Distributed merge forest: A new fast and scalable approach for topological analysis at scale

Xuan Huang
xuanhuang@sci.utah.edu
SCI Institute, University of Utah

Pavol Klacansky
klacansky@sci.utah.edu
SCI Institute, University of Utah

Steve Petruzza
steve.petruzza@usu.edu
Utah State University

Attila Gyulassy
jediati@sci.utah.edu
SCI Institute, University of Utah

Peer-Timo Bremer
bremer5@llnl.gov
Lawrence Livermore National Laboratory

Valerio Pascucci
pascucci@sci.utah.edu
SCI Institute, University of Utah

ABSTRACT

Topological analysis is used in several domains to identify and characterize important features in scientific data, and is now one of the established classes of techniques of proven practical use in scientific computing. The growth in parallelism and problem size tackled by modern simulations poses a particular challenge for these approaches. Fundamentally, the global encoding of topological features necessitates interprocess communication that limits their scaling. In this paper, we extend a new topological paradigm to the case of distributed computing, where the construction of a global merge tree is replaced by a distributed data structure, the merge forest, trading slower individual queries on the structure for faster end-to-end performance and scaling. Empirically, the queries that are most negatively affected also tend to have limited practical use. Our experimental results demonstrate the scalability of both the merge forest construction and the parallel queries needed in scientific workflows, and contrast this scalability with the two established alternatives that construct variations of a global tree.

CCS CONCEPTS

• **Theory of computation** → **Distributed algorithms**; • **Mathematics of computing** → *Exploratory data analysis.*

KEYWORDS

distributed algorithms, topological data analysis, feature extraction

ACM Reference Format:
Xuan Huang, Pavol Klacansky, Steve Petruzza, Attila Gyulassy, Peer-Timo Bremer, and Valerio Pascucci. 2021. Distributed merge forest: A new fast and scalable approach for topological analysis at scale. In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3447818.3460358

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICS '21, June 14–17, 2021, Virtual Event, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8335-6/21/06…$15.00
https://doi.org/10.1145/3447818.3460358

1 INTRODUCTION

Analysis of modern large-scale simulation and experimental data continues to be an increasing challenge as the data grow in size and complexity. In particular, contemporary supercomputers present an increasing divergence between computing power and I/O bandwidth. As a result, the quality or temporal resolution of data that can be stored to disk for analysis is increasingly a small fraction of the data generated by the simulation, limiting the quality of analysis results. For example, saving only every n-th time step makes feature tracking challenging. Furthermore, the data generated already exceed what can be analyzed post hoc on shared-memory systems. Both factors contribute to a pressing need for scalable data analysis on distributed systems: either operating in situ with the simulation to access the highest quality data or to process massive data in a timely manner.

Topological approaches provide a mathematical language that encapsulates the relationships among intrinsic features of the data, allowing scientists to formulate robust definitions of domain-specific phenomena, studying them at the level of individual features. Although certain classes of analyses are well suited for distributed computation, topological techniques face a particular challenge, because they inherently encode data that span many regions of the domain decomposition typical of large-scale distributed simulations. Topological approaches first encode the features of a function in data structure that can then be queried efficiently to answer domain-specific questions on a per-feature level. These approaches have traditionally been limited by the need to resolve global information in the data structure encoding stage, with interprocess communication limiting the scalability of most approaches. Our work is motivated by the empirical observation that for the majority of scientific workflows, only a small subset of available features is used, and therefore much work is wasted in the encoding stage. We extend to the distributed setting a reorganization of topological computation and query workflows, where data-parallel partial results are computed rather than a full encoding of the topological features, delaying communication to the query stage to resolve only the necessary features.

The merge tree [6] is a topological structure that encodes the merging and nesting of superlevel-set components associated with local maxima. Once computed, the merge tree acts both as an acceleration structure and as a means of defining phenomena of interest. It reduces the input size for answering queries about superlevel-set components from $O(N)$, where N is the number of samples in a

dataset, to $O(k)$, where k is the number of maxima. Furthermore, the maxima and branch points of the tree can assist in locating phenomena of interest and defining local thresholds, enabling a user to programmatically initiate queries at those locations that adapt to the local scale of the data.

Components of the merge tree often correspond to phenomenological features. For instance, in combustion, regions around high-valued local maxima of the temperature field identify ignition kernels [15]. In turbulent flows, a threshold that is defined based on the local tree structure around each maximum is used to extract multi-scale vortex structures [5]. In atmospheric science, leaves and sub-trees of the merge tree are used to extract locally thresholded superlevel-set components around maxima to track high-pressure regions [21]. The volume of superlevel-set components is used to compute a percolation threshold, which is useful in studying a flow's turbulence and validating normalization schemes [7, 10]. Although there exist several approaches to computing merge trees in a distributed setting [11, 16, 18], they are mainly focused on building a global data structure that is later used to answer topological queries. The construction of a global tree is often a very expensive task and requires a series of collective operations that dramatically affect performance at scale. For the example of in situ analysis, due to lower scalability compared to the simulation codes, these algorithms become a bottleneck at a higher core count as they take an increasingly larger proportion of a simulation time step. In this paper, we extend a new approach of localized data structures to the case of distributed computing, where the construction of a global merge tree is replaced by a distributed data structure called the merge forest.

We describe the distributed computation of a merge forest, and adapt the algorithms for querying this structure to the distributed setting. This topological data structure is later used to introduce the first experimental performance study of topological queries on distributed large-scale scientific data. Along with a demonstration of linear scaling of the data structure computation, we present an experimental study of topological queries and provide new insights into resource contention and scalability when multiple queries are executed asynchronously in parallel. We demonstrate a practical use case in using the merge forest to find density clusters in a cosmology simulation. The observed performance demonstrates that the introduced techniques represent a good choice for distributed topological analysis in two important regimes: (i) at low core counts where users perform feature space exploration and (ii) at large scale where production in situ analytics are performed.

In summary, our contributions are:

- a scalable and distributed approach to compute localized topological data structures (i.e., merge forest);
- a performance comparison with three distributed merge tree algorithms for the construction of topological data structures using different thresholds;
- implementation of fundamental distributed topological queries on a merge forest using different parallelism strategies;
- a performance study of distributed topological queries at scale, and a comparison with state-of-the-art implementations; and
- a performance analysis at scale of a distributed analysis use case in cosmology simulation data.

2 RELATED WORK

The need to compute merge trees for ever larger data has resulted in several approaches with varying degrees of success. Shared-memory parallel approaches tend to be most useful for post hoc exploratory analysis, at the cost of performance or limited flexibility. Shared-memory parallel approaches using divide-and-conquer of the domain [19], path tracing [2, 14], or range partition [8] all face limited scalability due to the need to build a global merge tree from constituent parts.

