
Article:
Carr, HA orcid.org/0000-0001-6739-0283, Rübel, O and Weber, GH (Accepted: 2022) Distributed Hierarchical Contour Trees. IEEE Transactions on Visualization and Computer Graphics. ISSN 1077-2626 (In Press)

Distributed Hierarchical Contour Trees

Hamish A. Carr*
University of Leeds

Oliver Rübel†
Lawrence Berkeley National Laboratory

Gunther H. Weber‡
Lawrence Berkeley National Laboratory

Abstract— Contour trees are a significant tool for data analysis as they capture both local and global variation. However, their utility has been limited by scalability, in particular for distributed computation and storage. We report a distributed data structure for storing the contour tree of a data set distributed on a cluster, based on a fan-in hierarchy, and an algorithm for computing it based on the *boundary tree* that represents only the superarcs of a contour tree that involve contours that cross boundaries between blocks. This allows us to limit the communication cost for contour tree computation to the complexity of the block boundaries rather than of the entire data set.

1 Introduction

Topological analysis helps comprehend data from numerical simulations. *Reeb analysis* exploits relationships between isocontours in data, often using the contour tree. For data analysis and visualization, features of a physical phenomenon often map to superarcs in a contour tree [8], and simplifying the contour tree constructs a semantically meaningful hierarchy called the branch decomposition [24]. Geometric properties of the topological zones can to guide the simplification [8], while individual contours can be extracted using the contour tree [2,27].

The principal limitation of these tools at scale has been scalability, as only one distributed algorithm has been defined [23] for the contour tree, but without exploiting on-node parallelism and with the disadvantage of computing the entire tree on a single machine with insufficient memory to store it.

We describe a distributed structure, the *hierarchical contour tree*, which represents local contours only on the machine where the data block itself resides, resulting in much lower communication cost and storage. This works with any local contour tree algorithm, but was built using the parallel peak pruning algorithm [10], as it exploits the hyperstructure [6] which supports augmentation, acceleration and computation in a massively parallel environment.

We contribute an explicit algorithm for distributed computation *and storage* of the contour tree, based on the PPP algorithm [10] but adaptable to any single-machine contour tree, that minimizes communication between ranks by retaining as many superarcs as possible on the rank, thus reducing the size to be stored on the top node of the hierarchy, and extending the distributed scalability and performance of previous algorithms. We have implemented this algorithm using vtk-m and DIY, made the code available in vtk-m, and report on its efficiency, both theoretical and practical.

2 Background

Given a function $f : M \to \mathscr{R}$ from a manifold M to the real numbers $\mathscr{R}$, known as a scalar field, we define the *level set* of an *isovalue* $h \in \mathscr{R}$ to be $f^{-1}(h) = x \in M : f(x) = h$. We then call the connected components of each level set *contours* or *isocontours*.

The quotient space of M by contour contraction gives the *Reeb graph* [25], whose vertices or *supernodes* are critical points of f, and whose edges are *superarcs*. Additional *nodes* represent other points in the domain, breaking the superarcs up into *arcs*. A contour tree with any such nodes and arcs added is called *augmented*.

An acyclic Reeb graph is a *contour tree* $T_f(M)$ for the function f on the manifold M [5]. If M is homeomorphic to a disk, the graph is always acyclic, but this is not a necessary condition. This observation is crucial for the boundary tree and the hierarchical contour tree, and we will return to it in due course.

We can also map from points in the manifold to the corresponding point in the contour tree, and use *topological zone* or *zone* for the inverse images under this mapping of superarcs, supernodes, or arbitrary subgraphs. Fig. 1 shows a small example of a contour tree, with color indicating topological zones, which are treated as features for the purpose of simplification, visualization and data analysis.

Computationally, we usually assume that the manifold is defined to be a mesh, most simply a simplicial mesh with barycentric interpolation, although other meshes and interpolants can be supported by using a *topology graph* to represent the relationships between contours [7]. For the balance of this paper, we will assume that the manifold is a simplicial mesh with barycentric interpolation.

For a function f on such a mesh M, critical points are at vertices [3], and contour tree computation collapses to a graph algorithm. Algorithmic analysis then relies on N - the number of nodes in the mesh, and t - the number of supernodes in the contour tree.

The contour tree can be computed from a topology graph by sweeping through the mesh in sorted order, testing at each vertex for creation, destruction, merge or separation of contours [27]. A more efficient approach sweeps through the mesh twice (once in each direction) to compute *merge trees* [8], which capture the connectivity of super- or sub-level sets of f (i.e. sets of the form $f(x) \geq h$ and $f(x) \leq h$), and which are also of interest in data analysis. A third phase then combines the merge trees to produce the contour tree, identifying leaf edges based on the information in the merge trees and transferring them to the contour tree [8]. While efficient ($O(N \lg N) + t\alpha(t)$), this algorithm is inherently serial, and most subsequent work on parallel contour tree computation has sought to improve its parallel efficiency.

2.1 Parallel Contour Tree Algorithms

Pascucci & Cole-McLaughlin [23] distributed the computation by dividing the mesh into *blocks* on individual *ranks*, and having each rank compute the contour tree for its own block. These trees were combined using a standard fan-in process, where the trees of adjacent blocks are united to build a topology graph for the combined block, and the contour tree was then computed for the combined block. The contour tree for the entire data set was computed recursively, and stored on a single machine at the top of the hierarchy, with the communication cost in each stage was linear in the tree size. However, many of the nodes ended up idle during the fan-in reduction, leading to inefficient resource utilization, and the top node of hierarchy is prone to run out of memory and compute resources, as the contour tree for the entire data set resides on it.

More recently, SMP parallel algorithms have emerged, as CPU core count increases and GPU processing becomes common. One approach took a path-following approach [11] and GPU-parallelized it to find a topology graph, then computed merge and contour trees on CPU [1]. Another approach breaks the mesh into segments by

*e-mail: h.carr@leeds.ac.uk
†e-mail: ORuebel@lbl.gov
‡e-mail: ghweber@lbl.gov

isovalue, computes contour trees for each segment on a thread, then glues together the segments across the boundaries [13]. Subsequently, a task-based approach used queued processing to construct the contour tree a few edges at a time [14]. Another merge tree construction in SMP is also task-based, in which edges are incrementally added, causing changes to propagate locally until the entire contour tree has been computed [26].

Distributed parallel algorithms for merge or contour trees suffer from two main problems: i) With increasing number of nodes, communication costs become prohibitive and impede scalings; ii) When computing a global contour (or merge) tree, after each step only half of the compute nodes continue calculating, until only a final compute node assembles the result. This approach results in poor compute node utilization and poor scaling behavior.

To alleviate these problems, Morozov & Weber introduced distributed merge trees [21]. This approach distributed the merge tree to multiple compute nodes and used the resulting distributed representation for analysis. In this, compute nodes exchange merge trees, simplifying away regular and critical points that do not belong to a more persistent path with an extremum on a neighbouring node. Effectively, each final local tree contains all nodes of the merge tree for the local domain as well as only those nodes of other domains that are on a path to an extremum in another domain.

Landge et al. [18] suppressed interior vertices of blocks, computed the blocks merge trees, and fanned in by gluing together the merge trees, using a *zipping* operation towards the root of the tree. Implicitly, sub- (or super-) level set components interior to a block are discarded. However, computations internal to a block such as zipping are expressed in serial forms, since the approach exploits distributed parallelism but not shared memory parallelism.

Subsequently, Klacansky et al. [17] represented virtual edges bridging between adjacent blocks, exploiting this to allow shared-memory parallel computation of merge trees with a similar approach.

