Efficient representation and analysis for a large tetrahedral mesh using Apache Spark

Yuehui Qian* University of Maryland, College Park
Guoxi Liu† Clemson University
Federico Iuricich‡ Clemson University
Leila De Floriani§ University of Maryland, College Park

ABSTRACT

Tetrahedral meshes are widely used due to their flexibility and adaptability in representing changes of complex geometries and topology. However, most existing data structures struggle to efficiently encode the irregular connectivity of tetrahedral meshes with billions of vertices.

We address this problem by proposing a novel framework for efficient and scalable analysis of large tetrahedral meshes using Apache Spark. The proposed framework, called *Tetra-Spark*, features optimized approaches to locally compute many connectivity relations by first retrieving the *Vertex-Tetrahedron (VT)* relation. This strategy significantly improves Tetra-Spark's efficiency in performing morphology computations on large tetrahedral meshes.

To prove the effectiveness and scalability of such a framework, we conduct a comprehensive comparison against a vanilla Spark implementation for the analysis of tetrahedral meshes. Our experimental evaluation shows that Tetra-Spark achieves up to a 78× speedup and reduces memory usage by up to 80% when retrieving connectivity relations with the VT relation available. This optimized design further accelerates subsequent morphology computations, resulting in up to a 47.7× speedup.

Index Terms: Data representation, connectivity relation, topological data analysis, Apache Spark.

1 INTRODUCTION

Tetrahedral meshes have found many applications in computational science and engineering due to their flexibility and adaptability in representing complex geometries [26, 36, 56, 44]. Unlike regular grids, tetrahedral meshes do not require the underlying domain to conform to a regular cube [49]. This helps eliminate ambiguities and simplifies the management of degeneracies in feature extraction and tracking [23]. One example application is in computational fluid dynamics, where tetrahedral meshes facilitate efficient and precise simulations of airflow around aircraft [8].

Despite their widespread adoption, processing this type of data still represents a major bottleneck in the analysis pipeline [7, 27, 53]. The primary challenges stem from the substantial memory requirements to compute and store the mesh connectivity when compared to regular data [34].

A common solution to address these challenges is to scale up the computational power by relying on distributed environments. Within large computational clusters, tetrahedral meshes can be distributed and processed, taking advantage of multiple CPU cores and extensive memory space [13, 31]. However, this approach comes with a cost in terms of usability. Since communication costs between cluster nodes are typically non-negligible, algorithms and data structures for processing tetrahedral meshes have to be redesigned from scratch. This redesign must consider not only the computational efficiency of the algorithm itself but also the distribution of data across cluster nodes.

To mitigate this issue, we investigate the use of a self-managing distributed computing system, *Apache Spark* [59]. Apache Spark is an in-memory distributed computing framework that can distribute computations across multiple nodes. By leveraging implicit data parallelism [1, 60], it eliminates the need for users to explicitly manage and design Map-Reduce diagrams for distributed computation. By abstracting away the underlying complexity of task distribution and fault tolerance, Spark enables users to quickly develop scalable applications without requiring in-depth knowledge of the intricacies of distributed systems.

In this work, we propose a novel framework for tetrahedral mesh representation and analysis in Apache Spark, named *Tetra-Spark*. Tetra-Spark incorporates optimized techniques for encoding a tetrahedral mesh and computing its connectivity relations. By first retrieving the *Vertex-Tetrahedron (VT)* relation (i.e., the set of tetrahedra incident to a given vertex), the framework enables embarrassingly parallel computation of various connectivity relations. This optimized approach for locally deriving relations significantly accelerates the computation of topological features. Specifically, our contributions include: (1) a minimal representation for encoding a tetrahedral mesh with scalar fields defined on it based on Spark *DataFrames* [1]; (2) new algorithms for deriving the connectivity relations of a tetrahedral mesh using DataFrame operations; (3) optimized algorithms to locally compute groups of connectivity relations using Spark *User Defined Functions*; (4) an evaluation of the effectiveness and scalability of our new algorithms across various cluster configurations and applications; and (5) an open-source Python implementation of our proposed framework[1].

2 BACKGROUND

2.1 Simplicial complex

A k-simplex is defined as the convex hull of $k+1$ linearly independent points in Euclidean space. When considering a simplex σ of dimension k, the convex hull of any nonempty subset of these $k+1$ points, consisting of $m+1$ points where $m < k$, forms an m-simplex τ. The simplex τ is known as an *m-face* of σ, and conversely, σ is a coface of τ. The collection of all cofaces of a simplex σ is called the *star* of σ. In general, 0-faces are called *vertices*, 1-faces are *edges*, and $(n-1)$-faces are termed *facets*.

A simplicial complex Σ is composed of a set of k-simplicies ($0 \leq k \leq d$), such that the face of every simplex σ is also included in Σ, and the intersection between any two simplices σ and τ is either a face common to both or is empty. A d-simplex is also called a *top simplex* if it is not a face of any other simplex in Σ. In this paper, we consider simplicial complexes up to dimension three ($d = 3$), and we refer to them as *tetrahedral meshes*.

Connectivity relations describe how simplices in a simplicial complex are "glued" together. There are three major categories of connectivity relations: a *boundary* relation maps a simplex to its faces, a *coboundary* relation maps a simplex to its cofaces, and an

*e-mail: yhqian@umd.edu
†e-mail: guoxil@clemson.edu
‡e-mail: fiurici@clemson.edu
§e-mail: deflo@umd.edu

[1] https://github.com/qyh-sunshine/Tetra_Spark

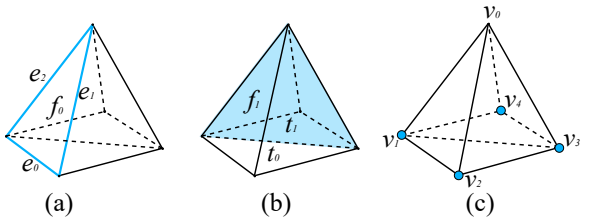


Figure 1: The (a) FE relation for the triangle f_0, (b) FT relation for the triangle f_1, and (c) VV relation for the vertex v_0 in Σ.

adjacency relation maps a simplex to neighboring simplices of the same dimension. Suppose two simplices σ and τ are in Σ, and σ is a face of τ: σ is on the *boundary* of τ, τ is on the *coboundary* of σ. Two k-simplices ($k > 0$) 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 there exists an edge connecting them.

In this paper, we focus on connectivity relations within a tetrahedral mesh and use capital letters to indicate whether the relation involves a vertex (V), edge (E), triangle (F), or tetrahedron (T). Each connectivity relation is denoted as a pair of letters. Fig. 1 shows three examples of connectivity relations for a mesh Σ consisting of two tetrahedra sharing a common triangular face. Specifically, Fig. 1a shows the FE relation for the triangle f_0, which relates f_0 to edges e_0, e_1, and e_2. Fig. 1b illustrates the FT relation for the triangle f_1, which relates f_1 to tetrahedra that have f_1 as boundary (i.e., t_0 and t_1). Fig. 1c shows the VV relation for the vertex v_0, which relates v_0 to all vertices sharing an edge with v_0 (i.e., v_1, v_2, v_3, and v_4).

2.2 Apache Spark

Apache Spark [59] is a robust in-memory computing framework that introduces a data abstraction called *Resilient Distributed Datasets (RDDs)*. RDDs are collections of read-only objects distributed across multiple machines, optimized for efficient data reuse in parallel operations. While functionally similar to legacy models like MapReduce [12], the in-memory storage of RDDs enables Spark to significantly outperform these models.

Spark provides a user-friendly programming interface for cluster environments [1, 60], which is facilitated by Spark DataFrame. *Spark DataFrame* [1] is an internal API of *Spark SQL*, which simplifies data manipulation through optimized query plans. DataFrames organize data in a structured format, similar to relational database tables. DataFrames in Spark are partitioned and distributed across a cluster, allowing Spark to leverage the full potential of distributed computing. Importantly, data within the same row of a DataFrame will always be stored on the same cluster node [48], enhancing data locality and minimizing the need for extensive data shuffling across the network. Moreover, the Spark engine transparently manages data distribution and task execution, ensuring efficient computation and optimized resource utilization across the cluster, all without requiring user intervention.

