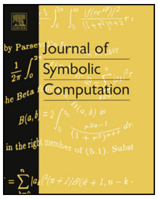
Journal of Symbolic Computation ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Symbolic Computation

www.elsevier.com/locate/jsc



PHAT – Persistent Homology Algorithms Toolbox

Ulrich Bauer[a], Michael Kerber[b], Jan Reininghaus[c], Hubert Wagner[d]

[a] *Technische Universität München (TUM), Munich, Germany*
[b] *Graz University of Technology, Graz, Austria*
[c] *CD-Adapco Inc., Vienna, Austria*
[d] *Institute of Science and Technology (IST) Austria, Klosterneuburg, Austria*

ARTICLE INFO

Article history:
Received 31 January 2015
Accepted 29 November 2015
Available online xxxx

Keywords:
Persistent homology
Topological data analysis
Matrix reduction
Algorithm engineering

ABSTRACT

PHAT is an open-source C++ library for the computation of persistent homology by matrix reduction, targeted towards developers of software for topological data analysis. We aim for a simple generic design that decouples algorithms from data structures without sacrificing efficiency or user-friendliness. We provide numerous different reduction strategies as well as data types to store and manipulate the boundary matrix. We compare the different combinations through extensive experimental evaluation and identify optimization techniques that work well in practical situations. We also compare our software with various other publicly available libraries for persistent homology.

1. Introduction

1.1. Motivation and related work

Persistent homology is one of the most widely applicable tools in the emerging field of computational topology. Intuitively, persistent homology tracks the topological features in a growing sequence of shapes; this includes the Betti numbers of each shape in the sequence, but also how homology classes appear and disappear in the process. This information can be summarized into a two-dimensional point plot summary (the *persistence diagram*) which has shown to be stable under

E-mail addresses: mail@ulrich-bauer.org (U. Bauer), kerber@tugraz.at (M. Kerber), hub.wag@gmail.com (H. Wagner).

http://dx.doi.org/10.1016/j.jsc.2016.03.008
0747-7171/© 2016 Elsevier Ltd. All rights reserved.

perturbations of the shape. For a comprehensive introduction to the theory and some applications, see Edelsbrunner and Harer (2008, 2010).

The computation of a persistence diagram usually includes two steps: the first step is the construction of a *filtered cell complex*, i.e., an ordered list of cells such that every prefix forms a combinatorial subcomplex. The filtered cell complex is often represented by its *boundary matrix*, a square matrix whose indices correspond to the ordering of the cells, and whose entries encode the boundary relation of the complex. We currently only consider homology with $\mathbb{Z}_2$-coefficients throughout, so that the boundary matrix has entries in $\{0, 1\}$. Given a boundary matrix, the second step is to compute the persistent homology itself. One approach is to transform the boundary matrix in *reduced form* using elementary column operations, similar to Gaussian elimination. A boundary matrix is called *reduced* if different columns have different pivots. The pivot of a column is the maximal index of the nonzero column entries. While alternative reduction methods based on matrix multiplication (Milosavljevic et al., 2011) and rank computations (Chen and Kerber, 2013) with superior asymptotic complexity have been presented, reduction by column operations is the basis of all efficient approaches for persistence computation to date.

For the first reduction algorithms (Edelsbrunner et al., 2002; Zomorodian and Carlsson, 2005), a quasi-linear complexity on many practical instance has been observed. However, the success of persistent homology has triggered the need of computing persistence on more and more complicated and larger datasets. In the last years, several heuristics with a tremendous effect on the performance of the algorithm have been proposed: replacing homology with cohomology (de Silva et al., 2011; Boissonnat et al., 2013), the usage of Discrete Morse Theory (Günther et al., 2011; Mischaikow and Nanda, 2013), exploiting the special structure of boundary matrices during the reduction (Chen and Kerber, 2011), and tuning the reductions towards parallelizable algorithms (Bauer et al., 2014a, 2014b; Lewis and Zomorodian, 2014; Lipsky et al., 2011). While some approaches also show favorable asymptotic bounds in special cases, the worst-case performance remains cubic in the number of cells, as in the original reduction algorithm.

The plethora of heuristics for persistence computations asks for a qualitative comparison of these approaches: previous comparisons show no clear "winner" among the approaches (e.g., Chen and Kerber, 2011; Boissonnat et al., 2013). While such experimental cross-evaluations are indisputably an important quality criterion, comparing two algorithms embedded in different software libraries reduces the informative value of such results, because the outcome is influenced by other factors than the algorithmic approach, for instance, programming language, implementation of low-level operations, and employed data structures.

1.2. Contributions

This paper introduces the PHAT library[1] as a platform for comparative evaluation of new and existing algorithms and data structures for matrix reduction. More precisely, PHAT provides a slim generic framework for reducing a boundary matrix and we have realized several of the aforementioned heuristics in this framework (see Section 3 for more details). Moreover, each algorithm also comes as a cohomology version by just running it on the anti-transposed matrix. We make the following contributions:

- We show by exhaustive experimental evaluation the tremendous impact of the *clearing* optimization in general, and of using cohomology on wide classes of inputs, confirming earlier reports (Chen and Kerber, 2011; Bauer et al., 2014a; de Silva et al., 2011) in a unified and easily reproducible software framework.
- PHAT provides several data structures to store matrix columns during the reduction process. Other libraries for persistent homology neglect the effect of choosing such a column representation (an exception is the *simplex tree* (Boissonnat and Maria, 2012) in the GUDHI library (Maria et al., 2014)). We implement various data structures in PHAT (Section 4) and provide the first systematic

[1] http://bitbucket.org/phat-code.

comparison. In particular, we propose the use of *accelerated column representations*, using different data structures for the storage and the addition of columns in a matrix. Our experiments reveal that the choice of a column representation has a similar effect on the performance as the choice of the reduction algorithm.

- We present a column representation based on a novel data structure called `bit_tree`. In essence, it implements a dense bit set, additionally supporting fast maximum queries, iteration, insertion, and deletion. This is currently our fastest data structure for additions of (binary) columns.
- We cross-evaluate PHAT with the publicly available persistence libraries DIONYSUS, JAVAPLEX, GUDHI and PERSEUS. We demonstrate that with the optimal choice of reduction strategy and column representation, PHAT outperforms these libraries for the computationally demanding task of matrix reduction. Moreover, our results suggest that substantial speed-ups could be achieved in other libraries by relatively simple optimization techniques.

