To appear in: Proceedings of the 9th IEEE Symposium on Large Data Analysis and Visualization (LDAV) 2019.

Distributed Percolation Analysis for Turbulent Flows

Anke Friederici[1,*] Wiebke Köpp[1,*]

Marco Atzori[2] Ricardo Vinuesa[2] Philipp Schlatter[2] Tino Weinkauf[1]

[1]Division of Computational Science and Technology, KTH Royal Institute of Technology, Stockholm
[2]Linné FLOW Centre, KTH Mechanics, Stockholm

Abstract

Percolation analysis is a valuable tool to study the statistical properties of turbulent flows. It is based on computing the percolation function for a derived scalar field, thereby quantifying the relative volume of the largest connected component in a superlevel set for a decreasing threshold. We propose a novel memory-distributed parallel algorithm to finely sample the percolation function. It is based on a parallel version of the union-find algorithm interleaved with a global synchronization step for each threshold sample. The efficiency of this algorithm stems from the fact that operations in-between threshold samples can be freely reordered, are mostly local and thus require no inter-process communication. Our algorithm is significantly faster than previous algorithms for this purpose, and is neither constrained by memory size nor number of compute nodes compared to the conceptually related algorithm for extracting augmented merge trees. This makes percolation analysis much more accessible in a large range of scenarios. We explore the scaling of our algorithm for different data sizes, number of samples and number of MPI processes. We demonstrate the utility of percolation analysis using large turbulent flow data sets.

Index Terms: Computing methodologies—Distributed computing methodologies—Distributed algorithms; Mathematics of computing—Discrete mathematics—Graph theory—Paths and connectivity problems

1 Introduction

Turbulent flows play a crucial role in many domains: fuel efficiency and maneuverability of airplanes, trucks, and cars depend directly on the understanding of turbulence during design to avoid detrimental flow characteristics and exploit beneficial ones. Turbulence appears not only on the outside of vehicles, but also in engine components or oil pipelines. Roughly half of the energy spent worldwide in transporting people and goods is dissipated by the turbulent motion in the immediate vicinity of the moving object, e.g., near the surface of a plane. A better understanding of turbulent flows can reduce this energy consumption and help make our life more sustainable.

Despite a century of research, a comprehensive characterization of turbulent flows is still an open and actively researched matter which is primarily addressed through *direct numerical simulations* (DNS) of the Navier-Stokes equations. It is the computationally most intensive form of flow simulation. For decades to come, turbulence research will exhaust the computational power (time and memory) of the largest supercomputers to perform more detailed simulations and get a better understanding of the underlying physical phenomena.

Percolation theory studies random connectivity in a graph or lattice using statistics. It has been initiated in 1957 by Broadbent and Hammersley [6] and is widely used today to characterize complex random systems. Its application in turbulence research has been initialized by Moisy and Jiménez [29]. Percolation analysis is used to understand intense Reynolds stress or vorticity events in turbulent flows, and to find suitable scalar indicators and corresponding thresholds for further analyses of coherent structures (Section 2).

Analyzing turbulent flows using percolation means to observe the connected components of the super-level sets in a derived scalar field for varying thresholds. Previous work [29] used a flood-fill like algorithm with a runtime of $O(nk)$, where n is the number of grid vertices and k the number of tested thresholds. While such an algorithm can easily be parallelized and distributed, it is inherently slow. Several publications reported on being restricted by the long computation times [10, 25] and resorting to remedies such as discarding parts of the data.

Recently, Friederici et al. [13] presented an algorithm for percolation analysis exploiting the *union-find* data structure with a runtime of $O(n \log n)$. While this algorithm is much faster than the previous work, it is a global algorithm and cannot easily be parallelized and distributed. Since most turbulent flow data is too large to be hosted on a single compute node, its percolation analysis has to run on distributed memory as well.

We present the first algorithm for percolation analysis of turbulent flows that works for out-of-core data sizes (Section 3). Our algorithm works in parallel and on distributed memory, so that even the largest data sets can be analyzed using it. Similar to Friederici et al. [13], our algorithm builds upon the *union-find* data structure, but we keep inter-process communication to a minimum by observing that many operations in the serial algorithm [13] can be executed out-of-order and synchronization is only necessary a small number of times.
We give the following contributions:

- We present a parallel and distributed algorithm for percolation analysis of turbulent flows.[1] To the best of our knowledge, it is the fastest one available today (Section 3).
- We provide a scaling analysis of our algorithm for data sets with different sizes (Section 4).
- We provide a comparison against related methods from topological data analysis (Section 4).
- We showcase the utility of percolation analysis for analyzing turbulent flows using different large data sets (Section 5).

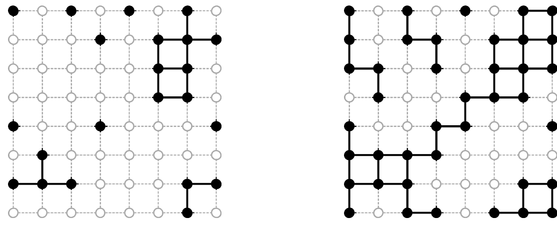
2 Related Work and Background

2.1 Percolation Theory

Percolation is a mathematical concept best explained by considering a porous stone put into water: What is the probability for the water to reach the center of the stone? This depends on how porous the stone's material is. One can model this using a stochastic approach proposed by Broadbent and Hammersley [6] that describes how the random properties of a medium (the stone in our example) influence the percolation of a fluid through it: consider an infinite 2D lattice **L**,

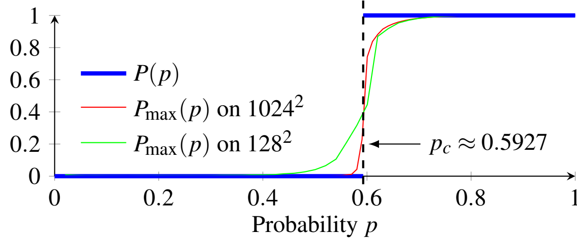
*Both authors contributed equally to this work.
e-mail:{ankef | wiebkek}@kth.se

[1]Source code: `https://github.com/KTHVisualization/percMPI`.



(a) Lattice with open subgraph for $p < p_c$.

(b) Lattice with open subgraph for $p > p_c$.



(c) Probability $P(p)$ of a percolating cluster for an infinite lattice (blue) and percolation function $P_{\max}(p)$ for a 1024^2 lattice (red), and a 128^2 lattice (green).

Figure 1: We consider the vertices in an infinite lattice to be *open* (●) with probability p or *closed* (○) otherwise. The probability $P(p)$ for the existence of an infinite *open* connected component is zero for $p < p_c$ and one for $p > p_c$, shown as the blue graph. The effect is less pronounced for real data and depends on the data size, but can still be observed clearly. See (1) for the definition of $P_{\max}$.

where we define for each of its vertices to be *open* with probability p, and *closed* otherwise. See Figure 1 for an illustration. Open vertices allow water to pass through, closed vertices do not. Hence, the parameter $0 \leq p \leq 1$ mediates the portion of open and closed vertices; high values of p correspond to a very porous stone, while small values of p correspond to impermeable material.

Let us concentrate on the *open* subgraph $\mathbf{L}'$ made from only the *open* vertices and their adjacent edges. We are interested in how the structure of $\mathbf{L}'$ depends on the value of p. For small values of p, this subgraph consists of many small connected components as illustrated in Figure 1a, each of them finite in size (remember that $\mathbf{L}$ and $\mathbf{L}'$ themselves are infinite). Strikingly, there is a *critical value* p_c for which large-scale structures form and the open subgraph contains an infinite connected component pervading the entire domain, called the *percolating cluster*. Figure 1b illustrates this as the dark black component extending through the entire lattice. One of the most intriguing aspects of percolation is the fact that this transition is sharp: for $p < p_c$, we see finite connected components in the open subgraph $\mathbf{L}'$, whereas the picture immediately changes for $p > p_c$, for which we see an infinite percolating cluster in $\mathbf{L}'$. The critical value p_c is often referred to as *percolation threshold*.

