A Study of the Locality of Persistence-Based Queries and Its Implications for the Efficiency of Localized Data Structures

Pavol Klacansky* SCI Institute University of Utah
Attila Gyulassy† SCI Institute University of Utah
Peer-Timo Bremer‡ LLNL
Valerio Pascucci§ SCI Institute University of Utah

2022 IEEE 15th Pacific Visualization Symposium (PacificVis) | 978-1-6654-2335-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/PACIFICVIS53943.2022.00021

Abstract

Scientific datasets are often analyzed and visualized using isosurfaces. The connected components at or above the isovalue defining these isosurfaces are called superlevel-set components. The vertex set of these superlevel-set components can be used to compute local statistics, such as mean temperature or histogram per component, or to segment the data. However, in datasets produced by acquisition devices or simulations, noise induces many spurious components that clutter the visualization and analysis results. Many of these spurious components would disappear if the data values were slightly adjusted. The notion of persistence captures the stability of a component with respect to function value changes, and so we are interested in computing persistence quickly. Locality of computation is critical for parallel scalability, minimization of communication in a distributed environment, or an out-of-core processing. The recently introduced merge forest attained high performance by exploiting locality, thereby avoiding communication until needed to resolve a feature query. We extend the merge forest to support persistence-based queries and study the locality of these queries by evaluating the traversals of regions of data during a query. We confirm that the majority of evaluated datasets have the property that the noise is mostly local, and thus can be efficiently eliminated without performing a global analysis. Finally, we compare the query running times with those of a triplet merge tree because a triplet merge tree answers all proposed queries in constant time and can be constructed from a merge tree in linear time.

Index Terms: Human-centered computing—Visualization—Visualization application domains—Scientific visualization

1 Introduction

Topological analysis techniques enable robust extraction of features from data. Alongside accelerating the extraction of connected components above a threshold, these techniques enable removal of components that are unstable under small function perturbation. Examples of these noisy components may be artifacts in computed tomography scans or insignificant clusters in cosmology simulations. Furthermore, topological analysis can define the importance of a component relative to its neighboring components and, for example, capture vortices across scales [3]. The stability of components and the choice of simplification threshold can be determined using summaries, such as a persistence diagram or persistence curve.

On the one hand, we have general purpose topological data structures that support both simplification and ranking of components, extraction of connected components, or computation of the relative importance of components. Unfortunately, construction of these data structures exhibits limited scalability [6, 10, 22]. On the other hand, specialized algorithms can exploit the locality of noise [15] but are limited to noise elimination or construction of partial summaries. To access the full capabilities, these specialized algorithms need to be combined with nonscalable global topological data structures.

We are building a new set of scalable algorithms and data structures [12, 13] based on two concepts: i) formal topological queries, and ii) localized topological data structures. In contrast to global topological data structures, the localized merge forest is a collection of local structures that are never assembled into a global data structure, i.e., a merge tree. When necessary, the global information is resolved during a query. It is unclear if persistence-based queries exhibit good locality and thus can be efficiently answered by localized structures. We present persistence-based queries, and generic algorithms for answering them. These algorithms rely only on a modified component maximum query, and thus can be accelerated by different data structures, such as a triplet merge tree, a merge tree, or a merge forest. We compare the performance of two implementations of these queries, on a merge forest and on a triplet merge tree, in a shared-memory setting. A triplet merge tree can answer all persistence-based queries in constant time, at the cost of nonscalable data structure construction [21].

If persistence-based queries can be resolved locally for common analysis use cases, then the applicability of the localized approach is broadened to more scientific use cases, which can benefit from the parallel scalability and low memory overhead of the merge forest, thus enabling more scientific applications of topological methods. Furthermore, the locality study is important to inform design of out-of-core and distributed-memory algorithms.

The contributions are:

- Three generic query algorithms for persistence-based analysis and optimizations specific to the merge forest data structure.
- An empirical study of the locality of the proposed query algorithms on a variety of datasets. The study explores the implications of varying region size and query specializations on locality of query traversal.
- A performance comparison of queries accelerated by a merge forest and those on a triplet merge tree, which optimally returns answers in constant time.

2 Background

Let K be a simplicial complex and g a function defined by a piecewise-linear extension of values at vertices. The function is generic if no two vertices have the same value (enforced by symbolic perturbation [8], e.g., by comparing memory addresses).

The upper star of a vertex v, $St^+(v)$, is the set of simplices that contain v as the vertex with the lowest function value. An upper link, $Lk^+(v)$, is the boundary of a closure of an upper star disjoint from v. We can build the complex by adding vertices and their upper stars in an order defined by the values on the vertices $v_1, \cdots, v_{i-1}, v_i, \cdots, v_n$ with $g(v_{i-1}) > g(v_i)$, creating an upper star filtration $K_i = K_{i-1} \cup St^+(v_i)$, $K_0 = \emptyset$. Connected component C is a subset of complex K_i, where there is a path between all vertices of the component.

*e-mail: klacansky@sci.utah.edu
†e-mail: jediati@sci.utah.edu
‡e-mail: bremer5@llnl.gov
§e-mail: pascucci@sci.utah.edu

Authorized licensed use limited to: The University of Utah. Downloaded on July 28,2022 at 00:42:23 UTC from IEEE Xplore. Restrictions apply.

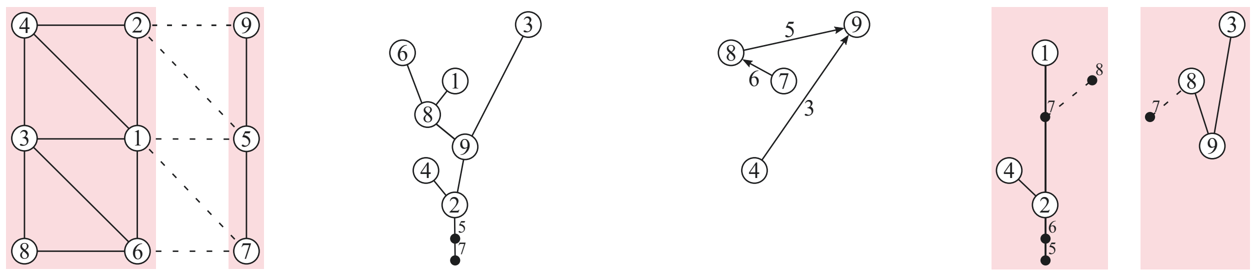

(a) Two regions and bridge set edges. (b) Merge tree. (c) Triplet merge tree. (d) Merge forest with two local trees.

Figure 1: An example dataset consisting of two regions (red) connected by bridge set edges (dashed lines) that has four maxima 4, 7, 8, 9 and three merge saddles 3, 5, 6. The vertex numbers denote function values. The merge tree has the maxima as its leaves, merge saddles as internal nodes, and regular vertices as solid circles. For example, the component defined by maximum 7 merges with the component created at maximum 8 when they reach the merge saddle 6, and this behavior is captured by two arcs (8,6) and (7,6). The triplet merge tree is a directed graph of maxima connected by edges recording the component merging with merge saddles as edge labels. In the previous example, this graph recorded that the maximum 7 merges into the maximum 8 via the merge saddle 6. Obtaining this information from the merge tree requires additional processing. For these global data structures, the bridge set edges are treated the same as the internal edges (solid lines). However, a localized merge forest constructs a local merge tree for each region and uses a subset of bridge set edges to resolve connectivity across regions. In this case, only the edge (6,7) is necessary to form a reduced bridge set. The merge saddle 6 is not represented as a node in the forest because locally it is a regular vertex. Only during a query is it resolved to a merge saddle.