1.3. *Outline*

We explain the design of our library in Section 2; in particular, our reduction scheme is parameterized by a choice of a reduction algorithm and a column representation. In Section 3, we describe the choices of reduction algorithms available in PHAT. The options for column representations are given in Section 4. We report on experiments in Section 5, testing the performance of different algorithms and data structures as well as comparing PHAT with related libraries. We conclude by summarizing our findings and giving an outlook to the future development of the library in Section 6.

2. Design

PHAT is an object-oriented C++ library consisting of about 3200 lines of code. This rather small number stems from its focus on computing persistent homology from a boundary matrix in an extendible, simple, and efficient way. In particular, PHAT does not provide code to *construct* a boundary matrix from point cloud or image data. Code redundancy is limited by using the *generic programming* paradigm, as explained below.

The main design aim is to decouple a matrix reduction algorithm from matrix representation. This way different combinations of representations and algorithms can be easily tested and benchmarked. Matrix representation takes care of implementation of storage and basic operations such as column addition. The reduction algorithm chooses the order in which matrix operations are performed.

We use the *policy-based design* paradigm, using C++'s mechanism of templates. The two orthogonal policies, or *strategies*, are modeled as C++ classes, and are combined at *compile-time*. There is no run-time overhead, as opposed to using *dynamic polymorphism*. This is crucial for performance, as the boundary matrix operations dominate the computation time and need to be as efficient as possible. For more details on *policy-based design* and *generic programming* in general, we refer to the textbooks Alexandrescu (2001), Austern (1999).

As an example, the main function of the library is declared as:

```
template<typename ReductionAlgorithm, typename Representation>
void compute_persistence_pairs(persistence_pairs& pairs,
    boundary_matrix<Representation>& matrix);
```

The function has two template parameters: `Representation` defines the data structure for the `boundary_matrix` class, and determines how columns are stored and manipulated. We will explain this indirection in a moment. `ReductionAlgorithm` defines a function object performing a particular matrix reduction strategy.

Intuitively, `compute_persistence_pairs` glues together the algorithm and matrix representation. The actual implementation simply takes a `matrix` as input and reduces it using the supplied reduction strategy. Then, it stores the resulting *persistent pairs* in the container `pairs`. PHAT currently implements 5 different reduction algorithms and 8 matrix representations, resulting in a total of 40 different combinations of polices.

Since a boundary matrix interfaces between the user code and our library, we explain its design rationale. It is implemented using the following class template:

```
template<typename Representation = default_representation>
class boundary_matrix {
   Representation rep;
   int get_max_index( int idx );
   void add_to( int source, int target );
   ...
};
```

The `boundary_matrix` class specifies the interface to a user. The actual implementation depends on the `Representation` data structure, transparent to the user. Moreover, a default representation is provided, which simplifies the first contact with the library.

The additional indirection introduced by the `boundary_matrix` class is motivated by the explicit specification of the interface of the class. This is useful, for example, for code completion tools, which do not work for template arguments without an explicit interface declaration. Additionally, the `boundary_matrix` class exposes other useful functions like file input/output, which are independent of the representation.

Several matrix representations are implemented using STL collections. Since we presently only work with binary matrices, all the data structures simply store the indices of the nonzero entries for each column. In addition, the dimensions of the cells in the filtration are stored in a `vector`. For more details see Section 4.

The design of PHAT decouples the reduction algorithm from the implementation of matrix storage and operations. As shown in Section 5, both aspects are equally important for an efficient implementation. Our design allows us to easily benchmark all configurations and identify the most efficient ones. We describe the options provided by the PHAT library below. Finally, we hide some of the complexity from the user by providing default strategies and easy to use wrapper classes.

3. Algorithms

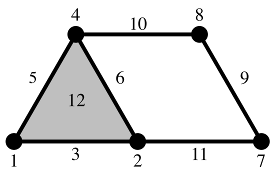
We use the following notation throughout this paper. The *pivot index* of a column in the matrix is the largest index of any nonzero entry. All our reduction algorithms perform left-to-right column additions until no two columns have the same pivot index. A matrix with this property is called *reduced*. Recall that we restrict to $\mathbb{Z}_2$-homology in this paper, so adding two columns with the same pivot index results in a column with a strictly smaller pivot index, or a zero columns if the columns were equal.

PHAT 1.4.1 provides five different choices of reduction strategies, two of which have a parallel implementation using the OpenMP API.[2] As a running (toy) example, we consider the complex depicted in Fig. 1 (left). The simplices are added in the order of their indices, and the resulting boundary matrix is given in Fig. 1 (right).

3.1. Sequential algorithms

The `standard` algorithm for reducing boundary matrices (Edelsbrunner et al., 2002; Cohen-Steiner et al., 2006) traverses the columns from left to right and maintains the invariant that after having traversed the j-th column, the first j columns of the matrix form a reduced submatrix. For handling the j-th column, we check whether its pivot index appears as pivot index of a previous column, say column i, which would violate the invariant. In this case, we add column i to column j, which eliminates the nonzero pivot entry and thus decreases the pivot index, and repeat until the pivot index does not appear as a pivot index in a previous column or the column becomes zero.

[2] www.openmp.org.



∂	1	2	3	4	5	6	7	8	9	10	11	12
1			1		1							
2			**1**			1					1	
3												1
4					**1**	**1**				1		
5												1
6												**1**
7									1		**1**	
8									**1**	**1**		
9												
10												
11												
12												

∂	1	2	3	4	5	6	7	8	9	10	11	12
1			1		1							
2			**1**									
3												1
4					**1**					1		
5												1
6												**1**
7									1	**1**		
8									**1**			
9												
10												
11												
12												

Fig. 1. An example complex of dimension 2 (left), its boundary matrix (middle), and the reduced boundary matrix computed by the standard algorithm (right). The pivot entry of a column is given in bold.