Another striking result of percolation theory is that the percolation threshold p_c depends *solely* on the lattice connectivity. In that regard, one first has to distinguish between *site* and *bond* percolation, which refers to considering either vertices or edges as open/closed, respectively. With this in mind, our above example of the 2D lattice with 4-connectivity (infinite uniform grid) has a site percolation threshold $p_c \approx 0.5927$ [12] and a bond percolation threshold $p_c = 1/2$ [17, 20]. Many other lattice types have been researched in the mathematics community [44]. While site and bond percolation are related, we concentrate on site percolation for the rest of the paper.

In order to apply percolation theory to real data, we turn to *level set* percolation [2, 35]: considering (seemingly) random scalar data values at the vertices of a finite lattice, we can observe the connected components of the superlevel sets for varying thresholds. The literature proposes different methods for determining the existence of the percolating cluster in this scenario, but it boils down to graphing a derived function, the *percolation function*, based on the volumes of the connected components. In Figure 1c, we applied a particular percolation function ($P_{\max}$, introduced in the next section) to random data sampled on either a 1024^2 or 128^2 lattice. As can be seen, the resolution of the lattice affects the steepness of the curve around the percolation threshold, but the transition is still clearly visible.

Percolation analysis finds its application in many domains such as cosmology, geology, material science and epidemiology [37]. We focus on the fluid dynamics domain as described in the next section.

2.2 Percolation Analysis for Turbulent Flows

Turbulence is not randomness. Turbulent flows are well-defined by the Navier-Stokes equations, some parameters such as the Reynolds number (loosely speaking, the mean speed of the flow) and the viscosity of the fluid, as well as the shape of the geometry around which the fluid flows. Yet, turbulent flows form many fine-grained structures, often throughout the entire domain, that are easily mistaken as random. They constitute the fundamental building blocks of turbulent flows and play a central role in the transport of mass and momentum. The study of their statistical properties with regards to frequency, intensity, spatial distribution and temporal evolution contributes to a better understanding of turbulent flows.

These structures of interest can be defined as regions where the flow exhibits a particular behavior over some amount of time. One speaks about spatial and temporal coherence meaning that this flow behavior can be observed in a continuous spatio-temporal region. This region is then called a *coherent structure*. Flow behaviors of interest include high vorticity magnitude [29], back-flow events near the wall [10], or high Reynolds stresses in channel [25], duct [5], boundary-layer [27], and shear-induced [11] flows.

While the specific definition of coherent structures varies with the flow case and the particular research topic, their technical characterization often comes down to being connected components in a scalar field $f(\mathbf{x},t)$ for a well-chosen threshold h. This means, we are looking at the superlevel set of the scalar field defined as the set of voxels fulfilling $f(\mathbf{x},t) \geq h$. Each of its connected components is then considered a coherent structure.

Not all turbulent flows are homogenous. For example, wall-bounded flows exhibit varying velocity magnitudes depending on the distance to the wall. The identification of coherent structures needs to account for the non-homogeneity, so that all structures can be dealt with on equal footing. This means to normalize the considered scalar field $f(\mathbf{x},t)$: different normalization strategies are employed, often including time averaging and disregarding parts of the domain near the wall. We describe a normalization procedure for a duct flow in Section 5. For now, it suffices to understand that a normalization process ideally yields an altered scalar field $f(\mathbf{x},t)$ in which we can assume to find comparable coherent structures at the same threshold. This leads to two questions:

- **Homogeneity:** Are the coherent structures distributed homogeneously in the domain as expected after normalization, i.e., was the normalization successful?
- **Choice of threshold:** Which threshold h is a suitable choice for further analysis such that the extracted structures are intense, numerous, and small enough to be meaningful?

Moisy and Jiménez [29] pioneered the use of level set percolation to answer these questions for isotropic turbulent flow: if $f(\mathbf{x},t)$ has been properly normalized, we can expect the typical steep phase transition around the percolation threshold as depicted in Figure 1c.

The corresponding percolation function $P_{\max}$ is defined as

$$P_{\max}(h) = \frac{V_{\max}}{V_{\text{total}}}, \tag{1}$$

where V_{total} denotes the total volume of the superlevel set for a given threshold h, and $V_{\max}$ is the volume of its largest connected component. The steep phase transition will only appear in this function if the coherent structures are distributed homogeneously in the domain. Furthermore, a suitable threshold for further analysis is then chosen relative to the percolation threshold H_c, which can be approximated from the percolation function $P_{\max}$ as the point of its highest slope, i.e.,

$$H_c = \text{argmax}_h \frac{\partial P_{\max}}{\partial h} . \quad (2)$$

This requires a dense sampling of $P_{\max}$, something that previous work struggled with due to the involved computational costs.

Since its inception due to Moisy and Jiménez [29], multiple studies have applied percolation analysis for different kinds of fully developed turbulent flows [5, 10, 11, 25, 27]. It is interesting to note that percolation analysis has also been applied in fluid dynamics in the context of the transition from the laminar to the turbulent regime [3]. In this paper, however, we focus on a fully turbulent regime. We perform percolation analyses for selected quantities of interest for isotropic and wall-bounded flows in Section 5.

2.3 Serial Algorithms for Percolation Analysis

Computing the percolation function (1) means to identify the total volume of the superlevel set $f(\mathbf{x},t) \geq h$ and the volume of its largest connected component. This has to be done for a reasonable number k of threshold samples $\{h_1, \ldots, h_i, \ldots, h_k\}$ such that we obtain a well-resolved graph for the percolation function (cf. Figure 1c). In practical settings, one strives for $10^2 < k < 10^4$.

Percolation analysis for turbulent flows [29] was introduced using a straightforward algorithm to be repeated for each threshold h_i: first, all vertices are marked as *open* if $f \leq h_i$, and *closed* otherwise. Second, we visit all vertices and detect all *open* connected components using a flood-fill algorithm. Then we can easily determine the total volume of the superlevel set and the volume of the largest connected component. This procedure has a runtime complexity of $O(nk)$, where n is the number of grid vertices and k is the number of thresholds h_i. This leads to long computation times. In fact, several publications [10, 25] report severe problems with this aspect to the extent that some of the data had been discarded in order to maintain reasonable computation costs.

Hoshen and Kopelman [19] pioneered the use of the *union-find* data structure for percolation analysis. Given a graph with a set of *open* and *closed* bonds or sites their algorithm determines the existence of a percolating cluster. This corresponds to a single threshold h in our setting. An iterative variant for efficient simulation of random valued lattices was introduced by Newman and Ziff [32].

Both algorithms do not consider level set percolation, for which Friederici et al. [13] recently presented an iterative algorithm based on the union-find data structure. It has a runtime complexity of $O(n \log n)$ and the measured computation times for practical cases decreased by an order of magnitude [13].

