Toward Localized Topological Data Structures: Querying the Forest for the Tree

Pavol Klacansky, Attila Gyulassy, Peer-Timo Bremer, and Valerio Pascucci

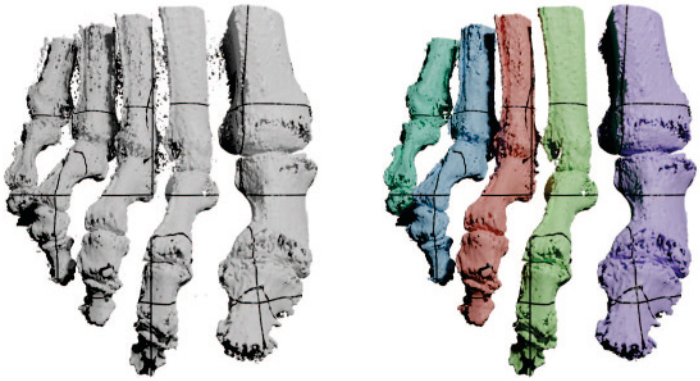


Fig. 1: Our localized *merge forest* data structure partitions the Foot data set, computes local merge trees, and connects them via a *bridge set*. On the left, the entire superlevel set at threshold 84, where the dark lines highlight the cells in the *bridge set* connecting the regions of the domain decomposition and the local trees of the forest. On the right, the *Components* query on the merge forest returns the five largest connected components in the superlevel set, shown in unique colors. The query correctly resolves the connectivity and eliminates the noise.

Abstract—Topological approaches to data analysis can answer complex questions about the number, connectivity, and scale of intrinsic features in scalar data. However, the global nature of many topological structures makes their computation challenging at scale, and thus often limits the size of data that can be processed. One key quality to achieving scalability and performance on modern architectures is data locality, i.e., a process operates on data that resides in a nearby memory system, avoiding frequent jumps in data access patterns. From this perspective, topological computations are particularly challenging because the implied data structures represent features that can span the entire data set, often requiring a global traversal phase that limits their scalability. Traditionally, expensive preprocessing is considered an acceptable trade-off as it accelerates all subsequent queries. Most published use cases, however, explore only a fraction of all possible queries, most often those returning small, local features. In these cases, much of the global information is not utilized, yet computing it dominates the overall response time. We address this challenge for merge trees, one of the most commonly used topological structures. In particular, we propose an alternative representation, the *merge forest*, a collection of local trees corresponding to regions in a domain decomposition. Local trees are connected by a *bridge set* that allows us to recover any necessary global information at query time. The resulting system couples (i) a preprocessing that scales linearly in practice with (ii) fast runtime queries that provide the same functionality as traditional queries of a global merge tree. We test the scalability of our approach on a shared-memory parallel computer and demonstrate how data structure locality enables the analysis of large data with an order of magnitude performance improvement over the status quo. Furthermore, a merge forest reduces the memory overhead compared to a global merge tree and enables the processing of data sets that are an order of magnitude larger than possible with previous algorithms.

Index Terms—Merge tree, parallel computation, topology

1 Introduction

Direct analysis of contemporary scientific data sets remains challenging due to the complexity and high resolution of the data. The extraction, effective analysis, and visualization of complex features require advanced techniques and algorithms. Complex features of interest include ignition kernels in combustion [29], organs in CT scans [11], or galaxies in cosmology [37]. However, these features are not well defined, the data may contain noise, and a range of parameters needs to be explored. In this context, topological techniques provide a rigorous mathematical framework to define and extract features across all thresholds, allowing interactive exploration. Furthermore, topological features can be ranked and simplified by persistence [17] or similar metrics, which enables a multiscale analysis as well as noise removal.

Traditionally, topological analysis has been divided into two stages: a preprocessing stage to compute topological structures, such as the merge tree or the Morse-Smale complex, and an analysis stage, where we use a topological structure to answer queries. Since we do not know which specific queries might be required, we compute a global structure that will accelerate all possible queries. However, not all queries may be necessary for the analysis, and computation of the global information that can accelerate all queries limits the parallelism. Furthermore, constructing and storing the fully resolved global topological structure has a large memory overhead. As a result, the analysis of larger data sets has typically required distributed computation on a supercomputer combined with a partial representation that stores spar-

- *Pavol Klacansky, Attila Gyulassy, and Valerio Pascucci are with the Scientific Computing and Imaging Institute, University of Utah. E-mail: {klacansky,jediati,pascucci}@sci.utah.edu.*
- *Peer-Timo Bremer is with Lawrence Livermore National Laboratory. E-mail: bremer5@llnl.gov.*

Manuscript received 31 Mar. 2019; accepted 1 Aug. 2019.
Date of publication 16 Aug. 2019; date of current version 20 Oct. 2019.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TVCG.2019.2934257

sified global information on each processor [30, 31]. In this paper, we focus on merge trees that accelerate the extraction of locally extremal features. Merge trees have a wide range of applications [29, 45], and their construction has been studied extensively. Parallel algorithms for shared-memory multicore processors [1, 21, 22, 28, 33, 39], distributed memory computers [25, 26, 30–32], and accelerators [12, 36] have been developed. Despite the substantial efforts of the research community, the best shared-memory algorithms on 16 cores still lose more than 40% of their parallel efficiency [22].

We overcome the scalability challenge by using a different split between precomputation and queries. In contrast to building a global merge tree, we decompose the domain, construct a *merge forest* of local trees connected by a local *reduced bridge set*, and compute the necessary global information at query time. The key insight is that, in practice, we rarely query the entire parameter range, and even during an automatic parameter sweep or an interactive exploration, we typically perform only a limited number of queries. Therefore, a fast and entirely local preprocessing coupled with efficient queries will significantly outperform existing solutions. The merge forest is a local structure, and thus its computation scales linearly with the available core counts and the number of regions in the domain decomposition. Although the queries are, in principle, more expensive than those in the merge tree, the merge forest reduces the first time to query by an order of magnitude. Furthermore, our queries are fast enough for an interactive data analysis. Finally, the local representation takes advantage of a smaller index space, which not only halves the memory footprint of the data structure for large data sets, but also improves the locality of the data within each region. A surprising result is that in some cases the nonlocal queries are faster than their equivalent query run on the global merge tree.

In summary, we introduce a new data structure called the *merge forest* that provides analysis capabilities equivalent to those of a global merge tree at a fraction of its computational cost and that can be constructed for significantly larger data sets than previously reported. We make the following contributions:

- A localized data structure, the *merge forest*, to represent topological features traditionally represented with a merge tree;
- A definition of elementary and derived topological queries independent of their underlying topological data structure, which is meant to open opportunities for research into new, nontraditional data structures with equivalent functionalities;
- A set of efficient query algorithms for feature extraction that can be used on a merge forest; and
- An extensive empirical evaluation of the merge forest construction algorithm on structured grids, executed serially and in shared-memory parallel, and a study of the impact of the merge forest representation on the execution time of topological queries.

2 Related Work

The construction of a merge tree (and related contour tree) has been studied extensively. We review the construction and representation of a merge tree and its applications to scientific data analysis.

Representation and construction. A merge tree captures the evolution of superlevel or sublevel sets, and thus it can be used to extract level-set-based features. The merge tree algorithm originates as a subroutine in a contour tree algorithm [10], a generalization and simplification of a 3D sweep algorithm [40]. The merge tree subroutine operates in two stages: first, it sorts the data, and second, it performs a sweep from high- to low-function values while recording maxima and merge saddles by tracking connected components of a superlevel set with a disjoint-set data structure [19]. A merge tree consists of a set of nodes that have pointers to their children and parent nodes. Since the algorithm visits all vertices, it can build an augmented merge tree, and thus enable data segmentation.

Alternatively, an unaugmented merge tree can be computed by identifying critical points locally, then sorting the points, and finally traversing the mesh upward to connect the points with monotone paths [13]. Sorting of critical points offers an advantage over the sweep algorithm due to the output sensitivity with respect to the topological complexity of the data. However, if the query extracts a segmentation - the set of vertices corresponding to a subtree - a mesh traversal is required because the unaugmented tree does not store all vertices. A recent result shows that only critical points along the leaf-root paths need to be sorted [35].

Both the sweep and monotone paths algorithms require the whole data set loaded in memory. Streaming algorithms construct a merge tree by processing a list of vertices and edges in any order and thus need less memory. If an added edge changes the connectivity, the corresponding subtrees are merged [8], resulting in quadratic complexity. Recently, a streaming algorithm based on a new representation of triplets [39], a form of branch decomposition [34], has provided a merge tree construction with linear complexity in practice. Triplets show that changing an underlying representation can result in a significant performance improvement. Additionally, the merge tree can be built using an I/O-efficient union-find algorithm [2].

