Distributed Computation of Persistent Cohomology

Arnur Nigmetov* Dmitriy Morozov*

Abstract. Persistent (co)homology is a central construction in topological data analysis, where it is used to quantify prominence of features in data to produce stable descriptors suitable for downstream analysis. Persistence is challenging to compute in parallel because it relies on global connectivity of the data. We propose a new algorithm to compute persistent cohomology in the distributed setting. It combines domain and range partitioning. The former is used to reduce and sparsify the coboundary matrix locally. After this initial local reduction, we redistribute the matrix across processors for the global reduction. We experimentally compare our cohomology algorithm with DIPHA, the only publicly available code for distributed computation of persistent (co)homology; our algorithm demonstrates a significant improvement in strong scaling.

1 Introduction Topological data analysis (TDA) is a research area at the intersection of computational geometry and algebraic topology. It aims to understand the "shape of data" by identifying prominent topological features that are stable to perturbations. One of its central constructions is persistent (co)homology [9, 10], which achieves this aim by examining the data across a range of scales and keeping track of the values where features are born and die. By pairing these events, it generates a *persistence diagram*, which is a stable descriptor that summarizes the distribution of topological features as a point set in the plane.

Persistence has found numerous applications. In cosmology, it's been used to describe the shape of the Cosmic Web [20]—the distribution of matter forming the large-scale structure of the Universe—and thus to compare different cosmological models. In materials science, persistence has been used to detect cavities and channels in nanoporous materials, information that was in turn used as an input for a machine learning algorithm to predict adsorption of a greenhouse gas [13]. In machine learning, persistence has been used to regularize the training loss [4, 18] to reduce topological complexity of the decision boundary, and thus minimize the model overfitting to outliers.

*Lawrence Berkeley National Laboratory (anigmetov@lbl.gov, dmorozov@lbl.gov).

Some of the applications of persistence require processing very large data sets. For example, cosmological simulations are some of the largest users of modern supercomputers, modeling domains on the order of $8{,}192^3$ grid cells. In other applications, even when the input data set is small, the combinatorial construction of simplicial complexes required to represent it for the persistence computation explodes the input complexity, scaling exponentially with homological dimension. These observations highlight the need for distributed algorithms and software to compute persistence, both to improve the running times and to gain access to the distributed (multi-node) memory.

Two types of distributed algorithms have been proposed in the literature. The first [14] partitions data by domain, processes as much of the computation locally as possible, and then performs a reduction among all processes to deduce the global connectivity of the topological features. The main construction is a gluing procedure called *Mayer–Vietoris blowup complex*, which adds extra cells to represent the structure of the intersecting regions of the domain. Specifically, given a cover $\mathcal{C} = \{C_i \mid i \in I\}$ of a simplicial complex K, the blowup is defined as $K^{\mathcal{C}} = \cup_{J \subseteq I} \cup_{\sigma \in K^J} \sigma \times J$, where K^J denotes the intersection $\cap_{i \in J} C_i$. In words, for every k-fold non-empty intersection of the cover elements J and for each simplex σ in the intersection, we add another simplex whose dimension is k higher. The *boundary matrix* of $K^{\mathcal{C}}$ has a special structure that allows one to perform some part of persistent *homology* computations locally. The cost is the necessity to implement the blowup construction. Also, this approach suffers from Amdahl's law: for small number of processors, the computation scales very well, since the local work dominates, but the running time quickly plateaus as the global gluing dominates the work and additional processes cannot help [14].

The second approach partitions the data by function value. The algorithm is based on the spectral sequence of the filtration [9]; it is implemented in the only publicly available code for this problem, DIPHA [2, 21]. The input matrix is reduced by blocks, moving away from the diagonal. This guarantees that each process only needs to access the data in the range it is responsible for and can pass the data to its neighbors, when

Copyright © 2026
Copyright retained by principal author's organization. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

the reduction leaves its local range.

Other techniques are available in shared memory, for example, identification of apparent pairs [15, 24] or waitfree column reduction that uses atomic operations for synchronization [16], but these do not help in the distributed memory setting.

Our contribution is three-fold.

1. We show that if one switches from persistent homology to persistent *cohomology*, the *coboundary* matrix that serves as input to the computation already has exactly the same structure as the boundary matrix of the blowup complex. This significantly simplifies the first part of the algorithm of [14], allowing us to run its local round on data partitioned by the domain, without additional combinatorial structures required by the blowup complex.
2. We combine the two approaches and after one round of local domain reduction, we switch to the spectral sequence algorithm to reduce the data globally.
3. We demonstrate that the combined approach scales significantly better than the persistent cohomology implementation in DIPHA.

2 Background

2.1 Simplicial Complexes We start with a finite *vertex set* V. For instance, V can be the set of nodes of a grid, on which a numerical simulation evaluates some function f. A *k-simplex* is a set of $k+1$ vertices. If $(k-1)$-simplex τ is a subset of k-simplex σ, $\tau \subset \sigma$, then τ is called a *facet* of σ. The set of all facets of σ forms its *boundary*, denoted $\partial\sigma$. For simplicity, we work with mod 2 coefficient (in $\mathbb{Z}/2\mathbb{Z}$), and we treat the boundary as a formal sum:

$$\partial\sigma = \sum_{\tau\subset\sigma, |\sigma\setminus\tau|=1} \tau.$$

The set of simplices K is called a *simplicial complex*, if for each $\sigma \in K$ it also contains all facets of σ (and hence, recursively, all subsets of σ). A *filtration* of K is a function on the simplicial complex, $f\colon K \to \mathbb{R}$, such that $f(\tau) \le f(\sigma)$ whenever $\tau \subset \sigma$. We call $f(\sigma)$ the *filtration value* of σ.

A filtration naturally defines a sequence of growing simplicial complexes. Let $v_1 < v_2 < \cdots < v_N$ be all filtration values $f(K)$ sorted in the ascending order. If we denote $K_i = \{\sigma \in K \mid f(\sigma) \le v_i\}$, we obtain a finite nested sequence

$$K_1 \subset K_2 \subset \cdots \subset K_N = K,$$

where each term is associated with a real number v_i. Sometimes this viewpoint is used as a definition of a filtration.

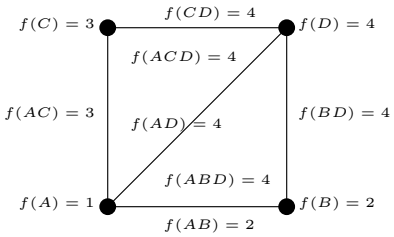


Figure 2.1: A toy example of a lower-star filtration on a 2×2 grid.

A specific construction, common for scientific computing data, that we use in our experiments is called a *lower-star* filtration. It extends a function $\hat{f} : V \to \mathbb{R}$ defined on the vertices (e.g., of a simulation domain) to a filtration $f : K \to \mathbb{R}$, defined on all the simplices (in the triangulation of the domain). It assigns to each simplex the maximum value of its vertices,

$$f(\sigma) = \max(\hat{f}(v_0), \hat{f}(v_1), \ldots, \hat{f}(v_k)). \tag{2.1}$$

Here $\sigma = \{v_0, \ldots, v_k\}$. Lower-star filtrations are used to to capture the topology of sublevel sets, $f^{-1}(-\infty, a]$ of scalar function.