Components partition the complex K_i, and we denote the partition size by $\#CC$. If the upper link is empty, $Lk^+(v_i) = \emptyset$, then the vertex v_i is a *maximum*. Otherwise, if the number of components in the complex $\#CC(K_{i-1})$ differs from $\#CC(K_{i-1} \cup St^+(v_i))$, the vertex v_i is a *merge saddle*. All other vertices are regular vertices because we ignore critical vertices where connectivity does not change during upper star filtration, i.e., minima and other saddles.

The *persistence pair* (u,s) consists of a maximum u and a merge saddle s, such that u is the highest vertex in the component C_{i-1} but not in the component $C_{i-1} \cup St^+(v_i)$, $v_i = s$. The global maximum, vertex v_1, has no persistence pair. The *persistence* of a maximum u is the function value difference of its persistence pair elements, $g(u) - g(s)$. The persistence of the global maximum is infinite.

The *triplet* (u,s,v) represents the nesting relationship of two connected components that merge at the saddle s. The first component has u as its highest valued maximum, and the maximum v is the highest in the second component. The component containing v subsumes the other component at the saddle s because the value at v is greater than at u. The persistence pair (u,s) captures that the component containing the maximum u ceases to exist at the merge saddle s.

The $ComponentMax(K,g,v,h)$ function returns a maximum with the highest function value in a connected component C that contains v and $C \subseteq K_i$, where $g(v_i) \geq h$ and $g(v_{i+1}) < h$. If no such component exists, the function returns the bottom.

We review data structures for the acceleration of superlevel-set queries: merge tree, triplet merge tree, and merge forest (Fig. 1).

Merge Tree. A tree rooted at the global minimum that captures the birth of components at maxima (leaves), and merging of these components at merge saddles (internal nodes). Arcs represent intervals without connectivity changes. A pair (arc, threshold) uniquely identifies a connected component. This tree is often represented as a set of arcs with pointers to children and a parent. The pointers to the children of each arc are usually stored in an array or as a linked list.

Triplet Merge Tree. A set of triplets in the form of (u,s,v) comprises a triplet merge tree, which is a directed graph with directed edges from maximum u to maximum v with a merge saddle s as an edge label. The component born at maximum u merges into the component born at v. Handling of regular vertices and merge saddles can be done via a triplet (u,u,v), and the global maximum has triplet (u,u,u). In this work, we represent only triplets of local maxima. A triplet merge tree is commonly implemented with a hash table that maps the first triplet element u to the pair (s,v).

Merge Forest. We partition the vertex set of complex K into regions $M_1, \cdots, M_m$, where simplices with all of their vertices inside a region belong to that region and the rest forms the *bridge set*. A subset of this bridge set that preserves component connectivity for all complexes K_i is called a *reduced bridge set*. A merge forest is a set of local merge trees for each region and reduced bridge set edges connecting those trees. The *merge forest graph* is the union of local trees and reduced bridge set edges. A maximum corresponds to a local merge tree leaf without an incident bridge-set edge that has a higher end vertex. The set of local merge saddles forms a superset of merge saddles. In general, the local merge trees do not form a subset of a global merge tree if there are two or more regions.

3 Related Work

We review merge trees, triplet merge trees, and merge forest data structures, and their ties to persistence simplification.

3.1 Data Structures

A *merge tree* [5] captures the changes in superlevel-set connectivity for all thresholds ranging from positive to negative infinity. As the threshold is reduced, components are born at maxima and merge at merge saddles. An augmented merge tree stores all vertices whereas an unaugmented merge tree captures only critical vertices (maxima and merge saddles).

A *triplet merge tree* [21] is a directed graph representing the global merging behavior of components in terms of their maxima. This triplet representation enables efficient serial construction from an unordered stream of edges. The triplet merge tree can answer persistence-based queries in constant time at the cost of limited parallel scalability of the data structure construction, potentially forming a bottleneck in data analysis. A merge tree can be augmented with the triplet information in linear time by a post-order traversal.

A *hyperstructure* [6] representation of superlevel-set topology is amenable to parallel processing compared to a standard branch decomposition [20]. This branch decomposition pairs maxima to corresponding merge saddles, and for analysis a branch is processed before its parent, potentially leading to sequential processing. Parallel tree contraction operations on a merge tree produce its hyperstructure and allow parallel computation of per-branch statistics. Moreover, this structure allows for logarithmic time vertex-to-arc lookups, important for data segmentation.

A *merge forest* [13] is a localized data structure that consists of a set of local merge trees and a reduced bridge set edges connecting

Authorized licensed use limited to: The University of Utah. Downloaded on July 28,2022 at 00:42:23 UTC from IEEE Xplore. Restrictions apply.

these trees. Each local tree corresponds to a region in a domain decomposition. The global information is resolved on demand during a query by traversing the local trees and moving between regions via reduced bridge set edges. The locality of the approach results in a linear scaling of the data structure construction and reduced memory overhead, and the queries are answered in comparable time to those of a merge tree.

3.2 Simplification

A key property of these data structures is they enable simplifying a function until noise is removed by capturing the lifetime of components, called a persistence [7]. The persistence of a maximum is the difference between the function value when its component appears (birth), and the value when it merges into a component created at a higher value (death). An appropriate simplification threshold can be discovered by visualizing the birth-death pairs in a scatter plot, called a persistence diagram. Alternatively, a persistence curve is formed by counting persistence pairs for every simplification threshold.

The idea of performing local simplification has its roots in distributed merge tree and Morse-Smale complex algorithms, where minimizing the size of communicated structures during the global reduction phase by sending only sparsified counterparts is paramount because this global phase limits scalability of these algorithms. Initial approaches require a user-specified simplification threshold while constructing the data structure [11], where components internal to a region are simplified if their persistence is below the specified persistence threshold. Unfortunately, the choice of the a priori threshold needs to be conservative to avoid oversimplifying the data and losing important features. This issue is mitigated by simplifying only components fully contained within a region of domain decomposition [14, 18, 19] and communicating only the parts of the structure that interact with the region's boundary. The small size of these communicated sparsified trees suggests that a large portion of maxima can be simplified locally.

A localized simplification algorithm [15] modifies the input data by removing features below the specified persistence. The algorithm exploits the locality of noise by terminating region growths started at extrema (maxima or minima) when a user-specified simplification threshold is reached. For a 1% simplification level, the algorithm traverses only 2% of a dataset. The result of partial region growths can then be used to build a persistence diagram or a persistence curve up to the specified simplification threshold. We use this idea of the partial growth to avoid excessive merge forest traversal during one of the persistence queries.

4 Persistence-based Queries

We review and define queries that consider the lifespan of components, and enable noise removal, exploration of functions at multiple simplification scales, or studying hierarchical nesting of components based on their importance. For example, we can filter all connected components with low persistence out of a visualization to eliminate noise or extract vortices of different intensities by computing their relative importance to the other components [3].

4.1 Triplet Query

A triplet represents the component, born at a maximum and dead at a merge saddle, and the component into which it merges. Moreover, triplets enable the computation of persistence and relevance [17] in constant time, and they are a representation of a branch decomposition [20]. This representation has been introduced as an alternative data structure to a merge tree [21], and we follow the same definition with a minor change of defining triplets neither for regular vertices nor a global maximum.

Definition 1. *We name* $\underline{Triplet(K,g,u)}$ *the function that outputs the triplet* (u,s,v) *for the maximum u. If such a triplet does not exist, the output is bottom* $\perp$ *(the case of the maximum u being the global maximum).*

The last element v in the triplet (u,s,v), the representative, is the highest vertex in a component defined by maximum u and threshold $g(s)$, i.e., $v = ComponentMax(K,g,u,g(s))$. Moreover, the persistence of a maximum u with a triplet (u,s,v) is the value $g(u)-g(s)$.

The complexity of answering the $Triplet$ query depends on the underlying data structure. For example, the worst case time complexity for a merge tree T is $O(|T|)$, whereas for a triplet merge tree it is $O(1)$.