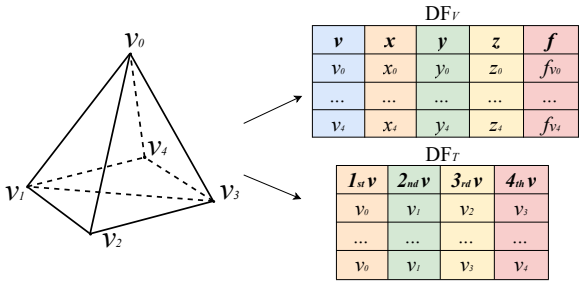
Fig. 2 presents an overview of the DataFrames used to encode a tetrahedral mesh, which can be viewed as tables formed by rows and columns. A detailed description of the information stored in these tables is provided in Sec. 4.

3 Related Work

In this section, we survey existing data structures that support the efficient extraction of connectivity relations on a simplicial complex in shared-memory and distributed environments.

3.1 Data structures for shared-memory systems

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



v	x	y	z	f
v_0	x_0	y_0	z_0	f_{v_0}
...	...	...	...	...
v_4	x_4	y_4	z_4	f_{v_4}

1_{st} v	2_{nd} v	3_{rd} v	4_{th} v
v_0	v_1	v_2	v_3
...	...	...	...
v_0	v_1	v_3	v_4

Figure 2: An example of (a) the DataFrame DF_V, which stores the x, y, and z coordinates plus scalar values f_v of vertices in Σ, and (b) the DataFrame DF_T, which stores four vertices of each tetrahedron.

Static data structures. The approach adopted by static data structures involves computing and storing connectivity relations at initialization time. Variations among these structures lie in the specific types of relations they encode.

The *Incidence graph* [14] is a 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 the memory footprint [5, 10, 11].

The *Simplex tree* [4] avoids encoding boundary relations by organizing all simplices of Σ in a trie [3]. This data structure supports the efficient query of coboundary relations but has limited scalability when working with simplicial complexes in high dimensions [19]. The *Half-edge* data structure [41] is designed for triangle meshes, which reduces the storage costs by only encoding edge-related connectivity relations. Extending this, *Half-faces* [30] adapt the half-edge concept to polyhedral complexes. *Indexed data structures* [32] provide a more compact option by encoding only vertices, top simplices, and the boundary relation from top simplices to their vertices. This structure contains sufficient information to efficiently extract all the boundary relations of cells but requires additional steps for deriving coboundary or adjacency relations.

Several data structures provide efficient access to connectivity relations through adjacency information. Examples include the *Indexed data structure with Adjacencies (IA data structure)* [43, 45] and the *Corner-Table* data structure [46] along with several extensions specifically proposed for triangle meshes [24, 37] and tetrahedral meshes [25]. The *Generalized Indexed data structure with Adjacencies (IA* data structure)* [6] extends the IA data structure to non-manifold simplicial complexes of arbitrary dimension. The IA* data structure has been shown to be the most compact among static topological data structures as the dimension increases [5].

Dynamic data structures. Unlike static data structures, dynamic data structures compute (and discard) connectivity relations at runtime rather than at initialization time. This strategy allows the control of the memory footprint and provides greater scalability.

The *PR-star octree* [54] is considered the first dynamic data structure for tetrahedral meshes. It supports the reconstruction of connectivity information by only encoding the list of tetrahedra incident at each vertex. This data structure uses a PR-Octree decomposition that focuses on the mesh vertices to segment the mesh into subsets. Then, it is capable of extracting any connectivity relation locally to a subset of the mesh.

The *Stellar tree* data structure 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 [19]. 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 [19].

The Stellar decomposition model has also been adopted by the

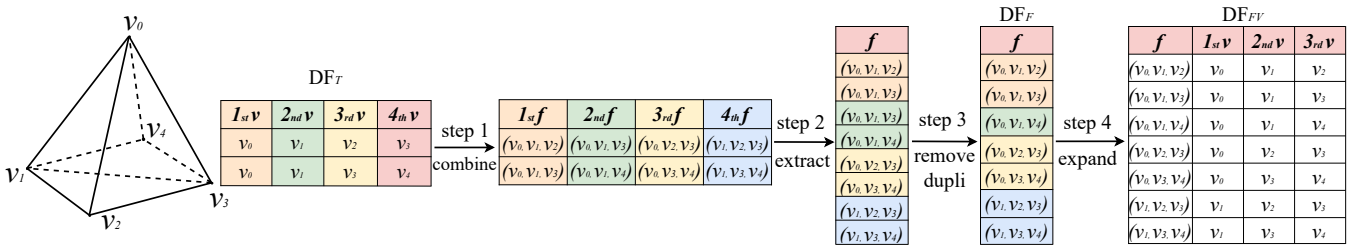


Figure 3: An example to extract the FV relation from DF_T using the *pure global* method.

TopoCluster data structure [35], 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 in a transparent manner for developers. The ease and flexibility of integrating TopoCluster were demonstrated by deploying the data structure in the TTK framework [38, 51], which allowed running any algorithm implemented in the framework out-of-the-box while drastically reducing the memory footprint.

The *Accelerated Clustered TOPOlogical (ACTOPO)* data structure [34] is the first data structure to introduce task-parallelism for connectivity relation computation. ACTOPO assigns different tasks to CPU threads, allowing threads that precompute connectivity relations and those executing the analysis algorithm to run concurrently. Such a task-parallel approach can improve the time performance in both sequential and parallel algorithms while maintaining a low memory footprint of dynamic data structures.

3.2 Data structures for distributed systems

Cluster computing systems are highly effective solutions for handling very large datasets [52, 58, 59]. A common approach employed in distributed systems is to subdivide the mesh into multiple partitions and distribute these partitions across the system. Boundary simplices are duplicated across partitions to allow local computation of connectivity relations and maintain efficiency.

Recently, researchers have developed distributed data structures to support specific topological algorithms [2, 42]. For instance, Bauer et al. [2] developed a data structure designed for encoding the boundary matrix to facilitate distributed computation of persistent homology. As another example, Nigmetov and Morozov [42] proposed a data structure for the distributed computation of merge trees. While efficient for the intended tasks, these data structures are specialized for extracting a particular connectivity relation and are not suitable for general topological algorithms.

A recent advancement is the distributed extension of TTK's triangulation data structure [33], which decomposes the input mesh into multiple disjoint blocks, each managed by a separate process. Each simplex in a block is assigned both a local identifier for internal tasks and a global identifier for inter-process communications. To reduce communication overhead and manage ownership, boundary simplices of each block are duplicated into adjacent blocks with exclusive ownership by the block with the smallest identifier [35].

TTK's distributed triangulation uses the Message Passing Interface (MPI) to handle communications across the cluster's nodes, which poses several challenges for a developer who is responsible for managing data distribution, synchronization, and communication. In this paper, we are interested in developing solutions based on existing frameworks for distributed processing.

Apache Hadoop [52] provides a simplified abstraction for managing complex distributed programs, drawing from the Google MapReduce model [12]. However, it struggles with high disk I/O operations and lacks native support for spatial data. SpatialHadoop [17] enhances the Hadoop framework by specifically addressing the need for geospatial data processing. Nevertheless, it does not support the extraction of connectivity relations in a mesh. Instead, a mesh is represented as a collection of polygons, requiring users to perform spatial joins to identify connections between them.

Apache Spark [59] is an in-memory distributed computing system that provides a platform for general-purpose cluster computing while hiding the details of parallelization, fault tolerance, and data distribution. The in-memory computing paradigm significantly enhances the speed of repeated data access, enabling it to outperform Hadoop by up to 100 times [60]. To efficiently process large spatial datasets, several Spark-based cluster computing systems have been developed, such as Simba [55], SpatialSpark [57], LocationSpark [50], and Apache Sedona [58]. Among them, Apache Sedona [58] stands out as the most widely used one for distributed spatial data processing and analysis. However, these systems treat simplicial complexes merely as general collections of polygons and do not support the computation and encoding of connectivity relations.

4 Encoding a Tetrahedral Mesh in Apache Spark

In this section, we introduce a new distributed data representation of static data structures, which we call *Tetra-Spark*, designed to efficiently encode scalar fields defined on a tetrahedral mesh using Apache Spark.

