Distributed Merge Trees

Dmitriy Morozov
Computational Research Division,
Lawrence Berkeley National Laboratory
One Cyclotron Road, MS 50F-1650,
Berkeley, CA 94720
dmitriy@mrzv.org

Gunther H. Weber
Computational Research Division,
Lawrence Berkeley National Laboratory
One Cyclotron Road, MS 50F-1650,
Berkeley, CA 94720 /
Department of Computer Science,
University of California, Davis
One Shields Avenue, Davis, CA 95616
GHWeber@lbl.gov

Abstract

Improved simulations and sensors are producing datasets whose increasing complexity exhausts our ability to visualize and comprehend them directly. To cope with this problem, we can detect and extract significant features in the data and use them as the basis for subsequent analysis. Topological methods are valuable in this context because they provide robust and general feature definitions.

As the growth of serial computational power has stalled, data analysis is becoming increasingly dependent on massively parallel machines. To satisfy the computational demand created by complex datasets, algorithms need to effectively utilize these computer architectures. The main strength of topological methods, their emphasis on global information, turns into an obstacle during parallelization.

We present two approaches to alleviate this problem. We develop a distributed representation of the merge tree that avoids computing the global tree on a single processor and lets us parallelize subsequent queries. To account for the increasing number of cores per processor, we develop a new data structure that lets us take advantage of multiple shared-memory cores to parallelize the work on a single node. Finally, we present experiments that illustrate the strengths of our approach as well as help identify future challenges.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; F.2.2 [*Analysis of Algorithms and Problem Complexity*]: Nonnumerical Algorithms and Problems.

Keywords topological data analysis, feature extraction, merge tree computation, parallelization, hybrid parallelization approaches

1. Introduction

Science and engineering increasingly rely on simulations to better understand natural phenomena. With the growing computational power available to these simulations, their results become more complex, and analysis becomes a challenge. Simultaneously, the improvements in the precision of physical instruments outpace our ability to process their measurements, creating a similar demand for tools capable of large-scale analysis. Traditional visualization techniques help gather new insights, but today's datasets achieve a level of complexity that hinders their direct understanding.

One way to reduce this complexity is to extract the salient features, subsequently using them for visualization and analysis. In this context, topological methods have proven useful as they supply flexible feature definitions for a range of scientific problems, including characterizing the mixing of fluids [14], analyzing combustion simulations [5, 6, 15, 23], and identifying states and transitions in chemical systems [3]. Often, topological features, such as isosurfaces and extrema, have a ready physical interpretation.

What makes topological analysis powerful is its principled approach. By drawing on algebraic insights, it establishes a connection between stability and persistence of topological features. Accordingly, it allows the user to zero in on what matters: to recognize the significance of the different parts of the input, to highlight its meaningful components, and to distinguish between important features and superficial noise. Feature detection provides a simpler, abstract view of the data.

But the increasing data sizes demand more computational power. While its growth was following Moore's law, serial computation was meeting this demand. Recently this growth has stalled. As a consequence, the need to speed up computation has put more emphasis on the use of parallelism. With modern supercomputers in mind, we aim to take full advantage not only of the large number of computational nodes, but also of the growing number of shared memory cores on each node.

To cope with the proverbial flood of data, we need to be able to run topological algorithms effectively on massively parallel machines. However, the analytical strength of topology is also its computational weakness. By definition, topology relies on global information: that's how it recognizes globally stable features. But global information is an obstacle to parallelization, which thrives on localized computation.

Accordingly, parallel approaches in topology are in their infancy. We are aware of only one result on parallel computation of merge trees [17], the main object of our study. Notable effort has gone into parallelizing Morse–Smale complex computation [13, 21, 22]. In all cases, the chief obstacle is the need to gather global information about the data. We propose to shift the focus from computation of topological descriptors in isolation and to consider their subsequent analysis. Our main contribution is a distributed representation of merge trees that can be quickly post-processed. We believe such an emphasis on analysis would be useful for many topological descriptors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'13, February 23–27, 2013, Shenzhen.
Copyright © 2013 ACM 978-1-4503-1922/13/02. . . $10.00

Merge trees. In this paper, we focus on a 0-dimensional topological invariant, a merge tree. By keeping track of the evolution of components in sublevel sets of a scalar function, merge trees capture their global connectivity. This information, in turn, has been used for computation of contour trees [8] and the analysis of scientific datasets [6].

Because of their connection to persistent homology [10] — the branches of a merge tree are equivalent to a 0-dimensional persistence diagram — merge trees allow the user to examine the amplitude of noise in the data. Furthermore, they let us make minimal changes to the function while eliminating as many (noisy) extrema as possible [1, 2, 12]. Consequently, such a simplified, denoised function can serve as an input to any visualization technique.

Reeb graphs [20] are descriptors closely related to merge trees. Instead of tracking components of sublevel sets, they track isosurfaces, i.e., components of level sets. Reeb graphs illuminate the stability of isosurfaces [4] by quantifying how small a perturbation eliminates any given component. When the domain is simply connected, Reeb graphs become trees, often called contour trees. The algorithm of Carr et al. [8] constructs contour trees from two merge trees: those of the function and of its negation. Flexible isosurfaces [7, 9] use these trees to manipulate individual contours.

In many data analysis applications, it is interesting how derived quantities change as we vary the classification threshold. For example, in combustion simulations one can classify regions as "burning" or "non-burning" by thresholding fuel consumption. Since superlevel sets correspond to segmentation of the domain above a threshold, it is possible to use merge trees to extract information, such as burning region size, for all possible thresholds [6]. Similarly, Mascarenhas et al. [15] use the merge tree to compute statistics about feature thickness depending on various scalar dissipation rate thresholds. Our own work is motivated by finding halos in astrophysical data. Here the goal is to identify clusters of high density without *a priori* knowledge of a suitable threshold. The merge tree provides a compact representation for all possible classifications, which not only lets us detect thresholds interesting for analysis, but also supports efficient computation of derived quantities such as the distribution of mass in a cluster. The salient point in all cases is that a merge tree is not an end goal in and of itself. Instead, it is a means to organize data for querying and post-processing.

Contribution. We attack the problem of computing merge trees on modern parallel computers on two fronts: (1) we develop a distributed representation of merge trees that not only improves their parallel computation, but also simplifies their subsequent analysis; (2) we develop a new algorithm for merging two trees. Specifically, after reviewing background and prior work in the next section, we present our contributions as follows:

- As a baseline, we assemble a global tree by performing in parallel a binary reduction. To fit the result on a single processor, we simplify the tree. In Section 3, we show that merging and simplification steps can be interleaved during the computation.
- Section 4 describes a distributed merge tree representation that not only speeds up the computation, but lets us parallelize subsequent queries. This representation is particularly interesting in the context of the so-called in-situ computation, where the input is partitioned among multiple processors by its producer, and any analysis must respect this partition, minimizing the movement of the data.
- Section 5 presents a new data structure for storing the merge tree. It not only speeds up the serial combination of two trees, a basic step both in our work and in prior work [17], but also lets this operation scale on multiple available shared-memory cores.

Section 6 illustrates our claims with experimental results.

2. Background

We begin by reviewing the relevant background, including related work.

