giotto-ph: A Python Library for High-Performance Computation of Persistent Homology of Vietoris–Rips Filtrations

Julián Burella Pérez[a], Sydney Hauke[a], Umberto Lupo[b,*], Matteo Caorsi[c], Alberto Dassatti[a]

[a]*HEIG-VD, HES-SO, Route de Cheseaux 1, Yverdon-les-Bains, Switzerland*
[b]*EPFL, Route Cantonale, Lausanne, Switzerland*
[c]*L2F SA, Rue du centre 9, Saint-Sulpice, Switzerland*

Abstract

We introduce *giotto-ph*, a high-performance, open-source software package for the computation of Vietoris–Rips barcodes. *giotto-ph* is based on Morozov and Nigmetov's lockfree (multicore) implementation of Ulrich Bauer's *Ripser* package. It also contains a re-implementation of Boissonnat and Pritam's "Edge Collapser", implemented so far only in the *GUDHI* library. Our contribution is twofold: on the one hand, we integrate existing state-of-the-art ideas coherently in a single library and provide Python bindings to the C++ code. On the other hand, we increase parallelization opportunities and improve overall performance by adopting higher performance data structures. The final implementation of our persistent homology backend establishes a new state of the art, surpassing even GPU-accelerated implementations such as *Ripser++* when using as few as 5–10 CPU cores. Furthermore, our implementation of the edge collapser algorithm has reduced dependencies and significantly improved run-times.

Keywords: persistent homology, Vietoris–Rips, concurrency, simplicial collapses

1. Introduction

In recent years, *persistent homology* (PH) (see e.g. [24, 19, 21, 34, 11, 35, 9, 31] for surveys) has been a key driving force behind the ever-increasing adoption of topological approaches in a wide variety of computational contexts, such as geometric inference [18, 6], signal processing [40, 36], data visualization [43], and, more generally, data analysis [8, 12] and machine learning [26].[2] Among the main invariants described by this theory,

*Corresponding author
Email addresses: julian.burellaperez@heig-vd.ch (Julián Burella Pérez), sydney.hauke@heig-vd.ch (Sydney Hauke), umberto.lupo@epfl.ch (Umberto Lupo), m.caorsi@l2f.ch (Matteo Caorsi), alberto.dassatti@heig-vd.ch (Alberto Dassatti)

[1]The first two authors contributed equally to this work.

[2]As a matter of fact, insights from the theory of persistent homology and *persistence modules* have also proved helpful in pure mathematics, e.g., in symplectic topology/geometry [37].

the (*persistence*) *barcode* [22, 23, 4, 39, 20, 47] has attracted the most attention due to a) its ability to track the appearance and disappearance of topological features in data throughout entire ranges of parameters, b) its succinct nature and ease of representation, as it simply consists of a (typically small) collection of intervals of the real line, c) its provable robustness under perturbations of the input data [17, 14], and d) its amenability to computation and algorithmic optimization, as demonstrated by the large number of existing implementations – see Sect. 1 in [5] for a review, and [1, 45] for recent entries not mentioned there.

Despite these successes, the computation of barcodes remains a challenge when dealing with large data sets and/or with high-dimensional[3] topological features. We now explain why this is the case. The input to any barcode computation is a growing, one-parameter family of combinatorial objects, called a *filtration* or a *filtered complex*. Filtrations consist of *cells* with assigned integer *dimensions* and values of the filtration-defining parameter, as well as *boundary* (resp. *co-boundary*) relations (mappings) between k-dimensional cells and $(k-1)$-dimensional [resp. $(k+1)$-dimensional] ones.[4] Arguably, the most common examples of filtrations in applications concern *simplicial* complexes, in which case the cells are referred to as *simplices*, and k-dimensional simplices consist of sets of $k+1$ points from a common vertex set V – for instance, a 0-simplex $\{v\}$ is one of the vertices $v \in V$, while a 1-simplex $\{v, w\} \subseteq V$ can be thought of as an edge connecting vertices v and w. These are the filtrations of interest in the present paper. In particular, we focus on the *Vietoris–Rips* (VR) (resp. *flag*) filtration of a finite metric space (resp. undirected graph with vertex and edge weights), in which simplices are arbitrary subsets of the available points (resp. vertices) and their filtration values are set to be their diameters (resp. the maximum weights of all vertices and edges they contain, with absent edges being given infinite filtration value).

Several simplicial filtrations of interest in applications, and the VR filtration chiefly among these, quickly become very large as their defining parameter increases (and hence more and more simplices are included in the growing complex). At the heart of all algorithms for computing PH barcodes lies the reduction of *boundary* or *co-boundary matrices* indexed by the full set of simplices in the filtration; the available reduction algorithms have asymptotic space and time complexities which are polynomial in the total number N of simplices. In the case of the VR filtration of an input metric space $\mathcal{M}$, if one is interested in computing the barcode up to and including homology dimension D, then $N = \sum_{k=0}^{D+1} \binom{|M|}{k}$. For sizeable data sets, this combinatorial explosion leads to a staggering number of elementary row or column operations (as well as memory) required to distil the desired barcode. PH computation for many other simplicial filtrations constructed from point clouds, finite metric spaces, or graphs are also ultimately limited by similar considerations.

1.1. Related work

To the best of our knowledge, at the time of writing, *Ripser*[5] [5] is the state-of-the-art and reference for computing VR persistence barcodes on CPUs. *Ripser* uses multiple

[3]In this paper, we use the word "dimension" as a synonym for "degree" when describing homology classes and therefore bars in a persistence barcode.

[4]We refer the reader to any of the aforementioned surveys of PH for rigorous definitions.

[5]`https://github.com/Ripser/ripser`

known optimizations like *clearing* [13] and *cohomology* [16]. Furthermore, it makes use of other performance-oriented ideas, such as the implicit representation of the (co)boundary and reduced (co)boundary matrices, and the *emergent/apparent pairs* optimizations (we refer to [5] for definitions and details). At the time of writing, the latest version of *Ripser* is v1.2[6]. In that version, to the emergent pairs optimization in use until that point was added an optimization based on apparent pairs.

