A Task-Parallel Approach for Localized Topological Data Structures

Guoxi Liu (iD), Student Member, IEEE, and Federico Iuricich (iD), Member, IEEE

Abstract—Unstructured meshes are characterized by data points irregularly distributed in the Euclidian space. Due to the irregular nature of these data, computing connectivity information between the mesh elements requires much more time and memory than on uniformly distributed data. To lower storage costs, dynamic data structures have been proposed. These data structures compute connectivity information on the fly and discard them when no longer needed. However, on-the-fly computation slows down algorithms and results in a negative impact on the time performance. To address this issue, we propose a new task-parallel approach to proactively compute mesh connectivity. Unlike previous approaches implementing data-parallel models, where all threads run the same type of instructions, our task-parallel approach allows threads to run different functions. Specifically, some threads run the algorithm of choice while other threads compute connectivity information before they are actually needed. The approach was implemented in the new Accelerated Clustered TOPOlogical (*ACTOPO*) data structure, which can support any processing algorithm requiring mesh connectivity information. Our experiments show that ACTOPO combines the benefits of state-of-the-art memory-efficient (TTK CompactTriangulation) and time-efficient (TTK ExplicitTriangulation) topological data structures. It occupies a similar amount of memory as TTK CompactTriangulation while providing up to 5x speedup. Moreover, it achieves comparable time performance as TTK ExplicitTriangulation while using only half of the memory space.

Index Terms—Data structures, parallel computation, topological data analysis, simplicial complex

---

1 INTRODUCTION

The proliferation of high-resolution scanning devices is increasing the availability and size of unstructured meshes in applications such as computer graphics [14, 42, 59], material science [3, 58], medical modeling [49, 51], environmental science [13, 32, 56], and autonomous navigation [1, 34].

Despite their widespread adoption, processing and visualizing unstructured meshes still represents a major bottleneck in the analysis pipeline. With regular data, computing and storing the connectivity of the mesh elements has a negligible cost since all information is implicitly provided by the data regularity. With unstructured data, instead, the same operation increases the memory footprint to the point of saturating the available memory.

Dynamic data structures have been proposed [20, 40] to cope with this problem by managing memory usage at runtime. The key idea of these approaches is to compute connectivity information only for a subset of the mesh at a time, discarding information when no longer needed. This approach provided advantages for memory consumption, but resulted in algorithms two to four times slower than state-of-the-art data structures [40].

In this paper, we propose a new block-based task-parallel computation model to obtain data structures that are both time and memory efficient. *Block-based* [20,40,55] indicates a data structure that executes fine-grain operations locally (on subsets of the input mesh) rather than processing the entire mesh at once. *Task-parallel* [36] indicates a data structure integrating pipelined data computation and data consumption tasks. The combination of these two characteristics allows the data structure to self-organize resources at runtime, computing information before they are needed for local data consumption and discarding information when no longer needed.

The main contributions of this work include:

- A new task-parallel computation model for unstructured mesh processing;

- A new data structure implementing the proposed model;

- *Guoxi Liu and Federico Iuricich are with School of Computing, Clemson University. E-mail: {guoxil | fiurici}@clemson.edu .*

- A comparison with state-of-the-art topological data structures on a wide range of unstructured meshes and processing algorithms (both sequential and parallel);

- An open-source integration of the proposed data structure in the TTK framework [53].

2 BACKGROUND

In this section, we introduce the necessary background information for topological data structures, including the notion of simplicial complex and topological relation.

2.1 Simplicial complex

A k-simplex (or simplex of dimension k) is defined as the convex hull of $k+1$ linearly independent points in the Euclidean space. A 0-simplex is also referred to as a point, a 1-simplex as an edge, a 2-simplex as a triangle, and so on. Given a k-simplex σ, the convex hull of a nonempty subset of size $m+1$ of the $k+1$ points (i.e., $m < k$) that defines an m-simplex τ is called an *m-face* of σ, and σ is said to be a coface of τ. The set of cofaces of a simplex σ forms the *star* of σ.

A simplicial complex Σ is a set of simplices such that every face of a simplex σ is also in Σ, and the intersection of any two simplices σ and τ is either a face of both or empty. A simplex that is not a proper face of any other simplex in Σ is called *top simplex*. The *dimension d* of Σ is equal to the largest dimension of any simplex in Σ.

2.2 Topological relations

Three types of topological relations describe the connectivity of the simplices in a simplicial complex Σ. The *boundary* relation maps a simplex to its faces, the *coboundary* relation maps a simplex to its cofaces, and the *adjacency* relation maps a simplex to other simplexes next to it. Suppose that two simplices σ and τ are in Σ, and σ is a face of τ, we say that σ is on the *boundary* of τ, and similarly, τ is on the *coboundary* of σ. Two k-simplices τ_1 and τ_2 are *adjacent* if and only if they share a common $(k-1)$-simplex σ, and two vertices are adjacent if they are on the same edge.

In this paper, we focus on the topological relations between the simplices of a tetrahedral mesh and use capital letters to indicate whether the relation involves a vertex (V), edge (E), triangle (F), or tetrahedron (T). Each topological relation is represented with a pair of letters, e.g., FE relation denotes the edges on the boundary of a triangle. For a tetrahedral mesh, there are six boundary relations (EV, FV, TV, FE, TE, TF), six coboundary relations (VE, VF, VT, EF, ET, FT), and four adjacency relations (VV, EE, FF, TT).

Figure 1 shows two example topological relations for a simple tetrahedral mesh composed by two tetrahedra sharing a common triangle face. Figure 1(a) shows the VE relation for the vertex v_0, which involves the edges e_0, e_1, e_2, and e_3 highlighted in red. Figure 1(b) shows the FV relation for the triangle f_0, which involves the vertices v_0, v_1, and v_2.
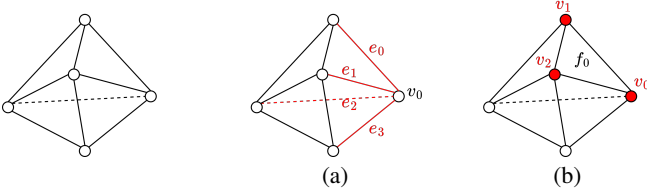

Fig. 1: A simplicial complex composed by two tetrahedra sharing a triangle face. (a) VE relation for vertex v_0. (b) FV relation for triangle f_0.

3 RELATED WORK

In this section, we review data structures designed for encoding simplicial complexes, and we provide an overview of how parallel computation is used for stuying the mesh topology.

3.1 Data structures for simplicial complexes

In this work we focus on data structures that allow the retrieval of any topological relation in a mesh. A number of data structures encode unstructured meshes without representing their connectivity information. These are useful for specific tasks (e.g., ray tracing [54]) but are inadequate to support topological algorithms.

Data structures that provide access to topological relations can be classified into two categories: *static* and *dynamic*.

Static data structures. The approach adopted by static data structures is to compute and store topological relations at initialization time. Differences among them are to be found in the types of relations they encode.

The *Incidence graph* [18] is the most general static data structure for simplicial complexes of arbitrary dimension, which explicitly encodes all simplices and all boundary and coboundary relations. Given the huge memory consumption it requires, several compact alternatives have been developed to reduce its memory footprint [16, 17].

The *Simplex tree* [6] avoids encoding boundary relations by organizing all simplices of Σ in a trie [4]. The result is a data structure that can efficiently support the query of coboundary relations, but that still has limited scalability when working with simplicial complexes in high dimensions [20].

The *Half-edge* data structure [43] is a well-known data structure for triangle meshes, which reduces the storage costs by only encoding the topological relations involving edges. *Half-faces* [38] generalizes the concept of the half-edge to polyhedral complexes.

Indexed data structures [39] provide a more compact option by encoding only vertices, top simplices, and the boundary relation from top simplices to their vertices. It contains sufficient information to extract efficiently all the boundary relations of cells, but it requires additional steps for coboundary or adjacency relations.

Several data structures have been developed to encode the connectivity through adjacency relations. Examples include the *Indexed data structure with Adjacencies (IA data structure)* [44, 47] and the *Corner-Table* data structure [48] along with its several extensions specifically proposed for triangle meshes [24, 41] and tetrahedral meshes [25]. The *Generalized Indexed data structure with Adjacencies (IA* data structure)* [8] extends the IA data structure to non-manifold simplicial complexes of arbitrary dimension. The IA* data structure has shown to be most compact among static topological data structures, especially as the dimension increases [7].

Dynamic data structures. Unlike static data structures, dynamic data structures compute (and discard) topological relations during runtime rather than at initialization time.

The *PR-star octree* [55] is considered the first dynamic topological data structure. It supports the reconstruction of the connectivity information of a simplicial complex by only encoding the list of tetrahedra incident in each vertex. The data structure is capable of extracting the boundary and coboundary relations locally to a subset of the mesh by using a PR-Octree decomposition of the mesh vertices.

The *Stellar tree* data structure [20] generalizes the PR-star octree to handle a broader class of complexes in arbitrary dimensions and is the first concrete realization of the *Stellar decomposition* model [20]. The Stellar tree is shown to be more compact than most state-of-the-art static data structures, requiring only a fraction of the memory space of the latter [20].

