Unordered Task-Parallel Augmented Merge Tree Construction

Kilian Werner and Christoph Garth

Abstract—Contemporary scientific data sets require fast and scalable topological analysis to enable visualization, simplification and interaction. Within this field, parallel merge tree construction has seen abundant recent contributions, with a trend of decentralized, task-parallel or SMP-oriented algorithms dominating in terms of total runtime. However, none of these recent approaches computed complete merge trees on distributed systems, leaving this field to traditional divide & conquer approaches. This article introduces a scalable, parallel and distributed algorithm for merge tree construction outperforming the previously fastest distributed solution by a factor of around three. This is achieved by a task-parallel identification of individual merge tree arcs by growing regions around critical points in the data, without any need for ordered progression or global data structures, based on a novel insight introducing a sufficient local boundary for region growth.

Index Terms—Scientific visualization, topological data analysis, task parallelism, distributed architecture

1 Introduction

With increasing size and complexity of scientific data arising from simulations and measurements, methods for efficient visualization and interactive exploration are becoming more and more important, forming a crucial component of modern, computational science. Topology-based methods have proven to reveal fundamental global features in such data, allowing for automatic analysis and simplification, and laying a basis for many such visualization methods [1], [2], [3].

However, this very same increase in size and complexity of data brings sequential algorithms of visualization and analysis to their feasible runtime limits. Scalable parallel reformulations have become necessary but proved especially challenging for topological methods that are often sequential in nature, due to their global feature-oriented concepts.

One of the most utilized [4], [5], [6], [7], [8], [9] topological concepts is the contour tree that represents level-set components of scalar functions on discretized manifold domains.

An augmented contour tree in addition embeds all vertices of the discretization to their corresponding level-set representation in the tree, allowing an additional bandwidth of applications [10], [11], [12].

In this paper we present work on the construction of the merge trees, from which the contour tree can be obtained with a well understood merging step, which is usually not the bottleneck in contour tree construction.

The global nature of these trees manifests in their well established sequential construction algorithm [13], that performs a single ordered scan through the data and manages a union-find data structure to track level sets.

Topological structures, and in particular the augmented merge trees, have considerably large memory footprints during computation, which, for larger data sets, exceeds shared-memory system capacity.

Therefore, the necessity for parallel and distributed merge tree construction algorithms, combined with the aforementioned intrinsic challenges to derive them, has led to a patched field of algorithms tailored for parallel execution or distributed systems, that received a lot of attention in recent years (see Section 2.2).

Within this field, a contemporary trend towards fine-grained parallelism, using e.g., task-parallel [14] or data-parallel [15] solutions, outperform more traditional divide & conquer strategies [16] on shared-memory systems. This is not surprising, since the core idea of maintaining a high processor occupancy through latency hiding and compensating load imbalance (which is found both in SIMD warps and task-parallel paradigms) fits the I/O heavy problem, with a tendency towards unpredictable load imbalance well. Additionally, the high-level concepts of task-parallel problem descriptions allow for easy portability, while machine-oriented low-level solutions do not fit the ever increasing size of modern clusters.

However, existing solutions within this trend either rely on sorted progression per arc [14], or global data structures [15], [17], [18], [19]. Therefore, improved versions [20] of the traditional divide & conquer remain unchallenged in the field of distributed merge tree construction.

Thus, the construction of the merge trees beyond the scope of shared-memory is hurt by either the need for data-boundary spanning, sorted progression, by some necessary global structures, or by the fan-in stages of traditional divide & conquer approaches. There are some specialized solutions that do not depend on shared-memory, however they compute a quantized approximation [21] or distributed

• *The authors are with Scientific Visualization Lab, University of Kaiserslautern, 67663 Kaiserslautern, Germany. E-mail: kwerner@rhrk.uni-kl.de, garth@cs.uni-kl.de.*

Manuscript received 20 Oct. 2020; revised 6 Apr. 2021; accepted 24 Apr. 2021. Date of publication 30 Apr. 2021; date of current version 30 June 2021. (Corresponding author: Kilian Werner.) Recommended for acceptance by I. Fujishiro. Digital Object Identifier no. 10.1109/TVCG.2021.3076875

representation [22] instead of the complete, un-modified contour tree.

In this context, we make the following contributions. We extend upon the monotone path-based core idea of methods, that do not rely on locally ordered progression [15], [18], [22], by deriving a novel insight of a "local boundary". This avoids global data structures, resulting in the identification of locally restricted, independent tasks, that can span beyond shared-memory and run on distributed computation nodes concurrently.

We derive a novel merge tree construction algorithm, that aims to bring strengths of recent, fine-grained solutions to distributed systems and present benchmarked runtimes and scalability on different data sets. The results show, that the novel algorithm performs almost on par with the current state of the art solution [14] on a shared-memory system, but is able to outperform the previously fastest distributed merge tree constructions [20], [23] by a factor of two to three, even though it constructs the augmented instead of un-augmented merge trees. Additionally, a proof-of-concept implementation of the SIMD-Hybrid variant is derived and benchmarked, and it demonstrates speed-ups of up to 20x.

2 Background and Previous Work

2.1 Preliminaries

Consider a piecewise-linear (PL), real-valued function f on the triangulation M of a d-manifold. We require that the restriction of f to the set $V(M)$ of vertices of M is injective, which can be achieved by slight function value perturbations achieving simulation of simplicity [24]. A *level set* is the pre-image $f^{-1}(h)$ of a given *level* $h \in \mathbb{R}$. A *sub-level set* $f^{-1}_{-\infty}(h)$ is the pre-image of the interval $(-\infty, h]$ - or *sur-level set* $f^{-1}_{+\infty}(h)$ for the interval $[h, \infty)$ respectively. A *contour* is a single connected component of any such level-set. Denote the contour containing a vertex p by $f^{-1}(h)_p$. Define the equivalence relation $\sim$ on vertices $p_1, p_2 \in V(M)$ as: $p_1 \sim p_2 \Leftrightarrow f(p_1) = f(p_2) \wedge p_2 \in f^{-1}(f(p_1))_{p_1}$. The *Reeb graph* of M is the quotient space $R(f) = M/\sim$. On domains that are topologically equivalent to a sphere of the same dimensionality $R(f)$ will be loop-free and called the *contour tree*. The *join tree* and *split tree* are defined similarly with regard to an equivalence relation on sub-level set components and sur-level set components respectively. Both join and split tree are called merge trees. Note that while contour, join and split trees are defined as quotient spaces, they are usually implicitly used as graphs. In these graphs nodes are vertices at which the number of connected components changes and edges follow the connectedness in the quotient space. For triangulations, all critical points and thus all tree nodes of f are guaranteed to be at vertices [25]. If v is such a vertex, we denote the corresponding node in the join tree, split tree or contour tree as the join node, split node or contour node $\tilde{v}$.

Intuitively, the join tree contracts all points that belong to a given sub-level set component and have the same function value to a point, see Fig. 1. Split and contour tree follow similarly for sur-level sets and level-sets. With this, the contour tree is a powerful topological abstraction, building a kind of skeleton of geometry with respect to a scalar field. Within this setting, each local minimum is a vertex with no smaller-valued neighbors and creates an isolated connected component of the sub-level set at its value, and thus it appears as a leaf in the join tree. As sub-level set components grow with increasing level h, vertex values of the data will gradually be reached, and corresponding vertices will join connected components. All vertices where such connected components merge with another will appear as inner nodes in the join tree, and they will again represent a now merged sub-level set connected component themselves. With edges connecting nodes to the node where their represented connected component merges, and vertices that are not nodes in the tree mapped to the edge that represents the sub-level set component they initially belonged to, the augmented join tree is complete. The augmented split tree follows symmetrically for sur-level sets and local maxima.

