Distributed Path Compression for Piecewise Linear Morse-Smale Segmentations and Connected Components

Michael Will, Jonas Lukasczyk, Julien Tierny, Christoph Garth

Distributed Path Compression for Piecewise Linear Morse-Smale Segmentations and Connected Components

Michael Will* RPTU Kaiserslautern-Landau
Jonas Lukasczyk† RPTU Kaiserslautern-Landau
Julien Tierny‡ CNRS and Sorbonne Université
Christoph Garth§ RPTU Kaiserslautern-Landau

ABSTRACT

This paper describes the adaptation to a distributed computational setting of a well-scaling parallel algorithm for computing Morse-Smale segmentations based on path compression. Additionally, we extend the algorithm to efficiently compute connected components in distributed structured and unstructured grids, based either on the connectivity of the underlying mesh or a feature mask. Our implementation is seamlessly integrated with the distributed extension of the Topology ToolKit (TTK), ensuring robust performance and scalability. To demonstrate the practicality and efficiency of our algorithms, we conducted a series of scaling experiments on large-scale datasets, with sizes of up to 4096^3 vertices on up to 64 nodes and 768 cores.

Index Terms: Distributed algorithms, Scientific visualization.

1 INTRODUCTION

Topological Data Analysis (TDA) has become a popular tool for capturing the inherent structure and features of interest of scalar field data. It has been used for a multitude of visualization and analysis tasks, for example in the areas of fluid and combustion dynamics [28, 30, 36], climate science [4, 9, 37] or astrophysics [45, 49] and more [26, 54]. Topological abstractions capture the global structure of data and allow researchers to extract relevant features more quickly and easily. As dataset sizes continue to grow, relying on a single machine for computing abstractions is becoming increasingly impractical. This is particularly true for methods involving larger data capture, such as increasingly accurate scientific simulations or medical imaging. Generally, shared computation is almost always preferred to distributed computation as communication can quickly become a bottleneck and impede the speed of the computation, especially for global problems such as the computation of TDA abstractions. However, memory limitations necessitate data distribution across multiple machines to efficiently process and analyze large-scale datasets.

One prominent topological abstraction is the Morse-Smale (MS) complex, which segments the domain into areas of similar gradient flow and has been used for the visualization of instabilities in hydrodynamic mixing layers [31], highlighting the dark matter cosmic web [49], feature extraction of combustion simulations [30], feature tracking [38], and many more applications. However, the distributed computation of Morse-Smale complexes is an underdeveloped field [22], and to the best of our knowledge at the time of writing, there is no publicly available implementation.

This paper describes Distributed Path Compression (DPC), an adaption of a well-scaling shared-memory parallel algorithm for computing the MS segmentation [33] to a distributed setting. To evaluate our implementation, we performed weak and strong scaling experiments for DPC. Our results show that due to the global nature of the problem and the additional communication overhead, the original shared-memory parallel implementation always outperforms the distributed version when working on a similar number of threads (e.g., 1 node with 48 threads performs better than 4 nodes with 12 threads). However, for large datasets it is often not possible to acquire singular nodes with the needed memory requirements, necessitating data distribution. Due to the lack of reference implementations, we were not able to compare DPC against other approaches. Thus, we consider the provision of a public implementation for future benchmarks to be a core contribution of this work.

To compare DPC at least somehow to existing implementations, we also describe a modification of DPC for the computation of connected components in structured and unstructured grids (Fig. 1). To this end, we compare the DPC-based connected component computation against the implementation provided in the Visualization Toolkit (VTK). Our results show that DPC requires much less memory and performs better or, at worst, similarly to the VTK equivalent for larger node counts and problem sizes.

To summarize, the main contributions of our work are:

1. a hybrid-parallel algorithm for the computation of Morse-Smale segmentations;
2. a hybrid-parallel algorithm for the computation of connected components based on the principle of path compression; and
3. the integration of both algorithms into TTK for reproducibility, future benchmarks, and utilization by end-users.

2 RELATED WORK

2.1 Morse-Smale Complex

The Morse-Smale complex (MS complex) provides an abstract overview over the gradient flow of the scalar field [47, 48]. Critical points represent areas where the gradient flow is zero and are connected by separatrices, boundary lines segmenting the domain into areas of similar flow. The complex was first formally defined for piecewise linear 2-manifolds by Edelsbrunner et al. [5] and later extended to 3-manifolds [7], where the authors first derive a *quasi MS complex*, which is structurally indistinguishable from the MS complex, but where the arcs may just be of *a* monotone ascent or descent, not the maximal ascent or descent, to then derive the MS complex from it. Bremer et al. [2] compute the MS complex by starting from saddle points and tracing two paths of steepest ascent and two of steepest descent. These paths naturally partition the domain in the cells of the MS complex. Gyulassy et al. [20, 21] extended these ideas to 3D domains and introduced topological simplification based on the MS complex.

Gyulassy et al [17] describe a divide-and-conquer algorithm with on-the-fly simplification to allow for a memory efficient computation. Their approach and the improvement done by Gyulassy in his thesis [23] are also based on discrete Morse theory. Robins et al. [40] extended these ideas to the first provably correct algorithm for computing the MS complex using discrete Morse theory.

*e-mail: mswill@rptu.de
†e-mail: lukasczyk@rptu.de
‡e-mail: julien.tierny@sorbonne-universite.fr
§e-mail: garth@rptu.de

Figure 1: Connected Component extraction for ctBones [52], the magnetic reconnection [16] and the AT complex [52] datasets based on a threshold, which characterize bones of the foot, high-density boundaries and low density areas, respectively. Running these computations on multiple nodes allows us to use much larger datasets by using the distributed memory of all the nodes.

Gyulassy et al. [19] presented a new approach for computing the MS complex, by using streamlines to compute mountains/basins, which leads to an MS complex, whose accuracy depends on the accuracy of the integration used. Günther et al. [15, 24], while mostly aiming to compute persistent homology, used the MS complex as an intermediate step to compute the persistence. The MS complex is extracted by integrating along critical points in parallel. Shivashankar et al. presented algorithms for the parallel computation of 2D [43] and 3D [44] MS complexes, based on discrete Morse theory. They first compute the discrete gradient field and then extract the relevant manifolds as collections of gradient paths, by breadth-first search starting from critical points. Gyulassy et al. [18] presented an algorithm with improved accuracy, while still presenting substantial speedups. They first generate a version on discrete Morse theory and then modify it using the numerically traced features. Subhash et al. [50] presented the first GPU based algorithm for the MS complex. One previous computational bottleneck was correctly identifying the structure of saddle-saddle connection. They compute the connectivity via a series of matrix operations which allows them to count paths completely lock free. Gerber et al. [13] presented the Morse-Smale Approximation which is conceptually similar to our segmentation, however, they follow the steepest k-nearest neighbors, instead of the steepest direct neighbor. Maack et al. [33] computed the ascending and descending segmentation of the domain using path compression on the discrete gradient field. These are then merged into a fast preview of the MS complex. This algorithm is the basis for the distributed version we use.

Concepts like local-global or fully distributed representations of the Morse-Smale complex are still underdeveloped. Gyulassy et al. [22] presented a method in which the data is distributed over multiple blocks, where each block computes their local gradient and MS complex, which are subsequently merged into one complex. However, this approach depends on partial and local simplification to make it feasible for larger complexes, and the distributed implementation is not publicly available.

2.2 Distributed Union-Find / Connected Components

Path Compression can be used as an efficient algorithm type for the *Find* component of the Union-Find data structure. While Union-Find by itself is inherently sequential, there have been efforts towards parallelizing and distributing it. In the following we will present some distributed Union-Find results and elaborate on how they differ from our approach. While there are many other algorithms for computing connected components (such as ones based on parallel domain decomposition [29, 46] or random edge sampling [12, 25]), we solely focus on ones based on Union-Find.

Using a variation of path compression for finding connectivity in shared memory environments was first described by Shiloach and Vishkin [42]. Their theoretical model worked on $n+2m$ processors, with n and m being the number of vertices and edges respectively. Later, Cybenko et al. [3] presented some of the first distributed parallel algorithms for computing connected components based on Union-Find, which can be seen as the basis for many of the following methods, but these algorithms exhibited poor scaling behaviour. However, their distributed model, which is based on merging local subsets of the data until one processor has the complete solution, showed weak scaling behaviour. In contrast, the algorithm by Manne and Patwary [34] is computing as much local Union-Find work as possible to minimize the needed merges. One important distinction to our approach is that we do not care for complete Union-Find forest, but only the final segmentation labeling for each component. Iverson et al. [27] evaluated multiple algorithms for computing connected component labeling of graphs distributed across multiple processors. Our algorithm is a variation of the distributed Union-Find algorithm (which is in turn a variation of the original one by Shiloach and Vishkin [42]), which was shown by them to scale well. Friederici et al. [11] used a distributed version of their previous [10] shared-memory Union-Find algorithm for efficient percolation analysis in turbulent flows. Their algorithm allows for arbitrary representatives for the components, which may lead to more efficient computation, as less pointers need to be rearranged.
In contrast to our method which uses one synchronous communication step and relies on a given distribution done by VTK, Xu et al. [53] present an asynchronous Union-Find method with dynamic redistribution for load-balancing reasons.