Although *Ripser v1.2* is arguably the fastest existing code for computing VR barcodes in a sequential (i.e., single CPU core) setting, it has no parallel capabilities. Overcoming this limitation is possible as two recent lines of work [30, 46] demonstrate. Based on a "pairing uniqueness lemma" proved by Cohen-Steiner *et al.* [15], Morozov and Nigmetov [30] observe that the reduction of the (co)boundary matrix can, in fact, be performed out of order as long as column additions are always performed left to right[7]. Therefore – these authors suggest – any column reduction can be efficiently performed in parallel provided adequate synchronisation is used. A functional proof of concept of this idea as applied to a now superseded version of *Ripser* was put in the public domain in June 2020 [29], but has not led to a distributable software package. A generic implementation of the ideas in [30], not tied to Vietoris–Rips filtrations and instead designed to make lock-free reduction possible on any (co)boundary matrix, has recently been published as the *Oineus* library[8] [32]. Although optimizations such as clearing and implicit matrix reduction appear to have been implemented there, the code and performance are not optimized for Vietoris–Rips filtrations, and in particular no implementation of the emergent *or* apparent pair optimization is present there at this time.[9]

Zhang *et al.*'s *Ripser++* [46] implements the idea of finding apparent pairs in parallel on a GPU to accelerate the computation of VR barcodes. Despite this, *Ripser++* is not fully parallel. For each dimension to process, it divides the computation into three sub-tasks: "*filtration construction and clearing*", "*finding apparent pairs*" and "*sub-matrix reduction*". The latter step is not parallel and it is executed on the CPU. Accordingly to Amdahl's law[10], this processing sequence is expected to yield only diminishing returns when augmenting the number of parallel resources. Although performance gains have been demonstrated in [46] – particularly when using high-end GPUs – there is room for extending parallelism to the third sub-task above, which we try to harvest in this work by integrating the aforementioned ideas from [30].

All implementations presented in this subsection (barring [32], which also provides some Python bindings) are written in low-level languages (C++, CUDA). However, researchers today make wide use of higher-level languages – Python, for instance, is the dominant one in several fields. It is therefore natural that libraries have been developed to provide the ease of use of Python coupled with these high-speed codes. *Ripser.py* [44] is probably the most notable of these implementations, providing an intuitive interface for VR filtrations wrapping *Ripser* at its core. The authors of the library forked the original *Ripser* implementation and added support for non-zero birth times, as well as the possibility to compute and retrieve cocycles.

[6]Release date: 25 February 2021.

[7]"Left to right" here is meant relative to the filtration order.

[8]At the time of writing, this library is in version 1.0.

[9]We were not aware of [32] – and thus did not use its implementations – during the development of our code.

[10]`https://en.wikipedia.org/wiki/Amdahl's_law`

Meanwhile, in 2020, Boissonnat and Pritam presented a new algorithm they called *Edge Collapser* (EC) [7]. Independently of the code used to compute barcodes, EC can be used as a pre-processing step on any flag filtration to remove "redundant" edges – and modify the filtration values of some others – while ensuring that the flag filtration obtained from the thus "sparsified" weighted graph has the same barcode as the original filtration. An implementation of EC has already been integrated into the *GUDHI* [42] library.

1.2. Our contribution

In this context, we present *giotto-ph*[11], a Python package built on top of a C++ backend that computes PH barcodes for VR filtrations on the CPU. To the best of our knowledge, this is the first package integrating the three state-of-the-art ideas described in Section 1.1 in a single portable, easy-to-use library. At the same time, we focused on increasing execution speeds throughout. In particular:

1. We built on the ideas for parallel reduction presented in [30] and on the prototype implementation described in [29], and improved execution speed and resource usage by implementing custom lock-free hash tables and a thread pool.
2. Similarly to *Ripser++*, we implemented a parallel version of the apparent pairs optimization, thus far only present in serial form in Ripser v1.2.
3. We re-implemented the EC algorithm to increase its execution speed compared to [38]. The simple observation that the well-known (minimum) *enclosing radius* optimization is applicable to EC is shown here to lead to even larger improvements.

Our results show that our code is often 1.5 to 2 times (and, in one example, almost 8 times) faster than [30] and able to beat *Ripser++* [46], the current state-of-the-art GPU implementation, while running only on CPU and with as few as 5–10 cores.

Finally, *giotto-ph* owes some architectural decisions to *Ripser.py* [44] – in particular, the support for node weights. At the level of the Python interface, our main contribution is supporting *weighted Rips filtrations* – in particular, the *distance-to-measure*–based filtrations described in [2].

Thanks to reduced memory usage and shorter run times, *giotto-ph* enables exploring data sets in dimensions higher than ever before.

1.3. Structure of the paper

In Section 2 we describe the package implementation, detailing optimizations adopted to increase speed and portability. In Section 3, we show how our library compares to other implementations. We also report performance when increasing PH dimension, observing that in that regime we are shifting the limiting factor from memory towards the enumeration of simplices, currently limited to 2^{64}. Finally, in Section 4 we draw conclusions and sketch future research directions.

[11] *giotto-ph* is available at `https://github.com/giotto-ai/giotto-ph`.

2. Implementation

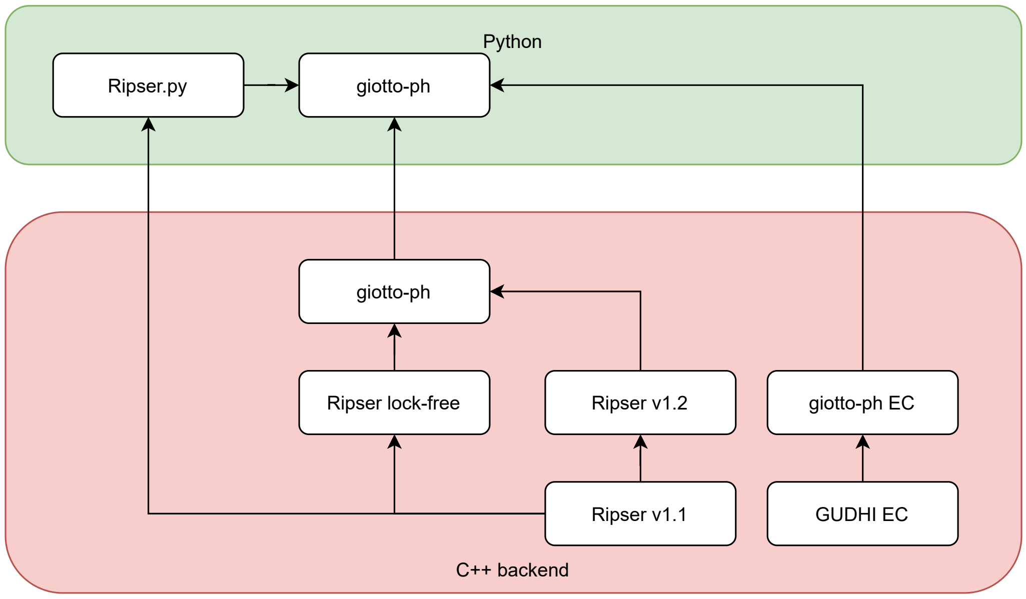

Figure 1: *giotto-ph* consists of a C++ backend and a Python frontend. The Python interface is based on *Ripser.py* [44] (see section 2.2 for details). The figure also shows the inheritance of *giotto-ph*'s C++ backend from pre-dating implementations.

