Chapter 1
Triplet Merge Trees

Dmitriy Smirnov and Dmitriy Morozov

Abstract Merge trees are fundamental data structures in computational topology. They track connected components in sublevel sets of scalar functions and can be used to compute 0-dimensional persistence diagrams, to construct contour trees on simply connected domains, and to quickly query the relationship between connected components in different sublevel sets. We introduce a representation of merge trees that tracks the nesting of their branches. We present algorithms to construct and manipulate the trees in this representation directly. We show that our algorithms are not only fast, outperforming Kruskal's algorithm, but they are easy to parallelize in shared memory using double-word compare-and-swap operations. We present experiments that illustrate the scaling of our algorithms as functions of the data size and of the number of threads.

1.1 Introduction

Merge trees are widely used in computational topology. These data structures track how components appear and merge in sublevel sets of functions, as one sweeps the threshold from negative to positive infinity. Once constructed, merge trees can be used to generate contour trees [1] or 0-dimensional persistence diagrams [2]. They are used in applications ranging from combustion to cosmology to materials science.

Most algorithms to compute merge trees are closely related to algorithms for minimum spanning tree construction. Of these, the most common is Kruskal's algorithm [3]. Recently, Carr et al. [4] introduced an algorithm that

Dmitriy Smirnov
Pomona College, Calremont, CA; e-mail: dimkasmir@gmail.com

Dmitriy Morozov
Lawrence Berkeley National Laboratory, Berkeley, CA; e-mail: dmitriy@mrzv.org

constructs merge trees by incrementally pruning extrema. It can be viewed as an adaptation of Borůvka's algorithm [5] to the merge tree problem, and, as Borůvka's, their algorithm is amenable to parallelization.

The algorithm of Bremer et al. [6] deserves special attention. It works by incrementally adding edges to the domain and updating the tree by merging paths inside it. This algorithm features several desirable properties. (1) It can process edges in a streaming fashion, without access to the full sequence. This property is most convenient when the domain is represented implicitly, and the number of edges is significantly higher than the number of vertices. (2) It can be used to combine a pair of merge trees to find the merge tree of the union of underlying domains, in sublinear time. In other words, it does not need to access those parts of the trees that don't change in the union. The need to combine merge trees comes up naturally during distributed computation [7, 8]. (3) It can be easily parallelized in shared memory, in a lock-free manner, using compare-and-swap operations for synchronization.

The algorithm of Bremer et al. would be perfect if it wasn't so slow. In theory, it can scale quadratically in the number of input vertices. In practice, it is orders of magnitude slower than Kruskal's algorithm. Section 1.3 includes this algorithm in several experiments, where it leaves a lot to be desired.

In this paper, we introduce a different representation of merge trees. Instead of recording the nesting of sublevel sets explicitly, the new *triplet representation* records the nesting of the branches of the merge tree. We present algorithms that construct merge trees in this representation directly by incrementally adding edges to the domain. They possess the same desirable properties (1)-(3) as above, and as our experiments show, the new algorithms perform better in practice than Kruskal's algorithm. The new algorithms are also sufficiently simple that they can be parallelized in shared memory using double-word compare-and-swap primitives for synchronization.

1.2 Background

Given a graph G and a scalar function $f : \operatorname{Vrt} G \to \mathbb{R}$ on its vertices, we assume that all vertex values are distinct, breaking ties lexicographically in practice. For $a \in \mathbb{R}$, the *sublevel graph at* a, denoted G_a, is the subgraph induced by the vertices whose function value does not exceed a. The *representative* of vertex u at level $a \geq f(u)$ is the vertex v with the minimum function value in the component of u in G_a. The *merge tree* of function f on graph G is the tree on the vertex set of G that has an edge (u, v), with $f(u) < f(v)$, if the component of u in $G_{f(u)}$ is a subset of the component of v in $G_{f(v)}$, and there is no vertex v' with $f(u) < f(v') < f(v)$ such that the component of u is a subset of the component of v' in $G_{f(v')}$. Figure 1.1 illustrates a function on a graph and its merge tree. (If G is disconnected, a "merge tree" is actually a

forest. We abuse terminology and don't distinguish this case, continuing to call it a tree, to not clutter the language.)

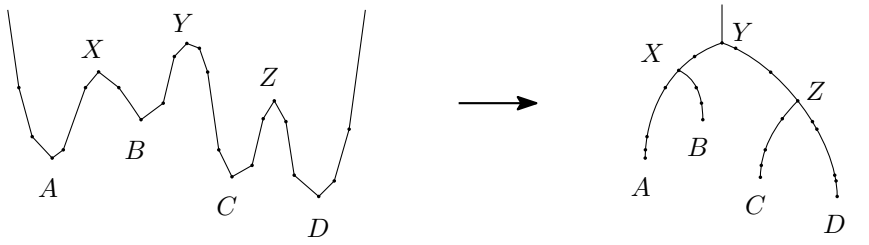

Fig. 1.1 A function on a graph and its merge tree.

