A Fast and Scalable Computational Topology Framework for the Euler Characteristic

Daniel J. Laky and Victor M. Zavala*
Department of Chemical and Biological Engineering
University of Wisconsin-Madison, 1415 Engineering Dr, Madison, WI 53706, USA

Abstract

The Euler characteristic (EC) is a powerful topological descriptor that can be used to quantify the shape of data objects that are represented as fields/manifolds. Fast methods for computing the EC are required to enable processing of high-throughput data and real-time implementations. This represents a challenge when processing high-resolution 2D field data (e.g., images) and 3D field data (e.g., video, hyperspectral images, and space-time data obtained from fluid dynamics and molecular simulations). In this work, we present parallel algorithms (and software implementations) to enable fast computations of the EC for 2D and 3D fields using vertex contributions. We test the proposed algorithms using synthetic data objects and data objects arising in real applications such as microscopy, 3D molecular dynamics simulations, and hyperspectral images. Results show that the proposed implementation can compute the EC a couple of orders of magnitude faster than `GUDHI` (an off-the-shelf and state-of-the art tool) and at speeds comparable to `CHUNKYEuler` (a tool tailored to scalable computation of the EC). The vertex contributions approach is flexible in that it can compute the EC as well as other topological descriptors such as perimeter, area, and volume (`CHUNKYEuler` can only compute the EC). Scalability with respect to memory use is also addressed by providing low-memory versions of the algorithms; this enables processing of data objects beyond the size of dynamic memory. All data and software needed for reproducing the results are shared as open-source code.

Keywords: Euler characteristic; topology; data science; fields; parallel computing

1 Introduction

Data objects that appear in the form of fields (e.g., images, video, space-time data) are common in science and engineering. These data objects can be processed using powerful algorithms such as convolution operations, Fourier transforms, and singular value decomposition to extract information and to enable reduction and visualization. Convolutional neural networks (CNNs), in particular, are a highly flexible tool that can be used to extract diverse types of feature information from field data. However, CNNs have significant scalability limitations (e.g., require repetitive convolutions to learn

*Corresponding Author: victor.zavala@wisc.edu

adequate operators) and might require large amounts of data to be trained. Moreover, attributing meaning to features extracted from CNNs is not straight-forward [10].

Topological data analysis (TDA) has recently emerged as a powerful framework to quantify the shape of field data [22, 23, 10]. A simple topological descriptor known as the Euler Characteristic (EC), in particular, has gained significant attention in diverse applications [12]. The EC is a descriptor that captures basic topological features of binary fields (e.g., connected components, voids, holes) and can be extended to continuous fields by using filtration/percolation procedures (i.e., the EC is computed at different filtration values). The EC has also been used for analyzing data in neuroscience [11, 6], medical imaging [13, 8], cosmology [19, 15], and plant biology [2]. The EC has also been recently used as a descriptor/feature to train simple machine learning models (e.g., linear regression) that have comparable prediction accuracy to those of CNNs but that are significantly less computationally expensive to build [23, 10, 21].

Enabling fast computations of topological descriptors is necessary to handle field data at high resolutions, high-throughput data, and to enable real-time applications (e.g., control). Methods for fast processing of small-scale images was proposed by Snidaro and Foresti [24]; specifically, they proposed to compute only the change in the EC over the filtration values. This idea was further explored in [9, 30, 17, 18]. Heiss and Wagner presented an algorithm that computes the EC for 3D fields that are too large to fit into memory and provide a software implementation called `CHUNKYEuler`[9]. A parallel implementation of this method using GPUs has also been recently developed [30].

In this work, we provide parallel implementations of the vertex contributions methods of Snidaro and Foresti [24]. We highlight that a major contribution of this work is the generalization of of this method to 3D fields (including handling of non-binary fields and parallel implementation); these capabilities allow us to process a broad range of data sets arising in applications. The vertex contributions method is scalable and flexible in that contributions can be used to compute the EC and other relevant topological descriptors such as perimeter, area, and volume (the approach implemented in `CHUNKYEuler` can only compute the EC). We provide background information on the computation of the EC with an in-depth look at 2D/3D field data and level set filtration and outline the vertex contribution method. We highlight aspects that make the proposed method highly parallelizable and describe our software implementation. In addition, we benchmark our implementations against state-of-the-art computational topology tools using synthetic data sets and data sets arising in real applications. Specifically, we analyze synthetic random fields that are systematically generated to obtain field data at different resolutions and we study data sets arising in real applications such as microscopy, molecular simulations, and hyperspectral images. Our results demonstrate that our implementation can compute the EC a couple of orders of magnitude faster than the off-the-shelf computational topology tool `GUDHI` [16]. We also compare times of the proposed methods to those implemented in the `CHUNKYEuler` software [9] and their GPU implementation [30].

2 Methodology

The EC of a data object is computed by counting the contribution of each fundamental component to the overall topology of the object. These fundamental components, which we will refer to as simplexes, are the building blocks that contribute to the topology of the object. The simplexes that are important to this work, and more broadly 2D/3D field analysis, include 0-dimensional through 3-dimensional cubical simplexes (Figure 1). In this work we will refer to a 0-dimensional cubical simplex as a vertex, 1-dimensional as an edge, 2-dimensional as a face or pixel, and 3-dimensional as a cell or voxel. A 2D field data object is a collection of face/pixel data (e.g., images) that contains intensity information to describe data at a particular position within the object. By representing each pixel as a face, an image object may be represented as a collection of cubical simplexes (Figure 2).

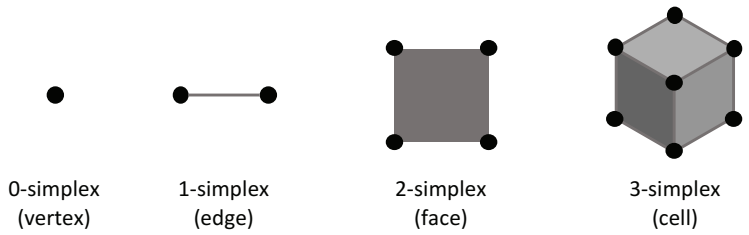


Figure 1: Cubical simplexes relevant to 2D and 3D field processing.

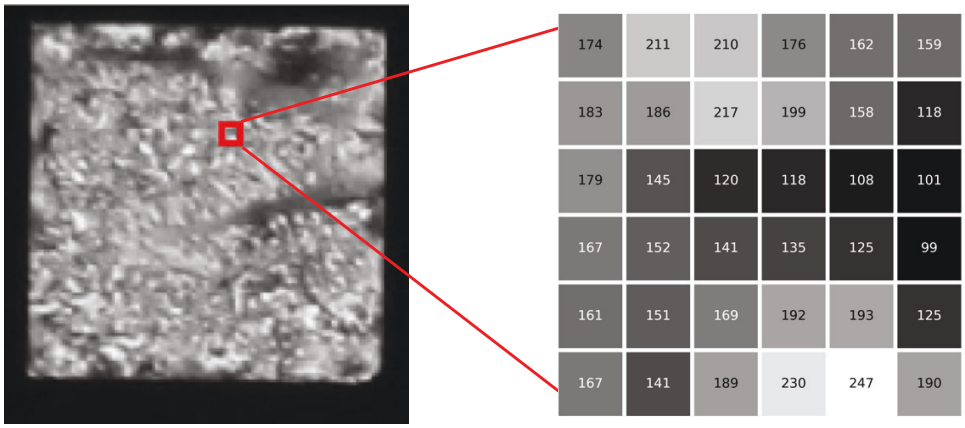


Figure 2: Liquid crystal micrograph (image) represented as a cubical simplicial complex; this is done by assigning pixel values to faces which are connected by lower dimensional cubical simplexes (i.e., edges and vertices).

The EC can be defined as the alternating sum of cubical simplexes; for 3D fields, this is:

$$\chi = V - E + F - C. \tag{2.1}$$

Here, V is the number of vertices, E is the number of edges, F is the number of faces/pixels, C is the number of cells or voxels, and χ represents the EC value.

To generate the EC for a field (a continuous object), one must first transform the original field into a binary field by applying a filtration at a desired face/pixel intensity level. A filtration/percolation is a function that is used to define which components (e.g., pixels in the case of a 2D image) should be included and which should not. The filtration function used throughout this work is a sublevel filtration:

$$g_c^-(f) = \{(x_w, x_h) | f(x_w, x_h) \leq c\}. \tag{2.2}$$

Here, c is the filtration level, faces are defined at position (x_w, x_h) with intensity f, and set g_c^- contains faces that are less than or equal to c only. Importantly, when a face is included, all edges and vertices relevant to that face are also included in the set. An example of filtration for multiple values of c over a small image is shown in Figure 3.

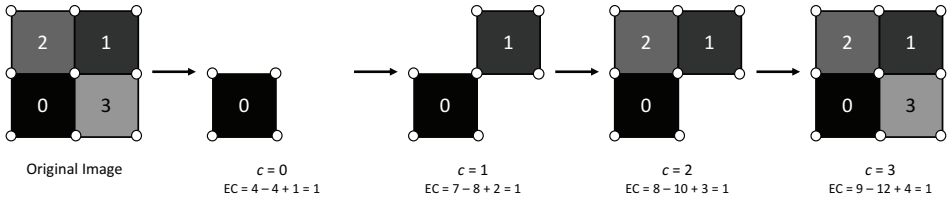


Figure 3: Example 2D field undergoing the process of filtration over the full range of its pixel intensity values.