Spaces, functions, and merge trees. In this paper we are concerned with triangulated domains that support continuous real-valued functions. Recall that a k-simplex is the convex hull of $k+1$ vertices. Its face is the convex hull of any subset of these vertices. A simplicial complex is a collection of simplices closed under taking faces. We assume that our input is a simplicial complex K together with a scalar function $\hat{f} : \mathrm{Vrt} K \to \mathbb{R}$ on its vertices. We extend this function to the underlying space $|K|$ of the simplicial complex, i.e., the union of its simplices, by linear interpolation and obtain $f : |K| \to \mathbb{R}$.

Merge trees capture the connectivity of the sublevel sets $K_a = f^{-1}(-\infty, a]$ of a real-valued function. (Symmetrically, they can keep track of the superlevel sets of a function, but it is convenient to think of this case as working with the negation, $-f$.) For a formal definition, we say that two points x and y in the domain $|K|$ of f are equivalent, $x \sim y$, if they belong to the same level set, $f(x) = f(y)$, and they belong to the same component of the sublevel set $f^{-1}(-\infty, f(x)]$. A *quotient space*, $|K|/\sim$, glues together points in $|K|$ that are equivalent under the relation $\sim$, i.e., it is the set of equivalence classes of $|K|$ with respect to $\sim$, equipped with the topology where open sets are those sets of equivalence classes whose unions are open sets in $|K|$. This quotient space is called a *merge tree*.

The following procedural description unravels the definition. As we sweep the function value from $-\infty$ to ∞, we create a new branch for each minimum. As its component in the sublevel set grows, we extend the branch until two of them merge (at a saddle). If the domain is connected, all the branches merge together by the time we reach the global maximum, which becomes the root of the tree. We can, thus, distinguish between three types of nodes in a merge tree: leaves represent the minima of the function; internal nodes correspond to the merge saddles; and the root of the tree captures the global maximum.

There is the fourth type of node, a *degree-two node*. These nodes are topologically trivial — they carry no information since the sublevel sets do not change as we pass the respective function value — but they are by far the most numerous. We prune such nodes from the tree, recording them in lists attached to the first non-trivial node below them.

For a linearly interpolated function only the 1-skeleton of the domain (vertices and edges) matters to the connectivity of the sublevel sets and, therefore, to the computation of the merge tree. We assume there are n vertices and m edges in the domain. The input function will be fixed throughout the paper. However, we will restrict it to different domains. We denote the merge tree of the function restricted to a set U by T_U.

Branch decomposition. To get a better handle on its structure, it is convenient to decompose a merge tree. At every internal node, we separate all but the deepest subtree. Here we mean the deepest in terms of the minimum function value. The result is a decomposition of the tree into half-open paths that start at an extremum and end just before a saddle, plus one closed path that starts at the global minimum and ends at the global maximum; see Figure 1. The intervals of function values covered by these paths nest into each other; they are precisely the persistence barcode [11]. Pascucci et al. [18] use a similar decomposition for the contour trees.

Computation. The standard approach to computing merge trees [8] is closely related to Kruskal's algorithm for finding minimum spanning trees. It maintains a disjoint-set data structure, initializing each vertex as its own set. Processing the vertices in order of increasing

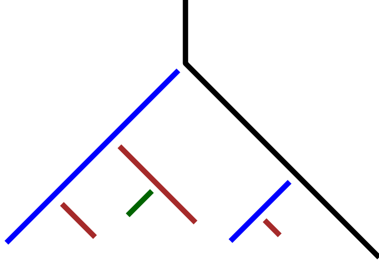


Figure 1: Branch decomposition.

function value, one finds the sets containing each lower neighbor v of the current vertex u. If the sets containing u and v are different, the highest-valued representative in the set of v is linked to u in the merge tree; the two sets are united. The procedure is dominated either by the initial sorting of the vertices or by the union–find algorithm; thus, it runs in time $O(n \log n + m\alpha(n))$[1], where $\alpha(n)$ is the inverse Ackermann function.

To avoid storing the entire input in memory, Bremer et al. [6] build merge trees by incrementally merging together paths. Processing the edges in arbitrary order, the authors merge together (in sorted order) the two paths in the tree that start from the vertices of an edge and go up to the root. It is not difficult to verify that the resulting tree is indeed the merge tree. (Here and throughout the paper, we abuse the terminology and refer to merge *trees* of disconnected domains, which are, in fact, forests.) The price for memory savings is speed: this construction technique is slower than the Kruskal's-like algorithm above.

Domain decomposition. Given multiple processors, the only existing technique [17] for parallel computation of merge trees follows the standard reduction strategy. The domain is covered by multiple sets; individual processors find merge trees of the function restricted to each of these sets. The resulting small trees are merged together in pairs — producing merge trees of the function restricted to larger and larger subsets of the domain — until all the covering sets are united. When this happens, we have the sought-after full merge tree.

It is convenient to think of the initial cover as the finest level in a hierarchical decomposition of the domain. The hierarchy follows in reverse the merging order of the sets; see Figure 2 for a two-dimensional example. The prototypical three-dimensional example of this construction is an octree subdivision of a cube. Because we are merging sets in pairs, we halve their number at each iteration and move higher up in the hierarchy: at level $3i$, we have 8^i smaller cubes. In this case, the intersection between two merging cubes is always a subset of their two-dimensional boundary. (In general, this need not be the case: for instance, if at the finest level the cubes were thickened by a layer of ghost cells.)

In the remainder of the paper we assume, for simplicity, that we are working with a hierarchy resulting from an octree subdivision of a cubical domain. In fact, we rely on no assumptions on the topology of the covering sets. This observation is irrelevant for cubical domains since the merged cubes remain contractible. However, it becomes beneficial on tori (periodic domains) and, especially, on adaptive mesh refinement (AMR) grids. In the latter case, higher-resolution grids refine subsets of the data at the coarser levels. This refinement, effectively, punches holes in coarser blocks

[1] In our implementation of union–find algorithm, we only use the path compression optimization, without the union-by-rank. Consequently, the running time is $O(m \log n)$.

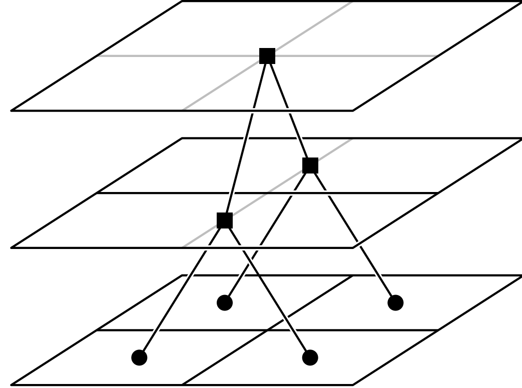


Figure 2: Hierarchy of merges resulting from a quadtree decomposition of the domain. The merges, represented by squares, follow in reverse the splitting of the sets in the quadtree.

leaving us no control over the topology of their uncovered portions. Our results in the next two sections apply directly in these settings.

