The Journal of Supercomputing
https://doi.org/10.1007/s11227-024-06374-5

A distributed approach for persistent homology computation on a large scale

Riccardo Ceccaroni[1] · Lorenzo Di Rocco[1] · Umberto Ferraro Petrillo[1] · Pierpaolo Brutti[1]

Accepted: 17 July 2024

Abstract
Persistent homology (PH) is a powerful mathematical method to automatically extract relevant insights from images, such as those obtained by high-resolution imaging devices like electron microscopes or new-generation telescopes. However, the application of this method comes at a very high computational cost that is bound to explode more because new imaging devices generate an ever-growing amount of data. In this paper, we present *PixHomology*, a novel algorithm for efficiently computing zero-dimensional PH on 2D images, optimizing memory and processing time. By leveraging the Apache Spark framework, we also present a distributed version of our algorithm with several optimized variants, able to concurrently process large batches of astronomical images. Finally, we present the results of an experimental analysis showing that our algorithm and its distributed version are efficient in terms of required memory, execution time, and scalability, consistently outperforming existing state-of-the-art PH computation tools when used to process large datasets.

Keywords Persistent homology · Distributed computing · Apache spark · Large-scale image analysis

This work was carried out while occupying the role of visiting student at Bielefeld University, Germany.

✉ Lorenzo Di Rocco
lorenzo.dirocco@uniroma1.it

Riccardo Ceccaroni
riccardo.ceccaroni@uniroma1.it

Umberto Ferraro Petrillo
umberto.ferraro@uniroma1.it

Pierpaolo Brutti
pierpaolo.brutti@uniroma1.it

[1] Department of Statistical Sciences, Università di Roma "La Sapienza", P.le Aldo Moro 5, Rome 00185, Italy

Published online: 12 August 2024

1 Introduction

Since the advent of the first electron microscopes, the field of electron microscopy (EM) has undergone significant transformations. Modern microscopy techniques are now achieving near-atomic resolution in the structural analysis of individual proteins and molecular complexes. As a result of this, it is becoming common to generate large digital image files, often above one terabyte in size, in a single data acquisition session [1]. This advancement poses serious performance challenges when it turns to the analysis of these images.

A similar problem affects also other application domains. For example, the Vera C Rubin Observatory is a new-generation ground-based telescope currently under construction (see [2]). When ready, it is expected to generate approximately 15 terabytes of data per night. Such an impressive amount of data requires very efficient methods to be analyzed [3].

In contexts like these, topological data analysis (TDA) [4] plays a significant role as a tool for the automatic extraction of relevant structural information from large datasets of images. This is especially true for persistent homology (PH) [5], a fundamental component of TDA, capable of constructing multiresolution, noise-resilient topological features from a variety of different data clouds [6].

Several techniques have been proposed so far for the efficient PH calculations (see, e.g., [7]). However, processing large images within a reasonable time is still impractical. The primary performance bottleneck is the complex computational procedure employed by many of these techniques, i.e., the filtration of simplicial and cubical complexes [8]. The execution cost of this procedure increases exponentially with the size of the input data. For this reason, innovative algorithms and software solutions that can efficiently handle vast datasets of very large images are required.

In this work, we propose *PixHomology*, a novel algorithm for computing the particular case of zero-dimensional PH on digital images that speed up the filtering of a simplicial complex. Our approach offers a substantial reduction in memory usage compared to existing methods, like the one employed by the state-of-the-art *Ripser* package, when applied to zero-dimensional PH computation. We also introduce a software pipeline for using *PixHomology* on a distributed system, to compute zero-dimensional PH on large batches of images.

We evaluate the performance of our algorithm and its distributed version using a reference dataset of images and show its efficiency compared to existing methods and software, making it the fastest algorithm available nowadays for zero-dimensional PH computation, in both its sequential and distributed versions.

Organization of the paper In Sect. 2, we provide a short introduction to the theoretical concepts behind the PH computation problem. Then, in Sect. 3 we review the existing literature on computational methods and software tools for efficient PH evaluation. In Sect. 4, we briefly describe the MapReduce distributed computing paradigm and its implementing framework, Apache Spark. Following this, in Sect. 5, we present our novel algorithm for efficiently computing zero-dimensional PH and its distributed version, together with several variants we developed to improve upon its original performance. In Sect. 6, we report the results of a thorough experimental analysis designed to assess the performance of our algorithm also in comparison with other existing state-of-the-art tools and methods for PH computation. Finally, some concluding remarks are given in Sect. 7.

2 Theoretical background

2.1 Simplicial and cubical complexes

Simplicial and cubical complexes are the fundamental building blocks of computational topology to fully describe topological spaces. Simplicial complexes consist of simplices, such as vertices, edges, and triangles. In general, a d-simplex represents the convex hull of $d+1$ points, and each subset of these $d+1$ points forms a face of this d-simplex. A collection of simplices, denoted by K, represents a simplicial complex if it satisfies two conditions: All faces of a simplex in K also belong to K, and the intersection of any two simplices in K is either empty or a common face.

For cubic complexes, an elementary interval can be described as a unit interval $[k, k+1]$ or as a degenerate interval $[k, k]$. For a d-dimensional space, a cube is the product of d elementary intervals, denoted $\prod_{i=1}^{d} I_i$. The dimension of a cube is determined by the number of non-degenerate intervals in this product. In particular, 0-cubes, 1-cubes, 2-cubes, and 3-cubes correspond to vertices, edges, squares, and 3D cubes, respectively. When comparing two cubes a and b in $\mathbb{R}^d$, a is considered to be the face of b only if a is contained in b. A cubic complex of dimension d consists of cubes of dimensions at most d. Similar to a simplicial complex, it must be closed under operations with faces and intersections.

For an in-depth discussion of these topics, we refer the interested reader to [4, 9].

2.2 Persistent homology

PH is a fundamental concept in TDA specifically focusing on Z^2 homology (see [4, 10] for a thorough introduction to this topic).

In the context of PH, we start with a topological space X and a filtering function $f : X \rightarrow R$. This method examines the homological transformations of the sublevel sets, denoted as $X_t = f^{-1}(-\infty, t]$. The algorithm captures the inception and extinction times of the homology classes as the subsets evolve from $X^{-\infty}$ to $X^{+\infty}$. For example, it identifies components as zero-dimensional homology classes, tunnels as one-dimensional classes, voids as two-dimensional classes, and so on. *Birth* implies the emergence of a homology class, while *death* implies its trivialization or amalgamation with another class that emerged earlier. The persistence or lifetime of a class represents the time difference between its death and birth. Homology classes

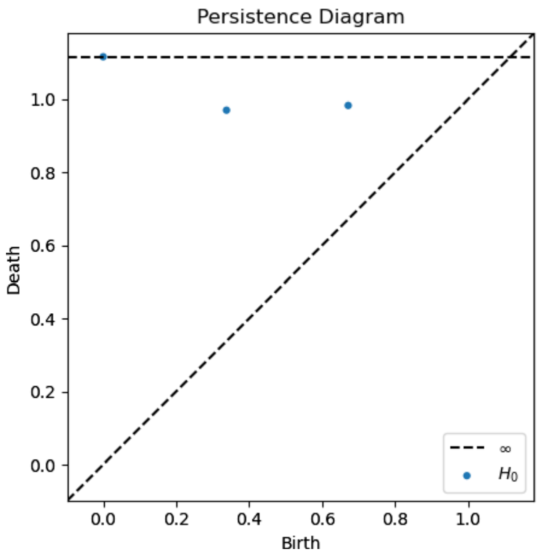

Fig. 1 A point (x, y) in the PD indicates a topological feature of dimension 0 (H_0) born at x and that persists until y. We call x the p_{birth} and y the p_{death}. By definition, all points should lie above the diagonal. The horizontal dashed line represents infinity

with greater persistence provide information about the global structure of the space X, which is affected by the function f.