Parallel algorithms. The parallel merge tree algorithm [33] works by subdividing the domain into regions and combining the merge trees of these regions from the bottom up. The final tree is assembled on a single core, limiting scalability. Moreover, the algorithm computes an unaugmented tree that is not suitable for data segmentation tasks. The domain subdivision approach has been combined with parallel monotone path tracing [28], resulting in a hybrid algorithm [1]. The hybrid algorithm computes an unaugmented merge tree by first building local trees with the parallel monotone path algorithm and then stitching the trees into a global merge tree. The computation per local region scales linearly, but the stitching step exhibits at best a factor of three speed-up and limits the overall scalability of the algorithm.

Instead of subdividing the domain, the function range can be partitioned and an augmented merge tree computed for each cell of the partition. These cells are then stitched to form the global merge tree [20]. However, partitioning the range does not mesh well with the visualization tools that usually load data in regions [24]. Moreover, the range partitioning is susceptible to load imbalance, because the number of boundary simplices can vary greatly.

The limitations of rigid subdivision and the extra work introduced at boundaries have inspired a more flexible task-based algorithm for computing an augmented merge tree [21, 22]. The algorithm creates a task for each arc, starting at maxima, and grows these arcs in parallel until reaching a saddle. The last task that reaches a saddle restarts arc growth. When a single task remains active, the unprocessed arcs, *trunk*, are sorted and processed in parallel. The parallel efficiency reported is 58% on 16 cores, which is the fastest shared-memory parallel approach to date. We use the task-based algorithm as a baseline in our comparisons. Additionally, the flexible streaming algorithm to compute the triplet representation has been parallelized using atomic variables [39]. However, the scaling trails off from two or more cores.

The presence of accelerators motivated a data-parallel merge tree algorithm [12]. The algorithm builds a merge tree by iteratively executing two steps, a monotone path construction and peak pruning, until no peak remains. As an optimization, a new graph can be constructed after the first iteration, which includes only candidate saddles and maxima, leading to improved performance. However, the algorithm then builds an unaugmented merge tree, and a postprocessing step is required to augment the tree. The algorithm executes either on a GPU or a CPU. The reported results are limited to 2D data sets, and the parallel efficiency is 33% on 32 cores. This data-parallel algorithm also has been combined with the domain and range subdivision approaches [36].

Distributed algorithms. Data sets, and the associated merge trees, can get larger than available memory on a single computer. The distributed merge trees [30, 31] present a local-global representation that distributes a merge tree over multiple computers and minimizes the network communication required during queries. Consequently, these queries can be efficiently executed in situ and in parallel. The distributed representation consists of two interconnected data structures stored on each computer, a local merge tree and a sparsified global tree with respect to the local tree. In some cases, the features span

only a limited amount of space, and it is wasteful to compute the sparsified global tree because these features can be extracted by growing regions of fixed size and computing the topology only to the extent of the region [26]. By limiting the global computation to a predefined extent, the scaling of the algorithm has been improved. This algorithm has been further extended to CW complexes [25]. We take inspiration from these approaches with the key difference of constructing only local structures and recovering the necessary global information during query. We focus on shared-memory parallel architectures.

A massive parallel communication (e.g., MapReduce) algorithm [32] computes a merge tree in 2D by first constructing contour trees for the domain partitions on multiple computers, and then using these trees on a single computer as an input graph for a serial merge tree construction with the sweep algorithm.

Applications of a merge tree (and contour tree). The first application of a contour tree accelerated the relief extraction from terrain maps in a radar simulator [6]. The relief is reconstructed by interpolating between nested contours, which are extracted by traversing a contour tree. Several decades later, a contour tree-based search data structure was used to answer path queries with respect to the contours of a terrain [15].

In volumetric data, seed sets [43] were used to accelerate the extraction of an isosurface. Furthermore, a flexible isosurface interface [11] or volume rendering interface [44] enabled interactive exploration of scalar fields by using a contour tree as part of a user interface. By selecting arcs of the contour tree, a user can display contours or subvolumes without the occlusion issues present in traditional isosurface and volume rendering, where two nested objects at the same threshold obscure one another.

The increasing complexity and resolution of the data sets poses a challenge for visualization techniques, because the number of features can be too large to inspect visually. A merge tree can be used to analyze features across all thresholds and simplification levels [17], and extract statistics such as feature count or their size to guide the analysis and visualization process [8, 30]. For example, a merge tree was used to extract and track features, such as extinction regions in combustion simulation [47] or pressure perturbations in weather data [46]. Both use cases rely on the *Components* query to extract, count, and visualize features at different thresholds. Moreover, the user interface utilizes the *MergeTree* query to display an unaugmented merge tree. Another example of merge tree application is extraction and tracking of ignition kernels [29]. First, high values are identified with the *Maxima* query, and the regions around them are extracted while the *Relevance* query values are below a specified relevance threshold. Finally, each of these regions is adjusted if the maximum obtained with *ComponentMax* query is below a threshold.

3 Background

We review the necessary concepts used throughout the paper. As a reference, we use the book Computational Topology - An Introduction [16].

Let K be a simplicial complex with values at vertices. The function g is a piecewise-linear extension of $g(v_i)$ to the simplicial complex K. The function g is generic if all vertices have different values, i.e., $g(v_i) = g(v_j) \Leftrightarrow v_i = v_j$. Simulation of simplicity [18] is used to symbolically perturb the input vertex values to ensure that g is generic.

A simplex $\sigma \in K$ is formed by the convex combination of its vertices, and its faces are simplices derived from proper subsets. The star $St(v)$ of a vertex v is the set of all simplices that have v as a face. The upper star $St^+(v)$ is a subset of the star $St(v)$ containing values above $g(v)$ and the vertex v, i.e., $St^+(v) = \{\sigma \in St(v) \mid u \in \sigma \Rightarrow g(u) \geq g(v)\}$. The link of a vertex v, $Lk(v)$ is the outer boundary of the star, i.e., its closure minus the star itself, $Lk(v) = \overline{St(v)} - St(v)$.

Given a complex K, vertices v_i and v_j are *connected* if there is a path between them in K. The *connected components* of K are the subcomplexes that partition K, such that $K = \bigcup C_k$, where $v_i, v_j \in C_k \iff v_i$ is connected to v_j in C_k. Each component is the maximal subcomplex induced by its vertex set. The number of connected components is denoted $\#CC = \|\{C_1, ..., C_m\}\| = m$. Because the function g is generic, we can order the vertices of K from highest to lowest $\{v_1, v_2, \ldots, v_n\}$, such that $i < j \iff g(v_i) > g(v_j)$. Moreover, since upper stars partition the complex K, an upper star filtration can be defined,

$$\emptyset = K_0 \subset K_1 \subset \cdots \subset K_n = K \tag{1}$$

where $g(v_i) > g(v_{i+1})$ and $K_{i+1} = K_i \cup St^+(v_{i+1})$. The superlevel complex of a scalar value h is $S_h = K_i$, such that $g(v_i) \geq h$ and v_i is the infimum.

A vertex can be classified by its neighborhood as regular or critical. A regular vertex has one connected component in the upper link and one in the lower link. Critical vertices are those that remain. A vertex is called a *maximum* if its upper link is empty, i.e., $Lk^+(v) = \emptyset$, indicating that all adjacent vertices have a lower function value. In the upper star filtration, a maximum vertex v_{i+1} introduces a new connected component in K_{i+1} with respect to the connected components of K_i, and $\#CC(K_{i+1}) = \#CC(K_i) + 1$. A vertex v_{i+1} is a *merge saddle* if K_{i+1} has fewer connected components than K_i, i.e., $\#CC(K_{i+1}) < \#CC(K_i)$. A vertex is a *minimum* if its lower link is empty, i.e., $Lk^-(v) = \emptyset$, and is the *global minimum* if no other vertex has a lower value.

Let X be a topological space and $f : X \to R$ be a scalar function. The *merge tree* is the quotient space $T = X/\sim$, where the equivalence relation $\sim$ is defined by $a, b \in X$, $a \sim b \iff f(a) = f(b)$, and a is connected to b in the superlevel set $S_{f(a)}$. In other words, T glues together all points on the boundaries of connected components of a superlevel set, and the neighborhood of each glued point in T is induced from the neighborhood in X.

For a function g defined on simplicial complex K, the merge tree T captures changes in the connected components of the superlevel complexes of the upper star filtration. Each arc of the tree tracks the growth of a single connected component. The nodes of T correspond to vertices that are a maximum, a merge saddle, or the global minimum.

4 Queries on a Merge Tree

Topology-based analysis seeks to understand the relationships between points of the domain and structural features, and topological data structures are merely intermediaries that distill properties of a function and accelerate answering queries. We capture the information available from a merge tree with elementary and derived queries, irrespective of the underlying data structures.

