TTK is Getting MPI-Ready

Eve Le Guillou, Michael Will, Pierre Guillou, Jonas Lukasczyk, Pierre Fortin, Christoph Garth, Julien Tierny

▶ **To cite this version:**

Eve Le Guillou, Michael Will, Pierre Guillou, Jonas Lukasczyk, Pierre Fortin, et al.. TTK is Getting MPI-Ready. IEEE Transactions on Visualization and Computer Graphics, inPress, pp.1-18. 10.1109/TVCG.2024.3390219 . hal-04552256

HAL Id: hal-04552256
https://hal.science/hal-04552256v1

Submitted on 19 Apr 2024

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

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

arXiv:2310.08339v2 [cs.DC] 15 Apr 2024

TTK is Getting MPI-Ready

E. Le Guillou, M. Will, P. Guillou, J. Lukasczyk, P. Fortin, C. Garth, J. Tierny

Abstract—This system paper documents the technical foundations for the extension of the *Topology ToolKit* (TTK) to distributed-memory parallelism with the *Message Passing Interface* (MPI). 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 paper 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 paper. 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). Finally, we provide a roadmap for the completion of TTK's MPI extension, along with generic recommendations for each algorithm communication category.

Index Terms—Topological data analysis, high-performance computing, distributed-memory algorithms.

1 Introduction

Modern datasets are constantly growing in size, due to the continuous improvements of acquisition technologies and computational systems. This growth induces finer level of details, in turn inducing more complex geometrical structures in the data. To apprehend this complexity, advanced techniques are required for the concise encoding of the core patterns in the data, to facilitate analysis and visualization.

Topological Data Analysis (TDA) [17] serves this purpose. It is based on robust, multi-scale algorithms [18], which capture a variety of structural features [33]. Examples of applications include combustion [10], [29], [41], material sciences [20], [31], [67], nuclear energy [47], fluid dynamics [37], [52], bioimaging [9], [12], data science [14], [15], quantum chemistry [7], [23], [55], [56] and astrophysics [64], [69].

However, 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 the combined memories of distributed systems.

The *Topology ToolKit* (TTK) [71] is an open-source library which implements a substantial collection of algorithms [8] for topological data analysis and visualization. In contrast to pre-existing, tailored, mono-algorithm implementations (see Sec. 1.1), TTK ***(1)*** supports multiple algorithms (Appendix A.3), ***(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* [74] for real-life examples). However, while most of its algorithms support shared-memory parallelism using multiple threads with OpenMP [57] (Appendix A.3), 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.

This system paper addresses this issue by documenting the technical foundations which are required for the extension of TTK to distributed-memory parallelism using multiple processes with the *Message Passing Interface* (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 (Sec. 3), we present the extension of TTK's internal triangulation data-structure (a central component of its performance and versatility) to the distributed setting (Sec. 4). We also document an interface between TTK and MPI (Sec. 5) enabling the consistent combination of multiple topological algorithms within a single, distributed pipeline.

Unlike previous work (Sec. 1.1), this paper 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 (Sec. 6), extending to the distributed setting a selection

- *E. Le Guillou is with the CNRS, Sorbonne Université and University of Lille. E-mail: eve.le_guillou@sorbonne- universite.fr*
- *M. Will, J. Lukasczyk and C. Garth are with RPTU Kaiserslautern-Landau. E-mails: {mswill,lukasczyk,garth}@rptu.de*
- *P. Guillou and J. Tierny are with the CNRS and Sorbonne Université. E-mails: {firstname.lastname}@sorbonne-universite.fr*
- *P. Fortin is with Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France. E-mail: pierre.fortin@univ-lille.fr*

Manuscript received May 21, 2021; revised May 11, 2021.

of topological algorithms. We also provide a taxonomy of TTK's topological algorithms (Sec. 6.1), depending on their communication needs and provide examples of hybrid MPI+thread parallelizations for each category (Sec. 6.3), with detailed performance analyses (Sec. 7.1). We illustrate the new distributed capabilities of TTK with an example of advanced analysis pipeline (Sec. 6.4), combining multiple algorithms, run on a dataset of 120 billion vertices distributed on 64 nodes (Sec. 7.2) of 24 cores each. Finally, we provide a roadmap for the completion of TTK's MPI extension, with generic recommendations for each algorithm communication category (Sec. 8). This work has been integrated in the main source code of TTK and is available in open-source.

1.1 Related work

Concepts from computational topology [17] have been investigated and extended by the visualization community [33] over the last two decades. Popular topological representations include the persistence diagram, the Reeb graphs and its variants, or the Morse-Smale complex [17].

To improve the time efficiency of the algorithms computing the above representations, 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 [6], [27], others focused on the merge and contour trees [1], [13], [24], [25], [46], [66] or the Reeb graph [26], while several other approaches have been proposed for the Morse-Smale complex [30], [62], [63]. Recently, a localized approach based on shared-memory parallelism has been introduced for the on-the-fly triangulation connectivity computation [42]. 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.

Fewer approaches have been documented for the computation of topological data representations in a distributed-memory environment. First, distributed-memory computers are much less accessible in practice than parallel shared-memory architectures, which have become ubiquitous in recent years (workstations, laptops, etc.). Second, the algorithmic advances in terms of parallelism described in the shared-memory approaches do not translate directly 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. 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 [34], [50], [51], 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 [5] as well as the merge and contour trees [11], [34], [50], [51], [53], [58], [75]. 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 [5] 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 [53], [54] focuses on merge tree computation. It is tailored for regular grids (with optional support of adaptive mesh refinement via AMReX [76]) and its data structure only models vertex adjacency relations (which is the only traversal functionality required for merge tree computation). In contrast, our work provides a data-structure (Sec. 4) which is *(1)* versatile (it supports both triangulated domains and regular grids), *(2)* compact and time-efficient (with an adaptive footprint for triangulated domains and no memory overhead for regular grids), *(3)* flexible (it supports a rich set of traversals, Sec. 4.1, required to support the entire TTK algorithm collection, Appendix A.3), and *(4)* 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. Several general purpose software frameworks have been documented. For instance, DIY [49] is a block-parallel library that facilitates the parallelization of pre-existing algorithms. Specifically, DIY enables developers to write a single implementation, which can be used for multiple runtime configurations (out-of-core, shared-memory or distributed-memory parallelism). As such, DIY is a general-purpose software component, sitting right on top of low-level parallel environments (e.g. MPI [48] or C++ threads). In contrast to our work, it does not provide any specific mechanism for topological algorithms. It does not provide a distributed data-structure for simplicial complexes (which our work contributes, Sec. 4) or the convenience functionalities needed by arbitrary simplicial complexes for consistently combining multiple topological algorithms into a distributed pipeline, such as the one supported by VTK (our work also contributes such pipeline functionalities, Sec. 5). Moreover, DIY's design philosophy eases parallelization at the cost of limiting the benefits of combining distributed and shared-memory parallelisms. For instance, to our un-

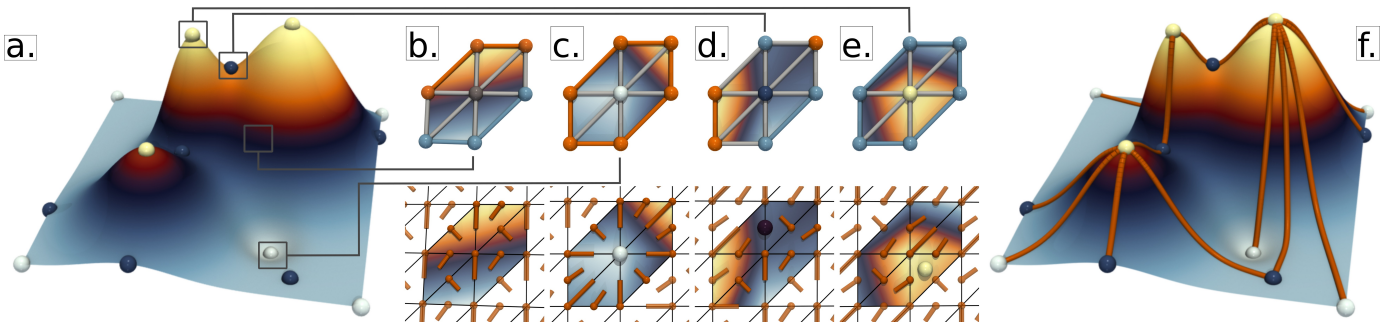

Fig. 1: Topological objects considered in this paper on a toy example (elevation f on a terrain $\mathcal{M}$, *(a)*). The vertices of $\mathcal{M}$ can be classified based on their star into regular vertices (*(b)*, top: PL setting, bottom: DMT setting), local minima *(c)*, saddle points *(d)* or local maxima *(e)*. Integral lines (orange curves, *(f)*) are curves which are tangential to the gradient of f.

derstanding, workload within a single data block cannot be shared with DIY among multiple threads (as done in contrast in our work, Sec. 6.2). To balance workload among threads with DIY, more (smaller) blocks would be required. This can result in non-optimal load balancing when the workload is not evenly distributed spatially, and can be detrimental to algorithms spanning multiple blocks (e.g. integral lines, Sec. 7.1.1).

To support topological algorithms, a data structure must be available to efficiently traverse the input dataset, with possibly advanced traversal queries. TTK [8], [71] 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 [19], [36], [76] (with a focus on simulation driven remeshing), we consider in this work the distribution of TTK's triangulation data structure (Sec. 4), 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. Sec. 3).