Example Suppose that f is a function on a 2×2 grid (see Figure 2.1). Namely, $f(A) = 1$, $f(B) = 2$, $f(C) = 3$ and $f(D) = 4$. We triangulate the domain by adding the diagonal AD and we obtain the simplicial complex with 4 vertices, 5 edges (1-dimensional simplices) and 2 triangles (2-dimensional simplices). According to the definition, the edge AB is assigned filtration value $f(AB) = \max(f(A), f(B)) = 2$. Similarly, $f(AC) = 3$ and $f(AD) = f(CD) = f(BD) = 4$. Intuitively, we can think of $f(\sigma)$ as time when σ appears in the filtration. (2.1) can be reformulated as follows: a simplex appears in the lower-star filtration at exactly the same moment when its last vertex appears. In case of lower-star filtrations, another natural interpretation is to think of f as a piecewise-linear function on the whole square (or cube) domain: its values at vertices are given, and we extend f to the interior of each triangle by linear interpolation (using barycentric coordinates). Then we should think about *sublevel sets* $\{p \mid f(p) \le t\}$ and how they change as we sweep threshold t from $-\infty$ to ∞. A simplex is present in the filtration, if all its vertices are in the sublevel set: a simplex appears when the *highest* of its vertices appears.

Lower-star filtrations highlight that f need not be injective. However, the computation of persistence requires a total order. So we assume that all values of f are distinct, and every simplex is uniquely determined by its value $f(\sigma)$. To achieve this in practice, we assign a unique identifier `uid` to each simplex and break ties

Copyright © 2026
Copyright retained by principal author's organization. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

by comparing the pairs $(f(\sigma), \texttt{uid}(\sigma))$ lexicographically. From the nested sequence viewpoint, this means that whenever two adjacent complexes K_i and K_{i+i} differ by more than one simplex, we refine this step by adding simplices one by one (lower-dimensional simplices are always added first: an edge cannot enter the filtration before its vertices). Thus we get the filtration whose length equals number of simplices.

2.2 Coboundary matrix For a fixed simplicial complex K, simplex τ is in the coboundary of σ if and only if σ is in the boundary of τ. The set of all simplices $\tau \in K$ such that $\sigma \in \partial\tau$ is called the *coboundary* of σ. Suppose that we have a total order on the simplices of K (given by some filtration). We enumerate the simplices of K in **reverse** filtration order and define the *coboundary matrix* of K by the formula $\mathbf{D}[i, j] = 1$ if and only if the j-th simplex is in the coboundary of the i-th simplex (in other words, the *rows* of $\mathbf{D}$ encode the boundaries). It is often convenient to extend the notation and use simplices themselves to refer to the matrix entries: $\mathbf{D}[\sigma, \tau]$ means $\mathbf{D}[i, j]$ when σ has index i and τ has index j in the reverse filtration order.

Remark 2.1. We work with the most common case of $\mathbb{Z}/2\mathbb{Z}$ coefficients, which allows us to treat rows and column of all our matrices as sets of simplices. The theory of persistent homology is valid for arbitrary fields, and all algorithms in the paper can be trivially re-written for any $\mathbb{Z}/p\mathbb{Z}$ (p prime).

Example We continue using Figure 2.1. We compute the coboundary matrix in dimension 1. First, we sort 1-simplices and 2-simplices by their filtration value in **descending** order, breaking the ties arbitrarily:

$$AD \succ BD \succ CD \succ AC \succ AB, \quad ABD \succ ACD$$

where the first three edges AD, BD and CD can be permuted in any order and the two triangles can be swapped.

The coboundary matrix has size 5×2, rows indexed by the triangles and columns indexed by the edges. Entry (i, j) is 1 if and only if edge j is in the boundary of triangle i in the geometric sense:

	AD	BD	CD	AC	AB
ABD	1	1			1
ACD	1		1	1	

.

Empty entries contain 0; every row has exactly 3 entries (the boundary of each triangle contains 3 edges). In this toy example, diagonal AD is the only edge that is shared by two triangles, so it is the only column where we have more than one non-zero entry. For a regular triangulation of bigger grids most edges are shared by 2 triangles (2D case) or 4 triangles (3D case).

ALGORITHM 2.1 Reduction algorithm.

```
1: function ReduceMatrix(D)
2:   R ← D
3:   for all column R[i] of R do
4:     while R[i] ≠ 0 do
5:       while ∃j < i : low(R[j]) = low(R[i]) do
6:         R[i] ← R[i] + R[j]
7:   return R
8: end function
```

2.3 Sequential algorithm We write $\mathbf{A}[k]$ to denote the k-th column of matrix $\mathbf{A}$ and $\mathbf{A}[k, \cdot]$ to denote the k-th row of $\mathbf{A}$. The index of the last non-zero entry of a column is denoted $\texttt{low}(\mathbf{A}[k])$; for zero columns, low is undefined. We say that two columns have a *collision*, if they have the same low. Matrix $\mathbf{A}$ is called *reduced*, if there are no collisions. A *valid* column operation is adding a column from left to right: $\mathbf{A}[k] \leftarrow \mathbf{A}[k] + \mathbf{A}[k']$ when $k' < k$. We usually perform this operation, if columns k and k' have a collision. In this case, this operation eliminates the collision, and either $\mathbf{A}[k] = 0$ or $\texttt{low}(\mathbf{A}[k]) < \texttt{low}(\mathbf{A}[k'])$. A *reduction* is any sequence of valid operations that brings matrix $\mathbf{A}$ to a reduced form.

Algorithm 2.1 presents a pseudocode for a generic reduction. As written, it is not sufficiently detailed to be an algorithm: we do not specify the order in which **for all** iterates over the columns $\mathbf{R}[i]$. Nor do we specify a concrete choice of a column to the left $\mathbf{R}[j]$ that has a collision with $\mathbf{R}[i]$ (there can be more than one).

The usual way to check existence of column $\mathbf{R}[j]$ that has a collision with $\mathbf{R}[i]$ (current column being reduced) is to maintain a table of *pivots*: given a row index a, the entry $\texttt{pivots}[\texttt{a}]$ contains the index of a column with $\texttt{low}(\mathbf{R}[\texttt{pivots}[\texttt{a}]]) = a$. If no such column has been seen by the algorithm, $\texttt{pivots}[\texttt{a}] = -1$. If we process all columns from left to right and, once a column cannot be further reduced, mark it as a pivot, we obtain the standard reduction algorithm [8]. Its pseudocode is in Algorithm 2.2.

From a reduced matrix, one obtains a *persistence pairing*: simplices σ and τ form a pair, if $\texttt{low}(\mathbf{R}[\sigma]) = \tau$. Simplex σ is called *negative* and simplex τ is *positive*. Every simplex can appear in at most one pair; in particular, if τ is positive, then its column $\mathbf{R}[\tau]$ must be 0.[1] The *persistence diagram* of K in dimension d is defined as the multi-set of all pairs $(f(\sigma), f(\tau)) \in \mathbb{R}^2$ such that simplices σ and τ are paired, σ is a d-simplex (hence τ must be a $d+1$-simplex) and $f(\sigma) \neq f(\tau)$.

[1] This follows from the fundamental property of the (co)boundary operator: its composition with itself is 0.

Copyright © 2026
Copyright retained by principal author's organization. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

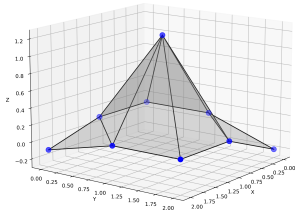
Although a reduced matrix is not unique, the pivots are, and so all reduced matrices produce the same persistence pairing [5].

We provide a detailed example of a reduction and the topological features that are captured by persistence pairs in the appendix, subsection 2.5.