From now on, let us refer to this algorithm as the *union-find algorithm.* It starts by constructing a list of vertices in descending order of the data values. Hence, the vertex with the global maximum appears first in the sorted list, the global minimum last. Next, we initialize a union-find data structure `UF` with path compression [39]: it has a slot for each vertex where we can store a pointer to another vertex. It allows us to identify connected components by following the pointers until we find an element that points to itself: this is the *representative* of the connected component. We will now iterate over the sorted list starting with the highest data value: for each vertex $\mathbf{v}$, we identify its neighbors in the underlying grid and count the number of connected components that have already been recorded in `UF` for these neighbors. The following cases occur (see Figure 2 for an example):

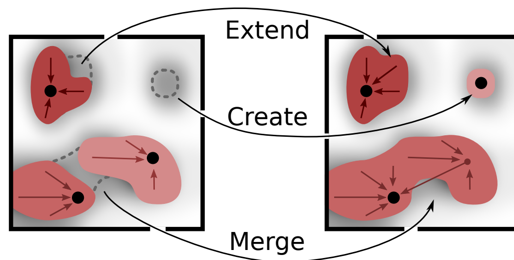


Figure 2: We can identify connected components of the superlevel set of a scalar field using a union-find data structure, which lets each vertex point (indirectly) to the representative of a component (black dots). While decreasing the threshold, we incorporate new vertices into the superlevel set, which leads to *creating* new components as well as *extending* or *merging* existing ones.

- **Create:** No neighbor has been recorded in `UF` yet. We create a new connected component in `UF` by letting the current vertex $\mathbf{v}$ point to itself. It is the representative of the component. Note also that $\mathbf{v}$ has to be a local maximum.
- **Extend:** One or several neighbors have been recorded in `UF` and belong to the same component. We extend this component with the current vertex $\mathbf{v}$ by letting it point to the component representative.
- **Merge:** Two or more neighbors have been recorded in `UF` and they belong to more than one component. Out of all their representatives we decide on any one of them and let all other representatives point to it. Thus, we merged the components. Furthermore, the current vertex $\mathbf{v}$ will also point to that representative.

To facilitate percolation analysis, we record statistics about the volumes of the connected components. In fact, these statistics are updated with each *create*, *extend*, and *merge* operation and just need to be stored when reaching a threshold sample h_i.

This procedure is very common in topological data analysis and finds its use in the computation of merge and contour trees [8], the Morse-Smale complex [9, 24, 42], or watershedding [36]. Merge trees are particularly interesting in this regard: they record the topology of the superlevel (or sublevel) sets of a scalar field, and the augmented version of a merge tree can be used to identify all connected components and their volume for any given threshold. Hence, one could postprocess an augmented merge tree for the purpose of percolation analysis. We will therefore compare our work against a parallel algorithm for merge tree computation, see the next section and Section 4.

2.4 Parallelization and Distribution

Regarding parallel computing, one has to distinguish between two different modes: *shared* memory and *distributed* memory.

Executing an algorithm in parallel with *shared* memory on a single computer allows fully parallel reading of the data, but requires some care when writing data to maintain consistency. This approach is also limited to the memory available on a single computer.

On the other hand, executing an algorithm in parallel with *distributed* memory means to segment the data across several parallel processes with independent memory This often includes the introduction of additional structures - called *ghost cells* [34] - at the segmentation boundaries. The processes run independently from each other and are often distributed to different compute nodes in a cluster. For large datasets that exceed the memory capacity of a single node, distribution is often the only possible approach. Depending on the algorithm, the processes may need to share some of their results with each other through an inter-process communication

Table 1: Overview of previously introduced algorithms for computing percolation functions as well as related algorithms based on the union-find data structure. Note that for augmented merge and contour trees [15, 16], obtaining the actual percolation function samples requires an additional post-processing step.

Algorithm	Multiple Samples	Parallel	Distributed
Algorithms for percolation analysis			
[19]			
[13,32]	✓		
[40,41]		✓	✓
Algorithms for union-find with graphs			
[18,28]		✓	✓
Algorithms for extracting merge and contour trees			
[1,7,26,33]		✓	
[15,16]	✓	✓	
[23,30,31]		✓	✓
Ours	✓	✓	✓

protocol such as MPI [14]. Our algorithm in this paper falls into this category.

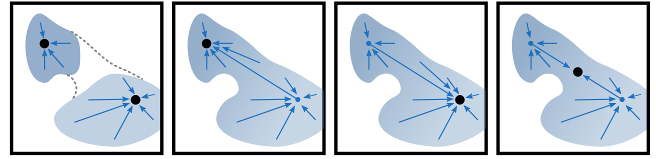
A number of algorithms in the realm of percolation analysis and related union-find based algorithms have been adapted to facilitate shared- and distributed memory parallelism. However, none of them readily support our setting of obtaining multiple percolation function samples with distributed memory, see Table 1.

Teuler and Gimel [40] as well as Tiggemann [41] each developed a parallel and distributed version of the Hoshen-Kopelman algorithm [19]. Both allow to obtain a single percolation function sample and do not pertain to level set percolation. Similarly, distributed versions of the union-find algorithm have been proposed [18, 28], but are tailored for unstructured graphs and cannot be updated from an existing threshold to another.

Several algorithms have been proposed to compute merge trees in parallel on a shared memory machine. Pascucci and Cole-McLaughlin [33] describe two parallel algorithms for computing contour trees. Acharya and Natarajan [1] propose a particularly memory efficient algorithm for computing contour trees that merges the intermediate results from different regions of the domain into one global result. This method has been designed for shared memory architectures, but could potentially also be adapted to a distributed environment. Carr et al. [7] propose a data-parallel algorithm for computing contour trees explicitly targeting SIMD architectures. Gueunet et al. [15, 16] exploit modern scheduling techniques for high-performance computing by providing task-based formulations for the merge and contour tree algorithms.

Some of the above methods are able to extract an *augmented* merge tree, which carries enough information to compute the percolation function. We will therefore compare our approach to the augmented merge tree computation method of Gueunet et al. [16]. The authors made an effort to provide code with their publication, so we are able to compare the results on the same machine. We detail this in Section 4.4.

A number of algorithms have been proposed to compute merge or contour trees in a distributed environment. Morozov and Weber compute separate simplified merge trees that are then merged pairwise [30, 31]. Landge et al. [23] propose two different in-situ algorithms for computing merge trees that work directly on the memory layout of the simulation. These algorithms extract unaugmented trees only or require an additional accumulation phase and can therefore not be used to compute the percolation function.



(a) Before *merge*. (b) Merge onto left representative. (c) Merge onto right representative. (d) Merge onto new representative.

Figure 3: We can choose different valid representatives after *merging* two components as long as the newly merged connected component is correctly identified, i.e., all paths lead to the new representative.

3 Parallel and Distributed Percolation Analysis

We introduce the first *distributed* algorithm for computing the percolation function. It is required since many turbulent flow data sets are too large to fit into the memory of a single compute node. We distribute the data across several MPI processes running on several compute nodes. The challenges with parallelizing the union-find algorithm in a memory-distributed setting are two-fold:

- The algorithm requires a list of cells sorted in descending order of the data values. It is not straightforward to sort a list when the data is distributed.
- The connected components grow with decreasing threshold across the boundaries of a single process. Synchronization between different processes is required to maintain consistency.

As discussed in the previous section, these challenges have been tackled in one way or another for similar algorithms such as computing merge or contour trees [23, 30, 31]. However, none of the existing methods allows us to compute the percolation function for memory-distributed data. Furthermore, the computation of the percolation function comes with its specific properties (Section 3.1) that we can exploit to design efficient data structures (Section 3.2) and a fast algorithm (Section 3.3).

3.1 Properties of Union-Find Algorithm for Percolation

We make the following observations that aid us in parallelizing the union-find algorithm of Friederici et al. [13] for distributed memory:

Low number of thresholds We want to sample the percolation function with k threshold samples $[h_1, \dots, h_i, \dots, h_k]$. This number is much smaller than the total number of cells n in our lattice. Hence, we can add several cells to the union-find data structure without evaluating the volume statistics of the connected components. We will exploit this to minimize global synchronization.

Arbitrary representatives A union-find data structure allows us to follow a path of pointers from each cell to the representative cell of the connected component. Many algorithms ascribe an inherent meaning to the exact position of that representative, e.g., the position marks the minimum/maximum in merge tree computations. However, we do not require this for computing the volume statistics of the connected components. We are free to decide on the direction of merges and to move the representative by redirecting a small number of pointers, as long as we ensure that the components can be correctly identified. This is illustrated in Figure 3.

