Clear and Compress: Computing Persistent Homology in Chunks

Ulrich Bauer[*]   Michael Kerber[†]   Jan Reininghaus[‡]

Abstract

We present a parallelizable algorithm for computing the persistent homology of a filtered chain complex. Our approach differs from the commonly used reduction algorithm by first computing persistence pairs within local chunks, then simplifying the unpaired columns, and finally applying standard reduction on the simplified matrix. The approach generalizes a technique by Günther et al., which uses discrete Morse Theory to compute persistence; we derive the same worst-case complexity bound in a more general context. The algorithm employs several practical optimization techniques which are of independent interest. Our sequential implementation of the algorithm is competitive with state-of-the-art methods, and we improve the performance through parallelized computation.

1   Introduction

Persistent homology has developed from a theoretical idea to an entire research area within the field of computational topology. One of its core features is its multi-scale approach to analyzing and quantifying topological features in data. Recent examples of application areas are shape classification [2], topological denoising [1], or developmental biology [7].

A second major feature of persistent homology is the existence of a simple yet efficient computation method: The standard reduction algorithm as described in [8, 17] computes the persistence pairs by a simple sequence of column operations; Algorithm 1 gives a complete description in just 10 lines of pseudo-code. The worst-case complexity is cubic in the input size of the complex, but the practical behavior has been observed to be closer to linear on average. Various variants have been proposed in order to improve the theoretical bounds [14, 3] or the practical behavior [5].

Our first contribution consists of two simple optimization techniques of the standard reduction algorithm, which we call *clearing* and *compression*. Both approaches exploit the special structure of a filtered chain complex in order to significantly reduce the number of operations on real-world instances. However, the two methods cannot be easily combined because they require the columns of the boundary matrix to be processed in different orders.

---

[*]Institute of Science and Technology Austria (IST Austria), Klosterneuburg, Austria

[†]IST Austria; Stanford University, CA, USA; Max-Planck-Center for Visual Computing and Communication, Saarbrücken, Germany

[‡]Institute of Science and Technology Austria (IST Austria), Klosterneuburg, Austria

Our second contribution is a novel algorithm that incorporates both of the above optimization techniques, and is also suitable for parallelization. It proceeds in three steps: In the first step, the columns of the matrix are partitioned into consecutive *chunks*. Each chunk is reduced independently, applying the clearing optimization mentioned above. In this step, the algorithm finds at least all persistence pairs with (index) persistence less than the size of the smallest chunk; let g be the number of columns not paired within this first step. In the second step, the g unpaired columns are compressed using the method mentioned above. After compression, each column has at most g non-zero entries and the unpaired columns form a nested $(g \times g)$-matrix. In the third and final step, this nested matrix is reduced, again applying the clearing optimization.

The chunk algorithm is closely related to two other methods for computing persistence. First of all, the *spectral sequence algorithm* [6, §VII.4] decomposes the matrix into blocks and proceeds in several phases, computing in phase r the persistence pairs lying $r - 1$ blocks apart. The first step of the chunk algorithm is equivalent to applying the first two phases of the spectral sequence approach. Furthermore, the three step chunk algorithm is inspired by the approach of Günther et al. [10], which combines persistence computation and discrete Morse theory for 3D image data. The first step of that algorithm consists in constructing a discrete gradient field consistent with the input function; such a gradient field can be interpreted as a set of persistence pairs that are incident in the complex and have persistence 0. We replace this method by local persistence computations, allowing us to find pairs with are not incident in the complex.

We analyze the chunk algorithm in terms of time complexity. Let n be the number of generators (simplices, cells) of the chain complex, m the number of chunks, ℓ the maximal size of a chunk, and g as above. We obtain a worst-case bound of

$$O(m\ell^3 + g\ell n + g^3),$$

where the three terms reflect the worst-case running times of the three steps. For the filtration of a cubical complex induced by a d-dimensional grayscale image (with d some fixed constant), this bound simplifies to

$$O(gn + g^3),$$

if the chunks are given by the cells appearing simultaneously in the filtration. This bound improves on the previous general bound of $O(g^2 n \log n)$ from [4]. Moreover, it matches the bound in [10], but applies to arbitrary dimensions. Of course, the bound is still cubic if expressed only in terms of n because $g \in O(n)$ in the worst case.

