Eurographics Conference on Visualization (EuroVis) 2017
J. Heer, T. Ropinski and J. van Wijk
(Guest Editors)

Volume 36 (2017), Number 3

Nested Tracking Graphs

Jonas Lukasczyk[1], Gunther Weber[2,3], Ross Maciejewski[4], Christoph Garth[1], and Heike Leitte[1]

[1]TU Kaiserslautern, Germany
[2]Lawrence Berkeley National Laboratory, USA
[3]University of California, USA
[4]Arizona State University, USA

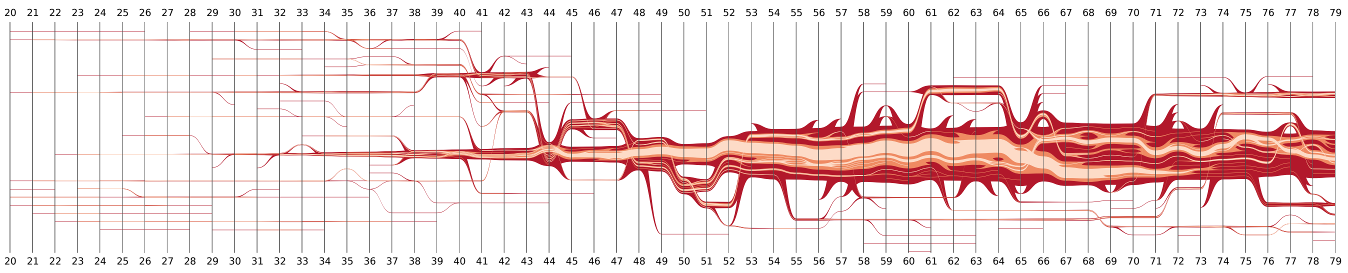


Figure 1: *Nested tracking graph for one ensemble member of the viscous finger dataset with highest particle resolution. In contrast to common tracking graphs—where edges only encode the evolution of components for a single level—nested tracking graphs additionally illustrate the nesting hierarchy of components across levels. The x-axis represents time and the y-axis is used to minimize edge crossings. The density levels* 25, 30, *and* 35 *are shown from dark to light red, and the width of edges encodes the size of their associated components.*

Abstract

Tracking graphs are a well established tool in topological analysis to visualize the evolution of components and their properties over time, i.e., when components appear, disappear, merge, and split. However, tracking graphs are limited to a single level threshold and the graphs may vary substantially even under small changes to the threshold. To examine the evolution of features for varying levels, users have to compare multiple tracking graphs without a direct visual link between them. We propose a novel, interactive, nested graph visualization based on the fact that the tracked superlevel set components for different levels are related to each other through their nesting hierarchy. This approach allows us to set multiple tracking graphs in context with each other and enables users to effectively follow the evolution of components for different levels simultaneously. We demonstrate the effectiveness of our approach on datasets from finite pointset methods, computational fluid dynamics, and cosmological simulations.

Categories and Subject Descriptors (according to ACM CCS): Data [Computer Graphics]: Data Structures—Graphs and Networks

1. Introduction

Detecting and tracking components of time varying scalar fields is a common approach in data analysis. Multiple techniques exist that identify components and visualize their evolution over time through so-called tracking graphs [WCPB12, ALS*16]. These tracking graphs are presented using layered graph drawing techniques where one axis represents time and the other axis serves to minimize edge crossings. Each edge represents a component, and nodes represent critical events such as when components appear, disappear, merge, or split. Thus, tracking graphs provide an overview and summarization of the evolution of components over time. Here, we focus on superlevel set components—connected areas in the scalar field that exceed a certain threshold—such as regions of a water-filled can that exceed a specified salt concentration threshold, or parts of the universe where dark matter exceeds a certain density. To track components over time one can apply methods of topological feature tracking [ALS*16, BBD*07, BKL*11, EHMP04, MS09] or simply test for spatial overlaps [SB06, LMGH15, SW98, LBM*06]. One can also augment the resulting graphs with additional information about the components or use the y-axis to visualize a metric—such as the position, size, or integral of components—to provide additional insights [ALS*16]. However, in general tracking graphs have the following limitations:

- a tracking graph can only represent one threshold and the choice of this threshold is not always known a priori;
- to examine multiple levels one has to compare multiple tracking graphs, and;
- tracking graphs may vary substantially even under small changes to the levels.

We propose a visualization that addresses these limitations. Specifically, since superlevel sets for different levels (thresholds) are nested inside each other by the definition of level sets, we can compute several tracking graphs for a set of levels and subsequently relate parts of them to each other through the nesting hierarchy of their associated components. This nesting relationship enables us to draw a nested representation of the tracking graphs that provides context over multiple levels as it additionally illustrates if components of one level contain higher level components, or are contained in lower level components. This visualization enables users to effectively follow the evolution of components for different levels at once. For large datasets, nested graphs might become extremely huge and detailed. To counteract this problem, we demonstrate in three case studies how to integrate nested tracking graphs as dynamic and interactive control devices in visual analytic frameworks. Linked to a 3D rendering of the original data, the graphs can be used to navigate through time and toggle the visibility of components, enabling users to perform temporal and spatial data peeling.

The main contribution of this paper is a novel drawing algorithm for nested tracking graphs that can be used to interactively examine the evolution and nesting hierarchy of components across levels in one compact visualization.

2. Related Work

Previous work has demonstrated that tracking graphs are ideally suited to visualize the evolution and relation of components over time [HLH*16]. Several methods have been proposed to identify and track components of time-varying scalar fields. Components can be superlevel sets [LMGH15, SB06, BWT*11, SW98, LBM*06], or subdomains that fulfill geometric and topological constraints [ALS*16, BKL*11, BBD*07]. Subsequently, the detected components can be tracked by testing for spatial overlaps in time [ALS*16, BWT*11, LBM*06, SB06], or applying methods of topological persistence [BKL*11, BBD*07, WCPB12]. For instance, Bremer et al. [BWT*11] visualize and track burning cells in large-scale combustion simulations where cells are defined as areas exceeding a fuel consumption rate threshold and are tracked by spatial overlaps. To effectively update the tracking graph when users change the level of interest, Widanagamaachchi et al. [WCPB12] propose a method that computes an intermediate graph structure called the meta-graph. Although the introduced technique enables users to navigate through different levels, it lacks a representation that visualizes several levels simultaneously.