The Stellar decomposition model has also been adopted by the *TopoCluster* data structure [40], which enriches the Stellar decomposition with an implicit enumeration scheme for the mesh simplices. This scheme provides an interface for the easy integration of *TopoCluster* into any algorithm for topological data analysis. The easy and general integration of TopoCluster was demonstrated by deploying the data structure in the TTK framework [53], which allowed running any algorithm implemented in the framework out-of-the-box while drastically reducing the memory footprint.

3.2 Parallel computation for topological data analysis

Rather than focusing on the speedup provided by the underlying data structure, some research work aimed at improving the performance of specific topological algorithms. To this end, parallel computation plays a major role in topology-based visualization [15, 31, 57]. While certain routines are embarrassingly parallel by nature, (e.g., critical points [2] or Forman gradient computation [47]), the extraction of many topological abstractions requires the study of dedicated parallel approaches. Notice that all the following methods have been evaluated on regular grids where the dataset subdivision and topological information are implicitly encoded.

Merge and contour trees. The *contour forests* algorithm [22] presents a fast, shared memory multi-threaded computation of contour trees on tetrahedral meshes. The approach partitions the domain first, computes the local contour trees for each partition, and stitches the resulting forest into the final augmented contour tree. Gueunet et al. proposed a new approach based on Fibonacci heaps [23] that skips the domain subdivision step by distributing the computations of the merge tree arcs to independent tasks on the CPU cores. A pure data-parallel algorithm with the support of GPU acceleration, *Parallel Peak Pruning (PPP)* [10, 11], has been developed for computing both merge and contour trees in unaugmented form using OpenMP for threading and using Thrust for GPU. The PPP algorithm presents up to 70x speedup compared to the serial sweep and merge algorithm supporting the contour tree computation for arbitrary (topology) graphs [9]. In contrast to building the global merge tree, the *merge forest* approach [37] decomposes the domain, maintains the local merge trees connected by a local *reduced bridge set*, and computes the necessary global information only at query time. As a result, the merge forest represents a localized data structure designed for answering queries related to merge trees without expensive precomputation costs. This idea has been generalized further for distributed environments [35].

Morse-Smale complex. Parallel algorithms for computing a 3D Morse-Smale (MS) complex [30, 45] extend the divide-and-conquer strategy presented by Gyulassy et al. [26]. The idea is to partition data into blocks, compute the MS complex for the individual blocks, and then merge the MS cells with a dedicated merge-and-simplify routine. Many approaches have focused on the geometric accuracy of the reconstructed model rather than the efficiency of the parallel approach [5, 27–29]. An exception is the hybrid (CPU-GPU) shared-memory algorithm proposed by Shivashankar et al. [50]. The algorithm assigns embarrassingly parallel tasks such as gradient computation and extreme traversals to the GPU, and thus results in substantial

speedup over CPU-based approaches. A pure GPU parallel algorithm for computing the MS complex has also been developed recently [52], which transforms the graph traversal operations into vector and matrix operations that are better suited for GPU parallel computation.

4 TASK-PARALLEL COMPUTATION APPROACH

In this section, we describe our proposed task-parallel model. To clarify the difference between our proposed model and existing approaches, we first review common computation models used by static and dynamic data structures.

Static data structures use a preprocessing approach, where topological relations are computed in bulk during the initialization of the data structure and then stored until the algorithm has been executed. As shown in Fig. 2(a), thread t_1 first computes the topological relations of the entire mesh, and then uses them in the selected algorithm.
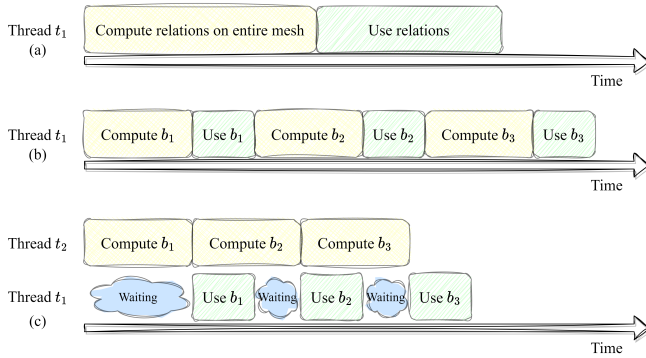

Fig. 2: (a) Workflow of typical static data structures. The thread computes topological relations for the entire mesh first and then use them in the algorithm. (b) Workflow of the traditional dynamic approach. The thread works with one block at a time. It computes and uses the topological relations in a sequential order. (c) Workflow of the proposed task-parallel approach with two threads. One is only responsible for precomputing topological relations, and the other executes the algorithm by using the topological relations.

Dynamic data structures compute information on-the-fly by processing one subset of the mesh at a time, which is achieved by dividing the dataset into blocks [20, 40, 55]. In practice, the running thread accesses one of the blocks, computes the required topological relations, and then runs the algorithm inside the block. Fig. 2(b) shows an example of such workflow executed by thread t_1 that computes the topological relations for the mesh subset b_1, executes the algorithm locally on b_1 using these relations, and then moves to the next portion of the mesh b_2 to repeat the same process.

The goal of our task-parallel model is to avoid the switching between algorithm execution and topological relations computation. Our model still assumes that the input mesh is subdivided into blocks. However, it introduces two types of threads, specialized in the computation of topological relations and in the algorithm execution, respectively. Fig. 2(c) shows an example of our proposed approach. Thread t_1 is created for running the processing algorithm, while thread t_2 is used for computing topological relations. In practice, thread t_2 precomputes topological relations for thread t_1 so that, as long as topological relations are provided, t_1 keeps executing instructions from the selected algorithm on each block.

The key difference in the execution of thread t_1 is that the time originally spent to compute topological relations in the traditional approach (Fig. 2(b)) is now replaced by a waiting time (Fig. 2(c)) in the new approach. The more efficient thread t_2 computes topological relations for t_1, the shorter t_1's waiting time will be.

4.1 Consumers and producers

Our approach integrates a classic producer-consumer paradigm with constrained consumers [33]. Threads that compute topological relations

are called *producers*. Threads that use these topological relations to run a processing algorithm are called *consumers*.

Our model involves two types of producers. *Worker producers* are unconstrained threads that can compute topological relations in any block without a specific request. *Leader producers* are constrained producers that compute topological relations only if specifically asked by a consumer. Moreover, they manage the communication between consumers and worker producers.

Fig. 3 shows the general workflow and communication strategy used by these types of threads. In the following, we provide a detailed description of the behavior of each thread type, assuming a sequential algorithm is being executed (i.e., a single consumer thread is involved).
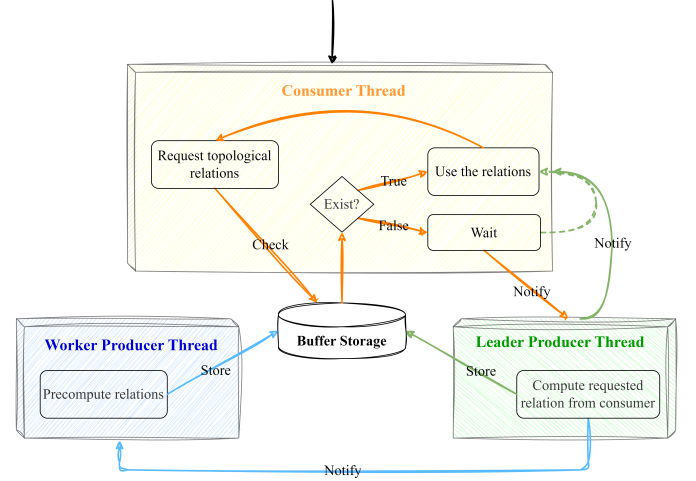

Fig. 3: The task-parallel producer-consumer paradigm. Orange, green, and blue colors indicate the activities of the consumer, leader producer, and worker producer threads, respectively.

Consumer. The consumer thread is the first thread to be spawned. The consumer thread spawns the leader producer thread and runs the selected processing algorithm that makes requests for topological relations. Our model involves a buffer system that all threads can use to store/retrieve topological relations (see Fig. 3). Assuming that a topological relation R for a simplex σ is requested, the consumer first identifies the block b_i containing σ. Then, it searches the buffer to verify if the relation R for the block b_i has been computed. If yes, the consumer keeps running the processing algorithm inside b. Otherwise, the consumer stops its execution and invokes a request to the leader producer.

Leader producer. The leader producer acts as a middle-layer between the consumer and the worker producers. During the initialization phase, the leader producer creates the worker producers and starts monitoring the consumer's working block (i.e., on which block the consumer is running the processing algorithm).

Every time the consumer requests a topological relation for a simplex $(e.g., R(\sigma))$ that is not stored in the buffer, the leader producer promptly computes the relation R for the block b_i containing the simplex σ and notifies the consumer that the topological relation $R(\sigma)$ is ready. Then, the leader producer notifies the worker producers that topological relations will be needed in the proximity of b_i.

Worker producer. Similar to the leader producer, the task of worker producers is to precompute topological relations in a block and store them in the buffer. The difference is that worker producers process blocks independently without an explicit request, unless they are notified by the leader producer.

Worker producers are spawned by the leader producer and are allocated to a block b_i to compute topological relations. Worker and leader producers share a variable indicating the index i of the block b_i where the consumer is working. Moreover, they share a second variable indicating the index j of the last topological relation R_j requested by

the consumer. These variables are controlled by a lock to prevent multiple producers from computing the same topological relation on the same block. Once a worker thread acquires the lock to access b_i, it modifies either b_i or R_j to the new value for next worker, releases the lock, and then proceeds with computing the topological relation with saved R_j and b_i before modification. This way, the next worker thread that acquires the lock will not compute the same relation on the same block (see Section 4.1.1 for details on how worker producers update the block b_i and topological relation index R_j).

