DOI: 10.1111/cgf.14784

COMPUTER GRAPHICS *forum*
Volume 42 (2023), number 6, e14784

TACHYON: Efficient Shared Memory Parallel Computation of Extremum Graphs

Abhijath Ande, Varshini Subhash and Vijay Natarajan

Department of Computer Science and Automation, Indian Institute of Science, Bangalore, India
abhijathande@alum.iisc.ac.in,varshini96@gmail.com

Abstract
The extremum graph is a succinct representation of the Morse decomposition of a scalar field. It has increasingly become a useful data structure that supports topological feature-directed visualization of 2D/3D scalar fields, and enables dimensionality reduction together with exploratory analysis of high-dimensional scalar fields. Current methods that employ the extremum graph compute it either using a simple sequential algorithm for computing the Morse decomposition or by computing the more detailed Morse–Smale complex. Both approaches are typically limited to two and three-dimensional scalar fields. We describe a GPU–CPU hybrid parallel algorithm for computing the extremum graph of scalar fields in all dimensions. The proposed shared memory algorithm utilizes both fine-grained parallelism and task parallelism to achieve efficiency. An open source software library, tachyon*, that implements the algorithm exhibits superior performance and good scaling behaviour.*

Keywords: visualization, scalar field, critical point, Morse theory, GPU, hybrid parallel algorithm

CCS Concepts: • Human-centred computing → Visualization techniques; • Computing methodologies → Parallel algorithms

1. Introduction

Topological analysis has become a key tool for the analysis and visualization of data from various branches of scientific research [CFST20, HMST21]. In the context of data represented as scalar fields, topological descriptors such as contour trees, Reeb graphs, merge trees, Morse–Smale complexes and extremum graphs have been widely studied [HLH*16]. Each abstraction provides a different perspective to analyse and interact with the scalar field. The extremum graph is a useful abstraction in situations where the extrema and gradient flow behaviour that determines their connectivity are crucial for understanding the scientific phenomenon [CLB11, TN13]. It is a subset of the Morse–Smale complex that is considerably simpler yet intuitive and detailed enough for various applications. It is a bipartite graph whose nodes are either extrema or saddles and arcs determine connectivity via a gradient path between a saddle-extremum pair. The simple structure of the graph makes it amenable for various applications including segmentation in 2D and 3D scalar fields, feature tracking in time-varying data, and clustering in higher dimensional data. While there exist efficient algorithms to compute the more general Morse–Smale complex, there is a need for a lightweight, fast and scalable algorithm that can compute the extremum graph in all dimensions.

1.1. Related work

The Morse complex and Morse–Smale complex were introduced as topological structures to represent the gradient flow behaviour of Morse functions [BHEP04, EHNP03]. While the extremum graph may be considered as a 1-skeleton of the Morse complex, it was first described in the currently known form by Correa *et al.* [CLB11] together with a new 2D visual representation called the topological spine. The extremum graph preserves the topological and geometric structure of a scalar field, while the topological spine provides a succinct and easily digestible 2D representation to the user. It essentially preserves the relative location of extrema and knowledge of their neighbourhoods with respect to gradient paths connecting them. Useful applications of extremum graphs include the robust detection of symmetry in noisy scalar fields [TN13], extrema proximity awareness and a feature-aware comparison of similar scalar fields using distance measures [NTN15].

The abundance of large-scale scientific data with increasing feature complexity and precision has directed the attention of the scientific visualization community towards developing parallel algorithms that effectively leverage modern compute power and massively parallel architectures. Specifically, the parallel computation of 3D Morse–Smale complexes has been studied extensively to this effect. Gyulassy *et al.* [GBHP08] developed a memory efficient algorithm for 3D Morse–Smale complex computation, where they partitioned large data into chunks called parcels that fit in memory. By computing the Morse–Smale complexes for each parcel and performing a cancellation-based merging of parcels, they were able to obtain Morse–Smale complexes for large-scale data which originally did not fit in memory. This approach was adapted to a distributed memory setting by Peterka *et al.* [PRG*11] and Gyulassy *et al.* [GPPR12] where they leveraged massively parallel clusters to handle the parcels in parallel. Robins *et al.* [RWS11] and Shivashankar and Natarajan [SSN11] introduced novel locally independent definitions for gradient pairs which facilitated an embarrassingly parallel gradient pair assignment. Novel traversal algorithms for the extraction of ascending and descending manifolds of the extrema and saddles further enhanced the runtime performance of the Morse–Smale complex computation [GRWH12, SN12, DFRS14].

Additionally, there have been advances in obtaining a higher degree of geometric accuracy during parallel Morse–Smale complex computation by Gyulassy *et al.* [GBP18, GGL*14, GBP12] and Bhatia *et al*. [BGL*18]. Several approaches dabbled in CPU-based shared memory parallelization techniques with the exception of Shivashankar and Natarajan [SN12], who introduced a hybrid approach. Their algorithm and the associated software library, pyms3d [SN17], leveraged GPU parallelism for gradient assignment and extrema traversal and CPU parallelism for saddle-saddle traversal. Recently, Subhash *et al.* [SPN20, SPN22] introduced GMSC, an end-to-end GPU parallel algorithm for 3D Morse–Smale complex computation, which resulted in substantial speedups over the CPU and hybrid approaches attempted thus far. GMSC transforms the saddle-saddle computation into a sequence of matrix operations that are amenable to fast parallel computation on the GPU, leading to significant improvement in runtime. We note that the gradient assignment and saddle-extrema traversal steps in GMSC are identical to those employed by pyms3d. Both pyms3d and GMSC are restricted to 3D scalar fields.

A discrete Morse theory-based approach is employed by several of the above-mentioned parallel algorithms and has yielded combinatorial and numerically robust algorithms [GNP*06, GRWH12, DFRS14, SSN11, SN12]. Fugacci *et al.* [FIDF19] also demonstrated an extension to higher dimensions by computing the Morse complex from simplicial complexes. This discrete Morse complex can be leveraged to compute homology [RWS11, HMMN14], perform shape analysis and study scalar fields [DFFIM15].

As mentioned earlier, the extremum graph is a sub-graph of the combinatorial structure or the 1-skeleton of the Morse–Smale complex. The succinct yet highly informative abstraction and its demonstrated role in many applications motivates the need for its efficient computation. To the best of our knowledge, no previous work specifically explores the efficient and scalable parallel computation of extremum graphs in all dimensions. We attempt to address this gap here and describe an open source implementation.

1.2. Contributions

In this paper, we describe a fast shared memory parallel algorithm for computing the extremum graph of an n-dimensional scalar field defined on a grid. The hybrid GPU–CPU parallel computation leverages the GPU for fine-grained parallel computation of nodes of the extremum graph and the CPU for effective task parallel computation of arcs of the graph. Key contributions of the paper include:

- A hybrid GPU–CPU algorithm that efficiently uses both computational resources while enabling the algorithm to be scalable to data that does not fit in the GPU memory.
- An efficient implicit representation of the neighbourhood of a grid vertex that supports fast parallel critical point classification of all vertices.
- TACHYON, a lightweight library that provides a fast implementation of the hybrid algorithm (bitbucket.org/vgl_iisc/tachyon).
- Multiple strategies for effective simplification of the extremum graph that makes it amenable for feature identification and applications.
- Experimental studies on multiple datasets to demonstrate superior performance and scalability.

2. Background

In this section, we define and briefly introduce the necessary terms required to define extremum graphs [EHNP03, CLB11, NTN15]. Given a smooth function $f : \mathbb{M} \rightarrow \mathbb{R}$ over a manifold $\mathbb{M}$ of dimension n, we say that a point $x \in \mathbb{M}$ is *critical* iff $\nabla f(x) = 0$, x is called *regular* otherwise. Also, f is a *Morse function* if all critical points have pairwise distinct function values and none of them are degenerate, i.e. the Hessian evaluated at the point is non-singular. For a critical point x of f, its Morse index is defined as the number of negative eigenvalues of its Hessian matrix evaluated at x. Critical points of index 0 are named *minima*, index n are *maxima* and the others with index k, $1 < k < n - 1$, are called *k-saddles*.