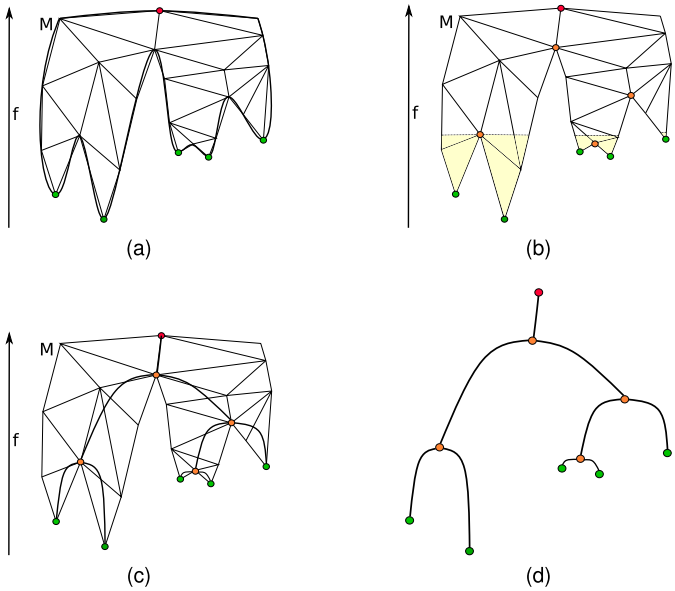

Fig. 1. An exemplary domain M and a given triangulation of it, with the scalar function f corresponding to the height (a). Local minima (green) and maxima (red) are illustrated as circles. In (b) vertices at which the number of connected components of sub-level sets change are illustrated as additional circles (orange). One such sub-level set is highlighted; it contains three components, one of which is created at this level by a join. The join tree (which is identical to the contour tree in this example due to the lack of split nodes) is shown embedded in the triangulation (c). (d) shows the join tree as a quotient space that is trivially interpretable as a graph.

The identification of local minima is trivial and embarassingly parallel, and the augmented contour tree can be constructed in parallel and in linear runtime from the augmented join and split tree [14]. The relevant task of our algorithm therefore is to identify inner nodes, edges and augmentations of the merge trees. This is also the aspect in which most contemporary solutions differ.

2.2 Contour Tree Computation

The abundant efforts in parallel contour tree construction have created a varied landscape of algorithms with different strengths, weaknesses and target hardware architectures. In the following, we group existing solutions into three families, giving a short characterization of each. Note that we will not address general Reeb graph construction in high detail as those algorithms are generally outperformed by their more specialized versions on simply connected domains. Especially within the scope of this paper instead of comparing to the

fastest known augmented Reeb graph construction [26], we compare to its faster version for merge trees [27].

2.2.1 Divide & Conquer Sequential Construction

Carr *et al.* [13] introduced constructing the contour tree from two merge trees; a method most existing algorithms utilize. Here, the join tree is constructed by a sequential sweep through all vertices, sorted by value. With a union-find data structure, adjacency to existing sub-level set components can be tested efficiently. Vertices that are not adjacent to any existing sub-level set appear as leafs in the tree. Vertices that are adjacent to exactly one sub-level set component just join that sub-level set (joining their augmentation too). Vertices that are adjacent to multiple components merge them and appear as inner nodes in the tree. As always, split tree computation follows symmetrically and will not be mentioned again below. Identification of nodes in the tree therefore relies on the sub-level set component configuration adjacent to a vertex at a level "just before" the vertices own value. This configuration is tracked with the union-find data structure and updated for each vertex in total order.

This naturally sequential approach has received a divide & conquer based parallelization [16], which divides the data into preferably equal chunks, computes the join tree sequentially for each chunk and then stitches those local trees together at common vertices on chunk boundaries. For this, a hierarchical fan-in is necessary, with each two local trees being merged into a new, partially-merged tree, that spans the combination of their chunks, repeating until two partially merged trees -spanning half the data each- merge into the complete join tree.

Some of the limitations of this approach could be overcome by an optimization by Landge *et al.* [20]. Here, the local trees and -after each merging step- the partially merged trees are pruned of any information that does not cross the remaining chunk-boundaries, before climbing the next merge step in the fan-in process. Depending on data complexity and chunk distribution, this reduces the communication and work in the fan-in stages dramatically and allows for a feasible treatment of augmentations, that would otherwise require the last fan-in merge to do approximately as much work as the sequential tree construction.

Another approach divides the data among level-sets, instead of spatial concepts [28]. Depending on the data, this approach might reduce chunk-boundary sizes and thus computational and communication overhead.

Members of this family of divide & conquer strategies [16], [20], [28], [29], [30] (and [31] for Reeb graphs) share common strengths and weaknesses. The number of parallel worker agents is limited to the number of data chunks. A growing number of data chunks increases not only synchronization effort, but actually introduces extra work. Lastly, the fan-in reduction stages of such approaches suffer from increased work and decreased involved processors, the higher up in the hierarchy the process reaches. All of this harms scalability of such approaches onto massively parallel systems like clusters and accelerators.

2.2.2 Fine-Grained Per-Arc Construction

A second family of algorithms focuses on mostly independent identification of individual arcs. As nodes in the join tree are always connected by monotone paths in the original data [25], tracing monotone paths from or to local minima and scanning for potential vertices where connected components may merge, allows for join tree construction. With this focus on local features, a less centralized workflow improves parallelization.

For example Chiang *et al.* [19] utilize the observation, that all inner nodes in the join tree have to be Morse critical saddle points. Otherwise, there could not ever be multiple, not yet connected sub-level set components to merge at the vertex. Restricting the data to local minima and Morse-critical saddle points, and following all remaining monotone paths from each remaining vertex downwards, yields all edges of the join tree. However, to be able to reconstruct the internal hierarchy of the join tree, these monotone paths have to be processed in sequential order of the remaining critical saddle vertices, and due to the restriction of the algorithm to critical points no augmentation can be computed.

Restricting the data even further, Maadasamy *et al.* [18] test for all critical saddle vertices whether their smaller neighbors are connected, by performing overlapping breadth-first searches in the sub-level set to their values.

Similarily, Carr *et al.* [15], [32] introduced an SMP oriented algorithm. From each vertex in the data an arbitrary descending monotone path is followed, until reaching a local minimum, labelling the vertices with that local minimum. In a second step, for each edge in the data it is checked, whether it connects differently labelled vertices and if so, the higher valued vertex of the edge is marked as potentially joining the two connected components of those minima. A global list of those $saddle\ candidates$ is sorted by function value, and the smallest saddle candidate per local minimum is identified as its join node.

While algorithms of this second family [17], [18], [19], [32], [33] have parts that expose fine-grained parallelism suitable for GPU or distributed settings, they introduce considerable extra work and communication, by tracing paths through the entire height of the tree. Most do not produce an augmentation and still contain globally sequential parts, harming scalability.

2.2.3 Independent Arc Growth

Gueuenet *et al.* [14] recently presented a task-parallel algorithm, with minimal dependencies between individual arc constructions. As it is neither based on a divide & conquer approach, nor requires any global or shared data structures, but instead constructs every arc in the join tree independently (with parent arcs depending on child arcs only), we classify this approach as the first in a third family of algorithms. As already hinted at in [17], finding the join node for a local minimum is ultimately a semi-local operation. In fact the smallest valued Morse-critical saddle point that is reachable by a local minimum m and at least one local minimum other than m through monotone paths is the join tree saddle for m. This observation will be discussed in Section 3.2, as our novel algorithm is also part of this third family.

In [14], again monotone paths are followed from minima, but instead of following all or one of them to their very end, only the path with the smallest next-to-visit vertex is extended, allowing to stop progression, once the join node is

reached. Again, all visited vertices are associated with the local minimum (and also added to its augmentation). The first node visited by a local minimum m that has smaller neighbors not visited by m is the join node for m. By merging the visited labels of merging components with a union-find data structure, the whole join tree is constructed in this manner. This method collects the augmentation on the fly and guarantees to only visit each vertex in the data exactly once. No communication or ordering spanning unrelated join tree arcs is necessary, leaving only the parent-child relations of the tree as inherent interdependencies. For an additional major speedup, explicit join tree construction can be skipped, as soon as only paths for one connected component are still followed. The authors call this trunk-skipping and demonstrate superior sequential and parallel runtime, while still computing the entire augmented contour tree. Since the progression of each connected component has to be strictly ordered to achieve correctness, parallel scalability is limited to the number of simultaneously growing components. This becomes problematic in a distributed setting, where the growth of a component spanning multiple computation nodes would require a sorted progression among all involved computation nodes, hurting distributed scalability.

Sarkar *et al.* [34] derived an algorithm on ad-hoc sensor networks, that is actually part of this third mentioned family of algorithms, as it computes arcs of the join tree over the network agents in parallel and independently of any unrelated arcs, of course without the use of global data structures, which would not be possible in an ad-hoc network. The general method is very similar to [32], with the global sorting of edge lists replaced by a broadcast.

The algorithm introduced in this paper is very similar to [14], but tries to overcome the mentioned limits, by allowing parallel work on each single component growth, without going back to the second families reliance on global data structures and the double work of tracing monotone paths through their entire length. This is made possible by the use of local restrictions for monotone path progression. We inherit sequential runtimes thanks to trunk-skipping and output-sensitiviy from [14], but try to challenge divide & conquer techniques in distributed settings [20], as we do not rely on shared memory. We compute the entire, un-altered augmented contour tree with an option to collect it on a single node, or to store a distributed representation. Our algorithm can also be adapted to a hybrid CPU-GPU solution.