giotto-ph is a library dedicated to the efficient computation of PH of VR filtrations (see Section 1). It inherits and extends ideas and code from many sources; Figure 1 gives a visual representation of the most important ones among them. Our aim with *giotto-ph* is to provide an alternative to the excellent *Ripser.py* library[12], while retaining its main advantages, namely portability and ease of use, while at the same time providing a new parallel and higher performance C++ backend.

2.1. C++ backend

The implementation of *giotto-ph*'s backend parallel algorithm is inspired by [29], a functional proof of concept of [30]. Starting from [30], we replaced the main data structure and the threading strategy to minimize the overhead introduced by adding parallelism. Furthermore, we introduced the apparent pairs approach, in its parallel form, to harvest its benefits in shortening run-times: a decreased number of columns to reduce [5] and an additional early stop condition when enumerating cofacets of a simplex.

The main data structure of the algorithm described in [30] is a lock-free hash table. A lock-free hash table is a concurrent hash map where concurrent operations do not make use of synchronization mechanisms such as mutexes. Instead, a lock-free hash table relies

[12]Currently, *giotto-ph* does not support retrieving cocycles, but there are no substantial challenges, in principle, to the future addition of this feature.

on atomic operations for manipulating its content; in particular, insertion is carried out by a mechanism called compare-and-swap (CAS). After benchmarking many alternatives, we created a custom hash map adapted to the needs of the core matrix reduction algorithm, using the Leapfrog implementation provided by the *Junction* library.[13]

As previously mentioned, we also adopted a different threading strategy: a *thread pool*[14]. A thread pool is a design pattern in which a "pool" of threads is created up front when the program starts, and the same threads are reused for different computations during the program's life span. This approach enables better amortization of the cost of the short-lived threads used in [29], where one thread is created whenever needed and destroyed at the end of its computation task. Table 1 compares the running time of a solution based on our thread pool with the former approach. The run-time improvements are highly data set dependent, but always measurable in the considered scenarios.

The final component in our C++ backend is a rewriting of the EC algorithm (see Section 1.1). Our new implementation focuses on performance and removes the dependencies on the *Boost* [3] and *Eigen* [25] libraries, present in the original implementation [38]. *giotto-ph*'s EC is more than 1.5 times faster than the original version as reported in Table 4. It also supports weighted graphs with arbitrary (possibly non-positive) edge weights as well as arbitrary node weights. Improvements were achieved mainly by reworking data structures, making the implementation more cache-friendly, and directly iterating over data without any transformation, hence reducing the pressure on the memory sub-system.

	giotto-ph backend			
	no thread pool		**thread pool**	
data set	**N=8**	**N=48**	**N=8**	**N=48**
sphere3	0.4	0.4	0.4	0.38
dragon	1.2	1.2	1.3	1.3
o3_1024	0.4	0.18	0.4	0.17
random16	0.9	0.4	0.9	0.24
fractal	0.9	0.35	0.9	0.34
o3_4096	6.9	2.7	6.9	2.6
torus4	19	14.7	19.1	14.3

Table 1: Running times, expressed in seconds, with and without the thread pool. All information regarding the data sets presented here are described in section 3 and summarized in Table 2.

2.2. Python Interface

Our Python interface is based on *Ripser.py* [44]. While it lacks some of *Ripser.py* features, such as the support for "greedy permutations" or for retrieving cocycles, it introduces the following notable improvements.

Support for Edge Collapser EC is disabled by default because it is expected, and empirically confirmed, that, unless the data is large and/or the maximum homology

[13]`https://github.com/preshing/junction`

[14]With optional CPU pinning option.

dimension to compute is high, preprocessing using EC can introduce a run-time overhead. However, users can easily enable it by means of the `collapse_edges` optional argument. In Table 3 we show the difference in run-times when this option is active. See also "Support for enclosing radius", below, and Table 4.

Support for enclosing radius The *(minimum) enclosing radius* of a finite metric space is the radius of the smallest enclosing ball of that space. Its computation, starting from a distance matrix, is trivial to implement and takes negligible run-time on modern CPUs. Above this filtration value, the Vietoris–Rips complex becomes a cone, and hence all homology groups are trivial. Hence, simplices with higher filtration values than the enclosing radius can be safely omitted from the enumeration and matrix reduction steps, without changing the final barcode. When the enclosing radius is considerably smaller than the maximum distance in the data, this can lead to dramatic improvements in run-time and memory usage, as observed in [27].[15] Unless the user specifies a threshold, both *Eirene* [28] and *Ripser* make use of the enclosing radius optimization, and the same is true in *giotto-ph*, where the enclosing radius computation is implemented in Python using highly-optimized *numpy* functions. An element of novelty in our interface is that, when both the enclosing radius is computed and EC is enabled, the input distance matrix/weighted graph is thresholded *before* being passed to the EC backend. As we experimentally find and report in Section 3.3, on several data sets this can lead to very high gains in the computational run-time of the EC step.

Weighted VR filtrations While standard stability results for VR barcodes [14, 10] guarantee robustness to small perturbations in the data, VR barcodes are generally *unstable* with respect to the insertion or deletion of even a single data point. Thus, even relatively small changes in the local density can greatly affect the resulting barcodes, rendering the vanilla VR persistence pipeline very vulnerable to statistical outliers. Distance-To-Measure (DTM) [2] based filtrations address this issue by reweighting vertices and distances according to the local neighbourhood structure. The user can toggle DTM-based reweighting (or more general reweightings) by appropriately setting the optional parameters `weights` and `weight_params`.

pybind11[16] **bindings** We added support for and used *pybind11* instead of *Cython*[17] for creating Python bindings. From our experience, it is easier to use without compromising performance. Furthermore, it is already used for the bindings in the *giotto-tda* library [41], our sibling project. The presence of Python bindings as well as the portability on different operating systems, namely Linux, Mac OS X, and Windows, have been two of our core objectives to facilitate the adoption of our library.