Overall, therefore, Pascucci & Cole-McLaughlin defined a distributed approach for contour tree computation, but one that was hampered by the need to fan-in the entire contour tree to a single node. Subsequent approaches have primarily built a distributed merge tree, which is known to be easier than building the contour tree, especially in the presence of W-structures [9, 15]. And, while subsequent work enabled some operations that usually require the contour treee [22], these operations are based on separate merge trees (join- and split-tree) that are *not* combined into a contour tree. As such, this representation does not support operations like persistence based simplification of contour trees. Finally, most of these methods do not exploit the massive on-node parallelism typical of modern architectures.

2.2 Parallel Peak Pruning

We introduced the Parallel Peak Pruning (PPP) algorithm [10], in which monotone paths to extrema are constructed from all vertices in the mesh. These paths define a topology graph, which is used to compute the merge trees. All extrema and their nearest saddle by isovalue are identified simultaneously and transferred to the merge tree (peak pruning). The topology graph is adjusted to remove these extrema, and saddles become extrema to be pruned in turn. Once both merge trees have been constructed, a batched merge phase transfers groups of edges to the contour tree.

In each iteration, monotone chains of superarcs transfer as *hyperarcs* to the contour tree - i.e. a specialised form of branch decomposition [24] that is related to rake-reduce parallel tree algorithms [16]. In this way, a *hyperstructure* is built up that allows logarithmic search in the contour tree for efficient parallel processing. We later exploited this hyperstructure to augment the contour tree with regular nodes at logarithmic overall cost [6].

While search is logarithmic, hyperstructure construction is not, due to *W-structures*, which zig-zag horizontally through the tree [9, 15], causing problems with parallel computation.

In subsequent work [16], we then showed that the hyperstructure was also related to the *Euler Tour* [4], which allows computations over trees to be collapsed into prefix sum operations. By exploiting this, we demonstrated efficient shared-memory parallel computation of geometric measures, simplification, branch decomposition and isocontour extraction.

We omit detailed discussion of this algorithm, since the distributed approach we outline does not depend strongly on it, and can also be built on top of other algorithms for rank-local computation of the contour tree.

3 Hybrid Parallel Contour Trees

While there are a number of approaches for SMP computation of the contour tree, distributed computation remains an important step, and it is this problem that we now approach.

We start with two limitations of distributed computation: communication cost and memory footprint. Since communication is often a bottleneck in distributed computation, we want to minimize it by reducing the amount of data transferred. Moreover, we want a data structure distributed across all ranks (a *rank* is an abstraction of an MPI process to allow load-balancing between machines and permit exploitation of multiple cores on each machine). This avoids concentrating the entire data structure (and the associated large memory requirements) on a single rank. These goals are mutually compatible, since partitioning the data to be stored on each rank will give rise fairly naturally to reduced communication cost.

One approach would be to add distributed array read/write primitives to PPP, but this is potentially expensive. Although many stages are heavily redundant in memory access patterns, sorting is a recurrent theme, and memory access is unlikely to be rank-local. Moreover, overall communication would require multiple logarithmic depth ($O(\lg N)$) fan-ins with $O(N)$ total communication each.

In comparison, a hybrid SMP-distributed algorithm should communicate only necessary data between ranks. Numerical computations often communicate data proportional to the size of the boundaries between the blocks, with communication cost of about $O(N^{2/3})$ in individual steps, where N is the number of vertices in the input mesh. For data sets in the range of $10^{12} - 10^{18}$, the potential savings in communication is of the order of $10^4 - 10^6$, and we therefore elected to build a hybrid algorithm.

We start by observing that many features are local to a block of data and do not have to be represented on other blocks (white zones in Fig. 1). Since a superarc whose topological zone does not intersect any boundary can be computed correctly on the block, there is no need to communicate it to any other block. We therefore remove all such superarcs / topological zones to produce a *boundary tree*. The algorithm then combines the boundary trees of individual blocks to compute the rest of the contour tree for the entire data set, following Pascucci & Cole-McLaughlin [23].

At each stage, we divide the contour tree into the *boundary tree*, which captures all contours intersecting the boundaries with other blocks, and the *interior forest*, which captures all contours that do not. We combine boundary trees pairwise in a fan-in, removing interior forests at each stage. When the fan-in is complete, we have a single shared tree for all ranks, plus a different history of removals on each rank, which we then re-insert to compute a set of hierarchical contour trees on individual compute ranks.

4 Boundary Trees and Interior Forests

We saw in Sect. 2 that a Reeb graph may be acyclic even if the manifold M is not homeomorphic to a disk. For example, removing a leaf superarc from a contour tree is equivalent to removing the corresponding topological zone from the domain of f [8]. Simplifying a contour tree thus cuts holes in the domain, which is not homeomorphic to a disk, but the corresponding Reeb graph is still acyclic and therefore a contour tree. In 2D, this causes the domain to have holes like a sheet of lace, while in 3D, it causes the block to look like a piece of Swiss cheese.

The gluing boundary $G(B)$ of a block B is the set of points shared with other blocks of data. We assume that it is simply connected. We define the boundary tree $B_f(B) = C^{-1}(G(B))$ to be the inverse image in the contour tree of the gluing boundary. For example, in Fig. 1, the