2.2.4 Specialized Solutions

Morozov *et al.* [22], [23] utilize globally extending monotone paths to calculate the augmentation of the contour tree in a distributed representation. While they do minimize communication and allow for the construction of an augmented contour tree in a distributed setting, the contour tree is never constructed explicitly, but rather stored in an implicit per-node representation, which differs greatly from the scope of this paper.

Carr *et al.* [21] developed a hybrid CPU-GPU and also distributed algorithm for augmented contour tree construction. By rasterizing the domain along quantized levels and computing the topology of the resulting fragments with a union-find data structure, a quantized approximation of the contour tree is constructed, which is augmented with respect to not vertices, but fragments. This again differs greatly from the scope discussed here.

3 Concept: Locally Restricted Monotone Paths

As mentioned above, contour tree construction is substantially achieved by constructing both merge trees and combining them; the approach presented in this paper does follow this idea. Combining is usually not the bottleneck for computational runtime and a partially parallel combination algorithm for merge trees exists [14].

Join and split tree construction is symmetrical and we will only be discussing join tree construction in the remainder of this paper. The join tree is taken to track sub-level sets; thus, local minima will appear as leafs in the tree.

3.1 Unordered Sweeping

The overall structure of our novel algorithm is as follows:

(1) For each vertex in M, test if it is a local minimum. If so, perform a depth-first search, (called a sweep as described in 2) at that local minimum m, i.e., push m to a newly created stack called sweep stack.
(2) Each sweep thus has its own stack and then performs the following:
  a) Pull a vertex v from the sweep stack. (On the first loop this will be the sweep starter m)
  b) If all smaller-valued neighbors of v are labelled "visited by m" push all larger-valued neighbors of v to the sweep stack and label v "visited by m", else label it "boundary accessed from m".
  c) If the sweep stack is not empty loop back to 2a.
  d) Find the minimal-valued vertex s_m that is labelled "boundary of m". s_m is the saddle for m in the join tree.
  e) After all smaller-valued neighbors of s_m have been labeled visited (by this or any other sweep), start a new sweep at s_m.
(3) Once a sweep finds no vertices labelled as boundary, the algorithm is done and thus the join tree has been constructed.

This structure is very similar to the task-parallel approach of Gueunet *et al.* [14]. By sweeping along ascending paths in the data, saddles are identified for each sweep starter independently. However, instead of progressing each sweep in a sorted order, our sweeps can follow multiple ascending paths in any order and in parallel.

3.2 Saddle Classification

The above algorithm depends on some observations for correctness. At its core the algorithm finds for each local minimum the corresponding saddle in the join tree by a depth-first search. To be more precise, the problem is to find for each local minimum m a vertex s_m, such that there exists an edge $(\tilde{m}, \tilde{s}_m)$ in the join tree. In the following we will show some properties of s_m that will ultimately prove the termination criteria in step 2d to be correct (see Fig. 2).

Let $V_{\min}$ denote the set of all local minima in M. We call a path p monotone ascending w.r.t. the scalar function f if

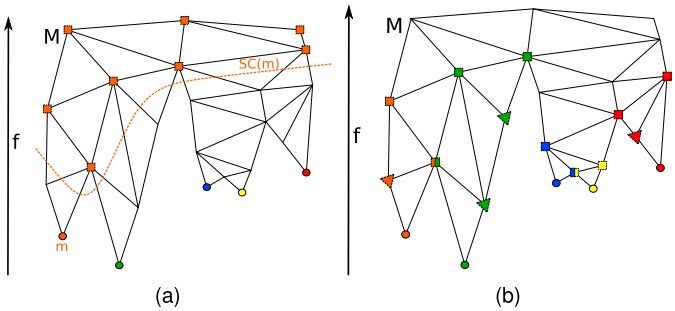

Fig. 2. (a) illustrates the saddle candidate set Sc for the leftmost local minimum with rectangles. Note that saddle candidate sets of local minima overlap and can span large portions of the domain. (b) illustrates the exclusively monotone reachable region set Ex for all local minima with triangles according to color. Note that those sets are mutually disjoint, connected and leave out large portions of the domain. Additionally it shows the boundary sets Bd for all local minima with rectangles according to color. The smallest valued vertex in each such set is a join node in the join tree.

$f(p(a)) \leq f(p(b))$ for all $0 \leq a < b \leq 1$. We denote the set of such paths by P^+. Symmetrically, we define P^- as the set of monotone descending paths. We call a path p monotone if $p \in P^+ \cup P^-$. With this, we want to define the set of all vertices reachable from a local minimum $m \in V_{\min}$ through a monotone ascending path as

$$\mathrm{Up}(m) := \{v \in V(M) : \exists p \in P^+ \text{ s.t. } p(0) = m, p(1) = v\}.$$

Lemma 1. *For a join node $\tilde{v}$ that is an ancestor in the join tree to a leaf $\tilde{m}$ one has $v \in \mathrm{Up}(m)$.*

Proof. For the proof, it is sufficient to show that for each parent $\tilde{u}$ and child $\tilde{v}$ in the join tree there exists a path $p \in P^+$ from u to v. However, this is already shown in the literature (Lemma 9 in [19]). The claim then follows through a transitivity argument. □

Now consider the set of *saddle candidates* of a local minimum $m \in V_{\min}$ as the set

$$\mathrm{Sc}(m) := \bigcup_{\substack{n \in V_{\min} \\ n \neq m}} (\mathrm{Up}(m) \cap \mathrm{Up}(n)).$$

In other words, a saddle candidate for a local minimum m is a vertex that is reachable by a monotone path from m and another local minimum in M. The motivation behind the chosen name *saddle candidates* becomes clear when considering the following lemma:

Lemma 2. *s_m is a saddle candidate for m, i.e., $s_m \in \mathrm{Sc}(m)$.*

Proof. From the definition of s_m follows $s_m \in \mathrm{Up}(m)$ due to Lemma 1. Furthermore, $\tilde{s}_m$ has to be connected to at least one other leaf $\tilde{n}$. Since a leaf in the join tree corresponds to a local minimum $n \in V_{\min}$ one has $s_m \in \mathrm{Up}(n)$ again due to Lemma 1 and therefore $s_m \in \mathrm{Sc}(m)$. □

This criterion is behind most monotone path-based merge tree construction algorithms [15], [17], [19]. However, since the sets of saddle candidates for different local minima overlap, searching their entirety often results in double work. This can be avoided by sorted progression, because of the following additional property:

Lemma 3. *The vertex s_m is the smallest valued saddle candidate of m. That means $s_m = \arg\min_{v \in \mathrm{Sc}(m)} f(v)$.*

Proof. We need to show that every saddle candidate's function value provides an upper bound for $f(s_m)$. Then, the claim follows from Lemma 2. Let $v \in \mathrm{Sc}(m)$ be an arbitrary saddle candidate of m. Then, there exist monotone ascending paths to v from m and at least one additional local minimum n. The entirety of these paths is in $f^{-1}_{-\infty}(f(v))_v$. This especially means that m and n are connected in $f^{-1}_{-\infty}(f(v))_v$ and therefore $f^{-1}_{-\infty}(f(v))_m = f^{-1}_{-\infty}(f(v))_n$. However, the initial connected components of m and n were disjoint i.e., $f^{-1}_{-\infty}(F)_m \cap f^{-1}_{-\infty}(F)_n = \emptyset$, where $F := \max\{f(m), f(n)\}$. Since both of these components are contained in $f^{-1}_{-\infty}(f(v))_v$, they must have joined by then; leaving the estimate $F < f(s_m) \leq f(v)$. □

Finally, we concluded the classification of the join node for a local minimum that we mentioned in Section 2.2: The smallest valued vertex that is reachable through monotone paths from m and at least one local minimum in M other than m, corresponds to the adjacent inner node for m in the join tree.

This observation summarizes why following monotone paths along the smallest-valued reachable vertex, until a vertex with an unvisited, smaller-valued neighbor is found, is sufficient to identify this vertex as the join node (as done in [14]). It is also behind the global monotone path-based approaches categorized in the second family in Section 2.2.