Building on these works, our algorithm computes a different intermediate graph structure that represents the nesting relationship between components and enables us to draw a nested graph representation of the tracking graphs. Similar to Aldrich et al. [ALS*16], Bremer et al. [BWT*11], and Sohn et al. [SB06], we identify superlevel set components and track them by testing their corresponding volumes for overlaps. To determine the nesting hierarchy of components, we also compute the spatial overlap of components for varying levels. We represent this hierarchy with simplified split trees that we refer to as nesting trees. In contrast to split and merge trees—as defined by Carr et al. [CSA03] to compute contour trees—nesting trees only represent the nesting hierarchy of superlevel sets for a predefined set of levels. This approach is similar to that of Hilaga et al. [HSKK01] who also discretize the function range to efficiently compute multiresolutional Reeb graphs. For a small number of levels nesting trees are faster to compute than split or merge trees, and store only necessary information. Nevertheless, split and merge trees, especially their temporal version, can be used to select levels for the nesting tree. Oesterling et al. [OHW*15] proposed an algorithm to compute the time-varying merge tree for piecewise linear functions in arbitrary dimensions. They use the fact that the merge tree's structure only changes when the sorting order of adjacent tree nodes changes, and determine a sequence of local updates of the merge tree. They visualize the tree by plotting a 1D landscape profile for each timestep and connecting its peaks with lines to indicate critical events. This enables them to illustrate the evolution of all critical values across time at the cost of cluttering and occlusion. Similarly, Sohn and Bajaj [SB06] track components via spatial overlaps and compute a correspondence between contour trees for each timestep pair. Their approach is capable of segmenting, tracking, quantifying, and visualizing the evolution of user defined levels but lacks a comprehensible visualization that displays the relationship between the different thresholds. Widanagamaachchi et al. [WBS*14] proposed a friends-of-friends halo detection algorithm that is based on a variable linking distance between particles. By increasing the linking distance more particles are grouped together which yields a nesting hierarchy. They also represent this hierarchy with a tree for each single timestep where nodes indicate linking lengths for which groups join. However, to examine the temporal evolution of halos one has to compare multiple trees. By contrast, our work focuses on examining the temporal evolution of multiple levels through a single, compact graph representation.

3. Nested Tracking Graphs

In this section we introduce a mathematical definition of nested tracking graphs, describe their construction and visualization, and provide details of how users can interact with these graphs in a visual analytics framework.

3.1. Definition

A nested tracking graph consists of nodes, multiple tracking graphs, and a nesting hierarchy between the nodes of the tracking graphs. Each node of the nested tracking graph has an associated time and level value. For instance, consider the bottom row of Figure 2 that illustrates a time-varying, 2D scalar-field via three contours. We can derive the superlevel set components for the levels of these contours and represent each of them as a node with a time and level value. Since the components for subsequent levels are nested inside each other, we can represent this nesting hierarchy by edges between nodes within the same timestep. In this set, an edge between two nodes indicates that the component of the second node is contained in the component of the first node. This yields one set of edges for each timestep, and since they represent the nesting hierarchy of components we refer to them as nesting trees. Similarly, for each individual level, we track the components for subsequent timesteps and connect their respective nodes via edges if they descend from one another. This yields a common tracking graph for each level. An advantage of our approach is that it does not specify which algorithm has to be used to track the components over time as long as their nodes have a nesting hierarchy that can be expressed by a nesting tree. The top of Figure 2 illustrates the nesting trees (red lines), tracking graphs (blue lines), and how their nodes are interconnected. Note that edges of the tracking graphs and nesting trees only connect nodes of the same level and timestep, respectively. Using the hierarchy of nodes described by the nesting trees we can draw edges of the tracking graphs inside each other (Figure 2 middle/center row).

Formally, a nested tracking graph $G = (N, E_T, E_N)$ consists of a node set N, a list of edge-sets E_T representing the tracking graphs, and a list of edge-sets E_N representing the nesting trees. Each node $n \in N$ has a time value t and a level l, which we denote in the following as n_t^l. Edges in E_T only connect nodes of the same level, while edges in E_N only connect nodes of the same timestep. Additionally, we constrain E_N such that for every node n_t^l with $l > 0$ there exists exactly one edge in E_N connecting it to a node n_t^{l-1} of one level lower. This constraint ensures that there exists a unique mapping between nodes of level l and $l-1$, i.e., each edge-set in the list E_N is a tree. E_N and E_T share the same nodes and they contain all edges of the nesting trees and tracking graphs, respectively. Finally, each node n_t^l has a width $w(n) \in R^+$ where $w(n)$ is greater or equal to the sum of the width of all nodes with level $l+1$ that are connected to n_t^l through E_N. This ensures that each node has enough space to host all its children. This property is also naturally satisfied by superlevel sets for subsequent levels.

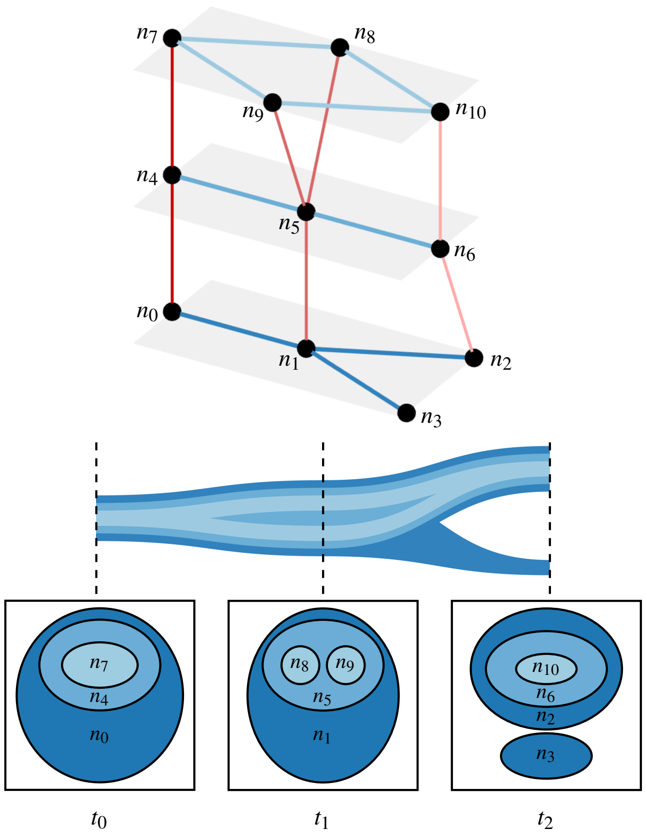


Figure 2: *(Top) 3D illustration of a nested tracking graph where tracking graphs for one level are shown in shades of blue, and nesting trees for each timestep in shades of red. Each level is also highlighted via gray planes. (Middle) Nested graph representation where edge colors encode levels, and edges are nested inside each other according to the nesting tree. (Bottom) Three timesteps of a time-varying scalar-field that was used to derive the nested tracking graph where each timestep is shown via three contours.*

3.2. Computation

This section provides details of how to derive the nested tracking graph. Specifically, we show how to compute nesting trees and tracking graphs for superlevel set components of time varying scalar fields given on a voxel grid. The key idea of this algorithm is to derive superlevel sets for a list of levels at each timestep and then test for spatial overlaps of subsequent levels and timesteps. We start by defining components for voxel volumes, then we introduce nesting trees, and finally we show how they are connected through time by tracking graphs.