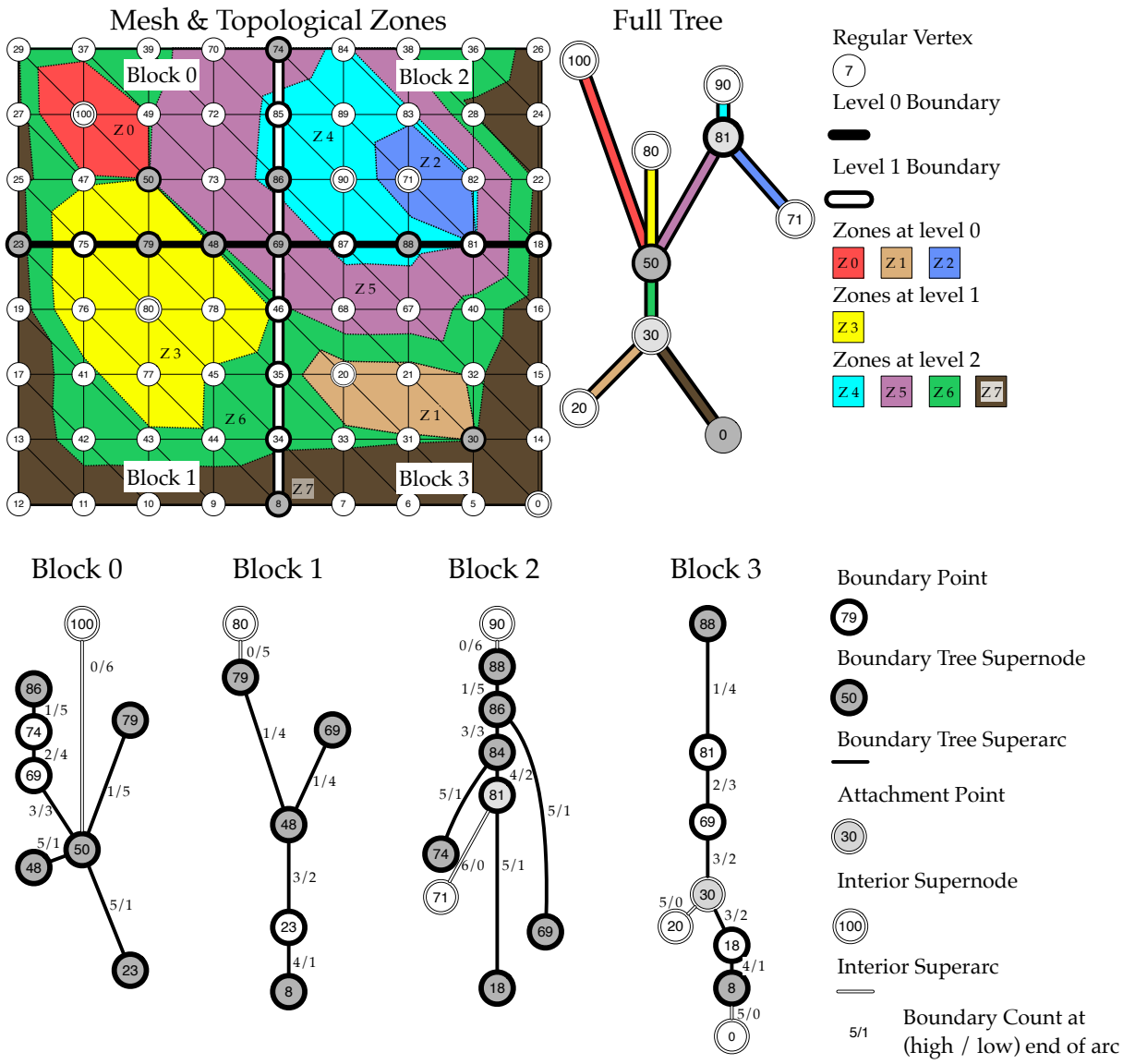


Fig. 1: Top: A small data set with topological zones marked, with the correct full contour tree. Bottom: Contour trees for individual blocks divided into boundary trees and interior forests. Note that a vertex may be a supernode in one block but not another (e.g. 23), and that superarcs in the blocks do not necessarily correspond to the zones shown. See text for full discussion.

gluing boundary of Block 0 consists of the heavy black edges between Block 0 and Blocks 1 and 2.

Since $G(B)$ is connected, the *boundary tree* of B, $B_f(B)$ is a connected subset of the tree $T_f(B)$. The remaining superarcs representing contours interior to the block then form a forest of zero or more trees, which we call the *interior forest* $I_f(B) = T_f(B) - B_f(B)$. For a barycentric simplicial mesh, the boundary tree is the closure under connectivity of the set of mesh vertices on the gluing boundary. We can then define the *necessary manifold* to be the manifold defined by the boundary tree – i.e. $N_f(B) = f(B_f(B))$, the portion of the domain whose connectivity is captured in the boundary tree. Trivially, $B_f(B)$ is then the contour tree for $N_f(B)$.

We illustrate this for the mesh in Fig. 1. We break this mesh into four blocks, and show the contour tree for each block augmented by all boundary points. For example, in Block 0, the maxima are 100, 86 and 79, while the minima are 48 and 23, and there is a saddle point at 50.

Since topological zones for superarcs in the interior forest do not intersect the boundary, they cannot connect to contours in other blocks. The invariant then follows that superarcs in the interior forest are guaranteed to be in the contour tree of the entire data set, while superarcs in the boundary tree are not. The supernode where a subtree of the interior attaches to the boundary tree is called an *attachment point*, and may or may not be represented explicitly in the boundary tree. For example, in Block 3 in Fig. 1, vertex 30 is an attachment point.

5 Hierarchical Contour Trees

Given boundary trees for each block, the union of the necessary manifolds is the union of all contours that intersect more than one block. We construct a topology graph by combining boundary trees, and compute the *shared contour tree* across all blocks. The contour tree for the entire dataset is then the shared contour tree plus the interior forests in each block. On each block, we compute the *hierarchical contour tree*, which represents the interior forest for that block plus the shared contour tree, thus distributing the storage of the contour tree and reducing communication cost during computation.

We follow Pascucci & Cole-McLaughlin [23] by applying this recursively, taking $\lg N_b$ iterations to combine N_b blocks, but removing interior forests at each stage of the fan-in. We note that the term *block* can refer to either the portion of M that corresponds to a single compute rank, or to the result of combining blocks hierarchically, and use $Hier(M)$ for the hierarchy of blocks that make up M.

For a given block B of data, define the *block manifold* $M(B) = B$ to be the block. For a higher level block that is the union of smaller blocks, $B_{ij} = B_i \bigcup B_j$, the block manifold $M(B_{ij}) = M_f(B_i) \bigcup M_f(B_j)$ is the union of the boundary tree manifolds of the smaller blocks, leaving out the interior forests. The shared contour tree $S_f(B_i j) = T_f(M(B_{ij}))$ is then the contour tree for the combined block with all remaining gluing boundaries represented, and the boundary tree $B_f(M(B_{ij}))$, interior forest $I_f(M(B_{ij}))$ and boundary tree manifold $M_f(M(B_{ij}))$ for B_{ij} are well defined. At the top level, no gluing boundary remains, so the boundary tree $B_f(M) = \emptyset$, and the interior forest $I_f(M) = S_f(B)$ is the shared contour tree.

The hierarchical contour tree for block B is then the union of the interior forests of all blocks in $Hier(M)$ that contain B, $T_f(B) = \bigcup I_f(B') : B' \in Hier(M), B \subseteq B'$. Neither shared contour tree nor hierarchical contour tree is canonical, as they depend on $Hier(M)$, not on M. The boundary tree for a given block however *is* canonical.

Not all contours in a parent block pass through B, but contours in M that are not represented in $H_f(B)$ do not intersect B. Moreover, if $B' \subset B$, then $H_f(B) \subset H_f(B')$: the hierarchical contour tree of a parent block is a subset of that for each of its children.

Finally, since every superarc in the contour tree is represented in the interior forest for at least one block, the union of all of the hierarchical contour trees is the contour tree of the entire mesh: $T_f(M) = \bigcap \{H_f(B) : B \in Hier(M)\}$.

Fig. 2 illustrates this construction for the dataset in Fig. 1. Here, Blocks 0 and 1 are combined to construct Block 01, and Blocks 01 and 23 to construct the full mesh. The left hand panel shows the fan-in that combines boundary trees to reach the shared contour tree, and the right hand panel shows the fan-out that constructs individual hierarchical contour trees for each block.

In the second row of Fig. 2, we see that combining boundary trees from Blocks 0 and 1 gives the topology graph shown, and that vertex 50 is correctly identified as the saddle for zone $50-79$. This zone is contained entirely in Block 01 even though it intersects both Blocks 0 and 1, which means it belongs to Block 01's interior forest, and can now be preserved for the fan-out phase.

Once the fan-in is complete (the bottom row), we have identified the only superarc shared between all four blocks, represented by the edge $8-86$, and note that the actual zone in the full contour tree, shown as Zones $7,6,5,4$ in Fig. 1 extends from $0-90$, but the top and bottom portions are represented in interior forests.

In the second row of the fan-in (right), superarc $50-79$ reattaches to the shared contour tree for Block 01 on the left, while superarc $8-86$ is extended up to 88 for Block 23 on the right.

Finally, in the top row, we reattach $50-100$ in Block 0, $79-80$ in Block 1, $71-81$ in Block 2, and both $20-30$ and $0-8$ in Block 3. We have now determined a unique hierarchical contour tree for each block, and can see that their union is the combined structure shown in the lower right of the figure, which is the correct contour tree for the entire mesh, as predicted.

6 Hybrid Parallel Algorithm

The distributed algorithm now follows from the definitions above, consisting of a fan-in to compute the shared contour tree, plus a fan-out to compute the hierarchical contour trees for each block.

To initialize the process, we compute the contour tree for each block. In each round, we separate the boundary tree and the interior forest for a block and retain the latter for the fan-out.

Computing a boundary tree in serial is easy: all boundary critical points are determined locally, and marked as immune to pruning. A queue is constructed with all leaves, and the tree is pruned until the boundary points are reached.

