filter.



AT: Simulation of the electronic density on the Adenine Thymine complex. This dataset is quite smooth.
Regular grid of dimensions $177 \times 95 \times 48$.
From [TTK20].



Turbulent Channel Flow: Pressure field of a simulation of a fully developed flow at different Reynolds numbers in a plane channel. To our knowledge, this is the largest publicly available dataset in the field of scientific visualization.
Regular grid of dimensions $10240 \times 7680 \times 1536$.
From [Kla20].



B

Appendix: comparing MPI+thread configurations

This appendix provides additional tests of thread configurations for the MPI+thread implementations of the parallel algorithms described in Section 3.5.3. We compare the following MPI+threads configurations on 1 to 16 nodes for the algorithm *IntegralLines*: 2×12 and 1×24 as well as a pure MPI configuration. The benchmark is run on the algorithm *IntegralLines*, as it is the only Data Dependent Communication (DDC) algorithm (see Section 3.5.1) of the four algorithms parallelized in Section 3.5.3. This makes *IntegralLines* the most likely to be affected by such configuration changes, as this algorithm generates numerous and unpredictable communications. Furthermore, of the four algorithms, it is also the one that could most benefit from dynamic load balancing as it is the most subject to workload imbalance. The higher the number of threads per process, the higher the number of cores involved in the same dynamic load balancing. For the following results, we rely on Sorbonne Université's supercomputer MeSU-beta (the same supercomputer as in Chapter 3). It is a compute cluster with 144 nodes of 24 cores each (totaling 3456 cores). Its nodes are composed of 2 Intel Xeon E5-2670v3 (see Section 3.6 for more details on MeSU-beta).

On top of having different numbers of threads and processes, each configuration possess its own thread and process placements to ensure threads are close to their originating process on the hardware to limit the extra cost caused by NUMA nodes. In the pure MPI configuration, 24 processes are generated per node (with only 1 thread per process). Processes are bound to sockets and mapped by cores (via the corresponding OpenMPI options) in order to fill a node with processes of adjacent ranks.

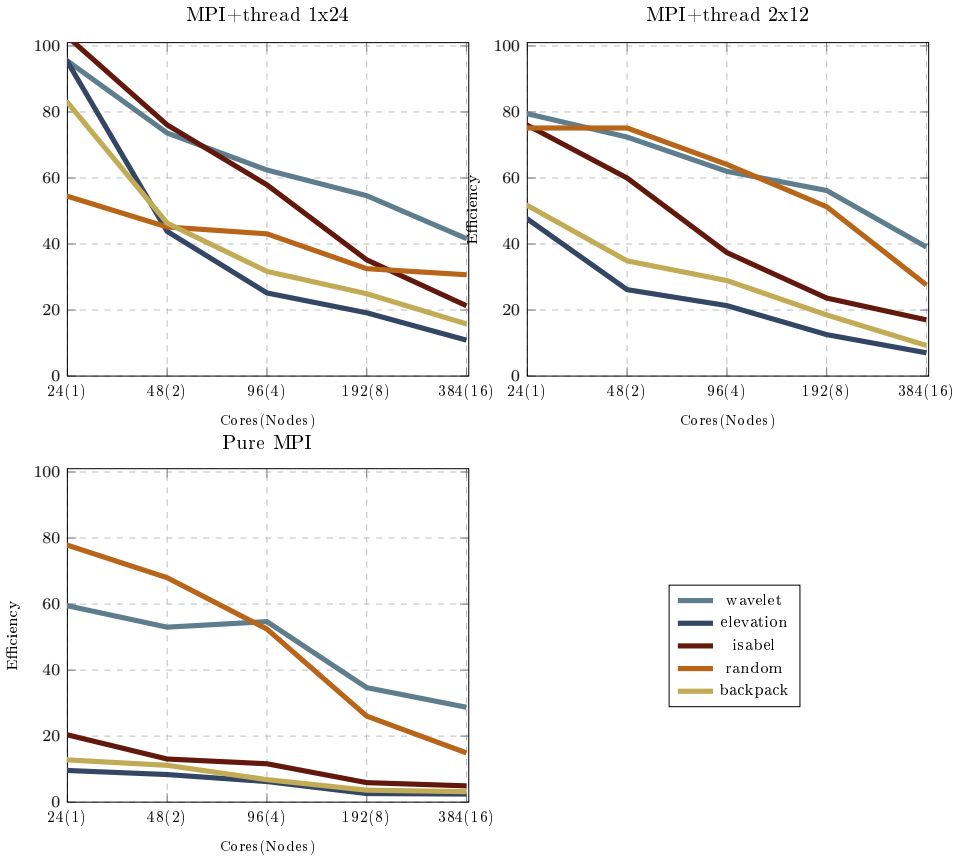


Figure B.1 – *Comparing three configurations of MPI+thread of our novel distributed-memory parallelism support for* IntegralLines*: MPI+thread* 2×12*, where there are 2 processes on each node, with 12 threads each, MPI+thread* 1×24 *where there is 1 process on each node, with 24 threads each and pure MPI, where there are 24 processes on each node (with therefore only 1 thread per process). The benchmark was computed on up to 16 nodes.*

For the 1×24 strategy, each process spawns 24 threads, with 1 process per node. Processes are bound to nothing and mapped by node, with threads bound to cores close to their originating process (via the corresponding OpenMP features). For the 2×12 strategy, each process spawns 12 threads, with 2 processes per node. The 2×12 configuration is of interest due to the hardware of the node: as mentioned before, each node is composed of two processors (here, two Intel Xeon). This architecture is quite common and induces Non Uniform Memory Access (NUMA) effects. The goal of the 2×12 configuration is to reduce these NUMA effects by having one process per processor of the node. In practice, each process is bound to a socket, with its threads bound to the cores close to their originating process. The associated threads will therefore always access data close to that particular processor and this will limit the NUMA effects. However, having more processes induces more MPI overhead. The gain therefore may not be worth the cost.