A superlevel set component is a set of voxels that exceed a certain level and are all connected via the 27-neighborhood. Moreover, there exists a nesting hierarchy of the components for varying levels. By the definition of superlevel sets, a component for level a is a subset of exactly one component of level b with $b < a$. Hence, this hierarchy can be represented with a structure we refer to as a nesting tree (Figure 3 (right)). Nodes of this tree and their vertical position represent an individual superlevel set and corresponding level, respectively. An edge between nodes represents the fact that the component of higher level is a subset of the lower level component. This is similar to a split tree [CSA03] (Figure 3 (left)), except that the leafs and inner nodes of the split tree indicate critical values of the underlying scalar field for which components cease to exist and split, respectively. In fact, as illustrated in Figure 3, the nesting tree is a subset of the split tree for a set of levels. It is important to note that we do not need to compute the entire split tree. Instead, we derive the nesting tree by choosing a sorted list of levels, compute the connected components for these values, represent each component by a node, and then test components of subsequent levels for overlaps to create edges between the nodes. If there exists an overlap, we add an edge between their respective nodes in the nesting tree. Since we only sample the nesting hierarchy with a discretized list of levels, it is possible that we miss critical values and thus not derive the same structure as the correct split tree.

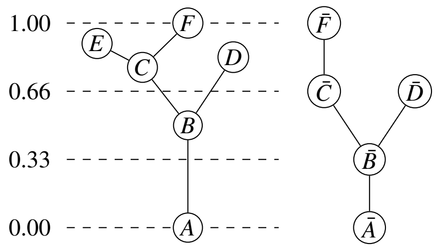


Figure 3: *A split tree (left) and a nesting tree (right).*

Figure 3 shows that with the given sampling we miss that component C splits into E and F between the samples. Although nodes in the nesting tree generally do not represent critical values of the data, their connectivity still represents the correct nesting relation of the respective superlevel set components. In other words, the nesting tree correctly records the hierarchy of the split tree restricted to the sample values. However, the critical values of split trees might indicate relevant levels for the nesting trees.

A tracking graph represents the temporal evolution of components where nodes indicate when components appear, disappear, merge, and split. Several methods exist to derive a tracking graph by computing superlevel set components for a fixed level for each timestep and then test the components of subsequent timesteps for spatial overlaps [ALS*16, BWT*11, LMGH15, SB06]. Each component is again represented by a node and there exists an edge between nodes if their respective volumes overlap. In our approach, we compute the tracking graphs for the same set of levels we chose for the nesting trees. Since the nodes of the nesting tree already represent the components of the tracking graphs, each tracking graph for a given level consists of a set of edges that connect the nodes of the nesting trees. Hence, the nesting trees and the tracking graphs share the same set of nodes N, and we only have to store their edges in E_N and E_T, respectively. In particular, E_N is a list of edge-sets where each single set represents the entire nesting tree of one timestep, and E_T is a list of edge-sets where each individual set represents an entire tracking graph for one level. Together they yield the nested tracking graph $G = (N, E_T, E_N)$.

3.3. Implementation

In order to implement this approach efficiently, we have to compute, label, and compare superlevel set components in the computational domain. The scalar values of a timestep t are stored as a 1D floating-point array D_t, and the list containing all these arrays is denoted as D. To store the ID of a component that is present at a voxel we use additional integer arrays of the same size as D_t called the component matrices. We denote with C_t^l the component matrix that stores at position i the ID of the component for level l and time t at voxel i. We can compute a component matrix C as described in Algorithm 1 which requires a data array D_t, a level l, a component counter n, and a set of nodes N as input. First, we mark in C the cells that are below the level l in D with -1, and the cells greater or equal to l with -2. In a subsequent iteration over the cells of C, a value of -2 indicates a component we have not identified yet. In these cells, we start a standard 3D flood fill algorithm that labels the new component by writing the current component counter value n into all voxels connected to the cell via the 27-neighborhood. The flood fill procedure returns a node that also stores additional information about the component such as its size, bounding box, integral, and ID. This node is added to the set of nodes N where n is used as a unique ID for that component. Finally, we increase the component counter by one.

Algorithm 1 Compute Component Matrix (D_t, l, n, N)

```
 1: // Initialize Component Matrix
 2: $C = [\,]$
 3: for $i = 0 \,..\, size(D_t)$ do
 4:     $C[i] = \begin{cases} -1 & \text{if } D_t[i] < l \\ -2 & otherwise \end{cases}$
 5: end for

 6: // Detect Components
 7: for $i = 0 \,..\, size(D_t)$ do
 8:     if $C[i] = -2$ then
 9:         $node = floodFill3D(C, i, n)$
10:         $N = N \cup \{node\}$
11:         $n{+}{+}$
12:     end if
13: end for

14: return $C$
```

To test for spatial overlaps of component matrices, Algorithm 2 iterates over two matrices C_1 and C_2 and adds edges to the edge-set E if components overlap. Specifically, if both arrays at index i store a value greater or equal to zero then the respective components overlap and we add an edge to E containing both IDs. Note that this algorithm can be used to test for component overlaps of different levels and timesteps.

Algorithm 2 Connect Nodes (C_1, C_2, E)

```
1: for $i = 0 \,..\, size(C_1)$ do
2:     if $C_1[i] \geq 0 \wedge C_2[i] \geq 0$ then
3:         $E = E \cup \{ (C_1[i], C_2[i]) \}$
4:     end if
5: end for
```

We can compute the nesting tree with Algorithm 3 that uses the previously described procedures. It requires the list of all voxel values D, the sorted list of levels L, and a sorted list of all timesteps T. First, the algorithm computes the component matrices for all levels and timesteps. Then, to generate the tracking graphs we test for each level all neighboring pairs of component matrices for subsequent timesteps for overlaps. Similarly, to generate the nesting trees, we test for each timestep all neighboring pairs of component matrices for different levels for overlaps. Each level yields a tracking graph and each timestep yields a nesting tree. This algorithm is embarrassingly parallel and can be easily rewritten to be more memory efficient since to fully connect two timesteps it is only necessary to keep their respective component matrices in memory. The most time critical part that impacts the total runtime is the 3D flood fill procedure as it depends on the size of the components. Table 1 provides performance results of our JavaScript implementation for single threaded computations of entire nesting graphs and associated component properties—such as size, bounding box, and integral—for three example datasets.