[15]For instance, the barcode computation for the `random16` data set (see Table 2) up to dimension 7 would not be completed after two hours without the enclosing radius optimization; with it, the run-time drops to seconds. Not all data sets can be expected to witness equally impressive improvements, but the cost of computing the enclosing radius is trivial compared to the computation of PH.

[16]`https://github.com/pybind/pybind11`

[17]`https://cython.org/`

3. Experimental results

All experiments presented in this paper were performed on a machine running Linux CentOS 7.9.2009 with kernel 5.4.92 equipped with two Intel XEON Gold 6248R (24 physical cores each) and a total of 128 GB of RAM.

We present measures on the data sets of Table 2 because they are publicly available, and they are used in publications [33, 5] describing established algorithms, making them a representative benchmark set and facilitating comparisons among competing solutions. All data sets are stored as point clouds. When the `threshold` parameter is empty, the tests report performances computed with the enclosing radius option active. The `dim` parameter corresponds to the maximum dimension for which we compute PH, and the `coeff` parameter corresponds to the prime modulus field (in our cases, the field will always be $\mathbb{F}_2$).

data set	**size**	**threshold**	**dim**	**coeff**
sphere3	192		2	2
dragon	2000		1	2
o3_1024	1024	1.8	3	2
random16	50		7	2
fractal	512		2	2
o3_4096	4096	1.4	3	2
torus4	50000	0.15	2	2

Table 2: Data sets used for the different bechmarking. The `size` represent the number of points in the data set, `threshold` the maximal diameter threshold (when empty an infinite value is assumed). `dim` indicates the maximal homology dimension computed.

3.1. Comparison with SOTA algorithms

In this section we compare our implementation with other implementations of PH using VR filtration that use an approach similar to Ripser. We do not compare with other existing libraries that can compute PH for VR filtrations, like *GUDHI* [42] and *Eirene* [28] because from [5] it is evident *Ripser* is always faster.

Figure 2 compares the `giotto-ph` backend and *Ripser v1.2*. Where the computation of the filtration is very fast, due to the reduced number of points or the low dimension of the computation, there is marginal or no benefit in adopting our parallel approach.

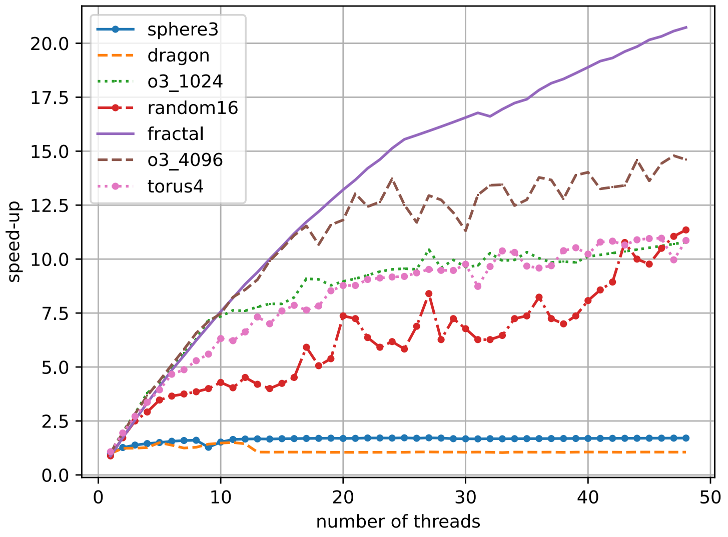

Figure 2: Speed-up of *giotto-ph* compared to *Ripser v1.2*. *giotto-ph* is always faster than *Ripser v1.2*, the only exception being the `dragon` data set. In this case we compute homology up to dimension 1, and the cost of setting up the parallel element of the library is non zero.

Figure 3 shows the scaling of *giotto-ph* when increasing the number of worker threads. Scaling is different for each data set due to the variable number of apparent and emergent pairs as well as the dimension parameter used. Observe that the larger the number of points in the data set, the better the scaling. Similar effect is visible with higher maximal homology dimension to compute.

As stated for run-times, scaling on small data sets combined with small persistent homology dimension (`sphere3` 192 points in dimension 2 or `dragon` 2000 points in dimension 1) is limited.

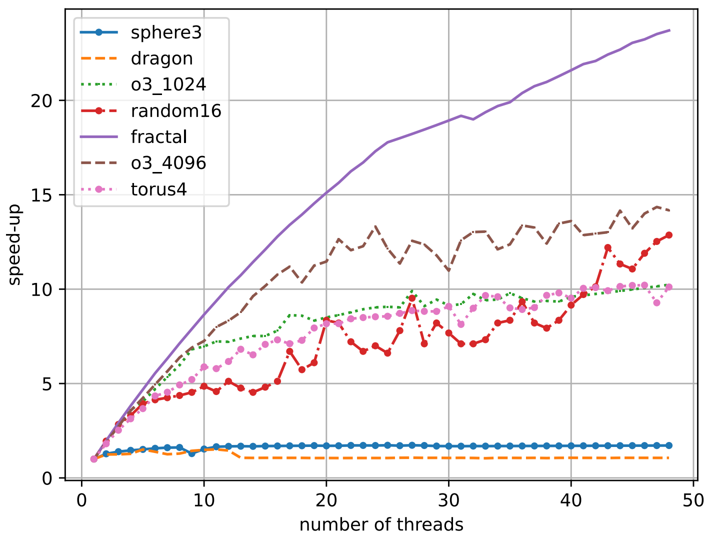

Figure 3: Scaling of *giotto-ph* when increasing the number of threads. This figure is similar to Figure 2 because in a single thread configuration *giotto-ph* performs very similarly to *Ripser v1.2*.

According to our measurements, our implementation outperforms most of the time [29] (Figure 4) when the number of parallel resources increases. The only exception is `dragon`. Comparing the two versions of [5], we noticed the same effect can be seen there. We concluded that the performance drop that `dragon` exhibits is due to the introduction of apparent pairs. It must be highlighted here that this effect is due to the low dimension (1) of the computation: indeed, introducing apparent pairs leads to a performance loss caused by the column assembly step that is done serially for dimension 0 (whereas in parallel for higher dimensions).