Design strategy The first strategy of Tetra-Spark aims to minimize the information stored during the loading stage and use memory only as required by an algorithm at runtime. This is achieved by limiting the encoded information only to vertices and tetrahedra of a mesh and to one connectivity relation (e.g., TV). All other simplices and connectivity relations are computed at runtime only if required by an algorithm.

The second strategy is to represent edges, triangles, and tetrahedra as arrays of vertices rather than as indices. This strategy is adopted due to the efficiency it affords by limiting internode communication. All information encoded by Tetra-Spark is contained in Spark DataFrames, which are automatically partitioned and distributed across a cluster of machines in a transparent manner to the user. The only guarantee regarding data distribution is that data within the same row of a DataFrame is stored on the same node of the cluster. By representing these simplices as tuples of vertices, retrieving vertices on the boundary of an edge (EV), triangle (FV), or tetrahedron (TV) becomes a local operation. Using global indexing, instead, would require traversing the entire DataFrame and triggering time-consuming internode communication.

Base encoding. The above design strategies are implemented using two DataFrames, called DF_V and DF_T (see Fig. 2). DF_V stores all vertices of a tetrahedral mesh Σ where each vertex v is characterized by five columns: the index of v, the x, y, and z coordinates, and the scalar value f_v. DF_T encodes the tetrahedra of Σ by storing the connectivity relation TV. Each row stores the indices of four vertices composing a tetrahedron.

The two DataFrames DF_V and DF_T are automatically partitioned and distributed among different nodes in Spark. Each partition includes part of the data, enabling parallel computation on the input mesh. All information encoded in DF_V and DF_T is provided as

input by common formats for mesh or scalar field encoding [47]. Since the number of tetrahedra is typically proportional to the number of vertices in Σ, the computational complexity of loading a mesh is linear to the number of vertices. Starting from DF_V and DF_T, any other connectivity relation can be computed on demand.

5 Computing connectivity relations in Apache Spark

In this section, we describe two strategies for extracting connectivity relations in Tetra-Spark: global methods and local methods. As we will show in Sec. 6, these two strategies have a radically different impact on the performance of the system.

Global methods work on an entire DataFrame by using native operations provided in Apache Spark, such as `groupby()`, `union()`, `explode()`, and `join()`. These operations can always be applied as long as DF_V and DF_T are provided. However, they necessitate data exchanges among different rows of a DataFrame or across several DataFrames, which may lead to inefficiencies.

Local methods offer a more efficient approach by applying *user-defined functions (UDFs)* to a DataFrame. UDFs [1] enable the implementation of custom logic at the row level. They operate by taking one or more columns from a single row as inputs and producing an output column based on the specified logic. Since all required data is contained within the same row and each row is stored on the same node, this method can significantly reduce the need for data redistribution and exchange, resulting in enhanced computational performance and scalability.

We begin this section with a discussion of global methods, followed by local methods. The performance analysis of every method is presented in Sec. 6.2. Due to space constraints, we only focus on a few connectivity relations. A comprehensive description of all connectivity relations is provided in the supplementary material.

5.1 Computing relations with global methods

There are two types of global methods for computing connectivity relations. *Pure global* methods extract connectivity relations starting from the base encoding DF_V and DF_T. *Symmetric global* methods retrieve connectivity relations starting from a symmetric relation (e.g., computing FV given VF).

Boundary relations The FV relation is a boundary relation representing the set of vertices bounding a triangle.

Since triangles are not explicitly encoded in Tetra-Spark, the *pure global* method first extracts all triangles from the base encoding, and then computes the relations between triangles and vertices. These steps are described in Fig. 3. For each tetrahedron in DF_T, four bounding triangles are computed as all possible combinations of three of its four vertices (step 1). By extracting all triangles in a dedicated list (step 2) and removing duplicates (step 3), the DataFrame D_F storing all triangles in Σ is created. Finally, the FV relation is computed by retrieving three vertices of each triangle, creating a new DataFrame DF_{FV} (step 4).

An alternative approach for computing a relation starts from the *symmetric* relation. For example, the FV relation can be computed using its symmetric VF relation (see Fig. 4). Assuming that the VF relation is already encoded in the DataFrame DF_{VF}, the column storing triangles in DF_{VF} is exploded so that each triangle is encoded in a separate row (step 1). Then, duplicate triangle entries are removed (step 2), and the individual FV relation is computed similarly to the pure method.

Coboundary relations The FT relation is an example of a coboundary relation which represents the set of tetrahedra incident to a triangular face.

Using the *pure global* method, the FT relation is computed starting from DF_T. Since only indices of extreme vertices of each tetrahedron are provided by DF_T, we have to use vertex indices to link

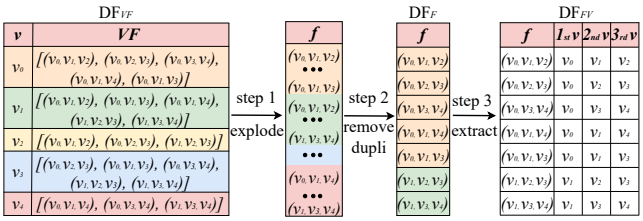


Figure 4: An example to extract the FV relation from its *symmetric* VF relation.

relations together. This process involves extracting FV and VT relations and then joining these relations to derive the FT relation.

We have already described the computation of the FV relation (see Fig. 3). The VT relation can only be extracted globally by utilizing DF_T (see Fig. 5). If we arbitrarily select a vertex (e.g., v_0) as dominant, the tetrahedron (v_0, v_1, v_2, v_3) then represents a partial VT relation for v_0 (step 1). To obtain a full VT relation, four copies of DF_T are created, each with a different vertex designated as dominant (step 2). These copies are subsequently merged into the unified DataFrame, DF_{union} (step 3). The rows in DF_{union} are grouped by the vertex column and consolidated into the DataFrame DF_{VT} (step 4).

Once the FV and VT relations are retrieved, we join the FV and VT relations three times, each time using a different vertex as the key. The FT relation is derived by iterating over the tetrahedra stored in the three VT lists.

The *symmetric global* method, instead, leverages a pre-computed TF relation, which is stored in a DataFrame DF_{TF}. DF_{TF} consists of five columns: a tetrahedron and its four bounding triangles. For each row in DF_{TF}, the tetrahedron represents a partial FT relation for its bounding triangles. To obtain a full FT relation, we create four copies of DF_{TF}, each with a different triangle designated as dominant. These copies are merged into a unified DataFrame DF_{union}. The FT relation is derived by grouping DF_{union} based on the dominant triangle column.

Adjacency relations The VV relation is an example adjacency relation representing the set of vertices that share a common edge with a given vertex.

The VV relation can only be computed from DF_T using a *pure global* method. Similar to the extraction of VT relation depicted in Fig. 5, for each tetrahedron (v_0, v_1, v_2, v_3) in DF_T, if we arbitrarily select one vertex (e.g., v_0) as dominant, the other three vertices denote a partial VV relation for v_0. To obtain a complete VV relation, four copies of DF_T are created, with each copy picking a different vertex as dominant. Within each copy, the other three columns are merged to indicate a partial VV relation. These four copies are then unified to form a new DataFrame DF_{union}, which is subsequently grouped by the dominant vertex column to get a full VV relation.

5.2 Computing relations with local methods

Local methods facilitate the computation of connectivity relations through row-level logic, which is feasible only if specific DataFrames are first computed globally. However, once the required DataFrames have been computed, local methods will utilize embarrassingly parallel routines to extract connectivity relations.