Results are shown in Figure B.1. The 2×12 configuration yields higher efficiencies than the pure MPI configuration for all datasets, thanks to fewer communications and dynamic load balancing between threads of the same process. On one node, the 2×12 configuration is significantly less efficient than 1×24, as the MPI overhead has not been triggered yet for 1×24. For two nodes or more, the efficiency of most datasets is either comparable or better for 1×24 compared to 2×12, to the exception of Random. This is most likely because there is very little work to do, as the computed integral lines are very short. In that case, the cost of the dynamic load balancing of the OpenMP work is not worth the gain and having more processes induces a more efficient execution.

In conclusion, the 1×24 configuration is the overall most efficient configuration compared to 2×12 and pure MPI. The additional MPI overhead is therefore costlier than the NUMA effects. For the same total number of cores, minimizing the number of processes and maximizing the number of threads provides the best performance.

Bibliography

[AAF+12] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 7490 of *Lecture Notes in Computer Science*, pages 298–299. Springer Berlin Heidelberg, 2012. (Cited pages 34 and 39.)

[AGL05] James Ahrens, Berk Geveci, and Charles Law. ParaView: An End-User Tool for Large-Data Visualization. *The Visualization Handbook*, pages 717–731, 2005. (Cited pages 45 and 63.)

[AN15] Aditya Acharya and Vijay Natarajan. A parallel and memory efficient algorithm for constructing the contour tree. In *IEEE Pacific Visualization Symposium*, pages 271–278, 2015. (Cited page 49.)

[Ban67] T. F. Banchoff. Critical points and curvature for embedded polyhedral surfaces. *The American Mathematical Monthly*, 45(1):245–256, 1967. (Cited page 18.)

[Bar94] S. A. Barannikov. The Framed Morse Complex and Its Invariants. In *ADVSOV*. 1994. (Cited page 97.)

[Bau19] Ulrich Bauer. Ripser: efficient computation of Vietoris-Rips persistence barcodes. *Journal of Applied and Computational Topology*, 5:391–423, 2019. https://github.com/Ripser/ripser/. (Cited pages 48, 97, and 98.)

[BBD+12] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for High Performance Computing. *Parallel Computing*, 2012. (Cited page 39.)

[BDSS18] Alexander Bock, Harish Doraiswamy, Adam Summers, and Cláudio T. Silva. TopoAngler: Interactive Topology-Based Extraction of Fishes. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):812–821, 2018. (Cited page 3.)

[BGL+18] Harsh Bhatia, Attila G. Gyulassy, Vincenzo Lordi, John E. Pask, Valerio Pascucci, and Peer-Timo Bremer. TopoMS: Comprehensive Topological Exploration for Molecular and Condensed-Matter Systems. *Journal of Computational Chemistry*, 39(16):936–952, 2018. (Cited page 3.)

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, 30(8):207–216, August 1995. (Cited page 34.)

[BKR14a] Ulrich Bauer, Michael Kerber, and Jan Reininghaus. Clear and Compress: Computing Persistent Homology in Chunks. In *Topological Methods in Data Analysis and Visualization III*, pages 103–117. Springer International Publishing, 2014. (Cited page 97.)

[BKR14b] Ulrich Bauer, Michael Kerber, and Jan Reininghaus. Distributed computation of persistent homology. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 31–38, 2014. (Cited pages 4, 58, 97, and 98.)

[BKRW17] Ulrich Bauer, Michael Kerber, Jan Reininghaus, and Hubert Wagner. Phat - persistent homology algorithms toolbox. *Journal of Symbolic Computation*, 78:76–90, 2017. https://bitbucket.org/phat-code/phat/src/master/. (Cited pages 48, 49, and 98.)

[BM14] Jean-Daniel Boissonnat and Clément Maria. The simplex tree: An efficient data structure for general simplicial complexes. *Algorithmica*, 70(3):406–427, 2014. (Cited page 48.)

[BMBF+19] Talha Bin Masood, Joseph Budin, Martin Falk, Guillaume Favelier, Christoph Garth, Charles Gueunet, Pierre Guillou, Lutz Hofmann, Petar Hristov, Adhitya Kamakshidasan, Christopher Kappe, Pavol Klacansky, Patrick Laurin, Joshua

Levine, Jonas Lukasczyk, Daisuke Sakurai, Maxime Soler, Peter Steneteg, Julien Tierny, Will Usher, Jules Vidal, and Michal Wozniak. An Overview of the Topology ToolKit. In *TopoInVis*, 2019. (Cited pages 45, 59, 100, and 119.)

[BW08] Sven Bachthaler and Daniel Weiskopf. Continuous Scatterplots. *IEEE Transactions on Visualization and Computer Graphics*, 14:1428–1435, 2008. (Cited pages 49 and 74.)

[BWT+11] P.T. Bremer, G. Weber, J. Tierny, V. Pascucci, M. Day, and J. Bell. Interactive exploration and analysis of large scale simulations using topology-based data segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 17(9):1307–1324, 2011. (Cited page 3.)

[CBW+12] Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, David Pugmire, Kathleen Biagas, Mark Miller, Cyrus Harrison, Gunther H. Weber, Hari Krishnan, Thomas Fogal, Allen Sanderson, Christoph Garth, E. Wes Bethel, David Camp, Oliver Rübel, Marc Durant, Jean M. Favre, and Paul Navrátil. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pages 357–372. October 2012. (Cited page 44.)