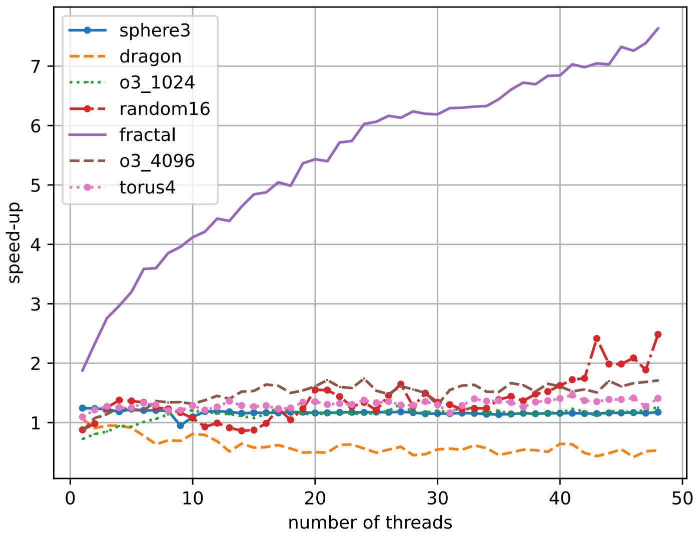

Figure 4: Speed-up of *giotto-ph* compared to the implementation in [29]. *giotto-ph* is slightly faster in general, but with `fractal` data set the speed up is larger, increased by a factor of 8. This phenomenon is explained by the large number of apparent pairs in this specific data set. On the other hand, performance on the `dragon` data set is worse because the homology dimension is only 1 and computing apparent pairs slows down the computation of dimension 0.

Considering the good performance obtained, we decided to compare our implementation with the state-of-the-art parallel code running on GPU: *Ripser++* [46]. For this test, we ran our code on the same data sets used in [46] (for full details check Table 2 at page 23 of [46]) and compared our run-times with the reported figures. Figure 5 shows that on our test machine, we achieve better performance when using only 4 to 10 threads, depending on the data set, confirming that a relatively new CPU with at least 8 cores should be able to beat a high end GPU on this computation.

There are multiple limitations in Ripser++ that were addressed in *giotto-ph*. First, Ripser++ does not perform the matrix reduction in parallel. Second, apparent pairs are stored in a sorted array in order to provide apparent pair lookups in $\mathcal{O}(\log n)$ time using binary searches. Since it is possible to carry out the matrix reduction without recording and/or sorting apparent pairs, *giotto-ph* results in a competitive solution, even if running on less high-performance hardware.

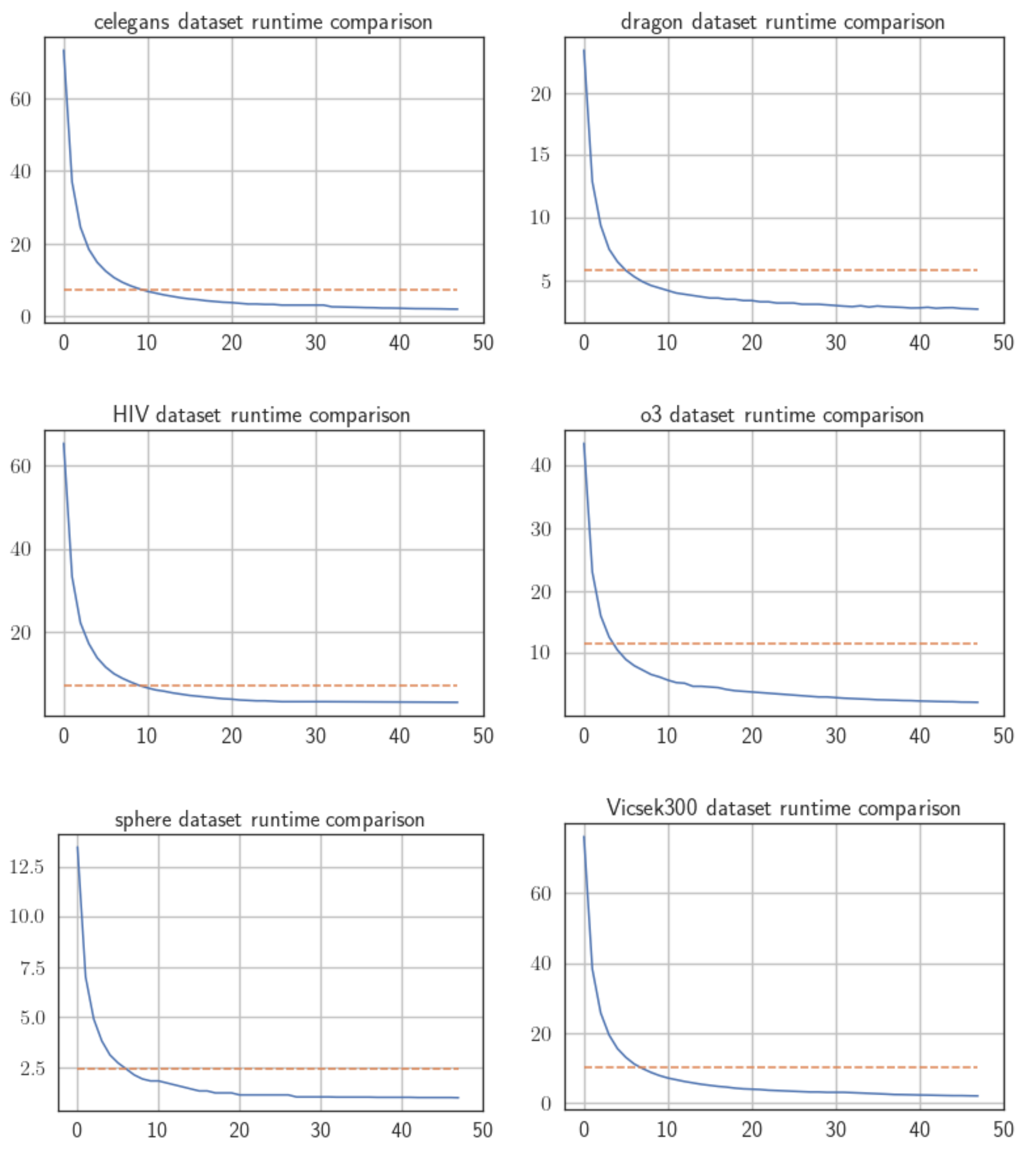

Figure 5: Run times comparison of *giotto-ph* (full blue line) and *Ripser++* (dashed orange line) using data sets from [46]. The x-axis represent the number of threads used and the y-axis the time (in seconds) to complete the PH computation.