3 Background

This section provides the theoretical background of the proposed approach and introduces the notations used throughout the manuscript. For a comprehensive introduction to computational topology, we refer the reader to the textbook of Edelsbrunner and Harer [6].

3.1 Scalar Fields

The input of our approach is a piecewise-linear (PL) *scalar field* $f : \mathcal{K} \to \mathbb{R}$, where real-valued data is given at the vertices of a connected simplicial complex $\mathcal{K}$, and values on edges are linearly interpolated. We denote the vertices (0-simplices) and edges (1-simplices) of the complex $\mathcal{K}$ with $\mathcal{V}(\mathcal{K})$ and $\mathcal{E}(\mathcal{K})$, respectively. Neighbor vertices of a vertex v are denoted by $\mathcal{N}(v,\mathcal{K}) = \{u \in \mathcal{V}(\mathcal{K}) \ : \ \langle v,u\rangle \in \mathcal{E}(\mathcal{K})\}$. $\mathcal{K}$ does not need to be simply connected, but we require that f is injective on the vertices of $\mathcal{K}$, which can always be enforced by applying a variant of *Simulation of Simplicity* [8].

3.2 Critical Points

The sub-level set $f^{-1}_{-\infty}(w)$ of an isovalue $w \in \mathbb{R}$ is defined as $f^{-1}_{-\infty}(w) = \{p \in \mathcal{M} \mid f(p) < w\}$. As w continuously increases, the topology of $f^{-1}_{-\infty}(w)$ changes at specific vertices of $\mathcal{K}$, called the *critical points* of f. Let $Lk^-(v)$ be the *lower link* of the vertex v: $Lk^-(v) = \{\sigma \in Lk(v) \mid \forall u \in \sigma : f(u) < f(v)\}$. The *upper link* of v

is defined symmetrically: $Lk^+(v) = \{\sigma \in Lk(v) \mid \forall u \in \sigma : f(u) > f(v)\}$. A vertex v is *regular* if and only if both $Lk^-(v)$ and $Lk^+(v)$ are simply connected. Otherwise, v is a *critical vertex* of f [1]. A critical vertex v can be classified by its *index* $\mathcal{I}(v)$, which is 0 for minima, 1 for 1-saddles, $(d-1)$ for $(d-1)$-saddles and d for maxima. Vertices for which the number of connected components of $Lk^-(v)$ or $Lk^+(v)$ are greater than 2 are called *degenerate saddles*.

3.3 Ascending and Descending Manifolds

Let $\mathcal{M}(\mathcal{K}) \subset \mathcal{V}(\mathcal{K})$ be the set of maxima of $\mathcal{K}$ for f. Then the *descending manifold* is a map $d : \mathcal{V}(\mathcal{K}) \to \mathcal{M}(\mathcal{K})$ that assigns to each vertex $v \in \mathcal{V}(\mathcal{K})$ the maximum $m \in \mathcal{M}(\mathcal{K})$ that would be reached by following the path starting at v along the steepest ascent on $\mathcal{K}$. This path is a sequence of n vertices $(v = v_1, v_2, ..., v_n = m)$ where $v_{i+1} = \text{argmax}_{u \in \mathcal{N}(v_i, \mathcal{K})} f(u)$. The *ascending manifold* is defined symmetrically for minima reached by following the path starting at v along the steepest descent. These paths are unambiguous since f is injective.

3.4 Distributed Model

Our work is integrated into the Topology ToolKit (TTK) [35,51] and is making use of its distributed capabilities. We will briefly recap some parts in the relevant places, while referring to the introduction paper by Le Guillou et al. [14] for more detail. One of the most important data-structures in TTK is the triangulation, which allows for constant time traversal queries, transformation of local into global ids and reverse, extraction of the rank ids of vertices and more. For this, a pre-processing step is necessary, in which the developer of an algorithm specifies which traversal types will be needed, so they can be pre-computed and cached. As TTK mostly relies on analysis *pipelines*, which allows for many analysis algorithms or *filters* to be chained together, this allows for relevant information to be computed once and used by the whole pipeline.

4 Methods

In this section, we will go over the preprocessing which needs to happen at a previous step of the pipeline; recap the algorithm from Maack et al. [33], present our distribution process; and show how the concept of Distributed Path Compression (DPC) can be applied to compute connected components.

4.1 Preprocessing

We need one layer of ghost vertices for our distribution scheme to work. While every rank knows which of its simplices are actually ghost simplices belonging to other ranks, the inverse is not true (without additional computation): a rank does not intuitively know which other ranks depend on its simplices. Therefore, we *request* information from other ranks and do not preemptively *supply* information. While we do not need unique scalar values for the connected components, we need to remove ambiguity for the MS segmentation. We apply a variant of *Simulation of Simplicity* [8], implemented by TTK in the `ttkArrayPreconditioning` filter, based on globally sorting the vertices according to their scalar values while breaking ties by their global vertex id, and then creating a global order field that assigns to each vertex its corresponding index in the sorted array.

In structured grids, the global ids and the value disambiguation can be computed on-the-fly by special data structures in TTK. For this and the neighborhood relations, we need to precondition the triangulation with the `preconditionDistributedVertices` and `preconditionVertexNeighbors` functions of TTK. In TTK taxonomy terms, our algorithms are in the class of data-dependent communication, because the amount of communication is dependent on the data distribution, even though we only have one communication step. Note that there are two types of vertex ids, local ids which are unique in the ranks, and global ids over the different ranks. For brevity, we omit detailed descriptions of when they have to be converted, but generally local ids are used for addressing arrays in the ranks and global ids are used during the communication phase.

