Distributed computation of persistent homology

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

October 3, 2013

Abstract

Persistent homology is a popular and powerful tool for capturing topological features of data. Advances in algorithms for computing persistent homology have reduced the computation time drastically – as long as the algorithm does not exhaust the available memory. Following up on a recently presented parallel method for persistence computation on shared memory systems, we demonstrate that a simple adaption of the standard reduction algorithm leads to a variant for distributed systems. Our algorithmic design ensures that the data is distributed over the nodes without redundancy; this permits the computation of much larger instances than on a single machine. Moreover, we observe that the parallelism at least compensates for the overhead caused by communication between nodes, and often even speeds up the computation compared to sequential and even parallel shared memory algorithms. In our experiments, we were able to compute the persistent homology of filtrations with more than a billion (10^9) elements within seconds on a cluster with 32 nodes using less than 10GB of memory per node.

---

[*]Institute of Science and Technology Austria, Klosterneuburg, Austria. http://ulrich-bauer.org

[†]Max-Planck-Center for Visual Computing and Communication, Saarbrücken, Germany. http://mpi-inf.mpg.de/~mkerber

[‡]Institute of Science and Technology Austria, Klosterneuburg, Austria. http://ist.ac.at/~reininghaus

1   Introduction

Background   A recent trend in data analysis is to understand the shape of data (possibly in high dimensions) using topological methods. The idea is to interpret the data as a growing sequence of topological spaces (a *filtration*), such as for example, sublevel sets of a function with an increasing threshold, or thickenings of a point set. The goal is now to compute a topological summary of the filtration, which can be used, for instance, to identify features, as well as to infer topological properties of a sampled shape. Persistent homology describes how homological features appear and disappear in the filtration (see Section 2 for more details). Besides significant theoretical advances, persistent homology has been used in various applications; see [7] for a survey. The success of persistent homology stems from its generality, which makes it applicable for various forms of data, from its stability with respect to perturbations [2, 5], from its ability to provide information on all scales, and, last but not least, from the availability of efficient algorithms for computing this information.

The standard algorithm for computing persistent homology assumes the input to be a boundary matrix of a chain complex, and proceeds by reducing that matrix using a variant of Gaussian elimination [8, 15]. The running time is cubic in the number of simplices; this can be improved to matrix multiplication time [12] or replaced by an output-sensitive bound [3]. However, on the practical side, it has been observed that the standard algorithm usually performs much better on real-world instances than predicted by the worst-case bounds, and relatively simple optimizations of the standard method yield remarkable speed-ups [4]. Recently, Maria et al. [11] implemented a memory-efficient and comparably fast method for computing persistent cohomology, which yields the same information about birth and death of features as its homology counterpart. Further improvements have been reported by using several processors in a shared memory environment [1] (see also [9, 10] for alternative parallelization schemes). With these optimizations, it is often the case that computing persistence actually takes less time than even reading the input into memory. Therefore, the limiting factor is not so much the time spent for computation but rather the memory available on the computer.

Contribution   We present a scalable algorithm for computing persistent homology in parallel in a distributed memory environment. This method the computation of much larger instances than using existing state-of-the art algorithms on a single machine, by using sufficiently many computing nodes such that the data fits into the distributed memory. While overcoming the memory bottleneck is the primary purpose of our approach, we aim for a time-efficient solution at the same time.

As demonstrated by our experiments, our implementation exhibits excellent scaling with respect to memory usage on a single node, and even outperforms similar parallel shared memory code in running time. This result is somewhat surprising, since the computation of topological properties like persistent homology is a global problem, and at first sight it is not obvious at all that the computation can be performed with the very simple and inexpensive pattern of communication that our algorithm exhibits.