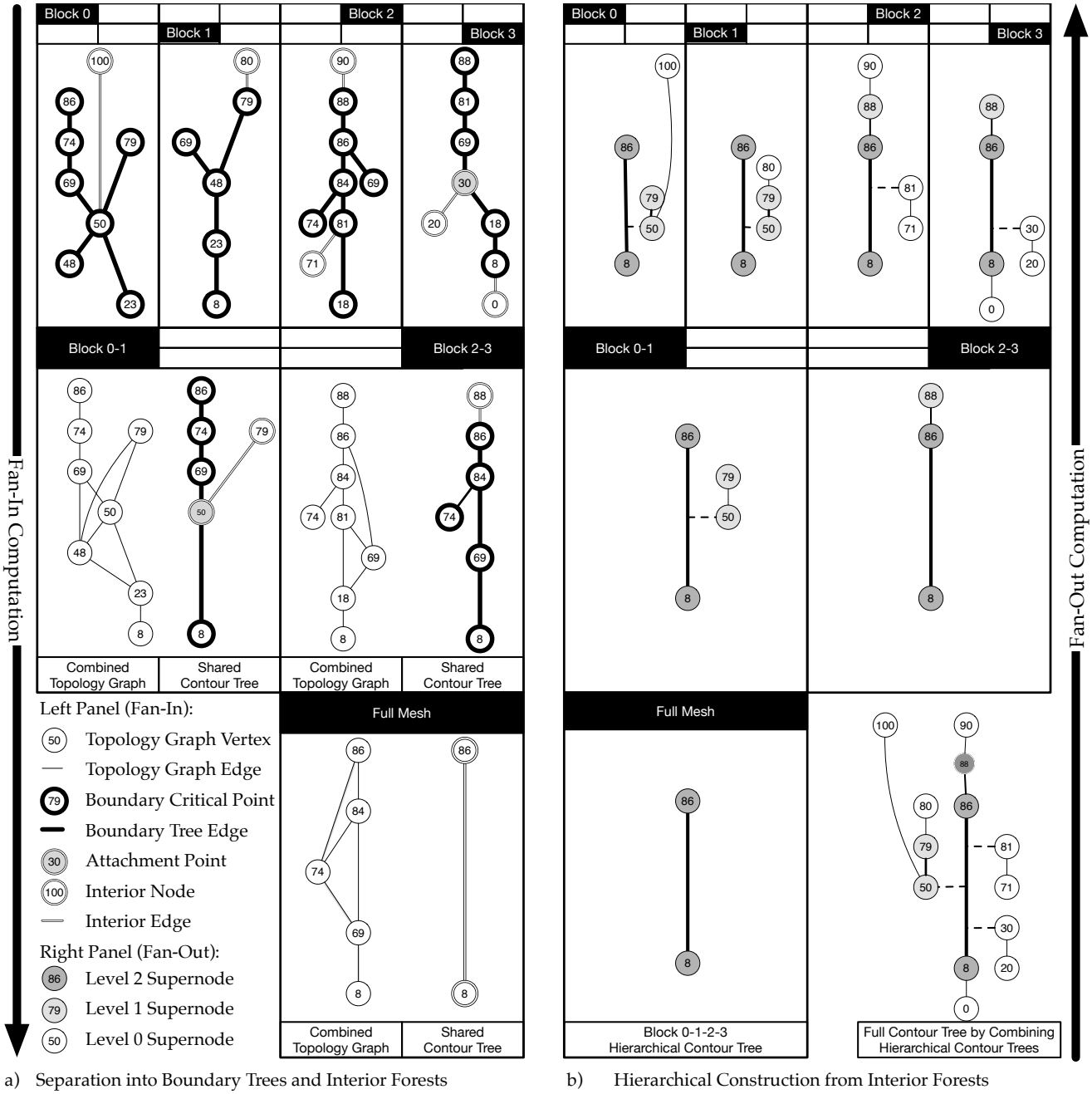
For shared-memory parallelism, we observe that the boundary tree is the closure under connectivity of the boundary points in the contour tree. For a superarc inside the boundary tree, there is thus at least one boundary point outside the superarc in each direction. In contrast, for superarcs in the interior forest, there are no boundary critical points in the subtree rooted at the superarc.

This summation is a modified subtree size computation, and can be performed efficiently in parallel with an Euler Tour or a hypersweep [16]. In practice, rather than split superarcs up, we define a superarc to be *necessary* for the boundary tree if none of the regular arcs on it are in the interior forest, and treat the supernodes at both ends as necessary as well.

The boundary tree is then exchanged with another rank which shares a boundary with it, and the two boundary trees are combined to get a topology graph from which the shared contour tree for the union of the two blocks is computed.

After a logarithmic number of such rounds, there is no remaining boundary, and the shared contour tree consists only of an interior tree. We initialise the hierarchical tree to the interior forest and fan back out, reinserting the interior forests in each round:

1. Compute a contour tree for each data block
2. For each block
   (a) For each of $O(\lg N)$ levels of fan-in:
      i. Split tree into *boundary tree* and *interior forest*
      ii. Exchange boundary tree with another block
      iii. Combine boundary trees into topology graph
      iv. Compute shared contour tree from topology graph
   (b) Set *hierarchical contour tree* to last combined tree
   (c) At each of $O(\lg N)$ levels of fan-out:
      i. Graft interior forest into hierarchical tree



a) Separation into Boundary Trees and Interior Forests

b) Hierarchical Construction from Interior Forests

Fig. 2: Computation of Distributed Hierarchical Contour Tree. During the fan-in (a), the contour tree for each block is separated into the boundary tree (dark) and interior forest (light). During the fan-out (b), the interior forests are inserted back in at each level. The lower right shows that the union of the individual hierarchical contour trees is the contour tree for the entire mesh.

7 Implementation Issues

In building the hierarchical contour tree several issues arise: boundary critical points, data structures, prefix IDs, partial superarcs, attachment points, augmentation and hierarchical storage.

Boundary Critical Points: While we can use every vertex on the gluing boundary to compute the contour tree, it is only necessary to include the critical points of the boundary [21, 23], and we have done so in the examples above. As we will see below, this has a significant impact on performance.

Data Structure: The choice of data structure depends on the algorithm for the contour tree: in our case the PPP algorithm [6] and its associated hyperstructure. Since the hyperstructure gathers superarcs in monotone chains iteratively, we treat each round of the fan-in as a separate layer of hyperstructure with its own iterations, preserving the ability to perform hypersweeps.

Prefix IDs: Since the parent's hyperstructure of a parent is a proper subset of a child's, ID numbers can be shared between blocks. We append any additional supernodes and superarcs to the arrays for the parent block, so the parent's hyperstructure becomes a prefix of every child's. This simplifies further computations using the hyperstructure as the ID numbers are implicitly shared between ranks rather than requiring explicit exchanges.

Partial Superarcs: Fig. 2 showed the example of superarc $50-80$ in the final contour tree, represented by $50-79$ and $50-80$. We modify the algorithm so that superarcs in the shared contour tree are preserved

as a whole rather than broken at boundary points. As a result, we would preserve $80-48$ in Block 0 rather than $79-48$.

Attachment Points: Some interior forests reattach at existing supernodes, as for example 50 in Block 0. Others, such as $20-30$ in Block 3, whose attachment point 30 can be carried forward in the boundary tree or not. If we carry them forward, we perform unnecessary communication between ranks, so we elected to omit them and reinsert them later by retaining their two adjacent boundary critical points (here, 18 and 69). On reinsertion, we search the hyperstructure between these two points as described in [6] to establish which superarc they insert onto.

Hierarchical Storage: The higher-level hierarchy can either be stored on a small number of nodes or distributed. We elected to swap boundary trees between pairs of ranks, and have both ranks compute the shared contour tree, as otherwise many nodes would lie idle. This means that, at the end of the fan-in phase, all of the information required to compute the hierarchical contour trees is already present on each rank, so no fan-out communication is needed, and all ranks can compute their own hierarchical contour tree directly.