The seminal work on distributed merge trees [16, 17] was motivated by two issues of topological analysis at scale: a global tree larger than the node's memory and parallel analysis. The described local-global representation addresses both of these issues. It consists of a local merge tree constructed for a restricted subset of data (such as a region in a data decomposition) and a sparsified global tree with respect to this local tree. The local trees are small and so is the sparsified global tree, avoiding the issue of storing a full global tree on each node. Moreover, queries have all the information available from the local-global tree and can be executed with minimal communication. Usually a reduction suffices at the end of a query to gather the final result. The computation of local trees depends only on a ghost layer, and thus exhibits ideal scaling. In contrast, the construction of the sparsified global tree requires a global reduction, limiting the overall scaling.

An observation of the locality of features in a subset of scientific datasets leads to an approach that builds merge trees only for data regions up to a predefined extent, improving scalability [11, 20]. Furthermore, in some cases, an a priori superlevel set is known and data can be thresholded [11]. Then region growing can be resolved until correct [18]. Unfortunately, if the exact threshold is unknown, a conservative lower bound must be used, leading to less scalable execution due to more work at the global reduction phase of the algorithms.

Overall, these approaches focus on minimizing the communication during query time at the cost of an expensive precomputation. An alternative is to avoid computing a global data structure and run the analysis directly on the data. A potential middle ground is to construct a partial localized data structure, accelerating local queries, and resolving any global information on demand [9]. This different split between precomputation and queries can result in an overall reduction of the time to perform data analysis. We therefore explore the direction of localized data structures and trade-offs between communication during precomputation and queries on a distributed machine.

3 BACKGROUND

An approach to studying a scalar function is to study its contours and their behavior. A level set (an isosurface) is a preimage of a threshold, and it can have multiple connected components. A superlevel set is a preimage of all values greater than or equal to a threshold, where the level set is its boundary. A sweep from the maximum to the minimum of the function range can result in two events in terms of superlevel sets: (i) a new component is born at a

local maximum, or (ii) two components merge into one at a merge saddle. A merge tree captures all these changes in the form of a tree, rooted at the global minimum. Leaves correspond to local maxima, internal nodes to merge saddles, and arcs encode the evolution of superlevel-set components between component creation and merging. Therefore, this tree can serve as an acceleration structure for querying superlevel-set-based features, because it allows a query to skip vertices and edges in the data where there is no connectivity change. For instance, finding the maximum of each connected component given a superlevel-set threshold involves a symbolic cut and connected component computation on the relatively sparse tree, rather than traversing the domain. Often, this tree is several orders of magnitude smaller than the input data. The strictly localized data structure [9], called a merge forest, accelerates the recovery of similar information, and we review the basic definitions and algorithms from this work.

3.1 The Merge Forest

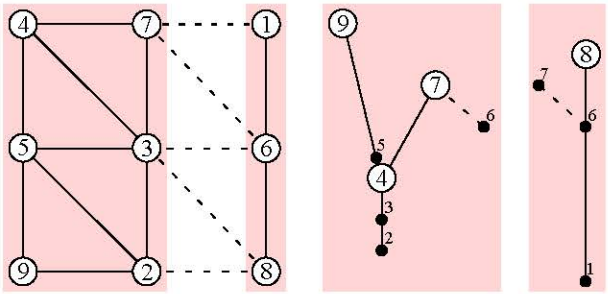
For parallel computing, the domain is usually partitioned into regions, such as in an in situ setting. Each region could also include layers of neighboring cells as *ghost regions*. We refer to the mesh edges connecting vertices owned by different regions as the *bridge set*. A merge forest (Fig. 1) is a collection of data structures local to each region recording topological changes only within, and at, region boundaries. Each region in the domain decomposition has its corresponding local merge tree, as well as a subset of the bridge set edges connecting the region to its neighboring regions (Fig. 1b). These edges, a *reduced bridge set*, tie the local merge trees together, and form a minimal set needed to reconstruct global features at query time. We find a reduced bridge set by running a union-find algorithm on sorted shared boundary between neighboring regions and insert only the edges that have end vertices in two different connected components. To extract a smaller set, the edges internal to a region are processed before the edges connected to neighboring regions (Alg. 1 [9]).

3.2 Querying the Merge Forest

Akin to a merge tree, a merge forest can serve as an acceleration structure for topological queries, such as *Maxima* query returning all local maxima in a dataset or *Component* query extracting the vertex set of a connected component, when compared to running those queries on the raw vertices of the mesh.

A forest-accelerated algorithm for extracting local maxima (Alg. 1) traverses each region's local tree (line 3) to find its leaves. In contrast to a global tree, each leaf needs to be checked for absence of a reduced bridge set edge connecting it to a higher valued vertex in neighboring region (lines 5-6).

A more complex query is to extract a connected component of a superlevel set containing a vertex (Alg. 2). The query starts by identifying a region containing the queried vertex, and tests if the value at the vertex is below the threshold or was previously visited (line 2). The vertex is marked as visited (line 4), and its corresponding arc in a local merge tree is accessed (line 5), then all vertices above the specified threshold are added to the segmentation (line 6). Now, the neighbors of an arc need to be traversed: local children, local parent, and neighboring arcs in adjacent regions through reduced



(a) Two regions and bridge edges. (b) Two local trees with their reduced bridge set.

Figure 1: An example of a domain partitioned into two regions (a) and its corresponding merge forest (b) (reproduced from [9], with permission). The input domain consists of vertices with function values, solid edges internal to a region, and dashed edges connecting regions. On a regular grid, both solid and dashed edges are defined implicitly by a 6-subdivision neighborhood. For example, as we sweep the function on the left from highest to lowest values, a component is born at value 9 and then 7. These two components merge at value 4. Recording these changes results in a merge tree structure (solid lines on the right). In a merge forest, we need to add a subset of the dashed lines (bridge set) connecting the neighboring regions (reduced bridge set) to maintain connectivity information.

Algorithm 1 The *Maxima* query algorithm that returns all local maxima in a domain.

1: **function** MAXIMA(function g, forest F)
2: $maxima \leftarrow \emptyset$
3: **for each** local merge tree T **in** F **do**
4: **for each** arc **in** T **do**
5: $edges \leftarrow \{(arc.maxVertex, v) \in arc.edges \mid g(v) > g(arc.maxVertex)\}$
6: **if** $arc.children = \emptyset$ **and** $edges = \emptyset$ **then**
7: $maxima \leftarrow maxima \cup arc.maxVertex$
8: **return** $maxima$

bridge set edges above the threshold (lines 7-9). For each arc, we use its vertex with maximum function value (*maxVertex*) to start the next recursive call. This recursive traversal ensures all arcs and reduced bridge set edges in the component are visited. The query terminates in linear time of the size of the forest in the worst case.