A common method for visualizing persistence is a persistence diagram (PD), shown in Fig. 1, consisting of points on a two-dimensional plane, each corresponding to a PH class. These points are defined by their birth and death times.

A key reason for using persistence is the stability theorem [11]: For any two filtering functions f and g, the difference in their persistence is always bounded by the L^∞norm of their dissimilarity:

$$||f - g||_\infty := \max_{x \in X} |f(x) - g(x)|.$$

This ensures that persistence serves as a distinctive signature. If two persistence outputs differ, it means that the functions are different.

2.3 Computation of persistence

The original algorithm for computing persistence [5] operates in cubic time relative to the size of the complex. This approach requires preprocessing of the data. In the case of images, the function f is defined for all pixels. These values are first interpreted as the values of the vertices of the complex. Then, the complex filtration is calculated and a sorted boundary matrix is created.

During filtration, the process entails adding cells with increasing values to the complex one by one. To achieve this, an algorithm for building the filtration extends the function to all cells within the complex by assigning each cell the maximum value among its vertices. Then, all cells are sorted in ascending order according to the function value. As a result, each cell is added to the filtration according to all of its faces, creating a sequence of cells known as *lower-star filtration*. This ordering of cells allows the creation of a sorted boundary matrix.

In the reduction phase, the algorithm performs column reductions on the sorted boundary matrix, proceeding from left to right. Each new column is reduced by adding it to already reduced columns, to maximize the lowest nonzero entry. The final reduced matrix contains all the information about PH.

3 Related work

Several methodological and software contributions have been proposed so far to support the efficient calculation of PH. One of the first software tools to be proposed for this purpose is the *Plex Library*, developed by the Computational Topology Group at Stanford University [12]. *Dionysus* [13] has been instead the first software package to implement the dual algorithm. Starting from the observation that cohomology groups are usually faster to compute, this algorithm reformulates the problem of homology group computations into a cohomology group computation problem (for more info see [14, 15]).

Phat [16] is the first software to implement a matrix reduction algorithm that can be executed in parallel, to accelerate the analysis of large datasets. *Gudhi* [17] implements a comprehensive library offering functionalities from basic to advanced PH, including new data structures for simplicial complexes and the boundary matrix. Finally, *Ripser* [18] is considered the gold standard solution in this field, thanks to its versatility and efficiency. It uses several optimizations and shortcuts to speed up the computation of PH in all dimensions and has demonstrated superior performance to other software tools in terms of both speed and memory efficiency [7].

Indeed, the analysis of very large datasets can be computationally prohibitive, even for efficient tools like *Ripser*. A natural solution to this problem is distributed computing. *DIPHA* [19] has been so far the first software we know of that implements PH computation with distributed computing, enabling efficient processing of large-dimensional data. It works by efficiently partitioning the PH computation problem into subprocesses to be concurrently run on the nodes of a distributed system. This allows *DIPHA* to compute PH on much larger instances than would be possible on a single machine. Moreover, the performance speed-up granted by parallelism introduced by DIPHA at least compensates for the overhead caused by communication between nodes.

Recently, alternative high-performance software solutions like CubicalRipser [20] and Cubicle [21] have been introduced for the PH computation on images.

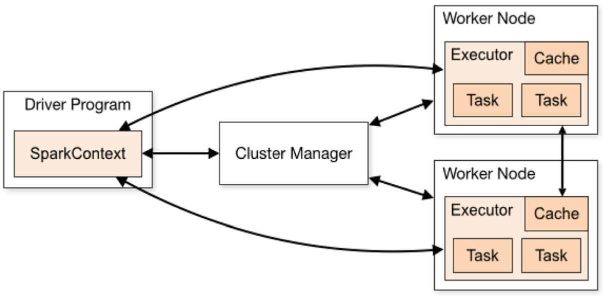

Fig. 2 Apache Spark architecture. Example for a reference installation featuring two worker nodes and one driver application. Each worker node in this figure runs one executor process and two tasks. The overall distributed execution is orchestrated by a cluster manager

4 Apache spark

Apache Spark [22] is one of the most popular engines for large-scale data processing and is based on RDDs (Resilient Distributed Datasets) and DataFrames. These are distributed memory abstractions that allow programmers to perform in-memory computations on large clusters in a fault-tolerant manner. The former are collection of key–value pairs to be processed by means of distributed transformations. The latter are table-like collections provided with the Spark SQL module, which optimizes structured data processing by introducing SQL-like logic in a distributed context.

The physical architecture of a Spark cluster, shown in Fig. 2, is characterized by a master node that oversees a set of *worker* nodes via daemon processes. Data are typically distributed among the worker nodes, and MapReduce is also supported.

A Spark application communicates with a *Cluster Manager*, which is the process that manages the computing and storage resources of the cluster. It includes a *driver* process and a set of *executor* processes. The driver process communicates with the Cluster Manager to learn where the data are located and what physical computing resources are available. Then, in each worker node, a set of parallel executor processes is activated according to the number of threads. For example, a cluster with three worker nodes, each with two threads, means a potential number of six parallel executor processes.

4.1 The MapReduce paradigm

The computations within Spark are formulated according to MapReduce, a programming paradigm that allows massive scalability [23]. MapReduce is composed of two tasks: mapping and reducing. The mapping phase transforms a dataset into another form with elements organized into key–value pairs. Subsequently, the reduction process uses the output of a map as input and combines those data tuples into a smaller set of tuples. As the name suggests, reduction always

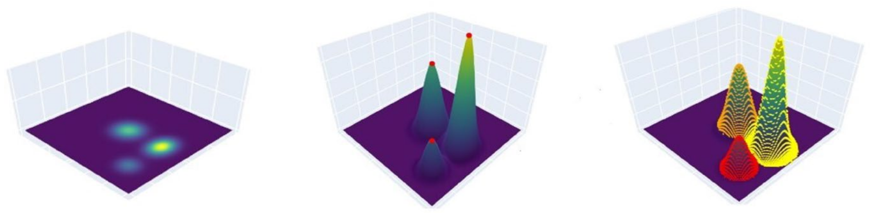

Fig. 3 PH calculation using *PixHomology* on an image containing three components defined by Gaussian functions. Initially, each pixel is linked to its neighbor with the highest value, and *PixHomology* detects relative maxima as *birth* values. Subsequently, all the minimum or saddle points are located. The first value of these points that connect the two components represents the *death* value of the component with the lower *birth* value. Finally, the process ends with identifying the absolute minimum in the image, which serves as the ultimate death point associated with the component relative to the absolute maximum

follows mapping. Consequently, assuming the input dataset is organized as a set of key–value pairs, it is initially distributed among the worker nodes of a cluster. Then, batches of key–value pairs are processed in parallel by concurrent executor processes on the worker nodes where these data are found. Reduce functions require a preliminary step to group on the same node all pairs having the same key (*shuffle* operation). When working with large datasets, this preliminary step makes the Reduce function potentially expensive from a computational perspective.