As an example, we explain the execution of the algorithm on the complex from Fig. 1. The only manipulation of the boundary matrix is a left-to-right column addition of the form $R_j \leftarrow R_j + R_i$, where R_i, R_j are columns of the matrix and $i < j$. We write $j \leftarrow j+i$ as a notational shortcut for this operation. Then, the standard algorithm traverses the columns from 1 to 12 in order and performs the following operations sequentially:

$$6 \leftarrow 6+5, 6 \leftarrow 6+3, 10 \leftarrow 10+9, 11 \leftarrow 11+10, 11 \leftarrow 11+5.$$

The resulting reduced matrix has pivots at indices

$$(2,3), (4,5), (8,9), (7,10), (6,12),$$

which are the persistence pairs. The standard algorithm computes two representative cycles of a homology class represented by the persistence pair (i, j): on the one hand, the (nonzero) j-th column of the reduced matrix encodes a cycle that becomes a boundary at this point, and thus represents the homology class just before it dies. On the other hand, the linear combination of columns that turns the i-th column to zero yields another representative of the homology class, at the moment the class is born. For instance, in our example above, we reduced column 6 by adding columns 3, 5, and 6, which indeed represents the cycle formed when adding edge 6 in Fig. 1 (left) (see also Edelsbrunner and Harer, 2010, §VII.1).

The algorithm `twist` (Chen and Kerber, 2011) is based on the standard algorithm and exploits the observation that a column will eventually be reduced to an empty column if its index appears as the pivot of another column. By reducing columns in decreasing order of the dimensions of the corresponding cells, we can explicitly *clear* the columns whose indices appear as pivot indices. This is done by performing the reduction in multiple passes, one for each dimension. Clearing does not affect the reduction of other columns – the resulting reduced matrix is therefore the same as in the standard algorithm. Since the omitted column operations often constitute the bulk of the column operations in the standard algorithm, the clearing optimization can have a tremendous impact on practical performance. It is therefore also used in all other algorithms described below. Due to its simplicity and efficiency, the `twist` algorithm is the default in PHAT.

In our example, the twist reduction traverses the columns in order

$$12, 3, 5, 6, 9, 10, 11, 1, 2, 4, 7, 8$$

(first triangles, then edges, then vertices). We let $i \leftarrow 0$ denote the operation of setting column i to zero. Then, the following operations are performed:

$$6 \leftarrow 0, 10 \leftarrow 10+9, 11 \leftarrow 11+10, 11 \leftarrow 11+5.$$

The reduced matrix is equal to the reduced matrix of the standard algorithm (for all inputs), and therefore, also the persistence pairs are the same.

The twist optimization only computes a single representative for each homology class: for a persistence pair (i, j), the j-th column represents the homology class just before it dies as in the standard algorithm, but the representation of the class at its time of birth is avoided. This is precisely the

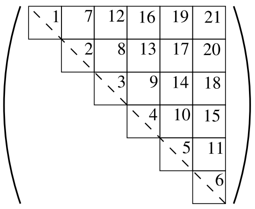


Fig. 2. The block-wise reduction order of the spectral sequence algorithm.

reason for the computational savings. The idea of avoiding the computation of multiple representatives is also implicitly employed in the persistent cohomology algorithms implemented in the libraries Dionysus and Gudhi.

In the `row` algorithm (de Silva et al., 2011), we maintain the invariant that after the j-th iteration, the submatrix formed by the lowest j rows of the matrix is reduced. To handle the j-th iteration, we let k denote the row index of the j-th row from the bottom, that is, $k = n - j + 1$ with n the number of simplices. If there are two or more columns with k as their pivot index, the invariant is violated. We let R_i denote the leftmost such column, and we add R_i to every further column with the same pivot index. We further integrate the clearing optimization in the algorithm: if k appears as a pivot index, we clear column k at the end of the j-th iteration.

In our example, the algorithm performs the following operations:

$$10 \leftarrow 10 + 9, 11 \leftarrow 11 + 10, 6 \leftarrow 0, 11 \leftarrow 11 + 5.$$

Note that this is a permutation of the operations performed in the twist variant and it yields the same reduced matrix. This is true in general. The advantage of the row algorithm is that the algorithm does not have to store any reduced columns for further iterations: with the notation above, after R_i has been added to all columns with same pivot index, R_i is not needed for any further operations in the algorithm. This can result in memory savings because there is no need to save the entire reduced matrix in memory (de Silva et al., 2011).

3.2. Parallel algorithms

The algorithm `spectral_sequence` is an implementation of the parallel algorithm outlined in Edelsbrunner and Harer (2010). We conceptually divide the boundary matrix into m^2 blocks of (roughly) equal size. We then reduce the matrix block-wise, starting from the diagonal, moving upwards towards the upper-right corner; see Fig. 2 for an illustration. Reducing a block means that we ensure that no two columns have the same pivot element within the block; in particular, reducing a block might only partially reduce the corresponding columns, and the reduction is finished at a later stage when another block is reduced. We can observe that the reductions of blocks within the same diagonal (i.e., blocks 1–6, blocks 7–11, etc.) are independent and can thus be computed in parallel. For that reason, we choose m as the number of available processors of the given machine.

The `chunk` algorithm (Bauer et al., 2014a) begins, similarly to the spectral sequence algorithm, with the reduction of the two main diagonals of blocks (blocks 1–6 and 7–11 in the example above). In a second step, it simplifies the partially reduced boundary matrix by eliminating the indices of the already found pairs from the matrix. Finally, the simplified matrix is reduced using the twist algorithm. The chunk algorithm is a generalized version of the approach in Günther et al. (2012) to compute persistence using discrete Morse theory (Forman, 1998): Collapses based on Morse matchings are replaced with elimination of local persistence pairs, allowing for the removal of pairs of non-incident simplices of the complex. The first and the second step of the algorithm can be run in parallel, respectively. The algorithm exploits the fact that the simplified matrix is much smaller than the original matrix for many inputs, and the final (and non-parallel) step therefore tends to finish fast in practice.