We implemented a sequential and a parallelized version of the chunk algorithm; both are publicly available in our new PHAT library (`http://phat.googlecode.com/`). The sequential code already outperforms the standard reduction algorithm and is competitive to other variants with a good practical behavior. The parallelized version using 12 cores yields a speed-up factor between 3 and 11 (depending on the example) in our tests, making the implementation the fastest among the considered choices. This is the first result where the usefulness of parallelization is shown for the problem of persistence computation through practical experiments.

2   Background

This section summarizes the theoretical foundations of persistent homology as needed in this work. We limit our scope to simplicial homology over $\mathbb{Z}_2$ just for the sake of simplicity in the

description; our methods generalize to chain complexes over arbitrary fields.

Homology Homology is an algebraic tool for analyzing the connectivity of topological spaces. Let K be a simplicial complex of dimension d. In any dimension p, we call a *p-chain* a formal sum of the p-simplices of K with $\mathbb{Z}_2$ coefficients. The p-chains form a group called the *pth chain group C_p*. The *boundary* of a p-simplex σ is the $(p-1)$-chain formed by the sum of all faces of σ of codimension 1. This operation extends linearly to a *boundary operator $\delta : C_p \to C_{p-1}$*. A p-chain γ is a *p-cycle* if $\delta(\gamma) = 0$. The p-cycles form a subgroup of the p-chains, which we call the *pth cycle group Z_p*. A p-chain γ is called a *p-boundary* if $\gamma = \delta(\xi)$ for some $(p+1)$-chain ξ. Again, the p-boundaries form a group B_p, and since $\delta(\delta(\xi)) = 0$ for any chain ξ, p-boundaries are p-cycles, and so B_p is a subgroup of Z_p. The *pth homology group H_p* is defined as the quotient group Z_p/B_p. The rank of H_p is denoted by β_p and is called the *pth Betti number*. In our case of $\mathbb{Z}_2$ coefficients, the homology group is a vector space isomorphic to $\mathbb{Z}_2^{\beta_p}$, hence it is completely determined by the Betti number. Roughly speaking, the Betti numbers in dimension 0, 1, and 2 yield the number of connected components, tunnels, and voids of K, respectively.

Persistence Let $\{\sigma_1, \ldots, \sigma_n\}$ denote the simplices of K. We assume that for each $i \leq n$, $K_i := \{\sigma_1, \ldots, \sigma_i\}$ is a simplicial complex again. The sequence of inclusions $\emptyset = K_0 \subset \ldots \subset K_i \ldots \subset K_n = K$ is called a *simplexwise filtration* of K. For every dimension p and every K_i, we have a homology group $H_p(K_i)$; we usually consider all dimensions at once and write $H(K_i)$ for the direct sum of the homology groups of K_i in all dimensions. The inclusion $K_i \hookrightarrow K_{i+1}$ induces a homomorphism $g_i^{i+1} : H(K_i) \to H(K_{i+1})$ on the homology groups. These homomorphisms compose and we can define $g_i^j : H(K_i) \to H(K_j)$ for any $i \leq j$. We say that a class $\alpha \in H(K_\ell)$ is *born at (index) i* if $\alpha \in \operatorname{im} g_i^\ell$ but $\alpha \notin \operatorname{im} g_{i-1}^\ell$. A class α born at index i *dies entering (index) j* if $g_i^j(\alpha) \in \operatorname{im} g_{i-1}^j$ but $g_i^{j-1}(\alpha) \notin \operatorname{im} g_{i-1}^{j-1}$. In this case, the index pair (i, j) is called a *persistence pair*, and the difference $j - i$ is the *(index) persistence* of the pair. The transition from K_{i-1} to K_i either causes the birth or the death of an homology class. We call the added simplex σ_i *positive* if it causes a birth and *negative* if it causes a death. Note that homology classes of the full complex K do not die during the filtration. We call a simplex σ_i that gives birth to such a class *essential*. All other simplices are called *inessential*.

Boundary matrix For a matrix $M \in \mathbb{Z}_2^{n \times n}$, we let M_j denote its j-th column, M^i its i-th row, and $M_j^i \in \mathbb{Z}_2$ its entry in row i and column j. For a non-zero column $0 \neq M_j = (m_1, \ldots, m_n) \in \mathbb{Z}_2^n$, we set $\operatorname{pivot}(M_j) := \max\{i = 1, \ldots, n \mid m_i = 1\}$ and call it the *pivot index* of that column.