[CDC+99] William Carlson, Jesse Draper, David Culler, Katherine Yelick, Eugene Brooks, Karen Warren, and Lawrence Livermore. Introduction to UPC and language specification. 04 1999. (Cited page 40.)

[CDRFPB24] Riccardo Ceccaroni, Lorenzo Di Rocco, Umberto Ferraro Petrillo, and Pierpaolo Brutti. A Distributed Approach for Persistent Homology Computation on a Large Scale. *Journal of Symbolic Computation*, 80:25510–25532, 2024. (Cited page 99.)

[CGOS13] Frédéric Chazal, Leonidas J. Guibas, Steve Y. Oudot, and Primoz Skraba. Persistence-Based Clustering in Riemannian Manifolds. *Journal of the ACM*, 60(6), 2013. (Cited page 3.)

[Con04] IEEE Visualization 2004 Contest. Simulation of the Isabel hurricane. http://sciviscontest-staging.ieeevis.org/2004/data.html, 2004. (Cited page 139.)

[CRW22] Hamish Carr, Oliver Ruebel, and Gunther Weber. Distributed Hierarchical Contour Trees. In *IEEE Symposium on Large Data Analysis and Visualization*, pages 1–10, 2022. (Cited page 58.)

[CSA00] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. In *Symposium on Distributed Algorithms*, 2000. (Cited page 27.)

[CSGE07] David Cheng, Viral Shah, John Gilbert, and Alan Edelman. A Novel Parallel Sorting Algorithm for Contemporary Architectures. 01 2007. https://github.com/DIPHA/dipha/tree/master/externals/psort-1.0. (Cited page 105.)

[CSvdP04] Hamish A. Carr, Jack Snoeyink, and Michiel van de Panne. Simplifying Flexible Isosurfaces Using Local Geometric Measures. In *IEEE VIS*, 2004. (Cited page 3.)

[CWS+21] Hamish A. Carr, Gunther H. Weber, Christopher M. Sewell, Oliver Rübel, Patricia K. Fasel, and James P. Ahrens. Scalable Contour Tree Computation by Data Parallel Peak Pruning. *IEEE Transactions on Visualization and Computer Graphics*, 27:2437–2454, 2021. (Cited page 49.)

[DF21] Lisandro Dalcin and Yao-Lung L. Fang. mpi4py: Status Update After 12 Years of Development. *Computing in Science and Engineering*, 23(4):47–54, July 2021. (Cited page 35.)

[Dil07] Scott Dillard. A contour tree library. https://github.com/sedillard/libtourtre, 2007. (Cited page 48.)

[DN12a] Harish Doraiswamy and Vijay Natarajan. LibRG (Reeb Graph computation). https://vgl.csa.iisc.ac.in/softwares.php?pid=001, 2012. (Cited page 48.)

[DN12b] Harish Doraiswamy and Vijay Natarajan. Recon (Reeb Graph computation). https://github.com/harishd10/recon, 2012. (Cited page 48.)

[DT16] Alexandre Denis and François Trahay. MPI Overlap: Benchmark and Analysis. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 258–267, 2016. (Cited pages 39 and 119.)

[DTS+20] Harish Doraiswamy, Julien Tierny, Paulo J. S. Silva, Luis Gustavo Nonato, and Cláudio T. Silva. TopoMap: A 0-dimensional Homology Preserving Projection of High-Dimensional Data. *IEEE Transactions on Visualization and Computer Graphics*, 27:561–571, 2020. (Cited pages 3 and 49.)

[EH04] Herbert Edelsbrunner and John Harer. Jacobi Sets of Multiple Morse Functions. Cambridge Books Online, 2004. (Cited pages 49 and 74.)

[EH09] H. Edelsbrunner and J. Harer. *Computational Topology: An Introduction*. American Mathematical Society, 2009. (Cited pages 2, 10, 26, 28, and 99.)

[EHNP03] Herbert Edelsbrunner, John Harer, Vijay Natarajan, and Valerio Pascucci. Morse-Smale complexes for piecewise linear 3-manifolds. In *Symposium on Computational Geometry*, pages 361–370, 2003. (Cited page 28.)

[ELZ02] Herbert Edelsbrunner, David Letscher, and Afra Zomorodian. Topological Persistence and Simplification. *Discrete Computational Geometry*, 28(4):511–533, 2002. (Cited pages 2 and 97.)

[EM90] Herbert Edelsbrunner and Ernst P Mucke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990. (Cited page 14.)

[EWS+10] H Carter Edwards, Alan B Williams, Gregory D Sjaardema, David G Baur, and William K Cochran. SIERRA Toolkit Computational Mesh Conceptual Model. Technical report, Sandia National Laboratories, 2010. (Cited page 59.)

[FFST18] Guillaume Favelier, Noura Faraj, Brian Summa, and Julien Tierny. Persistence Atlas for Critical Point Variability in Ensembles. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):1152–1162, 2018. (Cited page 4.)

[FGT16] G. Favelier, C. Gueunet, and J. Tierny. Visualizing ensembles of viscous fingers. In *IEEE SciVis Contest*, 2016. (Cited page 3.)

[FKLM14] Brittany Fasy, Jisu Kim, Fabrizio Lecci, and Clément Maria. Introduction to the R package TDA. https://CRAN.R-project.org/package=TDA, 2014. (Cited page 48.)

