Parallel Computation of Persistent Homology using the Blowup Complex

Ryan Lewis
Stanford University
Stanford, CA
me@ryanlewis.net

Dmitriy Morozov
Lawrence Berkeley National Laboratory
Berkeley, CA
dmitriy@mrzv.org

ABSTRACT

We describe a parallel algorithm that computes persistent homology, an algebraic descriptor of a filtered topological space. Our algorithm is distinguished by operating on a spatial decomposition of the domain, as opposed to a decomposition with respect to the filtration. We rely on a classical construction, called the Mayer–Vietoris blowup complex, to glue global topological information about a space from its disjoint subsets. We introduce an efficient algorithm to perform this gluing operation, which may be of independent interest, and describe how to process the domain hierarchically. We report on a set of experiments that help assess the strengths and identify the limitations of our method.

Categories and Subject Descriptors

F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Parallelism and concurrency*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

Keywords

persistent homology; Mayer–Vietoris blowup complex

1. INTRODUCTION

Persistent homology was introduced fifteen years ago by Edelsbrunner, Letscher, and Zomorodian [1] and later placed on a firm algebraic footing by Carlsson and Zomorodian [2]. From its roots as a method to measure shape across scales, it has evolved into a rich mathematical theory with applications to clustering [3] and dimensionality reduction [4], to materials science [5] and cosmology [6], to integral geometry [7] and image analysis [8], to name just a few. We refer the reader to the several excellent surveys [9, 10, 11, 12, 13] for a more detailed look at what makes persistence so exciting both to mathematicians and practitioners.

Without losing too much generality, we can think of persistence as operating on scalar functions, $f : X \to \mathbb{R}$. It tracks evolution of homology classes (an algebraic formalization of "holes") in the sublevel sets, $f^{-1}(-\infty, a]$, of these functions for varying values of threshold a. Specifically, it pairs those values of a where new homology classes appear and where they die. Such birth–death information is valuable for inference (e.g., when the domain X is high-dimensional and cannot be seen directly) and as a statistical descriptor of the function that can be used, for example, to compare different measurements of a physical phenomenon.

One ingredient in the success of persistent homology has been the development of efficient algorithms for its computation. The original paper [1] introduced a cubic-time algorithm that computes persistence by reducing a boundary matrix via a constrained Gaussian elimination. A different, output-sensitive analysis in the same paper hints at why the algorithm performs so much more efficiently in practice than the worst-case analysis would suggest. Since then an algorithm for computing persistence in matrix multiplication time has appeared [14], with a matching lower bound [15]. Other notable theoretical results include an output-sensitive algorithm that computes persistence in a top-down manner [16], the connection between different variations of the original algorithm and algebraic dualities [17], including an algorithm to compute persistent cohomology [4] and a data structure that improves its performance [18].

Another important direction in understanding computational aspects of persistence is the work on various optimizations of the algorithms. Already the original paper [1] observed that one can get rid of the so-called negative simplices during the computation. Another notable result is the *clearing* optimization [19], which zeroes out entire columns of the matrix without processing them. It's also possible to combine the two optimizations together [20], although doing so requires a different algorithm.

Despite significant successes, there is still a large gap between the sizes of data sets that persistent homology algorithms can process and what's needed in practice. One way to close this gap is to develop parallel algorithms and take advantage of the modern massively parallel computers. But any such attempt faces a fundamental difficulty: topology tracks global information, while parallel computation thrives on locality. In this paper we explore one approach to computing persistent homology in parallel. It's distinguished by dividing the work with respect to a spatial decomposition of the domain of the function, a feature important, for example, for integrating with existing decompositions used in simulation codes.

Related work. Ours is not the first attempt to design a parallel algorithm for persistent homology. Broadly, such algorithms can be split into two categories based on how they partition the computation between processors. Algorithms in the first category divide the data by function value, grouping ranges of function values together. Edelsbrunner and Harer [28] introduced the "spectral sequence algorithm," which divides the input matrix into blocks and processes them along diagonals. The algorithm is naturally parallel: blocks within a diagonal can be processed independently. Bauer, Kerber, and Reininghaus [20, 21] have examined the practical aspects of this algorithm. Notably they found clever ways to combine seemingly incompatible optimizations and implemented the algorithm, both in shared and distributed memory.

Another way to parallelize persistence, when dividing the data by function value, is using zigzag persistent homology [22]. The approach, suggested at the end of that paper, has not yet been tried in practice, as far as we know.

The second way to break up the computation between processors is to partition the domain of the function, assigning different regions to different processors. Although the approach sounds simple — it is probably the most common way to divide data for independent processing — for persistence such a partition presents a distinct challenge. Homology captures global properties of a topological space; persistent homology does so for a large collection of topological spaces at once. Gluing information from different subsets of the domain requires resolving certain algebraic issues. Translating any such resolution into a practical algorithm is a challenge of its own.

Although formulated in a different language (of spectral sequences), the work of Lipsky et al. [23] is close to ours, at least in spirit. They describe an algebraic construction that prescribes how to combine information from different subsets of the domain into a coherent whole. There are two problems with that paper. First, there is a serious technical error[1], which the authors are aware of and have a possible correction (according to a personal communication). The second problem is that, even if we restrict attention to ordinary homology (where the mentioned algebraic problem does not exist), the algorithm is not sufficiently detailed. It does not actually consider which information is available locally to any given processor, or what exactly is required to compute a particular algebraic object in parallel. There are also subtler differences between our approaches (for example, Lipsky et al. propose to collect information in the order of increasing dimension of the nerve, whereas we pursue a hierarchical decomposition of the domain), but they are less important. In this paper, we replace the algebraic construction with a geometric construction of Mayer–Vietoris blowup complex, which contains the same information, but is much more transparent computationally.

Mayer–Vietoris blowup complex was introduced to the computational topology community by Zomorodian and Carlsson [24], who used it to compute, what they called, localized

---

[1] Briefly, the paper assumes that given a map between two persistent homology modules, the target module decomposes as the direct sum of the image module and the cokernel module, which is not the case. Recovering the former decomposition from the latter requires a certain amount of "repair." In effect, Sections 3 and 4 of our paper are dedicated to performing such a repair efficiently.

homology. Although there is some overlap between the keywords of our papers, the details diverge significantly. They use persistent homology as a tool to localize homology classes to the individual sets of the cover. (And, accordingly, take a filtration with respect to the second factor of the blowup.) We are concerned with using the blowup complex as a tool to compute persistent homology of the base space. (And, therefore, filter by the first factor.)