4.2 Ascending and Descending Segmentation

Algorithm 1: DistributedPathCompression

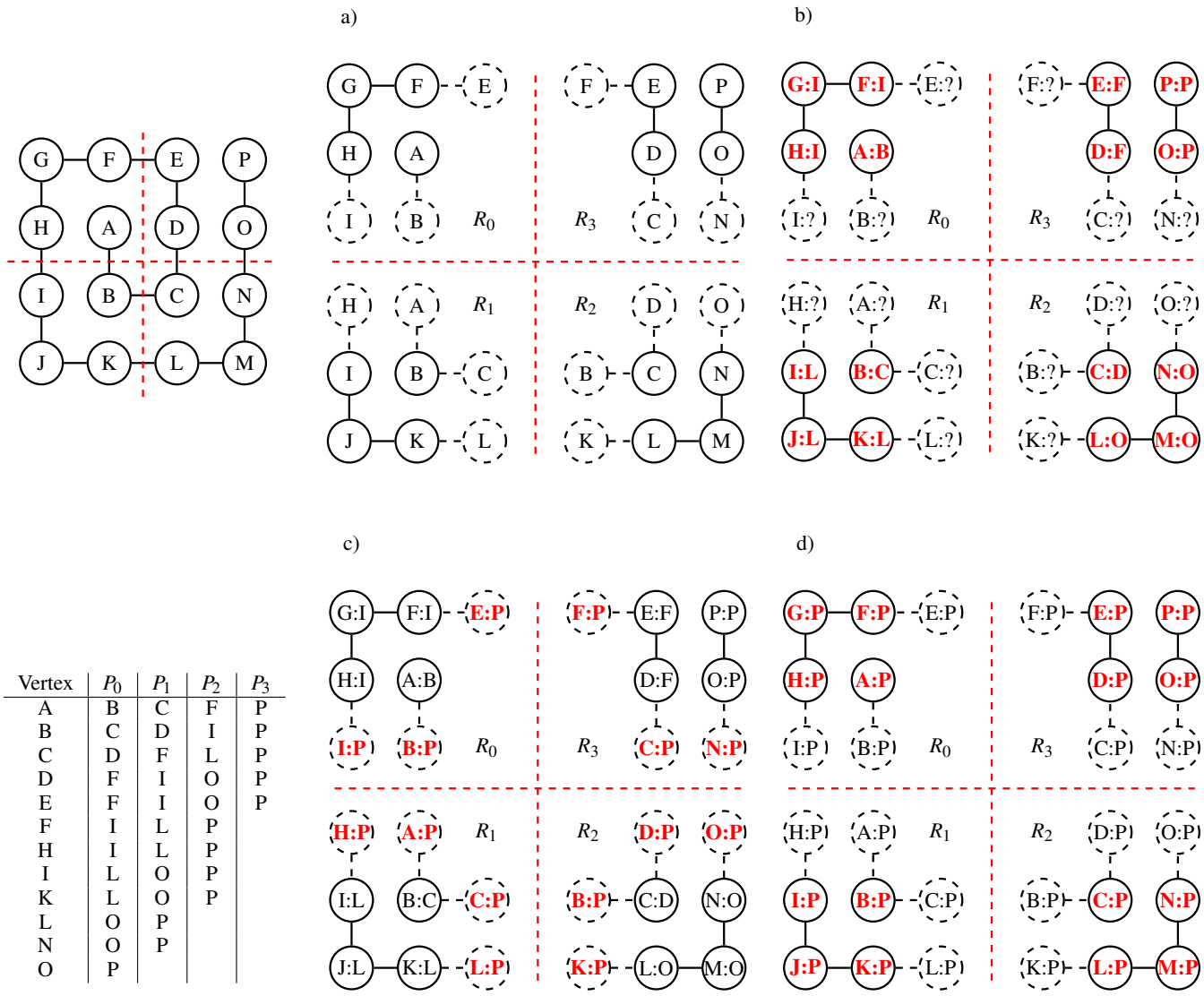
Inputs: • simplical complex $\mathcal{K}$
• scalar field $f : \mathcal{K} \to \mathbb{R}$
Outputs: ○ descending manifold $d : \mathcal{V}(\mathcal{K}) \to \mathcal{M}$ where $\mathcal{M}$ are the maxima of f on $\mathcal{K}$

```
1  d  ← array( |V(K)| )  // create int array with |V(K)| entries
2  gv ← array()  // create (id, rank(id), target)-struct array
3  parallel foreach vertex v ∈ V(K) do
4      if v belongs to the current rank then
5          d[v] ← argmax_{u∈N(v,K)} f(u) // assign v to largest
             neighbor
6      else
7          d[v] ← v  // treat ghost cell vertices as maxima
8          gv.add(v, rank(v))
9  parallel foreach thread t do
10     A ← AssignVerticesToThread(t, V(K))
11     while |A| > 0 do
12         foreach vertex v ∈ A do
13             u ← d[v]  // current pointer of v
14             # atomic read
15             w ← d[u]  // current pointer of u
16             if u = w then
17                 A ← A \ {v}  // delete v from active vertices
18             else
19                 d[v] ← w  // assign w to v
20 // every rank is now finished with their local computation,
     we now need to share segmentations over the ghost vertices
21 ExchangeGhostVertices(K, d, gv)
22 return d
```

The MS segmentations divide the scalar field into areas of similar gradient flow, therefore all vertices of which the steepest ascent and descent terminate in the same extrema, are in the same segment. In the following we will only describe the process for the descending segmentation (steepest ascent), the process for the ascending segmentation is symmetrical.

We use *path compression*, also known as pointer doubling, to efficiently compute the integral lines, as it has been shown to scale well [33, 41]. Initially, each vertex points to its largest neighbor and then, in each global iteration, each vertex points to the vertex its current pointer pointers to. This effectively doubles the step size in each iteration.

We outline the overall distributed algorithm in Alg. 1. At first, each vertex is assigned to its largest neighbor (line 3-5). One important change from the non-distributed to the distributed setting is seen in line 7: as previously described, we make use of ghost cells and vertices. If a vertex is a ghost vertex, it lies on the boundary between two ranks and is present in one rank while actually belonging to another rank. This means that the non-owning rank has no information about the gradient behaviour in the owning rank. Therefore, we pretend those vertices are maxima and let them point to themselves (lines 7-8). We additionally save them separately in a vector to handle them later in the communication step. For this, we use a C++ structure containing the vertex id, the owning rank of this vertex and the actual target to which it should point. The target is initially set to -1 and needs to be filled by the owning rank and communicated back.



Vertex	P_0	P_1	P_2	P_3
A	B	C	F	P
B	C	D	I	P
C	D	F	L	P
D	F	I	O	P
E	F	I	O	P
F	I	L	P	
H	I	L	P	
I	L	O	P	
K	L	O	P	
L	O	P		
N	O	P		
O	P			

Figure 2: Illustration of the distributed path compression (DPC) procedure for one connected polyline (top left). The polyline has the shape of a spiral and is distributed on four ranks whose boundaries are shown by red dashed lines. To compute connectivity, every rank needs one layer of ghost vertices (**a**), dashed nodes and edges). Note, in VTK a vertex can be a ghost vertex in multiple ranks, but every vertex belongs exclusively to a single rank, which is called the vertex owner. The goal of the DPC procedure is to assign to every vertex the largest vertex identifier of its connected component, here P. In the first step of DPC, every rank computes a path compression for all its non-ghost vertices (**b)**). For instance, after this step the R_3 assigns to vertex D and E the ghost vertex F, and vertex O is pointing towards vertex P. For details regarding the path compression on a single rank we refer the reader to the work of Maack et al. [33] and the summary described in Sec. 4.2. The next step involves a cross-rank communication in which all ghost vertices retrieve the current pointers of their owners (table, column P_0). For example, ghost vertex A is owned by the R_0 and is currently pointing towards B, so the R_1 (which contains A as a ghost vertex) retrieves this assignment. Next, DPC performs a path compression on the ghost vertices (table, columns P_1, P_2, P_3). Here, after three iterations all ghost vertices point towards vertex P, which is communicated across ranks (**c)**). Finally, every rank needs to perform one more iteration of a local path compression to correctly update all pointers (**d)**).

To locally perform path compression in parallel, we distribute all vertices among the available threads (lines 9-10). Each thread processes its own list of active vertices, where a vertex remains active as long as it is not pointing to a maximum. For each active vertex v, the thread first retrieves the current pointer u of v (line 13), then retrieves the vertex w pointed to by u (lines 14-15). While only the current thread updates the pointer of v, another thread might update the pointer of u during this process. Therefore, the first lookup does not need synchronization, but the second lookup requires an atomic read lock (line 14). This could be prevented by maintaining two arrays where in one iteration of path compression, you only read from one array and write into the other, and switching the arrays after each iteration. However, this would come at higher memory usage and our experiments have shown that light synchronization outperforms the solution with two arrays. If u and w are equal, indicating that v now points to a maximum, v can be removed from the list of active vertices (line 17). Otherwise, v's pointer is updated to point to w (line 19). Since only the current thread updates v's pointer, this write operation does not require synchronization.

4.3 Distributed Communication

After computing the local segmentations per rank, we need to do one communication phase to compute the correct segmentations over all ranks, described in Alg. 2. For this, each rank sends ids of the needed ghost vertices with their owners to rank 0 (line 4). Rank 0 then changes the ordering from who *needs* the ranks to who *owns* the ranks (line 5-8) and request this information from the actual ranks (line 10). Using the *MPI_Allgather* command (line 13), the actual targets for requested vertices get shared with all ranks. We need to share this with every rank, because a vertex in the owning rank may actually point to another ghost vertex at the other end of the rank, when segmentations are stretching over multiple ranks. Each rank now has all the information needed to build up a local ghost vertex pointer graph which can be locally compressed one last time (line 15-25). Finally, they need to walk over their vertices and replace all the ones pointing to ghost vertices with the correct targets received and compressed earlier (line 27-33).

In our distribution approach (highlighted in Alg. 2 and used in the other algorithms), rank 0 performs additional organizational work. Specifically, rank 0 performs a global path compression on the ghost vertices and then communicates the result to all ranks such that they can update their pointers in a single iteration. Alternatively, this could be implemented solely based on neighbor-to-neighbor communication as opposed to one-to-all. However, to resolve segments that stretch across multiple ranks, this approach requires multiple iterations in which ghost-cell pointers are updated between neighbors. Additionally, this approach also introduces technical problems while resolving local and global ids, because a rank might retrieve pointers to vertices not belonging to it (neither as a ghost vertex, nor a normal vertex). The build-in data structures in TTK do not support this case. Our initial experiments have shown that the alternative approach incurs a higher communication overhead and requires custom data structures to resolve ids, which is why we opted to the first approach that guarantees convergence in one iteration at the expense of a global one-to-all communication.