Augmentation: Since the hierarchical contour tree builds on the hyperstructure, it is trivial to augment with all regular nodes on each block by performing the same parallel search as in PPP to locate the parent superarc for each regular node. Computing geometric measures is however less trivial, as the sorting order of insertion points along each parent superarc is not recorded. We defer this to a future discussion, as the details are complex.

8 Computational Complexity

Before we turn to empirical measures of performance, we can perform the usual asymptotic analysis of computational efficiency.

We test all boundary vertices for criticality, taking $O(N^2)$ work and $O(N)$ time in pathological meshes with N vertices, but $O(N)$ work and $O(1)$ time in a regular mesh, or as little as $O(b)$ work and $O(1)$ time if the boundary (of size b) is already known. Counting the number on each superarc then takes a single prefix sum operation taking $O(N\lg N)$ work and $O(\lg N)$ time to perform, or $O(t)+O(b)\lg b$ work and $O(1)$ time if the boundary set is known.

Computing the boundary critical point counts for the end of each superarc then takes a single Euler Tour ($O(t\lg t)$ work, $O(\lg t)$ time) or hypersweep (usually more efficient in practice). Testing them to identify necessary superarcs and interior forest superarcs takes $O(t)$ work, $O(1)$ time, followed by a round of pointer-doubling which costs $O(N\lg N)$ work and $O(\lg N)$ time (or $O((t+b)\lg(t+b))$ work and $O(\lg(t+b))$ time). For regular meshes (our principal concern at present), we can therefore compute this in $O((t+b)\lg(t+b)) < O(N\lg N)$ work and $O(\lg(t+b)) < O(\lg N)$ time overall.

The relationship between t, the size of the contour tree, b, the size of the boundary, b', the number of boundary critical points, and N_b, the size of the boundary tree, is key. In the absence of W-structures [15], $N_b < 2b' < 2b$, i.e. the complexity of the boundary tree is linear in the size of the boundary. For a 3D mesh, the contour tree would scale with the cube of the linear dimension, but the boundary tree with the square. For W-structures, however, it is possible for $N_b = \Omega(t)$, although we have never see this occur, and have reported typical statistics [15]. We assume that W-structures do not have a major impact on the size of the boundary tree.

During fan-in, we compute a contour tree and boundary tree on each rank, then perform r rounds combining boundary trees, computing new contour and boundary trees, grafting interior forests during r rounds of fan-out to produce the hierarchical contour tree.

We know [10] that computing the initial contour tree costs $O(N\lg N + t\lg t)$ work and $O(\lg N + (\lg t)^2)$ time. We saw above that the boundary tree can be computed in $O((t+b)\lg(t+b)) < O(N\lg N)$ work and $O(\lg(t+b)) < O(\lg N)$ time - i.e. the $O(N\lg N)$ work cost and $O((\lg t)^2)$ time cost are still dominant.

Thereafter, the cost in each round depends on N_b, the size of the boundary tree, which we have argued is bounded by the size b of the boundary in practice, and by b', the number of boundary critical points, except pathologically. For simplicity, we assume that $N_b = b$.

We exchange boundary trees with another block, generating a topology graph of size $2b$: this may require sorting operations, so we assume $O(2b\lg 2b)$ work and $O(\lg 2b)$ time. Computing the shared contour tree then takes $O(2b\lg 2b)$ work and $O((\lg 2b)^2)$ time, and reduction to a new boundary tree takes $O(2b\lg 2b)$ time and $O(\lg 2b)$ work, with the block boundary increasing at each stage by a factor of at most 2. After r rounds of this, the fan-in is complete, at a total cost of $O(2^r b\lg b)$ work and $O(\lg b)^2)$ time

During the fan-out, our search operations will typically be logarithmic in the size of the hierarchical contour tree (which we expect to be approximately rb), but we will have update cost to support search structures of $rb\lg b$, and r rounds of merge phase computation costing $O(b\lg b$ work and $O((\lg b)^2)$ time. Since all of our variables are bounded by N, the number of vertices in the input, we can simplify this to $rN\lg N$ work and $r\lg N + r(\lg t)^2$ time, but we accept that there is a lot of looseness in this.

8.1 Predictions

Based on the formal analysis, we expect to see the following:

1. The communication cost should be small relative to data size
2. The proportion should scale with $N^{2/3}$ for 3D, $N^{1/2}$ for 2D,
3. Good strong scaling is expected for large data, with local parallelism used effectively in early stages.
4. Strong scaling will fall off once the boundary trees rival the interior forests in size,
5. Communication costs will grow as larger trees transfer,
6. Weak scaling will be less effective, as boundary trees will get larger relative to interior forests.

9 Results

The key question we seek to answer now is the practical performance of our algorithm. We describe our implementation and experiment design (Sect. 9.1 - 9.2), then discuss strong scaling (Sect. 9.3) and weak scaling (Sect. 9.4) performance.

9.1 Implementation

The implementation of our algorithm utilizes the parallel peak pruning (PPP) algorithm [6] in the VTK-m library for on-node computation of contour trees. Based on PPP we have developed a new *ContourTreeDistributed* filter and collection of worklets for distributed computation of the hierarchical contour tree. Using VTK-m enables our algorithms to run on a broad range of parallel compute architectures, from multi-core CPUs using TBB or OpenMP to many-core GPUs using CUDA. For distributed computations we utilize the DIY [20] block-parallel library. Using DIY enables in-core and out-of-core execution and allows for improved flexibility with regard to data decomposition so that, e.g., one or more data blocks may be placed on each MPI rank. We have released our implementation as open source through the VTK-m library available at `https://gitlab.kitware.com/vtk/vtk-m`.

9.2 Experiment Design

In this evaluation we focus in particular on the distributed strong scaling (Sect. 9.3) and weak scaling Sect. 9.4) characteristics of our algorithms. The on-node scaling characteristics of PPP have been described in detail previously in [6].

Data: We chose a set of large-scale scientific datasets from a range of applications (Fig. 3). **GTOPO30** [12] is a global digital elevation model with a horizontal grid spacing of 30 arc seconds ($\approx 1km$) and a resolution of (21601×43201) pixel. **WarpX** is a laser-driven, plasma-based particle accelerator simulation dataset describing the electric field in the transverse direction (Ex) with a resolution of $(6791 \times 371 \times 371)$. **Nyx** is an astrophysics simulation dataset of particle mass density with a resolution of 1024^3. Finally, **MICrONS** is a 6800^3

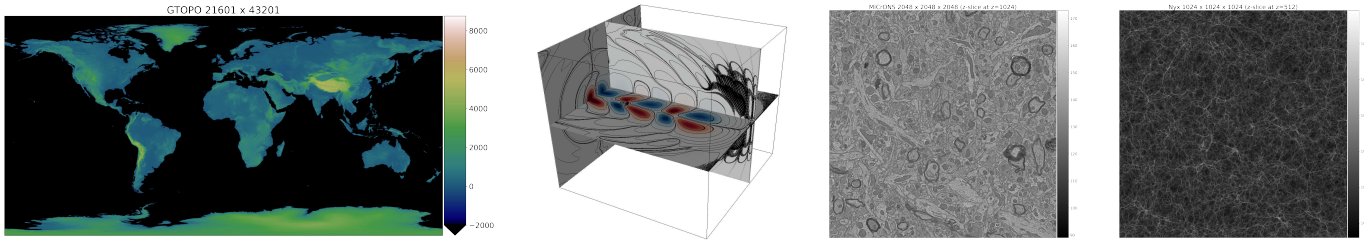


Fig. 3: Some Test Data Sets. On the left, GTOPO30 is 922M data points representing terrain altitude. Next, WarpX is a plasma physics simulation with complex isosurfaces but relatively clean topology. Both Microns (electron microscopy) and Nyx (cosmological simulation) have complex topology and massive contour trees.

subvolume of the minnie65 Electron Microscopy (EM) image dataset of a $1.4 \times 0.87 \times 0.84 mm^3$ volume of a neural cortex in a P60 mouse from the Machine Intelligence from Cortical Networks (MICrONS) program [19], available via BossDB [28]. In the scaling studies we select varying subvolumes of 512^3 to $2048 \times 2048 \times 4096$ from the MICrONS dataset.