This workflow can only get interrupted by the leader producer at any time, which will force all worker threads to move to $b_{i'}$ by updating the corresponding shared variable.

4.1.1 Computing new blocks for worker producers

After studying a wide range of processing algorithms, particularly those related to topology-based visualization, we have recognized key differences in their access patterns (the order in which they access blocks) and the types of topological relations they request. To address this diversity, we have defined multiple computing modes for worker producers to experiment with different orders for visiting blocks and the number of topological relations to compute. All computing modes have been implemented and experimentally evaluated (see Section 6).

Moving directions. As discussed previously, worker producers control the exclusive access to blocks through a shared variable. We have defined two possible strategies that a worker thread can use to update the variable.

The first strategy aims at precomputing relations for blocks with the index following the block b_i, i.e., the next block to compute is $Next(b_i) = b_{i+1}$. The strategy is designed based on the fact that many topology-based visualization algorithms loop through the simplices in the mesh based on their indices. In most block-based data structures [20, 40, 55], block indices follow the same order as the global indices of simplices. We refer to this buffer strategy as *linear* buffer, which is demonstrated in Fig. 4(a). Given the current block is b_5, the worker thread will precompute block b_6, followed by b_7, b_8, and b_9.

The second strategy aims at precomputing relations for all neighbors of a block b_i. We say two blocks are neighbors if they share a common tetrahedron. This strategy is motivated by the fact that certain algorithms (e.g. Morse-Smale computation [19]) visit simplices based on their connectivity rather than their indexing. These algorithms visit neighboring blocks in an unpredictable order which motivates the precomputation of all blocks surrounding the current one. We refer to this buffer strategy as *spatial* buffer. Fig. 4(b) shows an example of the moving direction in the spatial order. Since the neighbor set of the current block b_5 is $Neighbors(b_5) = \{b_1, b_2, b_3, b_4, b_6, b_7, b_8, b_9\}$, after b_5 is computed, the worker thread will start visiting blocks in $Neighbors(b_5)$, e.g., b_1.
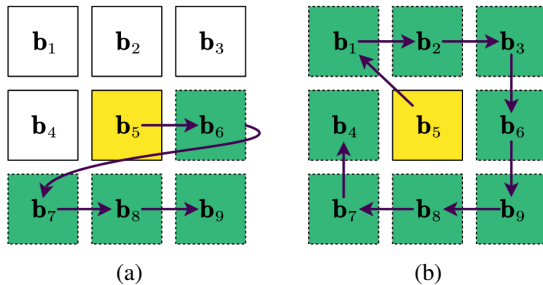

(a)   (b)

Fig. 4: Different moving directions (in green color) of worker producers for block b_5: (a) linear order, (b) spatial order.

Topological relations. After moving to a new block b_i, the worker producer will compute topological relations R for the simplices in b_i. At this point, the worker thread has two options, and the most straightforward one is to compute only the topological relation R specified in the last consumer's request. In this case, the worker

producer will update the block index once finished (based on the moving direction mentioned earlier) while leaving the topological relation index R_j untouched.

However, we observed that processing algorithms rarely use only one topological relation at a time. Instead, multiple topological relations are usually requested for the same block or for the same simplex. Therefore, an alternative approach for the worker producer is to compute all types of topological relations used by the algorithm when accessing a new block. In this case, each worker producer will update the relation index R_j to indicate that more topological relations need to be extracted in this block. Once all topological relations are precomputed, the last worker thread accessing the block will increment the block index b_i, according to the specified moving direction.

5 THE ACTOPO DATA STRUCTURE

Using the task-parallel framework described in Sec. 4, we implement a new data structure called Accelerated Clustered TOPOlogical (*ACTOPO*) data structure. In the remainder of this section, we describe the encodings of ACTOPO and the implementation choices made to integrate consumer, leader producer, and worker producer threads in the data structure.

5.1 Encodings of static information

The static information encoded by ACTOPO comprises the representation of the input unstructured mesh and additional information computed at initialization time. In the following, we assume that the input is a tetrahedral mesh that has already been subdivided into blocks. However, the same approach could be used to encode meshes defined by general polytopes.

Input. The input tetrahedral mesh contains only information for vertices and tetrahedra. An indexed list V stores the coordinate values of each vertex. An indexed list T stores the vertices forming each tetrahedron (*TV* relation).

The data structure also assumes a subdivision of the mesh is defined based on the mesh vertices and is also provided in the input. Vertices with the same label belong to the same *block*. An indexed list I is used to encode the block that each vertex belongs to. It is a requirement of the subdivision that each vertex is contained in exactly one block.

Fig. 5 shows an example of the encodings of an input mesh. The example tetrahedral mesh shows that v_0 and v_1 belong to the same block (i.e., $I[v_0] = I[v_1] = b_1$), while the remaining vertices are in the block b_2 (i.e., $I[v_2] = I[v_3] = I[v_4] = I[v_5] = 2$).

Based on the subdivision of the mesh vertices we can create an association between a simplex σ and a block b_i. Specifically, we say that a simplex σ is *internal* to the block b_i iff. the vertex v of σ with the lowest index also belongs to b_i and is *external* to all other intersecting blocks.
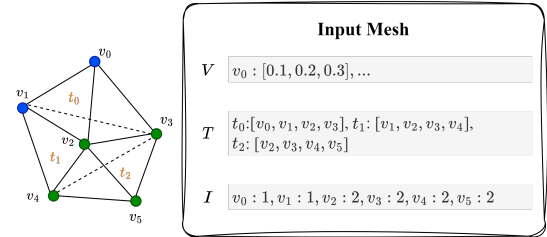

Fig. 5: The input mesh that includes the vertex list V, tetrahedron list T, and a subdivision I.

Initialization. In addition to the input mesh, the data structure computes and stores the boundary relations of each simplex σ with its vertices. Since this information is not provided for edges and triangles (i.e., EV, FV), these relations are computed during the initialization phase.

The extraction is performed by iterating through the tetrahedra in each block. For each tetrahedron, all possible combinations of two

vertices (*EV* relation) and three vertices (*FV* relation) are collected and stored in an indexed list after removing duplicates. Fig. 6 shows the additional information encoded by the data structure after the initialization. Two indexed lists E and F are used to store the edges and triangles globally. For each block b_i, we also encode a list of external tetrahedra T_{ex}.
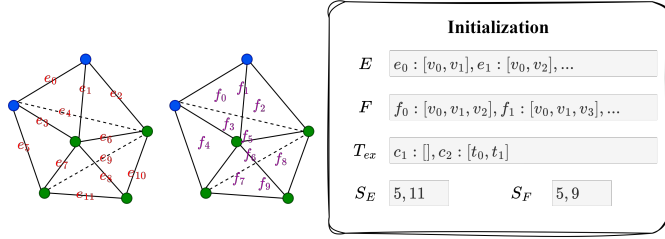

Fig. 6: Encodings of ACTOPO after initialization, which include edge list E, triangle list F, external tetrahedra list T_{ex}, and interval arrays for edges and triangles S_E and S_F.

In addition, the data structure maintains an interval array for edges and triangles, S_E and S_F respectively, indicating the largest edge and triangle indices contained in each block. These arrays are used to get local edges and triangles of a block from the global edge and triangle lists E and F.

5.2 Operations supported at runtime

As mentioned earlier in Sec. 4.1, every time the algorithm requires a topological relation of a simplex σ, the consumer will first locate the block b_i containing σ. This step requires constant time, as $b_i = I(v)$ for a vertex $v = \sigma$ and $b_i = min_{v \in \sigma} I(v)$ for any other simplex σ.

The buffer system is implemented using a linked list that stores the index i of each precomputed block b_i. While the linked list ensures fast insertion and deletion operations, a supplemental hash table of block indices is used for quick lookup. Therefore, it just takes constant time for the consumer thread to check whether a block b_i is available in the buffer. From the consumer thread's perspective, the buffer works like a black box. This feature makes the model universally applicable to any topological algorithm.

While the consumer thread can only read data from the buffer, the leader producer is the thread that actually manages the buffer. The buffer system in ACTOPO operates on a First-In-First-Out (FIFO) order, and an additional integer value caps the maximum number of blocks allowed in the buffer. Before notifying worker producers, the leader producer checks the free space remaining in the buffer. If the buffer is at capacity, the leader producer cleans up half of the buffer based on the chronological order in which blocks were added.

It is worth noting that the computation of a local topological relation R within a block b_i is not different from the approach used by other dynamic data structures [20, 40]. Typically, this involves iterating through internal, and sometimes external, tetrahedron list of b_i, and constructing the simplices of interest from vertices of the tetrahedron (e.g, a pair of vertices can form an edge). Due to the limited space available, we report the pseudocode used to extract a representative set of these operations in supplemental materials.

6 EVALUATION OF PERFORMANCE

In this section, we present an experimental evaluation of our proposed data structure when used in combination with sequential algorithms. All experiments are performed on a desktop computer equipped with a 3.2 GHz Intel i7-8700 CPU and 32 gigabytes of RAM. We report the average values obtained over 10 runs for each experiment.