4.2 Persistence Query

Persistence captures the lifetime of a component, and therefore it is useful to filter components with short lifetimes or to study the stability of the input function by simplifying components at multiple persistence thresholds. Furthermore, this query can be combined with filtering the input maxima by isovalue. This query finds the merge saddle for a given maximum, forming a persistence pair, which is used to compute its persistence. The difference between this query and the triplet query is that the last element of the triplet (representative) is not needed.

Definition 2. *We call* $\underline{PersistencePair(K,g,u)}$ *the function that returns the pair* (u,s) *where u is the queried maximum and s is a merge saddle. If the vertex u is a global maximum, it returns bottom* $\perp$*.*

The *Persistence* query can be implemented using the *PersistencePair* query's output (u,s) to compute the difference of function values, $g(u)-g(s)$.

Definition 3. *We call* $\underline{Persistence(K,g,u)}$ *the function that returns the persistence of the maximum u. If the vertex u is a global maximum, it returns* ∞*.*

Unfortunately, querying persistence pairs for all maxima amounts to global analysis. Often, only the maxima of persistence greater than or equal to a simplification threshold are of interest. Therefore, we define an additional query that allows us to test if a maximum has at least some persistence, and if not it returns the persistence pair of that maximum. This latter property is useful for constructing a persistence diagram or curve up to a certain simplification threshold [15]. More importantly, the simplification threshold allows the query to avoid unnecessary traversal, such as the global traversal of *Persistence* query when a global maximum is queried.

Definition 4. *We name* $\underline{PersistencePairBelow(K,g,u,p)}$ *the function that returns the persistence pair of the maximum u, if its persistence is below the simplification threshold p. Otherwise, it returns bottom* $\perp$*.*

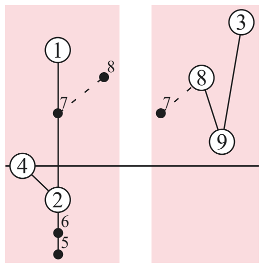
The $PersistencePairBelow(K,g,u,\infty)$ query is equivalent to the $PersistencePair(K,g,u)$ query.

5 Triplet Query Algorithm

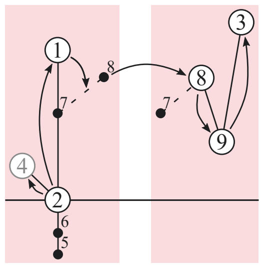
The *Triplet* query of a given maximum searches a topological data structure to obtain the maximum's merge saddle and representative to form the output triplet.

The algorithm (Alg. 1) starts traversal at the queried maximum u, and grows a connected component until u is no longer the highest vertex in the component (Fig. 2). Since there could be multiple ways to grow the component in a forest (a component can span multiple regions), we use a priority queue to process the vertices in order. This traversal is similar to that of the task-based merge tree algorithm [10]; however, we need to recover the representative while the task-based growth terminates at a merge saddle. In contrast to a merge forest, a merge tree has only one path toward the root, and thus the priority queue contains at most one element.

Authorized licensed use limited to: The University of Utah. Downloaded on July 28,2022 at 00:42:23 UTC from IEEE Xplore. Restrictions apply.



(a) First loop iteration.



(b) Second loop iteration.

Figure 2: An example of TRIPLET(u = 4) call that computes the triplet (4, 3, 9) of a maximum 4, highlighting the expensive query in the lower portion of the data values that traverses both local merge trees (red). The query starts with the priority queue containing the queried maximum 4 and an empty visited set. Then, it executes in two loop iterations. In the first iteration, vertex 4 is dequeued and COMPONENTMAX(u = 4, h = 4) returns 4 plus enqueues 3; the visited set now contains 4. Since 4 is not above 4, the loop reaches the second iteration. In the second iteration, vertex 3 is dequeued, and COMPONENTMAX(u = 3, h = 3) traverses both regions and returns 9 (it skips already visited 4, colored gray) and enqueues no vertex. Since 9 is above 4, the triplet (4, 3, 9) is returned. At the end of the query, the visited set contains all critical vertices, {3,4,5,6,7,8,9}.

Algorithm 1 The merge forest-accelerated algorithm for computing the *Triplet* query that returns the (maximum, merge saddle, representative) triplet, or bottom for a global maximum.

```
1:  function TRIPLET(function $g$, forest $F$, maximum $u$)
2:      priority queue $PQ \leftarrow \emptyset$
3:      visited set $VS \leftarrow \emptyset$
4:      ENQUEUE$(PQ, g(u), u)$
5:      while $PQ \neq \emptyset$ do
6:          $v \leftarrow$ DEQUEUE$(PQ)$
7:          $v^*, VS, PQ \leftarrow$ COMPONENTMAX$(g, F, v, g(v), VS, PQ)$
8:          if $v^* \neq \perp$ and $g(v^*) > g(u)$ then
9:              return $(u, v, v^*)$
10:     return $\perp$                                    ▷ Global maximum
```

The subroutine COMPONENTMAX is a modified *ComponentMax* query that enqueues all crossed arcs and reduced bridge set edges into the priority queue for further traversal. A crossed arc has its highest value above the queried threshold and its parent is below the threshold. A crossed reduced bridge set edge has one or both end vertices below the threshold passed into COMPONENTMAX. Recall that the COMPONENTMAX subroutine returns the bottom value if the queried vertex is below the threshold (never the case here) or the vertex was previously visited. The visited set contains all visited vertices to avoid repeated traversal.

Correctness. A merge tree is the same as a merge forest when the region size is equal to the domain size, and thus we first show that the triplet query algorithm (Alg. 1) returns the correct triplet when used with a merge tree. Then, we show the algorithm still works for different region sizes, and thus the merge forest.

Lemma 1 (triplet query on a merge tree)**.** *The triplet query algorithm computes a triplet of a given maximum u.*

Proof. The algorithm processes an ordered sequence of vertices $v_1 = u, v_2, \cdots, v_n$. We maintain the invariant that at the beginning of each loop iteration (line 5), the maximum u is the highest vertex in the connected component C_{i-1} in a complex K_{i-1}. The other invariant is that all maxima in the connected component C_{i-1} were visited.

Base case. $v_1 = u$, $C_1 = \{u\}$, and thus u is the highest vertex (pushed onto the priority queue on line 4).

Induction step. The component C_{i-1} has the highest vertex of the maximum u (invariant). The dequeued vertex v_i is not in C_{i-1}, because $g(v_{i-1}) > g(v_i)$ (line 6). Line 7 returns the highest vertex in C_i, where C_{i-1} is a subset of C_i. If the result of the COMPONENTMAX call on C_i, the vertex v^*, differs from the input maximum u, we return the triplet of the maximum, (u, v, v^*). Otherwise, the invariant holds because u is the highest vertex in C_i. If the whole sequence is processed and no higher maximum is found, the maximum u is the global maximum.

□

Lemma 2 (triplet query on a merge forest)**.** *The triplet query algorithm computes the triplet of a given maximum u from a merge forest.*

Proof. The nodes in merge forest form a superset of nodes in a merge tree. Since each node is tested in the inner loop, the nodes form a superset of the sequence of vertices processed in the merge tree, and they are processed in decreasing order (because of the priority queue). We conclude the query returns the same triplet. We rely on the property of reduced bridge set that guarantees unchanged connected components compared to the input domain, and thus the COMPONENTMAX calls return the same output. □

The triplet algorithm can be applied directly to the input complex K and function g without any acceleration data structure, such as a merge forest or merge tree, since a merge forest with a region size 1^3 is equivalent to the input complex (assuming a reduced bridge set is constructed pairwise between neighboring regions). Moreover, any topological data structure implementing the modified *ComponentMax* query can answer the triplet query. For example, for a triplet merge tree, the crossed arc's lower end vertex would be a merge saddle s of a triplet (u, s, v).