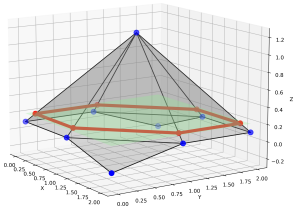
ALGORITHM 2.2 Reduction algorithm.

```
1:  function ReduceMatrixELZ(D)
2:      R ← D
3:      m, n ← number of rows and columns of R
4:      pivots = [−1, −1, . . . , −1] // array of length m
5:      for c ∈ [1, 2, . . . , n] do
6:          while R[c] ≠ 0 do
7:              ℓ ← low(R[c])
8:              p ← pivots[ℓ]
9:              if p ≠ −1 then
10:                 R[c] = R[c] + R[p]
11:             else
12:                 pivots[ℓ] ← c
13:                 break
14:     return R
15: end function
```

2.4 Clearing Clearing was first suggested in [3]. The idea follows from the structure of the (co)boundary matrix: if we know that a simplex σ is positive, then we do not need to reduce its column—it has to be 0. When computing cohomology, after reducing the coboundary matrix of k-simplices, we can identify all the $(k+1)$-simplices that appear as pivots `low` of some column and then set their columns to 0 before reducing them in dimension $k+1$.



(a) Linearly interpolated function f. The leftmost and rightmost triangles correspond to triangles DGH and BCF in Figure 2.3.



(b) Section of the graph of f with horizontal plane $z = 0.2$.

Figure 2.2: Example of a function with non-trivial homology in dimension 1. The graph of f is shown from two different viewpoints. The domain of the function is a triangulated square, see Figure 2.3.

2.5 Example of reduction To make it clear what we mean by topological features, let us consider a simple example of a function f on a 3×3 grid. We denote the vertices $A, B, \ldots, I$ and set $f(C) = -0.2$, $f(G) = -0.1$, $f(E) = 1.2$ and at all other vertices $f(v) = 0.1$, $v \in \{A, B, D, F, H\}$. The plot of f and its intersection with the plane $z = 0.2$ are shown in Figure 2.2. The plane is shown in light green, and we highlight the closed loop formed by the intersection in red. Clearly, this loop is present in the intersection with a plane $z = t$ if and only if $0.1 \le t < 1.2$; it creates a "hole" in sublevel set of f. Figure 2.3 shows the relevant part of the lower-star filtration.

(2.2)

$f(\tau)$	$f(\sigma)$	1.2 DE	1.2 EF	1.2 BE	1.2 EH	1.2 AE	1.2 EI	0.1 AB	… …
1.2	DEH	1			1				…
1.2	EHI				1		1		…
1.2	EFI		1				1		…
1.2	ADE	1				1			…
1.2	ABE			1		1		1	…
1.2	BEF		1	1					…
0.1	GDH								…
0.1	BFC								…

Formula (2.2) shows a part of the coboundary matrix in dimension 1. The numbers to the left from triangles and over edges indicate their filtration value. After we apply Algorithm 2.2 to it, we obtain the matrix in (2.3). For example, columns DE and EF require no action, because they do not have a collision. We mark DE as the pivot column responsible for eliminating lowest non-zeros in row ADE and EF is the pivot responsible for eliminating lowest non-zeros in row BEF. Column BE also has `low` $= BEF$, so the pivots table instructs us to add column EF to BE (modulo 2) to get rid of this collision. After this addition, BE has no collisions with any column to the left from it. We update the pivot table: column BE will be used to eliminate lowest non-zeros in row ABE; etc.

(2.3)

$f(\tau)$	$f(\sigma)$	1.2 DE	1.2 EF	1.2 BE	1.2 EH	1.2 AE	1.2 EI	0.1 AB	… …
1.2	DEH	1			1	1		1	…
1.2	EHI				1				…
1.2	EFI		1	1		1			…
1.2	ADE	1							…
1.2	ABE			1					…
1.2	BEF		1						…
0.1	GDH								…
0.1	BFC								…

Lowest non-zero entries in each column are highlighted, and we read off the pairing: e.g., triangle ADE is paired with edge DE. The interpretation is: when

Copyright © 2026
Copyright retained by principal author's organization. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

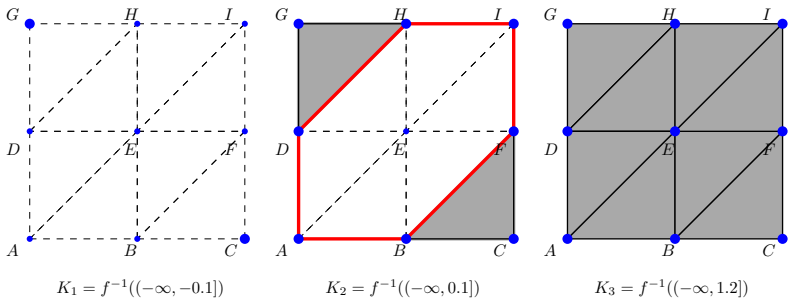


Figure 2.3: The lower-star filtration of function from Figure 2.2. Vertices which are present in the filtration at a given step are drawn with bigger markers. Edges which are not present are drawn as dashed lines. Triangles that are present are shaded. In K_2, edges in red form a closed loop which is not filled by the interior triangles.

positive simplex DE enters the filtration, it closes the loop around triangle ADE, forming a hole. This hole is filled when triangle ADE enters the filtration; recall that we call such simplices negative, meaning that they kill topological features. However, triangle ADE enters the filtration after DE only in the refined filtration, i.e., after we break ties. Since $f(ADE) = f(DE) = 1.2$, both simplices appear in the sublevel set simultaneously (in K_3 in Figure 2.3): a homological feature is born and dies immediately. This is the reason why we do not include such pairs in the persistence diagram. Other pairs highlighted in blue are equally ephemeral and also do not contribute to the persistence diagram.

The only non-trivial pair is AB and DEH (highlighted in red). AB is the last edge of the path in red that entered the filtration. All other path edges $BC, GH, \ldots$ are to the right from it in the coboundary matrix, and we omit them for space reasons. The triangle DEH is the last triangle to fill the hole formed by the red loop in K_2. Since $0.1 = f(AB) < f(DEH) = 1.2$, we add point $(0.1, 1.2)$ to the persistence diagram and we say that this hole *persists* in the filtration from 0.1 to 1.2, its *persistence* being the difference $1.2 - 0.1 = 1.1$.

This is a general way to think about persistence diagrams of scalar functions: as we vary threshold t from $-\infty$ to ∞, at some values of t holes of different dimensions are formed in the sublevel set $\{f \le t\}$, to be filled later. The moments when these holes appear and become completely filled are the birth and death coordinates of the corresponding point in the persistence diagram. For instance, 2-dimensional features appear when we consider a function of 3 variables (discretized as a 3D array): triangles form a void which is then filled by tetrahedra.

3 Distributed Algorithm As is often the case in applications, we assume our input data is spatially partitioned. We model this as a cover of the domain with sub-complexes. To make this precise, we use the following notation. Let K be a simplicial complex covered by simplicial sub-complexes K_i, $K = \bigcup_i K_i$. A simplex σ is called *interior to* K_i, if it belongs only to K_i and no other sub-complex, $\sigma \in K_i \setminus \left(\bigcup_{j \neq i} K_i \cap K_j\right)$. The rest of the simplices, $\bigcup_{i \neq j}(K_i \cap K_j)$, are called *shared*. The intersection of simplicial complexes is a simplicial complex; therefore, a simplex is shared if and only if all of its vertices are shared. If at least one vertex of a simplex lies in the interior $K_i \setminus \left(\bigcup_{j \neq i} K_i \cap K_j\right)$, then the simplex is interior.

The coboundary matrix $\mathbf{D}_i$ of K_i consists of two parts.[2] The columns of the interior simplices in $\mathbf{D}_i$ consist entirely of interior simplices and, therefore, they are the same as in the full coboundary matrix of K. The columns of shared simplices in $\mathbf{D}_i$ are subsets of the full columns in $\mathbf{D}_i$; the rows of simplices outside K_i are missing.

