Brief Announcement: Towards Lockfree Persistent Homology

Dmitriy Morozov
Lawrence Berkeley National Laboratory
dmorozov@lbl.gov

Arnur Nigmetov
Lawrence Berkeley National Laboratory
anigmetov@lbl.gov

ABSTRACT

Persistent homology, which describes the shape of data by quantifying the sizes of its topological features, is one of the most ubiquitous algorithms in topological data analysis. All existing algorithms that compute persistence in parallel rely on the algebraic structure of the problem to subdivide the computation, either by partitioning the range, or the domain of the underlying scalar measurement. Instead, we exploit the inherent parallelism of the reduction algorithm and rely on hardware synchronization primitives, namely compare-and-swap operations, to develop a lockfree shared-memory algorithm that avoids having to decide how to partition the underlying data set. We demonstrate the algorithm's performance and scaling using a set of computational experiments.

ACM Reference Format:
Dmitriy Morozov and Arnur Nigmetov. 2020. Brief Announcement: Towards Lockfree Persistent Homology. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20), July 15–17, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3350755.3400244

1 INTRODUCTION

Persistent homology is a key method in the field of Topological Data Analysis. It describes the shape of a data set, such as a high-dimensional point cloud or a scalar measurement, by quantifying the distribution of its topological features across scales. The technical (algebraic topological) details are not needed for this paper, so we summarize them intuitively. Persistence tracks how topology of a data set changes, as we vary some natural threshold. For example, given a point cloud, we can grow balls around the points and track at what radii *holes* (components, loops, voids, and their higher dimensional analogs) appear and disappear. There exists a unique pairing between such events, and the entire process can be summarized as a set of birth–death pairs, called a *bar code* or a *persistence diagram*.

To compute persistence, one captures the topology of the data in a *filtered simplicial complex*, which is a generalization of a graph. A boundary matrix represents the nesting relationship between different simplices (vertices, edges, triangles, tetrahedra, etc.). All algorithms for computing persistence manipulate the boundary matrix. The core of the procedure is a *reduction* of this matrix, which follows a special form of Gaussian elimination over a finite field (typically $\mathbb{Z}_2$).

ACM ISBN 978-1-4503-6935-0/20/07.
https://doi.org/10.1145/3350755.3400244

Several methods exist for parallel computation of persistence. All of them exploit algebraic structure of the problem and broadly fall into two categories: (1) methods that partition the range of the underlying scalar measurement and follow the *spectral sequence of the filtration*; (2) methods that partition the domain of the underlying scalar measurement and follow the *Mayer–Vietoris spectral sequence*. Examples of the former approach include PHAT for shared memory [3] and DIPHA for distributed memory [2]. Examples of the latter include computation using Mayer–Vietoris blowup complex [10]. HYPHA [13] and Ripser++ [14] accelerate the persistence reduction by preprocessing the data on a GPU.

In contrast, we explore a different tactic in this paper. Instead of making decisions about how to partition the data — decisions that are especially complicated when decomposing the domain (NP-hard in certain formulations [11]) — we take advantage of the inherent parallelism of the reduction algorithm itself and rely on the appropriate synchronization primitives, specifically, compare-and-swap operations, to develop its lockfree version. A key advantage of this approach is that our algorithm is a drop-in replacement for the original reduction algorithm and can be combined with other optimizations, including their state-of-the-art collection Ripser [1].

2 PERSISTENCE ALGORITHM

We refer the reader to a book by Edelsbrunner and Harer [7] or a book chapter by Edelsbrunner and Morozov [8] for detailed introduction to the relevant background.

The filtered simplicial complex is represented as a boundary matrix, D, which is reduced to a matrix R. To perform this reduction and to extract persistence pairs, for any matrix M, let $\mathbf{low}(M[j])$ denote the row of the lowest non-zero entry in the j-th column. The function is undefined if the column is zero. Edelsbrunner, Letscher, and Zomorodian [9] introduced the following greedy Algorithm 1 for computing persistence.