6.1 Experimental setup

The tetrahedral meshes selected for the performance analysis are listed in Tab. 1. Datasets Shapes and Hole are irregular tetrahedral meshes. The remaining datasets (i.e., Red Sea, Engine, Earthquake, Foot, Asteroid, and Stent) are tetrahedral meshes created by tetrahedralizing

volume images while filtering out elements corresponding to empty parts of the original data domain.

Table 1: Overview of the experimental datasets, including the number of vertices $|V|$, edges $|E|$, triangles $|F|$, and tetrahedra $|T|$. The type field indicates whether the dataset was originally a volume image (*Regular*) or points are irregularly distributed within the domain (*Irregular*).

Dataset	Type	$|V|$	$|E|$	$|F|$	$|T|$
Red sea	Regular	0.95M	6.33M	10.58M	5.20M
Engine	Regular	1.39M	9.14M	15.18M	7.43M
Earthquake	Regular	1.62M	11.14M	18.92M	9.41M
Foot	Regular	4.60M	30.79M	51.51M	25.32M
Asteroid	Regular	5.07M	35.00M	59.58M	29.65M
Shapes	Irregular	7.87M	52.37M	87.63M	43.13M
Hole	Irregular	9.26M	63.70M	108.29M	53.85M
Stent	Regular	17.37M	118.79M	201.40M	99.98M

ACTOPO has been implemented as a new module of the Topology Toolkit (TTK version 1.1.0), and thus all plugins implemented in TTK can run seamlessly using the new data structure. We used four different TTK plugins to evaluate the performance under different runtime conditions.

TestTopoRelations plugin is a plugin developed in-house for profiling the data structure under simplified conditions. The plugin iterates over the simplices of the mesh starting from the vertices and successively moving to edges, triangles, and tetrahedra. For each simplex σ the plugin requires the computation of every boundary and coboundary relation involving σ. The plugin traverses the tetrahedral mesh following the linear order of simplex indices, i.e., from σ_0 to σ_n, where σ denotes a k-simplex in the tetrahedral mesh and n is the total number of k-simplices in the mesh. It is selected to test the performance of the topological data structure in computing topological relations one at a time and to avoid recomputation for localized topological data structures since the topological relations of a block will only be requested once.

ScalarFieldCriticalPoints plugin is used to compute critical points based on an input scalar function. The plugin requires topological relations involving the vertices (i.e., VV, VF, and VT). The only exception is the FT relation, which is used to identify the list of vertices on the boundary of the mesh. The plugin traverses the vertices of the mesh in the same linear order as TestTopoRelations plugin, and then marks vertices that are on the boundary of the mesh. However, the plugin requires multiple topological relations to be computed when visiting one block instead of only one relation as in the previous testing plugin.

DiscreteGradient plugin [21] computes a *discrete Morse gradient field* based on an input function. The function F is a discrete Morse function if for any p-simplex σ, all the $(p-1)$-simplices on its boundary have a lower F value and all the $(p+1)$-simplices on its coboundary have a higher F value, with at most one exception. If such exception exists, it defines a pair of cells called a *discrete gradient vector*. Otherwise, p-simplex σ is a *critical* simplex of index p. Intuitively, a discrete vector field can be viewed as a collection of arrows, connecting a p-simplex of mesh Σ to an incident $(p+1)$-simplex in such a way that each simplex is a head, or a tail of at most one arrow and the critical simplices are neither the head nor the tail of any arrow. The plugin iterates all the 0-simplices (i.e., vertices), and for each vertex, it adds all k-simplices ($k > 0$) in the lower star of the vertex into a list. If the list is not empty, the plugin finds the pairable 1-simplex (i.e., edge) for the vertex, 2-simplices (i.e., triangles) for the remaining 1-simplices, and so on. Multiple topological relations are used when computing the discrete gradient vector, specifically, VE, VF, VT, EF, ET, FE, FT, and TF relations. Even though the plugin visits vertices sequentially, a simplex can be the face/coface of multiple simplices, and the recomputation of its topological relations will be requested multiple times during the process.

MorseSmaleComplex plugin [50] computes a Morse-Smale (MS) complex from an input scalar function on the given tetrahedral mesh. An *integral line* is a path on the mesh that is tangent to the

gradient of the function everywhere. Intuitively, the *MS complex* is a segmentation of the input scalar field in regions where integral lines are connected to the same pair of critical points. The plugin first computes the discrete gradient on the tetrahedral mesh as described in the `DiscreteGradient` plugin, and the remaining steps require to visit the mesh based on the discrete gradient vector, including the computation of 1-separatrices, saddle connectors, and segmentation. The order in which simplices are visited is defined at runtime. For this reason, this plugin denotes the worst-case scenario of the localized data structure, as recomputation of one same block could happen multiple times during the process. Since the first step of the plugin is overlapped with the previous `DiscreteGradient` plugin, we measure its performance by only counting the steps after the discrete gradient computation.

6.2 Evaluating robustness on sequential algorithms

In this section, we test how robust the proposed approach is when varying the model's parameters, namely, the four different computing modes and the total number of producers. For each execution, we track the amount of time to finish the algorithm execution and the peak memory usage observed during the process. Notice that the execution time of the plugin includes the waiting time of the consumer (i.e., time spent waiting for a topological relation to be ready) and algorithm execution time (i.e., time spent executing the algorithm's instructions). Since parameter changes in the data structure do not affect the algorithm, a reduction in the overall execution time is attributed to a reduction in the waiting time.

6.2.1 Comparison of computing modes for worker producers

The working modes described in Sec. 4.1.1 generate four possible configurations for the worker producers. Producers can visit blocks either linearly or following a spatial order. Moreover, at each request, producers can either compute the requested topological relation or all topological relations predefined by the algorithm. Our hypothesis is that each mode may be preferable based on the algorithm's characteristics. Specifically, the linear computing mode should benefit plugins that iterate through simplices in sequential order, i.e., `TestTopoRelations`, `ScalarFieldCriticalPoints`, and `DiscreteGradient` plugins. The spatial computing mode should be beneficial for plugins that visit simplices with an unpredictable order (e.g., `MorseSmaleComplex` plugin). In the following experiments, we focus on the computing mode and test these hypotheses using 6 producers in total (i.e., 1 leader producer and 5 worker producers).

Fig. 7 shows the results obtained on three plugins where the linear buffer was expected to provide the best performance.

For `TestTopoRelations` plugin (Fig. 7(a)), we can notice that the topological relation computation makes the real difference. While the linear buffer provides a small benefit compared to the spatial buffer, computing only the required topological relation provides a 5.2 times speedup compared to computing all topological relations used by the algorithm.

The opposite is true for `ScalarfieldCriticalPoints` (Fig. 7(b)) and `DiscreteGradient` plugins (Fig. 7(c)). Although differences across modalities are less pronounced, computing all listed topological relations using the linear buffer results in algorithms running 14% faster. This is reasonable since the plugin uses multiple relations for each vertex at a time. Not surprisingly, precomputing only one single topological relation also works better for the `MorseSmaleComplex` plugin (Fig. 8(a)). Unlike the other two plugins, this plugin utilizes topological relations to reconstruct Morse-Smale cells while navigating the entire mesh. Due to the navigation process only focusing on one topological relation at a time, the single mode is beneficial. Moreover, the spatial buffer provides slightly better performance than the linear one. This is because the computation of the MS complex follows the path according to a discrete gradient field and does not always visit blocks in a linear order. The spatial buffer precomputing a single relation is 85% faster on average.

Memory usage does not change significantly across modalities. As we can expect, the work modes precomputing multiple relations use more space compared to those precomputing only one relation, up to 18% as observed in our experiments. Overall memory usage grows linearly with the number of tetrahedra in the input mesh. We include detailed graphs in the additional materials only. The only exception is the `MorseSmaleComplex` plugin (Fig. 8). For this plugin, we can also notice that the time and memory do not increase consistently as the size of datasets increases; this is because extracting the MS cells depends on the size of the MS complex instead of the input mesh (i.e., output-sensitive). Still, computing multiple relations for each block requires about 12% more memory space than computing a single relation.

Lessons learned. Overall, the results match our hypothesis. Specifically, the computing mode of linear buffer with the single relation works best for `TestTopoRelations` plugin, the mode of linear buffer with all required relations performs best for `ScalarField-CriticalPoints` and `DiscreteGradient` plugins, and the mode of spatial buffer with single relation is optimal for `MorseSmaleComplex` plugin. Furthermore, using a spatial buffer with computing all required relations is not optimal for any of the testing plugins.

6.2.2 Comparison of different numbers of producers

The number of producer threads is another parameter that affects performance. Our hypothesis is that more producer threads should accelerate the execution of the plugin by reducing the waiting time of the consumer thread. For the following experiments, we adopted the best computing mode for each plugin, as observed in Sec. 6.2.1, while changing the number of producer threads from 1 to 10. Given that the same computing mode and buffer capacity are used, differences in memory costs are very limited (with a maximum increase of 16% for `DiscreteGradient` plugin and less than 1% for all others). Therefore, we only discuss time performance in the following, while providing detailed graphs in supplemental materials.