Merging. Pascucci and Cole-McLaughlin [17] follow the decomposition strategy outlined above. They merge the trees by repeating the Kruskal's algorithm with the union of the trees as the domain. The simplicity of this procedure is appealing: if you can compute a merge tree, you can merge two trees. But the entire approach is limited. As we merge more and more sets, fewer processors remain in use while the trees grow larger. As a result, the binary reduction underlying the merge is top-heavy, making this algorithm effective only for a small number of processors (relative to the size of the input).

The algorithm of Bremer et al. [6] can also be used for merging trees — after all, we just need the merge tree of the function on the union of the two trees. Although slower than [17], their technique serves as the foundation for our new merging algorithm in Section 5. Once we augment the underlying tree data structure, we get a merging algorithm that significantly outperforms the alternative.

3. Simplified Global Tree

For modern input sizes, the merge trees are too large to store in memory of a single node. That's the main motivation behind our work. It is also the main obstacle to a direct comparison of our work to prior approaches [6, 17], both of which try to compute the entire merge tree in one place. In this section we state auxiliary results and describe a scheme that serves as a comparison baseline for our main contribution.

The theory of persistent homology [10] suggests a way to limit the growth of the merge trees. Instead of computing the full tree, we could find a simplified tree that keeps only the stable branches. To distinguish between the unstable noise and the stable features, we need an extra parameter ε: any branch we can remove by changing the input by at most ε is noisy; every remaining branch is stable. When the threshold ε is sufficiently high, the simplified tree becomes small enough to fit in memory.

Simplification. A crucial realization is that we do not need to compute the full merge tree in order to simplify it[2]. Simplification and computation steps can be interleaved in the course of the reduction. Once we compute the merge tree for one of the regions

[2] The idea of simplification is very natural in computational topology and has a long history. Gyulassy et al. [13] independently suggested to use it in the context of parallel computation of Morse–Smale complexes.

in the decomposed domain, we can remove all the branches of length at most ε that are completely internal to the domain, i.e., the ones without boundary nodes. The following theorem justifies this procedure.

THEOREM 3.1. Given a function f, let T_U and $\mathrm{T}_{U \cup V}$ be the merge trees of its restriction to sets U and $U \cup V$. If every node in a subtree of T_U lies outside $U \cap V$, then that subtree appears (unaltered) in $\mathrm{T}_{U \cup V}$.

PROOF. We prove the claim by induction on the size of the subtree. Let x be a node in T_U. If x is a leaf (i.e., a subtree of size one), it is a minimum of the function restricted to U. If it lies outside $U \cap V$, it remains a minimum in $U \cup V$. So it remains a leaf in $\mathrm{T}_{U \cup V}$.

Assume the claim is true for all subtrees of size up to k. Let x be the root of a subtree of size $k+1$; let x and all of its descendants be outside $U \cap V$. From the inductive assumption, none of the subtrees of its children change in $\mathrm{T}_{U \cup V}$. Therefore, the only way for the subtree of x to change is if its component of the sublevel set changes when we restrict the function to the union $U \cup V$. But this is impossible: since x and all of its descendants are outside $U \cap V$, its component in the sublevel set of the function restricted to $U \cup V$ lies entirely in $U - (U \cap V)$. □

The final merge tree has a convenient interpretation in light of the work on persistence-sensitive simplification [1, 2, 12]. We can find a function g, close to f in the sense that $\sup_x |f(x) - g(x)| \leq \varepsilon$, such that the simplified tree is really the merge tree of g. This view justifies our strategy: if the input was perturbed up to ε by noise, we can choose any nearby function without sacrificing precision — we might as well choose the function that gives the clearest view of the data.

Pruning. The efficiency of the merging and simplification procedures depends on the sizes of the trees involved. So we try to make them as small as possible. It is easy to prune the internal nodes: all degree-two nodes are redundant. The boundary nodes, on the other hand, present a problem. A degree-two boundary node in the restricted domain can end up a merge node once the two domains are united. The next lemma, however, grants us some freedom by elucidating which of the boundary nodes are definitely superfluous.

LEMMA 3.2. Given a function f, suppose we have merge trees T_U and T_V of the restriction of f to sets U and V. If $x \in U \cap V$ is not a minimum of $f_{|U \cap V}$, then the path from x to the root in T_U is a sub-path of a path from y to the root for some node $y \in U \cap V$. The claim is true for the same pair of nodes in T_V.

PROOF. Let x be a vertex in $U \cap V$ with a lower neighbor $y \in U \cap V$, $f(y) < f(x)$, i.e., x is not a minimum in the intersection. Then, since the function is piecewise-linearly interpolated on the interiors of the edges, y appears in the subtree of x both in T_U and T_V. Therefore, the paths from x to the roots in both trees are subsets of the paths from y. The claim follows. □

What is the implication? A degree-two node x that's not a minimum in the boundary of a cover set contributes no information about the connectivity of the sublevel sets in the union. Some other node y makes redundant x's only contribution, the evidence that the sublevel sets on its path to the root in the two trees are connected. In other words, if a boundary node is not a minimum (within the boundary), we are free to prune it away. On a $1,024^3$ grid we use for our experiments in Section 6, such an aggressive pruning reduces (already pruned) trees by an additional factor between three and four.

Limitation. The simplified-global-tree strategy has an obvious downside: we must know the simplification threshold ε in advance. Merge trees are interesting because they offer a comprehensive view of the data. For example, the distribution of the branches lets us recognize the amplitude of the noise. But now we face a chicken-and-egg problem: we need a merge tree to choose a simplification threshold, and we need a threshold to get a merge tree. In the next section, we show that one need not choose: we can distribute the full merge tree among all the processors.

4. Local–Global Representation

No amount of simplification and pruning can save us from the sizes of the global trees, which grow in lockstep with the input. So instead of trying to assemble the entire output on a single processor, we distribute its representation.

There are many ways to do this. For instance, each processor could store those nodes in the tree that correspond to its local input data. Additionally, each such node could record its parent in the global tree. In principle, such a distribution represents the whole tree. But it is too expensive for analysis. Any non-trivial query requires a traversal of the tree, which, in turn, would generate too much communication: we may have to bounce between processors at every step.

With this consideration, we arrive at the main idea behind our work. By focusing on how a merge tree is used for analysis, we distribute it so as to minimize communication required to answer queries. To this end, each processor stores the detailed connectivity of the branches with its local input data, but keeps only a coarse view of the entire tree.

Sparsification. Instead of treating all the nodes equally, we bestow some with a special status. The chosen nodes are significant: we can answer detailed queries about them. We *sparsify* the tree as much as possible outside of these nodes. To do so, we remove all the branches not reachable from a special node by a monotonically increasing path; see Figure 3. Here we think of a branch as a minimum–saddle segment, open at the saddle. If the saddle loses enough branches to have only one child, i.e., when it becomes a degree-two node, we prune it from the tree. Such sparsification can be performed in linear time via a post-order traversal.

DEFINITION 4.1. A merge tree T_X is *sparsified with respect to* $Y \subset X$ if all its branches $[m, s)$ that do not intersect ascending paths $[u, \mathrm{ROOT}(\mathrm{T}_X)]$ for any $u \in Y$ are removed. We refer to the removed branches as *ghost* and to the remaining branches as *live*.

We note that in this definition the branches are open at the saddle. So if a path from a node in Y goes through a saddle, it does not preclude the removal of the branch that ends in that saddle (as well as the subsequent pruning of the saddle); see Figure 3.