In our implementation, the reduced bridge set end vertices do not split arcs in local trees. Instead, we store an unordered list of these edges per arc. This design allows us to avoid splitting an arc segmentation and update the reduced bridge set without changing the local trees. However, this choice may cause more reduced bridge set edges enqueued onto the priority queue.

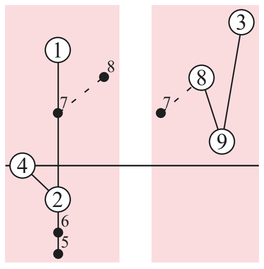
Time Complexity. In the worst case, when global maximum u is queried, the COMPONENTMAX (line 7) returns $v^* = u$ every time, and thus the if statement (line 8) is never true. Therefore, the algorithm traverses the whole merge tree or merge forest. For a merge tree, the priority queue PQ has at most one entry (a parent arc), yielding overall time complexity $O(|T|)$. For a merge forest, the queue may contain $O(|F|)$ entries, resulting in $O(|F| \log |F|)$ time complexity. Similar worst-case analysis can be applied to the *Persistence* and *PersistenceBelow* queries.

5.1 Optimizing Internal Query

In a merge tree, triplets can be precomputed in linear time, constructing a triplet merge tree, and thus rendering the triplet query in constant time. A similar optimization, limited to components contained inside a region, can be applied to a merge forest. Additionally, the portions of noninternal queries may be accelerated, e.g., during the *ComponentMax* query. We leave this form of query acceleration for future work.

Definition 5. *A triplet (u, s, v) is an internal triplet if the component $C \subseteq K_i \cap M_j$, s.t., $s = v_i$ and $u, s, v \in \overline{C}$, contains no reduced bridge set edge (a, b) with its end vertices greater than or equal to the value of the merge saddle s.*

Authorized licensed use limited to: The University of Utah. Downloaded on July 28,2022 at 00:42:23 UTC from IEEE Xplore. Restrictions apply.


(a) First loop iteration.

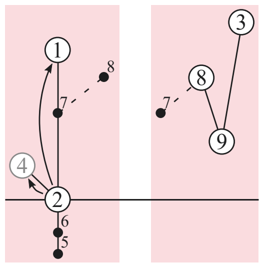
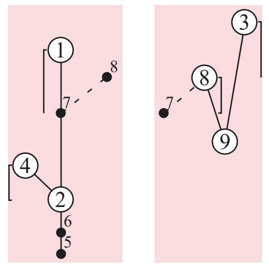

(b) Second loop iteration.

Figure 3: Execution of PERSISTENCEQUERY(u = 4) for maximum 4 to compute its persistence of 1. The query starts with the priority queue containing the queried maximum 4. Then it executes in two loop iterations: 1) the vertex 4 is dequeued and COMPONENTMAXIMUMEARLY(maximum = 4, u = 4, h = 4) returns 4 plus enqueues 3 just like the first iteration of the TRIPLET query algorithm. Since 4 is not above 4, the loop reaches the second iteration. 2) The vertex 3 is dequeued and COMPONENTMAXIMUMEARLY(maximum = 4, u = 3, h = 3) returns 8 and enqueues nothing. Since 8 is above 4, the query terminates. In contrast to the triplet query, the persistence query traverses only one region, and the visited set contains only local vertices {3,4,8}. Moreover, the arc 4 is internal, and thus its query is computable in constant time, thereby avoiding forest traversal.


(a) Simplification threshold 2.

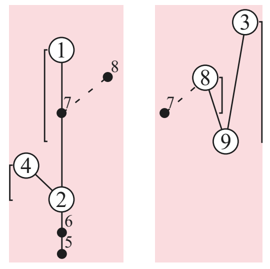

(b) Simplification threshold 4.

Figure 4: The PERSISTENCEBELOW query algorithm runs on all maxima for two simplification thresholds, 2 and 4. The closed intervals highlight maxima with known persistence, and the open intervals denote maxima with persistence at or above the simplification threshold. At threshold 2, this query is local for maxima 4, 8, and 9, and returns persistence values 1, ∞, and ∞. In contrast, the *Persistence* query has only the maximum 4 local to a region, because it is internal as the maximum 8 in the same region has a higher value. Threshold 4 requires traversal of both regions for maxima 7 and 8. Since maximum 9 has the next vertex to process 5, its persistence is at least 4, and the query can terminate early.

Internal triplets can be computed in a post-order traversal of a local merge tree. For each arc, we track the representative (highest vertex of its children) and the next reduced bridge set edge to be processed. An arc, with a different representative than its parent's representative and merge saddle above the next bridge set edge, can set its representative to this arc (merge saddle) and the parent arc's representative (representative).

Definition 6. *A pair (u,s) is an internal persistence pair of the component $C \subseteq K_i \cap M_j$, s.t., $s = v_i$ and $u,s \in C$, if there exists no reduced bridge set edge (a,b) with either end vertex in the component $C \setminus \{s\}$ while having their function values greater than the value of the merge saddle s, and the component C contains vertex v with a higher value than the maximum u, $g(v) > g(u)$.*

Similarly to the internal triplets, the internal persistence pairs are constructed by a post-order traversal of each local merge tree. However, the arc with a higher valued maximum may have reduced bridge set edges to other regions because it is not necessary to recover the actual representative.

The number of queries answered internally in a region depends on the region size. Region size 1^3 has no internal queries, and the size equal to the input dataset size has all queries answered internally. On the one hand, smaller regions result in more parallelism during forest construction [13]. On the other hand, larger regions render more triplet queries internal and thus are constant time.

5.2 Persistence Query Specialization of Triplet Query

The triplet query needs to recover the component maximum into which the queried maximum merges (representative). In contrast, for the persistence query, it is sufficient to find the first two elements of the triplet, the input maximum and its merge saddle, to calculate the persistence of a maximum. To certify a vertex is a merge saddle of the maximum, only a reachable vertex higher than the maximum needs to be found (Fig. 3). The motivation for this query specialization is the assumption that, in many cases, this vertex shares the same region with the maximum, rendering the query internal and thus constant time.

Therefore, the TRIPLET subroutine (Alg. 1) is modified with two changes to create the PERSISTENCE subroutine. First, the COMPONENTMAX call (line 7) is replaced with the COMPMAXEARLY function that terminates when any vertex above the queried maximum u is reached. This procedure follows a greedy depth-first traversal, but other heuristics are possible, such as breadth-first traversal or the use of a priority queue to always try the highest arc. The objective is to find a vertex higher than the queried maximum quickly. The second change is the query constructs only the persistence pair (u,v) and then calculates the persistence using this pair. Moreover, for a global maximum, it returns infinity instead of bottom (line 10).

5.3 Further Optimization to Persistence Below Query

We have seen that restricting the output of a query may reduce the amount of work, such as not needing the last element of a triplet to calculate the persistence using the *Persistence* query. However, some maxima may still require traversal of large portions of a merge forest. Take the example of a global maximum where both the TRIPLET and PERSISTENCE query algorithms traverse all arcs in the forest. Instead of adding a special case for the global maximum to address this issue, we modify the query algorithm to limit the traversal.

The main use of persistence is to simplify noise. Therefore, the query needed for such simplification must answer the question: Does a given maximum exist at a specific persistence threshold? An example would be to remove all maxima with a persistence value less than 1% of a function range before using superlevel-set components in visualization or analysis. Moreover, the query could be progressively executed by lowering the simplification threshold and keeping its internal state in memory (query inputs, priority queue, and visited set). This iterative query execution could be used to progressively build a persistence curve.

The *PersistenceBelow* query has the advantage of potentially terminating the traversal after the function value difference between the maximum and currently dequeued vertex from the priority queue is above or equal to the simplification threshold (Fig. 4). This early termination prevents traversal of many regions for long-lived components, such as the component defined by a global maximum.