Architecture: For evaluation, we use the Haswell partition of the Cray XC40 **Cori** compute system at the National Energy Research Scientific Computing Center, consisting of 2,388 nodes connected via a high-speed "Dragonfly" interconnect. Each compute node has two 2.3 GHz 16-core Intel Xeon E5-2698 v3 ('Haswell') processors and 128 GB DDR4 2133 MHz memory. Each core supports two hyperthreads and has two 256-bit-wide vector units, supporting 32 physical threads and 64 hyperthreads.

For parallel scaling with GPUs we use the **Summit** supercomputer at the Oak Ridge Leadership Computing Facility, consisting of 4,608 compute nodes—each equipped with 2 IBM POWER9 CPUs and 6 NVIDIA Tesla V100 GPUs with 608 GB of memory (96 GB HBM2 + 512 GB DDR4)—connected in a non-blocking fat-tree using a dual-rail Mellanox EDR InfiniBand interconnect.

Measuring Runtime: The implementation is divided into three main phases: 1) local computation of contour trees, 2) fan-in, and 3) fan-out. At the end of each phase the ranks synchronize in order to begin the next phase. To determine the overall contribution of each main algorithm phase to the total time we look at the maximum time across ranks in each phase. We then examine load imbalances between ranks by looking at the wait times required for synchronization at the end of each phase. Communication between ranks occurs only during fan-in: we estimate communication cost by measuring the compute time within the individual fan-in steps as well as the total compute time for the fan-in. The difference between these then provides an estimate of the time required by DIY for communication and coordination between ranks.

9.3 Strong Scaling

First, we evaluated distributed scalability for constant problem size with increasing number of processors for five datasets on Cori (Fig. 4, a-e). To evaluate our resource usage, we varied the number of ranks per compute node and show performance for a range of configurations (inset matrix plots). The available threads on each node are distributed evenly across the local ranks, e.g., at 256 nodes we are using 16,384 threads across up to 2,048 ranks.

Scaling of the main compute phases: The local contour tree computation phase does not require any coordination between ranks, so this phase shows close to ideal scaling relative to the reduction in workload per rank with increasing number of processors (Fig. 4, a-e, blue bars). In contrast, the runtime for the fan-in phase (green) is similar across scales in the strong scaling scenario. While the fan-in is a parallel compute phase, the work we need to complete in this phase increases proportionately as we add more ranks, requiring more iterations to complete the fan-in as well as increasing the size of the boundary between data blocks as the data is divided into increasingly smaller blocks.

Fig. 5 shows the sizes of the boundary tree during fan-in for WarpX, illustrating the increase in workload for the fan-in with increasing number of ranks. The fan-in, hence, behaves here like a weak-scaling experiment where we double workload as we double number of ranks. The fact that the fan-in time remains roughly constant, hence, indicates that the implementation itself indeed scales quite well with growing number of ranks. The fan-out then behaves similar to the fan-in with the key difference that the fan-out is a strictly local computation without interprocess communication. The strong scaling is, hence, characterized by reductions in workload during the local contour tree computation with scaling being limited primarily by the time required for fan-in/out.

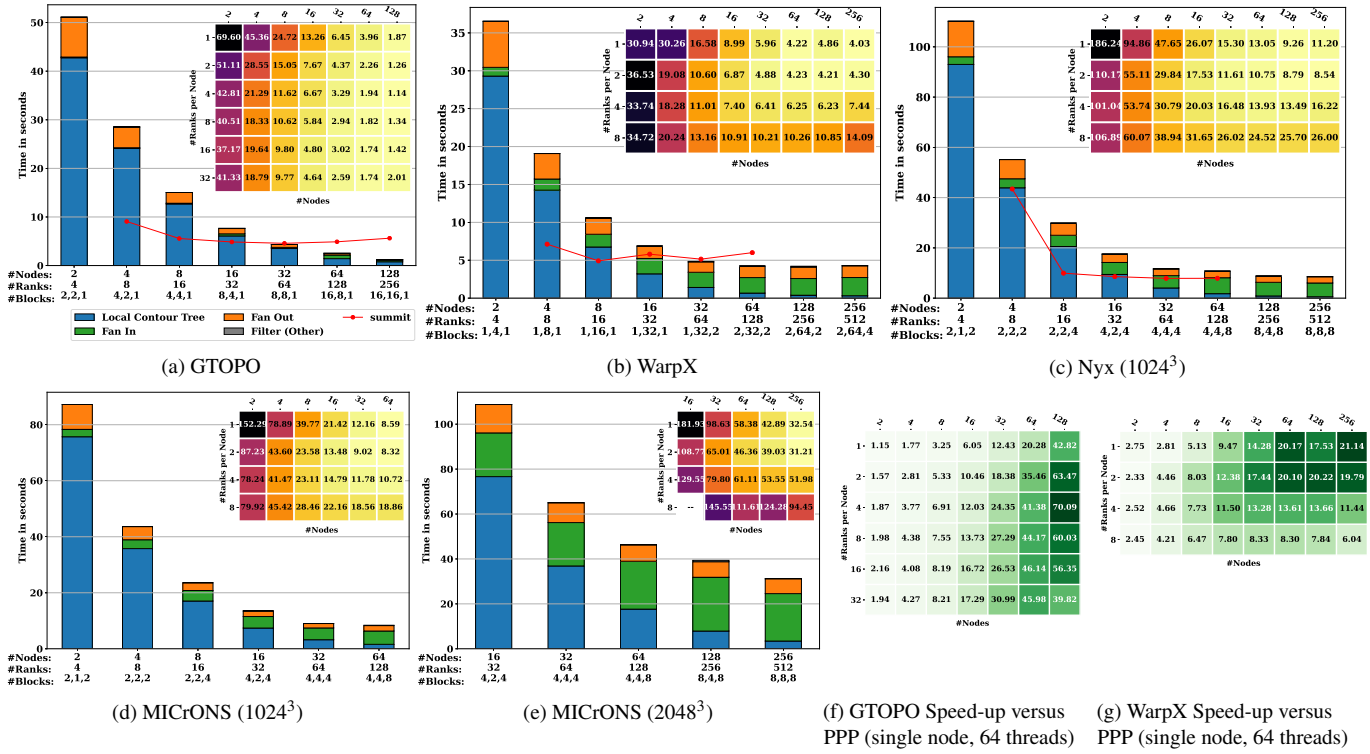
Speed up: Overall, we observe that the algorithm shows good strong scaling with speed-ups dropping off eventually due to the workload on each node becoming too small. Compared to the single-node PPP algorithm with threading we observe maximum speed-ups of $\approx 70\times$ for GTOPO and $\approx 21\times$ for WarpX (Fig. 4, f-g) and up to $\approx 700\times$ and $\approx 280\times$, respectively compared to serial PPP. For the 2D GTOPO dataset, the fan-in/out phases contribute significantly less to the overall compute time than for the 3D datasets. This is likely due to the much smaller relative boundary-size in 2D compared to 3D, which in turn leads to improved parallel scaling in 2D. The Nyx and MICrONS datasets are too large to be processed on a single Cori node so that we can only evaluate relative speed-up compared to the smallest number of nodes required to process the given dataset.