The *boundary matrix $D \in (\mathbb{Z}_2)^{n \times n}$* of a simplexwise filtration $(K_i)_i$ is a $n \times n$ matrix with $D_j^i = 1$ if and only if σ_i is a face of σ_j of codimension 1. In other words, the jth column of D encodes the boundary of σ_j. D is an upper-triangular matrix because any face of σ_j must precede σ_j in the filtration. Since the jth row and column of D corresponds to the jth simplex σ_j of the filtration, we can talk about *positive columns*, *negative columns*, and *essential columns* in a natural way, and similarly for rows.

The reduction algorithm A column operation of the form $M_j \leftarrow M_j + M_k$ is called *left-to-right* if $k < j$. We call a matrix M' *derived from M* if M can be transformed into M' by left-to-right operations. Note that in a derivation M' of M, the jth column can be expressed as

a linear combination of the columns $1, \ldots, j$ of M, and this linear combination includes M_j. We call a matrix R *reduced* if no two non-zero columns have the same pivot index. If R is derived from M, we call it a *reduction of M*. In this case, we define

$$
\begin{aligned}
P_R &:= \{(i,j) \mid R_j \neq 0 \wedge i = \text{pivot}(R_j)\} \\
E_R &:= \{i \mid R_i = 0 \wedge \text{pivot}(R_j) \neq i \forall j = 1, \ldots, n\}.
\end{aligned}
$$

Although the reduction matrix R is not unique, the sets P_R and E_R are the same for any choice of reduction; therefore, we can define P_M and E_M to be equal to P_R and E_R for any reduction R of M. We call the set P the *persistence pairs* of M. When obvious from the context, we omit the subscripts and simply write P for the persistence pairs. For the boundary matrix D of K, the pairs $(i,j) \in P$ are the persistence pairs of the filtration $(K_i)_{0 \leq i \leq n}$, and the indices in E correspond to the essential simplices of the complex. Note that E is uniquely determined by P and n as the indices between 1 and n that do not appear in any pair of P.

The simplest way of reducing D is to process columns from left to right; for every column, other columns are added from the left until the pivot index is unique (Algorithm 1). A lookup table can be used to identify the next column to be added in constant time. A flag is used for every column denoting whether a persistence pair with the column index has already been found. After termination, the unpaired columns correspond to the essential columns. The running time is at most cubic in n, and this bound is actually tight for certain input filtrations, as demonstrated in [16].

Algorithm 1 Left-to-right persistence computation

1: **procedure** PERSISTENCE_LEFT_RIGHT(D)
2: $\quad R \leftarrow D; L \leftarrow [0, \ldots, 0]; P \leftarrow \emptyset$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad \triangleright L \in \mathbb{Z}^n$
3: $\quad$ **for** $j = 1, \ldots, n$ **do**
4: $\qquad$ **while** $R_j \neq 0$ and $L[\text{pivot}(R_j)] \neq 0$ **do**
5: $\qquad\qquad R_j \leftarrow R_j + R_{L[\text{pivot}(R_j)]}$
6: $\qquad$ **if** $R_j \neq 0$ **then**
7: $\qquad\qquad i \leftarrow \text{pivot}(R_j)$
8: $\qquad\qquad L[i] \leftarrow j$
9: $\qquad\qquad$ Mark columns i and j as paired and add (i,j) to P
10: $\quad$ **return** P

Let M be derived from D. A column M_j of M is called *reduced* if either it is zero, or if $(i,j) \in P$ with $i = \text{pivot}(M_j)$. With this definition, a matrix M is a reduction of D if and only if every column is reduced.

3 Speed-ups

Algorithm 1 describes the simplest way of reducing the boundary matrix, but it performs more operations than actually necessary to compute the persistence pairs. We now present two simple techniques which both lead to a significant decrease in the number of required operations.

Clearing positive columns The key insight behind our first optimization is the following fact: if i appears as the pivot in a reduced column of M, the index i is positive and hence there

exists a sequence of left-to-right operations on M_i that turn it to zero. Instead of explicitly executing this sequence of operations, we define the *clear* operation by setting column M_i to zero directly. Informally speaking, a clear is a shortcut to avoid some column operations in the reduction when the result is evident.

In order to apply this optimization, we change the traversal order in the reduction by first reducing the columns corresponding to simplices with dimension d (from left to right), then all columns with dimension $d-1$, and so on. After having reduced all columns with dimension δ, we have found all positive inessential columns with dimension $\delta - 1$ and clear them before continuing with $\delta - 1$. This way all positive inessential columns of the complex are cleared without performing any column additions on them. See [4] for a more detailed description.