3.3. *Duality of persistence*

Every algorithm for persistent homology also yields an algorithm for persistent cohomology by applying it to the corresponding coboundary matrix. This matrix is given by the *anti-transpose* of the boundary matrix D, obtained by swapping $D_{i,j}$ with $D_{n+1-j,n+1-i}$. Reducing the coboundary matrix yields the same persistence pairs, up to reindexing. As an example, we consider the matrix from above, whose anti-transpose is the following matrix (the dashed line shows the anti-diagonal along which the entries are reflected):

$$\begin{array}{c|cccccccccccc}
\partial & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
1 & & & & & & & \mathbf{1} & \mathbf{1} & & \mathbf{1} & & \\
2 & & & & & & 1 & & & & & 1 & \\
3 & & & & & 1 & & & & 1 & & & \\
4 & & & & & \mathbf{1} & \mathbf{1} & & & & & & \\
5 & & & & & & & & & & & & \\
6 & & & & & & & & & & & & \\
7 & & & & & & & & & 1 & & 1 & \\
8 & & & & & & & & & \mathbf{1} & & & 1 \\
9 & & & & & & & & & & & & \\
10 & & & & & & & & & & & \mathbf{1} & \mathbf{1} \\
11 & & & & & & & & & & & & \\
12 & & & & & & & & & & & &
\end{array}$$

Applying the standard algorithm yields the following sequence of operations:

$$6 \leftarrow 6+5, 8 \leftarrow 8+7, 10 \leftarrow 10+7, 12 \leftarrow 12+11, 12 \leftarrow 12+9, 12 \leftarrow 12+6.$$

We obtain the persistence pairs $(4,5),(3,6),(1,7),(8,9),(10,11)$. Replacing a pair (i,j) with $(n-j+1, n-i+1)$, we obtain the same persistence pairs as above.

In some cases, computing persistent cohomology is significantly faster than persistent homology (de Silva et al., 2011), in particular for the common case of Vietoris–Rips filtrations. For convenience, PHAT contains a dualization option, which anti-transposes the matrix before applying the reduction algorithm. However, in practice this expensive dualization process should be avoided and the coboundary matrix should be generated directly from the input, as done in the library DIPHA (Bauer et al., 2014b). Instead of reducing the coboundary matrix, one may alternatively apply a *dual* algorithm: the aforementioned libraries DIONYSUS and GUDHI implement persistent cohomology algorithms that directly operate on the boundary matrix.

4. Data structures

All boundary matrix data structures currently implemented in PHAT use a `vector` containing the individual columns. The column type is defined by the representation. There are two groups of representations in PHAT, direct and accelerated, which are described below. To simplify notation, we refer to the number of nonzero entries of a column as its *size*.

4.1. *Direct representations*

A direct representation simply stores every column of the matrix using the same data type. We provide various options which are based on several container types from the STL library (Austern, 1999). The class `vector_list` represents a column as a doubly-linked list (`list<int>`) storing the indices of nonzero entries in increasing order, as suggested in Edelsbrunner and Harer (2010). Adding two columns of sizes k and m can therefore be performed in time $O(k+m)$ by computing the symmetric difference of the lists. The pivot of a column can be found in $O(1)$ by querying the last element of the list.

The representation `vector_vector` is analogous, using a dynamically growing array (`vector<int>`) instead of a linked list. This representation is more machine friendly, since it makes use of a contiguous memory region. However, both representations have the disadvantage that column additions are expensive when a small column is added to a large column.

An alternative representation is `vector_set`, where columns are stored as balanced binary search trees (`set<int>`). Adding a column A of size k to a column B of size m can be performed as follows. We iterate through the entries of A, removing the entry from B if it is already present, and inserting it otherwise. The complexity of such an addition is $O(k \log(k+m))$, which can be much better than $O(k+m)$ when $k \ll m$. The pivot of a column can be found in $O(1)$.

The representation `vector_heap` combines the advantages of contiguous storage and efficient column addition. Columns are again stored as `vector<int>`, but the indices are now arranged in heap order. Adding a column A of size k to a column B of size m can be lazily performed by inserting the indices of A into B in amortized time $O(k \log(k+m))$. This implies that an index may temporarily appear multiple times in the heap. The symmetric difference operation is delayed until a certain number of insertions is exceeded or the content of the column is queried. This allows for the pivot of a column of size k to be found in amortized time $O(1)$.

4.2. Accelerated representations

An accelerated representation is an extension of `vector_vector` from above. It provides an additional data type optimized for fast column additions. The representation contains one object of this type, called the *working column* (the parallel versions contain one working column per execution thread). Before a column gets manipulated through column addition, it is loaded into the working column (that is, converted into the working column type), and converted back into vector representation when another column gets manipulated. This conversion is done in the method `make_pivot_col` of the class `abstract_pivot_column` whenever a column with a different index is accessed, thereby always keeping the last accessed column in the working column data structure. Similar to `vector_heap`, this strategy combines contiguous storage with efficient column additions. For a net gain in performance, efficient conversion between `vector<int>` and the working column type is required. Moreover, the employed algorithm needs to exhibit a cache-friendly access pattern for manipulating columns. This is the case for all (sequential) algorithms of Section 3 except for the `row` algorithm. The use of accelerated representation in connection with the `row` algorithm is therefore discouraged.

A simple yet reasonably efficient choice for the working column is `set<int>`. The resulting representation is called `sparse_pivot_column`. Another option is the use of a heap as explained in the description of `vector_heap`. The corresponding representation is called `heap_pivot_column`.

Another accelerated representation, `full_pivot_column`, explicitly stores a dense bit vector corresponding to the currently reduced column. The bit vector requires n bits of memory, and allows for fast insertions and removal. To facilitate finding the pivot (maximum) value, we pair the bit vector with a max-heap. When converting the bit array back into a `vector<int>`, we repeatedly extract and remove the pivot in order to clear the structure for further use.

Adding a column of size k to this representation still takes time $O(k \log(k+m))$ due to the heap operations. However, compared to `sparse_pivot_column`, memory locality is significantly improved.

4.3. Bit tree

As an extension, we propose the `bit_tree_pivot_column` representation, which uses a specialized data structure we call a *bit tree*. A bit tree is a hierarchical, dense bit vector, stored as a contiguous block of bits, which implicitly encodes an 64-ary tree. It simplifies the idea behind the classical van Emde–Boas trees (Cormen et al., 2009). The *summary* structure is different: Instead of using recursion, we use a single integer, interpreted as a bit vector of size 64. All relevant operations exploit the current hardware and are very efficient. Additionally, the height of the tree is only $\lceil \log_{64} n \rceil$. Hence, the height is bounded by 6 for all currently realistic inputs.

