Scalable Contour Tree Computation by Data Parallel Peak Pruning

Hamish A. Carr, Gunther H. Weber, Christopher M. Sewell, Oliver Rübel, Patricia Fasel, James P. Ahrens

Abstract— As data sets grow to exascale, automated data analysis and visualization are increasingly important, to intermediate human understanding and to reduce demands on disk storage via *in situ* analysis. Trends in architecture of high performance computing systems necessitate analysis algorithms to make effective use of combinations of massively multicore and distributed systems. One of the principal analytic tools is the contour tree, which analyses relationships between contours to identify features of more than local importance. Unfortunately, the predominant algorithms for computing the contour tree are explicitly serial, and founded on serial metaphors, which has limited the scalability of this form of analysis. While there is some work on distributed contour tree computation, and separately on hybrid GPU-CPU computation, there is no efficient algorithm with strong formal guarantees on performance allied with fast practical performance. We report the first shared SMP algorithm for fully parallel contour tree computation, with formal guarantees of $O(\lg V \lg t)$ parallel steps and $O(V \lg V)$ work for data with V samples and t contour tree supernodes, and implementations with more than $30\times$ parallel speed up on both CPU using TBB and GPU using Thrust and up $70\times$ speed up compared to the serial sweep and merge algorithm.

Index Terms—topological analysis, contour tree, merge tree, data parallel algorithms

1 Introduction

Modern computational science and engineering depend heavily on ever-larger simulations of physical phenomena. These simulations are a major driver for hardware advances, and have led to clusters with hundreds of thousands of cores, petaflops of performance and petabytes of data storage, with exaflop performance predictable within the next seven to nine years. For recent hardware, the I/O cost of data storage and movement dominates, and increasingly requires *in situ* data analysis and visualization, making algorithms which identify key features such as contours during the simulation and store only features to disk rather than the full data more appealing.

In situ analysis and visualization require analytic tools to identify relevant features for further analysis and/or output to disk. This has stimulated research into areas such as computational topology, which constructs models of the mathematical structure of the data for analysis and visualization. One of the principal mathematical tools is the *contour tree* or *Reeb graph*, which summarizes the development of contours in the data set as the isovalue varies. Since contours occur in many visualizations, the contour tree and the related *merge tree* are of prime interest in automated analysis of massive data sets.

The value of these computations has been limited by available algorithms, and the goal of this paper is to give a data parallel, shared memory algorithm for contour tree computation. This goal is motivated by the increased per-node parallelism of modern computing architectures. Multi-core accelerator boards, such as NVIDIA GPU and Intel Xeon Phi increasingly provide data parallel compute power to personal work stations as well as supercomputers like Titan at the Oak Ridge Leadership Facility (NVIDIA Kepler), Trinity at the Los Alamos National Laboratory & Sandia National Laboratories (Intel Xeon Phi) and Cori at the National Energy Research Scientific Computing Center (Intel Xeon Phi). Furthermore, high performance computing already uses hybrid shared-memory/distributed memory architectures with 16 or more cores per compute node with high-speed interconnect. Finally, machines like Silicon Graphics UV racks make it feasible to have up to 512 processors and 4TB of RAM in a shared memory space, using OpenMP as the programming paradigm, while NVIDIA's Tesla V100 cards have up to 5120 cores and 16GB of VRAM. Problem sizes of $> 1,000,000,000$ data values however mean that each core can be expected to process many vertices. The effective model of parallelism is thus CRCW—concurrent read, concurrent write, with no guarantees on write order. Moreover, although parallel algorithmic efficiency is crucial, the total work to be performed also matters.

These factors make efficient data parallel contour tree algorithms desirable, but existing approaches are largely serial or distributed. While there is a well-established algorithm [9] for merge trees and contour trees, the picture is patchier for distributed and data-parallel algorithms. Although some approaches exist, they either target a distributed model [1], or have serial sections [27], do not come with strong formal guarantees on performance, or do not report methods for augmenting the contour tree with regular vertices, which is required for secondary computations such as geometric measures [10].

Developing a new algorithm for data parallel contour tree calculation requires reformulating the problem for parallelism. Our new approach builds on the two phases of Carr et al. [9] of computing merge trees (join & split tree) and combining them into a contour tree. To parallelize merge tree calculation, we replace the union-find based approach with a new algorithm that constructs monotone paths from saddles to extrema then iteratively "prunes" peaks, i.e., cuts off merge tree branches ending in an extremum (Section 4). Many extrema can be "pruned" simultaneously, making this approach easily parallelizable.

Once join and split trees are computed, we combine them into the contour tree. While the original algorithm [9] uses priority queues to serialize transferring arcs from join and split tree into the contour tree, these operations are not inherently serial, and, with some extensions to the algorithm, we can perform them in parallel.

While this approach is inherently parallel, a naïve implementation is slow, and we spent considerable effort in optimizing not only the formal analysis but the practical efficiency. As a result, while it is possible to compute the fully augmented contour tree with our algorithm by treating every node as a critical point, we concentrate on computing the unaugmented contour tree in this paper for reasons of speed.

We review previous work (Section 2), then add some new terminology (Section 3). Since the merge trees are a precursor to contour trees, we report our *parallel peak-pruning (PPP)* algorithm for merge trees first in a simple naïve form (Section 4), then in a developed form with a number of significant optimizations (Section 5). We then extend the approach to 3D simplicial meshes and cubic (MC) meshes in Section 7, and describe a parallel algorithm for contour tree computation from merge trees (Section 6), finishing with performance results (Section 8)

- *Hamish A. Carr is with the University of Leeds, United Kingdom. E-mail: H.Carr@leeds.ac.uk.*
- *Gunther H. Weber and Oliver Rübel are with the Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, California, United States. E-mail: {GHWeber, ORuebel}@lbl.gov.*
- *Gunther H. Weber is with the Department of Computer Science, University of California, Davis, California, United States.*
- *Christopher M. Sewell, Patricia Fasel, and James P. Ahrens are with the Los Alamos National Laboratory, Los Alamos, New Mexico, United States. E-mail: {csewell,pkf,ahrens}@lanl.gov.*

Mansucript received 24 July 2018; revised 12 February 2019; revised 10 June 2019.

and our conclusions (Section 9).

2 Background

Since the goal of this work is to use data-parallel computation to construct an algorithm for contour tree computation, we split relevant prior work between data-parallel computation (Section 2.1) and contour tree computation (Section 2.2). This divide is not strict, since some work has been published on distributed and parallel contour tree computation, but is convenient for the sake of clarity.

2.1 Data-Parallel Computation

Data-parallelism exploits the shared-memory parallelism on accelerators such as GPUs and multi-core CPUs. Blelloch [3] defined a scan vector model and showed that many algorithms can be implemented using a small set of "primitives"—such as transform, reduce, and scan—which can be implemented in a constant or logarithmic number of parallel steps. NVIDIA's open-source Thrust library provides an STL-like interface for such primitive operators, with backends for CUDA, OpenMP, Intel TBB, and serial STL. An algorithm written using this model can utilize this abstraction to run portably across all supported multi-core and many-core backends, with the architecture-specific optimizations isolated to the implementations of the data-parallel primitives in the backends.

PISTON [25] and VTK-m use Thrust for algorithms such as isosurfaces, cut surfaces, thresholds, Kd-trees [36] and halo finders [20]. Halo finding [20, 35] makes use of a data-parallel union-find algorithm, which most contour tree algorithms depend on.

We will also rely on a technique known as "pointer-jumping," which is used to find the root of each node in a forest of directed trees [23]. In this approach, the successor for each node is initialized to be its parent: thereafter, the successor of the node is updated to the successor's successor in each iteration. After at most logarithmic iterations, all vertices are guaranteed to point to the root of their forest.

2.2 Contour Trees

Given a function $f : \mathbb{R}^d \to \mathbb{R}$, a *level set*—usually termed *isosurface* in scientific visualization—is the inverse image $f^{-1}(h)$ of an *isovalue* h, and a *contour* is a single connected component of a level set. The *Reeb graph* is obtained by contracting each contour to a single point [34], and is well defined for Euclidean spaces or for general manifolds. The number and, in three dimensions, the genus of contours changes only at isolated *critical points*. Critical points where the number of contours changes appear as nodes in the Reeb graph, while critical points where only the genus changes do not. For simple domains, the graph is guaranteed to be a tree, and is called the *contour tree*.

The structure of the contour tree emphasizes the critical points at which topological change occurs: these are called *supernodes*, and the edges between them *superarcs*. Note that while all supernodes are critical points, the reverse is not true, as in the case of critical points where the genus changes but the number of connected components does not. For clarity, critical point should only be used to refer to a point in the original domain, while supernode should be used to refer to the corresponding vertex in the contour tree.

Strictly speaking, the supernode corresponds not to the critical point itself, but rather to the *critical contour*, i.e. the contour passing through the critical point. Each superarc is then the class of contours lying between the critical contours for the supernodes at the ends of the superarc. A single contour at isovalue h is then represented by the point on the corresponding superarc with the same isovalue h. For many purposes, the contour tree is then *augmented* with additional nodes along the superarc, most commonly the vertices in the original mesh. This is significant for data analysis and visualization, as it permits annotation of the tree with additional geometric information.

The contour tree abstracts isosurface behavior, as seen in Figure 1. By contracting contours to single points, it indexes all possible contours. If the contour tree is laid out so that the y-coordinates correspond to function value (Figure 1), a horizontal cut intersects one edge of the contour tree per connected isosurface component at the corresponding isovalue. We show three such cuts: orange at 6.5 (two contours), cyan at 11.5 (three contours) and blue at 21.25 (4 contours). This property was exploited in one of the early visualization applications: accelerated extraction by generating seed cells for isosurface extraction by contour following [39, 40].

As well as relating contours and critical points, contour trees also allow assigning importance to features [10] and ignoring features below an importance threshold. Features are defined by pairs of critical points, usually an extremum-saddle pair. The most common pairing is the *branch decomposition* of Pascucci et al. [32], which is very similar to the *topological persistence* [15] used in persistent homology, although only identical [22] for join and split trees, not for contour trees. We therefore use the language of branch decomposition rather than topological persistence.

We illustrate a small data set, the corresponding join, split and contour trees, and the branch decomposition in Figure 1. At saddle 15, peaks 24 and 22 meet, and we pair one with 15: the choice is based on isovalues or geometric properties [10]. Here, peak 24 has a higher value ($24-15=9$) than peak 22 does ($22-15=7$), so 22 pairs with 15 and is subordinate to peak 24. 24 is now a single peak with saddle at 10: this pair has a wider range of values than peak 13 and saddle 10 ($13-10<24-10$), so peak 24 is paired with the minimum at 0.

This process, applied to all critical points, results in the hierarchical *branch decomposition* shown in Figure 1(d). Simplification of the tree then consists of cancelling the extremum with the saddle, e.g. by "flattening" it, chopping off the peak or "filling in" the valley.

For data analysis, we assume the domain is a mesh—i.e., a tessellated subvolume of $\mathbb{R}^d$, as used for numerical simulation. For simplicial meshes, all critical points of the function are guaranteed to be at vertices of the mesh [2], simplifying topological computations.

We will refer to the number of vertices in a graph as V and the number of edges as E, and note that pathological tetrahedral meshes may have $E = \Theta(V^2)$. For regular meshes (our principal targets at present), $V = \Theta(E)$. In all practical cases, however, $V < E$.

2.2.1 Sweep And Merge Algorithm for Contour Trees

For simplicial meshes on simple domains, the *sweep and merge* algorithm [9] incrementally adds vertices in sorted order to a union-find data structure [38]. As components are created or merged in the union-find, critical points are identified, and a partial contour tree is created, called a merge tree. After performing ascending and descending sweeps, the two resultant merge trees, known as the *join* and *split* trees are combined to produce the contour tree. The conference and journal versions flipped the meaning of "join" and "split", which led to some confusion. We will follow the journal version, and use "join" for a saddle where peaks meet and "split" for a saddle where pits meet.

2.2.2 Topology Graph

For a simplicial mesh, the contour tree is normally computed by taking the edges of a triangulated mesh as the input to a graph-based algorithm (see below for details). However, while this is a sufficient input, it is not necessary, and may cause unnecessary workload. Carr & Snoeyink [8] abstracted this to a *topology graph*, in which all critical points must be represented, along with a set of edges that can represent any critical path through the underlying scalar field. Moreover, one can use separate topology graphs to compute the join and split trees, in which case we may refer to them as join and split graphs. This approach is also visible in other algorithms [37, 13, 27], and is essential to the performance of our new approach.

2.2.3 Scaling Sweep and Merge

While the sweep and merge algorithm is simple and efficient, it uses a sequential sweep through the contours, hindering the development of parallel algorithms. Pascucci & Cole-McLaughlin [31] described a distributed method that divides the data into spatial blocks, computes the contour tree separately for each block and combines the contour trees of individual blocks in a fan-in process combined until a single master node holds the entire contour tree.

Similarly, Acharya & Natarajan [1] computed the contour tree by splitting the data into blocks and combining the resulting local trees.

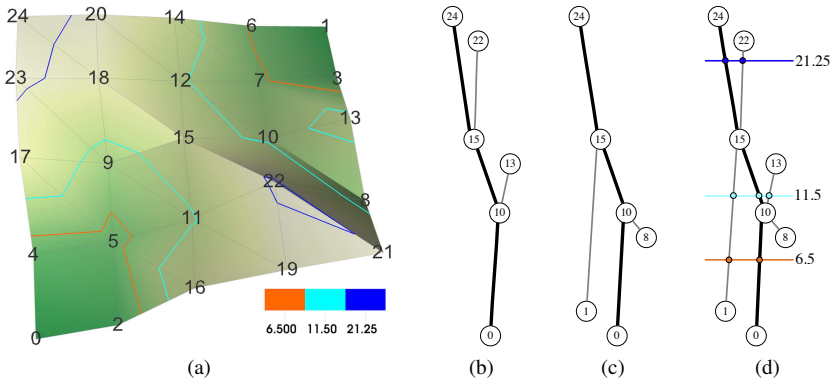


Fig. 1: (a) shows a small triangulated mesh as a landscape, with some selected isolines shown. (b) shows the corresponding join tree, which captures the connectivity of the super-level sets (regions above a given value). Cancellation pairs under topological persistence are indicated by line color and width, and are equivalent to branch decomposition. (c) shows the corresponding split tree with cancellation pairs. (d) shows the contour tree, which captures the connectivity of the contours in mesh (a) - for example, the isolines shown in (a) are marked as augmenting nodes in the tree. The branch decomposition of the contour tree, which provides a hierarchy for data simplification, is indicated by line color and width, but is not always identical to cancellation under topological persistence.

Within each block, their algorithm identifies critical points, and constructs monotone paths from saddles to extrema to build topology graphs, following Chiang et al. [13]. Once this is done, they stitch together the join & split trees for the blocks, to produce join & split graphs for computing the global contour tree.

In practice, contour trees have a significant memory footprint, and, for noisy or complex data set, their size is nearly linear in input size, which forces the contour tree for the entire data set to reside on the master node, defeating one of the purposes of parallelization: distribution of cost both in computation and in storage.

More recently, Morozov & Weber [29] distributed a merge tree computation by observing that each vertex in the mesh belongs to a unique component based at a single root maximum, and to a corresponding component at a minimum. Thus, by storing the location of each vertex in a merge tree, the merge tree is held implicitly, distributed across the nodes of the computation. They then generalized this further [30] and stored unique maximal and minimal roots for each vertex. Since this combination is unique for each edge of the contour tree, the contour tree is stored implicitly across a cluster. These algorithms focus on distributed computing but not data-parallelism, limiting efficient utilization of individual compute nodes.

Landge et al. [24] used segmented merge trees to segment data and identify threshold-based features. They constructed local merge trees and corrected them based on neighboring domains. By considering features only up to a predefined size, this correction process requires less communication than the approach by Morozov & Weber [29].

Related to this, Widanagamaachchi et al. [41] described a data-parallel model for the merge tree, breaking the computation into a finite number of fan-in stages. This approach in effect quantized the merge tree, an effect that was acceptable for the task in hand.

The hybrid GPU-CPU algorithm by Maadasamy et al. [27] finds critical points then monotone paths [13] from saddles to extrema, to build join & split graphs to identify equivalence classes of vertices that share a set of accessible extrema to compute the merge trees.

Once the merge trees are computed, the computation continues in serial on the CPU, using the merge phase of Carr et al. [9]. Where $E \ll V$, this is practical, but as shown by Carr et al. [10], there are classes of data (principally empirical) for which $E \approx V$. Moreover, even the GPU phase is not pure data-parallel, as the search from saddles to extrema is serial for each vertex, and the number of steps needed is bounded by the longest such path in the mesh. Although this tends to average out over a large number of vertices, it limits the formal guarantees on performance. Lastly, this algorithm computes the unaugmented contour tree, limiting the forms of analysis that are feasible.

Some of the work on Reeb graph and higher-dimensional topological computation is also relevant. In particular, Hilaga et al. [21] quantized the range of the function, explicitly dividing an input mesh into slabs—i.e., the inverse image of intervals rather than of single isovalues. They then identified the neighborhood relationships between these slabs to approximate the Reeb graph of a 2-manifold. More recently, Carr & Duke [5] generalized this with the Joint Contour Net—which approximates the Reeb space [14] for higher dimensional cases—by quantizing all variables in the range.