We find several changes compared to the triplet query algorithm (Alg. 1). The algorithm (Alg. 2) traverses a sequence of vertices of decreasing function values, but it can terminate if the persistence of the component is known to be greater than or equal to the simplification threshold (lines 7-8). The algorithm uses the same helper function to early exit the *ComponentMax* query, COMPMAXEARLY, as does the PERSISTENCE subroutine. The input maximum u is passed to this helper call to enable the early exit when the func-

Authorized licensed use limited to: The University of Utah. Downloaded on July 28,2022 at 00:42:23 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2 The algorithm for answering the *PersistenceBelow* query that computes the persistence of a maximum if its persistence is below a simplification threshold. Otherwise, it returns infinity. The queried maximum is passed into the subroutine COMPMAXEARLY to allow for an early termination during a query's traversal.

```
1:  function PERSISTENCEBELOW(function $g$, forest $F$, maximum $u$, persistence threshold $p$)
2:      priority queue $PQ \leftarrow \emptyset$
3:      visited set $VS \leftarrow \emptyset$
4:      ENQUEUE$(PQ, g(u), u)$
5:      while $PQ \neq \emptyset$ do
6:          $v \leftarrow$ DEQUEUE$(PQ)$
7:          if $g(u) - g(v) \geq p$ then
8:              return $\infty$            ▷ Early exit for persistent maximum
9:          $v' \leftarrow$ COMPMAXEARLY$(g, F, u, v, g(v), VS, PQ)$
10:         if $v' \neq \perp$ then
11:             return $g(u) - g(v)$
12:     return $\infty$                    ▷ Global maximum
```

tion encounters higher vertex, returned as v' (which may be different from v^* returned by the COMPONENTMAX subroutine).

Correctness. The two changes compared to the triplet query algorithm are an early exit if the persistence of the maximum is greater than or equal to the simplification threshold (lines 7-8) and early termination of COMPMAXEARLY (line 9). The former can happen when dequeued vertex v is the merge saddle of the maximum u, or v is some other merge saddle or a regular vertex. If v is the merge saddle, then the persistence of u is $g(u) - g(v) = p$; otherwise, it is greater than the persistence threshold. The latter always returns a bottom or vertex v' where $g(v') > g(u)$. Since $g(v') > g(u)$ and u is a maximum, then v' must belong to a component with a higher maximum, and by Elder's rule, the component defined by a lower maximum merges into the component with a higher maximum.

6 Results

The evaluation has two primary goals. The first is to empirically measure the locality and running time of the triplet query algorithm (Alg. 1), to test if internal query optimization is beneficial, and to determine if specialization into the persistence and persistence below algorithms (Alg. 2) reduces the number of visited regions and running time. The second is to compare query execution times with a triplet merge tree, which answers each query in constant time, and thus provides an optimal baseline.

The datasets used for the tests (Table 1) consist of both simulation data (Fuel, Cosmology subset [16], Duct [1], HCCI [2], JICF Q [9]) and acquired data (Foot). For all the datasets, we use superlevel-set analysis that captures components useful for further processing. The perturbation scheme is based on a local index and then a region index, and it is the same for a merge forest and triplet merge tree. Both data structures are constructed using the same six-subdivision neighborhood. This subdivision may introduce artifacts [4].

The machine used to obtain the timings has AMD Threadripper 1950X (16 cores at 3.4 GHz) and 64 GB RAM. The tests were compiled with MSVC 19.28.29912 (/O2). All tests are run once.

6.1 Locality Study of Triplet Queries

We start the evaluation with the most general query, the *Triplet* query, which computes both the merge saddle and representative for a given maximum. We run the query for all maxima in each dataset and measure its running time and the number of visited regions, our choice of locality unit. Since the locality depends on the region size, we test three region sizes: 32^3, 64^3, and 128^3.

We inspect the locality of these queries by computing a histogram of the number of visited regions (Fig. 5). All histograms have 20 bins. The datasets exhibit bimodal distributions, with the majority of triplet queries visiting only a few regions. The exception is the noisy Foot and Cosmology datasets where a large portion of queries traverses a large number of regions. Only the histograms of the Foot dataset change significantly with the increasing region size (Fig. 6).

We shift our focus on the wall clock time of the queries (Fig. 7). The prevalent time dependence is on the topological complexity rather than the number of visited regions. For example, at 64^3 region

Table 1: A list of acquisition and simulation datasets used for the evaluation. In all datasets, the components of superlevel sets are of interest. The datasets have different topological complexity. Datasets with an integer data type are converted to the float32 type.

Dataset	Resolution	Data type	# maxima
Fuel	64x64x64	uint8	65
Foot	256x256x256	uint8	192,375
Cosmology	256x256x256	float32	726,942
Duct	193x194x1000	float32	35,381
HCCI	560x560x560	float32	11,274
JICF Q	1408×1080×1100	float32	228,459

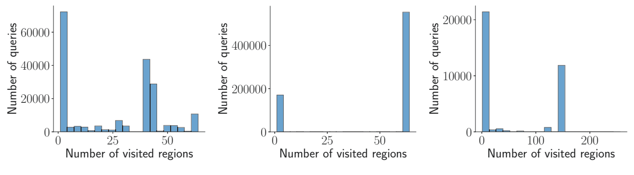

(a) Foot dataset. (b) Cosmology dataset. (c) Duct dataset.

Figure 5: Histograms at region size 64^3 of triplet queries based on the number of regions visited. Even though most datasets exhibit a bimodal distribution, the shape is largely unpredictable and data dependent. The Cosmology dataset has a majority of the queries traversing the data background, and filtering maxima below threshold 5 reduces the number of queries from 726,942 to 38,761, bringing the run-time from 66,172 to 121 seconds.

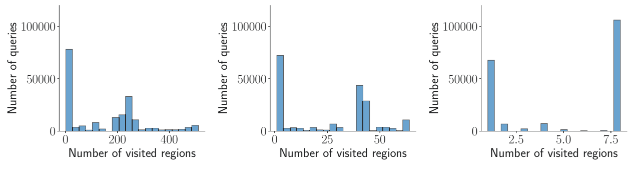

(a) Region 32^3. (b) Region 64^3. (c) Region 128^3.

Figure 6: Histograms at region sizes 32^3, 64^3, and 128^3 of triplet queries based on the number of regions visited for the Foot dataset. We observe a shift of the middle of the distribution into right. This shift can happen, for example, for a feature that lies in the center of the dataset, intersecting 8 regions regardless of the region size.

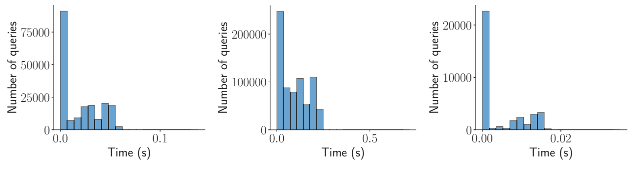
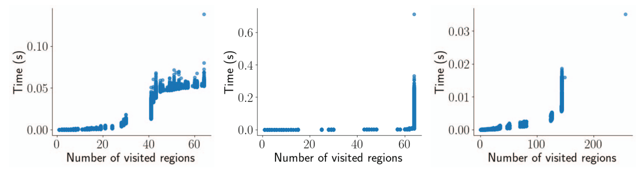

(a) Foot dataset. (b) Cosmology dataset. (c) Duct dataset.

Figure 7: Triplet query running times histograms where a large portion of the queries is near instant (region size 64^3). However, for some datasets, such as Cosmology, the majority of queries take at least 10 milliseconds, rendering aggregate analysis slow.

Authorized licensed use limited to: The University of Utah. Downloaded on July 28,2022 at 00:42:23 UTC from IEEE Xplore. Restrictions apply.



(a) Foot dataset. (b) Cosmology dataset. (c) Duct dataset.