4.4 Connected Components

Our algorithm for computing the MS segmentations can easily be adapted for computing connected components, either implicitly based on a given feature mask or explicitly based on extracted geometry. We describe it in Alg. 3.

Our algorithm for connected components follows out of our method for MS segmentations. Before our main algorithm, we compute a feature mask on our scalar field, which is generic and can highlight any areas of interest e.g. all vertices with a value exceeding a given threshold. Then, we start as previously, by letting all vertices

Algorithm 2: ExchangeGhostVertices

Inputs: • simplical complex $\mathcal{K}$
• some segmentation $d : \mathcal{K} \to \mathbb{R}$, with
• array of (id, rank(id), target)-structs gv
Outputs: ◦ segmentation $d : \mathcal{K} \to \mathbb{R}$, with rank boundaries correctly resolved

```
1  globalSize ← 0
2  allValuesFromRanks ← array()
3  Allreduce(gv.size(), globalSize, +) // each rank knows how
     many ghost vertices are needed
4  Gather(gv, 0, allValuesFromRanks) // Rank 0 gets all the
     needed ids, along with the ranks which need them
5  if rankId == 0 then
6  |  neededPerRanks ← array(array(|ranks|))
7  |  foreach struct s ∈ allValuesFromRanks do
8  |  |  neededPerRanks[s.rank].add(s)
9  receivedIds ← array()
10 Scatter(neededPerRanks, receivedIds, 0)  // each rank gets
     the information which of its ids are needed by some other
     rank
11 foreach struct s ∈ receivedIds do
12 |  s.target ← d[s.id]
13 Allgather(receivedIds, allValuesFromRanks) // every rank
     now know where every ghost points to in the neighboring
     rank
14 // now we need one last path compression to resolve
     segmentations over multiple ranks
15 parallel foreach thread t do
16 |  A ← AssignIdsToThread(t, i ∈ receivedIds)
17 |  while |A| > 0 do
18 |  |  foreach id i ∈ A do
19 |  |  |  u ← receivedIds[v].target // current pointer of target
20 |  |  |  # atomic read
21 |  |  |  w ← receivedIds[u].target // current pointer of u
22 |  |  |  if u = w then
23 |  |  |  |  A ← A \ {v}  // delete v from active vertices
24 |  |  |  else
25 |  |  |  |  receivedIds[v].target ← w  // assign w to v
26 // finally, replace each vertex pointing to a ghost vertex
     with the correct value (possible from multiple ranks away)
27 parallel foreach thread t do
28 |  A ← AssignVerticesToThread(t, V(K))
29 |  while |A| > 0 do
30 |  |  foreach vertex v ∈ A do
31 |  |  |  u ← d[v]  // current pointer of v
32 |  |  |  if u does not belong to the current rank then
33 |  |  |  |  d[v] ← receivedIds[u].target
34 return d
```

with a positive feature mask point to their largest neighbors, but with two important changes: the largest neighbor is not chosen based on the actual scalar value, but on the scalar id (therefore they can also be computed on pure geometry without any scalar data on it) and we only consider the largest neighbor for which the feature mask is also positive (line 6). Similar to the segmentation algorithm, we let ghost vertices point to themselves and collect them for later(lines 8-9), but unlike in the segmentation algorithm, we do not need to exchange all the ghost vertices with our neighboring ranks, but only the masked ones, which can significantly improve performance. Vertices with a non-positive feature mask immediately point to some negative value and are not considered for the further path compression steps (line 12).

Algorithm 3: ComputeConnectedComponents

Inputs: • simplicial complex $\mathcal{K}$
• feature mask $m : \mathcal{K} \to \{0,1\}$
Outputs: ○ segmentation d of $\mathcal{K}$ into connected components, with the segmentation label being the highest vertex id in the segmentation

1 $d \leftarrow array(\,|\mathcal{V}(\mathcal{K})|\,)$ *// create int array with $|\mathcal{V}(\mathcal{K})|$ entries*
2 $fv \leftarrow array()$ *// create (id, rank(id), target)-struct array*
3 **parallel foreach** vertex $v \in \mathcal{V}(\mathcal{K})$ **do**
4 **if** $m(v) = 1$ **then**
5 **if** v belongs to the current rank **then**
6 $d[v] \leftarrow \text{argmax}_{u \in \mathcal{N}(v,\mathcal{K})\ \&\ m(u)==1}\ id(u)$ *// assign v to the neighbor with the largest id, which is also part of the feature*
7 **else**
8 $d[v] \leftarrow$ v *// treat ghost cell vertices as maxima*
9 fv.add(v, rank(v))
10
11 **else**
12 $d[v] \leftarrow -1$
13 **parallel foreach** thread t **do**
14 $A \leftarrow$ AssignVerticesToThread(t, $\mathcal{V}(\mathcal{K})$)
15 **while** $|A| > 0$ **do**
16 **foreach** vertex $v \in A$ **do**
17 $u \leftarrow d[v]$ *// current pointer of v*
18 # atomic read
19 $w \leftarrow d[u]$ *// current pointer of u*
20 **if** $u = w$ **then**
21 $A \leftarrow A \setminus \{v\}$ *// delete v from active vertices*
22 **else**
23 $d[v] \leftarrow w$ *// assign w to v*
24 *// finished first pathbcompression, now we need to stitch segments together*
25 **parallel foreach** vertex $v \in \mathcal{V}(\mathcal{K})$ **do**
26 **if** $m(v) = 1$ **then**
27 **foreach** $u \in \mathcal{N}(v,\mathcal{K})$ **do**
28 **if** $d[u] > d[v]$ **then**
29 d[d[v]] $\leftarrow d[u]$ *// the target of this segmentation will point to the target of the neighboring segmentation, such that one further path compression merges them*
30 **parallel foreach** thread t **do**
31 $A \leftarrow$ AssignVerticesToThread(t, $\mathcal{V}(\mathcal{K})$)
32 **while** $|A| > 0$ **do**
33 **foreach** vertex $v \in A$ **do**
34 $u \leftarrow d[v]$ *// current pointer of v*
35 # atomic read
36 $w \leftarrow d[u]$ *// current pointer of u*
37 **if** $u = w$ **then**
38 $A \leftarrow A \setminus \{v\}$ *// delete v from active vertices*
39 **else**
40 $d[v] \leftarrow w$ *// assign w to v*
41 ExchangeForeignVertices(K, d, fV)
42 **return** d

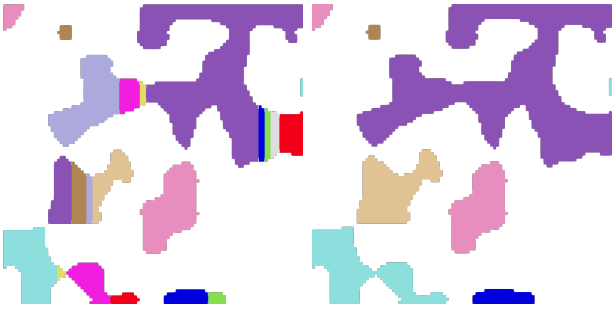


Figure 3: Before (left) and after (right) applying the second pass path compression to merge the sub-segmentations in the connected components. How the segmentations are actually merged is not relevant, it just has to be done in a consistent manner. We have chosen that the segmentation whose target has a lower id gets attached to the one with the higher id. In our example dataset, the id-generation is dominated by the y-direction, therefore the remaining labeling is the one whose segment stretches furthest in positive x-direction.

We then run our path compression on the relevant vertices until convergence (lines 13-23). These are not the correct final segmentations (as can be seen in Fig. 3), as one connected component can have multiple local maxima in their id distribution. Therefore we need to initialize a second iteration of path compression with slightly different starting conditions: each (featured) vertex does not point to the largest neighbor, but to the neighbor with the largest pointer (line 29). After one more path compression (lines 30-40) we have the correct local segmentation into connected components. Finally, the distributed communication phase is the same as in the MS segmentation algorithm, the actual distribution does not care what the pointers actually signify.

5 Experiments

Both of our algorithms are implemented within the *Topology ToolKit (TTK)* [14, 51] and heavily utilize its data structures. The experimental workloads were delivered via python pipelines executed with ParaView v5.12.1. All experiments were run on the Elwetritsch cluster of RPTU Kaiserslautern-Landau on up to 64 nodes, with up to 12 cores per node, for a maximum of 768 cores in a hybrid distributed-shared setting.

We expect the computation of the ascending and descending segmentation to not scale well, due to the global nature of the problem. Our goal is therefore to *be able* to compute this in a distributed setting at all.