4.1 Elementary Queries

Three fundamental queries are: 1) find all maxima in a domain; 2) given a point in the domain, determine to which feature it belongs; and 3) find the spatial extent of a topological feature. Independent of a data structure, these three queries can then be used as building blocks for constructing other queries. The first query returns a list of all maxima. The second query returns the identifier of the highest maximum of a connected component of a superlevel complex, when given a vertex and threshold. With the third query, a connected component given a vertex and a threshold, also referred to as a segment, can be obtained.

4.1.1 Maxima Query

The simplest topological query is understanding where new connected components are created.

Definition 1. *We call $\underline{Maxima}(K, g)$ a function that returns a set of vertices that are maxima in the simplicial complex K.*

4.1.2 Component Maximum Query

The maximum of a connected component in a superlevel complex query can be used to test if two vertices are connected in the superlevel complex, useful for feature tracking [47]. We choose the component maximum query as a running example throughout the paper due to its simplicity and because the other queries build on it.

Definition 2. *We call the vertex v_C^* the global maximum of a connected component C if $g(v_C^*) > g(v)$ for any vertex $v \neq v_C^*$ in C.*

Definition 3. *Given a function g defined on a simplicial complex K, a vertex $v \in K$, and threshold h, we call $\underline{ComponentMax(K,g,v,h)}$ the function that returns the v^*_C, where C is the connected component of a superlevel complex S_h such that $v \in C$. If $g(v) < h$, the function returns bottom $\perp$.*

4.1.3 Connected Component Query

Another common query extracts a connected component in a superlevel complex given a vertex and a threshold. For example, we can extract the segmentation corresponding to the subtree given a saddle.

Definition 4. *Let v_i be the lowest vertex such that $g(v_i) \geq h$. We define $\underline{Component(K,g,v,h)}$ as a function that returns the vertices of the connected component containing v from the subcomplex K_i associated with v_i, where i is the index in the upper star filtration (Equation 1).*

4.2 Derived Queries

The elementary queries can be used as subroutines to build more sophisticated queries. We define a collection of derived queries that are commonly used in a topological data analysis, such as extracting all connected components of a superlevel complex.

4.2.1 Connected Components Query

Extraction of connected components, also called *segmentation*, is one of the applications of a merge tree data structure. For example, the connected components query can be used to perform a parameter sweep and count the number of components, their size, or surface area, to determine the appropriate threshold for the analysis [8]. Moreover, the segmentation can be visualized as part of an interactive tool where a user can explore different thresholds in the context of feature tracking [45].

Definition 5. *We call $\underline{Components(K,g,h)}$ a function that, for a superlevel complex S_h, returns a set of connected components.*

Note that the number of components of the superlevel complex S_h is equal to the number of connected components of K_i in the filtration (Equation 1), where i is the index of the lowest valued vertex higher than h. Every pair of vertices $v_i, v_j \in C_k$, in the same connected component $C_k \in Components(K,g,h)$, has the same component maximum, $ComponentMax(K,g,h,v_i) = ComponentMax(K,g,h,v_j)$.

4.2.2 Relevance Query

In many applications, features exist as locally extremal regions relative to their surroundings, and different thresholds are needed for each connected component. For example, an indicator function has different values for vortices of different speeds, and a localized threshold is needed to extract vortices across scales [7]. The relevance metric [29] can be used to locally extract features by a threshold relative to the local maximum of a connected component.

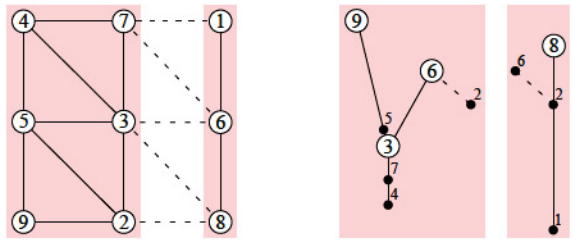
Definition 6. *Given a vertex v and a function g on a simplicial complex K, the relevance query $\underline{Relevance(K,g,v)}$ is defined as a function that evaluates $\frac{g(v^*)-g(v)}{g(v^*)-g_{min}}$, where $v^* = ComponentMax(K,g,v,g(v))$.*

4.2.3 Merge Tree Query

In some cases, a global unaugmented merge tree is visualized, either as part of an interface to control the visualization [11,44], or embedded in the data as a direct visualization of feature relationships [34,42]. In our case, we use the merge tree query to experimentally validate the forest implementation by comparing the merge tree query output with existing implementations.

Definition 7. *Given a simplicial complex K and a generic function g, we call $\underline{MergeTree(K,\ g)}$ a function that outputs an unaugmented merge tree T.*

Once an unaugmented merge tree with N nodes is available, a branch decomposition can be computed in $O(N \log N)$ [34], which allows adding a persistence simplification threshold to all queries.



(a) Two regions and bridge edges. (b) Local trees for the two regions.

Fig. 2: On the left are two local regions (red) with a scalar field (solid edges) connected by the bridge set (dashed edges), with three maxima (7, 9, and 8) and two saddles (6 and 4). On the right are the local trees of the two regions (filled circles are locally regular vertices) and their local copy of reduced bridge-set edges necessary for traversal between the local trees. Note that the boundary restricted maxima [30] are not sufficient due to the lack of a region overlap. For example, the saddle at the vertex with value 6 is not present as a node in either of the two trees (it is either a bridge-set edge end vertex or a regular vertex).

5 Forest Representation

We introduce an intermediate representation called the *merge forest* that is local with respect to a domain partition, yet allows fast computation of the queries. The domain is partitioned, and each region builds and maintains local data structures that are traversed during queries.

Domain decomposition. Consider a simplicial complex K with a set of vertices $V = \{v_1, \ldots, v_n\}$. Let $V_i \subseteq V$; the *region* associated with V_i, denoted M_i, is the maximal subcomplex of K containing only vertices in V_i. We partition V into m subsets $V_1, \ldots, V_m$ such that the associated regions $M_1, \ldots, M_m$ are simply connected. This partition can be obtained, for example, by a recursive bisection on the vertex coordinates. The *global bridge set* is the set of cells not included in any region, i.e., $B = K - \bigcup_{i=1}^{m} M_i$. Note that the intersection of any two regions is empty, $M_i \cap M_j = \emptyset$, and the intersection of a region with the global bridge set is also empty, $M_i \cap B = \emptyset$. The *local bridge set* B_i with respect to a region M_i is the intersection of the global bridge set with all cells incident on M_i. Formally, the local bridge set $B_i = \bigcup_{v \in M_i} St(v) \cap B$. The intersection of local bridge sets of two adjacent regions contains the cells of K that have vertices in both regions, and we use B_{ij} to denote $B_i \cap B_j$.

Forest data structure. The forest data structure is a collection of local data structures associated with each region of a domain decomposition (Fig. 2). A region M_i has associated three components: 1) an augmented merge tree T_i, 2) a map from vertices to arcs of T_i, and 3) a subset of the edges from the local bridge set B_i. The local merge tree T_i, the merge tree of region M_i, is represented as a set of arcs. Each arc has pointers to children arcs and a pointer to a parent arc, allowing navigating the tree in both directions, toward the leaves and toward the root. Additionally, an arc has a pointer to a sorted list of vertices that project to it, called the *arc segmentation*. The first vertex in the list is a maximum or saddle in the local tree. The second component of the forest data structure is a map from each vertex in M_i to the arc in T_i that contains it. The third component is a subset of the edges of the local bridge set B_i that allows navigation between regions. This subset, called the *reduced bridge set* RB_i, locally minimizes the number of edges that must be stored to recover global information when querying the forest. We consider only edges, because the function is piecewise linear, and thus connectivity changes only at vertices [3].

Definition 8. *A $\underline{reduced\ bridge\ set\ RB}$ is defined as a set $RB \subseteq B$, such that, for all indices i in the reduced filtration of $K' = RB \cup M_1 \cup \cdots \cup M_m$ (Equation 1), the number of connected components of K_i and the corresponding K'_i is the same:*

$$\#CC(K_i) = \#CC(K'_i)$$

Note that the number of components is sufficient to show the component equivalence because the vertex set of K and K' is the same, and

removing edges from a bridge set B can only split components, thereby increasing the number of components.

6 Forest Construction

We construct a local tree and find its reduced bridge-set edges independently for each region. Both stages, local tree computation and reduced bridge set construction, build on the sweep algorithm [10]. The sweep algorithm processes vertices in decreasing order, building the filtration of superlevel complexes (Equation 1) one upper star at a time. The individual connected components are maintained during the sweep with a union-find algorithm operating on a disjoint-set data structure [19] that allows logarithmic set membership query [41] (only the path compression is used) as well as joining of sets.

6.1 Local Merge Tree Construction

For each region M_i, a local tree is computed using the sweep algorithm. During the sweep, a vertex to arc map is created. After the local tree is built, the vertex to arc map is used to construct an arc segmentation. The arc segmentation enables output-sensitive computation of connected components of a superlevel complex.