Based on quantized Joint Contour Net computation, Carr et al. [7] used Reeb's characterization to contract contours to points. They achived data-parallel computation by using explicit quantization to break cells into fragments representing fat contours as in the work on Joint Contour Nets [5], then used the parallel union-find algorithm of Sewell et al. [35] to collapse the contours nodes in the quantized contour tree. A second union-find pass then constructed superarcs out of the nodes. However, this was profligate of memory, and processed c. 1M samples on a single Tesla K40 card, with a memory footprint even larger than the sweep and merge algorithm.

In contrast, Gueunet et al.[17, 18] employ task-based parallelism in shared memory to work around the serial sweep, first by computing separate contour trees for subranges of the scalar value [17], then by a task-based algorithm that decouples the sweep for separate peaks, with a common pool of small sweeps shared by all threads [18].

3 New Terminology

We start by introducing two new terms that will help us build our algorithm: governing saddles, and pseudo-extrema. The motivation for this is that the new algorithm does not rely on cancelling critical points in the same way as previous work. New terminology was therefore inevitable, but we have tried to limit the number of new terms, which we describe in the following subsections.

3.1 Governing Saddles

Previous simplifications pair peaks with saddles to build a hierarchy. Instead, we allow multiple peaks to pair with one saddle. We therefore define the *governing saddle* for a peak to be the highest point from which monotone paths exist to that peak and at least one other: such a point will always be a saddle point. Moreover, where a Y-structure is broken into one long and one short edge in branch decomposition, we break it into two short branches and a residuum. Inversely, the governing saddle of a minimum is the lowest point from which monotone paths exist to the minimum and at least one other.

In Figure 1, we also show the decomposition of the join and split trees based on governing saddles. For example, 15 is the governing

saddle for *both* 24 and 22, and 10 is the governing saddle for both 13 and 15—i.e. we can assign governing saddles to saddles themselves.

3.2 Pseudo-Extrema

During this process, we prune peaks to their governing saddles, and will often prune all peaks sharing a governing saddle at once. In this case, the saddle will effectively become a peak in its own right. This can be handled mathematically by contracting the region for the peak, by excerpting the peak from the domain, or by operations on the mesh [4]. We refer to such saddles as *pseudo-extrema* in order to make it clear that a vertex currently being treated as an extremum may be a saddle in the original data. We will see later that not all saddles are pseudo-extrema: hence the introduction of a separate term.

One reason for this choice is that pair cancellation is a serial process, proven by linear induction. For parallelism, we want properties that are not defined by simple linear induction, preferring rather to use properties that reduce problem size by many elements at once.

Moreover, saddles at one iteration may become regular points at a later iteration. Where needed, we will refer to these as *pseudo-regular points*, but we will not rely as heavily on them as the pseudo-extrema.

In the base case, all saddles will have been pruned down to pseudo-regular points except the global maximum, and we will refer to the resulting structure as the *trunk* of the tree.

In branch decomposition or simplification, this equates to choosing the vertex depth in the tree as an importance measure, then batching simplification and degree 2 vertex reduction.

4 Parallel Peak Pruning for Merge Trees

Our new algorithm, *Parallel Peak Pruning* (PPP), is fully data-parallel and computes both merge and contour trees. Since the join tree and split tree computations are symmetric in nature, we describe and illustrate the algorithm for the join tree only. At heart, our algorithm is similar to simplifying a contour tree: we identify peaks and find their governing saddles to establish superarcs in the join tree, then delete (prune) the regions defined by each peak/saddle pair, and process the remaining data recursively. When only one peak remains, there is no saddle, and all remaining vertices form the "trunk" of the tree.

Since the details are somewhat complex after optimization, we will build it in several stages:

S. 4 Parallel Peak Pruning to Construct Merge Trees

S. 5 Optimising Parallel Peak Pruning

S. 6 Parallel Combination of Merge Trees

We assume that the input is a triangulated mesh in 2D, and reduce it to the edge graph of the mesh, as we know that this is sufficient to compute the join tree [8]. We will see in Section 7 how to extend the current work to other mesh types, but defer details until then.

We start with all data values sorted using simulation of simplicity [16] and assign a unique sorting index which we then use throughout for comparisons. This simplifies the code significantly and reduces memory access, at the cost of an initial sort over all data values.

The parallel peak pruning algorithm then operates as follows:

1. Iterate Until No Saddles Remain:
   (a) **Monotone Path Construction:** from vertices to peaks
   (b) **Peak Pruning:** to governing saddles
2. **Trunk Construction:** from remaining vertices
3. **Join Arc Construction:** along superarcs

Monotone Path Construction: In this phase, we build one monotone path from each vertex to a peak. No canonicity is assumed, as any peak reachable from the vertex can be chosen. The simplest way is to choose the first ascending edge from each vertex, except for peaks, as shown in Figure 2. Since every edge points to a higher vertex (except at peaks), we have no cycles, and the directed graph is therefore a forest. In this forest, each tree consists of a set of vertices which are guaranteed to have a monotone path to the peak at the root of the tree. We then set each peak to point to itself to simplify the computation.

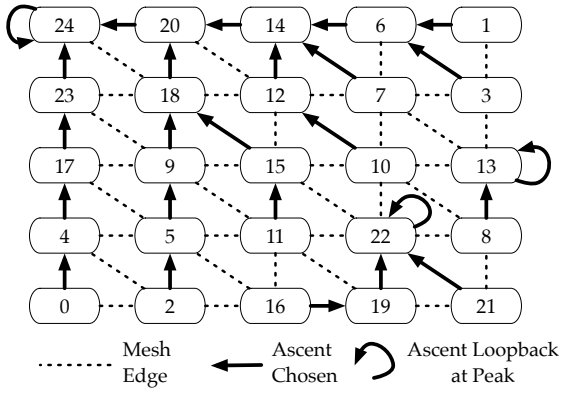


Fig. 2: Selection of Initial Ascending Edge

Since the trees are connected components of the forest, we use pointer-doubling [23] to collect the trees, as shown in Figure 3. In each iteration, each vertex points to its ascending neighbor's neighbor, terminating at the peak. At the end of this process, every vertex has been assigned to a peak, shown by the coloured groups: the colour attests to the existence of a monotone path from the vertex to the peak.

Peak Pruning: In the second phase, we identify the governing saddles for each peak. Recall from Section 3 that the governing saddle g of a peak p is the highest saddle from which a monotone path to p exists. Since identifying saddles accurately is more difficult than it might seem, we define a *saddle candidate* to be a vertex which has ascending edges whose upper ends are labelled with at least two peaks: i.e. vertices in Figure 4 with ascending edges in two different colours. Every saddle is guaranteed to be a saddle candidate, but not vice versa: for example, vertex 5 is a saddle candidate but not a saddle.

A saddle candidate c from which a path ascends to peak p cannot be higher than the governing saddle g for the peak, as otherwise c would govern. It then follows that the governing saddle is the highest saddle *candidate* from which a monotone path to p exists. This gives a parallel test to find the governing saddles for all peaks. We simply sort all of the edges ascending from saddle candidates by four criteria, using either a single lexicographic sort or a sequence of stable sorts:

1. Whether the lower end is a saddle candidate
2. The peak ID assigned to the upper end
3. The ID of the lower end
4. The ID of the edge

The first of these criteria, illustrated in the first row of Figure 5 is used to ignore all edges whose lower end is not a saddle candidate. Alternately, these edges can be omitted from the sort, but the cost of doing so in parallel is the same as sorting.

The second criterion sorts the edges into equivalence classes by peak, illustrated by the colour in the second row of Figure 5, while the third criterion ensures that an edge from the governing saddle is at the beginning of the group, as shown by the boxes in Figure 5.

The fourth criterion is not needed, but forces a canonical order, which makes debugging in parallel considerably easier.

After this sort, all edges leading to each peak p are clumped, with the leftmost (highest) such edge adjacent to the end of the array or to an edge leading to a different peak. This test for peak-saddle pairs is fully parallelized over all edges: only the edge that satisfies the conditions is allowed to assign the saddle s to the peak p, precluding write conflicts.

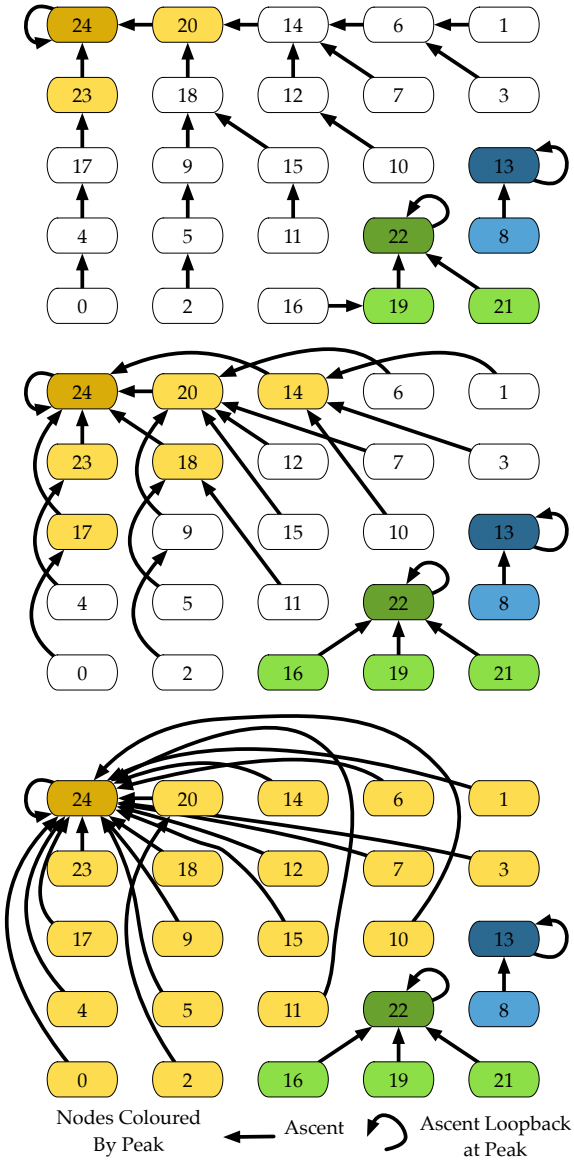


Fig. 3: Monotone Path Construction. 3 iterations are required.

This process pairs critical points, but we must still process the regular points (which may previously have been critical points). We therefore exploit a simple property: any regular vertex above the governing saddle s of a peak p can only have monotone paths to p, and can therefore be assigned to that peak, shown in Figure 6 by color-coding.

Once peak p and its regular vertices are found, they are no longer needed. We will exploit this later, but for now, once a vertex has a peak assigned, it is ignored. This can be achieved if each vertex checks to see whether it has a peak assigned before it is considered in each pass: in the sorting pass, we force all such vertices to sort high. However, monotone ascents to vertices inside this region are still needed, which is handled by redirecting any edge leading to a marked vertex to ascend to the governing saddle, as shown in Figure 7.

Trunk Construction: In each pass, we prune all peaks, flattening (or deleting them) to remove the region above the governing saddle. Each governing saddle then becomes either a peak (e.g. 15 in our example) or a regular point (e.g. 10 in our example). We recompute monotone paths and iterate: Figure 8 illustrates the next iteration for our example. Here, there is only one peak left at 15 and no saddles, so we have hit the base case and can assign all remaining vertices to the trunk which leads downwards from 15 to a virtual saddle at $-\infty$.

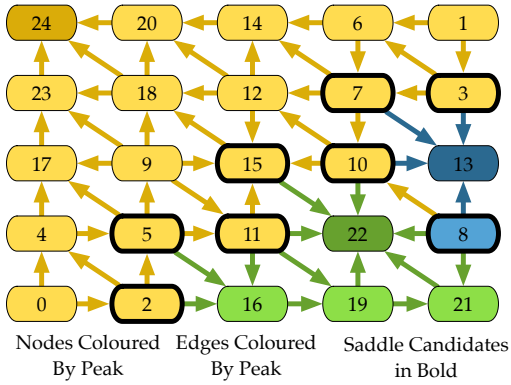


Fig. 4: Saddle Candidate Identification. Edges are assigned to the same peak as their upper end. Any vertex whose edges lead to multiple peaks is a saddle candidate.

Regular Arc Connection: One final step remains: to compute join arcs connecting the vertices together in the fully-augmented join tree. Observe that each vertex points to the next highest vertex on the same peak: we construct this by sorting first on the peak ID, then the vertex value, as shown in Figure 9. Each vertex then points to its right-hand neighbor, unless this belongs to a different peak, in which case the vertex points to the governing saddle of the peak.

We note that this does not give the join superarcs in the same form as existing algorithms. To see this, consider Figure 9, which shows the join arcs we have just computed. We extracted peak-saddle pairs $(24,15),(22,15),(13,10),(15,-\infty)$: note that in the second pass, 10 was treated as a regular point, not a critical point.

4.1 Algorithmic Analysis of the Naïve Algorithm

To analyse the performance of this naïve algorithm, we first ask how many iterations it will take to compute the entire merge tree. Notice that in each iteration, all (upper) leaves are pruned as peaks simultaneously. We observe that, if a tree only has vertices of degree 1 (leaves, or the global minimum) or > 3 (saddles), then at least half of the vertices must be leaves. As long as each iteration removes all of them, and preserves the degree condition, we are guaranteed to terminate in at most $O(\lg V)$ iterations, or more precisely, in $O(\lg t)$, where t is the number of supernodes in the merge tree.

Pruning the upper leaves may result in a large number of internal saddle points becoming regular points during the computation. This is why we recompute the monotone paths to peaks in each pass: this treats these points as regular points for the next iteration. As a result, our invariant is preserved, and we take at most $O(\lg t)$ passes. In this case, notice that we take exactly 2 passes for a tree of 28 nodes, so $O(\lg t)$ is a loose bound in practice. Oddly, this difference is most pronounced for unbalanced merge trees such as this example, and balanced trees are tight to this bound. Thus, unlike many graph algorithms, the more balanced the tree, the slower the computation!

Within each pass, assuming we carry all vertices forward as described above, we will perform $O(V \lg V)$ work in $O(\lg V)$ steps to recompute the monotone paths, followed by $O(E \lg E)$ work in $O(E)$ steps to sort the E edges. Identifying this iteration's peaks then takes $O(E)$ work in $O(1)$ steps, while assignment of regular vertices to peaks takes $O(N)$ work in $O(1)$ steps.

Finally, in the base case, construction of the trunk takes $O(V)$ work in $O(1)$ steps, while the final join arc construction requires a sort in $O(V \lg V)$ work and $O(\lg V)$ steps followed by $O(V)$ work in $O(1)$ steps to connect the join arcs together. If the superarcs are to be constructed as well, then a similar sort of the supernodes along the peaks can be performed in $O(t \lg t)$ work in $O(1)$ time.

Overall, naïve parallel peak pruning (PPP) takes $O(E \lg V \lg E)$ work in $O(\lg V \lg E)$ steps. This is *less* efficient than sweep and merge, so massive parallelism would be needed for practical gains. Initial

From Saddle Candidate	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Peak for Upper End	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	22	22	22	22	22	22	22	22	22	13	13	13	13
Lower End	15	11	10	10	8	7	7	7	5	5	5	3	3	2	2	15	11	11	11	10	8	8	5	2	10	8	7	3

From Saddle Candidate	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Peak for Upper End	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	22	22	22	22
Lower End	23	20	18	18	18	17	14	12	12	12	12	9	9	9	9	9	6	6	4	4	1	1	0	0	21	19	19	16

Fig. 5: Peak-Saddle Pairing. In the sorted array, edges from non-saddle candidates (grey) are ignored. Edges whose left neighbor has a different peak identify the peak-saddle pairings (marked as a box).

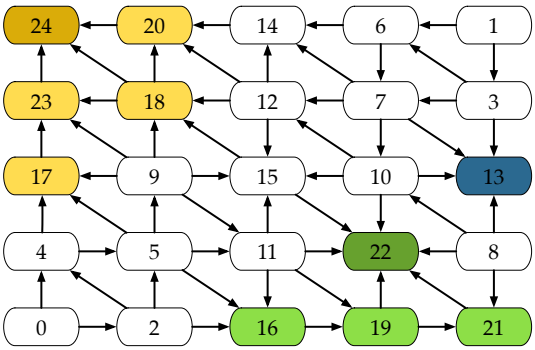


Fig. 6: Assigning Regular Points. Since peak 24 has 15 as its governing saddle, vertices labelled 24 whose value is higher than 15 can now be assigned to 24. Similarly, vertices 16, 19, 21 can be assigned to 22, while no regular vertices labelled 13 lie above 10.

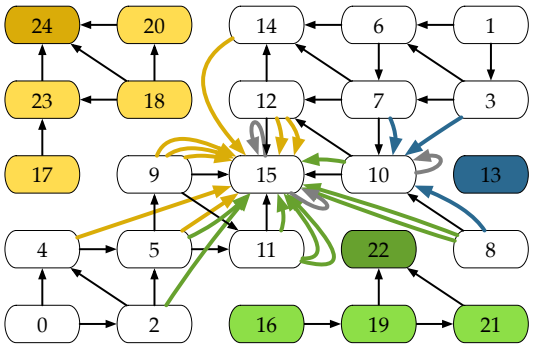


Fig. 7: Peak Pruning. Edges leading up to pruned vertices are truncated at the saddle's isovalue, represented by redirecting them to the saddle. Pruned vertices and looped edges are removed.

runs confirmed this, with computation times several orders of magnitude slower than sweep and merge. This led to consideration of the trade-off between work and steps, and identification of bottlenecks.