We evaluate both of our algorithms on synthetically generated datasets of various resolutions based on one layer of *Perlin Noise* [39] with an amplitude of one and frequency in every dimension of 0.1. Additionally, our weak scaling study is run on a simulation of the electronic density in the Adenine Thymine complex (AT). This dataset is resampled using the pipelines from Guillou et al. [14] according to the number of nodes used. A complete overview of the timings can be seen in Tab. 1. For all algorithms we exclude any preprocessing steps (exchanging ghost cells, computing order arrays, computing feature masks, extracting geometry etc), as they would probably have to be done at some point in the pipeline for some of the MPI filters either way.

We compare our connectivity computation via Distributed Path Compression (DPC) with the existing VTK Connectivity filter, which works in a distributed setting. It follows the same principles as computing connected components via DPC, with first running a local connectivity algorithm (a connected wave propagation), creating a graph of region connections across ranks (every rank gets this graph) and then running a connected component algorithm on this graph and relabeling to the correct ids. However, it automati-

cally transforms the data from structured grids to unstructured grids which may lead to extremely high memory usage, depending on the size of the region of interest. In contrast, DPC can work on *implicitly thresholded grids*, which maintains the data structure, but assigns a negative value and segmentation id to non-relevant areas. Therefore, we always need one extra array of memory that is the same size as the original grid and uses the same type of ids (either 32- or 64-bit ids). Most of the computation is done in-place, apart from the communication step. However, the additional memory needed is bounded upwards by the amount of ghost vertices, which is negligible compared to the size of the whole dataset, for sensible dataset size / node count configurations. If the regions of interest are sufficiently small or need to be extracted explicitly, this can easily be done post-hoc after computing them implicitly with a simple threshold operation for values larger than zero.

We ran into several problems with VTK and TTK when trying to run experiments with additional large-scale datasets:

VTK Ghost Cell Computation When running our analysis pipelines with 16 or more nodes letting VTK compute ghost simplices became increasingly unreliable. When using verbose output of pvbatch one could see that `vtkDIYGhostUtilities` froze in the "Exchanging ghost data between blocks" step. Explicitly calling the Ghost Cell Generator helped in some cases, but for many datasets it still froze at that point (but not always). Using the AT example pipeline with resampling worked, but also only by adding an explicit Ghost Cell Generator. We also tried saving the datasets into a more parallel friend format on fewer nodes and then processing it on more nodes, to no avail. ParaView, the visualization software we used to call VTK, changed the GhostCellGenerator and removed the legacy generator in version 5.11.0, it is currently being investigated whether this issue also occurred in the legacy generator.

VTK / TTK Unstructured Grid Distribution On more than 8 nodes, when switching from an Implicit to an Explicit Triangulation, such as when explicitly extracting all values above a specific threshold, and afterwards triangulating it, TTK would sometimes get such in an endless loop while trying to precondition this triangulation. This issue is known, however it is still unclear what exactly causes this issue and whether it is due to ParaView distribution or due to the TTK triangulation.

5.1 Strong Scaling

For our strong scaling study we have run the algorithms on Perlin Noise of multiple grid sizes, at 512^3, 1024^3, 2048^3 and 4096^3. While the larger grid was more beneficial for larger node counts, due to memory constraints only the two highest node count configurations could be run. Therefore, we mainly focus on the large grid and compare timings starting from 4 nodes and doubling the node count up to 64 nodes. The timings can be seen plotted for DPC in Fig. 4 and for computing connected components in Fig. 6. The raw timing in seconds are presented in Tab. 1. A comparison of speedup and parallel efficiency can be seen in Fig. 5 for DPC and in Fig. 7 for computing connected components.

We have seen that the distribution step of path compression does not scale well, as more nodes significantly increases the size of the needed communication. However, the computation of connected components scales well as only few components stretch over multiple ranks and need to be distributed. This shows that the computation of connected components is more dependant on the actual data distribution, while path compression is more independent but can be much slower.

5.2 Weak Scaling

We run weak scaling experiments based on Perlin Noise with a high threshold and on the resampled AT complex, the timings can be found in Tab. 2. Fig. 8 plots the timings and weak parallel efficiency for the AT complex and Perlin Noise.

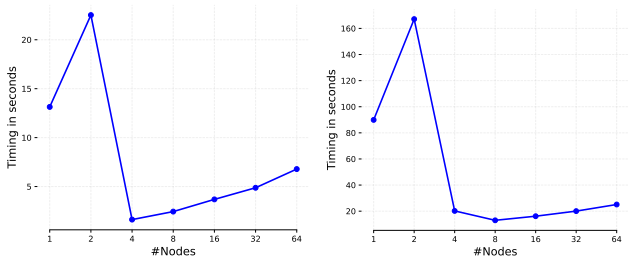


Figure 4: Timing for the strong scaling experiments of DPC based on Perlin Noise at a grid size of 512^3 (left) and 1024^3 (right) vertices.

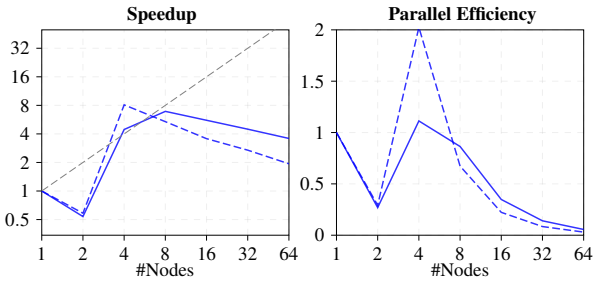


Figure 5: Illustration of parallel speedup and efficiency for DPC on Perlin Noise at 512^3 (dashed blue line) and 1024^3 (solid blue line). The left plot shows the parallel speedup defined as the runtime of one node divided by the runtime of n nodes, perfect scalability is marked with the dashed gray line. The right plot shows the parallel efficiency as the speedup divided by the number of nodes. The plot shows that the distribution step of path compression does not scale well, as more nodes significantly increase the size of the needed communication.

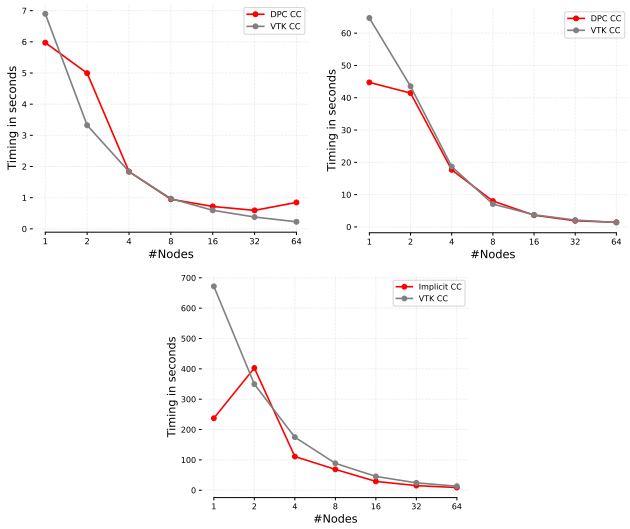


Figure 6: Timing for the strong scaling experiments for computing connected components with DPC (red) and with the VTK Connectivity filter (gray) based on Perlin Noise at a grid size of 512^3 (upper left), 1024^3 (upper right) and 2048^3 (lower) vertices.

One can see that computing the connected components implicitly is at least as fast as computing it with the VTK connectivity filter and they show very similar scaling behaviour, with the implicit connected components scaling slightly worse for the Perlin Noise

Size in #Vertices	Algorithm	1N	2N	4N	8N	16N	32N	64N
512^3	Segmentation	11.291	20.629	1.468	2.440	3.687	4.875	6.785
1024^3	Segmentation	86.044	167.134	17.417	13.010	16.148	20.046	25.103
2048^3	Segmentation	905.872	2165.203	215.517	106.430	91.730	-	-
512^3	DPC CC	5.973	4.994	1.835	0.950	0.718	0.592	0.847
	VTK CC	6.898	3.323	1.836	0.969	0.595	0.379	0.226
1024^3	DPC CC	44.7504	41.451	17.683	8.023	3.640	1.908	1.446
	VTK CC	64.692	43.553	18.686	7.086	3.756	2.131	1.367
2048^3	DPC CC	237.242	402.772	111.073	68.752	29.125	15.188	8.912
	VTK CC	671.906	349.608	174.838	88.756	45.556	24.410	13.350
4096^3	DPC CC	-	-	-	-	277.634	129.562	73.757
	VTK CC	-	-	-	-	372.957	198.001	109.066

Table 1: Timing results of our experiments run on Perlin Noise first grouped by dataset size, and then by algorithm, comparing connected component computation using Distributed Path Compression (DPC) and the VTK Connectivity filter. The remaining columns show, per node count, the total runtime of the different algorithms in seconds. For some datasets the computations ran out of memory or failed because of the previously described reasons and are therefore missing from the table.