Our method closely resembles the *spectral sequence algorithm* for persistent homology [6, S VII.4]. However, several adaptions are necessary for an efficient implementation in distributed memory. Most importantly, reduced columns are not stored in order of their index in the matrix, but rather according to the order of their *pivot*, the largest index of a non-zero entry. This allows a node to perform eliminations in its associated rows, and to determine if a column with pivot in these rows is reduced, without further communication with other nodes. Furthermore, we minimize the number of messages sent through the network by collecting blocks of messages, and we simplify the communication structure by letting node j only communicate with nodes $j \pm 1$. Finally, we incorporate the clear optimization [4] into the algorithm in order to avoid unnecessary column operations.

Organization   We introduce the necessary background on persistent homology in Section 2, describe our distributed memory algorithm in Section 3, report on experimental evaluation in Section 4, and conclude in Section 5.

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 however easily generalize to chain complexes over arbitrary fields. For a more detailed introduction, we refer to [6, 7, 15].

Homology   Homology is an algebraic invariant for analyzing the connectivity of simplicial complexes. Let K be a finite simplicial complex. For a given dimension d, a *d-chain* is a formal sum of d-simplices of K with $\mathbb{Z}_2$ coefficients. The d-chains form a group $C_d(K)$ under addition. Equivalently, a d-chain can be interpreted as a subset of the d-simplices, with the group operation being the symmetric set difference. The *boundary* of a d-simplex σ is the $(d-1)$-chain formed by the sum of all faces of σ of codimension 1. This operation extends linearly to a *boundary operator* $\partial_d : C_d(K) \to C_{d-1}(K)$. A d-chain γ is a *d-cycle* if $\partial_d(\gamma) = 0$. The d-cycles form a subgroup of the d-chains, denoted by $Z_d(K)$. A d-chain γ is called a *d-boundary* if $\gamma = \partial_d(\xi)$ for some $(d+1)$-chain ξ. Again, the d-boundaries form a subgroup $B_d(K)$ of $C_d(K)$, and since $\partial_d(\partial_d(\xi)) = 0$ for any chain ξ, d-boundaries are d-cycles, and so $B_d(K)$ is a subgroup of $Z_d(K)$. The dth *homology group* $H_d(K)$ is defined as the quotient group $Z_d(K)/B_d(K)$. In fact, the groups $C_d(K)$, $Z_d(K)$, $B_d(K)$, and $H_d(K)$ are $\mathbb{Z}_2$-vector spaces. The dimension of $H_d(K)$ is called the dth *Betti number* β_d. Roughly speaking, the Betti numbers in dimension 0, 1, and 2 count the number of connected components, tunnels, and voids of K, respectively.

Persistence   Consider a *simplexwise filtration* of K, i.e., a sequence of inclusions $\emptyset = K_0 \subset \ldots \subset K_n = K$ such that $K_i = K_{i-1} \cup \{\sigma_i\}$, where σ_i is a simplex of K. We write $H_*(K_i)$ for the direct sum of the homology groups of K_i in all dimensions. For $i \leq j$, the inclusion $K_i \hookrightarrow K_j$ induces a homomorphism $h_i^j : H_*(K_i) \to H_*(K_j)$ on the homology groups. We say that a class $\alpha \in H_*(K_i)$ is *born at (index) i* if

$$\alpha \notin \operatorname{im} h_{i-1}^i.$$

A class $\alpha \in H_*(K_i)$ born at index i *dies entering (index) j* if

$$h_i^{j-1}(\alpha) \notin \operatorname{im} h_{i-1}^{j-1} \text{ but } h_i^j(\alpha) \in \operatorname{im} h_{i-1}^j.$$

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 some homology class. This homology class is not unique in general.

Boundary matrix   For a matrix $M \in (\mathbb{Z}_2)^{n \times n}$, let M_j denote its jth column and $m_{ij} \in \mathbb{Z}_2$ its entry in row i and column j. For a column M_j, we define $\operatorname{pivot}(M_j) = \min\{p \in \mathbb{N}_0 : m_{ij} = 0 \text{ for all } i > p\}$ and call it the *pivot index* of that column. When obvious from the context, we omit explicit mention of the matrix M and write $\operatorname{pivot}(j)$ for $\operatorname{pivot}(M_j)$.