Algorithm 3 Compute Nested Tracking Graph (D, L, T)

```
 1: // Initialize Graph and Component Counter
 2: $N, E_T, E_N = \{\}$
 3: $n = 0$

 4: // Compute Component Matrices
 5: for $i = 0 .. size(T)$ do
 6:     $t = T[i]$
 7:     for $j = 0 .. size(L)$ do
 8:         $l = L[j]$
 9:         $C^l_t = computeComponentMatrix(D_t, l, n, N)$
10:     end for
11: end for

12: // Compute Tracking Graphs
13: for $i = 0 .. size(L)$ do
14:     $l = L[i]$
15:     $E_T[v] = \{\}$
16:     for $j = 0 .. size(T) - 1$ do
17:         $t_0 = T[j]$
18:         $t_1 = T[j+1]$
19:         $connectNodes(C^l_{t_0}, C^l_{t_1}, E_T[v])$
20:     end for
21: end for

22: // Compute Nesting Trees
23: for $i = 0 .. size(T)$ do
24:     $t = T[i]$
25:     $E_N[t] = \{\}$
26:     for $j = 0 .. size(L) - 1$ do
27:         $l_0 = L[j]$
28:         $l_1 = L[j+1]$
29:         $connectNodes(C^{l_0}_t, C^{l_1}_t, E_N[t])$
30:     end for
31: end for

32: return $(N, E_T, E_N)$
```

	Viscous Fingers	**Jet**	**Halo Simulation**
Resolution	64^3	$128^2 \times 256$	256^3
Timesteps	100	600	854
Components	1 k	139 k	4,559 k
Time	8 sec	3 min	22 min

Table 1: *Performance of our JavaScript implementation for three example datasets on a machine with 6 GB RAM and an Intel i7. The time stated in the last row excludes the computation of the layout which can be done at interactive framerates.*

3.4. Visualization

In order to visualize the nested tracking graphs, we first calculate an optimized layout for each individual tracking graph and subsequently draw them from lowest to highest level by determining the position of new nodes according to already drawn levels. Specifically, to draw the nested tracking graph $G = (N, E_T, E_N)$ we first represent each individual tracking graph in the edge-set E_T in the DOT language and compute an optimized layout using the graph library Graphviz [Gra13]. These layouts assign to each node $n \in N$ a position $p(n) = (x, y)$ on a x- and y-axes where the x-axis is used to represent time and the other to minimize edge crossings. We start by drawing the lowest level according to its optimized layout. To draw an edge (n, m) we render a Bézier curve from $p(n)$ to $p(m)$ where the width is linearly interpolated between $w(n)$ and $w(m)$. Alternatively, we can choose the smaller of both widths. The remaining vertical levels are drawn iteratively in a bottom-up approach: after drawing level l for $l \geq 0$ we render the next level $l+1$ by mapping the position of nodes of the current level (children) to the locations of the already rendered nodes of one level lower (parents). A child node n^{l+1}_t is connected via exactly one edge in E_N to its parent m^l_t since this was a constraint on E_N. The render position $p(n)$ of a child n depends on the number and width of all other children of its parent node. Since we require that the total width of all children does not exceed the width of the parent we can draw them below each other inside the available space of the parent where the order depends on the optimized layout calculated for the tracking graph of level $l+1$. The remaining space of the parent can be used to create a gap between its children. After determining the positions of the children according to their parents, we can draw the edges of the current level as described before. Although the color scheme used to encode the different levels can be domain specific, in general it appears sensible to use a sequential or diverging color map.

The width used for each node is either fixed for a level or encodes some metric of its associated component—such as its size or integral. In the former case, the resulting graphs clearly show the topological relation between components where in the latter case the graphs also show the evolution of some component property. The only requirement is that the width of a parent is equal or larger than the sum of the widths of its children. For instance, we can visualize the size of components since it is not possible that the nested components are larger than the parent containing them.

Nested tracking graphs for many components might become extremely large and cluttered. To counteract this problem, we integrate nested tracking graphs in a visual analytics framework that enables users to interactively explore the graphs and their correspondence to the original dataset. Figure 4 shows our web-based tool that consists of a direct volume rendering (DVR) window (left), and the nested tracking graph (right). Per default, each level of the nested graph is shown in a different color to provide an overview across the different levels. Selecting a level of interest by clicking on one of its edges grays out all other levels and uses edge colors to encode the individual components of the selected level. The resulting highlighted graph is a common tracking graph where color is used to illustrate the history of the single components as described in Aldrich et al. [ALS*16] and Lukasczyk et

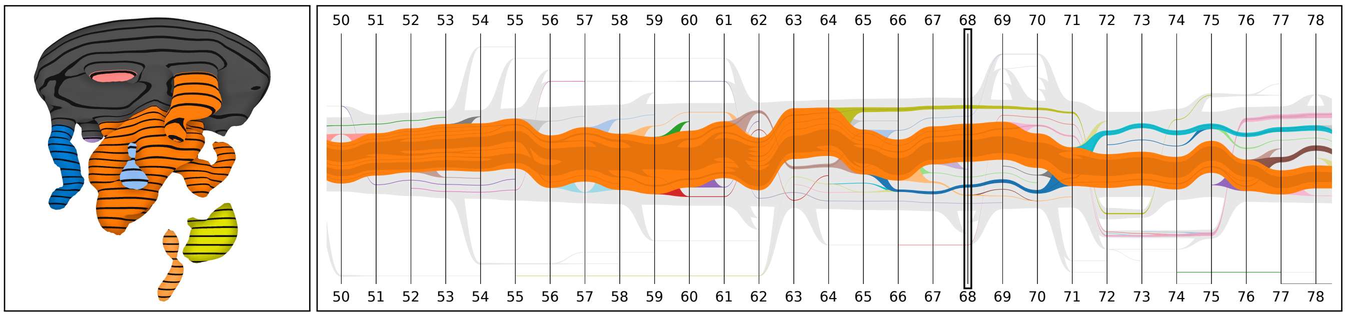


Figure 4: *Interface of our visual analytics framework consisting of a DVR window (left) and the interactive nested tracking graph (right).*

al. [LMGH15]. Selecting a layer also updates the extracted superlevel sets in the DVR window. Although the other levels are grayed out, they still provide context as they indicate the nesting hierarchy with respect to the current level. For instance, the nested graph in Figure 4 shows 1) that the huge orange component contains multiple components of higher value, 2) that all components of the selected level are contained in one single component of lower level, and 3) that sometimes small components split from this low level component. Components in the DVR and nested graph window are shown in the same color to link both views. One can use the nested tracking graph to interact with the components visualized in the DVR window and vice versa by selecting either components in the 3D rendering or nodes, edges, and entire paths of the nested graph. This selection mechanism is used to manage white- and blacklists that control which components are visible and which ones are grayed out.