4 DISTRIBUTED TOPOLOGICAL QUERIES

We revisit the data structures and algorithms described in Section 3 and extend them to the context of a distributed setting. Let the region graph be the adjacency graph of the partition of the domain into regions, where each node represents a region in the domain and an edge represents the spatial adjacency of the regions. We model the queries and subsequent traversal of the merge forest as a traversal of the region adjacency graph, where we move among neighbor region nodes, each containing its local merge tree (Fig. 2). In this model, we will consider the root node of the graph traversal

Algorithm 2 The *Component* query algorithm that returns the vertices of a given component.

1: **function** COMPONENT(function g, forest F, vertex v, threshold h, visited set VS)
2: **if** $g(v) < h$ **or** $v \in VS$ **then**
3: **return** $\emptyset, VS$
4: $VS \leftarrow VS \cup \{v\}$
5: $arc_v \leftarrow$ VERTEXTOARC(F, v)
6: $vertices \leftarrow \{u \in arc_v.vertices \mid g(u) \geq h\}$
7: **for each** arc_n **in** ARCNEIGHBORS(g, F, arc_v, h) **do**
8: $vertices', VS \leftarrow$ COMPONENT(g, F, $arc_n.maxVertex$, h, VS)
9: $vertices \leftarrow vertices \cup vertices'$
10: **return** $vertices, VS$

to be the *shard* (i.e., the computing element associated to a single block of data) containing the input vertex. The traversal will then start from the root node and visit all the children nodes recursively.

We present distributed algorithms for three fundamental queries useful for feature extraction, i) a query to find all *maxima* in the domain; ii) a query to find the *maximum* with the highest function value in a connected component (its representative) for a given vertex and threshold; iii) a query to extract the mesh vertices constituting a component for a given vertex and threshold. These queries can be used to build more complex queries.

4.1 Maxima Query

This query represents the simplest topological query and returns a list of all maxima in a dataset. If a vertex has no higher neighbor, it is a maximum. Therefore, the query needs only its local neighborhood, and thus the distributed implementation is data parallel where each shard independently traverses its local merge tree (the inner loop on line 4 in Alg. 1) and its local reduced bridge set to find all local maxima. Each region's maxima are gathered and saved, or further passed to subsequent queries. This query involves no interprocess communication except to report results. The time complexity of this query is $O(rn)$ when executed directly on the dataset, and $O(|F|)$ when accelerated with a merge forest (r is the number of regions, n is the number of vertices inside a region, and $|F|$ is the number of nodes in a forest). The maxima query is usually followed by other queries, such as to extract components, and thus building a data structure is beneficial if the number of queries is large.

4.2 Component Query

The *Component* and *ComponentMax* queries are similar because both traverse a connected component of a superlevel set. Therefore, we discuss only the *Component* query.

The *Component* query receives as input a vertex and threshold that are used to start the traversal of the merge forest. Starting from the local tree corresponding to the region containing the input vertex, we find the connected neighbors using the local reduced bridge set. This strategy allows us to limit the communication only to the neighbors that share a component (the ARCNEIGHBORS subroutine on line 7 in Alg. 2). The traversal continues, potentially in other regions (line 8), until the portion of the merge forest containing the component is explored. Finally, the completion status is propagated back in the reverse order, reaching the starting shard.

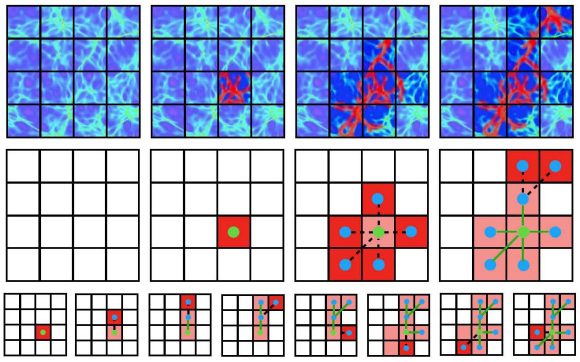


Figure 2: An illustration of a potential merge forest traversal. For each query, we traverse the forest without exploring all neighbors at each step. Instead, we use the local reduced bridge set to look up only the neighbors that have arcs in common. In the first row, we depict the traversal of data regions containing a feature (colored in red) in a cosmology dataset. In the second and third rows, we highlight the visited regions, following respectively, a breadth-first and depth-first traversal of the regions. The depth-first algorithm serializes the execution of the query, whereas the breadth-first approach enables more parallelism.

We note that this same traversal is not only useful to find the component's highest vertex (*ComponentMax* query) or extract a component, but also can be used as well for computing metrics along the components, such as volume, surface area, or other statistics.

Each shard starts its local traversal only when it receives incoming query request. Each local merge tree has a finite number of arcs. Each local traversal will mark some of the arcs visited, and continue with the unvisited arcs. Eventually, all arcs required by the local traversal are visited, and the shard reports the result to the parent node. The communication pattern can be different depending on the execution order (Fig. 3).

The execution of the query across regions is captured by a traversal of the region adjacency graph. Each node of the tree (i.e., shard) performs the following operations:

- execute incoming queries (e.g., collecting local vertices of a component in the local tree);
- evaluate incoming results from children nodes;
- forward queries to connected neighbors; and
- forward results to the parent node.

4.3 Regions Traversal Order

A traversal of regions implicitly defines a graph, and thus we can apply different strategies from graph theory to traverse it, such as a depth-first search (DFS) or breadth-first search (BFS). To avoid an infinite traversal due to potential cycles in the graph, each region maintains a set of visited vertices per query. We evaluate both approaches and analyze their impact on performance. Furthermore, to reduce the number of messages between shards, we aggregate all requests per neighboring shard into a single message. Moreover, we send a source vertex alongside the target vertex during a

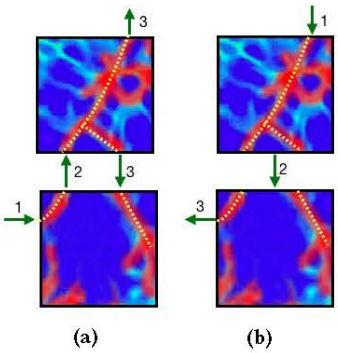


Figure 3: *Component* query traversal of a merge forest may vary depending on the order the messages are received by neighboring regions. For example, (a) the bottom region receives the message, first resulting in four messages in total, and (b) the top region is traversed first followed by the bottom region (total three messages, because we batch all messages to the same region such as the arrow number 2). This traversal variability can impact both the number of messages and the amount of parallelism during a query.