5 Optimization with Active Topology Graphs

This algorithm is asymptotically efficient in parallel but slow empirically, with the most expensive routine being the sort, which had to sort E edges per iteration. Ironically, we found ourselves bemoaning the fact that we "only" had 3,000 processors on a GPU, and turned our attention to optimising the work performed as well as the time required.

This sort is repeated in every iteration, carrying forward every vertex and edge until the end of the computation. We therefore introduced two fundamental optimizations, reducing the size of working arrays in every iteration, and restricting the computation to the supernodes.

5.1 Active Working Graph

The first optimization we applied was to observe that, once a vertex has been pruned, it can be dropped out of the arrays for subsequent iterations. This adds an additional step at the end of the loop, compressing the *active working set* of vertices and edges. Formally, this adds an extra reduction to the iteration, but since we are sorting all edges anyway, it does not increase the asymptotic cost. In practice, however, this optimization alone improved runtimes by an order of magnitude, but further analysis will be deferred until we have described the second optimization: critical topology graphs.

5.2 Critical Topology Graph

We saw in Section 2.2.2 that contour trees need not be computed from the edges of the mesh: instead, it suffices to use a *topology graph* capturing all monotone paths between critical points. We therefore borrow from the work of Chiang et al. [13], and define a *critical graph* on which the PPP algorithm is executed.

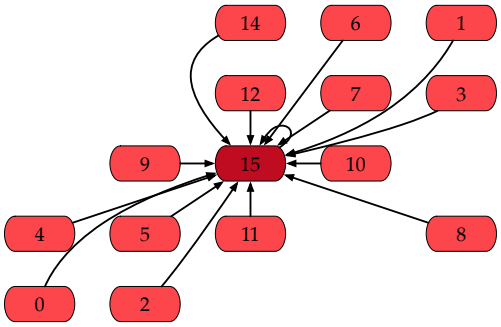


Fig. 8: Second Round of Monotone Path Construction. 15 is the last peak, and we transfer all vertices to the trunk of the tree.

In this graph, the vertices are a superset of the critical points, defined conservatively by the invariant that edges incident to join saddles ascend to at least two different maxima during the initial peak-finding routine. This is necessary but not sufficient, with no guarantee that only critical points are selected. We use a simple local test based on the link to consider only Morse critical points. In 2D, this allows us to define bounds based on the size t of the contour tree, but in 3D and higher the size t means the number of Morse critical points.

To construct the critical graph, we identify a small superset of the critical points, and a set of edges ascending from them to peaks [13]. The first task, critical point detection, looks at the link of each vertex to detect criticality. In a triangulated mesh, any vertex whose link has

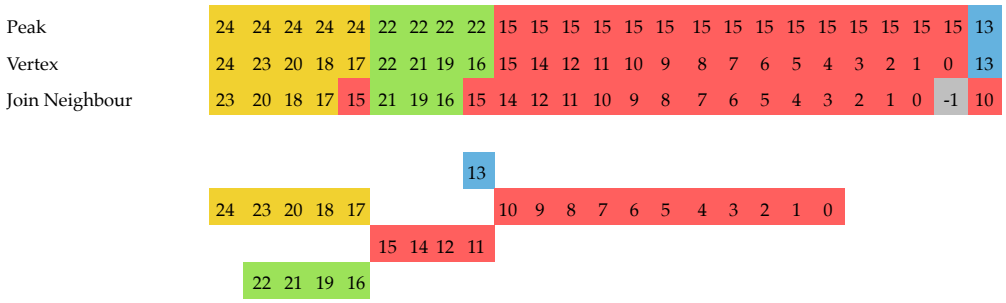


Fig. 9: Join Arc Construction. Each vertex points to the next lowest in its peak-saddle superarc. Each run of nodes along a superarc transfers as a unit to the join tree (shown horizontally beneath)

either zero or two or more connected components of neighbors with higher values is a critical point, either a peak or a saddle. All such vertices are put in the critical topology graph.

For each saddle u, we want an ascending edge for each upper link component: we select one vertex v in each link arbitrarily. We then extend this edge to a path to a peak for three reasons. First, v may not be a critical point, but every peak is. Second, by extending the path to a peak p, we satisfy the conditions necessary for the critical graph to be a topology graph for contour tree computation [8].

We initialize the critical graph by computing paths from all vertices in the mesh to a peak as in the naïve algorithm. We then identify critical points with a conservative test: any vertex whose higher neighbors belong to either zero peaks or more than one peak could be a saddle point, while any vertex whose higher neighbors belong to exactly one peak must be a regular point. Note that we could have an arbitrary number of unnecessary points at this stage, and that this will complicate formal analysis. For each such vertex u transferred to the critical graph, we choose at least one neighbor v per upper link component and follow to its peak $p = peak(u)$, then add (u, p) to the active graph.

This neighborhood examination can be done in several ways. First, we can sort to find equivalence classes of upper link components, as with the naïve algorithm. Second, in 2D we can iterate around the link, counting sign changes along the way. Third, we can use union-find on the link of u. And last, for regular lattices, we can compute bitflags for the sign of each neighbor, then use a lookup table.

Clearly, which choice we make here will have a significant impact on performance, but none of them alter the algorithmic analysis, since we cannot guarantee how many vertices will be transferred to the active graph. In practice, however, this step typically reduces the graph size by at least one order of magnitude, reducing practical run times accordingly. Moreover, the cost of constructing the active graph is at worst $n\lg(n)$ work in $\lg(n)$ time, the same as the initial pointer-doubling computation of paths from every vertex to peaks.

5.2.1 Active Graph Peak Pruning

After computing the critical graph, we run the PPP algorithm. In each pass, pseudo-extrema are pruned to saddles. Regular and pseudo-regular points are assigned to pseudo-extrema and also pruned. Vertices are *only* labelled as supernodes when a superarc begins or ends at them. As a result, any regular point included in the critical graph by our conservative test is left unlabelled.

In the ideal case, our active graph would consist only of the $O(t)$ supernodes belonging to the merge tree. Moreover, half of them are pruned in each pass, then removed from the working (active) vertex set. We therefore perform $O(t)$ iterations with $O(\lg t_i)$ work each, where $t_i < 2t_{i+1}$. This results in an overall work cost of $O(t\lg(t))$ and time cost of $\lg^2(t)$, reducing the work by a log factor, but not the time cost. As noted above, where the number of processors is much smaller than the number of elements, the dominant cost is the work cost, and we would therefore expect significant speedup. Moreover, when we compress the vertices and arcs in each pass, we can test the remaining number of vertices: when it reaches 0, no more iterations are required, and we may terminate long before $\lg(t)$ passes are required.

Most of the time, the size of the active graph is a small multiple of the tree size, and the performance improvement is in line with the ideal case. In fact, it is commonly the case that fewer than $\lg(t)$ iterations are required, indicating that the trees are generally unbalanced.

6 Parallel Combination of Merge Trees

Given our parallel merge tree algorithm, we now identify parallelism in the merge phase. The merge phase of the sweep and merge algorithm is based on the observation that an upper leaf in the join tree is always a leaf in the contour tree. From this, a queue is constructed with all current leaves, which are transferred one by one to the contour tree, updating both join and split trees as each vertex is transferred.

The choice of a queue to process the leaves was arbitrary, and on principle we can transfer all leaves simultaneously, subject to preserving the necessary invariants in the two merge trees. In serial, this was performed by deleting vertices one at a time, reconnecting their neighbors if necessary. Since a given vertex may (for example) be a saddle for multiple leaves, this means that read/write collisions in parallel will happen, and we therefore replace the queue processing with a batching approach. For simplicity, we process upper and lower leaves in alternating batches, using a lazy deletion strategy.

In the first batch, every upper leaf identifies its governing saddle, and records this in the contour tree. This is illustrated in the first Phase of Figure 10, where vertices $20, 19, 18, 16, 14, 12, 10$ are transferred to the contour tree and deleted from join and split tree. In the join tree, deletion is trivial since all of these vertices are leaves. In the split tree, these vertices lie between other vertices, whose connectivity therefore needs adjustment, and it is clear that write conflicts on the chain $20-19-18-16$ are to be avoided.

These write conflicts are avoided by having each vertex mark itself as logically deleted. We then collapse the chain $20-19-18-16$ with an additional path-doubling computation. For this, we initialize each vertex to point downwards. In each iteration of the path-doubling, each vertex updates to its neighbor's neighbor, but only if its neighbor has been flagged for deletion. At the end of this, vertices $20, 19, 18$ will all point to 17, 17 will point to 15, and so forth.

At the end of this step, we have the trees shown in Phase Ib: note the chain $17-15-13-11-7$. If we continue processing leaves in batches as we have just described, each iteration will reduce a single step on this chain, with the algorithm serialising along all such chains. To avoid this, we insert an additional phase which (again) uses pointer-doubling to collapse the chain down to vertex 7 in the join tree, removing the additional vertices from the split tree as well.

Phases II, III, IV then repeat the process, alternating upper and lower leaves until all vertices have been processed.

6.1 Algorithmic Analysis of Parallel Merge Phase

The performance of this parallel merge phase is dominated by the number of iterations I required to process all supernodes. Transferring leaves takes constant time per vertex, for a time bound of $O(1)$ and

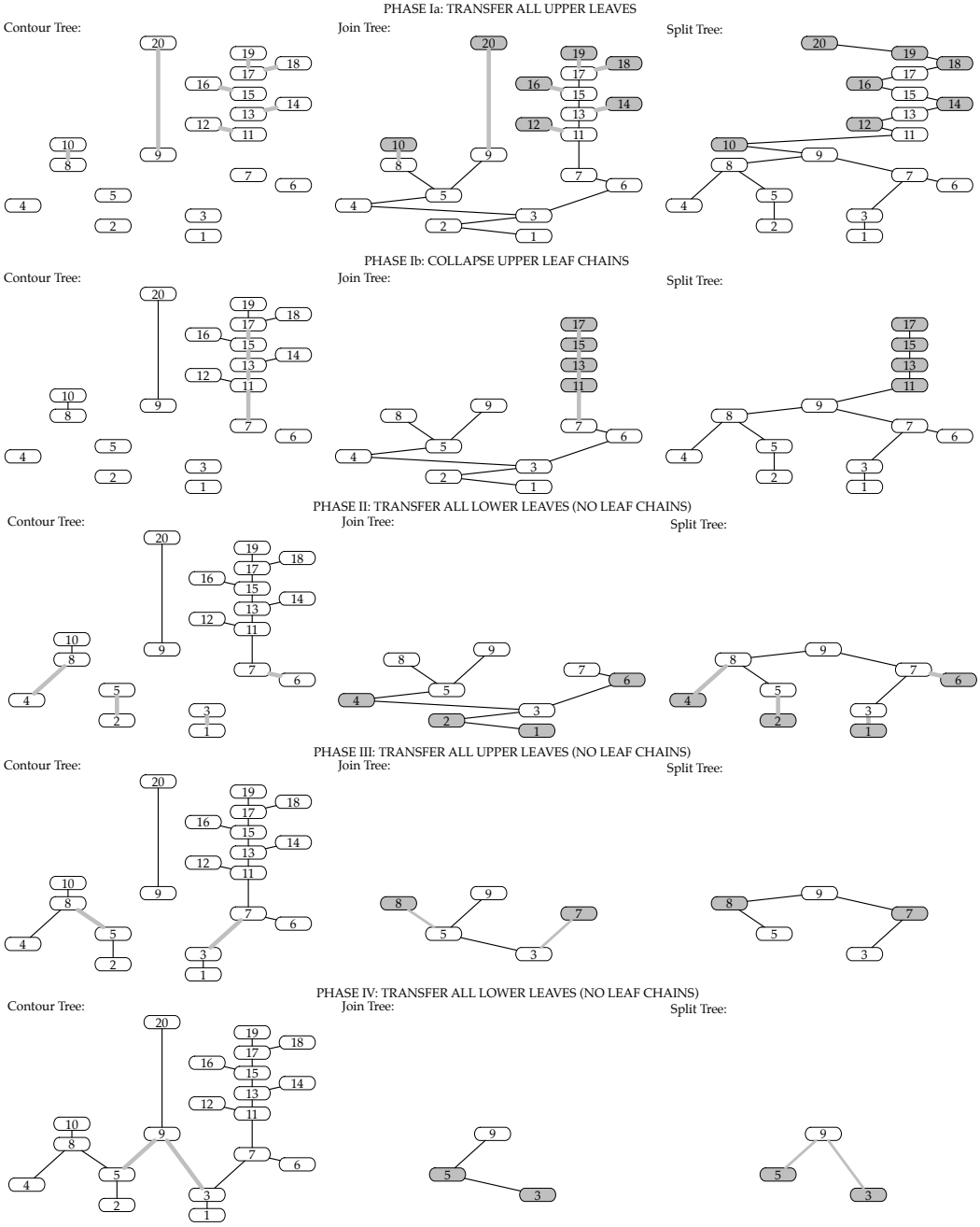


Fig. 10: Parallelization of Contour Tree Merge Phase. We first identify all upper leaves in parallel (top) and transfer them to the contour tree. After deleting these vertices from the join and split trees, we collapse regular chains from upper leaves (second), then repeat with lower and upper leaves, omitting collapses if there are no chains available. Note how the algorithm serializes along the W shape of vertices 8-5-9-3-7.

work bound of $O(t_i)$ in pass i, where t_i is the number of vertices still in the tree. However, the two path-doubling phases for collapse take $O(\lg t_i)$ time and $O(t_i \lg t_i)$. Formally, therefore, the bound is $O(I \lg t)$ time and $O(It \lg t)$ work, although this can be refined to $O(\sum_1^I \lg t_i)$ time and $O(\sum_1^I \lg t_i t_i)$ work.

In the ideal case, we would have a logarithmic collapse as for the merge tree construction phase, with $I = O(\lg t)$ and $t_i = O(t/2^i)$. In this case, the overall cost collapses to $O(\lg t^2)$ time and $O(t \lg t)$ work. Unfortunately, however, this does not occur in practice, due to three effects which serialize the computation: leaf chain collapse, interior chain collapse, and W-structures.

Leaf Chain Collapse: We observed in Phase Ib of Figure 10 that the basic algorithm serializes along chains of vertices. The result of this is that the computation may require $O(d)$ iterations, where d is the diameter of the contour tree, which in the worst case is $O(t)$. While the additional collapse described in Phase Ib accelerates this, a formal improvement only occurs if we preserve the invariant that all leaves are pruned and all degree 2 vertices are collapsed.

Interior Chain Collapse: The reason we can collapse the leaf chains easily is that chains starting from leaves are easily identified by propagating the leaf inwards. Similar interior chains are harder to collapse, and we do not do so at present, because interior chains are typically bound up with W-structures (below), and resolving these is logically prior. Moreover, applying the leaf chain collapse alone re-

duces the practical cost of the merge phase to roughly the same as the construction of the two merge trees. We therefore did not optimize this stage further, as it would give us marginal advantage (for now).

W-Structures: In Figure 10 (bottom left), the sequence of edges $4-8-5-9-3-7-6$ form a horizontal zigzag. We call this a *W-structure*, and have reported elsewhere on its properties and how to find the largest one in a contour tree [11, 22]. Because of this zigzag, at most one edge at each end is removed in each alternating pass, and therefore w, the size of the largest W-structure is a lower bound on the number of iterations required. We are actively seeking accelerations for this part of the computation, but as noted above, are not focussing on it due to limited performance gains currently available.

Algorithmic Summary: The most that we can therefore claim is $O(I \lg t)$ time and $O(It \lg t)$ work, but this is a very loose bound in practice, and as we will see below, sufficiently fast in practice.

7 Adaptation to Non-Triangulated Meshes

Like sweep and merge [9], parallel pruning is most straightforward and intuitive for triangulated meshes. Linear interpolation ensures that critical points coincide with mesh vertices, and concepts such as vertex neighborhood and link have long established definitions. As a result, sweep and merge is effectively a graph algorithm whose input is the set of vertices and edges of the triangulated mesh.

Carr & Snoeyink [8] showed how to construct *topology graphs* that adapt the algorithm to other types of meshes. The general principle is that a topology graph must contain all critical points, and that any continuous path through a super-level set or sub-level set must be representable by a path through the topology graph. It therefore becomes feasible to define local graphs for each cell in the mesh that capture the topology of the cell and its interpolant, then glue them together to get a global topology graph on which the algorithm is run. This allowed adaptation of the algorithm to bilinear and trilinear interpolants, and on principle to higher-order interpolants.

These graphs could, however, be fairly complex, and very few packages use trilinear interpolants for isosurface extraction and visualization. Instead, most isosurfacing algorithms use variants of the original Marching Cubes [26] in which the cracks in the original version are fixed [28]. These cases have simple topology graphs, as the isosurfaces extracted always have super-level sets where only connectivity along the edges of the cubes was needed, while sub-level sets require connectivity along diagonals as well. As a result, separate topology graphs can be defined for join and split sweeps, which are equivalent to a standard rule in digital image processing: that foreground pixels connect orthogonally, but background pixels also connect diagonally. In 3D cubic meshes, this means processing the six orthogonal neighbors in the downward (join) sweep, the 18 orthogonal and face diagonal neighbors in the upward (split) sweep.