The *boundary matrix* $D \in (\mathbb{Z}_2)^{n \times n}$ of a simplexwise filtration $(K_i)_{i=1}^n$ is the $n \times n$ matrix of the boundary operator $\partial_* : C_*(K) \to C_*(K)$ with respect to the ordered basis $(\sigma_i)_{i=1}^n$ of $C_*(K)$. We have $D_{ij} = 1$ 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, since any face of σ_j must precede σ_j in the filtration.

Matrix reduction A column operation of the form $M_j \leftarrow M_j + M_k$ is called *left-to-right addition* if $k < j$. A left-to-right addition is called *eliminating* if it decreases pivot(j). A column M_j is called *reduced* if pivot(j) cannot be decreased by applying any sequence of left-to-right additions. In particular, there is no non-zero column M_k with $k < j$ and pivot(k) = pivot(j). Clearly a zero column is reduced. Note that a reduced column remains reduced under eliminating left-to-right column additions.

We call a matrix M *reduced* if all columns are reduced, or equivalently, if no two non-zero columns have the same pivot index. We call M *reduced at index* (i, j) if the lower left submatrix of M with rows of index $> i$ and columns of index $\leq j$ is reduced. A sufficient condition for column M_j to be reduced is that M is reduced at index (i, j) with $i = $ pivot(j).

If R is a reduced matrix obtained by applying left-to-right additions to M, we call it a *reduction of M*. In this case, we define

$$P_R := \{(i, j) \mid i = \text{pivot}(R_j) > 0\}$$

Although the reduction matrix R is not unique, the set P_R is the same for any reduction of M; therefore, we can define P_M to be equal to P_R for any reduction R of M.

Persistence by reduction For the boundary matrix D of the filtration $(K_i)_{i=1}^n$, the first i columns generate the boundary group $B_*(K_i)$. This property is invariant under left-to-right column additions. For a reduction R of D, the non-zero columns among the first i columns actually form a basis of $B_*(K_i)$. Note that

$$i = \dim C_*(K_i) = \dim Z_*(K_i) + \dim B_*(K_i) \text{ and}$$
$$\dim H_*(K_i) = \dim Z_*(K_i) - \dim B_*(K_i).$$

Hence, if R_i is zero, we have

$$\dim B_*(K_i) = \dim B_*(K_{i-1}),$$
$$\dim Z_*(K_i) = \dim Z_*(K_{i-1}) + 1, \text{ and}$$
$$\dim H_*(K_i) = \dim H_*(K_{i-1}) + 1,$$

and so some homology class is born at i. If on the other hand R_j is non-zero with $i = $ pivot(j), we have

$$\dim B_*(K_i) = \dim B_*(K_{i-1}) + 1,$$
$$\dim Z_*(K_i) = \dim Z_*(K_{i-1}), \text{ and}$$
$$\dim H_*(K_i) = \dim H_*(K_{i-1}) - 1.$$

The fact that R_j has pivot i means that $R_j \in Z_*(K_i)$ and hence

$$[R_j] \in H_*(K_i);$$

the fact that it is reduced means that there is no $b \in B_*(K_{j-1})$ with $b + R_j \in Z_*(K_{i-1})$ and hence

$$[R_j] \notin \text{im } h_{i-1}^i.$$

We conclude that $[R_j]$ is born at i. We even have

$$[R_j] \notin \text{im } h_{i-1}^{j-1}.$$

Moreover, R_j is a boundary in K_{j-1}, and so

$$[R_j] = 0 \in \text{im } h_{i-1}^j.$$

We conclude that the pairs $(i, j) \in P_D$ are the persistence pairs of the filtration.

The standard way to reduce D is to process columns from left to right; for every column, previously reduced columns are added from the left until the pivot index is unique. A lookup table can be used to identify the next column to be added in constant time. The running time is at most cubic in n, and this bound is actually tight for certain input filtrations, as demonstrated in [13].

3

Clearing optimization   Despite its worst-case behavior, there are techniques to speed up the reduction significantly in practice. A particularly simple yet powerful improvement has been presented in [4]. It is based on the following observations.