Boundary relations The FV relation can be derived *locally* from a pre-computed VT relation. Assuming that the DataFrame DF_{VT} storing the VT relation consists of two columns: a vertex v, and a list of tetrahedra incident to v (e.g., $VT_{(v)}$). Within each row of DF_{VT}, the objective is to derive the FV relation for all triangles in $VT_{(v)}$ where v is the vertex with the highest index. This can be achieved by applying a UDF to DF_{VT}. Within this UDF, the set of triangles is identified by iterating over the tetrahedra stored

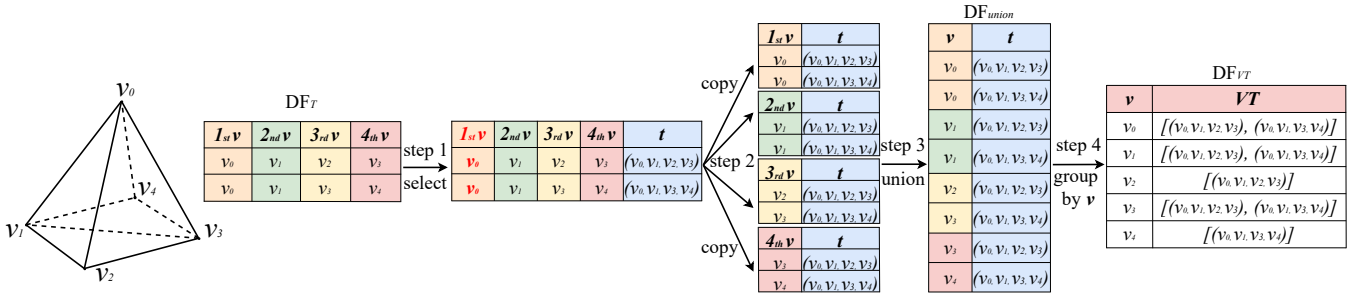


Figure 5: An example to extract the VT relation based on DF_T using the *pure global* method.

in $VT_{(v)}$. The FV relation is then derived by extracting the three extreme vertices from each triangle.

Coboundary relations The *local method* for computing the FT relation leverages the previously extracted VT relation (see Fig. 6). Given a vertex v, it aims to comput the FT relation for triangles in $VT_{(v)}$ where v is the vertex with the highest index. To achieve this, a UDF is applied to DF_{VT} to retrieve the list of triangles incident to v that have the largest index (step 1). Subsequently, another UDF iterates over the triangle list and identifies the coboundary tetrahedra for each triangle in this list (step 2). The resultant DataFrame DF_{FT} is composed of two columns: a vertex v and the FT relation for triangles where v has the highest index.

Adjacency relations The VV relation can also be computed with a *local method* if the VT relation is provided. The VV relation is derived by applying a UDF to DF_{VT}. Within this UDF, we iterate over the tetrahedra incident to a given vertex v, extracting all extreme vertices while excluding v itself.

6 Experimental Evaluation

Local methods are typically more efficient than global methods. However, the effectiveness of localized strategies for computing connectivity relations depends on the prior global computation of a specific relation (e.g., VT). To evaluate the trade-offs between global and local methods, we conduct extensive evaluations focusing on extracting connectivity relations (see Sec. 6.2) and computing topological features (see Sec. 6.3). Additionally, we evaluate the node scalability of Tetra-Spark when more nodes are available in a cluster in Sec. 6.4.

6.1 Experimental settings

In this section, we describe our experimental settings.

Datasets We use six tetrahedral meshes with the number of vertices ranging from about 300K to 1.8 billion. The number of simplices for each mesh is shown in Tab. 1. Among these, four datasets, *Brain, Foot, Stent*, and *Synthetic*, are obtained by thresholding and tetrahedralizing points from volume images. The remaining two datasets, *Lander_small*, and *Lander_huge*, are obtained by tetrahedralizing meshes defined on general polytopes.

System and parameters All experiments are conducted on a cluster equipped with one gateway node, three name nodes, and six worker nodes. The cluster operates using Apache Hadoop 3.0 and Apache Spark 2.4. The gateway node serves as the user interface for submitting Spark jobs and managing cluster resources. The name nodes manage the file system namespace, maintain the metadata for all files and directories, and regulate clients' access to files. The worker nodes store the actual data in the Hadoop file system and execute the actual computational tasks. The gateway node boasts 20 processor cores and 256 GB of RAM, while each of the three name nodes has 16 processing cores and 128 GB of RAM. Additionally, each of the six worker nodes is equipped with 28 processing cores (168 physical cores in total) and 768 GB of RAM (4.6 TB in total).

Table 1: Overview of experimental datasets. We list the number of vertices $|\Sigma_V|$, edges $|\Sigma_E|$, triangles $|\Sigma_F|$, and tetrahedra $|\Sigma_T|$ for each tetrahedral mesh. *Regular* means that the mesh is tetrahedralized from regular volume images. *Irregular* means the mesh is tetrahedralized from irregularly distributed points in polytopes.

Dataset	Type	$\lvert\Sigma_V\rvert$	$\lvert\Sigma_E\rvert$	$\lvert\Sigma_F\rvert$	$\lvert\Sigma_T\rvert$
Brain	Regular	0.3 M	2.1 M	3.6 M	1.8 M
Foot	Regular	5.9 M	40.7 M	69.5 M	34.7 M
Stent	Regular	17 M	118.8 M	201.4 M	99.9 M
Synthetic	Regular	118.7 M	747.3 M	1.2 B	588.7 M
Lander_small	Irregular	224.9 M	1.4 B	2.4 B	1.2 B
Lander_huge	Irregular	1.8 B	11.5 B	19.4 B	9.7 B

We have performed preliminary experiments to determine the impact of various parameters on execution performance, considering the fixed number of physical cores and available memory. Our pilot tests focus on computing the VV relation using the *pure global* method with the largest dataset *Lander_huge*. We varied the number of executors, ranging from 1 to 64, the number of cores per executor from 1 to 20, and the memory allocated per executor, from 4 GB to 100 GB. The results, shown in the supplementary material, indicate that the cluster performs well when using 64 executors, with each executor equipped with 5 cores and 64 GB of memory plus an 8 GB overhead, thereby fully utilizing the cluster's total memory capacity of 4.6 TB.

In assessing the performance of these data structures, we focus on execution time and peak memory usage. Execution time refers to the total time required to complete a specific task. Peak memory usage, on the other hand, is more involved. In Spark, memory usage can be categorized into two main types [59]: execution and storage. *Execution memory* is the memory used at runtime by computational operations such as shuffles and joins. *Storage memory*, instead, is used for caching and distributing internal data throughout the cluster. In our evaluation, we report the peak total memory, defined as the sum of execution and storage memory.

6.2 Computing connectivity relations

In this section, we evaluate the performance of different methods used to extract each connectivity relation. Due to space constraints, we present results only for the relations discussed in Sec. 5, which are also relevant to the computation of the topological features discussed in Sec. 6.3. In the supplementary material, we provide performance evaluations for all connectivity relations.

For each relation, we evaluate the time and memory consumption using three methods: *pure global*, *symmetric global*, and *local*. In the *pure global* method, we report the time and memory required to compute the desired relation starting from the initial DataFrame DF_T. For the *symmetric global* and *local* methods, we measure the time needed to extract the pre-computed relation (e.g., VT) from

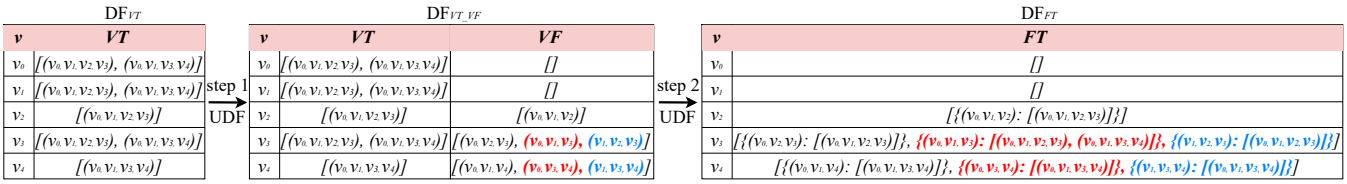


Figure 6: An example to *locally* extract the FT relation from the pre-computed VT relation.

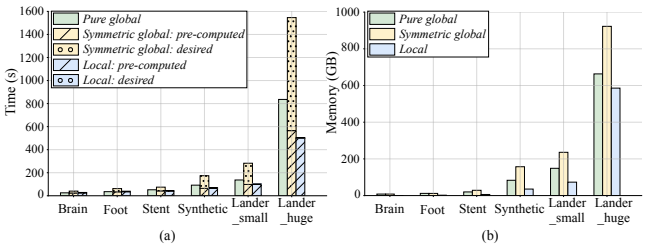


Figure 7: (a) time and (b) memory costs of deriving FV relation.