In the following we provide some implementation details of the tool. On initialization, we load the precomputed nested tracking graph—represented as a JSON-Object—from the server and generate in real-time on the client via D3.js [BOH11] an interactive SVG representation using the previously described layout algorithm. To visualize a timestep with direct volume rendering (DVR), the tool first loads the respective voxel volume and subsequently uses Algorithm 1 to compute at interactive framerates the superlevel sets and component IDs. However, unless the correct component counter n is chosen the computed IDs will not match the IDs stored in the graph. To ensure a consistent labeling, in the preprocessing when we compute the nested tracking graph we also store each time we execute Algorithm 1 its respective input values and attach them to the JSON-Object. As the user selects a timestep we can simply lookup the input values and the resulting IDs will match due to the deterministic nature of the algorithm. The computed component matrix and the scalar field are both passed as data textures to a WebGL shader that performs direct volume rendering with screen space ambient occlusion to enhance spatial perception. The shader renders for a given level the components and colors them according to the component matrix. In addition to the resulting DVR image, we also generate a selection buffer where each pixel stores the ID of the component shown at that location. This way, by clicking on the DVR window it is possible to identify the selected component and highlight relevant parts of the nested graph.

3.5. Limitations

Although the proposed layout algorithm produces smooth and streamlined layouts, it still contains edge crossings. Sometimes, these are unavoidable or occur due to limitations of the used layout algorithm. For example, the red lines of Figure 8 between timestep 835 and 840 cross even though their corresponding components do not merge. These "false" crossings, which could be misinterpreted, can only occur between timesteps and never at an exact timestep. Hence, edge crossings between timesteps are only layout based and do not have any semantic interpretation.

Another limitation is the maximum number of visible levels, which depends on the dataset, the amount of cluttering of the resulting graphs, and on whether the graphs are shown statically or in an interactive interface. In our experience, static drawings of nested graphs—such as those presented in this work—should not show more than three levels at once. An interactive interface can use zooming and focus-and-context techniques to compensate for cluttering, making it practical to show up to 8 levels in a nested graph. However, using only two levels is already a significant improvement over previous visualization techniques as the nested tracking graph shows the tracking graphs for both levels simultaneously and sets them in context with each other.

4. Results

In the following, we demonstrate how nested graphs can be used for ensemble comparison, semantic decomposition, and interactive exploration.

4.1. Viscous Fingers

In this case study we examine an ensemble of finite pointset method (FPM) simulations that was provided for the 2016 scientific visualization contest [IEE16]. FPMs are a common tool to solve numerical problems in fluid dynamics and continuum mechanics where the medium is represented as a numerical point cloud with each particle storing properties of the medium at its location. The ensemble consists of simulations that model the viscous fingering process of salt solutions inside water. Specifically, a cylinder is filled with pure water and contains an infinite salt supply at its top. As soon as the salt mixes with the water the resulting solutions sink down to the bottom since they have a higher density than the surrounding water. In course of the simulation, the solutions form structures with

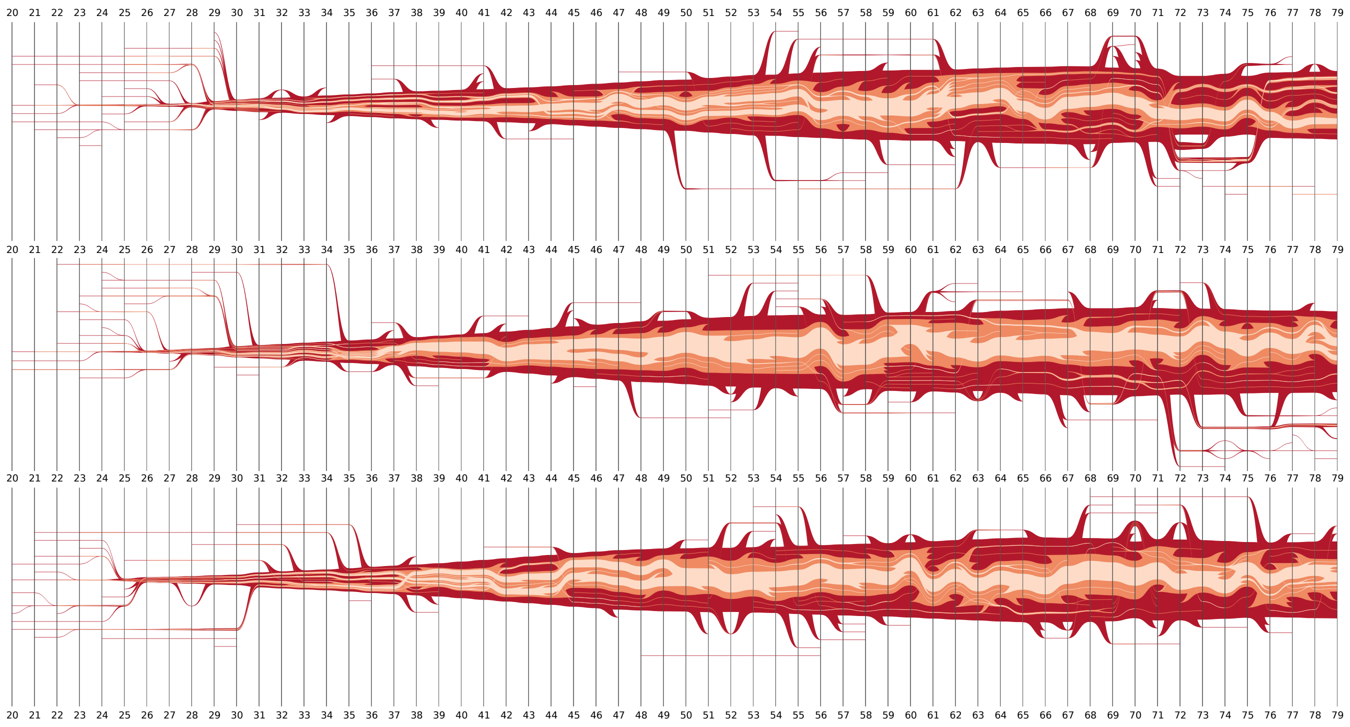


Figure 5: *Nested graphs for three simulation runs with the lowest particle resolution of the viscous finger dataset. Edges encode the evolution of components, and their width the size of their associated components. Different levels are shown in different colors, i.e., the levels* 25*,* 30*, and* 35 *are shown from dark to light red, respectively. The x-axis represents time and the y-axis is used to minimize edge crossings. Although stochastic effects alter simulation results, the graphs show similar trends such as the initial phase where small fingers originate from the salt supply and then merge into larger finger structures.*

increased salt concentration value, called viscous fingers. However, it is not deterministic when and where viscous fingers appear and how they evolve. To examine the aleatoric uncertainty of the mixing process, the simulations incorporate stochastic effects that alter simulation results. Furthermore, the accuracy of the mean solution rate and properties of the fingers also depend on the used point cloud resolution. The ensemble consists of 50 simulations at three resolution levels, i.e., 250*k*, 650*k*, and 1900*k* particles. Each simulation run provides around 100 timesteps.