Compression   Alternatively, we can try to save arithmetic operations by reducing the number of non-zero rows among the unpaired columns. A useful observation in this context is given next.

Lemma 1. *Let M_j be a non-zero column of M with $i = \mathrm{pivot}(M_j)$. Then M_i is a positive and inessential column.*

Proof. The statement is clearly true if M_j is reduced, because in this case (i, j) is a persistence pair. If M_j is not reduced, this means that after applying some sequence of left-to-right column operations, some reduced column has i as pivot index. □

Corollary 2. *Let M_i be a negative column of M. Then i is not the pivot index of any column in M.*

As a consequence, whenever a negative column with index j has been reduced, row j can be set to zero before further reducing.

Corollary 3. *Let M_i be a negative column and let M_j be a column with $M_j^i = 1$. Then setting M_j^i to zero does not affect the pairs.*

We can even do more: let i be the pivot index of the reduced column M_j and assume that the submatrix of M with column indices $\{1, \ldots, j\}$ and row indices $\{i, \ldots, n\}$ is reduced, i.e., the pivot indices are unique in this submatrix. By adding column j to each unreduced column in the matrix that has a non-zero entry at row i, we can eliminate all non-zero entries in row i from the unreduced columns. Note that if $k < j$ and $M_k^i \neq 0$, then $\mathrm{pivot}(M_k) \geq i$ and thus, by assumption, M_k must be a reduced negative column. Therefore, for each unreduced column M_k, the operation $M_k \leftarrow M_k + M_j$ is a left-to-right addition and thus does not affect the pairs.

4   Reduction in chunks

The two optimization techniques from Section 3 both yield significant speed-ups, but they are not easily combinable, because clearing requires to process a simplex before its faces, whereas compression works in the opposite direction. In this section, we present an algorithm which combines both optimization techniques.

Let $m \in \mathbb{N}$. Fix $m+1$ numbers $0 = t_0 < t_1 < \ldots < t_{m-1} < t_m = n$ and define the ith *chunk* of D to be the columns of D with indices $\{t_{i-1}+1, \ldots, t_i\}$. We call a column D_j *local* if it forms a persistence pair with another column in the same chunk or in one of the adjacent chunks. In this case, we also call the persistence pair local. Non-local columns (and pairs) are

called *global*. If ℓ is a lower bound on the size of each chuck, then every global persistence pair has index persistence at least ℓ. We also call an index j local if the jth column of D is local, and the same for global. We denote the number of global columns in D by g. The high-level description of our new algorithm consists of three steps:

1. Partially reduce every chunk independently, applying the clearing optimization, so that all local columns are completely reduced.

2. Independently compress every global column such that all its non-zero entries are global.

3. Reduce the submatrix consisting only of the global rows and columns.

We give details about the three steps in the rest of this section. The first two steps can be performed in parallel, whereas the third step only needs to reduce a matrix of size $g \times g$ instead of $n \times n$. In many situations, g is significantly smaller than n.

Local chunk reduction The first step of our algorithm computes the local pairs by performing two phases of the spectral sequence algorithm [6, §VII.4]. Concretely, we apply left-to-right operations as usual, but in the first phase we only add columns from the same chunk, and in the second phase we only add columns from both the same chunk and its left neighbor. After phase r, for each $b \in \{r, \ldots, m\}$ the submatrix with column indices $\{1, \ldots, t_b\}$ and row indices $\{t_{b-2} + 1, \ldots, n\}$ is reduced. If the reduction of column j stops at a pivot index $i > t_{b-r}$, row j cannot be reduced any further by adding any column, so we identify (i, j) as a local persistence pair. Conversely, any local pair (i, j) is detected by this method after two phases. We incorporate the clearing operation for efficiency, that is, we proceed in decreasing dimension and set detected local positive columns to zero; see Algorithm 2. After its execution, $L[i]$ contains the index of the local negative column with pivot index i for any local positive column i, and the resulting matrix R is a derivation of D in which all local columns are reduced.