1.2 Contributions

This system paper makes the following new contributions.

1) *An efficient, distributed triangulation data structure* (Sec. 4): We introduce an extension of TTK's triangulation data structure for the support of distributed datasets.
2) *A software infrastructure for distributed topological pipelines* (Sec. 5): 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* (Sec. 6): 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.
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).

2 Background

This section describes our formal setting and formalizes a few topological data representations, used later in the paper when discussing examples (Sec. 6). All these descriptions are given in a *non-distributed* context. The formalization of our distributed model is documented in Sec. 3. We refer the reader to reference textbooks [17] for a comprehensive introduction to computational topology.

2.1 Input data

The input is a piecewise linear (PL) scalar field $f : \mathcal{M} \rightarrow \mathbb{R}$ defined on a d-dimensional simplicial complex, with $d \leqslant 3$ in our applications (Fig. 1*(a)*). The set of i-simplices of $\mathcal{M}$ is denoted $\mathcal{M}^i$. The *star* $St(\sigma)$ of a simplex σ is the set of simplices of $\mathcal{M}$ which contain σ as a face. The *link* $Lk(\sigma)$ is the set of faces of the simplices of $St(\sigma)$ which do not intersect σ. The input field f is provided on the vertices of $\mathcal{M}$ and is interpolated on the simplices of higher dimension. f is assumed to be injective on the vertices, which is achieved by substituting the f value of a vertex by its position in the vertex order (by increasing f values).

2.2 Critical points

The sub-level set $f^{-1}_{-\infty}(w)$ of an isovalue $w \in \mathbb{R}$ is defined as $f^{-1}_{-\infty}(w) = \{p \in \mathcal{M} \mid f(p) < w\}$. It can be interpreted as a subset of the data, below the isovalue w. As w continuously increases, the topology of $f^{-1}_{-\infty}(w)$ changes at specific vertices of $\mathcal{M}$, called the *critical points* of f. Let $Lk^-(v)$ be the *lower link* of the vertex v: $Lk^-(v) = \{\sigma \in Lk(v) \mid \forall u \in \sigma : f(u) < f(v)\}$ (blue edges and vertices in Fig. 1*(b-e)*, top). The *upper link* of v is defined symmetrically: $Lk^+(v) = \{\sigma \in Lk(v) \mid \forall u \in \sigma : f(u) > f(v)\}$ (orange

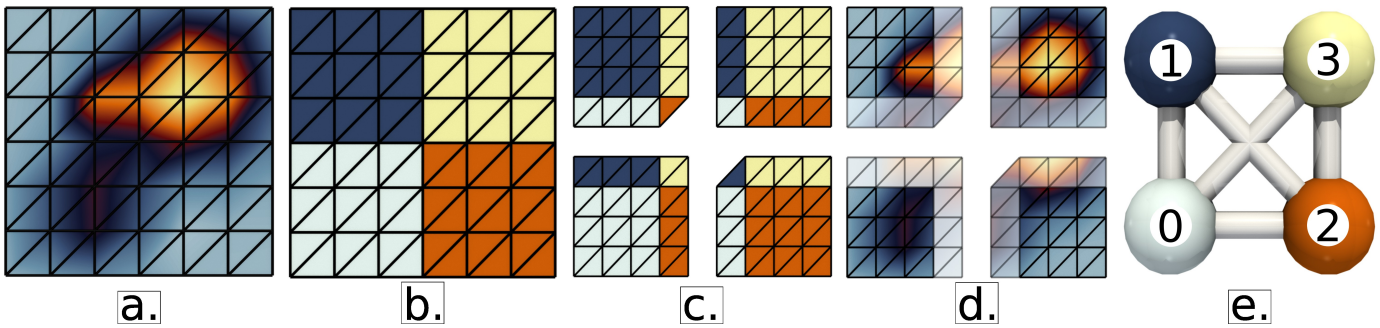

Fig. 2: The input data *(a)* is assumed to be loaded in the memory of n_p independent processes in the form of n_p disjoint *blocks* of data (*(b)*, one color per block, $n_p = 4$ in this example). A layer of *ghost* simplices (*(c)*, coming from adjacent blocks, matching colors) is added to each block. 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)*.

edges and vertices in Fig. 1*(b-e)*, top). 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 [4]. A critical vertex v can be classified by its *index* $\mathcal{I}(v)$, which is 0 for minima (Fig. 1*(c)*), 1 for 1-saddles (Fig. 1*(d)*), $(d-1)$ for $(d-1)$-saddles and d for maxima (Fig. 1*(e)*). Vertices for which the number of connected components of $Lk^-(v)$ or $Lk^+(v)$ are greater than 2 are called *degenerate saddles*. Prior to this work, this critical point classification was implemented in TTK with shared-memory parallelism with OpenMP (see Appendix A.3), as each vertex classification is independent.

2.3 Integral lines

Integral lines are curves on $\mathcal{M}$ which locally describe the gradient of f (orange curves in Fig. 1*(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*. 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). Prior to this work, this computation was implemented sequentially in TTK (see Appendix A.3), i.e. given a list of input seeds, each integral line was computed sequentially one after the other.

2.4 Discrete gradient

In recent years, an alternative emerged to the PL formalism of critical points described above (Sec. 2.2), namely Discrete Morse Theory (DMT) [22]. This formalism implicitly resolves several challenging configurations (such as degenerate saddles on manifold domains), which has been particularly useful for the development of robust algorithms in the context of Morse-Smale complex computation [30], [62]. We also consider in this work this alternative representation to critical points, as it nicely exemplifies a large set of the traversal features supported by TTK's triangulation (Sec. 4).

A *discrete vector* (small orange arrows, Fig. 1*(b-e)*, bottom) 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 $\{\sigma_i < \sigma_{i+1}\}$. σ_{i+1} is usually referred to as the *head* of the vector (represented with a small orange cylinder in Fig. 1*(b-e)*, bottom), while σ_i is its *tail* (represented with a small orange sphere in Fig. 1*(b-e)*, bottom). 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. 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. A simplex σ_i which is involved in no discrete vector $\mathcal{V}$ is called a *critical simplex*. A *v-path* is a sequence of discrete vectors $\big\{\{\sigma_i^0 < \sigma_{i+1}^0\}, \ldots, \{\sigma_i^k < \sigma_{i+1}^k\}\big\}$, 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 $0 < j < k$. A *discrete gradient field* is a discrete vector field such that all its possible *v-paths* are loop-free. 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. [62], 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 (Sec. 2.2). Prior to this work, this computation was implemented in TTK with shared-memory parallelism with OpenMP (Appendix A.3), as each lower star can be processed independently.

3 Distributed Model

We now formalize our distributed model, which will eventually be used as a blueprint to port the algorithms described above (Sec. 2) to distributed computations (Sec. 6).

3.1 Input distribution formalization

3.1.1 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 (Fig. 2*(a-b)*). Specifically, each process $i \in \{0, \ldots, n_p - 1\}$ is associated with a local block $f_i : \mathcal{M}_i \to \mathbb{R}$, such that:

- $\mathcal{M}_i \subset \mathcal{M}$: each block $\mathcal{M}_i$ 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 $\sigma \in \mathcal{M}_i$ which is not exclusively owned by the process i is called a *ghost simplex* (Sec. 3.1.2).
- $\cup_{\mathcal{M}_i} = \mathcal{M}$: the union of the blocks is equal to the input.

3.1.2 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 Fig. 2, (d)). We note $\mathcal{M}_i'$ the d-dimensional simplicial complex obtained by considering a layer of ghost simplices, i.e. by adding to $\mathcal{M}_i$ the d-simplices of $\mathcal{M}$ which share a face with a d-simplex of $\mathcal{M}_i$, 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 $\mathcal{M}_i$ to form the *ghosted block* $\mathcal{M}_i'$ are *ghost simplices* (Fig. 2*(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, Sec. 2.2). 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 Sec. 4.2.1).

The blocks are also positioned in relation to one another. Processes i and j will be considered adjacent (Fig. 2*(e)*) if $\mathcal{M}_i'$ contains d-simplices that are exclusively owned by j and if $\mathcal{M}_j'$ contains d-simplices that are exclusively owned by i.

3.1.3 Global simplex identifiers

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

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, Sec. 2.2) 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, Sec. 6.3.5, 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. Sec. 4), 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 ($n_p(P_2)$).

3.1.4 Simplex-to-process maps

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

3.2 Output distribution formalization

Topological algorithms typically consume an input (possibly complex), to produce a (usually) simpler output (such as the topological representations described in Sec. 2). Moreover, multiple topological algorithms can be combined sequentially to form an analysis pipeline. For instance, a first algorithm A_1 may compute integral lines (Sec. 6.3.5) for a first field f, while a second algorithm A_2 may extract the critical vertices (Sec. 2.2) 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 (Sec. 3.1) 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 complexity of this output, specialized manipulation algorithms (handling communication between processes) may need to be later developed to exploit them appropriately in a post-process.

3.3 Implementation specification

We now review the building blocks which are necessary to support the distributed model specified in Secs. 3.1 and 3.2.

The pipeline combining the different topological algorithms can be encoded in the form of a Python script (c.f. contribution 5, Sec. 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 [2]. 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 (Sec. 5.2).