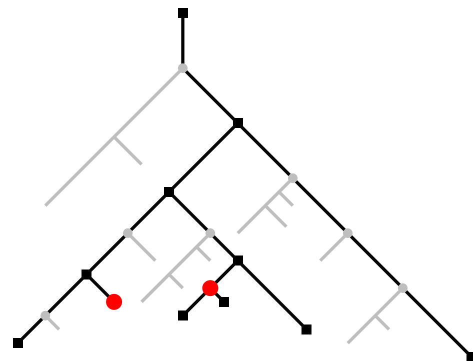


Figure 3: A tree sparsified with respect to the two nodes marked with red disks. Removed branches (and saddles) appear in gray. The remaining nodes are marked with black squares.

Our ultimate goal is to assemble on each processor the global tree sparsified with respect to the processor's local data. But to do so we need intermediate information. As before, our Algorithm 1, SPARSEEXCHANGE, grows the domain of the sparsified tree in iterations: starting from the finest sets in the cover, the trees are merged in pairs. After each iteration, each processor sparsifies its result with respect to the local data and the boundary of the larger domain; see Figure 4. The processors exchange data in pairs: they send to each other their local trees sparsified with respect to the boundary. After i iterations, 2^i nodes have the same outgoing information. In our implementation, they exchange it directly with their counterpart, a node with the same MPI rank except for the inverted i-th bit. Figure 5 illustrates the communication pattern.

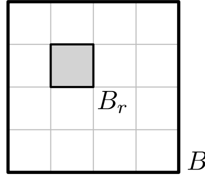


Figure 4: Special nodes in the tree after several iterations of merging are the vertices in the shaded local domain B_r and in the boundary of the larger domain ∂B.

Unlike in the previous section and in the prior work [17], the number of workers does not halve after each iteration. Every processor remains busy computing how its local data fits into the global (sparsified) tree.

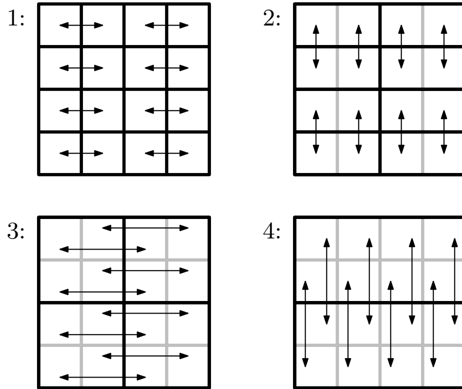


Figure 5: Communication pattern between 16 nodes. The nodes boxed together have the same outgoing information that corresponds to the boundary ∂B of the expanded domain in Figure 4.

The following theorem implies the correctness of our algorithm.

THEOREM 4.2. Ghost branches of T_U with respect to $Y_U \supseteq U \cap V$ remain ghosts in $\mathrm{T}_{U \cup V}$ with respect to $Y_U \cup Y_V$ for any $Y_V \subset V$.

PROOF. Consider the merging of the trees T_U and T_V. It operates on the paths to the root that start from nodes in $U \cap V$. Since Y_U contains $U \cap V$, by definition, the ghost branches in T_U with respect to Y_U are not reachable from these paths. Therefore, they remain unreachable from $Y_U \cup V$ in $\mathrm{T}_{U \cup V}$. □

In other words, once a branch is a ghost with respect to $X \cup \partial U$, it remains a ghost no matter how many other trees it is merged with. (Here we assume, for simplicity, that the sets in the cover intersect only along their boundaries.) Equivalently, sparsification and merge operations can be safely interleaved. So, once the SPARSEEXCHANGE procedure terminates, we have the global tree sparsified with respect to the processor's local data.

Algorithm 1

SPARSEEXCHANGE(f):
(Assumes 2^k compute nodes. BOX(r) refers to the initial set assigned to the node r.)

$r \leftarrow$ MPI-RANK
$B \leftarrow B_r \leftarrow$ BOX(r)
$T \leftarrow$ MERGETREE($f_{|B_r}$)
for i **from** 0 **to** $k-1$ **do**
  $p \leftarrow r$ XOR 2^i # partner rank
  SEND($p, (T, B)$)
  $(T_p, B_p) \leftarrow$ RECEIVE(p)
  $T \leftarrow$ MERGE(T, T_p)
  $B \leftarrow B \cup B_p$
  $T \leftarrow$ SPARSIFY($T, B_r \cup \partial B$)
$T \leftarrow$ SPARSIFY(T, B_r)

Post-processing. Usually merge tree computation serves as a preprocessing step; the real analysis is performed via subsequent queries. In this case, the local–global representation is especially powerful. The initial investment in its computation pays off: the queries can be answered in parallel employing all the available processors.

The simplest example of such post-analysis is the extraction of the persistence diagram, a completely parallel process requiring no communication: each node reports the branches started by the minima in its local domain. It is similarly simple to find which noisy minimum gets remapped into which persistent minimum when we simplify the function by a given threshold: all the information is available locally.

Histograms. A more interesting example is querying the distribution of volumes of sublevel set components containing a given point. Such analysis is useful, for example, in astrophysics when comparing simulated and observed data. Unlike the nodes of the merge tree, the sublevel sets are not localized; their different parts belong to different processors. Fortunately, we can find the sublevel set component of any given local node x without any communication. We identify each component with the point that started it, i.e., the lowest minimum it contains. This identification suggests an alternative view of the local–global representation: for every point in the local domain, it records all the sublevel set components that contain that point.

So for any point x we identify a sequence s_i of saddles that we encounter on the path from x to the root. Additionally, we get the sequence m_i of minima that identify the components of the sublevel sets, so that between $f(s_i)$ and $f(s_{i+1})$ point x belongs to the component identified with m_{i+1}; see Figure 6. Accordingly, the function values on these pairs nest into each other:

$$f(x) \in [f(m_1), f(s_1)] \subseteq [f(m_2), f(s_2)] \subseteq [f(m_3), f(s_3)] \subseteq \ldots$$

Now for any value $a \in [f(s_{k-1}), f(s_k)]$ we broadcast the minimum m_k to the rest of the processors, so that each one of them can find its contribution to the volume of the component of m_k in the sublevel set $f^{-1}(-\infty, a]$. The numbers are summed up by a single reduction. Similarly, for a histogram of volumes with bins defined by intervals I: we broadcast both sequences $f(s_i)$ and m_i; each processor finds its local contribution to each interval in I; the processors sum up all the contributions via a reduction.

Persistent components. Another common query is the location of components that cannot be removed by a small perturbation. Besides the merge tree, we are given two thresholds, m and t. The goal is to report all components of the sublevel set at t that contain a minimum lower than m. Typically, we want to find some additional

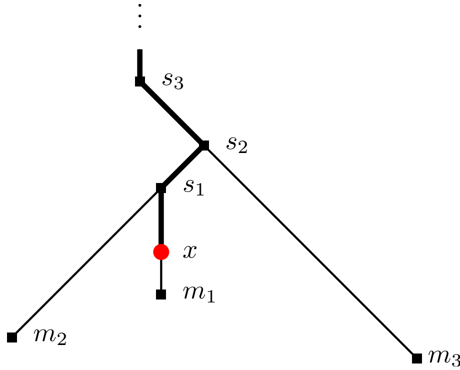


Figure 6: The saddles on the path from x to the root designate its membership in different sublevel set components, represented by the respective minima.

information for such components, for example, their volume or integral. In applications, such queries appear when we are looking for clusters in density functions; in cosmological simulations this problem is called "halo finding."