Table 1: Datasets used in our experiments. *Neurons* is a volume of high-resolution neurons images (two-photon microscopy), *Cosmology* is produced by a simulation of universe formation, and *DNS* is a channel flow simulation (produced by a direct numerical combustion simulation).

Dataset	Resolution	Size (GB)	Arc count
Neurons	2048x2048x2048	16	910,677,279
Cosmology	2048x2048x2048	32	759,760,093
DNS	10240x7680x1536	900	138,601,501

query. This source vertex enables the neighbor to avoid traversing back alongside the same bridge set edge and can be checked in a logarithmic time.

Fig. 2 depicts the two types of traversal, DFS and BFS, indicating the time when each computation can be executed on each region's data structure. This illustration highlights how the DFS approach serializes the execution of the query, because each parent node has to wait for the result of one of the children before sending the query request to its other children. Conversely, the BFS approach allows us to forward query requests to all children in parallel (i.e., spatial neighbors in the domain containing a connected merge tree).

Finally, a set of queries can be executed sequentially or concurrently. The sequential execution waits for a previous query to finish before starting the next query, and thus limits the throughput. In contrast, queries run concurrently reveal opportunities for parallelism and overlapping of computation and communication, maximizing throughput at the cost of increased memory usage.

5 EVALUATION

For our experiments, we used two supercomputers: the National Energy Research Scientific Computing Center (NERSC) Cori supercomputer and the Texas Advanced Computing Center (TACC) Stampede 2. Cori is a Cray XC40 supercomputer equipped with 9,688 KNL nodes, each with a single-socket 68-core Intel Xeon Phi 7250 processor, and 2,388 Haswell nodes, each with two sockets of a 16-core Intel Xeon E5-2698 v3 processor. Stampede2 is the University of Texas at Austin's flagship supercomputer. It hosts 4,200 KNL compute nodes with the a single-socket 68-core Intel Xeon Phi 7250 processor (i.e., same as Cori's KNL nodes), and 1,736 SKX compute nodes with 48 Intel Xeon Platinum 8160 cores on two sockets. Due to intermittent machine availability, we have performed the comparison between Reeber [1] (a library implementing the local-global representation [16, 18]) and the merge forest (Fig. 13) on Stampede 2 and all other experiments on Cori. On both machines, we used exclusively KNL nodes in order to obtain comparable results. In all our experiments, we map each region of data to a separate MPI rank and use MPI asynchronous communication API (i.e, *Isend*, *Irecv*) for data exchange between ranks. Finally, for some of the local debugging and profiling, we used the Intel® Trace Analyzer [4].

The size of topological data structures is data dependent. Therefore, we test the distributed construction and queries on a range of different datasets: (i) a volume of high-resolution neurons images (two-photon microscopy), (ii) a cosmology [13], and (iii) a channel flow simulation dataset [12] (i.e., computed via direct numeric simulation; we will refer to it as DNS). Table 1 reports the dimensions, size, and topological complexity of each dataset. The arc count refers to the number of arcs in the global merge tree. We performed experiments to measure the performance scalability of the merge forest construction and three fundamental queries in distributed setting: *Maxima*, *ComponentMax*, and *Component* query. We compare the computation of the merge forest with the state-of-the-art distributed merge tree implementations Parallel Merge Tree (PMT) [11] and Reeber [1].

5.1 Distributed Merge Forest Construction Performance

Large-scale simulations on supercomputers produce data in small patches (or regions) that are distributed in memory among different nodes and cores. The construction of the merge forest builds local merge trees and reduced bridge sets for each region of the domain. In our distributed implementation, we load a single layer of ghost cells around each region to avoid communication with neighbors during the computation. This approach makes the distributed construction of the merge forest a data parallel problem. We demonstrate the scalability of this approach by performing both weak and strong scaling experiments.

In a distributed setting, the domain decomposition plays a large role in assigning data to computational resources. In many simulations, the size of the region that each shard (or task) will ingest or produce can either stay constant or vary in size at scale. For our weak scaling experiments, we use a proxy of the cosmology simulation dataset of increasing size to test the performance of the data structure construction using three different region sizes.

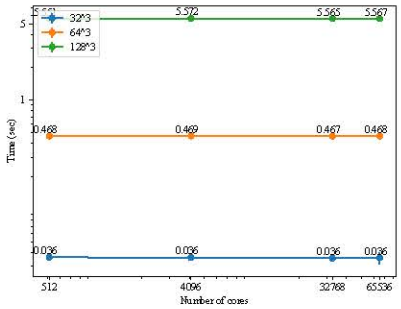


Figure 4: Weak scaling of the merge forest computation for different region sizes. The performance results demonstrate near-ideal scaling. In this plot, we report average time error bars. The error bars are very short and, hence, are not visible in the plot.

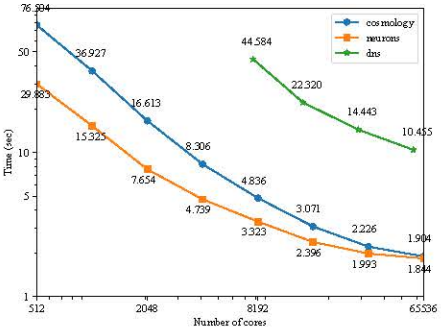


Figure 5: Strong scaling of the forest construction. Each core computes a local tree and local reduced bridge set, and thus we observe ideal scaling up to a point where load imbalance and machine variability limits further scaling. The superlinear scaling of *Cosmology* and *DNS* datasets is a consequence of decreasing region size and thus reducing the number of vertices to be sorted per region (i.e., $O(r(n \log n))$ complexity of the forest computation).

The experimental results (Fig. 4) demonstrate that the data parallel nature of the merge forest data structure exhibits scalability performance very close to the ideal scaling.

For strong scaling experiments, the region size decreases as we increase the number of cores (i.e., from 256x256x256 to 64x64x32). The results reported in Fig. 5 show good scalability of the approach, which decreases its efficiency when the region size becomes very small. For the cosmology and DNS datasets, we note an initial superlinear scaling that is caused by the logarithmic complexity of the local computation. In particular, the time complexity is $O(r(n \log n))$, where r is the number of regions (cores) and n is the number of samples inside a region. For example, a dataset with 128^3 samples would take $O(1(128^3 \log 128^3))$ on one core, but $O(8(64^3 \log 64^3)) = O(128^3 \log 64^3)$ on eight cores. In the case of the neurons dataset, we do not see this effect because the increased size of the reduced bridge set offsets the savings from the local

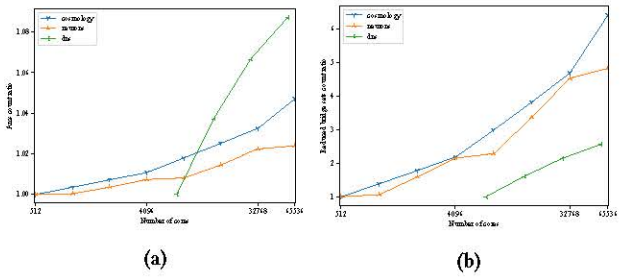


Figure 6: Ratios representing the variation of the number of arcs (a) and reduced bridge-set edges (b) in a merge forest at a particular number of cores with respect to the forest of initial experimental scale (i.e., 512 cores for the *Neurons* and *Cosmology* datasets, 7,700 cores for the DNS dataset).