Re-ordering Since we query the volume statistics only at the threshold samples h_i, the kind and order of union-find operations between thresholds is of no particular interest, as long as the final result remains the same. Figure 4 illustrates this, where two *extend* operations lead to the same final connected component as a *create* with a subsequent *merge*. As exactly the same cells will be added, the final connected component is the same. This also means we do not need to iterate over all cells in data-value order, but rather only

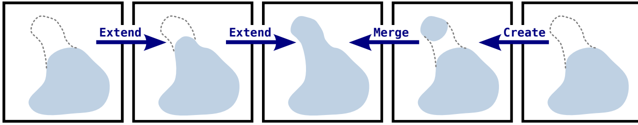


Figure 4: Processing two cells in different order. In one case (left to right), the component is *extended* twice. The other way around (right to left), a new component is *created* and immediately *merged*. Both result in the exact same connected components.

make sure to have processed all cells with data values smaller than h_i (and only those cells) before querying the volume statistics for that threshold. We will exploit this to increase parallelism.

Prevalence of *extend* operations The least expensive operation is to *extend* an existing component: while the *create* and *merge* operations need to update the list of connected components, *extending* only adds to an already existing component and its volume statistics. An analysis with the sequential union-find algorithm on the duct data set (see Section 5) has shown that approximately 99.6% of all operations were *extends*, with only a tiny fraction of *creates* (0.21%) and *merges* (0.19%). We can exploit this in the design of the distributed algorithm to balance the costs of the three operations.

Locality To process a cell in the union-find algorithm, we only need access to its neighbors and their information. More precisely, we need to be able to identify the representatives of the connected components of the neighbors.

Contrast to other algorithms Some of the above properties are unique to the percolation algorithm and in stark contrast to other algorithms making use of the union-find data structure. For example, merge tree algorithms record the topology changes of the superlevel sets, i.e. all *creates* and *merges*, and can therefore not afford to re-order their union-find operations.

3.2 Data Structure

Our algorithm follows the same structure as the non-distributed version. For a set of values h_i, every cell with value $c \geq h_i$ indicates which connected component they belong to within a union-find (`UF`) data structure. To that end, every cell with scalar value $h_i \leq c < h_{i-1}$ is added into `UF` by either a *create*, *extend* or *merge* operation. Then, the necessary statistical values are recorded, namely the volume of the largest connected component and total superlevel set.

Our algorithm executes as follows for each value range $[h_i, h_{i-1})$: first, every process updates the union-find data structures of a partial domain. Small slices of these union-find cells and lists of connected components are sent to a unique global process. In a sequential step, this global process updates the few remaining cells and resolves all *merge* operations that happen across processes. Finally, this aggregated data is distributed back to all other processes. These four steps will be described in detail in Section 3.3.

3.2.1 Spatial Distribution

While our algorithm is applicable to different lattices, we will focus the description on three-dimensional curvilinear grids with 6-neighborhood. The basic idea to facilitate distributed processing is to partition the domain across several processes. Each process contains all data necessary to perform union-find operations: the scalar data ordered by value and a union-find data structure `UF`.

Each cell update entails gathering the connected component identifiers of all neighboring cells to enable *creating*, *extending* and *merging* union-find operations. Updating the outer cell layer on a processor thus requires the `UF` data from a different processor. This `UF` data is transferred in the form of *ghost cells* - small `UF` layers that are not updated on the receiving process. Ghost cells are a common concept for interfacing between processes in distributed computation. Note that in our application, special care has to be taken that all ghost cells pointers can be traced. In general, ghost cells can only point to other ghost cells, or the receiving process cannot match the cells to their respective connected components.

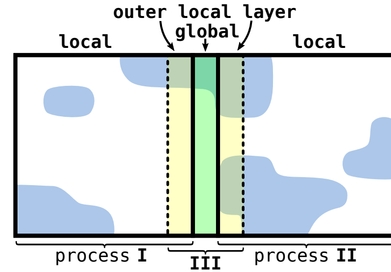
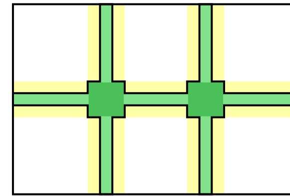
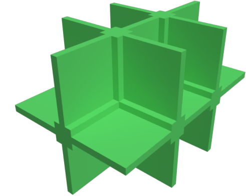


Figure 5: Basic layout of our memory distribution. *Local cells* reside on local processes. *Global cells* (green) are held by the unique global process. Both *global cells* and the outer cell layers (yellow) of local processes will be sent between processes as *ghost cells*.



(a) 2×3, outlines separate processes



(b) Global cells of a $2 \times 2 \times 3$ grid

Figure 6: The spatial distribution in a 2D and 3D domain. A skeleton of global cells (green) requires some neighboring cells as ghost cells (yellow). The local cells (white) take up the most part.

Since each union-find step requires full information of its neighbors, we cannot update two neighboring cells in parallel. A simple example where the attempt would fail is two neighboring cells concurrently creating a new connected component next to each other.

To circumvent this problem, we introduce a *global process*. It contains layers of cells between each set of processor blocks. A simple case with two common *local processes* is shown in Figure 5. While the local processes update their cells, they have access to the global cells as ghost cells. In a subsequent step, the global cells are updated by the global process, given their neighbors as ghost cells. This way, no neighboring cells are ever written to simultaneously.

In general, more than two local processes are employed concurrently, with global cells in between in form of a skeleton structure. The resulting block distribution is shown in Figure 6 for a 2D and 3D field. Local processes contain the majority of cells. To optimize efficiency, their volume should be compact and large in comparison to their surface to keep the number of ghost cells requiring communication small.

3.2.2 Connected Component Lists

The ultimate goal of the percolation algorithm is to gather statistics of connected components. As these grow, they will lay partially within several processes - see for example Figure 5, where the top connected component is distributed between all 3 processes present. Such a connected component is called a *global component*.

Several processes hold partial volumes about these components. Additionally, merges between several such components are rather complicated and need to be resolved consistently. In order to communicate merges and collect volumes, we need to re-think the way a cell is linked to a connected component.

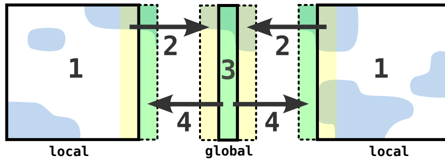
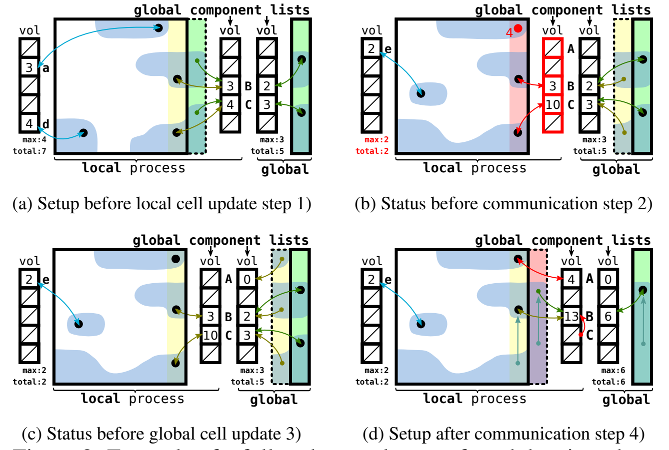


Figure 7: The four basic phases of our algorithm:
1) update the local processes in parallel
2) send data to the global process
3) update the global process and collect statistics
4) distribute data to the local processes
Thick lines represent the scalar data per process, while dotted lines mark ghost cells. The green and yellow regions respectively overlap.