Dataset	#Nodes	1	2	4	8	16	32	64
AT complex	Segmentation Computation	1.963	8.505	1.359	2.569	5.885	11.839	26.006
	Implicit CC	0.560	1.226	1.735	4.038	8.709	16.236	25.810
	Explicit CC	0.012	0.312	1.079	7.384	20.126	50.634	-
	VTK CC	0.242	0.691	4.600	18.867	47.442	109.149	-
Perlin Noise	Segmentation Computation	1.477	5.453	0.798	2.435	5.900	10.452	25.414
	Implicit CC	0.604	1.542	0.931	1.034	1.098	1.312	1.615
	Explicit CC	0.010	0.886	0.819	1.048	-	-	-
	VTK CC	0.730	0.813	0.832	0.968	1.031	1.241	1.361

Table 2: Timing for Weak Scaling experiments based on the Adenine Thymine (AT) complex and on Perlin Noise, resampled to different sizes to increase with increased node count, starting from 256^3 vertices and doubling with every doubled node count, up to 1024^3 vertices. Algorithms marked with - could not be run due to memory constraints.

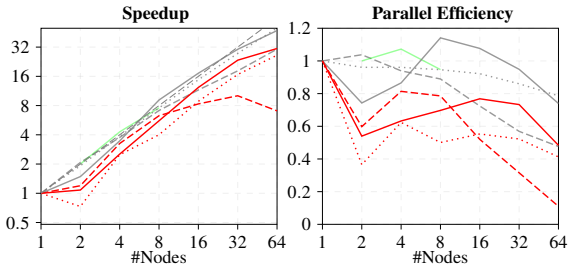


Figure 7: Comparison of parallel speedup and efficiency between computing connected components based on a feature mask implicitly with DPC (red), extracting this feature and explicitly computing connected components with DPC (green, could not be run on all configurations and starting at 2 nodes) and with the VTK Connectivity filter (gray) on Perlin Noise at 512^3 (dashed), 1024^3 (solid) and 2048^3 (dotted) vertices, thresholded such that the top 10% of vertices are masked as relevant and are extracted. The left plot shows the parallel speedup (y-axis) defined as the runtime of one node divided by the runtime of n nodes (x-axis), perfect scalability is marked with the dashed gray line. The right plot shows the parallel efficiency (y-axis) as the speedup divided by the number of nodes (x-axis). The plot shows that the computation of connected components scales well when only few components stretch over multiple ranks and need to be distributed. The smaller grid leads to highly unstable timings at high node counts.

and slightly better for the AT complex. However, Perlin Noise was thresholded with a high scalar value, extracting only little geometry and leaving little actual work to do. In contrast, looking at the AT complex, where more geometry was extracted, VTK connectivity failed at the highest node-count / dataset size configuration, due to memory issues.

5.3 Extraction based on different thresholds

One major advantage of our connected components computation is that it can run *implicitly* on a pre-computed feature mask. Therefore we also ran experiments with Perlin Noise of size 1024^3 and three different computations of features: first we normalize the scalar values and then we either mark everything above 0.9, above 0.5 or above 0.1 as a feature. The values of Perlin Noise are normally distributed, so therefore we either mark almost nothing ($\approx 0.06\%$, an unstructured grid with 671 960 vertices and 206 993 cells, for an original grid size of 1024^3), roughly half of the domain, or almost everything($100\% - \approx 0.06\%$) as a feature. While this feature masking works easily for our approach, the VTK connectivity filter only works on actively extracted geometry. This leads to drastically higher memory usage and possibly worse timing behaviours. We present the timings for this in Tab. 3, run on different node counts with a Intel Xeon Gold 6126 and large amounts of memory each (256GB per node for 1,2 and 4, 64GB per node for the rest). Note, that for the lowest thresholds, most of the configurations could not be run for the VTK connectivity, due to the high memory usage.

While the VTK connectivity is faster than our method for high thresholds (less geometry / less work), we see that it scales worse with increased geometry than our method, with our method being

Size in #Vertices	Algorithm	1N	2N	4N	8N	16N	32N
Top 10%	Implicit CC	47.285	23.433	9.086	4.686	2.582	1.431
	VTK CC	2.394	1.308	0.737	0.410	0.255	0.256
Top 50%	Implicit CC	163.410	217.288	189.417	95.898	49.360	26.345
	VTK CC	675.794	396.708	210.169	94.0211	62.176	38.642
Top 90%	Implicit CC	226.593	368.826	356.776	181.304	92.330	48.426
	VTK CC	-	-	523.862	241.716	118.921	78.366

Table 3: Timing results of implicit connected components and VTK connectivity run on Perlin Noise of size 1024^3 with different feature thresholds, extracting the Top 10%, Top 50% and Top 90% of the values. We see that, while being faster for small data, for larger data / more extraction, VTK connectivity drastically slows down for larger extraction. Additionally, most of the more intensive workflows could not be run at all due to memory constraints.

significantly faster at lower thresholds and the connected components being able to be computed even on one node (even when restricting their memory to 64GB). In contrast, for the two lower thresholds, the distributed memory of 4 nodes, so 1024GB in total, were needed.

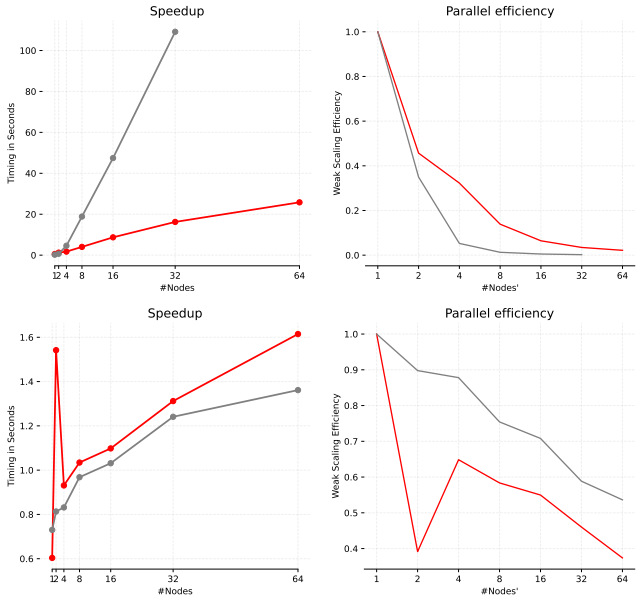


Figure 8: Timing (left) and parallel efficiency (right) for the weak scaling experiments of DPC (red) and the VTK Connectivity filter (gray) based on the AT complex (upper) and Perlin Noise (lower) with a low amount of masked vertices and extracted geometry. Perfect weak scaling parallel efficiency would be 1. The VTK based connected components, running on unstructured grids, could not be run on the AT complex for the last configuration, due to memory constraints.

5.4 Limitations

It is not only possible to implicitly compute the connected components via DPC, but to extract geometry and then run it on the geometry without a feature mask. However, in addition to the previously described problems with the distribution of unstructured grids in TTK, this requires the usage of a triangulation and further increases the memory usage, making it less efficient than the existing VTK connectivity filter.

One of the primary limitations of DPC is its poor scalable efficiency. The global nature of the path compression paradigm necessitates extensive data exchange between nodes. As the number of ranks grows, the overhead associated with maintaining global consistency increases significantly, leading to worse performance and reduced scalability. This issue becomes particularly pronounced when the size of the dataset grows slower than the number of nodes.

Another limitation is related to the current global communication, which may lead to congestion and redundant work done on the nodes. Currently, rank 0 collects the edges for inter-rank connectivity with the specific path compression targets and sends them to all the other ranks, which build up a connectivity graph and path compress it on their own. Other approaches would be to let rank 0 compress the graph and only send the relevant edges to each rank or to shift to inter-node, neighborhood communication, which would lead to more communication steps with interleaved path compressions, but less work in each communication step.

There may also be ways to further reduce the amount of ghost vertices which need to be sent to minimize communication. For connected components, it may be feasible to only send a few ghost vertices at the boundaries because they may belong to the same component. Including those ghost vertices which only point to other ghost vertices will also reduce the communication sent, but increase the program complexity to account for all edge cases.

6 Conclusion

We described an adaption of a well-scaling parallel algorithm for computing Morse-Smale segmentations to a distributed setting, additionally using it as a base to efficiently compute connected components on distributed structured and unstructured grids. Furthermore, we provide an implementation in TTK, which is open source and integrated in the widely-used ParaView visualization environment, and conduct a series of scaling experiments on large-scale datasets in distributed environments.