Lewis and Zomorodian [25] use the Mayer–Vietoris blowup complex to compute ordinary homology in parallel, in shared memory. The basic ideas overlap between our papers, but persistent homology imposes more constraints on the operations one may perform during the matrix reduction. In the context of [25], Sections 3 and 4 of our paper can be seen as a way to adapt their results to persistent homology. Our work also finds more parallelism (afforded by the row operations) than exploited by Lewis and Zomorodian. As such it can be seen as an improved way to compute ordinary homology (over a field), which comes out as a byproduct of persistent homology. Also notably, Lewis and Zomorodian consider the complexity of cover construction, an important question that we ignore entirely in this paper.

To conclude our review of related work, we also note the work on distributed computation and representation of merge trees [26]. Merge trees can be used to recover 0-dimensional persistent homology, which is a very special case computationally (with different complexity characteristics than the general case). In the present paper, we are interested in the general problem in all dimensions.

Contributions. Our contributions are three-fold:

- we present the first parallel algorithm that computes persistent homology from a spatial decomposition of the domain;

- we present an efficient procedure, **Cascade** in Section 4, that combines persistence pairs from different subspaces of the domain into persistence pairs for the entire domain; that algorithm operates on matrices with a particular structure and may be of independent interest in other contexts;

- we present experiments and describe practical limitations of our results.

2. BACKGROUND

We briefly review the necessary background, but encourage the reader to consult a textbook on algebraic topology (e.g., by Hatcher [29]) or on computational topology (e.g., by Edelsbrunner and Harer [28]) for a thorough introduction to the subject.

Homology. Given a universal vertex set V, an (abstract) *simplex* is a subset of V, $\sigma \subseteq V$. The dimension of a simplex is one less than its cardinality, $\dim \sigma = \operatorname{card} \sigma - 1$. A subset of a simplex is called its *face*. A *simplicial complex* K is a collection of simplices closed under the face relation, i.e., if $\sigma \in K$ and $\tau \subseteq \sigma$, then $\tau \in K$.

In applications, simplicial complexes often arise from geometric constructions. For example, given a point set P with a metric $d : P \times P \to \mathbb{R}$, one can build a Vietoris–Rips complex for a parameter r by recording every subset $\sigma \subseteq P$,

with $d(x, y) \leq r \; \forall x, y \in \sigma$. It's easy to verify that this construction is closed under the face relation.

A *k-chain* is a formal sum of k-dimensional simplices, $c = \sum a_i \sigma_i$. Throughout the paper, for ease of exposition, we assume the coefficients a_i are elements of the field $\mathbb{Z}/2\mathbb{Z}$, so a chain can be thought of as a set of simplices. The k-chains form an abelian group under addition, which we denote by C_k. The *boundary map* ∂_k takes a k-simplex σ into the sum of its $(k-1)$-dimensional faces. It extends linearly to the chain groups C_k, giving an operator $\partial_k : C_k \to C_{k-1}$.

The kernel of this operator, $\mathsf{Z}_k = \ker \partial_k$, is called the *cycle group*; by definition, it consists of all the chains with an empty boundary. The image of the operator, $\mathsf{B}_k = \operatorname{im} \partial_{k+1}$, is called the *boundary group*. A *homology group* is the quotient, $\mathsf{H}_k = \mathsf{Z}_k / \mathsf{B}_k$; it's the group of non-bounding cycles. It is convenient to suppress the dimensions by taking the direct sum of chain groups across all dimensions, $C(K) = \oplus C_k(K)$, with $\partial : C(K) \to C(K)$, and $\mathsf{H}(K) = \ker \partial / \operatorname{im} \partial$.

Persistent homology. A nested sequence of simplicial complexes, $K_0 \subseteq K_1 \subseteq \ldots \subseteq K_n = K$, is called a *filtration* of K. Passing to homology, we get a sequence of homology groups (one for each complex in the filtration), connected by linear maps induced by the inclusions.

$$\mathsf{H}(K_0) \to \ldots \to \mathsf{H}(K_i) \to \ldots \to \mathsf{H}(K_n) \qquad (1)$$

We denote the maps induced by inclusions of the simplicial complexes by $\mathbf{f}_i^j : \mathsf{H}(K_i) \to \mathsf{H}(K_j)$. Persistent homology tracks how elements appear and disappear in this sequence. We say that an element $\alpha \in \mathsf{H}(K_i)$ is born in $\mathsf{H}(K_i)$ if it's not in the image of the map $\mathbf{f}_{i-1}^i$. We say that α dies at $\mathsf{H}(K_j)$ if $\mathbf{f}_i^j(\alpha) \in \operatorname{im} \mathbf{f}_{i-1}^j$, but $\mathbf{f}_i^{j-1}(\alpha) \notin \operatorname{im} \mathbf{f}_{i-1}^{j-1}$. If there is an element α born in $\mathsf{H}(K_i)$ that dies in $\mathsf{H}(K_j)$, we record this as a pair (i, j). The goal of algorithms for computing persistent homology is to compute all such pairs (i, j) for a given filtration.

The following theorem characterizes when two sequences of homology groups produce the same persistence pairing.

PERSISTENCE EQUIVALENCE THEOREM [28]. *Given two filtrations $L_0 \subseteq \ldots \subseteq L_n$ and $K_0 \subseteq \ldots \subseteq K_n$, the induced sequences of homology groups produce the same persistence pairs, if there are isomorphisms $\mathsf{H}(L_i) \to \mathsf{H}(K_i)$ that commute with inclusions. In other words, if the vertical maps in the following diagrams are isomorphisms, and the diagram commutes.*

$$
\begin{array}{ccccccccc}
\mathsf{H}(K_1) & \longrightarrow & \cdots & \longrightarrow & \mathsf{H}(K_i) & \longrightarrow & \cdots & \longrightarrow & \mathsf{H}(K_n) \\
\uparrow & & & & \uparrow & & & & \uparrow \\
\mathsf{H}(L_1) & \longrightarrow & \cdots & \longrightarrow & \mathsf{H}(L_i) & \longrightarrow & \cdots & \longrightarrow & \mathsf{H}(L_n)
\end{array}
$$

Algorithms. Assume that we are given a filtration $K_0 \subseteq \ldots \subseteq K_i \subseteq \ldots \subseteq K_n = K$, where $K_{i+1} = K_i \cup \sigma_{i+1}$. We represent the boundary map, $\partial : C(K) \to C(K)$, as a matrix D, where $D[i, j] = 1$ if $(k-1)$-simplex σ_i is a face of k-simplex σ_j; $D[i, j] = 0$ otherwise. Notice that given such a representation, every upper-left square sub-matrix of D, sub-matrix consisting of rows and columns $[0 \ldots i]$, represents the boundary map of the subcomplex K_i, $\partial_i : C(K_i) \to C(K_i)$.