Figure 6 compares the memory consumption of *giotto-ph* and [5]. Considering these numbers are quite remarkable, we investigated the source of these differences and discovered that [5] missed an optimization when the required dimension is greater than 2. We contributed with a pull request[18] to [5] that, once accepted, will make the memory

[18]`https://github.com/Ripser/ripser/pull/37`

benefit of our solution vanish.

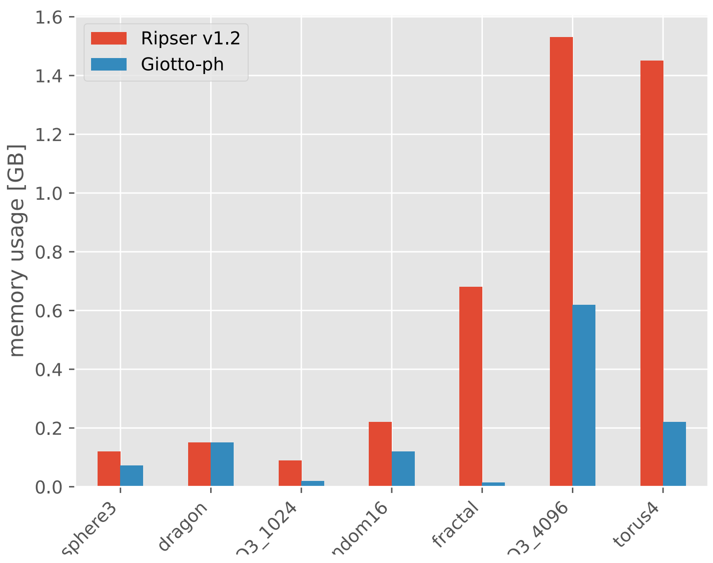

Figure 6: Memory consumption of *giotto-ph* and *Ripser v1.2*. For the `dragon` data set, the memory consumption is exactly the same because the memory optimization only kicks in from dimension 2.

3.2. Higher homology dimensions

Table 3 compares *Ripser* and *giotto-ph* when increasing the homology dimension parameter. We included the measurements using EC to show the potential benefits. It is important to note that timings reported using EC do not include EC processing time; the interested reader can find them in Table 4. The first dimension reported in Table 3 is the one in the Table 2 setup.

`sphere3` and `random16` are the only data sets where **MI**[19] is not attained. `sphere3` is a highly-regular data set and computing higher dimensions will not produce any interesting result. `random16` produces no barcodes also at dimension 20. We arbitrarily decided to stop at dimension 10 and report the data.

Using EC on the `sphere3` data set, as reported in Table 3, is a special case. On this data set, EC is seen to be slightly detrimental to run-times. The reason for this is implementational in nature. Indeed, the EC step outputs a sparse matrix, but, for `sphere3`, EC removes very few edges, producing a highly-filled sparse matrix. In the subsequent processing stage, when the input is sparse, the data points are stored in a sparse matrix, whereas when the data input is a distance matrix, it will have a dense representation. The dense representation has better cache behaviour and thus can compute faster than

[19]**MI** stands for Maximal Index and it corresponds to the maximum number of retrievable entries in a data structure.

`sphere3`	2	3	4	5
ripser	0.7	11.6	392	**OOM**
ripser after EC	0.8	18.6	658	**OOM**
giotto-ph	0.4	2.2	63	1826
giotto-ph after EC	0.4	3.3	97	2900

`o3_1024`	3	4	5	6
ripser	2.3	9.9	34.2	**MI**
ripser after EC	0.2	0.3	0.4	
giotto-ph	0.5	2.2	7.9	
giotto-ph after EC	0.1	0.1	0.1	

`o3_4096`			
data set	**3**	**4**	**5**
ripser	45.6	334	**MI**
ripser after EC	1.9	3.6	
giotto-ph	9.3	69	
giotto-ph after EC	0.5	0.9	

`random16`	7	8	9	10
ripser	3.9	7.9	13.6	20.3
ripser after EC	0.1	0.1	0.1	0.1
giotto-ph	1	2.2	3.9	5.9
giotto-ph after EC	0.1	0.1	0.1	0.1

`fractal`	2	3	4	5	6	7	8
ripser	5.4	**OOM**					**MI**
ripser after EC	0.1	0.2	1	3.4	10	23.5	
giotto-ph	0.9	90	9100	**OOM**			
giotto-ph after EC	0.1	0.1	0.2	0.9	2.5	6	

`dragon`	1	2	3	4	5
ripser	2.2	168	**OOM**		**MI**
ripser after EC	0.1	0.9	8.6	85	
giotto-ph	1.4	40	6508	**OOM**	
giotto-ph after EC	0.1	0.3	1.8	18	

Table 3: These tables shows the timing in seconds of *Ripser v1.2* and *giotto-ph* when increasing the maximal homology dimension to compute. *giotto-ph* was run using 8 threads. For both software, we included the run-time when using the data produced by EC, but the timings do not include EC computation time. **OOM** stands for out of memory while **MI** is the Maximal Index. The `torus4` data set is not present here because MI is reached at $dim = 3$.

a badly-used sparse one. We are working on an heuristic to automatically select the best data format.

3.3. Edge Collapser

We now report experimental findings concerning our EC implementation, summarized in Table 4. We remind the reader that, as explained in Section 2.2, a novelty of our implementation is the use of the enclosing radius computation to shorten the run-time of the EC step even beyond what is already made possible by our use of faster routines and data structures.

