Distributed Asynchronous Union-Find for Scalable Feature Tracking

Jiayi Xu[*1] Hanqi Guo[†2] Han-Wei Shen[‡1] Mukund Raj[§2] Xueqiao Xu[¶3] Xueyun Wang[‖3]
Zhehui Wang[**4] Tom Peterka[††2]

[1] The Ohio State University
[2] Argonne National Laboratory
[3] Lawrence Livermore National Laboratory
[4] Los Alamos National Laboratory

ABSTRACT

Feature tracking and the visualizations of the resulting trajectories make it possible to derive insights from scientific data and thus reduce the amount of data to be stored. However, existing serial methods are not scalable enough to handle fast increasing data size. In this paper, we tackle the problem of distributed parallelism for feature tracking and visualization by introducing a scalable, asynchronous union-find algorithm. We show that asynchronous communication can improve the scalability of distributed union-find operation in comparison to synchronous communication, as seen in existing methods. In the proposed feature tracking pipeline, we first construct and partition a high-dimensional mesh that incorporates both space and time. Then, the trajectories of features are built distributively across parallel processes, and trajectory pieces are merged asynchronously by using our distributed union-find implementation. Results demonstrate the scalability of tracking critical points on exploding wire experimental data and tracking super level sets on BOUT++ fusion plasma simulations.

Index Terms: Human-centered computing—Visualization—Visualization application domains—Scientific visualization;

1 INTRODUCTION

During this big data era, one of the major challenges in current and future supercomputers is limited I/O bandwidth and computer storage paired with a high data generation rate. For in situ visualization and data analysis, an effective solution of the big data challenge is to store feature trajectories as opposed to raw data for two reasons: (1) visualizations of feature trajectories lead to various scientific insights, and (2) features usually have smaller sizes than the raw data. However, most existing feature tracking algorithms are not scalable; hence, they are hard to be used to handle large data and track features in situ. A challenging problem for scalable feature tracking is how to group related features efficiently (i.e., the Connected Component Labeling (CCL) problem) for distributed memory.

Identifying connected components is useful in many areas. For example, in mesh-based data sets seen in scientific applications, connected components provide a way to highlight the connectivity of objects of interest that exist on the mesh [6]. Another application of connected components is for determining trajectories of moving objects from a temporal sequence of images in order to understand the characteristics of the object [22]. Over the years, many methods have been introduced to determine connected components for data in local [18, 34] as well as distributed settings [3, 10]. Distributed data often present a challenge since a single connected component can span multiple processors (or cores). As a result, communication between processes becomes necessary to resolve such instances as a single connected component, and the associated communication patterns become a key factor affecting the efficiency of methods tackling such data.

State-of-the-art methods to compute connected components in distributed datasets rely on *synchronous* communication for resolving parts of individual connected components that span over the memory of multiple processes. For example, Harrison et al. [10] proposed a method that proceeds in the following phases: identifying local connected components, identifying components that span multiple processes, merging components that span multiple processes, and producing consistent component labels across all processes; these phases broadly correspond to embarrassingly parallel, local, global all-to-all and local communication patterns respectively. The global all-to-all operation required for merging is a potential performance bottleneck, particularly in case of data where connected components span over many processes [5, 36]. Furthermore, their load balancing approach involves the division of the domain extents rather than the features; this can lead to an imbalance in cases where the features are not uniformly distributed across the domain.

In order to address the above limitations, we introduce a novel distributed, a union-find method that can operate in both synchronous as well as asynchronous way. We show that our asynchronous union-find algorithm leads to better performance and scaling in comparison to synchronous union-find for computing connected components on a range of distributed data sets. Our method is general and can work with any kind of structured or unstructured mesh. In this paper, we demonstrate the effectiveness of our method by using it to construct connected components for tracking two kinds of features: critical points and super level sets. Additionally, since the number of mesh elements or *features* that compose the connected components can have a non-uniform distribution across the global domain, we also use a kd-tree based approach to divide the features across all available processes evenly. To summarize, we enumerate our contributions in this paper as follows:

- A novel asynchronous, distributed union-find algorithm that compares favorably to state-of-the-art approaches with regard to scaling characteristics.
- A kd-tree based load balancing for the proposed union-find algorithm that partitions the domain based on feature distribution.
- Demonstration of our method with synthetic and real scientific data sets and a discussion of the performance characteristics of our method in light of the results.

[*]e-mail: xu.2205@osu.edu
[†]e-mail: hguo@anl.gov
[‡]e-mail: shen.94@osu.edu
[§]e-mail: mraj@anl.gov
[¶]e-mail: xu2@llnl.gov
[‖]e-mail: wxy2015@pku.edu.cn
[**]e-mail: zwang@lanl.gov
[††]e-mail: tpeterka@mcs.anl.gov

The remainder of the paper is organized as follows. Following a brief survey of related works (Section 2), we present a high-level overview (Section 3) and a detailed description of our method (Section 4). Next, we present the results of our experiments on synthetic data (Section 5) and demonstrate our method on two real data sets (Section 6). Finally, we discuss a few nuances of our method including limitations (Section 7).