Assuming the neighborhood size is constant, each local tree can be built in $O(r \log r)$, where r denotes the number of vertices in a region. There are $\frac{n}{r}$ regions; thus, the overall complexity to build all local trees is $O(n \log r)$. In contrast, the global tree requires $O(n \log n)$ steps.

6.2 Local Reduced Bridge Set Construction

In addition to a local tree, each region M_i stores a local reduced bridge set $RB_i = \bigcup RB_{ij}$, where M_j is a neighbor of M_i, to allow for an efficient traversal to the neighboring local trees. For each neighboring region M_j, the local reduced bridge set is computed by the function REDUCEDBRIDGESET(B_{ij}, vertices in the closure of B_{ij}, $\overline{B_{ij}}$).

The reduced bridge-set algorithm (Alg. 1), similarly to the merge tree algorithm, performs a sweep through the sorted vertices while maintaining connected components with a disjoint-set data structure. However, instead of iterating over all vertices in an upper link, the edges outside the bridge set are processed first (lines 6-8) to minimize the size of the reduced bridge set. All components in the upper link are then merged, and the edges that have end vertices in different components are recorded (lines 9-12). These edges are added to the reduced bridge set, because they are necessary to maintain the connectivity between regions. We note that the disjoint-set data structure used in the reduced bridge-set algorithm is independent of the disjoint sets used during the local merge tree construction.

In total, there are $\frac{n}{r}$ regions, each containing r vertices. A region has 6 faces of size $\sqrt[3]{r^2}$, 12 edges of size $\sqrt[3]{r}$, and 8 corners of size 1. The faces dominate the number of processed elements. Each region does $O(\sqrt[3]{r^2} \log \sqrt[3]{r^2})$ work, and thus the time complexity of building all local reduced bridge sets is $O(\frac{n}{r} \sqrt[3]{r^2} \log \sqrt[3]{r^2})$.

Algorithm 1 A reduced bridge-set algorithm. The arguments determine if the algorithm computes a minimum reduced bridge set or a local reduced bridge set.

```
1:  function REDUCEDBRIDGESET(bridge set B, vertex set V)
2:      V' ← SORTBYDECREASINGVALUE(V)
3:      RB ← ∅
4:      for i ← 1 to |V'| do
5:          CREATECOMPONENT(v_i)                          ▷ 1. add vertex
6:          for each v_j in Lk+(v_i) do                   ▷ 2. connect inside region
7:              if (v_i, v_j) ∉ B then
8:                  MERGE(COMPONENT(v_i), COMPONENT(v_j))
9:          for each v_j in Lk+(v_i) do                   ▷ 3. connect between regions
10:             if COMPONENT(v_i) ≠ COMPONENT(v_j) then
11:                 MERGE(COMPONENT(v_i), COMPONENT(v_j))
12:                 RB ← RB ∪ (v_i, v_j)
13:     return RB
```

7 Analysis of Reduced Bridge Set Construction

The goal is to prove that the forest data structure computed with the local trees and local reduced bridge sets admits the same superlevel-complex components as the global mesh K. First, we prove that the complex created by the local regions and the global reduced bridge set RB produce the same components as K. Finally, we show that the union of the local reduced bridge sets produced by the REDUCEDBRIDGESET function (Alg. 1) is a superset of the global reduced bridge set, and hence is guaranteed to have the same superlevel-complex components for all thresholds.

Lemma 1 (Reduced bridge set algorithm). *Let RB be the set returned by the function* REDUCEDBRIDGESET*(B, V), given the global bridge set B and all vertices of K, V. Then RB is a reduced bridge set.*

Proof. We prove that RB is a reduced bridge set, i.e., the invariant $\#CC(K_i) = \#CC(K'_i)$ holds at the end of each outer loop iteration (lines 4-12), by induction on the filtration index.

Base case: $i = 0$, $K_0 = K'_0 = \emptyset$, $RB = \emptyset$, and $\#CC(K_0) = \#CC(K'_0) = 0$.

Induction step: By induction, the precondition holds $\#CC(K_{i-1}) = \#CC(K'_{i-1})$. The first step (line 5) adds the vertex v_i and the number of components increases by one, $\#CC(K_{i-1} \cup v_i) = \#CC(K'_{i-1} \cup v_i) = \#CC(K_{i-1}) + 1$. In the second step, the first inner loop (lines 6-8) adds the edges outside B, E_{out}. Since the complexes differ only in the bridge set, i.e., $K \setminus B = K' \setminus B$, it follows $\#CC(K_{i-1} \cup v_i \cup E_{out}) = \#CC(K'_{i-1} \cup v_i \cup E_{out})$. In the third step, the second inner loop (lines 9-12) adds the edges inside a bridge set B that change the number of connected components, E_{in}, because COMPONENT$(v_i) \neq$ COMPONENT(v_j) implies v_i and v_j are not connected. Therefore, $\#CC(K_{i-1} \cup St^+(v_i)) = \#CC(K'_{i-1} \cup v_i \cup E_{out} \cup E_{in})$. Thus the postcondition $\#CC(K_i) = \#CC(K'_i)$ holds.

□

Definition 9. *A minimum reduced bridge set RB^* is such that no reduced bridge set $RB^{*\prime}$ with $|RB^{*\prime}| < |RB^*|$ exists.*

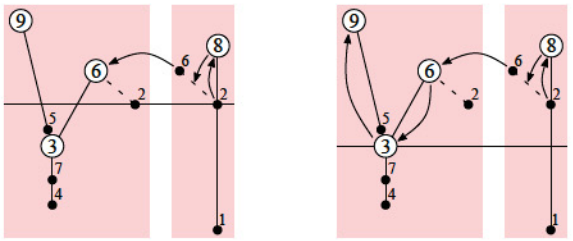
Lemma 2 (Minimum reduced bridge set construction). *The reduced bridge set RB constructed by* REDUCEDBRIDGESET*(B, V) is a minimum reduced bridge set.*

Proof. By the definition of a reduced bridge set, $RB \cap K_1 \subseteq RB \cap K_2 \subseteq \cdots \subseteq RB \cap K_n = RB$. Suppose a set of simplices Y exists, such that $|RB| > |Y|$. Assume for some filtration index i, $|RB \cap K_i| = |Y \cap K_i|$, and $|RB \cap K_{i+1}| > |Y \cap K_{i+1}|$. The function REDUCEDBRIDGESET adds only edges that change the number of connected components (lines 10-12); thus Y is not a reduced bridge set. □

The algorithm for constructing a reduced bridge set (Alg. 1) has the property that if run on any subset of B and the vertices in its closure, it will return a superset of a minimum reduced bridge set restricted to that subset. We use this property to build a local reduced bridge set that allows us to answer global queries.

Lemma 3 (Reduced bridge set for domain subset). *Let B' be the subset of B, arbitrary set $Y \subseteq K$, and RB' the set constructed by* REDUCEDBRIDGESET*$(B',$ vertex set of $\overline{B'} \cup Y)$. Then the set RB' is a superset of $RB \cap B'$.*

Proof. The sweep in REDUCEDBRIDGESET(B, V) implicitly defines an edge sequence E, and its subsequence E^* corresponds to the edges in a minimum reduced bridge set RB. Similarly, the sweep in REDUCEDBRIDGESET$(B',$ vertex set of $\overline{B'} \cup Y)$ implicitly defines an edge sequence E', and its subsequence E'^* corresponds to the edges in the set RB'. We need to show that $E^* \cap B' \subseteq E'^*$, that is, the set RB' contains all edges from RB restricted to the set B', and potentially some extra edges. Suppose there exists an edge $e \in E^* \cap B'$, and thus it connects two components C_a and C_b. Assume $e \notin E'^*$. Then another edge $e' \in E'^*$ before e in the edge sequence must connect C_a and C_b. Then, however, the edge e would not be in $E^* \cap B'$ given the component test (line 10). □



(a) *ComponentMax*$(v=6,h=6)=8$ (b) *ComponentMax*$(v=6,h=4)=9$

Fig. 3: The *ComponentMax* query for a vertex v with function value 6 and two thresholds, 6 and 4. The forest consists of two local trees (red) connected by a local reduced bridge set. The algorithm starts by finding an arc that contains the vertex v using the vertex to arc map, and then it traverses the whole subforest above a given threshold and performs a reduction on the visited leaves. On the left, the threshold is 6, and thus the query returns the vertex 8 (maximum of the leaves 7 and 8). On the right, the threshold is 4, and the query returns the vertex 9 (maximum of leaves 7, 8, and 9), because the saddle 4 is in the connected component.

8 Querying Forest

We now use the forest to accelerate the elementary queries (Sect. 4.1) and derived queries (Sect. 4.2).

8.1 Maxima Query

The query returns all maxima in a data set. It is implemented by iterating over all leaves in each local tree, and returning only the leaves that do not have incident a bridge-set edge with a higher end vertex.