Algorithm 2 Local chunk reduction

 1: **procedure** LOCAL_REDUCTION$(M, t_0, \ldots, t_m)$
 2:     $R \leftarrow M; L \leftarrow [0, \ldots, 0]; P \leftarrow \emptyset$ $\triangleright L \in \mathbb{Z}^n$
 3:     **for** $\delta = d, \ldots, 0$ **do**
 4:         **for** $r = 1, 2$ **do** $\triangleright$ Perform two phases of the spectral sequence algorithm
 5:             **for** $b = r, \ldots, m$ **do** $\triangleright$ Loop is parallelizable
 6:                 **for** $j = t_{b-1} + 1, \ldots, t_b$ with $\dim \sigma_j = \delta$ **do**
 7:                     **if** j is not marked as paired **then**
 8:                         **while** $R_j \neq 0 \wedge L[\mathrm{pivot}(R_j)] \neq 0 \wedge \mathrm{pivot}(R_j) > t_{b-r}$ **do**
 9:                             $R_j \leftarrow R_j + R_{L[\mathrm{pivot}(j)]}$
10:                         **if** $R_j \neq 0$ **then**
11:                             $i \leftarrow \mathrm{pivot}(R_j)$
12:                             **if** $i > t_{b-r}$ **then**
13:                                 $L[i] \leftarrow j$
14:                                 $R_j \leftarrow 0$ $\triangleright$ Clear column i
15:                                 Mark i and j as paired and add (i, j) to P
16:     **return** (R, L, P)

Algorithm 3 Determining active entries

1: **procedure** MARK_ACTIVE_ENTRIES(R)
2:     **for** each unpaired column k **do**   ▷ Loop is parallelizable
3:         MARK_COLUMN(R, k)
4: **function** MARK_COLUMN(R, k)
5:     **if** k is marked as active/inactive **then return** true/false
6:     **for** each non-zero entry index i of R_k **do**
7:         **if** ℓ is unpaired **then**
8:             mark k as active and **return** true
9:         **else if** i is positive **then**
10:             $j \leftarrow L[i]$ ▷ (i, j) is persistence pair
11:             **if** $j \neq k$ and MARK_COLUMN(R, j) **then**
12:                 mark k as active and **return** true
13:     mark k as inactive and **return** false

Global column compression Let R be the matrix returned by Algorithm 2. Before computing the global persistence pairs, we first compress the global columns, using the ideas from Section 3; recall that negative rows can simply be set to zero, while entries in positive rows can be eliminated by an appropriate column addition. Note, however, that a full column addition might actually be unnecessary: for instance, if all non-zero row indices in the added column belong to negative columns (except for the pivot), the entry in the local positive row could just have been zeroed out in the same way as in Corollary 3. Speaking more generally, it is more efficient to avoid column additions that have no consequences for global indices, neither directly nor indirectly.

In the spirit of this observation, we call an index i *inactive* if either it is a local negative index or if (i, j) is a local pair and all indices of non-zero entries in column R_j apart from i are inactive. Otherwise, the index is called *active*. By induction and Corollary 3, we can show:

Lemma 4. *Let i be an inactive index and let M_j be any column with $M_j^i = 1$. Then setting M_j^i to zero does not affect the persistence pairs.*

The compression proceeds in two steps: first, every non-zero entry of a global column is classified as active or inactive (using depth-first search; see Algorithm 3). Then, we iterate over the global columns, set all entries with inactive index to zero, and eliminate any non-zero entry with a local positive index ℓ by column addition with $L[\ell]$ (see Algorithm 4). After this process, we obtain a matrix R' with the same persistence pairs as R, such that the global columns of R' have non-zero entries only in the global rows.

Submatrix reduction After having compressed all global columns, these form a $g \times g$ matrix "nested" in R (recall that g is the number of global columns). To complete the computation of the persistence pairs, we simply perform standard reduction on the remaining matrix. For efficiency, we perform steps 2 and 3 alternatingly for all dimensions in decreasing order and apply the clearing optimization; this way, we avoid the compression of positive global columns. Algorithm 5 summarizes the whole method.

Algorithm 4 Global column compression

1: **procedure** COMPRESS(R, k)
2:     **Uses variables:** L
3:     **for** each non-zero entry index ℓ of R_k in decreasing order **do**
4:         **if** ℓ is paired **then**
5:             **if** ℓ is inactive **then**
6:                 $R_\ell^k \leftarrow 0$
7:             **else**
8:                 $j \leftarrow L[\ell]$ ▷ (ℓ, j) is persistence pair
9:                 $R_k \leftarrow R_k + R_j$