Given such a matrix, we can compute the persistence pairing in the sequence of homology groups (1) using algorithm

$\mathbf{ELZ}(D)$, Algorithm 1, introduced by Edelsbrunner et al. [1]. Let "low $R[\cdot, i]$" denote the lowest non-zero entry in the column $R[\cdot, i]$ (the map is undefined if the column is zero). We say that the matrix R is *reduced* if the lowest non-zero entry in every column falls in a unique row; in other words, if the map low is injective. The original algorithm [1] computes only the matrix R by greedily subtracting columns from left to right until no two columns have the lowest non-zero entry in the same row. When this happens, the lowest non-zero entries of the reduced matrix R record the sought after persistence pairing, i.e., we have a pair (i, j) in the sequence $\mathsf{H}(K_0) \to \ldots \to \mathsf{H}(K_n)$ if and only if low $R[\cdot, j] = i$.

An additional matrix U was introduced in a later work [27], which reinterpreted persistence computation as finding a decomposition $D = RU$, where the matrix R is reduced and the matrix U is invertible upper triangular. The crucial insight of that paper was that for any such decomposition, we get the same lowest non-zero entries in the columns of R (i.e., the same map low R). In particular, this means that one can subtract columns of R in any order, as long as such operations always happen from left to right. If in the end the matrix R is reduced, then we recover the correct pairing. We denote by $r_D(i, j)$ the following inclusion-exclusion of ranks of lower-left sub-matrices of D,

$$r_D(i, j) = \operatorname{rk} D_i^j - \operatorname{rk} D_{i+1}^j + \operatorname{rk} D_{i+1}^{j-1} - \operatorname{rk} D_i^{j-1},$$

where $\operatorname{rk} D_i^j$ denotes the rank of the lower-left sub-matrix of D, namely the sub-matrix that retains rows $[i..n]$ in columns $[0..j]$.

LEMMA 1 (PAIRING UNIQUENESS LEMMA [27]). *Letting $D = RU$, we have low $R[\cdot, j] = i$ iff $r_D(i, j) = 1$. In particular, the pairing function does not depend on the matrix R in the RU-decomposition.*

We do not repeat the proof of the lemma here, but briefly recall that it depends on the fact that if R is reduced then low $R[\cdot, j] = i$ iff $r_R(i, j) = 1$. Then it's easy to see that left-to-right column operations will not change the ranks of lower-left sub-matrices and, therefore, $r_R(i, j) = r_D(i, j)$.

For the present work, we need to extend this lemma to allow row operations on the matrix R.

LEMMA 2. *Letting $D = SRU$, where S and U are invertible upper-triangular and R is reduced, we have low $R[\cdot, j] = i$ iff $r_D(i, j) = 1$.*

The proof is the same as before. Neither the left-to-right column operations, expressed by the matrix U, nor the bottom-up row operations, expressed by matrix S, change the ranks of lower-left sub-matrices. Therefore, $r_R(i, j) = r_D(i, j)$.

As an immediate consequence of the amended lemma, we can choose the matrix S to be such that we get a decomposition $D = SPU$, where P is not only reduced, but it has at most one non-zero entry per column. To achieve this, we first perform the reduction prescribed by the $\mathbf{ELZ}(D)$ algorithm to get the decomposition $D = RU$. Then we go through the rows of the matrix R from the bottom up and, whenever we encounter a (unique, by induction) non-zero in a row, we subtract it from the rest of the entries in its column. This operation is formalized in the algorithm $\mathbf{Sparsify}(R)$, Algorithm 2. For clarity, we prove its correctness.

PROOF (CORRECTNESS OF ALGORITHM 2). We want to show by induction that after the outer loop processes row

Algorithm 1 Persistence reduction algorithm.

$\mathbf{ELZ}(D)$:
    $R = D$, $U = I$
    for all columns $R[\cdot, i]$ **do**
        while $R[\cdot, i] \neq 0$ and $\exists j < i$ with low $R[\cdot, i] = $ low $R[\cdot, j]$ **do**
            $R[\cdot, i] = R[\cdot, i] - R[\cdot, j]$
            $U[j, i] = 1$

Algorithm 2 Column sparsification algorithm.

$\mathbf{Sparsify}(R)$:
    $P = R$, $S = I$
    for all rows $P[i, \cdot]$ of P from bottom up **do**
        if $P[i, j]$ is not zero **then**
            for all non-zero entries $P[i', j]$ in $P[\cdot, j]$, with $i' \neq i$ **do**
                $P[i', \cdot] = P[i', \cdot] - P[i, \cdot]$ # zero out the entry $P[i', j]$
                $S[\cdot, i] = S[\cdot, i] + S[\cdot, i']$

---

i, the columns with the lowest non-zero entries in rows i and below have a unique non-zero entry, and the matrix P remains reduced. The base case is trivially true: either the lowest row of the matrix P is zero, or it has a unique non-zero entry since the input matrix R is reduced. Suppose the statement is true after the outer loop completes processing row $i + 1$. When processing row i, either the row is empty, and we are done, or again it has a unique non-zero entry. Why? Suppose there is more than one non-zero entry in the row. Then either all of them are the lowest entries in their columns, which would violate the assumption that the matrix P is reduced, or some of them have lower entries in their columns, which would violate the inductive hypothesis. Since the non-zero entry $P[i, j]$ is unique, after the inner loop of the algorithm, the column above it is zero, matrix P remains reduced, and the statement is true for all the rows up to i. By induction, at termination, the algorithm produces matrix P that is reduced and each of whose columns has at most one non-zero entry. $\square$

Blowup complex. We recall a classical construction used for gluing topological information from multiple disjoint subsets of a space. Its features relevant to our work are explained perspicuously by Zomorodian and Carlsson [24]; the original description of the structure was given by Segal [30]. Given a cover $\mathcal{C} = \{K^i\}_{i \in I}$ of a complex K by simplicial subcomplexes $K^i \subseteq K$, we denote the intersection of subcomplexes indexed by $J \subseteq I$ by $K^J = \cap_{j \in J} K^j$.