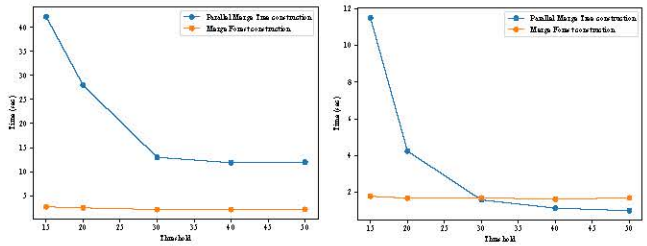
merge tree construction. In Fig. 6, we report the number of arcs and bridge sets in each dataset at scale. These numbers show that for smaller regions, we have an increasing number of both arcs and reduced bridge-set edges. This increase happens because when a region splits into smaller regions, some arcs are also split between the two new regions, causing an increase in the total number of arcs and reduced bridge-set edges.

5.2 Computation with Thresholds

In some cases, only high values encode features of interest, and the computation of the topological data structures can be sped up by ignoring all the data below a certain threshold. All three evaluated implementations, merge forest, PMT, and Reeber, support specification of this threshold. For this comparison, we performed experiments using the cosmology dataset with varying core counts and thresholds, where low values represent empty space, and high values represent density clusters. More details on the cosmology use case are given in Sec. 5.5.

At varying thresholds, we observe (Fig. 7) that the distributed merge tree performance is greatly affected by the threshold value, whereas the forest construction maintains a fairly constant performance. This behavior is caused by the size of the global merge tree, which for low thresholds is very dense and requires more communication when using the PMT or Reeber. On the other hand, the merge forest is not affected by communication overhead, and its computation is mostly bounded by the local data structures' creation time.

Finally, we selected a typical use case threshold (T=50) and performed a strong scaling study (Fig. 8) to compare thresholded computation of the merge forest with PMT. As expected, with increasing core counts, the v-cycle in communication dominates the distributed merge tree construction, degrading performance, whereas the forest construction maintains almost constant performance at any scale.



(a) 512 cores (b) 4,096 cores

Figure 7: Time to compute the distributed merge tree vs the merge forest at different thresholds using 512 (a) and 4,096 (b) cores. Note that at threshold 50 only 0.1% data is processed compared to 0.6% at threshold 15.

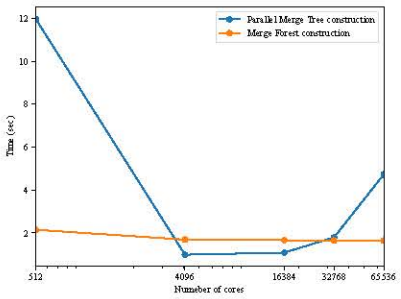


Figure 8: Strong scaling study of time to compute the global merge tree vs the merge forest for a fixed threshold (T=50). The higher runtime in the distributed merge tree between 4k and 32k cores is caused by the computation of the reduced bridge set. We choose to use the bridge set instead of a simpler ghost zone overlap in order to support AMR data.

5.3 Comparison of Distributed-Query Strategies

The distributed topological queries described in Sec. 4 intersperse local computation with occasional neighbor communication to traverse features that span different regions of the domain (Fig. 2). Specifically, in a distributed setting, the recursive calls of Component and ComponentMax queries traversing bridge set edges move the computation from one region of the domain to another, requiring communication.

We can model this region traversal using a graph that we start traversing from the vertex where the query originates through the neighbors containing connecting arcs of the feature of interest. This graph can be traversed in either a depth-first search (DFS) or breadth-first search (BFS) order. To evaluate the performance of the BFS and DFS approaches, we performed experiments at a small scale, using eight MPI ranks, and observed the allocation of computation and communication using the Intel® Trace Analyzer. In Fig. 10a, we report the profiler measurements for the computation of a ComponentMax query using a DFS (on top) vs a BFS (on the bottom) approach. In this figure, the blue areas represent computation and the red represent communication or waiting. In the DFS (on top) profiling, we see how the query is effectively executed serially (as depicted in at the bottom of Fig. 2). The BFS approach clearly enables a much higher parallelism where all children nodes (i.e., neighboring regions) can receive and execute queries and return their results to the parent in parallel. Given the higher parallelism and better performance of the BFS traversal, we use this approach in all our query experiments.

In the study of distributed topological queries, we performed experiments in different settings to assess the performance of: (i) queries executed sequentially, (ii) queries executed concurrently, (iii) queries traversing large portions of the domain (iv), and (v) queries targeting a specific topological analysis use case.

We start by observing and comparing the performance of topological queries when executed sequentially or concurrently. In our first experiment, we sample one hundred random vertices uniformly distributed in the domain, and we use the vertex data value as the threshold for each query. In particular, we execute 100 queries to find the maximum of the component containing the vertex at that value in the *Cosmology* dataset. With important features being sparse in this data, the majority of these queries occur in the low-valued background of the function, testing the worst-case performance of the queries. In Fig. 9, we show the total time to compute these 100 queries executed sequentially, concurrently, and individually (i.e., the average time to run one query). The single query performance, shown in Fig. 9, scales reasonably well, even for worst-case queries. Parallel efficiency decreases at scale as the smaller region sizes generate a larger overall data structure in terms of arcs and reduced bridge sets (as shown in Fig. 6), and also longer communication paths to identify the full extent of the features. However, when running the same 100 queries concurrently, the utilization and scalability improves significantly. We profile the execution of sequential (i.e., executed sequentially) and concurrent queries to observe the allocation of computation and communication time. In Fig. 10b, 10c, we report the profiling for running sequential and concurrent queries on a 512x512x512 subset of the neurons dataset using eight cores. The profiling results show how the concurrent execution of multiple queries allows a much better utilization of the cores and a significantly faster execution.

5.4 Scaling Performance of Concurrent Queries

We perform experiments at scale running 100 randomly sampled queries using three datasets described in Table 1. For each dataset, we perform strong scaling studies of three fundamental queries: *Maxima*, *ComponentMax*, and *Component*.