We now comment on our use of the enclosing radius optimization on EC. One would expect that the more "random" data sets, where "central points" are likely to be present, will benefit the most from thresholding by the enclosing radius. Among our standard data sets from Table 2, `random16`, `o3_1024` and `o3_4096` are random data sets, but we do not witness such an impact. While, in the case of `random16`, the reason is likely that the data set it too small (50 points), in the case of the `o3` data sets the reason is that a threshold lower than the enclosing radius is provided, meaning that the enclosing radius optimization is not used at all there. To demonstrate that our expectation is valid despite the limitations caused by our choice of data sets and configurations, we have added a last entry in Table 2, representing a data set of $N = 3000$ points sampled from the uniform distribution on the unit cube in $\mathbb{R}^3$. Here, it can be seen that large gains can be made by using the enclosing radius on certain data sets.

data set[20]	***GUDHI* EC [38]**	***giotto-ph* EC**	***giotto-ph* EC with encl. rad.**	**Speed-up**
`sphere3`	1.6	0.9	0.9	1.78
`dragon`	63	36	28	1.75
`o3_1024`	0.2	0.13	0.13*	1.53
`random16`	0.004	0.001	0.001	4.00
`fractal`	1.32	0.8	0.8	1.65
`o3_4096`	2.1	1.2	1.2*	1.75
`torus4`	10	6.7	6.7*	1.49
3000 points in unit cube	180	125	78	1.44

Table 4: Execution times in seconds across different implementations of the EC algorithm of Boissonnat and Pritam [7]. The second-to-last column reports run-times when sparsifying by the enclosing radius before calling *giotto-ph*'s EC. The last column refers to the speed-up of our own implementation, with the enclosing radius optimization disabled, relative to *GUDHI*'s implementation of EC [38]. Cells marked with an asterisk mean that a threshold is provided and therefore the enclosing radius is not computed by default. The last entry is unique to this table and better demonstrates the impact of the enclosing radius optimization on favourable data sets and configurations.

3.4. Low-end CPU

Not all researchers have access to high-end machines. In this section we report performance figures when using a low-mid end CPU, showing that similar results can be achieved. The CPU used in this test is an Intel(R) Core(TM) i7-7700 CPU with 4 physical cores. Figure 7 shows the speed-up of *giotto-ph*. This plot can be compared to Figure 2: performances are very similar also on a less high-performing hardware platform.

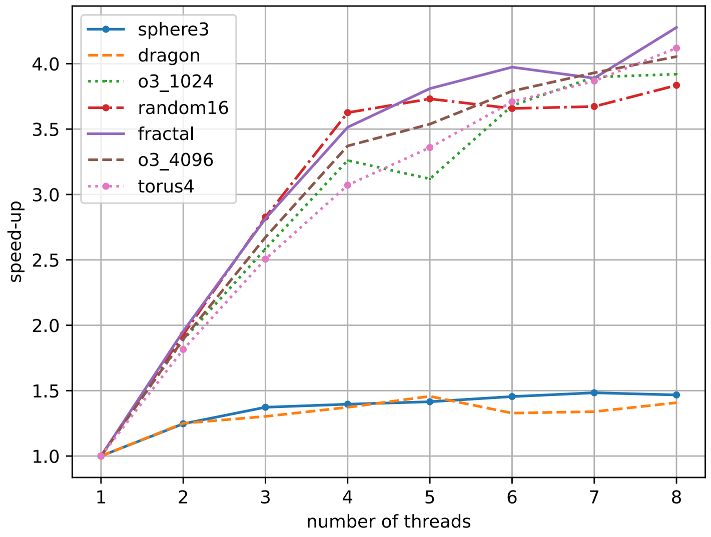

Figure 7: Speed-up of *giotto-ph* on a low-end CPU. *giotto-ph* uses 8 threads fully exploiting the hardware resources: 4 physical cores and 8 logical threads ("hyper-threading").

4. Conclusion and future work

We integrated multiple, existing and novel, algorithmic ideas to obtain a state-of-the-art implementation of the computation of persistent homology for Vietoris–Rips filtrations. This implementation enables the use of parallel CPU resources to speed up the computation and outperforms even state-of-the-art GPU implementations.

We plan to extend *giotto-ph* by supporting a wider range of filtrations in a modular way. We also plan to add features (e.g., simplex pairings) needed for back-propagation in a deep learning context, and seamless integration with frameworks such as *Pytorch*.[21]

Acknowledgements

The authors would like to thank Kathryn Hess Bellwald for numerous fruitful discussions as well as Ulrich Bauer for very helpful conversations about *Ripser*. This work was supported by the Swiss Innovation Agency (Innosuisse project 41665.1 IP-ICT).

References

[1] Aggarwal, M., and Periwal, V. Dory: Overcoming Barriers to Computing Persistent Homology. *arXiv:2103.05608* (2021).

[21] `https://pytorch.org/`

[2] Anai, H., Chazal, F., Glisse, M., Ike, Y., Inakoshi, H., Tinarrage, R., and Umeda, Y. DTM-based Filtrations. *arXiv:1811.04757* (2020).

[3] authors, T. B. Boost C++ Libraries. `http://www.boost.org/`, 2015.

[4] Barannikov, S. A. The framed Morse complex and its invariants. In *Singularities and bifurcations*, vol. 21 of *Adv. Soviet Math.* Amer. Math. Soc., Providence, RI, 1994, pp. 93–115.

[5] Bauer, U. Ripser: efficient computation of Vietoris–Rips persistence barcodes. *J Appl. and Comput. Topology* (2021).

[6] Boissonnat, J.-D., Chazal, F., and Yvinec, M. *Geometric and Topological Inference.* Cambridge Texts in Applied Mathematics. Cambridge University Press, 2018.

[7] Boissonnat, J.-D., and Pritam, S. Edge Collapse and Persistence of Flag Complexes. In *36th International Symposium on Computational Geometry (SoCG 2020)* (Dagstuhl, Germany, 2020), S. Cabello and D. Z. Chen, Eds., vol. 164 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 19:1–19:15.

[8] Carlsson, G. Topology and data. *Bull. Amer. Math. Soc. 46* (2009), 255–308.

[9] Carlsson, G. *Persistent homology and applied homotopy theory.* CRC Press, 2019, ch. 8, pp. 297–330.

[10] Chazal, F., Cohen-Steiner, D., Glisse, M., Guibas, L. J., and Oudot, S. Y. Proximity of persistence modules and their diagrams. In *Proceedings of the Twenty-Fifth Annual Symposium on Computational Geometry* (New York, NY, USA, 2009), SCG '09, Association for Computing Machinery, pp. 237–246.