After the initial local reduction of matrices $\mathbf{D}_i$, we assemble the full matrix $\mathbf{D}$, partitioned among the processors. Let N be the number of columns of $\mathbf{D}$, p be the number of processors. We make processor i responsible for the column and row range $[s_i, s_{i+1})$, where the sequence

$$-\infty = s_{-1} < s_0 < s_1 < \cdots < s_{p-1} < s_p = \infty$$

is computed to split the number of columns between processors approximately evenly. We use a sample sort to identify the splitters. Finding a rank that is responsible for value v is done by binary search, as expressed in Algorithm 3.1.

The overall algorithm, expressed in pseudocode in Algorithm 3.2, consists of three parts:

1. Local reduction and sparsification. We reduce each $\mathbf{D}_i$ on a separate processor and then sparsify

[2]Normally it would be denoted $\mathbf{D}_i^{\perp}$ to distinguish it from the boundary matrix, but we only work with cohomology in this paper, so we drop the $\perp$ superscript everywhere for convenience.

Copyright © 2026
Copyright retained by principal author's organization. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

Algorithm 3.1 Obtain rank by value.

```
function RankByValue(v)
    r ← first i such that s_{i-1} ≤ v < s_i
    // r is found by binary search
    return r
end function
```

Algorithm 3.2 Full Algorithm

```
R_i ← ReduceLocal(K_i)
[s_{-1}, ..., s_p] ← SampleSort(all simplex values)
R^F ← RedistributeColumns(R_i, [s_{-1}, ..., s_p])
R^F ← Reduce(R^F)
all_done ← False
round ← 1
updated ← ∅
for dim = 0..max_dim do
    while not all_done do
        SendColumns(dim, updated, round)
        all_done ← none of the ranks sent a column
        updated ← ReceiveColumns
        round ← round + 1
    ClearColumns(dim)
```

it, as discussed in subsections 3.1 and 3.2. At this phase, rank i has the columns of $\mathbf{D}_i$ that correspond to simplices in cover sub-complex K_i (domain partitioning). The columns of shared simplices are incomplete.

2. Rearranging matrix across ranks in filtration order. After determining the splitters s_i via a sample sort, the processors exchange their domain partitioned columns, to partition them by function value. Rank i gathers a contiguous chunk of columns of $\mathbf{D}$ whose **column values** are in $[s_{i-1}, s_i)$. All columns are now complete (they include both interior and shared simplices). Each rank runs a local reduction on its own chunk once.
3. Global reduction. At this step, rank i becomes responsible for the columns whose `low` belongs to segment $[s_{i-1}, s_i)$. In a reduction loop, each rank (a) receives columns sent to it, (b) runs a local reduction using these received columns, (c) sends columns whose `low` is not in $[s_{i-1}, s_i)$ to the rank responsible for it. The loop finishes when all columns are reduced.

3.1 Local Reduction: Matrix Structure The initial local computations of our algorithm are as in [14]. We denote by $\mathring{\mathbf{D}}_i$ the submatrix of $\mathbf{D}_i$ formed by the columns that correspond to the interior simplices. We denote by $\mathbf{D}_i^S$ the submatrix of $\mathbf{D}_i$ formed by the columns that correspond to shared simplices. The

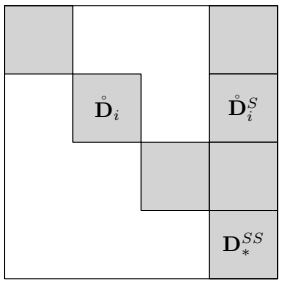


Figure 3.1: The overall structure of the coboundary matrix $\mathbf{D}$. Non-zero blocks are highlighted in gray, and different blocks of submatrix $\mathbf{D}_i$ are labeled. $\mathbf{D}_*^{SS}$ contains rows and columns from $\mathbf{D}_i^{SS}$ for all i. This coboundary matrix structure is the same as the boundary matrix structure of the blowup complex used in [14].

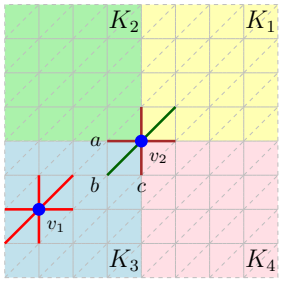


Figure 3.2: A triangulation of a 2-dimensional grid, partitioned among four processes. The four cover subcomplexes are shown in four colors. Coboundaries of interior (v_1) and shared (v_2) simplices are illustrated.

coboundary of a shared simplex can contain both shared and interior simplices. We split $\mathbf{D}_i^S$ further into the submatrix $\mathring{\mathbf{D}}_i^S$ formed by the rows of interior simplices and $\mathbf{D}_i^{SS}$ formed by the rows of shared simplices. See Figure 3.1. The order of columns and rows inside each $\mathring{\mathbf{D}}_i$, $\mathring{\mathbf{D}}_i^S$ and $\mathbf{D}_i^{SS}$ is the same as in $\mathbf{D}_i$.

An example is given in Figure 3.2. The global domain is split between 4 ranks. We use Freudenthal triangulation to create a simplicial complex K on the regular grid (dashed lines), covered by the four subcomplexes $K_1, \ldots, K_4$. The coboundary of an interior vertex v_1 consists of the 6 highlighted edges, all interior. The part of the coboundary of a shared vertex v_2 that belongs to sub-complex K_3, and therefore to the coboundary matrix $\mathbf{D}_3$ on rank 3, consists of two shared edges $\{a, v_2\}$ and $\{c, v_2\}$ and one interior edge $\{b, v_2\}$. This example also illustrates why none of the ranks have complete columns of $\mathbf{D}$ for the shared simplices.

Crucially, only the rows of simplices interior to K_i have non-zero entries in $\mathring{\mathbf{D}}_i$. In other words, each rank has complete columns of the global coboundary matrix

Copyright © 2026
Copyright retained by principal author's organization. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

$\mathbf{D}$ that correspond to its interior simplices. Therefore, rank i can locally reduce matrix $\mathring{\mathbf{D}}_i$. However, there is no way to guarantee (yet) that a local pair (σ, τ) identified from the reduced $\mathring{\mathbf{D}}_i$ is a true global pair.

Algorithm 3.3 contains pseudocode for the local part. Since we do not run any reduction operations on the columns of $\mathbf{D}_i^S$, and all further parts of the algorithm operate on matrix $\mathbf{R}$ which was initially $\mathbf{D}$, in the pseudocode we directly write the columns of matrices $\mathring{\mathbf{D}}_i^S$ and $\mathbf{D}_i^{SS}$ into the corresponding matrices $\mathring{\mathbf{R}}_i^S$, $\mathbf{R}_i^{SS}$.

Algorithm 3.3 Local Reduction.

```
1: function ReduceLocal(K_i)
2:   D_i ← coboundary matrix of K_i
3:   D̊_i ← empty matrix
4:   for all columns D_i[σ] of D_i do
5:     if σ is interior then
6:       append D_i[σ] to D̊_i
7:     else
8:       append empty columns to R̊_i^S, R_i^SS
9:       for all entries τ of D_i[σ] do
10:        if τ is interior then
11:          append τ to the last column of R̊_i^S
12:        else
13:          append τ to the last column of R_i^SS
14:  R̊_i ← ReduceMatrixELZ(D̊_i)
15: end function
```

3.2 Local Reduction: Ultrasparsification Ultrasparsification is a procedure described in [14] that allowed its authors to lower both the space and time complexity of the global hierarchical reduction for spatially partitioned data. The idea is to zero out all but one elements of the interior columns via row operations. The validity of this operation, stated in the following lemma, follows from [5].

Lemma 3.1. *The persistence pairing determined by* $\mathbf{R}$ *does not change, if in the reduction process we also perform row operations* $\mathbf{R}[i,\cdot] \leftarrow \mathbf{R}[i,\cdot]+\mathbf{R}[j,\cdot]$ *provided that we add rows bottom-up,* $j > i$.