In the following we utilize the approach of Aldrich et al. [ALS*16] to identify and track viscous fingers within the particle clouds. We represent the entire computational domain as a voxel grid with 64^3 cells and estimate the density distribution of the salt concentration by averaging the salt density values of all particles in the cells. To handle gridding artifacts and reduce high-frequency noise in the density estimate, we apply low-pass filtering using a Gaussian kernel with a bandwidth of two voxels. Viscous fingers are identified as areas within the domain with increased salt concentration density, i.e., superlevel set components in the voxel grid that exceed a certain concentration level (Figure 4 left). However, it is not sufficient to simply partition the domain according to a threshold since most fingers are connected to the salt supply at the top of the domain. All fingers emerging from the salt supply together with the supply itself are identified as one single superlevel set component. To separate fingers and to filter voxels that are considered noise or belong to the salt supply we also derive the spatial Reeb graph. After we use this graph to filter out the salt supply and noise, we determine the components within the remaining voxels and proceed to track them according to Section 3.3. In the approach of Aldrich et al. [ALS*16] the authors compute tracking graphs for multiple density levels. Although each individual tracking graph effectively summarizes the evolution of fingers, they are limited to said level and do not provide context. It is not apparent if two fingers of the same concentration level are contained in a finger of lower concentration level. To tackle this problem, we can derive the nested tracking graph for a set of salt concentration levels. We obtain this set of levels by uniform sampling values between the minimum and maximum concentration or use a set of values manually chosen by domain scientists who are interested in certain thresholds. Considering that we do not only analyze a single run, but rather summarize runs and compare them to each other it makes sense to determine a set of levels that can be used across all simulations. Therefore, we use a uniform sampling and later enable the user to toggle the visibility of levels within the visual analytics framework.

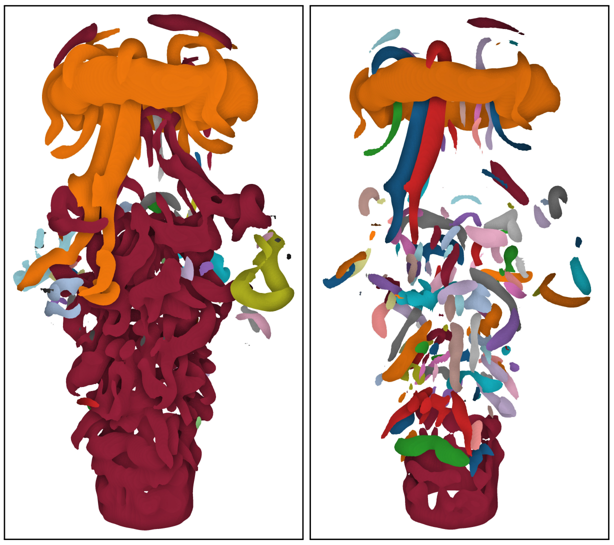
Figure 5 shows the nested tracking graphs for three ensemble members with the lowest particle resolution and the same smoothing bandwidth for three levels. The graphs show the density levels 25, 30, and 35 where layers are encoded with colors ranging from dark to light red. In previous approaches, users had to compare three separate tracking graphs per ensemble member. In contrast, nested tracking graphs enable users to compare ensemble members directly by illustrating the evolution and nesting hierarchy of the fingers across levels in one compact visualization. The width of edges encodes the size of their associated components. Obviously, the stochastic effects of the simulation have an impact on the resulting finger structures. Although the graphs differ in various aspects, some trends become apparent. For instance, the fingers seem to evolve in two phases. From timestep 0 to around 37 small fingers emerge from the salt supply that subsequently merge into one huge component shown in dark red. In all runs there exists only one of these huge components that sometimes splits into—or merges with—small components. Furthermore, the number of fingers and their nesting hierarchy are in general highly similar for these runs. We also explored the impact of the particle resolution on the simulation to see whether those effects are visible in the nested graphs. We use the same smoothing bandwidth to process ensemble members with the highest particle resolution. Figure 1 shows a nested graph for one of these members. As expected, the graph becomes more complex since the finger structures are now very detailed. Nonetheless, simulations with higher particle resolution still show similar trends as they also evolve in two phases and on average have the same number and nesting hierarchy of fingers.

With an increasing number of levels and timesteps the static graphs become more cluttered and suffer from information overload. To solve this problem we created the visual analytic framework described in Section 3.4 to interact with the graphs in a level-of-detail approach. Specifically, one can toggle the visibility of levels and components, encode different metrics of the fingers, and directly validate the encoded information of the nested graphs by linking them to a 3D rendering of the fingers. Through comparison of the graphs and the 3D rendering, we observed that the graphs correctly represent the evolution of components, their properties, and their nesting hierarchy. Furthermore, the tool provides a link between the graph and the DVR by enabling users to select components in either view and directly see highlighted parts in the other.

4.2. Jet

The *Jet* dataset results from a direct, numerical, computational fluid dynamics (CFD) simulation capturing the injection of a jet into a medium at rest and the formation of vortical structures due to friction. Initially, a large vortex is formed at the tip of the jet. As the simulation progresses, this vortex decays into progressive smaller vortical structures as the system moves towards turbulence. Velocity data is given on a regular grid of resolution $128 \times 256 \times 128$ for a total of 600 timesteps. From this, we compute the vorticity magnitude as a criterion for the local strength of rotation. Superlevel sets of high vorticity magnitude are used to identify vortices.

With this case study we demonstrate how the nested tracking graphs can be used to create semantic partitions of a dataset, helping a user to effectively peal through the data. For example, consider the components for the two vorticity magnitude levels 85 and 117 shown at the top of Figure 6. A standard tracking graph that illustrates the evolution of the numerous components at level 117 is heavily cluttered and does not provide context. However, the components of level 117 are contained in components of lower levels,



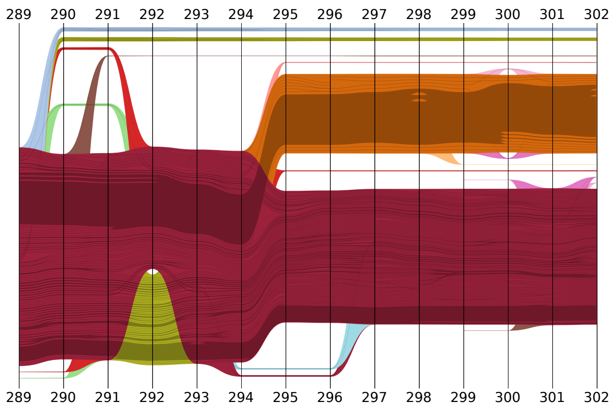


Figure 6: *(Top) Individual components of the jet dataset at timestep* 302 *for vorticity magnitude level* 85 *(left) and* 117 *(right). (Bottom) Nested tracking graph with focus on layer* 85*, i.e., layer* 117 *is grayed out and the edge colors of layer* 85 *match the components of Figure* 6 *top left. The graph indicates that the top component (orange) split from the main component (red) at timestep* 295*.*

which yields a group hierarchy that can be illustrated via a nested graph. Figure 6 (bottom) shows the nested tracking graph for these two values, where the layer for level 85 is highlighted and the layer for level 117 is grayed out. The colors of the graph match the ones used to show the individual components for level 85 of Figure 6 (top left). At timestep 302 there exist two major components, i.e., the main jet (red) and the top ring (orange). These components contain the smaller components with higher vorticity magnitude and thus provide context by partitioning them into groups. The graph shows that the ring—and thus its subcomponents—split from the main jet at timestep 295; an information that is not conveyed by conventional tracking graphs. Furthermore, if users want to examine the components within the ring, they can click on an edge that represents the ring to highlight the history of its subcomponents and filter out others. The nested tracking graph can be used to organize multiple tracking graphs and their respective components.