The EC at each filtration value in Figure 3 can be verified by counting the number of components included and utilizing (2.1). A generalization of the EC, called the EC curve, encodes information for the entire field by applying g_c^- for values of c that cover the entire range of face intensities for a given field. A small example of an EC curve is shown in Figure 4 for a 2D field.

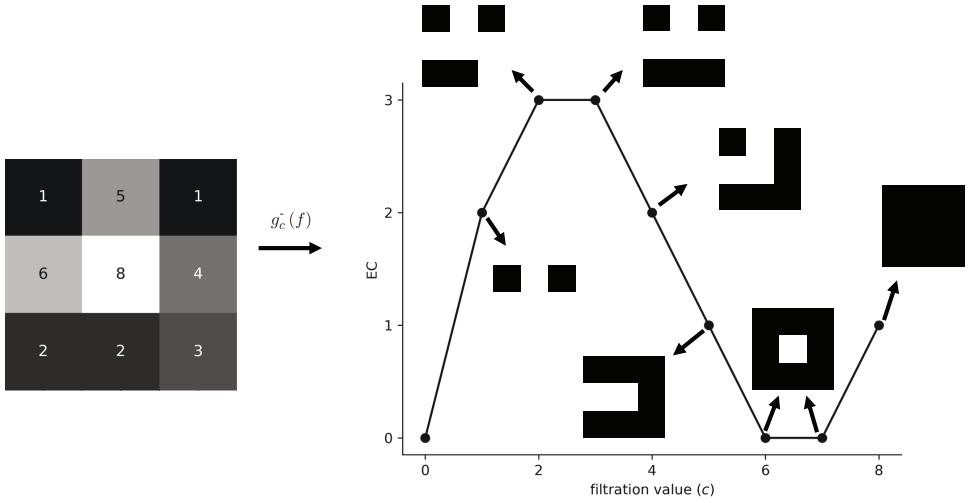


Figure 4: EC curve for a 2D field. Note that, in this example, we see the emergence of a hole in the simplicial complex at a filtration level of $c = 6$.

A fundamental aspect to consider in EC computations is connectivity. There are a couple of types of adjacency in a 2D field: (i) vertex adjacency and (ii) edge adjacency. Vertex adjacency defines that a face that shares a vertex with another face is adjacent. This is often referred to as 8-connectedness (or 8-C) as all 8 of the faces that surround an arbitrary central face are considered connected to the central face, as seen in Figure 5. Edge adjacency is more strict, as a face is adjacent to another face only if an edge is shared. This is often referred to as 4-connectedness (or 4-C) as there are only 4 edge-connected faces from an arbitrary central face (also shown in Figure 5). In the above example, where the EC was computed for Figure 3, 8-C was assumed. Throughout this work, we will be using vertex adjacency for all dimensions: 8-C for 2D field analysis and 26-C for 3D field analysis.

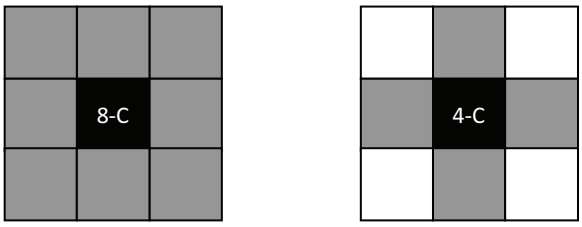


Figure 5: Types of connectivity; on the left is vertex-adjacency (or 8-C) and, on the right, is edge-adjacency (or 4-C).

A naïve approach for computing the EC curve would be to count the vertices, edges, and faces at each filtration level. This would require iterations over every vertex, edge, and face at each level, resulting in poor computational scalability: $\mathcal{O}(whn_c)$ where w and h are the width and height of the field, respectively, and n_c is the number of filtration levels. Another method would be to consider the vertices, edges, and faces from the previous filtration level and only add the new ones to the current filtration; unfortunately, adding a new face does not necessarily add all vertices and edges associated with the face as new components to the level set. This is because the EC follows the inclusion-exclusion principle:

$$\chi(A \cup B) = \chi(A) + \chi(B) - \chi(A \cap B). \tag{2.3}$$

As such, the unique components added to set A by adding set B, or a new face, would be those in B minus the intersection: $A \cap B$. An illustration of this concept is included in Figure 6 where edges and vertices will be double-counted if we do not subtract the intersection of the new face and the existing set.

Figure 6: Adding a single face to the bottom right of a 3x3 field complex requires the subtraction of two edges and 3 vertices which are double counted.

One can avoid the aforementioned problem by tracking vertices and edges which are included in the set. This provides a means to check new vertices and edges against the existing set and only

add novel components. However, this comes at large memory cost, as these additional edge and vertex arrays would require storage of a binary or integer value for each entry, exceeding the element size of the original face data array from the field data itself. A solution to both of these problems is to use vertex neighborhoods to determine impact on EC [7]. Vertices can be used to compute EC contribution by considering a face-sized neighborhood centered on a vertex. A quarter of each face that share the vertex contribute to make up this neighborhood. Subsequently, one-half of each edge that shares the vertex are also included in this vertex-centered neighborhood. An illustration of this concept is shown in Figure 7.

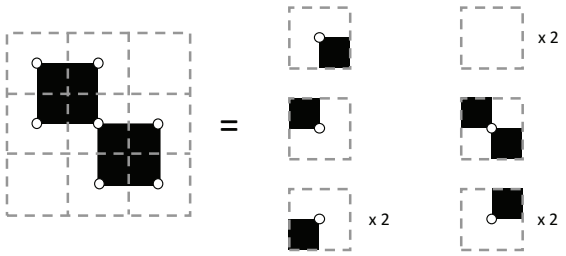


Figure 7: Examining a face-sized neighborhood about each vertex in a 2x2 binary image.

There are 16 positionally-different types of vertex contributions of which 6 are unique subject to symmetry, as shown in Table 1. From these representations we can compute the EC contribution and we can compute the contribution to the perimeter and area of the filtered field. One advantage to this method over strictly keeping track of the EC contribution is that the perimeter cannot be computed from the edges in the set, it can only be computed from exterior edges, which would require the tracking of additional information. The contribution of these 6 types of vertex contributions is given for EC (both 4-C and 8-C), area, and perimeter in Table 1.

Table 1: The 2D vertex contributions to the overall EC for both 4-C and 8-C cases. Area and perimeter contributions (equivalent in both 4-C and 8-C cases) are given as well.

Type	V (8-C)	E (8-C)	F (8-C)	EC (8-C)	V (4-C)	E (4-C)	F (4-C)	EC (4-C)	Perimeter	Area
	0	0	0	0	0	0	0	0	0	0
	1	2 * (0.5)	1 * (0.25)	0.25	1	2 * (0.5)	1 * (0.25)	0.25	1	0.25
	1	3 * (0.5)	2 * (0.25)	0	1	3 * (0.5)	2 * (0.25)	0	1	0.5
	1	4 * (0.5)	2 * (0.25)	-0.5	2	4 * (0.5)	2 * (0.25)	0.5	2	0.5
	1	4 * (0.5)	3 * (0.25)	-0.25	1	4 * (0.5)	3 * (0.25)	-0.25	1	0.75
	1	4 * (0.5)	4 * (0.25)	0	1	4 * (0.5)	4 * (0.25)	0	0	1

Because there is no intersection between the contributions of each vertex, the EC can be computed by the strict addition of contributions from each vertex. In other words, (2.3) loses the $\chi(A \cap B)$ term

because it is zero. By providing a collar of values about the edge of the field, or by determining if a given vertex is a corner, edge, or central vertex, the EC of that field may be computed by iterating over all vertices for each value of the level set. Although this is clearly an improvement over checking each vertex, edge, and face, this method still is computationally expensive in large-scale fields as the order of computation remains at $\mathcal{O}(whn_c)$. However, by tracking only the change in contribution types (i.e., the 6 unique vertex contribution types) for a given level set, time complexity can be reduced from $\mathcal{O}(whn_c)$ to $\mathcal{O}(wh + n_c)$, as shown in Snidaro and Foresti [24].

The method implemented in `CHUNKYEuler` yields the same reduction in time complexity by analyzing change in EC through changes in which top-dimensional cells are active; however, only the changes in EC are tracked. Modifications would need to be made to `CHUNKYEuler` to consider other topological descriptors proposing new challenges when considering indirect metrics such as perimeter. `GUDHI` computes the so-called persistence homology of a cubical complex over filtration values using the compressed annotation matrix method [4] and currently cannot benefit from the reduction in time complexity by looking at changes instead of evaluating each filtration level. The limitations of `GUDHI` (in terms of computational speed) and `CHUNKYEuler` (in terms of ability to compute diverse descriptors) are overcome by vertex contribution algorithms.

