Extremely Scalable Distributed Computation of Contour Trees via Pre-Simplification

Mingzhe Li* University of Utah
Hamish Carr† University of Leeds
Oliver Rübel‡ Lawrence Berkeley National Laboratory
Bei Wang§ University of Utah
Gunther H. Weber¶ Lawrence Berkeley National Laboratory

Abstract

Contour trees offer an abstract representation of the level set topology in scalar fields and are widely used in topological data analysis and visualization. However, applying contour trees to large-scale scientific datasets remains challenging due to scalability limitations. Recent developments in distributed hierarchical contour trees have addressed these challenges by enabling scalable computation across distributed systems. Building on these structures, advanced analytical tasks—such as volumetric branch decomposition and contour extraction—have been introduced to facilitate large-scale scientific analysis. Despite these advancements, such analytical tasks substantially increase memory usage, which hampers scalability. In this paper, we propose a pre-simplification strategy to significantly reduce the memory overhead associated with analytical tasks on distributed hierarchical contour trees. We demonstrate enhanced scalability through strong scaling experiments, constructing the largest known contour tree—comprising over half a trillion nodes with complex topology—in under 15 minutes on a dataset containing 550 billion elements.

Index Terms: Contour Tree, Computational Topology, Distributed Algorithm, Branch Decomposition, Topological Data Analysis

1 Introduction

Scientific and engineering simulations have long been cornerstones of supercomputing. However, as the scale and complexity of data have continued to expand, the human capacity to assimilate and comprehend such data has remained relatively static. As the bandwidth of the human visual system remains inherently limited, the need for advanced data analysis and visualization tools has become increasingly pressing. One of the most successful tools in this domain to date has been computational topology, which leverages rigorous mathematical frameworks to enhance the visualization and interpretation of complex data. As topology-based analysis and visualization techniques have matured, attention has increasingly turned toward their parallelization and distribution—key steps required to fully realize their potential at the *exascale* where they are most critically needed.

One important topological descriptor is the contour tree, which analyzes contours (level sets of scalar fields) to reveal relationships between features defined by critical points, enables hierarchical simplification based on geometric properties, and serves as a foundational structure for isosurface-based visualization techniques.

*e-mail: mingzhe.li@utah.edu
†e-mail: h.carr@leeds.ac.uk
‡e-mail: oruebel@lbl.gov
§e-mail: beiwang@sci.utah.edu
¶e-mail: ghweber@lbl.gov

Recent work under the ECP ALPINE project [2] introduced efficient parallel algorithms within the PRAM model (Parallel Random Access Machine) for contour tree construction [14, 15], and methods for augmenting the contour tree and accelerating access to its structure [8]. Advances have also been made in computing geometric properties, performing simplification, and supporting visualization [26]. Building upon efficient on-node parallelism, subsequent work has achieved additional performance gains by leveraging distributed parallelism—both for contour tree computation [7] and for tasks such as augmentation, geometric property computation, and visualization [29].

Despite its strengths, contour tree computation has constraints that can limit the effectiveness of even the most advanced algorithms. First, the memory footprint can be $100 \sim 200$ bytes per cell for regular meshes, whose intrinsic format stores neighborhood information implicitly. Second, because topological analysis derives inherently global properties, substantial data exchange is needed between machines in a cluster, leading to bottlenecks in both inter-node communication and per-node storage. As a result, although the most recent work [29] achieved up to a $100\times$ speedup in distributed settings—on top of prior single-node gains of up to $200\times$—the full analytic pipeline became impractical for data sizes beyond 1024^3. This was the case even though the core contour tree computation [7] successfully handled volumes as large as $2048 \times 2048 \times 4096$ on a previous-generation system, and a related but simpler computation of merge trees scaled to 8192^3 data [35].

Contribution. In this paper, we apply a novel pre-simplification strategy that substantially reduces the memory overhead associated with analytical tasks on distributed hierarchical contour trees, thereby enabling highly scalable computation. For simulation data, the final stages of analysis and visualization typically rely on contour tree simplification to separate features from noise. We enhance the existing implementation by performing simplification both *before* and *after* the construction of the distributed contour tree, thereby reducing per-node memory footprint as well as inter-node communication cost.

Our framework enables contour tree analysis of the largest dataset reported to date—8192^3 grid points, or approximately 550 billion elements, corresponding to 4 TiB (tebibytes) of data—in less than 15 minutes using approximately 120 TiB of working memory distributed across 512 nodes.

Overview. We review contour tree computation in Sec. 2 and introduce pre-simplification for improved scalability in Sec. 3. Implementation details are provided in Sec. 4, where our code is integrated into VTK-m [31] (recently renamed to Viskores). Results are presented in Sec. 5, and conclusions are given in Sec. 6.

2 Background

Large-scale simulations investigate physical phenomena by numerically modeling physical properties, typically as continuous functions defined over a spatial domain. This domain is usually discretized into a mesh composed of individual cells, most commonly tetrahedral or cubic in shape. In this paper, we assume the data of

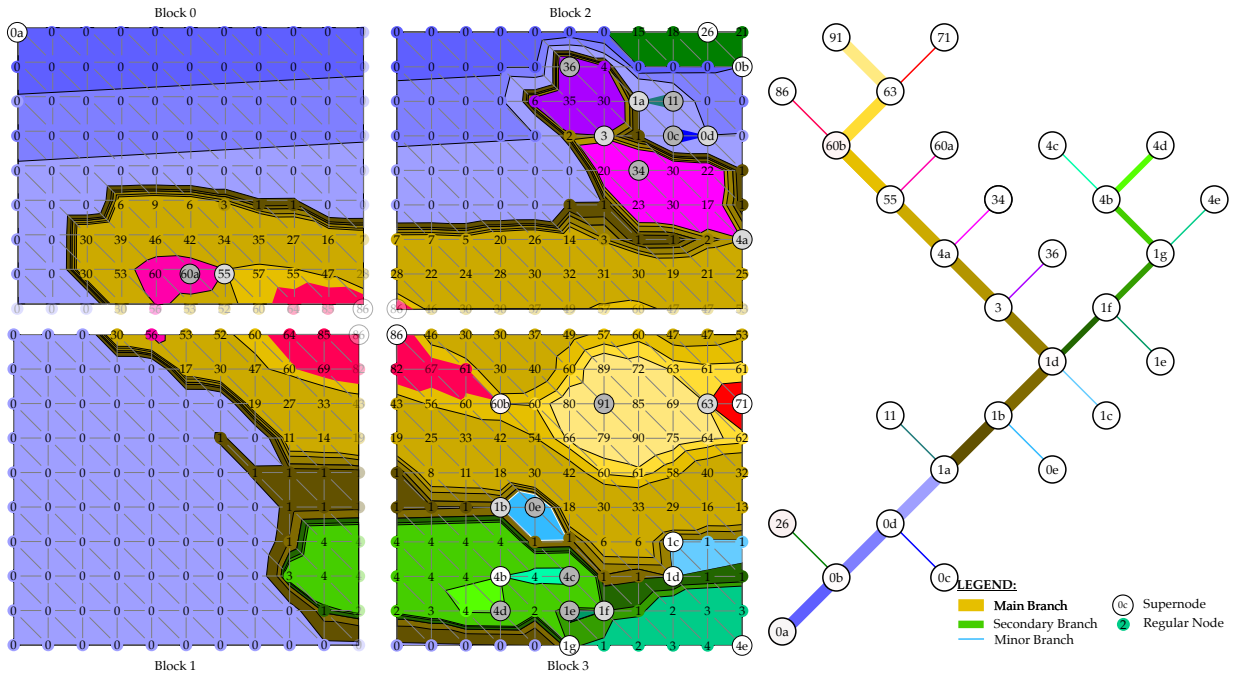

Fig. 1: Contour tree of a small dataset for the elevation of Vancouver, color-coded by topological zone/feature, with branch decomposition by area measure approximated with vertex count. Lettered values indicate the sorting order under simulation of simplicity [19]. Figure adapted from Li et al. [29, Fig. 2] © IEEE, with permission.

interest is given as a function $f : \mathbb{M} \to \mathbb{R}$ (where the manifold $\mathbb{M}$ is a subset of $\mathbb{R}^2$ or $\mathbb{R}^3$) with a suitable discretization.

We start by introducing the contour tree and its utilization in visualization (Sec. 2.1). We then introduce the computation of contour tree via serial (Sec. 2.2), locally parallel (Sec. 2.3), and distributed algorithms (Sec. 2.4).

2.1 Contour Trees

Given a function $f : \mathbb{M} \to \mathbb{R}$, the *level set* $f^{-1}(a)$ defines a subset of points in $\mathbb{M}$ at a given *isovalue* a; these are called *isolines* in 2D and *isosurfaces* in 3D. Each non-empty level set contains one or more connected components called *contours*, and we can study the function f by analyzing how these contours change as the isovalue a varies. Two points $x, y \in \mathbb{M}$ are considered *equivalent*, denoted $x \sim y$, if they have the same isovalue and belong to the same contour. Based on such an equivalent relation, we contract each contour to a single point, giving rise to a quotient space $\mathbb{M}/\sim$ first described by Reeb [37]. This quotient space collapses the data $(\mathbb{M}, f)$ to a skeletal structure—called the *Reeb graph*—which connects local minima, local maxima, and saddles of f.

If a Reeb graph contains no cycles (i.e., when the domain $\mathbb{M}$ is simply connected), it is a *contour tree*. The contour tree was originally defined independently as a data structure for efficient access to polygonal contours stored in memory [6]. We show a small example of a 2D contour tree in Fig. 1.