[FL99] P. Frosini and C. Landi. Size theory as a topological tool for computer vision. *Pattern Recognition and Image Analysis*, 9, 1999. (Cited page 97.)

[For98] Robin Forman. A User's Guide to Discrete Morse Theory. *Séminaire Lotharingien de Combinatoire*, 1998. (Cited pages 19, 97, and 98.)

[Fou] The R Foundation. The R project. https://www.r-project.org/. (Cited page 48.)

[Fre42] H. Freudenthal. Simplizialzerlegungen von beschrankter Flachheit. *Annals of Mathematics*, 43:580–582, 1942. (Cited pages 47, 67, and 69.)

[GABCG+14] D. Guenther, R. Alvarez-Boto, J. Contreras-Garcia, J.-P. Piquemal, and J. Tierny. Characterizing Molecular Interactions in Chemical Systems. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2476–2485, 2014. (Cited page 3.)

[GBG+14] A. Gyulassy, P.T. Bremer, R. Grout, H. Kolla, J. Chen, and V. Pascucci. Stability of dissipation elements: A case study in combustion. *Computer Graphics Forum*, 33(3):51–60, 2014. (Cited page 3.)

[GBP19] Attila Gyulassy, Peer-Timo Bremer, and Valerio Pascucci. Shared-Memory Parallel Computation of Morse-Smale Complexes with Improved Accuracy. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):1183–1192, 2019. (Cited pages 20 and 49.)

[GFJT16] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Contour forests: Fast multi-threaded augmented contour trees. In *IEEE Symposium on Large Data Analysis and Visualization*, 2016. (Cited page 49.)

[GFJT19a] Charles Gueunet, Pierre Fortin, Julien Jomier, and Julien Tierny. Task-Based Augmented Contour Trees with Fibonacci Heaps. *IEEE Transactions on Parallel and Distributed Systems*, 30(8):1889–1905, 2019. (Cited pages 49, 50, and 74.)

[GFJT19b] Charles Gueunet, Pierre Fortin, Julien Jomier, and Julien Tierny. Task-based Augmented Reeb Graphs with Dynamic ST-Trees. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, 2019. (Cited pages 49 and 74.)

[GKL+16] A. Gyulassy, A. Knoll, K.C. Lau, B. Wang, P.T. 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, 2016. (Cited page 3.)

[GRWH12] David Günther, Jan Reininghaus, Hubert Wagner, and Ingrid Hotz. Efficient computation of 3d morse-smale complexes and persistent homology using discrete morse theory. *The Visual Computer*, 2012. (Cited page 98.)

[GST14] David Günther, Joseph Salmon, and Julien Tierny. Mandatory critical points of 2D uncertain scalar fields. *Computer Graphics Forum*, 2014. (Cited page 49.)

[Gue19] Charles Gueunet. *High Performance Level-Set Based Topological Data Analysis*. PhD thesis, Sorbonne Université, 2019. (Cited page 10.)

[GVT23] Pierre Guillou, Jules Vidal, and Julien Tierny. Discrete Morse Sandwich: Fast Computation of Persistence Diagrams for Scalar Data – An Algorithm and A Benchmark. *IEEE Transactions on Visualization and Computer Graphics*, 2023. (Cited pages 3, 5, 49, 74, 98, 100, 102, 111, and 120.)

[HKP+21] Xuan Huang, Pavol Klacansky, Steve Petruzza, Attila Gyulassy, Peer-Timo Bremer, and Valerio Pascucci. Distributed merge forest: a new fast and scalable approach for topological analysis at scale. In Huiyang Zhou, Jose Moreira, Frank Mueller, and Yoav Etsion, editors, *International Conference on Supercomputing*, pages 367–377. ACM, 2021. (Cited page 58.)

[HLH+16] C. Heine, H. Leitte, M. Hlawitschka, F. Iuricich, L. De Floriani, G. Scheuermann, H. Hagen, and C. Garth. A survey of topology-based methods in visualization. *Computer Graphics Forum*, 35(3):643–667, 2016. (Cited page 2.)

[HP18] Gregory Henselman-Petrusek. Eirene.jl package for homological algebra. https://github.com/Eetion/Eirene.jl, 2018. (Cited page 98.)

[HSSW11] Georg Hager, Gerald Schubert, Thomas Schoenemeyer, and Gerhard Wellein. Prospects for Truly Asynchronous Communication with Pure MPI and Hybrid MPI/OpenMP on Current Supercomputing Platforms. *The Cray User Group*, 2011. (Cited pages 39 and 119.)

[Inta] Intel Corporation. Open Source Code of oneAPI Threading Building Blocks (oneTBB) . https://github.com/uxlfoundation/oneTBB. (Cited page 34.)

[Intb] Intel Corporation. Specifications of oneAPI Threading Building Blocks (oneTBB). https://uxlfoundation.github.io/oneTBB/. (Cited page 34.)

[ISSS16] Daniel Ibanez, E. Seegyoung Seol, Cameron W. Smith, and Mark S. Shephard. PUMI: parallel unstructured mesh infrastructure. *ACM Transactions on Mathematical Software*, 42(3), 2016. (Cited page 59.)

[Iur21] Federico Iuricich. Persistence cycles for visual exploration of persistent homology. *IEEE Transactions on Visualization and Computer Graphics*, 28:4966–4979, 2021. https://github.com/IuricichF/PersistenceCycles. (Cited page 98.)

[Kel17] Bryn Keller. Python bindings for PHAT. https://pypi.org/project/phat/, 2017. (Cited page 48.)

[Khr25] The Khronos Group Inc. The OpenCL specification version 3.0.18, 2025. https://registry.khronos.org/OpenCL/. (Cited page 42.)