8.2 Component Maximum Query

The *ComponentMax* query returns the global maximum of a connected component in a superlevel complex that contains a given vertex (Fig. 3). Instead of traversing a mesh to find the maximum, a global merge tree (Alg. 2) can be traversed because its leaves represent maxima, and arcs connect at saddle vertices where the connectivity changes. However, in the forest representation, we have a collection of local merge trees connected by reduced bridge-set edges. Thus, compared to global tree traversal, we need to traverse between regions (Alg. 3) and keep track of already visited arcs.

The COMPONENTMAX function (Alg. 2) returns the correct maximum of a component, because the forest has the same components as the underlying complex K. Effectively, COMPONENTMAX performs a depth-first search in the graph defined by the portions of the forest data structure above the given threshold.

Algorithm 2 The algorithm for computing query *ComponentMax* that returns the highest vertex reachable from a given vertex in a superlevel complex.

```
function COMPONENTMAX(function g, forest F, vertex v, threshold
h, visited set VS)
    if g(v) < h or v ∈ VS then
        return ⊥
    VS ← VS ∪ {v}
    arc_v ← VERTEXTOARC(F, v)
    max ← arc_v.maxVertex
    for each arc_n in ARCNEIGHBORS(g, F, arc_v, h) do
        v' ← arc_n.maxVertex
        result, VS ← COMPONENTMAX(g, F, v', h, VS)
        if result ≠ ⊥ and g(result) > g(max) then
            max ← result
    return max, VS
```

Optimizations. The *ComponentMax* query on a forest needs to keep track of visited vertices (or arcs) compared to the global tree, where a predecessor is sufficient. The bookkeeping adds to the cost of the traversal. This cost is minimized by traversing the local tree first and adding only the arcs with reduced bridge-set edges to the visited set. An additional benefit of the optimization is increased cache locality due to local traversal. Altogether, we observe more than a factor of two speed-up over the nonoptimized implementation.

Algorithm 3 A function that returns all neighboring arcs of an arc. The set may include arc's children, parent, and all arcs reachable through the arc's bridge-set edges.

```
function ARCNEIGHBORS(function g, forest F, arc arc_v, threshold
h)
    neighbors ← arc_v.children ∪ {arc_v.parent}
    for each edge in arc_v.bridgeSet do
        if g(edge.localVertex) ≥ h then
            arc_n ← VERTEXTOARC(F, edge.neighborVertex)
            neighbors ← neighbors ∪ {arc_n}
    return neighbors
```

8.3 Connected Component Query

We implement the *Component* query by modifying the COMPONENTMAX function (Alg. 2) to collect the arc's vertices during the forest traversal. For arcs that intersect the threshold, only vertices above the threshold are collected. Compared to the *ComponentMax* query, the forest traversal cost is, in practice, many times smaller than the cost associated with copying the vertices to the output array.

8.4 Relevance Query

Instead of traversing the complex K to compute the *Relevance* query, we use the forest-accelerated function COMPONENTMAX. This function is called with the queried vertex and its function value, and the returned maximum is used in the relevance metric equation (Def. 6).

8.5 Merge Tree Query

One of the main bottlenecks in constructing the augmented tree is the need to maintain a list of vertices per arc to support the extraction of connected components. Fortunately, such an augmented tree is not necessary, because we have the *Component* query on the forest, and thus we compute the unaugmented global tree that contains only critical vertices. For example, extracting a superlevel-complex component corresponding to a subtree in the global merge tree is performed by the *Component* query using a merge saddle and its function value.

We query the forest for the tree using the sweep algorithm [10] on the forest graph. First, all local arcs and reduced bridge-set edges are collected into an array. The local bridge-set edges that have a lower end vertex in other region are ignored, because the other region will add that edge. Additionally, the higher of the two end vertices is pointed directly to the corresponding arc's highest vertex, reducing the number of edges in the array by about 10% and leading to a 10-20% speed-up. Then, the array is sorted and a sweep is performed to identify global arcs.

9 Results

We evaluate the forest construction time and its scaling on a multicore computer and test the performance impact of the forest representation on queries. Both the construction and queries depend on a chosen region size, and we focus our attention on the region size first. Then, we evaluate the parallel scaling of the construction and conclude with a comparison of queries on a forest with those on a global tree.

Topological data structures, including a merge tree, are output sensitive, i.e., the construction and query time is impacted not only by the data size, but also by its topological complexity. Therefore, we use a wide variety of data sets (Table 1), such as imaged scans (Foot, Vertebra, Neurons) and simulations (Magnetic [23], HCCI [4], TJ [5], Miranda [14], DNS [27]). Some data sets have mostly local features (TJ), but some have long, thin features that span a large portion of the data set (Vertebra). We choose these data sets to test the overhead imposed by the region decomposition, during both precomputation and the queries. For the data sets HCCI, TJ, and Neurons, a subset with a

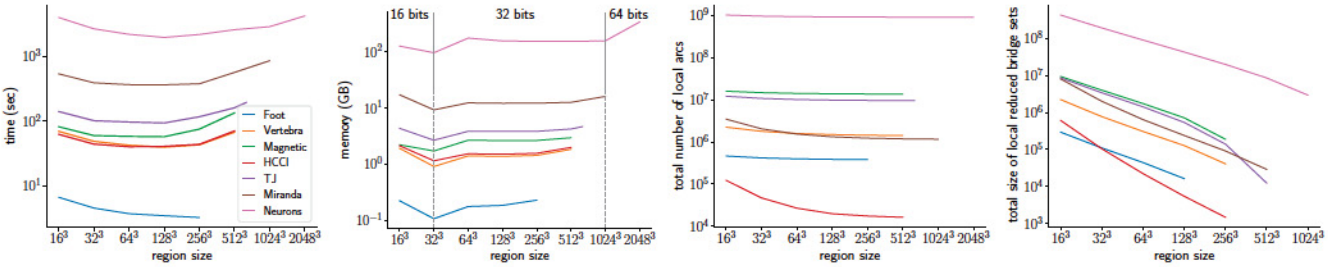


Fig. 4: The region size affects the performance of the serial forest construction. Due to better locality and a smaller log factor from the sort, we can build the forest faster than the global tree. Moreover, we use a smaller index data type to reduce the memory needed to store the forest, i.e., the two jumps, from 32^3 to 64^3 and from 1024^3 to 2048^3, are caused by switching to a larger index type. We do not split arcs in local trees with bridge-set end vertices, and thus the cost of the domain decomposition is mainly reflected in the total size (number of edges) of the local reduced bridge sets.

Table 1: A list of data sets used for the evaluation. The selected data sets have a wide range of sizes and topological complexity.

Data set	Resolution	Size (GB)	Arc count
Foot	256x256x256	0.01	380,654
Vertebra	512x512x512	0.25	1,419,062
Magnetic	512x512x512	0.5	13,661,086
HCCI	512x512x512	0.5	16,342
TJ	512x1024x512	1	9,619,909
Miranda	1024x1024x1024	4	1,162,520
Neurons	2048x2048x2048	16	910,677,279
DNS	10240x7680x1536	900	138,601,501

resolution that is a multiple of the region resolution is extracted. Data sets are represented as structured grids and are implicitly triangulated.

In the comparison with the global augmented tree construction, we use FTC [22] and TTK [42]. We show that forfeiting the goal of computing the global tree results in an order of magnitude speed-up in precomputation with a small impact on queries. All algorithms use the same six-subdivision neighborhood and build a forest or a superlevel-complex merge tree (the split tree setting in FTC and TTK).

All tests are run on a machine with four Intel Xeon CPU E7-8890 v3 @ 2.5 GHz (18 cores each) and 3 TB RAM (1.6 GT/s) running CentOS Linux 7.5. The compiler used is GCC 7.3.1 with optimizations enabled (-O3), and FTC and TTK 0.9.7 are compiled with additional libraries Boost 1.69.0 and VTK 8.2.0. All reported precomputation times are a mean of 3 runs with a coefficient of variation (standard deviation divided by mean) less than 15% for forest and 30% for FTC and TTK. Some of the runs with FTC did not complete successfully, and thus can have a mean of less than three runs. The queries are run serially.

9.1 Forest Construction

The forest algorithm computes, for each region in a domain partition, a local tree and a local reduced bridge set. The region size determines the size of a partition, and we focus on exploring its impact on the serial construction time. From the complexity analysis (Sect. 6), we know that local trees take $O(n \log r)$ and local reduced bridge sets take $O(\frac{n}{r}\sqrt[3]{r^2}\log\sqrt[3]{r^2})$ time, and thus both depend directly on the region size. Moreover, data locality can play a role in the forest construction performance, especially for larger data sets that do not fit into a cache, because modern processors rely on several layers of a cache hierarchy to minimize the latency of memory accesses.