However, an additional, novel observation allows us to search for saddle candidates without the need for ordered progression and without tracing the entirety of $\mathrm{Sc}(m)$ for all minima. We define

$$\mathrm{Ex}(m) := \mathrm{Up}(m) \setminus \mathrm{Sc}(m),$$

for $m \in V_{\min}$ as the set of vertices that are exclusively reachable through a monotone path from m. We write $v_1 \leftrightarrow v_2$ for vertices $v_1, v_2 \in M$, if there exists a 1-simplex in M being the convex hull of these vertices. In this case we also call v_1 and v_2 *adjacent* to each other in M. With this, let furthermore

$$\mathrm{Bd}(m) := \{v \in M \setminus \mathrm{Ex}(m) : \exists v' \in \mathrm{Ex}(m) : v' \leftrightarrow v \text{ in } M\},$$

be the set of vertices forming a boundary around $\mathrm{Ex}(m)$. Regarding this set, consider the following properties.

Lemma 4. *The set $\mathrm{Bd}(m)$ is a subset of $\mathrm{Sc}(m)$.*

Proof. Let $v \in \mathrm{Bd}(m)$. By definition $v \notin \mathrm{Ex}(m)$ which is only possible if either $v \notin \mathrm{Up}(m)$ or if $v \in \mathrm{Sc}(m)$. Thus, we have to rule out the first case by proving $v \in \mathrm{Up}(m)$. Again due to the definition of $\mathrm{Bd}(m)$, there exists $v' \in \mathrm{Ex}(m)$ such that $v \leftrightarrow v'$. If $f(v') < f(v)$, the vertex v is reachable from m by a monotone ascending path through v'; therefore, $v \in \mathrm{Up}(m)$. The remaining case can be ruled out by contradiction. Assume that $v \notin \mathrm{Up}(m)$ and $f(v') \geq f(v)$. Note that then, there has to exist at least one local minimum $n \in V_{\min}, n \neq m$, such that $v \in \mathrm{Up}(n)$. This becomes clear when considering the following construction: by successively choosing adjacent vertices in M with decreasing function values, one eventually ends up in such a local minimum n. By traversing the involved 1-simplices in reverse order, one thus obtains a monotone

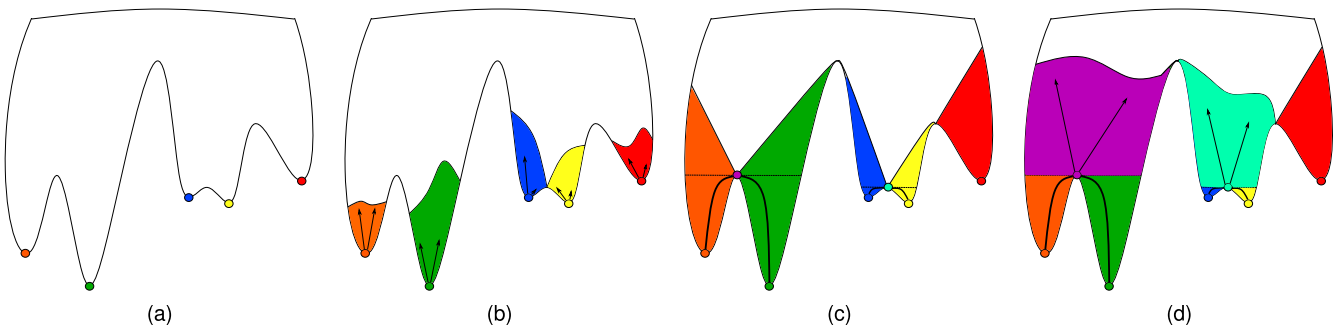

Fig. 3. Exemplary join tree computation on the height function on a manifold, deliberately made comparable to an example in [14]. In (a) local minima and thus join tree leaves are found according to Section 4.1. In (b) independent sweeps grow a region around each local minimum following arbitrary monotone paths in parallel. In (c) sweeps terminate at non-exclusively monotone reachable vertices, namely boundary sets according to Section 4.2. The smallest valued boundary vertices are identified and prepared for their own sweep according to Section 4.3. Additionally, according to Section 4.6, swept vertices are split at the saddle value to retrieve the augmentation. In (d) prepared saddles continue their own sweeps in the same manner, constructing the entire join tree.

ascending path from n to v, proving $v \in \mathrm{Up}(n)$. Since v' was chosen adjacent to v and has a larger function value by assumption, v' is reachable by a monotone path through v from n, thus $v' \in \mathrm{Up}(n)$. This however contradicts $v' \in \mathrm{Ex}(m)$. With this contradiction we prove $f(v') < f(v)$, thus $v \in \mathrm{Up}(m)$ and finally $v \in \mathrm{Sc}(m)$. □

Lemma 5. *s_m is in the boundary set of m, i.e., $s_m \in \mathrm{Bd}(m)$.*

Proof. Because of Lemma 2, s_m is reachable from m through at least one monotone ascending path. Let p denote one such path. Since $m \in \mathrm{Ex}(m)$ and $s_m \notin \mathrm{Ex}(m)$, there has to exist a vertex $v \notin \mathrm{Ex}(m)$ on p that is adjacent in M to a vertex in $\mathrm{Ex}(m)$. Thus, $v \in \mathrm{Bd}(m)$ and therefore due to Lemma 4 $v \in \mathrm{Sc}(m)$. Because p is monotone ascending, the estimate $f(v) <= f(s_m)$ holds true and due to Lemma 3 we obtain $s_m = v \in \mathrm{Bd}(m)$. □

Lemmas 4 and 5 together directly allow for a stronger variant of Lemma 3:

Lemma 6. *The vertex s_m is the smallest valued vertex in the boundary set of m. That means $s_m = \arg\min_{v \in \mathrm{Bd}(m)} f(v)$.*

With this novel observation, we can locally restrict our sweeps with $\mathrm{Bd}(m)$. This is why the algorithm in Section 3.1 is correct, as from these observations our novel algorithm directly follows: For each local minimum m, a sweep grows exclusively monotone reachable regions from m. Those regions are subsets of $\mathrm{Ex}(m)$ and can therefore not contain s_m. This growth terminates at the boundary $\mathrm{Bd}(m)$, which must contain s_m. We identify the actual saddle as the smallest vertex on that boundary. After this, contraction of $\mathrm{Ex}(m)$ onto the saddle is simulated. This results in the saddle becoming a local minimum and an identical sweep can be started from there. This is repeated until the join tree is constructed.

4 Parallel Implementation

In this section, we present our novel algorithm in detail, focusing on involved methods and data structures. We first concentrate on the fundamental algorithm, as if running on a single, shared-memory system. The necessary adaption to a distributed implementation is elaborated in Section 5.

The described algorithm is formulated for a task-parallel setting. In this setting, light-weight tasks are defined by the algorithm, each of which performs a set of instructions. A runtime engine assigns tasks to and suspends them from computational resources, according to dependencies to other tasks and external latencies like memory and network communication. This allows for a dependency driven, parallel execution of tasks. In the presented algorithm, every call to START_SWEEP (see Algorithm 1) creates a new, independent task for execution.

The presented approach calls for a number of data structures. A Union-Find data structure UF is used to track visited labels of vertices and allow for fast simulated contraction to saddles. It is thus updated and queried by the sweeps. Note that we do not rely on the connected component algorithm that is also often called *Union-Find*. A boundary data structure BD represents boundary sets, to efficiently determine the smallest-valued vertex and thus saddle for each sweep. An augmentation data structure AU holds information about the arc each vertex is augmented to.

4.1 Local Minimum Search

Iterating over the complete dataset, a function called IS_MINIMUM is evaluated for each vertex. The function returns true, iff no neighbors of the vertex have smaller scalar values than itself. If so, a sweep is started at the local minimum m by START_SWEEP, this work is encapsulated in a new task and thus runs in parallel to all other sweeps. Compare to Fig. 3a.

4.2 Exclusively Monotone Reachable Region Growth

In START_SWEEP(m), a modified depth-first search is performed, successively evaluating CAN_SWEEP(v) for all vertices that are in $\mathrm{Ex}(m)$ or $\mathrm{Bd}(m)$. CAN_SWEEP(v) performs a find operation for each smaller-valued neighbor of v in a Union-Find data structure and returns true if all return m. Each vertex v that passes this test is visited; thus, all its neighbors are added to a search stack and a union is performed on v and m in the Union-Find structure, see Fig. 3b. Vertices not passing this test are tracked in a boundary data structure and removed if they are visited later on. This representation of the extending boundary will be identical to $\mathrm{Bd}(m)$ once the sweep stack is empty, and thus it contains

the join tree saddle s_m for m as the smallest valued element, see Fig. 3c.