[Kit03] Kitware, Inc. *The Visualization Toolkit User's Guide*, 2003. (Cited page 44.)

[KK93] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on C++. *SIGPLAN Notices*, 28(10):91–108, October 1993. (Cited page 34.)

[Kla20] Pavol Klacansky. Open Scientific Visualization Data Sets. https://klacansky.com/open-scivis-datasets/, 2020. (Cited pages 80, 81, 120, 126, 128, 139, 140, and 141.)

[KRHH11] J. Kasten, J. Reininghaus, I. Hotz, and H.C. Hege. Two-dimensional time-dependent vortex regions based on the acceleration magnitude. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2080–2087, 2011. (Cited page 3.)

[KTCG17] Pavol Klacansky, Julien Tierny, Hamish A. Carr, and Zhao Geng. Fast and Exact Fiber Surfaces for Tetrahedral Meshes. *IEEE Transactions on Visualization and Computer Graphics*, 2017. (Cited pages 49 and 74.)

[Kuh60] H.W. Kuhn. Some combinatorial lemmas in topology. *IBM Journal of Research and Development*, 45:518–524, 1960. (Cited pages 47, 67, and 69.)

[KVT19] Max Kontak, Jules Vidal, and Julien Tierny. Statistical parameter selection for clustering persistence diagrams. In *2019 IEEE/ACM UrgentHPC@SC*, 2019. (Cited page 4.)

[LBCH22] Matthew Larsen, Eric Brugger, Hank Childs, and Cyrus Harrison. Ascent: A Flyweight In Situ Library for Exascale Simulations. In *In Situ Visualization For Computational Science*, pages 255 – 279. Mathematics and Visualization book series from Springer Publishing, Cham, Switzerland, May 2022. (Cited page 44.)

[LBM+06] D. E. Laney, P.T. Bremer, A. Mascarenhas, P. Miller, and V. Pascucci. Understanding the structure of the turbulent mixing layer in hydrodynamic instabilities. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1053–1060, 2006. (Cited page 3.)

[LGMT20] Jonas Lukasczyk, Christoph Garth, Ross Maciejewski, and Julien Tierny. Localized topological simplification of scalar

data. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):572–582, 2020. (Cited pages 50 and 74.)

[LGW+19] Jonas Lukasczyk, Christoph Garth, Gunther H. Weber, Tim Biedert, Ross Maciejewski, and Heike Leitte. Dynamic nested tracking graphs. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):249–258, 2019. (Cited page 4.)

[LI24] Guoxi Liu and Federico Iuricich. A task-parallel approach for localized topological data structures. *IEEE Transactions on Visualization and Computer Graphics*, 30:1271–1281, 2024. (Cited page 49.)

[LWG+24] Eve Le Guillou, Michael Will, Pierre Guillou, Jonas Lukasczyk, Pierre Fortin, Christoph Garth, and Julien Tierny. TTK is Getting MPI-Ready. *IEEE Transactions on Visualization and Computer Graphics*, 30(8):5875–5892, 2024. (Cited pages 54, 100, 105, and 120.)

[LWW+23] Jonas Lukasczyk, Michael Will, Florian Wetzels, Gunther H. Weber, and Christoph Garth. ExTreeM: Scalable Augmented Merge Tree Computation via Extremum Graphs. *IEEE Transactions on Visualization and Computer Graphics*, 30(1):1085–1094, 2023. (Cited pages 49 and 135.)

[MAB+24] Kenneth Moreland, Tushar M Athawale, Vicente Bolea, Mark Bolstad, Eric Brugger, Hank Childs, Axel Huebl, Li-Ta Lo, Berk Geveci, Nicole Marsaglia, Sujin Philip, David Pugmire, Silvio Rizzi, Zhe Wang, and Abhishek Yenpure. Visualization at exascale: Making it all work with vtk-m. *The International Journal of High Performance Computing Applications*, 38(5):508–526, 2024. (Cited pages 44 and 135.)

[MBGY14] Clément Maria, Jean-Daniel Boissonnat, Marc Glisse, and Mariette Yvinec. The gudhi library: Simplicial complexes and persistent homology. In *Mathematical Software*, 2014. https://github.com/GUDHI/. (Cited pages 48 and 98.)

[MDN12] Senthilnathan Maadasamy, Harish Doraiswamy, and Vijay Natarajan. A hybrid parallel algorithm for computing and tracking level set topology. In *International Conference on High Performance Computing*, pages 1–10, 2012. (Cited page 49.)

[Mes23] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 4.1. https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf, 2023. (Cited pages 35 and 76.)

[Mil63] J. Milnor. *Morse Theory*. Princeton University Press, 1963. (Cited pages 21 and 98.)

[MLT+23] Robin Maack, Jonas Lukasczyk, Julien Tierny, Hans Hagen, Ross Maciejewski, and Christoph Garth. Parallel Computation of Piecewise Linear Morse-Smale Segmentations. *IEEE Transactions on Visualization and Computer Graphics*, 30(4):1942–1955, 2023. (Cited pages 49 and 74.)

[MN12] Konstantin Mischaikow and Vidit Nanda. Morse theory for filtrations and efficient computation of persistent homology. *Discrete & Computational Geometry*, 50:330–353, 2012. (Cited pages 48, 97, and 98.)

[MN20] Dmitriy Morozov and Arnur Nigmetov. Brief announcement: Towards lockfree persistent homology. In *Symposium on Parallelism in Algorithms and Architectures*, 2020. (Cited pages 102 and 103.)