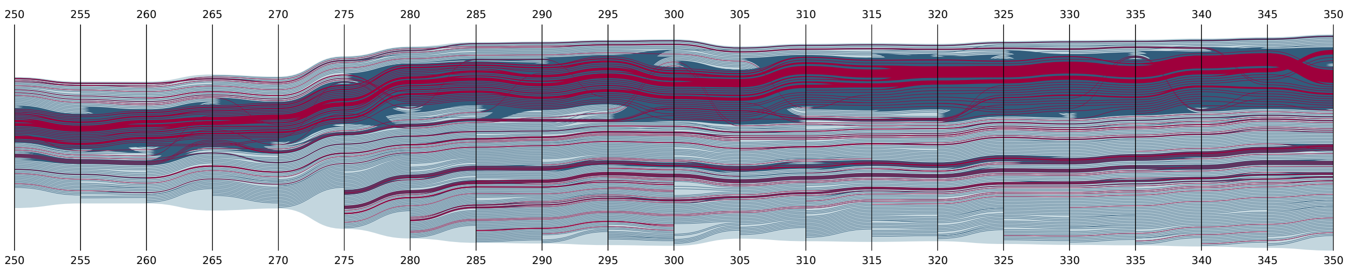


Figure 7: *Nested tracking graph for one filament of the halo dataset where the density levels* 2.632497×10^{12}, 1×10^{12}, *and* 5×10^{11} *are shown in red, dark blue, and light blue, respectively.*

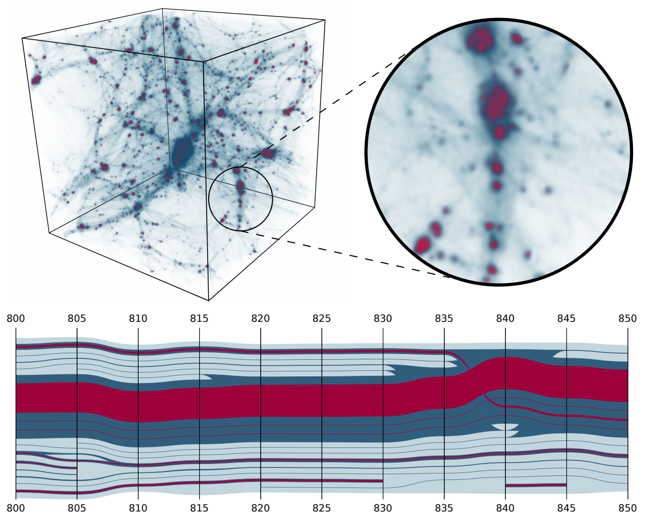


Figure 8: *(Top) DVR image of the Halo dataset at timestep* 850 *that highlights halos with density* 2.632497×10^{12} *(red),* 1×10^{12} *(dark blue), and* 5×10^{11} *(light blue). (Bottom) Nested tracking graph for one galaxy filament (light blue) that illustrates the evolution of halos (dark blue) and sub-halos (red).*

4.3. Halos

In this case study we use nested tracking graphs to visualize cosmology simulation data in collaboration with a domain expert. The data was generated using the Nyx [ABL*13] code co-developed by the Center for Computational Sciences and Engineering (CCSE) and the Center for Computational Cosmology (C^3) at the Lawrence Berkeley National Laboratory to explore structure formation in the universe. We consider a data set with 256^3 voxels from a scaling study of a simulation of the Lyman α forest [LSN*15]. The simulation covers a cubic domain with an edge length of approximately 93 million light years and contains 850 timesteps spanning the interval from redshift $z = 159$ (approximately 10 million years after the Big Bang) to redshift $z = 0$ (today, approximately 13.5 billion years later). It uses hydrodynamics to evolve Baryon density and treats dark matter as collisionless particles evolved via a particle-mesh method. We consider the total mass density, which is the sum of Baryon and dark matter density. Our method detects halos—i.e., gravitationally collapsed regions of locally higher density—as superlevel sets exceeding a density threshold and tracks them over time. At the scale of this simulation, individual halos correspond to clumps of matter hosting galaxies and groups of galaxies.

This dataset is challenging due to the vast number of components and their complex evolution; especially at the beginning of the simulation where halos start to form and then progressively cluster together. To illustrate their evolution, we derive a nested tracking graph for density levels 2.632497×10^{12}, 1×10^{12}, and 5×10^{11}. The largest level was suggested by the domain scientists and the other ones were chosen heuristically based on the indicated structures of the cosmic web that are visible in the volume rendered images of the halo dataset. Figure 8 shows that the halos with highest density (red) are embedded in halos with density 1×10^{12} (dark blue), that are in turn embedded in galaxy filaments with density 5×10^{11} (light blue). This hierarchy can be well represented with a nested tracking graph. It is not possible to interactively render the entire graph due to the large number of nodes and edges. We extended our interface to explore the nested graph in a level-of-detail approach by filtering halos below a certain voxel size, collapsing intermediate timesteps, and focusing on single components. Figures 8 and 7 show the nested tracking graph for the same filament but for different timesteps and volume filters. The figures illustrate the evolution of halos for each fifth timestep and filter halos below 40 and 20 voxels. Figure 7 shows an important phase of the simulation where until timestep 285 a large number of new halos are born within the same filament, that are then attracted to each other and merge. E.g., notice the vast number of small, isolated halos (thin red lines) at timestep 250 that merge into two large clusters (thick red lines) until timestep 350. Another observable trend is that the filament and its halos do not significantly increase in size after timestep 285 although new halos are born. Figure 8 shows the same filament at a much later time of the simulation where most halos already converged. The most prominent feature of Figure 8 is already visible in Figure 7, i.e., a single, huge halo containing the most and largest sub-halos.

In contrast to standard tracking graphs, the nested representation shows the evolution of galaxy filaments, contained halos, and how they cluster together over time. According to the domain scientist, containment and clustering are currently not well captured by traditional tracking graphs used in cosmology visualization. Cosmological simulations model very large volumes and cannot account for detailed physics, for example physics relevant for galaxy formation. Instead, reduced models—like gravitational N-body simulations—are used. A substantial challenge in this context is to connect clumps of matter, i.e., halos with certain galaxy types as are observed with telescopes. This is ongoing research [HCT*16], and the connection approaches rely on both assembly history of halos, as well as their environment, both of which are conveniently captured with nested tracking graphs.