DEFINITION 3. *The* Mayer–Vietoris blowup complex *of simplicial complex K and cover $\mathcal{C}$, $K^{\mathcal{C}} \subseteq K \times I$, is defined by*

$$K^{\mathcal{C}} = \bigcup_{J \subseteq I} \bigcup_{\sigma \in K^J} \sigma \times J.$$

Figure 1 illustrates a simplicial complex, covered by three subcomplexes, and the resulting blowup complex.

Chain complex of the blowup. A basis for the chain complex may be prescribed via tensor products $C_*(K^{\mathcal{C}}) = \langle \sigma \times J \mid \sigma \times J \in K^{\mathcal{C}} \rangle$ [24, Section 4.3]. The tensor product structure endows the boundary operator of the blowup complex with a useful structure. The boundary of a cell $\sigma \times J \in K^{\mathcal{C}}$ is given by [24, Lemma 4],

$$\partial(\sigma \otimes J) = \partial\sigma \otimes J + (-1)^{\dim \sigma} \sigma \otimes \partial J. \qquad (2)$$
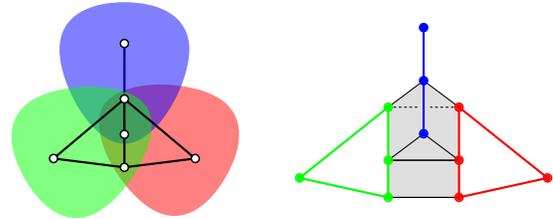
Figure 1: Left: Simplicial complex K. The subcomplexes of the cover are highlighted with three shaded regions. Right: The resulting blowup complex $K^{\mathcal{C}}$, with the disjoint subcomplexes of the cover shown in red, green, and blue.

With a boundary operator defined we may now consider the homology of the blowup complex $\mathsf{H}(K^{\mathcal{C}})$. Let $\pi : K^{\mathcal{C}} \to K$ denote the projection of the blowup complex onto the first factor. This map is a homotopy equivalence [24]. We do not define this technical term, but note that it implies that the map $\pi^* : \mathsf{H}(K^{\mathcal{C}}) \to \mathsf{H}(K)$, induced on homology, is an isomorphism.

In moving to persistent homology, we need only specify a partial order on the $K^{\mathcal{C}}$. Given a subcomplex $L \subseteq K$, its blowup complex (where the cover is the restriction of the original cover, $L^i = K^i \cap L$),

$$L^{\mathcal{C}} = \bigcup_{J \subseteq I} \bigcup_{\sigma \in (K^J \cap L)} \sigma \times J,$$

is a subcomplex of the blowup complex of $K^{\mathcal{C}}$, $L^{\mathcal{C}} \subseteq K^{\mathcal{C}}$. The projection map, $\pi_L : L^{\mathcal{C}} \to L$, is also a homotopy equivalence, so the map $\pi_L^* : \mathsf{H}(L^{\mathcal{C}}) \to \mathsf{H}(L)$, induced on homology groups, is an isomorphism. We arrive at the main reason for using the Mayer–Vietoris blowup complex.

THEOREM 4. *A filtration $K_1 \subseteq \ldots \subseteq K_i \subseteq \ldots \subseteq K$ of the base complex K induces a filtration $K_1^{\mathcal{C}} \subseteq \ldots \subseteq K_i^{\mathcal{C}} \subseteq \ldots \subseteq K^{\mathcal{C}}$ of the blowup complex. Passing to homology, we get two*

$$K_{>0}^{\mathcal{C}}$$
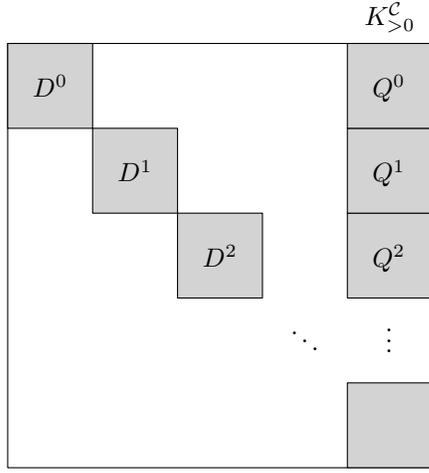
Figure 2: Schematic structure of the boundary matrix of the blowup complex. Only shaded regions may be non-zero.

$$K_{>0}^{\mathcal{C}}$$
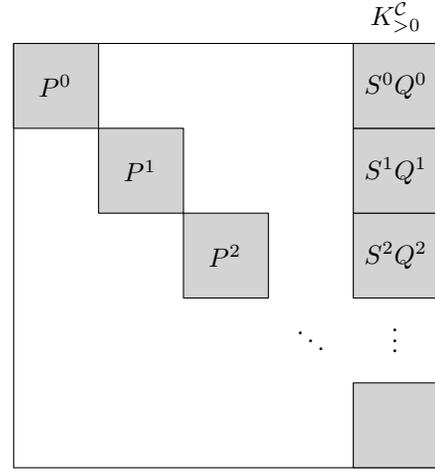
Figure 3: The matrix T', formed out of the boundary matrix of the blowup complex after the initial column and row reductions.

sequences of homology groups connected by isomorphisms,

$$\mathsf{H}(K_1) \longrightarrow \cdots \longrightarrow \mathsf{H}(K_i) \longrightarrow \cdots \longrightarrow \mathsf{H}(K_n)$$

$$\Big\uparrow \pi_1^* \qquad\qquad \Big\uparrow \pi_i^* \qquad\qquad \Big\uparrow \pi_n^*$$

$$\mathsf{H}(K_1^{\mathcal{C}}) \longrightarrow \cdots \longrightarrow \mathsf{H}(K_i^{\mathcal{C}}) \longrightarrow \cdots \longrightarrow \mathsf{H}(K_n^{\mathcal{C}})$$

The persistence pairs in the two sequences are the same.

PROOF. The vertical maps are isomorphisms. The projections π_i commute with the inclusions, so the entire diagram commutes. Persistence Equivalence Theorem implies that the persistence pairs in the top and bottom sequences are the same. □

In other words, we can compute the persistence pairing of the filtration of K from the filtration of $K^{\mathcal{C}}$.