Intuitively, a merge tree keeps track of the connected components in the sublevel graphs. If we sweep threshold a from $-\infty$ to ∞, as we cross values of specific vertices, they either start new components in the tree, or they are added to existing components, possibly joining multiple such components together. Such a sweep lies at the core of the most common algorithm for merge tree construction, which adapts Kruskal's algorithm, originally developed for the minimum spanning tree construction. This algorithm processes vertices in sorted order and maintains connected components in a disjoint set data structure that supports fast component identification and merging. For each vertex, it queries the representatives of the connected components of its neighbors with lower function value and unites them, recording the changes to the connected components in the merge tree.

The problem of constructing a merge tree is related to the minimum spanning tree construction. The latter can be reduced to the former by subdividing the edges of the input graph; the new vertices are assigned the edge values, while the values of the original vertices are set to $-\infty$. The merge tree of the resulting function identifies the minimum spanning tree of the original graph: degree-3 nodes of the merge tree are the vertices subdividing the edges of the minimum spanning tree.

However, the two problems are distinct. In general merge trees, the values of the minima vary, a feature important for the algorithms presented in the next section. Furthermore, the goal is to compute the tree itself rather than to find the identity of the edges. As a consequence, a folklore reduction from sorting[1] shows that the lower bound for merge tree construction is $\Omega(m + n \log n)$, where n, m are the numbers of vertices and edges in the input graph. In contrast, a minimum spanning tree can be built in $\mathrm{O}(m \cdot \alpha(m, n))$ time [9], where α is the inverse Ackermann function.

[1] Given a sequence of n values, take a star graph with n leaves and assign the input values to the leaves; assign $-\infty$ to the central vertex. The merge tree of this function is a path, with vertices assigned the input values in sorted order.

The algorithm of Bremer et al. [6], mentioned in the introduction, works as follows. It initializes the merge tree to be a set of disjoint vertices. It then processes all the edges one by one, in arbitrary order. Given an edge (u, v), it merges, in sorted order, the paths from u and v to the root of the merge tree. The correctness of the algorithm is easy to see: an edge (u, v) certifies that the two vertices belong to the same connected components in all sublevel graphs G_a, with $a \geq \max\{f(u), f(v)\}$; the connected components of the sublevel graphs that contain vertex u are represented by the vertices on the path from u to the root. We use this algorithm for comparison in the next section.

1.3 Triplet Merge Tree

Merge trees are typically stored in a graph data structure: each node stores a pointer to its parent, or to its children, or both. These pointers allow one to traverse the tree and answer various queries: for example, what is the volume of the component that contains a given vertex; or how many connected components there are at level b that have a vertex with function value below a; etc.

We are interested in an alternative representation that records global information. Instead of storing its immediate parent, each vertex stores the range of values for which it remains its own representative (i.e., the deepest vertex in its connected component), together with the reference to its representative at the end of this range. The necessary information can be thought of as a triplet of vertices (u, s, v), such that u represents itself at levels $a \in [f(u), f(s))$, and v becomes its representative at level $f(s)$. Recursively following the chain of representatives of v, we can find the representative of u at any level, which in turn allows us to answer the queries mentioned before. Figure 1.2 illustrates the triplet representation of the merge tree in Figure 1.1.

A reader familiar with persistent homology will immediately recognize pairs $(f(u), f(s))$ as the birth–death pairs in the 0-dimensional persistence diagram. Similarly, pairs (u, s) are the branches in the decomposition introduced by Pascucci et al. [10] for contour trees. The extra information stored in the triplet — which branch a given branch merges into — is crucial for our main goal: to construct and manipulate the trees directly in the triplet representation. The rest of this section introduces such algorithms and structural formalism necessary to prove their correctness.

Structure. Given a graph G with a scalar function $f : \operatorname{Vrt} G \to \mathbb{R}$, we define a *merge triplet* to be a 3-tuple of vertices, (u, s, v), such that u and v are in the same component of $G_{f(s)}$ and $f(v) < f(u) \leq f(s)$, or (u, u, u) if u is the minimum in its component of G. We define the *triplet representation* T to be a set of merge triplets. A triplet representation induces a directed graph $D(T)$, with the same vertex set as the graph G; $D(T)$ contains a directed edge (u, v),

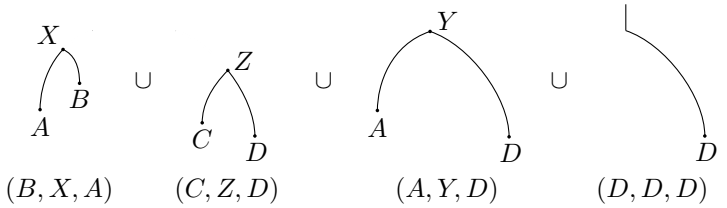

Fig. 1.2 A subset of the minimal, normalized triplet representation of the merge tree in Figure 1.1. The full representation includes triplets $(X, X, A), (Z, Z, D), (Y, Y, D)$, as well as triplets for the unlabeled degree-2 nodes.

with label s, if and only if (u, s, v) is a triplet in T. Figure 1.3 illustrates the directed graph induced by the triplet representation in Figure 1.2.