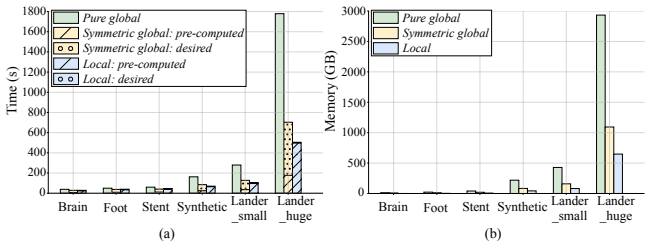


Figure 8: (a) time and (b) memory costs of computing FT relation.

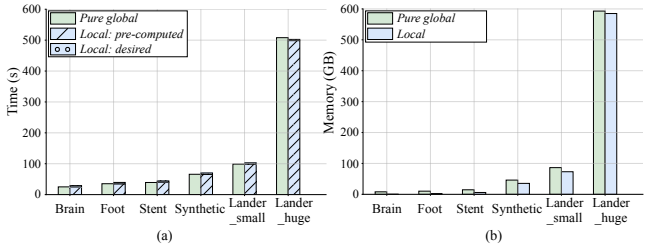


Figure 9: (a) time and (b) memory cost of deriving VV relation.

DF_T, and the time required to derive the desired relation from this pre-retrieved relation. For instance, when assessing the FV relation using the *symmetric global* method, we detail both the time to compute VF from DF_T and the time to derive FV from VF. Similarly, in the *local* method, we specify the time for computing VT from DF_T and the time for locally deriving FV from VT.

6.2.1 Boundary relations

The time and memory required for extracting the FV relation are shown in Fig. 7, where the slashed bar denotes the time taken to derive the pre-computed relation and the dotted bar denotes the time required for extracting the desired relation after preparing the pre-computed one.

The *local* method outperforms both the *pure global* and *symmetric global* methods, achieving speedups of up to 1.7× and 3.1× (see Fig. 7a), and reducing memory usage by up to 57% and 69% (see Fig. 7b), respectively. We recall that the *pure global* method requires global DataFrame operations to gather and deduplicate triangle entries, which is why it underperforms the localized strategy. The inefficiency of the *symmetric global* method mainly stems from its *explode* operation, which generates a new row for each triangle in the $VF_{(v)}$ list, causing extensive data movement and reduced performance.

If we assume the pre-computed relation has already been retrieved in the system and focus solely on the timings for extracting the desired relation from it, the *local* method offers speedups of up to 42× and 50× over the *pure global* and *symmetric global* methods, respectively. Such superior efficiency of the *local* method is primarily attributed to the pre-computed VT relation, which eliminates the need to retrieve triangles from different partitions, significantly reducing internode communication.

6.2.2 Coboundary relations

The time and memory consumption for computing the FT relation are displayed in Fig. 8. As described above, the time to derive the pre-computed relation is denoted with slashed bars, and the time to compute the desired relation is illustrated using dotted bars.

The *local* method still performs the best among all three approaches, achieving up to 3.5× and 1.4× speedups (see Fig. 8a) and saving up to 80% and 48% memory (see Fig. 8b) compared to the *pure global* and *symmetric global* approaches, respectively. Such great advantage stems from the localized strategy of deriving FT from VT in the *local* method, allowing operations to be completed within the same row of the DataFrame DF_{VT}, minimizing data movements between partitions.

When considering only the time requirement for extracting the expected relation with the pre-computed relation already available, the *local* method is up to 78× and 24× faster than the *pure global* and *symmetric global* approaches, respectively. This substantial increase in efficiency can be attributed to the embarrassingly parallel nature of the computations in the *local* method, where data exchange between nodes is completely eliminated.

6.2.3 Adjacency relations

Recall that we have two different approaches for computing the VV relation: *pure global* and *local*. The corresponding time and memory costs for each approach are detailed in Fig. 9.

The *local* method exhibits comparable performance to the *pure global* method in terms of total time and memory efficiency. This similarity arises because the *local* method initially relies on an analogous global step to retrieve the VT relation.

When focusing solely on the time needed to derive the VV relation from the already retrieved VT relation, the *local* method outperforms the *pure global* method, providing a speedup of up to 33× (see Fig. 9a). This notable increase in time efficiency comes from the localized strategy, where adjacent vertices are directly retrieved from the VT list encoded in the same row of a DataFrame, and thus no data exchange among nodes is required.

6.3 Computing topological features

The previous section highlights the importance of local methods for efficiently computing connectivity relations. In this section, we compare global and local methods for computing groups of relations in support of real algorithms for topological feature extraction. The objective is to quantify the improvements offered by our optimized approach in retrieving relations for practical applications.

6.3.1 Topological algorithms

We have selected three algorithms for extracting different topological features on a given scalar field: discrete distortion, critical points, and the Forman gradient. Detailed descriptions of these algorithms are provided in the supplementary material.

Discrete distortion The concept of curvature is fundamental to comprehending the geometric and topological characteristics of surfaces [39]. In the context of scalar fields defined on tetrahedral meshes, vertex distortion is a generalization of the concept of concentrated curvature and allows analyzing the local geometry and topology of a three-dimensional manifold. In this work, we approximate vertex distortion using the discrete approach proposed by Mesmoudi et al. [39].

From the perspective of connectivity relations, vertex distortion computation is an example of an algorithm requiring only coboundary relations involving vertices. Specifically, this computation only needs the VT and VF relations.

Critical points Critical points are fundamental topological features for identifying the regions of interest in a scalar field. To classify a vertex v in Σ as critical, we adopt the approach proposed by Edelsbrunner et al. [15], which works on the vertices adjacent to v.

In terms of connectivity relations, computing critical points requires the VT, VF, VV, and TE relations. This algorithm explores the capability of Tetra-Spark to manage both boundary and coboundary relations involving vertices.

Forman gradient The Forman gradient is a fundamental tool rooted in discrete Morse theory [20], which aids the computation of many topological abstractions, such as the Morse complex [40] and the persistent diagram [16]. We compute the Forman gradient using the *coreduction-based* algorithm proposed by Robins et al. [45]. The core idea of this algorithm involves establishing a total order I for the vertices of Σ. The total order will serve as a guiding schema for the subdivision of tetrahedra, triangles, edges, and vertices of Σ into independent sets. This partitioning facilitates the parallel computation of the Forman gradient.

Regarding required connectivity relations, computing the Forman gradient is the most demanding among all algorithms since it involves the VT, VF, VE, EF, and FT relations.

6.3.2 Analysis of results

In this section, we evaluate the performance of computing discrete distortion, critical points, and the Forman gradient. While the implementation of these topological algorithms does not change, the underlying data structures that provide the required connectivity relations work differently. Tetra-Spark always computes connectivity relations using *local* methods after initially deriving the VT relation globally. The alternative implementation, referred to as Vanilla-Spark, uses *global* methods for all connectivity computations.

Discrete distortion computation We recall that computing vertex distortion relies solely on VT and VF relations. As the VT relation is retrieved globally in both Tetra-Spark and Vanilla-Spark, the performance of these two implementations is expected to be similar. This scenario provides a good example to demonstrate the limited effectiveness of Tetra-Spark when only a few connectivity relations are involved for topological algorithms.

The time and memory required for computing vertex distortion in Tetra-Spark and Vanilla-Spark are presented in Fig. 10. Specifically, we outline the time costs for extracting connectivity relations using slashed bars and the time costs for computing vertex distortion after these relations are prepared using dotted bars in Fig. 10a. Fig. 10b illustrates the memory consumption for solely computing the involved relations, while Fig. 10c shows the overall memory usage for the entire algorithm computation.

Tetra-Spark is approximately 1.3× faster than Vanilla-Spark for the entire computation. Specifically, it offers a 1.5× to 2.8× speedup in deriving relations (see the slashed bar chart in Fig. 10a), and about a 1.2× speedup for the actual algorithm computation even though the operations involved are completely the same once the relations are retrieved (see the dotted bar chart in Fig. 10a).

The speedup for computing relations in Tetra-Spark is attributed to its localized strategy. Conversely, both the VT and VF relations are extracted globally and separately in Vanilla-Spark. Furthermore, Vanilla-Spark requires a join operation to align VT and VF in the same DataFrame row, causing substantial data movements and increased inter-node communication.