4.2 Fault-tolerant applications

Spark supports various fault tolerance strategies including checkpointing, task replication, and error handling, all of which contribute to the reliability and robustness of the computing process. For instance, checkpointing allows Spark to periodically save the state of distributed data structures to resilient storage, enabling recovery from failures without recomputing from scratch, while task replication ensures that tasks are rerun on different nodes in case of failures [24, 25].

5 Our contribution

In this section, we introduce a novel algorithm for computing PH on 2D images: the *PixHomology* algorithm.[1] Our algorithm comes with two relevant advantages concerning the existing literature. First, it offers a substantial reduction in memory usage compared to existing methods like the *lower_star_img* function of *Ripser* package, when applied to zero-dimensional PH computation. Second, it has been conceived to process large batches of images in parallel using a distributed system in a more efficient way than other distributed systems (i.e., *DIPHA*).

The first goal has been achieved by overcoming a relevant performance bottleneck existing in traditional general filtration PH algorithms, i.e., the computation of

[1] The source code of *PixHomology* is available at https://github.com/riccardoc95/Sparksistence

adjacency matrices. Being purposely designed to deal with the particular case of zero-dimensional PH computation, our algorithm can avoid all the computational burden of constructing and analyzing these matrices while dramatically reducing the overall amount of memory required for its execution.

The second goal has been reached by using the MapReduce paradigm to develop a distributed PH pipeline based on our algorithm. This allows the execution of the algorithm concurrently on very large batches of images, in an efficient and scalable way.

5.1 The *PixHomology* algorithm

The algorithm we propose, here called *PixHomology*, has been designed to process efficiently very large images as input while yielding zero-dimensional PH as its output. Specifically, the algorithm will provide the *birth* and *death* values of each object within the image, along with their pixel coordinates.

The straightforward implementation and the computational efficiency result from constraining the application to 2D images with specific characteristics. The algorithm initiates by linking each pixel to its highest value neighbor among the 8 surrounding pixels. This process enables the division of the image into connected components, which are later united to generate the points on the PD (see Fig. 3). A crucial condition for the proper operation of *PixHomology* is that the pixel containing a local maximum value must not have any neighboring pixels with the same value among its 8 neighbors. While the application domains of this algorithm may appear limited at this stage, it is important to note for example that whenever Gaussian noise is added to a signal in an image, the image satisfies the necessary conditions for the application of *PixHomology*.

One of the key points of our proposal, as visible in Algorithm 1, is the usage of a *maxpool2d* function with a kernel size of 3, a stride of 1, and a padding of 1. This function yields an image of the same initial dimensions, with each pixel's value being set to the maximum among the pixel and its eight neighboring pixels. The *arg-maxpool2d* function employs identical parameters as the *maxpool2d* function. However, instead of returning the pixel values with the highest value, it returns the indices corresponding to these pixels.

Given an input image I, the algorithm we propose requires the following steps.

- *Step 1: Identification of the concave components.*

  This step is about the identification of the concave components within the image I.

  To accomplish this, we use the *arg-maxpool2d* function to compute a new image M of the same initial dimensions of I, with each pixel's value being set to the maximum among the pixel and its eight neighboring pixels. Then M is processed in a loop.

  In each iteration, every element $x \in M$ is replaced by the value $M[x]$, which corresponds to the value located at position x of M. This process ends when x is equal to $M[x]$ for every element $x \in M$.

- *Step 2: detection of birth points and re-indexing of components*

  At this point, all the identified elements in M are marked with the index of their corresponding relative maximum neighbor found in I. The unique values of M are the position of the relative maximums that are stored in an array labeled p_{birth}, i.e., the birth points. A separate *birth* array records the values of I at these p_{birth} points. We sort *birth* and p_{birth} array so that *birth* array is in descending order. Subsequently, we update the values in M with positions corresponding to the values in the p_{birth} array. This process assigns incremental numbers to the components of I, starting with the component that has the smallest relative maximum and ending with the component that has the largest relative maximum.
- *Step 3: edge points detection*

  Upon partitioning the image into distinct components, we calculate $maxpool2d(M)$ and $-maxpool2d(-M)$. The region where these two outcomes differ represents the edges of the components within the image I. It is essential to note that matrix M comprises integer values, each signifying a unique component in the image. These component values are not arbitrarily assigned; rather, they follow the order of the relative maxima present in I.

  An array B that contains the indices of all the edges of the components is generated.
- *Step 4: distillation*

  By definition, death points are located along the edges of the components. To connect two neighboring components, they must be either relative minimum points or saddle points. In this step, we verify whether the points with an index in B are minimum or saddle points. If these criteria are unsatisfied, the index is removed from B. We characterize a minimum point as a pixel with the lowest value in comparison to its 8 neighboring pixels. In contrast, a saddle point is a pixel that serves as a minimum along one axis and a maximum along the other axis, always about its 8 neighboring pixels.
- *Step 5: dead points identification and partition merging*

  We sort B in descending order to maintain the chronological sequence of partition merges in M. Each point x in B that is adjacent to two partitions triggers their merger. We call x the p_{death} point for the lesser-indexed partition. Merger history is captured in vector C, which stores the new index of each partition after the merger.

  To further improve the efficiency of the algorithm, we restrict the changes to the eight pixels around the point x, rather than the entire array M.
- *Step 6: PD construction*

  In this step, we create a PD, to associate each birth point with its respective death point within the same partition. We then extract the *birth* and *death* values for the p_{birth} and p_{death} points from I and aggregate them into a *DGM* matrix.

When compared with the existing literature, PH achieves minimal memory usage and efficient execution times by eliminating the classical *adjacency matrix* in step 1 and accelerating filtration in step 5 by avoiding pixel-by-pixel control. In terms of computational complexity, each operation in Algorithm 1 incurs a cost of $O(n)$, where n is the number of pixels in image I, except for the while loop. In a highly improbable scenario where there is only one component in the image with *birth* value in the last pixel of I, the while loop concludes after $n(n-1)$ operations.