In a contour tree, *supernodes* occur at critical points of f (points with zero gradient), which are connected by *superarcs* representing equivalence classes of contours that map to *topological zones* in the input domain. Critical points where the number of contours does not change are not supernodes, for example, when an isosurface changes its genus. Most algorithms in computing contour trees begin by operating on simplicial meshes, where critical points are guaranteed to lie at the mesh vertices [5].

A contour tree can be *augmented* with *regular nodes*—most commonly the mesh vertices that are not critical points—as these nodes are essential for computing the geometric properties of the associated topological zones. If needed, these regular nodes are connected by *regular arcs* that break up the parent superarcs. Analysis of contour tree algorithms therefore depends primarily on the data size n (normally the same as the number of mesh vertices), sometimes the number of edges in the mesh m (which is $O(n)$ for regular meshes but may be $O(n^2)$ for irregular meshes), and the number of supernodes t in the contour tree.

The contour tree is used for terrain [6, 20, 21], to accelerate isosurface extraction [27, 40, 10], and for feature extraction [13], with variations adapted for volume rendering [41], protein molecule comparison [43], and energy landscape analysis [24].

2.2 Serial Contour Tree Computation

van Kreveld et al. [40] reported a serial algorithm for computing a contour tree. The algorithm performs a sweep of a contour through the simplicial mesh from high to low isovalues, explicitly tracking the cells intersected by the contour and updating this set as the sweep progresses through each mesh vertex. The algorithm has a time complexity of $O(m \log m)$ in 2D and $O(m^2)$ in higher dimensions. Tarasov and Vyalyi [39] extended this framework to $O(m \log m)$ in 3D with a complex three-sweep algorithm that requires special processing at the boundaries as well as repeated subdivision of the mesh to avoid topological problems.

Carr et al. [12] gave a serial algorithm that sweeps downwards through the data to identify relationships between local maxima and saddles, then upwards for local minima and saddles, producing two *merge trees*. Leaf edges are then transferred serially from the merge trees to construct the contour tree recursively. With an initial sorting and a modified union-find algorithm, the overall cost is $O(n \log n + \alpha(m, n))$, where α is the inverse Ackermann function. Later work [36] relaxed the initial assumption of a simplicial mesh, allowing trilinear interpolation on cubical cells. A more general approach [11] identified that the algorithm could use a *topology graph* constructed for arbitrary meshes and interpolants, such as the approximate trilinear interpolation from Marching Cube isosurfaces.

Computing the contour tree alone is insufficient for many analytical tasks such as branch decomposition, contour tree simplifica-

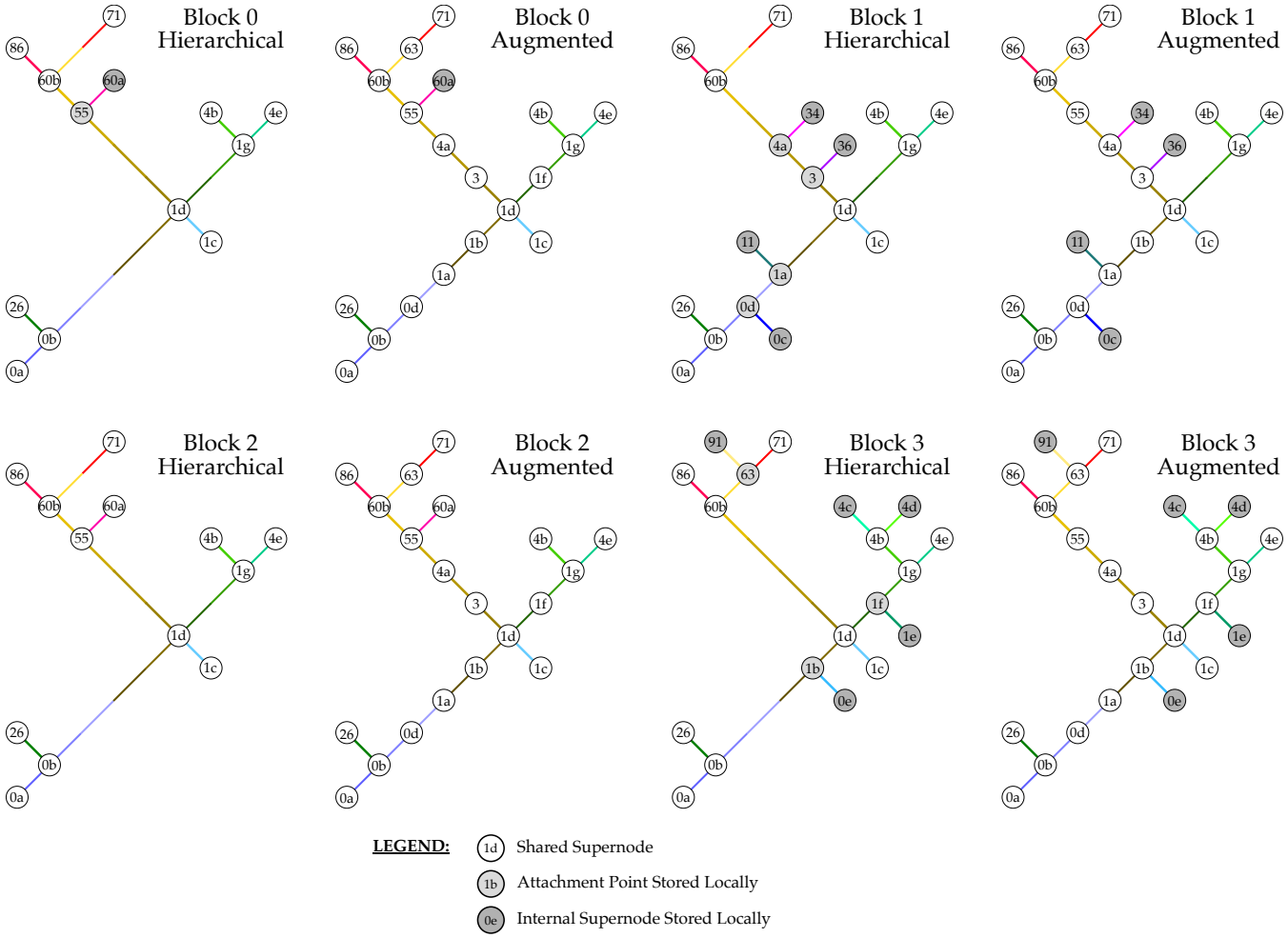

Fig. 2: Distributed hierarchical contour tree for Fig. 1, augmented for volume computation and branch decomposition. Augmentation increases the number of shared vertices, which becomes a bottleneck for distributed storage and communication. Figure adapted from Li et al. [29, Fig. 3] © IEEE, with permission.

tion, and contour extraction. For instance, contour tree simplification recursively discards leaves representing insignificant features to produce a *branch decomposition* [36]. This process is different from the cancellation of critical point pairs [18], when complex structures (so-called *W-structures* [25]) are present in the tree.

Geometric properties—such as contour surface area and enclosed volume—are also useful for identifying significant isosurfaces [4]. Subsequent work computed these properties for individual contours by sweeping inward through the contour tree [13], enabling their use as measures of importance for simplification. Fig. 1 shows the decomposition of the contour tree into branches based on surface area, with more important branches shown as thicker lines.

While branches are often thought of as linear paths within the contour tree, it is better to think of them as subtrees anchored to a single parent branch or trunk. This is because a contour taken just above the attachment point—where the branch connects to its parent—encloses all regions within the corresponding subtree. For instance, in Fig. 1, the branch $4d$–$1d$ connects to its parent at $1d$, and the contour at $1d + \epsilon$ (for an arbitrarily small ϵ) encloses not only the sequential regions $1d$–$1f$, $1f$–$1g$, $1g$–$4b$, and $4b$–$4d$ along the branch itself, but also the child branches $1g$–$4e$ and $4b$–$4c$.

2.3 Parallel Contour Tree Computation

Although a distributed parallel algorithm was introduced relatively early [36], the development of shared-memory parallel algorithms for contour trees emerged only after a significant delay. Acharya and Natarajan [1] used GPU parallelism to build monotone paths between critical points, then used them as a topology graph for the serial algorithm on CPU. Carr et al. [9] gave a distributed algorithm for computing contour trees on GPU based on quantizing the mesh, then contracting all contours in parallel with union-find. However, this approach was difficult to validate and incurred a substantial memory footprint, significantly limiting its practical utility.

Gueunet et al. [22, 23] segmented the mesh by isovalues, then computed separate contour trees for each segment on individual threads and merged them with a task-based approach, achieving peak speedups of approximately $10\times$. Smirnov and Morozov [38] used a task-based algorithm to add mesh edges incrementally, collapsing redundant edges to build merge trees. Carr et al. [14] presented a PRAM algorithm that employs pointer-jumping to construct monotone paths, which are then used to compute a topology graph. The algorithm then identifies in parallel one superarc from each remaining extremum to its governing saddle—i.e. the last saddle at which a monotone path can reach another extremum instead. Removing each extremum from the topology graph and redirecting its paths to its governing saddle then constructs a smaller topology graph in which some or all of the saddles become extrema. After a logarithmic number of passes, all superarcs in the merge tree have been identified.