With our implementation of the vertex contribution method, an array that includes two entries for each contribution type at each level set value is initialized at zero. Then, for each vertex neighborhood, the integer values of the pixels are used as the index to be incremented. As an example, in Figure 8 we show the central face neighborhood of Figure 3; the first face becomes active at a value of 0. Subsequently, the vertex contribution with one active face is incremented as born at index 0. Since the second face becomes active at a value of 1, the one-face vertex contribution is incremented as dying at index 1, and the two-face diagonal vertex contribution is incremented as being born. This process continues for each face value until the maximum value is reached. At this point, the neighborhood will be filled under the assumption that the maximum value for filtration exceeds the maximum intensity of faces in the field. The only additional step is determining if the first two face values are in a diagonal position or not as the two types of 2-face neighborhoods have different contributions to the topological descriptors. In the example in Figure 8, it can be seen that there will be a diagonal vertex neighborhood as the first two active faces are diagonally adjacent. Importantly, the output of this method is not just the EC curve, but rather an array of the quantity of vertex contributions of each type that are present at each level set. This means that one can compute EC, perimeter, area, and any other descriptors/metrics that can utilize these vertex contributions as input for each filtration value. The process of gathering vertex contributions of a 2D field considering filtration values using this method is summarized in Algorithm 1. In the following sections, we propose algorithms for computing vertex contributions and extend the method to 3D fields.

2.1 Parallel Implementation

Parallel implementation follows the same pattern as the serial implementation but requires: either (i) a vertex contribution array for each thread, or (ii) a lock-based structure for a global vertex contribution array. The length of the 2D vertex contribution array would be one more than the number

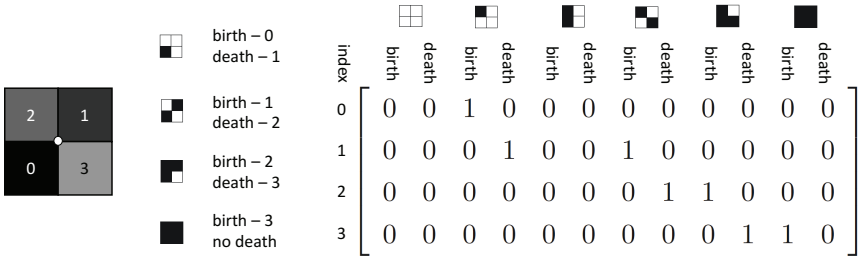


Figure 8: Visualization of the contribution of a single vertex neighborhood over its face values.

Algorithm 1 Bitmap contribution (2D, serial)

Field $\leftarrow$ read(raw data file)
$w \leftarrow$ field width
$h \leftarrow$ field height
$M \leftarrow$ largest filtration value
$q_{k,\text{in}}, q_{k,\text{out}} \leftarrow$ Zeros of length M+2 $\forall k \in \{0, 1, ..., 9\}$
for all $i < h + 1$ **do**
  for all $j < w + 1$ **do**
    data $\leftarrow [M + 1, M + 1, M + 1, M + 1]$
    if $i > 0 \wedge j > 0$ **then**
      data[0] $\leftarrow$ Field$(i - 1, j - 1)$
    else if $i > 0 \wedge j < w$ **then**
      data[1] $\leftarrow$ Field$(i - 1, j)$
    end if
    if $i < h \wedge j > 0$ **then**
      data[2] $\leftarrow$ Field$(i, j - 1)$
    else if $i < h \wedge j < w$ **then**
      data[3] $\leftarrow$ Field(i, j)
    end if
    pos $\leftarrow$ argsort(data)
    adjacency $\leftarrow$ distance(pos[0], pos[1])
    $q_{1,\text{in}}$[data[0]]++, $q_{1,\text{out}}$[data[1]]++
    if adjacency == 1 **then**
      $q_{2,\text{in}}$[data[1]]++, $q_{2,\text{out}}$[data[2]]++
    else
      $q_{d,\text{in}}$[data[1]]++, $q_{d,\text{out}}$[data[2]]++
    end if
    $q_{3,\text{in}}$[data[2]]++, $q_{3,\text{out}}$[data[3]]++
    $q_{4,\text{in}}$[data[3]]++
  end for
end for

of levels, and the width would be 10. Although there are 6 vertex contribution types which would result in an array width of 12, the empty type contributes nothing to perimeter, area, and EC (see Table 1); in addition, the number of empty contributions at a given level can be back-calculated using the total number of vertices. In addition, when considering lock-based methods, at the entry value, every vertex will increment the empty contribution at the initial filtration level, causing a pile-up that could exceed serial computation time. A standard 8-bit image with pixel values ranging from 0-255 would require 2570 entries in a vertex contribution matrix, requiring 10280 bytes of storage per array instance using 32-bit integers. In most CPU systems, holding these arrays in memory is not expensive and can be done easily for most modern systems, even with one storage array per core for highly distributed systems. The size of the image in memory will greatly exceed that of vertex contributions in most cases. For perspective, if we consider a standard high-definition (HD) image size of 1280x720, the storage for only one color channel using arrays of unsigned 8-bit integers would require 921600 bytes of memory, which exceeds the size of approximately 90 vertex contribution arrays. If a vertex contribution array can be afforded for each thread, Algorithm 2 can and should be used. The lock-based method for these algorithms will not be described in detail in this work, however in highly distributed systems, such as GPUs, a global contribution array with locks should be used where local or register memory does not exceed the size of a thread vertex contribution array plus dynamic memory required to compute vertex contributions. Instead of increasing the global array values directly, the algorithm would use a locking system for each increment operation to prevent a data race. A light-weight locking mechanism would be best for these purposes, such as a spin-lock, which would reduce overhead of waking up a thread as in a mutex-based locking system.

2.2 3D Field Processing

For processing 3D fields we need to define vertex contributions in 3D. To the best of our knowledge, there are no extensions of the contribution methods of Snidaro and Foresti to 3D fields reported in the literature. Following the definition in [27], there are 22 unique types of vertex contribution subject to symmetry in 3D, binary fields. All 22 unique contributors are described in Table 2. Similarly, the empty type contributes nothing to EC, perimeter, area, or volume, so is only back-calculated in post if desired. This means that with 21 unique contribution types without the empty type, there are a total of 42 array entries per level. Therefore with an 8-bit 3D data file, a storage size of 43176 bytes per vertex contribution array would be required. Again, the size is not overwhelmingly large and therefore a 3D extension of Algorithm 2 is sufficient. It should be noted that many 3D fields, especially those from computer tomography (CT) scans, can use much higher resolution (e.g., 14-bit or 16-bit integers). The size of a 512x512x512 scan with 16-bit data requires almost 24 times the memory of the vertex contribution array.

A major difference between the 2D and 3D case is that diagonal adjacency of cells in the vertex neighborhood must be determined to differentiate between contributor types in the 3D case whereas only one adjacency needs to be calculated for the 2D case. Using the sum of Euclidean distance between all pairs of points, one can completely differentiate contribution types and use a running

Algorithm 2 Bitmap contribution (2D, parallel)

Field $\leftarrow$ read(Field file)
$M \leftarrow$ largest filtration value
$n_{\text{threads}} \leftarrow$ number of threads
$q_{tk,\text{in}}, q_{tk,\text{out}} \leftarrow$ Zeros of length M+2 $\forall k \in \{0, 1, ..., 9\}, \forall t \in \{1, 2, ..., n_{\text{threads}}\}$
jobs$_t \leftarrow$ total jobs / n_{threads}
for all $t < n_{\text{threads}}$ **do**
  for all $(i, j) \in$ jobs$_t$ **do**
    data $\leftarrow [M+1, M+1, M+1, M+1]$
    if $i > 0 \wedge j > 0$ **then**
      data[0] $\leftarrow$ Field$(i-1, j-1)$
    else if $i > 0 \wedge j < w$ **then**
      data[1] $\leftarrow$ Field$(i-1, j)$
    end if
    if $i < h \wedge j > 0$ **then**
      data[2] $\leftarrow$ Field$(i, j-1)$
    else if $i < h \wedge j < w$ **then**
      data[3] $\leftarrow$ Field(i, j)
    end if
    pos $\leftarrow$ argsort(data)
    adjacency $\leftarrow$ distance(pos[0], pos[1])
    $q_{t1,\text{in}}$[data[0]]++, $q_{t1,\text{out}}$[data[1]]++
    if adjacency == 1 **then**
      $q_{t2,\text{in}}$[data[1]]++, $q_{t2,\text{out}}$[data[2]]++
    else
      $q_{td,\text{in}}$[data[1]]++, $q_{td,\text{out}}$[data[2]]++
    end if
    $q_{t3,\text{in}}$[data[2]]++, $q_{t3,\text{out}}$[data[3]]++
    $q_{t4,\text{in}}$[data[3]]++
  end for
end for
join threads
$q_{k,\text{in}} \leftarrow \sum_{t=0}^{n_{\text{threads}}} q_{tk,\text{in}} \forall k \in \{0, 1, ..., 9\}$
$q_{k,\text{out}} \leftarrow \sum_{t=0}^{n_{\text{threads}}} q_{tk,\text{out}} \forall k \in \{0, 1, ..., 9\}$

(in parallel)

Table 2: 3D vertex contributions to EC for the 26-C case along with perimeter, area, and volume contributions.