Algorithm 1 Outline of the *PixHomology* algorithm. It assumes the availability of the *unique* function, which is used to extract the unique elements from an array, the*reindex* function, which is used to reset the indexes of the components so that the highest index corresponds to the component containing the pixel with the greatest value, and the maxpool2d and arg-maxpool2d functions which return, respectively, the output of maxpool operation on 2D images and the indexes of this operation. Additionally, the *distillation* function, detailed in the step with the same name, identifies and removes unnecessary pixel indices in subsequent steps.

```
Input: Image $I$
Output: PD, the matrix $dgm$ with birth and death values
 1: $M = \text{arg-maxpool2d}(I)$
 2: while $\exists\, x \in M : x! = M[x]$ do
 3:     $x == M[x]\ \forall x \in M$
 4: end while
 5: $p_{birth} = unique(M)$, $M = reindex(M)$
 6: $B = \text{where}(\text{maxpool2d}(M)! = -\text{maxpool2d}(-M))$
 7: $B = distillation(B)$
 8: the elements within B are arranged in a descending order
 9: for all $x$ in $B$ do:
10:     if $x$ is border of two partitions $P_1$ and $P_2$ then
11:         $x$ is added in $p_{death}$
12:         $C[x] = min(P_1, P_2)$
13:     end if
14: end for
15: $dgm = (I[p_{birth}], I[p_{death}])$
```

5.2 The distributed *PixHomology* pipeline

Tools like *DIPHA* overcome the heavy memory and computational requirements of many PH algorithms by distributing the computation across multiple nodes. However, this approach presents a significant challenge: Dividing the image into patches for parallel processing reduces the memory required per node, but it also leads to substantial data traffic between nodes, as the computational units handling adjacent patches must communicate frequently to detect *birth* and *death* coordinates within their respective regions.

Given the expected smaller memory footprint of our algorithm, we were able to adopt a different solution: concurrently processing multiple images at once using the different computational units of a distributed system. Based on this idea, we developed a simple distributed pipeline for our PH algorithm using the Apache Spark framework. The choice of this technology over other distributed computing

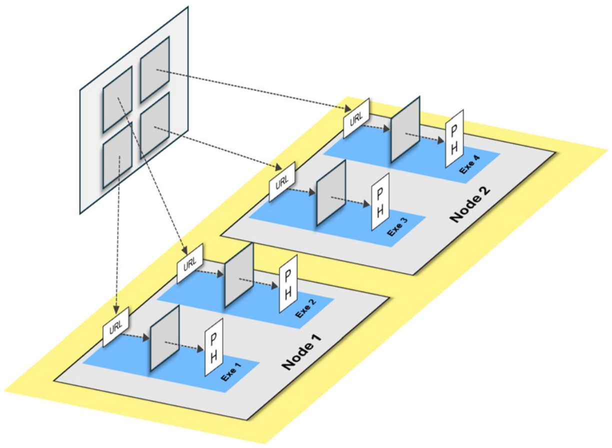

Fig. 4 An overview of the distributed workflow of *PixHomology* on a Spark cluster involving four executor processes scattered across two computing nodes. After partitioning the URLs of the images across the various executors, each executor performs two map operations. The former operation loads the image into memory, while the latter performs the PixHomology algorithm to compute the zero-dimensional PH

frameworks has been motivated by its inherent scalability and by its ability to better operate on cloud-based big data processing infrastructures.

The proposed pipeline, shown in Fig. 4, consists of two Spark distributed transformations, followed by one Spark distributed action to collect the computation results. The first distributed transformation, implemented as a *map* operation, is named *load_preprocess_image*(). It is used by each Spark executor to load each input image in memory and prepare auxiliary data structures used by the algorithm for subsequent steps. Images are retained in memory using 2D array representations.

The second distributed transformation, also implemented as a *map* operation, is named *process_image*(). It is used by each executor to apply the PH algorithm to each of the arrays loaded in the previous step, yielding zero-dimensional PH.

5.2.1 Variants

Once ready, we performed a preliminary experimental analysis targeting our distributed algorithm, to identify hotspots and address possible performance bottlenecks. The insights gained from this analysis were used for the development of the following more efficient variants.

- **Variant 1: reducing images loading time.**
  A simple yet effective way to handle input data in Spark is to load the entire dataset in the memory of the driver application and then convert it into a distributed RDD representation using the `parallelize()` method. This approach fails with huge datasets both because of the large amount of memory required by the driver application to initially store the datasets and because of

the long execution times needed to first load in memory the dataset and then distribute it across the cluster.

We developed an alternative approach where each executor loads the images to be processed on its own, either from the local disk or from a remote web server. This solution alleviates the pressure on the driver application and reduces loading times. From the technical viewpoint, this solution was implemented by avoiding the need for the driver to load any images. Instead, we modified the original *load_preprocess_image*() distributed transformation to instruct executors about the physical location of the images to load and process. This variant is much faster than the original one, so we will use it in all the following experiments. Notice that our improved approach is still vulnerable to the case where the size of the images to be processed exceeds the amount of memory available to a single executor. Such a problem could be circumvented by reducing the number of executors to use, so as to increase the amount of memory available to each of them.

- **Variant 2: reducing the amount of pixels to process.**

  In the analysis of digital images, processing every individual pixel might not be imperative. Hence, there could be situations where it proves beneficial to set some sort of threshold value t, below which pixels do not necessitate examination.

  Applying this technique can be of significant help when analyzing large batches of high-resolution images, as it can significantly reduce the computational burden. In our case, we consider a very simple preprocessing method, where we assign a threshold to each image within the dataset. This threshold is acquired with each image and then applied to the image itself using the *load_preprocess_image*() function in each experiment. Consequently, when a new image is loaded, all pixels with a value under this threshold are identified as *background pixels* and excluded from the subsequent analysis. Notice that we do not aim to find an optimal thresholding strategy, but rather to assess the potential performance gain achievable by applying this technique.
- **Variant 3: improve the workload distribution.**

  When a distributed data structure is created in Spark, it is automatically partitioned into a number of partitions based on factors such as the number of executors and the size of the data. The developer can override these settings and specify the number of partitions to use.

  Partitioning is important because partitions are the basic units of parallelism in Spark. A good partitioning strategy will ensure that all executors in a distributed system have approximately the same workload, resulting in shorter execution times.

  In our particular case, the processing time of each image is relatively long. Provided that each executor is fed with approximately the same number of partitions, this does not necessarily guarantee a balanced workload, as the processing time for each image may vary depending on several factors. Moreover, because the amount of data stored in each partition is very small (i.e., the address where each image is stored) and the time spent transmitting it over the network is

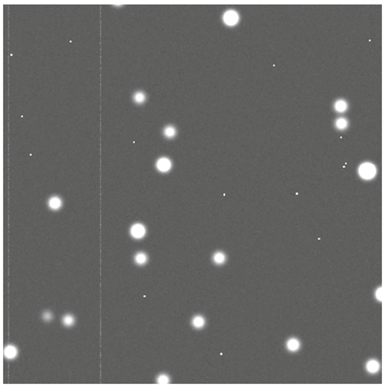
Fig. 5 A cropped section of 500×500 pixels from an image in the dataset

negligible, there is no need to ensure the locality of data in the worker nodes. To account for these factors, we tried several partitioning strategies for our pipeline.