An *integral line* is a maximal curve in $\mathbb{M}$ whose tangent at every point is equal to the gradient of f at that point. Naturally, f monotonically increases along the integral line and its two end points (limit points) are critical points of f. The Morse function f determines a decomposition of $\mathbb{M}$ based on the integral lines. The union of integral lines that terminate at a critical point is called its *descending manifold*. The descending manifold of a maximum is an n-dimensional manifold. The collection of descending manifolds of critical points of f partition $\mathbb{M}$ and is called the *Morse decomposition*. The *ascending manifold* of a minimum is similarly defined as the union of integral lines that originate at a minimum. Again, the collection of ascending manifolds of critical points of f partition $\mathbb{M}$. The *extremum graph* is a representation of the Morse decomposition. It is called a maximum/minimum graph if it represents the decomposition into descending/ascending manifolds, respectively. Without loss of generality, we restrict the discussion in this paper to maximum graphs, and refer to them as extremum graphs.

An $(n-1)$-saddle s of f lies on the boundary of the descending manifold of a maximum m, i.e. an integral line originating at s terminates at m. The extremum graph captures this relationship between maxima and $(n-1)$-saddles of f, and thereby captures the combinatorial structure of the Morse decomposition. The node set of the extremum graph consists of the maxima and $(n-1)$-saddles of f. An arc (s, m) belongs to the extremum graph if s lies on the boundary of the descending manifold of m. Figure 1 shows the

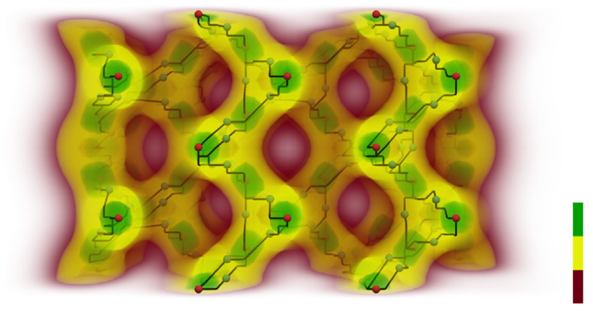


Figure 1: *Extremum graph for the silicium grid. Maxima (red) and 2-saddles (green) are connected by arcs. The geometry of an arc is determined by the integral line (black) that connects the maximum and the 2-saddle. The scalar field is shown as a volume rendering using a simple banded colour map.*

extremum graph for a 3D scalar field, the output of a simulation of a silicium grid.

3. Computation of Extremum Graphs

There are two popular approaches to compute extremum graphs, namely flood fill and gradient path tracing. While both methods produce equivalent results, each method has its own associated advantages and disadvantages. We begin with a brief discussion about the two approaches and justify why we propose gradient path tracing towards the development of an efficient and scalable parallel algorithm.

3.1. Flood fill versus gradient path tracing

The flood fill approach computes the extremum graph in one sweep over the list of grid vertices by incrementally growing the descending n-dimensional manifolds of all maxima. The vertices are processed in decreasing order of scalar value. Processing a vertex v includes expanding the descending manifolds that have reached v in the steepest descent direction, namely the vertex with the smallest value in the lower link. This approach has been employed to compute the Morse decomposition [EHNP03]. The method implicitly recognizes a maximum as a vertex where no descending manifold have reached and an $(n-1)$-saddle as a vertex where multiple descending manifolds merge. The adjacency between maxima and saddles is recorded when the saddle vertex is processed. While this method computes the connectivity of the extremum graph, it does not explicitly capture the gradient paths between the maxima and saddles. A subsequent step needs to be invoked to compute these gradient paths as necessary, a limitation of this approach. Further, this method has a high memory footprint because it requires the collection of all descending manifolds to be stored.

The gradient path tracing approach computes the extremum graph in two steps. The first step locates and classifies the maxima and $(n-1)$-saddles. It analyses the link of a vertex to classify the vertex. Importantly, the classification of a vertex is independent of the classification of other vertices and depends on a local neighbourhood. The second step traces gradient paths from each $(n-1)$-saddle towards extrema. This tracing requires gradients to be computed only for a small subset of regular vertices, thereby resulting in a low computational load. The memory footprint is also low because the first step requires a simple query within a constant sized local neighbourhood and the second step requires only the end point of the path to be stored.

There exist multiple parallel algorithms for flood fill. However, these method do not scale due to the communication required to handle the merge events and subsequent serialization of the descending manifold growth computation. In contrast, the first step of the gradient path tracing is amenable to fine-grained parallelism, is simple, and highly scalable. The second step can also be accelerated via task parallelism. These advantages motivate us to develop a gradient path tracing algorithm for parallel computation of the extremum graph. In the following, we describe the two steps of the algorithm with a focus on how the steps are parallelized, followed by a discussion on effective simplification of the extremum graph to make it suitable for applications.

3.2. Grid tessellation

We assume that the scalar field is available as a collection of samples at vertices of an n-dimensional grid. Vertices of the grid that represent the domain $\mathcal{D}^n$ have integral coordinates

$$V(\mathcal{D}^n) = \{v \mid v \in \underbrace{\mathbb{Z} \times \mathbb{Z} \times \cdots \times \mathbb{Z}}_{n}\}.$$

Edges of the grid are between two vertices that differ in exactly one coordinate by a value of 1. Methods for topological analysis and visualization of this scalar field typically assume an input piecewise multilinear or piecewise linear scalar function. The samples at vertices may be extended into a scalar function by either using a multilinear interpolant (trilinear in 3D) within each grid cell or by tessellating the grid cells into linear cells (tetrahedra in 3D) followed by linear interpolation within each linear cell. In either case, the local maxima of the scalar field are located at grid vertices.

We tessellate the uniform grid into irregular linear cells (tetrahedra in 3D) with the aim of simplifying the extremum graph computation and to support future extensions of the method to unstructured grids. We choose a tessellation that decomposes all grid cells in the same manner and one that is consistent on the common face between adjacent cells, called the Freudenthal subdivision [CMS06]. Figure 2 shows the tessellated grid in 2D and 3D (for illustration). The resulting edge set of the tessellated grid $E(\mathcal{D}^n)$ can be stored implicitly and recovered using a simple routine, see Algorithm 1. The algorithm essentially dictates that two distinct vertices are connected by an edge if and only if the difference vector between the two points consists entirely of non-negative or non-positive values and the magnitude of the non-zero values is 1.

The edge set of the tessellated grid

$$E(\mathcal{D}^n) = \{(u, v) \mid u, v \in V(\mathcal{D}^n) \wedge GridAdjacency(u, v)\}.$$

Figure 2 also shows the edges incident on a given vertex of the tessellated grid. It follows from the symmetry of the edge set that the number of edges incident on a vertex v is equal to twice the number

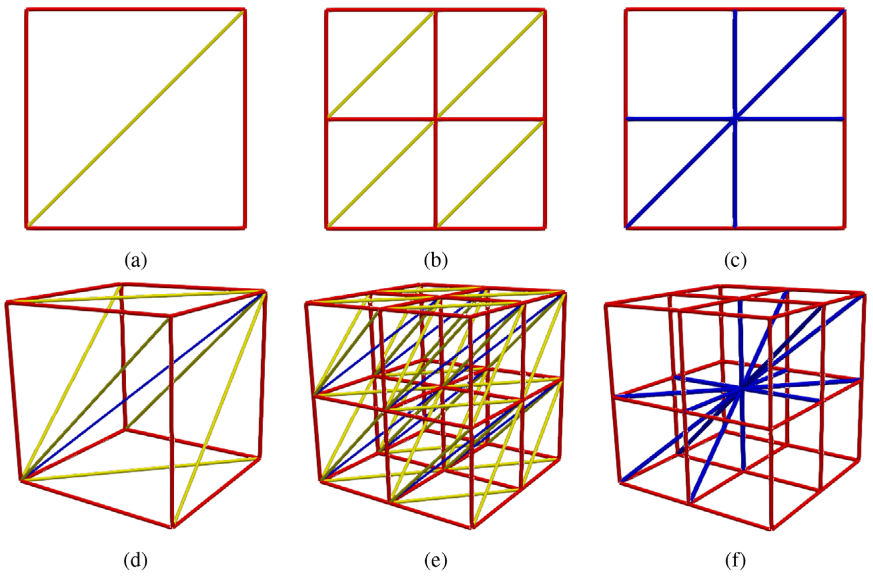


Figure 2: *Tessellating 2D and 3D grids. (a) A 2D cell is decomposed into two triangles. (b) Tessellating a 2D grid. (d) A 3D grid cell is decomposed into six tetrahedra. (e) Tessellating a 3D grid. The decomposition of the common faces between two adjacent grid cells is consistent. (c,f) Edges of the neighbourhood graph (blue) of the vertex at the centre. The vertex has six neighbours in a 2D grid and 14 neighbours in a 3D grid.*