4.3 Moving up the Tree

So far we only talked about local minima as sweep starters, which allows us to identify only leaf edges. However, with a little caution, we are able to virtually contract the sub-level set connected component of a saddle to itself, once all incoming edges in the join tree have been identified. This in turn makes the saddle a local minimum and we can apply the algorithm and observations from Section 3, see Fig. 3d. To prepare the saddle s to start its own sweep (and simulate contraction), a union operation on m and s is performed in the Union-Find structure. To avoid the sweep of s having to revisit Ex sets of its join tree children, the boundary of such a sweep is initialized in UNITE_CHILD_BD to contain the union of the boundaries of their children in the tree. Additionally, pairwise intersections of child boundaries might contain vertices, that are exclusively, monotonely reachable by a saddle, once the child minima are pruned. Thus, pairwise child boundary intersections have to be added to the initial sweep stack of s in INIT_STACK. Once all smaller-valued neighbors of s point to s in the Union-Find data structure, the sweep for s can commence. This sweep will calculate boundary unions and intersections first and then run on its initialized stack in an identical procedure to the local minimum sweeps.

4.4 Union-Find Data Structure

To keep track of vertices that have been visited by sweeps, we employ a data structure that is capable of union and find operations, a *Disjoint-Set* or *Union-Find* data structure UF. Find operations are used in CAN_SWEEP and union operations are performed, whenever a sweep visits a vertex or finds a saddle. We chose UF to be implemented as a fixed size array, holding one vertex index to point to, for each vertex in the data. This allows unrestricted parallel access to unrelated parts of the structure. Some care has to be taken when two smaller-valued neighbors of a vertex v are visited. If both subsequent considerations of v do not see the changes of each others visiting step, growth might terminate too early. However, no mutex and ordering is needed here and a simple memory fence, guaranteeing the visibility of previous changes, is sufficient.

4.5 Boundary Data Structure

A boundary data structure BD is needed to track all vertices that have been considered by a sweep, but could not be visited due to smaller-valued, unvisited neighbors. The data structure must support insertion and deletion of vertex indices, because critical points within Ex sets may not be visited on the first consideration, but later on. Additionally, a fast min-search operation on this data structure is needed, to identify the smallest saddle candidate on the final boundary. Also, union and intersection operations will be performed at every saddle between all child boundary structures. For implementation of this structure a set was chosen, as it allows logarithmic min-search, insertion and deletion, and a union and intersection in O(n*log(m)) with n the size of the smaller and m the size of the larger set. Note that the parallelization of a data structure with these requirements is not trivial, which limits the in-arc parallel abilities of our algorithm. However, this restriction does not transfer to distribution as discussed below and there is no need for any kind of synchronization regarding this data structure between different sweeps.

Algorithm 1. START_SWEEP

```
Input: Local minimum or contracted saddle m
  Initialization :
  UF[m] = m
  Bd(m) = UNITE_CHILD_Bd(m)
  AU(m) = UNITE_CHILD_AU(m)
  Initial_Stack = INIT_STACK(m)
  while !Stack.empty() do
    v = Stack.pop()
    if (CAN_SWEEP(v)) then
      UF[v] = m
      Bd(m).erase(v)
      AU(m).insert(v)
      Stack.insert(NEIGHBORS(v))
    else
      Bd(m).insert(v)
    end if
  end while
  s = Bd(m).MIN()
  if (s == null) then
    GLOBAL_TERMINATE()
  end if
  UF[m] = s
  REGISTER_CHILD_Bd(s, Bd(m))
  REGISTER_CHILD_AU(s, AU(m).SPLIT(s))
  if (SADDLE_READY(s) then
    START_SWEEP(s)
  end if
  return P
```

4.6 Augmentation Handling and Data Structure

To understand the augmentation data structure AU, consider the following observation: The augmentation of an arc from a local minimum m to a join tree saddle s is the set of all vertices in $\mathrm{Ex}(m)$ with smaller values than s. To again avoid double work for visited vertices, a dedicated data structure holding a set of all vertices augmented to an arc started at a local minimum m is maintained. A vertex v is added to the augmentation of m, when v is visited by the sweep of m. When pruning an arc to its saddle s, the augmentation needs to be split into vertices with smaller values than s, that form the final augmentation of m's arc, and vertices with higher values than s, that are then used to initialize the augmentation of s's sweep. This split is illustrated as a dotted line in Fig. 3c resulting in the colored augmentation of Fig. 3d. For this purpose we implemented a concurrent skip-list [35]. This allows for statistically logarithmic insertion and constant time splitting of the list at a given value, and again a union in $\mathcal{O}(n \cdot \log(m))$. Note that tracking the augmentation is optional, as our algorithm at no point requires augmentation information for correctness. Results in section 6 include augmentation tracking. Omitting this data structure and the methods described above resulted in

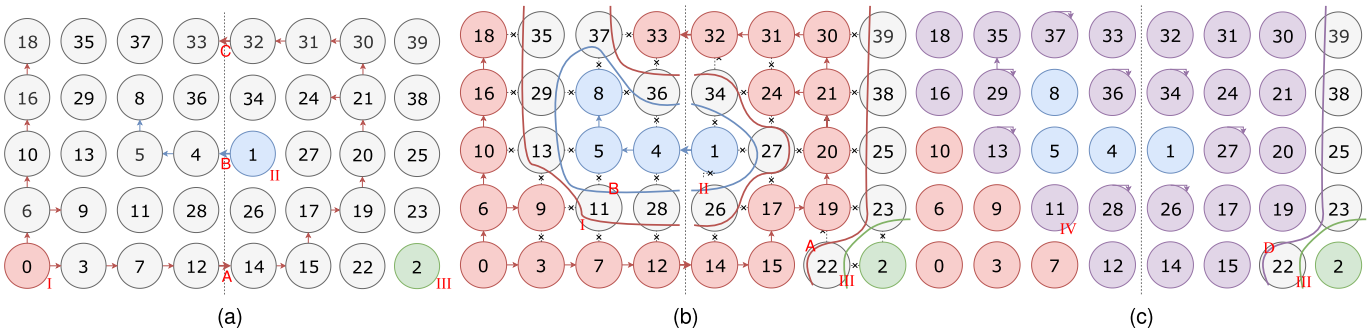

Fig. 4. Circles represent nodes in a regular 2D-Grid with their scalar function value written inside. Starting in the local minima (colored red, blue and green) a task (I, II, III) sweeps along ascending paths, visiting vertices that are exclusively monotone reachable by the starting local minimum; thus, tracing the interior of the corresponding Ex set. According to Section 5.2 when such a sweep encounters a data boundary between two nodes (dotted line) a network message is sent to start a new task on the neighboring node continuing the sweep (A, B, C). Information about visited and boundary labels is stored separately on each node for the local vertices. Once the sweep stack of task is empty and all remotely started sweeps returned, the smallest boundary vertex on the local node is returned by the sweep. In (b) each task (I, II, III, A, B, C) is positioned at the smallest valued vertex of the local boundary set representation of the starting local minimum. In this case A would return the node valued 22, which would be compared to 11, which would be identified as the saddle for 0. In (c) , after saddle identification, the saddle is communicated to all involved nodes. A new task is started for each involved node (IV, D). Here augmentations are cut at the saddle value, boundary intersections are pushed on the sweep stack and a new sweep is started. The task IV encounters an empty boundary, D returns vertex values 22 which will be handled as the saddle in the same way 11 was. Note that trunk skipping would skip all work shown in (c).

almost identical runtimes, which can be explained by the I/O and message heavy nature of the algorithm and the latency hiding capabilities of task parallel programming. Note however, that depending on the data, resulting output sizes of augmented merge trees may be orders of magnitude larger than un-augmented output sizes.

4.7 Tree Representation

The resulting tree can be extracted from these data structures in multiple ways. The most simple representation of the augmented join tree would be to start an additional parallel scan over the data in UPDATE_UF and let all vertices in UF point to the local minimum they are augmented to in AU. After this step, UF would be a flat representation of the join tree, with all non-node vertices pointing to the lower vertex of their augmented edge.

5 Distributed Task-Parallel Merge Trees

With the distributed setting, we refer to a parallel hardware environment in which not all threads access a shared memory. Instead, multiple computational nodes host a set of threads each, and accessing memory of a remote node requires explicit communication. In our task-parallel setting [36], no direct memory access or message passing across nodes is possible, but rather all information has to be shared as method input or output.