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. 4.2 and 4.3.

The input PL scalar field f is required to be injective on the vertices (c.f. Sec. 2.1). 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 $\mathcal{M}'_i$), the process i 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.

Sec. 4 documents the extension of TTK's triangulation data structure to support our model of distributed input and output (Secs. 3.1 and 3.2).

Additional procedures easing the combination of multiple algorithms into a single pipeline (adjacency graph computation, ghost data exchange) are documented in Sec. 5.2.

4 Distributed Triangulation

This section describes the distributed extension of TTK's triangulation data structure, later used by each topological algorithm. In the following, we assume that the input block is loaded in the memory of the local process i and ghosted (i.e. we consider the ghosted block $\mathcal{M}'_i$, Sec. 3.1.2). Moreover, we consider that, for each process i, a list of *neighbor processes* is available (Fig. 2*(e)*).

4.1 Initial design

For completeness, we briefly summarize the pre-existing implementation of TTK's triangulation data structure [71]. 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 enumerate 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 [71], 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 (i.e. an adaptive, precomputation stage, described in Appendix A.2). For regular grids, periodic (along all dimensions) or not, the traversal queries can be computed on-the-fly given the regular structure of the Freudenthal triangulation of the grid [32], [35].

4.2 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.

4.2.1 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, Sec. 3.1, its inverse, ϕ_0^{-1}). This step is not optional and is triggered automatically. For each ghosted block $\mathcal{M}'_i$, the number n_{v_i} of *non-ghost* vertices that the block exclusively owns is computed (Fig. 3*(a)*). Next, an MPI prefix sum is performed to determine the offset that each block i 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. $d' \in \{1, \ldots, d-1\}$, Fig. 3*(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 i 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 i will generate an identifier for σ only if i is the lowest simplex-to-process identifier among the exclusive owners of the d-simplices in $St(\sigma)$ (Sec. 3.1). 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 $\mathcal{M}'_i$ 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, Sec. 6).

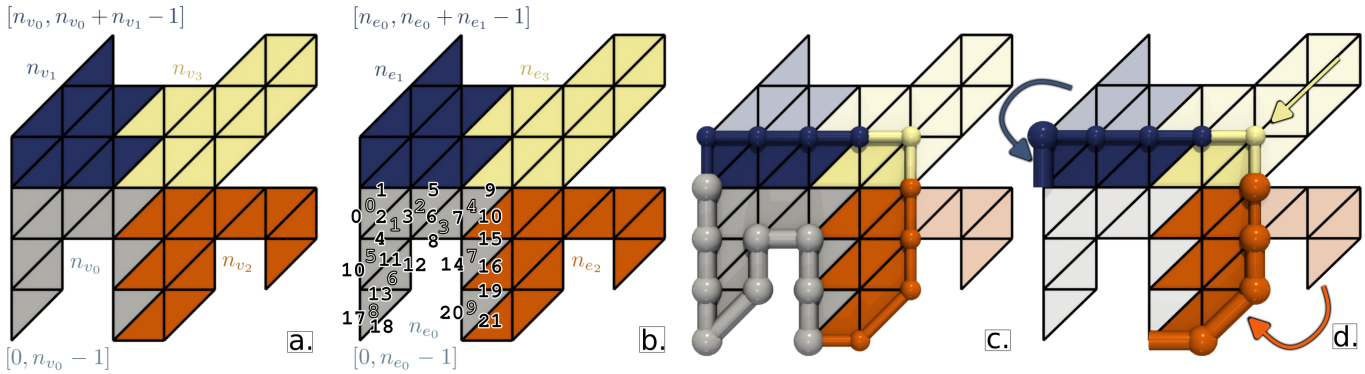

Fig. 3: Preconditioning of our distributed explicit triangulation. *(a)* Each process i enumerates its number n_{v_i} of exclusively owned vertices and d-simplices. Next, an MPI prefix sum provides a local offset for each process to generate global identifiers. *(b)* For each process i, simplices of intermediate dimensions (edges (n_{e_i}), 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).

Once all the processes have established their vertex global identifiers, each process i 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 $\mathcal{M}'_i$, 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, Sec. 6).

Each process i identifies the boundary vertices of its *ghosted block* $\mathcal{M}'_i$ (See Fig. 3*(c)*), with exactly the same procedure as the one used in the non-distributed setting [71]. 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 i and its neighbors enables the retrieval of the ghost simplices of $\mathcal{M}'_i$ which are also on the boundary (Fig. 3*(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.

4.2.2 *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 $\mathcal{M}'_i$ by an algorithm instantiated on the process i 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 Sec. 3.1.3).

4.3 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 [32], [35] of the input grid.

4.3.1 *Distributed implicit preconditioning*

In implicit mode, the preconditioning of the triangulation identifies the position of the local ghosted grid $\mathcal{M}'_i$ within the global grid $\mathcal{M}$, as detailed in Fig. 4. 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).

4.3.2 *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 $\mathcal{M}'_i$ by an algorithm instantiated on the process i is performed identically to the non-distributed setting, with local simplex identifiers.

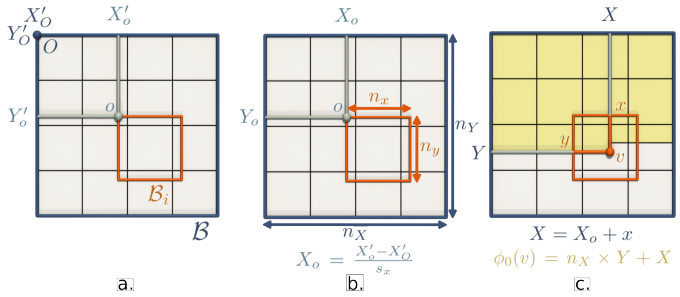

Fig. 4: Preconditioning of our distributed implicit triangulation. *(a)* Each process i computes (with shared-memory parallelism) the bounding box $\mathcal{B}_i$ of its ghosted block $\mathcal{M}'_i$. The vertex o, respectively O, is the origin of $\mathcal{M}'_i$, respectively $\mathcal{M}$, with (X'_o, 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 $\mathcal{B}_i$. *(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 $\mathcal{M}'_i$ 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.

Similarly to the explicit case (Sec. 4.2.2), the calling algorithm must now translate input (and output) *global* simplex identifiers into *local* ones (i.e. with the maps from Sec. 3.1.3).

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 Sec. 4.2.1) 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 Fig. 4, 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 $\mathcal{M}'_i$ (Fig. 4*(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 [71] (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 [71]).

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

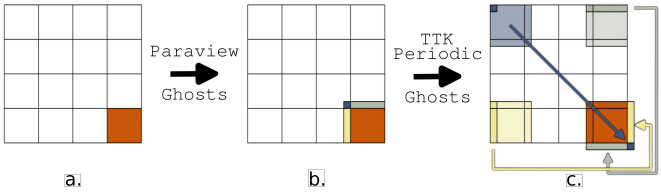

Fig. 5: Preconditioning of our distributed periodic implicit triangulation. This triangulation type is handled similarly to the implicit case, but additional ghost simplices need to be computed. Given a data block $\mathcal{M}_i$ (*(a)*, orange), ParaView generates a first layer of ghost d-simplices (*(b)*, blue, grey, yellow). If $\mathcal{M}_i$ 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)*.

4.4 Distributed implicit periodic triangulation

Periodic grids (with periodicity in all dimensions) are supported via implicit Freudenthal triangulation [32], [35] 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 Fig. 5. Specifically, each process i checks if its block $\mathcal{M}'_i$ 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 $\mathcal{M}'_i$ 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 Fig. 5*(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 Sec. 4.3, 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).

5 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, Sec. 4) in order to complete the support of the distributed model specified in Sec. 3. We refer the reader to Appendix A.1 for further details on the integration of TTK with VTK and ParaView.

5.1 Overview

The input data is provided in the form of a distributed dataset (see Sec. 3.1) loaded from a filesystem (e.g. *PVTI* file format) or provided in-situ (e.g. with *Catalyst*). As shown in Fig. 6 (and detailed in Appendix A.1), ParaView's execution flow enters TTK via the function `ProcessRequest`, which triggers TTK's preconditioning, including the *Distributed*

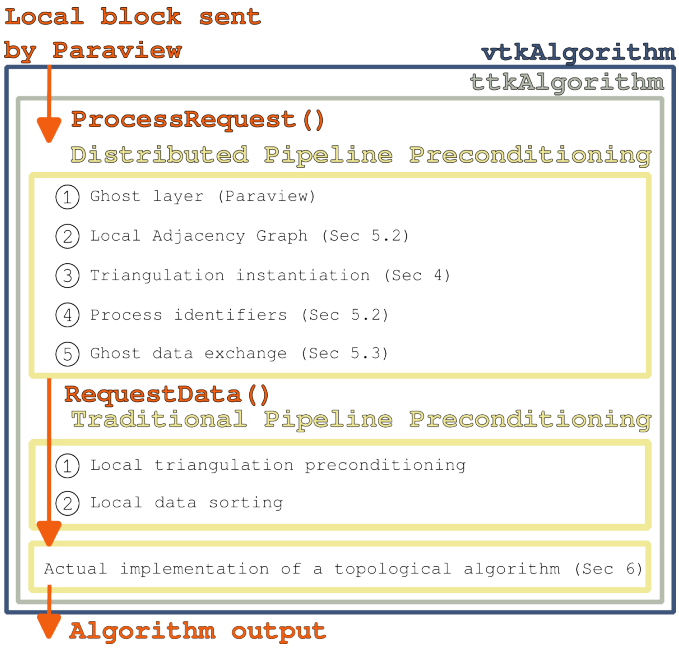

Fig. 6: Overview of the overall pipeline upon the the delivery of a data block $\mathcal{M}_i$ by ParaView (top). A step of pipeline preconditioning specialized for the distributed setting (top yellow frame) is automatically triggered before calling the actual implementation of the topological algorithm. 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.