Algorithm 1. GridAdjacency

```
Input: $p, q$ : Two vertices. Test if they are adjacent in the
       tessellated $n$-dimensional grid.
Result: $True$ if $p, q$ are adjacent and $False$ otherwise
1  $n \leftarrow len(p)$                 ▷ $p$ is a vertex in an $n$-dimensional grid
2  $U \leftarrow \phi$
3  foreach $i \in 1..n$ do
4      $d \leftarrow p_i - q_i$          ▷ difference between $i^{th}$ coordinates
5      $U \leftarrow U \cup \{d\}$       ▷ collect component wise difference
6  end
7  if $U \subseteq \{0,1\}$ or $U \subseteq \{0,-1\}$ then
8      return $True$
9  end
10 else
11     return $False$
12 end
```

of non-zero difference vectors, namely $2 \times (2^n - 1)$ for nD grids. So, the number of vertices in the neighbourhood of v in a 3D grid equals 14. Critical points are identified and classified based on a connected component labelling of this neighbourhood. We discuss this classification next.

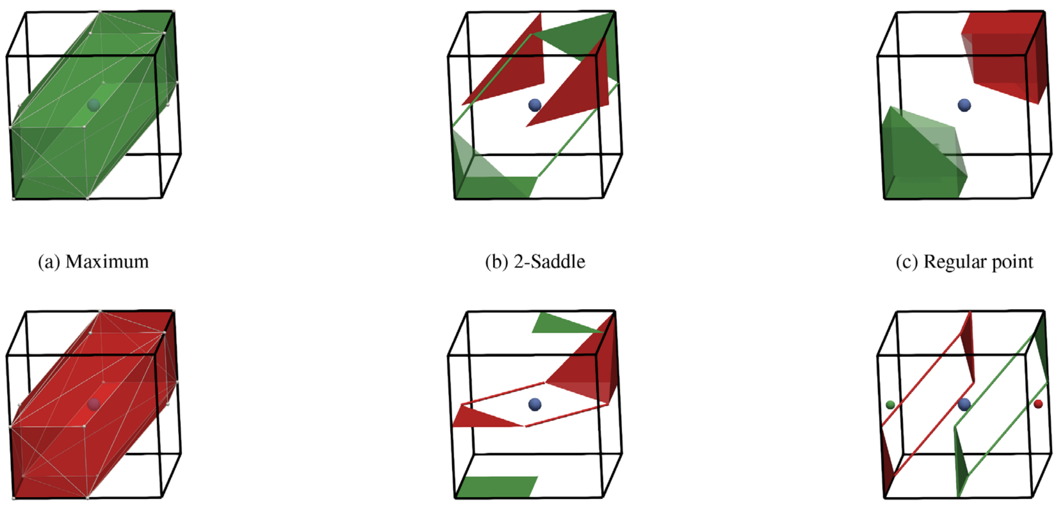
3.3. Critical point classification

Critical points of the piecewise linear function defined over a tessellated grid can be identified and classified based on a local neighbourhood [Ban70, EHNP03]. All critical points are located at vertices of the grid. The *star* of a vertex v consists of v together with the collection of edges, triangles, tetrahedra and higher dimensional cells incident on it. Figure 2f shows the edges (blue) of the star of a vertex in a cube grid. The collection of end points of these edges, excluding v, together with the induced edges and triangles form the *link* of v. The star and link are two useful notions of neighbourhood of a vertex within the tessellated grid. Assuming that there are no degeneracies, scalar values at vertices in the link are either lower or higher than at v. Vertices of the link with scalar value lower than at v together with the induced edges and triangles form the *lower link* of v. Similarly, the *upper link* of v is defined as the collection of vertices of the link with scalar values greater than at v together with the induced edges and triangles.

Critical points are classified based on the number of connected components of the upper and lower link, denoted β_0^+, β_0^- and called the zeroth Betti number [EHNP03]. Figure 3 illustrates the upper (red) and lower (green) link for various types of critical points and for a regular point. Since our objective is to compute the extremum graph, we are specifically interested in identifying the maxima and 2-saddles. The first step of the algorithm visits each grid vertex and classifies it as critical or regular as shown in Table 1.

Both β_0^+ and β_0^- are typically computed by performing a BFS graph traversal on the upper and lower link of the vertex v. The BFS traversal of the upper and lower link is an embarrassingly parallel task across all vertices. We utilize GPU parallelism to perform this traversal in a massively parallel fashion by launching a CUDA thread for each vertex. Our implementation moves this task automatically to a multicore CPU in the absence of a CUDA enabled GPU.

In order to make the computation GPU friendly, we utilize a union-find-based connected component tracking instead of the BFS traversal, similar to previous approaches [CWS*19]. The union-find



(a) Maximum (b) 2-Saddle (c) Regular point

(d) Minimum (e) 1-Saddle (f) Multi-saddle

Figure 3: *Critical points can be identified and classified based on the upper (red) and lower (green) link connectivity. The upper link of a maximum (a) is empty and the lower link is equivalent to a sphere, whereas the upper link of a minimum (d) is equivalent to a sphere and the lower link is empty. The upper link of a simple 2-saddle (b) consists of two connected components and its lower link consists of a single component, whereas the upper link of a 1-saddle (e) consists of a single component and the lower link consists of two connected components. Both the upper and lower links of a regular point (c) consist of a single component. The multi-saddle (f) is a degenerate structure that does exist in piecewise linear functions.*

Table 1: *Classifying a vertex based on the topology of the upper and lower link.*

	β_0^+	β_0^-
Maximum	0	1
$(n-1)$-saddle	≥2	•
1-saddle	•	≥2
Minimum	1	0
Regular	1	1

β_0^+ and β_0^- count the number of connected components of the upper and lower link, respectively. • indicates that the value does not affect the classification.

data structure is initialized with a collection of singleton sets, each containing a vertex of the link of v. Next, we test for the existence of an edge between a pair of vertices in the link using Algorithm 1. If the edge exists and both end points of the edge belong to the upper link (lower link), we perform a union operation on the corresponding sets. Degeneracies are handled using a simulated perturbation [EH09, Section 1.4], which consistently determines if the scalar value at a link vertex is lower or higher than the value at v.

After identifying the connected components in the link, we label each component as upper or lower link by testing one vertex within the component and hence compute β_0^+ and β_0^-. This method is GPU friendly as all working threads test the same number of edges and require almost the same time, thereby causing a very low thread divergence. If a vertex is identified as regular, we additionally compute the gradient of the scalar field at the vertex. The gradient is approximated as the vector towards the vertex with the highest scalar value in the upper link.

3.4. Path tracing

We opt to use a direct path tracing algorithm to trace the saddle-maximum arcs that constitute the extremum graph over the flood-fill approach. The primary reason for this choice is the benefit in terms of scalability of the computation. In addition, path tracing does not require each vertex of the grid be visited. Further, a flood-fill-based algorithm requires an ordered list of vertices, sorted on the scalar values.

The path tracing algorithm requires only the gradient at vertices. The gradient paths in an extremum graph originate from $(n-1)$-saddles, so the algorithm visits only a small fraction of vertices of the grid. However, this approach is not without challenges. The operation of tracing a gradient path is inherently serial in nature. One approach towards parallelism is to trace the gradient paths for all $(n-1)$-saddles concurrently.

Algorithm 2 computes the collection of gradient paths that originate at an $(n-1)$-saddle. The subroutine *UpperLinkRep* returns the highest vertex in each upper link component of the saddle. For each such vertex, it iteratively follows the gradient vectors until termination at a maximum. This algorithm is executed in parallel for each $(n-1)$-saddle. Unlike the critical point classification, Algorithm 2 is not amenable to efficient GPU parallelism. Varying lengths of the gradient paths causes unequal division of work among CUDA threads, and results in significant thread divergence.