When we are done reducing a column $\mathbf{R}[k]$, we can add the row $\texttt{low}(\mathbf{R}[k])$ to all other rows j such that $\mathbf{R}[j,k] \neq 0$. As a result, the only non-zero entry remaining in the column is the pivot. In a sequential algorithm, this would add extra work, but it would produce the same persistence pairing. In the distributed setting, this procedure both reduces the size of the columns that we send between ranks and makes column addition faster.

In our setting, we can perform this operation locally, because each rank i has complete rows of matrix $\mathbf{D}$ that correspond to the simplices of K_i. Since we split the local matrix into parts, we must apply row additions simultaneously to $\mathring{\mathbf{R}}_i$ and $\mathring{\mathbf{R}}_i^S$. Note that $\mathbf{R}_i^{SS}$, which is shared among multiple ranks, remains untouched. A convenient optimization is to perform ultrasparsification itself bottom-up, starting with the column of $\mathring{\mathbf{R}}_i$ whose `low` is maximal. Then we do not actually need to do any computations on $\mathring{\mathbf{R}}_i$, because we maintain the invariant: when we add row ℓ to row t above it, the only non-zero element in row ℓ is the lowest one of the column that we are processing, so we can just remove all elements from the column of $\mathring{\mathbf{R}}_i$ except for the lowest one. We must, however, perform a row addition on $\mathring{\mathbf{R}}_i^S$. So, while the interior columns become ultrasparse, the columns of shared simplices can actually become denser.

We perform one more operation to sparsify the columns of $\mathring{\mathbf{R}}_i^S$. In the global matrix $\mathbf{R}$, the columns of $\mathring{\mathbf{R}}_i$ and $\mathring{\mathbf{R}}_i^S$ are interleaved, following the reverse filtration order. Since we can always perform column operations, we can add the ultrasparse column $\mathring{\mathbf{R}}_i[t]$ to every column of $\mathring{\mathbf{R}}_i^S$ that succeeds it in the reverse filtration order and contains $x = \texttt{low}(\mathring{\mathbf{R}}_i[t])$. This operation removes x from the column $\mathring{\mathbf{R}}_i^S$.

Figure 3.3 illustrates the overall procedure. We write $\alpha \prec \beta$ to indicate that in the global matrix $\mathbf{R}$ the column of α precedes the column of β. Global order of simplices in matrix $\mathbf{R}$ corresponds to simplex indices: $\sigma_i \prec \sigma_{i+1}$ for $i = 1, \ldots, 4$. We start the ultrasparsification at column $\mathring{\mathbf{R}}_i[\sigma_4]$ because its pivot is the lowest. We add the row of τ_3 to the rows of τ_2 and τ_1 to remove all non-zero entries from the column. We must perform the same additions on the rows of $\mathring{\mathbf{R}}_i^S$. Then we sparsify the column $\mathring{\mathbf{R}}_i[\sigma_2]$. After the columns of $\mathring{\mathbf{R}}_i$ are ultrasparse, we can perform column operations to sparsify the rows. We add column $\mathring{\mathbf{R}}_i[\sigma_2]$ to all columns of $\mathring{\mathbf{R}}_i^S$ that are to the right from it and contain 1 in row τ_2 (in the figure, we remove τ_2 from both columns $\mathring{\mathbf{R}}_i^S[\sigma_3]$ and $\mathring{\mathbf{R}}_i^S[\sigma_5]$). When we sparsify the row of τ_3, we cannot add $\mathring{\mathbf{R}}_i[\sigma_4]$ to $\mathring{\mathbf{R}}_i^S[\sigma_1]$, because σ_4 is to the left from σ_1, so we can only remove one entry by adding $\mathring{\mathbf{R}}_i[\sigma_4]$ to $\mathring{\mathbf{R}}_i^S[\sigma_5]$.

Pseudocode for this procedure is given in Algorithm 3.4. Here we slightly abuse the notation: since we work over $\mathbb{Z}/2\mathbb{Z}$, we treat columns as sets.

3.3 Rearranging When we compute local coboundary matrix $\mathbf{D}_i$, we only have access to the values of simplices in K_i. Suppose

$$K_i = \{\sigma_1^i, \ldots, \sigma_{N_i}^i\},$$

where we list simplices in the reverse filtration order:

$$v_1^{(i)} > \cdots > v_{N_i}^{(i)}, \text{ where } v_t^{(i)} = f(\sigma_t^i).$$

Copyright © 2026
Copyright retained by principal author's organization. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

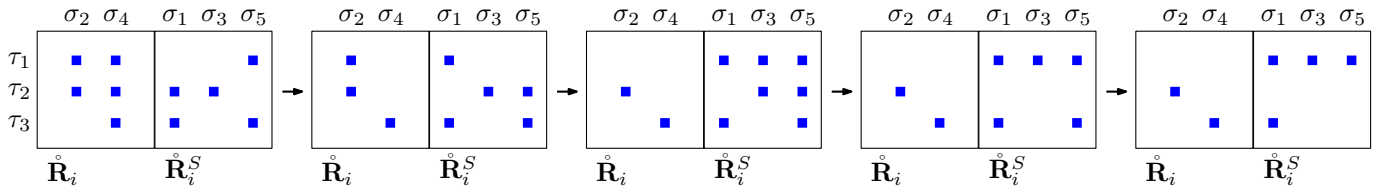


Figure 3.3: Illustration of the ultrasparsification procedure.

Algorithm 3.4 Ultrasparsification.

```
1:  function Sparsify(R̊_i, R̊_i^S)
2:      L ← sorted {low(R̊_i[c]) | non-zero columns c of R̊_i}
3:      for ℓ ∈ L do
4:          // t the column of R̊_i whose low is ℓ
5:          t ← pivots[ℓ]
6:          for all e ∈ R[t] except ℓ do
7:              R̊_i^S[e,·] ← R̊_i^S[e,·] + R̊_i^S[ℓ,·]
8:          // the only entry left is low
9:          R̊_i[t] ← {ℓ}
10:     for all column R̊_i^S[j] of R̊_i^S do
11:         for all entry e ∈ R̊_i^S[j] do
12:             if ∃p s.t. low(R̊_i[t]) = e and p ≺ j then
13:                 // removes e from R̊_i^S[j]
14:                 R̊_i^S[j] ← R̊_i^S[j] + R̊_i[p]
15: end function
```

In the compressed column representation, if a column of $\mathbf{D}_i$ is $[a_1, a_2, \dots]$, $0 \le a_1 < a_2 < \cdots$, it means that it contains simplices $\sigma^i_{a_1}$, $\sigma^i_{a_2}$, etc. In order to perform the global reduction, we must use consistent global indexing of simplices. If we order all simplices following the filtration,

$$K = \{\sigma_1, \dots, \sigma_N\}, \text{ and } v_1 > \cdots > v_N,$$

where $v_t = f(\sigma_t)$, then, since $K = \bigcup K_i$, every simplex σ^i_t in the local order of K_i maps to some $\sigma_{t'}$ in the global order,

$$\forall i \in \{1, \dots, p\},\ \forall t \in \{1, 2, \dots, N_i\},\ \exists!\, t' \text{ with } \sigma^{(i)}_t = \sigma_{t'}.$$

This generates a monotonically increasing map,

$$r_i \colon \{1, \dots, N_i\} \to \{1, \dots, N\}, \quad t \mapsto t'.$$

To go from matrices $\mathbf{R}_i$ to a global matrix $\mathbf{R}$, we must translate all entries using the map r_i.

This approach is not feasible in practice: it requires having the sorted values, $\{v_1, \dots, v_N\}$, of the entire domain in the memory of each rank. This is impossible for large datasets, especially when we take into account that the reduction algorithm is itself memory-intensive.