Algorithm 1 Original Persistence Algorithm [9]

```
1: $R := D$
2: for $j = 0$ to $n-1$ do
3:     while $\exists$piv $< j$ such that $\mathbf{low}(R[j]) = \mathbf{low}(R[\text{piv}])$ do
4:         $R[j] := R[j] + \alpha R[\text{piv}]$
```

Here $\alpha = -R[\mathbf{low}(j), j]/R[\mathbf{low}(\text{piv}), \text{piv}]$. The role of the column operation is to cancel the lowest non-zero entry in column $R[j]$, until it is either unique leftmost such entry, or the column $R[j]$ becomes 0. When the algorithm terminates, the map **low** is injective: all lowest non-zero entries of matrix R are in unique rows. We call matrix R that satisfies this condition *reduced.* The map **low** gives a pairing between simplicies, which lets us determine the persistence barcode: the i-th simplex creates a "hole" that the j-th simplex kills if and only if $\mathbf{low}(R[j]) = i$.

Edelsbrunner et al. [5] reinterpret Algorithm 1 as a matrix decomposition. If we initialize an auxiliary matrix V to be identity, and perform operations on V in parallel to those on R — we add $V[j] := V[j] + \alpha V[\texttt{piv}]$ to the inner loop of Algorithm 1 — it follows immediately that the algorithm computes a matrix decomposition, $R = DV$, where R is reduced and V is upper-triangular with all ones on the diagonal (so it is invertible) Edelsbrunner et al. [5] show that any such decomposition determines the same map **low** on matrix R, and therefore provides the same simplex pairing.

Lemma 2.1 (Pairing Uniqueness Lemma [5]). *Letting $R = DV$, if R is reduced and V is invertible upper-triangular, then the map* $\mathbf{low}(R[\cdot])$ *is unique.*

This lemma has an important algorithmic implication: It does not matter in what order we add the columns of R. As long as all such operations happen from left to right and matrix R gets reduced, we get the correct pairing. This is the starting point for our work; it suggests a tremendous amount of parallelism. We can operate on multiple columns of R simultaneously. As long as the operations are atomic and left-to-right, once we get a reduced matrix, we get the same (correct) answer.

Parallel algorithm. It is conceptually straightforward to implement this idea. We split matrix R into chunks, create a pool of threads, and let each thread process the next available chunk, as in Algorithm 2. The chunking plays a dual role: (1) it helps the outer parallel loop to scale better (as is a standard technique); (2) it breaks up the computation into epochs separated by *quiescent states*, which is necessary for memory management, an important detail that we omit from this abridged version of the paper.

Algorithm 2 Outer parallel loop

```
parallel for chunk ∈ [0, #chunks) do
    compute start, end from chunk
    REDUCE_CHUNK(start, end)
    MM.QUIESCENT()
```

All the real work is done in Algorithm 3. To determine which column of matrix R has the lowest non-zero entry in a particular row, we use a `vector` of integers, `pivots`: `pivots`$[i] = j$ iff $\mathbf{low}(R[j]) = i$. Because this vector is modified by multiple threads at once, we make it a `vector` of `atomic` integers. The matrix R is stored as a `vector` of `atomic` pointers to columns, which in turn are sorted `vectors` of indices of the non-zero entries. Each thread reduces column $R[j]$ by making its local copy `curr_column` and working on it. Only when the thread finishes reducing `curr_column` is the pointer to `curr_column` written to R.

To accommodate the possibility that after we read the pivot `piv := pivots`$[\ell]$ and before we read the pivot column $R[\texttt{piv}]$, another thread updates $R[\texttt{piv}]$, we must check that $\mathbf{low}(R[\texttt{piv}]) = \mathbf{low}(\texttt{curr\_column})$. If this condition fails, we re-read the pivot. That's the point of the **`while`**-loop in line 6 of Algorithm 3.

Once the pivot is read, there are three possibilities: the pivot doesn't exist, it lies to the left of the current column, or it lies to the right. These correspond to the three conditions of the **`if`**-statement. In the first case, we record the current column as the pivot; in the second case, we perform an ordinary column reduction; in the third case, we overwrite the pivot with the current column and switch to reducing the old pivot column.