A bit tree supports fast insertions, deletions and lookup of entries, as well as maximum, minimum, successor, and predecessor queries, all in time $O(\log_{64} n) = O(\log n)$. It can also be cleared in time proportional to the number of *nonzero* entries, so sparse columns are handled efficiently and once the structure is initialized, it can be reused for all consecutive reductions.

Table 1
Running times (in seconds) of different combinations of algorithms and data structures. The prefix "A-" refers to accelerated representations, while (·)* denotes dualization. The input data is the 3-skeleton of the Vietoris–Rips filtration of 64 points on a 2-sphere.

	List	Vector	Set	Heap	A-Heap	A-Set	A-Full	**A-Bit-Tree**
standard	17.1	2.8	7.5	5.9	5.6	8.6	2.3	1.62
standard*	2580.0	168.0	17.3	13.5	14.6	16.0	3.9	0.57
twist	16.3	2.7	7.0	5.7	5.4	6.5	2.2	1.59
twist*	0.23	0.03	0.04	0.02	0.03	0.03	0.02	**0.02**
row	39.9	4.3	20.3	7.2	21.4	37.9	15.5	13.8
row*	0.25	0.06	0.06	0.05	0.08	0.09	0.05	0.05
chunk	0.63	0.19	0.6	0.50	0.35	0.5	0.24	0.24
chunk*	2.9	0.42	0.1	0.11	0.17	0.14	0.07	0.04
spectral	10.7	1.8	4.03	3.3	3.4	4.3	2.1	1.2
spectral*	0.35	0.03	0.05	0.04	0.04	0.04	0.02	**0.01**

While the worst-case complexity of the main operations is theoretically worse than the $O(\log\log n)$ complexity of van Emde–Boas trees, our data structure is more efficient in practice, and simpler to implement.

5. Experiments

To evaluate the performance of the different algorithms and data structures, we perform computational experiments using three data sets. The requisite boundary matrices were generated using the `create_phat_filtration` tool included in DIPHA. The running times are measured on a workstation with two Intel Xeon E5645 CPUs using the integrated benchmark utility from PHAT 1.4.1. Parallel algorithms are performed on 24 logical cores, the maximum available on the workstation. We use Visual Studio 2013 to produce 64 bit executables, and Java 8.0_66 in 64 bit mode. All data sets are available on the project homepage (http://bitbucket.org/phat-code).

5.1. *Rips filtrations*

The first data set is the 3-skeleton of the Vietoris–Rips filtration of a point cloud generated by a uniform random sample of the 2-sphere. Using 64 points, the resulting boundary matrix has 679 120 columns and 2 670 528 nonzero entries. The running times in seconds for the matrix reduction of this data set are shown in Table 1, where we denote the computation of persistent cohomology (i.e., the algorithm applied on the coboundary matrix) by an asterisk. The combination of persistent cohomology with algorithms employing the clearing optimization leads to drastically shorter running times compared to other choices. To admit a meaningful comparison for these fast algorithms, we repeat the experiment using 192 points, resulting in a boundary matrix with 56 050 288 columns and 223 002 432 nonzero entries. The running times in Table 3 also show the importance of choosing an appropriate data structure for column operations; in particular, the "standard choices" of `list` and `vector` yield running times significantly worse than other choices. The `twist` and `row` algorithms show comparable running times for non-accelerated column types, which is not surprising because both algorithms perform the same column operations, just in different orders. The striking performance penalty when using accelerated representations for the row algorithm is due to the reduction strategy of the `row` reduction, which interleaves the reduction of several columns. This results in many changes of the working column (cf. Section 4.2) and slows down the algorithm because of the numerous conversions from `vector` into the working column type and vice versa. Finally, the simple sequential `twist` algorithm is about as fast as the parallel algorithms on these examples.

5.2. *Three-dimensional images*

The second data set is a sublevel set filtration of a cubical complex generated from a 3D image that indicates separation behavior in a vector field (Kasten et al., 2014). The cubical complex is a grid

Table 2
Running times (in seconds) of selected combinations of algorithms and data structures for the 3-skeleton of the Vietoris–Rips filtration of 64 points on a 2-sphere. See Table 1 for details.

	List	Vector	Set	Heap	A-Heap	A-Set	A-Full	**A-Bit-Tree**
twist*	0.23	0.03	0.04	0.02	0.03	0.03	0.02	**0.02**
spectral*	0.35	0.03	0.05	0.04	0.04	0.04	0.02	**0.01**

	standard*	**twist***	chunk*	**spectral***
A-Bit-Tree	0.57	**0.02**	0.04	**0.01**

Table 3
Running times in seconds for the 3-skeleton of the Vietoris–Rips filtration of 192 points on a 2-sphere. See Table 1 for details.

	List	Vector	Set	Heap	A-Heap	A-Set	A-Full	**A-Bit-Tree**
twist*	2635.4	339.9	4.9	2.0	2.5	6.1	2.1	**1.0**
spectral*	2644.8	349.2	5.2	1.9	3.3	6.6	3.1	**1.0**

Table 4
Running times in seconds for a 64^3 image data set. See Table 1 for details.

	List	Vector	Set	Heap	A-Heap	A-Set	A-Full	**A-Bit-Tree**
standard	141.1	16.1	23.9	17.5	19.4	21.5	10.8	10.7
standard*	460.2	39.6	27.5	18.8	20.8	22.8	14.0	9.8
twist	9.8	0.54	0.30	0.11	0.13	0.13	0.09	**0.07**
twist*	337.1	18.6	0.99	0.52	0.52	0.72	0.24	0.11
row	9.9	1.5	0.50	0.48	1.5	2.1	1.2	0.78
row*	350	43.8	1.0	0.93	24.5	44.0	15.5	7.0
chunk	1.8	0.19	0.19	0.12	0.09	0.09	0.08	0.08
chunk*	5.7	0.53	0.27	0.22	0.19	0.17	0.13	0.14
spectral	8.4	0.78	0.19	0.11	0.11	0.11	0.08	**0.06**
spectral*	339.2	21.9	0.90	0.65	0.74	0.74	0.35	0.12

Table 5
Running times in seconds of selected combinations of algorithms and data structures for a 64^3 image data set. See Table 1 for details.