An alternative strategy is to avoid integral indices and to store the values themselves: instead of having columns of $\mathbf{R}^F$ of the form $[a_1, a_2, \dots]$, where a_t is the global index of $\sigma_t \in K$, we will represent the columns as sorted arrays of values v_t directly. The complexity of column addition does not change; it is the same symmetric difference.

Algorithm 3.5 Rearranging Matrix.

```
1:  function RedistributeColumns(R_i)
2:      for all columns R_i[σ] of R_i do
3:          r ← RankByValue(low(R_i[σ]))
4:          convert R_i[σ] from indices to values
5:          send R_i[σ] to r
6:      for all incoming columns R_k[σ] do
7:          // merge R_k[σ] into R^F
8:          if R^F already has column for σ then
9:              R^F[σ] ← R_k[σ] ∪ R^F[σ]
10:         else
11:             R^F[σ] ← R_k[σ]
12:     R^F ← ReduceMatrixELZ(R^F)
13: end function
```

Many columns are zeroed by local reduction, so we choose to represent the global matrix not as an `std::vector` of `std::vector`s, but as an `std::map` that uses column values as keys. The columns themselves are still stored as `std::vector`s of values. The fact that a `map` is ordered makes iterating over columns in the filtration order easy. If during the global reduction a column becomes zero, we remove it from the map entirely, to avoid storing empty vectors.

We must eliminate all collisions between columns in the whole matrix. Therefore, it is convenient to rearrange columns among ranks according to the values `low` of their pivots (this idea was proposed in [2]). We cannot determine this assignment immediately after the local reduction step: ranks do not have complete columns for shared simplices. Before we go into the global reduction loop, we rearrange the matrix according to

Copyright © 2026
Copyright retained by principal author's organization. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

column values: the column of σ is sent to rank i such that $f(\sigma) \in [s_{i-1}, s_i)$. When a rank receives columns, it needs to merge different parts of a column together. The ultrasparsification procedure does not create any conflicts, because each rank was operating on the rows that correspond to its interior simplices. After we assemble a contiguous chunk of columns $\mathbf{R}^F$ on each rank, we run a local reduction on this chunk again. The pseudocode for this part is in Algorithm 3.5. To avoid additional verbosity, in this algorithm we do not explicitly state that columns of the local matrix $\mathbf{R}_i$ are actually split between $\mathring{\mathbf{R}}_i$, $\mathring{\mathbf{R}}_i^S$ and $\mathbf{R}_i^{SS}$.

3.4 Global reduction loop The main reduction loop is conceptually simple: rank i receives columns whose `low` belongs to the range $[s_{i-1}, s_i)$ and reduces them until there are no more collisions among local columns. Columns that were reduced to zero are removed from the map; non-zero columns are sent to the rank responsible for their new `low`. Note that when we reduce a column, its `low` only gets smaller (moves up in the matrix). Therefore we only send columns from rank i to rank $j < i$.

There is one exception: in the first iteration of the loop, we must send all columns to new ranks. Recall that after rearranging column $\mathbf{R}[\sigma]$ is on the rank that corresponds to $f(\sigma)$, not to the value of its `low`. In all other iterations, we must send every column whose `low` moved out of $[s_{i-1}, s_i)$ after we received columns and reduced them. These columns are contained in the `updated` argument in Algorithm 3.6.

The local reduction procedure is slightly more complicated than the standard reduction. In Algorithm 2.2, when we reduce column $\mathbf{R}[\sigma]$ and query the pivots table for $\texttt{pivots}[\texttt{low}(\mathbf{R}[\sigma])]$, the column in the table is always to the left. In the global reduction loop, we may receive columns that lie to the left of us in the global order, and so the existing pivot may lie to the right; see Figure 3.4 for an illustration. In this case, we mark the column we are reducing as the new pivot and switch to reducing the column to the right that was marked as the pivot before. This algorithm is closely related to the algorithm in [16], where it is used for shared-memory parallelism. Its adaptation to our setting makes it simpler and obviates the need for atomic operations and complicated memory management. The resulting pseudocode is in Algorithm 3.7.

4 Clearing An important aforementioned optimization is clearing: zeroing out columns of positive simplices without reducing them explicitly. Applying it during the initial local phase is straightforward. If the row of some simplex σ contains the lowest non-zero entry of at least one column, this will be true until the

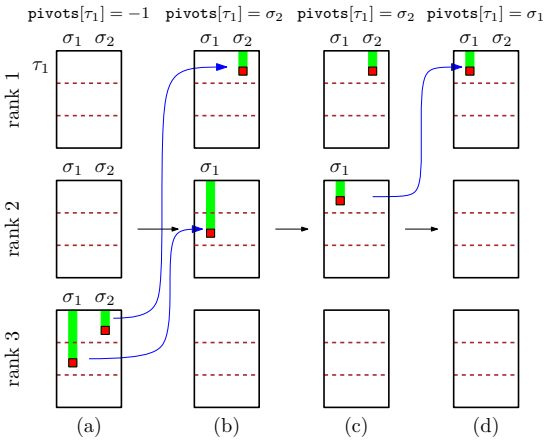


Figure 3.4: An example of the pivot to the right situation. (a) Rank 3 has $\mathbf{R}[\sigma_1]$ and $\mathbf{R}[\sigma_2]$. Rank 1 does not have any column whose `low` is τ_1, so in its local pivots table the corresponding entry is -1. (b) Based on their `low` values, rank 3 sends the columns to ranks 2 and 1, respectively. (c) After the local reduction, on rank 1, $\mathbf{R}[\sigma_2]$ has no collisions, so σ_2 remains a pivot for τ_1. Rank 2 reduces $\mathbf{R}[\sigma_1]$ (via columns that are not shown) and its new `low` is now τ_1. (d) Rank 2 sends $\mathbf{R}[\sigma_1]$ to rank 1. Now, when rank 1 runs local reduction on $\mathbf{R}[\sigma_1]$, its local pivots table points to $\mathbf{R}[\sigma_2]$ which is *to the right* from σ_1.

Algorithm 3.6 Sending columns in global reduction.

```
1:  function SendColumns(dim, updated, round)
2:      if round = 1 then
3:          // first round: rearrange all columns by low
4:          for all column R^F[σ] of R^F in dimension dim
        do
5:              r ← RankByValue(low(R^F[σ]))
6:              send R^F[σ] to rank r
7:      else
8:          for all column R^F[σ] where σ ∈ updated do
9:              // guaranteed: dim(σ) = dim
10:             r ← RankByValue(low(R^F[σ]))
11:             send R^F[σ] to rank r
12: end function
```

Copyright © 2026
Copyright retained by principal author's organization. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

ALGORITHM 3.7 Receiving Columns in Distributed Reduction.

```
1: function ReceiveColumns(C) // C: received columns
2:   updated ← ∅
3:   σ ← leftmost of all columns in C // start from σ
4:   insert columns of C into R^F
5:   while true do
6:     if σ is not the last column in R^F then
7:       σ_next ← next simplex after σ in R^F
8:     else
9:       σ_next ← σ
10:    while R^F[σ] ≠ 0 do
11:      ℓ ← low(R^F[σ])
12:      if pivots[ℓ] ≺ σ and pivots[ℓ] ≠ −1 then
13:        // standard reduction
14:        R^F[σ] ← R^F[σ] + R^F[pivots[ℓ]]
15:      else if pivots[ℓ] ≻ σ then
16:        // Pivot to the right
17:        Swap(pivots[ℓ], σ)
18:        R^F[σ] ← R^F[σ] + R^F[pivots[ℓ]]
19:      else
20:        // We saw that column and reduced it
21:        break
22:    if R^F[σ] ≠ 0 then
23:      pivots[low(R^F[σ])] ← σ
24:      if low(R^F[σ]) ∉ [s_{i−1}, s_i) then
25:        // R^F[σ] no longer belongs
26:        // to current rank i, mark it to send further
27:        insert low(R^F[σ]) into updated
28:    else
29:      erase column R^F[σ]
30:    if σ_next ≠ σ then
31:      σ ← σ_next
32:    else
33:      break
34:  return updated
35: end function
```

ALGORITHM 4.1 Global Clearing.

```
1: function ClearColumns(dim)
2:   for all non-zero columns R^F[τ] of R^F in dimension dim do
3:     σ ← low(R^F[τ]) // Dimension of σ is dim + 1
4:     r ← RankByValue(σ)
5:     send σ to r
6:   for all incoming σ do
7:     erase column R^F[σ]
8: end function
```

end of the reduction (either there will be no collisions, or this particular lowest one will be killed by adding another column *with the same* `low`). Therefore, if the column of σ happens to be at the same rank, it can be zeroed out immediately. We only need to make sure that we perform reduction in the increasing dimension order. This can be achieved by (stable) sorting the simplices in the filtration with respect to the dimension.