– *Strategy 1: partitioning by the number of executors*

  Let n be the number of input images and m be the number of executors available on the distributed system. We partition the image dataset into m partitions, one for each executor. To do so, we first create a list of n integers, where each integer represents the index of an image in the dataset. We then shuffle the list of integers and divide it into m partitions, each containing n/m integers. Finally, we assign each partition to an executor.

  This partitioning strategy ensures that all executors have approximately the same workload, as each partition contains the same number of images. It also minimizes the amount of data that need to be transmitted over the network, as each executor processes the images in its assigned partition. However, it can perform poorly with datasets that have a skewed image complexity distribution. If an executor is assigned a partition with images requiring very long processing times, its execution time will dominate the overall algorithm execution time, making it a *straggler*.

– *Strategy 2: partitioning by the number of images*

  Let n be the number of input images and m be the number of executors available on the distributed system. We partition the image dataset into n partitions, one for each image. Then, we let Spark assign partitions to executors using the default partitioning strategy.

  As before, this partitioning strategy ensures that all executors have approximately the same workload, as each executor is initially assigned approximately the same number of partitions. Differently from the previous case, we expect this strategy to be more effective in handling datasets that have a skewed image complexity distribution. If an executor is assigned a partition with an image requiring processing times much longer than other partitions, Spark may automatically reassign the remaining partitions to other executors, thus mitigating the effects of that straggler on the overall algorithm execution time.

– *Strategy 3: overriding standard partitioning rules*

We expect strategy 2 to help mitigate the effects of stragglers, but still under the possibly wrong assumption that all images have approximately the same processing time. To overcome this problem, we introduce a third partitioning strategy. We estimate the computational cost for the analysis of input images and then use the lasting processing time (LPT) rule [26] to schedule the analysis of input images on all executors to ensure a uniformly distributed workload.

LPT provides a heuristic solution to the NP-hard scheduling problem with identical parallel machines and no preemption. It has already been used in the literature to minimize the completion time (also referred to as *makespan*) of Spark jobs, as shown in [27].

The goal consists of distributing n jobs, characterized by specific processing times $\{p_j, j = 1, \ldots, n\}$, among a set of m executors operating in parallel to minimize the maximum makespan. Basically, the LPT rule sorts the jobs in a non-increasing order according to their processing times and, iteratively, assigns a job to the machine that currently has the minimum completion time.

In our case, each job corresponds to the PH calculation on a single image, and we estimate the processing time as proportional to the number of pixels that have been identified as non-background during the preprocessing phase (see Variant 2).

6 Experimental analysis

In this section, we present the results of an experimental analysis done to assess the performance of our algorithm and its variants, according to several metrics. We also compare its performance with the state-of-the-art software in this field. While recent software [20, 21] has demonstrated outstanding performance in computing PH on images, this section will focus on comparing results with the *Ripser* package. This decision is based on potential differences in output between software using cubical complexes and those analyzing simplicial complexes. However, a comparison will also be conducted with *DIPHA*, the only software implementing distributed calculation, despite its use of cubical complexes.

6.1 Computing environment

Our experiments were conducted on the Terastat HPC infrastructure of 24 nodes, where each node is equipped with 128 computing units and a maximum of 2 GB RAM of memory per computing unit (see [28] for more details). Experiments with our pipeline were performed using a Spark installation with multiple computing nodes, providing a total of 256 GB RAM of memory per node.

6.2 Dataset

The dimensions of the images in the dataset were selected to enable a fair comparison between the software using the hardware resources available to us. Smaller dimensions were omitted as they fell outside the scope of our study.

We generated for our experiments a dataset of 90 astronomical images (see Fig. 5) using the Astropy library [29]. The creation of the image starts with the creation of an array of dimensions $10,000 \times 10,000$, where all pixels are set to zero. Gaussian readout noise and sky background are added. Stars are then generated and added to the image. The details of the process are described in [30]. To make the images more realistic, we set the number of objects within each image to approximately 340, 000. The resulting images with a precision of *float32* were saved in *.fits* format, resulting in a dataset size of about 36 GB.

6.3 Performance assessment of PH

In our first round of experiments, we analyzed the experimental performance of several different variants of our *PixHomology* algorithm in order to find the combination leading to the best performance.

6.3.1 Variant 1: reducing images loading times

In this experiment, we conducted a comparative analysis of the execution time required by two distinct approaches for handling the input dataset of images. The first approach requires the driver program to be responsible for loading the entire dataset in memory and distributing it to the computing nodes (i.e., **load_driver**). The second approach (i.e., **load_self**), detailed in Sect. 5.2.1, allows the executors to load the images on their own.

This latter approach resulted to be significantly faster than the former (results not reported but available upon request). The observed efficiency of **load_self** is mostly due to its ability to greatly reduce the communication overhead for the driver program, as well as to deeply decrease its memory requirements.

6.3.2 Variant 2: reducing the amount of pixels to process

In this experiment, we assessed the impact of different filtering rules on the performance of our algorithm. Namely, we evaluated how the number of background pixels removed by the filtering rule introduced in variant 2 (see Sect. 5.2.1) affected the efficiency of the algorithm. We investigated four different scenarios:

- **vanilla**: Baseline scenario: no filtering rule is applied.
- **filter_std**: Standard filtering rule: all pixels below a predefined threshold are classified as background.

Table 1 Execution time comparison of *PixHomology* using different filtering threshold levels

	Dropped pixels (%)	PixHom. time (min)	*Ripser* time (min)
Vanilla	–	7.08 ± 0.04	9.28 ± 0.04
Filter_light	4.19 ± 1.98	7.01 ± 0.04	9.28 ± 0.03
Filter_std	4.68 ± 2.25	6.49 ± 0.04	9.29 ± 0.04
Filter_heavy	5.08 ± 2.48	6.30 ± 0.04	9.29 ± 0.04

For each test conducted on 10 images of the dataset, the median and standard deviation are provided, along with the percentage of background pixels dropped. These times are compared with those of *Ripser*

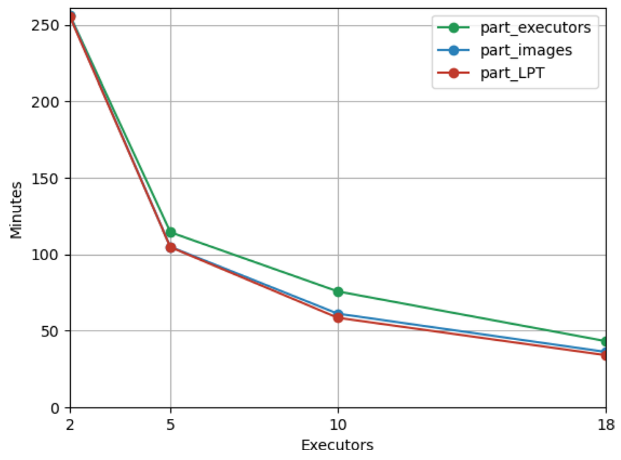

Fig. 6 Comparative analysis of the execution time for analyzing the entire input dataset when using different partitioning strategies and an increasing number of executors

- **filter_light**: More conservative filtering rule: only pixels below 30% of the predefined threshold are classified as background.
- **filter_heavy**: More aggressive filtering rule: all pixels up to 30% above the predefined threshold are classified as background.