	List	Vector	Set	Heap	A-Heap	A-Set	A-Full	**A-Bit-Tree**
twist	9.8	0.54	0.30	0.11	0.13	0.13	0.09	**0.07**
chunk	1.8	0.19	0.19	0.12	0.09	0.09	0.08	0.08
spectral	8.4	0.78	0.19	0.11	0.11	0.11	0.08	**0.06**

	Standard	**Twist**	Chunk	**Spectral**
A-Bit-Tree	10.7	**0.07**	0.08	**0.06**

with one vertex per voxel, and the function assigns to each vertex the corresponding voxel value and to each higher-dimensional cube the maximum value of its vertices (lower star filtration). Using a 64^3 sub-region of the image, we get a boundary matrix with 2 048 383 columns and 6 096 762 nonzero entries. The running times for this data set are shown in Table 4. We observe that homology computation is generally faster than cohomology computation for this data set, and the clearing optimization is again crucial for a fast algorithm. To investigate the performance behavior further, we also apply a selection of the fastest algorithms to the full data set consisting of 256^3 voxels – the corresponding boundary matrix has 133 432 831 columns and 399 515 130 nonzero entries. The results in Table 6 demonstrate the usefulness of the accelerated data structures introduced in Section 4.2. In contrast to the first data set, the parallel algorithms perform faster than the sequential methods in this case. The speedup compared to the implementation as proposed in Edelsbrunner and Harer (2010) exceeds a factor of 1000.

Table 6
Running times in seconds for a 256^3 image data set. See Table 1 for details.

	List	Vector	Set	Heap	A-Heap	A-Set	A-Full	**A-Bit-Tree**
twist	2080.2	101.7	26.4	11.3	11.1	12.3	10.4	**8.8**
chunk	894.5	156.1	9.8	6.5	6.3	6.2	5.6	**4.7**
spectral	1197.7	261.7	11.9	8.5	8.5	8.4	7.1	6.1

Table 7
Running times in seconds for a flag complex filtration of graphs induced by the cosine dissimilarity measure of text documents.

	List	Vector	Set	Heap	A-Heap	A-Set	A-Full	**A-Bit-Tree**
twist*	424.9	24.6	37.2	32.0	25.2	29.1	8.3	**2.80**
chunk*	418.7	25.2	37.6	28.8	25.2	29.4	9.0	**2.86**
spectral*	381.1	27.2	31.7	25.1	22.7	26.3	8.6	2.92

5.3. *Dissimilarity measures*

The third data set comes from an application of persistent homology in text mining (Wagner and Dłotko, 2014). We consider a filtration of *flag complexes* of graphs induced by dissimilarities between 1500 text documents. The data set represents the boundary matrix of the 4-skeleton of this complex. It contains 1 448 504 columns and 7 002 178 nonzero entries.

The running times for this data set are shown in Table 7. This data set is challenging, and we show only the algorithms which could finish within reasonable time frame. We can observe that all the algorithms behave similarly. As for the data structures, A-Full and A-Bit-Tree are clearly more efficient than the remaining ones, the latter one outperforming the non-accelerated data structures by a factor of 8. We attribute this effect in part to the small number of dynamic memory allocations required in A-Full and A-Bit-Tree.

5.4. *Comparison with other packages*

We compare PHAT (version 1.4.1) with several publicly available software packages for persistence: DIONYSUS (Morozov, 2010) (revision r280), JAVAPLEX (Adams et al., 2014) (version 4.2.1), PERSEUS (Nanda, 2013) (version 4 beta 4) and GUDHI (Maria et al., 2014) (version 1.1.0). In our experiment, we are focusing entirely on the matrix reduction performance of these libraries; we ignore peripheral computations, such as the creation of the (co)boundary matrix, which is provided by most libraries. All timings were measured using `std::clock()` in C++ and `System.currentTimeMillis()` in Java.

There are several other libraries for persistence that we have not included in our comparison for various reasons: The DIPHA library (Bauer et al., 2014b) performs reduction using a spectral sequence algorithm and heap representation of the columns. On a shared memory system, it therefore exhibits running times roughly comparable to PHAT with the appropriate parameters. The CTL library[3] is currently still under development. The R package TDA (Fasy et al., 2015) provides an interface to GUDHI, DIONYSUS, and PHAT. The package REDHOM[4] uses PHAT internally for matrix reduction.

5.5. *Configuration details*

For evaluating the performance of DIONYSUS, we used the example file `rips-pairwise-cohomology.cpp` provided with the package. The algorithm is described in de Silva et al. (2011). It is a *dual* algorithm, in the sense that it takes a boundary matrix as input

[3] http://ctl.appliedtopology.org/.
[4] http://capd.sourceforge.net/capdRedHom/index.php.

Table 8
Running times in seconds for the datasets from Tables 1–6. For the first two instances, PHAT is applied on the coboundary matrix. (m) indicates that the program failed to finish due to consuming more than 64 gigabytes of RAM. A missing number indicates that the data set is not supported by the software.

	Dionysus	JavaPlex	Perseus	Gudhi	Phat (simple)	Phat (opt)
Rips 64	2.6	4.4	18.0	0.15	0.02	0.01
Rips 192	359	465	(m)	9.8	2.0	1.0
Image 64^3		163	11 139		0.11	0.06
Image 256^3		(m)	(m)		11.3	4.7

and computes persistent cohomology, so it should be compared to PHAT with a coboundary matrix as input. We note that this algorithm also employs an equivalent to the clearing optimization. Specifically, in contrast to the standard algorithm applied to the coboundary matrix, this algorithm only computes one cocycle corresponding to a birth in persistent cohomology, similarly to the effect achieved by the clearing optimization. Consequently, it performs significantly faster than the alternative `rips-pairwise.cpp`, which computes persistent homology using the standard algorithm.