In practice, this observation is significant for topological analysis and visualization for two reasons. First, Marching Cubes are nearly ubiquitous in visualization, and computing contour trees under the same assumption assures that we use exactly the same isosurfaces as those extracted for other purposes. Second, Marching Cubes generates 60-70% fewer triangles than a mesh constructed by subdividing the cubes into tetrahedra [6].

To implement parallel peak pruning for non-triangulated meshes, we start with the existing mechanism of topology graphs [8], but note an important difference: sweep and merge detects merge events between sets in a disjoint set (union find) data structure while parallel peak pruning identifies candidate neighbors for distinct paths to maxima (or minima). In the split and merge approach, it suffices to identify all neighbors of a given vertex since it determines "automatically," using union find, whether multiple neighboring vertices belong to the same connected component of the link. For parallel peak pruning, this is not true, and we also wish to have a single representative edge for each connected components to keep our computational cost down.

The general solution to this problem is to extract the upper link of each vertex, and compute its connected components directly using a union-find algorithm. One vertex is then chosen from each connected component as a representative, and the ascending path traced through it. We note that this is not strictly necessary, as we can simply include ascending paths through every adjacent vertex in the upper link, but this can result in twice as many edges in the active graph, with obvious knock-on effects on performance.

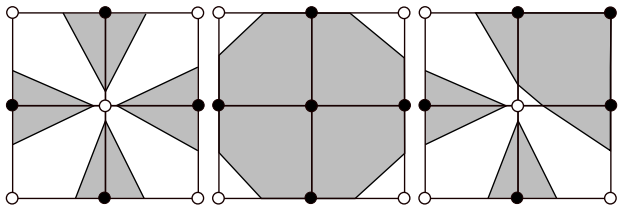


Fig. 11: The choice of ascending paths is dictated by the number of connected components in the upper link, illustrated by contours at the center vertex value plus ε (grey regions). Left: only the edge-connected vertices are in the upper link, and there are four connected components. Center: For a contour at the center value minus ε (gray region), the four components in the upper link have merged. Right: one additional vertex is in the upper link means there are only three connected components. The calculation can therefore not be restricted to only edge-connected neighbors.

For regular meshes, the upper link is bounded in size, so we optimize further with a case table based on the relative sign of the neighbors in the link, and look up the result rather than performing union-find repeatedly. Appendix I provides more implementation details.

7.1 PPP for Marching Cubes Connectivity

We have just observed that we need one ascending path per connected component of the upper link of each vertex v. For a cubic lattice, this still holds, provided we understand the upper link to mean the outer boundary of the set of cells incident to v. To see that this is true, consider Figure 11, which shows the two-dimensional equivalent with marching squares. In the left-hand example, each adjacent vertex is in a separate component (shown as grey regions), and processing is straightforward. In the right-hand example, however, the fact that the upper right vertex is also above the central vertex' value means that we only have three connected components.

We therefore consider all twenty-six neighbors for our union-find, and either add them one at a time using the existing rule [8] or precompute a lookup table for all possible configurations. However, with twenty-six neighbors, the lookup table would have $2^{26} = 67,108,864$ entries. We therefore use a lookup table to represent the connectivity inside each adjacent cube. Since there are 7 vertices in the cube in addition to v, we only need 128 entries, and these record how the edge-adjacent vertices are connected to each other in the upper link.

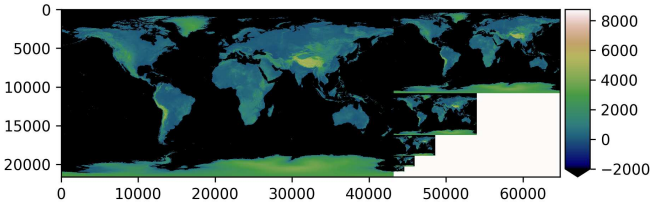
Over the entire neighborhood, we use these per-cube connections in a union-find pass to compute the connected components of the upper link, and choose one ascending edge per component. Since the total number of vertices is a small constant in each neighborhood, we do not use either path-compression or union-by-rank.

8 Results & Performance Analysis

In the following we seek to characterize the performance of our algorithm in practice. We first describe the design of our performance evaluation study (Section 8.1). Based on the results from this study we then try to answer critical questions concerning the correctness and runtime performance of our algorithm as well as the impact of algorithmic improvements, data size, data complexity, number of threads, and compute architecture on runtime performance (Section 8.2).

8.1 Experiment Design

Implementation: We implemented the parallel peak pruning algorithm for 2D and 3D data in C++ in several variations. We first



Scale	1.0	0.5	0.25	0.125	0.0625	0.03125
#Pixel $nx \times ny$	21601 × 43201	10800 × 21600	5400 × 10400	2700 × 5400	1350 × 2700	675 × 1350
#Nodes (s)	36,912,523	12,688,670	3,579,117	991,480	271,772	72,276
Complexity	3.96%	5.44%	6.14%	6.8%	7.46%	7.93%

Fig. 12: Visualization showing the full GTOPO dataset at the various levels of resolution used for the scaling study. The table then shows for each dataset the spatial resolution ($nx \times ny$), number of supernodes s in the contour tree, and relative topological complexity $s/(nx * ny)$.

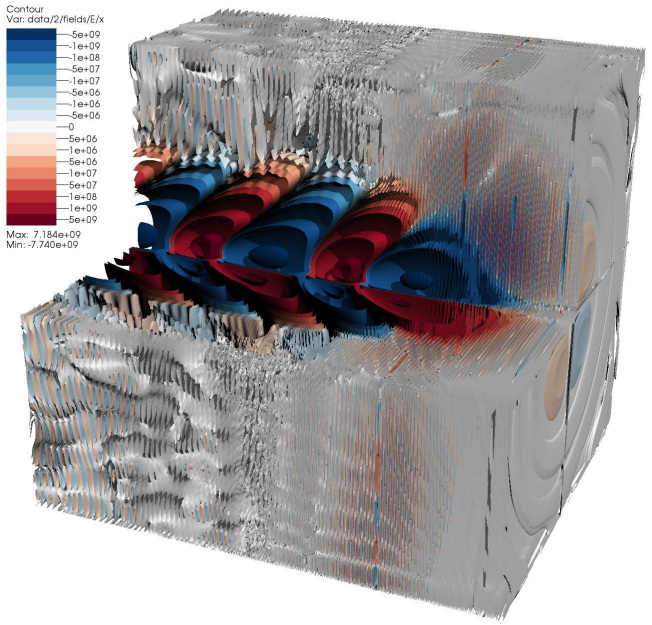


Fig. 13: Isosurface visualization of the Ex field of a single timestep of a 3D WarpX laser plasma particle accelerator simulation with a resolution of $(425 \times 371 \times 371)$.

implemented a serial reference implementation, focusing on correctness rather than performance. Subsequently we then optimized and parallelized the code using OpenMP (**PPP (OMP-ref)**) for threading and using Thrust (**PPP (Thrust-ref)**) for GPU. Unless indicated otherwise, we focus on results from our final implementation in VTK-m (**PPP (VTK-m)**), which is now publicly available. Using VTK-m enables us to compile the code using various backends. We use the serial backend for serial performance and the TBB backend for parallel scaling using the Intel Thread Building Blocks library for threading. As reference for comparison we use the original serial sweep and merge algorithm (**SAM**) [9] and naïve implementation of PPP (**PPP (na)**), i.e., without the active topology graph optimization. Finally, we also compare with the task-based parallel contour tree algorithm [18] available in the topology toolkit (**TTK**).

Data: To evaluate performance on real data we use the 2D GTOPO30 dataset and 3D WarpX dataset. The GTOPO30 dataset describes the Earth land topography and consists of 34 tiles; 27 at 6000×4800 pixel, 6 at 3600×4800 pixel, and 1 at 5400×5400 pixel. When combining the tiles, the full dataset consists of (21601×43201) pixel. Finally, we rescaled the full image by progressively reducing resolution by half, i.e., $G(1.0)$ to $G(0.03125)$ (Figure 12).

The WarpX dataset models a laser-driven, plasma-based particle accelerator in 3D (Figure 13). We use the electric field in the x direction Ex for our tests. The scientists executed the same simulation model at a spatial resolution of $(425 \times 371 \times 371)$ and $(6791 \times 371 \times 371)$ nodes, respective. The higher resolution in the acceleration direction enables more accurate resolution of the high frequencies of the laser pulse whereas lower mesh resolutions are sufficient in the other dimensions to resolve the electromagnetic fields induced by the comparatively lower-frequency plasma waves.

We also use several 3D data sets made available to the visualization community for our scaling studies. Most of these data sets are from the Open SciVis Dataset page (`https://klacansky.com/open-scivis-datasets/`), which includes many data sets from the no longer available VolVis page (`http://volvis.org`). Furthermore, we use a time step from the SciVis contest asteroid data set [33], which is available from the Los Alamos National Laboratory (`https://dssdata.org`). Appendix A provides a list of all 3D data sets including source and size.

Architecture: To evaluate performance across different compute architectures we used the following systems. **Haswell** refers to a single compute node of the Haswell partition of the Cori supercomputer at NERSC, equipped with two 16-core Intel® Xeon ™ E5-2698 v3 ("Haswell") processors at 2.3 GHz and 128 GB DDR4 2133 MHz memory. Each core supports two hyper-threads and has two 256-bit-wide vector units, i.e., each node support 32 physical and 64 hyper-threads. **KNL** refers to a single compute node of the KNL parition of Cori, equipped with a single-socket Intel® Xeon Phi™ 7250 ("Knights Landing") processor with 68 cores at 1.4 GHz with 4 hardware threads (272 threads total), two 512-bit-wide vector processing units, and 96 GB DDR4 2400 MHz memory. All codes were compiled using gcc (version 7.2.0) with optimization options `-O3 -ffast-math -funroll-loops -march=native -dynamic -w -DNDEBUG -std=c++11`. (We note that `-dynamic` is an option of the compiler wrappers on Cori that enables dynamic linking.) For evaluation of performance on GPU we use **P100**, which is equipped with a single Intel® Xeon ™ CPU E5-2650 v4 processor at 2.20GHz and a NVidia® Tesla P100-PCIE graphics card with 12GB of memory and 1328 MHz base clock. For our performance tests we repeated each test 5 times and report the time of the fastest run to mitigate the impact of system background processes on measured runtimes. Since all tests run on a single compute node, observed differences were low within a range of 1-3 seconds. Varying I/O time, which we exclude from our measurements, accounts for the majority of this difference, and the remaining variations in actual compute time were only fractions of a second.

8.2 Evaluation

When designing this evaluation we set out to answer a series of questions with regard to the performance of PPP in practice.

Are the results of PPP correct? To validate results, we saved for all versions of PPP (i.e., serial, OpenMP, Thrust, and VTK-m) the superarcs generated, sorted the arcs lexicographically, and compared results using the Unix diff utility with the corresponding results of the sweep and merge algorithms. We compared results for all tiles of the GTOPO dataset as well as for the full GTOPO dataset (except for GPU due to insufficient memory) and verified that the generated contour trees match.

What is the impact of the active topology graph on compute performance? In serial, PPP (VTK-m) requires $\approx 4.5 - 21.2s$ per 6000×4800 GTOPO tile (Figure 14, top), whereas PPP (na) (i.e., the naïve implementation without the active topology graph optimizations) did not complete a single tile within the allocated 24 hour runtime. Even on $G(0.03125)$, consisting of just 675×1350 pixels, PPP (na) required at least $6567s$ in serial, whereas PPP (VTK-m) required only $0.27s$, i.e., a speed-up of $\approx 24300\times$. These results highlight the

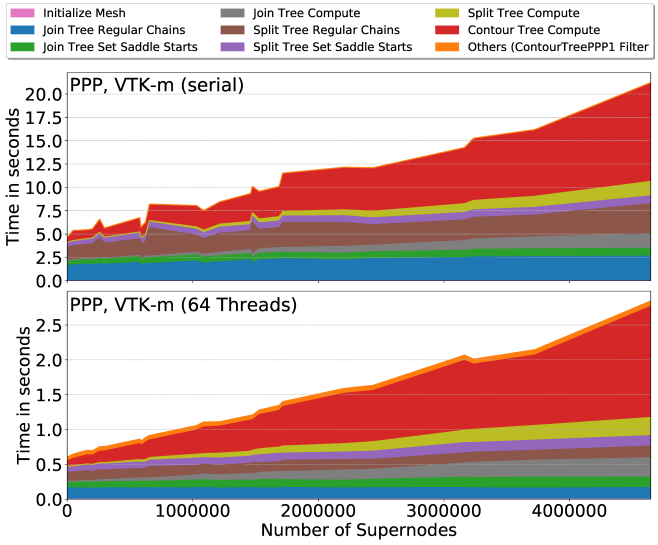


Fig. 14: Performance of PPP (VTK-m) in serial (top) and using 64 threads (bottom) across 27 GTOPO tiles with a resolution 6000×4800. The tiles are sorted along the x-axis based on their topological complexity given by the number of supernodes in the contour tree.

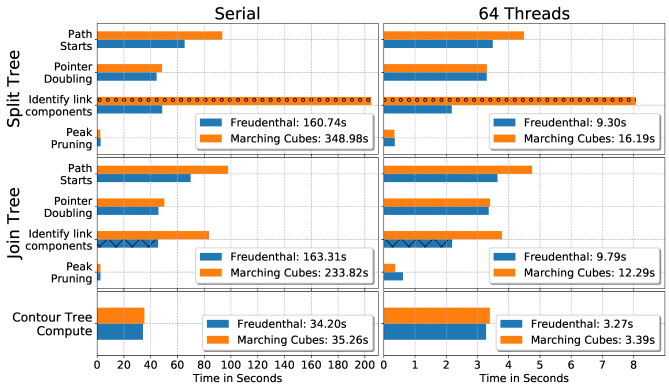


Fig. 15: Performance of the subphases of the split (top), join (middle), and contour tree (bottom) in serial (left) and using 64 threads (right) on Haswell for processing the Ex field of the $(6791 \times 371 \times 371)$ WarpX dataset. The corresponding total times are shown in the legends of the subplots. Here we divide the monotone path constructing phase into the "path starts" and "pointer doubling" sub-phases.

critical role of the algorithmic improvements described in Section 5 on performance.

What is the impact of using marching cubes connectivity on performance? To answer this question we measured the performance of PPP (VTK-m) on the 3D Warp dataset using Freudenthal (six tets per cube [6]) and marching cubes connectivity, respectively. The implementation of the contour tree phase is identical in both cases as it depends only on the join and split trees (not the input mesh). For the WarpX dataset with $(6791 \times 371 \times 371)$ nodes we see that using marching cubes is in parallel 1.74 and 1.26× slower and in serial 2.17× and 1.43× slower for the split and join tree, respectively (Figure 15), whereas the performance of the contour tree phase remains roughly the same. We also observe a combined parallel speed-up of 16× for Freudenthal and 19.4× for marching cubes. The main subphases responsible for the slow down (as well as the larger speed ups) are mainly "identify link components" and to a lesser extend "path starts". (We note that "path starts" and "pointer doubling" together constitute the "monotone path construction" phase mentioned in Section 4, while "identify link components" is the most expensive step of identifying governing saddles that are subsequently used during "peak pruning.") This behavior is expected since marching cubes connectivity considers a larger number of neighbors during these phases. Furthermore, marching cubes connectivity requires an explicit "union find" to identify distinct components in the upper link, which is considerably more expensive than using a look-up table for the Freudenthal triangulation. Also, marching cubes connectivity considers significantly more neighbors during the split tree construction, making the "identify link components" phase more expensive for split trees than for join trees. We also observe similar behavior for the smaller $(425 \times 371 \times 371)$ dataset (not shown).

How well does PPP scale with topological data complexity? To address this question we computed the contour tree independently using varying numbers of threads for the 27 GTOPO tiles with a resolution of 6000×4800 pixel. As the resolution of the data is constant, any performance differences must be attributed to variations in the topology of the data. Figure 14 shows the runtime performance of PPP (VTK-m) in serial and using 64 threads on the Haswell system. The roughly linear slope in the plots with varying number of supernodes reflects the expected dependence of PPP on topological complexity. Note, while the topological complexity increases by a factor of 19525× from the simplest tile (with just 238 supernodes) to the most complex tile (with $4,647,099$ supernodes), the compute time increases only by a factor of $\approx 4.68\times$ (from $4.53s$ to $21.23s$ in serial and $0.61s$ to $2.85s$ in parallel). Overall, these result indicate that the dependence of PPP on topological complexity is well-behaved in practice. We also observe that the breakdown of compute times across phases of PPP is similar in serial and parallel, indicating that the parallel scaling of the phases is well-behaved under varying topological complexity.