Algorithm 5 Persistence in chunks

1: **procedure** PERSISTENCE_IN_CHUNKS($D, t_0, \ldots, t_m$)
2:     $(R, L, P) \leftarrow$ LOCAL_REDUCTION($D, t_0, \ldots, t_m$)   ▷ step 1: reduce local columns
3:     MARK_ACTIVE_ENTRIES(R)
4:     **for** $\delta = d, \ldots, 0$ **do**
5:                                 ▷ step 2: compress global columns
6:         **for** $j = 1, \ldots, n$ with $\dim \sigma_j = \delta$ **do**   ▷ Loop is parallelizable
7:             **if** column j is not paired **then**
8:                 COMPRESS(R, j)
9:         **for** $j = 1, \ldots, n$ with $\dim \sigma_j = \delta$ **do**   ▷ step 3: reduce global columns
10:             **while** $R_j \neq 0 \wedge L[\text{pivot}(R_j)] \neq 0$ **do**
11:                 $R_j \leftarrow R_j + R_{L[\text{pivot}(j)]}$
12:             **if** $R_j \neq 0$ **then**
13:                 $i \leftarrow \text{pivot}(R_j)$
14:                 $L[i] \leftarrow j$
15:                 $R_i \leftarrow 0$ ▷ Clear column i
16:                 Mark i and j as paired and add (i, j) to P
17:     **return** P

---

5 Analysis

Algorithm 5 permits a complexity analysis depending on the following parameters: n, the number of simplices; m, the number of chunks; ℓ, the maximal size of a chunk; and g, the number of global columns. We assume that for any simplex, the number of non-zero faces of codimension 1 is bounded by a constant (this is equivalent to assuming that the dimension of the complex is a constant).

General complexity   We show that the complexity of Algorithm 5 is bounded by

$$O(m\ell^3 + g\ell n + g^3). \tag{1}$$

The three summands correspond to the running times of the three steps[1]. Note that $g \in O(n)$ in the worst case.

For the complexity of Algorithm 2, we consider the complexity of reducing one chunk, which consists of up to ℓ columns. Within the local chunk reduction, every column is only added with columns of the same or the previous chunk, so there are only up to 2ℓ column additions per column. Moreover, since the number of non-zero entries per column in D is assumed to be constant, there are only $O(\ell)$ many entries that can possibly become non-zero during the local chunk reduction. It follows that the local chunk reduction can be considered as a reduction on a matrix with ℓ columns and $O(\ell)$ rows. If we represent columns by linked lists (containing the non-zero indices in sorted order), one column operation can be done in $O(\ell)$ primitive operations, which leads to a total complexity of $O(\ell^3)$ per chunk.

The computation of active columns in Algorithm 3 is done by depth-first search on a graph whose vertices are given by the columns and whose edges correspond to their non-zero entries. The number of edges is $O(n\ell)$, so we obtain a running time of $O(n\ell)$.

Next, we consider the cost of compressing a global column with index j. After the previous step, the column has at most $O(\ell)$ non-zero entries. We transform the presentation of the column from a linked list into a bit vector of size n. In this representation, adding another column in list representation with v entries to column j takes time proportional to v. In the worst case, we need to add all columns with indices $1,\ldots,j-1$ to j. Each such column has $O(\ell)$ entries. At the end, we transform the bit vector back into a linked list representation. The total cost is $O(n + (j-1)\ell + n) = O(n\ell)$ per global column.

Finally, the complexity of the global reduction is $O(g^3)$, as in the standard reduction.

Choosing chunks We discuss different choices of chunk size and their complexities. A generic choice for an arbitrary complex is to choose $O(\sqrt{n})$ chunks of size $O(\sqrt{n})$ each. With that, the complexity of (1) becomes

$$O(n^2 + g_1 n \sqrt{n} + g_1^3).$$

Alternatively, choosing $O(\frac{n}{\log n})$ chunks of size $O(\log n)$, the complexity becomes

$$O(n \log^2 n + g_2 n \log n + g_2^3).$$

We replaced g by g_1 and g_2 to express that the number of global columns is different in both variants. In general, choosing larger chunks is likely to produce less global columns, since every global persistence pair has index persistence at least ℓ (the size of the smallest chunk) .