Algorithm 2. TraceGradientPaths

```
Input: $s, M$: Source $(n-1)$ saddle and the set of all
       maxima
Result: Set of gradient paths originating from $s$
1  $P \leftarrow \phi$
2  foreach $u \in UpperLinkRep(s)$ do
3      $p \leftarrow s$                          ▷ Initialize path $p$ with $s$
4      while $u \notin M$ do
5          $p \leftarrow p \,||\, u$               ▷ Append $u$ to $p$
6          $u \leftarrow gradient(u)$           ▷ Follow gradient at $u$
7      end
8      $P \leftarrow P \cup \{p\}$
9  end
10 return $P$
```

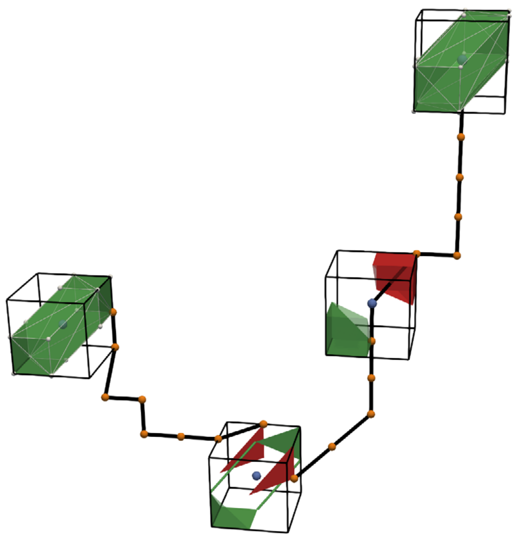


Figure 4: *Gradient path tracing for computing arcs of extremum graph. Ascending one-dimensional manifolds are computed for each 2-saddle. The gradient paths that constitute these ascending manifolds represent the arcs of the extremum graph. Note that they pass through regular points, entering and exiting its neighbourhood via the lower and upper link, respectively.*

Figure 4 illustrates the ascending one-dimensional manifolds for a 2-saddle. Notice how the gradient paths originate from the 2-saddle via each upper link component and terminate at maxima. Given two adjacent regular points (orange) (p, q) on a gradient path, p lies in the lower link of q and q lies in the upper link of p. The neighbourhood of one regular point is shown in detail. The gradient path enters through its lower link and exits via its upper link.

3.5. Extremum graph simplification

Scientific data are often noisy, which manifests as an increased number of critical points and hence a larger sized extremum graph. Topological simplification identifies noisy topological features and removes them in a controlled manner, often as a sequence of critical point pair cancellation operations. The simplification is typically directed by the notion of persistence [ELZ02]. Indeed, the extremum graph computed using the algorithm that we describe in the previous sections contains a large number of saddles and extrema for all datasets. Several critical points and their incident gradient paths correspond to noisy topological features. The presence of these noisy elements cause occlusion, thereby adversely affecting data visualization tasks, and hinder feature selection. We propose three simplification operations to aid the proper identification of key features from the extremum graph. The *persistence directed cancellation* and *saturated persistence directed simplification* algorithms are implemented in TACHYON as they are useful in removing noise and potentially uninteresting features from the extremum graph [CLB11]. We introduce the *arc-bundling* algorithm as a method to remove multiple connections between a pair of extrema. Any sequence of these operations may be applied on an extremum graph. We describe the simplification operations below. Their implementations in TACHYON are serial in nature.

Arc bundling:. The gradient path tracing algorithm computes individual arcs between an $(n-1)$-saddle-maximum pair independent of each other. So, it is likely that a pair of maxima contain more than one common $(n-1)$-saddle in their neighbourhood. We select a single representative saddle based on a specified criterion and discard the remaining saddles that are shared by the two maxima within their neighbourhood. We term this pruning operation as arc bundling, as this reduces the number of arcs between an extrema pair. Note that this operation does not disturb the connectivity between extrema and the resulting graph continues to represent the Morse decomposition. In our current implementation, we select the saddle with the highest scalar value as the representative.

Persistence directed cancellation:. This operation cancels an $(n-1)$-saddle-maximum pair that is connected by an arc in the extremum graph and reconnects the neighbourhood of both nodes. The critical point pairs are ordered based on their difference in scalar value, motivated by the notion of topological persistence, and scheduled for cancellation. The simplification begins by initializing a priority queue with all $(n-1)$-saddles, where the priority is inversely related to the persistence of the saddle. Computing the topological persistence of all saddles upfront is unnecessary and computationally expensive. We use lazy updates where the true persistence of a saddle is computed when it reaches the top of the queue and is ready for cancellation. We initialize the cost of cancelling a simple $(n-1)$-saddle as the smallest difference in scalar value between the saddle and its two adjacent maxima in the extrema graph. This cost is recomputed whenever a saddle is removed from the top of the queue.

The next step of the simplification iteratively removes the saddle s from the top of the queue, re-computes its cost and decides whether to discard, reinsert or cancel the saddle. If the updated cost is above a user-specified persistence threshold, the saddle should not be cancelled. Note that subsequent cancellations will not decrease the cost of this saddle and so, it is safely discarded. If the updated cost is greater than the cost of the top of the queue (but smaller than the persistence threshold), then the saddle is reinserted into the queue because the queue contains other saddles with lower costs that need