Type	f_{adj}	V (26-C)	E (26-C)	F (26-C)	Vox (26-C)	EC (26-C)	Perimeter	Area	Volume
	0	0	0	0	0	0	0	0	0
	0	1	3 * (0.5)	3 * (0.25)	1 * (0.125)	0.125	1.5	0.75	0.125
	1	1	4 * (0.5)	5 * (0.25)	2 * (0.125)	0	1	1	0.25
	1.414	1	5 * (0.5)	6 * (0.25)	2 * (0.125)	-0.25	3	1.5	0.25
	1. 732	1	6 * (0.5)	6 * (0.25)	2 * (0.125)	-0.75	3	1.5	0.25
	3.414	1	5 * (0.5)	7 * (0.25)	3 * (0.125)	-0.125	1.5	1.25	0.375
	4.146	1	6 * (0.5)	8 * (0.25)	3 * (0.125)	-0.375	2.5	1.75	0.375
	4.243	1	6 * (0.5)	9 * (0.25)	3 * (0.125)	-0.125	4.5	2.25	0.375
	6.828	1	5 * (0.5)	8 * (0.25)	4 * (0.125)	0	0	1	0.5
	7.243	1	6 * (0.5)	9 * (0.25)	4 * (0.125)	-0.25	3	1.75	0.5
	7.560	1	6 * (0.5)	9 * (0.25)	4 * (0.125)	-0.25	2	1.5	0.5
	7.975	1	6 * (0.5)	10 * (0.25)	4 * (0.125)	0	3	2	0.5
	8.293	1	6 * (0.5)	10 * (0.25)	4 * (0.125)	0	2	2	0.5
	8.485	1	6 * (0.5)	12 * (0.25)	4 * (0.125)	0.5	6	3	0.5
	3.414†	1	6 * (0.5)	10 * (0.25)	5 * (0.125)	-0.125	1.5	1.25	0.625
	4.146†	1	6 * (0.5)	11 * (0.25)	5 * (0.125)	0.125	2.5	1.75	0.625
	4.243†	1	6 * (0.5)	12 * (0.25)	5 * (0.125)	0.375	4.5	2.25	0.625
	1†	1	6 * (0.5)	11 * (0.25)	6 * (0.125)	0	1	1	0.75
	1.414†	1	6 * (0.5)	12 * (0.25)	6 * (0.125)	0.25	3	1.5	0.75
	1.732†	1	6 * (0.5)	12 * (0.25)	6 * (0.125)	0.25	3	1.5	0.75
	0†	1	6 * (0.5)	12 * (0.25)	7 * (0.125)	0.125	1.5	0.75	0.875
	0†	1	6 * (0.5)	12 * (0.25)	8 * (0.125)	0	0	0	1

† Adjacency of empty voxels used to avoid more computation

sum to avoid computing the same number twice. We call this function f_{adj} which is defined as:

$$f_{\text{adj}}(\mathcal{V}) = \sum_{i=1}^{|\mathcal{V}|-1} \sum_{j=i+1}^{|\mathcal{V}|} ||V_i - V_j||_2 . \tag{2.4}$$

Here, $\mathcal{V}$ represents the set of vertex coordinates V_i. The 2-norm is taken to get the Euclidean distance between all coordinate pairs of the vertices, allowing for distinction between which type of contribution is present. For instance, with 2 active cells, they may share only one face ($f_{\text{adj}} = 1$), share only one edge ($f_{\text{adj}} = \sqrt{2}$), or share only one vertex ($f_{\text{adj}} = \sqrt{3}$). Adjacency for all configurations of vertex neighborhoods in 3D are included in Table 2. Also, for the case of 5, 6, and 7 cells active, the position of the inactive cells, or empty cells, is used to reduce the number of adjacencies computed. For the sake of brevity, the serial version for computing 3D vertex contributions is shown in the appendix in Algorithm 3. When considering memory requirements, the implementation of a low-memory algorithm for 3D field processing is all the more important. For this case, the 3D algorithm can be extended exactly the same way as the 2D case, by replacing the calls to the 8-cell values in the field file in memory with calls to read the 8 cell-specific indices from the field file which is described in more detail in the following section.

2.3 Low-Memory Processing

Reading a data file for processing often requires that the entire field is held in memory during execution. To mitigate this, the same method that is utilized for the serial and parallel cases can be implemented by forgoing reading the whole field into memory, and instead reading only the values needed at a given vertex of the field from the file directly. To compute a vertex contribution in 2D, only four values are required to be read into memory at a given time. Exploiting the file structure in the BMP file format (for bitmap images), we utilize a low memory version of Algorithm 1 (lines 9 through 18) and Algorithm 2 (lines 9 through 18) by replacing calls to the Field data (i.e., data$[0] \leftarrow$ Field$(i-1, j-1)$) with calls to the raw data file (i.e., data$[0] \leftarrow$ read(Field face$(i-1, j-1)$)).

3 Computational Results

The proposed algorithms were implemented in C++ and invoked from `Python`. For the low-memory cases, C++ was also used to read in the partial field data. A comparison between the `GUDHI` software package [16] invoked from `Python` and the algorithms for 2D and 3D field analysis shown above was performed and reported in the following sections. The EC was first computed on synthetic random fields of standard sizes for 2D and 3D fields. We then analyze data arising in real applications: microscopy (2D), molecular dynamics simulations (3D), and hyperspectral images (3D). For all case studies, data was translated from raw format (i.e., float or integer) to integer format using values from 0 to 255. For each case, benefit of parallelization was measured by computing the EC with up to 24 CPU cores. All timing results were obtained on a 24-core (Intel Xeon E5-2697—2.7 GHz) computing server with 256 GB of RAM running Red Hat Enterprise release 6.10. All data

and software needed for reproducing the results are shared as open-source code and are available at https://github.com/zavalab/ML/tree/master/FastTopology.

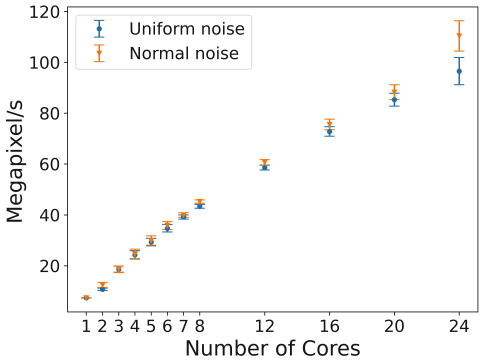
3.1 Analysis of Synthetic Random Fields

We generated synthetic 2D random fields in order to test the scalability of the proposed algorithms in a systematic manner. Standard 2D image sizes of 1280x720 and 1920x1080 were tested along with a couple of large square fields of dimension 2048x2048 and 4096x4096. Fields at each size were generated using 500 random samples of both uniform and random noise to show run-to-run timing variation. A summary of the results is provided in Table 3. We can see that the processing time for computing the EC using parallel Algorithm 2 is nearly 3 orders of magnitude faster than `GUDHI`, with serial operation already being 50 times faster. The speedup obtained using more cores can be seen in Figure 9 for a field of size 2048x2048. On the left we show that the total Megapixels (MP) processed per second exceeds 100 when using 24 cores and on the right, the parallel speedup efficiency is 62%. Processing random fields of similar size using 8 cores with `CHUNKYEuler` results in a processing speed of about 46.5 MP/s as reported in [30], compared to 43.3 - 45.2 MP/s using Algorithm 2 with 8 cores. As such, our implementation is as scalable as that of `CHUNKYEuler`.

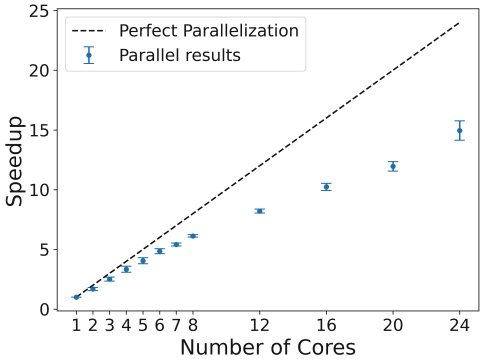
Table 3: Timing results in seconds and million pixels (faces) processed per second for random 2D fields.

Type	GUDHI (s)	GUDHI (MP/s)	Alg. 2, 24 cores (s)	Alg. 2, 24 cores (MP/s)
1280x720 U	5.31 ± 0.16	0.174 ± 0.005	0.0095 ± 0.0006	97.7 ± 4.7
1280x720 N	5.11 ± 0.16	0.181 ± 0.005	0.0093 ± 0.0006	99.3 ± 4.9
1920x1080 U	14.99 ± 0.44	0.138 ± 0.003	0.0193 ± 0.0010	107.4 ± 4.1
1920x1080 N	14.57 ± 0.44	0.142 ± 0.004	0.0188 ± 0.0011	110.8 ± 4.6
2048^2 U	32.50 ± 1.24	0.129 ± 0.004	0.0436 ± 0.0031	96.5 ± 5.3
2048^2 N	31.35 ± 1.07	0.133 ± 0.004	0.0381 ± 0.0029	110.4 ± 6.0
4096^2 U	151.1 ± 4.8	0.111 ± 0.003	0.207 ± 0.024	82.2 ± 9.6
4096^2 N	145.2 ± 5.0	0.116 ± 0.003	0.210 ± 0.032	81.95 ± 13.2

The 3D algorithm was also tested on both uniform and normal noise for fields of dimensions 128x128x128 and 256x256x256. Given the larger size of the data and processing time required, only 100 samples were run for each field size with both uniform and random noise. The processing results are summarized in Table 4; a similar trend is observed, as `GUDHI` takes nearly 3 orders of magnitude longer than Algorithm 2 using 24 cores. With fields of size 256x256x256, the parallel implementation with 24 cores speeds up execution by about 19 times, leading to an efficiency of 75-80% (Figure 10). For reference, `CHUNKYEuler` processes voxels/cells at a speed of 26.6 MV/s on 3D random fields with similar size [30], compared to 10.4 MV/s with Algorithm 2 for 3D, both using 8 CPU cores. The reduction in speed by our implementation is likely due to the higher number of comparisons and computation required to determine adjacency for vertex neighborhood classification. However, this reduction in speed comes with the benefit of more flexibility of enabling computation of alternative



(a) Processing speed



(b) Speedup from parallelization

Figure 9: Timing results for 2D fields of size 2048x2048. On the left, the processing speed of faces for Algorithm 2 with 2-24 cores. On the right, the speedup due to parallelization of Algorithm 2 with 2-24 cores compared to a perfect parallel efficiency line (dashed).