Best node configuration: With regard to on-node configurations we observe that in general 2 ranks per node (i.e., 1 rank per processor) is a good configuration for Cori. However, for small numbers of nodes using more ranks per node can help improve performance as at that scale overall runtime is dominated by the local tree computations so that runtime gains due to the reduction in workload per ranks outpaces cost for the fan-in/out.

Impact of Boundary on Scaling: By using only boundary critical points in the fan-in, we see significant reductions in boundary tree sizes (Fig. 5) and corresponding speed-ups in particular for larger number of ranks. This is due to the fact that the size of the boundary relative to the full data is increasing as we split the data into more blocks. For WarpX and GTOPO we observe speed-ups of up to $\approx 2.8\times$ when using only boundary critical points vs. using the full block boundary. Further, using the full boundary of blocks increases not just the compute time of all phases but it also exasperates imbalance in wait times during the fan-in. We completed the same experiment also for GTOPO and with OpenMP which showed similar overall effects.

Scaling with Different Device Backends: So far we have discussed scaling using TBB for on-node computation. We repeated all experiments using VTK-m's OpenMP backend. The overall scaling behavior for TBB and OpenMP are very similar with TBB being typically between $1.06\times$ to $1.96\times$ faster compared to OpenMP.

We also performed initial scaling runs on Summit using CUDA (Fig. 4, a-c, red curves). Consistent with the scaling studies presented in [6], we observe $\approx 3\times$ speed-ups for the local tree computations using GPUs compared to using threading on CPUs. However, when using GPUs, cost for moving data between the device and host are a key cost factor. As such, we observe significant slowdown at low numbers of ranks, when individual data blocks are too large to fit the required data structures into GPU memory. When using large numbers of ranks with CUDA, the fan-in/out phases then become dominant factors in the



(a) GTOPO (b) WarpX (c) Nyx (1024^3)

(d) MICrONS (1024^3) (e) MICrONS (2048^3) (f) GTOPO Speed-up versus PPP (single node, 64 threads) (g) WarpX Speed-up versus PPP (single node, 64 threads)

Fig. 4: Strong scaling on Cori Haswell using TBB. **(a-e)** The bar charts show the breakdown of runtime into the 3 main compute phases for runs with 2 ranks per node (i.e., 1 rank per CPU). The inset plot shows the runtimes for all evaluated node/rank configurations. **(a-c, red curve)** Runtime on Summit using the same number of ranks as on Cori with 1 GPU per rank and using 4 ranks/GPUs per compute node. **(f-g)** Speed-up compared to the single-node, threaded contour tree implementation.

total runtime, with memory movement due to the need for inter-rank communication being the likely cause. In the future, this suggest that hybrid models using CUDA for on-node tree computations and using CPU threading for fan-in/out could help improve scaling when using larger numbers of ranks.

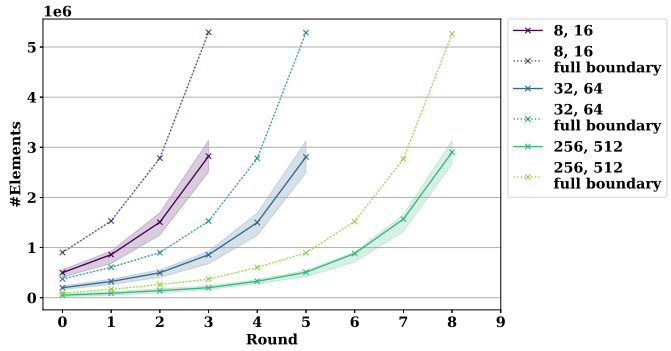


Fig. 5: Comparison of boundary tree sizes during fan-in for WarpX using 8, 32, and 256 nodes (2 ranks per node) using the full boundary (dashed) and using only critical boundary (solid).

9.4 Weak Scaling

For this experiment we begin by selecting a subvolume of 512^3 from the MICrONS dataset. As we increase the number of compute nodes we simultaneously grow the data selection accordingly so that independent of the number of nodes, each compute node is assigned a 512^3 subvolume. This dataset is particularly challenging as each new block adds new topological structures (neurons in the brain) which are in turn connected with structures in neighboring blocks (via axons and dendrites). This use-case models the scenario where increase in data size is driven by expansion of the observable space.

In Fig. 7, the local contour tree compute (blue) remains roughly constant (i.e., perfect weak scaling). This is expected as the computation of the local contour trees is independent across ranks and the data blocks exhibit similar topological complexity.

The fan-in phase (green) and to a lesser degree also the fan-out phase (orange) then become increasingly expensive. While each node is responsible for a 512^3 subvolume, Fig. 8 shows that in addition to more iterations in the fan-in, we increase the amount of data and work in subsequent rounds. For the fan-out, the majority of the work is in round 0 with the work per round growing much slower than for the fan-in, explaining the more modest growth in compute time for the fan-out. The growth in compute time we observe in Fig. 7 is, hence, a reflection of the growth in the amount of work required to assemble the contour tree for the data, rather than describing a decrease in efficiency of the algorithm itself.

Using the GTOPO and WarpX datasets, we next modeled the use case where increase in data size is driven by increasing resolution to more accurately resolve physical processes and with it topology. We observed similar increases in compute time during the fan-in/Out due to increases in workload to resolve the global structures. For WarpX the local tree computation times remain stable whereas for GTOPO the local tree computation times grow with increasing data resolution due to the increased topological complexity within the individual tiles as more local topographic structures are being resolved.

9.5 Comparison with Other Approaches

Ideally, we would have liked to compare against the parallel contour tree algorithm of Pascucci and Cole-McLaughlin [23], but we did not have access to that code. We did however, implement their algorithm in our own framework and ran some tests against it at an earlier stage

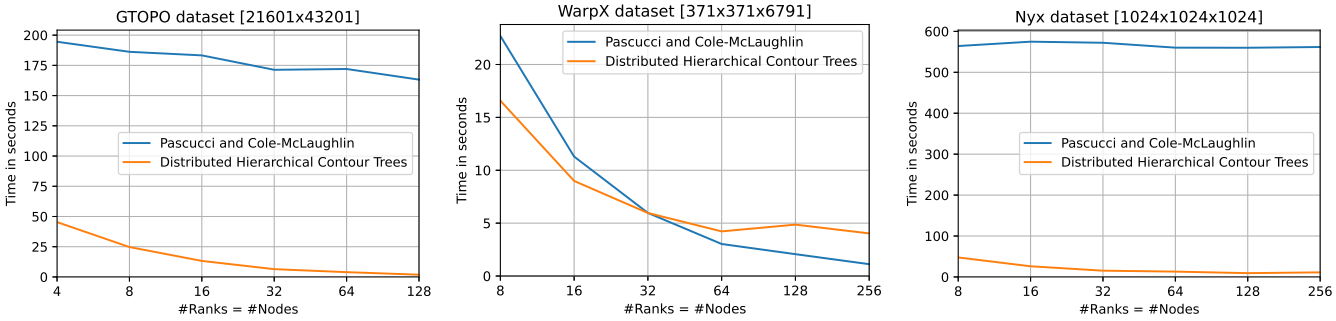


Fig. 6: Scaling results for a variation of the distributed parallel algorithm, by Pascucci and Cole-McLaughlin [23]