Future work will focus on further optimizing communication patterns and exploring additional applications of our methods in various scientific domains. The integration of our algorithms into TTK provides a solid foundation for continued development and application of topological data analysis tools in distributed computing environments. Some experiments could not be run due to shortcomings of VTK and TTK. We aim to further investigate these problems, address them and run more comprehensive benchmarks, with different scientific visualization datasets and feature masks, and comparing different communication approaches. Currently, resolving path compression on the ghost vertices requires global communication. It is possible to change this to only need rank neighborhood communication, at the cost of additional compression and communication steps. Additionally, there is a clear trade-off between dataset size and number of ranks. In contrast to shared memory parallelism where more threads will often give at least *some* performance improvement, with distributed computations, more ranks can significantly worsen performance, due to the communication increase. Therefore it is paramount to use as many threads as possible per rank and only as many ranks as are actually needed for the dataset. We plan to conduct more thorough experiments on these trade-offs to find out when this is worth it. Additionally, there are ways to additionally minimize

the amount of ghost vertices which need to be communicated, such as compressing paths along ghost vertices, but only if they point to other ghost vertices. These optimizations possibly introduce many edge cases and therefore need to be cautiously evaluated.

Having the ability to compute the ascending and descending segmentations in a distributed setting allows us to efficiently compute the *extremum graph* used in the merge tree algorithm ExTreeM [32]. As there is currently no distributed merge tree algorithm available in TTK, extending ExTreeM to a distributed setting would allow for much more sophisticated analysis on large datasets.

Acknowledgments

This work is partially supported by the European Commission grant ERC- 2019-COG "TORI" (ref. 863464, https://erc-tori.github.io/).

References

[1] T. F. Banchoff. Critical Points and Curvature for Embedded Polyhedral Surfaces. *The American Mathematical Monthly*, 77(5):475, May 1970. doi: 10.2307/2317380 3

[2] P.-T. Bremer, B. Hamann, H. Edelsbrunner, and V. Pascucci. A topological hierarchy for functions on triangulated surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):385–396, July 2004. Conference Name: IEEE Transactions on Visualization and Computer Graphics. doi: 10.1109/TVCG.2004.3 1

[3] G. Cybenko, T. G. Allen, and J. E. Polito. Practical parallel Union-Find algorithms for transitive closure and clustering. *International Journal of Parallel Programming*, 17(5):403–423, Oct. 1988. doi: 10.1007/BF01383882 2

[4] H. Doraiswamy, V. Natarajan, and R. S. Nanjundiah. An Exploration Framework to Identify and Track Movement of Cloud Systems. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2896–2905, Dec. 2013. doi: 10.1109/TVCG.2013.131 1

[5] Edelsbrunner, Harer, and Zomorodian. Hierarchical Morse—Smale Complexes for Piecewise Linear 2-Manifolds. *Discrete & Computational Geometry*, 30(1):87–107, May 2003. doi: 10.1007/s00454-003-2926-5 1

[6] H. Edelsbrunner and J. Harer. *Computational Topology*. American Mathematical Society, Providence, Rhode Island, Dec. 2009. doi: 10.1090/mbk/069 2

[7] H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci. Morse-smale complexes for piecewise linear 3-manifolds. In *Proceedings of the nineteenth annual symposium on Computational geometry*, SCG '03, pp. 361–370. Association for Computing Machinery, New York, NY, USA, June 2003. doi: 10.1145/777792.777846 1

[8] 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, Jan. 1990. doi: 10.1145/77635.77639 2, 3

[9] W. Engelke, T. B. Masood, J. Beran, R. Caballero, and I. Hotz. Topology-Based Feature Design and Tracking for Multi-center Cyclones. In I. Hotz, T. Bin Masood, F. Sadlo, and J. Tierny, eds., *Topological Methods in Data Analysis and Visualization VI*, pp. 71–85. Springer International Publishing, Cham, 2021. Series Title: Mathematics and Visualization. doi: 10.1007/978-3-030-83500-2_5 1

[10] A. Friederici, M. Atzori, R. Vinuesa, P. Schlatter, and T. Weinkauf. An efficient algorithm for percolation analysis and its appliction to turbulent duct flow. In *Euromech Colloquium*, vol. 598, 2018. 2

[11] A. Friederici, W. Köpp, M. Atzori, R. Vinuesa, P. Schlatter, and T. Weinkauf. Distributed Percolation Analysis for Turbulent Flows. In *2019 IEEE 9th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 42–51, Oct. 2019. doi: 10.1109/LDAV48142.2019.8944383 2

[12] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pp. 492–501, Oct. 1986. ISSN: 0272-5428. doi: 10.1109/SFCS.1986.9 2

[13] S. Gerber, P.-T. Bremer, V. Pascucci, and R. Whitaker. Visual Exploration of High Dimensional Scalar Functions. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1271–1280, Nov. 2010. Conference Name: IEEE Transactions on Visualization and Computer Graphics. doi: 10.1109/TVCG.2010.213 2

[14] E. L. Guillou, M. Will, P. Guillou, J. Lukasczyk, P. Fortin, C. Garth, and J. Tierny. TTK is Getting MPI-Ready, 2024. 3, 6

[15] D. Gunther, J. Reininghaus, I. Hotz, and H. Wagner. Memory-Efficient Computation of Persistent Homology for 3D Images Using Discrete Morse Theory. In *2011 24th SIBGRAPI Conference on Graphics, Patterns and Images*, pp. 25–32, Aug. 2011. ISSN: 2377-5416. doi: 10.1109/SIBGRAPI.2011.24 2

[16] 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. doi: 10.1103/PhysRevLett.113.155005 2

[17] A. Gyulassy, P.-T. Bremer, B. Hamann, and V. Pascucci. A Practical Approach to Morse-Smale Complex Computation: Scalability and Generality. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1619–1626, Nov. 2008. Conference Name: IEEE Transactions on Visualization and Computer Graphics. doi: 10.1109/TVCG.2008.110 1

[18] A. Gyulassy, P.-T. Bremer, and V. Pascucci. Shared-Memory Parallel Computation of Morse-Smale Complexes with Improved Accuracy. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):1183–1192, Jan. 2019. Conference Name: IEEE Transactions on Visualization and Computer Graphics. doi: 10.1109/TVCG.2018.2864848 2

[19] A. Gyulassy, D. Gunther, J. A. Levine, J. Tierny, and V. Pascucci. Conforming Morse-Smale Complexes. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2595–2603, Dec. 2014. doi: 10.1109/TVCG.2014.2346434 2

[20] A. Gyulassy, V. Natarajan, V. Pascucci, P.-T. Bremer, and B. Hamann. A topological approach to simplification of three-dimensional scalar functions. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):474–484, July 2006. Conference Name: IEEE Transactions on Visualization and Computer Graphics. doi: 10.1109/TVCG.2006.57 1

[21] A. Gyulassy, V. Natarajan, V. Pascucci, and B. Hamann. Efficient computation of Morse-Smale complexes for three-dimensional scalar functions. *IEEE transactions on visualization and computer graphics*, 13(6):1440–1447, 2007. doi: 10.1109/TVCG.2007.70552 1

[22] A. Gyulassy, V. Pascucci, T. Peterka, and R. Ross. The Parallel Computation of Morse-Smale Complexes. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 484–495, May 2012. ISSN: 1530-2075. doi: 10.1109/IPDPS.2012.52 1, 2

[23] A. G. Gyulassy. *Combinatorial construction of morse-smale complexes for data analysis and visualization*. phd, University of California at Davis, USA, 2008. AAI3350730 ISBN-13: 9781109061529. 1

[24] D. Günther, J. Reininghaus, H. Wagner, and I. Hotz. Efficient computation of 3D Morse–Smale complexes and persistent homology using discrete Morse theory. *The Visual Computer*, 28(10):959–969, Oct. 2012. doi: 10.1007/s00371-012-0726-8 2

[25] S. Halperin and U. Zwick. Optimal Randomized EREW PRAM Algorithms for Finding Spanning Forests. *Journal of Algorithms*, 39(1):1–46, Apr. 2001. doi: 10.1006/jagm.2000.1146 2

[26] C. Heine, H. Leitte, M. Hlawitschka, F. Iuricich, L. De Floriani, G. Scheuermann, H. Hagen, and C. Garth. A Survey of Topology-based Methods in Visualization. *Computer Graphics Forum*, 35(3):643–667, June 2016. doi: 10.1111/cgf.12933 1