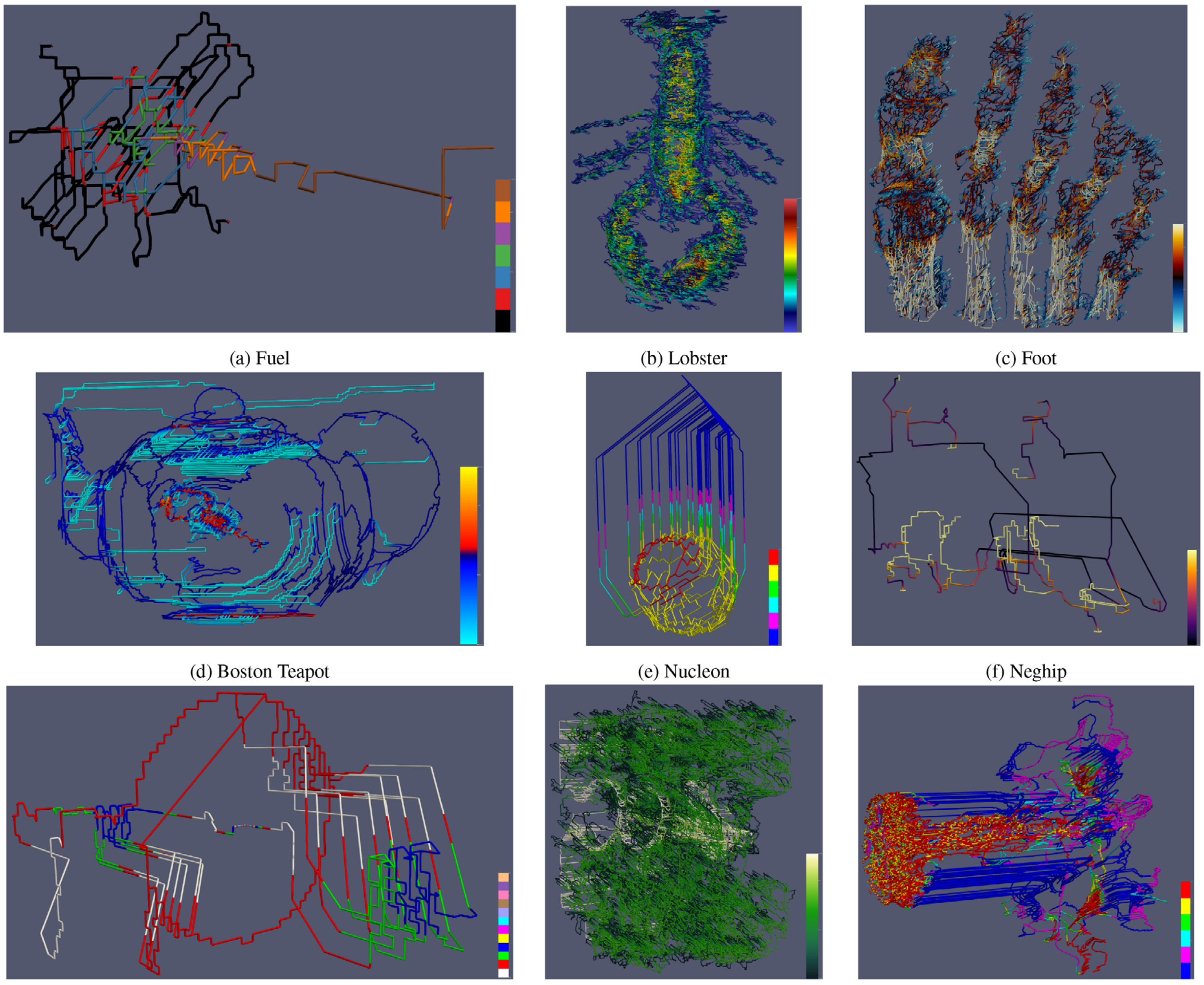

(a) Fuel (b) Lobster (c) Foot
(d) Boston Teapot (e) Nucleon (f) Neghip
(g) Hydrogen (h) Engine (i) Heptane Gas

Figure 5: *Simplified extremum graphs for various datasets. Nodes (extrema and saddles) are not displayed to reduce clutter. The scalar field is mapped to colour as indicated by the legend.*

to be cancelled. Else, the saddle should be cancelled. The gradient path from s towards its persistence pair m is reversed to cancel the critical point pair. The above step is repeated until the queue is empty. This simplification removes all low persistence critical point pairs (arcs) from the extremum graph, while retaining features of potential interest.

Multi-saddles have greater than two arcs incident on them. The cost of a multi-saddle is computed as the difference in scalar value with the second highest maximum adjacent to it. The multi-saddle is cancelled by reversing gradient paths that originate from it towards all maxima, except for the highest maximum, also called the surviving maximum.

Saturated persistence directed simplification:. While persistence directed cancellation removes saddles whose persistence falls below a given threshold, it does not remove long and potentially unintuitive arcs that connect spatially distant maxima via a saddle. These additional arcs often do not represent structural features. In addition, they cause clutter and occlude potentially interesting features. We propose the use of saturated persistence directed extremum graph simplification [CLB11], which prunes such arcs.

We apply all three operations described above to obtain simplified extremum graphs for various scientific datasets as shown in Figure 5.

4. Hybrid GPU–CPU Parallel Computation

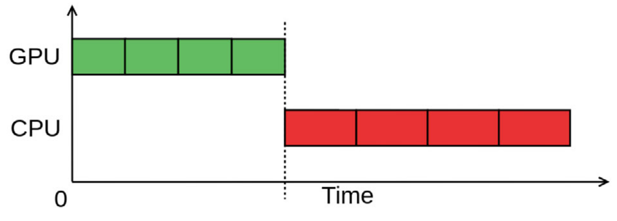
The parallel algorithm described in the previous section works when the dataset together with the associated data structures fit within the GPU and main memory. The GPU memory is often much smaller and determines the size of the data that can be processed. We now describe an extension of both steps of the gradient path tracing algorithm to handle datasets that are too large to fit within the GPU memory. We do assume, however, that the data fits within the main memory. In order to classify critical points in large datasets, we view the dataset as an array of scalar field values in row major order. Thus, each vertex of the grid is mapped to an index of this array. We partition the array into contiguous sub-arrays called *blocks*. These contiguous blocks partition the n-dimensional dataset along one of its dimensions, for example, along the y-axis for 2D data and along the z-axis for 3D data. A block is typically adjacent to two blocks, except for the first and last block, which are adjacent to exactly one block. The domain is partitioned such that each block fits in the available GPU memory during runtime. The size of a block is computed based on a linear regression model that predicts the number of vertices that can be accommodated in the GPU. The linear regression model is constructed from a series of experiments on datasets of different sizes executed on different GPUs.

Critical point classification of vertices that lie on the boundary of a block requires access to their neighbouring vertices. So, we append each block with a set of ghost vertices constituted by vertices from the link of the boundary. These ghost vertices are not classified into regular or critical while processing the block. Critical point classification of all vertices in a block is computed on the GPU and the results are transferred to main memory. This process is repeated for all blocks in sequence. If multiple GPUs are available, then multiple blocks could be processed concurrently. TACHYON currently does not utilize multiple GPUs.

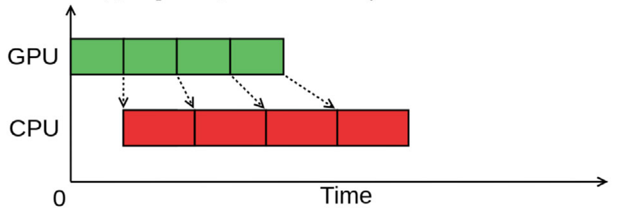
We could employ a simple approach for parallel gradient path tracing after all vertices are processed in the first step. However, this leads to a waste of CPU resources while the GPU is executing the first step. An optimal approach utilizes the CPU simultaneously for gradient path tracing within blocks that are already processed in the GPU. However, this path tracing is non-trivial. The gradient path tracing is indeed executed within each block but the task needs to be temporarily paused when the path reaches the block boundary and attempts to exit the block. Figure 6 illustrates how such an approach reduces the idle CPU time.

Gradients paths that need to exit a block are recorded and their tracing is resumed once the adjacent block is processed by the GPU. Path tracing in this hybrid mode is performed by maintaining a list of partially traced paths, whose tracing could not be completed because it entered a block that is not yet processed by the GPU to classify the critical points. A partial path is stored as a triple—the origin saddle, the first vertex on the path and the last visited vertex. The first vertex is required to distinguish between all paths that originate from the saddle. For each partial path, the last visited vertex is updated when path tracing is paused at a block boundary. Path tracing is resumed when the adjacent block containing the vertex is processed by the GPU. The tracing of a specific path continues until a maximum is reached and the overall tracing process terminates when the list of partial paths is exhausted. Thus, each vertex on a saddle-maximum path is visited once, which implies that the worst case running time is determined by the total number of vertices across all traced paths. The space required for path tracing is equal to the maximum size of the partial paths list.

The above method can be further optimized by noticing that at any instant the collection of blocks processed by the GPU thus far



(a) Sequential, GPU followed by CPU execution.



(b) Hybrid GPU-CPU mode.

Figure 6: *Hybrid GPU–CPU parallel computation for a dataset partitioned into four blocks. In the sequential execution model (a), point classification step (green) processes the blocks one by one on the GPU. Gradient path tracing (red) for the blocks executes on the CPU after point classification is complete for all blocks. The CPU cores lie idle, waiting for the completion of critical point classification for all blocks on the GPU. In the hybrid GPU–CPU mode (b), gradient path tracing within the first block begins soon after it is processed by the GPU. The transfer of control from GPU to CPU for a block is indicated by arrows. The base of the arrow lies at the finishing time for a block on GPU and the tip lies at the time when that block is taken up for processing on the CPU.*

can be viewed as a single but larger contiguous block. This is true if the blocks are processed in sequence. We fuse the blocks, iteratively including the next block after point classification is complete. This fuse step reduces the number of boundary crossings that may occur during gradient path tracing by simply reducing the number of possible boundaries to one, thus allowing for a majority of threads to complete tracing. By nature of this scheme, the extremum graph is computed without the need of any further processing, once gradient path tracing is completed for the last block.

5. The TACHYON Library

TACHYON (bitbucket.org/vgl_iisc/tachyon) is a C++ software library that enables efficient computation of the extremum graph by offloading major portions of the computation onto an Nvidia GPU that supports CUDA. We list below a few salient features of the software library:

- Leverages GPU computation power via CUDA kernels to perform point classification, an embarrassingly parallel task.
- Written in C++14. Utilizes standard threading libraries and synchronization utilities to implement parallelism for most tasks that cannot be performed on GPUs, thereby ensuring ease-of-installation and portability.
- Generalized to work for n-dimensional scalar fields. Computes both maximum and minimum graph, as required.