Figure 8: Correlations between visited region count and the running time of the triplet query (region size 64^3). We observe a sharp increase in the running time when resolving queries that have the global maximum as their representative or lie in the data background. The dot in the top right corner of each plot is the global maximum, which requires traversal of the whole merge forest. The large gap in the number of visited regions between the global maximum and other maxima for the Duct dataset is caused by many components suddenly merging into the global component. Other metrics such as number of traversed arcs may be more suitable; however, we are primarily interested in locality, and thus using regions is a reasonable compromise.

size, the slowest query for the Cosmology dataset takes 0.7 s and visits all 64 regions, and the slowest query on JICF Q executes in 0.3 s and visits all 6,732 regions. A change in the region size yields a small query time reduction.

The scatter plot of number of visited regions and the query time (Fig. 8) shows that the unit of locality, visited regions, correlates well only for some datasets. The Cosmology, Duct, and JICF Q display a spike where the query time changes by an order of magnitude suddenly, when a query visits a majority or all regions. This result suggests that additional metrics such as the number of traversed arcs or executed loop iterations may better correlate to a query's running time because even for the smallest region 32^3, the visited count does not capture the running time spike for the global queries (Cosmology).

Overall, the $O(|F|\log|F|)$ time complexity of the triplet query algorithm (Alg. 1) makes these queries expensive, and we recommend this algorithm only when few maxima need to be queried.

6.2 Evaluation of Internal Query Optimization

From the histograms in the previous section, we observe that many queries traverse a single region and thus are internal to that region. In general, around 20% of triplet queries are internal, and with the increasing region size, a larger proportion becomes internal (Table 2). Unfortunately, this optimization provides speed-up in a fraction of a percent for the triplet queries because we are optimizing already fast queries. After all, the expensive queries are ones that traverse many regions and do not benefit from this optimization.

The memory overhead of this optimization is an extra index per arc, each maximum stores an index of its merge saddle, and merge saddles point to their representatives. The impact on the forest construction time in the worst case is 1% (the Cosmology dataset at region size 64^3).

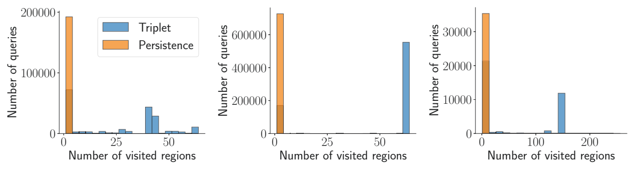
We conclude that this optimization has a negligible impact on the overall running time of triplet query. However, we revisit this optimization for the persistence query algorithm in the next section.

6.3 Locality Study of Persistence Queries

The persistence query may traverse a smaller fraction of the dataset compared to the triplet query because it needs to recover only the merge saddle and not its representative. Finding the representative is especially expensive for maxima appearing in the dark background portions of the data (Fig. 8). Fortunately, the persistence query is sufficient for many analysis tasks, such as noise removal by simplifying components with low persistence or the construction of a persistence diagram.

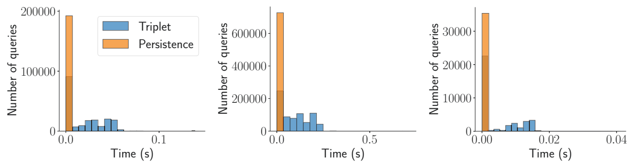
Table 2: Proportions of triplet queries performed internally inside a region out of all queries. These queries can be answered in constant time. The region size 64^3 is usually preferable for the forest construction, and only about 20% of triplet queries are internal. However, this optimization does not translate into a noticeable speed-up (numbers in parentheses) because the internal queries were fast in the first place, and the overall time is dominated by the expensive queries.

Dataset	Region 32^3	Region 64^3	Region 128^3
Fuel	0.18 (1.03)	-	-
Foot	0.26 (1.00)	0.31 (1.00)	0.35 (1.00)
Cosmology	0.18 (1.00)	0.21 (1.00)	0.22 (1.00)
Duct	0.29 (1.00)	0.40 (1.00)	0.47 (1.00)
HCCI	0.12 (1.00)	0.18 (1.00)	0.22 (1.00)
JICF Q	0.38 (1.00)	0.47 (1.00)	0.53 (1.00)



(a) Foot dataset. (b) Cosmology dataset. (c) Duct dataset.

Figure 9: Histograms of visited regions for persistence queries at region size 64^3 overlaid on the triplet query histograms. The results confirm that persistence query specialization significantly reduces the number of regions traversed and renders many queries internal to a region.



(a) Foot dataset. (b) Cosmology dataset. (c) Duct dataset.

Figure 10: Histograms of query time for persistence queries at region size 64^3 overlaid on the triplet query histograms. The reduced number of visited regions of the persistence query directly translates to a run-time improvement.

For each dataset, we run the persistence query for all maxima. This scenario is common to determine what maxima belong to noise, to rank maxima and extract components of the most stable ones, or to study the function stability as the persistence changes in the form of a persistence diagram or curve that is used to determine the simplification threshold.

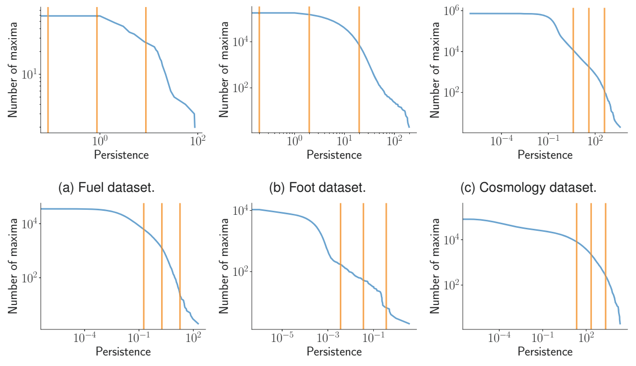
The majority of persistence queries traverse few regions (Fig. 9). The difference between persistence and triplet query is striking; by not recovering the representative, we gain locality. Recall that the persistence query can terminate as early as it discovers a vertex higher than the queried maximum. Moreover, this increase in locality directly translates to running time improvements (Fig. 10), which makes this query useful for more use cases.

We further investigate these results by computing the proportion of internal persistence queries, which can be answered in constant time. As expected, the proportion is greater than or equal to that of triplet queries (Table 3). Unexpectedly, around 90% of queries are internal compared to only 20% of triplets, resulting in 1.2 to 3x speed-up (region size 64^3) compared to queries without the internal region optimization.

Authorized licensed use limited to: The University of Utah. Downloaded on July 28,2022 at 00:42:23 UTC from IEEE Xplore. Restrictions apply.

Table 3: The running times of persistence queries with and without internal query optimization and proportions of internal queries (answered in constant time). Queries are run for all maxima in a dataset. The proportion is always greater than or equal to the proportion for triplet queries. The run-time speed-ups compared to the query without optimization are in parentheses. Only the Duct dataset time exhibits smaller speed-up at region size 128^3 than 64^3, where the larger proportion of the running time is dominated by the global queries. Furthermore, larger region size does not imply lower aggregate running time because, as the region size increases, the compMaxEarly procedure may visit more arcs (it performs depth-first traversal of local trees), depending on the dataset. For example, for the Foot dataset, the persistence query without the internal optimization takes longer for region size 128^3 than for 64^3 whereas the optimized query takes the same time, resulting in seemingly greater speed-up compared to other datasets. The locality results support the idea of sparsification during distributed merge tree construction [14, 18, 19] and the reduced strong scaling with increasing core counts because the regions become smaller and fewer maxima can be internally sparsified, increasing the communication cost during the reduction phase.