First, the reduction of the matrix can be performed separately for each dimension d, by restricting to the submatrix corresponding to columns of dimension d and rows of dimension $d - 1$. This submatrix is exactly the matrix of the boundary operator $\partial_d : C_p(K) \rightarrow C_{p-1}(K)$. The second basic fact to note is that in any reduction of D, if i is a pivot of some column j, the ith column is zero.

This leads to the following variant of the reduction algorithm: the boundary matrix is reduced separately in each dimension in decreasing order. After the reduction in dimension d, all columns corresponding to pivots indices are set to zero – we call this process *clearing*. Note that columns corresponding to d-simplices have pivots corresponding to $d - 1$-simplices. After clearing, we proceed with the reduction in dimension $d - 1$.

3   Algorithm

Throughout the section, let $(K_i)_{i=1}^n$ be a filtration of a simplicial complex consisting of n simplices, represented by its boundary matrix D. Our goal is to compute the persistence pairs of $(K_i)_i$ on a cluster of p processor units, called *nodes*, which are indexed by the integers $\{1, \ldots, p\}$.

Reduction in blocks   Let $0 = r_0 < \cdots < r_i < \cdots < r_p = n$ be an integer partition of the interval $\{0, \ldots, n\}$. Let the ith *range* be the interval of integers k with $r_{i-1} < k \leq r_i$. We define the *block* (i, j) of M as the block submatrix with rows from the ith row range and columns from the jth columns range. The blocks partition the matrix into p^2 submatrices. Any block (i, j) with $i > j$ is completely zero, since D is lower triangular.

To simplify notation, we call M *reduced at block* (i, j) if M is reduced at index (r_{i-1}, r_j). Moreover, we call M *reducible in block* (i, j) if M is reduced at block $(i, j - 1)$ and at block $(i + 1, j)$. This terminology is motivated by the fact that in order to obtain a matrix that is reduced at block (i, j), only entries in block (i, j) have to be eliminated, as described in Algorithm 1 and shown in the following lemma.

Algorithm 1 Block reduction

Require: input M is reducible in block (i, j)
Ensure: result M is reduced at block (i, j)
  1: **procedure** REDUCEBLOCK(i, j)
  2:    **for** each l in range j in increasing order **do**
  3:       **while** $\exists k$ with pivot(k) = pivot(l) in range i **do**
  4:          add column k to column l
  5:       **if** pivot(l) is in range i **then**
  6:          add column l to collection of reduced columns

Lemma 1. *Algorithm 1 is correct: if M is reducible in block* (i, j)*, then applying Algorithm 1 yields a matrix which is reduced at block* (i, j)*.*

Proof. By induction on l, M is reduced at index (r_{i-1}, l) after each iteration of the main **for** loop (Line 2). This follows directly from the exit condition of the **while** loop in Line 3, together with the induction hypothesis and the precondition that M is reduced at index (r_i, r_j) and hence also at index (r_i, l). □

Lemma 2. *Algorithm 1 only requires access to the unreduced columns of M in range j and the reduced columns with pivot in range i.*

4

Proof. Let l be in range j and let $k < l$ be such that $\text{pivot}(k) = \text{pivot}(l)$ is in range i, as in the **while** loop in Line 3. Then clearly column l is unreduced. Moreover, as shown in the proof of Theorem 1, M is reduced at index (r_{i-1}, l). Since $\text{pivot}(k)$ is in range i, we have $r_{i-1} < \text{pivot}(k)$, and by assumption $k < l$. Hence M is also reduced at index $(\text{pivot}(k), k)$, i.e., column k is reduced. □

Parallel reduction   We now describe a parallel algorithm to reduce a boundary matrix D by applying block reduction on all blocks (i, j) with $i \leq j$ in a certain order.