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 adjacency graph (i.e. connecting the data block to its neighbors) is constructed. This step (described in Sec. 5.2) is omitted if a valid LAG is already present.

***(3)* Triangulation instantiation:** this step instantiates a new TTK triangulation data structure (Sec. 4). 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 Sec. 3.1). This step (described in Sec. 5.2) is omitted if valid simplex-to-process maps are already present.

***(5)* Ghost data exchange:** this step computes for each neighbor process j the list of vertices or cells exclusively owned by it, and which are ghosts in the process i. This step (described in Sec. 5.2) 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, Fig. 6).

5.2 Infrastructure details

This section describes the implementation of the pipeline preconditioning mentioned in the above overview (Sec. 5.1), specifically, the routines which are not directly related to the distributed triangulation (which has been covered in Sec. 4).

Local adjacency graph (LAG) initialization: Given a ghosted block $\mathcal{M}'_i$, the goal of this step is to store a list of processes, which are responsible for the blocks adjacent to $\mathcal{M}'_i$ (Fig. 2*(e)*). First, each process i computes the bounding box $\mathcal{B}_i$ of its ghosted block $\mathcal{M}'_i$. Next, all processes exchange their bounding boxes. Finally, each process i can initialize a list of neighbor processes by collecting the processes whose bounding box intersects with $\mathcal{B}_i$. 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 Sec. 3.1, 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, Sec. 6.3.5).

Each vertex $v \in \mathcal{M}'_i$ is classified by ParaView as ghost or non-ghost. For each non-ghost vertex v, we set its simplex-to-process identifier to i. Then, the *ghost global identifier list* is computed (it contains the global identifiers of all the ghost vertices of $\mathcal{M}'_i$). Next, this list is sent to each process j marked as being adjacent in the LAG (previous paragraph). Then, the process j will return its identifier (j) and the subset of the *ghost global identifier list*, corresponding to non-ghost vertices in $\mathcal{M}'_j$. Finally, the process i will set the simplex-to-process identifier of v to j, for each vertex v returned by j. 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 Sec. 3.1. Following the generation of the simplex-to-process maps, the LAG is updated, by only considering the block i and j as neighbors if i contains ghost vertices which are exclusively owned by j 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 $\mathcal{M}_i$. 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 $\mathcal{M}'_i$. For instance, when considering smoothing (Sec. 6.3.4), at each iteration, the process i 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 i will, for each neighbor process j, send the global identifiers of the simplices which are ghost for i and owned by j (using their simplex-to-process maps). This is computed once, in an optional preconditioning step (step 5, Sec. 5.1). This list of ghost vertex identifiers is cached in j and used at runtime, when necessary, to send to i the updated values (exchange

data buffers are updated with shared-memory parallelism). A similar procedure is available for d-simplices.

6 EXAMPLES

Secs. 4 and 5 documented the implementation of the distributed model specified in Sec. 3. In this section, we now describe how to make use of this model to extend topological algorithms to the distributed setting. Specifically, we will mostly focus on the algorithms described in Sec. 2.

6.1 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 Sec. 2.2, discrete gradient computation Sec. 2.4, Jacobi set extraction [16], Fiber surface computation [40] 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 (Sec. 6.3), or continuous scatter plots [3].

***(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 (Sec. 2.3), persistence diagrams [27], merge and contour trees [25], path compression [45], Reeb graphs [26], Morse-Smale complexes [71], Rips complexes, topological simplification [43], [72], Reeb spaces [70], etc.

6.2 Hybrid MPI+thread strategy

In this section, we present general aspects regarding the combination of multi-process parallelism (with MPI; running on both distributed- and shared-memory architectures) and multi-thread parallelism (restricted to shared-memory architectures), that are used within the examples (Sec. 6.3).

Current compute clusters are based on multiple nodes, each node including multi-core processors. For performance reasons, one then runs one execution flow per core following either a *pure MPI* strategy, i.e. with one MPI process per core, or a *MPI+thread* strategy, i.e. with one MPI process per node (or per processor) and multiple threads within each MPI process. The latter can improve performance thanks to fewer MPI communications (due to fewer MPI processes), to a (better) dynamic load balancing among threads within each MPI process, and to a multi-core speedup for computations specifically performed by the MPI process 0 (e.g. Sec. 4.2.1). The overall memory footprint is also lower with the latter, since using fewer MPI processes implies fewer ghost simplices and less data duplication.

Regarding the MPI+thread strategy and the port examples described in Sec. 6.3, we rely in TTK on the `MPI_THREAD_FUNNELED` thread support level in MPI [48]. 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 (Appendix A.3).

Besides, when using the MPI+thread implementations, one can choose to run one MPI process per node or one MPI process per processor, hence e.g. two MPI processes on a node with two processors. The former leads to fewer MPI processes in total, and enables one to balance the compute load among all the cores of the node. The latter can avoid performance issues due to NUMA (non-uniform memory access) effects which occur when a thread running on a given processor accesses data on the memory local to the other processor. Options specific to the MPI implementation enable the user to choose one of the two possibilities. The threads are also bound to the CPU cores using the OpenMP thread affinity features [57].

6.3 Distributed algorithm examples

We now illustrate the taxonomy of Sec. 6.1 by describing the distributed-memory parallelization of algorithms belonging to each of the categories, while exploiting the distributed model we introduced (Sec. 3).

6.3.1 NC: Scalar Field Critical Points

This algorithm processes each vertex v of the domain independently and performs the classification presented in Sec. 2.2. 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 (Fig. 7*(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 (Sec. 3.2), each output critical point is associated with its *global* vertex identifier (instead of its local one).

6.3.2 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 (Sec. 2.4). Similarly to the previous case,

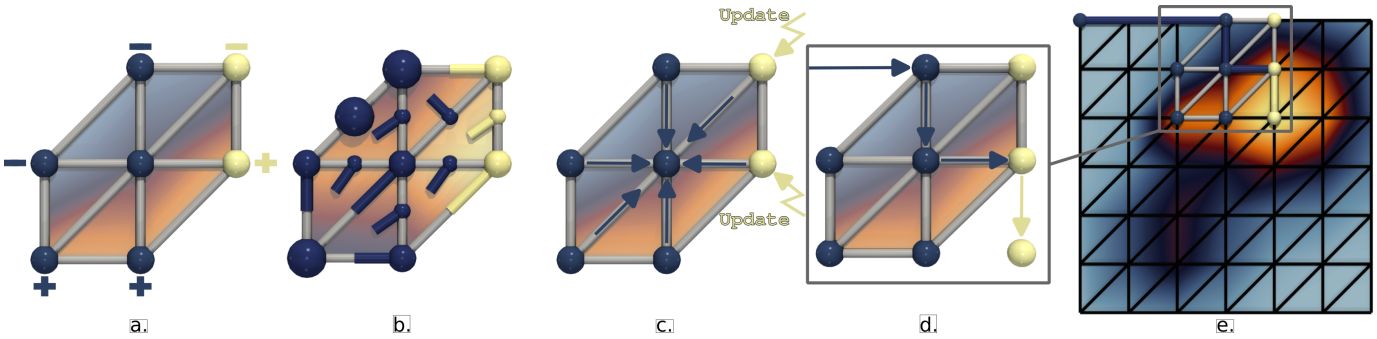

Fig. 7: Examples of topological algorithm modifications for the support of distributed memory computation. *(a)* Scalar Field Critical Points (NC): Critical points are generated similarly to the sequential mode. Upper and lower links ($+$ and $-$ signs in the figure) of non-ghost vertices on the boundary of $\mathcal{M}_i$ 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 $\mathcal{M}_i$, 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 $\mathcal{M}_i$. Then either the integral line reaches its final vertex within $\mathcal{M}_i$, *completing* the computation, or the integral line reaches a vertex outside of $\mathcal{M}_i$ (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.

this algorithm only requires local data and only produces local outputs, without needing communications (hence its NC classification) (Fig. 7 *(b)*). The port of this embarrassingly parallel algorithm requires two modifications. First, only the vertices which are exclusively owned by the current process (Sec. 3.1) 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).

6.3.3 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).

6.3.4 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 (Fig. 7*(c)*). Thus, at the end of each iteration, each process i 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 Sec. 5.2 (hence the DIC classification for this algorithm).

6.3.5 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 2.3): each process i will compute the integral lines whose seeds lie within its block $\mathcal{M}_i$ (each seed is processed independently via shared-memory parallelism with OpenMP). Moreover, the process i 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 $\mathcal{M}_i$, *completing* the computation, or the integral line reaches a ghost vertex owned by another process j 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 j (Fig. 7*(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).