The input data to our algorithm is assumed to be partitioned onto the nodes, so that each node is responsible for a subset of M. We require one layer of ghost cells [37] that is assumed in the input data but could be constructed with one message between all nodes holding adjacent vertices. Within this setting, the neighbors and value of a vertex v are available only on nodes that are responsible for v or any of its neighbors. In the following, it is assumed that the partition of M onto the nodes is along axis aligned boxes of preferably similar volume. While this is a typical standard setting, as it minimizes boundary sizes between nodes, it is not required by our algorithm.

Above, we described a novel algorithm to compute augmented join trees. However, we assumed shared-memory so far. Thus, the algorithm has similar structure to [14], with the option to parallelize the work on a single join tree edge to some degree, but paying the price of management of the boundary and augmentation data structures for it. While for augmentations a parallel and efficient data structure could be found, boundary management will harm our sequential runtime.

Since most complex and large data sets arising from simulations and measurements tend to have a large number of join tree arcs with rather small augmentations each, the option of parallelizing individual arc constructions is hardly ever worth this price. However, as we will discuss in this section, the ability to split work on single arcs allows our algorithm to scale to distributed systems without shared memory quite well. As we still utilize task-parallel, non-global methodology, global communication can be kept to a minimum. In the following, data structures and methods will be revisited and their adaption to a distributed setting is explained, see Fig. 4.

5.1 Local Minimum Search

The function IS_MINIMUM does not require an adaption to the distributed setting and thus the identification of local minima, which start sweeps can be done in parallel for each node and within each node, scaling embarrassingly parallely.

5.2 Exclusively Monotone Reachable Region Growth

The sweeps themselves are running just like in the shared-memory scenario with one major adaption. If the node responsible for a vertex v from the sweep-stack is not the node on which the current task is running, the responsible node is informed about a sweep from m having reached v; on the responsible node, a new sweep-task is started for m, with the initial stack containing v. Note that this way any number of nodes can get involved in a single sweep. If such

continuing sweeps encounter a data boundary to another node again, they recursively start a continuing sweep at the responsible node. This process might spread back onto nodes that already work on a sweep for a given local minimum. In this case, the respective vertex is added to the back of the sweep stack of an already running sweep and the newly started sweep instantly returns. Each continuing sweep returns once its stack is empty and all recursively started sweeps return.

For each sweep, only one instance of $\mathrm{Bd}(m)$ and $AU(m)$ is maintained, per node. Since insert and delete operations for given vertices v will always arise only on the node responsible for v, those data structures are disjoint and do not need any communication during the sweep. Once the stack of a continuing sweep is empty, the current minimum of $\mathrm{Bd}(m)$ is returned to the node that initiated the continuing sweep. Since even after that a second continuing sweep on the same node can be started, only the most recent such return value per node is maintained in a dictionary on the node responsible for the local minimum that started the sweep.

With this, once the stack of the original sweep is empty and all continuing sweeps have returned, the overall minimum s in the dictionary is the overall minimum of all involved $\mathrm{Bd}(m)$ structures and thus the saddle for m. s is shared with all nodes involved in the sweep, so that they can initialize their local portions of $\mathrm{Bd}(s)$, $AU(s)$ and initial sweep stack, with the respective elements in $\mathrm{Bd}(m)$ and $AU(m)$. This will allow continuing sweeps of s to run within a correct setting on those nodes. The node responsible for s will check whether all smaller neighbors of s have been visited by (continuing) sweeps of some minima and if so start a sweep at s.

5.3 Distributed Union-Find Data Structure

The structure for UF therefore is the only data structure, that will need to store vertices, for which the storing node is not responsible. For that, the structure is composed of a fixed sized array for all local vertices and a dictionary for all remote vertices. Modifications to the dictionary need to be synchronized. The maintenance of this data structure does not require additional network messages. Union operations that may involve remote vertices only occur when a sweep moves across data chunks, or when a saddle is identified. In both cases the algorithm requires a network message anyway and the involved vertices can be stored to all involved dictionaries. With that, as find operations on a vertex v only traverse ancestors of v in the join tree, find operations will never look up a vertex that is not already in the dictionary.

5.4 Communication and Load Balance

Note that an uneven distribution of local minima (and sweep starting saddles higher up in the tree) among nodes causes little imbalance. Each vertex will be visited by exactly one sweep. BD and AU structures will only contain vertices a node is responsible for. Computational expense of unions and intersections will be proportional to the size of the respective Ex set within the nodes responsibility; this is independent of whether a node is responsible for the local minimum or saddle of the sweep. Communication is limited to directly adjacent nodes. The only exception to this occurs when informing all involved nodes of a found saddle.

In other words, sweeps that span across multiple nodes can grow on each node individually, overall communication being limited to one message per vertex on the boundary between two nodes. There will be latencies introduced due to messaging, for example when waiting for continuing sweeps return values, when the local stack is empty. However, since all sweeps can run in parallel, and the task-parallel paradigm allows for fast context switches, these wait times can —to a large degree— be filled with work on different sweeps.

5.5 Trunk Skipping

However, in practice for typical data sets, a large part of the computations in regions of the tree with great height can be avoided, since the concept of trunk-skipping [14] translates to our distributed solution. We maintain a global counter that is increased for each identified local minimum. It is decreased each time a sweep is finished and does not spawn a new sweep at its saddle immediately. With this counter, the number of globally active sweeps can be tracked. Note that this is the only completely global data structure in our algorithm that serves only as an optimization, without sequential dependencies. Only a single integer is stored, and counter modification tasks can be run with low priority, running only whenever all remaining worker tasks are in a waiting state.

Once all nodes have identified all minima, and the global counter reaches the value one, only one sweep is still growing, and the remaining computation can be trivialized as follows: All remaining saddles that were identified, but still have smaller-valued unvisited neighbors, will be reached by the last remaining sweep in ascending order of their values. Therefore, they are collected on a single node, sorted by value and broadcasted back to all nodes. There, all un-augmented vertices will be augmented to the highest valued saddle in the list that has a smaller value than them with a parallel binary search. While this is a very global and sequential operation, it typically helps avoiding to sweep through large regions of the data and can significantly speed up computation on noisy data sets, or data sets with large homogeneous regions, as illustrated by Guenet *et al.* [14].

5.6 Tree Representation

After construction, the augmented join tree is implicitly represented by all UF data structures on all nodes. However, we additionally collected the un-augmented join tree arcs during construction. Whenever a sweep terminates, the corresponding arc/edge is stored as a pair (m, s) and asynchronously sent to a master node. The resulting messages are combined with the messages required to maintain the trunk skipping counter above. Thus, after construction, the un-augmented join tree is available at the master node, runtimes in Section 6 include this process. Note that if the resulting join tree does not fit within the main memory of a single node, more sophisticated representations will be necessary.

5.7 GPU Adaption

In addition to the task-parallel and distributed setting above, our novel algorithm can also translate to a GPU

TABLE 1
Data Set Overview Including Runtimes on an Ideal Number of Nodes and Dimensionality for all Involved Data Sets

Data set	Dimensionality	Arc count [million]	Runtime [seconds]
Foot	256^3	0.54	1.19
Vertebra	512^3	1.5	2.48
Meteor [38]	300^3	0.038	3.31
Backpack	512^2x373	4.8	6.23
Jet [39]	256^2x512	0.24	4.49
Aneurism	256^3	0.007	0.23
Vertebra 1024^3	1024^3	1.7	10.02
Foot 1024^3	1024^3	2.2	14.64
Miranda	1024^3	3.4	29
Spathorhynchus	1024^2x750	30	117.33

environment. As a proof of concept, we derived a CUDA implementation of the following idea: One thread for each vertex evaluates IS_MINIMUM. Using an atomic counter, all minima are collected in a list. For each element of the list, a megakernel performs the sweeps like above. After a fixed number of CAN_SWEEP calls, one thread for each vertex in the data tests, whether the vertex is adjacent to two vertices in different Ex sets. If so, and the sweeps for both those sets already finished, the tested vertex is on the boundary set Bd of both. This boundary vertex is compared in an atomic minimum operation for both involved sweeps. The resulting minimum for a sweep is the join tree saddle and is therefore added to the mega kernel list as a new (contracted) local minimum. It is typically faster to extract current boundary and sweep states after a certain number of rounds and switch remaining computation and trunk skipping back to CPU.

6 Results

In this section, we present performance and scalability of the resulting implementations. All results emerged from experiments run on the AHRP High Performance Computer 'Elwetritsch' at TU Kaiserslautern. All involved processors were of type Intel XEON SP 6126 (19.25M Cache, 2.6 GHz, 12 CPU cores, 96 GB RAM) with two processors per cluster node. All times were measured for join tree construction, including the augmentation and gathering of resulting arcs at a master node.