We end this section by noting that in the filtration of $K^{\mathcal{C}}$, the complex is not constructed one cell at a time, i.e., the difference between $K_{i+1}^{\mathcal{C}}$ and $K_i^{\mathcal{C}}$ may consist of multiple cells. Within an algorithm we may break these ties in the partial order arbitrarily, as long as we respect the dimension of the cells. So we order the cells in $K_{i+1}^{\mathcal{C}} - K_i^{\mathcal{C}}$ by the dimension of their second factor.

3. ALGORITHM

It is helpful to understand the special structure of the boundary map (2) in the blowup complex. Over $\mathbb{Z}/2\mathbb{Z}$ coefficients, the boundary of a cell $\sigma \times J \in K^{\mathcal{C}}$ becomes

$$\partial(\sigma \otimes J) = \partial\sigma \otimes J + \sigma \otimes \partial J.$$

Let the matrix $D^{\mathcal{C}}$ represent this boundary map, with columns and rows ordered to respect the given filtration of $K^{\mathcal{C}}$. In other words, if we were to reduce $D^{\mathcal{C}}$ using the $\mathbf{ELZ}(D^{\mathcal{C}})$ algorithm, we would get the correct persistence pairing.

Suppose we reorder the columns and the rows of $D^{\mathcal{C}}$ as follows. We group together the rows and the columns $\sigma \times \{i\}$ that belong to the individual disjoint sets of the cover (ordering them by filtration within these sets), and we group together columns that correspond to the cells $\sigma \times J$, where

the second factor J has dimension higher than 0, again ordering by filtration within those columns. Figure 2 shows this structure schematically: D^0, D^1, D^2 denote the sub-matrices of the disjoint sets, i.e., the columns of D^i record the boundaries of the cells $\sigma \times \{i\}$, where $\sigma \in K^i$. The right-most block of shaded columns represents the cells with higher-dimensional second factor, i.e., the cells $\sigma \times J$, where $\dim J > 0$. We denote by Q^i their sub-blocks that fall into the rows $\sigma \times \{i\}$ that correspond to the cells of the disjoint union. Notice that outside the shaded regions in Figure 2, the matrix is necessarily zero.

We observe that because we've ordered the cells within the blocks by filtration, we may carry out operations within the blocks — column operations from left to right, row operations from bottom up — without violating the column and row order within the original matrix $D^{\mathcal{C}}$.

Accordingly, we may column-reduce individual matrix blocks independently, decomposing $D^i = R^i U^i$ using $\mathbf{ELZ}(D^i)$ algorithm. We may further row-reduce matrices R^i, getting decomposition $D^i = S^i P^i U^i$ using $\mathbf{Sparsify}(R^i)$ algorithm. (The matrix P^i has an immediate interpretation: it records the persistence pairs in the restriction of the input filtration to the cover set K^i.) To be consistent in the full boundary matrix $D^{\mathcal{C}}$, we must perform the row operations on the full rows, thus replacing blocks Q^i with blocks $S^i Q^i$. We call the resulting matrix T', see Figure 3.

It is helpful to note the structure of the blocks Q_i. Which blowup cells have the base cells of the disjoint union, $\sigma \times \{i\}$, in their boundary? First of all, these are the cells $\tau \times \{i\}$, with $\sigma \in \partial\tau$; the first factor of their boundary map (2) consists of the cells $\sigma \times \{i\}$. Their boundaries define the columns of the sub-matrices D^i. The second type of a cell that has $\sigma \times \{i\}$ in its boundary are the cells $\sigma \times \{i, j\}$, where cover set K^j intersects cover set K^i. The part of the row of cell $\sigma \times \{i\}$ that falls into the block Q^i has non-zero entries in the columns of such cells. These are the only two possibilities allowed by the boundary formula (2). Most rows of matrices Q^i are zero, since only cells that fall into more than one cover set have non-zero entries in the columns of Q^i.

Now, if we reorder the columns and the rows of the matrix T' back into the original filtration order, call the resulting

matrix T, and reduce it using algorithm $\mathbf{ELZ}(T)$, the low map on the resulting matrix R_T will produce the correct persistence pairing.

THEOREM 5. (i, j) *is a pair in* $\mathsf{H}(K_0^{\mathcal{C}}) \to \ldots \to \mathsf{H}(K_n^{\mathcal{C}})$ *if and only if* $\mathrm{low}\, R_T[j] = i$.

PROOF. We pad and reorder the rows and columns of matrices S^i and U^i so that they match the original blowup boundary matrix $D^{\mathcal{C}}$. (The padding is to identity, i.e., the newly added rows and columns have 1s on the diagonal, so the padded matrices remain invertible upper-triangular.) Since the operations in matrices D^i respected the filtration order, the padded matrices S^i and U^i remain invertible upper-triangular. Therefore, so are their products $S = S^0 \cdot S^1 \cdot \ldots$ and $U = U^0 \cdot U^1 \cdot \ldots$. Therefore, the first set of independent reductions results in the decomposition $D^{\mathcal{C}} = STU$, with T appropriately reordered. Now reducing the matrix T using $\mathbf{ELZ}(T)$ algorithm produces decomposition $T = R_T U_T$, where R_T is reduced and U_T is invertible upper-triangular. Therefore, the complete decomposition is $D^{\mathcal{C}} = SR_T(U_T U)$, and Lemma 2 implies the claim. $\square$

Parallel setup. If we have p cover sets, i.e., $p = \mathrm{card}\, I$, then we can split the initial operations $\mathbf{ELZ}(D^i)$ and $\mathbf{Sparsify}(R^i)$ between p processors. The final reduction $\mathbf{ELZ}(T)$ can be performed by either one of them. If all the processors share the same memory, we may be satisfied with this solution, although we can perform a little more work in parallel, as explained in Section 5.

If the memory is distributed, the separate bits of information P^i and $S^i Q^i$, necessary for the final reduction, need to be brought to a single processor. In Section 5 we explain how this operation can be performed using a binary reduction. Meanwhile, we mention a simple, but important optimization: it suffices to send only those rows of P^i that are not zero in $S^i Q^i$; the rows that are zero in $S^i Q^i$ record the pairs that will not change.

4. CASCADE

So far it is unclear why we performed the seemingly unnecessary sparsification step, converting matrices R^i into matrices P^i. The sparsification is advantageous since it reduces the potentially quadratic size of matrices R^i down to the linear size of matrices P^i. In the distributed setting, this means less data to send to the processor responsible for the final reduction. But there is another advantage. The combined matrix T has a special sparsity pattern. This structure allows for a faster reduction even using the standard $\mathbf{ELZ}(T)$ algorithm. In addition, with a little extra work, presented in algorithm $\mathbf{Cascade}(T)$, Algorithm 3, we can preserve this structure throughout the computation and thus gain efficiency both in time and in space.