The speedup of the actual algorithm computation in Tetra-Spark stems from its memory efficiency in deriving relations. As shown in Fig. 10b, Tera-Spark uses 20% to 46% less memory than Vanilla-Spark for computing relations, allowing more intermediate data to be kept in memory to enhance subsequent processing efficiency through Spark's dynamic memory allocation strategy [60].

Interestingly, the overall peak memory usage for the entire algorithm computation, as shown in Fig. 10c, is consistent across Tetra-Spark and Vanilla-Spark frameworks. This similarity occurs because the largest DataFrames cached in both implementations are the same, which are used to encode VT and VF relations. Furthermore, both implementations employ a global strategy to derive the VT relation. As a result, we observe the same peak memory usage.

Critical points extraction Note that extracting critical points requires multiple connectivity relations. This scenario is a great example to demonstrate the significant improvements offered by the localized strategy in Tetra-Spark for deriving connectivity relations.

The time and memory consumption for extracting critical points using Tera-Spark and Vanilla-Spark are presented in Fig. 11. Similar to the algorithm for computing vertex distortion, we illustrate the time spent on computing connectivity relations using the slashed bars in Fig. 11a, and display the time taken for extracting critical points from these relations using the dotted bars.

Tetra-Spark is 2.2× to 13.1× faster than Vanilla-Spark for the entire computation. Specifically, it provides a 2.5× to 16.5× speedup in computing connectivity relations and a 1.6× to 9.0× speedup in the actual computation of critical points.

The efficiency of relation extraction in Tetra-Spark is primarily attributed to its localized approach for retrieving the VF, VV, and TE relations, coupled with the elimination of join operations to align these relations in the same DataFrame. Moreover, the efficiency gains are more pronounced with larger datasets. This is because the data shuffling associated with joining two DataFrames in Vanilla-Spark escalates with the dataset size, leading to substantial computational overheads.

The speedup of actual algorithm computation in Tetra-Spark derives from its memory efficiency in extracting relations. As shown in Fig. 11b, Tetra-Spark utilizes approximately 60% less memory than Vanilla-Spark for deriving relations. Similar to the findings in distortion calculation, the memory saved allows for more intermediate data to be cached in Spark, thereby improving the time efficiency of subsequent computations.

Regarding the peak memory usage for the entire algorithm, Tetra-Spark consumes between 12.7% and 35.8% less memory than Vanilla-Spark. This efficiency gain, while significant, is not as pronounced as it is for the extraction of connectivity relations alone. This is attributed to the similar storage memory requirements in Tetra-Spark and Vanilla-Spark, as they both cache the largest DataFrame that encodes the VV and TE relations.

Forman gradient computation The time and memory requirements for computing the Forman gradient are displayed in Fig. 12. Similarly, we show the time taken for solely deriving relations with the slashed bars and the time needed for only computing the Forman gradient using the dotted bars in Fig. 12a.

Tetra-Spark achieves an overall speedup ranging from 2.4× to 55.6× compared to Vanilla-Spark. Specifically, it is 3.0× to 61.5×

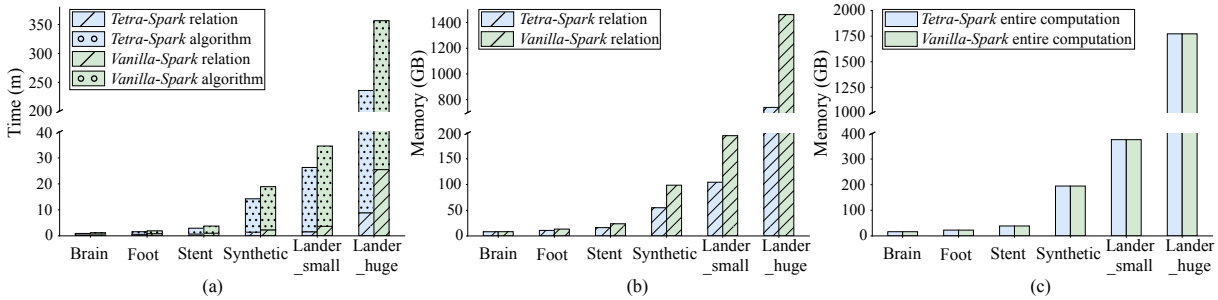


Figure 10: (a) The time cost (in minutes) for deriving connectivity relations and executing the algorithm in computing vertex distortion. (b) The peak memory consumption (in GB) for extracting relations. (c) The overall peak memory usage (in GB) for the entire computation.

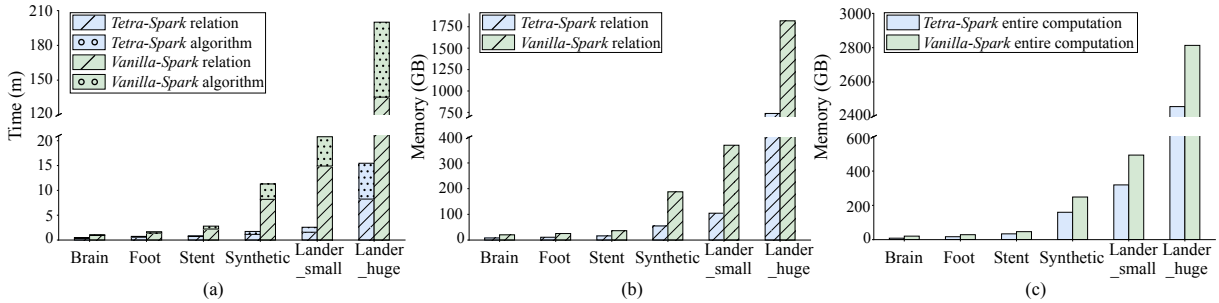


Figure 11: (a) The time cost (in minutes) for extracting connectivity relations and executing the algorithm in extracting critical points. (b) The peak memory consumption (in GB) for extracting relations. (c) The peak memory usage (in GB) for the entire computation.

faster in retrieving the involved relations, and up to 47.7× faster for computing the Forman gradient after preparing the relations.

This notable increase in time efficiency for deriving relations in Tetra-Spark is due to its approach of retrieving only the VT relation globally, while all other relations are computed locally from VT. Additionally, the requirement for several join operations to associate these relations within the same DataFrame row in Vanilla-Spark contributes to its lower performance.

The speedup of actual algorithm computation in Tetra-Spark arises from its memory efficiency in extracting relations. It uses approximately 60% less memory than Vanilla-Spark, as shown in Fig. 12b. This allows an effective utilization of memory to cache more intermediate data during subsequent computations.

Tetra-Spark demonstrates an overall memory usage reduction by up to 37% over Vanilla-Spark, as shown in Fig. 12c. This substantial decrease is primarily due to the reduced execution memory required for locally retrieving relations in Tetra-Spark. Conversely, Vanilla-Spark necessitates join operations to associate tetrahedra, triangles, and edges incident to the same vertex within the same row of a DataFrame. This process requires massive data exchanges among nodes and results in higher memory consumption.

6.4 Node scalability analysis

To explore whether the performance of Tetra-Spark consistently improves when more nodes are available in a cluster, we conducted various experiments focusing on the speedup and efficiency achieved with varying numbers of executors. Since the observed performance is similar on all datasets, in this section, we show only the plots from the largest dataset *Lander_huge*. In the supplementary material, we provide figures describing the performance trends of other datasets.

Fig. 13a illustrates the speedup achieved by Tetra-Spark as the number of executors increases from 2 to 64. The results show that Tetra-Spark experiences almost linear speedup as the executor count rises from 2 to 32. The speedup is achieved by more executors available for parallel computation. However, a diminishing return on speedup is observed when the number of executors rises to 64. Specifically, the speedup with 64 executors is only approximately 1.1× compared to 32 executors. Further analysis indicates that the total number of cores configured for use with 64 executors reaches 320, which substantially exceeds the 168 available physical cores among the worker nodes in the cluster. Additionally, the workload is spread across more nodes when configuring 64 executors, leading to greater complexity in coordination and resource management, which further diminishes the speedup gains.