(a) Setup before local cell update step 1) (b) Status before communication step 2)

(c) Status before global cell update 3) (d) Setup after communication step 4)

Figure 8: Example of a full update cycle, transferred data in red.
(a) Before step 1), local components **a** and **d** and global components **B** and **C** are known. The value $P_{\max} = 4/19$ was just recorded.
(b) In step 1, a new local component, **e**, is *created*. Furthermore, the local component **a** is *extended* onto the boundary and converted into a global component, yet without identifier. Components **d** and **C** *merge*, which is resolved locally by removing **d** and *extending* **C**. The data marked in red is transferred to the global process in step 2): the maximum and total volume of the local components, the newly created global component representative and volume, the outer cell layer as ghost cells and the local copy of the global component list.
(c) Before updating the global cells, the global component **A** is created from former local component **a**.
(d) In the global cell update step 3, adding a global cell *merges* $\mathbf{C} \to \mathbf{B}$. Component **C** is deleted from the global component list, the volumes are added and the former representative of **C** is pointed towards the representative of **B**. At this point, $P_{\max} = 13/25$ is recorded. The *merge* $\mathbf{C} \to \mathbf{B}$ and the *creation* of **A** (red) are communicated to the local process, which mirrors the *merge* changes.

To this end, we introduce *component lists*. A representative of a connected component in `UF` will point into this list by means of a unique component identifier, corresponding to the list index. By storing the volume of the (partial) region in the list, it now contains all statistically relevant information, namely the number and volume of all connected components. To avoid a gathering step, the largest and total volume of all components are stored additionally per list and are expected to correctly correspond to the list's content.

All local processes keep their own connected component list. In addition, each process holds a copy of a *global component list*. It holds all global components, with their partial volume on the respective process. When a global component is shared between processes, the partial regions within each process have their own representative, each using the same component identifier. See for example Figure 8, where both the local and global process point into a copy of the global component list.

Union-find operations can call for one of three different actions on a component list: *create* a component **A**, *extend* a component **A** or *merge* one component **A** onto a second component **B**, noted as $\mathbf{A} \to \mathbf{B}$. Each operation will call for a change in volume of one component, easily achieved by adapting the respective value. *Creating* or *merging* operations will in addition add or remove connected components from the list, respectively. To prevent ambiguity, only the global process is ever creating or merging global components.

3.3 Algorithm

After loading and sorting the scalar data per process, we repeat these steps (see also Figure 7) for each value range $(h_i, h_{i-1}], i = 1, 2, \ldots k$:
1) All local processes update by adding all cells with values in $[h_{i-1}, h_i)$ into `UF` . Merges between global components are recorded.
2) Each local process sends the outer ghost cell layer, a global component list copy, global merges and statistics to the global process.
3) The global process updates the global cells. All global component merges are resolved. The number of components as well as the largest and total volume are recorded.
4) The resolved merges are broadcast to all local processes. Repeat from 1) for the next value range $i \leftarrow i+1$.

We provide an example of these steps in Figure 8 as an overview of the data structures involved and several union-find operations.

Our source code[1] is provided for further detail and reproducibility as a FreeBSD licensed, standalone C++ application, including all settings for the experiments shown. It compiles out-of-the-box and supports random generation, raw data as well as normalization.

1) Local Process Update

The main purpose of this step is to update the `UF` data structure of local processes by adding all cells in the current value range $[h_i, h_{i-1})$. As mentioned, we distinguish local and global components.

Local components lay fully within the local process and are very easy to handle, see for example components **a** and **d** in Figure 8a. In fact, we can apply all union-find operations in exactly the same way as the sequential union-find algorithm described by Friederici et al. [13] when we only encounter local components as neighbors: When no neighboring cell points at a connected component, we *create* a new one by adding an element to the local component list and pointing at it. If the current cell neighbors exactly one connected component, it gets pointed to that component's representative in an *extend*. In the case of several neighboring connected components, we need to perform a *merge* operation. All *merges* of more than two connected components can be split up into several two-component-merges, so we will from here on only cover those. To merge two local components $\mathbf{A} \to \mathbf{B}$, the representative of component **A** gets repointered to the representative of **B**, effectively transferring all cells from **A** to **B**. This increased connected component size is expressed by adding the volume of **A** to **B**. Finally, the component **A** is deleted from the component list. Note that all these operations will also add to the volume of the connected component ultimately pointed to. We also update the total volume and potentially the largest volume we keep for the local components.

Global components contain at least one cell of the outer cell layer or a ghost cell. They are recorded in the global component list. Only the global process is allowed to *create* new global components. Local processes record all *creations* of global components and are assigned component identifiers by the global process in step 4).

Similarly, *merging* two global components removes one of them from the global component list. Again, *merges* will be coordinated from the global process: say that we encounter the merges $\mathbf{A} \to \mathbf{B}$ and $\mathbf{A} \to \mathbf{C}$ on two different local processes. If both were to apply their respective *merge* operation, we would end up with the components **B** and **C**, instead of one large component. Instead, the global

process will later tell both processes to perform the merges $\mathbf{A} \to \mathbf{C}$ and $\mathbf{B} \to \mathbf{C}$, resulting in one large component $\mathbf{C}$.

Extend operations are performed as usual, with the additional constraint that extending into the outer cell layer transforms the local component into a global component. The local component is removed and a *create* operation is recorded. It can be observed over all steps in Figure 8 in the transformation of **a** into **A**.

All in all, we need to adhere to the following rules when applying the union-find algorithm with neighboring global components: 1) global component representatives must lay on the outer layer to be visible to the global process, 2) *create* and *merge* operations will be recorded but not applied directly.

2) Local to Global Process Communication

When a local process was fully updated for the current threshold range, we send a collection of data structures to the global process.

The statistics of the local components are sent as three values: the maximal volume, the total volume and the number of components.

A list of newly *created* and *merged* global components is sent.

The first action taken by the global process is to add all newly *created* global components into the global component list.

3) Global Process Update

We add all global cells in the current value range $(h_i, h_{i+1}]$ to the union-find data structure. The last remaining step is to pool all recorded global *merges*. These *merges* might be redundant and cannot simply be applied one after the other. Instead, we construct one final merge set, where each element contains a number of global components to be merged onto the last element. From the earlier example, the merge list $\{\mathbf{A} \to \mathbf{B},\ \mathbf{A} \to \mathbf{C}\}$ result in the merge set $\{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$, meaning that $\mathbf{A}$ and $\mathbf{B}$ are to be merged onto $\mathbf{C}$ on each process. These *merges* are now applied on the global process.

Finally, the statistics for $(h_{i-1}, h_i]$ are assembled. By adding all instances of the global component list together, we get the full volume of each global component. We combining that data with the statistics of the local components and end up with the overall total volume, largest connected component and number of both local and global components. The fraction between the largest and total volume is recorded according to (1).

4) Global to Local Process Communication

Before computing the next value range $(h_i, h_{i+1}]$, the local process needs to enter a valid state. First, each local process *creates* new global components according to the global process. Similarly, global *merge* operations need to be performed. To this end, the same merge list is sent to every local process and applied there.

The local process update step 1) requires the latest version of all neighboring global ghost cells, which are thus sent to the respective local processes. At this point, the full domain is now in a valid state: neighboring cells belong to the same components on every process. On that grounds, we can start at step 1) again.

4 Evaluation

We evaluate the technical aspects of our algorithm using three fully turbulent flows: a *duct flow* [5] of size $193 \times 194 \times 1000$; and two *isotropic flows* [43] with sizes 512^3 and 4096^3. More information about the data sets and analysis results can be found in Section 5.