A second phase replaced recursive leaf transfer from the merge trees with batched alternating transfers of all upper or lower leaves, using pointer-doubling to collapse vertical chains of vertices left by

removing leaves in each pass. This algorithm achieves a polylogarithmic computational cost and demonstrates practical speedups of nearly $50\times$ over the standard serial implementation.

Later work [8] added an accelerating *hyperstructure* related to rake-and-contract [30] to provide logarithmic access into the tree, allowing efficient augmentation with regular nodes. Each *hyperarc* in this hyperstructure captures a vertical chain collapsed in each pass of the batched transfer, and stores the superarcs (and supernodes) on each hyperarc in sorted order, permitting binary search by data value for rapid access into the tree.

Hristov et al. [26] employed segmented prefix-scan operations to sum values along the hyperarcs, replacing serial sweeps through the tree with parallelized *hypersweeps* over the hyperarcs to compute geometric properties. They then extracted a branch decomposition by using approximated volume as the measure, selecting the "most important" up and down arcs at each vertex in parallel, and applying pointer-jumping to collapse the branches.

Analytically, these algorithms run polylogarithmically in time, except where W-structures are present in the contour tree [25]; however, the impact of these structures is minimal in practice.

2.4 Distributed Contour Trees

Pascucci and Cole-McLaughlin [36] presented a distributed algorithm that computes individual contour trees for blocks of a dataset and then unites them into a topology graph for the combined contour tree. While effective in principle, the approach computes and stores the final contour tree on a single node. Given the substantial memory footprint discussed earlier, this limitation renders the algorithm ineffective for distributed computation until a fully distributed data structure is developed.

Nigmetov and Morozov [35] modified a previous local task-based parallel approach [33] with a triplet merge tree representation [38] to compute a distributed representation of the merge tree. Landge et al. [28] gave an algorithm that discards local features of merge tree, combining the remainder incrementatlly using a fan-in process. However, these approaches did not exploit local parallelism, and only computed the merge trees, without any of the secondary geometric or topological properties.

Carr et al. [7] adapted the previous distributed approach [36] by removing *interior forests* of superarcs that only exist on a given block from the contour tree at each stage of a fan-in process. The interior forests are then reinserted during a fan-out phase to build a *distributed hierarchical contour tree* (DHCT), which distributes shared superarcs across multiple machines, in a layered version of the hyperstructure. Peak speedups were reported of $70\times$ compared to the local PRAM algorithm, and maximum data size on NERSC's Cori supercomputer was $2048 \times 2048 \times 4096$.

Recently, Li et al. [29] observed that certain geometric computations depend on the precise ordering of superarcs along a hyperarc. Because the distribution strategy for the contour tree treated critical points with only local significance as regular points during construction, they introduced an additional stage to augment the tree with these points before implementing a distributed version of the hypersweep and simplification; see Fig. 2 for augmented contour trees corresponding to the example in Fig. 1. While this approach improved performance, the extra memory and communication required for all machines to process these supplementary *attachment points* limited the largest dataset computed and analyzed on NERSC's Perlmutter supercomputer to 1024^3.

3 Method

In the work of Li et al. [29], the contour tree computation itself scaled well, but secondary computations—such as geometric property evaluation and simplification—did not, due to the memory overhead of augmenting the local data structure with attachment points for interior forests on other machines. In this paper, to restore scalability following [29], we focus on analytic tasks with distributed contour trees, namely simplification and extraction.

The primary goal of the analysis is to extract a manageable set of significant contours that capture the key features of the data. This is commonly accomplished by simplifying the contour tree to a fixed number b of branches or by applying a fixed threshold Λ on an importance measure, which may—but need not—coincide with the measure originally used for branch definition. For clarity, we assume volume is used as the criterion for both branch decomposition and simplification, with a predefined volume threshold delineating the features of interest.

Since only a small number of important contours are selected, and most augmenting vertices serve as attachment points for small branches or subtrees, we simplify as many of these subtrees as possible *prior* to tree construction. This reduces the number of augmenting vertices and, in turn, lowers both memory footprint and communication cost. We introduce an additional threshold λ to designate subtrees as sufficiently unimportant to exclude from communication with neighboring nodes. We refer to this step as *pre-simplification*.

If $\lambda \leq \Lambda$, pre-simplification does not affect the final selection of contours for measures such as volume, which increase monotonically as one sweeps through the tree. Pre-simplifying a given subtree is equivalent [13] to replacing the function value throughout the subtree's topological zone with the value of the saddle point—or, in the case of a simplicial mesh, setting the values of all regular nodes in that zone to the saddle value. Consequently, we can treat all such nodes as belonging to the same superarc as the attachment point, ensuring that their contribution to the parent subtree's measure remains unaffected by pre-simplification. As a result, any subtree represented in the pre-simplified contour tree will retain the correct volume, and the desired outcome follows.

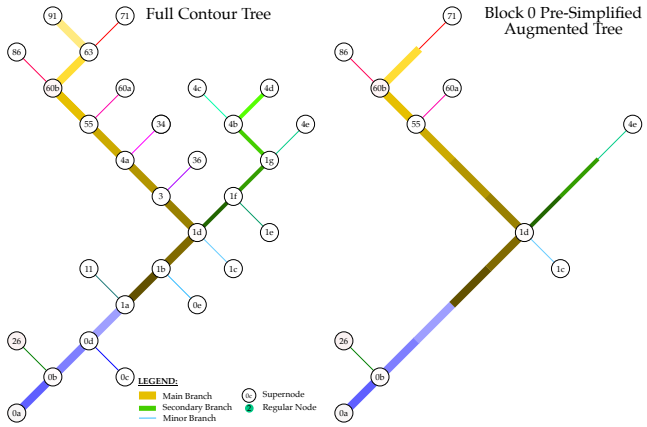

Fig. 3: A branch with the wrong leaf. Vertex $4d$ lies within the interior of a block and can thus be pre-simplified, whereas vertex $4e$, located on the block's boundary, cannot. After pre-simplification and augmentation, a branch will be selected from $1d - 4e$ rather than $1d - 4d$. Nonetheless, the subtree rooted at $1d$ is accurately represented.

One side effect is that, although remaining subtrees maintain the correct volume, the branch representing the subtree may have a different outer leaf in the pre-simplified contour tree. To understand why this occurs, consider Fig. 3. In this case, vertex $4d$ would typically be selected as the outer leaf of a branch rooted at $1d$, presenting the subtree containing $1f$, $1e$, $1g$, $4e$, $4b$, $4c$, and $4d$. However, since $4d$ lies within the interior of a block and $4e$ cannot serve as an attachment point due to its position on the boundary, the algorithm retains $4e$ and reports the branch $1d - 4e$ instead.

We outline the high-level steps for pre-simplified distributed hierarchical contour tree computation and branch decomposition, adapted from the pipeline in [29]:

1. Compute a distributed hierarchical contour tree;
2. Compute the correct volume for all superarcs in the tree;
3. For each block, list attachment points with measure $> \lambda$;
4. Augment the distributed tree with these attachment points;
5. Recompute the volumes for all superarcs in the tree;
6. Compute the distributed branch decomposition;
7. Simplify the contour tree to threshold Λ or to b branches;
8. Extract and render the contours.

Steps 1 and 2 remain unchanged, while Steps 7 and 8 are only minimally affected. The remaining steps require further explanation.

Step 3 (listing attachment points) identifies the subtrees of the interior forest with measure $> \lambda$ in a single logarithmic-cost PRAM pass. At this stage, the internal logic must be substantially revised to support augmentation [29]. The original goal in [29] was to avoid explicitly representing the supernodes connecting interior trees, thereby reducing communication costs in both fan-in and fan-out operations. However, situations arise—such as at vertex $1d$ in Fig. 1—where insertions occur at multiple levels of the hierarchy.

In the original DHCT, interior trees were stored by setting the superarc of the attachment point to null, as with the root in the non-distributed structure. A second array stored the *superparent*—the superarc to which each regular node (including supernodes) belonged. For efficiency, each superarc was indexed by the ID of its outer end; thus, supernodes normally had their superparent set to their own ID.

For attachment points already in a higher layer, the superparent could be assumed correct. Otherwise, the superarc ID into which the attachment point was to be inserted was stored as the superparent, avoiding the need for an additional array.

In the augmented distributed tree [29], a new distributed structure was constructed rather than editing the existing one. All attachment points were inserted from the outset, so the distributed hypersweeps, branch decomposition, and simplification routines did not need to check both superarc and superparent. With the introduction of pre-simplification, however, these routines needed to handle the possibility of uninserted attachment points, complicating Steps 4 through 7.

Step 4 (augmentation) proceeded largely as before, but with only a subset of attachment points exchanged during fan-in. After augmentation, the hierarchical tree's supernode IDs often differed, requiring recomputation of the volumes for each superarc and subtree (Step 5).

Steps 6, 7, and 8 were also modified to use new representative branch IDs. In the original approach [29], the correct extremum for each branch was known, and the branch ID was taken to be the outer end's ID. The mislabeling introduced by pre-simplification rendered this strategy unreliable. Instead, the saddle at the root of the subtree was used as the representative, again necessitating substantial changes in the detailed processing.

4 Implementation and Experiments

We begin with implementation details in Sec. 4.1, followed by a description of the datasets in Sec. 4.2. We then present the experimental setup in Sec. 4.3, and conclude with a parameter sensitivity analysis of the pre-simplification threshold in Sec. 4.4.