Fig. 9 shows the time performance of all four tested plugins. For all of them, we can notice that using more producers reduces the waiting time of the consumer thread. For all plugins, except the `MorseSmale-Complex` plugin and `ScalarFieldCriticalPoints` plugin on the Stent dataset, the waiting time disappears after using 6 producers (as indicated by the cyan colored bar in the figure). Overall, the speedup plateaus at around 6 producers, after which producers will end up competing with the consumer for computing resource (i.e., which thread to be scheduled), and the context switching adds overhead to the performance. Using 6 threads results in a 4.2 times speedup if we consider the extraction of topological relations only (`TestTopoRelations`), and about 1.7 times speedup for all remaining plugins.

Lessons learned. Overall, the results show that multiple producer threads can improve the time performance of the plugin, but such speedup is limited. For most testing plugins, using 4 producer threads shows significant improvements over the single producer. However, configuring the number of producer threads to exceed the number of physical cores will result in performance degradation. This is expected due to issues such as operating system's scheduling policy, context switching, and critical resource competition.

6.3 Comparing with state-of-the-art

In this section, we compare our proposed data structure with state-of-the-art static and dynamic data structures. For static data structure, we use *ExplicitTriangulation* implemented in TTK. This data structure precomputes and stores all required topological relations during a pre-processing stage so that they are readily available during the algorithm execution. For dynamic data structure, we use *CompactTriangulation*, an implementation of the dynamic data structure proposed by Liu et al. [40]. Similar to ACTOPO, CompactTriangulation organizes the mesh using blocks, computes and discards topological relations locally to each block.

The comparison is intended to evaluate the parallelism at the data structure level instead of the algorithm level. For this reason, the testing algorithms are restricted to the sequential execution. In practice, multiple threads are used for initializing the data structures or, in the case of ACTOPO, threads are used for supporting the extraction of
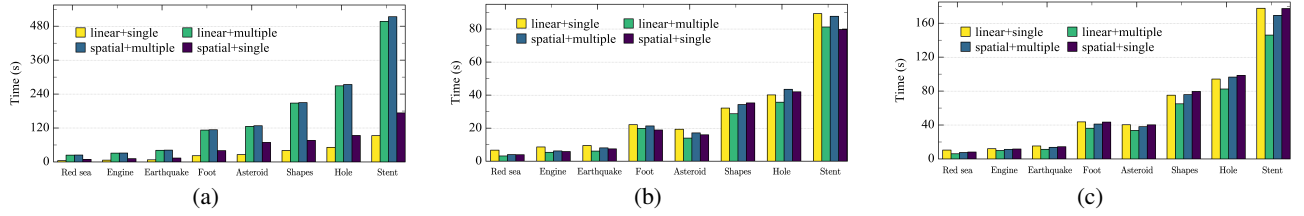
Fig. 7: The execution time (in seconds) used by `TestTopoRelations`, `ScalarfieldCriticalPoints`, and `DiscreteGradient` plugins when running with different computing modes using six producer threads.
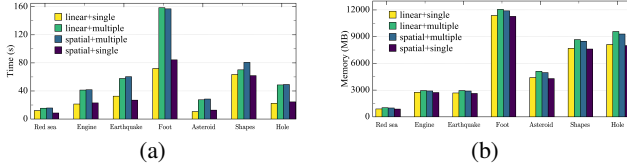

Fig. 8: The execution time (in seconds) and memory usage (in megabytes) used by `MorseSmaleComplex` plugin when running with different computing modes.
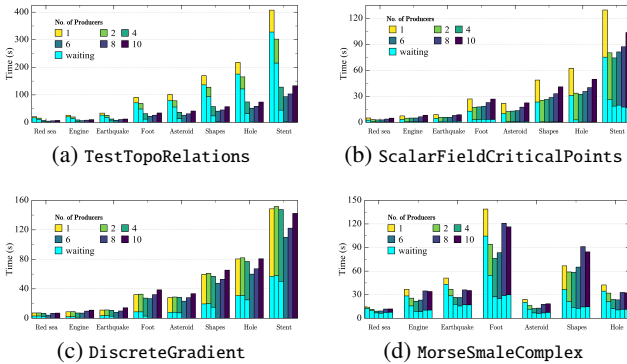

Fig. 9: The total time (in seconds) spent by four testing plugins when running with different numbers of producer threads, and the cyan colored bar denotes the waiting time of the consumer thread.

topological relations (6 producers). Instructions implemented by each plugin are executed by a single thread (1 consumer for ACTOPO) in a sequential manner.

Since the initialization of each data structure is different, we report detailed results about the preprocessing time (steps required to initialize the data structure) and the algorithm execution time. For the following experiments, we have set the number of threads to 6 for all data structures and limited the buffer/cache capacity to 20% of the total number of blocks for both our proposed data structure and TTK CompactTriangulation.

Fig. 10 shows the total time and memory consumptions obtained with three linear plugins. Focusing on the preprocessing step, we notice that our proposed data structure is the second best on average in terms of time performance. *ExplicitTriangulation* is the slowest being in general, 2 times as slow as ACTOPO, and *CompactTriangulation* is 4 times as fast as ACTOPO. For the `TestTopoRelations` plugin (Fig. 10(a)), CompactTriangulation shows the worst performance instead, underlying the limitation of computing topological relations on-the-fly while the algorithm is blocked to wait. Our approach provides a dramatic improvement and is 5.3 times as fast as CompactTriangulation.

In the remaining two plugins (Fig. 10(c) and (e)), topological relations are used for computing additional information. Producer threads used in our approach have enough time to precompute topological relations and save time for the consumer. This allows

ACTOPO to close the gap with ExplicitTriangulation and to show roughly the similar performance. The difference with CompactTriangulation is still significant, since ACTOPO is 3.4 times as fast as CompactTriangulation for `ScalarFieldCriticalPoints` plugin and 1.83 times for `DiscreteGradient` plugin.

In terms of memory usage, even if *CompactTriangulation* is the most compact data structure in general, ACTOPO shows a very similar memory footprint using only 7% more memory on average. As opposed, ExplitTriangulation uses about 3 times the memory of our proposed data structure. The large memory requirement causes ExplicitTriangulation to run out of memory when processing the Stent dataset with `Test-TopoRelations` plugin.

Particular attention is dedicated to the results collected on the `MorseSmaleComplex` plugin, shown in Fig. 11. Since the plugin does not simply iterate through the simplices of the mesh, block-based data structures like CompactTriangulation are forced to recompute topological relations inside a block multiple times with an evident loss in performance, especially when handling the input dataset with a complicated scalar field, such as Engine, Earthquake, and Foot datasets. ACTOPO provides a considerable improvement being 3 times as fast as CompactTriangulation. However, the unpredictable pattern used by the plugin to access blocks makes it harder for the producers to predict which block the consumer will move to next. As a result, the gap between ACTOPO and ExplicitTriangulation is wider with ACTOPO being 2.6 times as slow as ExplicitTriangulation on average. Nevertheless, this is paid off by much better scalability in terms of memory usage, with ACTOPO being 1.8 times as compact as ExplicitTriangulation. As mentioned earlier, the plugin is output-sensitive, which causes both localized data structures to use the most memory when processing the Foot dataset.

7 SUPPORTING PARALLEL ALGORITHMS

Parallel computation makes use of multiple processors to execute computational tasks simultaneously, thus reducing the total time required to complete the task. Various topology-based visualization algorithms have implemented different parallel computation techniques, and one common approach is the data-parallel technique. In this method, data is divided into smaller pieces and processed independently by multiple threads. In this section, we describe how ACTOPO supports a parallel topological algorithm.

For static data structures, data-parallel techniques are used to parallelize the computation of topological relations, or the selected algorithm, independently. For dynamic data structures, data-parallel techniques are used to allow multiple threads to process multiple blocks at the same time [40].

Our proposed ACTOPO data structure can also be adapted to execute parallel algorithms. As mentioned previously, the consumer thread is the one responsible for running the processing algorithm. When an algorithm allows for parallel execution, multiple consumer threads can be used to run instructions in parallel. To this end, we have generalized our model by duplicating producer-consumer associations.

Consumer threads are spawned at the initialization phase of the ACTOPO. Once created, each consumer thread has its own set of

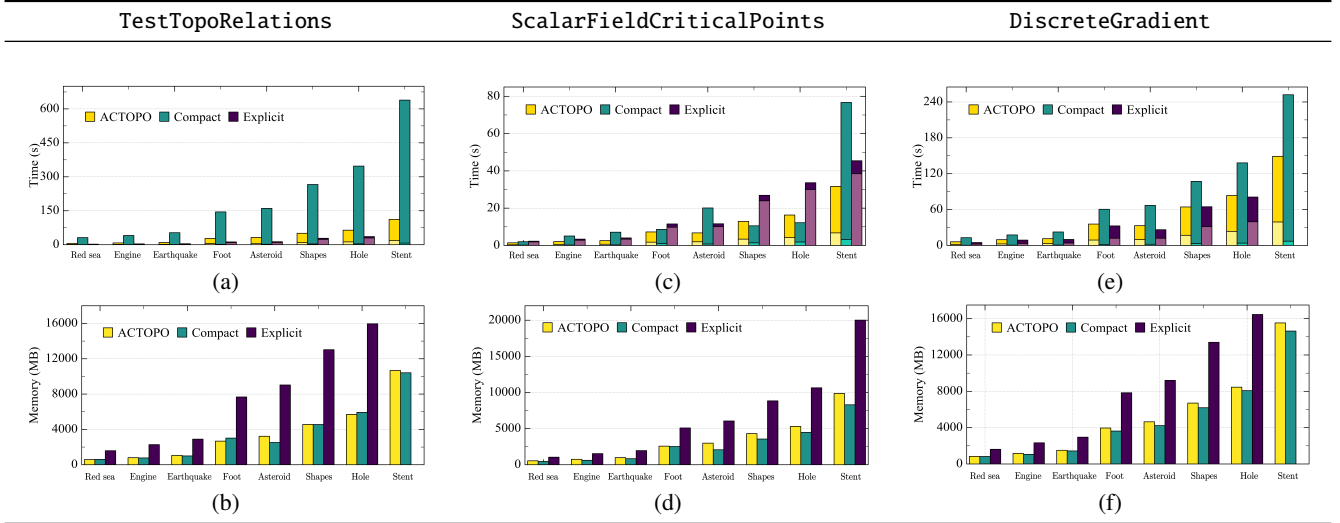| TestTopoRelations	ScalarFieldCriticalPoints	DiscreteGradient