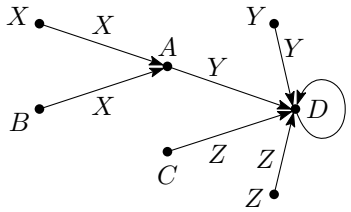

Fig. 1.3 Directed graph induced by the triplet representation in Figure 1.2, with leaves representing the unlabeled degree-2 nodes omitted.

We let $D(T, a)$ be the subgraph of $D(T)$ that consists of all vertices u with $f(u) \leq a$ and all edges (u, v) with label s such that $f(s) \leq a$. The *representative* of vertex u at level a in the triplet representation is vertex v with the lowest function value in the connected component of u in $D(T, a)$.

A representation T is *equivalent* to the graph G if for every vertex u and every level $a \geq f(u)$, the representative of u at a in G is the same as the representative of u at a in T. Similarly, we say that two representations T_1 and T_2 are equivalent, $T_1 \simeq T_2$, if every vertex, at every level, has the same representative in both T_1 and T_2.

A triplet representation is *normalized* if every vertex $u \in \operatorname{Vrt} G$ appears as the first element of exactly one triplet. It follows immediately that the digraph induced by a normalized representation is a forest, if we ignore the self-loops at the roots. (We note that the internal nodes in such a forest represent extrema of the function, while the internal nodes of the regular merge tree appear as leaves in the digraph; see Figures 1.1 and 1.3.) Furthermore, as the following lemma states, a normalized representation recovers the branches of the merge tree (but not necessarily what they merge into).

Lemma 1. *If representation T is normalized, then if $(u, s, v) \in T$, then u is its own representative for all levels a, with $f(u) \leq a < f(s)$. Furthermore, it*

is not its own representative at level $f(s)$. (The only exception are the triplets $(u, u, u) \in T$, corresponding to minima of the connected components. In this case, u represents itself at all levels.)

Proof. If T is normalized, its induced digraph $D(T)$ is a forest (ignoring the self-loops at the roots). The component of the subgraph $D(T, a)$, induced by vertices and edges below $a < f(s)$, that contains vertex u is a subtree rooted at u (unless $u = s$, in which case there is no such component). Since vertex values decrease along the directed edges, u is the minimum of its component, and, therefore, its own representative, by definition. At level $f(s)$, we add the edge (u, v) to the digraph $D(T, f(s))$. Since $f(v) < f(u)$, u stops being its own representative. □

In general, existence of triplet (u, s, v) does not imply that v represents u at level $f(s)$. We say that a representation is *minimal* if along every path in the digraph, the values of the edge labels are increasing. It follows immediately that if T is normalized and minimal, then for every triplet $(u, s, v) \in T$, v represents u at level $f(s)$.

Our main algorithm, Algorithm 3, relies on the following theorem to construct a triplet representation.

Theorem 1. *Suppose a representation T is equivalent to a graph G. Let G' denote graph G with an extra edge (u, v); assume without loss of generality that $f(u) < f(v)$. Then representation $T' = T \cup (v, v, u)$ is equivalent to graph G'.*

Proof. If representation T is equivalent to graph G, then at every level $a \in \mathbb{R}$, there is a bijection between the connected components of the sublevel graph G_a and those of the subgraph $D(T, a)$ induced by the representation. If $a < f(v)$, then neither vertex v nor the new edge is present in either, and so there is no change between G_a and G'_a and $D(T, a)$ and $D(T', a)$.

Suppose $a \geq f(v)$. If u and v are in the same connected component of G_a, then they are in the same connected component of the induced subgraph $D(T, a)$. They remain in the same connected component after we add edges (u, v) to G and (v, v, u) to $D(T, a)$. If they are in different connected components, then adding edge (u, v) to G_a merges the two connected components, but edge (v, v, u) merges the corresponding connected components in $D(T, a)$. Thus, there is a bijection between the connected components of G'_a and $D(T', a)$. □

Algorithms. It is possible to construct a triplet representation from a graph using Kruskal's algorithm. Algorithm 1 spells out the details. It uses algorithm FindDeepest(T, u), given in Algorithm 2, to find the deepest element in a connected component, using path compression. (Because we use path compression alone, the algorithm takes $\mathrm{O}(m \log n)$ steps in the worse case, on a graph with n vertices and m edges.) In the next section, we use this algorithm as a baseline for comparison.

Algorithm 1: ComputeMergeTree(G)

```
1  Sort vertices by f value;
2  foreach vertex u ∈ G do
3      T[u] ← (u, u);
4      CurDeepest[u] ← u;
5      nbrs ← {v | (u, v) ∈ G, f(v) < f(u)};
6      if #nbrs > 0 then
7          leaves ← {FindDeepest(T, v) : v ∈ nbrs};
8          v̂ ← oldest(leaves);
9          T[u] ← (u, v̂);
10         if #leaves > 1 then
11             foreach v ∈ leaves \ {v̂} do
12                 T[v] ← (u, v̂)
13 return T
```

Algorithm 2: FindDeepest(T, u)

```
1  û ← CurDeepest[u]
2  (_, v) ← T[û]
3  while û ≠ v do
4      û ← CurDeepest[v]
5      (_, v) ← T[û]
6  d ← û
7  û ← CurDeepest[u]
8  (_, v) ← T[û]
9  while û ≠ v do
10     û ← CurDeepest[v]
11     CurDeepest[v] ← d
12     (_, v) ← T[û]
13 CurDeepest[u] ← d
14 return d
```

Our main contribution is Algorithm 3 and its auxiliary Algorithms 4, 5, and 6. Together they construct a normalized minimal triplet representation by processing the edges of the input graph, adding them one by one into the triplet representation. Crucially, it doesn't matter in what order the edges are processed; we harness this opportunity for parallelization in the next section.