Cubical complexes We consider an important special case of boundary matrices: consider a d-dimensional image with p hypercubes, where each vertex contains a grayscale value. We assume that the cubes are triangulated conformally in order to get simplicial input – the argument also works, however, for the case of cubical cells. We assign function values inductively, assigning to each simplex the maximal value of its faces. Assuming that all vertex values are distinct, the *lower star* of vertex v is the set of all simplices which have the same function value as v. Filtering the simplices in a way that respects the order of the function values, we get a *lower star filtration* of the image. Now choose the lower stars as the chunks in

---

[1]The running time of the third step could be lowered to g^ω, where ω is the matrix-multiplication exponent, using the method of [14].

our reduction algorithm. Note that the lower star is a subset of the star of the corresponding vertex, which is of constant size (assuming that the dimension d is constant). Therefore, the complexity bound (1) reduces to

$$O(n + gn + g^3) = O(gn + g^3).$$

Note that global columns with large index persistence might still have very small, or even zero, persistence with respect to the function values, for instance in the presence of a flat region in the image where many vertices have similar values.

6 Experiments

We implemented two versions of the algorithm presented in Section 4: a sequential and a parallel version (using OPENMP), in which the first two steps of the algorithm are performed simultaneously on each chunk and on each global column, respectively. In both cases, we use $\lfloor \sqrt{n} \rfloor$ as the chunk size. For a fair comparison, we also re-implemented the algorithms introduced in [4, 8] in the same framework, that means, using the same data representations and low-level operations such as column additions. Our implementation is publicly available in our new *PHAT* library for computing persistence homology, available at `http://phat.googlecode.com/`. Additionally, we compare to the memory efficient algorithm [10] based on discrete Morse theory [9] and to the implementation of the persistent cohomology algorithm [5] found in the DIONYSUS library [15].

To find out how these algorithms behave in practice, we apply them to five representative data sets. The first three are 3D image data sets with a resolution of 128^3. The first of these is given by a Fourier sum with random coefficients and is representative of smooth data. The second is uniform noise. The third is the sum of the first two and represents large-scale structures with some small-scale noise. These data sets are illustrated in Figure 1 by an isosurface.

a)   b)   c)
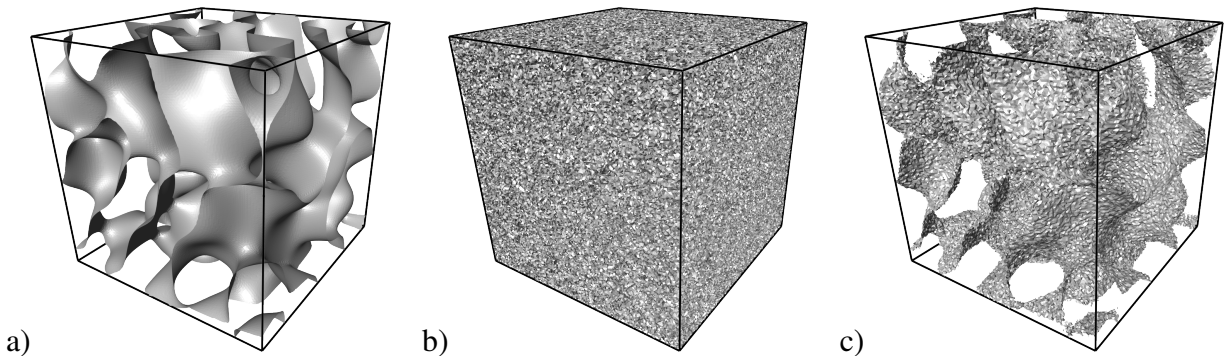
Figure 1: A single isosurface of the representative data sets used to benchmark and compare our algorithm: a) a smooth data set, b) uniformly distributed noise, c) the sum of a) and b).

In addition to the lower star filtrations of these image data sets, we also consider an alpha shape filtration defined by 10000 samples of a torus embedded in $\mathbb{R}^3$, and the 4-skeleton of the Rips filtration given by 50 points randomly chosen from the Mumford data set [11].

As pointed out in [5], the pairs of persistent cohomology are the same as those of persistent homology. We therefore also applied all algorithms to the corresponding cochain filtration, given in matrix form by transposing the boundary matrix and reversing the order of the

columns; this operation is denoted by $(\cdot)^{\perp}$. When reducing such a coboundary matrix with the clearing optimization, columns are processed in order of increasing instead of decreasing dimension.

Table 1 contains the running times of the above algorithms applied to filtrations of these five data sets run on a PC with two Intel Xeon E5645 CPUs. We can observe a huge speed-up caused by the clearing optimization as already reported in [4]. We can see that our chunk algorithm performs slightly worse than the one of [4] when executed sequentially, but faster when parallelized. We also observe that standard, twist, and chunk generally behave worse when computing persistent cohomology, except for the Rips filtration of the Mumford data set, where the converse is true.