The algorithm reduces the blocks starting with the diagonal blocks (i, i) with $1 \leq i \leq p$. Indeed, note that the boundary matrix D is (i, i)-reducible for any diagonal block (i, i). All block reductions for diagonal blocks are independent and can be performed in parallel. Now consider a block of the form (i, j) with $i < j$. Note that this block can be reduced as soon as blocks the $(i, j - 1)$ and $(i - 1, j)$ have been reduced. This relation defines a partial order on the blocks (i, j) with $i \leq j$. If the order of execution of the block reductions is consistent with that partial order, the preconditions of block reduction are satisfied in every block. Note that two blocks (i, j) and (i', j') can be reduced independently iff either $(i < i'$ and $j < j')$ or $(i > i'$ and $j > j')$. After having reduced the block $(1, p)$, the postcondition of Algorithm 1 yields that the resulting matrix is a reduction of the input boundary matrix D.

Note that a special case of this block-reduction scheme is the *spectral sequence algorithm* presented in [6, S VII.4]. This algorithm sweeps the blocks diagonally, and in each phase $r \in \{1, \ldots, p\}$ of the sweep it reduces all blocks (i, j) with $j - i = r - 1$ in order of increasing index i. The algorithm as described is sequential, however, as discussed above, within a given phase r the blocks can be reduced independently.

Distributed reduction   We now describe how the data and the workload are distributed and transferred between the nodes.

Each node i is assigned a row of blocks for reduction. The blocks are necessarily processed from left to right. Recall that reducing a block (i, j) requires access to the unreduced columns in range j, and to the reduced columns with pivot in range i. During the execution, each node i maintains a collection of all reduced columns with pivot in the ith range, indexed by pivot. The unreduced columns in a given range j, on the other hand, are passed on from node to node. No data is duplicated among the nodes; each column of the matrix is stored in exactly one node throughout the execution of the algorithm. The union of the locally stored unreduced and reduced columns yields a distributed representation of the partially reduced boundary matrix.

Initially, each node i loads the columns of the input boundary matrix in range i. The following procedure is now repeated, with j ranging from i to m. Node i performs reduction in block (i, j) and retains the reduced columns with pivot in range i in its collection. After that, it sends a package to node $i - 1$ containing the remaining unreduced columns in range j (if $i > 1$), and receives a package from node $i + 1$ containing the unreduced columns in range $j + 1$ (if $j < p$).

Observe that in each iteration, node i has all the information required to perform reduction in block (i, j), namely, the unreduced columns in range j and the reduced columns with pivot in range i. Moreover, the preconditions for block reduction are satisfied, since block $(i, j - 1)$ is reduced on the same node i before block (i, j), and block $(i + 1, j)$ is reduced on node $i + 1$ before node i receives the unreduced columns in range j from node $i + 1$. We conclude:

Lemma 3. *If Algorithm 2 is executed on a cluster with p nodes, it computes a reduction of the input matrix.*

Note that the structure of communication between the nodes is very simple: each node i only receives data from node $i + 1$ and only sends data to node $i - 1$. Moreover, less than p messages are sent between each pair of consecutive nodes. This is highly beneficial for distributed computing, as the communication overhead and the network latency become negligible.

5

Algorithm 2 Distributed matrix reduction

Require: access to columns of input boundary matrix D in range j
Ensure: resulting output matrix R is a reduction of D
 1: **procedure** REDUCEONNODE(i)
 2:         input package with columns of D in range i
 3:      **for** $j = i, \ldots, p$ **do**
 4:            REDUCEBLOCK(i, j)
 5:         **if** $i > 1$ **then**
 6:               send package with unreduced columns in range j to node $i - 1$
 7:         **if** $j < p$ **then**
 8:               receive package with unreduced columns in range $j + 1$ from node $i + 1$
 9:      **return** reduced columns with pivot in range i

Clearing in parallel   The clearing optimization from Section 2 can be implemented in the distributed reduction algorithm with minor changes. Recall that the the clearing optimization iterates over the dimensions d in decreasing order and processes only the columns of a given dimension d at a time.