(a) (c) (e)

(b) (d) (f)

Fig. 10: The total time (in seconds) and memory usage (in megabytes) used by three experimental plugins with different data structures: ACTOPO, TTK Compact Triangulation, and TTK Explicit Triangulation when the algorithm runs in sequential. In (a), (c), and (e), the top bar indicates the time required by running the algorithm, while the bottom bar indicates the preprocessing time.
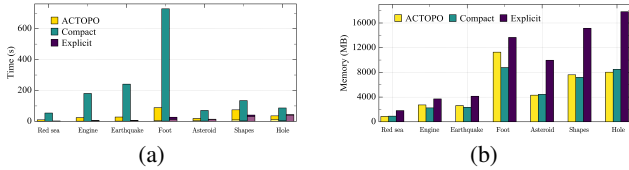

(a) (b)

Fig. 11: The total time (in seconds) and memory usage (in megabytes) used by `MorseSmaleComplex` plugin when running with different data structures: ACTOPO, TTK Compact Triangulation, and TTK Explicit Triangulation.

producers and a dedicated buffer system that is not shared with any other consumer/producer threads. Each type of thread behaves in the same way as described in Sec. 5, i.e., the consumer thread is responsible for the algorithm execution, the worker producer threads precompute topological relations and saves them to the buffer storage, and the leader producer thread monitors both threads and manages the buffer system. The only difference is the exclusive association between consumers and producers.

In the following, we present an evaluation of this producer-consumer paradigm when the topological algorithm runs in parallel. We keep only three of the four TTK plugins originally selected since `MorseSmaleComplex` uses parallelism only for computing the discrete gradient (equivalent to the `DiscreteGradient` plugin), while the extraction of the MS cells is done in a sequential manner.

7.1 Evaluating robustness on parallel algorithms

When working with a parallel algorithm, the total number of consumer and producer threads involved is an important parameter to evaluate. The total number of threads used by ACTOPO is $t_c + t_c \cdot t_{pc}$, where t_c indicates the number of consumer threads, t_{pc} indicates the number of producer threads per consumer thread, including one leader producer and $(t_{pc} - 1)$ worker producers. Thus, each plugin can obtain a relative speedup by either increasing t_c to distribute the computation across more consumer threads, or by increasing t_{pc} to let more worker threads precompute topological relations. Since our testing machine can support at most 12 threads, we can use at least 2 consumers with 5 producers each (i.e., $t_c = 2$ and $t_{pc} = 5$), and at most 6 consumers with 1 (leader) producer assigned to each one of them (i.e., $t_c = 6$ and $t_{pc} = 1$).

Fig. 12 shows the total time and peak memory tracked for three testing plugins. We also reuse the results obtained with a single consumer thread (sequential computation) in Sec. 6.2.2 as a reference.

In general, using the maximum number of consumer threads achieves the best time performance. Since the memory usages are at a similar level (within 3% difference) except the `DiscreteGradient` plugin (within 10% difference), we only discuss the time performance here and shared the detailed figures in supplemental materials.

The `TestTopoRelations` plugin only extracts the topological relations from the mesh without computing any additional information, the precomputation from five producer threads helps speed up the execution, but it is still on average 5% slower than fully distributing the task to consumers. With one more consumer thread, it is 1.7 times as fast as the single consumer thread.

The `ScalarFieldCriticalPoints` and `DiscreteGradient` plugins show the advantage of maximizing the number of consumer threads over other cases, and it is on average 2.4 times as fast as maximizing the number of producer threads for both plugins.

Lessons learned. If the parallel processing algorithm is intensive with topological relations, i.e., the use of topological relations is faster than the production, both maximizing the consumer and producer threads will speed up the execution. Otherwise, it is preferred to increase the number of consumer threads to distribute the workload instead of increasing the number of producer threads for precomputation. Even though more work producers can reduce the waiting time of the consumer, more consumer threads can make the algorithm proceed in parallel and thus reduce the total time spent by the algorithm.

7.2 Comparing with state-of-the-art

In this section, we repeat our comparison with CompacTriangulation and ExplicitTriangulation data structures, allowing testing plugins to run in parallel. In the following experiments, all data structures use 12 threads for their initialization (preprocessing). CompacTriangulation and ExplicitTriangulation use 12 threads to run the requested algorithm. Our data structure uses 6 consumer threads supported by 1 leader producer each (i.e., $t_c = 6$ and $t_{pc} = 1$).

Fig. 13 shows the total time and peak memory consumption. Notice that all algorithms are parallelized in a way that is agnostic to data structures (using OpenMP [12]). Specifically, for loops that iterate over simplices are split and distributed across threads. This is a challenge for block-based data structures like CompacTriangulation and ACTOPO. Suppose that the simplices indexed by a block are distributed between two threads, topological relations of that block will be computed and stored by each thread.

The resulting loss of performance is clearly visible in the results of `TestTopoRelations` (Fig. 13(a)), where ACTOPO is about 1.2

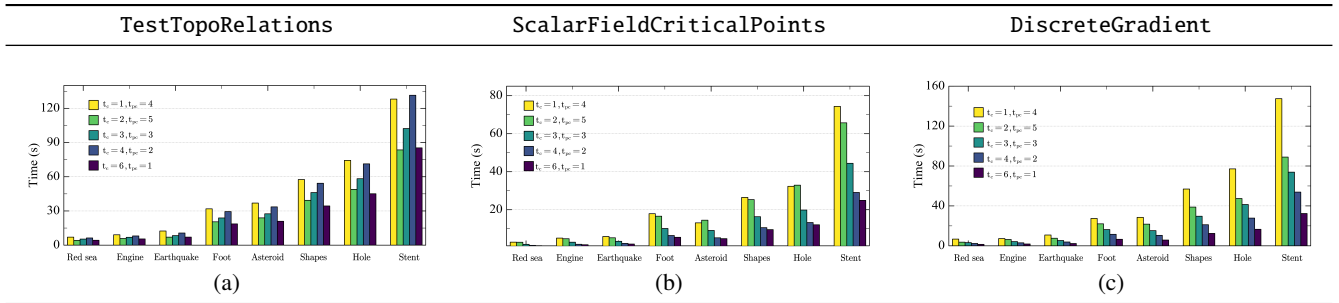| TestTopoRelations	ScalarFieldCriticalPoints	DiscreteGradient

Fig. 12: The total time (in seconds) used by three experimental plugins with different numbers of consumer and producer threads when the algorithm runs in parallel with OpenMP.
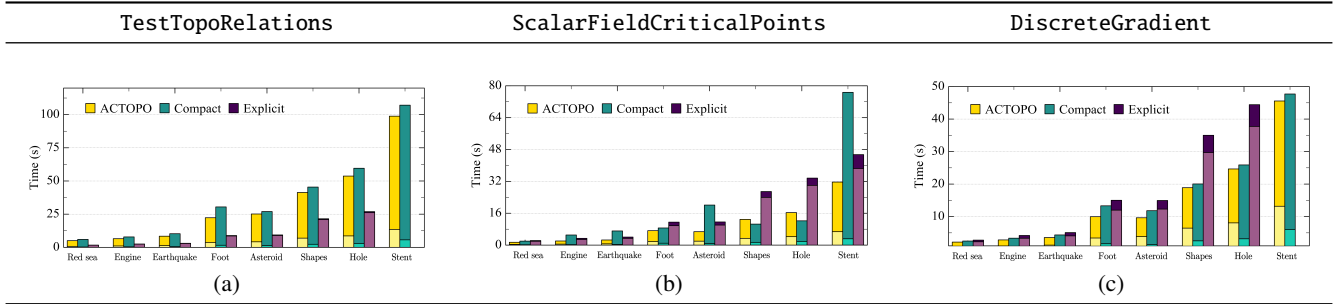
TestTopoRelations	ScalarFieldCriticalPoints	DiscreteGradient

Fig. 13: The total time (in seconds) used by three experimental plugins with different data structures: ACTOPO, TTK Compact Triangulation, and TTK Explicit Triangulation when the algorithm runs in parallel with OpenMP. The top bar indicates the time required by running the algorithm, while the bottom bar indicates the preprocessing time.

times as fast as CompactTriangulation but 2.5 times as slow as Explicit-Triangulation. However, this testing plugin already demonstrates the key advantage of block-based data structures, which is to speed up the preprocessing stage. During this step, ExplicitTriangulation is about 2.5 times as slow as ACTOPO. This advantage becomes fundamental depending on the topological relations required by the algorithm. For real plugins (see Fig. 13(b) and (c)), Explicit Triangulation is always the slowest data structure, being 4.8 times as slow as ACTOPO at the preprocessing stage and requires 1.6 times to complete the execution on average.