5. Conclusion and Future Work

We presented nested versions of tracking graphs that represent the evolution of components and their properties while illustrating their nesting hierarchy. This visualization sets multiple tracking graphs in context with each other and enables users to follow the evolution of components for different levels simultaneously. We have demonstrated our method on three datasets from different application areas and how it can be integrated in a visual analytic framework for interactive data exploration and analysis.

In future work, we plan to extend our methodology to handle large scale datasets that will require more interactive ways to effectively use the resulting nested graphs. To cope with a large number of timesteps and components we will search for ways to summarize time intervals and branches of the graphs. Edges and nodes of the graph can also be linked to other visualizations such as histograms to provide additional information about the components. To improve the layout of the graphs we have to further reduce the number of edge crossings by considering split and merge events of other levels while computing the final layout. Furthermore, to make it easier to link parts of the graph to a rendered image of their associated components we have to incorporate the spatial relationship between components in the layout algorithm. We also plan to explore nested graphs for varying levels for each timestep. In particular, we plan to choose levels according to the contour trees of the individual timesteps and create a link between them. Nested tracking graphs might be capable of visualizing time-varying split and merge trees. We think nested tracking graphs can be applied in various other fields as well. They could be used in general for visualizing time-varying hierarchies that are present in hierarchical clustering, hierarchical diffusion, and threshold based methods.

Acknowledgments

This work was supported in part by the DFG IRTG 2057 “Physical Modeling for Virtual Manufacturing”, EU grant #678727, and the Center for Mathematical and Computational Modeling (CM^2) at the University of Kaiserslautern. Gunther H. Weber was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 through the project “Towards Exascale: High Performance Visualization and Analytics Program”, program manager Lucy Nowell.

References

[ABL*13] Almgren A. S., Bell J. B., Lijewski M. J., Lukić Z., Andel E. V.: Nyx: A Massively Parallel AMR Code for Computational Cosmology. *The Astrophysical Journal 765*, 1 (2013), 39. 9

[ALS*16] Aldrich G., Lukasczyk J., Steptoe M., Maciejewski R., Leitte H., Hamann B.: Viscous Fingers: A Topological Visual Analytics Approach. *IEEE Scientific Visualization Contest* (2016). 1, 2, 4, 5, 7

[BBD*07] Bremer P.-T., Bringa E. M., Duchaineau M. A., Gyulassy A. G., Laney D., Mascarenhas A., Pascucci V.: Topological feature extraction and tracking. *Journal of Physics: Conference Series 78*, 1 (2007), 012007. 1, 2

[BKL*11] Bennett J., Krishnamoorthy V., Liu S., Grout R. W., Hawkes E. R., Chen J. H., Shepherd J., Pascucci V., Bremer P.-T.: Feature-Based Statistical Analysis of Combustion Simulation Data. *IEEE Trans. Vis. Comput. Graph. 17*, 12 (2011), 1822–1831. 1, 2

[BOH11] Bostock M., Ogievetsky V., Heer J.: D3: Data-Driven Documents. *IEEE Trans. Visualization & Comp. Graphics* (2011). 6

[BWT*11] Bremer P. T., Weber G., Tierny J., Pascucci V., Day M., Bell J.: Interactive Exploration and Analysis of Large-Scale Simulations Using Topology-Based Data Segmentation. *IEEE Transactions on Visualization and Computer Graphics 17*, 9 (2011), 1307–1324. 2, 4

[CSA03] Carr H., Snoeyink J., Axen U.: Computing contour trees in all dimensions. *Computational Geometry 24*, 2 (2003), 75 – 94. 2, 3

[EHMP04] Edelsbrunner H., Harer J., Mascarenhas A., Pascucci V.: Time-varying Reeb Graphs for Continuous Space-time Data. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry* (New York, NY, USA, 2004), SCG ’04, ACM, pp. 366–372. 1

[Gra13] Graphviz. http://www.graphviz.org/, 2013. 5

[HCT*16] Hearin A., Campbell D., Tollerud E., Behroozi P., Diemer B., Goldbaum N. J., Jennings E., Leauthaud A., Mao Y.-Y., More S., et al.: High-precision forward modeling of large-scale structure: An open-source approach with halotools. 9

[HLH*16] Heine C., Leitte H., Hlawitschka M., Iuricich F., De Floriani L., Scheuermann G., Hagen H., Garth C.: A Survey of Topology-based Methods in Visualization. *Computer Graphics Forum 35*, 3 (2016), 643–667. 2

[HSKK01] Hilaga M., Shinagawa Y., Kohmura T., Kunii T. L.: Topology Matching for Fully Automatic Similarity Estimation of 3D Shapes. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2001), SIGGRAPH ’01, ACM, pp. 203–212. 2

[IEE16] IEEEVIS: Scientific Visualization Contest. http://www.uni-kl.de/sciviscontest/, 2016. 6

[LBM*06] Laney D., Bremer P. T., Mascarenhas A., Miller P., Pascucci V.: Understanding the Structure of the Turbulent Mixing Layer in Hydrodynamic Instabilities. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (Sept. 2006), 1053–1060. 1, 2

[LMGH15] Lukasczyk J., Maciejewski R., Garth C., Hagen H.: Understanding Hotspots: A Topological Visual Analytics Approach. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2015), GIS ’15, ACM, pp. 36:1–36:10. 1, 2, 4, 6

[LSN*15] Lukić Z., Stark C. W., Nugent P., White M., Meiksin A. A., Almgren A.: The Lyman α forest in optically thin hydrodynamical simulations. *Monthly Notices of the Royal Astronomical Society 446*, 4 (2015), 3697–3724. 9

[MS09] Mascarenhas A., Snoeyink J.: *Isocontour based Visualization of Time-varying Scalar Fields*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 41–68. 1

[OHW*15] Oesterling P., Heine C., Weber G. H., Morozov D., Scheuermann G.: Computing and Visualizing Time-Varying Merge Trees for High-Dimensional Data. Springer. 2

[SB06] Sohn B. S., Bajaj C.: Time-varying contour topology. *IEEE Transactions on Visualization and Computer Graphics 12*, 1 (2006), 14–25. 1, 2, 4

[SW98] Silver D., Wang X.: Tracking scalar features in unstructured data sets. In *Visualization ’98. Proceedings* (1998), pp. 79–86. 1, 2

[WBS*14] Widanagamaachchi W., Bremer P. T., Sewell C., Lo L. T., Ahrens J., Pascuccik V.: Data-parallel halo finding with variable linking lengths. In *2014 IEEE 4th Symposium on Large Data Analysis and Visualization (LDAV)* (2014), pp. 27–34. 2

[WCPB12] Widanagamaachchi W., Christensen C., Pascucci V., Bremer P. T.: Interactive exploration of large-scale time-varying data using dynamic tracking graphs. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (Oct 2012), pp. 9–17. 1, 2