The threshold for each query is defined by the value at each chosen vertex. This approach produces a selection of queries that will traverse, in most cases, the largest features contained in the dataset. In Fig. 11, we report the distribution of the number of arcs traversed by 100 random *ComponentMax* queries in each dataset. These histograms show that most queries traverse a large portion or the entire dataset.

In Fig. 12, we report the strong scaling performance of executing 100 concurrent queries. The experimental performance results show good scalability for all queries. In particular, the *Maxima* performance is similar for the three datasets, because they perform

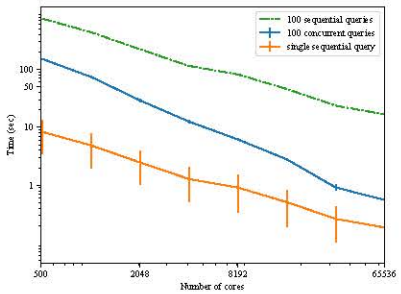


Figure 9: Time for executing 100 ComponentMax queries sequentially, in parallel, and the individual query (i.e., time to execute each of the 100 queries one at a time). The orange line shows the average and standard deviation times to run a single query. The green line represents running all 100 queries sequentially. Running queries concurrently allows for greater utilization of the machine (the blue line).

a data parallel operation, which essentially consists of the traversal of the local merge trees. This traversal does not require any communication resulting in a very fast query. The *ComponentMax* and *Component* queries' performance results exhibit good strong scalability. In particular, for the *DNS* dataset, we notice how *ComponentMax* queries are faster due to fewer arcs and *Component* queries are slower because of much larger features (i.e., more vertices) present in the dataset (Table 1). Finally, queries on *Cosmology* and *Neurons* datasets show very similar performance, with slightly slower execution time for *Neurons* due to more arcs.

5.5 Topological Queries at Scale: Density Clusters in Cosmology

Cosmological simulations compute the clumping of matter into gravitationally bound objects called halos, which include galaxies and dark matter [3]. Spatial regions of local high density can be used as a proxy to identify potential halos, and statistics of these regions mirror the relationship among the shape, density profile, and substructure of halos. We propose an algorithm for extracting high-density objects from grid data as a three-stage process: (i) perform a *Maxima* query to identify all maxima and keep only those above a given density threshold, (ii) perform a *ComponentMax* query for each maximum to identify the highest maximum within each component, and (iii) extract the vertices of each component highest maximum using the *Component* query, and gather the component sizes. Although in this work we use these queries in analyzing matter distribution in cosmology, the same queries can be used to identify vortex cores in turbulence and ignition kernels in combustion simulations, among other applications.

The *Maxima* query is data parallel whereas the cost of the *ComponentMax* and *Component* depends on the size of the halos and how they expand throughout the regions. The threshold that we choose will affect the number of local maxima and the size of the forest traversed by the *ComponentMax* queries as reported in Table 2. We performed experiments using the *Cosmology* dataset, dark

Table 2: The thresholds used in the cosmology density-region analysis with a corresponding number of maxima and the proportion of data values above the threshold. High thresholds capture only the densest clusters, whereas both the number and size of clusters increase with lower thresholds.

Threshold	Maxima count	Data proportion
500	45,167	0.003%
200	207,830	0.02%
100	504,153	0.05%
50	1,130,895	0.1%

matter density field, using two thresholds, one of interest to cosmologists (T=50) and one artificially high (T=5000) (dataset's value range is [0, 20104.877]). At these thresholds, the *Maxima* query finds 388 (at T=5000) and 1,130,045 (at T = 50) local maxima, which then trigger the execution of the same number of *ComponentMax* queries, starting at these maxima. We find the component of each highest maximum using the *Component* query and gather a histogram of sizes, one of the statistics of typical interest in studying the evolution of halos.

In this topological analysis use case configuration, the features are relatively small and span few regions. Fig. 13 shows the performance of the high-density-region finding query. The time scale for this query is very small for two reasons: (i) using a threshold limits the number of resulting maxima from the *Maxima* query, and (ii) the features identified in the dataset are small and so the *ComponentMax* and *Component* query do not traverse a large portion of the domain. The splitting of small features due to the region decomposition may introduce communication overhead at higher core counts, but since these features remain relatively local, the overall time is fast. The experimental results also show how the split of features among ranks at high core count can produce an increase in execution time.

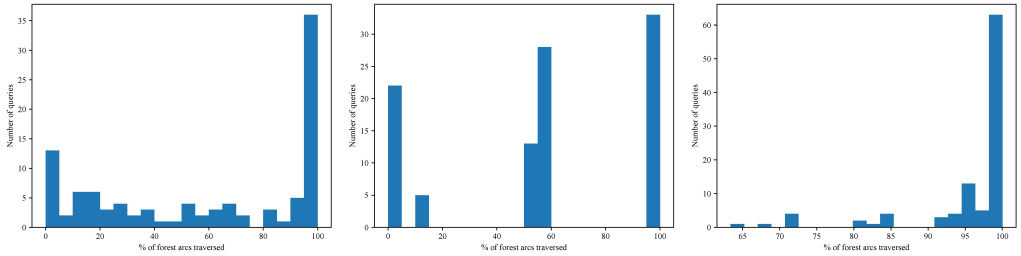
These experiments demonstrate not only the performance of our approach but also that in many practical use cases topological analysis can be performed efficiently without the need for a global data structure or expensive preprocessing steps. For example, at 65,536 cores and threshold 5000, the forest computation took 1.92 seconds and subsequent queries 0.02 seconds, a fast end-to-end time in the absolute sense.

Unfortunately, neither of the *PMT* nor the *Reeber* libraries provide a distributed query implementation to compare against (with the exception of halo component extraction using the components algorithm [18] in Reeber). Hence, in order to compare the performance of queries using the different data structures, we implemented Maxima and ComponentMax queries in the publicly available local-global implementation (*Reeber*). In our implementation of the ComponentMax query, we precompute a subtree maximum for each saddle to avoid traversing the tree toward leaves.