Algorithm 3 Chunk reduction with CAS synchronization.

```
1:  function REDUCE_CHUNK(start, end)
2:      for $j$ := start to end do
3:          curr_column := copy of $R[j]$
4:          while curr_column $\neq 0$ do
5:              $\ell$ := low(curr_column)
6:              do                                          ▷ read pivot
7:                  piv := pivots[$\ell$]
8:                  piv_column := $R$[piv]
9:              while $\ell \neq$ low(piv_column)
10:             if piv = $-1$ then
11:                 MM.RETIRE($R[j]$)
12:                 $R[j]$ := curr_column
13:                 if CAS(pivots[$\ell$], piv, $j$) then      ▷ write pivot
14:                     break                          ▷ go to next column
15:                 else
16:                     goto 3             ▷ start over with curr_column
17:             else if piv < $j$ then
18:                 curr_column := curr_column + $\alpha \cdot$ piv_column
19:             else if piv > $j$ then
20:                 MM.RETIRE($R[j]$)
21:                 $R[j]$ := curr_column
22:                 if CAS(pivots[$\ell$], piv, $j$) then      ▷ write pivot
23:                     $j$ := piv                      ▷ reduce next column
24:                 goto 3
25:         if curr_column = 0 then
26:             MM.RETIRE($R[j]$)
27:             $R[j]$ := curr_column
```

We must be careful when updating `pivots`. Two threads from different chunks may end up with the same lowest non-zero, and they may try to record their columns as pivots at the same time. We use the standard compare-and-swap idiom to solve this problem. The two **`if`**-statements in Algorithm 3 in lines 13, 22 ensure that in such a situation only one thread writes to `pivots` atomically, and the other threads will be notified via a failed `CAS` that they must start over the reduction of the current column.

We omit the discussion of correctness for lack of space, but note that there cannot be any conflict in updating the columns of matrix R: only one thread can be working on reducing a particular column $R[j]$, and therefore only one thread will try to record this column. The columns interact with each other through the `pivots` vector, and so the change of the column R becomes visible to the other columns during the `CAS` operation that follows the assignment. In between, once the column $R[j]$ is updated, but not its pivot, the column is inconsistent — `pivots`$[\mathbf{low}(R[j])] \neq j$ — and any thread that tries to read it will fail in the **do-while**-loop in line 6.

Each `CAS` operation (for fixed `piv`, ℓ, j) can fail at most $n - j$ times, since every failure means the pivot is moved to the left by another thread. It follows that Algorithm 3 is waitfree. It is not difficult to check that the matrix R ends up being reduced, when the algorithm terminates, even though the columns are processed out of order. Because column operations are still performed from left to right, Lemma 2.1 implies that we get the correct pairing, given by the function $\mathbf{low}(R[\cdot])$.

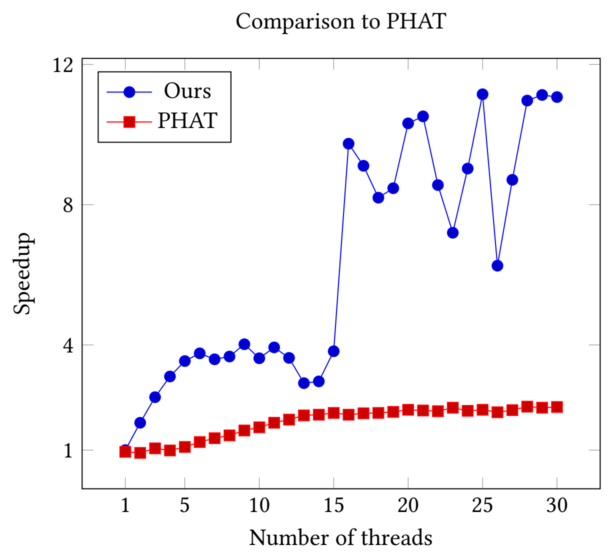

Figure 1: Speedup for varying number of threads on the upper-star filtration of a density field from a cosmological simulation.

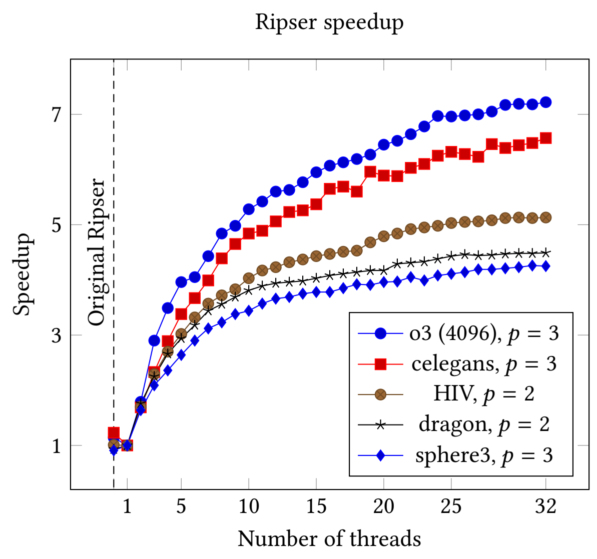

Figure 2: Speedup for varying number of threads on the Vietoris–Rips complexes of different data sets, processed using original and parallel versions of Ripser.