Here again the local–global representation proves convenient: every processor has complete information about all the components that intersect its local domain. Therefore, in a single traversal, it can identify both the persistent components and how much volume its local domain contributes to each one. Making each processor responsible for the minima local to its input, an exchange is still necessary to sum up the contributions from the different processors, but the required communication is minimal.

5. Skip Trees

Both in Sections 3 and 4 as well as in [17] the main computational step is the merging of two trees. The original approach of Pascucci and Cole-McLaughlin [17] finds the result by repeating the union–find procedure with the union of the two trees as the input. It is unsatisfying for two reasons: (1) the underlying union–find procedure is inherently serial; (2) the algorithm is not sensitive to the output. The former trait is particularly unfortunate for modern computer architectures where the number of cores on an individual node has been steadily rising. Sharing the same memory — the real limiting resource — these processing units could be simultaneously working on the same pair of trees. The goal of this section is to describe an alternative to the algorithm of Pascucci and Cole-McLaughlin that takes advantage of multiple shared-memory cores.

The work of Bremer et al. [6] can be used for merging just as well as for the initial construction. It would mitigate both of the above problems — their algorithm is easy to parallelize using shared memory, and it ignores the parts of the tree that remain unchanged — if only it was competitive with the union–find procedure. Below we adjust our data structure to gain both performance and scaling.

The algorithm of Bremer et al. creates n vertices (on-demand, if necessary) and then processes edges in an arbitrary order. Assuming we already have a merge tree for a domain K, the algorithm finds the merge tree for the domain $K \cup e$, where e is an arbitrary edge $e = (u, v)$ with $f(u) \leq f(v)$. This new edge indicates that all the sublevel set components containing vertex v also contain vertex u. The nodes on the path from v to the root represent all the former components; the nodes on the path from u to the root represent the latter. So all it takes to get the merge tree for $K \cup e$ is to merge these two paths in the sorted order.

The core of this algorithm is the merging of sorted paths. Unfortunately, we have no control over the order in which the edges arrive. Merging together n singleton paths can take quadratic time in the worst case. Although, in practice, the situation is not nearly as bad, profiling shows that the average node access is too high, indicating that the same nodes get traversed many times during the merge process. To alleviate this problem, we turn to skip lists [19] for inspiration and build additional shortcut levels over each parent pointer in the merge tree.

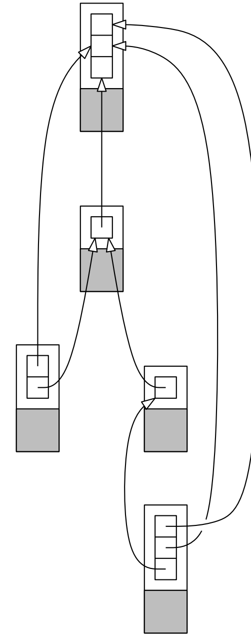


Figure 7: A skip tree with five nodes. The heights of parent stacks are randomized.

Like skip lists, the resulting data structure is randomized. Each node has a parent pointer stack, called a *finger*. The heights of these stacks follow the geometric distribution with probability parameter $1/2$: a node gets a stack of height at least i with probability $(1/2)^{i-1}$. Thus, the expected size of the stack is less than two. Together with the overhead to store the size and the location of the stack, our data structure grows by three words per node compared to the regular merge tree. At each available level i, the node stores the pointer to its first ancestor in the merge tree with the stack of height at least i; see Figure 7. In the resulting tree each path from a leaf to the root is a skip list.

To merge two skip trees, we proceed similarly to Bremer et al. Starting from the shared vertices (alternatively, from the user-provided extra edges), we merge the ascending paths in the two trees. Unlike their algorithm, we can use the additional levels to skip over many nodes at once — the ability responsible for the speedup in Figure 11. Although more involved, it is still simple to simultaneously perform multiple such merges in shared memory. (In fact, our implementation is lock-free.) The scaling results in Table 3 are encouraging.

Remark. We stress that our findings about skip trees apply only to the merging of two trees. When constructing two merge trees from scratch, the union–find-based construction outperforms the skip-tree approach.

6. Experimental Results

We perform experiments on four different datasets: **V** is a rotational angiography scan of a head with an aneurysm (a 512^3 image from http://volvis.org); **A1** and **A2** are two datasets of dark matter particle mass, results of astrophysics simulations ($1,024^3$ and $2,048^3$ fields, respectively); **C** is the rate of methane consumption during a combustion process ($1,024^2 \times 2,048$ flattened two-level AMR grid).

We ran all benchmarks on the "Hopper" system at the National Energy Research Scientific Computing Center (NERSC). "Hopper" is a Cray XE6 with 6,384 nodes connected by a proprietary "Cray Gemini Interconnect". Each node contains 24 cores — provided by two twelve-core AMD 'MagnyCours' 2.1GHz processors — with 32GB (6,000 nodes) or 64GB (384 nodes) of shared memory. Each core has its own L1 and L2 caches, of size 64KB and 512KB respectively, and six cores on the 'MagnyCours' processor share one 6MB L3 cache.

Figure 8 on the following page expresses the relationship between the computation of the local–global representation and of the global simplified trees. The salient point of the four graphs is clear: even when the computation of the most aggressively simplified tree is faster than the local–global representation, it is only slightly so. At the same time the local–global representation contains the same information as the entire unsimplified tree. As the graphs suggest, the less one simplifies the tree, the longer its global computation. We note that the algorithm of [17] computes the global unsimplified tree, which in our context is equivalent to the 0-simplification, meaning it is at least as slow as the slowest running times in those graphs.

To get a better idea about this behavior, we focus on one of the runs. Specifically, we examine the tree growth during the computation of the local–global representation and the global simplified trees for different thresholds using 512 processors on **A1** dataset. Table 1 shows the sizes of the largest tree on any processor during the different iterations of the algorithm, for varying thresholds. The tree growth reflects the memory demands on the processors, our most constraining resource. It also explains the slower computation of the global simplified trees: larger trees take longer to merge. Figure 9 displays the same information graphically. What do we observe? Although the local–global representation starts out larger than even the least aggressive simplification — indeed, it starts with the unsimplified tree — and even though the trees grow during the computation of the local–global representation, they do so significantly slower than during the computation of the global simplified tree. So much so that already before the final sparsification the local–global trees are smaller than the most aggressively simplified, '1e13', trees. The most simplified tree provides little detail about the underlying function. In contrast, the local–global representation gives access to the full merge tree, providing more detail than the most timid simplification, '8e11', which also happens to be several times slower. The situation becomes even more dramatic after the final sparsification: the sparsified local–global tree is not only significantly smaller than the global simplified trees, but it is barely larger than the initial local tree, 44,824 nodes at the end versus 42,594 nodes in the beginning.