The benchmark for GUDHI is based on the examples `rips_persistence.cpp` and `performance_rips_persistence.cpp` provided with the package. The algorithm is an improved implementation of the persistent cohomology algorithm found in DIONYSUS (de Silva et al., 2011), using specialized data structures (Boissonnat et al., 2013). We used the data structure `Hasse_complex<>` for comparing the reduction time, since it provides faster performance for reduction than the more memory-efficient data structure `Simplex_tree<>` (Boissonnat and Maria, 2012), which is only used for constructing the filtration here. We note that generating and subsequently performing reduction the `Hasse_complex<>` takes about the same time as performing reduction directly on the `Simplex_tree<>`.

In JAVAPLEX, we used `Plex4.getDefaultSimplicialAlgorithm` for Rips filtrations and `Plex4.getDefaultCellularAlgorithm` for image data to compute persistence with $\mathbb{Z}_2$ coefficients using the standard algorithm. JAVAPLEX provides the option of dualizing the input using the class `DualStream`. We have not used this option, as it did not lead to an improvement in performance. We attribute this to our observation that computing persistent cohomology of Rips filtrations is only beneficial in combination with the clearing optimization (see Table 1), which apparently is not implemented in JAVAPLEX.

The benchmark for PERSEUS is based on the file `Pers.cpp` provided with the software. We chose the option to perform alternating Morse reductions and coreductions before the computation of persistence pairs, which leads to the best performance. Since the Morse (co)reductions are part of the reduction strategy and corresponds to boundary matrix operations, they are also accounted for in our timing. We stress that PHAT and PERSEUS interpret cubical data differently: PHAT interprets the image voxel values as values at the vertices of a cubical complex, whereas PERSEUS interprets them as values at the maximal cubes. Up to boundary effects, the filtration based on maximal cubes is dual to the vertex-based filtration of the negated function. Note, however, that the two constructions are not entirely symmetric with respect to dimension. The results in Table 4 indicate that dualization has a significant effect on performance.

In Table 8, we list the running times of the reduction using the Vietoris–Rips and image data sets described earlier. Since no tested library except PERSEUS provided direct support for cubical image data, we omit a comparison for these libraries.

We compare with two configurations from PHAT: The *simple* configuration uses the `twist` algorithm and the column type `vector_heap`, both having an elementary implementation. The *opt* configuration refers to the fastest reduction among all options in PHAT, as determined in Tables 1, 3, 4 and 6.

The results of Table 8 indicate that the matrix reduction performance of PHAT is faster but still in the same order of magnitude as GUDHI (confirming the observation in Boissonnat et al., 2013) and clearly ahead of other libraries.

Table 9 shows memory usage, measured as the peak working set size in bytes. DIONYSUS, GUDHI and PHAT show similar memory consumption. GUDHI is most memory-efficient when us-

Table 9
Memory consumption in megabytes or gigabytes. See Table 8 for more details.

	Dionysus	JavaPlex	Perseus	Gudhi	Phat (simple)	Phat (opt)
Rips 64	60 MB	270 MB	718 MB	44 MB	53 MB	61 MB
Rips 192	4.9 GB	12.3 GB	(m)	3.1 GB	3.6 GB	3.8 GB
Image 64^3		2.04 GB	1.5 GB		0.16 GB	0.16 GB
Image 256^3		(m)	(m)		10.2 GB	10.3 GB

ing the `Simplex_tree<>` data structure, which is therefore used in this comparison instead of `Hasse_complex<>`.

We stress that all of those libraries provide important additional functionality, e.g., creating boundary matrices from point cloud data or support for field coefficients other than $\mathbb{Z}_2$, while PHAT concentrates entirely on the reduction process and is currently limited to $\mathbb{Z}_2$ coefficients. However, our results show a large potential for speed-ups in a computationally important subtask. In particular, our simple configuration using the twist algorithm and heap column type yields near-optimal speed-ups. We assume that an adaption of these simple techniques would yield a performance boost in other libraries with little implementation effort.

6. Conclusion

The experiments clearly show that the choice of reduction algorithms and column data structures has significant impact on practical efficiency. The generic programming design of PHAT was instrumental to achieving this goal – it made testing different configuration of algorithms and data structures easy.

Our experimental results point out three significant insights:

1. Algorithms using the clearing optimization significantly improve over the standard algorithm in all cases.
2. The choice of data structures for column reduction and storage can have a large impact.
3. The combination of computing persistent *cohomology* and employing the clearing optimization is highly beneficial for (low-dimensional skeleta of) Rips filtrations. The same advice applies to other filtrations of spaces with large Betti numbers in the top dimension. However, in this scenario the two optimizations have to be used in combination in order to achieve their full effect.

We recommend the following usage (as of PHAT 1.4.1):

1. Algorithm: `twist_reduction` (default for the `compute_persistence_pairs` function and for the command-line tool). In comparison, the parallel algorithms `chunk` and `spectral_sequence` can speed up the computation, but often not by much. They require a compiler that supports OpenMP.
2. Column addition strategy: `bit_tree_pivot_column` (default representation for the `boundary_matrix` and the command-line tool). This data structure is the fastest in all instances we tested.
2. Computing persistent *cohomology* for Rips filtrations. While this can be done for an input boundary matrix using the convenience function `compute_persistence_pairs_dualized` or the `--dual` command-line option, for best performance we highly recommend generating a coboundary matrix already at the previous stage of the pipeline. The same advice applies to other data known to have many infinite persistent intervals in the top dimension.

We conclude that recent persistence software can handle large data sets efficiently. Moreover, this is achieved by a single choice of (sequential) algorithm and data structure. It is also apparent that existing packages can benefit from the new algorithmic techniques. In particular, the standard algorithm and the list data structure of columns are not recommended for handling large inputs.

In future developments of PHAT, we plan to implement support for coefficients other than $\mathbb{Z}_2$, and the computation of generating cycles and Smith normal forms. Moreover, we plan to include PHAT in a larger user-friendly package for topological data analysis that also supports the efficient generation of filtrations from data.

Acknowledgements

Michael Kerber acknowledges support by the Max Planck Center for Visual Computing and Communications (FKZ-01IMC01 and FKZ-01IM10001). Ulrich Bauer, Jan Reininghaus, and Hubert Wagner acknowledge support by the EU Project TOPOSYS (FP7-ICT-318493-STREP).