6.4 Integrated pipeline

In this section, we describe an integrated pipeline that produces a real-life use case combining all the the port examples

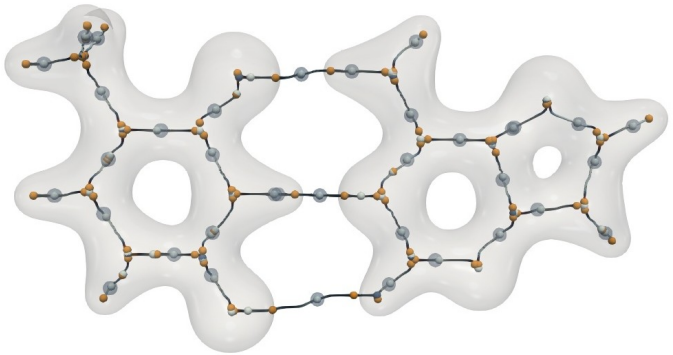

Fig. 8: Output of the integrated pipeline on the AT dataset, a three-dimensional regular grid of the electronic density (and its gradient magnitude) in the Adenine Thymine complex (AT). The extracted integral lines capture the covalent and hydrogen bonds within the molecule complex. The transparent spheres 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 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.

presented in Sec.6.3. All of the algorithms, their order as well as their input are described in Table 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 (Fig. 8). 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 (Fig. 8).

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 [39]). 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).

7 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 (using the timing method described in Appendix B). 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 representation of performance in a more complicated pipeline. They are therefore excluded from the individual benchmarks (Sec. 7.1) but included in the study of the global, integrated pipeline Sec. 7.2 (which is timed using ParaView's internal timer).

The benchmark is performed on five different datasets: *Wavelet* (3D wavelets on a cube), *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), *Isabel* (magnitude of the wind velocity in a simulation of the hurricane Isabel that hit the east coast of the USA in 2003), *Random* (random field on a cube) and *Backpack* (density in the CT-scan of a backpack filled with items). The datasets all originate from publicly available repositories [39], [73].

7.1 Distributed algorithms performance

This section evaluates the practical performance of the extension to the distributed setting (Sec. 6.3) of the algorithms presented in Sec. 2, by considering strong and weak scaling.

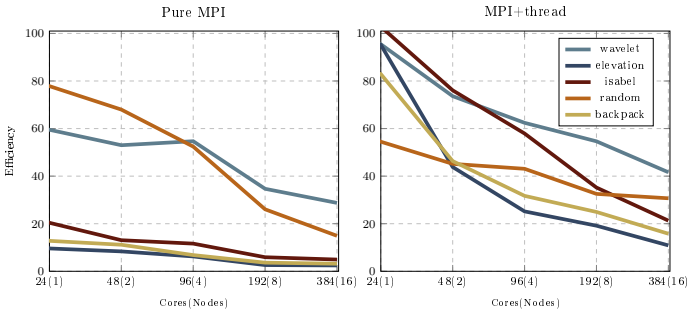

Fig. 9: Strong scaling efficiencies for the Integral line computation algorithm with $500,000$ seeds, randomly distributed on all processes, using the pure MPI strategy (left) and the MPI+thread one (right) with 1 MPI process and 24 threads per node. The MPI+thread strategy is significantly more efficient than the pure MPI one.

7.1.1 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, where the size of the input data is constant (each dataset is resampled to 512^3 via trilinear interpolation) while the number of available cores increases. The speedup s_p for p cores is defined as $s_p = \frac{t_1}{t_p}$, with t_p being the execution time for p cores. Then, we define the *strong scaling efficiency* for p cores as $\frac{s_p}{p} \times 100$. Appendix C shows the same results as presented in Fig. 9 (right) and Fig. 10, but in terms of execution time instead of parallel efficiency.

We first compare the pure MPI and the MPI+thread strategies (Sec. 6.2). 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 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 Fig. 9, 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.

We have tried to improve the parallel efficiencies of the integral line algorithm, by dedicating a thread to MPI communications. 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. However this design based on a communication thread adds a significant amount of complexity to the implementation (due to the required thread synchronizations), and did not improve the parallel efficicency since the main performance bottleneck is the load imbalance among processes. As a result, we do not rely on this communication thread design in our distributed integral lines implementation.

The performance results for the other distributed algorithms can be found in Fig. 10. 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, Sec. 6.3.5), 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.

7.1.2 Weak scaling

Next, we evaluate, the ability of our framework to process datasets of increasing sizes. For this, we conduct a weak scaling analysis, where the workload increases proportionally to the number of available cores, starting at 24 cores (1 node). 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. Then, we define the *weak scaling*

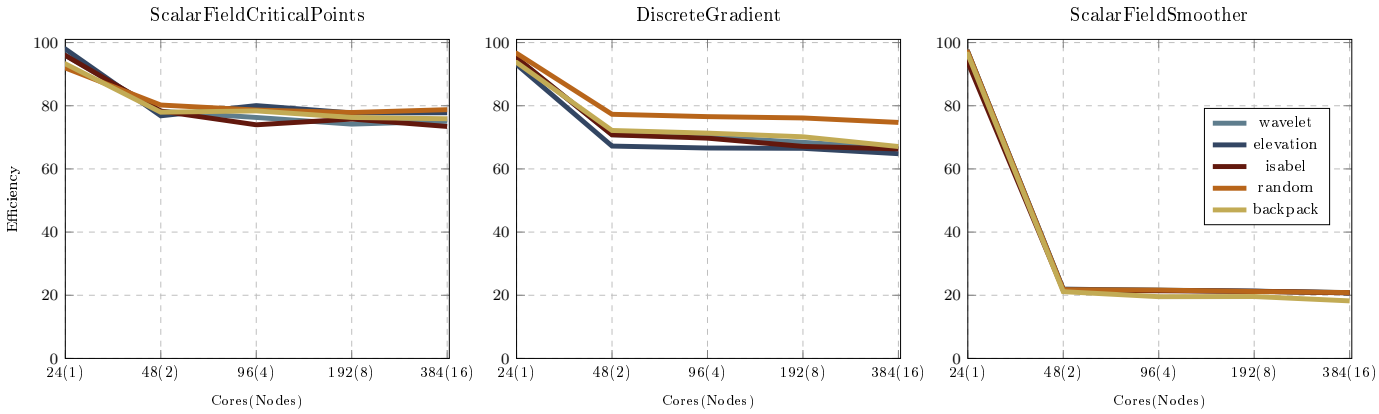

Fig. 10: Strong scaling efficiencies for various algorithms (MPI+thread: 1 MPI process and 24 threads per node).

efficiency for p cores as $\frac{t_1}{t_p} \times 100$, with t_1 and t_p being the execution times on 1 and p nodes. Appendix C shows the same results as presented in Fig. 11, but in terms of execution time instead of efficiency.

As shown in Fig. 11, 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 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.

7.2 Integrated pipeline performance

We now present experimental results for the integrated pipeline (Sec. 6.4), which exemplifies a real-life use case

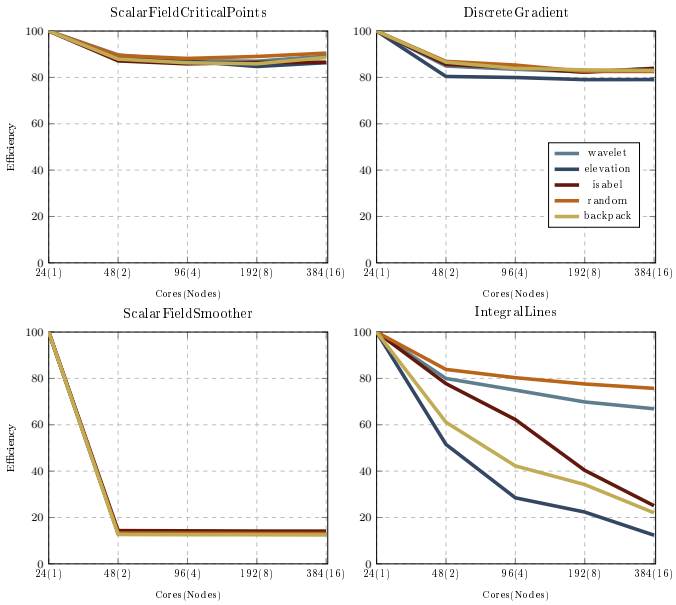

Fig. 11: Weak scaling efficiencies for various algorithms (MPI+thread: 1 MPI process and 24 threads per node)

combining all of the port examples described in Sec. 6.3, on datasets which were too large (8.5 and 120 billion vertices, Sec. 6.4) to be handled by TTK prior to this work.

The results for the integrated pipeline are twofold: an output image (Fig. 8 and Fig. 13) and the time profiling of the pipeline (Fig. 12). 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 defined in Sec. 5.2 (for a fine-grain account of the execution time within an algorithm and its preconditioning).