Suppose there are $n = \sum_{i \in I} \mathrm{card}\, K^i$ cells in the blowup complex that fall into the disjoint union, and there are m cells with higher-dimensional second factor, $m = \mathrm{card}\{\sigma \times J \mid \sigma \times J \in K^{\mathcal{C}}, \dim J > 0\}$. We assume $m < n$. Then reducing the matrix T using $\mathbf{ELZ}(T)$ algorithm takes, in the worst case, $\mathrm{O}\left(n^2 m\right)$ time and $\mathrm{O}\left(n^2\right)$ space. (The reason why the time complexity is tighter than $\mathrm{O}\left(n^3\right)$ is similar to the analysis in the proof of Theorem 7 below.)

The matrix T has special structure, see Figure 4. The n columns of the disjoint union are formed by matrices P^i,
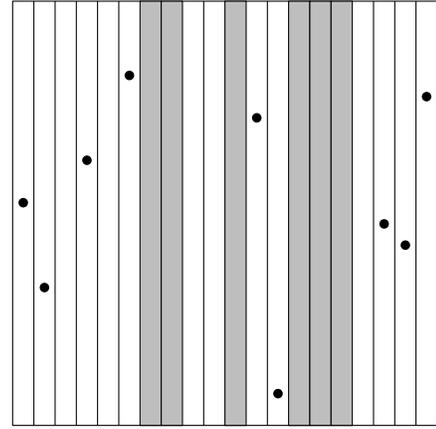
Figure 4: **Structure of the matrix** T **prior to the reduction. Shaded columns are dense. The rest of the columns are ultra-sparse, they have at most one non-zero entry.**

which have at most one non-zero entry per column. We call such columns *ultra-sparse*. The remaining m columns are *dense*. Given such a matrix, with n ultra-sparse columns and m dense columns, we can reduce it as follows.

We iterate over the rows of the matrix T from the bottom up, and consider the columns whose lowest ones fall into a given row. Let J be the set of their indices. At most one such column can be ultra-sparse because the matrices P^i are reduced. Let j denote the first column in the set J. We can subtract it from every other column in J. If one of those columns is ultra-sparse and column j is dense, then the ultra-sparse column becomes dense. To keep the number of dense columns constant, we subtract the current row (which after the column operations has a single non-zero in column j) from every other row with a non-zero entry in column j. In other words, we zero out column j, making it ultra-sparse. Algorithm 3 performs the described operations.

Since $\mathbf{Cascade}(T)$ performs operations from left to right and from bottom up, Lemma 2 immediately implies its correctness.

THEOREM 6. *Lowest ones of the matrix T reduced using* $\mathbf{Cascade}(T)$ *algorithm produce the correct pairing of the sequence of homology groups,* $\mathsf{H}(K_0^{\mathcal{C}}) \to \ldots \mathsf{H}(K_n^{\mathcal{C}})$.

What is the worst case complexity of $\mathbf{Cascade}(T)$?

THEOREM 7. $\mathbf{Cascade}(T)$ *algorithm reduces the matrix T with n ultra-sparse and m dense columns in time* $\mathrm{O}\left(n^2 m\right)$, *while keeping its size* $\mathrm{O}\left(nm\right)$.

PROOF. The initial number of nonzero elements in the matrix T is in $\mathrm{O}\left((n+m)m + n\right) = \mathrm{O}\left(nm\right)$. Since the number of dense columns is kept constant (or, more accurately, it never increases) thanks to the row operations that clear out column j, the space used during the cascade remains in $\mathrm{O}\left(nm\right)$.

It takes $\mathrm{O}\left(n+m\right)$ time to add two dense columns, or to add a dense column to a sparse column. How many such operations are there? At most m per row. There are $n + m$ rows, for a total of $\mathrm{O}\left((n+m)^2 m\right) = \mathrm{O}\left(n^2 m\right)$ operations. $\square$

Algorithm 3 Cascade algorithm for the reduction of the matrix T with ultra-sparse columns.

Cascade(T) :

    for all rows $T[i, \cdot]$, from bottom up **do**
        J = columns with the lowest non-zero entry in row $T[i, \cdot]$
        $j = \min J$
        for all $j' \in J, j' > j$ **do**
            subtract $T[\cdot, j]$ from $T[\cdot, j']$
        for all $i' < i$ with $T[i, j] \neq 0$ **do**
            subtract $T[i, \cdot]$ from $T[i', \cdot]$ # zero out all but the lowest entry of column $T[\cdot, j]$

---

5. HIERARCHY

So far we have considered the case of a single cover of the domain, a collection of simplicial complexes $\mathcal{C} = \{K^i\}_I$ with domain $K = \bigcup \mathcal{C}$. But in many applications, it is natural to build a hierarchy of such covers. For example, if the domain K triangulates a cube or a flat torus (a cube with periodic boundary conditions), both exceedingly common scenarios for simulation data, one can build a refinement of covers following an oct-tree partition of the cube. The cubes at each level of the oct-tree become the cover sets. (Technically, the cover sets are the closures of the subcomplexes of K that intersect those cubes.)

Given such a hierarchy, it becomes possible to follow the standard reduction pattern and merge sets together in pairs (or, more generally, in small groups) and thus to extend the amount of useful work a processor can do. It also limits how many dense columns m a single processor has to handle at once. Consider the prototypical oct-tree example. If we have $p = 8^k$ processors and descend down to the k-th level in the tree, we end up with p cubes and $m = 3 \cdot 2^k \cdot c$ shared simplices, where c is the number of simplices in a side of the cube. On the other hand, if we merge the cubes in pairs (proceeding to a higher level in the oct-tree after each merge), m never exceeds c, the size of the initial cut that splits the domain into two.

We can abstract the hierarchical partition of an oct-tree as a nested collection of covers $\mathcal{C}_0, \mathcal{C}_1, \ldots$, such that $K = \bigcup \mathcal{C}_0 = \bigcup \mathcal{C}_1 = \ldots$ and every cover set $L^i \in \mathcal{C}_a$ is contained in exactly one cover set $K^j \in \mathcal{C}_{a-1}$, $L^i \subseteq K^j$.