Our C++ and CUDA based implementations utilize the HPX framework on top of an OpenMPI parcelport and VTK for data input, using gcc version 9.1, nvcc version 9.2, hpx 1.3.0 and OpenMPI version 4.0. They are made publicly available via codeocean [40].

Experiments are performed on openly accessible, well known data sets to allow better comparability, see Table 1. Most data sets are from the Open SciVis Dataset page (https://klacansky.com/open-scivis-datasets/). Additionally, we use time step 15,422 from simulation yA31 of the SciVis contest asteroid data set [38], the foot ct scan from the TTK example data (https://topology-tool-kit.github.io/downloads.html) and a simulation of a jet fluid stream [39].

Considering Amdahl's law, a deciding factor for the usefulness of a distributed algorithm is the turnaround point at which the total runtime is no longer reduced by additional resources, as communication and synchronization overhead

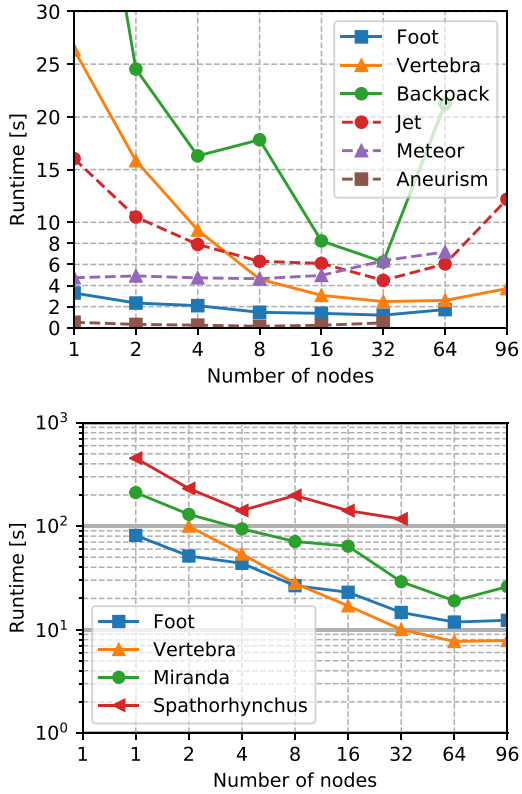

Fig. 5. Strong scaling for different data sets. Runtimes are illustrated for growing number of parallel agents showing feasible scalability up to 96 nodes depending on data size. On the bottom, data sets Vertebra and Foot are resampled to 1024^3 grid size.

excel parallelization benefits. Our results show sufficient strong scalability of up to 96 nodes (2,304 cores), see upper Fig. 5, although ideal runtimes were typically achieved with 32 or 64 nodes. This sound scalability allows us to utilize the HPC hardware to reduce the necessary runtime for augmented merge tree construction on large data sets by an order of magnitude, see lower Fig. 5. Note, that missing entries in the diagram represent configurations, that timed out consistently. These may be due to a sharp increase in parallel overhead, due to network congestion, or similar hardware related thresholds.

In weak scaling experiments problem sizes are grown proportionally to the number of utilized cores to determine the maximal problem sizes the algorithm can feasibly solve, before growing overheads increase runtimes unsustainably. Runtimes of our algorithm stayed within the same order of magnitude when scaling problem sizes up to 2 billion data points, see Fig. 6. As the problem requires/enforces unique vertex values, the number of values a float can represent will become a scalability issue before the runtime of our algorithm does.

The sequential runtime of our novel algorithm on a single, shared memory system is almost on par with the current state-of-the-art TTK implementation [14], see Fig. 7. Additionally, the GPU-hybrid solution demonstrates speed ups between x5 and x20. While the algorithm is targeted towards distributed systems and faster and more readily available solutions for shared-memory systems exist, this comparison shows, that speedups of parallelization do not have to compensate for a subpar sequential runtime.

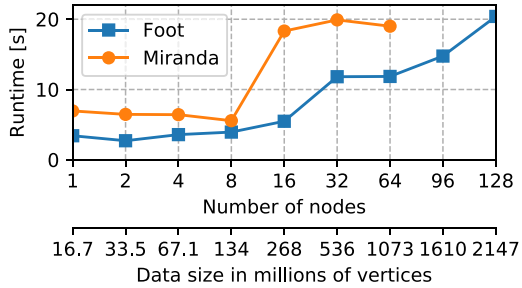

Fig. 6. Weak Scaling demonstrated on the Foot and Miranda data sets. To achieve adjustable data size, the Foot data set has been upsampled and the Miranda data set has been downsampled accordingly.

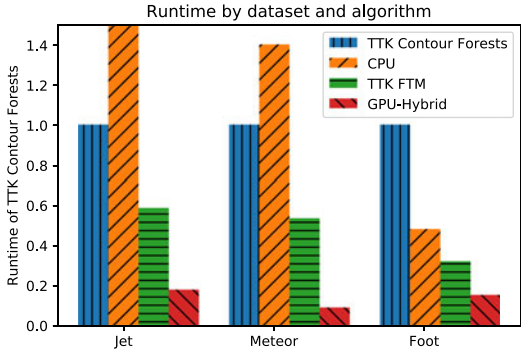

Fig. 7. Runtime comparison between state of the art [14] and [28] and our novel solution along with the GPU-hybrid version. All algorithms constructed the augmented join tree running on a single cluster node.

Comparison to related work in a distributed setting shows, that our novel algorithm outperforms both [20] and [22] significantly (compared to their reported runtimes on the volvis.org vertebra data set, see Fig. 8). Thus, we conclude that the algorithm described here is at least competitive with the state of the art with respect to performance and scalability.

7 Conclusion

We have described a task-parallel algorithm for augmented merge tree construction, that is efficient in shared-memory, scalable in distributed settings, adaptable to a hybrid GPU solution and can compute the entire, un-altered merge tree with or without augmentation.

This was made possible by identifying a local boundary for the recent region growing construction concept, avoiding any need for ordered progression and shared-memory-dependent data structures.

A review of contemporary contour tree construction methods showed, that this unique profile of strengths addresses a gap in the current algorithm landscape, and we were able to demonstrate a perfomance and scalability that outperforms any other means of distributed, large-data merge tree construction known to the authors at the time of writing.

Additionally, as a task-parallel algorithm, combinations of the algorithm with related computations would allow for additional latency hiding; thus, interleaved operations like topological simplification [41], topology based transfer functions for volume rendering and in-situ visualization

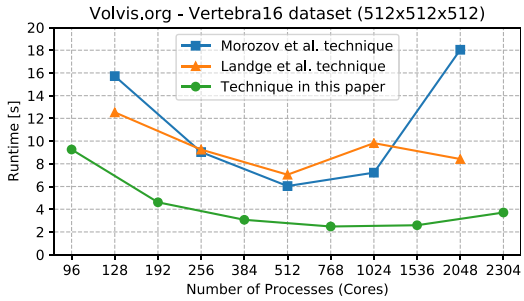

Fig. 8. Runtime comparison between our algorithm and reported runtimes of [20] and [22] on the volvis.org vertebra data set. We therefore present the fastest distributed merge tree construction (augmented and not) known to the authors at the time of writing.

scenarios would present themselves as interesting topics for future work.

Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 398122172. Special thanks to Jan-Tobias Sohns for fruitful discussions and Felix Dietrich for mathematical support.