7.2.1 The Adenine Thymine complex (AT) dataset

For the experiments of Figs. 8 and 12 (left), the selected resampling dimensions for the input regular grid are 2048^3, a choice explained in Sec. 6.4. The overall computation takes 241.2 seconds. Preconditioning is triggered once, before executing the first TTK algorithm. The longest preconditioning

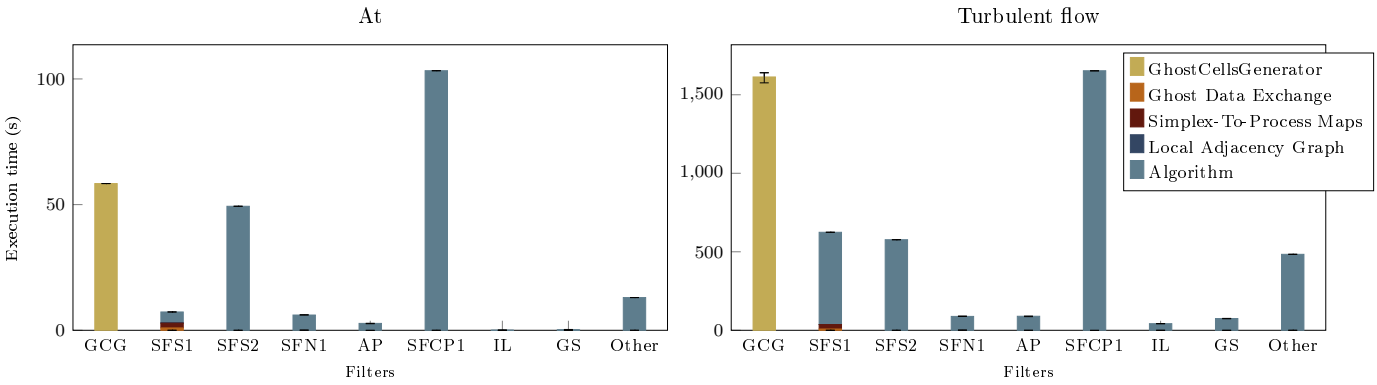

Fig. 12: Time profiling for the integrated pipeline for the AT dataset resampled to roughly 8.5 billion vertices (left) and the *Turbulent Channel Flow* dataset (right) of 120 billion vertices. The execution was conducted using 64 nodes of 24 cores each (1536 cores in total) on MeSU-beta. Each bar corresponds 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 Tab. 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%).

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.

7.2.2 The Turbulent Channel Flow dataset

The computation shown in Fig. 12 (right) was performed on the complete dataset (120 billion vertices, single-precision, Sec. 6.4). The overall computation takes 5257.5 seconds.

The execution time of this pipeline includes the algorithms listed in Tab. 1. Note that the rendering time is not included in the time profiling reported in Fig. 12 (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 Fig. 13 was performed on only a quarter of the dataset. The pipeline profiled in Fig. 12, 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).

7.3 Limitations

Sec. 4 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 address 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.

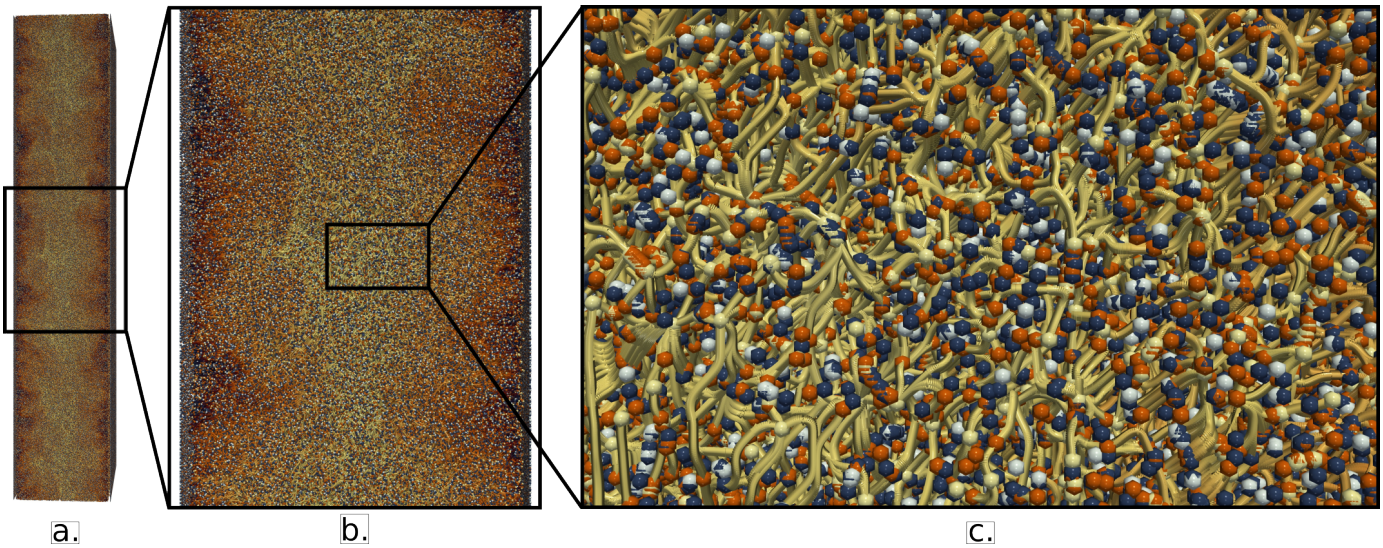

Fig. 13: Output of the integrated pipeline on the *Turbulent Channel Flow* dataset (120 billion vertices), a three-dimensional regular grid with two scalar fields, the pressure of the fluid and its gradient magnitude. The pipeline was executed up to the Geometry Smoother algorithm. The spheres correspond 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 Sec. 7.2.2), while Fig. 12 was produced on the full dataset.

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. In the future, we will investigate extensions of our distributed triangulation to support these representations natively, without pre-process.

8 Conclusion and Roadmap

In this paper, 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 the Topology ToolKit (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. Our framework is available in TTK 1.2.0, enabling others to reproduce our results or extend TTK's distributed capabilities. Also, as TTK is now officially integrated in ParaView, this distributed version of TTK will be available to a wide audience in the next release of ParaView.

The next step consists in adding distributed-memory support to all of TTK's topological algorithms. The challenge here depends on the algorithm class (see Sec. 6.1). The port of NC and DIC algorithms (such as ContinuousScatterPlot, ManifoldCheck, DistanceField, JacobiSet or FiberSurface) is relatively straightforward. For DIC algorithms, the initial step entails identifying the data to be exchanged, the processes involved in the exchange, and the appropriate timing for 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). This could for example be done during a hackathon. For 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, on which we will focus in future work.

Acknowledgments

This work is partially supported by the European Commission grant ERC-2019-COG *"TORI"* (ref. 863464, https://erc-tori.github.io/).

References

[1] A. Acharya and V. Natarajan. A parallel and memory efficient algorithm for constructing the contour tree. In *IEEE PV*, 2015.
[2] J. Ahrens, B. Geveci, and C. Law. ParaView: An End-User Tool for Large-Data Visualization. *The Visualization Handbook*, 2005.
[3] S. Bachthaler and D. Weiskopf. Continuous Scatterplots. *IEEE TVCG*, 2008.
[4] T. F. Banchoff. Critical points and curvature for embedded polyhedral surfaces. *The American Mathematical Monthly*, 1970.
[5] U. Bauer, M. Kerber, and J. Reininghaus. Distributed computation of persistent homology. In *AEE*, 2014.

[6] U. Bauer, M. Kerber, J. Reininghaus, and H. Wagner. Phat - persistent homology algorithms toolbox. *J. of S. C.*, 2017.

[7] H. Bhatia, A. G. Gyulassy, V. Lordi, J. E. Pask, V. Pascucci, and P.-T. Bremer. Topoms: Comprehensive topological exploration for molecular and condensed-matter systems. *JCC*, 2018.

[8] T. Bin Masood, J. Budin, M. Falk, G. Favelier, C. Garth, C. Gueunet, P. Guillou, L. Hofmann, P. Hristov, A. Kamakshidasan, C. Kappe, P. Klacansky, P. Laurin, J. Levine, J. Lukasczyk, D. Sakurai, M. Soler, P. Steneteg, J. Tierny, W. Usher, J. Vidal, and M. Wozniak. An Overview of the Topology ToolKit. In *TopoInVis*, 2019.

[9] A. Bock, H. Doraiswamy, A. Summers, and C. T. Silva. TopoAngler: Interactive Topology-Based Extraction of Fishes. *IEEE TVCG*, 2018.

[10] P. Bremer, G. Weber, J. Tierny, V. Pascucci, M. Day, and J. Bell. Interactive exploration and analysis of large scale simulations using topology-based data segmentation. *IEEE TVCG*, 2011.

[11] H. Carr, O. Ruebel, and G. Weber. Distributed Hierarchical Contour Trees. In *IEEE LDAV*, 2022.

[12] H. A. Carr, J. Snoeyink, and M. van de Panne. Simplifying Flexible Isosurfaces Using Local Geometric Measures. In *IEEE VIS*, 2004.

[13] H. A. Carr, G. H. Weber, C. M. Sewell, O. Rübel, P. K. Fasel, and J. P. Ahrens. Scalable Contour Tree Computation by Data Parallel Peak Pruning. *IEEE TVCG*, 2021.

[14] F. Chazal, L. J. Guibas, S. Y. Oudot, and P. Skraba. Persistence-based clustering in riemannian manifolds. *J. ACM*, 2013.

[15] H. Doraiswamy, J. Tierny, P. J. S. Silva, L. G. Nonato, and C. T. Silva. Topomap: A 0-dimensional homology preserving projection of high-dimensional data. *IEEE TVCG*, 2020.

[16] H. Edelsbrunner and J. Harer. Jacobi Sets of Multiple Morse Functions. Cambridge Books Online, 2004.

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

[18] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. *DCG*, 2002.