The two smaller data sets still fit into the memory of a compute node, which allows us to compare against the serial algorithm of Friederici et al. [13] (Sections 4.1 – 4.2). We use the 4096^3 data set to evaluate our algorithm in a distributed setup (Section 4.3). Finally, we compare our algorithm against a parallel algorithm for computing merge trees in Section 4.4.

All computation times were taken on – where applicable multiple – compute nodes with 2 Intel Xeon E5-2698v3 Haswell 2.3 GHz CPUs (16 cores per CPU) and 64 GB primary memory. Our largest computations were run on 64 of such compute nodes which are connected through the Cray Aries interconnect technology [4].

Table 2: Computation times for parts of the algorithm for varying number of threshold samples. All times in seconds. Obtained by averaging 10 runs on the Duct Flow data set with 32 processes.

Algorithmic Part	Number of threshold samples			
	10^1	10^2	10^3	10^4
Sorting	0.2	0.2	0.2	0.2
Computation	0.6	0.7	0.8	1.1
Communication	0.2	0.4	3.0	31.9

4.1 Strong Scaling

A strong scaling analysis measures how the wall clock time changes with an increasing number of processes working on the same total problem size. We use 1000 threshold samples and run the algorithm 10 times to measure the speedup as the average parallel runtime in relation to the average serial runtime.

Figure 9 shows the speedup of the parallel (blue) over the serial algorithm (red). This is based on average serial computation times of ≈ 82 and ≈ 651 seconds for the duct and isotropic flow, respectively.

First, we notice that the performance increases drastically when going from the serial to the parallel algorithm with 2-4 processes. In fact, the performance more than doubles when doubling the number of processes in this range. This is mostly due to reducing cache misses, which we confirmed with Cray performance tools. When splitting that data over several processors, neighboring data moves closer together in memory and the performance per processor increases on top of the overall performance increase due to the distribution. This effect is stronger for larger data sets (isotropic flow) and increases the performance for up to 32 processors.

Second, we notice that the speedup throttles and goes down around 16-32 processors. While the fastest computation times have been achieved with 32 local processes for both data sets, the computations with 16 processes had almost the same speedup with much less effort. Increasing the number of local processes further provides lesser work for the local processes and induces more work for the global communication. The reason is simple: the connected components of the superlevel sets will quickly outgrow their small local block and span over several blocks, thereby inducing communication costs. Note that at 32 local processes, the processes are already distributed over multiple - in this case two - compute nodes. From that point on we observe an increase in the standard deviation for the runtime up to ≈ 15.8 seconds (duct flow) and ≈ 82.3 seconds (isotropic flow). This indicates high communication costs, since they are rather variable between compute nodes.

Overall, the algorithm does not scale well for small data loads. This is acceptable as the goal is rather to process large out-of-core datasets than to increase performance on formerly manageable sizes.

4.2 Scaling with the Number of Threshold Samples

Increasing the number of threshold samples yields a higher precision for the percolation function, but it also entails more global synchronization. In fact, our algorithm only performs global synchronization steps when a threshold sample h_i is hit. Table 2 reveals the strong overhead induced by the global synchronization. Fortunately, most practical scenarios require only 1000 threshold samples or less, which keeps the ratio of computation to communication time in an acceptable range.

4.3 Data Set Larger than Single Compute Node

We run the percolation analysis for our 4096^3 data set on 64 local processes. Computing the percolation function for this data set size

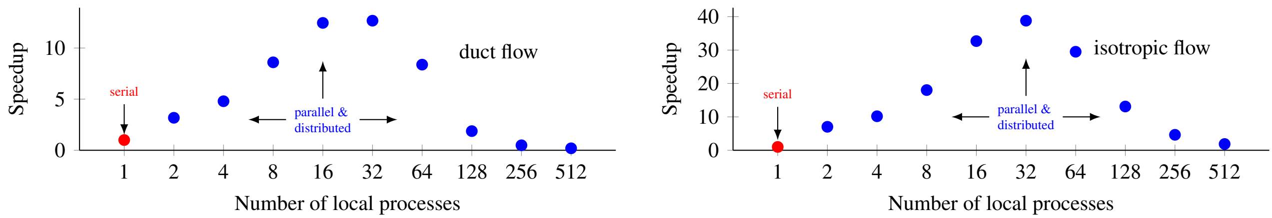


Figure 9: The strong scaling analysis shows the speedup of our distributed algorithm (blue) over the serial algorithm (red). Our algorithm scales very well and benefits from fewer cache misses. Increasing the number of processors further than 16 induces large communication costs leading to a lower speedup. The plots are based on average computation times (10 runs) and 1000 threshold samples.

Table 3: Our method performs *on par* with a state-of-the-art method for computing augmented merge trees due to Gueunet et al. [16]. While the two methods have some similar algorithmic ingredients, the comparison needs to be taken with caution, since the two methods compute different features, use different parallelization technologies (MPI versus OpenMP), and run either distributed (ours) or in a shared memory setting ([16]).

Data Set	Augmented Merge Trees [16]		Our Algorithm	
	# Threads	Computation (s)	# Processes	Computation (s)
Duct Flow	16	5.2	16	4.1
Isotropic Flow	16	15.3	16	18.3

has thus far been impossible with any method known to us.

Each process is run on their own compute node, since a local process requires approximately 31.8GB of input data and data structures. The global process holds about 6.8GB for this case.

Out of a total of 68.4 billion cells, we hold 67.9 billion local cells and 150.8 million global cells. The outer local cell layers consist of 299.6 million cells. This means, we exchange information of about 0.66% of the cells for each threshold sample.

We ran our algorithm four times on this data and the computation times varied between 14 and 21 minutes. This is another indication for the rather variable communication costs between compute nodes.

4.4 Comparison to Parallel Merge Tree Algorithm

We compare our approach with the augmented merge tree computation method of Gueunet et al. [16], which computes in parallel on a shared memory machine. It needs to be stressed that the algorithm of Gueunet et al. [16] is *not* designed to compute the percolation function: a direct comparison is ill-posed. But since both algorithms use similar algorithmic ingredients and address the sub/superlevel sets of a scalar function, we can expect that the computation times are in the same ballpark. We can see this comparison as a sanity check whether or not our implementation performs *on par* with a related state-of-the-art method. Table 3 confirms that both methods need a similar amount of computation time on the same data set and with a similar level of concurrency. Note, however, that our method uses MPI processes and distributes the memory, whereas Gueunet et al. [16] uses shared memory and OpenMP threads.

5 Applications

Percolation analysis has two major applications in turbulence research: First, it can validate a chosen normalization scheme. Second, it is crucial for determining a threshold that defines the coherent structures. We detail these aspects using two turbulent data sets with quite different characteristics: a wall-bounded flow in a duct, and a wall-free flow exhibiting homogeneous isotropic turbulence.

5.1 Normalization

If we can observe a sharp transition in the percolation function, then we know that a chosen normalization scheme accounts for the variations in the data properly. Normalization schemes are required to compare coherent structures in different regions of the flow (e.g., distance to wall) on equal footing.

The duct flow is a good example for this. It models water flowing through a square periodic pipe. Wall-bounded flows are neither homogeneous nor isotropic. Instead, the flow properties depend highly on both the angle and the relative position to the nearest walls. The flow follows vastly different laws depending on the distance to the wall and the in-stream speed. To make up for these variations, the velocity components are normalized. We use the normalization approach of Atzori et al. [5] on their original simulation data, sampled to $193 \times 194 \times 1000$ data points. Average velocity and the root mean square (RMS) are accumulated over long simulation runs. Every component u, v and w is normalized as $\overline{u} = \|\frac{u - u_{average}}{u_{rms}}\|$. Turbulent structures are regions of intense Reynolds stress: $\overline{uv}$, the product of the streamwise and one cross-stream normalized component.