References

[1] H. Edelsbrunner and J. Harer, *Computational Topology: An Introduction*. Providence, RI, USA: Amer. Math. Soc., 2009.
[2] V. Pascucci, X. Tricoche, H. Hagen, and J. Tierny, *Topological Methods in Data Analysis and Visualization: Theory, Algorithms, and Applications*, 1st ed. Berlin, Germany: Springer, 2011.
[3] C. Heine *et al.*, "A survey of topology-based methods in visualization," *Comput. Graph. Forum*, vol. 35, no. 3, pp. 643–667, Jun. 2016.
[4] P. Rosen *et al.*, "Using contour trees in the analysis and visualization of radio astronomy data cubes," 2017, *arXiv: 1704.04561*.
[5] W. Widanagamaachchi, C. Christensen, V. Pascucci, and P.-T. Bremer, "Interactive exploration of large-scale time-varying data using dynamic tracking graphs," in *Proc. IEEE Symp. Large Data Anal. Visual.*, 2012, pp. 9–17.
[6] H. Saikia, H.-P. Seidel, and T. Weinkauf, "Extended branch decomposition graphs: Structural comparison of scalar data," *Comput. Graph. Forum*, vol. 33, no. 3, pp. 41–50, 2014.
[7] E. Zhang, K. Mischaikow, and G. Turk, "Feature-based surface parameterization and texture mapping," *ACM Trans. Graph.*, vol. 24, no. 1, pp. 1–27, Jan. 2005.
[8] Z. Wood, H. Hoppe, M. Desbrun, and P. Schröder, "Removing excess topology from isosurfaces," *ACM Trans. Graph.*, vol. 23, no. 2, pp. 190–208, Apr. 2004.
[9] J. Tierny, J.-P. Vandeborre, and M. Daoudi, "3D mesh skeleton extraction using topological and geometrical analyses," in *Proc. 14th Pacific Conf. Comput. Graph. Appl.*, 2006, pp. 85–94.
[10] G. Weber, P.-T. Bremer, and V. Pascucci, "Topological landscapes: A terrain metaphor for scientific data," *IEEE Trans. Visual. Comput. Graph.*, vol. 13, no. 6, pp. 1416–1423, Nov. 2007.
[11] G. H. Weber, S. E. Dillard, H. A. Carr, V. Pascucci, and B. Hamann, "Topology-controlled volume rendering," *IEEE Trans. Visual. Comput. Graph.*, vol. 13, no. 2, pp. 330–341, Mar./Apr. 2007.
[12] H. Carr, J. Snoeyink, and M. van de Panne, "Simplifying flexible isosurfaces using local geometric measures," in *Proc. Visual.*, 2004, pp. 497–504.
[13] H. Carr, J. Snoeyink, and U. Axen, "Computing contour trees in all dimensions," *Comput. Geometry*, vol. 24, no. 2, pp. 75–94, 2003.
[14] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny, "Task-based augmented merge trees with Fibonacci heaps," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 8, pp. 1889–1905, Aug. 2019.
[15] H. Carr, G. H. Weber, C. Sewell, O. Rubel, P. Fasel, and J. Ahrens, "Scalable contour tree computation by data parallel peak pruning," *IEEE Trans. Visual. Comput. Graph.*, vol. 27, no. 4, pp. 2437–2454, Apr. 2021.

[16] K. C.-M. V. Pascucci, "Parallel computation of the topology of level sets," *Algorithmica*, vol. 38, no. 1, pp. 249–268, 2003.
[17] B. Raichel and C. Seshadhri, "A mountaintop view requires minimal sorting: A faster contour tree algorithm," 2014, *arXiv:1411.2689* .
[18] S. Maadasamy, H. Doraiswamy, and V. Natarajan, "A hybrid parallel algorithm for computing and tracking level set topology," in *Proc. 19th Int. Conf. High Perform. Comput.*, 2012, pp. 1–10.
[19] Y.-J. Chiang, T. Lenz, X. Lu, and G. Rote, "Simple and optimal output-sensitive construction of contour trees using monotone paths," *Comput. Geometry*, vol. 30, pp. 165–195, 2005.
[20] A. G. Landge *et al.*"In-situ feature extraction of large scale combustion simulations using segmented merge trees," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2014, pp. 1020–1031.
[21] H. Carr, C. Sewell, L.-T. Lo, and J. Ahrens, "Hybrid data-parallel contour tree computation," in *Proc. Conf. Comput. Graph. Vis. Comput.*, 2016, pp. 73–80.
[22] D. Morozov and G. Weber, "Distributed contour trees," in *Topological Methods in Data Analysis and Visualization III*. Berlin, Germany: Springer, 2014, pp. 89–102.
[23] D. Morozov and G. Weber, "Distributed merge trees," *SIGPLAN Notices*, vol. 48, no. 8, pp. 93–102, Feb. 2013.
[24] H. Edelsbrunner and E. P. Mücke, "Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms," *ACM Trans. Graph.*, vol. 9, no. 1, pp. 66–104, Jan. 1990.
[25] T. Banchoff, "Critical points and curvature for embedded polyhedra," *J. Differ. Geometry*, vol. 1, no. 3–4, pp. 245–256, 1967.
[26] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny, "Task-based Augmented Reeb Graphs with Dynamic ST-Trees," in *Proc. Eurograph. Symp. Parallel Graph. Visual.*, Porto, Portugal, Jun. 2019, pp. 27–37.
[27] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny, "Task-based augmented merge trees with fibonacci heaps," in *Proc. IEEE 7th Symp. Large Data Anal. Visual.*, 2017, pp. 6–15.
[28] C. Gueunet, P. Fortin, J. Jomier, and Vijay, "Contour forests: Fast multi-threaded augmented contour trees," in *Proc. IEEE Symp. Large Data Anal. Visualization*, 2016, pp. 85–92.
[29] A. Acharya and V. Natarajan, "A parallel and memory efficient algorithm for constructing the contour tree," in *Proc. IEEE Pacific Visual. Symp.*, 2015, pp. 271–278.
[30] A. Nath, K. Fox, P. K. Agarwal, and K. Munagala, "Massively parallel algorithms for computing tin dems and contour trees for large terrains," in *Proc. 24th ACM SIGSPATIAL Int. Conf. Adv. Geogr. Inf. Syst.*, 2016, pp. 1–10.
[31] M. Hajij and P. Rosen, "An efficient data retrieval parallel reeb graph algorithm," *Alogrithms*, vol. 13, 2020, Art. no. 258.
[32] H. A. Carr, G. H. Weber, C. M. Sewell, and J. P. Ahrens, "Parallel peak pruning for scalable SMP contour tree computation," in *Proc. IEEE Symp. Large Data Anal. Visual.*, 2016, pp. 75–84.
[33] P. Rosen, J. Tu, and L. Piegl, "A hybrid solution to parallel calculation of augmented join trees of scalar fields in any dimension," *Comput.-Aided Des. Appl.*, vol. 15, no. 1, pp. 610–618, 2018.
[34] R. Sarkar, X. Zhu, J. Gao, L. J. Guibas, and J. Mitchell, "Iso-contour queries and gradient descent with guaranteed delivery in sensor networks," in *IEEE INFOCOM 27th Conf. Comput. Commun.*, 2008, pp. 960–967.
[35] W. Pugh, "Concurrent maintenance of skip lists," Dept. Comput. Sci., Inst. Adv. Comput. Stud., Univ. Maryland, College Park, MD, USA, Tech. Rep. *CS-TR-2222.1*, 1989.
[36] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex an advanced parallel execution model for scaling-impaired applications," in *Proc. Int. Conf. Parallel Process. Workshops*, 2009, pp. 394–401.
[37] J. Patchett, B. Nouanesengesy, J. Pouderoux, J. Ahrens, and H. Hagen, "Parallel multi-level ghost cell generation for distributed unstructured grids," in *Proc. IEEE 7th Symp. Large Data Anal. Visual.*, 2017, pp. 84–91.
[38] J. Patchett and G. Gisler, "Deep water impact ensemble data set," U.S. Dept. Energy, Washington, D.C., USA, Tech. Rep.*TR-LA-UR-17–21595*, Apr. 27, 2017. [Online]. Available: https://datascience.dsscale.org/wp-content/uploads/2017/03/DeepWaterImpactEnsembleDataSet.pdf
[39] C. Garth, "Simulation of a jet flow," 2020. [Online]. Available: http://dx.doi.org/10.21227/qjxp-kc31
[40] K. Werner, "Reproducible source code for unordered task-parallel augmented merge tree construction," 2020. [Online]. Available: https://codeocean.com/capsule/0498480/tree/v1
[41] K. Werner and C. Garth, "Alternative parameters for on-the-fly simplification of MergeTrees," in *Proc. Eurograph. Symp. Parallel Graph. Visual.*, 2020, pp. 75–79.

Kilian Werner received the bachelor's and master's degrees in 2016 and 2018, respectively, in computer science from Technische UniversitÃ¤t Kaiserslautern, where he is currently working toward the PhD degree. His research interests include topology-based methods in visualization, large-scale data analysis, and visualization.

Christoph Garth received the PhD degree in computer science from Technische UniversitÃ¤t (TU) Kaiserslautern in 2007. After four years as a postdoctoral researcher with the University of California, Davis, he rejoined TU Kaiserslautern where he is currently a full professor of computer science. His research interests include large-scale data analysis and visualization, *in situ* visualization, topology-based methods in visualization, and interdisciplinary applications of visualization.