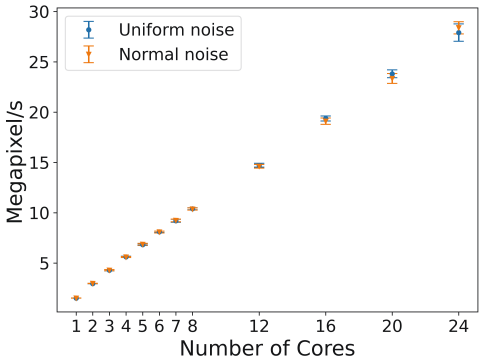
topological descriptors.

Table 4: Timing results in seconds and million voxels/cells processed per second for random 3D fields.

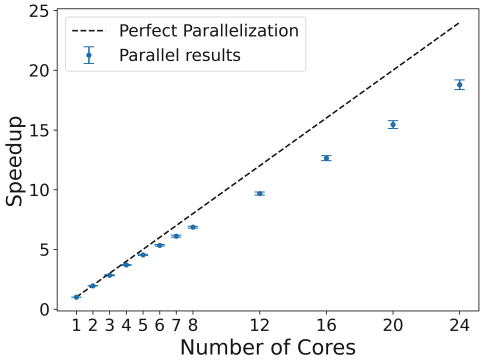
Type	GUDHI (s)	GUDHI (MV/s)	Alg. 2 (3D), 24 cores (s)	Alg. 2 (3D), 24 cores (MV/s)
128^3 U	42.61 ± 1.19	0.0493 ± 0.0011	0.0795 ± 0.0067	26.67 ± 1.67
128^3 N	42.13 ± 1.46	0.0498 ± 0.0015	0.0765 ± 0.0045	27.47 ± 1.21
256^3 U	472.3 ± 15.5	0.0356 ± 0.0011	0.602 ± 0.020	27.90 ± 0.87
256^3 N	470.7 ± 12.2	0.0357 ± 0.0009	0.591 ± 0.013	28.38 ± 0.61

3.2 Memory analysis

The low-memory version of Algorithms 1 and 2 were also tested on random 2D fields. First, random matrices of uniform and normal distribution were generated for fields of size 1920x1080. Then, the fields were saved as bitmap image files (an easily readable, raw data format). Finally, the C++ algorithm directly read and computed the EC for each field. 500 random 2D fields were generated with statistics on the processing speed reported in Table 5. There is a drastic processing speed loss from reading individual pixel values to populate only the 4 values from each vertex neighborhood during computation of the EC curve. However, even the serial methods are still faster than `GUDHI` (see Table 3) and now require the minimal memory to compute the EC curve. Also, data beyond the size of working memory can be analyzed given that this is a streaming approach, similar to that used in `CHUNKYEuler` [9]. In addition, only 2D fields were explored here and memory management issues are even more important in the 3D case. Similar slowdown should be expected when, if not slightly more given the increase in values analyzed from 4 faces/pixels to 8 cells/voxels.



(a) Processing speed



(b) Speedup from parallelization

Figure 10: Timing results for 3D fields of size 256x256x256. On the left, the processing speed of voxels for Algorithm 2 with 3D modifications with 2-24 cores. On the right, the speedup due to parallelization of Algorithm 2 with 3D modifications with 2-24 cores compared to a perfect parallel efficiency line (dashed).

Table 5: Timing results in million pixels processed per second for the low memory processing algorithm.

Type	Alg. 1 (MP/s)	Alg. 2, 12 cores (MP/s)	Alg. 2, 24 cores (MP/s)
1920x1080 U	0.346 ± 0.005	3.040 ± 0.045	4.285 ± 0.036
1920x1080 N	0.347 ± 0.005	2.991 ± 0.029	4.159 ± 0.047

3.3 Microscopy Case Study

Liquid crystal sensors elicit optical responses in the presence of target compounds or contaminants. The concentration and environment in which the target compound binds to the liquid has significant impact on the topological state of the system. In this case study, data presented in Jiang et al. [10] was used to benchmark real data and explore the potential of real-time analysis. The system we are showcasing here is analysis on a gas-based liquid crystal detection system with varying sulfur dioxide (SO_2) concentrations, from 0.5 ppm to 5 ppm, in an environment with 40% relative humidity. An example of how filtration values impact the topology of the system is shown in Figure 11.

As shown in Figure 12, the optical responses at differing SO_2 concentrations have distinct EC curves. On a liquid crystal micrograph plate, anywhere from 1 to 36 fully readable grid squares may be used to predict the state of the system. Therefore, the speed in which the images are read dictate how close to real time the sensor can be monitored.

In Table 6, with image sizes that are so small (approximately 134x134 for each grid-square), parallelization does not receive the same benefit as the larger field data as explored above. There is even a performance decrease by using more than 12 cores for parallel computation; however, the data is processed about 20-30 times faster than `GUDHI`.

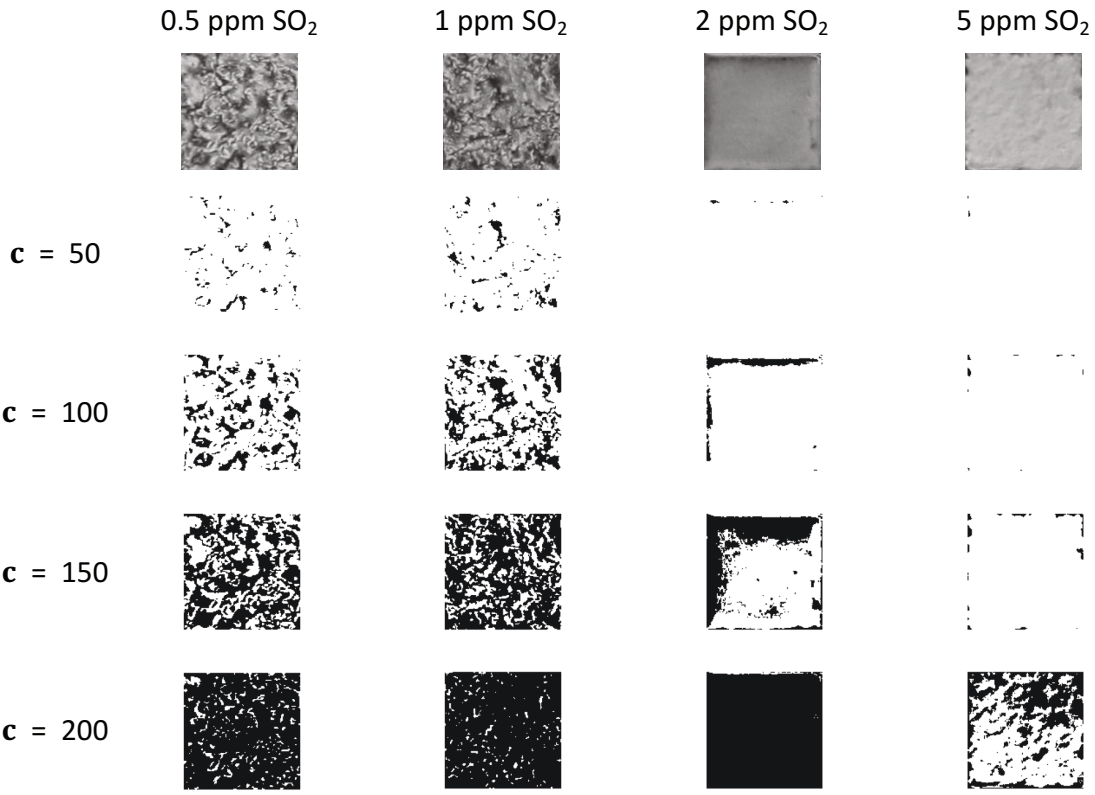
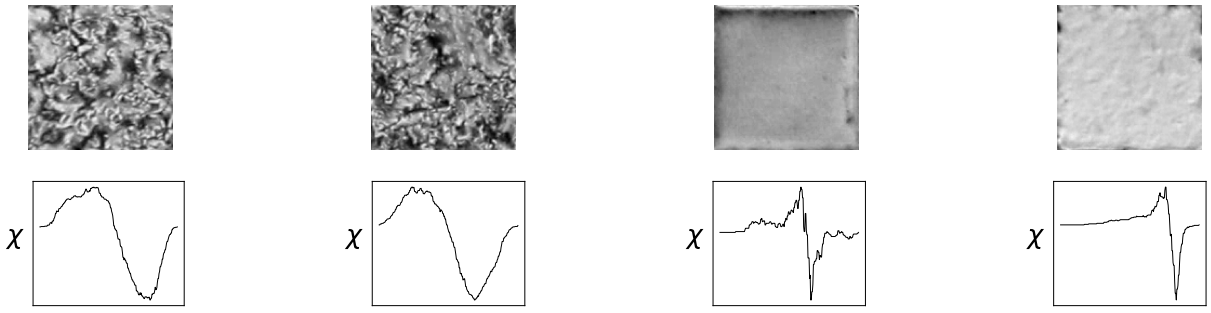


Figure 11: Filtration process for various liquid crystal micrographs. For some concentrations of SO_2, no activity occurs until over a filtration value of 150 while other concentrations show activity at low filtration values.