Because we maintain a normalized representation, each vertex occurs as the first component of exactly one triplet. In the pseudo-code, we use the associative map notation, $T[u] \leftarrow (s, v)$, to record triplet (u, s, v) and, similarly, $(s, v) \leftarrow T[u]$ to access triplet $(u, s, v) \in T$.

The first loop of Algorithm 3 initializes the vertices of representation T. The result is trivially equivalent to graph G without any edges.

The second loop employs the main workhorse, Merge(T, u, s, v) operation, presented in Algorithm 4. Given a normalized representation T, it finds a

Algorithm 3: ComputeMergeTree2(G)

```
1  foreach vertex u ∈ G do
2      T[u] ← (u, u);
3  foreach edge (u, v) ∈ G do
4      if f(u) < f(v) then
5          Merge(T, v, v, u)
6      else
7          Merge(T, u, u, v)
8  foreach u ∈ T do
9      Repair(T, u)
10 return T
```

normalized representation equivalent to $T \cup (u, s, v)$. Figure 1.4 illustrates a single transformation performed by the Merge algorithm.

Algorithm 4: Merge(T, u, s, v)

```
1 (u', s_u, u'') ← Representative(T, u, f(s));
2 (v', s_v, v'') ← Representative(T, v, f(s));
3 if u' = v' then return;
4 if f(v') < f(u') then
5     swap((u', s_u, u''), (v', s_v, v''))
6 T[v'] ← (s, u');
7 Merge(T, u', s_v, v'') ;
```

Because Merge(T, u, s, v) does not guarantee to preserve minimality, Algorithm 3 restores this property in the third loop by calling Repair(T, u), presented in Algorithm 6, which finds for each edge the next edge with a higher label. One could maintain minimality on the fly by keeping back-pointers to the branches that merge into the given branch (making the induced graph undirected), but this would obstruct our goal of lock-free shared-memory parallelism in the next section. So we opt to use the Repair procedure, instead.

Algorithm 5: Representative(T, u, a)

```
1 (s, v) ← T[u]
2 while f(s) ≤ a and s ≠ v do
3     u ← v
4     (s, v) ← T[u]
5 return (u, s, v)
```

Algorithm 6: Repair(T, u)

```
1 (s, _) ← T[u]
2 v ← Representative(T, u, f(s))
3 if u ≠ v then
4     T[u] ← (s, v)
5 return T[u]
```

Correctness. The core of the algorithm is in the Merge procedure, in Algorithm 4. Assuming that representation T is normalized, we want to show that

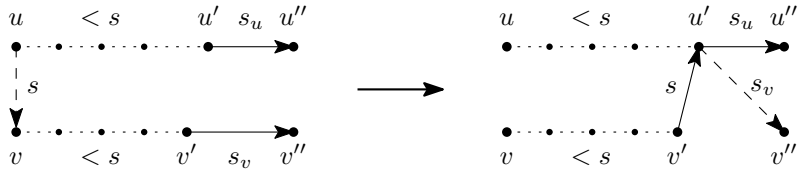

Fig. 1.4 Induced graph of triplet representations before and after a single transformation in the Merge(u, s, v) algorithm.

after Merge(u, s, v) returns, we get a normalized representation equivalent to $T \cup (u, s, v)$.

Lemma 2. *Let T denote the triplet representation at the beginning of Algorithm 4, and T' the representation before the recursive call in step 7. Then, $T \cup (u, s, v) \simeq T' \cup (u', s_v, v'')$.*

Proof. Fix a level $a \in \mathbb{R}$. Let $D_1 = D(T \cup (u, s, v), a)$ and $D_2 = D(T' \cup (u', s_v, v''), a)$ denote the directed graphs induced by the representations at the beginning of the algorithm and before the recursive call; see Figure 1.4. Let x and y be any two vertices in the same connected component in D_1, and let p be the (undirected) path that connects them. If p does not contain edges (u, s, v) or (v', s_v, v''), then it exists in D_2, and therefore x and y are still connected. If p goes through edge (u, s, v), we can replace that edge by the path $u \dots u', (v', s, u'), v' \dots v$ in D_2, since all the edges in the subpaths connecting vertices u and v have values that don't exceed $f(s)$. If p contains edge (v', s_v, v''), we can replace the edge by path $(v', s, u'), (u', s_v, v'')$ since $f(s) < f(s_v)$.

Similarly, in the other direction. If x and y are in the same component of D_2, then either the path connecting them does not contain edges (v', s, u') or (u', s_v, v'') and, therefore, exists in D_1; or we can replace (u', s_v, v'') by $u' \dots u, (u, s, v), v \dots v', (v', s_v, v'')$, and (v', s, u') by $v' \dots v$, $(u, s, v), u \dots u'$.