Dataset	Region 32^3			Region 64^3			Region 128^3		
	no opt (s)	opt (s)	internal	no opt (s)	opt (s)	internal	no opt (s)	opt (s)	internal
Fuel	0.00	0.00	0.55 (1.44)	0.00	0.00	-	-	-	-
Foot	0.58	0.35	0.86 (1.68)	0.98	0.31	0.94 (3.17)	3.43	0.29	0.98 (12.00)
Cosmology	2.09	1.37	0.89 (1.53)	2.54	1.42	0.95 (1.79)	5.19	2.42	0.98 (2.14)
Duct	0.18	0.14	0.77 (1.26)	0.15	0.08	0.87 (1.93)	0.10	0.05	0.93 (1.84)
HCCI	0.10	0.09	0.88 (1.13)	0.04	0.03	0.93 (1.42)	0.03	0.02	0.96 (1.50)
JICF Q	1.72	1.54	0.81 (1.11)	0.76	0.61	0.89 (1.23)	1.32	0.64	0.94 (2.08)



(a) Fuel dataset. (b) Foot dataset. (c) Cosmology dataset.

(d) Duct dataset. (e) HCCI dataset. (f) JICF Q dataset.

Figure 11: Log-log persistence curve plots using region size 64^3, but the persistence curve is virtually unchanged for other region sizes. The orange lines indicate 0.1%, 1%, and 10% simplification thresholds. The choice of fixed simplification percentages is due to the inability to know the simplification threshold a priori without computing the curve. Notice this issue with fixed thresholds on the Fuel and Foot datasets, where a higher simplification threshold may be of interest.

6.4 Locality Study of Persistence Below Queries

The locality of persistence query improves significantly compared to the triplet query. However, some of the queries still traverse the whole forest, e.g., a global maximum. The most controllable query, with potentially the fastest running time, is the persistence below query because it can limit the traversal needed to compute the result. We use three simplification thresholds, 0.1%, 1%, and 10% of the function range, by gleaning the persistence curve of each dataset (Fig. 11).

We explore the effects of the simplification threshold on the query locality (Table 4). At the threshold 0.1%, the queries visit between 1 and 910 regions. As the threshold increases to 1% and 10%, more regions get traversed in most datasets because the algorithm needs to visit larger portions of the components to determine if their persistence is below the threshold. This reduction in traversal directly translates into run-time improvements, and even at a large 10% simplification threshold, the queries are two to three times faster than persistence queries.

Table 4: The proportion of queries visiting single region as the simplification threshold increases. Ranges denote the minimum and maximum number of visited regions at the given threshold. Region size 64^3. The JICF Q dataset contains a low persistence component spanning a large portion of the domain, and its query visits 910 regions out of 6,732.

Dataset	0.1%	1%	10%
Fuel	1.000 [1,1]	1.000 [1,1]	1.000 [1,1]
Foot	0.997 [1,6]	0.984 [1,6]	0.938 [1,6]
Cosmology	0.950 [1,5]	0.949 [1,7]	0.948 [1,7]
Duct	0.914 [1,8]	0.878 [1,9]	0.874 [1,67]
HCCI	0.935 [1,18]	0.932 [1,37]	0.932 [1,111]
JICF Q	0.897 [1,910]	0.894 [1,910]	0.893 [1,910]

6.5 Comparison of Queries Accelerated by Triplet Merge Tree and Merge Forest

How do these queries compare to those of a triplet merge tree? A triplet merge tree can answer all presented queries in constant time, and in our implementation that amounts to two memory accesses for the triplet query (one for the maximum to get the merge saddle index, and one for the merge saddle to obtain the representative index).

We compute the aggregate times to answer the *Triplet*, *Persistence*, and *PersistenceBelow* queries for all maxima in each dataset. Even though the *Persistence* and *PersistenceBelow* queries take the same time for triplet merge trees, they differ for the merge forest due to early traversal termination. All queries use the internal query optimization.

In contrast to previous measurements of individual query times, all comparison results with triplet merge tree are produced by timing all queries at once because timing individual query skews the results by penalizing the fast queries; the faster the query, the larger the proportion of its time taken by computing the two timestamps to calculate its running time. This penalty can change the time by an order of magnitude for the triplet merge tree queries.

6.5.1 Triplet Query

We look at the aggregate of all query running times for different region sizes. The tests report significant slowdown compared to the triplet merge tree (Table 5), even with the internal query optimization. The need to recover the representative of a triplet often results in a traversal of almost the whole forest, e.g., 76% of the maxima in the

Authorized licensed use limited to: The University of Utah. Downloaded on July 28,2022 at 00:42:23 UTC from IEEE Xplore. Restrictions apply.

Table 5: Serial execution times for running the *Triplet* query on each maximum in the dataset, region size 64^3, with internal query optimization. Clearly, using the triplet query on the merge forest to build a triplet merge tree is inefficient, and we observe quadratic scaling with respect to the topological complexity.

Dataset	Triplet merge tree (s)	Merge forest (s)
Fuel	0.0000	0.0000
Foot	0.0074	3,613.8402
Cosmology	0.0293	66,171.5902
Duct	0.0018	145.7138
HCCI	0.0006	12.7924
JICF Q	0.0107	6,677.2230

Cosmology dataset has the global maximum as their representative and merge saddle in the data background.

In the comparison with the global triplet merge tree, we observe that if all maxima need to be queried, building the triplets directly from the merge forest graph may be beneficial. However, this performance is expected when a global analysis is needed. Querying the triplet of each maximum is an inefficient approach to building the triplet merge tree. After all, a triplet merge tree can be constructed in $O(n \log n)$ (n is number of edges in the domain) time [21] while the forest is performing $O(|F|^2 \log |F|)$ operations. When the number of queries is much smaller than the input size, the scalable construction of the forest may absorb the less efficient triplet queries. We would construct triplet merge tree from the merge forest graph directly if a large number of global queries is going to be executed.

6.5.2 Persistence and Persistence Below Queries

As we did for the triplet query, we look at the aggregate running times. The early termination and high proportion of internal queries result in a much better performance compared to the triplet query (Table 6). We observe a slowdown ranging from 1.4x (Fuel) to 39x (Cosmology) in serial. The query algorithms share no mutable state, and thus we can execute queries simultaneously in parallel. Unfortunately, the scalability is limited by the slowest query, such as the global maximum query taking 0.71 s, whereas running all queries takes 0.79 s (Cosmology dataset). Moreover, the running time of the global maximum and the next slowest query differs by one to two orders of magnitude, and introducing a special handling of this case may be worthwhile after all. Further improvement would require parallel query algorithm, increasing the complexity of the implementation.

Even though it may seem that the persistence query builds a branch decomposition behind the scenes by finding persistence pairs for all maxima, thinking so is misleading because additional work would be needed to construct a map from a merge saddle to a branch containing it. One approach could be to collect all vertices dequeued from the priority queue during the query in a hash table.

Finally, we show aggregate results at various simplification thresholds, because noise is mostly local (Table 4). The performance gap narrowed, making forest a suitable data structure for noise elimination. At the simplification level 0.1% of the function range, the queries accelerated by the triplet merge tree are about ten times faster than those of the merge forest (Table 7). Increasing the simplification to 1% and 10% widens the gap due to more global queries. Moreover, compared to presimplification [15], which takes 1.39 s to perform 10% simplification of the Foot dataset, a merge forest has a construction time of 0.29 s and a simplified components query time of 0.03 s, for a total of 0.32 s analysis time.

Table 6: Serial and parallel execution times for running the *Persistence* query for all maxima, region size 64^3. The overall scalability is limited by the slowest query for global maximum that traverses the whole forest (the numbers in parentheses). The serial query times of Foot and HCCI datasets are comparable with those of serial merge tree construction from the merge forest graph, but without building a global structure [13].