- Automatically switches to a hybrid GPU–CPU mode, on the fly, for datasets that do not fit in GPU memory.
- Provides graph simplification and cancellation algorithms for downstream applications of the extremum graph.
- Utilizes code and memory optimization tricks, such as bit manipulation operations and caching, for effective use of memory.
- Provides a user friendly command line utility to specify input data format, choice of graph simplification, cancellation algorithms etc.

The library is designed in a modular fashion. For example, it is possible to configure TACHYON to perform point classification on the CPU if NVidia GPUs are not available. On a multi-GPU setup, it allows a user to select which GPU to utilize for computations. Further, it can be extended to support multi-GPU computation. The library has only a small set of dependencies, making it a versatile tool that can be built and used on a wide variety of workstations.

TACHYON is bundled with a user friendly interface that enables a non-expert to use the library for computing the extremum graph. The user interface supports various desired functionalities such as computation of maximum and/or minimum graphs based on user requirement, graph cancellation and simplification operations, support for various data types for storing the scalar field (8/16/32/64-bit signed/unsigned integers and single/double precision floating point values).

6. Experimental Results

Experiments were conducted on datasets of sizes varying from $64 \times 64 \times 64$ to $2048 \times 2048 \times 2612$. These datasets were obtained from the Open Scientific Visualization Dataset Repository [Ope22]. All experiments, unless stated otherwise, were run on a workstation with an Intel(R) Xeon(R) Gold 6258R CPU @ 2.70GHz 28 Cores/56 Threads, 640GB RAM, and an Nvidia GeForce RTX 3080 GPU with 10GB GDDR6X RAM. Runtimes were computed as an average over five samples after dropping the best and worst timings out of seven runs. GPU temperatures were also kept within optimal operating ranges of <85ºC. We have performed experiments to highlight different properties of TACHYON and to evaluate its performance. Table 2 lists the various datasets used in the following experiments.

6.1. Internal memory computation

We first evaluate the runtime performance of the algorithm for datasets that fit within the GPU memory. In this experiment, we compute the wall clock time for computing the extremum graphs on a large collection of datasets, see Table 2. These computations did not require the hybrid GPU–CPU execution. Figure 7 shows the total runtime and the split up between the two major steps and the time for data structure updates following the computation. Datasets that are smaller in size compared to Magnetic Reconnection require under 2.3 s. The figure shows runtimes only for larger datasets that fit in memory. Our key observations are:

- Time required for critical point classification on the GPU scales linearly with the domain size.

Table 2: *Datasets used in the computational experiments [Ope22], their size, and the number of critical points.*

Dataset	Size	#Critical Pts
Nucleon	41×41×41	421
Silicium	98×34×34	548
Neghip	64×64×64	1646
Fuel	64×64×64	296
Hydrogen	128×128×128	11370
Shockwave	64×64×512	991
Lobster	301×324×56	347207
Head Mri Ventricles	256×256×124	1719145
Engine	256×256×128	501799
Statue Leg	341×341×93	454011
Boston Teapot	256×256×178	101967
Skull	256×256×256	1877366
Foot	256×256×256	788482
Aneurism	256×256×256	60767
Bonsai	256×256×256	210570
Mrt Angio	416×512×112	5146370
Heptane Gas	302×302×302	61212
Stent	512×512×174	3083028
Pancreas	240×512×512	7215261
Backpack	512×512×373	7066081
Magnetic Reconnection	512×512×512	31147751
Zeiss	680×680×680	3209058
Marmoset Neurons	1024×1024×314	56384791
Stag Beetle	832×832×494	852383
Pawpawsaurus	958×646×1088	83014508
Spathorhynchus	1024×1024×750	47569170
Kingsnake	1024×1024×795	32483997
Chameleon	1024×1024×1080	49325513
Beechnut	1024×1024×1546	159472969
Richtmyer Meshkov	2048×2048×1920	31585707
Woodbranch	2048×2048×2048	1520839571
3D Neurons	2048×2048×2384	1913765935
Pig Heart	2048×2048×2612	906730898
Schwefel 3D	128×128×128 to 1024×1024×1024	1688

The Schwefel dataset [Sch81] is re-sampled on domains of different sizes.

- Time required for gradient path tracing on the CPU scales roughly linear with the number of identified saddles.

We also perform this experiment for higher dimensional data by sampling the Schwefel function [Sch81] on 3-to-6 dimensional domains of varying sizes. Results are shown in Table 3. We again observe an increase with size of data, particularly the time taken for critical point classification.

6.2. Hybrid GPU–CPU hybrid computation

In this experiment, we choose the large datasets that do not fit in GPU memory and required partitioning into multiple blocks. Figure 8 shows the runtime results for these datasets. We make the following observations:

- Critical point classification time is nearly identical for all datasets because they have similar size.

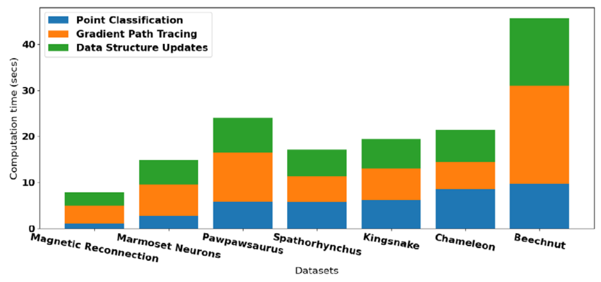


Figure 7: *Extremum graph computation for datasets that fit in GPU memory. The time taken increases with size of dataset and naturally depends on the number of critical points.*

Table 3: *Extremum graph computation for higher dimensional scalar fields.*

Dataset	Size	#Critical Pts	Computation time (in s)
Schwefel 4D	32^4	8449	0.20
	64^4	15072	0.71
	128^4	15072	7.84
Schwefel 5D	16^5	9525	0.33
	32^5	62167	3.87
	64^5	128808	110.01
Schwefel 6D	16^6	46345	9.13
	24^6	444241	104.46
	32^6	444241	432.54

A significant fraction of time was taken for the critical point classification step due to the increased neighbourhood size.

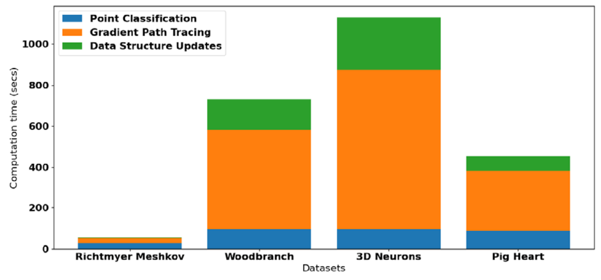


Figure 8: *Performance for large datasets that do not fit in GPU memory. Runtime broadly depends on data size. Pig Heart has a small number of critical points, which explains the smaller runtime. Gradient path tracing constitutes a significant fraction of total runtime in all cases.*

- The gradient path tracing step consumes a large fraction of total runtime.
- An additional factor that influences gradient path tracing time is the number of paths that cross block boundaries. Such paths carry a computational overhead.

6.3. GPU scaling—varying number of cores

We evaluate the performance of the critical point classification step across different GPUs. These experiments are performed on the dataset Zeiss. The aim is to understand whether the computation scales to the number of available compute units on the GPU. Total computation time is not compared in this experiment because it is strongly influenced by the number of CPU cores and additional hardware features. The results are shown in Figure 9. We observe a monotonically decreasing trend with a good slope, indicating efficient usage of the GPU cores.