Such shrinking of the final trees occurs in all our examples, most vividly illustrated in the bottom graph of Figure 10. This behavior highlights our main interest in the local–global representation. We can make the output of our algorithm (which is also the input to the subsequent analysis routines) small by increasing the number of processors we use. So even in the case of the **V** dataset, where, as the top graph in Figure 10 illustrates, it is disadvantageous to increase the number of processors past 512 — the problem size is too small, and the communication overhead becomes overwhelming — it is still advantageous to increase processor count to decrease the size of the per-processor input to the analysis routines.

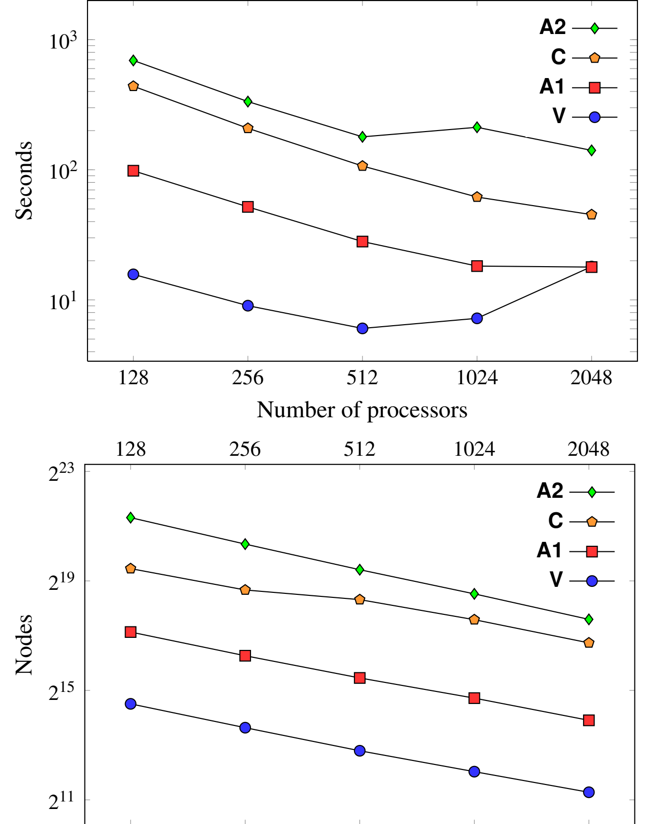


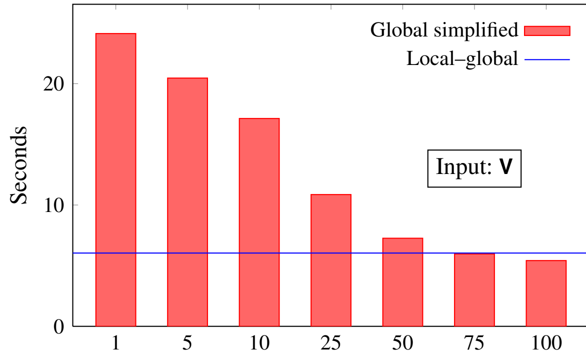
Figure 10: Times to compute the local–global representation (top) and its final (largest) tree sizes (bottom) for varying numbers of processors. Same data as in Table 2.

To get a sense of how the two methods influence analysis, we extract a persistence diagram for the **A1** dataset. When working with the local–global representation all 512 processors are participating in the computation; for the global simplified trees, we are limited to a single processor. It takes 3 seconds to get the persistence diagram in the former case versus 58 seconds in the latter (for the '8e11' threshold). Of course, the simplified diagram is less detailed. Although this comparison may seem strange — comparing one processor against 512 is hardly fair — it underlines our key message. Local–global representation lets us use all the available processors for post-processing. With global simplified trees we are stuck with a monolithic representation. It is worth noting that, in this case, the time to read the global simplified tree is about 14 seconds, i.e., it is significantly longer than the entire diagram extraction from the local–global representation.

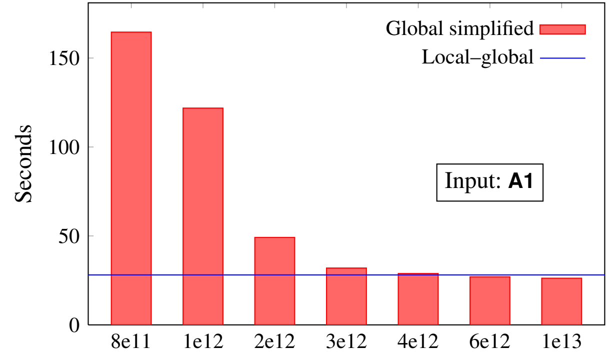
In a sequel [16] we show how a more computationally intensive analysis routine (level set extraction) takes advantage of our representation and speeds up as the local–global trees shrink with the increasing number of processors.

Skip trees. Figure 11 and Table 3 below illustrate the time it takes to merge two trees restricted to adjacent $1024^2 \times 512$ subsets of a $1,024^3$ grid. The trees have 803,589 and 821,941 nodes.

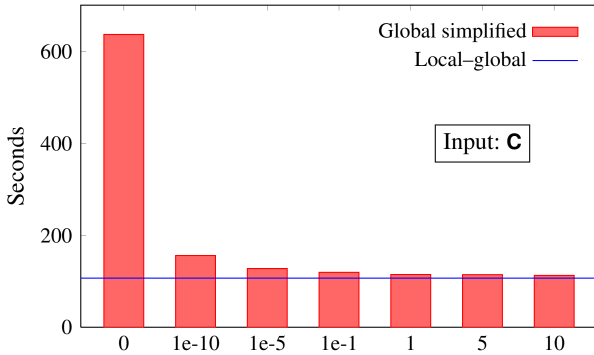
Figure 11 shows how the merging time decreases as we increase the finger size. The run times stop improving once the finger size becomes larger than six, but naturally this depends on the size of the input trees. More importantly, the run times do not increase as we increase the finger size further. Thus, to be safe, we set this constant to 16 in our code. A skip tree with all fingers of size one



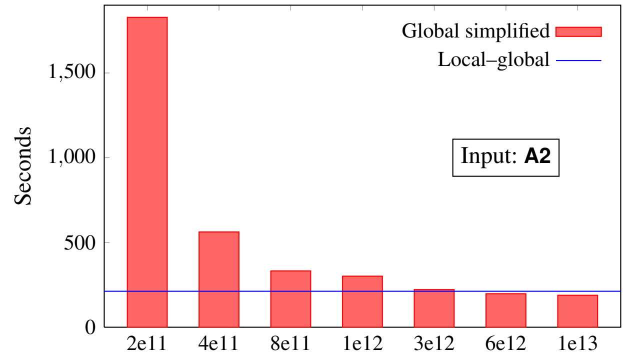
(a) Time to compute using 512 processors the global simplified trees, using different simplification thresholds, for the **V** dataset. It takes 6.04 seconds to compute the local–global representation of this dataset on 512 processors.



(b) Time to compute using 512 processors the global simplified trees, using different simplification thresholds, for the **A1** dataset. It takes 28.09 seconds to compute the local–global representation of this dataset on 512 processors.



(c) Time to compute using 512 processors the global simplified trees, using different simplification thresholds, for the **C** dataset. It takes 106.92 seconds to compute the local–global representation of this dataset on 512 processors.