In summary, two vertices are connected in D_1 if and only if they are connected in D_2. Therefore, the two triplet representations are equivalent. □

At every point of the Merge algorithm, the representation remains normalized. When the exit condition is detected ($u' = v'$), the extra triplet does not add any new information, i.e., in this case $T \simeq T \cup (u, s, v)$. The recursive calls are made with vertices u' and v''. By definition, $f(v'') < f(v)$. Consequently, the algorithm is always making progress and must eventually terminate. (The progress is most evident in the induced digraph: the algorithm always moves up the directed paths in $D(T)$.) We get the following corollary.

Corollary 1. *Algorithm 4 updates normalized triplet representation T, making it equivalent to representation $T \cup (u, s, v)$. The representation remains normalized.*

It follows that Algorithm 3 computes the correct result, as summarized in the following theorem.

Theorem 2. *Algorithm 3 computes a normalized minimal triplet representation equivalent to the input graph G.*

Proof. The first for-loop in Algorithm 3 initializes representation T to be equivalent to a graph on the same vertex set as G, but without any edges. The second for-loop uses Algorithm 4 to update the representation, adding the edges of the graph. The correctness of this loop follows from Theorem 1 and Corollary 1. The third for-loop ensures that the representation is not only normalized, but also minimal by finding the representative of every vertex at the first level where it does not represent itself; its correctness follows from Lemma 1. □

Evaluation. Throughout the paper we use three data sets to evaluate the algorithms. The first, Z2, is a 512^3 snapshot of a cosmological simulation. The second, Vertebra, is a 512^3 scan of a head aneurysm, once available online as part of the `volvis.org` collection of data sets. The third, Pumice, is a $640^2 \times 540$ signed distance function to a porous material.

All experiments were performed on a compute node with two sockets, each with a 16-core Intel Xeon Processor E5-2698 v3 ("Haswell") at 2.3 GHz.

Figures 1.5, 1.6, and 1.7 illustrate the scaling of Algorithm 1 (labeled "Kruskal's") and Algorithm 3 (labeled "Triplet") as a function of input size; the three inputs were downsampled to the sizes specified on the x-axis. They also show two data points each[2] of the path merging algorithm of Bremer et al. (labeled "Path merging"). The salient point in all cases is that Algorithm 3 is not only competitive with Algorithm 1, but performs significantly better for larger domain sizes, more than five times better for 512^3 Z2 input. The path merging algorithm is slower and scales significantly worse.

1.4 Shared Memory

We adapt our new algorithms to allow multiple threads to concurrently modify the triplet representation in shared memory. We use compare-and-swap (CAS) primitive for synchronization between the threads. This primitive atomically checks whether a variable contains a given `expected` value; if it does, CAS replaces it with the given `desired` value. The operation is equivalent to atomically executing Algorithm 7.

Since the variables we need to update atomically store pairs of values, (s, v), we use double-word compare-and-swap (DWCAS) operations. These should not be confused with double compare-and-swap (DCAS) operations, which

[2] We stopped at two data points because by the third, the jobs exhausted the 4-hour wallclock request.

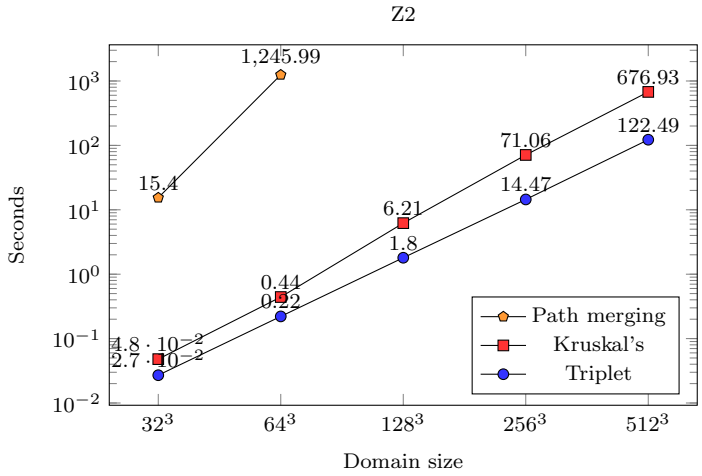

Fig. 1.5 Running time of the three algorithms as a function of input size, on downsampled Z2 data set.

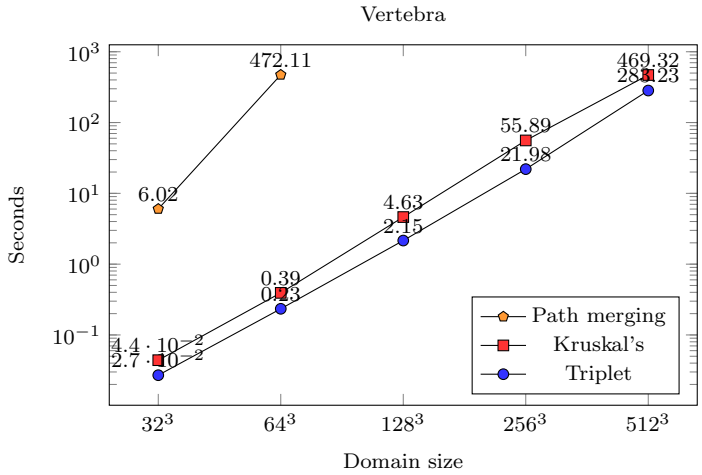

Fig. 1.6 Running time of the three algorithms as a function of input size, on downsampled Vertebra data set.