In Fig. 13, we report a strong scaling study that compares the data structure construction and query run times for *Reeber* and the distributed *Merge Forest* using the Cosmology dataset with varying thresholds. Although the queries accelerated by *Reeber* are much

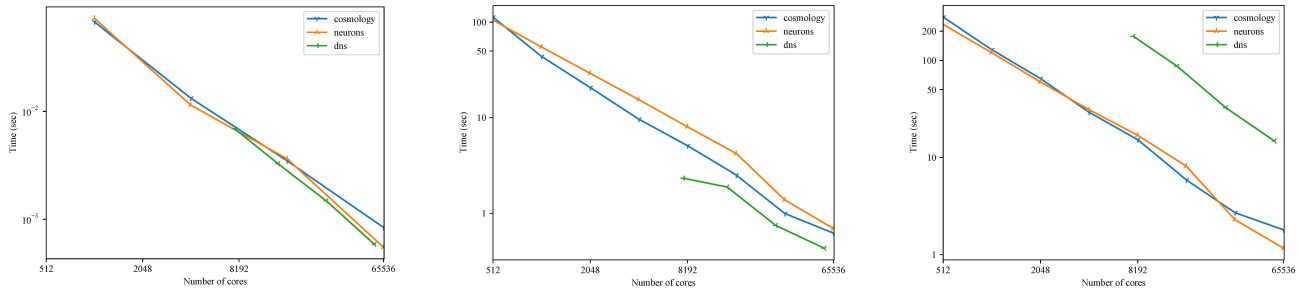
(a) (b) (c)

Figure 10: Profiling of *ComponentMax* queries on a 512x512x512 subset of the *Neurons* dataset using eight cores. In (a) we compare executions of a query using the DFS and BFS approach. The selection on top (for DFS) represents a 3.5 sec execution time and 1.3 sec for the one on the bottom (for BFS). The blue areas represent computation, and the red represent communication or wait (for results) time. Black lines indicate communication between ranks. The DFS shows a very poor parallel utilization, resulting in a serial computation. The BFS scheme, instead, sends out the workload across all neighbor ranks as soon as possible and thus results in more parallelism and utilization. In (b), we report the execution of 10 sequential (top, 10.7 sec) and concurrent (bottom, 5.1 sec), which shows how the parallel execution of multiple concurrent queries allows a better utilization of the cores and a faster execution. In (c), we report the execution of 10 (top, 5.2 sec) and 40 (bottom, 16.5 sec) random concurrent queries. Although the number of queries grows by a factor of four, the time only grows by a factor of three, which indicates a good degree of parallelism and overall utilization.



(a) Cosmology (b) Neurons (c) DNS

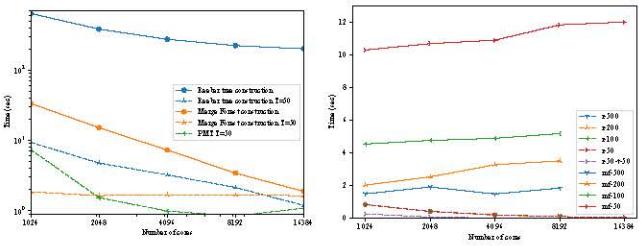
Figure 11: Percentage of arcs traversed by 100 random *ComponentMax* queries when using 65,536 cores for the cosmology (a) and neurons (b) and 57,600 cores for the DNS (c) dataset.



(a) *Maxima* (b) *ComponentMax* (c) *Component*

Figure 12: Strong scaling performance of executing 100 concurrent queries. Experiments are performed using three fundamental queries (i.e., *Maxima, ComponentMax, Component*) and three datasets.

faster than those accelerated by the merge forest, the overhead of precomputing the full local-global tree in *Reeber* is enormous (e.g., 20x slower). Thus, with non-thresholded dataset the merge forest gives the shortest time to result, for every combination of core count and query threshold. For instance, at 16k cores, with a full data structure and query threshold T=50, the merge forest computes the result in 13.89 sec = 1.88 sec compute + 12.01 sec query, whereas *Reeber* computes the result in 201.5 sec = 201.4 sec compute + 0.05 sec query, for an overall workflow speed-up of 14.6x. However, if the threshold T=50 is used, the *Reeber* construction time



(a) Tree construction (b) High-density region queries

Figure 13: Strong scaling of data structures computation and queries using the Cosmology dataset. In Figure (a) we compare the construction of data structures for *Reeber*, *PMT*, and the distributed *Merge Forest* with a threshold of T=50, and *Reeber* and distributed *Merge Forest* without threshold. The *Reeber* construction of the merge tree is significantly slower than that of the distributed *Merge Forest* without thresholding (solid lines). However, if a portion of the data is removed by thresholding (dashed lines), all three data structures can be constructed in approximately the same time. In (b), we report the strong scaling of the queries for high-density regions in Cosmology. In this use case, we perform a *Maxima* query to find the local maxima in each region of the domain and then execute a *ComponentMax* and *Component* query for each of them to find the individual components. The reported queries are executed using different thresholds (Table 2) using *Reeber* (dashed lines) and the *Merge Forest* (solid lines). Results show that queries performed using the local-global data structure are faster than queries on the merge forest and are not affected by thresholding. However, this performance gain on the query run-time is quite small when compared with the 20-times-slower tree construction (a).

can be greatly reduced. Thus, when combined with the fast query execution, *Reeber* outperforms the merge forest in all configurations we tested.

6 CONCLUSION

Large-scale topological data analysis is a growing challenge due to limited scalability of existing algorithms. In particular, traditional topological analysis techniques are limited by preprocessing steps to encode global information into the data structures later used for the analysis. In practice, we observe that, for most scientific workflows, only a small subset of features is relevant to the user. This observation suggests that we can use a lazy approach where topological data structure can be computed locally, and all communication can be performed on demand at query time.

In this work, we describe the distributed computation of a merge forest, a localized topological data structure that can be computed in parallel without requiring any communication. Furthermore, we introduce the first experimental study of topological queries at scale using the merge forest.

We performed experiments on two supercomputers using three large-scale scientific datasets and demonstrated that our approach allows us to compute the merge forest with near-ideal performance scalability. Moreover, we presented techniques to compute sequential and concurrent queries at scale and provided experimental evidence of strong scalability performance of three fundamental queries: *Maxima*, *ComponentMax*, and *Component*. Furthermore, we have compared our approach with a state-of-the-art distributed merge tree implementation using different thresholds and core counts demonstrating faster construction time at scale. Additionally, we measured the query performance when using local-global representation that trades construction performance for faster queries.

Finally, we demonstrated the effectiveness and high performance of the introduced techniques on an experimental topological analysis use case of density cluster finding in a cosmology simulation dataset. In this case, we showed how a common topological analysis task can be performed quickly at scale by avoiding global operations and only using the distributed merge forest. This scalability makes our approach a promising solution for future topological analysis at scale.

We acknowledge that large-scale in situ analytics often use a wide range of parameters at run-time for each analysis pipeline. In the case of topological analysis, a specific parameter choice could produce a very large number of topological queries to be processed. In the current distributed query implementation, each rank is processing query requests from neighbor ranks in order of arrival. This fixed ordering can cause congestion and load imbalance when the number of requests to some rank grows. Our experiments executing queries on a very large number of components (i.e., in the order of millions) demonstrate how performance and scalability can be affected. Future work will investigate the usage of load balancing and other techniques to handle those requests more efficiently and improve the overall performance in those use cases. More importantly, it is likely that a combination of the two extremes, merge forest and local-global representation, driven by queries will yield the best results. For example, if a certain number of queries is reached traversing between two regions, a merge of the two regions could be triggered to build a local-global tree in both regions to avoid the communication cost by jumping between the two regions.