(d) Time to compute using 1,024 processors the global simplified trees, using different simplification thresholds, for the **A2** dataset. It takes 212.19 seconds to compute the local–global representation of this dataset on 1,024 processors.

Figure 8: Running times to compute global simplified trees for different simplification thresholds.

iteration	local–global	1e13	8e12	6e12	4e12	3e12	2e12	1e12	8e11
Initial	42,594	6,396	6,464	6,580	6,889	7,502	9,960	24,171	28,733
1	46,656	10,612	10,733	10,933	11,484	12,519	17,112	43,572	52,247
2	53,814	16,848	17,041	17,428	18,439	20,259	28,887	80,538	96,969
3	62,983	26,207	26,483	27,017	28,736	31,613	46,917	145,089	176,516
4	75,836	39,366	39,929	41,056	44,387	50,082	80,182	271,390	332,159
5	98,981	64,605	65,845	68,045	74,462	85,364	144,362	520,391	637,634
6	134,004	102,184	104,389	108,566	120,849	142,282	258,474	1,010,099	1,244,505
7	173,369	148,638	152,808	161,053	186,206	229,442	463,442	1,967,922	2,435,072
8	236,334	226,894	235,175	251,101	298,308	378,679	828,682	3,774,931	4,689,989
9	297,786	315,921	332,831	365,347	460,797	623,497	1,529,351	7,437,553	9,269,091
Final	44,824	247,755	264,829	297,705	393,969	557,679	1,467,172	7,392,027	9,228,241

Table 1: Maximum size of a tree on any processor at every iteration of the computation of the local–global representation and of the global simplified trees for different simplification thresholds for **A1** dataset, using 512 processors. Figure 9 displays the same numbers graphically.

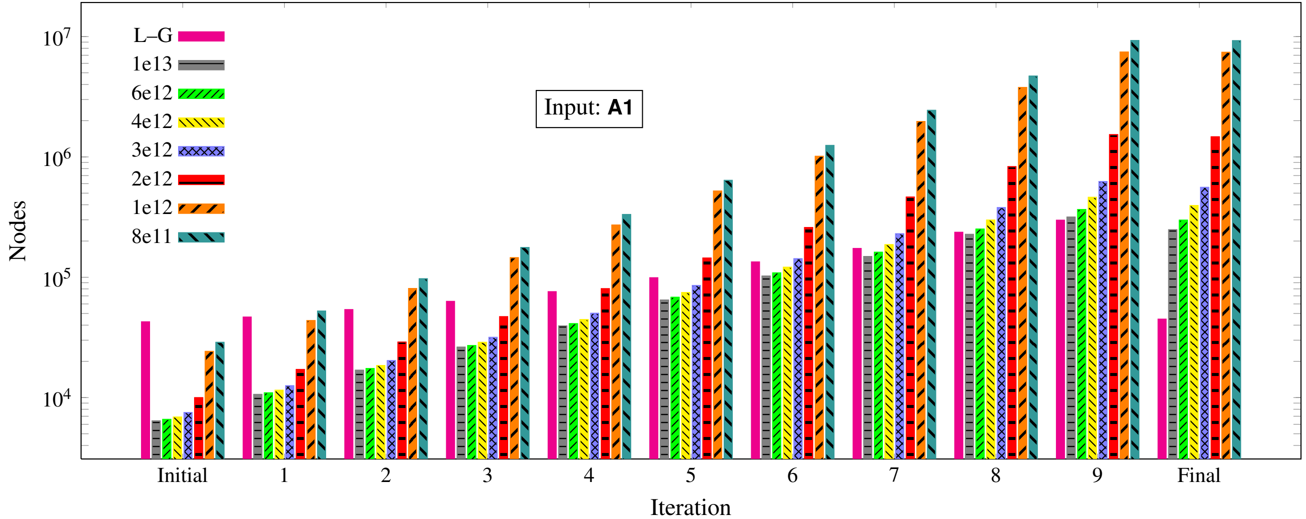


Figure 9: Maximum size of a tree on any processor at every iteration, using 512 processors. The raw data appears in Table 1.

	V		**A1**		**C**		**A2**	
processors	seconds	nodes	seconds	nodes	seconds	nodes	seconds	nodes
128	15.72	23,258	98.36	143,371	438.88	713,998	692.46	2,595,954
256	9.03	12,721	51.88	78,665	208.49	415,927	334.48	1,328,602
512	6.04	7,093	28.09	44,824	106.92	325,708	179.20	694,689
1,024	7.22	4,179	18.20	26,895	61.75	196,009	212.19	376,917
2,048	18.04	2,480	17.88	15,330	45.23	108,937	141.01	197,579

Table 2: Times to compute the local–global representation and its final tree sizes, in terms of the largest number of nodes on any processor, for varying numbers of processors. Figure 10 displays the same numbers as graphs.

is the same as the ordinary merge tree. Accordingly, merging two such skip trees is equivalent to applying the algorithm of [6]. It is also notable that with the increased finger size our merging strategy becomes faster than the union–find-based approach in [17]. We stress that this statement is true only for merging of two trees and not for the initial construction, where the Kruskal's-like algorithm is still superior.

The main reason we look for alternative merging strategies that do not depend on the serial constraints of the union–find algorithm is to improve the running time on multiple shared memory cores. As Table 3 illustrates, the skip tree merging is able to benefit from multiple cores on a 2×12-core AMD MagnyCours processor. Although all 24 cores share the same memory, the non-uniformity of memory access is noticeable: we saw no real improvement past 12 threads as well as a significantly diminishing improvement past six threads. Nonetheless, the union-find algorithm took 5.39 seconds and could only utilize one core, while the skip tree merge took 1.74 seconds with one thread and dropped to 0.42 seconds with 12 threads.

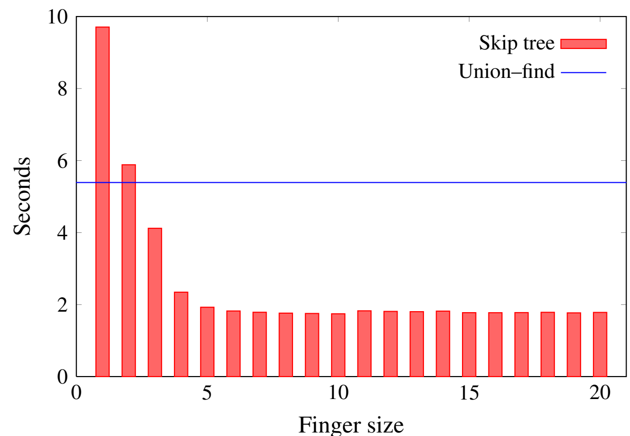


Figure 11: Merge time depending on the maximum size of the fingers in the skip tree. Finger size one is equivalent to applying the algorithm of [6] to an ordinary merge tree.

7. Conclusion

It is challenging to compute topological descriptors in parallel because they rely on global information. On the other hand, one of the main strengths of topological analysis is its ability to distinguish between stable and noisy features. We take advantage of this connection in Section 3 and reduce the size of the intermediate trees by interleaving computation and simplification.