[11] Chazal, F., de Silva, V., Glisse, M., and Oudot, S. Y. *The Structure and Stability of Persistence Modules.* SpringerBriefs in Mathematics. Springer, 2016.

[12] Chazal, F., and Michel, B. An introduction to Topological Data Analysis: fundamental and practical aspects for data scientists. *arXiv:1710.04019* (2021).

[13] Chen, C., and Kerber, M. Persistent homology computation with a twist. In *27th European Workshop on Computational Geometry (EuroCG)* (2011), pp. 197–200.

[14] Cohen-Steiner, D., Edelsbrunner, H., and Harer, J. Stability of persistence diagrams. *Discrete Comput. Geom. 37* (2007), 103–120.

[15] Cohen-Steiner, D., Edelsbrunner, H., and Morozov, D. Vines and vineyards by updating persistence in linear time. In *Proceedings of the Twenty-Second Annual Symposium on Computational Geometry* (New York, NY, USA, 2006), SCG '06, Association for Computing Machinery, pp. 119–126.

[16] de Silva, V., Morozov, D., and Vejdemo-Johansson, M. Dualities in persistent (co)homology. *Inverse Problems 27*, 12 (Nov 2011), 124003.

[17] d'Amico, M., Frosini, P., and Landi, C. Optimal matching between reduced size functions. *DISMI, Univ. di Modena e Reggio Emilia, Italy, Technical report*, 35 (2003).

[18] Edelsbrunner, H. *A Short Course in Computational Geometry and Topology.* SpringerBriefs in Mathematical Methods. Springer, 2014.

[19] Edelsbrunner, H., and Harer, J. Persistent homology—a survey. In *Surveys on discrete and computational geometry* (2008), vol. 453 of *Contemp. Math.*, Amer. Math. Soc., Providence, RI, pp. 257–282.

[20] Edelsbrunner, H., Letscher, D., and Zomorodian, A. Topological persistence and simplification. In *Proceedings 41st Annual Symposium on Foundations of Computer Science* (2000), IEEE, pp. 454–463.

[21] Edelsbrunner, H., and Morozov, D. Persistent Homology: Theory and Practice. In *European Congress of Mathematics Kraków, 2–7 July, 2012* (2014), European Mathematical Society, pp. 31–50.

[22] Frosini, P. A distance for similarity classes of submanifolds of a Euclidean space. *Bull. Austral. Math. Soc. 42*, 3 (1990), 407–416.

[23] Frosini, P. Measuring shapes by size functions. In *Intelligent Robots and Computer Vision X: Algorithms and Techniques* (1992), D. P. Casasent, Ed., vol. 1607, International Society for Optics and Photonics, SPIE, pp. 122–133.

[24] Ghrist, R. Barcodes: The persistent topology of data. *Bull. Amer. Math. Soc. 45* (2007), 61–75.

[25] Guennebaud, G., Jacob, B., et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[26] Hensel, F., Moor, M., and Rieck, B. A Survey of Topological Machine Learning Methods. *Front. Artif. Intell. 4* (2021), 52.

[27] Henselman-Petrusek, G. Matroids and canonical forms: Theory and applications, 2020.

[28] Henselman-Petrusek, G., and Ghrist, R. Matroid Filtrations and Computational Persistent Homology. *arXiv:1606.00199* (2016).

[29] Morozov, D., and Nigmetov, A. Lock-Free Ripser. `https://github.com/mrzv/ripser/tree/lockfree`, 2020.

[30] Morozov, D., and Nigmetov, A. Towards Lockfree Persistent Homology. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2020), SPAA '20, Association for Computing Machinery, pp. 555–557.

[31] Nanda, V. Computational Algebraic Topology. `http://people.maths.ox.ac.uk/nanda/cat/TDANotes.pdf`, 2021. [Online lecture notes; accessed 4 July 2021].

[32] Nigmetov, A. Oineus. `https://github.com/grey-narn/oineus`, 2020.

[33] Otter, N., Porter, M. A., Tillmann, U., Grindrod, P., and Harrington, H. A. A roadmap for the computation of persistent homology. *EPJ Data Science 6*, 1 (Aug 2017).

[34] Oudot, S. Y. *Persistence Theory: From Quiver Representations to Data Analysis*, vol. 209 of *Mathematical Surveys and Monographs*. American Mathematical Society, 2015.

[35] Perea, J. A. A brief history of persistence. *arXiv:1809.03624* (2018).

[36] Perea, J. A., and Harer, J. Sliding windows and persistence: An application of topological methods to signal analysis. *Found. Comput. Math. 15*, 3 (2015), 799–838.

[37] Polterovich, L., Rosen, D., Samvelyan, K., and Zhang, J. *Topological Persistence in Geometry and Analysis*. University Lecture Series. American Mathematical Society, 2020.

[38] Pritam, S. *Edge collapse*, 3.4.1 ed. GUDHI Editorial Board, 2021.

[39] Robins, V. Towards computing homology from finite approximations. In *Proceedings of the 14th Summer Conference on General Topology and its Applications (Brookville, NY, 1999)* (1999), vol. 24, pp. 503–532 (2001).

[40] Robinson, M. *Topological Signal Processing*. Mathematical Engineering. Springer, 2014.

[41] Tauzin, G., Lupo, U., Tunstall, L., Pérez, J. B., Caorsi, M., Reise, W., Medina-Mardones, A., Dassatti, A., and Hess, K. giotto-tda: A Topological Data Analysis Toolkit for Machine Learning and Data Exploration. *JMLR 22*, 39 (2021), 1–6.

[42] The GUDHI Project. *GUDHI User and Reference Manual*, 3.4.1 ed. GUDHI Editorial Board, 2021.

[43] Tierny, J. *Topological Data Analysis for Scientific Visualization*. Mathematics and Visualization. Springer, 2017.

[44] Tralie, C., Saul, N., and Bar-On, R. Ripser.py: A lean persistent homology library for Python. *The Journal of Open Source Software 3*, 29 (Sep 2018), 925.

[45] von Brömssen, E. Computing persistent homology in parallel with a functional language. Master's thesis, Chalmers University of Technology, 2021.

[46] Zhang, S., Xiao, M., and Wang, H. GPU-accelerated computation of Vietoris–Rips persistence barcodes. *arXiv:2003.07989* (2020).

[47] Zomorodian, A., and Carlsson, G. Computing persistent homology. *Discrete & Computational Geometry 33*, 2 (2005), 249–274.