Consider the structure of the boundary matrix for two consecutive levels in the cover, illustrated in Figure 5. Suppose at level $i + 1$ the cover consists of four sets, $\mathcal{C}_{i+1} = \{K^1, K^2, K^3, K^4\}$, and at level i the cover consists of two sets $\mathcal{C}_i = \{K^1 \cup K^2, K^3 \cup K^4\}$. The procedure outlined in Sections 3 and 4 would operate independently on the matrices D^1, D^2, D^3, and D^4 (on four different processors). The first two results would then be combined by first performing row updates on the matrices Q^1 and Q^2, then reordering the matrix and reducing it using the **Cascade** algorithm. The second pair of results would be combined similarly. The two combined matrices contain the same information as the reduced boundary matrices R^{12} and R^{34} for the cover sets $K^1 \cup K^2$ and $K^3 \cup K^4$ at level i. We could proceed with the algorithm at level i, with one caveat: the higher rows (the second to last column in the figure) involved in the combination did not get updated during the execution of **Sparsify** and **Cascade** algorithms. The fix is straight-forward: when processing any given level of the cover, we perform the row operations on the full rows, rather than on their restriction to the given level. The only information necessary to construct
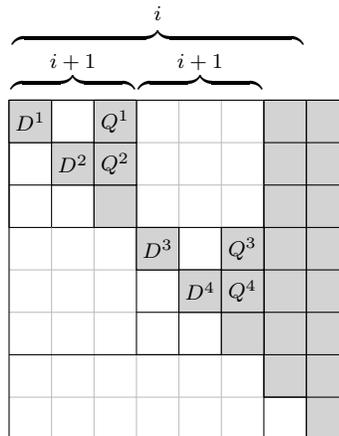
Figure 5: The structure of the boundary matrix for three consecutive levels of the hierarchy.

such rows is knowing which sets of the cover contain any given simplex.

6. EXPERIMENTS

We have implemented the described algorithm on top of MPI, and ran a strong scaling experiment on Edison supercomputer at the National Energy Research Scientific Computing Center (NERSC). Edison is a Cray XC30 system; its individual compute nodes have 24 Intel 'Ivy Bridge' processor cores, at 2.4 GHz, with 64KB and 256KB of L1 and L2 cache, respectively; the 24 cores share 64GB of RAM.

Our input is a snapshot of a combustion simulation, a $256^2 \times 512$ scalar field. The input simplicial complex is a Freudenthal triangulation of the grid, with $\sim 870 \cdot 10^6$ simplices. It is covered hierarchically via an oct-tree.

Figure 6 summarizes the running times (wallclock as reported by the PBS job resource manager). Going from 8 to 32 processes we see a near perfect scaling, with another improvement by a factor of ~ 1.5 going from 32 to 64 processes. But then the returns diminish rapidly. The behavior is not surprising: past 64 processes the binary reduction used to merge different cover sets is top-heavy, i.e., it's dominated by the merge of the information from the final two sets. As a result, adding more processes only speeds up the initial computation, which is already plenty fast.

What is worse is what the figure does not show. We have tried our code on larger data sets, but ran out of memory because the merge reduction is dominated by its final stages. Although the $(n + m)$ term in the space analysis of the **Cascade** algorithm is a gross worst-case overestimate, the
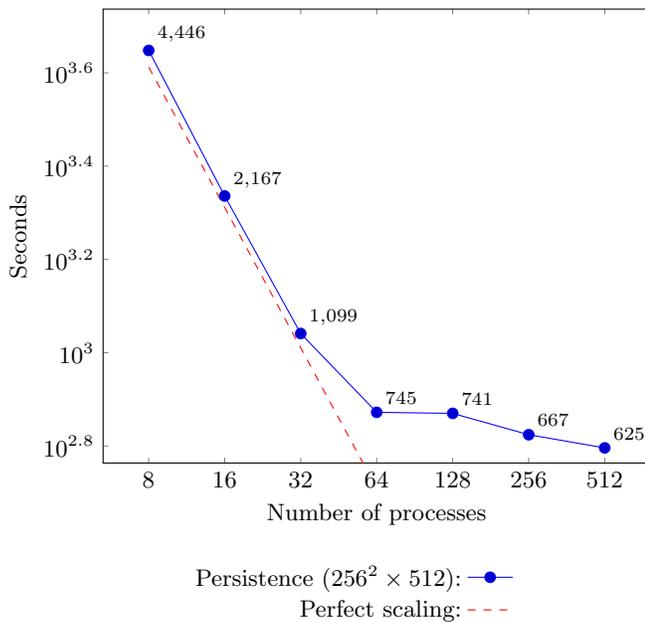
Figure 6: Times to compute persistence diagram for the $256^2 \times 512$ combustion data set.

growth of this term does follow the growth of n, the size of the domain, in many practical examples. So our technique does not solve the memory limitations of the persistence algorithm for the large data sets. We address this issue in the next section.

7. CONCLUSION

Despite the evident limitation of a merge reduction, we believe our theoretical results have a place as building blocks of a practical parallel persistence algorithm. In particular, an interesting (and, we believe, promising) research direction is combining the domain decomposition approach of our paper with the spectral sequence algorithm [28, 21]. One could initially distribute the data with respect to a domain decomposition: often such a distribution is either very cheap to compute, or entirely free in the cases when the data is supplied already decomposed directly from a simulation code, or it is stored decomposed (for I/O-efficiency). In this case, much processing could be performed on the individual chunks of the data; this part of our algorithm scales perfectly. Then, when combining the individual results, one could redistribute the remaining matrix (the input to the **Cascade** algorithm) with respect to the diagonal block partition of the spectral sequence algorithm. This way we could limit the space needed on any given processor.

Another important research topic is adapting our algorithm to the computation of persistent cohomology. Although the resulting pairing is the same, algorithms that keep track of cocycles rather than cycles have been reported to perform significantly better in practice [17, 18]. On the other hand, they are built around tracking the matrix U^{-1} in the $D = RU$ decomposition, while the algorithm that we've presented depends on manipulating the matrix R.

Similarly, it would be fruitful to understand the relationship of our algorithm to various practical optimizations [20].

Unexpectedly to us, the original optimization [1] that removes negative simplices from the columns of the matrix R cannot be used in our context. The reason is that a simplex that is negative in the filtration of a cover subcomplex K^i may be positive in the filtration of the entire space K. On the other hand, if a simplex creates a cycle in the filtration of K^i (i.e., it is positive), it must be positive in the filtration of the entire K. Thus the clearing optimization of Chen and Kerber [19] is readily applicable.