We also benefit from taking a longer view by considering not just the computation of the individual descriptors, but also how we analyze them afterwards. To this end, we develop a local–global representation in Section 4. It is practical because it dramatically reduces the size of the trees during the computation. Even more useful is its output: instead of assembling all the information on

Threads	s	$\times$	Threads	s	$\times$
1	1.742	1.00	7	0.509	3.42
2	1.067	1.63	8	0.489	3.56
3	0.835	2.08	9	0.447	3.89
4	0.663	2.62	10	0.453	3.84
5	0.590	2.95	11	0.442	3.94
6	0.524	3.32	12	0.424	4.10

Table 3: Running times for skip tree merging using different number of threads. Columns labeled s show the times in seconds; $\times$ denotes the speed-up compared to the single thread. Merging the same dataset using the union–find algorithm takes 5.39 seconds.

one node, it distributes the tree across the processors, making the subsequent analysis fast. So even if the intermediate tree sizes grow throughout the computation, posing a challenge for future scaling — after all, individual nodes are still performing a binary reduction — we gain in the post-processing stage where the size of the input shrinks with the growing number of processors. Such an approach is likely to be useful for other topological descriptors.

On the shared-memory front, we present a parallel algorithm for merging two trees. It improves on the existing union–find-based techniques. Our experiments make it clear that new ideas will be necessary for the future architectures that will have many more cores with non-uniform memory access. Developing data structures that explicitly take this asymmetry into account is an important research direction.

Although our theoretical contributions apply in more general settings than the octree partitions we use as a running example in this paper, in practice, they require more work to deal with more complicated boundary and membership tests. We are currently focused on extending our implementation to adaptive mesh refinement grids.

Both because of the gains we find with the local–global representation and because of the challenges we discover along the way, we hope our contributions bring the community closer to the goal of parallel, scalable computation and analysis of topological descriptors.

Acknowledgments

This work was supported by the Director, Office of Science, Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 through the grant "Topology-based Visualization and Analysis of High-dimensional Data and Time-varying Data at the Extreme Scale," program manager Lucy Nowell, and by the use resources of the National Energy Research Scientific Computing Center (NERSC). The authors wish to thank Hank Childs, Terry Ligocki, Zarija Lukić, Peter Nugent, Casey Stark, and Matthew Turk for their encouragement and help.

References

[1] D. Attali, M. Glisse, S. Hornus, F. Lazarus, and D. Morozov. Persistence-sensitive simplification of functions on surfaces in linear time, 2009. Manuscript, presented at the Workshop on Topology In Visualization (TopoInVis'09).

[2] U. Bauer, C. Lange, and M. Wardetzky. Optimal topological simplification of discrete functions on surfaces. *Discrete and Computational Geometry*, 47(2):347–377, 2012.

[3] K. Beketayev, G. H. Weber, M. Haranczyk, P.-T. Bremer, M. Hlawitschka, and B. Hamann. Visualization of topology of transformation pathways in complex chemical systems. *Computer Graphics Forum (EuroVis 2011)*, pages 663–672, May/June 2011.

[4] P. Bendich, H. Edelsbrunner, D. Morozov, and A. Patel. Homology and robustness of level and interlevel sets. *Homology, Homotopy, and Application*, 2012. Accepted.

[5] P.-T. Bremer, G. H. Weber, V. Pascucci, M. S. Day, and J. B. Bell. Analyzing and tracking burning structures in lean premixed hydrogen flames. *IEEE Transactions on Visualization and Computer Graphics*, 16(2):248–260, 2010. doi: doi:10.1109/TVCG.2009.69.

[6] P.-T. Bremer, G. H. Weber, J. Tierny, V. Pascucci, M. S. Day, and J. B. Bell. Interactive exploration and analysis of large scale simulations using topology-based data segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 17(9):1307–1325, 2011.

[7] H. Carr and J. Snoeyink. Path seeds and flexible isosurfaces using topology for exploratory visualization. In *Data Visualization 2003 (Proceedings VisSym 2003)*, pages 49–58, New York, NY, 2003.

[8] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. *Computational Geometry—Theory and Applications*, 24 (2):75–94, 2003.

[9] H. Carr, J. Snoeyink, and M. van de Panne. Simplifying flexible isosurfaces using local geometric measures. In *Proceedings IEEE Visualization 2004*, pages 497–504, Oct. 2004. doi: http://dx.doi.org/10.1109/VISUAL.2004.96.

[10] H. Edelsbrunner and J. Harer. *Persistent homology—a survey*, volume 453 of *Contemporary Mathematics*, pages 257–282. American Mathematical Society, 2008.

[11] H. Edelsbrunner and J. Harer. *Computational Topology. An Introduction.* American Mathematical Society, Providence, Rhode Island, 2010.

[12] H. Edelsbrunner, D. Morozov, and V. Pascucci. Persistence-sensitive simplification of functions on 2-manifolds. In *Proceedings of the Symposium on Computational Geometry*, pages 127–134, 2006.

[13] A. Gyulassy, V. Pascucci, T. Peterka, and R. Ross. The parallel computation of Morse–Smale complexes. In *Parallel & Distributed Processing Symposium (IPDPS)*, pages 484–495, 2012.

[14] D. Laney, P.-T. Bremer, A. Macarenhas, P. Miller, and V. Pascucci. Understanding the structure of the turbulent mixing layer in hydrodynamic instabilities. *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1053–1060, 2006.

[15] A. Mascarenhas, R. Grout, P.-T. Bremer, V. Pascucci, E. Hawkes, and J. Chen. Topological feature extraction for comparison of length scales in terascale combustion simulation data. In V. Pascucci, X. Tricoche, H. Hagen, and J. Tierny, editors, *Topological Methods in Data Analysis and Visualization: Theory, Algorithms, and Applications*, pages 229–240, 2011.

[16] D. Morozov and G. Weber. Distributed contour trees. Manuscript, 2012.

[17] V. Pascucci and K. Cole-McLaughlin. Parallel computation of the topology of level sets. *Algorithmica*, 38(1):249–268, 2003. doi: 10.1007/s00453-003-1052-3.

[18] V. Pascucci, K. Cole-McLaughlin, and G. Scorzelli. *The Toporrery: Computation and Presentation of Multi-Resolution Topology*, pages 19–40. Springer-Verlag, 2009.

[19] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[20] G. Reeb. Sur les points singuliers d'une forme de pfaff complètement intégrable ou d'une fonction numérique. *Comptes Rendus de l'Acadèmie des Sciences de Paris*, 222:847–849, 1946.

[21] N. Shivashankar and V. Natarajan. Parallel computation of 3d Morse–Smale complexes. *Computer Graphics Forum (EuroVis 2012)*, 31(3): 965–974, 2012.

[22] N. Shivashankar, Senthilnathan M., and V. Natarajan. Parallel computation of 2d Morse–Smale complexes. *IEEE Transactions on Visualization and Computer Graphics*, 2012. To appear.

[23] G. H. Weber, P.-T. Bremer, M. S. Day, J. B. Bell, and V. Pascucci. Feature tracking using reeb graphs. In V. Pascucci, X. Tricoche, H. Hagen, and J. Tierny, editors, *Topological Methods in Data Analysis and Visualization: Theory, Algorithms, and Applications*, pages 241–253, 2011.