How well does PPP scale with varying data size? Figure 16 shows the performance of PPP (VTK-m) at varying numbers of threads for all scaled versions of the GTOPO dataset. Figure 17 (top) also shows the curves for $G(1.0)$ in serial and using 64 threads. First, we observe that the slope of the curves for PPP (VTK-m) in Figure 17 (top) is similar to the gray line showing the slope for perfect linear scaling. For additional detail, we can further see from Figure 16 that the increase in compute time between $G(0.5)$ to $G(1.0)$ is on the order of 3.5 to 3.86× and $\approx 4.5\times$ for $G(0.25)$ to $G(0.5)$ across all numbers of threads. As the mesh size increases by a factor of 4 between each scaled version, these results indicate that PPP scales well with increasing mesh size. Figure 18 (top) furthermore confirms that the relative times per compute phase of the algorithm are similar for the different scaled versions of GTOPO. This is expected given the similar topology of the datasets and gives further evidence that the algorithm and its different compute phases overall behave well with growing data size. Finally, Figure 18 (bottom) shows the speed-ups using 64 threads on Haswell compared to serial for all scaled GTOPO datasets. We observe that speed-ups improve across all phases of the algorithm as the data size increases, in particular from $G(0.03125)$ to $G(0.25)$, with speed-ups leveling off afterwards. This indicates that for the smaller scaled datasets ($G(0.125)$ and smaller) there is likely simply not enough work to utilize all available compute and memory resources while scaling is consistent for the larger datasets.

How well does PPP scale with varying number of threads and on different architectures? Using 64 threads we observe a 11× speed-up on Haswell and a 26.2× speed-up on KNL compared to serial for the full GTOPO dataset (Figure 17). Using all 272 threads on KNL we then observe a speed-up of 31.2×. Profiling of the Thrust-based PPP (Thrust-ref) using the VTune Performance Analyzer with one GTOPO30 tile as input indicated an overall memory bound metric of 43.4%, with the tree compression step and the monotone path construction reporting memory-bound metrics above 70%. One possible explanation for the improved scaling on KNL compared to Haswell may be that the algorithm hits the memory bound later on KNL, which is consistent with the observation that the ratio of memory clock speed to CPU clock speed is 1.7× on KNL and 0.93× on Haswell. KNL also has twice the L2 cache per core and has two 512bit vector processing

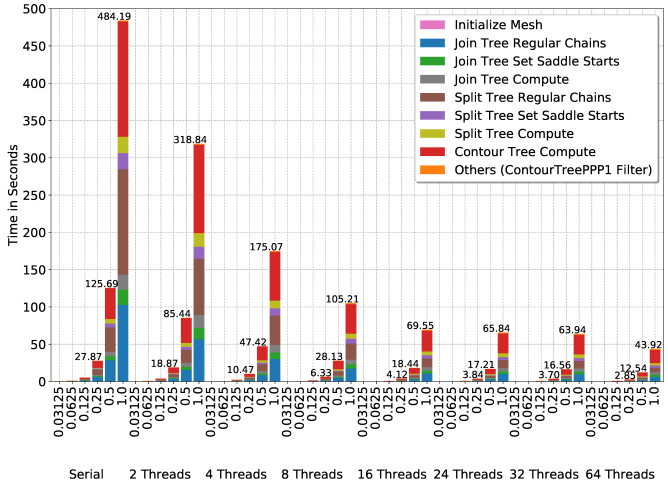


Fig. 16: PPP (VTK-m) scaling on Haswell with varying numbers of threads and data sizes. Each stacked bar shows the results for a particular scaled version of the GTOPO dataset (i.e., $G(0.03125)$ to $G(1.0)$) and number of threads used. The breakdown of the bars corresponds to the main logical phases of the PPP implementation. The total runtime is indicated by the total height of the stacked bars and for the three largest datasets also via additional text labels on top of each bar.

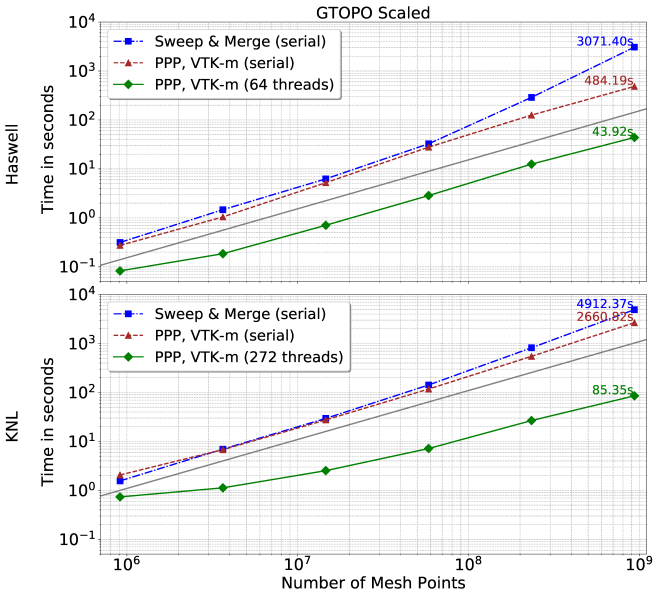


Fig. 17: Comparison of the performance of SAM and PPP, VTK-m in serial and parallel for the different scaled GTOPO datasets and on the Haswell (top) and KNL (bottom) compute system. Note the logarithmic scale on both coordinate axes. A gray line illustrates the slope for perfect linear scaling.

units compared to two 256bit units on Haswell.

Compared to PPP (OMP-ref) in serial on Haswell, PPP (Thrust-ref) on P100 is 18 to 34.4× and on average 22.4× faster for the GTOPO tiles (Figure 19). Even when PPP (OMP-ref) uses 64 threads on Haswell, PPP (Thrust-ref) on P100 is 5 to 16.6× and on average 8.13× faster (Figure 19, bottom). While CPU and GPU cores are hard to compare due to their vastly different architecture, the fact that the clock rate on the Haswell cores is 1.73× higher than on the P100 GPU cores provides some evidence that there may be more parallelism present than is immediately visible from the direct speed-up rates.

How well does PPP scale for 3D data? To evaluate the performance of PPP for 3D data, we computed the contour tree using PPP (VTK-m) on 43 3D datasets (see Appendix A) on Haswell using varying numbers of threads. Table 1 shows the runtime performance and Figure 20 the corresponding parallel speed ups for twelve of the 3D datasets. The complete times and speed-ups for PPP and TTK for all 43 datasets on Haswell are available in Appendix B-G. The overall scaling and speed ups we observe for the 3D datasets is consistent with the behavior we have seen for the 2D GTOPO datasets. For the larger 3D datasets we observe speed-ups of 12 to 13×, while our implementation shows consistent speed-ups and scaling for all of the medium ($256^3 - 400^3$) and large ($> 512^3$) 3D datasets. Unsurprisingly, we observe that the use of hyperthreading generally yields greater speed-ups for the larger datasets than for the medium-sized datasets.

How does the performance of PPP compare to Sweep and Merge? On the full GTOPO dataset (Figure 17), serial PPP is already 6.3× faster than SAM. Using 64 threads on Haswell, we then observe an $\approx 70\times$ speed-up for PPP compared to the serial SAM. Overall, we see that PPP is consistently significantly faster than SAM in parallel and even in serial PPP is faster than SAM across all scales of the GTOPO dataset on Haswell and on KNL for all but the very small $G(0.03125)$. When comparing the slope of the curves in Figure 17, we also observe that PPP scales significantly better with growing data size than SAM.

With regard to topological complexity, we observe that PPP (VTK-m) is already in serial on average 1.34× faster than SAM across all 27 tiles of the GTOPO datasets with a resolution of 6000×4800 (and 2.3× faster for gt30e020n90 where SAM is slowest). Using 64 threads, PPP (VTK-m) is than between $8.8 - 15.2\times$ and on average 10.2× faster than the serial SAM.

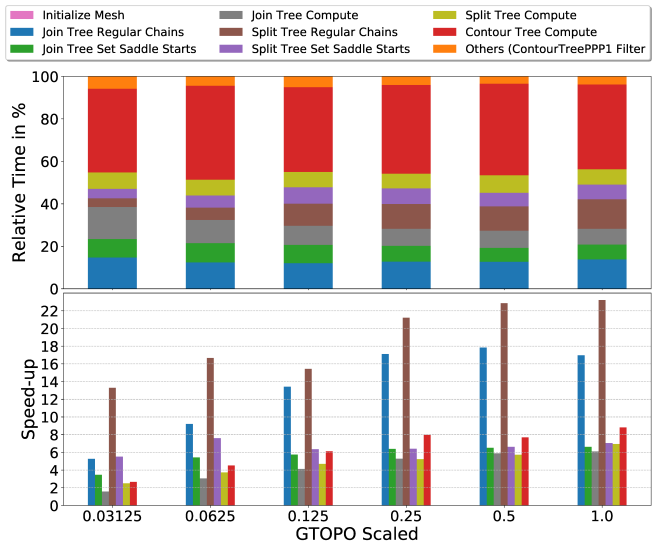


Fig. 18: Analysis of the relative time per compute phase (top) and speed-up compared to serial (bottom) for PPP, VTK-m (64 threads) for all scaled GTOPO datasets. The corresponding absolute compute performance times are shown in Figure 17 (top, green curve) as well as Figure 16 (right-most bars labeled "64 Threads").

How does the performance of PPP compare to TTK? The topology toolkit (TTK) provides a task-based parallel implementation of the contour tree [18]. The TTK implementation computes the augmented contour tree (i.e., the contour tree including all regular nodes) while PPP computes the contour tree without augmentation. As such, TTK has to perform a certain amount of additional work, so any comparison should assume a significant margin for leeway.

Table 1 shows an overview of the performance results for PPP and TTK on Haswell for 13 3D datasets of varying size. We present the complete scaling results for all 43 3D datasets in Appendix B-H.

Across all 3D datasets we observe that PPP outperforms TTK in se-

		64 Threads			32 Threads			16 Threads			4 Threads			Serial		
name	**shape**	**PPP**	**TTK**	↑	**PPP**	**TTK**	↑	**PPP**	**TTK**	↑	**PPP**	**TTK**	↑	**PPP**	**TTK**	↑
tacc turbulence	256^3	0.89	5.02	5.63	0.97	3.75	3.85	1.09	3.66	3.36	2.77	8.77	3.16	9.20	21.70	2.36
aneurism	256^3	0.50	2.15	4.33	0.60	2.20	3.67	0.63	2.20	3.49	1.54	3.21	2.09	5.22	8.63	1.66
bonsai	256^3	0.64	2.78	4.34	0.74	2.06	2.80	0.80	1.86	2.34	2.05	3.77	1.84	6.89	8.42	1.22
skull	256^3	1.94	19.76	10.20	2.16	12.48	5.77	2.35	9.79	4.15	5.59	14.07	2.52	19.67	35.14	1.79
mrt angio	$516 \times 512 \times 112$	5.61	59.38	10.58	6.44	36.00	5.59	6.78	29.12	4.30	15.03	40.63	2.70	52.26	71.84	1.37
warpx ex (small)	425×371^2	2.19	19.84	9.07	2.60	22.41	8.62	2.91	22.32	7.68	7.72	42.99	5.57	27.43	127.65	4.65
prone	$512^2 \times 463$	17.15	156.11	8.91	20.68	92.65	4.48	21.93	80.50	3.67	49.62	139.14	2.80	169.27	300.99	1.77
asteroid	500^3	3.91	22.48	5.74	4.83	20.11	4.16	5.08	20.64	4.06	12.92	45.74	3.54	45.23	94.13	2.08
vertebra	512^3	6.81	44.78	6.57	8.01	30.76	3.84	8.92	29.98	3.36	21.91	65.84	3.01	77.14	157.12	2.04
magnetic reconnection	512^3	42.06	881.57	20.96	47.92	777.47	16.22	51.04	777.34	15.23	115.07	934.53	8.12	394.36	1108.90	2.81
kingsnake	$1024^2 \times 795$	92.55	OOM	—	121.54	OOM	—	130.37	OOM	—	295.01	OOM	—	999.99	OOM	—
warpx ex (large)*	6791×371^2	19.51	OOM	—	22.85	OOM	—	30.55	OOM	—	92.62	OOM	—	325.28	OOM	—
nyx*	1024^3	250.49	OOM	—	262.51	OOM	—	330.06	OOM	—	874.86	OOM	—	3589.51	OOM	—

Table 1: Performance of PPP and TTK on 3D data for varying numbers of threads on Haswell. For each number of threads we show the runtime for PPP and TTK in seconds as well as the corresponding speed-up (↑) of PPP compared to TTK (OOM=out-of-memory). Datasets marked with * were processed on a Cori login node, which has the same configuration as the Haswell compute nodes but with 512GB of main memory.

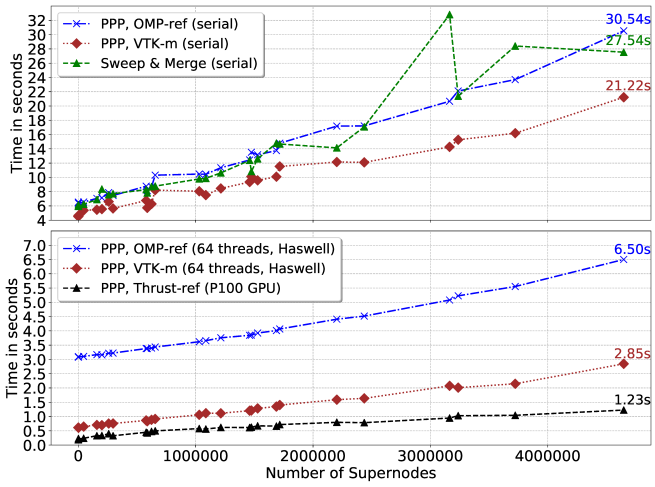


Fig. 19: Comparison of the performance of the different algorithms across 27 tiles of the GTOPO dataset. Each tile has a resolution 6000 × 4800. The tiles are sorted along the x-axis of the plot based on the complexity of their topology indicated by the number of supernodes in the contour tree. For PPP, Thrust-ref we used the P100 system and for all other results we used the Haswell system.

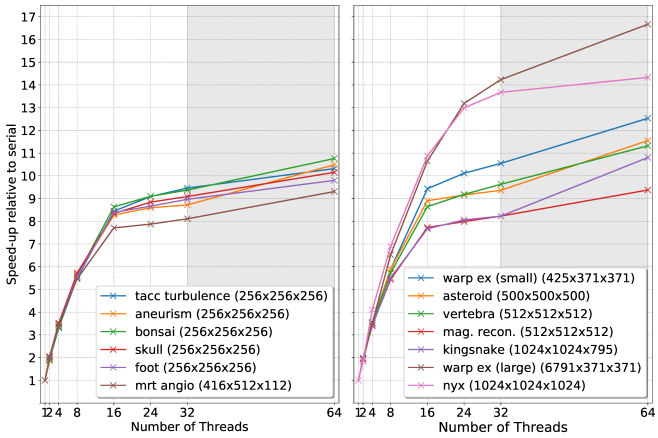


Fig. 20: Speed up compared to serial on Haswell for PPP on 6 medium (left) and 7 large (right) 3D datasets See Tab. 1 for the corresponding runtime performance values The gray area denotes the use of 2 threads per core (i.e., hyperthreading).

rial between 1.1× to 5.5×. In parallel, we observe that TTK achieves a maximum speed-up of 1.5× to 11.8× depending on the 3D datasets. Here, TTK achieved its best performance for most datasets (20 of 34) with 16 or fewer threads on Haswell. For larger numbers of threads we then typically observe a reduction in performance for TTK.

In contrast, for PPP we observe maximum speed-ups of 1.5× to 17.5× depending on the dataset. For all dataset larger than 64^3, PPP then shows a maximum speed-up of at least 4.1× and on average 10.9×. Further, for all datasets larger than 128^3, PPP achieved its best performance at the maximum available 64 threads and showed continued improvement in performance with increasing number of threads. With this improved scaling, the best PPP time is between 2.4× and 18× faster than the best TTK time for all dataset larger than 64^3. When comparing performance with the maximum number of 64 threads, PPP is between 1.4× to 21× faster. Allowing for the differences in the computation, we can therefore make the reasonable claim that PPP as embedded in VTK-m is at least as efficient as the TTK implementation in serial while showing significantly better parallel scaling and accordingly much faster performance in parallel.

How does the performance of PPP compare to Distributed Merge Trees In addition to testing against the TTK implementation of the contour tree computation, we also ran some preliminary tests against the Distributed Merge Trees (DMT) computation [29], which was designed to run on a distributed cluster rather than in shared memory. In serial, however, it collapses to the sweep and merge algorithm for computing the merge tree.

For the mrt angio dataset ($416 \times 512 \times 112$), DMT took 68 seconds for the fully augmented merge tree, where a serial run of PPP took 18.14 seconds. We also ran it on the large WarpX Ex data set ($6791 \times 371 \times 371$) with 64 tasks and compared it with PPP running on 64 threads. As shown in Table 1, PPP took a total of 19.51 seconds for the contour tree: this included 8.34 seconds for the join tree, 8.8 seconds for the split tree, and 2.3 seconds to combine them into the contour tree. Against this, DMT took around 100 seconds to compute the join

tree, the bulk of which (90 seconds) was used to compute 64 join trees on $1/64$ of the data each, where 10 seconds were used to reconcile them into the distributed structure.

Since the two computations are significantly different, the strongest conclusion we are comfortable drawing at present is that DMT is less efficient than sweep and merge when run in serial on a single node, and that it appears to be considerably slower than PPP.