Acknowledgements

8. REFERENCES

[1] HERBERT EDELSBRUNNER, DAVID LETSCHER, AND AFRA ZOMORODIAN. Topological Persistence and Simplification. *Proceedings of the Annual Symposium on Foundations of Computer Science*, pages 454–463, 2000. *Discrete and Computational Geometry,* **28**:511–533, 2002.

[2] AFRA ZOMORODIAN AND GUNNAR CARLSSON. Computing Persistent Homology. *Discrete and Computational Geometry,* **33**:249–274, 2005.

[3] FRÉDÉRIC CHAZAL, LEONIDAS J. GUIBAS, STEVE Y. OUDOT, PRIMOŽ ŠKRABA. Persistence-Based Clustering in Riemannian Manifolds. *Journal of the ACM*, **60**, 2013.

[4] VIN DE SILVA, DMITRIY MOROZOV, MIKAEL VEJDEMO-JOHANSSON. Persistent Cohomology and Circular Coordinates. *Discrete and Computational Geometry*, **45**:737–759, 2011.

[5] ROBERT D. MACPHERSON AND BENJAMIN SCHWEINHART. Measuring Shape with Topology. *Journal of Mathematical Physics,* **53**, 2012.

[6] RIEN VAN DE WEYGAERT ET AL. Alpha, Betti and the megaparsec Universe: on the topology of the cosmic web. *Trans. Comput. Sci. XIV*, pages 60–101, 2011.

[7] DAVID COHEN-STEINER AND HERBERT EDELSBRUNNER. Inequalities for the curvature of curves and surfaces. *Foundations of Computational Mathematics*, **7**:391–404, 2007.

[8] GUNNAR CARLSSON, TIGRAN ISHKHANOV, VIN DE SILVA, AFRA ZOMORODIAN. On the Local Behavior of Spaces of Natural Images. *International Journal of Computer Vision*, **76**:1–12, 2008.

[9] HERBERT EDELSBRUNNER AND JOHN HARER. Persistent homology — a survey. *Surveys on Discrete and Computational Geometry. Twenty Years Later.*, pages 257–282, 2008.

[10] HERBERT EDELSBRUNNER AND DMITRIY MOROZOV. Persistent homology: theory and practice. *Proceedings of European Congress of Mathematics*, pages 31–50, 2012.

[11] ROBERT GHRIST. Barcodes: The persistent topology of data. *Bulletin of the American Mathematical Society*, **45**:61–75, 2007.

[12] SHMUEL WEINBERGER. What is... Persistent Homology? *Notices of the American Mathematical Society*, **58**:36–39, 2011.

[13] GUNNAR CARLSSON. Topology and data. *Bulletin of the American Mathematical Society*, **46**:255–308, 2009.

[14] NIKOLA MILOSAVLJEVIĆ, DMITRIY MOROZOV, AND PRIMOŽ ŠKRABA. Zigzag persistent homology in matrix multiplication time. *Proceedings of the Annual Symposium on Computational Geometry,* pages 216–225, 2011.

[15] HERBERT EDELSBRUNNER AND SALMAN PARSA. On the computational complexity of Betti numbers: reductions from matrix rank. *Proceedings of the Annual Symposium on Discrete Algorithms*, pages 152–160, 2014.

[16] CHAO CHEN AND MICHAEL KERBER. An output-sensitive algorithm for persistent homology. *Computational Geometry: Theory and Applications,* **46**:435–447, 2013.

[17] VIN DE SILVA, DMITRIY MOROZOV, MIKAEL VEJDEMO-JOHANSSON. Dualities in Persistent (Co)Homology. *Inverse Problems*, **27**, 2011.

[18] JEAN-DANIEL BOISSONNAT, TAMAL K. DEY, CLÉMENT MARIA. The Compressed Annotation Matrix: An Efficient Data Structure for Computing Persistent Cohomology. *Proceedings of European Symposium on Algorithms*, pages 695–706, 2013.

[19] CHAO CHEN AND MICHAEL KERBER. Persistent Homology Computation With a Twist. *Proceedings of the European Workshop on Computational Geometry*, 2011.

[20] ULRICH BAUER, MICHAEL KERBER, JAN REININGHAUS. Clear and Compress: Computing Persistent Homology in Chunks. *Topological Methods in Data Analysis and Visualization III*, pages 103-117, 2014.

[21] ULRICH BAUER, MICHAEL KERBER, JAN REININGHAUS. Distributed Computation of Persistent Homology. *Proceedings of Algorithm Engineering and Experiments (ALENEX)*, 2014.

[22] GUNNAR CARLSSON, VIN DE SILVA, DMITRIY MOROZOV. Zigzag Persistent Homology and Real-valued Functions. *Proceedings of the Annual Symposium on Computational Geometry*, pages 247–256, 2009.

[23] DAVID LIPSKY, PRIMOŽ ŠKRABA, MIKAEL VEJDEMO-JOHANSSON. A spectral sequence for parallelized persistence. arXiv:1112.1245, 2011.

[24] AFRA ZOMORODIAN AND GUNNAR CARLSSON. Localized Homology. *Computational Geometry: Theory and Applications*, **41**:126–148, 2008.

[25] RYAN H. LEWIS AND AFRA ZOMORODIAN. Multicore Homology via Mayer Vietoris. arXiv:1407.2275, submitted to *Computational Geometry: Theory and Applications*, 2014.

[26] DMITRIY MOROZOV AND GUNTHER WEBER. Distributed Merge Trees. *Proceedings of the Annual Symposium on Principles and Practice of Parallel Programming*, pages 93–102, 2013.

[27] DAVID COHEN-STEINER, HERBERT EDELSBRUNNER, AND DMITRIY MOROZOV. Vines and Vineyards by Updating Persistence in Linear Time. *Proceedings of the Annual Symposium on Computational Geometry,* pages 119–126, 2006.

[28] HERBERT EDELSBRUNNER AND JOHN HARER. *Computational Topology: an Introduction.* AMS Press, 2010.

[29] ALLEN HATCHER. *Algebraic Topology.* Cambridge University Press, 2002.

[30] GRAEME SEGAL. Classifying spaces and spectral sequences *Publications Mathématiques de l'Institut des Hautes Études Scientifiques*, **34**:105–112, 1968.