As for the global phase, we choose to explicitly split the matrix into dimension components and perform clearing after we finish each dimension. The reason is that before we enter the global reduction loop (before *SendColumns* is executed), the columns of the matrix in the next dimension are assigned to ranks by their column value. This allows us to easily identify the rank holding the column that we are going to zero out and send the message to this rank only, as in Algorithm 4.1.

Our experiments show that, while the fraction of the columns zeroed by the global clearing is small, this optimization is crucial in reducing the running time, because those columns would require many more communication rounds to explicitly reduce to zero.

5 Experiments We compare our implementation, called Cadmus, with DIPHA [2], the only publicly available distributed implementation of persistence. Both codes use MPI; our code uses block-parallel library DIY [17] for domain partitioning and exchange between blocks. Experiments were performed on the Perlmutter supercomputer (CPU Nodes) at the National Energy Research Scientific Center (NERSC). Every node of which has 2 AMD Epyc 7763 processors (64 CPU cores, 2.45GHz) and 512 GB of RAM. Our code and DIPHA were compiled with Clang 13, using MPI version 3.1 (Cray-MPICH 8.1.30).

For all the experiments, both DIPHA and Cadmus use *cubical* complexes, not simplicial ones described in the background. The reduction algorithm remains the same, but the basic blocks are cubes. This approach is much more natural for functions on grids and leads to filtrations of smaller size.

Timings for DIPHA are reported according to their own benchmark option. We only report the time for the reduction algorithm itself. Since we use cohomology, we also run DIPHA for cohomology.

All dataset files were downloaded from P. Klacansky's "Open scientific visualization datasets" repository [12]. We used the following data sets.

1. Woodbranch is a microCT scan of a branch of hazelnut.
2. Magnetic reconnection dataset is from [11]. It is a simulation of an interaction between magnetic fields that were concentrated in two different regions and were pointing in two different directions;

Copyright © 2026
Copyright retained by principal author's organization. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

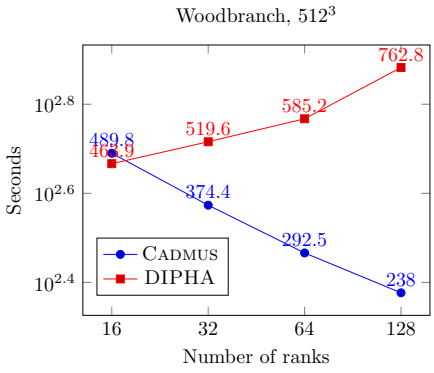


Figure 5.1: Strong scaling on a 3D scan data (Woodbranch) of size 512^3. Reduction time only.

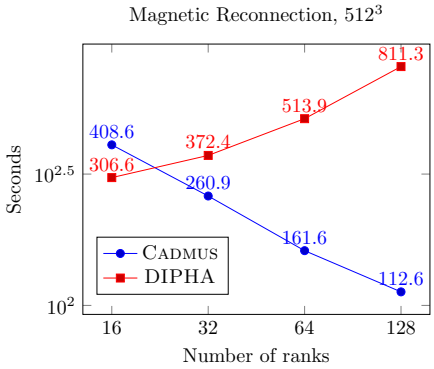


Figure 5.2: Strong scaling on magnetic reconnection data set of size 512^3. Reduction time only.

at the boundary between the regions, magnetic lines change connectivity.

3. Isotropic pressure [23]. The scalar function here is enstrophy. Its high values correspond to regions where the magnitude of the vorticity vector is large.
4. Temperature field of the rotating stratified turbulence simulation [22].

The strong scaling plots in Figures 5.1 to 5.3 are for 512^3 data sets. In all these examples Cadmus scales significantly better than DIPHA and usually outperforms it. Same holds for Figures 5.4 and 5.5, where the size is 1024^3.

It is difficult to evaluate weak scaling for persistence computation for lack of suitable data. If one takes a large data set and cuts out a small subset, one inevitably loses large-scale features that often take the most of time to process. If one coarsens the data set by smoothing, the numerous fine-scale features get eliminated. Nonetheless, we investigated the weak scaling properties of our algorithm following the second procedure. Specifically, we downsampled a 1024^3 dataset to sizes 512^3

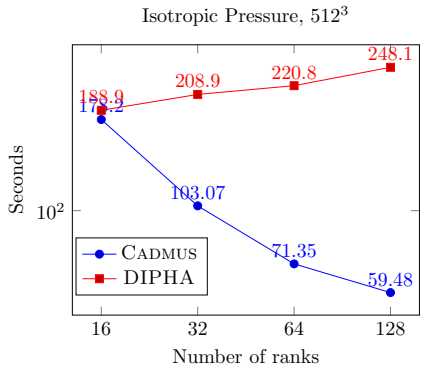


Figure 5.3: Strong scaling on isotropic pressure data set of size 512^3. Reduction time only.

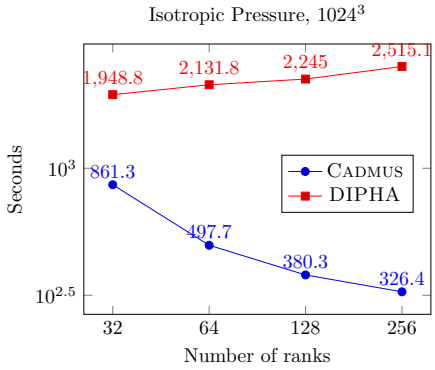


Figure 5.4: Strong scaling on isotropic pressure data set of size 1024^3. Reduction time only.

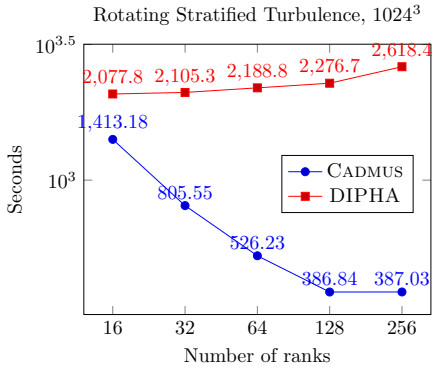


Figure 5.5: Strong scaling on the temperature field of rotating stratified turbulence simulation data set of size 1024^3. Reduction time only.