[19] H. C. Edwards, A. B. Williams, G. D. Sjaardema, D. G. Baur, and W. K. Cochran. SIERRA Toolkit Computational Mesh Conceptual Model. Technical report, Sandia National Laboratories, 2010.

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

[21] Florain Wetzels and Mathieu Pont and Julien Tierny and Christoph Garth. Merge Tree Geodesics and Barycenters with Path Mappings. *IEEE TVCG*, 2023.

[22] R. Forman. A User's Guide to Discrete Morse Theory. *AM*, 1998.

[23] D. Guenther, R. Alvarez-Boto, J. Contreras-Garcia, J.-P. Piquemal, and J. Tierny. Characterizing molecular interactions in chemical systems. *IEEE TVCG*, 2014.

[24] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Contour forests: Fast multi-threaded augmented contour trees. In *IEEE LDAV*, 2016.

[25] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Task-Based Augmented Contour Trees with Fibonacci Heaps. *IEEE TPDS*, 2019.

[26] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Task-based Augmented Reeb Graphs with Dynamic ST-Trees. In *EGPGV*, 2019.

[27] P. Guillou, J. Vidal, and J. Tierny. Discrete Morse Sandwich: Fast Computation of Persistence Diagrams for Scalar Data – An Algorithm and A Benchmark. *IEEE TVCG*, 2023.

[28] D. Günther, J. Salmon, and J. Tierny. Mandatory critical points of 2D uncertain scalar fields. *CGF*, 2014.

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

[30] A. Gyulassy, P. Bremer, and V. Pascucci. Shared-Memory Parallel Computation of Morse-Smale Complexes with Improved Accuracy. *IEEE TVCG*, 2018.

[31] A. Gyulassy, A. Knoll, K. Lau, B. Wang, P. Bremer, M. Papka, L. A. Curtiss, and V. Pascucci. Interstitial and interlayer ion diffusion geometry extraction in graphitic nanosphere battery materials. *IEEE TVCG*, 2015.

[32] H. Freudenthal. Simplizialzerlegungen von beschrankter Flachheit. *Annals of Mathematics*, 43:580–582, 1942.

[33] 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. *CGF*, 2016.

[34] X. Huang, P. Klacansky, S. Petruzza, A. Gyulassy, P. Bremer, and V. Pascucci. Distributed merge forest: a new fast and scalable approach for topological analysis at scale. In *ICS*, 2021.

[35] H.W. Kuhn. Some combinatorial lemmas in topology. *IBM Journal of Research and Development*, 45:518–524, 1960.

[36] D. Ibanez, E. S. Seol, C. W. Smith, and M. S. Shephard. PUMI: parallel unstructured mesh infrastructure. *ACM TMS*, 2016.

[37] J. Kasten, J. Reininghaus, I. Hotz, and H. Hege. Two-dimensional time-dependent vortex regions based on the acceleration magnitude. *IEEE TVCG*, 2011.

[38] Kitware, Inc. *The Visualization Toolkit User's Guide*, January 2003.

[39] P. Klacansky. Open Scientific Visualization Data Sets. https://klacansky.com/open-scivis-datasets/, 2020.

[40] P. Klacansky, J. Tierny, H. A. Carr, and Z. Geng. Fast and Exact Fiber Surfaces for Tetrahedral Meshes. *IEEE TVCG*, 2017.

[41] D. E. Laney, P. Bremer, A. Mascarenhas, P. Miller, and V. Pascucci. Understanding the structure of the turbulent mixing layer in hydrodynamic instabilities. *IEEE TVCG*, 2006.

[42] G. Liu and F. Iuricich. A task-parallel approach for localized topological data structures. *IEEE TVCG*, 2024.

[43] J. Lukasczyk, C. Garth, R. Maciejewski, and J. Tierny. Localized topological simplification of scalar data. *IEEE TVCG*, 2020.

[44] J. Lukasczyk, M. Will, F. Wetzels, G. H. Weber, and C. Garth. ExTreeM: Scalable Augmented Merge Tree Computation via Extremum Graphs. *IEEE TVCG*, 2023.

[45] R. Maack, J. Lukasczyk, J. Tierny, H. Hagen, R. Maciejewski, and C. Garth. Parallel Computation of Piecewise Linear Morse-Smale Segmentations. *IEEE TVCG*, 2023.

[46] S. Maadasamy, H. Doraiswamy, and V. Natarajan. A hybrid parallel algorithm for computing and tracking level set topology. In *Proc. of HiPC*, 2012.

[47] D. Maljovec, B. Wang, P. Rosen, A. Alfonsi, G. Pastore, C. Rabiti, and V. Pascucci. Topology-inspired partition-based sensitivity analysis and visualization of nuclear simulations. In *IEEE PV*, 2016.

[48] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 4.0. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[49] D. Morozov and T. Peterka. Block-parallel data analysis with DIY2. In M. Hadwiger, R. Maciejewski, and K. Moreland, eds., *IEEE LDAV*, 2016.

[50] D. Morozov and G. H. Weber. Distributed merge trees. In *ACM PPoPP*, 2013.

[51] D. Morozov and G. H. Weber. Distributed contour trees. In *TopoInVis*. 2014.

[52] F. Nauleau, F. Vivodtzev, T. Bridel-Bertomeu, H. Beaugendre, and J. Tierny. Topological Analysis of Ensembles of Hydrodynamic Turbulent Flows – An Experimental Study. In *IEEE LDAV*, 2022.

[53] A. Nigmetov and D. Morozov. Local-global merge tree computation with local exchanges. In *SC*, 2019.

[54] A. Nigmetov and D. Morozov. Reeber: A library for shared- and distributed-memory parallel computation of merge trees, 2020. https://github.com/mrzv/reeber.

[55] M. Olejniczak, A. S. P. Gomes, and J. Tierny. A Topological Data Analysis Perspective on Non-Covalent Interactions in Relativistic Calculations. *International Journal of Quantum Chemistry*, 2019.

[56] M. Olejniczak and J. Tierny. Topological Data Analysis of Vortices in the Magnetically-Induced Current Density in LiH Molecule. *Physical Chemistry Chemical Physics*, 2023.

[57] OpenMP Architecture Review Board. OpenMP application program interface version 5.1.

[58] V. Pascucci and K. Cole-McLaughlin. Parallel Computation of the Topology of Level Sets. *Algorithmica*, 2004.

[59] M. Pont and J. Tierny. Wasserstein Auto-Encoders of Merge Trees (and Persistence Diagrams). *IEEE TVCG*, 2024.

[60] M. Pont, J. Vidal, J. Delon, and J. Tierny. Wasserstein Distances, Geodesics and Barycenters of Merge Trees. *IEEE TVCG*, 2021.

[61] M. Pont, J. Vidal, and J. Tierny. Principal Geodesic Analysis of Merge Trees (and Persistence Diagrams). *IEEE TVCG*, 2023.

[62] V. Robins, P. J. Wood, and A. P. Sheppard. Theory and Algorithms for Constructing Discrete Morse Complexes from Grayscale Digital Images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 2011.

[63] N. Shivashankar and V. Natarajan. Parallel Computation of 3D Morse-Smale Complexes. *CGF*, 2012.

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

[65] K. Sisouk, J. Delon, and J. Tierny. Wasserstein Dictionaries of Persistence Diagrams. *IEEE TVCG*, 2024.

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

[67] M. Soler, M. Petitfrere, G. Darche, M. Plainchault, B. Conche, and J. Tierny. Ranking Viscous Finger Simulations to an Acquired Ground Truth with Topology-Aware Matchings. In *IEEE LDAV*, 2019.
[68] M. Soler, M. Plainchault, B. Conche, and J. Tierny. Topologically controlled lossy compression. In *IEEE PV*, 2018.
[69] T. Sousbie. The persistent cosmic web and its filamentary structure: Theory and implementations. *RAS*, 2011.
[70] J. Tierny and H. A. Carr. Jacobi Fiber Surfaces for Bivariate Reeb Space Computation. *IEEE TVCG*, 2016.
[71] J. Tierny, G. Favelier, J. A. Levine, C. Gueunet, and M. Michaux. The Topology ToolKit. *IEEE TVCG*, 2017. https://topology-tool-kit.github.io/.
[72] J. Tierny and V. Pascucci. Generalized topological simplification of scalar fields on surfaces. *IEEE TVCG*, 2012.
[73] TTK Contributors. TTK Data. https://github.com/topology-tool-kit/ttk-data/tree/dev, 2020.
[74] TTK Contributors. TTK Online Example Database. https://topology-tool-kit.github.io/examples/, 2022.
[75] K. Werner and C. Garth. Unordered Task-Parallel Augmented Merge Tree Construction. *IEEE TVCG*, 2021.
[76] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, M. Katz, A. Myers, T. Nguyen, A. Nonaka, M. Rosso, S. Williams, and M. Zingale. AMReX: A Framework for Block-Structured Adaptive Mesh Refinement. *Journal of Open Source Software*, 2019.

Eve Le Guillou is a Ph.D. student at Sorbonne Université. She received a M.S. degree in 2020 in Computer Science from Cranfield University as well as an engineering degree in 2021 from École Centrale de Lille. She is an active contributor to the Topology ToolKit (TTK), an open source library for topological data analysis. Her notable contributions to TTK include the port to MPI of TTK's data structure and of several algorithms.