Dataset	$n \cdot 10^{-6}$	std. [8]	twist [4]	cohom. [5]	DMT [10]	g/n	chunk (1x)	chunk (12x)
Smooth	16.6	383s	3.1s	65.8s	2.0s	0%	5.0s	0.9s
Smooth$^{\perp}$	16.6	432s	11.3s	20.8s	–	0%	6.3s	0.9s
Noise	16.6	336s	17.2s	15971s	13.0s	9%	28.3s	6.3s
Noise$^{\perp}$	16.6	1200s	29.0s	190.1s	–	9%	31.1s	5.8s
Mixed	16.6	330s	5.8s	50927s	12.3s	5%	21.6s	2.4s
Mixed$^{\perp}$	16.6	446s	13.0s	32.7s	–	5%	32.0s	2.9s
Torus	0.6	52s	0.3s	1.6s	–	7%	0.3s	0.1s
Torus$^{\perp}$	0.6	24s	0.3s	1.4s	–	7%	0.9s	0.2s
Mumford	2.4	38s	35.2s	2.8s	–	82%	14.6s	1.8s
Mumford$^{\perp}$	2.4	58s	0.2s	184.1s	–	82%	1.5s	0.4s

Table 1: Running time comparison of various persistent homology algorithms applied to the data sets described in Section 6. The last three columns contain information of the algorithm presented in this paper: the fraction of global columns g/n, and the running times using one and twelve cores, respectively.

7 Conclusion and Outlook

We have presented an algorithm for persistent homology that includes two simple optimization techniques into the reduction algorithm. It can be fully parallelized, except for the reduction of compressed global columns, whose number is often small. Besides our asymptotic complexity bounds, which give a detailed dependence on the parameters of the algorithm, our experiments show that significant speed-ups can be achieved through parallelized persistence computation. Similar observations have been made recently by Lewis and Zomorodian [12] for the computation of (non-persistent) homology; see also [13]. We plan a more extensive discussion of the practical effects of our optimizations and parallelization in an extended version of this paper.

References

[1] U. Bauer, C. Lange, and M. Wardetzky. Optimal topological simplification of discrete functions on surfaces. *Discrete and Computational Geometry*, 47:347–377, 2012.

[2] F. Chazal, D. Cohen-Steiner, L. Guibas, F. Memoli, and S. Oudot. Gromov–Hausdorff stable signatures for shapes using persistence. In *Eurographics Symposium on Geometry Processing*, pages 1393–1403, 2009.

[3] C. Chen and M. Kerber. An output-sensitive algorithm for persistent homology. In *Proceedings of the 27th Annual Symposium on Computational Geometry*, pages 207–215, 2011.

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

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

[6] H. Edelsbrunner and J. Harer. *Computational Topology, An Introduction*. American Mathematical Society, 2010.

[7] H. Edelsbrunner, C.-P. Heisenberg, M. Kerber, and G. Krens. The medusa of spatial sorting: Topological construction. arXiv:1207.6474, 2012.

[8] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. *Discrete and Computational Geometry*, 28:511–533, 2002.

[9] R. Forman. Morse theory for cell complexes. *Advances in Mathematics*, 134:90–145, 1998.

[10] D. Günther, J. Reininghaus, H. Wagner, and I. Hotz. Efficient computation of 3D Morse–Smale complexes and persistent homology using discrete Morse theory. *The Visual Computer*, pages 1–11, 2012.

[11] A. B. Lee, K. S. Pedersen, and D. Mumford. The nonlinear statistics of high-contrast patches in natural images. *International Journal of Computer Vision*, 54:83–103, 2003.

[12] R. H. Lewis and A. Zomorodian. Multicore homology. Manuscript, 2012.

[13] D. Lipsky, P. Skraba, and M. Vejdemo-Johansson. A spectral sequence for parallelized persistence. arXiv:1112.1245, 2011.

[14] N. Milosavljević, D. Morozov, and P. Škraba. Zigzag persistent homology in matrix multiplication time. In *Proceedings of the 27th Annual Symposium on Computational Geometry*, pages 216–225, 2011.

[15] D. Morozov. Dionysus: a C++ library for computing persistent homology. `http://www.mrzv.org/software/dionysus/`.

[16] D. Morozov. Persistence algorithm takes cubic time in the worst case. In *BioGeometry News*. Duke Computer Science, Durham, NC, 2005.

[17] A. Zomorodian and G. Carlsson. Computing persistent homology. *Discrete and Computational Geometry*, 33:249–274, 2005.