(a) 0.5 ppm SO_2 (b) 1 ppm SO_2 (c) 2 ppm SO_2 (d) 5 ppm SO_2

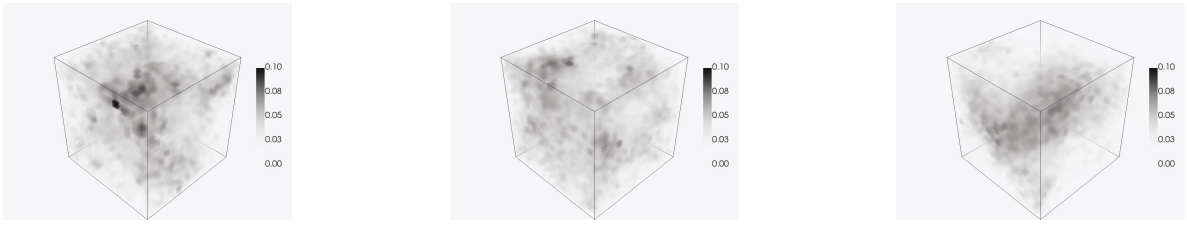
Figure 12: Varying concentration of SO_2 in the system changes both the optical response and the topological description of the image through the EC. The optical response and subsequent EC curve are shown for 0.5 ppm, 1 ppm, 2 ppm, and 5 ppm SO_2 systems in 40% relative humidity.

Table 6: Timing results in million pixels processed per second for the liquid crystal sensor case study.

Type	GUDHI (MP/s)	Alg. 1 (MP/s)	Alg. 2, 12 cores (MP/s)	Alg. 2, 24 cores (MP/s)
SO_2 Images	0.332 ± 0.009	6.268 ± 0.171	11.33 ± 0.49	9.449 ± 0.225
Random Fields	0.271 ± 0.009	6.127 ± 0.132	11.20 ± 0.39	9.600 ± 0.223

3.4 Molecular Dynamics Data

Simulations that result in 3D fields are also relevant to the algorithms provided in this work. One such example is molecular dynamics simulations, where chemical systems can be simulated to understand molecular-scale interactions between participating species. Here, we analyze a system shown by Chew and co-workers [5], and later with the EC by Smith and co-workers [21]. The simulations capture molecular interactions of biomass reactants in cosolvent/water mixtures. As an example, Figure 13 shows the density of water molecules in a 20x20x20 grid for fructose in three different cosolvent/water systems with 10 weight % water. The topology of the solvent environment has been shown to correlate strongly with reactivity. The process of filtering one of these fields is shown in Figure 14.



(a) dioxane (b) γ-valerolactone (c) tetrahydrofuran

Figure 13: Water densities from simulations of fructose with various cosolvent species listed.

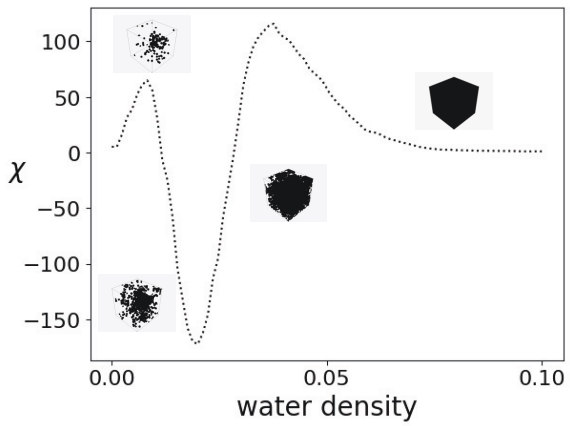


Figure 14: EC curve evolution with 3D binary fields at various filtration values.

Table 7: Timing results in million cells processed per second for the molecular dynamics case study.

Type	GUDHI (MC/s)	Alg. 1 (3D) (MC/s)	Alg. 2 (3D), 12 cores (MC/s)	Alg. 2 (3D), 12 cores (MC/s)
MD Fields	0.108 ± 0.004	1.065 ± 0.025	2.105 ± 0.111	1.853 ± 0.045

The small size of the data set shows much less scaling advantage for the parallel case, as shown in Table 7. Although the timing advantage is less, the proposed algorithm is still one order of magnitude faster than `GUDHI`. To process the entire data set, `GUDHI` took 56.6 seconds, while the serial algorithm took 5.7 seconds, and 12-core parallel implementation took 2.9 seconds. The EC curve of each solvent system is shown in Figure 15.

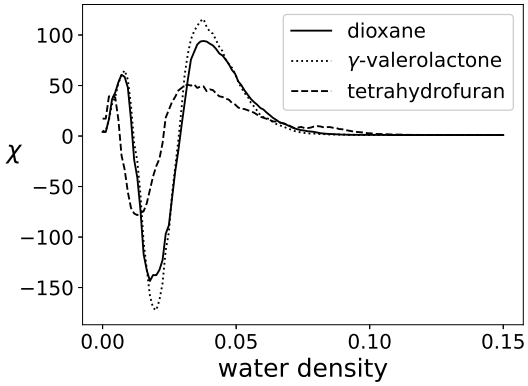


Figure 15: EC Curves for each cosolvent system.

3.5 Hyperspectral Image Data

Another example relevant to the chemical sensing community is hyperspectral image analysis. In this research domain, each pixel of a 2D image has a spectral dimension, meaning each pixel of the image is associated with a unique spectra. This requires each image to be analyzed in 3D to take full advantage of the information contained in the spectral dimension. One can view each spectral wavelength as a 2D image, as shown in Figure 16.

A sample data set on the ripeness of fruits [28] is used to benchmark the performance. Hyperspectral images were taken of both the front and back of the fruit each day. Fruits were removed from the set when considered overripe. A total of 360 hyperspectral images make up the kiwifruit data set. Some sample images translated from visible spectra to RGB are shown in Figure 17, where it is clear that using only RGB, it is difficult to tell which kiwifruit is ripe, overripe, or under ripe.

Statistics on the processing speed are shown in Table 8. The processing time for these images was removed as a column of the table due to the varying size of the images and instead strictly analyzed processing speed in MV/s, similar to the liquid crystal case study. For comparison, the EC curve from

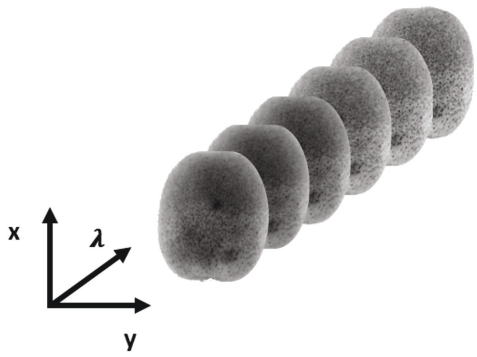


Figure 16: Visualization of hyperspectral image data. Here, a near infrared image with 6 selected wavelengths (λ) is illustrated. In totality, the hyperspectral image has 252 unique wavelengths.

(a) Under ripe (b) Ripe (c) Overripe

Figure 17: RGB equivalent images of kiwifruits presenting with varying ripeness levels. RGB alone is difficult to discern whether a kiwifruit is ripe or not without feeling the firmness of the kiwifruit.

filtering the hyperspectral images of ripe, overripe, and under ripe kiwifruits are shown in Figure 18.

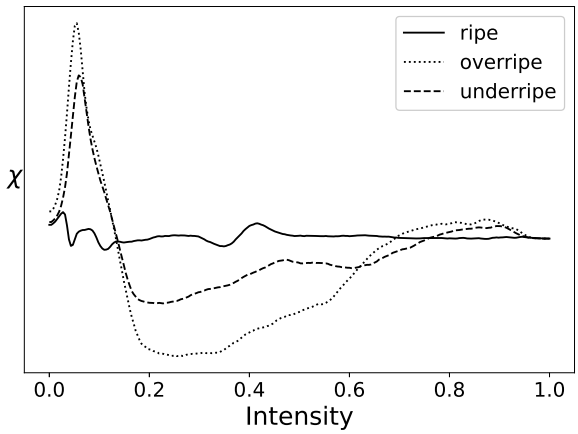


Figure 18: EC Curves for each stage of kiwifruit ripeness.

Table 8: Timing results in million voxels processed per second for the hyperspectral imaging case study.

Type	GUDHI (MV/s)	Alg. 1 (3D) (MV/s)	Alg. 2 (3D), 24 cores (MV/s)
Kiwifruit Images	0.0701 ± 0.0078	1.973 ± 0.112	34.57 ± 2.38
Random Fields	0.0427 ± 0.0031	1.506 ± 0.006	28.19 ± 1.02

4 Discussion

Using Algorithms 1 and 2 in both 2D and 3D field data, we find that there is a clear scaling advantage in both time and memory usage while computing EC when compared with off-the-shelf computational tools such as `GUDHI`. Also, when compared to state-of-the-art parallel implementations, such as `CHUNKYEuler`, there is comparable speed. The time complexity of `CHUNKYEuler` and the algorithms presented in this work are nearly identical, as both methods stem from the same principle that using change in features is more computationally scalable than counting features at each filtration value.