Michael Will is a Ph.D. student at RPTU Kaiserslautern-Landau. He received a M.S. degree in Computer Science from the Technical University of Kaiserslautern in 2021. He is an active contributor to TTK's MPI section. His research interest include topological data analysis, especially using high performance computing on large-scale data.

Pierre Guillou is a research engineer at Sorbonne Université. After graduating from MINES ParisTech, a top French engineering school in 2013, he received his Ph.D., also from MINES ParisTech, in 2016. His Ph.D. work revolved around parallel image processing algorithms for embedded accelerators. Since 2019, he has been an active contributor to TTK and the author of many modules created for the VESTEC and TORI projects.

Jonas Lukasczyk received his Ph.D. degree from the Visual Information Analysis Group, Technische Universitat Kaiserslautern, Germany, where he also studied applied computer science and mathematics. His recent work focuses on topology-based characterization of features and their evolution in large-scale simulations.

Pierre Fortin received his Ph.D. degree in Computer Science from the University of Bordeaux in 2006, and his Habilitation degree (HDR) from Sorbonne University in 2018. He has been Assistant Professor from 2007 to 2020 at Sorbonne University, and he is Associate Professor at University of Lille since 2020. His research interests include parallel algorithmic and high performance scientific computing.

Christoph Garth received the PhD degree in computer science from Technische Universität (TU) Kaiserslautern in 2007. After four years as a postdoctoral researcher with the University of California, Davis, he rejoined TU Kaiserslautern where he is currently a full professor of computer science. His research interests include large scale data analysis and visualization, in situ visualization, topology-based methods, and interdisciplinary applications of visualization.

Julien Tierny received the Ph.D. degree in Computer Science from the University of Lille in 2008. He is a CNRS research director at Sorbonne University. Prior to his CNRS tenure, he held a Fulbright fellowship and was a post-doctoral researcher at the University of Utah. His research expertise lies in topological methods for data analysis and visualization. He is the founder and lead developer of the Topology ToolKit (TTK), an open source library for topological data analysis.

APPENDIX A
BACKGROUND ON THE TOPOLOGY TOOLKIT (TTK)

This section provides some background regarding the software library *the Topology ToolKit* (TTK), and it details its pre-existing support (i.e. prior to this work) for triangulation traversal and parallelization.

A.1 Scope and interfaces

TTK is an open-source software library for topological data analysis and visualization, written in C++. While TTK can be used directly via its raw, low-level C++ interface, TTK also provides an interface of higher-level, for the *Visualization ToolKit* (VTK [38], another open-source C++ library, dedicated to data visualization and analysis, but with a broader scope than TTK). In particular, as described in its companion paper [71], 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 inheriting 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 Fig. 6). The re-implementation of this function in the `ttkAlgorithm` class will trigger all the necessary preconditioning before calling the actual topological algorithm (see Sec. 6 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 the open-source application *ParaView* [2], which is a de-facto standard for the visualization and analysis of large-scale data. 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).

A.2 Pre-existing triangulation

Internally, each topological algorithm implemented in TTK is exploiting TTK's generic data-structure for the efficient traversal of simplicial complexes (Sec. 4.1), a central aspect in most topological algorithms. 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 preconditioning 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 [32], [35] 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 [71]. 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.

A.3 Pre-existing parallel algorithms (shared-memory)

TTK implements a substantial collection of topological algorithms for scalar data, bivariate data, ensemble data or even point cloud data. For more details, we refer the reader to an overview paper [8] as well as to *TTK's Online Example Database* [74] (a database of real-life data analysis use cases, implementing advanced topological analysis pipelines, combining multiple algorithms).

Prior to this work, only shared-memory parallelism was implemented in TTK, using multiple threads with OpenMP [57]. Then parallel computations could only be carried out on a single computer. Specifically, some of the topological objects introduced in the manuscript (the critical points and the discrete gradient) could be computed in parallel (the integral line extraction implementation however was sequential). TTK provides shared-memory parallel computations for various objects, including the following, non-exhaustive list: continuous scatterplots [3], data or geometry smoothing, dimensionality reduction [15], fiber surfaces [40], Jacobi sets [16], mandatory critical points [28], marching tetrahedra, merge and contour trees [25], [44], merge tree distances and encoding [21], [59], [60], [61], Morse-Smale complexes [71], path compression [45], persistence diagrams [27], persistence diagram encoding [65], Reeb graphs [26], Reeb spaces [70], Rips complexes, scalar field normalizer, topological compression [68], topological simplification [43], [72].

A.4 Contributions

In this work, we document the infrastructure evolution that is required for TTK to support distributed parallelism, via MPI, as documented in Secs. 3 (distributed data model), 4 (distributed mesh data-structure) and 5 (distributed pipeline management). We also provide examples of topological algorithms (Sec. 6) which we extended to support distributed computations with MPI, on top of their pre-existing shared-memory parallelization with OpenMP. In particular, note that Sec. 6.3.5 describes a shared-memory parallelization with OpenMP of integral line computation which is novel in this work (this computation was sequential in the pre-existing version of TTK). Finally, we provide in Sec. 8 a roadmap for the extension to the distributed computation of the remaining algorithms of TTK.

APPENDIX B
FINE-SCALE TIME PERFORMANCE MEASUREMENTS

When timing the execution of a specific distributed algorithm, simply measuring the execution time on one process may not represent the execution time of the whole algorithm, as the local execution time may greatly vary from one process to the next. An established way to measure time in a distributed-memory environment consists in adding a MPI barrier before starting and stopping the timer. The call before starting the timer forces all processes to start simultaneously and the call before stopping the timer ensures that the time measurement includes the slowest process. Doing so, the execution time from (e.g.) process 0 then corresponds to the overall MPI execution time. In TTK, this can be done using the two functions `startMPITimer` and `stopMPITimer`.

However, the two MPI barriers add synchronization points, slowing down the execution. Hence, the execution time is not measured by default but only when the compilation variable `TTK_ENABLE_MPI_TIME` is set to `ON` for TTK.

APPENDIX C
ADDITIONAL PERFORMANCE RESULTS

This appendix provides further details regarding the performance of the distributed algorithms presented in this paper.

Specifically, Fig. 14 and Fig. 15 show the same performance results as in the section 7.1.1 and section 7.1.2 of the main manuscript (*"Strong scaling"*) and (*"Weak scaling"*), but expressed in terms of running time instead of efficiency.

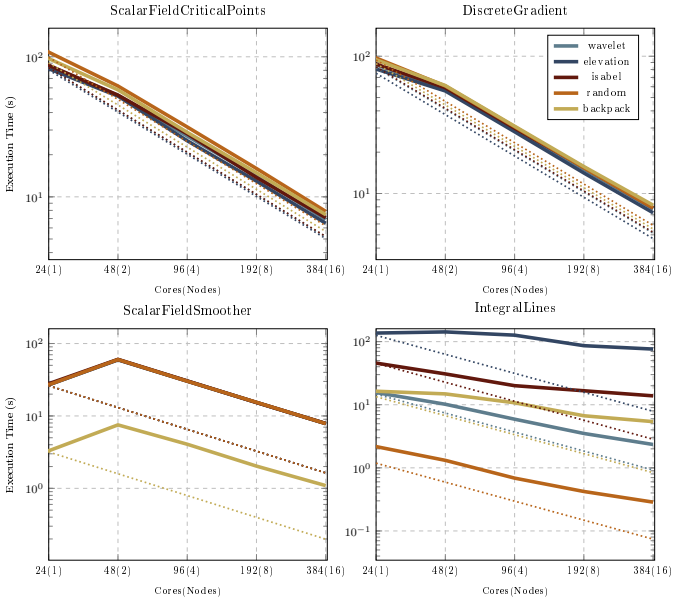

Fig. 14: Strong scaling (execution times) for various algorithms (MPI+thread: 1 MPI process and 24 threads per node). The dotted lines indicate ideal performances.

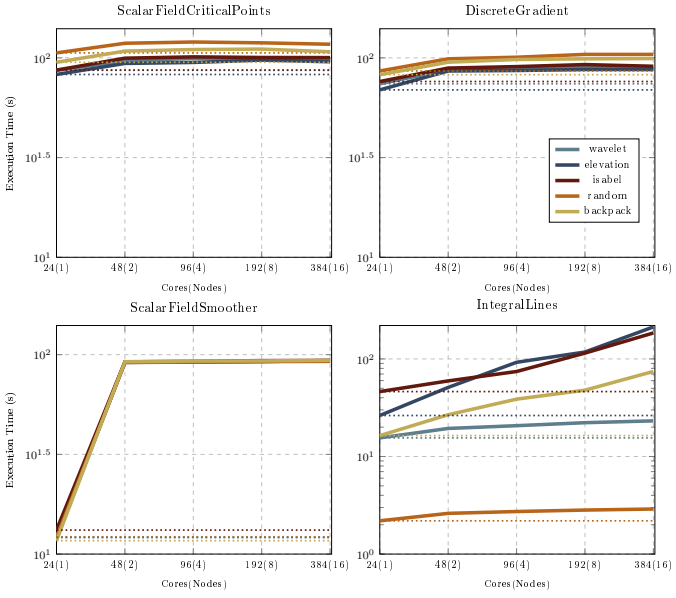

Fig. 15: Weak scaling (execution times) for various algorithms (MPI+thread: 1 MPI process and 24 threads per node). The dotted lines indicate ideal performances.