Algorithm 7: CAS(v, `expected`, `desired`)

```
1 if v = expected then
2     v ← desired;
3     return True
4 else
5     return False
```

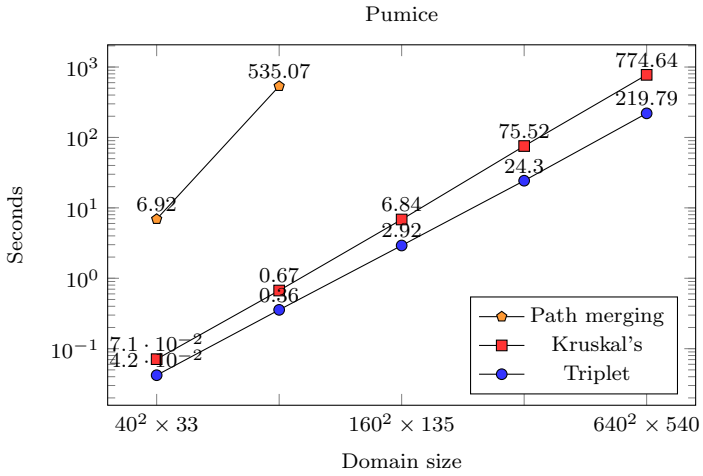

Fig. 1.7 Running time of the three algorithms as a function of input size, on downsampled Pumice data set.

Algorithm 8: Parallel ComputeMergeTree2(G)

```
1  foreach vertex u ∈ G do in parallel
2      T[u] ← (u, u);
3  foreach edge (u, v) ∈ G do in parallel
4      if f(v) < f(u) then
5          Merge(T, u, u, v)
6      else
7          Merge(T, v, v, u)
8  foreach u ∈ T do in parallel
9      Repair(T, u)
10 return T
```

update two arbitrary words in memory. DWCAS updates two *contiguous* words in memory, and, unlike DCAS, it is supported in modern hardware.[3]

Algorithm 8 adapts Algorithm 3 to the parallel setting by executing its for-loops in parallel. We assume that the triplet representation T is stored in a container that allows concurrent insertion of elements (for example, a concurrent hash map), so that the first for-loop can execute in parallel.

The second for-loop runs in parallel over the edges of the input graph and invokes the Merge algorithm, Algorithm 9, adjusted to use DWCAS for synchronization. The third for-loop performs Repair, as before.

[3] On x86-64 architecture, where we conducted all our experiments, DWCAS is performed by instruction `CMPXCHG16B`. It is automatically emitted by the C++ compilers, when provided with appropriate flags.

Algorithm 9: Parallel Merge(T, u, s, v)

```
   ▷ equivalent to (u, s_u, u') ← Representative(T, u, f(s))
 1 (s_u, u') ← T[u];
 2 if f(s_u) < f(s) then
 3     return Merge(T, u', s, v)
   ▷ equivalent to (v, s_v, v') ← Representative(T, v, f(s))
 4 (s_v, v') ← T[v];
 5 if f(s_v) < f(s) then
 6     return Merge(T, u, s, v')
 7 if u = v then return;
 8 if f(v) < f(u) then
 9     swap((u, s_u, u'), (v, s_v, v'))
10 if DWCAS(T[v], (s_v, v'), (s, u)) then
11     Merge(T, u, s_v, v');
12 else
13     Merge(T, u, s, v);
```

Correctness. To understand the correctness of the second for-loop and Algorithm 9, we interpret the state of the data structure as the normalized representation T, together with a complete list of triplets, (u, u, v) or (v, v, u), one for each edge, as prescribed by Theorem 1. Each invocation of Algorithm 9 is assigned one of the outstanding triplets — the algorithm receives the triplet as the arguments u, s, v. In a single instantiation, before or via the recursive call, it transforms the state of the data structure:

1. by modifying the triplet it's assigned, from (u, s, v) to (u', s, v) or (u, s, v'), via the recursive calls in lines 3 or 6;
2. by modifying the triplet representation T in line 10 and replacing the triplet (u, s, v) with (u, s_v, v'), in line 11;
3. by implicitly removing the triplet by returning in line 7.

Each such operation, when applied in isolation, keeps the state before the transformation equivalent to the state after the transformation. This follows from Lemmas 1 and 2 (or very similar proofs).

$$T \cup (u_1, s_1, v_1) \cup \ldots \cup (u, s, v) \cup \ldots \cup (u_m, s_m, v_m)$$

$$\Big\downarrow \simeq$$

$$T' \cup (u_1, s_1, m_1) \cup \ldots \cup (u, s_v, v') \cup \ldots \cup (u_m, s_m, v_m)$$

We claim that if multiple invocations of the algorithm modify the state concurrently, the transformations that they apply are linearizable, in the sense of Herlihy and Wing [11].