4.1 Implementation Details

Our implementation is based on VTK-m [31], an open-source library for efficient scientific visualization algorithms enhanced with on-node SMP parallelism. For distributed computation, we use the DIY [32] block-parallel library. We implement the pre-simplification strategy on top of the distributed hierarchical contour tree implementation in the VTK-m library. Specifically, we modify the `ContourTreeUniformDistributed` filter to update the pipeline with the pre-simplification strategy in Sec. 3. We add a hypersweep computation to superarc volumes before the augmentation, referred to as the *pre-augmentation hypersweep*. We only augment the contour tree with attachment points whose corresponding interior forest volume is higher than λ. In addition, we update the `SelectTopVolumeBranches` filter to limit branch selection in the simplified contour tree to branches with a volume above λ (i.e., to ensure $\lambda \leq \Lambda$). Our implementations are available at https://github.com/Viskores/viskores.

4.2 Datasets

We experiment with two large datasets: Nyx and MICrONS. Nyx is a 4096^3 dataset from cosmological simulations of particle mass density [3]: we use matter density Ω_m [34]) (the sum of baryon density and dark matter density) as the scalar field in a 3D volume.

MICrONS [16] is a volume of Electron Microscopy (EM) image data of a P60 mouse cortex with a volume of $1.4mm \times 0.87mm \times 0.84mm$ from the BossDB [42]. We work on a volume of 8192^3 voxels cropped from the original data.

Fig. 4 showcases visualizations for both datasets. In the 3D visualization for the Nyx dataset (1st and 2nd column), there are two 3D contours rendered in different colors, corresponding to the 68th and 89th highest volume branch of the contour tree. These extracted contours highlight the filamentary structures of matter density in the universe. The 3rd and 4th columns of Fig. 4 show the visualizations for 2D slices of the Nyx and the MICrONS dataset, respectively. In the 2D image of the Nyx dataset, there are similar filamentary structures to the 3D contours. In contrast, the MICrONS image shows many cell-structure shapes in the mouse cortex. The irregular and intricate shapes of features in both datasets contribute to a contour tree structure that is correspondingly large and complex.

4.3 Experimental Settings

Hardware configurations. All experiments are conducted on the National Energy Research Scientific Computing Center (NERSC)'s Perlmutter supercomputer with $3,072$ CPU-only and $1,792$ GPU-accelerated nodes. Our experiments were conducted on CPU-only nodes, each with two 2.45 GHz (up to 3.5 GHz) AMD EPYC 7763 (Milan) CPUs with 64 cores per CPU and two hardware threads per physical core, and 512 GB of DDR4 memory per node.

Computational parameter configurations. For all the experiments, we fix one data block per MPI rank and one MPI rank per CPU node. This is to reduce the number of volume subdivisions to minimize the size of boundary information that has to be shared across blocks, which leads to a scalability bottleneck in the distributed hierarchical contour tree framework [7]. For each compute node, we use 128 threads with OpenMP [17] for thread parallelism.

Algorithmic configurations. We retain the b branches with the largest volumes to simplify the contour tree based on the branch decomposition, fixing $b = 100$ for all experiments. Li et al. [29] showed that the runtime of contour tree simplification is insensitive to b; we select a small value so that no branch retained in the simplified contour tree is subject to pre-simplification. Parameter sensitivity with respect to λ is analyzed in Sec. 4.4. We omit contour extraction runtimes—reported in [29]—as they depend primarily on contour size rather than contour tree structure, and thus their scalability lies outside the scope of this work.

Runtime evaluation configuration. We report runtime of different pipeline phases: the first three phases, namely (1) computing the local contour tree, (2) fan-in, and (3) fan-out, contribute to the contour tree construction, which are the same as in the previous work [29]. The subsequent phases are the analytical tasks, including (4) the

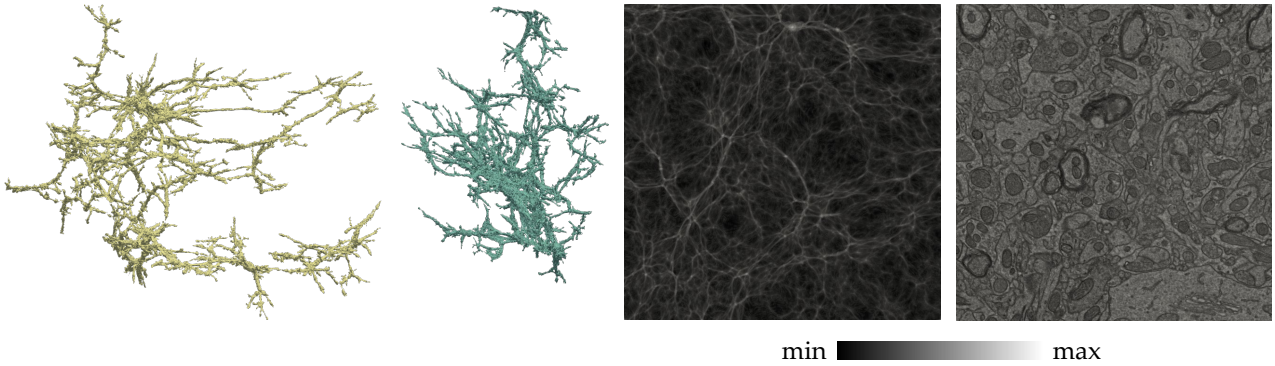

Fig. 4: Visualizations of the Nyx and MICrONS datasets using our framework. Left: two 3D contours of high-volume features extracted from a 1024^3 subvolume of the Nyx dataset. Right: 2D slices of the Nyx dataset and the MICrONS dataset respectively.

newly introduced pre-augmentation hypersweep, (5) augmentation, (6) post-augmentation hypersweep, (7) branch decomposition, and (8) extraction of the top-volume branches. Any remaining computational costs are grouped as "Other."

To collect runtime statistics, we place synchronization barriers after each phase. Due to the sequential nature of the pipeline, these barriers do not introduce significant overhead. For each phase, we report maximum runtime observed across all MPI ranks. Note that inter-rank communication occurs during the fan-in phase (Phase 2) and throughout the analytical computation stages (Phases 4–8).

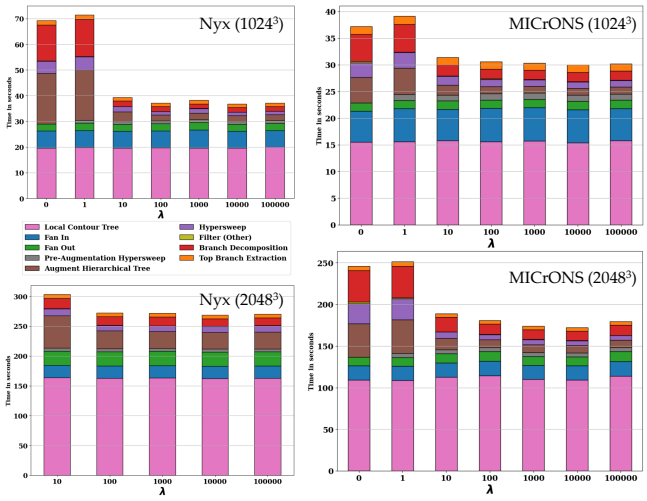

Fig. 5: Overall runtime using 16 nodes for the 1024^3 and 2048^3 subvolumes of the Nyx (left column) and the MICrONS (right column) datasets, respectively, with λ ranging from 0 to 10^5.

4.4 Parameter Sensitivity Analysis

Recall that we only exchange attachment points whose subtree volumes are higher than λ during augmentation, reducing the overall communication cost for augmentation and subsequent steps. In theory, the correctness of the final selection and extraction of contours is guaranteed if $\lambda < \Lambda$, which is the volume for the branch with the b-th highest volume. Here, we conduct a parameter sensitivity analysis for λ to observe the performance on reducing communication costs and to provide a summary on choosing λ.

Configurations. We use 16 CPU-only nodes for the analysis, evaluating performance for $\lambda \in \{0, 1, 10, 10^2, 10^3, 10^4, 10^5\}$ (all smaller than Λ for both datasets). When $\lambda = 0$, there is no pre-simplification, leaving the pipeline unchanged [29].

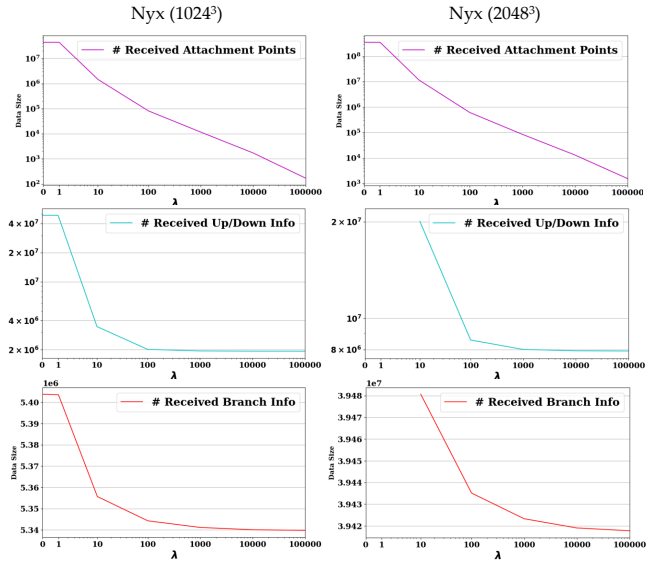

Fig. 6: The communication workload of attachment points (1st row, log-log), best up/down volume information (2nd row, log-log), and the branch information (3rd row, log-linear) for the 1024^3 and 2048^3 subvolumes of the Nyx dataset, respectively, with λ between 0 and 10^5. Each statistic is collected on the rank with the highest workload.