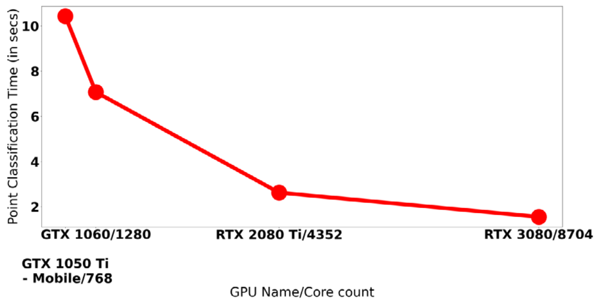


Figure 9: *GPU scaling results.*

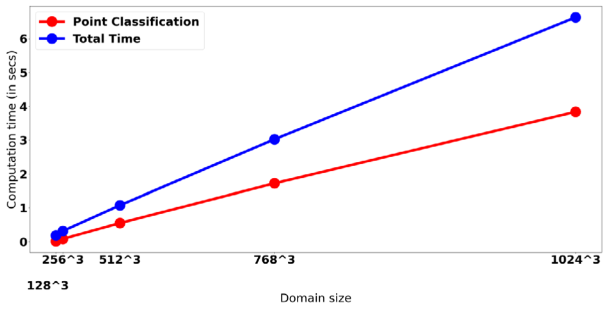


Figure 10: *GPU scaling study for varying domain size. Experiments on datasets obtained by sampling the 3D Schwefel function [Sch81] over different domain sizes reveal a linear relationship between computation time and domain size.*

6.4. GPU scaling—varying domain size

This experiment aims to establish how well TACHYON utilizes the available GPU compute resources. We study how performance varies upon increasing the domain size while fixing the number of critical points. The Schwefel function [Sch81] is sampled on a 3D domain. The data, originally available over a $500 \times 500 \times 500$, is sampled to obtain datasets over domains whose sizes range from $128 \times 128 \times 128$ to $1024 \times 1024 \times 1024$. Across all these sampled functions, the number of critical points and gradient paths do not change thereby allowing a study of the variation of GPU computation time with domain size. All GPU cores are used to compute the extremum graph for this dataset available at different resolutions. The results are shown in Figure 10. We observe a strong linear dependency on domain size.

6.5. CPU scaling—varying CPU threads

Next, we study scaling behaviour by varying the number of CPU threads. The aim is to study how well TACHYON utilizes the

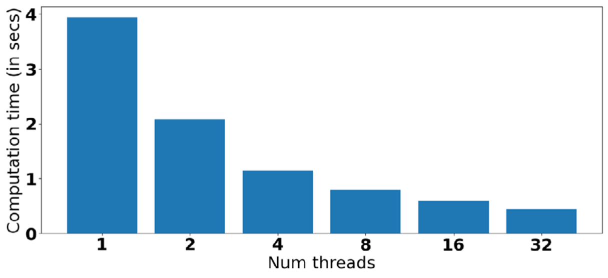


Figure 11: *CPU scaling study for varying number of CPU threads. Experiments on the dataset Zeiss show good scaling up to eight threads.*

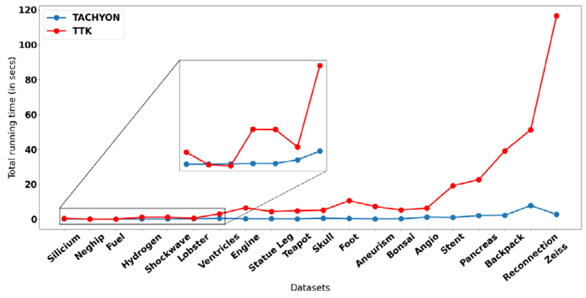


Figure 12: *Runtime comparison between* tachyon *and TTK[TFL*17].* tachyon *performs consistently better, often several orders of magnitude faster.*

available parallelism provided by the CPU. In the ideal case, we expect the computation time to reduce by a factor of two on doubling the number of threads. The results of the experiment are shown in Figure 11. We observe that the performance improvements continue up to eight threads and taper off afterwards.

6.6. Performance comparisons

We now describe results of runtime comparisons against TTK and pyms3d. Both TTK [TFL*17] and pyms3d [SN17] support computation of the complete Morse–Smale complex. The key difference from TACHYON is that they employ a discrete Morse theory-based approach. As a first step, they compute a discrete gradient field on the GPU to locate all critical cells. In contrast, TACHYON uses a lower link-based critical point classification to locate nodes of the extremum graph. They employ a parallel root finding algorithm to compute the entire descending 3-manifold of all extrema and hence compute saddle-maxima arcs. In contrast, TACHYON employs gradient path tracing from each saddle to compute the arcs. The gradient field and saddle-extrema arc computation in GMSC [SPN22] and pyms3d are identical. The only difference between the two implementations is that GMSC uses CUDA whereas pyms3d uses OpenCL.

In order to ensure a fair comparison, we configure pyms3d and TTK to selectively compute the ascending 1-manifolds of 2-saddles and hence output the extremum graph. The results show that our implementation is able to compute extremum graphs faster. In some instances, both TTK and pyms3d fail due to system (GPU/main memory) resource limitations. We find that TACHYON performs better, both in terms of running time and utilization of system resources. Figures 12 and 13 plot the runtimes for increasing dataset sizes. The running time for TACHYON rises slower compared to TTK and pyms3d. Further, pyms3d ran out of memory and could not complete execution for larger data sizes. The comparison with pyms3d was performed on a workstation with an Intel(R) Core(TM) i5-4590 CPU @ 3.30 GHz 2 Cores/4 Threads, 16-GB RAM, and an Nvidia GeForce GTX 1060 GPU with 6-GB GDDR5 RAM.

6.7. Simplification

Finally, we study the ability of the simplification operations to remove noise and clutter. We perform three operations on various datasets with a fixed parameter set: edge bundling, persistence directed cancellation with 5% threshold and saturated persistence-directed simplification with high and low thresholds $p_{lo} = 5\%$ and $p_{hi} = 95\%$. Figure 5 shows the simplified extremum graphs. We also study statistics of the size of extremum graph before and after simplification, see Figure 14. In some noisy datasets, a small threshold already removes a large fraction of nodes and arcs whereas in other datasets that contain a smaller number of features, the reduction is not significant.

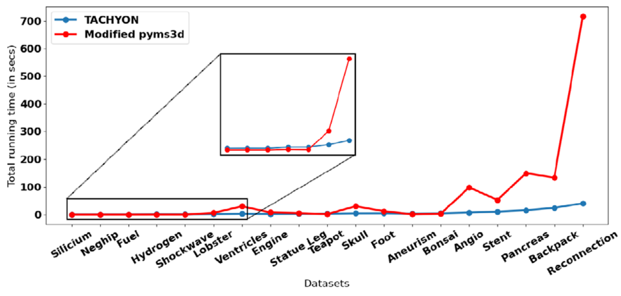


Figure 13: *Runtime comparison between* tachyon *and pyms3d [SN17].* tachyon *performs consistently better, often several orders of magnitude faster.*

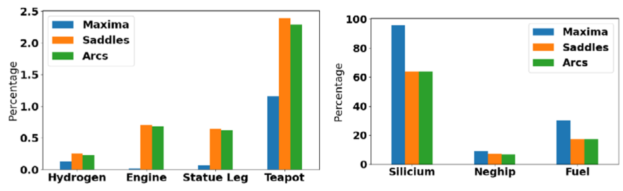


Figure 14: *Statistics on the fraction of extremum graph nodes and arcs that survive post simplification. The number of surviving extrema, saddles and arcs after applying the three simplification operations in sequence are shown as a fraction of the corresponding numbers in the unsimplified extremum graph.*

Across all results shown in Figure 14, we observe that the number of arcs is a little over twice the number of saddles, although these absolute numbers are not depicted in the graph plot. This is expected because all saddles have at least two outgoing arcs.

7. Conclusions and Future Work

We have developed TACHYON, a dimension independent, scalable extremum graph computation library that also supports a hybrid GPU–CPU mode to process large datasets. This software library will be released in the public domain for use by the community. We also describe three methods for simplifying the extremum graph with the aim of identifying and removing noise, resulting in a clutter free graph that is more amenable for further analysis. Computational experiments demonstrate good running times and scaling properties.

Future work includes incorporating support for generic non-uniform grids and an extension to a distributed setting where the dataset does not fit in system memory. For the latter problem, an approach similar to the hybrid GPU–CPU mode where different processors work on different blocks of the same dataset may work but the challenge is to reduce the communication necessary to consolidate the output.

Acknowledgements

This work is partially supported by a Swarnajayanti Fellowship from the Department of Science and Technology, India (DST/SJF/ETA-02/2015-16), an SERB Grant CRG/2021/00527, MoE Govt. of India, and a Mindtree Chair research grant.

References

[Ban70] BANCHOFF T. F.: Critical points and curvature for embedded polyhedral surfaces. *The American Mathematical Monthly 77* (1970), 475–485.