What is the impact of VTK-m on performance? In serial, we observe that PPP (VTK-m) is between $1.19 - 1.46\times$ and on average $1.35\times$ faster than the reference PPP (OMP-ref) (Figure 19, top). Possible sources for this gain in performance may be differences in performance between VTK-m and C++ standard library (STL) data structures and algorithms and differences in auto-vectorization. In parallel using 64 threads the VTK-m implementation is between $2.3 - 5.1\times$ and on average $2.7\times$ faster than the reference PPP (OMP-ref) implementation. This additional gain in performance is likely due to differences in scaling caused by the use of TBB compared to OpenMP.

9 Conclusions & Future Work

We have provided an extended description and benchmarks of the first pure data-parallel algorithm for the merge tree and contour tree in unaugmented (canonical) form [12] with strong guarantees on computation time, and practical performance faster than the sweep and merge algorithm, with parallel speedup of at least one order of magnitude on GPU. We have further extended this algorithm to 3D data, both for Freudenthal triangulations and marching cubes connectivity. Our implementation is being released through VTK-m: while we have targetted regular mesh structures, we have engineered it so that minimal changes will be required for computations on irregular meshes in the future.

In future work, we will improve performance for computing the fully augmented contour tree, extend the scalability with a hybrid distributed/ data-parallel stage, and add geometric computation and simplification to allow the contour tree to be used for *in situ* analysis.

Acknowledgements

We would like to acknowledge EPSRC Grant EP/J013072/1 and the University of Leeds for supporting the first author's study leave at Los Alamos National Laboratory. This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 to the Lawrence Berkeley National Laboratory ("Towards Exascale: High Performance Visualization and Analytics Program") and under Award Number 14-017566 at Los Alamos National Laboratory ("XVis: Visualization for the Extreme-Scale Scientific-Computation Ecosystem"), with Lucy Nowell the program manager for both awards. This research used resources of the University of Leeds Advanced Research Computing facility and 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 Li-ta Lo for his contributions. 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)*, pages 271–278, Apr. 2015.

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

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

[4] H. Carr. *Topological Manipulation of Isosurfaces*. PhD thesis, University of British Columbia, Vancouver, BC, Canada, 2004.

[5] H. Carr and D. Duke. Joint Contour Nets. *IEEE Transactions on Visualization and Computer Graphics*, 20(8):1100–1113, 2014.

[6] H. Carr, T. Möller, and J. Snoeyink. Artifacts Caused by Simplicial Subdivision. *IEEE Transactions on Visualization and Computer Graphics*, 12(2):231–242, March/April 2006.

[7] 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.

[8] H. Carr and J. Snoeyink. Representing Interpolant Topology for Contour Tree Computation. In H.-C. Hege, K. Polthier, and G. Scheuermann, editors, *Topology-Based Methods in Visualization II*, Mathematics and Visualization, pages 59–74. Springer, 2009.

[9] H. Carr, J. Snoeyink, and U. Axen. Computing Contour Trees in All Dimensions. *Computational Geometry: Theory and Applications*, 24(2):75–94, 2003.

[10] 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.

[11] H. Carr, J. Tierny, and G. H. Weber. Pathological and Test Cases For Reeb Analysis. Accepted for Publication.

[12] H. Carr, G. H. Weber, C. Sewell, and J. Ahrens. Parallel Peak Pruning for Scalable SMP Contour Tree Computation. In *Large Scale Data Analysis and Visualization*, pages 75–84, 2016.

[13] 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.

[14] H. Edelsbrunner, J. Harer, and A. K. Patel. Reeb Spaces of Piecewise Linear Mappings. In *Proceedings of ACM Symposium on Computational Geometry*, pages 242–250., 2008.

[15] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological Persistence and Simplification. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 454–463. IEEE, 2000.

[16] 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.

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

[18] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Task-based Augmented Merge Trees with Fibonacci Heaps. In *Large Scale Data Analysis and Visualization*, 2017.

[19] F. Guo, H. Li, W. Daughton, and Y.-H. Liu. Formation of hard power laws in the energetic particle spectra resulting from relativistic magnetic reconnection. *Phys. Rev. Lett.*, 113:155005, Oct 2014.

[20] K. Heitmann, N. Frontiere, C. Sewell, S. Habib, A. Pope, H. Finkel, S. Rizzi, J. Insley, and S. Bhattacharya. The Q Continuum Simulation: Harnessing the Power of GPU Accelerated Supercomputers. To appear in the Astrophysical Journal Supplement., 2015.

[21] M. Hilaga, Y. Shinagawa, T. Kohmura, and T. L. Kunii. Topology Matching for Fully Automatic Similarity Estimation of 3D Shapes. *ACM Transactions on Graphics*, pages 203–212, 2001.

[22] P. Hristov and H. Carr. W-Structures in Contour Trees. In preparation.

[23] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[24] 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*, pages 1020–1031, Nov. 2014.

[25] L.-T. Lo, C. Sewell, and J. Ahrens. PISTON: A Portable Cross-Platform Framework for Data-Parallel Visualization Operators. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, pages 11–20, 2012.

[26] W. E. Lorenson and H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21(4):163–169, 1987.

[27] S. Maadasamy, H. Doraiswamy, and V. Natarajan. A hybrid parallel algorithm for computing and tracking level set topology. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–10. IEEE, Dec. 2012.

[28] C. Montani, R. Scateni, and R. Scopigno. A modified look-up table for implicit disambiguation of Marching Cubes. *Visual Computer*, 10:353–355, 1994.

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

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

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

[32] V. Pascucci, K. Cole-McLaughlin, and G. Scorzelli. *The TOPORRERY: Computation and Presentation of Multi-Resolution Topology*, pages 19–40. Springer-Verlag, Berlin Heidelberg, Germany, 2009. Preliminary version appeared in the proceedings of the IASTED conference on Visualization, Imaging, and Image Processing (VIIP 2004), 2004, pp.452-290.

[33] J. Patchett and G. Gisler. Deep water impact ensemble data set. Technical Report LA-UR-17-21595, Los Alamos National Laboratory, 2017.

[34] 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.

[35] C. Sewell, K. Heitmann, L.-T. Lo, S. Habib, and J. Ahrens. Utilizing Many-Core Accelerators for Halo and Center Finding within a Cosmology Simulation. In submission., 2015.

[36] C. Sewell, L.-T. Lo, and J. Ahrens. Portable Data-Parallel Visualization and Analysis in Distributed Memory Environments. In *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, pages 25–33, 2013.

[37] S. Takahashi, T. Ikeda, Y. Shinagawa, T. L. Kunii, and M. Ueda. Algorithms for Extracting Correct Critical Points and Constructing Topological Graphs from Discrete Geographical Elevation Data. *Computer Graphics Forum*, 14(3):C–181–C–192, 1995.

[38] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

[39] 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*, pages 212–220, 1997.

[40] M. J. van Kreveld, R. van Oostrum, C. L. Bajaj, V. Pascucci, and D. R. Schikore. *Efficient contour tree and minimum seed set construction*, pages 71–86. John Wiley & Sons, May 2004.

[41] W. Widanagamaachchi, C. Christensen, P.-T. Bremer, and V. Pascucci. Interactive Exploration of Large-Scale Time-Varying Data Using Dynamic Tracking Graphs. In *Proceedings of Large-Scale Data Analysis and Visualization (LDAV)*, pages 9–17, 2012.

APPENDIX

A SHAPE AND SIZE OF THE 3D DATASETS USED IN THE PERFORMANCE EVALUATION

name	source	shape	# nodes	# supernodes
marschner lobb	Marschner and Lobb	[41, 41, 41]	68,921	1,506
nucleon	SFB 382 of the German Research Council (DFG)	[41, 41, 41]	68,921	579
silicium	VolVis distribution of SUNY Stony Brook, NY, USA	[98, 34, 34]	113,288	458
neghip	VolVis distribution of SUNY Stony Brook, NY, USA	[64, 64, 64]	262,144	2,242
fuel	SFB 382 of the German Research Council (DFG)	[64, 64, 64]	262,144	344
tooth	TBD	[103, 94, 161]	1,558,802	231,242
shockwave	TBD	[64, 64, 512]	2,097,152	1,133
hydrogen atom	SFB 382 of the German Research Council (DFG)	[128, 128, 128]	2,097,152	13,593
lobster	VolVis distribution of SUNY Stony Brook, NY, USA	[301, 324, 56]	5,461,344	323,349
mri ventricles	Dirk Bartz, VCM, University of Tübingen, Germany	[256, 256, 124]	8,126,464	1,562,438
engine	General Electric	[256, 256, 128]	8,388,608	467,702
statue leg	German Federal Institute for Material Research and Testing (BAM), Berlin, Germany	[341, 341, 93]	10,814,133	353,877
tacc turbulence	Gregory D. Abram and Gregory P. Johnson, Texas Advanced Computing Center, The University of Texas at Austin. Simulation by Diego A. Donzis, Texas A&M University, P.K. Yeung, Georgia Tech	[256, 256, 256]	16,777,216	313,281
aneurism	Philips Research, Hamburg, Germany	[256, 256, 256]	16,777,216	54,197
bonsai	S. Roettger, VIS, University of Stuttgart	[256, 256, 256]	16,777,216	179,627
skull	Siemens Medical Solutions, Forchheim, Germany	[256, 256, 256]	16,777,216	1,710,477
foot	Philips Research, Hamburg, Germany	[256, 256, 256]	16,777,216	719,091
mrt angio	Özlem Gürvit, Institute for Neuroradiology, Frankfurt, Germany	[416, 512, 112]	23,855,104	4,818,339
stent	Michael Meißner, Viatronix Inc., USA	[512, 512, 174]	45,613,056	2,856,825
warpx small Ez	WarpX collaboration	[425, 371, 371]	58,497,425	111,396
warpx small Ex	WarpX collaboration	[425, 371, 371]	58,497,425	358,203
warpx small rho	WarpX collaboration	[425, 371, 371]	58,497,425	106,908
warpx small Ey	WarpX collaboration	[425, 371, 371]	58,497,425	100,467
pancreas	Roth HR, Lu L, Farag A, Shin H-C, Liu J, Turkbey EB, Summers RM. DeepOrgan: Multi-level Deep Convolutional Networks for Automated Pancreas Segmentation	[240, 512, 512]	62,914,560	6,682,631
bunny	Stanford Radiology & Computer Science Departments	[512, 512, 361]	94,633,984	11,110,783
backpack	Kevin Kreeger, Viatronix Inc., USA	[512, 512, 373]	97,779,712	5,693,268
present	Christoph Heinzl, 2006	[492, 492, 442]	106,992,288	11,547,958
neocortical layer 1 axons	V De Paola, MRC Clinical Sciences Center, Imperial College London	[1464, 1033, 76]	114,935,712	9,289,314
prone	Walter Reed Army Medical Center, USA	[512, 512, 463]	121,372,672	12,087,883
asteroid	John Patchett and Galen Gisler, Los Alamos National Laboratory [33]	[500, 500, 500]	1250,00,000	804,757
christmas tree	Armin Kanitsar, 2002	[512, 499, 512]	130,809,856	19,962,839
vertebra	Michael Meißner, Viatronix Inc., USA	[512, 512, 512]	134,217,728	2,808,594
magnetic reconnection	Bill Daughton (LANL) and Berk Geveci (KitWare) [19]	[512, 512, 512]	134,217,728	27,860,405
marmoset neurons	Frederick Federer, Moran Eye Institute, University of Utah	[1024, 1024, 314]	329,252,864	48,399,592
stag beetle	Meister Eduard GrÏler, Georg Glaeser, Johannes Kastner, 2005	[832, 832, 494]	341,958,656	712,098
pawpawsaurus	Matthew Colbert, 4 February 2014	[958, 646, 1088]	673,328,384	76,373,336
spathorhynchus	Matthew Colbert, 17 February 2005	[1024, 1024, 750]	786,432,000	39,376,047
kingsnake	DigiMorph.org, The University of Texas High-Resolution X-ray CT Facility (UTCT), and NSF grant IIS-9874781	[1024, 1024, 795]	833,617,920	50,552,413
warpx large Ey	WarpX collaboration	[6791, 371, 371]	934,720,031	330,912
warpx large rho	WarpX collaboration	[6791, 371, 371]	934,720,031	322,028
warpx large Ez	WarpX collaboration	[6791, 371, 371]	934,720,031	378,067
warpx large Ex	WarpX collaboration	[6791, 371, 371]	934,720,031	242,442
Nyx particle mass density	Zarija Lukić, Center for Computational Cosmology, Lawrence Berkeley National Laboratory	[1024, 1024, 1024]	1,073,741,824	144,464,050

Table A1: Overview of the shape and size of the 3D datasets used in the performance evaluation with datasets sorted by size. Most data sets are from the Open SciVis Dataset page (`https://klacansky.com/open-scivis-datasets/`). The WarpX and Nyx data sets are courtesy of collaborators at Lawrence Berkeley National Laboratory and not publically available. The asteroid data set is available from the Los Alamos National Laboratory (`https://dssdata.org`).

B PPP RESULTS FOR 3D DATASETS ON HASWELL

	64	32	24	16	8	4	2	1
marschner lobb	0.033055	0.030258	0.031103	0.027123	0.028659	0.031116	0.036576	0.039753
nucleon	0.031096	0.026896	0.026699	0.026854	0.024182	0.027759	0.034495	0.036596
silicium	0.034715	0.029465	0.029583	0.032216	0.028851	0.034026	0.043325	0.050460
neghip	0.043687	0.037691	0.037577	0.040677	0.041616	0.052291	0.068589	0.097084
fuel	0.039960	0.036719	0.037327	0.038628	0.040577	0.047113	0.063771	0.085372
tooth	0.267086	0.243260	0.248558	0.254357	0.342220	0.518939	0.816788	1.591930
shockwave	0.097280	0.094213	0.093131	0.095394	0.130586	0.187419	0.317553	0.587685
hydrogen atom	0.109471	0.108616	0.107407	0.111205	0.148911	0.216281	0.356003	0.621548
lobster	0.465318	0.478225	0.487549	0.512951	0.724602	1.132370	1.912070	3.564980
mri ventricles	1.481870	1.623060	1.691120	1.745950	2.498200	3.986720	6.927940	14.049000
engine	0.667967	0.717370	0.740272	0.785006	1.139780	1.787140	3.105430	6.028240
statue leg	0.603635	0.653635	0.673348	0.695886	1.036650	1.669470	2.894090	5.458320
tacc turbulence	0.891400	0.971992	1.011880	1.089030	1.668750	2.771720	4.841090	9.196700
aneurism	0.497326	0.598067	0.606651	0.630344	0.921900	1.537790	2.795060	5.215160
bonsai	0.640259	0.735186	0.757202	0.798286	1.228990	2.050770	3.577160	6.890100
skull	1.937480	2.164390	2.224800	2.357210	3.437780	5.590730	9.591030	19.672300
foot	1.081000	1.181380	1.222440	1.262570	1.898590	3.062520	5.324510	10.592500
mrt angio	5.615060	6.444860	6.636290	6.779310	9.555340	15.031600	25.812000	52.262700
stent	3.942580	4.485490	4.674960	4.850010	7.195720	11.500400	20.411000	41.098100
warpx small Ez	1.863410	2.244090	2.319450	2.459290	3.870990	6.617620	11.867700	23.476900
warpx small rho	1.747890	2.122500	2.189420	2.303060	3.623980	6.133930	11.015500	21.554800
warpx small Ex	2.188610	2.600140	2.712100	2.907770	4.653710	7.716100	13.842300	27.427500
warpx small Ey	1.822880	2.248890	2.339130	2.482620	3.924610	6.682670	12.007900	23.438100
pancreas	8.418060	10.037000	10.324500	10.664300	15.006200	23.516300	41.645100	81.438400
bunny	15.296000	17.878300	18.425300	18.800800	26.908300	42.765500	72.919300	144.755000
backpack	9.077100	10.402100	10.779200	11.118400	15.877700	25.664500	44.431200	86.976700
present	16.814100	19.730600	20.246400	20.893400	29.354200	45.740100	78.807700	161.914000
neocortical layer 1 axons	13.962900	15.741300	16.328500	16.895600	24.072200	38.760300	67.291800	131.801000
prone	17.512400	20.680000	21.173900	21.929300	31.087700	49.612900	86.311500	169.273000
asteroid	3.915640	4.831200	4.939870	5.079580	7.670890	12.924200	23.023200	45.232200
christmas tree	28.032000	32.277800	33.110200	33.840500	46.654300	72.853800	125.615000	244.865000
vertebra	6.814380	8.012720	8.401800	8.921680	13.465800	21.907900	39.625500	77.141100
magnetic reconnection	42.059000	47.922300	49.375900	51.038000	72.470400	115.074000	204.075000	394.356000
marmoset neurons	72.946000	83.369200	85.129100	87.734900	123.638000	200.107000	344.231000	678.229000
stag beetle	9.713370	11.887400	11.998100	12.258100	18.675600	31.438200	58.305700	112.556000
pawpawsaurus	116.667000	154.088000	158.855000	161.947000	237.537000	379.057000	659.068000	1301.130000
spathorhynchus*	78.139400	80.754100	88.111600	102.787000	157.699000	255.936000	437.597000	844.489000
kingsnake	92.553600	121.540000	124.073000	130.367000	181.377000	295.014000	502.689000	999.994000
warpx large Ey*	19.926100	22.028700	25.209700	31.117700	52.116100	99.732400	185.629000	332.137000
warpx large rho*	19.398200	22.225600	25.102500	30.858400	50.491600	94.365700	180.433000	325.116000
warpx large Ez*	19.698200	22.551600	25.811100	31.523200	51.664700	94.584300	182.051000	345.387000
warpx large Ex*	19.514700	22.853700	24.664400	30.554600	49.821300	92.615300	176.965000	325.281000
Nyx particle mass density*	250.491000	262.512000	276.267000	330.055000	520.232000	874.856000	1950.600000	3589.510000

Table A2: PPP runtime in seconds on Haswell for all 3D datasets. We repeated each evaluation 5 times and report here the best time. Datasets marked with * were processed on a Cori login node, which has the same configuration as the Haswell compute node but with 512GB of main memory, to accommodate the larger memory requirements to process the largest files.