When comparing the speedup across different applications, the computation of the Forman gradient achieves the highest speedup, while vertex distortion records the lowest. The superior performance of the Forman gradient stems from its local processing approach. Once the required connectivity relations are obtained, the Forman gradient is computed within the same row and partition, making optimal use of available computational resources. Conversely, computing vertex distortion requires a *join* operation to retrieve vertex coordinates and scalar values. This process involves substantial data shuffling and inter-node communication, which can significantly reduce the speedup gains.

Fig. 13b depicts the efficiency of Tetra-Spark when the number of executors is varied. Efficiency is defined as the ratio of the time cost with a single node to the product of the time requirements with N nodes and N. As shown in Fig. 13b, the efficiency drops as the executor number increases, which is a common characteristic in a distributed computing system. This decrease is due to the increased inter-executor communication overhead and potential load imbalance across the cluster executors. Notably, the drop becomes more pronounced when the number of executors exceeds 32. This is because the number of allocated cores with 64 executors, which is 320, exceeds the number of available physical cores, which is 168.

When it comes to the comparison of efficiency among different applications, vertex distortion calculation drops more rapidly than other applications. As highlighted in the speedup analysis, this decrease is primarily driven by the significant data shuffling and inter-node communication required by the *join* operation for retrieving

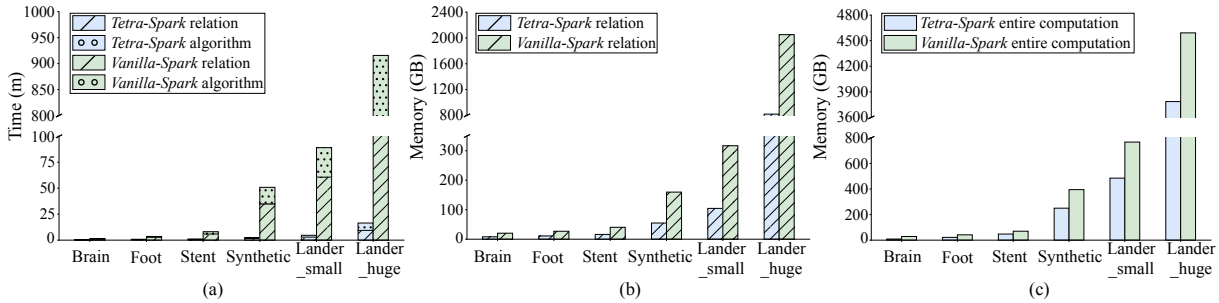


Figure 12: (a) The time cost (in minutes) for extracting connectivity relations and executing the algorithm in computing Forman gradient. (b) The peak memory consumption (in GB) for extracting relations. (c) The peak memory usage (in GB) for the entire computation.

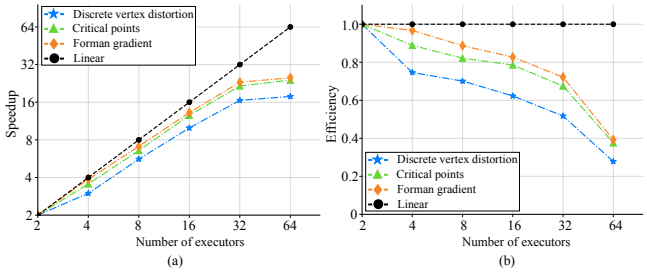


Figure 13: The (a) speedup and (b) efficiency achieved by Tetra-Spark.

vertex coordinates and scalar values.

7 Discussion of Limitations

We have introduced three methods for computing connectivity relations in Spark: the *pure global*, *symmetric global*, and *local* methods. Our experimental evaluations have demonstrated the efficiency of the localized strategy in deriving connectivity relations and computing topological features. In this section, we explore the limitations of the localized methodology implemented in Tetra-Spark.

As detailed in Sec. 6.2, the localized method achieves significant speedup when the VT relation is pre-computed, allowing for embarrassingly parallel computations of many connectivity relations. However, this strategy provides limited speedup for applications requiring only a few relations, due to the preliminary global computation of the VT relation. A potential improvement could involve embedding the VT relation directly within the original DataFrame DF_V. This would eliminate the need for an initial global retrieval step but increase the storage requirements of Tetra-Spark.

Additionally, as discussed in Sec. 6.3, the localized methodology exhibits limited efficiency when computing topological features that require vertex coordinates. This inefficiency arises from the lack of a global array storing all vertex data. Consequently, a time-consuming *join* operation with DF_V is necessary to access vertex coordinates for analyses requiring this data. To enhance efficiency, one possible solution is to incorporate vertex data directly into the original DataFrame DF_T. Specifically, the vertex index in DF_T could be replaced with an array that includes the vertex's coordinates and associated scalar values. Since these attributes are unique to each vertex, this array could effectively serve as a new vertex index. This approach would transform the encoding of a tetrahedral mesh into a single DataFrame composed of four columns, each containing an array of coordinates and scalar fields for one of the tetrahedron's vertices.

Furthermore, as described in Sec. 6.3, the localized methodology leverages the power of embarrassingly parallel processing for computing topological features like critical points and the Forman gradient. However, these computations are predominantly local, relying mainly on the connectivity relations limited to a vertex's neighbors. For applications requiring extensive mesh navigation, the performance gains would not be the same. This limitation stems from the need for frequent data access between nodes, which can slow down processes due to increased communication overhead. Additionally, Spark does not support granular control over inter-node communication, which makes it more challenging to minimize the communication overhead. One potential solution involves defining custom partitioning for DataFrames, which could implicitly reduce network traffic by strategically distributing data across nodes to align with the computation needs. Another scheme is to conceptualize the mesh as a graph and integrate Tetra-Spark with graph-parallel computation systems like Apache Giraph [9] and GraphX [22] to facilitate more efficient graph navigation.

8 Conclusions

We have proposed Tetra-Spark, a new framework for processing large tetrahedral meshes within Apache Spark. This framework encodes only the essential information necessary to represent a tetrahedral mesh with scalar fields defined on it. Tetra-Spark supports the optimized extraction of boundary, co-boundary, and adjacency relations in batches, thereby providing a substantial speedup compared to global methods that rely on native Spark DataFrame operations. This optimized design of locally computing groups of relations also greatly benefits subsequent algorithm executions.

We have experimentally demonstrated the efficiency and scalability of Tetra-Spark compared to a vanilla Spark system. With the pre-computed VT relation available, Tetra-Spark exhibits substantial performance enhancements: it achieves up to a 42× speedup and uses up to 57% less memory for deriving boundary relations. For coboundary relations, it offers up to a 78× speedup while saving up to 80% of memory. Moreover, in computing adjacency relations, Tetra-Spark reaches up to a 33× speedup with comparable memory usage. The memory efficiency in deriving relations significantly enhances the performance of subsequent algorithm computations, leading to speedups ranging from 1.2× to 47.7×. Overall, Tetra-Spark achieves a 1.3× to 55.6× speedup across the entire algorithmic process for computing topological features.

Our future work involves expanding Tetra-Spark to include the computation of topological abstractions that require heavy inter-node communications. This includes the computation of Morse complexes by leveraging the navigation of the Forman gradient [45], merge trees [28], and persistent homology [16]. Additionally, implementing geometry-aware partitioning for the input data is a promising direction, which could reduce inter-node communication during the extraction of connectivity relations. Furthermore, we aim to adapt Tetra-Spark for use with higher dimensional simplicial complexes or cell (CW) complexes to increase its versatility and applicability [18, 21].

Supplemental Materials

The source code for *Tetra-Spark library (TSL)* is available on GitHub[2]. All supplemental materials are available on OSF at https://osf.io/68hwe/?view_only=088db835e5e341d88b350c2071df66bd, which include (1) a thorough introduction to the extraction of all connectivity relations, (2) detailed descriptions of three algorithms used for computing different topological features, (3) the experimental results from pilot tests used for tuning Spark parameters, (4) the experimental results focused on computing all connectivity relations, and (5) experimental figures to evaluate the node scalability using remaining five datasets.

Acknowledgments

The authors would like to thank Philips Research, Hamburg, Germany, for the *Brain* and *Foot* datasets, and Michael Meißner, Viatronix Inc, for the *Stent* dataset. The *Synthetic* dataset is provided by Klacansky et al.[29]. The *Lander_small* and *Lander_huge* datasets are provided by Ingo Wald. This work has been supported by the US National Science Foundation under grant numbers IIS-1910766 and OAC-2403022.