[Mor34] Marston Morse. The calculus of variations in the large. In *American Mathematical Society*, 1934. (Cited pages 21 and 98.)

[Mor17] Dmitriy Morozov. Dionysus2. http://www.mrzv.org/software/dionysus2, 2017. (Cited page 48.)

[MP16a] Dmitriy Morozov and Tom Peterka. Block-parallel data analysis with DIY2. In Markus Hadwiger, Ross Maciejewski, and Kenneth Moreland, editors, *IEEE Symposium on Large Data Analysis and Visualization*, 2016. (Cited page 41.)

[MP16b] Dmitriy Morozov and Tom Peterka. Efficient Delaunay Tessellation through K-D Tree Decomposition. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 728–738. IEEE, 2016. (Cited page 41.)

[MW13] Dmitriy Morozov and Gunther H. Weber. Distributed merge trees. In *ACM Principles and Practice of Parallel Programming*, 2013. (Cited page 58.)

[MW14] Dmitriy Morozov and Gunther H. Weber. Distributed contour trees. In *Topological Methods in Data Analysis and Visualization III, Theory, Algorithms, and Applications*. 2014. (Cited page 58.)

[MWR+16] Daniel Maljovec, Bei Wang, Paul Rosen, Andrea Alfonsi, Giovanni Pastore, Cristian Rabiti, and Valerio Pascucci. Topology-inspired partition-based sensitivity analysis and visualization of nuclear simulations. In *IEEE Pacific Visualization Symposium*, 2016. (Cited page 3.)

[Nan21] Vidit Nanda. Perseus, the persistent homology software. http://people.maths.ox.ac.uk/nanda/perseus/, 2021. (Cited page 48.)

[NM19] Arnur Nigmetov and Dmitriy Morozov. Local-global merge tree computation with local exchanges. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019. (Cited page 58.)

[NM20] Arnur Nigmetov and Dmitriy Morozov. Reeber: A library for shared- and distributed-memory parallel computation of merge trees, 2020. https://github.com/mrzv/reeber. (Cited pages 41 and 58.)

[NM24] Arnur Nigmetov and Dmitriy Morozov. Distributed Computation of Persistent Cohomology. Unpublished, https://arxiv.org/abs/2410.16553, 2024. (Cited page 99.)

[NR98] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998. (Cited page 40.)

[NVBB+22] Florent Nauleau, Fabien Vivodtzev, Thibault Bridel-Bertomeu, Heloise Beaugendre, and Julien Tierny. Topological Analysis of Ensembles of Hydrodynamic Turbulent Flows – An Experimental Study. In *IEEE Symposium on Large Data Analysis and Visualization*, 2022. (Cited page 3.)

[Nvi25] Nvidia Corporation. CUDA C Programming Guide, Release 12.8, 2025. (Cited page 42.)

[OGT19] Malgorzata Olejniczak, André Severo Pereira Gomes, and Julien Tierny. A Topological Data Analysis Perspective on

Non-Covalent Interactions in Relativistic Calculations. *International Journal of Quantum Chemistry*, 120(8):e26133, 2019. (Cited page 3.)

[Ope20] OpenMP Architecture Review Board. OpenMP application program interface version 5.1, 2020. https://www.openmp.org/specifications/. (Cited pages 31 and 38.)

[OT23] Malgorzata Olejniczak and Julien Tierny. Topological Data Analysis of Vortices in the Magnetically-Induced Current Density in LiH Molecule. *Physical Chemistry Chemical Physics*, 2023. (Cited page 3.)

[PC04] Valerio Pascucci and Kree Cole-McLaughlin. Parallel Computation of the Topology of Level Sets. *Algorithmica*, 2004. (Cited page 58.)

[Pon23] Mathieu Pont. *Analysis of Ensembles of Topological Descriptors*. PhD thesis, Sorbonne Université, 2023. (Cited page 10.)

[PT24] Mathieu Pont and Julien Tierny. Wasserstein Auto-Encoders of Merge Trees (and Persistence Diagrams). *IEEE Transactions on Visualization and Computer Graphics*, 2024. (Cited page 49.)

[PVDT22] Mathieu Pont, Jules Vidal, Julie Delon, and Julien Tierny. Wasserstein Distances, Geodesics and Barycenters of Merge Trees. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):291–301, 2022. (Cited pages 49 and 50.)

[PVT23] Mathieu Pont, Jules Vidal, and Julien Tierny. Principal Geodesic Analysis of Merge Trees (and Persistence Diagrams). *IEEE Transactions on Visualization and Computer Graphics*, 2023. (Cited pages 49 and 50.)

[QLIF24] Yuehui Qian, Guoxi Liu, Federico Iuricich, and Leila De Floriani. Efficient representation and analysis for a large tetrahedral mesh using apache spark. In *2024 IEEE Topological Data Analysis and Visualization (TopoInVis)*, pages 1–11, 2024. (Cited page 41.)

[Ree46] Georges Reeb. Sur les points singuliers d'une forme de Pfaff complètement intégrable ou d'une fonction numérique.

Comptes Rendus des séances de l'Académie des sciences, 222(847-849):76, 1946. (Cited page 28.)

[Rob99] Vanessa Robins. Toward computing homology from finite approximations. *Topology Proceedings*, 1999. (Cited page 97.)

[RWS11] Vanessa Robins, Peter John Wood, and Adrian P. Sheppard. Theory and Algorithms for Constructing Discrete Morse Complexes from Grayscale Digital Images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(8):1646–1658, 2011. (Cited pages 20, 21, 49, 97, 98, 105, and 127.)

[SCI23] SCIRun, 2023. https://github.com/SCIInstitute/SCIRun. (Cited page 44.)