Evaluation metrics. To evaluate the runtime and communication workload for attachment points, we consider statistics for three types of communication workload: the highest number of attachment points received by a block, which depends on the pre-simplification thresholding process; the highest number of best up/down volume information received, which tracks the number of supernodes in the largest augmented contour tree of other blocks; the highest number of received branch outer end information, reflecting the highest number of branches in other blocks.

Nyx **dataset.** We examine parameter sensitivity for the Nyx dataset using subvolumes with 1024^3 and 2048^3 voxels. The statistics for runs on the 2048^3 subvolume with $\lambda \in \{0, 1\}$ are incomplete because these runs failed midway because the data size during branch decomposition exceeded the MPI communication size limit.

Fig. 5 (top left) shows that runtime on the 1024^3 Nyx subvolume is stable for $\lambda \geq 10$. Similarly, on the 2048^3 subvolume of Nyx (Fig. 5 bottom left), runtime stabilizes after $\lambda = 10$.

Fig. 6 shows communication statistics for the Nyx dataset at two subvolume sizes (1024^3 and 2048^3). As the pre-simplification threshold λ increases, the number of attachment points exchanged decreases nearly linearly (1st row), while the amount of best

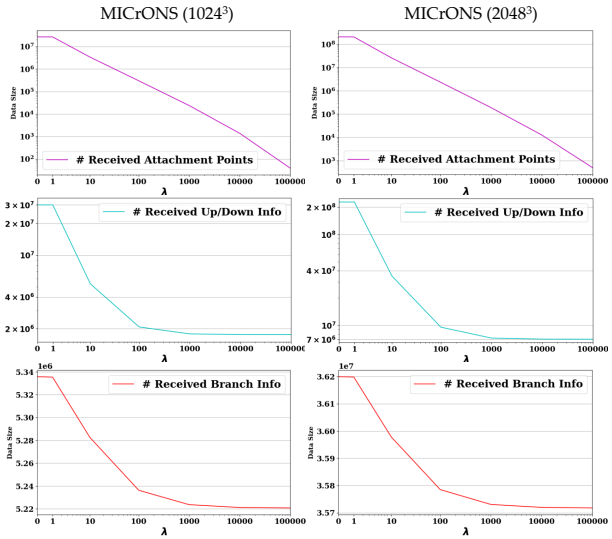

Fig. 7: Communication workload of attachment points (1st row, log-log), best up/down volume information (2nd row, log-log), and branch information (3rd row, log-linear) for 1024^3 and 2048^3 subvolumes of MICrONS, with λ ranging from 0 to 10^5. Each statistic is collected on the rank with the highest workload.

up/down information plateaus beyond $\lambda = 100$ (2nd row). The amount of branch information exchanged decreases only slightly, showing minimal sensitivity to λ (3rd row). Among the three metrics, the number of attachment points becomes less important because it is consistently smaller than the number of supernodes, which reflects the amount of best up/down information.

Improvements in both runtime and communication cost fall off after $\lambda = 100$, which is consistent across both subvolumes of different sizes. The consistency arises because the increase in data size primarily reflects a larger observable domain, while the typical feature volume remains relatively stable. Therefore, increasing λ for larger subvolumes of the Nyx dataset is unnecessary.

MICrONS dataset. We apply the same evaluations to MICrONS: runtime performance is in the right column of Fig. 5, and communication cost in Fig. 7. As with Nyx, we observe the bottleneck effect for the speed and communication cost improvement at $\lambda = 100$ for both the 1024^3 and 2048^3 subvolumes.

Summary. In summary, we can choose $\lambda \geq 100$ for the optimal performance on both datasets. If users can estimate Λ (the volume of the smallest preserved feature), they can choose λ with any value in $[100, \Lambda)$. Otherwise, without knowledge of the data to determine Λ, one would choose a small λ. Unless otherwise specified, for the remaining experiments on both datasets, we choose $\lambda = 100$, which is very small for the domain of size 1024^3 or larger. In addition, we provide a discussion for choosing λ in the appendix.

5 Results and Evaluation

In this section, we give experimental results and evaluate scalability. We start with our largest computed contour tree in Sec. 5.1, followed by the performance improvement in speed and data size compared to previous work in Sec. 5.2, then evaluate the algorithm's scalability in Sec. 5.3 using strong scaling and weak scaling.

5.1 Performance Evaluation

We compute a distributed hierarchical contour tree and its volumetric branch decomposition with pre-simplification on a 8192^3 subvolume (roughly 550 billion of voxels) of MICrONS, in less than 15 minutes; see Fig. 8 (right). We use 512 CPU-only nodes and approximately 120 TiB of memory. We apply $\lambda = 1000$ for pre-simplification, $10\times$ higher than indicated by the parameter sensitivity analysis. We produce a contour tree with more than half a trillion regular nodes, which is, to the best of our knowledge, the largest computed contour tree in the literature that supports advanced analytic tasks such as branch decomposition.

5.2 Performance Comparison

Existing implementations. We compare our distributed computation with pre-simplification to some existing implementations of contour tree computation and volumetric branch decomposition, in particular the standard serial algorithm—Sweep and Merge [12]—and the state-of-the-art shared-memory parallel implementation—Parallel Peak Pruning (PPP) [8, 26], as well as the distributed implementations with [29] and without pre-simplification.

All experiments are conducted on the same type of nodes on the Perlmutter supercomputer; see Sec. 4.3 for configurations. Due to memory limits, all shared-memory experiments use the 1024^3 subvolume of MICrONS. We collect runtime and memory costs, with the number of threads ranging from 1 to 128 for the PPP shared-memory implementation. We do not include runtime for top branch extraction (Phase 8, see Sec. 4.3) in the comparison because shared-memory implementations do not include this step.

Runtime performance. Fig. 9 gives performance results for all shared-memory implementations. The PPP implementation in serial is $2.08\times$ faster than the serial implementation due to algorithmic improvements and the internal optimization of VTK-m.

In parallel, however, runtime performance of PPP reaches up to about $11\times$ speedup with 128 threads compared to serial runs. On top of shared-memory parallelism, the distributed contour tree implementation provides additional improvements. Fig. 10 (right) presents the runtime performance of distributed computation without ($\lambda = 0$) and with ($\lambda = 100$) pre-simplification on the same subvolume of MICrONS. Using 64 nodes, runs without and with pre-simplification spent 26.52 seconds and 16.09 seconds, respectively. Therefore, on the 1024^3 subvolume of MICrONS, our implementation with pre-simplification reaches an estimated speedup of nearly $334\times$ over the Sweep and Merge serial version and roughly $160\times$ that of the PPP implementation with one thread. Compared to the parallel run with 128 threads, our distributed computation with pre-simplification reaches about $14.4\times$ speedup.

While memory limits prevent a single serial run for comparison, we can estimate serial runtime for a 8192^3 volume, assuming sufficient memory. We know that serial time complexity is dominated by the $O(n \log n)$ term from sorting, and recall the runtime for size $n = 2^{30}$ (1024^3) from in Fig. 9. A volume of 2^{39} (8192^3) voxels is $512 \times larger$, with an increased log factor of $39/30 = 1.3$, and we therefore estimate runtime for the Sweep and Merge implementation and the PPP (one thread) implementation to be approximately 3.58×10^6 seconds and 1.72×10^6 seconds, respectively. Our distributed computation with pre-simplification instead completes in 871.72 seconds; see Fig. 8 right column. The estimated speedup of our distributed implementation with pre-simplification is thus up to $4100\times$ over the Sweep and Merge implementation and roughly $1970\times$ that of the PPP implementation in serial.

While Li et al. [29] showed significant speedup of distributed computation over shared-memory, our pre-simplification process further reduces the communication cost for the distributed analytic computations; see Sec. 4.4. We demonstrate runtime improvement on the 1024^3 subvolumes of Nyx and MICrONSs in Fig. 10, in which we compare the runtime of experiments via the grouped bar charts. For each group of the bar plot in Fig. 10, the left bar shows the run without pre-simplification ($\lambda = 0$), and the right bar is the runtime with pre-simplification ($\lambda = 100$).

We can see the speed improvement on both datasets using pre-simplification under all node/rank configurations. Specifically, the runtime for analytical computation phases (all phases after fan-out)

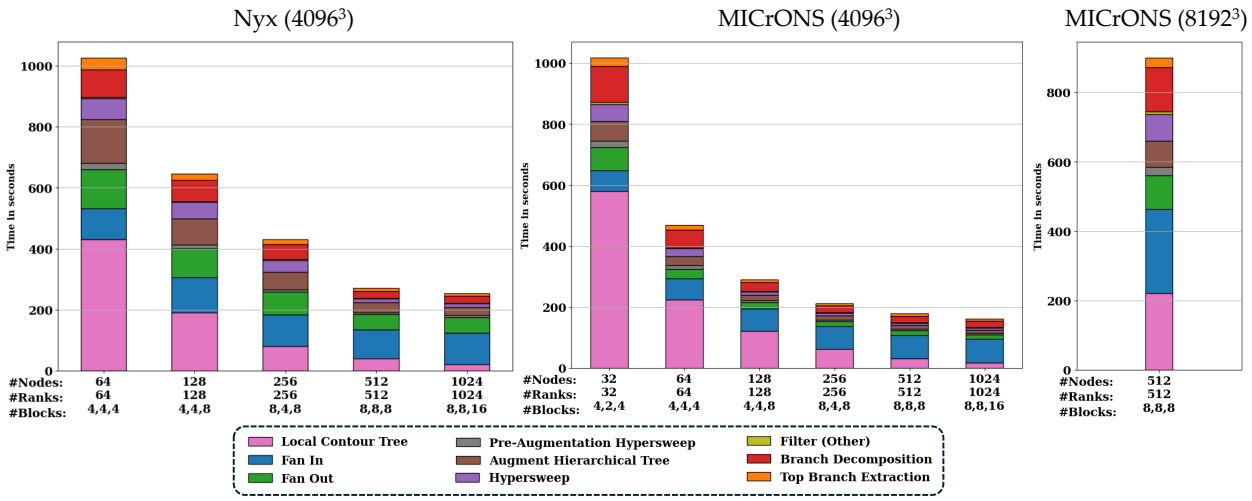

Fig. 8: Runtime performance using OpenMP on Perlmutter for the 4096^3 volume of Nyx (left), the 4096^3 subvolume of MICrONS (middle), and the 8192^3 volume of MICrONS (right), respectively.