For the microscopy case study for liquid crystals, the average size of images in that data set was 134x134, or 17956 pixels, meaning that it would require a processing speed of 19 MP/s to fully process a liquid crystal sensor with 36 readable grid-squares at a video frame rate of 30 frames per second (FPS). Considering control applications, the serial version of the tool is capable of processing 11 grid-squares in real-time considering a 30 FPS video. However, some control applications will not require the reading of all 36 grid-squares, or reading data as quick as 30 measurements per second. Also, when implementing a complete control scheme, the time required to read and crop the raw

video data to these processable grid-squares must be considered, introducing a time delay between measurement and data reconciliation.

Also, the potential limitations of parallelization are noticed when considering the microscopy case. For larger random fields for benchmarking, more cores yielded significantly faster performance, whereas in the liquid crystal case study, more cores did not yield favorable speedup. This is seen with a peak efficiency at about 12 cores before dropping as more cores were used (Figure 19). This analysis does not consider simply using Algorithm 1 for each individual image, but running the dispatch of the image analysis in parallel (which may increase throughput). This highlights the importance to design a computational framework that accurately addresses data analysis and data structure from case to case. This is especially relevant in the case of designing lightweight, standalone sensors with limited computational capabilities.

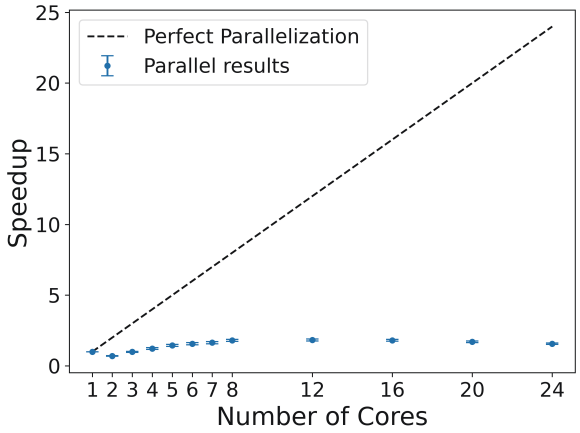


Figure 19: Speedup curve through parallelization for processing liquid crystal images. Perfect speedup line shown for reference.

The molecular dynamics study shows similar trends as the microscopy study, indicating the data was small enough that parallelization did not realize the same benefit as random 3D fields of larger size. However, this framework is capable a handling molecular dynamics simulations which may include thousands or millions of molecules. This becomes especially relevant with the more recent use of parallel computation using GPUs to accelerate computations in density functional theory[29, 14, 1] where more detailed or holistic simulation results can be analyzed quickly. The granularity of the grid (20x20x20) could be expanded to have more fine-grained analysis on larger systems and see significant speed-up similar to that found in the random 3D fields analyzed earlier. Also, any 3D field data that can be translated to a cubical lattice may be analyzed in the same way. For instance, computational fluid dynamics data with millions of finite elements could be analyzed for trends in EC which could transform large-scale, fine-grained data into a feasible size for input into a prediction algorithm without losing important topological information. For reference, a couple of computational fluid dynamics simulations were analyzed to demonstrate scalability: first, a 302x302x302 element simulation of heptane gas undergoing combustion (The University of Utah Center for the Simulation of Accidental Fires and Explosions), took 405 seconds for `GUDHI`, and 3.5 seconds for Algorithm 3 in

parallel with 4 cores; and second, simulation of duct flow [3] of size 193x194x1000 took GUDHI 673 seconds, whereas Algorithm 3 in parallel with 4 cores took 5.5 seconds. Clearly, scaling advantages exist in these data-rich applications.

For the hyperspectral image case study, the size of the images indicated that parallelizing the identification of vertex contributions was economical. The average size of images in this data set was 6.88 MV, meaning it takes about 100 seconds to process a single hyperspectral image with GUDHI, 3 seconds in serial with Algorithm 1 in 3D mode, and only about 0.2 seconds when using 2 in 3D mode with 24 cores. Applications where sensors are developed with a hyperspectral camera may require processing speeds that allow data analysis at speeds faster than those presented in this work. However, in certain applications, such as monitoring pharmaceutical powder composition, standard sampling times for single-sample analysis (only spectral dimension; no 2D image component) with near infrared sensors can be on the order of 5 - 15 seconds [26]. On the other hand, commercially available hyperspectral cameras can achieve frame rates that could facilitate real-time data collection and analysis [25]. Once again, understanding the needs of the system and the dynamic response to control actions should be considered case-to-case when implementing topology-informed control schemes to hyperspectral data.

An important distinction with respect to hyperspectral imaging is that we used the raw data for the case study. Performing dimensionality reduction in hyperspectral image is not uncommon, for instance, using principal component analysis to reduce the number of wavelengths used in the spectral dimension [20]. Reducing the number of wavelengths through dimensionality reduction or selecting regions that are known to contain the most important topological information could reduce computation time enough to process these 3D hyperspectral images in real time as well.

Also, the output of this tool is not just the EC curve, it is the vertex contribution map over the filtration values. Thus, by simply changing the weights each vertex contributes to an overall topological descriptor, the same vertex contribution map can be used to compute any number of topological descriptors as a function of vertex contributions. Also, connectivity need not be assumed in the beginning, as the connectivity only impacts the weights, not the vertex types. This allows for analyses to identify that alternate definitions of connectivity may be more suitable for a given physical system.

With this being said, sensitivity of the system to connectivity can be explored alongside sensitivity of topological descriptors to image resolution, number of filtration values, or image preprocessing techniques. It is commonplace to blur images or treat images with another convolutional operator while utilizing neural networks in machine learning. However, in this work, we use the raw data for both applications. Since the tool is scalable and allows for large-scale analysis, performing studies to understand which combination of resolution, number of filtrations, preprocessing technique, and connectivity type may lead to more topologically-based physical intuition of chemical systems discerned through field/image analysis.

5 Conclusions

In this work, we presented algorithms addressing the scalable computation of vertex contributions in field data. Ultimately, the tool was tested using EC, but the analysis that can be done with vertex contribution maps is not limited just to EC. The speedup was significant, 2 to 3 orders of magnitude, when compared with a common package used for topological persistence in Python (`GUDHI`). Also, recent advancement of the `CHUNKYEuler` software show similar timing capabilities, indicating that methods that address changes in topological contributions are much more scalable than counterparts that compute topological characteristics at each level of a filtration set.

Future research directions could address the usage of vertex contribution maps for more complex topological descriptors, such as the fractal dimension. Also, generalizing these methods to larger dimensions (4D data, such as hyperspectral imaging with a temporal dimension) requires the set of unique vertex contributors for a given dimension. The number of unique contributors and subsequent impact on scalability in terms of computational time and storage for vertex contribution methods in higher dimensions is an open question currently being explored by the authors. In terms of low memory methods, `CHUNKYEuler` uses chunks of each image, whereas the low memory version of Algorithms 1 and 2 reads only the minimum data required to compute the EC curve. Exploration into the optimal decomposition scheme for a cubical field/image could be an interesting direction for research, as having minimum data representations is good for memory scaling, but results in a slow down of just over an order of magnitude.

Also, generalizing vertex contributions to regular, non-cubical lattices, or even more generally to Voronoi cells, with sets of vertex contributions corresponding to the degree of that vertex could be impactful for data that is not the classical cubical layout that physical fields/images possess. Ultimately, if an efficient method exists to compute vertex contribution maps of these more abstract structures, one could calculate a plethora of topological and general system descriptors to control or characterize complex and more abstract physical systems in real time.

6 Acknowledgements

This research was supported by the U.S. National Science Foundation through the University of Wisconsin Materials Research Science and Engineering Center (DMR-2309000) and via grant IIS-1837812.

References

[1] S. I. Allec, Y. Sun, J. Sun, C.-e. A. Chang, and B. M. Wong. Heterogeneous cpu+gpu-enabled simulations for dftb molecular dynamics of large chemical and biological systems. *Journal of Chemical Theory and Computation*, 15(5):2807–2815, 2019. PMID: 30916958.

[2] E. J. Amézquita, M. Y. Quigley, T. Ophelders, J. B. Landis, D. Koenig, E. Munch, and D. H. Chitwood. Measuring hidden phenotype: quantifying the shape of barley seeds using the euler characteristic transform. *in silico Plants*, 4(1):1–15, 2021.

[3] M. Atzori, R. Vinuesa, A. Lozano-Durán, and P. Schlatter. Intense reynolds-stress events in turbulent ducts. *International Journal of Heat and Fluid Flow*, 89:108802, 2021.

[4] J. Boissonnat, T. K. Dey, and C. Maria. The compressed annotation matrix: an efficient data structure for computing persistent cohomology. *CoRR*, abs/1304.6813, 2013.

[5] A. K. Chew, S. Jiang, W. Zhang, V. M. Zavala, and R. C. Van Lehn. Fast predictions of liquid-phase acid-catalyzed reaction rates using molecular dynamics simulations and convolutional neural networks. *Chemical science*, 11(46):12464–12476, 2020.

[6] M. K. Chung, H. Lee, A. DiChristofano, H. Ombao, and V. Solo. Exact topological inference of the resting-state brain networks in twins. *Network Neuroscience*, 3:674–694, 2019.