C PPP Speed-up for 3D datasets on Haswell

	64	32	24	16	8	4	2	1
marschner lobb	1.202617	1.313797	1.278088	1.465629	1.387084	1.277562	1.086851	1.0
nucleon	1.176891	1.360659	1.370709	1.362787	1.513382	1.318338	1.060895	1.0
silicium	1.453571	1.712511	1.705681	1.566298	1.748980	1.483001	1.164683	1.0
neghip	2.222263	2.575780	2.583574	2.386687	2.332853	1.856607	1.415441	1.0
fuel	2.136447	2.325041	2.287120	2.210095	2.103956	1.812084	1.338734	1.0
tooth	5.960365	6.544150	6.404662	6.258644	4.651774	3.067663	1.949012	1.0
shockwave	6.041170	6.237827	6.310318	6.160633	4.500368	3.135675	1.850667	1.0
hydrogen atom	5.677741	5.722435	5.786848	5.589209	4.173956	2.873798	1.745907	1.0
lobster	7.661384	7.454608	7.312045	6.949943	4.919915	3.148247	1.864461	1.0
mri ventricles	9.480589	8.655872	8.307512	8.046622	5.623649	3.523950	2.027876	1.0
engine	9.024757	8.403251	8.143277	7.679228	5.288950	3.373121	1.941193	1.0
statue leg	9.042418	8.350716	8.106239	7.843699	5.265345	3.269493	1.886023	1.0
tacc turbulence	10.317142	9.461703	9.088726	8.444855	5.511131	3.318048	1.899717	1.0
aneurism	10.486401	8.720026	8.596640	8.273514	5.656969	3.391334	1.865849	1.0
bonsai	10.761426	9.371914	9.099421	8.631117	5.606311	3.359762	1.926137	1.0
skull	10.153550	9.089074	8.842278	8.345587	5.722385	3.518735	2.051114	1.0
foot	9.798797	8.966209	8.665047	8.389634	5.579140	3.458753	1.989385	1.0
mrt angio	9.307594	8.109206	7.875289	7.709147	5.469476	3.476855	2.024744	1.0
stent	10.424164	9.162455	8.791113	8.473818	5.711465	3.573624	2.013527	1.0
warpx small Ez	12.598891	10.461657	10.121753	9.546210	6.064831	3.547635	1.978218	1.0
warpx small rho	12.331897	10.155383	9.844982	9.359200	5.947825	3.514028	1.956770	1.0
warpx small Ex	12.531927	10.548470	10.113012	9.432486	5.893685	3.554581	1.981426	1.0
warpx small Ey	12.857731	10.422075	10.020007	9.440873	5.972084	3.507296	1.951890	1.0
pancreas	9.674248	8.113819	7.887878	7.636544	5.426984	3.463062	1.955534	1.0
bunny	9.463585	8.096687	7.856317	7.699406	5.379567	3.384855	1.985140	1.0
backpack	9.581992	8.361456	8.068938	7.822771	5.477916	3.388989	1.957559	1.0
present	9.629656	8.206238	7.997175	7.749529	5.515872	3.539870	2.054545	1.0
neocortical layer 1 axons	9.439371	8.372943	8.071838	7.800907	5.475237	3.400412	1.958649	1.0
prone	9.665894	8.185348	7.994418	7.719033	5.445015	3.411875	1.961187	1.0
asteroid	11.551675	9.362519	9.156557	8.904713	5.896604	3.499807	1.964636	1.0
christmas tree	8.735195	7.586174	7.395455	7.235856	5.248498	3.361046	1.949329	1.0
vertebra	11.320340	9.627330	9.181497	8.646477	5.728668	3.521154	1.946754	1.0
magnetic reconnection	9.376257	8.229071	7.986811	7.726713	5.441615	3.426977	1.932407	1.0
marmoset neurons	9.297686	8.135247	7.967064	7.730436	5.485603	3.389332	1.970273	1.0
stag beetle	11.587739	9.468513	9.381152	9.182173	6.026901	3.580230	1.930446	1.0
pawpawsaurus	11.152511	8.444071	8.190677	8.034295	5.477589	3.432544	1.974197	1.0
spathorhynchus	10.807467	10.457537	9.584311	8.215913	5.355069	3.299610	1.929833	1.0
kingsnake	10.804485	8.227695	8.059723	7.670607	5.513345	3.389649	1.989290	1.0
warpx large Ey	16.668440	15.077467	13.174968	10.673572	6.373021	3.330282	1.789252	1.0
warpx large rho	16.760112	14.627997	12.951539	10.535737	6.439012	3.445277	1.801866	1.0
warpx large Ez	17.533937	15.315410	13.381336	10.956597	6.685164	3.651631	1.897199	1.0
warpx large Ex	16.668511	14.233188	13.188279	10.645893	6.528954	3.512173	1.838109	1.0
Nyx particle mass density	14.329896	13.673699	12.992902	10.875490	6.899825	4.102972	1.840208	1.0

Table A3: PPP speed-up compared to serial on Haswell for all 3D datasets. See Appendix B for the corresponding timings.

D PPP RESULTS FOR 3D DATASETS ON KNL

	272	64	32	16	8	1
marschner lobb	0.218029	0.083152	0.083680	0.082941	0.084798	0.175909
nucleon	0.171768	0.069165	0.069328	0.079099	0.071143	0.159650
silicium	0.169682	0.083132	0.083929	0.086475	0.080865	0.238604
neghip	0.266221	0.096589	0.096296	0.102804	0.130781	0.533827
fuel	0.195899	0.081277	0.083251	0.100624	0.116542	0.511699
tooth	1.867430	0.764428	0.774809	0.954735	1.394060	7.995990
shockwave	0.543238	0.214694	0.231911	0.341925	0.572874	3.968180
hydrogen atom	0.788345	0.255941	0.271915	0.382899	0.620761	4.149850
lobster	3.490710	1.241450	1.389550	1.887140	3.014330	19.525300
mri ventricles	7.221250	3.489950	4.077420	5.816410	9.654530	66.537300
engine	4.415880	1.678270	1.943990	2.777880	4.594360	30.638600
statue leg	4.080970	1.492490	1.719010	2.520060	4.306190	29.492300
tacc turbulence	4.784990	2.014940	2.424700	3.757840	6.581700	46.564300
aneurism	2.572440	0.990454	1.342770	2.336510	4.414440	33.832100
bonsai	4.110850	1.354680	1.762940	2.888570	5.239290	38.565800
skull	8.397010	4.320860	5.160200	7.556220	12.854200	89.934200
foot	5.563140	2.565990	3.045920	4.587310	7.849450	55.026300
mrt angio	16.969500	10.171000	12.179100	18.137300	31.672600	229.592000
stent	13.120000	7.147150	9.318360	14.934100	26.700400	197.327000
warpx small Ez	6.432710	3.427210	5.186660	9.210940	17.596200	135.205000
warpx small rhow	5.720800	3.085190	4.856140	8.754000	16.788400	129.846000
warpx small Ex	7.983250	4.175520	6.047640	10.492100	19.696400	149.263000
warpx small Ey	5.882770	3.416140	5.267330	9.407430	18.123800	139.202000
pancreas	20.680000	13.174500	17.258100	27.691100	49.470400	359.771000
bunny	31.624500	22.063800	29.156000	47.135700	85.452900	632.973000
backpack	22.696700	15.054300	19.978500	31.758800	57.158600	416.063000
present	34.516700	25.162500	32.908500	52.947800	94.975300	708.680000
neocortical layer 1 axons	30.848300	21.947200	28.349400	44.635700	79.366200	579.790000
prone	34.981000	24.836100	33.474400	55.221800	99.523300	738.359000
asteroid	10.286100	6.906800	10.598400	19.046900	36.157000	275.539000
christmas tree	54.173600	39.821500	51.445000	81.023400	144.990000	1060.290000
vertebra	21.007100	11.795500	17.119300	27.991600	52.008900	387.730000
magnetic reconnection	67.358500	53.246100	71.011800	117.312000	207.845000	1553.140000
marmoset neurons	116.085000	97.842400	129.522000	203.693000	366.471000	2804.410000
stag beetle	15.669100	14.915100	25.613000	48.529200	94.083600	748.051000
pawpawsaurus	OOM	OOM	OOM	OOM	OOM	OOM
spathorhynchus	OOM	OOM	OOM	OOM	OOM	OOM
kingsnake	OOM	OOM	OOM	OOM	OOM	OOM
warpx large Ey	OOM	OOM	OOM	OOM	OOM	OOM
warpx large rho	OOM	OOM	OOM	OOM	OOM	OOM
warpx large Ez	OOM	OOM	OOM	OOM	OOM	OOM
warpx large Ex	OOM	OOM	OOM	OOM	OOM	OOM
Nyx particle mass density	OOM	OOM	OOM	OOM	OOM	OOM

Table A4: PPP runtime in seconds on KNL for all 3D datasets. We repeated each evaluation 5 times and report here the best time. OOM indicates that PPP did not complete computation of the contour tree in any of the 5 tries due to out-of-memory error.

E PPP Speed-up for 3D datasets on KNL

	272	64	32	16	8	1
marschner lobb	0.806815	2.115504	2.102171	2.120891	2.074440	1.0
nucleon	0.929451	2.308235	2.302818	2.018367	2.244072	1.0
silicium	1.406183	2.870179	2.842920	2.759235	2.950653	1.0
neghip	2.005202	5.526812	5.543605	5.192668	4.081839	1.0
fuel	2.612055	6.295780	6.146475	5.085258	4.390683	1.0
tooth	4.281815	10.460096	10.319950	8.375088	5.735757	1.0
shockwave	7.304680	18.482957	17.110788	11.605411	6.926794	1.0
hydrogen atom	5.264002	16.214088	15.261571	10.837976	6.685101	1.0
lobster	5.593504	15.727818	14.051527	10.346503	6.477493	1.0
mri ventricles	9.214097	19.065402	16.318481	11.439582	6.891822	1.0
engine	6.938277	18.256061	15.760678	11.029490	6.668742	1.0
statue leg	7.226787	19.760467	17.156561	11.703015	6.848815	1.0
tacc turbulence	9.731327	23.109522	19.204149	12.391241	7.074813	1.0
aneurism	13.151755	34.158174	25.195752	14.479758	7.663962	1.0
bonsai	9.381466	28.468568	21.875844	13.351174	7.360883	1.0
skull	10.710265	20.813958	17.428433	11.902009	6.996484	1.0
foot	9.891230	21.444472	18.065576	11.995331	7.010211	1.0
mrt angio	13.529686	22.573198	18.851311	12.658554	7.248915	1.0
stent	15.040168	27.609187	21.176151	13.213183	7.390414	1.0
warpx small Ez	21.018358	39.450457	26.067836	14.678741	7.683761	1.0
warpx small rho	22.697175	42.086873	26.738521	14.832762	7.734269	1.0
warpx small Ex	18.697022	35.747164	24.681198	14.226227	7.578187	1.0
warpx small Ey	23.662662	40.748330	26.427431	14.797027	7.680619	1.0
pancreas	17.397050	27.308133	20.846501	12.992297	7.272450	1.0
bunny	20.015273	28.688304	21.709871	13.428739	7.407273	1.0
backpack	18.331431	27.637486	20.825537	13.100715	7.279097	1.0
present	20.531511	28.164133	21.534862	13.384503	7.461730	1.0
neocortical layer 1 axons	18.794877	26.417493	20.451579	12.989378	7.305251	1.0
prone	21.107430	29.729265	22.057423	13.370788	7.418956	1.0
asteroid	26.787509	39.893873	25.998170	14.466344	7.620627	1.0
christmas tree	19.572079	26.626069	20.610166	13.086220	7.312849	1.0
vertebra	18.457093	32.871010	22.648706	13.851655	7.455070	1.0
magnetic reconnection	23.057817	29.169085	21.871576	13.239396	7.472588	1.0
marmoset neurons	24.158246	28.662523	21.651997	13.767827	7.652475	1.0
stag beetle	47.740521	50.153938	29.205911	15.414452	7.950918	1.0
pawpawsaurus	—	—	—	—	—	—
spathorhynchus	—	—	—	—	—	—
kingsnake	—	—	—	—	—	—
warpx large Ey	—	—	—	—	—	—
warpx large rho	—	—	—	—	—	—
warpx large Ez	—	—	—	—	—	—
warpx large Ex	—	—	—	—	—	—
Nyx particle mass density	—	—	—	—	—	—

Table A5: PPP speed-up compared to serial on KNL for all 3D datasets. See Appendix D for the corresponding timings.

F TTK RESULTS FOR 3D DATASETS ON HASWELL

	64	32	24	16	8	4	2	1
marschner lobb	0.049862	0.030217	0.021816	0.020424	0.021291	0.028796	0.040681	0.062984
nucleon	0.042875	0.031371	0.021521	0.019456	0.020031	0.023282	0.029632	0.044781
silicium	0.102043	0.095027	0.088266	0.086240	0.040611	0.045389	0.058960	0.097439
neghip	0.060087	0.038009	0.040597	0.039004	0.046466	0.056421	0.077084	0.118688
fuel	0.097933	0.087743	0.085085	0.082077	0.085127	0.092970	0.106507	0.144186
tooth	2.626720	1.544830	1.328990	1.190950	1.227900	1.540450	1.893480	5.271760
shockwave	1.447200	0.969234	1.046470	0.795929	0.792436	0.850987	1.411480	1.711060
hydrogen atom	0.286965	0.261202	0.260882	0.276810	0.290801	0.379583	0.489017	0.897027
lobster	3.843540	2.287700	2.035030	1.843810	1.979520	2.754030	3.591520	5.494350
mri ventricles	17.827900	11.219400	9.730420	9.069210	9.191580	11.388500	14.368400	20.242700
engine	6.107600	3.232240	2.835140	2.598120	2.701860	3.525560	4.711360	6.789390
statue leg	4.236080	2.700060	2.360520	2.062490	3.153440	3.943640	5.647150	8.613430
tacc turbulence	5.014680	3.745910	3.636350	3.657250	5.271280	8.767310	12.792500	21.701800
aneurism	2.153240	2.196500	2.160850	2.197950	2.604890	3.207560	4.602450	8.633830
bonsai	2.779400	2.057730	1.906650	1.864640	2.420320	3.772570	5.308880	8.423000
skull	19.759800	12.478700	10.645100	9.790570	10.379700	14.066000	22.974500	35.141300
foot	8.985310	5.485160	5.023670	4.620460	5.114590	7.657740	9.003580	14.319000
mrt angio	59.381800	36.001600	31.195700	29.119300	31.687900	40.628400	52.898300	71.837000
stent	45.805100	37.771200	35.688400	35.734300	38.835400	51.775200	71.398900	96.321700
warpx small Ez	23.403300	26.716600	26.585500	26.563900	33.289400	47.797600	71.774000	128.230000
warpx small rho	16.450300	18.315400	18.300300	18.181100	24.474100	39.437200	57.300200	89.415300
warpx small Ex	19.844500	22.408400	22.677000	22.319200	26.031200	42.990200	67.780100	127.652000
warpx small Ey	10.581800	12.152500	12.408900	12.826600	20.718100	40.466900	64.301700	124.791000
pancreas	85.582200	61.432200	58.113900	58.978300	69.786700	103.438000	148.792000	206.390000
bunny	159.189000	114.513000	108.009000	108.468000	113.140000	159.588000	209.711000	298.799000
backpack	84.777800	52.132100	47.268200	42.080200	46.355000	72.303500	88.007000	127.822000
present	183.657000	137.897000	130.239000	128.413000	137.205000	159.860000	229.482000	297.514000
neocortical layer 1 axons	151.719000	87.135400	80.580400	68.908500	68.542100	90.822700	115.003000	170.920000
prone	156.111000	92.651400	84.815800	80.490000	100.541000	139.140000	199.069000	300.989000
asteroid	22.478600	20.114100	19.870700	20.637600	25.492800	45.738400	59.076000	94.126300
christmas tree	231.260000	155.274000	138.467000	131.247000	145.541000	198.457000	246.902000	368.184000
vertebra	44.784300	30.756800	30.008600	29.982500	40.847800	65.843900	93.591300	157.115000
magnetic reconnection	881.570000	777.465000	756.458000	777.343000	825.301000	934.525000	940.863000	1108.890000
marmoset neurons	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
stag beetle	31.648700	36.864300	37.601900	38.680500	52.992100	76.769800	119.961000	221.407000
pawpawsaurus	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
spathorhynchus	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
kingsnake	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
warpx large Ey	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
warpx large rho	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
warpx large Ez	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
warpx large Ex	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
Nyx particle mass density	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM

Table A6: TTK runtime in seconds on Haswell for all 3D datasets. We repeated each evaluation 5 times and report here the best time. OOM indicates that TTK did not complete computation of the contour tree in any of the 5 tries due to out-of-memory error. Similar to PPP, we attempted to process the OOM files on a Cori login node, with 512GB of main memory, to accommodate the larger memory requirements, but even with *OMP_STACKSIZE* set to 1000M the files did not complete successfully.