[BGL*18] BHATIA H., GYULASSY A. G., LORDI V., PASK J. E., PASCUCCI V., BREMER P.-T.: TopoMS: Comprehensive topological exploration for molecular and condensed-matter systems. *Journal of Computational Chemistry 39*, 16 (2018), 936–952.

[BHEP04] BREMER P.-T., HAMANN B., EDELSBRUNNER H., PASCUCCI V.: A topological hierarchy for functions on triangulated surfaces. *IEEE Transactions on Visualization and Computer Graphics 10*, 4 (2004), 385–396.

[CFST20] CARR H., FUJISHIRO I., SADLO F., TAKAHASHI S.: *Topological Methods in Data Analysis and Visualization V*. Springer, Cham, 2020.

[CLB11] CORREA C., LINDSTROM P., BREMER P.-T.: Topological spines: A structure-preserving visual representation of scalar fields. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (2011), 1842–1851.

[CMS06] CARR H., MOLLER T., SNOEYINK J.: Artifacts caused by simplicial subdivision. *IEEE Transactions on Visualization and Computer Graphics 12*, 2 (2006), 231–242.

[CWS*19] CARR H. A., WEBER G. H., SEWELL C. M., RÜBEL O., FASEL P., AHRENS J. P.: Scalable contour tree computation by data parallel peak pruning. *IEEE Transactions on Visualization and Computer Graphics 27*, 4 (2019), 2437–2454.

[DFFIM15] DE FLORIANI L., FUGACCI U., IURICICH F., MAGILLO P.: Morse complexes for shape segmentation and homological analysis: Discrete models and algorithms. *Computer Graphics Forum 34*, 2 (2015), 761–785.

[DFRS14] DELGADO-FRIEDRICHS O., ROBINS V., SHEPPARD A.: Skeletonization and partitioning of digital images using discrete Morse theory. *IEEE Transactions on Pattern Analysis and Machine Intelligence 37*, 3 (2014), 654–666.

[EH09] EDELSBRUNNER H., HARER J.: *Computational Topology: An Introduction*. American Mathematical Society, Providence, Rhode Island, 2009.

[EHNP03] EDELSBRUNNER H., HARER J., NATARAJAN V., PASCUCCI V.: Morse-Smale complexes for piecewise linear 3-manifolds. In *Proceedings of the Nineteenth Annual Symposium on Computational Geometry* (2003), pp. 361–370.

[ELZ02] EDELSBRUNNER H., LETSCHER D., ZOMORODIAN A.: Topological persistence and simplification. *Discrete & Computational Geometry 28*, 4 (2002), 511–533.

[FIDF19] FUGACCI U., IURICICH F., DE FLORIANI L.: Computing discrete Morse complexes from simplicial complexes. *Graphical Models 103*, (2019), 101023.

[GBHP08] GYULASSY A., BREMER P.-T., HAMANN B., PASCUCCI V.: A practical approach to Morse-Smale complex computation: Scalability and generality. *IEEE Transactions on Visualization and Computer Graphics 14*, 6 (2008), 1619–1626. https://doi.org/10.1109/TVCG.2008.110

[GBP12] GYULASSY A., BREMER P.-T., PASCUCCI V.: Computing Morse-Smale complexes with accurate geometry. *IEEE Transactions on Visualization and Computer Graphics 18*, 12 (2012), 2014–2022.

[GBP18] GYULASSY A., BREMER P.-T., PASCUCCI V.: Shared-memory parallel computation of Morse-Smale complexes with improved accuracy. *IEEE Transactions on Visualization and Computer Graphics 25*, 1 (2018), 1183–1192.

[GGL*14] GYULASSY A., GÜNTHER D., LEVINE J. A., TIERNY J., PASCUCCI V.: Conforming Morse-Smale complexes. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (2014), 2595–2603.

[GNP*06] GYULASSY A., NATARAJAN V., PASCUCCI V., BREMER P.-T., SOCIETY C., HAMANN B.: A topological approach to simplification of three-dimensional scalar functions. *IEEE Transactions on Visualization and Computer Graphics 12*, (Aug. 2006), 474–84. https://doi.org/10.1109/TVCG.2006.57

[GPPR12] GYULASSY A., PASCUCCI V., PETERKA T., ROSS R.: The parallel computation of Morse-Smale complexes. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium* (2012), IEEE, pp. 484–495.

[GRWH12] GÜNTHER D., REININGHAUS J., WAGNER H., HOTZ I.: Efficient computation of 3D Morse–Smale complexes and persistent homology using discrete Morse theory. *The Visual Computer 28*, 10 (2012), 959–969.

[HLH*16] HEINE C., LEITTE H., HLAWITSCHKA M., IURICICH F., DE FLORIANI L., SCHEUERMANN G., HAGEN H., GARTH C.: A survey of topology-based methods in visualization. *Computer Graphics Forum 35*, 3 (2016), 643–667.

[HMMN14] HARKER S., MISCHAIKOW K., MROZEK M., NANDA V.: Discrete Morse theoretic algorithms for computing homology of complexes and maps. *Foundations of Computational Mathematics 14*, 1 (2014), 151–184.

[HMST21] HOTZ I., MASOOD T. B., SADLO F., TIERNY J.: *Topological Methods in Data Analysis and Visualization VI*. Springer, Cham, 2021.

[NTN15] NARAYANAN V., THOMAS D. M., NATARAJAN V.: Distance between extremum graphs. In *Proceedings of the 2015 IEEE Pacific Visualization Symposium (PacificVis)* (2015), IEEE, pp. 263–270.

[Ope22] KLACANSKY P.: Open scientific visualization datasets. https://klacansky.com/open-scivis-datasets/ (2022). Accessed 01-July-2022.

[PRG*11] PETERKA T., ROSS R., GYULASSY A., PASCUCCI V., KENDALL W., SHEN H.-W., LEE T.-Y., CHAUDHURI A.: Scalable parallel building blocks for custom data analysis. In *Proceedings of the 2011 IEEE Symposium on Large Data Analysis and Visualization* (2011), IEEE, pp. 105–112.

[RWS11] ROBINS V., WOOD P. J., SHEPPARD A. P.: Theory and algorithms for constructing discrete Morse complexes from grayscale digital images. *IEEE Transactions on Pattern Analysis and Machine Intelligence 33*, 8 (2011), 1646–1658.

[Sch81] SCHWEFEL H.-P.: *Numerical Optimization of Computer Models*. John Wiley & Sons, Inc., Chichester, 1981.

[SN12] SHIVASHANKAR N., NATARAJAN V.: Parallel computation of 3D Morse-Smale complexes. *Computer Graphics Forum 31*, 3 (2012), 965–974.

[SN17] SHIVASHANKAR N., NATARAJAN V.: Efficient software for programmable visual analysis using Morse-Smale complexes. In *Topological Methods in Data Analysis and Visualization*. Springer, Cham (2017), pp. 317–331. https://doi.org/10.1007/978-3-319-44684-4_19

[SPN20] SUBHASH V., PANDEY K., NATARAJAN V.: GPU parallel computation of Morse-Smale complexes. In *Proceedings of the IEEE Visualization Conference, IEEE VIS 2020 - Short Papers* (2020), IEEE, pp. 36–40. https://doi.org/10.1109/VIS47514.2020.00014

[SPN22] SUBHASH V., PANDEY K., NATARAJAN V.: A GPU parallel algorithm for computing Morse-Smale complexes. *IEEE Transactions on Visualization and Computer Graphics* (2022), 1. https://doi.org/10.1109/TVCG.2022.3174769

[SSN11] SHIVASHANKAR N., SENTHILNATHAN M., NATARAJAN V.: Parallel computation of 2D Morse-Smale complexes. *IEEE Transactions on Visualization and Computer Graphics 18*, 10 (2011), 1757–1770.

[TFL*17] TIERNY J., FAVELIER G., LEVINE J. A., GUEUNET C., MICHAUX M.: The Topology ToolKit. *IEEE Transactions on Visualization and Computer Graphics* (2017). https://topology-tool-kit.github.io/

[TN13] THOMAS D. M., NATARAJAN V.: Detecting symmetry in scalar fields using augmented extremum graphs. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (2013), 2663–2672.