We vary the region size (Fig. 4) and also the necessary index type. For example, a region size 32^3 can use 16-bit integers as indices, but a size 64^3 to 1024^3 requires 32 bits, and with size 2048^3, 64-bit indices are required. The use of 32-bit indices halves the memory overhead of the vertex to arc map and arc segmentation compared to 64-bit indices, which allows us to analyze the DNS data set.

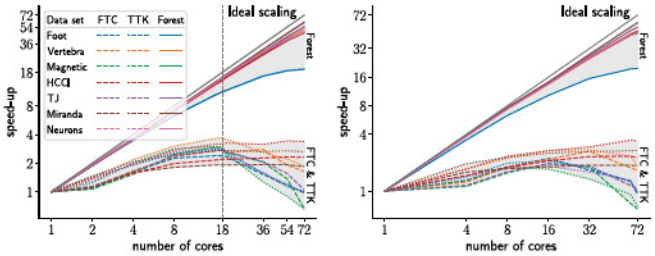
The spectrum of region sizes has two extremes, a region with a single vertex and a region with all vertices. For practical reasons, we start with the region size 16^3. As we sweep the region size range, we observe the serial construction reaching a minimum around size 64^3 and 128^3. The computation of the global tree (the region size is the same as the data size) is slower than that of the forest, due to both algorithmic complexity of sorting and worse data locality. The locality of the global sort is improved by duplicating the data set and combining each vertex with its value to avoid an indirection to read the scalar value from the data set. However, duplication may not always be possible due to limited memory capacity. Since the region size is much smaller than the data size, the optimization has negligible memory overhead.

9.2 Parallel Forest Construction and Comparison with Parallel Merge Tree Construction

The serial algorithm iterates over all regions and processes them independently, and the natural next step is to execute the iteration in parallel. In parallel tests, we use a region size 64^3 because it minimizes the time per region for most of the data sets (Fig. 4) and still provides a sufficient amount of work for 72 cores. An exception is the Foot data set with only 64 regions. Moreover, the larger number of small regions reduces the load imbalance.

The parallel tests are run in two configurations by setting the OpenMP environment variables

- *close* with `OMP_PROC_BIND=close` and `OMP_PLACES=cores`
- *spread* with `OMP_PROC_BIND=spread` and `OMP_PLACES=cores`



(a) Scaling with *close* configuration. (b) Scaling with *spread* configuration.

Fig. 5: On the left, scaling with the *close* configuration (a single socket left of the stippled vertical line). FTC and TTK do not scale over multiple sockets, whereas the forest scales linearly. The exception is the Foot data set that has only 64 regions, which limits its scalability and increases load imbalance. On the right, scaling with the *spread* configuration shows that the forest benefits from more cache available across sockets and is more robust to the system configuration.

The *close* runs start with 1 core, and then 2, 4, 8, 16, and 18 cores, all on the same socket (a workstation-like configuration). Then one socket is added at a time (19 and 36, 54, and 72 cores) to explore the nonuniform memory access effects on scaling. However, the processor cache is underutilized with the *close* configuration, and thus we also

Table 2: Construction times (in seconds) of a merge tree with FTC and TTK and a forest (region size 64^3). We run these scaling tests in the *close* configuration (environment variables `OMP_PROC_BIND=close` and `OMP_PLACES=cores`) to force the 18 core runs to reside on a single socket, because FTC and TTK are designed for a workstation. The forest is an order of magnitude faster than FTC and TTK and scales to 72 cores. The missing results did not run successfully with FTC and TTK, and the DNS data set took excessive time for the forest on a single core.

	FTC (sec)			TTK (sec)			Forest (sec)		
Data set	1 core	18 cores	72 cores	1 core	18 cores	72 cores	1 core	18 cores	72 cores
Foot	7.8	3.3	8.0	9.8	3.5	9.6	3.6	0.3	0.2
Vertebra	69.2	25.3	42.1	103.1	28.0	55.5	40.1	2.6	0.8
Magnetic	197.1	67.2	294.2	235.4	81.0	356.1	54.5	3.6	1.2
HCCI	60.3	27.4	26.1	96.3	30.3	28.5	37.0	2.5	0.8
TJ	234.2	86.6	223.2	299.5	104.5	307.4	89.4	5.8	1.7
Miranda	764.2	398.4	416.0	1204.2	442.8	457.3	340.8	21.4	5.6
Neurons	-	-	-	-	-	-	2097.3	133.5	44.2
DNS	-	-	-	-	-	-	-	2667.4	750.1

run the tests in a *spread* mode, which maximizes the available cache by evenly distributing the work across sockets (1, 4, 8, 16, 32, 64, and 72 cores). We compare the scalability of the forest construction with the shared-memory augmented merge tree construction algorithms, FTC and TTK. All reported times exclude the I/O and measure the time from the start of the precomputation until we are able to run the first query. We were unable to process the Neurons and DNS data sets with FTC or TTK. The DNS data set is run only in the *close* configuration at 18 and 72 cores due to excessive runtime.

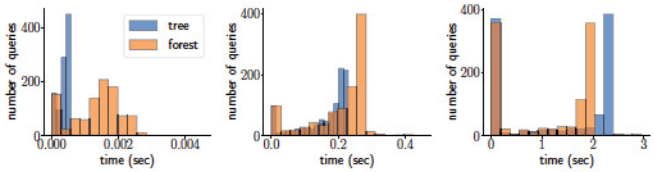
We start with the *close* runs (Table 2) and observe forest construction times under 10 seconds for most of the data sets on 18 cores (a workstation scenario). Furthermore, if all 72 cores are employed, only the DNS data set cannot be preprocessed under a minute. Moreover, the linear scaling of the forest construction exhibits a mean 87% parallel efficiency on 8 cores, 82% on 18 cores, and 65% on 72 cores. The decrease in parallel efficiency on 72 cores can be attributed to greater load imbalance on the small data set (Foot) and increased resource contention. Furthermore, the *spread* configuration increases the parallel efficiency to 92% on 8 cores because it maximizes the cache utilization (Fig. 5). In contrast to the forest parallel efficiency, FTC achieves a mean 27% parallel efficiency on 8 cores and 14% on 18 cores, and the scaling trails off on more cores. Similarly, TTK has a mean parallel efficiency of 34% on 8 cores and 17% on 18 cores.

The memory overhead of a construction algorithm and data structure limits the size of data sets that can be processed. Due to the localized organization of the forest, the overhead during construction is negligible compared to the global approach (Table 3). Moreover, 32-bit indices are used to halve the size of the data structures needed to support the data segmentation. The low computation overhead and smaller data structure combined allow us to analyze the large DNS data set.

Overall, the linear scaling and reduced memory footprint allow us to process up to 10 times larger data than previously reported.

Table 3: Memory usage (in GB) of FTC and TTK and the forest, measured as the peak resident set with the *time* utility. The forest construction uses an order of magnitude less memory compared to the construction of the global augmented tree, enabling the analysis of larger data sets. The reported memory usage is on 18 cores with the coefficient of variation less than 30% between different core counts.

Data set	FTC (GB)	TTK (GB)	Forest (GB)
Foot	2.8	2.5	0.2
Vertebra	20.6	18.0	1.5
Magnetic	39.7	38.0	2.7
HCCI	17.9	15.0	1.6
TJ	51.6	47.0	3.9
Miranda	143.4	122.0	12.4
Neurons	-	-	161.3
DNS	-	-	1835.7



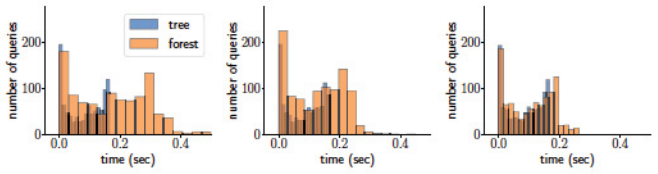
(a) HCCI data set. (b) Vertebra data set. (c) Magnetic data set.

Fig. 6: Distributions of execution times for 1000 *ComponentMax* queries on the forest with region size 64^3 and the global tree. The query vertices are selected randomly and use thresholds equal to the function value at a given vertex. Generally, we observe that the forest-query histograms are scaled to the right compared to the global-tree-query histograms, as expected. The queries on the far right require traversal of a large portion of the forest, because the superlevel complex includes almost the entire data set. Furthermore, this difference indicates that for queries traversing small components, the running time is effectively the same between forest and tree. However, responding to queries that require traversing virtually the entire data set may incur a measurable overhead. Fortunately, this tends to happen only for query vertices in the noise/background of the data that are, therefore, not interesting for many applications where the range of features is known a priori. We conclude that the proposed change in data structure is a good choice in most practical situations.