[27] J. Iverson, C. Kamath, and G. Karypis. Evaluation of connected-component labeling algorithms for distributed-memory systems. *Parallel Computing*, 44:53–68, May 2015. doi: 10.1016/j.parco.2015.02.005 2

[28] J. Kasten, J. Reininghaus, I. Hotz, and H.-C. Hege. Two-Dimensional Time-Dependent Vortex Regions Based on the Acceleration Magnitude. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2080–2087, Dec. 2011. doi: 10.1109/TVCG.2011.249 1

[29] S. Lamm and P. Sanders. Communication-efficient Massively Distributed Connected Components. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 302–312, May 2022. ISSN: 1530-2075. doi: 10.1109/IPDPS53621.2022.00037 2

[30] A. G. Landge, V. Pascucci, A. Gyulassy, J. C. Bennett, H. Kolla, J. Chen, and P.-T. Bremer. In-Situ Feature Extraction of Large Scale

Combustion Simulations Using Segmented Merge Trees. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1020–1031. IEEE, New Orleans, LA, USA, Nov. 2014. doi: 10.1109/SC.2014.88 1

[31] D. Laney, P.-t. Bremer, A. Mascarenhas, P. Miller, and V. Pascucci. Understanding the Structure of the Turbulent Mixing Layer in Hydrodynamic Instabilities. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1053–1060, Sept. 2006. Conference Name: IEEE Transactions on Visualization and Computer Graphics. doi: 10.1109/TVCG.2006.186 1

[32] J. Lukasczyk, M. Will, F. Wetzels, G. H. Weber, and C. Garth. ExTreeM: Scalable Augmented Merge Tree Computation via Extremum Graphs. *IEEE transactions on visualization and computer graphics*, 30(1):1085–1094, Jan. 2024. doi: 10.1109/TVCG.2023.3326526 10

[33] R. G. C. Maack, J. Lukasczyk, J. Tierny, H. Hagen, R. Maciejewski, and C. Garth. Parallel Computation of Piecewise Linear Morse-Smale Segmentations. *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–14, 2023. doi: 10.1109/TVCG.2023.3261981 1, 2, 3, 4

[34] F. Manne and M. M. A. Patwary. A Scalable Parallel Union-Find Algorithm for Distributed Memory Computers. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, eds., *Parallel Processing and Applied Mathematics*, pp. 186–195. Springer, Berlin, Heidelberg, 2010. doi: 10.1007/978-3-642-14390-8_20 2

[35] T. B. Masood, J. Budin, M. Falk, G. Favelier, C. Garth, C. Gueunet, P. Guillou, L. Hofmann, P. Hristov, A. Kamakshidasan, C. Kappe, P. Klacansky, P. Laurin, J. A. Levine, J. Lukasczyk, D. Sakurai, M. Soler, P. Steneteg, J. Tierny, W. Usher, J. Vidal, and M. Wozniak. An Overview of the Topology ToolKit. In I. Hotz, T. Bin Masood, F. Sadlo, and J. Tierny, eds., *Topological Methods in Data Analysis and Visualization VI*, pp. 327–342. Springer International Publishing, Cham, 2021. Series Title: Mathematics and Visualization. doi: 10.1007/978-3-030-83500-2_16 3

[36] F. Nauleau, F. Vivodtzev, T. Bridel-Bertomeu, H. Beaugendre, and J. Tierny. Topological Analysis of Ensembles of Hydrodynamic Turbulent Flows An Experimental Study. In *2022 IEEE 12th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 1–11. IEEE, Oklahoma City, OK, USA, Oct. 2022. doi: 10.1109/LDAV57265.2022.9966403 1

[37] E. Nilsson, J. Lukasczyk, W. Engelke, T. B. Masood, G. Svensson, R. Caballero, C. Garth, and I. Hotz. Exploring Cyclone Evolution with Hierarchical Features. In *2022 Topological Data Analysis and Visualization (TopoInVis)*, pp. 92–102. IEEE, Oklahoma City, OK, USA, Oct. 2022. doi: 10.1109/TopoInVis57755.2022.00016 1

[38] E. Nilsson, J. Lukasczyk, T. B. Masood, C. Garth, and I. Hotz. Probabilistic Gradient-Based Extrema Tracking. In *2023 Topological Data Analysis and Visualization (TopoInVis)*, pp. 72–81, Oct. 2023. doi: 10.1109/TopoInVis60193.2023.00014 1

[39] K. Perlin. An image synthesizer. *ACM SIGGRAPH Computer Graphics*, 19(3):287–296, July 1985. doi: 10.1145/325165.325247 6

[40] V. Robins, P. J. Wood, and A. P. Sheppard. Theory and Algorithms for Constructing Discrete Morse Complexes from Grayscale Digital Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(8):1646–1658, Aug. 2011. doi: 10.1109/TPAMI.2011.95 1

[41] R. Seidel and M. Sharir. Top-Down Analysis of Path Compression. *SIAM Journal on Computing*, 34(3):515–525, Jan. 2005. doi: 10.1137/S0097539703439088 3

[42] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, Mar. 1982. doi: 10.1016/0196-6774(82)90008-6 2

[43] N. Shivashankar, S. M, and V. Natarajan. Parallel Computation of 2D Morse-Smale Complexes. *IEEE Transactions on Visualization and Computer Graphics*, 18(10):1757–1770, Oct. 2012. Conference Name: IEEE Transactions on Visualization and Computer Graphics. doi: 10.1109/TVCG.2011.284 2

[44] N. Shivashankar and V. Natarajan. Parallel Computation of 3D Morse-Smale Complexes. *Computer Graphics Forum*, 31(3pt1):965–974, 2012. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2012.03089.x. doi: 10.1111/j.1467-8659.2012.03089.x 2

[45] N. Shivashankar, P. Pranav, V. Natarajan, R. V. D. Weygaert, E. P. Bos, and S. Rieder. Felix: A Topology Based Framework for Visual Exploration of Cosmic Filaments. *IEEE Transactions on Visualization and Computer Graphics*, 22(6):1745–1759, June 2016. doi: 10.1109/TVCG.2015.2452919 1

[46] J. Shun, L. Dhulipala, and G. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, SPAA '14, pp. 143–153. Association for Computing Machinery, New York, NY, USA, June 2014. doi: 10.1145/2612669.2612692 2

[47] S. Smale. Generalized Poincare's Conjecture in Dimensions Greater Than Four. *Annals of Mathematics*, 74(2):391–406, 1961. Publisher: Annals of Mathematics. doi: 10.2307/1970239 1

[48] S. Smale. On Gradient Dynamical Systems. *Annals of Mathematics*, 74(1):199–206, 1961. Publisher: Annals of Mathematics. doi: 10.2307/1970311 1

[49] T. Sousbie. The persistent cosmic web and its filamentary structure - I. Theory and implementation: Persistent cosmic web - I: Theory and implementation. *Monthly Notices of the Royal Astronomical Society*, 414(1):350–383, June 2011. doi: 10.1111/j.1365-2966.2011.18394.x 1

[50] V. Subhash, K. Pandey, and V. Natarajan. GPU Parallel Computation of Morse-Smale Complexes. In *2020 IEEE Visualization Conference (VIS)*, pp. 36–40. IEEE, Salt Lake City, UT, USA, Oct. 2020. doi: 10.1109/VIS47514.2020.00014 2

[51] J. Tierny, G. Favelier, J. A. Levine, C. Gueunet, and M. Michaux. The Topology ToolKit. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):832–842, Jan. 2018. doi: 10.1109/TVCG.2017.2743938 3, 6

[52] TTK Contributers. TTK data examples. `https://github.com/topology-tool-kit/ttk-data`. [Accessed 18-Jun-2024]. 2

[53] J. Xu, H. Guo, H.-W. Shen, M. Raj, X. Wang, X. Xu, Z. Wang, and T. Peterka. Asynchronous and Load-Balanced Union-Find for Distributed and Parallel Scientific Data Visualization and Analysis. *IEEE Transactions on Visualization and Computer Graphics*, 27(6):2808–2820, June 2021. Conference Name: IEEE Transactions on Visualization and Computer Graphics. doi: 10.1109/TVCG.2021.3074584 2

[54] L. Yan, T. B. Masood, R. Sridharamurthy, F. Rasheed, V. Natarajan, I. Hotz, and B. Wang. Scalar Field Comparison with Topological Descriptors: Properties and Applications for Scientific Visualization. *Computer Graphics Forum*, 40(3):599–633, June 2021. doi: 10.1111/cgf.14331 1