[SDT24] Keanu Sisouk, Julie Delon, and Julien Tierny. Wasserstein Dictionaries of Persistence Diagrams. *IEEE Transactions on Visualization and Computer Graphics*, 2024. (Cited page 49.)

[SFLC18] Issam Said, Pierre Fortin, Jean-Luc Lamotte, and Henri Calandra. Leveraging the accelerated processing units for seismic imaging: A performance and power efficiency comparison against CPUs and GPUs. *International Journal of High Performance Computing Applications*, 32(6):819–837, 2018. (Cited page 39.)

[SM17] Dmitriy Smirnov and Dmitriy Morozov. Triplet Merge Trees. In *TopoInVis*, 2017. (Cited pages 49 and 111.)

[SMC07] Gurjeet Singh, Facundo Memoli, and Gunnar Carlsson. Topological Methods for the Analysis of High Dimensional Data Sets and 3D Object Recognition. In M. Botsch, R. Pajarola, B. Chen, and M. Zwicker, editors, *Eurographics Symposium on Point-Based Graphics*. The Eurographics Association, 2007. https://danifold.net/mapper/. (Cited page 48.)

[SN12] Nithin Shivashankar and Vijay Natarajan. Parallel Computation of 3D Morse-Smale Complexes. *Computer Graphics Forum*, 31(3):965–974, 2012. (Cited page 49.)

[SN17] Nithin Shivashankar and Vijay Natarajan. Efficient Software for Programmable Visual Analysis Using Morse-Smale

Complexes. In *Topological Methods in Data Analysis and Visualization IV*, pages 317–331, Cham, 2017. Springer International Publishing. https://vgl.csa.iisc.ac.in/mscomplex/software.html. (Cited page 48.)

[Sou11] Thierry Sousbie. The Persistent Cosmic Web and its Filamentary Structure: Theory and Implementations. *Royal Astronomical Society*, 414(1):384–403, 2011. https://thierry-sousbie.github.io/DisPerSE/. (Cited pages 3 and 48.)

[SPCT18a] 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*, 2018. (Cited page 4.)

[SPCT18b] Maxime Soler, Mélanie Plainchault, Bruno Conche, and Julien Tierny. Topologically controlled lossy compression. In *IEEE Pacific Visualization Symposium*, 2018. (Cited page 49.)

[SPD+19] Maxime Soler, Martin Petitfrere, Gilles Darche, Melanie Plainchault, Bruno Conche, and Julien Tierny. Ranking Viscous Finger Simulations to an Acquired Ground Truth with Topology-Aware Matchings. In *IEEE Symposium on Large Data Analysis and Visualization*, 2019. (Cited pages 3 and 4.)

[SPN+16] 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.)

[TC16] Julien Tierny and Hamish A. Carr. Jacobi Fiber Surfaces for Bivariate Reeb Space Computation. *IEEE Transactions on Visualization and Computer Graphics*, 2016. (Cited pages 49 and 74.)

[TFL+17] Julien Tierny, Guillaume Favelier, Joshua A. Levine, Charles Gueunet, and Michael Michaux. The Topology ToolKit. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):832–842, 2017. https://topology-tool-kit.github.io/.

(Cited pages 3, 4, 10, 45, 46, 47, 49, 59, 66, 69, 74, 100, and 119.)

[TGSP09] Julien Tierny, Attila Gyulassy, Eddie Simon, and Valerio Pascucci. Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1177–1184, 2009. (Cited page 48.)

[Tie18] Julien Tierny. *Topological Data Analysis for Scientific Visualization*. Springer, 2018. (Cited page 10.)

[top] top500.org. The top 500. https://top500.org/. (Cited page 30.)

[TP12] Julien Tierny and Valerio Pascucci. Generalized topological simplification of scalar fields on surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 2012. (Cited pages 50 and 74.)

[TSBO18] Christopher Tralie, Nathaniel Saul, and Rann Bar-On. Ripser.py: A lean persistent homology library for python. *The Journal of Open Source Software*, 3(29):925, Sep 2018. (Cited page 48.)

[TTK20] TTK Contributors. TTK Data. https://github.com/topology-tool-kit/ttk-data/tree/dev, 2020. (Cited pages 81, 120, and 141.)

[TTK22] TTK Contributors. TTK Online Example Database. https://topology-tool-kit.github.io/examples/, 2022. (Cited page 45.)

[TVJA14] Andrew Tausz, Mikael Vejdemo-Johansson, and Henry Adams. JavaPlex: A research software package for persistent (co)homology. In *ICMS*, 2014. http://appliedtopology.github.io/javaplex/. (Cited page 48.)

[VBT20] Jules Vidal, Joseph Budin, and Julien Tierny. Progressive Wasserstein Barycenters of Persistence Diagrams. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):151–161, 2020. (Cited page 4.)

[Vid21] Jules Vidal. *Progressivity in Topological Data Analysis*. PhD thesis, Sorbonne Université, 2021. (Cited page 10.)

[VKP+15] Karthikeyan Vaidyanathan, Dhiraj D. Kalamkar, Kiran Pamnany, Jeff R. Hammond, Pavan Balaji, Dipankar Das, Jongsoo Park, and Bàlint Joó. Improving concurrency and asynchrony in multithreaded MPI applications using software offloading. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015. (Cited page 39.)

[Wag23] Hubert Wagner. Slice, Simplify and Stitch: Topology-Preserving Simplification Scheme for Massive Voxel Data. In *Symposium on Computational Geometry*, 2023. (Cited pages 98 and 126.)