9.3 Component Maximum Query

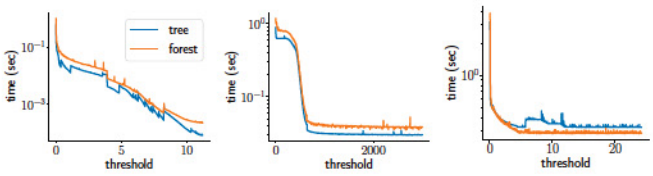
We now demonstrate that queries on a forest perform comparably to those on a global tree, and any slowdown is acceptable for the interactive data analysis that is typical for a visualization environment. The main difference between the global merge tree and a forest is that the forest contains the bridge-set edges, and these edges increase the cost of the traversal during a query. Because the region size is an input parameter, we can develop different trade-offs by shifting more cost to the precomputation by increasing the region size, which reduces the cost of traversing the bridge set at runtime. Note that the global merge tree is a special case of a forest with a single region. Therefore, we compare the forest to the query on the global tree (no bridge-set edges needed) and explore the impact of region size on the query time.

We evaluate the query performance by randomly sampling 1000 vertices in the domain and by running the query with a threshold set to the function value of a given vertex. We run the query on a forest and global tree, and for each run we compute the distribution by splitting the elapsed time range into 16 bins and counting the number of queries that fall into each bin. From the experiments, we observe the performance of the forest representation is comparable to that of the global tree (Fig. 6), with a surprising result for the Magnetic and TJ data sets, where the forest outperforms the tree despite the extra overhead imposed by the local reduced bridge set. These data sets have an order of magnitude more arcs than other similarly sized data sets, and thus the better locality of the traversal outweighs the extra overhead imposed by the merge



(a) Region size 64^3. (b) Region size 128^3. (c) Region size 256^3.

Fig. 7: The *ComponentMax* query on the Miranda data set with varying region size on the global tree and forest. We clamped the histogram (a) after 0.5 second. As the region size increases, the overhead decreases, and the absolute time difference is negligible and well within interactive time. We expect most practical queries to be in the left to the middle of the histogram (the ones on the right traverse the whole forest).



(a) HCCI data set. (b) Vertebra data set. (c) Magnetic data set.

Fig. 8: Measuring the query time (log scale) as a function of threshold for the *Components* query on the forest with region size 64^3 and the global tree. In each case, a threshold of 0 extracts every data point. We observe the queries execute in comparable time, and more topologically complex data sets perform better with the forest.

forest. The absolute time differences at region size 64^3 are less than 100 milliseconds and are unlikely to impact an interactive data analysis.

We look closer at the impact of the region size on the Miranda data set (Fig. 7), because it is the largest data set we use to test queries, and the number of regions is higher than the number of cores even for larger regions. For the region size 64^3, the *ComponentMax* queries are 2.1 times slower, but at the region size 256^3, we get only a 1.3x slowdown.

9.4 Connected Component Query

The connected component query needs to collect vertices to construct a segment, which serves as an opportunity to amortize the overhead of the forest traversal. We test the query by extracting the components of 1000 superlevel complexes evenly spaced in the range. The first superlevel complex has a single component containing all vertices in the domain.

We observe times to extract the components using the forest that are similar to the times using the global tree (Fig. 8). The difference between the query times is, in most cases, on the order of 10 milliseconds, which is acceptable for an interactive analysis. Furthermore, when the topological complexity increases, the forest outperforms the global tree due to better data locality. Similarly to the *ComponentMax* query, increasing the region size narrows the gap between the forest and the global tree, both the speed-ups and the slowdowns. For example, in the largest tested data set (Miranda), the worst case slowdown is 2.8x at the region size 64^3, 1.7x at 128^3, and only 1.2x at 256^3.

9.5 Merge Tree Query

The *MergeTree* query returns a global unaugmented merge tree. Compared to the global approach where the tree is readily available, the forest requires an additional computation. The query time increases with the input graph size (Table 4) and on most of the data sets is below 10 seconds. However, for the Neurons data set, the query takes several minutes, further corroborating our pursuit of localized data structures.

Table 4: The *MergeTree* query times for all data sets with the input graph size in millions. An unaugmented merge tree can be constructed in a few seconds for most of the data sets. However, the more topologically complex data sets (Magnetic, Neurons) take an order of magnitude longer than the forest preprocessing.

	Region size 64^3		Region size 128^3	
Data set	Graph size	Time (sec)	Graph size	Time (sec)
Foot	0.64 M	0.25	0.60 M	0.24
Vertebra	2.69 M	1.10	2.37 M	1.04
Magnetic	23.19 M	12.31	21.75 M	10.47
HCCI	0.06 M	0.03	0.04 M	0.02
TJ	16.72 M	8.52	15.38 M	7.12
Miranda	2.95 M	1.14	2.24 M	0.91
Neurons	1502.76 M	879.42	1435.34 M	816.29
DNS	374.23 M	546.53	273.88 M	446.89

10 Limitations

The forest construction requires an input parameter, the region size, which may be challenging to choose appropriately a priori. Even though our experiments suggest the region size 64^3 is a reasonable default for the precomputation, the region size choice affects the performance of the queries (Fig. 7). If we had a way of determining that a region size is not appropriate given the number of expected queries to be executed, we could quickly recompute the forest with a larger region size. However, it is unclear what a good technique would be to judge if such recomputation is beneficial.

Persistence simplification enables the reduction of noise by removing features with low persistence; however, the presented queries do not support such a simplification directly. Currently, we need to compute the unaugmented global tree with the *MergeTree* query first, forming a bottleneck to the scalability of analysis. Moreover, the queries presented are executed serially, and thus underuse the resources available. Additionally, 1000 queries may be insufficient to adequately sample the *ComponentMax* query parameter space.

The forest representation is limited to a simplicial complex, because we assume that critical points are vertices inside a region. For example, a trilinear interpolant can cause critical points to be inside a face or a cell [33], and thus potentially in a bridge set. The simplicial complex requirement forces an implicit triangulation of a grid cell, potentially changing its topology [9].

11 Conclusion

We present the merge forest, a localized data structure that processes each region in the domain decomposition in parallel and with linear scaling. We define a set of queries useful for data analysis and describe algorithms that can execute them quickly using the forest. Through an extensive evaluation, we confirm that these queries are only marginally slower on the forest compared to the global tree, and in some cases, even faster. The linear scaling and fast queries combined enable analysis of an order of magnitude larger data sets than previously possible.

As future work, we would like to explore the applicability of localized data structures to different topological structures, such as the Morse-Smale complex. Moreover, an extension of the merge forest to distributed-memory computers would be interesting, and it remains an open question if the queries can execute quickly despite the communication cost. Another possibility is to evaluate the forest construction and queries on a wide variety of unstructured meshes. Especially interesting would be a study of the impact of different mesh partitioning schemes [38] and mesh layouts [48] on the forest performance.

Acknowledgments

The authors wish to thank reviewers, Amy Gooch, Duong Hoang, John Holmen, and Christine Pickett. The Neurons data set was provided by Alessandra Angelucci and Frederick Federer. This work was supported in part by NSF:CGV Award: 1314896, NSF:IIP Award: 1602127, NSF:ACI Award:1649923, DOE/SciDAC DESC0007446, PSAAP CCMSC DE-NA0002375, NSF:OAC Award: 1842042, and Intel Graphics and Visualization Institutes of XeLLENCE program.

REFERENCES

[1] A. Acharya and V. Natarajan. A parallel and memory efficient algorithm for constructing the contour tree. In *2015 IEEE Pacific Visualization Symposium (PacificVis)*, pp. 271–278, Apr. 2015. doi: 10.1109/PACIFICVIS.2015.7156387

[2] P. K. Agarwal, L. Arge, and K. Yi. I/O-efficient batched union-find and its applications to terrain analysis. *ACM Trans. Algorithms*, 7(1):11:1–11:21, Dec. 2010. doi: 10.1145/1868237.1868249

[3] T. F. Banchoff. Critical points and curvature for embedded polyhedral surfaces. *The American Mathematical Monthly*, 77(5):475–485, May 1970. doi: 10.2307/2317380

[4] G. Bansal, A. Mascarenhas, and J. H. Chen. Direct numerical simulations of autoignition in stratified dimethyl-ether (DME)/air turbulent mixtures. *Combustion and Flame*, 162(3):688–702, 2015. doi: 10.1016/j.combustflame.2014.08.021

[5] A. Bhagatwala, Z. Luo, H. Shen, J. A. Sutton, T. Lu, and J. H. Chen. Numerical and experimental investigation of turbulent DME jet flames. *Proceedings of the Combustion Institute*, 35(2):1157–1166, 2015. doi: 10.1016/j.proci.2014.05.147

[6] R. L. Boyell and H. Ruston. Hybrid techniques for real-time radar simulation. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS '63 (Fall), pp. 445–458. ACM, New York, NY, USA, 1963. doi: 10.1145/1463822.1463869

[7] P.-T. Bremer, A. Gruber, J. C. Bennett, A. Gyulassy, H. Kolla, J. H. Chen, and R. W. Grout. Identifying turbulent structures through topological segmentation. *Commun. Appl. Math. Comput. Sci.*, 11(1):37–53, 2016. doi: 10.2140/camcos.2016.11.37