3 EXPERIMENTAL RESULTS

We ran our experiments on a dual socket machine with 40 Intel Xeon Gold 6230 cores (20 cores in each socket), and 128 GB of RAM. All timings are averaged over 5 runs. We measure only the reduction time, without I/O and initialization.

We compare our algorithm with the chunk reduction algorithm implemented in PHAT [3], which combines two optimizations, clearing [4] and compression [9]. We have integrated the clearing optimization in our implementation as well. Figure 1 illustrates the speedup relative to the time it takes our code using a single thread to compute persistence of the density function on a snapshot of a cosmological simulation. (The data is 128^3, and its upper-star filtration contains 53.5 million simplices.)

We have also incorporated our algorithm into Ripser [1], a state-of-the-art software to compute persistence of Vietoris–Rips complexes. It incorporates a large number of optimizations as well as ideas about computing persistent cohomology, while maintaining the matrix V instead of the matrix R [1, 6]. Figure 2 illustrates the speedup relative to the single-threaded version of our code. Note that this is typically a little slower than unmodified Ripser (illustrated in the figure with the points on the dashed line) because of the extra overhead from using atomic operations. The figure shows multiple datasets, explained in the original Ripser paper [1] and by Otter et al. [12]. The homological dimension used for each data set is specified in the legend.

4 CONCLUSION

We introduced a non-blocking algorithm for computing persistent homology, which exploits the inherent parallelism of the problem. Its main subroutine is waitfree, but not its outer loop. A key open question is how to make the entire algorithm waitfree.

ACKNOWLEDGEMENTS

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] U. Bauer. Ripser: efficient computation of Vietoris-Rips persistence barcodes, August 2019. arXiv:1908.02518.
[2] U. Bauer, M. Kerber, and J. Reininghaus. Distributed computation of persistent homology. In *2014 proceedings of the sixteenth workshop on algorithm engineering and experiments (ALENEX)*, pages 31–38. SIAM, 2014.
[3] U. Bauer, M. Kerber, J. Reininghaus, and H. Wagner. Phat–persistent homology algorithms toolbox. *Journal of symbolic computation*, 78:76–90, 2017.
[4] C. Chen and M. Kerber. Persistent homology computation with a twist. In *Proceedings 27th European Workshop on Computational Geometry*, volume 11, pages 197–200, 2011.
[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*, SCG '06, pages 119–126, New York, NY, USA, 2006. ACM.
[6] V. de Silva, D. Morozov, and M. Vejdemo-Johansson. Dualities in persistent (co)homology. *Inverse problems*, 27(12):124003, November 2011.
[7] H. Edelsbrunner and J. Harer. *Computational topology: an introduction*. American Mathematical Soc., 2010.
[8] H. Edelsbrunner and D. Morozov. Persistent homology. In Jacob E Goodman, Joseph O'Rourke, and Csaba D Tóth, editors, *Handbook of Discrete and Computational Geometry*. CRC Press, 2017.
[9] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. *Discrete & computational geometry*, 28(4):511–533, November 2002.
[10] R. Lewis and D. Morozov. Parallel computation of persistent homology using the blowup complex. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, pages 323–331, New York, NY, USA, 2015. ACM.
[11] R. Lewis and A. Zomorodian. Multicore homology via mayer vietoris, July 2014. arXiv:1407.2275.
[12] N. Otter, M. A. Porter, U. Tillmann, P. Grindrod, and H. A. Harrington. A roadmap for the computation of persistent homology. *EPJ Data Science*, 6(1):17, 2017.
[13] 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*, pages 69–81, 2019.
[14] S. Zhang, M. Xiao, and H. Wang. GPU-Accelerated computation of Vietoris-Rips persistence barcodes. March 2020.