G TTK Speed-up for 3D Datasets on Haswell

	64	32	24	16	8	4	2	1
marschner lobb	1.263164	2.084396	2.887055	3.083838	2.958245	2.187233	1.548245	1.0
nucleon	1.044460	1.427460	2.080795	2.301667	2.235585	1.923409	1.511233	1.0
silicium	0.954880	1.025378	1.103921	1.129855	2.399320	2.146740	1.652626	1.0
neghip	1.975269	3.122637	2.923566	3.042962	2.554303	2.103621	1.539721	1.0
fuel	1.472291	1.643276	1.694607	1.756721	1.693777	1.550889	1.353770	1.0
tooth	2.006974	3.412518	3.966742	4.426517	4.293314	3.422221	2.784165	1.0
shockwave	1.182324	1.765373	1.635078	2.149765	2.159241	2.010677	1.212245	1.0
hydrogen atom	3.125911	3.434227	3.438440	3.240587	3.084676	2.363191	1.834347	1.0
lobster	1.429502	2.401692	2.699886	2.979889	2.775597	1.995022	1.529812	1.0
mri ventricles	1.135451	1.804259	2.080352	2.232025	2.202309	1.777468	1.408835	1.0
engine	1.111630	2.100522	2.394728	2.613193	2.512858	1.925762	1.441068	1.0
statue leg	2.033349	3.190088	3.648954	4.176229	2.731439	2.184132	1.525270	1.0
tacc turbulence	4.327654	5.793465	5.968017	5.933912	4.116989	2.475309	1.696447	1.0
aneurism	4.009692	3.930722	3.995571	3.928128	3.314470	2.691713	1.875920	1.0
bonsai	3.030510	4.093346	4.417696	4.517226	3.480118	2.232695	1.586587	1.0
skull	1.778424	2.816103	3.301171	3.589301	3.385580	2.498315	1.529578	1.0
foot	1.593601	2.610498	2.850307	3.099042	2.799638	1.869873	1.590367	1.0
mrt angio	1.209748	1.995384	2.302785	2.466989	2.267017	1.768147	1.358021	1.0
stent	2.102860	2.550136	2.698964	2.695497	2.480255	1.860383	1.349064	1.0
warpx small Ez	5.479142	4.799638	4.823306	4.827228	3.851977	2.682771	1.786580	1.0
warpx small rho	5.435481	4.881974	4.886002	4.918036	3.653466	2.267283	1.560471	1.0
warpx small Ex	6.432614	5.696614	5.629140	5.719381	4.903808	2.969328	1.883326	1.0
warpx small Ey	11.792984	10.268751	10.056572	9.729079	6.023284	3.083780	1.940711	1.0
pancreas	2.411600	3.359639	3.551474	3.499423	2.957440	1.995302	1.387104	1.0
bunny	1.877008	2.609302	2.766427	2.754720	2.640967	1.872315	1.424813	1.0
backpack	1.507730	2.451887	2.704186	3.037581	2.757459	1.767854	1.452407	1.0
present	1.619944	2.157509	2.284370	2.316853	2.168390	1.861091	1.296459	1.0
neocortical layer 1 axons	1.126556	1.961545	2.121111	2.480391	2.493650	1.881908	1.486222	1.0
prone	1.928045	3.248618	3.548737	3.739458	2.993694	2.163210	1.511983	1.0
asteroid	4.187374	4.679618	4.736939	4.560913	3.692270	2.057927	1.593309	1.0
christmas tree	1.592078	2.371189	2.659002	2.805276	2.529761	1.855233	1.491215	1.0
vertebra	3.508261	5.108301	5.235666	5.240223	3.846352	2.386174	1.678735	1.0
magnetic reconnection	1.257858	1.426289	1.465898	1.426513	1.343619	1.186581	1.178588	1.0
marmoset neurons	—	—	—	—	—	—	—	—
stag beetle	6.995769	6.006000	5.888187	5.723995	4.178113	2.884038	1.845658	1.0
pawpawsaurus	—	—	—	—	—	—	—	—
spathorhynchus	—	—	—	—	—	—	—	—
kingsnake	—	—	—	—	—	—	—	—
warpx large Ey	—	—	—	—	—	—	—	—
warpx large rho	—	—	—	—	—	—	—	—
warpx large Ez	—	—	—	—	—	—	—	—
warpx large Ex	—	—	—	—	—	—	—	—
Nyx particle mass density	—	—	—	—	—	—	—	—

Table A7: TTK speed-up compared to serial on Haswell for all 3D datasets. See Appendix F for the corresponding timings.

H PPP SPEED-UP COMPARED TO TTK FOR 3D DATASETS ON HASWELL

	64	32	24	16	8	4	2	1
marschner lobb	1.508440	0.998638	0.701400	0.752996	0.742898	0.925438	1.112220	1.584384
nucleon	1.378810	1.166394	0.806076	0.724512	0.828357	0.838717	0.859016	1.223658
silicium	2.939492	3.225033	2.983626	2.676926	1.407607	1.333972	1.360874	1.931011
neghip	1.375398	1.008432	1.080357	0.958866	1.116539	1.078975	1.123852	1.222529
fuel	2.450791	2.389613	2.279436	2.124790	2.097915	1.973355	1.670156	1.688914
tooth	9.834735	6.350530	5.346800	4.682199	3.588043	2.968461	2.318203	3.311553
shockwave	14.876645	10.287678	11.236562	8.343631	6.068307	4.540559	4.444864	2.911526
hydrogen atom	2.621379	2.404821	2.428911	2.489187	1.952851	1.755046	1.373632	1.443214
lobster	8.260029	4.783732	4.174001	3.594515	2.731872	2.432094	1.878341	1.541201
mri ventricles	12.030677	6.912499	5.753832	5.194427	3.679281	2.856609	2.073979	1.440864
engine	9.143565	4.505680	3.829863	3.309682	2.370510	1.972739	1.517136	1.126264
statue leg	7.017618	4.130838	3.505646	2.963833	3.041952	2.362211	1.951270	1.578037
tacc turbulence	5.625623	3.853849	3.593657	3.358264	3.158819	3.163130	2.642483	2.359738
aneurism	4.329635	3.672665	3.561933	3.486906	2.825567	2.085824	1.646637	1.655525
bonsai	4.341056	2.798924	2.518020	2.335804	1.969357	1.839587	1.484105	1.222479
skull	10.198712	5.765458	4.784745	4.153457	3.019303	2.515951	2.395415	1.786334
foot	8.312035	4.643011	4.109543	3.659567	2.693889	2.500470	1.690969	1.351806
mrt angio	10.575452	5.586095	4.700774	4.295319	3.316250	2.702866	2.049369	1.374537
stent	11.618052	8.420752	7.633948	7.367882	5.397014	4.502035	3.498060	2.343702
warpx small Ez	12.559394	11.905316	11.461985	10.801451	8.599712	7.222778	6.047844	5.461965
warpx small rho	9.411519	8.629164	8.358515	7.894323	6.753376	6.429353	5.201779	4.148278
warpx small Ex	9.067170	8.618151	8.361417	7.675710	5.593645	5.571493	4.896592	4.654161
warpx small Ey	5.804990	5.403777	5.304921	5.166558	5.279021	6.055499	5.354950	5.324280
pancreas	10.166499	6.120574	5.628737	5.530443	4.650524	4.398566	3.572857	2.534308
bunny	10.407231	6.405139	5.861994	5.769329	4.204651	3.731700	2.875933	2.064170
backpack	9.339745	5.011690	4.385131	3.784735	2.919503	2.817257	1.980748	1.469612
present	10.922797	6.988992	6.432699	6.146104	4.674118	3.494964	2.911924	1.837482
neocortical layer 1 axons	10.865866	5.535464	4.934954	4.078488	2.847355	2.343189	1.709020	1.296804
prone	8.914312	4.480242	4.005677	3.670432	3.234109	2.804513	2.306402	1.778128
asteroid	5.740722	4.163376	4.022515	4.062856	3.323317	3.538973	2.565933	2.080958
christmas tree	8.249857	4.810551	4.182004	3.878400	3.119562	2.724045	1.965546	1.503620
vertebra	6.572029	3.838497	3.571687	3.360634	3.033448	3.005487	2.361896	2.036722
magnetic reconnection	20.960318	16.223449	15.320389	15.230671	11.388112	8.121079	4.610379	2.811901
marmoset neurons	—	—	—	—	—	—	—	—
stag beetle	3.258262	3.101124	3.133988	3.155505	2.837505	2.441927	2.057449	1.967083
pawpawsaurus	—	—	—	—	—	—	—	—
spathorhynchus	—	—	—	—	—	—	—	—
kingsnake	—	—	—	—	—	—	—	—
warpx large Ey	—	—	—	—	—	—	—	—
warpx large rho	—	—	—	—	—	—	—	—
warpx large Ez	—	—	—	—	—	—	—	—
warpx large Ex	—	—	—	—	—	—	—	—
Nyx particle mass density	—	—	—	—	—	—	—	—

Table A8: PPP speed-up compared to TTK using the same number of threads on on Haswell for all 3D datasets. See Appendix B and F for the corresponding timings for PPP and TTK, respectively.

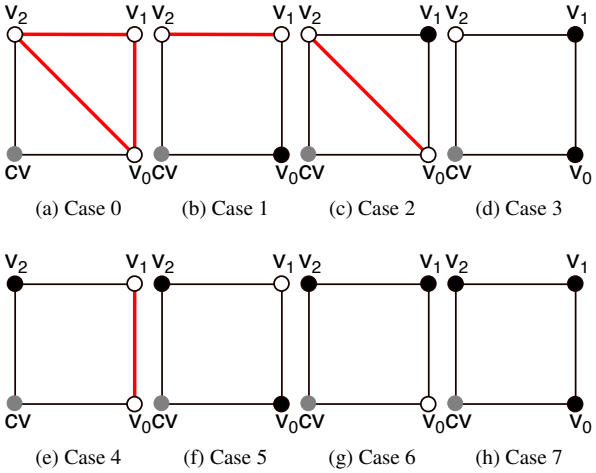


Fig. 21: Possible quadrilateral-internal connection scenarios for the upper link. v_0–v_2 are all edge- and diagonal connected neighbors of the centre vertex. Bold red lines indicate internal connections between vertices. A case table (Table A9) contains a bit for each possible vertex pair indicating whether these vertices belong to the same link component.

Case No	$v_0v_1v_2$	Connected		
		v_0v_1	v_0v_2	v_1v_2
0	—	T	T	T
1	–+	F	F	T
2	-+-	F	T	F
3	-++	F	F	F
4	+–	T	F	F
5	+-+	F	F	F
6	++-	F	F	F
7	+++	F	F	F

Table A9: Example case table for computing the connected components in the lower link by iterating over quadrilaterals adjacent to the centre vertex.

I Implementation Details for Computing Marching Cubes Connectivity

We identify individual components of the upper (or lower) link, based on vertex *polarities*, similar to finding appropriate marching cubes cases. The polarity of a vertex is determined by its value relative to the value at the centre vertex: positive if its value is larger than the centre, negative if smaller.

We compute connected components in the upper link as follows. First, we add all edge connected neighbors with positive polarity as separate sets to a disjoint set (union find) data structure. (Neighbors with negative polarity do not belong to the upper link.) To determine the number of distinct upper link components, we iterate over the four quadrilaterals adjacent to the centre vertex to determine if they connect any upper link components. For quadrilaterals in the 2D case, this is simple: If all vertices have positive polarity, the two edge-connected neighbors of the centre vertex belonging to the quadrilateral belong to the same upper-link component, and we perform a union operation on their respective sets. If at least one vertex has negative polarity, then either one of the edge-connected neighbors has negative polarity and does not belong to the upper link, or the vertex between the edge-connected neighbors has negative polarity, implying that there is no connection between the upper link components in this quadrilateral. After this iteration, the union-find structure will have one set per connected component, and we pick one representative neighbor (e.g., the neighbor with the lowest index) as the start of a new path.

The determination whether a quadrilateral connects components in the upper link is easy to make for two-dimensional quadrilaterals, but we observe that it is also possible to encode it as a case table, which is useful for implementing three-dimensional marching cubes connectivity. First, we observe that using rotation symmetry it is possible to restrict our considerations to the quadrilateral whose lower, left vertex corresponds to the centre vertex and whose upper, left and lower, right vertices correspond to its edge connected neighbors. We compute the case index as three-bit integer, where each bit corresponds to the three non-centre vertices, and each bit is set if the corresponding vertex has positive polarity. Since for the upper link each quadrilateral contains at most two neighbor of the centre vertex, it maps the case number to a single boolean stating whether these neighbors are connected.

Computing the connected components in the lower-link proceeds in the same way as the upper-link by adding all edge-connected and diagonal-connected neighbors of the centre vertex with negative polarity as separate sets to the disjoint set (union find) data structure. During the subsequent iteration over four quadrilaterals more cases occur that determine whether two neighbors belong to the same connected component. Furthermore, each quadrilateral now contains three neighbors of the centre vertex. To account for this, we store three bits per case table entry—one bit for each of the three vertex pairs that are possibly connected.

We pre-compute this case table (Figure 21 and Table A9) itself by iterating over all eight possible vertex polarity combinations, using union find find to determine which vertices are connected within the quadrilateral. To determine which sets need to be merged, it is sufficient (for the lower-link case) to merge all sets corresponding to vertices of negative polarity connected by a quadrilateral edge. We do not need to consider internal connections along diagonals, since marching cubes/marching quadrilateral always separates vertices with negative polarity. (We note that for the lower link in 2D all neighbors of the central vertex uniquely determine its connected components, unlike for the upper link or in the marching cubes case.)

These considerations carry over to constructing the connected components for marching cubes, albeit it is more complicated to construct an appropriate case table. To determine the number of connected components in the upper (or lower) link for marching cubes, we start by adding the eight edge connected (or eighteen edge and face-diagonal connected) neighbors for consideration to the disjoint set data structure. We then iterate over the eight cube cells adjacent to the centre vertex (i.e., having a vertex coinciding with the centre vertex). Each cube contains three edge connected (or six edge- and diagonal neighbors). We compute the connection configuration case for the cube as seven bit integer (i.e., an integer where each bit corresponds to a vertex that is not the centre vertex). The resulting case table contains three (or fifteen) bit entries, one for each pair of neighbors potentially connected by the cell. For each vertex pair that is marked as connected, we perform a union of the sets corresponding to the neighbors. After iterating over all cubes and vertex pair, we chose one representative vertex as path start. We note that the number of union operation is so small, that a standard disjoint set implementation without optimizations (path-compression and union-by-rank) is sufficient.

We use a single case table for all eight-cubes, using rotation symmetry to map the configuration such that the centre vertex corresponds to the "bottom, front, left" vertex and use two tables that map vertex and edge indices between the configuration of the currently considered cell and the standard configuration used for the lookup table.

We pre-compute case tables for upper- and lower-link connectivity for marching cubes. Both for upper- and lower-link, we iterate over the $2^7 = 128$ possible vertex polarity cases. The centre vertex always corresponds to the bottom-left-front vertex of the cell, and we assign polarities to the other vertices according to the case number. For the upper-link, we create a three-bit case table, one for each pair of edge-connected neighbors of the centre vertex. For the lower-link, the centre vertex has six edge- and diagonal-connected neighbors, resulting in fifteen pairs of possibly connected vertices. To determine what ver-

tices are connected inside the cube, we observe, that marching cubes cases always separate vertices of positive polarity, and never connect cell-diagonal vertices. Thus, for the upper link, two vertices of the cube are connected if and only if a path along the cube edges connects them that only connects vertices of positive polarity. For the lower link, two vertices are connected if and only if a path of cube edges and cube face diagonals connects them that contains only vertices of negative polarity. We can compute the case table by adding each cube vertex to a disjoint set data structure. We then iterate over all cube edges connecting vertices of positive polarity and merge their respective sets to compute the connectivity of the neighbors in the upper link. For each pair, the bit in the case table is true if the two vertices belong to the same set. For the case table for the lower link, we add all cube edges and cube face diagonals connecting vertices of negative polarity. Again, the case table for each pair of potentially connected neighbors is set to true if the vertices belong to the same set.

Finally, we note that for the neighborhood configurations for triangulations of a rectilinear mesh, we could construct a case table that directly mapped from neighbor polarity to starts of monotone paths. However, for marching cubes, the number of components in upper (or lower) link depends on 26 surrounding vertices, making a case table too large to be efficient. Thus, we use the approach of iterating over adjacent cells and using a case table only for connections within those cells, making storage costs more manageable.