ACTOPO is generally the fastest data structure performing on par with CompactTriangulation. Exceptions are the Foot and Stent dataset, where CompactTriangulation performs sensibly worse than ACTOPO. This is because these two datasets contain many more critical points than other datasets, and CompactTriangulation is slower in computing boundary vertices than our data structure, as it needs to recompute triangle list for the same block multiple times while ACTOPO already has those information encoded at initialization time.

This is an impressive result if we consider that, with ACTOPO, the algorithm's instructions are executed by half the number of threads (6 instead of 12). This demonstrates the advantage that a task-parallel data structure can provide for unstructured mesh processing, leading to competitive time performance and efficient memory usage. ACTOPO constantly performs on par with CompactTriangulation using only 2% more memory for `ScalarFieldCriticalPoints` and `TestTopoRelations`, and 8% less memory for `DiscreteGradient`. Notably, both data structures are at least 2 times as compact as ExplicitTriangulation. Given that using parallel algorithm has very limited effects on the peak memory usage, we provide detailed figures in supplemental materials.

8 CONCLUSION

In this paper, we present a new block-based task-parallel approach that integrates a producer-consumer paradigm to improve data structures' performance. A concrete implementation of this approach, called Accelerated Clustered Topological (ACTOPO) data structure, was also introduced and evaluated. In our experimental evaluation, we test ACTOPO in two different scenarios. With sequential algorithms,

ACTOPO provides a substantial speedup compared to state-of-the-art block-based data structures (TopoCluster [40]). It provides timings comparable to static data structures (ExplicitTriangulation [53]), using only half the memory. With parallel algorithms, ACTOPO maintains its compactness while providing the best time performance in most cases, when compared to other data structures.

Limitations. The main limitation of ACTOPO is the spatial buffer. Our evaluation has shown that this type of buffer does not allow to completely eliminate the waiting time of the consumer thread when running the `MorseSmaleComplex` plugin. Although the implementation of spatial buffer helps reduce the overall waiting time compared to the linear order, ACTOPO still faces difficulties in handling a processing algorithm exhibiting non-linear access patterns.

The main objective of ACTOPO is to provide an algorithm-agnostic interface for mesh processing. One possible direction for our future work is to explore learning-based methods to guide the producers in selecting the blocks to precompute. This may further reduce the waiting time for the consumer thread and result in faster algorithm execution. Another direction is to refine the block-based task-parallel model for specific algorithms. This will allow the development of new buffering strategies tailored to an algorithm's access patterns.

SUPPLEMENTAL MATERIALS

All supplemental materials are available on OSF at https://osf.io/b3d6k/, released under a CC BY 4.0 license. In particular, they include (1) Excel files containing the results obtained during experiments, (2) pseudocode for computing a representative set of localized topological relations with ACTOPO, and (3) figure images geenrated from the result data. The source code of ACTOPO data structure, including the plugins we used during the experiments, can be found at the same URL, released under BSD License.

REFERENCES

[1] H. Azpúrua, M. F. M. Campos, and D. G. Macharet. Three-dimensional terrain aware autonomous exploration for subterranean and confined spaces. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2443–2449. IEEE, Xi'an, China, 2021. doi: 10.1109/ICRA48506.2021.9561099 1

[2] T. F. Banchoff. Critical points and curvature for embedded polyhedral surfaces. *The American Mathematical Monthly*, 77(5):475–485, 1970. doi: 10.1080/00029890.1970.11992523 2

[3] X. Bao, N. Karthikeyan, U. D. Schiller, and F. Iuricich. Application-oriented analysis of material interface reconstruction algorithms in time-varying Bijel simulations. In *EuroVis 2022 - Short Papers*. The Eurographics Association, Rome, Italy, 2022. doi: 10.2312/evs.20221104 1

[4] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '97, pp. 360–369. Society for Industrial and Applied Mathematics, New Orleans, LA, USA, Jan. 1997. 2

[5] H. Bhatia, A. Gyulassy, V. Lordi, J. Pask, V. Pascucci, and P.-T. Bremer. TopoMS : Comprehensive topological exploration for molecular and condensed-matter systems: Comprehensive topological exploration for molecular and condensed-matter systems. *Journal of Computational Chemistry*, 39:936–952, Mar. 2018. doi: 10.1002/jcc.25181 2

[6] J.-D. Boissonnat and C. Maria. The Simplex tree: An efficient data structure for general simplicial complexes. *Algorithmica*, 70(3):406–427, 2014. doi: 10.1007/s00453-014-9887-3 2

[7] D. Canino and L. De Floriani. Representing simplicial complexes with Mangroves. In *Proceedings of the 22nd International Meshing Roundtable*, pp. 465–483. Springer, 2014. doi: 10.1007/978-3-319-02335-9_26 2

[8] D. Canino, L. De Floriani, and K. Weiss. IA*: An adjacency-based representation for non-manifold simplicial shapes in arbitrary dimensions. *Computers & Graphics*, 35(3):747–753, 2011. doi: 10.1016/j.cag.2011.03.009 2

[9] H. A. Carr, O. Rübel, and G. H. Weber. Distributed hierarchical contour trees. In *2022 IEEE 12th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 1–10. IEEE, Oklahoma City, OK, USA, 2022. doi: 10.1109/LDAV57265.2022.9966394 2

[10] H. A. Carr, G. H. Weber, C. M. Sewell, and J. P. Ahrens. Parallel peak pruning for scalable SMP contour tree computation. In *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 75–84. IEEE, Baltimore, MD, USA, 2016. doi: 10.1109/LDAV.2016.7874312 2

[11] H. A. Carr, G. H. Weber, C. M. Sewell, O. Rübel, P. Fasel, and J. P. Ahrens. Scalable contour tree computation by data parallel peak pruning. *IEEE Transactions on Visualization and Computer Graphics*, 27(4):2437–2454, 2021. doi: 10.1109/TVCG.2019.2948616 2

[12] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. 8

[13] A. L. Codd and L. Gross. Three-dimensional inversion for sparse potential data using first-order system least squares with application to gravity anomalies in Western Queensland. *Geophysical Journal International*, 227(3):2095–2120, 2021. doi: 10.1093/gji/ggab323 1

[14] A. Dai and M. Nießner. Scan2mesh: From unstructured range scans to 3D meshes. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5569–5578. IEEE, Long Beach, CA, USA, 2019. doi: 10.1109/CVPR.2019.00572 1

[15] L. De Floriani, U. Fugacci, F. Iuricich, and P. Magillo. Morse complexes for shape segmentation and homological analysis: Discrete models and algorithms. *Computer Graphics Forum*, 34(2):761–785, 2015. doi: 10.1111/cgf.12596 2

[16] L. De Floriani, D. Greenfieldboyce, and A. Hui. A data structure for non-manifold simplicial d-complexes. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pp. 83–92. ACM, Nice, France, 2004. doi: 10.1145/1057432.1057444 2

[17] L. De Floriani, A. Hui, D. Panozzo, and D. Canino. A dimension-independent data structure for simplicial complexes. *Proceedings of the 19th International Meshing Roundtable*, pp. 403–420, 2010. doi: 10.1007/978-3-642-15414-0_24 2

[18] H. Edelsbrunner. *Algorithms in combinatorial geometry*, vol. 10. Springer Verlag, 1987. doi: 10.1007/978-3-642-61568-9 2

[19] H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci. Morse-smale complexes for piecewise linear 3-manifolds. In *Proceedings of the 19th Annual Symposium on Computational Geometry*, SCG '03, pp. 361–370. ACM, New York, NY, USA, June 2003. doi: 10.1145/777792.777846 4

[20] R. Fellegara, K. Weiss, and L. De Floriani. The Stellar decomposition: A compact representation for simplicial complexes and beyond. *Computers & Graphics*, 98:322–343, Aug. 2021. doi: 10.1016/j.cag.2021.05.002 1, 2, 3, 4, 5

[21] R. Forman. A user's guide to discrete morse theory. *Séminaire Lotharingien de Combinatoire*, 48:B48c, 35 p., 2002. 5

[22] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Contour forests: Fast multi-threaded augmented contour trees. In *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 85–92. IEEE, Baltimore, MD, USA, 2016. doi: 10.1109/LDAV.2016.7874333 2

[23] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Task-based augmented contour trees with Fibonacci heaps. *IEEE Transactions on Parallel and Distributed Systems*, 30(8):1889–1905, 2019. doi: 10.1109/TPDS.2019.2898436 2

[24] T. Gurung, D. Laney, P. Lindstrom, and J. Rossignac. SQuad: Compact representation for triangle meshes. *Computer Graphics Forum*, 30(2):355–364, 2011. doi: 10.1111/j.1467-8659.2011.01866.x 2

[25] T. Gurung and J. Rossignac. SOT: Compact representation for tetrahedral meshes. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, SPM '09, p. 79–88. Association for Computing Machinery, New York, NY, USA, 2009. doi: 10.1145/1629255.1629266 2