To see why this is so, we identify the linearization points, at which the transformations appear to occur atomically. In case of the first two recursive

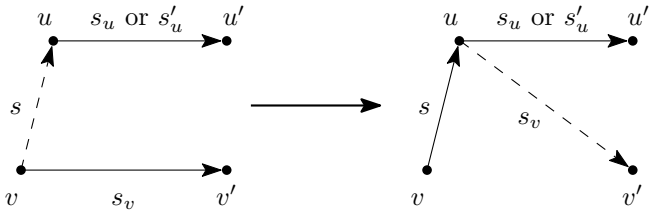

Fig. 1.8 The effect of the DWCAS transformation in line 10 of Algorithm 9.

calls, in lines 3 and 6, the preceding reads in lines 1 and 4 act as linearization points. Even though there are no side effects — different invocations of Algorithm 9 don't see each other's triplets, and the triplet representation is not modified — the operation is equivalent to atomically deciding that $T \cup (u, s, v)$ is equivalent to $T \cup (u', s, v)$ or $T \cup (u, s, v')$.

If the condition $u = v$ in line 7 is satisfied, then triplet (u, s, v) is redundant and the return statement in that line is equivalent to its removal, which also appears atomically.

What happens when we reach line 10? In this case, the function values are ordered: $f(u) < f(v) < f(s) < f(s_v)$. When this is true, the representation $T \cup (v, s, u)$ is equivalent to the representation $T' \cup (u, s_v, v')$, where T' has triplet (v, s_v, v') replaced by the triplet (v, s, u); see Figure 1.8. (The proof here is similar to the proof of Lemma 2.) If DWCAS in line 10 succeeds, then this transformation appears atomically. If it fails, we simply retry via the recursive call in line 13. We note that in this case we don't guarantee that the triplet (u, s_u, u') is not changed in the representation T, i.e., the edge going out of the vertex u in the induced digraph may change, so that its new label s'_u is such that $f(s'_u) < f(s)$. In this case, the transformation is still valid — the representations remain equivalent — but the transformed representation ceases to be minimal. Since we already do not maintain minimality during the second for-loop of Algorithm 8, but rather restore it via the Repair procedure in the third for-loop, this situation requires no special handling.

Finally, the third for-loop of Algorithm 8 requires no special attention since Repair procedure, Algorithm 6, can already execute in parallel. Since it only changes the target of any edge in the diagraph, i.e., only the third component of any triplet, the result that it obtains is the same even if other threads are simultaneously modifying the triplets it encounters. Suppose Repair changes entry $T[u]$ from (s, v) to (s, v'), and suppose a different Repair procedure queries the entry $T[u]$ as part of its search for a representative of a vertex u' at level a. If $a < f(s)$, then the change is irrelevant, since u is the desired representative. If $a \geq f(s)$, then the search needs to test the entry $T[v']$, since if the original Repair procedure was changing $T[u]$ from (s, v) to (s, v'), then all the edges on the path from v to v' have labels s_i, with $f(s_i) < f(s)$. It makes no difference whether the second Repair procedure goes directly to v' or traverses the original path.

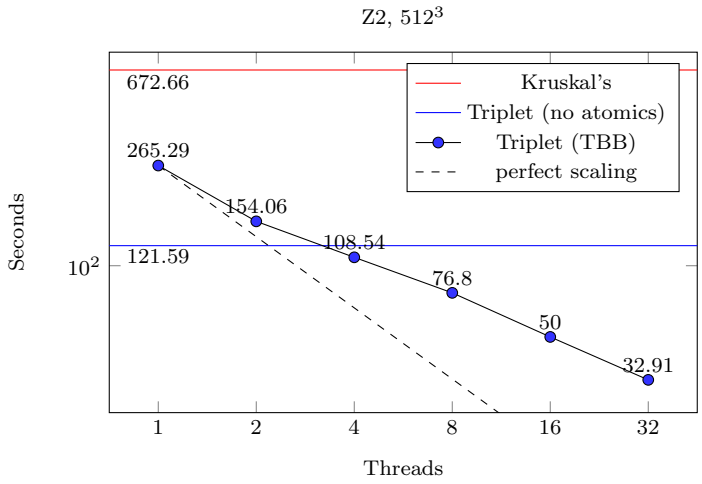

Fig. 1.9 Scaling of Algorithm 9 as a function of the number of threads, on Z2 dataset.

Evaluation. We perform experiments on the same datasets, Z2, Vertebra, Pumice, as in the previous section, but now we vary the number of threads used for the computation. We use Intel's Thread Building Blocks library for parallelization, using its `concurrent_unordered_map` to store the triplet representation.

As a baseline, we compare to Algorithm 1 and Algorithm 3. The significance of the latter is that unlike Algorithm 8, it's implemented without atomic variables.[4] Figures 1.9, 1.10, 1.11 illustrate the results of our experiments. Unfortunately, simply turning on atomics roughly doubles the running time. As the thread count increases, the parallel algorithm outperforms the serial (when using 4 threads in all cases). Although the scaling trails off, we believe the higher thread counts are still worthwhile, especially, in the (common) situation where those cores would remain idle otherwise.