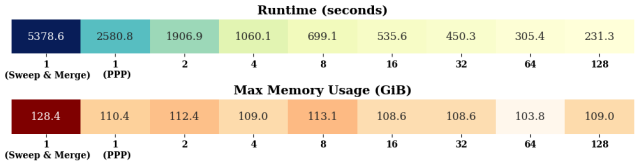

Fig. 9: Performance metrics by number of threads for shared-memory implementations on the 1024^3 subvolume of MICrONS.

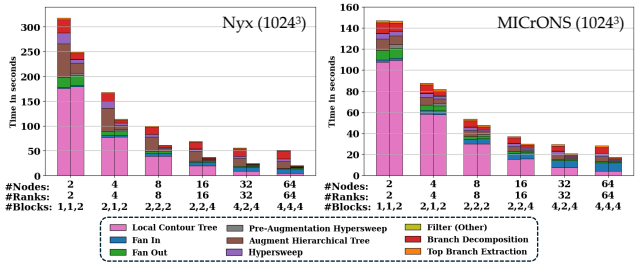

Fig. 10: Grouped bar charts for distributed runtime on 1024^3 subvolumes of Nyx and MICrONS. The left and right bars of each group show runtime with $\lambda = 0$ and $\lambda = 100$, respectively.

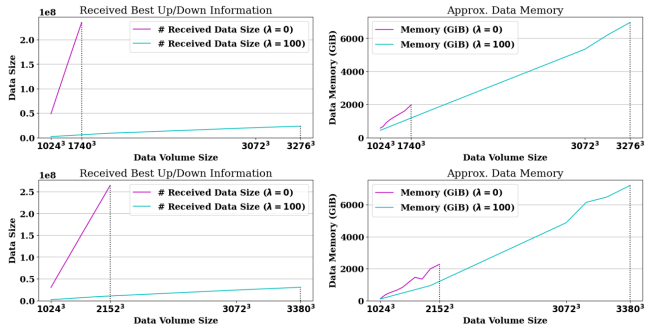

Fig. 11: The left column shows exchanged information size for best up/down volume data of supernodes w.r.t. data volume size. The right column shows approximate total data memory compared to data volume size. Top and bottom rows are for Nyx and MICrONS, respectively. Vertical dotted lines indicate the largest feasible data volume size for the corresponding λ choices using 16 nodes.

is improved by a large margin compared to the runs without pre-simplification. For Nyx, the largest total runtime improvement is about $2.58\times$ using 64 nodes. Similarly, for MICrONS, the largest total runtime improvement is about $1.60\times$.

Maximum feasible data size. The maximum feasible data size for shared-memory implementations is limited by the available memory per compute node. As shown in Fig. 9, the contour tree for a 1024^3 subvolume takes $103 \sim 129$ GiB of memory. Given a limit of 512 GB (≈ 476 GiB) per CPU-only node, linear memory growth would predict an upper limit around $1582^3 \sim 1700^3$ voxels. In contrast, our distributed approach with pre-simplification successfully processes a much larger volume of 8192^3 voxels, which is difficult to process on existing shared-memory platforms.

While previous work [7, 29] significantly increased data scale through distributed computation, this came at the cost of memory overhead due to augmentation, which reduced the maximum data size that can be processed under fixed memory constraints.

We examine maximum data size due to pre-simplification with ($\lambda = 100$) and without ($\lambda = 0$) pre-simplification, by gradually increasing the subvolume size from 1024^3 to 4096^3 in steps of 100 or 104 ($\approx 0.1 \times 1024$, while the boundary size needs to be divisible by 4) for both datasets. We fix the number of nodes at 16. We measure the communication size for the best up/down volume data information during the branch decomposition computation and the total memory consumed by the run reported by Perlmutter.

Fig. 11 demonstrates the statistics of evaluated metrics for Nyx in the top row and MICrONS in the bottom, in which the vertical dotted lines represent the maximum feasible data size for the two experimental configurations. Without pre-simplification ($\lambda = 0$), the framework can only compute a subvolume of up to 1740^3 voxels for Nyx and 2152^3 for MICrONS. In contrast, as we apply the pre-simplification with $\lambda = 100$, the largest feasible data size we can compute grows to 3276^3 for Nyx and 3380^3 for MICrONS. The maximum feasible volume size increase owing to the pre-simplification is roughly $6.67\times$ for Nyx and $3.87\times$ for MICrONS.

For the runs without pre-simplification, the data communication size grows significantly faster than that of the runs with pre-simplification for our implementation; see Fig. 11 left column. The runs without pre-simplification crashed after exceeding the one-time MPI communication limit. For finished runs without pre-simplification, their memory consumption (magenta line) is consistently higher than that of the runs with pre-simplification (cyan line); see Fig. 11 right column.

Lastly, we discuss memory efficiency by evaluating the footprint

per input voxel for all implementations. In a distributed hierarchical contour tree, each block stores a copy of the shared contour tree structure, increasing the total memory footprint. In addition, augmentation exchanges attachment points across blocks, leading to multiple copies of attachment points in the memory. Therefore, it is expected that the distributed implementations would have lower memory efficiency than the shared-memory ones.

Following Fig. 9, shared-memory requires approximately $103 \sim 129$ bytes per voxel in the input mesh on MICrONS. For distributed computations, the memory footprint needed for each voxel is reflected in Fig. 11 right column. On MICrONS, the run without pre-simplification on the 2152^3 subvolume needs 2273 GiB; the average memory footprint per input voxel is roughly 244 bytes. In contrast, with pre-simplification, the distributed computation on the 3380^3 subvolume consumes 7199 GiB; the average memory usage for each input voxel is approximately 200 bytes. In other words, while the pre-simplification strategy improves the overall memory efficiency, the extent of the improvement is moderate. Regardless of pre-simplification, the memory usage for distributed contour tree computation is roughly twice as much as the shared-memory implementations. Such sacrifice in memory efficiency is acceptable since we can increase the total memory size by adding compute nodes.

5.3 Scalability Evaluation

Strong scaling. We evaluate strong scaling of the pre-simplified pipeline by measuring performance at a fixed problem size while varying the number of compute nodes. For this, we use fixed input volumes of size 4096^3 for both Nyx and MICrONS. The number of nodes is increased from the minimum feasible option for each dataset—64 nodes for Nyx and 32 nodes for MICrONS—up to 1024 nodes, with one MPI rank assigned per node.

Fig. 8 demonstrates the strong scaling plot for the runtime performance of our framework on the 4096^3 volume of both datasets. The stacked box plots separate the overall runtime by the pipeline phases. As we add more nodes, the runtime on both datasets consistently decreases, reaching the near-optimal value at 512 nodes.

Among the runtime of phases, all but the fan-in phase have gained a noticeable amount of speedup as the number of nodes increases for both datasets. Recall that the analytical computation phases require communication, the size of which is heavily affected by the number of attachment points. Without pre-simplification, Li et al. [29] have shown the scalability limitation of these analytical computation steps due to attachment points (reflected in Fig. 10). With the pre-simplification strategy, the limitation in scalability is largely mitigated. While there is still communication overhead for the shared tree structure, the analytical computation steps are no longer the scalability bottleneck.

With pre-simplification, the bottleneck of scalability becomes the fan-in operation, which is a critical step in constructing the contour tree. In this step, shared boundary information needs to be exchanged between adjacent blocks to construct the shared contour tree structure that goes across the block boundary. We are unsure whether this can be optimized, so we leave it for future work.

Weak scaling. We provide the weak scaling analysis for the optimized framework. For both Nyx and MICrONS, we start with a subvolume of $1024 \times 1024 \times 512$ voxels. As we increase the number of nodes and blocks, we simultaneously grow the subvolume size accordingly so that each compute node is assigned a subvolume at a fixed size.

We report the runtime and the weak scaling efficiency in Fig. 12. Among all the phases, only the local contour tree computation has a roughly constant runtime. All other phases become increasingly expensive as the number of nodes grows. This is because the amount of work to communicate and process the data for the shared tree structure naturally increases, leading to increasing communication overheads and a drop in the weak scaling efficiency [7]. On the other hand, the runtime growth for all the analytical computation phases is comparable to or slower than that of the fan-in phase. In other words, with our pre-simplification strategy, such analytical computations are not the bottlenecks of the algorithm's scalability.