The results of this experiment, shown in Table 1, indicate that filtering rules can reduce the execution time of *PixHomology* by up to 10%, without any relevant degradation in the output quality. Instead, this optimization seems to have no significant effect on the performance of traditional PH tools like *Ripser*.

6.3.3 Variant 3: improve the workload distribution

In this experiment, we measured the overall time required to analyze our entire dataset using the three partitioning strategies outlined in Sect. 5.2.1: partitioning by the number of executors (**part_executors**), partitioning by the number of images (**part_images**) and partitioning according to the Longest Processing Time (LPT) rule (**part_LPT**).

We evaluated the performance of these strategies, including the impact of the number of available executors, under several different scenarios. We expected that the impact of these strategies would be more relevant when considering a large number of executors, as the average number of images to be processed per executor would decrease, leading to possible performance bottlenecks due to straggler executors (i.e., executors requiring a much longer time to complete than the other ones, causing delays in the overall execution). For this reason, we varied the number of executors used in our test from 2 to 18, while processing a constant number of images.

As expected, the results in Fig. 6 show that there is little difference between the three partitioning strategies when using a very small number of executors or a very large number of images. In this scenario, all strategies resulted in a very similar workload distribution. However, we observed a relevant change when increasing the number of executors and decreasing their average workload. Under these conditions, the **part_executors** strategy began to underperform, also because of its inability to reassign tasks to balance the workload.

Moreover, the experiments proved the LPT-based strategy to require a lightweight computation (taking at most 20 s) and be slightly faster than the others, although the performance improvement was very small. On closer inspection, we found that the execution cost estimation, which is based on the number of background pixels, can be inaccurate, sometimes leading to suboptimal task assignments in the LPT strategy.

6.3.4 Final configuration

Based on the results of the experiments presented so far, we determined the most effective variants to use for the next experiments. The selected variants are:

- Variant 1: we chose **load_self** for its efficiency when loading input images.
- Variant 2: we chose **filter_std** for its balanced approach to image filtering.
- Variant 3: we chose **part_LPT** thanks to its improved performance in workload distribution.

6.4 Experimental comparative analysis: *Ripser*

As described in Sect. 3, *Ripser* is the current gold standard software for computing PH filtrations across all dimensions. Thus, we used it in our experiments to evaluate both the quality of the output of our algorithm and its performance, when applied to the particular case of zero-dimensional PH computation.

In our first experiment, we performed a qualitative comparison of the outputs from the two algorithms using the *bottleneck distance* [31]. It is an index measuring distances between two persistence diagrams as a minimal matching between them, allowing points to be matched with the diagonal Δ, which is the set of all $(x, x) \in \mathbb{R}^2$.

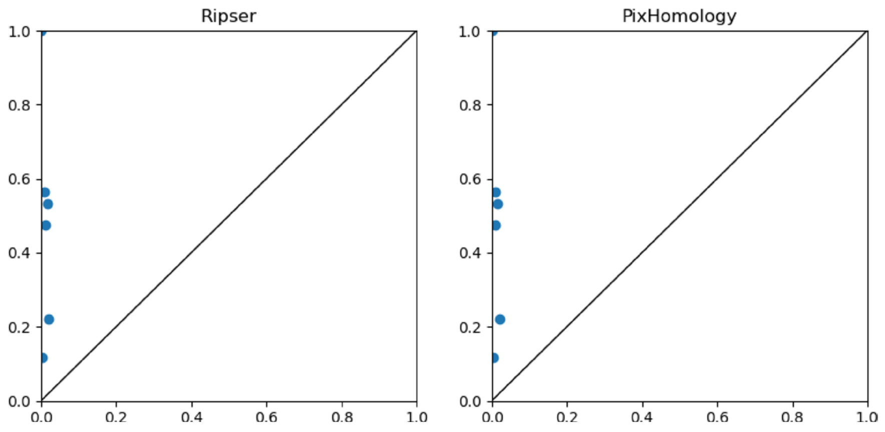

Fig. 7 PD representation of the outputs from *Ripser* and *PixHomology*, when used to process a 50×50 pixels reference image obtained by cropping one image chosen at random from our dataset. The p_{birth} and p_{death} points identified by the two algorithms are consistent

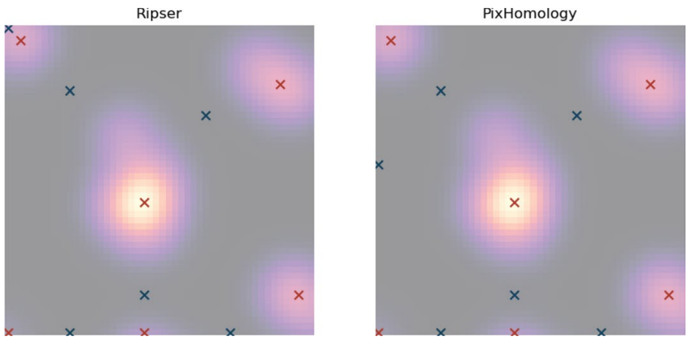

Fig. 8 Pixel representation of the outputs from *Ripser* and *PixHomology* for a 50×50 pixels reference image obtained by cropping one image chosen at random from our dataset. The p_{birth} and p_{death} identified on the filtered image reveal a discrepancy at the point of infinity between the two methods

To this end and for visualization purposes, we selected a 50×50 pixels reference image, obtained by cropping one image chosen at random from our dataset.

As shown in Fig. 7, the PDs returned by the two algorithms are consistent, suggesting that the two methodologies produce similar results. To validate this outcome, we computed the bottleneck distances between the two diagrams, which resulted to be zero.