[8] P.-T. Bremer, G. H. Weber, J. Tierny, V. Pascucci, M. S. Day, and J. B. Bell. Interactive exploration and analysis of large-scale simulations using topology-based data segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 17(9):1307–1324, Sept. 2011. doi: 10.1109/TVCG.2010.253

[9] H. Carr, T. Möller, and J. Snoeyink. Artifacts caused by simplicial subdivision. *IEEE Transactions on Visualization and Computer Graphics*, 12(2):231–242, Mar. 2006. doi: 10.1109/TVCG.2006.22

[10] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. *Computational Geometry*, 24(2):75–94, 2003. Special Issue on the Fourth CGC Workshop on Computational Geometry. doi: 10.1016/S0925-7721(02)00093-7

[11] H. Carr, J. Snoeyink, and M. van de Panne. Flexible isosurfaces: Simplifying and displaying scalar topology using the contour tree. *Comput. Geom. Theory Appl.*, 43(1):42–58, Jan. 2010. doi: 10.1016/j.comgeo.2006.05.009

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

[13] Y.-J. Chiang, T. Lenz, X. Lu, and G. Rote. Simple and optimal output-sensitive construction of contour trees using monotone paths. *Computational Geometry*, 30(2):165–195, 2005. Special Issue on the 19th European Workshop on Computational Geometry. doi: 10.1016/j.comgeo.2004.05.002

[14] A. W. Cook, W. Cabot, and P. L. Miller. The mixing transition in Rayleigh-Taylor instability. *Journal of Fluid Mechanics*, 511:333–362, 2004. doi: 10.1017/S0022112004009681

[15] M. de Berg and M. van Kreveld. Trekking in the alps without freezing or getting tired. *Algorithmica*, 18(3):306–323, July 1997. doi: 10.1007/PL00009159

[16] H. Edelsbrunner and J. L. Harer. *Computational Topology: An Introduction*, vol. 69. American Mathematical Society, 2010. doi: 10.1090/mbk/069

[17] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. *Discrete & Computational Geometry*, 28(4):511–533, Nov. 2002. doi: 10.1007/s00454-002-2885-2

[18] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9(1):66–104, Jan. 1990. doi: 10.1145/77635.77639

[19] B. A. Galler and M. J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, May 1964. doi: 10.1145/364099.364331

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

[21] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Task-based augmented merge trees with Fibonacci heaps. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 6–15, Oct. 2017. doi: 10.1109/LDAV.2017.8231846

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

[23] F. Guo, H. Li, W. Daughton, and Y.-H. Liu. Formation of hard power laws in the energetic particle spectra resulting from relativistic magnetic reconnection. *Phys. Rev. Lett.*, 113:155005, Oct. 2014. doi: 10.1103/PhysRevLett.113.155005

[24] M. Hadwiger, A. K. Al-Awami, J. Beyer, M. Agus, and H. Pfister. SparseLeap: Efficient empty space skipping for large-scale volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):974–983, Jan. 2018. doi: 10.1109/TVCG.2017.2744238

[25] A. G. Landge, P.-T. Bremer, A. Gyulassy, and V. Pascucci. Notes on the distributed computation of merge trees on CW-complexes. In *Topological Methods in Data Analysis and Visualization IV*, pp. 333–348. Springer International Publishing, Cham, 2017. doi: 10.1007/978-3-319-44684-4_20

[26] A. G. Landge, V. Pascucci, A. Gyulassy, J. C. Bennett, H. Kolla, J. Chen, and P.-T. Bremer. In-situ feature extraction of large scale combustion simulations using segmented merge trees. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pp. 1020–1031. IEEE Press, Piscataway, NJ, USA, 2014. doi: 10.1109/SC.2014.88

[27] M. Lee and R. D. Moser. Direct numerical simulation of turbulent channel flow up to $Re_\tau \approx 5200$. *Journal of Fluid Mechanics*, 774:395–415, July 2015. doi: 10.1017/jfm.2015.268

[28] S. Maadasamy, H. Doraiswamy, and V. Natarajan. A hybrid parallel algorithm for computing and tracking level set topology. In *2012 19th International Conference on High Performance Computing*, pp. 1–10, Dec. 2012. doi: 10.1109/HiPC.2012.6507496

[29] A. Mascarenhas, R. W. Grout, C. S. Yoo, and J. H. Chen. Tracking flame base movement and interaction with ignition kernels using topological methods. *Journal of Physics: Conference Series*, 180:012086, July 2009. doi: 10.1088/1742-6596/180/1/012086

[30] D. Morozov and G. H. Weber. Distributed merge trees. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pp. 93–102. ACM, New York, NY, USA, 2013. doi: 10.1145/2442516.2442526

[31] D. Morozov and G. H. Weber. Distributed contour trees. In *Topological Methods in Data Analysis and Visualization III*, pp. 89–102. Springer, 2014. doi: 10.1007/978-3-319-04099-8_6

[32] A. Nath, K. Fox, P. K. Agarwal, and K. Munagala. Massively parallel algorithms for computing TIN DEMs and contour trees for large terrains. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPACIAL '16, pp. 25:1–25:10. ACM, New York, NY, USA, 2016. doi: 10.1145/2996913.2996952

[33] V. Pascucci and K. Cole-McLaughlin. Parallel computation of the topology of level sets. *Algorithmica*, 38(1):249–268, Jan. 2004. doi: 10.1007/s00453-003-1052-3

[34] V. Pascucci, K. Cole-McLaughlin, and G. Scorzelli. *The TOPORRERY: computation and presentation of multi-resolution topology*, pp. 19–40. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi: 10.1007/b106657_2

[35] B. Raichel and C. Seshadhri. Avoiding the global sort: A faster contour tree algorithm. *Discrete & Computational Geometry*, 58(4):946–985, Dec. 2017. doi: 10.1007/s00454-017-9901-z

[36] P. Rosen, J. Tu, and L. A. Piegl. A hybrid solution to parallel calculation of augmented join trees of scalar fields in any dimension. *Computer-Aided Design and Applications*, 15(4):610–618, Jan. 2018. doi: 10.1080/16864360.2017.1419648

[37] N. Shivashankar, P. Pranav, V. Natarajan, R. van de Weygaert, E. G. P. Bos, and S. Rieder. Felix: A topology based framework for visual exploration of cosmic filaments. *IEEE Transactions on Visualization and Computer Graphics*, 22(6):1745–1759, June 2016. doi: 10.1109/TVCG.2015.2452919

[38] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2):135–148, 1991. Special Issue

on the Parallel Methods on Large-scale Structural Analysis and Physics Applications. doi: 10.1016/0956-0521(91)90014-V

[39] D. Smirnov and D. Morozov. Triplet merge trees. In *Workshop on Topology-based Methods in Visualization (TopoInVis)*, Feb. 2017.

[40] S. P. Tarasov and M. N. Vyalyi. Construction of contour trees in 3D in $O(n \log n)$ steps. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, SCG '98, pp. 68–75. ACM, New York, NY, USA, 1998. doi: 10.1145/276884.276892

[41] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, Mar. 1984. doi: 10.1145/62.2160

[42] 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, Jan. 2018. doi: 10.1109/TVCG.2017.2743938

[43] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, SCG '97, pp. 212–220. ACM, New York, NY, USA, 1997. doi: 10.1145/262839.269238

[44] G. H. Weber, S. E. Dillard, H. Carr, V. Pascucci, and B. Hamann. Topology-controlled volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):330–341, Mar. 2007. doi: 10.1109/TVCG.2007.47

[45] W. Widanagamaachchi, C. Christensen, P.-T. Bremer, and V. Pascucci. Interactive exploration of large-scale time-varying data using dynamic tracking graphs. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 9–17, Oct. 2012. doi: 10.1109/LDAV.2012.6378962

[46] W. Widanagamaachchi, A. Jacques, B. Wang, E. Crosman, P.-T. Bremer, V. Pascucci, and J. Horel. Exploring the evolution of pressure-perturbations to understand atmospheric phenomena. In *2017 IEEE Pacific Visualization Symposium (PacificVis)*, pp. 101–110, Apr. 2017. doi: 10.1109/PACIFICVIS.2017.8031584

[47] W. Widanagamaachchi, P. Klacansky, H. Kolla, A. Bhagatwala, J. Chen, V. Pascucci, and P.-T. Bremer. Tracking features in embedded surfaces: Understanding extinction in turbulent combustion. In *2015 IEEE 5th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 9–16, Oct. 2015. doi: 10.1109/LDAV.2015.7348066

[48] S.-E. Yoon and P. Lindstrom. Mesh layouts for block-based caches. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1213–1220, Sept. 2006. doi: 10.1109/TVCG.2006.162