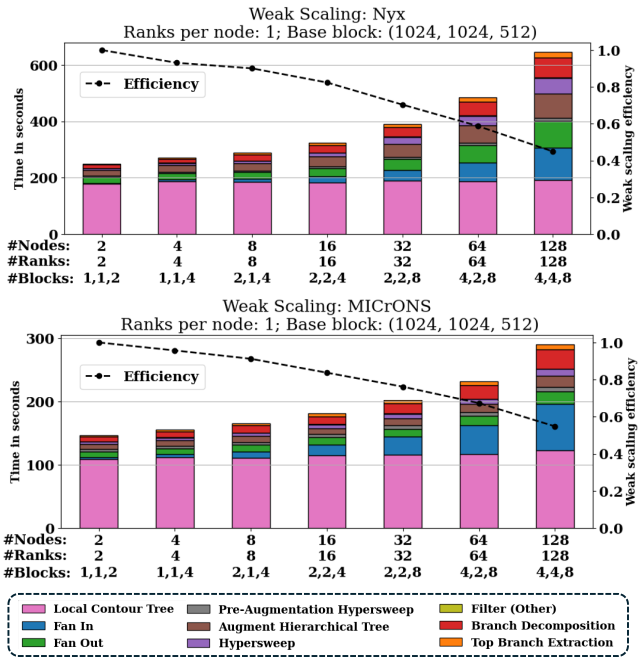

Fig. 12: Runtime performance with a growing number of nodes (and blocks) and data size, while each node is assigned a $1024 \times 1024 \times 512$ subvolume (weak scaling) with one MPI rank per node. The top and bottom rows are for Nyx and MICrONS, respectively.

6 Conclusion

We introduce a pre-simplification strategy to optimize the analytical computation for distributed hierarchical contour trees. First, with the pre-simplification process, we generate the largest known contour tree on a volume of size 8192^3 with complex topology within 15 minutes. Second, we demonstrate a runtime speedup of up to $334\times$ over the serial computation and $14.4\times$ over the parallel implementation on a 1024^3 dataset. Assuming there is enough memory for the serial computation on the data volume of size 8192^3, our performance is expected to reach up to $4100\times$ speedup over the serial version in theory. Moreover, our pre-simplification strategy enables $1.60 \sim 2.58\times$ speedup for the distributed computation and supports $3.87 \sim 6.67\times$ larger size of data volume with a fixed number of compute nodes. Lastly, our pre-simplification strategy has largely mitigated the scalability issue of the distributed contour tree computations in [29].

Acknowledgments

This research was supported by the U.S. Department of Energy (DOE), Office of Science, Advanced Scientific Computing Research (ASCR) program and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the DOE Office of Science and the National Nuclear Security Administration under Contract No. DE-AC02-05CH11231 to the Lawrence Berkeley National Laboratory. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility using NERSC award ASCR-ERCAP0026937. Additionally, Mingzhe Li and Bei Wang were partially supported by DOE DE-SC0021015 and National Science Foundation (NSF) IIS-2145499. Hamish Carr was supported by the University of Leeds.

REFERENCES

[1] A. Acharya and V. Natarajan. A parallel and memory efficient algorithm for constructing the contour tree. In *Proceedings of the 2015 IEEE Pacific Visualization Symposium (PacificVis)*, pp. 271–278. IEEE, New York, 2015. doi: 10.1109/PACIFICVIS.2015.7156387 3

[2] J. Ahrens, M. Arienti, U. Ayachit, J. Bennett, R. Binyahib, A. Biswas, P.-T. Bremer, E. Brugger, R. Bujack, H. Carr, et al. The ECP ALPINE project: In situ and post hoc visualization infrastructure and analysis capabilities for exascale. *International Journal of High Performance Computing Applications (IJHPCA)*, 39(1):32–51, 2025. doi: 10.1177/10943420241286521 1

[3] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. Van Andel. Nyx: A massively parallel AMR code for computational cosmology. *The Astrophysical Journal*, 765(1):39, 2013. doi: 10.1088/0004-637X/765/1/39 5

[4] C. L. Bajaj, V. Pascucci, and D. R. Schikore. The contour spectrum. In *Proceedings of Visualization 1997*, pp. 167–173. IEEE, New York, 1997. doi: 10.1109/VISUAL.1997.663875 3

[5] T. F. Banchoff. Critical Points and Curvature for Embedded Polyhedra. *Journal of Differential Geometry*, 1:245–256, 1967. doi: 10.4310/jdg/1214428092 2

[6] R. L. Boyell and H. Ruston. Hybrid techniques for real-time radar simulation. In *Proceedings of the 1963 Fall Joint Computer Conference*, pp. 445–458. ACM, New York, 1963. doi: 10.1145/1463822.1463869 2

[7] H. Carr, O. Rübel, and G. H. Weber. Distributed hierarchical contour trees. In *2022 IEEE 12th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 1–10. IEEE, New York, 2022. doi: 10.1109/LDAV57265.2022.9966394 1, 4, 5, 8, 9, 12

[8] H. Carr, O. Rübel, G. H. Weber, and J. Ahrens. Optimization and augmentation for data parallel contour trees. *IEEE Transactions on Visualization and Computer Graphics*, 28(10):3471–3485, 2022. doi: 10.1109/TVCG.2021.3064385 1, 4, 7

[9] H. Carr, C. Sewell, L.-T. Lo, and J. Ahrens. Hybrid Data-Parallel Contour Tree Computation. Technical Report LA-UR-15-24579, Los Alamos National Laboratory, 2015. 3

[10] H. Carr and J. Snoeyink. Path seeds and flexible isosurfaces: Using topology for exploratory visualization. In *Proceedings of Eurographics Visualization Symposium 2003*, pp. 49–58, 285, 2003. doi: 10.5555/769922.769927 2

[11] H. Carr and J. Snoeyink. Representing interpolant topology for contour tree computation. In H.-C. Hege, K. Polthier, and G. Scheuermann, eds., *Topology-Based Methods in Visualization II*, Mathematics and Visualization, pp. 59–73. Springer, Berlin, Heidelberg, 2009. doi: 10.1007/978-3-540-88606-8_5 2

[12] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. *Computational Geometry Theory and Applications*, 24(2):75–94, 2003. doi: 10.1016/S0925-7721(02)00093-7 2, 7

[13] H. Carr, J. Snoeyink, and M. van de Panne. Flexible isosurfaces: Simplifying and displaying scalar topology using the contour tree. *Computational Geometry: Theory and Applications*, 43(1):42–58, 2010. doi: 10.1016/j.comgeo.2006.05.009 2, 3, 4

[14] H. Carr, G. H. Weber, C. Sewell, and J. Ahrens. Parallel peak pruning for scalable SMP contour tree computation. In *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 75–84. IEEE, New York, 2016. doi: 10.1109/LDAV.2016.7874312 1, 3

[15] H. A. Carr, G. H. Weber, C. M. Sewell, O. Rubel, P. Fasel, and J. P. Ahrens. Scalable contour tree computation by data parallel peak pruning. *IEEE Transactions on Visualization and Computer Graphics*, 27(4):2437–2454, 2021. doi: 10.1109/TVCG.2019.2948616 1

[16] M. Consortium, J. A. Bae, M. Baptiste, A. L. Bodor, D. Brittain, J. Buchanan, D. J. Bumbarger, M. A. Castro, B. Celii, E. Cobos, F. Collman, N. M. da Costa, S. Dorkenwald, L. Elabbady, P. G. Fahey, T. Fliss, E. Froudarakis, J. Gager, C. Gamlin, A. Halageri, J. Hebditch, Z. Jia, C. Jordan, D. Kapner, N. Kemnitz, S. Kinn, S. Koolman, K. Kuehner, K. Lee, K. Li, R. Lu, T. Macrina, G. Mahalingam, S. McReynolds, E. Miranda, E. Mitchell, S. S. Mondal, M. Moore, S. Mu, T. Muhammad, B. Nehoran, O. Ogedengbe, C. Papadopoulos, S. Papadopoulos, S. Patel, X. Pitkow, S. Popovych, A. Ramos, R. C. Reid, J. Reimer, C. M. Schneider-Mizell, H. S. Seung, B. Silverman, W. Silversmith, A. Sterling, F. H. Sinz, C. L. Smith, S. Suckow, M. Takeno, Z. H. Tan, A. S. Tolias, R. Torres, N. L. Turner, E. Y. Walker, T. Wang, G. Williams, S. Williams, K. Willie, R. Willie, W. Wong, J. Wu, C. Xu, R. Yang, D. Yatsenko, F. Ye, W. Yin, and S.-c. Yu. Functional connectomics spanning multiple areas of mouse visual cortex. *Nature*, 640(8058):435–447, 2025. doi: 10.1038/s41586-025-08790-w 5

[17] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998. doi: 10.1109/99.660313 5

[18] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological Persistence and Simplification. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pp. 454–463. IEEE, 2000. doi: 10.1109/SFCS.2000.892133 3

[19] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990. doi: 10.1145/77635.77639 2

[20] H. Freeman and S. P. Morse. On Searching A Contour Map for a Given Terrain Elevation Profile. *Journal of the Franklin Institute*, 284(1):1–25, 1967. doi: 10.1016/0016-0032(67)90568-6 2

[21] C. Gold and S. Cormack. Spatially Ordered Networks and Topographic Reconstruction. In *Proceedings of the 2nd International ACM Symposium on Spatial Data Handling*, pp. 74–85, 1986. doi: 10.1080/02693798708927800 2