The percolation function for the normalized Reynolds stress in Figure 10 reveals a sharp transition. Hence, the normalization was successful in putting all coherent structures on an equal footing. This is in contrast to the original data where the percolation function of the non-normalized counterpart uv exhibits a vastly different behavior as shown in Figure 12.

The impact of different kinds of normalization has further been explored by Köpp and Friederici et al. [22], who also study the general application of percolation analysis on sampled data.

5.2 Choice of Threshold

After establishing a proper normalization scheme, coherent structures are defined as a superlevel set and one is interested in gathering statistics about them, e.g., their shape and wall distance [5]. But at which threshold? Percolation analysis helps in determining an appropriate threshold: we know that setting the threshold after the percolation crisis means that most of the volume is consumed by a single component. That would invalidate any further analysis of the coherent structures. Hence, the threshold is chosen to be before the percolation crisis.

We can observe this for the duct flow in Figure 10. We determined the percolation threshold $H_c \approx 0.86$ using our method. The component renderings in Figure 10 confirm that a threshold $h = 2H_c$ before the percolation crisis captures the coherent structures well, whereas thresholds at and after the crisis contain few connected components that consume most of the volume, which is unsuitable for the turbulence analysis.

It is important to note that percolation analysis is quite unique in discovering this property. For example, one may be tempted to look at the number of components over the threshold, since one could argue that the threshold with the most components brings out most coherent structures. We plotted that function in Figure 10 as well and it is remarkably featureless over the majority of the threshold range:

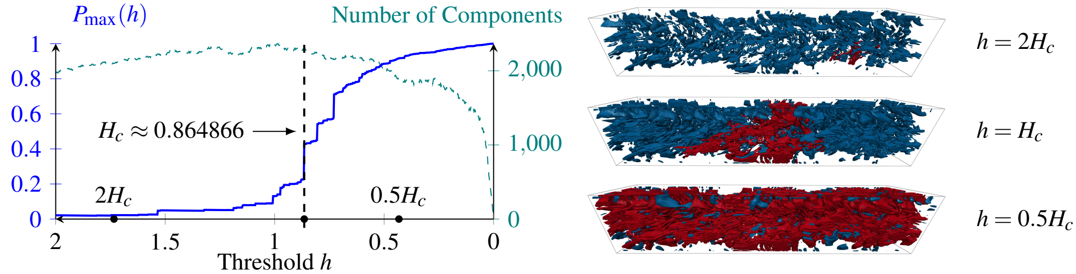


Figure 10: Percolation analysis for the Duct Flow data set. On the right, superlevel sets for thresholds $2H_c, H_c$ and $0.5H_c$ are shown, with the respective largest component highlighted in red. We observe the existence of the percolating cluster for thresholds below H_c and many small connected components for larger thresholds.

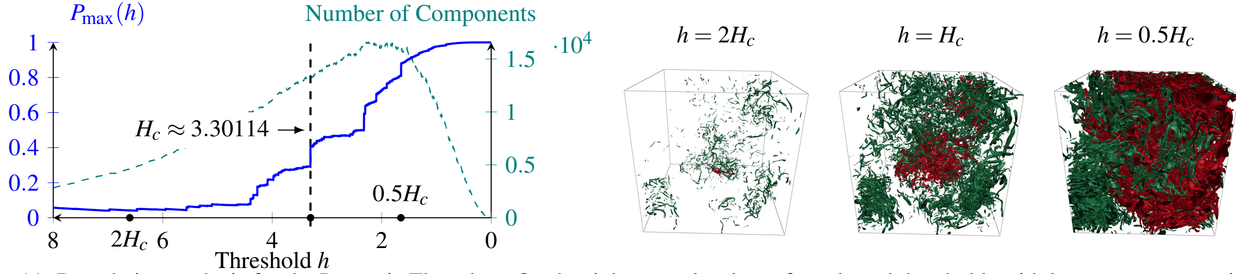


Figure 11: Percolation analysis for the Isotropic Flow data. On the right, superlevel sets for selected thresholds with largest component in red.

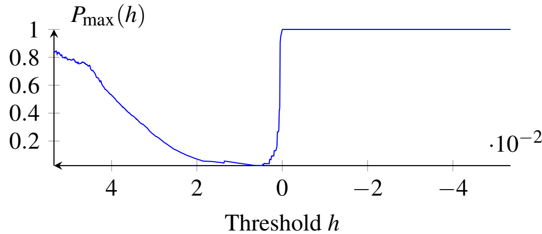


Figure 12: Wall-bounded flows such as the Duct Flow are neither homogeneous nor isotropic: the flow velocity varies greatly with the distance to the wall. Normalization is required to perform successful further analyses. This plot shows $P_{max}(h)$ for the non-normalized, original data. The typical percolation crisis cannot be observed. See Figure 10 for the normalized data.

the number of connected components does not change much. In fact, in this data set, even the volume distribution of the components does not change much over the majority of the threshold range.

Our second data set is a turbulent flow exhibiting homogeneous isotropic turbulence due to Yeung et al. [43]. We consider a 512^3 subset of the data and show the percolation function and volume renderings for some selected thresholds in Figure 11. The data has been computed with periodic boundary conditions, which renders the domain infinite. This data has turbulent structures that share the same statistical properties independent on the spatial translation (homogeneous) and rotation (isotropic) of the underlying frame of reference. A normalization is thus not necessary here: note the clear transition in the percolation function for validation. However, we still need to find an appropriate threshold. Moisy and Jiménez [29] use normalized vorticity magnitude to analyze such flows: regions of high vorticity magnitude correspond to turbulent structures, and after finding an appropriate threshold, they investigate the surface geometry of the connected components. Figure 11 reveals rather few and large turbulent structures after the percolation crisis at $h = 0.5H_c$. Again, a threshold larger than the percolation threshold H_c needs to be chosen to find structures with an appropriate granularity.

6 Conclusions and Future Work

We presented a novel algorithm for percolation analysis of turbulent flows. To the best of our knowledge, it is the first algorithm allowing for this analysis in a distributed memory setting, so that even the largest data sets can be analyzed. Our evaluation shows that the algorithm scales well when increasing the number of processes – up to a limit, when the workload for each process becomes too small and global communication costs rise. As a sanity check, we compared our computation times to the algorithmically loosely related method of Gueunet et al. [16], which perform similar. We showcased the utility of percolation analysis using different large data sets. Our source code is freely available for reproducing these experiments.

The bottleneck of the current approach is the communication during the synchronization step. It is worthwhile to investigate whether a multi-step synchronization could mitigate this issue: ranks within physical compute nodes are synchronized first and then treated as a single local block for a synchronization across physical nodes.

Since percolation analysis is a rather new topic for the visualization community and it is not restricted to the analysis of turbulent flows, we deem it quite interesting to explore other use cases for it.

In relation to our algorithm and its applications, we could fairly easily extend our method to gather additional statistics about superlevel sets. Examples would be the value histogram and the isosurface size, whose relationship has been explored by Scheidegger et al. [38]. Note that these statistics depend on the sampling resolution of the given scalar field, as discussed by both Scheidegger et al. and Khoury and Wenger [21]. An adapted version of our algorithm could enable these computations on a very large, densely sampled grid.

Acknowledgments

The authors wish to thank Sergio Rivas-Gomez and Steven Wei Der Chien for various discussions regarding handling of the super computer. This work was supported through grants from s the Swedish Foundation for Strategic Research (SSF, Project BD15-0082) and the Swedish e-Science Research Centre (SeRC). The computations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at the PDC Centre for High Performance Computing (PDC-HPC).

REFERENCES

[1] A. Acharya and V. Natarajan. A parallel and memory efficient algorithm for constructing the contour tree. In *2015 IEEE Pacific Visualization Symposium (PacificVis)*, pages 271–278, April 2015.