1.5 Conclusion

We have described a new representation of merge trees. Instead of recording the nesting of sublevel set components, it records the nesting of the branches of merge trees. We have presented algorithms to construct the merge trees in this representations directly, as well as their parallel versions, together with experimental results demonstrating that these algorithms are efficient in practice.

[4] This is important because in C++, once a variable is declared `std::atomic`, all operations on it are protected by atomic primitives and incur the corresponding overheads.

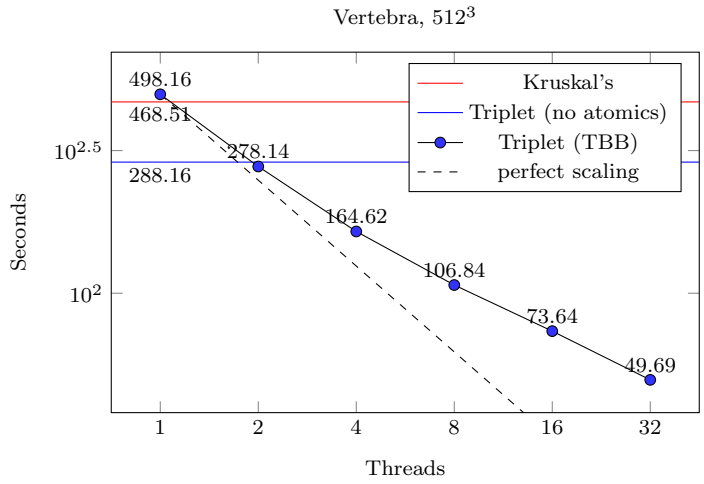

Fig. 1.10 Scaling of Algorithm 9 as a function of the number of threads, on Vertebra dataset.

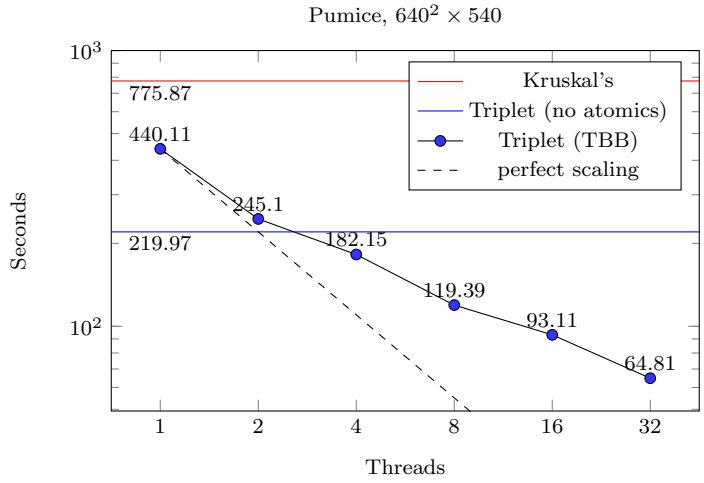

Fig. 1.11 Scaling of Algorithm 9 as a function of the number of threads, on Pumice dataset.

Conspicuously missing from our paper is the complexity analysis of the new algorithms. An $O(mn)$ upper bound for Algorithm 3 is obvious: for each edge, the algorithm touches each vertex at most once (in the amortized sense). It is also not difficult to construct an example on which Algorithm 3 would take quadratic time. However, all such examples that we know of require specifying not only the graph and the function, but also the sequence in which edges must be added. This leaves us hopeful that it's possible to show that

the randomized version of Algorithm 3 is also efficient in theory. We view this as the main open question left by our paper.

References

1. Hamish Carr, Jack Snoeyink, and Ulrike Axen. Computing contour trees in all dimensions. *Computational Geometry—Theory and Applications*, 24(2):75–94, 2003.
2. Herbert Edelsbrunner and John Harer. *Computational Topology. An Introduction.* American Mathematical Society, Providence, Rhode Island, 2010.
3. Thomas Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction To Algorithms.* MIT Press, 2009.
4. Hamish Carr, Gunther H. Weber, Chris Sewell, and James Ahrens. Parallel peak pruning for scalable SMP contour tree computation. In *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, 2016.
5. Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar Borůvka on minimum spanning tree problem: translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233:3–36, 2001.
6. Peer-Timo Bremer, Gunther H. Weber, Julien Tierny, Valerio Pascucci, Marcus S. Day, and John B. Bell. Interactive exploration and analysis of large scale simulations using topology-based data segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 17(9):1307–1325, 2011.
7. Valerio Pascucci and Kree Cole-McLaughlin. Parallel computation of the topology of level sets. *Algorithmica*, 38(1):249–268, 2003.
8. Dmitriy Morozov and Gunther H. Weber. Distributed merge trees. In *Proceedings of the Annual Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 93–102, 2013.
9. Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the Association for Computing Machinery*, 47:1028–1047, 2000.
10. Valerio Pascucci, Kree Cole-McLaughlin, and Georgio Scorzelli. *The Toporrery: Computation and Presentation of Multi-Resolution Topology*, pages 19–40. Springer-Verlag, 2009.
11. Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12:463–492, 1990.