[22] C. Gueunet, P. Fortin, and J. Jomier. Contour forests: Fast multi-threaded augmented contour trees. In *2016 IEEE 6th IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 85–92. IEEE, New York, 2016. doi: 10.1109/LDAV.2016.7874333 3

[23] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Task-based augmented merge trees with Fibonacci heaps. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 6–15. IEEE, New York, 2017. doi: 10.1109/LDAV.2017.8231846 3

[24] W. Harvey and Y. Wang. Topological landscape ensembles for visualization of scalar-valued functions. *Computer Graphics Forum*, 29(3):993–1002, 2010. doi: 10.1111/j.1467-8659.2009.01706.x 2

[25] P. Hristov and H. Carr. W-structures in contour trees. In I. Hotz, T. Bin Masood, F. Sadlo, and J. Tierny, eds., *Topological Methods in Data Analysis and Visualization VI*, pp. 3–18. Springer, Cham, 2021. doi: 10.1007/978-3-030-83500-2_1 3, 4

[26] P. Hristov, G. H. Weber, H. Carr, O. Rübel, and J. Ahrens. Data parallel hypersweeps for in situ topological analysis. In *2020 IEEE 10th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 12–21. IEEE, New York, 2020. doi: 10.1109/LDAV51489.2020.00008 1, 4, 7, 12

[27] T. Itoh and K. Koyamada. Isosurface generation by using extrema graphs. In *Proceedings Visualization '94*, pp. 77–83, 1994. doi: 10.1109/VISUAL.1994.346334 2

[28] A. G. Landge, V. Pascucci, A. Gyulassy, J. C. Bennett, H. Kolla, J. Chen, and P.-T. Bremer. In-situ feature extraction of large scale combustion simulations using segmented merge trees. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1020–1031, Nov. 2014. doi: 10.1109/SC.2014.88 4

[29] M. Li, H. Carr, O. Rübel, B. Wang, and G. H. Weber. Distributed Augmentation, Hypersweeps, and Branch Decomposition of Contour Trees for Scientific Exploration. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 31(1):152–162, 2025. doi: 10.1109/TVCG.2024.3456322 1, 2, 3, 4, 5, 6, 7, 8, 9, 12

[30] G. L. Miller and J. H. Reif. Parallel Tree Contraction Part I: Fundamentals. *Advances in Computing Research*, 5:47–72, 1989. 4

[31] K. Moreland, C. Sewell, W. Usher, L. ta Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, M. Larsen, C.-M. Chen, R. Maynard, and B. Geveci. VTK-m: Accelerating the visualization toolkit for massively threaded architectures. *IEEE Computer Graphics and Applications*, 36(3):48–58, 2016. doi: 10.1109/MCG.2016.48 1, 5

[32] D. Morozov and T. Peterka. Block-parallel data analysis with DIY2. In *2016 IEEE 6th Symposium on Large Data Analysis and Visualiza-*

tion (LDAV), pp. 29–36. IEEE, New York, 2016. doi: 10.1109/LDAV.2016.7874307 5

[33] D. Morozov and G. Weber. Distributed merge trees. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 93–102. ACM, New York, 2013. doi: 10.1145/2442516.2442526 4

[34] NASA / LAMBDA Archive Team. LAMBDA - Matter Density, 2019. Accessed Apr 10th, 2025, https://lambda.gsfc.nasa.gov/education/graphic_history/matterd.html. 5

[35] A. Nigmetov and D. Morozov. Local-Global Merge Tree Computation with Local Exchanges . In *SC19: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13. IEEE Computer Society, Los Alamitos, CA, USA, Nov 2019. doi: 10.1145/3295500.3356188 1, 4

[36] 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 2, 3, 4

[37] 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. 2

[38] D. Smirnov and D. Morozov. Triplet merge trees. In *Topological Methods in Data Analysis and Visualization V. TopoInVis 2017.*, Mathematics and Visualization, pp. 19–36. Springer, Cham, 2020. doi: 10.1007/978-3-030-43036-8_2 3, 4

[39] S. P. Tarasov and M. N. Vyalyi. Construction of Contour Trees in 3D in $O(n \log n)$ steps. In *Proceedings, 14th ACM Symposium on Computational Geometry*, pp. 68–75, 1998. doi: 10.1145/276884.276892 2

[40] M. van Kreveld, R. van Oostrum, C. L. Bajaj, V. Pascucci, and D. R. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proceedings of the 13th ACM Symposium on Computational Geometry*, pp. 212–220. ACM, New York, 1997. doi: 10.1145/262839.269238 2

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

[42] B. Wester, W. Gray-Roncal, S. Hider, T. Gion, J. Matelsky, J. Downs, D. Xenes, T. Rose, K. Romero, L. Kitchell, D. Ramsden, M. Sanchez, and D. Moore. The brain observatory storage service & database (BossDB). Accessed March 30th, 2022, https://bossdb.org/project/microns-minnie. 5

[43] X. Zhang, C. L. Bajaj, and N. Baker. Fast matching of volumetric functions using multi-resolution dual contour trees. Technical report, Texas Institute for Computational and Applied Mathematics, Austin, Texas, 2004. 2

A Discussion: Parameter Estimation

In Sec. 4.4, we conducted a parameter sensitivity analysis on the two datasets to determine λ, running a sequence of exponentially increasing λ values. This analysis provides a systematic approach for identifying the optimal λ. In this section, we further discuss criteria to help users estimate an appropriate value for λ, focusing in particular on determining its minimum value.

Estimating the minimum value of λ. We begin by defining the terms used in the discussion. Let N denote the total data volume and R the number of ranks. The parameter λ determines which attachment points are exchanged: specifically, only those whose interior forest volume exceeds λ are included. Let α be the number of attachment points received from other ranks. An upper bound on α for each rank is

$$\frac{N - N/R}{\lambda + 1},$$

where $N - N/R$ represents the maximum external data volume for a rank, and $\lambda + 1$ specifies the minimum interior forest volume of attachment points to be exchanged. This bound is loose, as part of the data volume is already represented in the shared contour tree structure.

The first criterion for determining λ, referred to as the *memory criterion*, ensures that sufficient memory is available for analytical computations. Based on this criterion, we estimate the minimum value of λ through two runs on a subvolume of the data, followed by a test run on the full dataset.

For example, to estimate the minimum λ satisfying the memory criterion for the 2048^3 volume of the Nyx dataset, we first run the framework on a 1024^3 subvolume with two values of λ: 0 and 100. The run without pre-simplification ($\lambda = 0$) consumes 574.66 GiB of memory, whereas the run with $\lambda = 100$ uses 439.17 GiB. In parallel, the number of attachment points drops from $697{,}320{,}285$ to $1{,}288{,}810$. This reduction implies that processing approximately 6.96×10^8 attachment points requires about 135.49 GiB of memory, or roughly 209.02 bytes per attachment point.

Next, we perform a test run on the full 2048^3 volume using a large λ value (e.g., $\lambda = 10^5$) to eliminate most attachment points and measure memory consumption. If this run fails due to insufficient memory, the hardware configuration is unlikely to be viable for the dataset, regardless of λ. If successful, we record the peak memory usage for any single rank—in our case, 133.26 GiB (note that this is not the total memory usage across all ranks). The remaining available memory must then be sufficient to handle attachment point computations.

In this example, $N = 2048^3$, $R = 16$, and each rank (i.e., compute node) has 512 GB of available memory. Since the upper bound on the number of attachment points is $\frac{N - N/R}{\lambda+1}$, and the memory required for attachment point computation is 209.02 bytes per voxel on average, we require

$$209.02\,\alpha < 209.02 \frac{N - N/R}{\lambda + 1} < 512 \times 10^9 - 133.26 \times 1024^3.$$

From this, we estimate the minimum λ for this example to be 4.

The second criterion concerns communication overhead, which we refer to as the *communication criterion*. Pre-simplification reduces the number of attachment points exchanged during communication, thereby mitigating scalability limits. As shown in Sec. 4.4, both the attachment points and the shared contour tree structure contribute to this overhead. With increasing λ, the number of attachment points decreases, and eventually the shared contour tree structure becomes the dominant factor. For optimal scalability, our goal is to reduce the number of attachment points to be comparable to, or smaller than, the size of the shared contour tree, which has been shown [26, 7, 29] to be bounded by $O(N^{2/3})$ for 3D data. This implies that λ should be on the order of $\Omega(N^{1/3})$ for optimal scalability.

Limitation. We conclude by discussing the limitations of our approach for estimating λ. First, the memory criterion requires recording statistics such as the number of attachment points and the memory usage for each rank. Although our implementation includes logging functionality, this method still entails additional effort. Second, the estimated minimum λ for the first criterion is likely higher than the true minimum, as it is based on the worst-case distribution of attachment points. Third, for the communication criterion, constant factors in the computation make it difficult to determine a precise minimum value of λ.

Parameter choices. While pre-simplification significantly reduces communication overhead and enables the processing of much larger datasets, it also removes some small-volume features that may be important in certain tasks or data contexts. To preserve such features, we aim to choose λ as small as possible, subject to satisfying the memory criterion (and optionally the communication criterion), and ensuring that $\lambda < \Lambda$. However, if the estimated λ is substantially larger than the expected size of the smallest relevant features, pre-simplification may not be suitable for the application.

Currently, our implementation supports only a single, global λ for pre-simplification as a means of reducing the number of attachment points in the computation. Since the augmentation step can be performed on arbitrary subsets of attachment points [29], it would be possible to customize λ for different subareas or subvolumes of the data, depending on the features of interest, or even to selectively preserve specific features—an extension we leave for future work.