References

[1] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pp. 1383–1394. New York, NY, USA, 2015. 1, 2, 4

[2] U. Bauer, M. Kerber, and J. Reininghaus. Distributed computation of persistent homology. In *2014 proceedings of the sixteenth workshop on algorithm engineering and experiments (ALENEX)*, pp. 31–38. Philadelphia, PA, USA, 2014. 3

[3] 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. New Orleans, LA, USA, 1997. 2

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

[5] D. Canino and L. De Floriani. Representing simplicial complexes with Mangroves. In *Proceedings of the 22nd International Meshing Roundtable*, pp. 465–483. Springer, Orlando, FL, USA, 2014. 2

[6] 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. 2

[7] J. Chen, D. Zhao, Y. Zheng, Y. Xu, C. Li, and J. Zheng. Domain decomposition approach for parallel improvement of tetrahedral meshes. *Journal of Parallel and Distributed Computing*, 107:101–113, 2017. 1

[8] X. Chen, J. Liu, Y. Pang, J. Chen, L. Chi, and C. Gong. Developing a new mesh quality evaluation method based on convolutional neural network. *Engineering Applications of Computational Fluid Mechanics*, 14(1):391–400, 2020. 1

[9] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015. 9

[10] 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. 2

[11] 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. 2

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 2, 3

[2] https://github.com/qyh-sunshine/Tetra_Spark

[13] J. P. D'Amato and M. Venere. A cpu–gpu framework for optimizing the quality of large meshes. *Journal of Parallel and Distributed Computing*, 73(8):1127–1134, 2013. 1

[14] H. Edelsbrunner. *Algorithms in combinatorial geometry*, vol. 10. Springer Science & Business Media, 1987. 2

[15] 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, 2003. 7

[16] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. *Discrete and Computational Geometry*, 28(4):511–533, 2002. 7, 9

[17] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st international conference on Data Engineering*, pp. 1352–1363. Seoul, Korea (South), 2015. 3

[18] R. Fellegara, F. Iuricich, L. De Floriani, and U. Fugacci. Efficient homology-preserving simplification of high-dimensional simplicial shapes. In *Computer Graphics Forum*, vol. 39, pp. 244–259. Wiley Online Library, 2020. 9

[19] R. Fellegara, K. Weiss, and L. De Floriani. The Stellar decomposition: A compact representation for simplicial complexes and beyond. *Computers & Graphics*, 98:322–343, 2021. 2

[20] R. Forman. Morse theory for cell complexes. *Advances in mathematics*, 134(1):90–145, 1998. 7

[21] U. Fugacci, F. Iuricich, and L. De Floriani. Computing discrete morse complexes from simplicial complexes. *Graphical models*, 103:101023, 2019. 9

[22] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)*, pp. 599–613, 2014. 9

[23] H. Guo, D. Lenz, J. Xu, X. Liang, W. He, I. R. Grindeanu, H.-W. Shen, T. Peterka, T. Munson, and I. Foster. Ftk: A simplicial spacetime meshing framework for robust and scalable feature tracking. *IEEE transactions on visualization and computer graphics*, 27(8):3463–3480, 2021. 1

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

[26] S. Hang. Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Softw*, 41(2):11, 2015. 1

[27] I. Hotz, R. Bujack, C. Garth, and B. Wang. Mathematical foundations in visualization. *Foundations of Data Visualization*, pp. 87–119, 2020. 1

[28] 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*, pp. 367–377. ACM, New York, NY, USA, 2021. 9

[29] P. Klacansky, H. Miao, A. Gyulassy, A. Townsend, K. Champley, J. Tringe, V. Pascucci, and P.-T. Bremer. Virtual inspection of additively manufactured parts. In *2022 IEEE 15th Pacific Visualization Symposium (PacificVis)*, pp. 81–90. Tsukuba, Japan, 2022. 10

[30] M. Kremer, D. Bommes, and L. Kobbelt. OpenVolumeMesh – a versatile index-based data structure for 3D polytopal complexes. In *Proceedings of the 21st International Meshing Roundtable*, pp. 531–548. Springer, Berlin, Heidelberg, 2013. 2

[31] J. Langguth, M. Sourouri, G. T. Lines, S. B. Baden, and X. Cai. Scalable heterogeneous cpu-gpu computations for unstructured tetrahedral meshes. *IEEE Micro*, 35(4):6–15, 2015. 1

[32] C. Lawson. Software for C^1 surface interpolation. In *Mathematical Software*, pp. 161–194. Academic Press, 1977. 2

[33] E. Le Guillou, M. Will, P. Guillou, J. Lukasczyk, P. Fortin, C. Garth, and J. Tierny. Ttk is getting mpi-ready. *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–18, 2024. 3

[34] G. Liu and F. Iuricich. A task-parallel approach for localized topo-

logical data structures. In *2023 IEEE Visualization Conference (VIS)*. IEEE, Melbourne, Australia, 2023. 1, 3

[35] 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. 3

[36] S. Lo. Finite element mesh generation and adaptive meshing. *Progress in Structural Engineering and Materials*, 4(4):381–399, 2002. 1

[37] 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. 2

[38] T. B. Masood, J. Budin, M. Falk, G. Favelier, C. Garth, C. Gueunet, P. Guillou, L. Hofmann, P. Hristov, A. Kamakshidasan, et al. An overview of the topology toolkit. *Topological Methods in Data Analysis and Visualization VI: Theory, Applications, and Software*, pp. 327–342, 2021. 3

[39] M. M. Mesmoudi, L. De Floriani, and U. Port. Discrete distortion in triangulated 3-manifolds. In *Computer Graphics Forum*, vol. 27, pp. 1333–1340, 2008. 7

[40] J. W. Milnor, M. Spivak, R. Wells, and R. Wells. *Morse theory*. Number 51. Princeton University Press, 1963. 7

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

[42] A. Nigmetov and D. Morozov. Local-global merge tree computation with local exchanges. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13. New York, NY, USA, 2019. 3

[43] 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. 2

[44] V. Pascucci, X. Tricoche, H. Hagen, and J. Tierny. *Topological Methods in Data Analysis and Visualization: Theory, Algorithms, and Applications*. Springer Berlin Heidelberg, 2010. 1

[45] 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. 2, 7, 9

[46] 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

[47] W. J. Schroeder, L. S. Avila, and W. Hoffman. Visualizing with vtk: a tutorial. *IEEE Computer graphics and applications*, 20(5):20–27, 2000. 4

[48] J. Shanahan and L. Dai. Large scale distributed data science from scratch using apache spark 2.0. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pp. 955–957. Perth, Australia, 2017. 2

[49] H. Si. Adaptive tetrahedral mesh generation by constrained delaunay refinement. *International Journal for Numerical Methods in Engineering*, 75(7):856–880, 2008. 1

[50] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref. Locationspark: A distributed in-memory data management system for big spatial data. *Proceedings of the VLDB Endowment*, 9(13):1565–1568, 2016. 3

[51] 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. 3

[52] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pp. 1–16. Santa Clara, CA, USA, 2013. 3

[53] H. T. Vo, J. Bronson, B. Summa, J. L. Comba, J. Freire, B. Howe, V. Pascucci, and C. T. Silva. Parallel visualization on large clusters using mapreduce. In *2011 IEEE Symposium on Large Data Analysis and Visualization*, pp. 81–88. Providence, RI, USA, 2011. 1

[54] 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*, pp. 92–101. ACM, New York, NY, USA, 2011. 2

[55] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *Proceedings of the 2016 international conference on management of data*, pp. 1071–1085. San Francisco, CA, USA, 2016. 3

[56] 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. In *Computer Graphics Forum*, vol. 40, pp. 599–633, 2021. 1

[57] S. You, J. Zhang, and L. Gruenwald. Large-scale spatial join query processing in cloud. In *2015 31st IEEE international conference on data engineering workshops*, pp. 34–41. Seoul, Korea (South), 2015. 3

[58] J. Yu, Z. Zhang, and M. Sarwat. Spatial data management in apache spark: the geospark perspective and beyond. *GeoInformatica*, 23:37–78, 2019. 3

[59] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010. 1, 2, 3, 5

[60] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016. 1, 2, 3, 7