The ranges are defined by a single global partition $r_0 < \ldots < r_p$ that does not change per dimension. Note that this might cause initial column packages of different sizes in a given dimension, even if the ranges are all of same size. However, it has the following advantage: when node i has performed its last block reduction for dimension d, it knows all pivots that fall in the ith range. All these pivots corresponds to $d - 1$-simplices that create homology and hence correspond to zero columns in any reduction. In the next iteration, node i is initialized to process the columns of dimension $d - 1$ in the ith range. Before it starts the block reduction, it can simply clear all columns with indices that were pivots in dimension d. In particular, no communication with other nodes is required.

Design rationale   We justify some design choices in our algorithm and discuss alternatives. First, we implemented the sending of packages in Algorithm 2 in a *blocking* fashion, i.e., a node does not start receiving the next package until has sent and discarded its current package. Clearly, this strategy can result in delayed processing of packages because a sending node has to wait for its predecessor to be ready to receive a package. On the other hand, the strategy guarantees that every node holds at most one package at a time; this prevents a slower node from accumulating more and more packages, possibly causing high memory consumption.

A possible strategy to reduce the overall amount of communication would be to have node i send a unreduced column with pivot in the kth range to node k directly, instead of the predecessor node $i-1$. However, this approach would complicate the communication structure and data management significantly. Any node would have to be able to receive unreduced columns any time, and it would not be possible to bound the number of unprocessed columns a node has to maintain in memory. It would also increase the number of messages send through the network.

A somewhat dual approach to our communication scheme would be to send the reduced columns from node i to $i + 1$ instead of sending the unreduced columns from node i to $i - 1$. In this variant, node i would perform reduction in block (j, i) for $j = i, i - 1, \ldots, 1$. However, in this approach, the package size would increase towards the end of the reduction, as the number of reduced columns increases, whereas in our implementation the package size decreases together with the number of reduced columns. Since typically most columns are reduced early on, we expect much more data to be sent between the nodes using this variant.

cores/nodes	PHAT		DIPHA				
	1	16	2	4	8	16	32
GRF2-256	10.2GB	10.5GB	11.1GB	5.6GB	2.8GB	1.4GB	0.74GB
GRF1-256	10.8GB	11.3GB	11.8GB	6.1GB	3.1GB	1.5GB	0.8GB
GRF2-512						11.1GB	5.7GB
GRF1-512							9.1GB
vertebra16							9.0GB

Table 1: Peak memory consumption for sequential and parallel shared memory (PHAT, left) algorithms and our distributed algorithm (DIPHA, right)

cores/nodes	PHAT		DIPHA				
	1	16	2	4	8	16	32
GRF2-256	14.6s	5.2s	10.1s	5.5s	3.4s	2.2s	1.6s
GRF1-256	28.8s	12.8s	27.2	20.3	15.4	12.1s	9.9s
GRF2-512						17.9s	11.2s
GRF1-512							95.3s
vertebra16							34.9s

Table 2: Running times for sequential and parallel shared memory (PHAT, left) algorithms and our distributed algorithm (DIPHA, right)

4 Experiments

Since our algorithm is, to the best of our knowledge, the first attempt at computing persistence in a distributed memory context, we concentrate our experimental evaluation on two aspects. First, how does our approach scale with an increasing number of nodes, in running time and memory consumption? Second, how does the our algorithm compare with state-of-the-art sequential and parallel shared memory implementations on instances which are still computable in this context?

We implemented Algorithm 2 in C++ using the OpenMPI implementation of the Message Parsing Interface standard[1]. We ran the distributed algorithm on a cluster with up to 32 nodes, each with two Intel Xeon CPU E5-2670 2.60GHz processors (8 cores each) and 64GB RAM, connected by a 40Gbit Infiniband interconnect.

For comparison, our tests also include results for the PHAT library[2], which contains efficient sequential and parallel shared memory algorithms for computing persistence. Among the sequential versions, the `--twist` algorithm option, which is the standard reduction with the clearing optimization described in Section 2, together with the `--bit_tree_pivot_column` data structure option, showed the overall best performance (see the PHAT documentation for more information). For parallel shared memory, the `--block_spectral_sequence` algorithm with the `--bit_tree_pivot_column` data structure showed the overall best performance on the tested examples. We therefore used these two variants for comparison. The sequential and parallel shared memory algorithms were run on a single machine of the cluster. In order to obtain a clear comparison between the shared memory and distributed memory algorithms, in our test of the distributed algorithm only one processor core per node was used.