[26] A. Gyulassy, P.-T. Bremer, B. Hamann, and V. Pascucci. A practical approach to Morse-Smale complex computation: Scalability and generality. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1619–1626, 2008. doi: 10.1109/TVCG.2008.110 2

[27] A. Gyulassy, P.-T. Bremer, and V. Pascucci. Computing Morse-Smale complexes with accurate geometry. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2014–2022, 2012. doi: 10.1109/TVCG.2012.209 2

[28] A. Gyulassy, P.-T. Bremer, and V. Pascucci. Shared-memory parallel computation of Morse-Smale complexes with improved accuracy. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):1183–1192, 2019. doi: 10.1109/TVCG.2018.2864848 2

[29] A. Gyulassy, D. Günther, J. A. Levine, J. Tierny, and V. Pascucci. Conforming Morse-Smale complexes. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2595–2603, 2014. doi: 10.1109/TVCG.2014.2346434 2

[30] A. Gyulassy, V. Pascucci, T. Peterka, and R. Ross. The parallel computation of Morse-Smale complexes. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 484–495. IEEE, Shanghai, China, 2012. doi: 10.1109/IPDPS.2012.52 2

[31] C. Heine, H. Leitte, M. Hlawitschka, F. Iuricich, L. De Floriani, G. Scheuermann, H. Hagen, and C. Garth. A survey of topology-based methods in visualization. *Computer Graphics Forum*, 35(3):643–667, June 2016. doi: 10.1111/cgf.12933 2

[32] A. Heinecke, A. Breuer, S. Rettenberger, M. Bader, A.-A. Gabriel, C. Pelties, A. Bode, W. Barth, X.-K. Liao, K. Vaidyanathan, M. Smelyanskiy, and P. Dubey. Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 3–14. IEEE, New Orleans, LA, USA, 2014. doi: 10.1109/SC.2014.6 1

[33] R. C. Hilzer. Synchronization of the producer/consumer problem using semaphores, monitors, and the ada rendezvous. *ACM SIGOPS Operating Systems Review*, 26(3):31–39, July 1992. doi: 10.1145/130888.130891 3

[34] H. Hu and P. Yadmellat. Spatial constraint generation for motion planning in dynamic environments. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 384–389. IEEE, Prague, Czech Republic, 2021. doi: 10.1109/IROS51168.2021.9636067 1

[35] X. Huang, P. Klacansky, S. Petruzza, A. Gyulassy, P.-T. Bremer, and V. Pascucci. Distributed merge forest: A new fast and scalable approach for topological analysis at scale. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, p. 367–377. ACM, New York, NY, USA, 2021. doi: 10.1145/3447818.3460358 2

[36] C. Kessler and J. Keller. Models for parallel computing : Review and

perspectives. *Mitteilungen - Gesellschaft für Informatik e.V., Parallel-Algorithmen und Rechnerstrukturen*, 24:13–29, 2007. 1

[37] P. Klacansky, A. Gyulassy, P.-T. Bremer, and V. Pascucci. Toward localized topological data structures: Querying the forest for the tree. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):173–183, 2020. doi: 10.1109/TVCG.2019.2934257 2

[38] M. Kremer, D. Bommes, and L. Kobbelt. OpenVolumeMesh – a versatile index-based data structure for 3D polytopal complexes. In X. Jiao and J.-C. Weill, eds., *Proceedings of the 21st International Meshing Roundtable*, pp. 531–548. Springer, Berlin, Heidelberg, 2013. doi: 10.1007/978-3-642-33573-0_31 2

[39] C. Lawson. Software for C^1 surface interpolation. In J. R. Rice, ed., *Mathematical Software*, pp. 161–194. Academic Press, 1977. doi: 10.1016/B978-0-12-587260-7.50011-X 2

[40] G. Liu, F. Iuricich, R. Fellegara, and L. De Floriani. TopoCluster: A localized data structure for topology-based visualization. *IEEE Transactions on Visualization and Computer Graphics*, 29(2):1506–1517, 2023. doi: 10.1109/TVCG.2021.3121229 1, 2, 3, 4, 5, 6, 7, 9

[41] M. Luffel, T. Gurung, P. Lindstrom, and J. Rossignac. Grouper: A compact, streamable triangle mesh data structure. *IEEE Transactions on Visualization and Computer Graphics*, 20(1):84–98, 2014. doi: 10.1109/TVCG.2013.81 2

[42] N. Morrical, W. Usher, I. Wald, and V. Pascucci. Efficient space skipping and adaptive sampling of unstructured volumes using hardware accelerated ray tracing. In *2019 IEEE Visualization Conference (VIS)*, pp. 256–260. IEEE, Vancouver, BC, Canada, 2019. doi: 10.1109/VISUAL.2019.8933539 1

[43] G. M. Nielson. Tools for triangulations and tetrahedralizations and constructing functions defined over them. In G. M. Nielson, H. Hagen, and H. Müller, eds., *Scientific Visualization: overviews, Methodologies and Techniques*, chap. 20, pp. 429–525. IEEE Computer Society, Silver Spring, MD, 1997. 2

[44] A. Paoluzzi, F. Bernardini, C. Cattani, and V. Ferrucci. Dimension-independent modeling with simplicial complexes. *ACM Transactions on Graphics (TOG)*, 12(1):56–102, 1993. doi: 10.1145/169728.169719 2

[45] T. Peterka, R. Ross, A. Gyulassy, V. Pascucci, W. Kendall, H.-W. Shen, T.-Y. Lee, and A. Chaudhuri. Scalable parallel building blocks for custom data analysis. In *2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 105–112. IEEE, Providence, RI, USA, 2011. doi: 10.1109/LDAV.2011.6092324 2

[46] M. Pont, J. Vidal, J. Delon, and J. Tierny. Wasserstein distances, geodesics and barycenters of merge trees. *IEEE Transactions on Visualization and Computer Graphics*, 28(01):291–301, Jan. 2022. doi: 10.1109/TVCG.2021.3114839 10

[47] V. Robins, P. J. Wood, and A. P. Sheppard. Theory and algorithms for constructing discrete morse complexes from grayscale digital images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(8):1646–1658, 2011. doi: 10.1109/TPAMI.2011.95 2

[48] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. 2

[49] S. Sathar, M. L. Trew, and L. K. Cheng. Tissue specific simulations of interstitial cells of cajal networks using unstructured meshes. In *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pp. 8062–8065. IEEE, Milan, Italy, 2015. doi: 10.1109/EMBC.2015.7320264 1

[50] N. Shivashankar and V. Natarajan. Parallel computation of 3D Morse-Smale complexes. *Computer Graphics Forum*, 31(3pt1):965–974, June 2012. doi: 10.1111/j.1467-8659.2012.03089.x 2, 5

[51] D. Shulga, O. Morozov, and P. Hunziker. A tensor B-spline approach for solving the diffusion pde with application to optical diffusion tomography. *IEEE Transactions on Medical Imaging*, 36(4):972–982, 2017. doi: 10.1109/TMI.2016.2641500 1

[52] V. Subhash, K. Pandey, and V. Natarajan. GPU parallel computation of Morse-Smale complexes. In *2020 IEEE Visualization Conference (VIS)*, pp. 36–40. IEEE, Salt Lake City, UT, USA, 2020. doi: 10.1109/VIS47514.2020.00014 3

[53] J. Tierny, G. Favelier, J. A. Levine, C. Gueunet, and M. Michaux. The Topology ToolKit. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):832–842, 2018. doi: 10.1109/TVCG.2017.2743938 1, 2, 9

[54] I. Wald, N. Morrical, and S. Zellmann. A memory efficient encoding for

ray tracing large unstructured data. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):583–592, 2022. doi: 10.1109/TVCG.2021.3114869 2

[55] K. Weiss, L. De Floriani, R. Fellegara, and M. Velloso. The PR-star octree: a spatio-topological data structure for tetrahedral meshes. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '11, pp. 92–101. ACM, ACM, New York, NY, USA, 2011. doi: 10.1145/2093973.2093987 1, 2, 3, 4

[56] X. Xu, F. Iuricich, K. Calders, J. Armston, and L. De Floriani. Topology-based individual tree segmentation for automated processing of terrestrial laser scanning point clouds. *International Journal of Applied Earth Observation and Geoinformation*, 116:103145, 2023. doi: 10.1016/j.jag.2022.103145 1

[57] L. Yan, T. B. Masood, R. Sridharamurthy, F. Rasheed, V. Natarajan, I. Hotz, and B. Wang. Scalar field comparison with topological descriptors: Properties and applications for scientific visualization. *Computer Graphics Forum*, 40(3):599–633, 2021. doi: 10.1111/cgf.14331 2

[58] X. Zhao, X. Yu, M. Qiu, F. Qing, and S. Zou. An arbitrary Lagrangian-Eulerian RKDG method for multi-material flows on adaptive unstructured meshes. *Computers & Fluids*, 207:104589, 2020. doi: 10.1016/j.compfluid.2020.104589 1

[59] A. Şahıstan, S. Demirci, N. Morrical, S. Zellmann, A. Aman, I. Wald, and U. Güdükbay. Ray-traced shell traversal of tetrahedral meshes for direct volume visualization. In *2021 IEEE Visualization Conference (VIS)*, pp. 91–95. IEEE, New Orleans, LA, USA, 2021. doi: 10.1109/VIS49827.2021.9623298 1