[2] K. S. Alexander and S. A. Molchanov. Percolation of level sets for two-dimensional random fields with lattice symmetry. *Journal of Statistical Physics*, 77(3):627–643, Nov 1994.

[3] K. T. Allhoff and B. Eckhardt. Directed percolation model for turbulence transition in shear flows. *Fluid Dynamics Research*, 44(3):031201, May 2012.

[4] B. Alverson, E. Froese, L. Kaplan, and D. Roweth. Cray XC series network. White Paper WP-Aries01-1112, Cray Inc., 2012.

[5] M. Atzori, R. Vinuesa, A. Lozano-Durán, and P. Schlatter. Characterization of turbulent coherent structures in square duct flow. *Journal of Physics: Conference Series*, 1001(1):012008, 2018.

[6] S. R. Broadbent and J. M. Hammersley. Percolation processes: I. Crystals and mazes. *Mathematical Proceedings of the Cambridge Philosophical Society*, 53(3):629–641, 1957.

[7] H. Carr, C. Sewell, L.-T. Lo, and J. Ahrens. Hybrid data-parallel contour tree computation. In *Proceedings of the Conference on Computer Graphics & Visual Computing*, CGVC '16, pages 73–80. Eurographics Association, 2016.

[8] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. *Computational Geometry*, 24(2):75 – 94, 2003. Special Issue on the Fourth CGC Workshop on Computational Geometry.

[9] F. Cazals, F. Chazal, and T. Lewiner. Molecular shape analysis based upon the Morse-Smale complex and the Connolly function. In *Proc. SoCG*, pages 351–360, 2003.

[10] J. C. Del Álamo, J. Jiménez, P. Zandonade, and R. D. Moser. Self-similar vortex clusters in the turbulent logarithmic region. *Journal of Fluid Mechanics*, 561:329–358, 2006.

[11] S. Dong, A. Lozano-Durán, A. Sekimoto, and J. Jiménez. Coherent structures in statistically stationary homogeneous shear turbulence. *Journal of Fluid Mechanics*, 816:167–208, 2017.

[12] X. Feng, Y. Deng, and H. W. J. Blöte. Percolation transitions in two dimensions. *Physical Review E*, 78(3):031136, 2008.

[13] 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 598: Coherent structures in wall-bounded turbulence*, 2018.

[14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 3 edition, 2014.

[15] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Task-based augmented merge trees with Fibonacci heaps. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 6–15, Oct 2017.

[16] C. Gueunet, P. Fortin, J. Tierny, and J. Jomier. Task-based augmented contour trees with Fibonacci heaps. *IEEE Transactions on Parallel and Distributed Systems*, 30(8):1889–1905, 2019.

[17] T. E. Harris. A lower bound for the critical probability in a certain percolation process. *Mathematical Proceedings of the Cambridge Philosophical Society*, 56(1):13–20, 1960.

[18] C. Harrison, J. Weiler, R. Bleile, K. Gaither, and H. Childs. A distributed-memory algorithm for connected components labeling of simulation data. In J. Bennett, F. Vivodtzev, and V. Pascucci, editors, *Topological and Statistical Methods for Complex Data*, pages 3–19, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[19] J. Hoshen and R. Kopelman. Percolation and cluster distribution. I. Cluster multiple labeling technique and critical concentration algorithm. *Phys. Rev. B*, 14:3438–3445, Oct 1976.

[20] H. Kesten. The critical probability of bond percolation on the square lattice equals $\frac{1}{2}$. *Communications in Mathematical Physics*, 74(1):41–59, 1980.

[21] M. Khoury and R. Wenger. On the fractal dimension of isosurfaces. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1198–1205, Nov. 2010.

[22] W. Köpp, A. Friederici, M. Atzori, R. Vinuesa, P. Schlatter, and T. Weinkauf. Notes on percolation analysis of sampled scalar fields. In *Topology-Based Methods in Visualization (TopoInVis)*, 2019.

[23] A. G. Landge, V. Pascucci, A. Gyulassy, J. C. Bennett, H. Kolla, J. Chen, and P. Bremer. In-situ feature extraction of large scale combustion simulations using segmented merge trees. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1020–1031, Nov 2014.

[24] T. Lewiner. Constructing discrete Morse functions. Master's thesis, Department of Mathematics, PUC-Rio, 2002.

[25] A. Lozano-Durán, O. Flores, and J. Jiménez. The three-dimensional structure of momentum transfer in turbulent channels. *Journal of Fluid Mechanics*, 694:100–130, 2012.

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

[27] Y. Maciel, M. P. Simens, and A. G. Gungor. Coherent structures in a non-equilibrium large-velocity-defect turbulent boundary layer. *Flow, Turbulence and Combustion*, 98(1):1–20, Jan 2017.

[28] 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, editors, *Parallel Processing and Applied Mathematics*, pages 186–195, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[29] F. Moisy and J. Jiménez. Geometry and clustering of intense structures in isotropic turbulence. *Journal of Fluid Mechanics*, 513:111–133, 2004.

[30] D. Morozov and G. Weber. Distributed merge trees. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 93–102, New York, NY, USA, 2013. ACM.

[31] D. Morozov and G. H. Weber. Distributed contour trees. In P.-T. Bremer, I. Hotz, V. Pascucci, and R. Peikert, editors, *Topological Methods in Data Analysis and Visualization III*, pages 89–102, Cham, 2014. Springer International Publishing.

[32] M. E. J. Newman and R. M. Ziff. Efficient Monte Carlo algorithm and high-precision results for percolation. *Phys. Rev. Lett.*, 85:4104–4107, Nov 2000.

[33] V. Pascucci and K. Cole-McLaughlin. Parallel computation of the topology of level sets. *Algorithmica*, 38(1):249–268, Jan 2004.

[34] J. M. Patchett, B. Nouanesengesy, J. Pouderoux, J. Ahrens, and H. Hagen. Parallel multi-layer ghost cell generation for distributed unstructured grids. In *IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 84–91, Oct 2017.

[35] P.-F. Rodriguez and A.-S. Sznitman. Phase transition and level-set percolation for the gaussian free field. *Communications in Mathematical Physics*, 320(2):571–601, Jun 2013.

[36] J. Roerdink and A. Meijster. The watershed transform: Definitions, algorithms and parallelization strategies. *Fundamenta Informaticae*, 41(1-2):187–228, 2000.

[37] M. Sahimi. *Applications of percolation theory*. CRC Press, 2014.

[38] C. Scheidegger, J. Schreiner, B. Duffy, H. Carr, and C. Silva. Revisiting histograms and isosurface statistics. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1659–1666, 11 2008.

[39] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, Apr. 1975.

[40] J. Teuler and J. Gimel. A direct parallel implementation of the Hoshen–Kopelman algorithm for distributed memory architectures. *Computer Physics Communications*, 130(1):118 – 129, 2000.

[41] D. Tiggemann. Simulation of percolation on massively-parallel computers. *International Journal of Modern Physics C*, 12(6):871–878, 2001.

[42] T. Weinkauf and D. Günther. Separatrix Persistence: Extraction of salient edges on surfaces using topological methods. *Computer Graphics Forum (Proc. SGP '09)*, 28(5):1519–1528, July 2009.

[43] P. K. Yeung, D. A. Donzis, and K. R. Sreenivasan. Dissipation, enstrophy and pressure statistics in turbulence simulations at high Reynolds numbers. *Journal of Fluid Mechanics*, 700:5–15, 2012.

[44] R. M. Ziff and C. R. Scullard. Exact bond percolation thresholds in two dimensions. *Journal of Physics A: Mathematical and General*, 39(49):15083–15090, Nov 2006.