Copyright © 2026
Copyright retained by principal author's organization. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

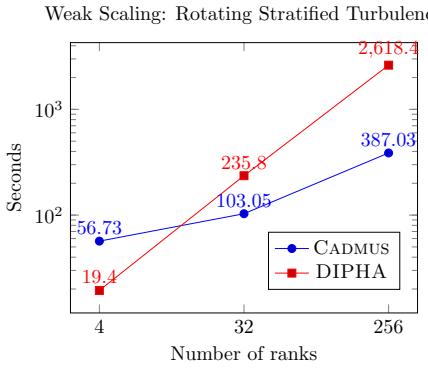


Figure 5.6: Weak scaling on the temperature field of rotating stratified turbulence simulation: data sets of size 256^3, 512^3 and 1024^3.

and 256^3 by averaging over $2 \times 2 \times 2$ voxels. Dividing the number of processors by 8 each time, each rank gets the same amount of input data. However, since persistence diagram also captures global characteristics of the input, and we effectively smoothed it, removing lots of topological features, one cannot expect ideal weak scaling. That is indeed the case for the temperature field of turbulence simulation, plotted in Figure 5.6. The running time of CADMUS increases by a factor of 2 as we go from 256^3 to 512^3 and by almost a factor of 4 as we go from 512^3 to 1024^3.

6 Conclusion We presented a new distributed algorithm for computing persistent cohomology. It combines space and range partitioning and avoids computing the Mayer–Vietoris blowup complex. Reducing data locally before switching to the global reduction is beneficial for scaling, and there is one more reason why a space partitioning approach is important — it is more GPU-friendly. If one runs local reduction on a GPU, then at least the columns that have been cleared locally need not even leave the device memory.

Although we do not discuss it in this work, persistent cohomology is known to be particularly efficient for computing persistence of Vietoris–Rips filtrations [6, 1], which makes our algorithm particularly promising for tackling large problems in that domain. Before doing so, we have to solve other subproblems involved in that computation, specifically, the efficient construction of (spacially localized) cliques in the near-neighbor graphs. We leave this as a direction for future research.

There are applications of persistence that rely on access to cocycles [7]. They immediately benefit from our new algorithm. At the same time, there are applications that need access to both cycles and cocycles (as well as their bounding chains and cochains) [19]. In other words, they need both homology and cohomology computation. Adapting our algorithm to the homology setting is another important future research direction. We use the cohomology implementation in DIPHA to get a fair comparison, but experiments with its homology implementation suggest that it is better suited for the data sets used in this paper.

Acknowledgment This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program and Mathematical Multifaceted Integrated Capability Center (MMICC) program. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

References

[1] U. BAUER, *Ripser: Efficient Computation of Vietoris–Rips Persistence barcodes*, Journal of Applied and Computational Topology, (2021), https://doi.org/10.1007/s41468-021-00071-5.

[2] U. BAUER, M. KERBER, AND J. REININGHAUS, *Distributed Computation of Persistent Homology*, in Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM, 2014, pp. 31–38.

[3] C. CHEN AND M. KERBER, *Persistent Homology Computation with a Twist*, in Proceedings 27th European workshop on Computational Geometry, vol. 11, 2011, pp. 197–200.

[4] C. CHEN, X. NI, Q. BAI, AND Y. WANG, *A Topological Regularizer for Classifiers via Persistent Homology*, in The 22nd International Conference on Artificial Intelligence and Statistics, PMLR, 2019, pp. 2573–2582.

[5] D. COHEN-STEINER, H. EDELSBRUNNER, AND D. MOROZOV, *Vines and Vineyards by Updating Persistence in Linear Time*, in Proceedings of the Twenty-Second Annual Symposium on Computational Geometry, 2006, pp. 119–126.

[6] V. DE SILVA, D. MOROZOV, AND M. VEJDEMO-JOHANSSON, *Dualities in Persistent (Co)Homology*, Inverse problems, 27 (2011), p. 124003, https://doi.org/10.1088/0266-5611/27/12/124003.

[7] V. DE SILVA, D. MOROZOV, AND M. VEJDEMO-JOHANSSON, *Persistent Cohomology and Circular Coordinates*, Discrete & Computational Geometry,

Copyright © 2026
Copyright retained by principal author's organization. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

45 (2011), pp. 737–759, https://doi.org/10.1007/s00454-011-9344-x.

[8] Edelsbrunner, Letscher, and Zomorodian, *Topological persistence and simplification*, Discrete & Computational Geometry, 28 (2002), pp. 511–533.

[9] H. Edelsbrunner and J. Harer, *Persistent Homology — a Survey*, Contemporary Mathematics, (2008).

[10] H. Edelsbrunner and D. Morozov, *Persistent Homology*, in Handbook of Discrete and Computational Geometry, J. E. Goodman, J. O'Rourke, and C. D. Tóth, eds., CRC Press, 2017, https://www.csun.edu/~ctoth/Handbook/chap24.pdf.

[11] F. Guo, H. Li, W. Daughton, and Y.-H. Liu, *Formation of Hard Power Laws in the Energetic Particle Spectra Resulting from Relativistic Magnetic Reconnection*, Phys. Rev. Lett., 113 (2014), p. 155005, https://doi.org/10.1103/PhysRevLett.113.155005.

[12] P. Klacansky, *Open Scientific Visualization Datasets*. https://klacansky.com/open-scivis-datasets, 2021.

[13] A. S. Krishnapriyan, M. Haranczyk, and D. Morozov, *Topological Descriptors Help Predict Guest Adsorption in Nanoporous Materials*, Journal of Physical Chemistry C, 124 (2020), pp. 9360–9368, https://doi.org/10.1021/acs.jpcc.0c01167.

[14] R. Lewis and D. Morozov, *Parallel Computation of Persistent Homology Using the Blowup Complex*, in Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures, 2015, pp. 323–331.

[15] R. Mendoza-Smith and J. Tanner, *Parallel Multi-scale Reduction of Persistent Homology Filtrations*, arXiv [math.AT], (2017), https://arxiv.org/abs/1708.04710.

[16] D. Morozov and A. Nigmetov, *Towards Lock-free Persistent Homology*, in Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures, 2020, pp. 555–557.

[17] D. Morozov and T. Peterka, *Block-Parallel Data Analysis with DIY2*, in Proceedings of the 6th Symposium on Large Data Analysis and Visualization (LDAV), IEEE, 2016, pp. 29–36.

[18] A. Nigmetov, A. Krishnapriyan, N. Sanderson, and D. Morozov, *Topological Regularization via Persistence-Sensitive Optimization*, Computational Geometry, (2024), p. 102086.

[19] A. Nigmetov and D. Morozov, *Topological Optimization with Big Steps*, Discrete & Computational Geometry, (2024), pp. 1–35, https://doi.org/10.1007/s00454-023-00613-x.

[20] P. Pranav, H. Edelsbrunner, R. van de Weygaert, G. Vegter, M. Kerber, B. J. T. Jones, and M. Wintraecken, *The Topology of the Cosmic Web in Terms of Persistent Betti Numbers*, Monthly Notices of the Royal Astronomical Society, 465 (2017), pp. 4281–4310, https://doi.org/10.1093/mnras/stw2862.

[21] J. Reininghaus, U. Bauer, and M. Kerber, *DIPHA: A Distributed Persistent Homology Algorithm*. https://github.com/DIPHA/dipha, 2014–2016.

[22] D. Rosenberg, A. Pouquet, R. Marino, and P. D. Mininni, *Evidence for Bolgiano-Obukhov Scaling in Rotating Stratified Turbulence Using High-Resolution Direct Numerical Simulations*, Physics of Fluids, 27 (2015).

[23] P. Yeung, D. Donzis, and K. Sreenivasan, *Dissipation, Enstrophy and Pressure Statistics in Turbulence Simulations at High Reynolds Numbers*, Journal of Fluid Mechanics, 700 (2012), p. 5.

[24] S. Zhang, M. Xiao, C. Guo, L. Geng, H. Wang, and X. Zhang, *HYPHA: A Framework based on Separation of Parallelisms to Accelerate Persistent Homology Matrix Reduction*, in Proceedings of the ACM International Conference on Supercomputing, 2019, pp. 69–81, https://dl.acm.org/doi/abs/10.1145/3330345.3332147.

Copyright © 2026
Copyright retained by principal author's organization. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.