Dataset	Triplet merge tree (s)		Merge forest (s)	
	1 core	16 cores	1 core	16 cores
Fuel	0.0000	0.0000	0.0000	0.0003 (0.00)
Foot	0.0087	0.0026	0.3079	0.1485 (0.14)
Cosmology	0.0368	0.0081	1.4201	0.7925 (0.71)
Duct	0.0033	0.0007	0.0792	0.0423 (0.04)
HCCI	0.0011	0.0003	0.0313	0.0239 (0.02)
JICF Q	0.0313	0.0066	0.6124	0.3650 (0.32)

Table 7: Serial execution times for running the *PersistenceBelow* query on each maximum in the dataset, region size 64^3. With an increasing simplification threshold, fewer queries are internal, and the gap between a triplet merge tree and a merge forest widens. The Cosmology and HCCI datasets have similar running times regardless of the threshold because the large portion of pairs is simplified already at 0.1% threshold.

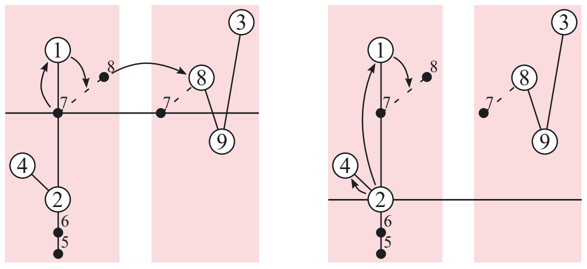
Dataset	Triplet merge tree (s)			Merge forest (s)		
	0.1%	1%	10%	0.1%	1%	10%
Fuel	0.00	0.00	0.00	0.00	0.00	0.00
Foot	0.01	0.01	0.01	0.01	0.03	0.12
Cosmology	0.04	0.04	0.04	0.77	0.67	0.72
Duct	0.00	0.00	0.00	0.01	0.02	0.04
HCCI	0.00	0.00	0.00	0.01	0.01	0.01
JICF Q	0.02	0.02	0.02	0.23	0.26	0.26

7 Limitations

The data values are symbolically perturbed by their local index and then by the region index to guarantee no two critical vertices have the same function value. Unfortunately, this perturbation scheme depends on the region size parameter, and thus may produce different results as this parameter changes. For example, the tested datasets (Table 1) have all triplets identical between region sizes 32^3 and 64^3 with the exception of the Foot dataset, where about half of the triplets are mismatched. This dataset has 8-bit data values, and thus many vertices have the same value, and their order is affected by the changing perturbation scheme. The alternatives are to preprocess datasets to remove equal values at the cost of changing the analysis output, e.g., the persistence, to transform local and region indices into a global index space that imposes run-time cost, or to specify an offset array to break the ties at the cost of extra memory overhead.

The proposed query algorithms do not cache any intermediate results of the *ComponentMax* query calls, which enables us to run multiple queries in parallel, at the cost of duplicated work. In essence, building a cache amounts to a lazy construction of a global data structure, and it may be a better choice to use one of the available algorithms to build this data structure directly from the merge forest. Finally, a cached entry per merge forest graph vertex may be of a size equal to the number of regions in the domain decomposition. In contrast to a merge tree with one-to-one mapping between an arc and a component, a component can have multiple corresponding arcs in a merge forest, which requires the cache to contain not only the query result but also the list of arcs to traverse next (Fig. 12).

Authorized licensed use limited to: The University of Utah. Downloaded on July 28,2022 at 00:42:23 UTC from IEEE Xplore. Restrictions apply.



(a) ComponentMaxCached(u = 6, h = 6) = 8. (b) ComponentMaxCached(u = 3, h = 3) = 8.

Figure 12: Caching the result of the *ComponentMax* query is insufficient for a merge forest because several ways may be available to continue the traversal. For example, if we cache the result of ComponentMaxCached(u = 6, h = 6) = 8, then the query ComponentMaxCached(u = 3, h = 3) would reach the cached value at vertex 6 and incorrectly return 8. For the cache to be correct, it must also store two possible traversals toward the roots, from arc 3 and arc 5. Then, the second call to the query would start traversing from arc 5 and reach the correct maximum 9. Moreover, the cache would need to store the list of children already visited to avoid repeated traversal.

8 Conclusion

We presented topological queries that depend on the persistence of superlevel-set components, and described generic algorithms for answering these queries. These algorithms can be accelerated by a variety of data structures, such as a merge tree, a triplet merge tree, or a merge forest. After the study of the locality of these queries, we compared the performance impact on the query time between the triplet merge tree and the merge forest. The conclusion from the analysis results is that many datasets have mostly local noise, and this noise can be removed by localized and scalable techniques. The competitive performance of the *Persistence* query to the triplet merge tree is surprising because we never precompute a global structure to accelerate this query. The significantly worse locality of the *Triplet* query suggests that relevance is largely global, and thus a global data structure is more suitable than the merge forest. However, in many analysis use cases only a subset of the triplet queries is of interest, which may be local. For example, querying triplets of maxima above analysis threshold in the JICF Q dataset takes 76 s compared to the 6,677 s when computing a triplet for all maxima. Therefore, we are working on creating a benchmark set consisting of reproduced analysis use cases and also synthetic datasets to improve evaluations of topological structures. Finally, we are investigating the locality implications of persistence simplification for the Morse-Smale complex.

Acknowledgments

The authors wish to thank reviewers and Christine Pickett. This work was supported in part by NSF OAC awards 2127548, 1941085, 2138811 NSF CMMI awards 1629660, DoE award DE-FE0031880, and the Intel Graphics and Visualization Institute of XeLLENCE, and oneAPI Center of Excellence.

References

[1] M. Atzori, R. Vinuesa, A. Lozano-Durán, and P. Schlatter. Characterization of turbulent coherent structures in square duct flow. *Journal of Physics: Conference Series*, 1001:012008, Apr. 2018.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] H. A. Carr, O. Rübel, G. H. Weber, and J. P. Ahrens. Optimization and augmentation for data parallel contour trees. *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–1, 2021.

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

[8] 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.

[9] R. W. Grout, A. Gruber, H. Kolla, P.-T. Bremer, J. C. Bennett, A. Gyulassy, and J. H. Chen. A direct numerical simulation study of turbulence and flame structure in transverse jets analysed in jet-trajectory based coordinates. *Journal of Fluid Mechanics*, 706:351—383, July 2012.

[10] 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.

[11] A. Gyulassy, V. Pascucci, T. Peterka, and R. Ross. The parallel computation of Morse-Smale complexes. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 484–495, 2012.

[12] X. Huang, P. Klacansky, S. Petruzza, A. Gyulassy, P.-T. Bremer, and V. Pascucci. Distributed merge forest: a new fast and scalable approach for topological analysis at scale. In *ICS '21: 2021 International Conference on Supercomputing, Virtual Event, USA, June 14-17, 2021*, pp. 367–377, 2021.

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

[14] 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.

[15] J. Lukasczyk, C. Garth, R. Maciejewski, and J. Tierny. Localized topological simplification of scalar data. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):572–582, 2021.

[16] Z. Lukic. Nyx cosmological simulation data, 2019.

[17] 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.

[18] 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.

[19] A. Nigmetov and D. Morozov. Local-global merge tree computation with local exchanges. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19. Association for Computing Machinery, New York, NY, USA, 2019.

[20] 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.

[21] D. Smirnov and D. Morozov. Triplet merge trees. In H. Carr, I. Fujishiro, F. Sadlo, and S. Takahashi, eds., *Topological Methods in Data Analysis and Visualization V*, pp. 19–36. Springer International Publishing, Cham, 2020.

[22] K. Werner and C. Garth. Unordered task-parallel augmented merge tree construction. *IEEE Transactions on Visualization and Computer Graphics*, 27(8):3585–3596, Aug. 2021.

Authorized licensed use limited to: The University of Utah. Downloaded on July 28,2022 at 00:42:23 UTC from IEEE Xplore. Restrictions apply.