References

Adams, H., Tausz, A., Vejdemo-Johansson, M., 2014. javaPlex: a research software package for persistent (co)homology. In: Hong, H., Yap, C. (Eds.), Mathematical Software – ICMS 2014. In: Lecture Notes in Computer Science, vol. 8592. Springer, Berlin, Heidelberg, pp. 129–136. http://dx.doi.org/10.1007/978-3-662-44199-2_ 23.
Alexandrescu, A., 2001. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison–Wesley.
Austern, M.H., 1999. Generic Programming and the STL. Addison–Wesley.
Bauer, U., Kerber, M., Reininghaus, J., 2014a. Clear and compress: computing persistent homology in chunks. In: Topological Methods in Data Analysis and Visualization III. In: Mathematics and Visualization. Springer, pp. 103–117. http://dx.doi.org/10.1007/978-3-319-04099-8_7.
Bauer, U., Kerber, M., Reininghaus, J., 2014b. Distributed computation of persistent homology. In: 2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2014. Portland, Oregon, USA, January 5, 2014, pp. 31–38.
Boissonnat, J., Dey, T.K., Maria, C., 2013. The compressed annotation matrix: an efficient data structure for computing persistent cohomology. In: Algorithms – ESA 2013 – 21st Annual European Symposium, Proceedings. Sophia Antipolis, France, September 2–4, 2013, pp. 695–706.
Boissonnat, J., Maria, C., 2012. The simplex tree: an efficient data structure for general simplicial complexes. In: Algorithms – ESA 2012 – 20th Annual European Symposium, Proceedings. Ljubljana, Slovenia, September 10–12, 2012, pp. 731–742.
Chen, C., Kerber, M., 2011. Persistent homology computation with a twist. In: 27th European Workshop on Computational Geometry (EuroCG), pp. 197–200. URL http://eurocg11.inf.ethz.ch/abstracts/22.pdf.
Chen, C., Kerber, M., 2013. An output-sensitive algorithm for persistent homology. Comput. Geom. 46 (4), 435–447. http://dx.doi.org/10.1016/j.comgeo.2012.02.010.
Cohen-Steiner, D., Edelsbrunner, H., Morozov, D., 2006. Vines and vineyards by updating persistence in linear time. In: Proceedings of the Twenty-second Annual Symposium on Computational Geometry. SCG'06. ACM, New York, NY, USA, pp. 119–126.
Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2009. Introduction to Algorithms, third edition. The MIT Press.
de Silva, V., Morozov, D., Vejdemo-Johansson, M., 2011. Dualities in persistent (co)homology. Inverse Probl. 27 (12), 124003+. http://dx.doi.org/10.1088/0266-5611/27/12/124003.
Edelsbrunner, H., Harer, J., 2008. Persistent homology – a survey. In: Surveys on Discrete and Computational Geometry: Twenty Years Later. In: Contemporary Mathematics, pp. 257–282.
Edelsbrunner, H., Harer, J., 2010. Computational Topology. An Introduction. American Mathematical Society.
Edelsbrunner, H., Letscher, D., Zomorodian, A., 2002. Topological persistence and simplification. Discrete Comput. Geom. 28 (4), 511–533. http://dx.doi.org/10.1007/s00454-002-2885-2.
Fasy, B.T., Kim, J., Lecci, F., Maria, C., 2015. Introduction to the R package TDA. arXiv:1411.1830 [cs.MS].
Forman, R., 1998. Morse theory for cell complexes. Adv. Math. 134 (1), 90–145. http://dx.doi.org/10.1006/aima.1997.1650.
Günther, D., Reininghaus, J., Hotz, I., Wagner, H., 2011. Memory-efficient computation of persistent homology for 3D images using discrete Morse theory. In: 24th SIBGRAPI Conference on Graphics, Patterns and Images, Sibgrapi 2011. Alagoas, Maceió, Brazil, August 28–31, 2011, pp. 25–32.
Günther, D., Reininghaus, J., Wagner, H., Hotz, I., 2012. Efficient computation of 3D Morse–Smale complexes and persistent homology using discrete Morse theory. Vis. Comput. 28 (10), 959–969. http://dx.doi.org/10.1007/s00371-012-0726-8.
Kasten, J., Reininghaus, J., Reich, W., Scheuermann, G., 2014. Toward the extraction of saddle periodic orbits. In: Topological Methods in Data Analysis and Visualization III. In: Mathematics and Visualization. Springer, pp. 55–69. http://dx.doi.org/10.1007/978-3-319-04099-8_4.
Lewis, R.H., Zomorodian, A., 2014. Multicore homology via Mayer Vietoris. arXiv:1407.2275 [cs.CG].
Lipsky, D., Skraba, P., Vejdemo-Johansson, M., 2011. A spectral sequence for parallelized persistence. arXiv:1112.1245 [cs.CG].
Maria, C., Boissonnat, J., Glisse, M., Yvinec, M., 2014. The Gudhi library: simplicial complexes and persistent homology. In: Mathematical Software – ICMS 2014 – 4th International Congress, Proceedings. Seoul, South Korea, August 5–9, 2014, pp. 167–174.
Milosavljevic, N., Morozov, D., Skraba, P., 2011. Zigzag persistent homology in matrix multiplication time. In: Proceedings of the 27th ACM Symposium on Computational Geometry. Paris, France, June 13–15, 2011, pp. 216–225.
Mischaikow, K., Nanda, V., 2013. Morse theory for filtrations and efficient computation of persistent homology. Discrete Comput. Geom. 50 (2), 330–353. http://dx.doi.org/10.1007/s00454-013-9529-6.
Morozov, D., 2010. Dionysus. URL http://www.mrzv.org/software/dionysus.

Nanda, V., 2013. Perseus: the persistent homology software. Accessed 30/01/15. URL http://www.sas.upenn.edu/~vnanda/perseus.
Wagner, H., Dłotko, P., 2014. Towards topological analysis of high-dimensional feature spaces. Comput. Vis. Image Underst. 121, 21–26. http://dx.doi.org/10.1016/j.cviu.2014.01.005.
Zomorodian, A., Carlsson, G.E., 2005. Computing persistent homology. Discrete Comput. Geom. 33 (2), 249–274. http://dx.doi.org/10.1007/s00454-004-1146-y.