An important direction is to identify more abstract queries that can serve as a framework to implement other queries. For example, ComponentMax and Component queries could be abstracted into TraverseComponent query. Similarly to MapReduce approach, this query would take a map and reduce functions. The ComponentMax would use a map function to extract the highest vertex of an arc, and reduce function to get the highest vertex of all arcs.

Similarly, in situ settings could use a limited amount of resources for analysis compared to the ones used to produce the data. In the current study, we do not explicitly address load balancing issues of queries, but this investigation could be an interesting direction for future investigations.

ACKNOWLEDGMENTS

This work was funded in part by NSF OAC awards 1842042, 1941085, NSF CMMI awards 1629660, LLNL LDRD project SI-20-001, DoE award DE-FE0031880, and the Intel Graphics and Visualization Institute of XeLLENCE. This material is based in part upon work supported by the DoE NNSA under award DE-NA0002375. This research was supported in part by the Exascale Computing Project

(17-SC-20-SC), a collaborative effort of the DoE and the NNSA. This work was performed in part under the auspices of the DoE by LLNL under contract DE-AC52-07NA27344, and UT-Battelle, LLC under contract DE-AC05-00OR22725. The authors thank the Texas Advanced Computing Center for access to Stampede2 and the National Energy Research Scientific Computing Center (NERSC) for access to the Cori supercomputer.

REFERENCES

[1] 2020. Reeber. https://github.com/mrzv/reeber.
[2] Aditya Acharya and Vijay Natarajan. 2015. A parallel and memory efficient algorithm for constructing the contour tree. In *2015 IEEE Pacific Visualization Symposium (PacificVis)*. 271–278. https://doi.org/10.1109/PACIFICVIS.2015.7156387
[3] Ann S. Almgren, John B. Bell, Mike J. Lijewski, Zarija Lukić, and Ethan Van Andel. 2013. Nyx: A MASSIVELY PARALLEL AMR CODE FOR COMPUTATIONAL COSMOLOGY. *The Astrophysical Journal* 765, 1 (feb 2013), 39. https://doi.org/10.1088/0004-637x/765/1/39
[4] Ray Asbury and M Wrinn. 2004. MPI tuning with Intel/spl copy/Trace Analyzer and Intel/spl copy/Trace Collector. In *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No. 04EX935)*. IEEE, 4.
[5] Peer-Timo Bremer, Andrea Gruber, Janine Bennett, Attila Gyulassy, Hemanth Kolla, Jacqueline Chen, and Ray Grout. 2016. Identifying turbulent structures through topological segmentation. *Commun. Appl. Math. Comput. Sci.* 11, 1 (2016), 37–53. https://doi.org/10.2140/camcos.2016.11.37
[6] Hamish Carr, Jack Snoeyink, and Ulrike Axen. 2003. Computing contour trees in all dimensions. *Computational Geometry* 24, 2 (2003), 75–94.
[7] Anke Friederici, Wiebke Köpp, Marco Atzori, Ricardo Vinuesa, Philipp Schlatter, and Tino Weinkauf. 2019. Distributed percolation analysis for turbulent flows. In *2019 IEEE 9th Symposium on Large Data Analysis and Visualization (LDAV)*. 42–51. https://doi.org/10.1109/LDAV48142.2019.8944383
[8] Charles Gueunet, Pierre Fortin, Julien Jomier, and Julien Tierny. 2016. Contour forests: Fast multi-threaded augmented contour trees. In *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*. 85–92. https://doi.org/10.1109/LDAV.2016.7874333
[9] Pavol Klacansky, Attila Gyulassy, Peer-Timo Bremer, and Valerio Pascucci. 2019. Toward localized topological data structures: Querying the forest for the tree. *IEEE Transactions on Visualization and Computer Graphics* 26, 1 (2019), 173–183.
[10] Wiebke Köpp, Anke Friederici, Marco Atzori, Ricardo Vinuesa, Philipp Schlatter, and Tino Weinkauf. 2019. Notes on percolation analysis of sampled scalar fields. In *Topology-Based Methods in Visualization (TopoInVis)*. Nyköping, Sweden.
[11] Aaditya G Landge, Valerio Pascucci, Attila Gyulassy, Janine C Bennett, Hemanth Kolla, Jacqueline Chen, and Peer-Timo Bremer. 2014. In-situ feature extraction of large scale combustion simulations using segmented merge trees. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1020–1031.
[12] Myoungkyu Lee and Robert D. Moser. 2015. Direct numerical simulation of turbulent channel flow up to $Re_\tau \approx 5200$. *Journal of Fluid Mechanics* 774 (July 2015), 395–415. https://doi.org/10.1017/jfm.2015.268
[13] Zarija Lukic. 2019. Nyx cosmological simulation data. https://doi.org/10.21227/k8gb-vq78
[14] Senthilnathan Maadasamy, Harish Doraiswamy, and Vijay Natarajan. 2012. A hybrid parallel algorithm for computing and tracking level set topology. In *2012 19th International Conference on High Performance Computing*. 1–10. https://doi.org/10.1109/HiPC.2012.6507496
[15] A Mascarenhas, RW Grout, Chun Sang Yoo, and JH Chen. 2009. Tracking flame base movement and interaction with ignition kernels using topological methods. In *Journal of Physics: Conference Series*, Vol. 180. IOP Publishing, 012086.
[16] Dmitriy Morozov and Gunther Weber. 2013. Distributed merge trees. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 93–102.
[17] Dmitriy Morozov and Gunther H. Weber. 2014. Distributed contour trees. In *Topological Methods in Data Analysis and Visualization III*. Springer, 89–102. https://doi.org/10.1007/978-3-319-04099-8_6
[18] Arnur Nigmetov and Dmitriy Morozov. 2019. Local-global merge tree computation with local exchanges. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.
[19] Valerio Pascucci and Kree Cole-McLaughlin. 2004. Parallel computation of the topology of level sets. *Algorithmica* 38, 1 (2004), 249–268.
[20] Steve Petruzza, Sean Treichler, Valerio Pascucci, and Peer-Timo Bremer. 2018. Babelflow: An embedded domain specific language for parallel analysis and visualization. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 463–473.
[21] Wathsala Widanagamaachchi, Alexander Jacques, Bei Wang, Erik Crosman, Peer-Timo Bremer, Valerio Pascucci, and John Horel. 2017. Exploring the evolution of pressure-perturbations to understand atmospheric phenomena. In *2017 IEEE Pacific Visualization Symposium (PacificVis)*. 101–110. https://doi.org/10.1109/PACIFICVIS.2017.8031584