[WG21] Kilian Werner and Christoph Garth. Unordered Task-Parallel Augmented Merge Tree Construction. *IEEE Transactions on Visualization and Computer Graphics*, 2021. (Cited page 58.)

[WPTG23] Florain Wetzels, Mathieu Pont, Julien Tierny, and Christoph Garth. Merge Tree Geodesics and Barycenters with Path Mappings. *IEEE Transactions on Visualization and Computer Graphics*, 2023. (Cited page 49.)

[YBC+07] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, pages 24–32, New York, NY, USA, 2007. Association for Computing Machinery. (Cited page 40.)

[ZAB+19] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, Max Katz, Andrew Myers, Tan Nguyen, Andrew Nonaka, Michele Rosso, Samuel Williams, and Michael Zingale. AMReX: A Framework

for Block-Structured Adaptive Mesh Refinement. *Journal of Open Source Software*, 2019. (Cited pages 58 and 59.)

[ZC05] Afra Zomorodian and Gunnar Carlsson. Computing Persistent Homology. *Discrete and Computational Geometry*, 33(2):249–274, 2005. (Cited page 48.)

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association. (Cited page 40.)

Distributed Topological Data Analysis

Topological Data Analysis (TDA) tackles the complexity of large-scale data by capturing its structural characteristics in a concise encoding for analysis and visualization. As datasets grow, it becomes frequent for a single dataset to exceed the memory limit of one machine, making distributed-memory systems, with their much larger capacities, a necessary solution. However, adapting an algorithm for distributed-memory systems requires substantial changes to ensure correctness and performance. In particular, TDA algorithms face challenges in this context, as they rely on global data accesses and multiple traversals with minimal computation, a combination that often scales poorly in a distributed-memory context. Furthermore, existing distributed-memory implementations are mono-tailored for one particular topological representation which induces practical drawbacks. The Topology ToolKit (TTK) aims at providing a unified framework for TDA algorithms with a reusable and efficient data structure. However, TTK was up until now limited to shared-memory parallelism. In this thesis, we add distributed support to TTK using the *Message Passing Interface* (MPI). First, we adapt TTK's core data structure and add distributed-memory support to several existing algorithms, both to demonstrate the new features and highlight their performance. Performance tests showcase the efficiency of each algorithm as well as of the overall software infrastructure. Additionally, we apply a real-life topological analysis pipeline to two massive datasets to demonstrate our software's effectiveness at scale. Then, we focus our effort on a much more complex abstraction: the persistence diagram. Its robustness and reliability make it one of the most used topological representation. The Discrete Morse Sandwich (DMS) is currently the most efficient algorithm for computing the diagram on one node. Our new method, the Distributed Discrete Morse Sandwich (DDMS), builds upon DMS and introduces tailored step-specific modifications, resulting in a hybrid MPI+thread implementation. Performance tests demonstrate the gain of our approach over the original DMS method as well as *Dipha*, the reference method for persistence diagram computation in a distributed-memory context. Our method successfully computes persistence diagrams on datasets containing up to 6 billion vertices.

Analyse Topologique de Données Distribuée

L'Analyse Topologique de Données (TDA) vise à encoder de manière concise les caractéristiques structurelles de jeux de données afin de faciliter leur analyse et leur visualisation. Avec l'augmentation constante de la taille de ces données, qui dépassent de plus en plus souvent la capacité mémoire d'un ordinateur, le recours à des systèmes à mémoire distribuée, ou superordinateurs, offrant des ressources bien plus importantes, devient indispensable. Toutefois, adapter un algorithme aux superordinateurs requiert des modifications substantielles pour assurer à la fois l'exactitude des résultats et l'efficacité des calculs. Les algorithmes de TDA posent notamment des défis dans ce contexte, car ils nécessitent un accès global aux données et plusieurs parcours du jeu de données, avec peu de calculs, une combinaison qui passe généralement mal à l'échelle. De plus, les implémentations existantes pour la mémoire distribuée se concentre sur le calcul d'une seule représentation topologique. Le Topology ToolKit (TTK) vise à fournir un cadre unifié pour les algorithmes TDA avec une structure de données réutilisable et efficace. Cependant, il était jusqu'à présent limité au parallélisme à mémoire partagée. Dans cette thèse, nous ajoutons le support pour la mémoire distribuée à TTK grâce à MPI (*Message Passing Interface*). Dans un premier temps, nous adaptons la structure de données centrale de TTK et ajoutons le support du distribué à plusieurs algorithmes existants. Les tests de performance montrent l'efficacité de chaque algorithme ainsi que de l'infrastructure logicielle globale. De plus, nous appliquons un pipeline d'analyse topologique réel à deux jeux de données massifs afin de prouver la capacité de notre logiciel à traiter des jeux de données de grande taille. Ensuite, nous concentrons nos efforts sur une abstraction beaucoup plus complexe : le diagramme de persistance. Sa robustesse et sa fiabilité en font l'une des représentations topologiques les plus utilisées. Le Discrete Morse Sandwich (DMS) est actuellement l'algorithme le plus efficace pour calculer le diagramme sur un nœud. Notre nouvelle méthode, le Distributed Discrete Morse Sandwich (DDMS), repose sur DMS et introduit des modifications adaptées à chaque étape du calcul, aboutissant à une implémentation hybride MPI+thread. Des tests de performance montrent les gains de notre approche par rapport à la méthode DMS originale ainsi qu'à *Dipha*, la méthode de référence pour le calcul des diagrammes de persistance en distribué. Notre approche permet le calcul de diagrammes de persistance sur des jeux de données contenant jusqu'à 6 milliards de sommets.