[7] S. B. Gray. Local properties of binary images in two dimensions. *IEEE Transactions on Computers*, C-20(5):551–561, 1971.

[8] L. Hakim, M. S. Kavitha, N. Yudistira, and K. Takio. Regularizer based on euler characteristic for retinal blood vessel segmentation. *Pattern Recognition Letters*, 149:83–90, 2021.

[9] T. Heiss and H. Wagner. Streaming algorithm for euler characteristic curves of multidimensional images. In *Computer Analysis of Images and Patterns: 17th International Conference, CAIP 2017, Ystad, Sweden, August 22-24, 2017, Proceedings, Part I 17*. Springer International Publishing, 2017.

[10] S. Jiang, N. Bao, A. D. Smith, S. Byndor, R. C. Van Lehn, M. Mavrikakis, N. L. Abbott, and V. M. Zavala. Scalable extraction of information from spatio-temporal patterns of chemoresponsive liquid crystals using topological data analysis. *J. Phys. Chem. C*, 2023.

[11] J. M. Kilner, S. J. Kiebel, and K. J. Friston. Applications of random field theory to electrophysiology. *Neuroscience Letters*, 3:174–178, 2005.

[12] E. Leonhard. Elementa dosctrinae solidorum. *Novi commentarii academiae scientiarum Petropolitanae*, pages 109–140, 1758.

[13] L. Marsh, F. Y. Zhou, X. Qin, X. Lu, H. M. Byrne, and H. A. Harrington. Detecting temporal shape changes with the euler characteristic transform. *arXiv preprint arXiv:2212.10883*, 2022.

[14] Y. Nishimura and H. Nakai. Dcdftbmd: Divide-and-conquer density functional tight-binding program for huge-system quantum mechanical molecular dynamics simulations. *Journal of Computational Chemistry*, 40(15):1538–1549, 2019.

[15] P. Pranav, R. J. Adler, T. Buchert, H. Edelsbrunner, B. J. T. Jones, A. Schwartzman, H. Wagner, and R. van de Weygaert. Unexpected topology of the temperature fluctuations in the cosmic microwave background. *A&A*, A163, 2019.

[16] T. G. Project. *GUDHI User and Reference Manual*. GUDHI Editorial Board, 2015.

[17] E. Richardson and M. Werman. Efficient classification using the euler characteristic. *Pattern Recognition Letters*, 49:99–106, 2014.

[18] A. Rieser. A note on the simplex-tree construction of the vietoris-rips complex, 2023.

[19] J. Schmalzing, M. Kerscher, and T. Buchert. Minkowski functionals in cosmology. *arXiv preprint astro-ph/9508154*, 1995.

[20] A. Signoroni, M. Savardi, A. Baronio, and S. Benini. Deep learning meets hyperspectral image analysis: A multidisciplinary review. *Journal of Imaging*, 5(5), 2019.

[21] A. Smith, S. Runde, A. K. Chew, A. S. Kelkar, U. Maheshwari, R. C. Van Lehn, and V. M. Zavala. Topological analysis of molecular dynamics simulations using the euler characteristic. *J. Chem. Theory Comput.*, 19(5):1553–1567, 2023.

[22] A. D. Smith, P. Dłotko, and V. M. Zavala. Topological data analysis: concepts, computation, and applications in chemical engineering. *Computers & Chemical Engineering*, 146, 2021.

[23] A. D. Smith and V. M. Zavala. The euler characteristic: a general topological descriptor for complex data. *Computers & Chemical Engineering*, 154, 2021.

[24] L. Snidaro and G. L. Foresti. Real-time thresholding with euler numbers. *Pattern Recognition Letters*, 24:1533–1544, 2003.

[25] SPECIM. The top 10 questions about hyperspectral imaging (part 2 of 2). https://www.specim.com/the-top-10-questions-about-hyperspectral-imaging-part-2-of-2/, 2016.

[26] I. technology. *MULTIEYE2 USER MANUAL*, v2.6.0+ edition.

[27] J. Toriwaki and T. Yonekura. Euler number and connectivity indexes of a three dimensional digital picture. *Forma*, 17:183–209, 2002.

[28] L. A. Varga, J. Makowski, and A. Zell. Measuring the ripeness of fruit with hyperspectral imaging and deep learning. *CoRR*, abs/2104.09808, 2021.

[29] V.-Q. Vuong, C. Cevallos, B. Hourahine, B. Aradi, J. Jakowski, S. Irle, and C. Camacho. Accelerating the density-functional tight-binding method using graphical processing units. *The Journal of Chemical Physics*, 158(8):084802, 02 2023.

[30] F. Wang, H. Wagner, and C. Chen. Gpu computation of the euler characteristic curve for imaging data. *arXiv preprint arXiv:2203.09087v2*, 2023.

A 3D Algorithms

Algorithm 3 Bitmap contribution (3D, serial). Parallel mode runs lines 7 to 35 on different threads.

1: Field $\leftarrow$ read(data file)
2: $w \leftarrow$ field width
3: $h \leftarrow$ field height
4: $d \leftarrow$ field depth
5: $M \leftarrow$ largest filtration value
6: $q_{k,\text{in}}, q_{k,\text{out}} \leftarrow$ Zeros of length M+2 $\forall k \in \{0, 1, ..., 42\}$
7: **for all** $i < h + 1$ **do**
8: **for all** $j < w + 1$ **do**
9: **for all** $k < d + 1$ **do**
10: data $\leftarrow [M + 1, M + 1, M + 1, M + 1, M + 1, M + 1, M + 1, M + 1]$
11: data $\leftarrow$ fill_data_3D (i, j, k) (Algorithm 4)
12: pos $\leftarrow$ argsort(data)
13: adjacency $\leftarrow$ compute $\left[f_{\text{adj}}\left(\text{pos}[0, 1]\right), f_{\text{adj}}\left(\text{pos}[0, 1, 2]\right), f_{\text{adj}}\left(\text{pos}[0, 1, 2, 3]\right)\right]$
14: $\text{adjacency}_{\text{empty}} \leftarrow$ compute $\left[f_{\text{adj}}\left(\text{pos}[7, 6]\right), f_{\text{adj}}\left(\text{pos}[7, 6, 5]\right)\right]$
15: $q_{10,\text{in}}[\text{data}[0]]{+}{+}, q_{10,\text{out}}[\text{data}[1]]{+}{+}$
16: **if** adjacency[0] $< \sqrt{2}$ **then**
17: $q_{20,\text{in}}[\text{data}[1]]{+}{+}, q_{20,\text{out}}[\text{data}[2]]{+}{+}$
18: **else if** adjacency[0] $< \sqrt{3}$ **then**
19: $q_{21,\text{in}}[\text{data}[1]]{+}{+}, q_{21,\text{out}}[\text{data}[2]]{+}{+}$
20: **else**
21: $q_{22,\text{in}}[\text{data}[1]]{+}{+}, q_{22,\text{out}}[\text{data}[2]]{+}{+}$
22: **end if**
23: **if** adjacency[0] $< 1 + \sqrt{2} + \sqrt{3}$ **then**
24: $q_{30,\text{in}}[\text{data}[2]]{+}{+}, q_{30,\text{out}}[\text{data}[3]]{+}{+}$
25: **else if** adjacency[0] $< \sqrt{3}$ **then**
26: $q_{31,\text{in}}[\text{data}[2]]{+}{+}, q_{31,\text{out}}[\text{data}[3]]{+}{+}$
27: **else**
28: $q_{32,\text{in}}[\text{data}[2]]{+}{+}, q_{32,\text{out}}[\text{data}[3]]{+}{+}$
29: **end if**
30: $\vdots$
31: Continue similarly for q_{40} through q_{80}. Use adjacency thresholds in Table 2
32: $\vdots$
33: **end for**
34: **end for**
35: **end for**

Algorithm 4 Filling data for 3D field.

1: $(i, j, k) \leftarrow$ input values
2: data $\leftarrow$ input data
3: **if** $i > 0 \wedge j > 0 \wedge k > 0$ **then**
4: data[0] $\leftarrow$ Field$(i-1, j-1, k-1)$
5: **end if**
6: **if** $i > 0 \wedge j > 0 \wedge k < d$ **then**
7: data[4] $\leftarrow$ Field$(i-1, j-1, k)$
8: **end if**
9: **if** $i > 0 \wedge j < w \wedge k > 0$ **then**
10: data[2] $\leftarrow$ Field$(i-1, j, k-1)$
11: **end if**
12: **if** $i > 0 \wedge j < w \wedge k < d$ **then**
13: data[6] $\leftarrow$ Field$(i-1, j, k)$
14: **end if**
15: **if** $i < h \wedge j > 0 \wedge k > 0$ **then**
16: data[1] $\leftarrow$ Field$(i, j-1, k-1)$
17: **end if**
18: **if** $i < h \wedge j > 0 \wedge k < d$ **then**
19: data[5] $\leftarrow$ Field$(i, j-1, k)$
20: **end if**
21: **if** $i < h \wedge j < w \wedge k > 0$ **then**
22: data[3] $\leftarrow$ Field$(i, j, k-1)$
23: **end if**
24: **if** $i < h \wedge j < w \wedge k < d$ **then**
25: data[7] $\leftarrow$ Field(i, j, k)
26: **end if**