For our tests, we focus on filtrations induced by 3D image data. In particular, we used isotropic Gaussian random fields whose power spectral density is given by a power law $\|x\|^{-p}$. This process is commonly used in physical cosmology as a model for cosmic microwave background [14]. We consider two images sizes: filtrations of images of size 256^3 have a length of $n = 511^3 \approx 133$ millions and a binary file size of around

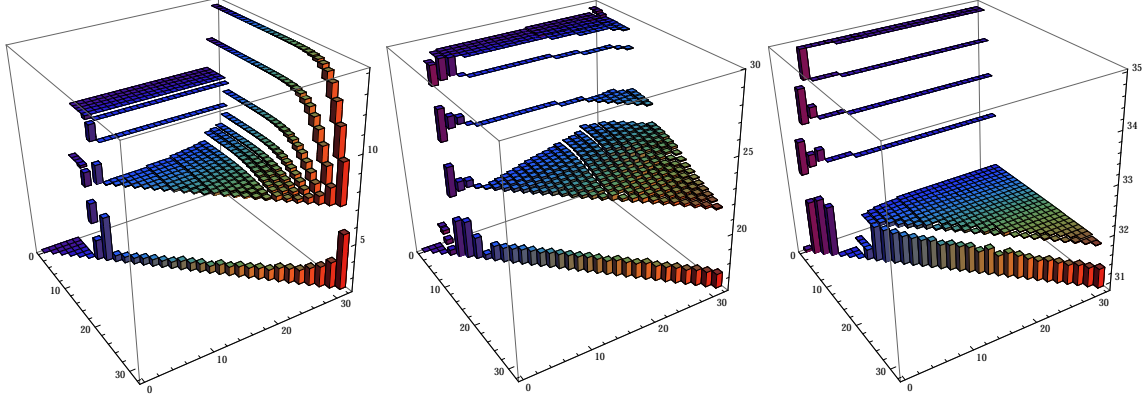---

[1] www.open-mpi.org
[2] http://phat.googlecode.com

Figure 1: Running times for each block reduction in dimensions $\delta = 3, 2, 1$ for the vertebra16 data set.

5GB, while images of size 512^3 yield a filtration of length $n = 1023^3 \approx 1.07$ billions and a file size of around 40GB. In addition, we included the 512^3 medical image `vertebra16` from the VoLVis repository[3] in our test set, a rotational angiography scan of a head with an aneurysm.

Scalability Tables 1 and 2 show the running time and peak memory consumption of our algorithm for images of size 256^3 and 512^3. The table is incomplete; the algorithm was not able to compute a result for the remaining cases because of address space limitations of the OpenMPI I/O API that we plan to circumvent in a forthcoming version. We observe that the memory usage per node is almost exactly halved when doubling the number of nodes. For the running time, the speed-up factor is not quite as high, but still the algorithm terminates faster when using more nodes. In summary, this provides strong evidence that our algorithm scales well with the number of nodes, both regarding time and space complexity.

Comparison Tables 1 and 2 also lists the results for the best sequential and parallel shared memory algorithms of the PHAT library. Both algorithms run out of memory when trying to compute persistence for larger examples on our testing machine, showing that our distributed approach indeed extends the set of feasible instances. Moreover, we observe that the running time on 16 nodes with distributed memory is actually lower than that of the parallel shared memory algorithm on a single machine with 16 processor cores. One reason might be that the distributed system has a much larger total amount of processor cache available than the shared memory system. Since matrix reduction is more memory intensive than processor intensive, this effect may actually outweigh the overhead of communication over the network. This suggests that the distributed approach may be preferable even if the solution is in principle computable in a non-distributed environment.