Figure 8 compares the pixel-based outputs of the two algorithms for the same image. Birth pixels are marked in red and death pixels in blue. The position of one point, specifically the point at infinity, differs between the two outputs. This is because *Ripser* does not return the positions of birth and death points in pixel coordinates, requiring an additional reconstruction step. This limitation can pose

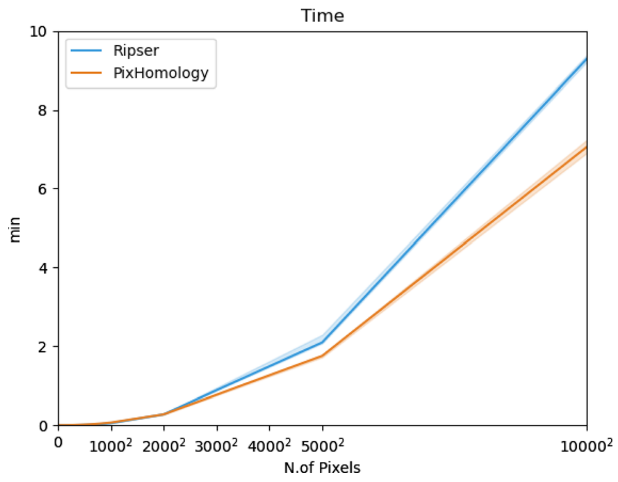

Fig. 9 Execution time comparison between *Ripser* and *PixHomology*, for analyzing the images of our dataset, using a random crop of each image, with a size varying from 20×20 to $10,000 \times 10,000$ pixels. *PixHomology* was executed on a single-core single-executor Spark distributed system

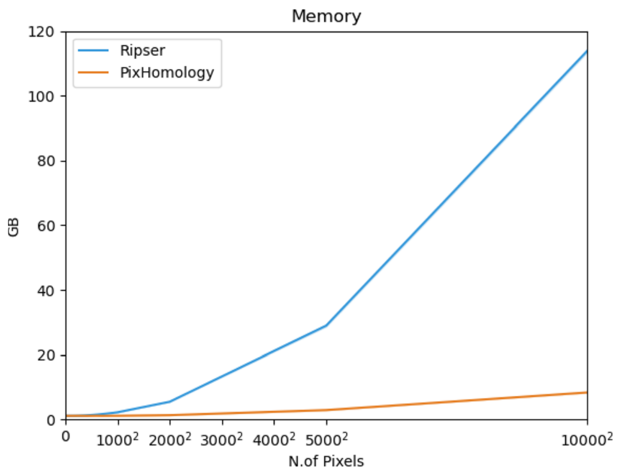

Fig. 10 Maximum memory usage comparison of *Ripser* and *PixHomology* for analyzing the images of our dataset, using a random crop of each image, with a size varying from 20×20 to $10,000 \times 10,000$ pixels. *PixHomology* was executed on a single-core single-executor Spark distributed system

serious identification challenges, especially when multiple pixels share the same value in the image. In contrast, *PixHomology* returns the positions of birth and death points in pixel coordinates, thereby circumventing this issue.

Next, we compare the experimental performance of the two algorithms in terms of total execution time and overall memory usage. To ensure a fair comparison and due to the inability of *Ripser* to take advantage of distributed or

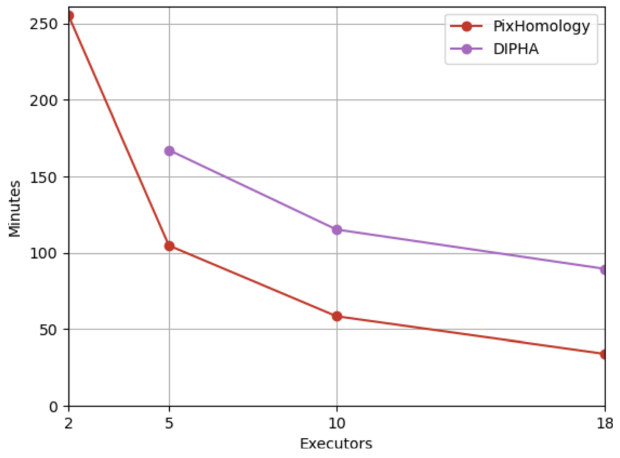

Fig. 11 Execution time comparison between *DIPHA* and *PixHomology*, for analyzing the images of our dataset, using an increasing number of computing units

multicore systems, we executed *PixHomology* using one single executor running on one single core.

We expect *Ripser* to perform very well on small images, while our algorithm is expected to handle better very large images. To assess this expectation, we designed our experiment to evaluate the performance of the two algorithms on our entire dataset, but using a random crop of each image, with a size varying from 20×20 to $10,000 \times 10,000$ pixels.

The resulting execution times are presented in Fig. 9. As expected, *Ripser* outperforms *PixHomology* on small patches. However, as the image size increases to 1000×1000 and beyond, *PixHomology* becomes much faster than *Ripser*. The differences in memory usage between the two algorithms are even more relevant, as shown in Fig. 10. Notably, for $10,000 \times 10,000$ pixel patches, *Ripser* requires approximately 112 GB of memory because of the adjacency matrix computations. In contrast, *PixHomology* only requires around 8 GB.

We remark that the large memory requirements of *Ripser* limit its applicability to the analysis of very large images and prevent running multiple instances of it concurrently on a single machine. Conversely, *PixHomology*'s efficient memory usage makes it a more viable option for large-scale parallel processing on many core systems.

6.5 Experimental comparative analysis: *DIPHA*

We were interested in assessing the performance of *PixHomology* when run as a distributed pipeline, in terms of execution times and scalability. For this purpose, we benchmarked its performance against *DIPHA*, the only existing distributed pipeline designed for large-scale PH computation.

We recall that the strategy adopted by *DIPHA* for the distributed PH calculation implies that each computing unit of the distributed system analyzes a segment of the input image, with the size of this segment being inversely proportional to the total number of computing units. Consequently, when processing very large images using a few computing units, each unit requires a significant amount of memory to run. This was a limiting factor in our experiments, as we were unable to execute *DIPHA* with only two computing units because of the memory required per unit, due to its high memory requirements (approximately 9 GB per unit).

For this reason, the analysis of our entire dataset with *DIPHA* was only possible when using a large number of computing units. In such a setting the results, available in Fig. 11 show that, although both algorithms exhibit a very good scalability, *PixHomology* consistently outperforms *DIPHA* in terms of execution time.

7 Conclusions and future directions

In this paper, we presented *PixHomology*, a novel and efficient algorithm for computing PH on large batches of images. We also presented a distributed version of *PixHomology* able to leverage the Apache Spark computing framework to concurrently process large batches of digital images. In addition, we also considered several variants of our distributed algorithm, for addressing performance bottlenecks identified during preliminary experimental evaluation.

We compared the performance of our algorithm to that of the state-of-the-art algorithm in this field, finding that *PixHomology* is significantly faster and less memory-demanding for processing batches of large digital images. We also compared *PixHomology* to the only other distributed pipeline for PH computation available and again found that our algorithm is more favorable in terms of execution time and memory requirements.

Despite this progress, there is still room for improvement and expansion. Primarily, there is potential to enhance the algorithm's versatility for application across diverse image types. To address this, a possible solution involves introducing controlled noise into the image, mitigating the likelihood of neighboring pixels adopting identical values. Another avenue for exploration is the potential to distribute the analysis of a single image by partitioning the image processing workflow into smaller, more manageable subprocesses. This partitioning strategy could significantly enhance the scalability of our pipeline and further optimize memory utilization. A further problem occurs when the size of input images exceeds the amount of memory available to each executor (see Sect. 5.2.1). Possible solutions to address this problem would be either to partition each images into independent parts to be processed by different executors or develop out-of-core algorithms for this purpose.