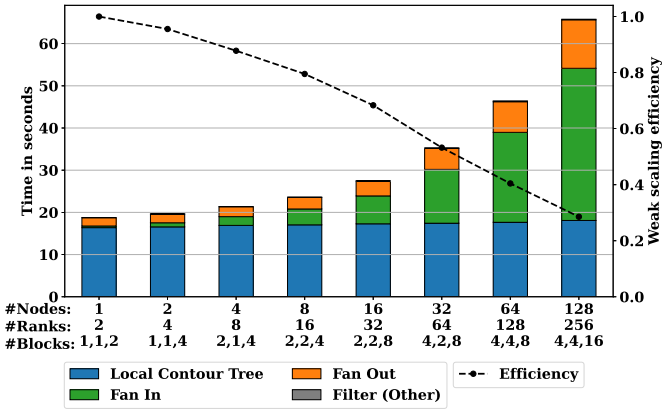
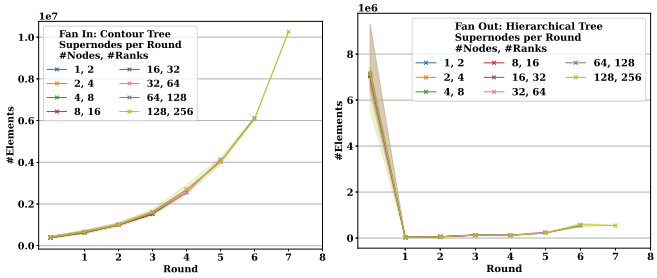


Fig. 7: Runtime for the hierarchical contour tree with growing number of compute resources (nodes) and data size, while each node is assigned a 512^3 subvolume (weak scaling) with 2 MPI ranks per node (i.e., each rank owns a $512x512x256$ block) .



(a) fan-in tree sizes (b) fan-out tree sizes

Fig. 8: Median (line) and min/max (area) number of supernodes per round and rank in the **(a)** fan-in and **(b)** fan-out.

of our project, as shown in Fig. 6.

As described by Pascucci and Cole-McLaughlin, we subdivide the data set into blocks, compute the contour tree in each block, and combine the resulting trees. We note that their original implementation used divide-and-conquer inside each block, but we substituted PPP [10] to take advantage of the degree of local parallelism available. Like Pascucci and Cole-McLaughlin, we only consider extrema on the boundary when gluing trees together and eliminate all other vertices, thus reducing communication cost and computational effort for combining trees.

We also note that our implementation of this algorithm used our original MPI+OpenMP parallel peak pruning implementation and *not* its VTK-m implementation (unlike the other scaling studies in this paper). Moreover, as with their original paper, we only computed the unaugmented contour tree, not the fully augmented contour tree in which every regular node is assigned to a superarc.

We found that their method performed very well on WarpX, which has a comparatively simple topological structure. However, as contour tree complexity increased for GTOPO30 and Nyx, the scaling behavior of this previous algorithm deteriorated badly, confirming the logic behind our attempt to minimize data transfer between ranks.

Since approaches such as distributed merge trees [21] only compute the merge trees, we did not perform systematic tests against them, as we compute the full contour tree, which is rather more complex, especially in the presence of W structures. We did however perform some preliminary tests against the publicly available implementation (`https://github.com/mrzv/reeber`), since the original multi-threaded implementation using skip lists is not available. In these tests, computing the merge tree alone for the WarpX data set took 6 minutes 48 seconds on 8 MPI ranks, which is significantly slower than our approach. This is probably because the implementation does not use on-node parallelism.

We also did not compare against shared memory algorithms, as this paper is primarily about the distributed computation, and is on principle compatible with any SMP algorithm on node.

10 Conclusions & Future Work

We have introduced and implemented a distributed algorithm for computing a hierarchical contour tree with good scaling efficiency and significantly improved performance over the existing state of the art. The task is not yet finished, as effective use of the contour tree for analytic purposes requires further computations, such as geometric measures and branch decompositions. We expect to publish further results on these tasks in future, together with application studies of contour tree analysis at scale.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration under Contract No. DE-AC02-05CH11231 to the Lawrence Berkeley National Laboratory and subcontract 7452335 to the University of Leeds. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. We thank Jean-Luc Vay and Maxence Thevenet for making the WarpX dataset available to us.

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, Apr. 2015. doi: 10.1109/PACIFICVIS.2015.7156387

[2] C. L. Bajaj, V. Pascucci, and D. R. Schikore. The Contour Spectrum. In *Proceedings of Visualization 1997*, pp. 167–173, 1997.

[3] T. F. Banchoff. Critical Points and Curvature for Embedded Polyhedra. *Journal of Differential Geometry*, 1:245–256, 1967.

[4] G. Blelloch. *Vector Models for Data-Parallel Computing*. PhD thesis, MIT, 1990.

[5] R. L. Boyell and H. Ruston. Hybrid Techniques for Real-time Radar Simulation. In *Proceedings, 1963 Fall Joint Computer Conference*, pp. 445–458. IEEE, 1963.

[6] 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*, March 2021. doi: 10.1109/TVCG.2021.3064385

[7] 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. 59–74. Springer, 2009.

[8] H. Carr, J. Snoeyink, and M. van de Panne. Simplifying Flexible Isosurfaces with Local Geometric Measures. In *Proceedings of Visualization 2004*, pp. 497–504, 2004.

[9] H. Carr, J. Tierny, and G. H. Weber. Pathological and Test Cases For Reeb Analysis. In H. Carr, I. Fujishiro, F. Sadlo, and S. Takahashi, eds., *Topological Methods in Data Analysis and Visualization V: Theory, Algorithms, and Applications*, Mathematics and Visualization, pp. 103–120. Springer International Publishing, Dec. 2020. doi: 10.1007/978-3-030-43036-8_7

[10] H. Carr, G. H. Weber, C. Sewell, O. Rübel, P. Fasel, and J. Ahrens. Scalable contour tree computation by data parallel peak pruning. *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–1, November 2019. doi: 10.1109/TVCG.2019.2948616

[11] Y.-J. Chiang, T. Lenz, X. Lu, and G. Rote. Simple and Optimal Output-Sensitive Construction of Contour Trees Using Monotone Paths. *Computational Geometry: Theory and Applications*, 30:165–195, 2005.

[12] Earth Resources Observation and Science Center, U.S. Geological Survey, U.S. Department of the Interior. USGS 30 Arc-second Global Elevation Data, GTOPO30, 1997.

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

[14] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Task-based Augmented Merge Trees with Fibonacci Heaps. In *7th IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, 2017.

[15] 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 International Publishing, 2021.

[16] 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, Oct. 2020. doi: 10.1109/LDAV51489.2020.00008

[17] P. Klacansky, A. Gyulassy, P.-T. Bremer, and V. Pascucci. Toward localized topological data structures: Querying the forest for the tree. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):173–183, 2020. doi: 10.1109/TVCG.2019.2934257

[18] 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.

[19] MICrONS 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. *bioRxiv*, 2021. doi: 10.1101/2021.07.28.454025

[20] D. Morozov and T. Peterka. Block-parallel data analysis with diy2. In *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 29–36. IEEE, 2016.

[21] D. Morozov and G. Weber. Distributed Merge Trees. *ACM SIGPLAN Notices*, 48(8):93–102, 2013.

[22] D. Morozov and G. Weber. Distributed Contour Trees. In P.-T. Bremer, I. Hotz, V. Pascucci, and R. Peikert, eds., *Topological Methods in Data Analysis and Visualization III*, Mathematics and Visualization, pp. 89–102. Springer, 2014.

[23] V. Pascucci and K. Cole-McLaughlin. Parallel Computation of the Topology of Level Sets. *Algorithmica*, 38(2):249–268, 2003.

[24] V. Pascucci, K. Cole-McLaughlin, and G. Scorzell. Multi-resolution computation and presentation of contour trees. In *Proceedings of the IASTED conference on Visualization, Imaging and Image Processing (VIIP 2004)*, pp. 452–290, 2004.

[25] 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.

[26] D. Smirnov and D. Morozov. Triplet merge trees. In *Topological Methods in Data Analysis and Visualization V*, Mathematics and Visualization. Springer, 2017.

[27] 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, 13th ACM Symposium on Computational Geometry*, pp. 212–220, 1997.

[28] 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`.