Communication analysis We give more details on the amount of data transmitted between the nodes by our algorithm. Table 3 shows the total amount of data exchanged; Table 4 shows the largest total amount of data transmitted between any pair of nodes; Table 5 shows the largest package size. For the more challenging examples, the amount is in the range of GBs. Considering the bandwidth of modern interconnects and the fact that communication is bundled in a small number of packages, the running time of the local block reductions dominates the time spent for communication. This is illustrated in Fig. 1, which shows a plot of the running times for each block reduction for the vertebra16 data set on 32 nodes.

---

[3]Available at http://volvis.org

8

nodes	2	4	8	16	32
GRF2-256	5.6MB	15.1MB	32.5MB	67.7MB	136MB
GRF1-256	69.2MB	218MB	497MB	1.0GB	2.0GB
GRF2-512				342MB	694MB
GRF1-512					34.0GB
vertebra16					19.1GB

Table 3: Total size of all packages sent over the network

nodes	2	4	8	16	32
GRF2-256	5.6MB	5.6MB	5.6MB	6.5MB	8.7MB
GRF1-256	69.2MB	90.3MB	109MB	162MB	238MB
GRF2-512				29.2MB	29.7MB
GRF1-512					5.0GB
vertebra16					4.2GB

Table 4: Maximum total size of all packages transmitted between any pair of nodes

nodes	2	4	8	16	32
GRF2-256	3.1MB	2.9MB	2.7MB	3.6MB	2.5MB
GRF1-256	61.4MB	52.0MB	69.0MB	50.9MB	38.4MB
GRF2-512				11.5MB	9.0MB
GRF1-512					1.9GB
vertebra16					1.5GB

Table 5: Maximum package size sent over the network

5   Conclusion

We presented the first implementation of an algorithm for computing persistent homology in a distributed memory environment. While our algorithm resembles the spectral sequence algorithm for persistence computation to a large extent, several lower-level design choices were necessary for an efficient realization. Our approach permits the computation of instances that were infeasible for previous methods, and the parallelism also speeds up the computation for previously feasible instances.

We plan to extend our experimental evaluation in future work. One problem in benchmarking our new approach is that persistence computation is only the second step in the pipeline: first, one has to generate a filtration that serves as the input for the algorithm. This itself usually requires a massive computation, which at some point becomes infeasible on single machines as well. We are currently working on methods for generating filtrations of large 3D images and Rips filtrations in a distributed memory environment.

References

[1] Ulrich Bauer, Michael Kerber, and Jan Reininghaus. Clear and compress: Computing persistent homology in chunks. In *TopoInVis 2013*, 2013.

[2] F. Chazal, D. Cohen-Steiner, M. Glisse, L. Guibas, and S. Oudot. Proximity of persistence modules and their diagrams. In *Proc. 25th ACM Symp. on Comp. Geom.*, pages 237–246, 2009.

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

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

[5] D. Cohen-Steiner, H. Edelsbrunner, and J. Harer. Stability of persistence diagrams. *Discrete and Computational Geometry*, 37:103–120, 2007.

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

[7] Herbert Edelsbrunner and John Harer. Persistent homology — a survey. In Jacob E. Goodman, János Pach, and Richard Pollack, editors, *Surveys on Discrete and Computational Geometry: Twenty Years Later*, Contemporary Mathematics, pages 257–282. 2008.

[8] Herbert Edelsbrunner, David Letscher, and Afra Zomorodian. Topological persistence and simplification. *Discrete and Computational Geometry*, 28:511–533, 2002.

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

[10] David Lipsky, Primoz Skraba, and Mikael Vejdemo-Johansson. A spectral sequence for parallelized persistence. arXiv:1112.1245, 2011.

[11] Clement Maria, Jean-Daniel Boissonnat, and Tamal Dey. The compressed annotation matrix: An efficient data structure for computing persistent cohomology. In *ESA 2013*, 2013.

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

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

[14] John Peacock. *Cosmological Physics*. Cambridge University Press, 1999.

[15] Afra Zomorodian and Gunnar Carlsson. Computing persistent homology. *Discrete and Computational Geometry*, 33:249–274, 2005.