Additionally, comprehensive testing and evaluation of the pipeline's performance on other application domains, such as the analysis of diagnostic images, is essential. This would thoroughly assess the versatility and adaptability of the pipeline and ensure that it remains a robust solution across diverse scientific disciplines. Notably, PH encounters challenges when applied to large-scale images, as evident in possible tasks like tumor segmentation from whole slide histology images [32] and astronomical image segmentation [33].

Finally, a research line that has still to be explored but has a potential for giving breakthrough results, is about the possibility to employ machine learning techniques to solve the PH problem. Indeed, such an approach may lead to results that may be, to some extent, less accurate than deterministic solutions but in a fraction of time.

Funding Open access funding provided by Università degli Studi di Roma La Sapienza within the CRUI-CARE Agreement. This work was supported in part by Sapienza University of Rome under project, "Caratterizzazione, sviluppo e sperimentazione di algoritmi efficienti"(nr. RM12117A8720C2A0). It was also supported in part by INdAM—GNCS under project "Approcci computazionali per il supporto alle decisioni nella Medicina di Precisione" (2023), and STILES "Strengthening the Italian leadership in ELT and SKA," financed by the EU and approved by the MUR (Ministerial Decree No. 3264 of December 28th, 2021, "Creation of new research infrastructures") based on the NRRP—NextGenerationEU—Mission 4 "Education and Research", Component 2 "From research to business" - Investment 3.1, "Fund for the realization of an integrated system of research and innovation infrastructures."

References

1. Poger D, Yen L, Braet F (2023) Big data in contemporary electron microscopy: challenges and opportunities in data transfer, compute and management. Histochem Cell Biol 160(3):169–192
2. Large synoptic survey telescope (2023)
3. Starck J, Murtagh F (2007) Astronomical image and data analysis. Astronomy and astrophysics library. Springer, Berlin
4. Edelsbrunner H, Harer J (2010) Computational topology: an introduction. American Mathematical Society, New York
5. Edelsbrunner H, Letscher D, Zomorodian A (2003) Topological persistence and simplification. Discrete & Computational Geometry, 01
6. Carlsson G (2009) Topology and data. Bull Am Math Soc 46:255–308
7. Otter N, Porter M, Tillmann U, Grindrod P, Harrington H (2015) A roadmap for the computation of persistent homology. EPJ Data Sci 6:06
8. Kaczynski T, Mischaikow K, Mrozek M (2004) Computational homology. Applied mathematical sciences. Springer, New York
9. Boissonnat J-D, Chazal F, Yvinec M (2017) Geometric and topological inference. Cambridge University Press, Cambridge (**10**)
10. Munkres J (1984) Elements of algebraic topology, 1st edn. Westview Press, Nashville
11. Cohen-Steiner D, Edelsbrunner H, Harer J (2007) Stability of persistence diagrams. Discrete and Computational Geometry
12. Adams H, Tausz A, Vejdemo-Johansson M (2014) Javaplex: a research software package for persistent (co) homology. In: Mathematical Software–ICMS 2014: 4th International Congress, Seoul, South Korea, August 5–9, 2014. Proceedings 4. Springer, pp 129–136
13. DM. et al. (2012) Dionysus: a software library for topological data analysis

14. De Silva V, Morozov D, Vejdemo-Johansson M (2011) Dualities in persistent (co) homology. Inverse Probl 27(12):124003
15. De Silva V, Vejdemo-Johansson M (2009) Persistent cohomology and circular coordinates. In: Proceedings of the Twenty-Fifth Annual Symposium on Computational Geometry, pp 227–236
16. Bauer U, Kerber M, Reininghaus J, Wagner H (2017) Phat—persistent homology algorithms toolbox. J Symb Comput 78:76–90 (**01**)
17. Maria C, Boissonnat J-D, Glisse M, Yvinec M (2014) The gudhi library: simplicial complexes and persistent homology. In: Mathematical software–ICMS 2014: 4th international congress, Seoul, South Korea, August 5–9, 2014. Proceedings 4. Springer, pp 167–174
18. Bauer U (2021) Ripser: efficient computation of Vietoris–Rips persistence barcodes. J Appl Comput Topol 5(3):391–423
19. Wagner H, Chen C, Vuçini E (2012) Efficient computation of persistent homology for cubical data. Springer Berlin, pp 91–106
20. Kaji S, Sudo T, Ahara K (2020) Cubical ripser: Software for computing persistent homology of image and volume data. arXiv preprint arXiv:2005.12692
21. Wagner H (2023) Slice, simplify and stitch: Topology-preserving simplification scheme for massive voxel data. In: 39th International Symposium on Computational Geometry (SoCG 2023). Schloss Dagstuhl-Leibniz-Zentrum für Informatik
22. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I et al (2010) Spark: cluster computing with working sets. HotCloud 10(10–10):95
23. Jeffrey D, Ghemawat S (2008) "MapReduce: simplified data processing on large clusters." Commun ACM 51.1:107–113
24. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin M, Shenker S, Stoica I (2012) Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, pp 2–2, 04
25. Apache Software Foundation. Apache spark documentation. https://spark.apache.org/docs/latest/. Accessed 2 April 2024 (2024)
26. Graham RL (1969) Bounds on multiprocessing timing anomalies. SIAM J Appl Math 17(2):416–429
27. Amorosi L, Rocco LD, Petrillo UF (2022) Scheduling k-mers counting in a distributed environment. In: Amorosi L, Dell'Olmo P, Lari I (eds) Optimization in artificial intelligence and data sciences. Springer, Cham, pp 73–83
28. Bompiani E, Petrillo U, Lasinio GJ, Palini F (2020) High-performance computing with terastat. In: 2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing (DASC/PiCom/CBDCom/CyberSciTech). Los Alamitos, CA, USA, Aug 2020. IEEE Computer Society, pp 499–506
29. Price-Whelan AM et al (2022) The astropy project: sustaining and growing a community-oriented open-source project and the latest major release (v5.0) of the core package. Astrophys J 935(2):167
30. Craig M (2023) A guide to CCD data reduction and stellar photometry using astropy and affiliated packages. github.io
31. Efrat A, Itai A, Katz M (2001) Geometry helps in bottleneck matching and related problems. Algorithmica 31:1–28
32. Qaiser T, Sirinukunwattana K, Nakane K, Tsang Y-W, Epstein D, Rajpoot N (2016) Persistent homology for fast tumor segmentation in whole slide histology images. Procedia Computer Science, 90:119–124, 2016. 20th Conference on Medical Image Understanding and Analysis (MIUA 2016)
33. Ceccaroni R, Brutti P, Castellano M, Fontana A, Merlin E (2022) Topological persistence for astronomical image segmentation. In: The 51st Scientific Meeting of the Italian Statistical Society, pp 1993–1998