2 Related Works

Here we briefly review existing literature related to various areas connected to the proposed method in this paper.

2.1 Feature Tracking

In time-varying datasets, identifying and tracking features of interest is important for making useful inferences in computer vision and scientific applications. In computer vision, the features typically involve objects in a sequence of real-world images. Identification and tracking of such image-based features are found usage in domains ranging from traffic management [28] to health care [1]. Many methods have been introduced for tracking features across a sequence of images, and this area continues to be an active area of research [22, 24].

Another key area of import with regard to feature tracking involves in-situ analysis in scientific applications. A range of in-situ feature tracking algorithms appear in literature such as halo tracking in cosmology [29], flame tracking in combustion sciences [15], blob tracking in plasma fusion science [35, 35], and vortex tracking in superconductivity [9, 16, 23]. A common challenge in scientific applications is the large size of data that does not fit in a single core and is usually split and processed in parallel using distributed memory machines.

The diverse nature of applications, data, and features has led to a variety of approaches that are used to track the features. An upcoming approach involves the use of deep learning techniques to identify and track features [25]. Other approaches for tracking include statistical approaches [32] as well as topology based approaches, such as connected components, which are popular due to characteristics such as robustness and provability of correctness [30]. The necessity and popularity of feature tracking in real applications have motivated the development of supporting the functionality in various specialized libraries [2, 8].

2.2 Connected Component Labeling

Our approach for feature tracking requires the determination of connected components in a spacetime mesh that is stored in a distributed memory setting. Connected component labeling involves assigning a unique label to *connected* elements in a set. Several methods to determine connected components have been proposed with applications in domains such as image analysis, computer vision, and social networks. The methods in literature can be broadly organized into the following classes based on the approach taken [12]: label-propagation [18] or label-equivalence resolving [11].

Label propagation methods rely on identifying an unlabeled element, assigning a new label, and propagating the same label to all connected elements. In this class of methods, the order in which unlabeled elements are traversed can be irregular depending upon the connectivity of connected components; this makes such methods unsuitable for parallel implementations. On the other hand, label-equivalence methods operate by first assigning a temporary label in a first pass, which is followed by a step that determines final labels of each element by *merging* equivalent labels or elements that are connected. Since this class of methods typically read the elements in a regular, predetermined order in the first pass, they are a popular candidate for parallelization [10].

There are different variations, even within the class of label-equivalence resolving methods based on how the resolving is carried out. One approach developed for image data involves alternatively traversing the image forward/ backward passes and updating the labels assigned to current pixel and neighbors until no further changes are needed during a pass [26]. The other approach, which also forms the basis of our proposed method, uses the *union-find* data structure [7]. The union-find data structure provides a way to organize and update the connected components efficiently [34]. It has been used for identifying connected components in graph-based data [31] as well as for data stored in a distributed setting [10].

Harrison et al. [10] introduce a data-parallel algorithm to determine connected components with a focus on mesh-based data stored in a distributed memory architecture, as is typical for processing large scientific datasets. Their method uses union-find to serially identify connected components at each process locally as well as to identify components that span multiple processes. As a prepossessing step, a binary search partition (BSP) tree is used to redistribute data with the aim of achieving a uniform workload across processes. While Harrison et al. [10] require synchronization between processes for resolving label equivalence, our method is capable of achieving this outcome using asynchronous (in addition to synchronous) communications. Connected component finding methods have also been developed for specialized data types such as those involving hierarchical meshes [37]. We refer the interested reader to the following surveys for more information on the connected component algorithms and their evaluation in distributed memory systems [12, 13].

2.3 Union-Find

The union-find (or, disjoint-set) is a data structure that supports queries for existing disjoint sets, and is backed by algorithms for efficient unions of disjoint sets.

2.3.1 Distributed Synchronous Union-Find

In this paper, we construct union-find data structures to deal with the problem that elements of disjoint sets are distributed amongst processors. Researchers have made many efforts. Cybenko et al. [4] probably was the first to present an algorithm of union-find for distributed-memory parallelism; however, their algorithm [4] duplicates all elements on each processor and distributes edges amongst processors, which is different from our problem.

Other published distributed union-find methods [10, 13, 17] can handle distributed elements. These existing algorithms separate the local computation and global communication, and use synchronous communications.

The algorithm of Manne et al. [17] has two stages: (1) processors complete the unions for local elements and (2) merge union-find data structures across processors to get the final result.

Different from Manne et al. [17] that used the ID of the root element as the label of each set, Harrison et al. [10] used a unique label (e.g., number or letter) for each set. Hence, the algorithm of Harrison et al. [10] has four stages for the construction of union-find. (1) Disjoints sets of local elements are identified, and each local disjoint set is assigned a locally unique label. (2) Global synchronization is performed to assign a globally unique number to each local disjoint set. (3) Union-find data structures across processors are merged to obtain global disjoint sets. (4) Each global disjoint set is assigned a unique label.

Also, different from the approaches of both Manne et al. [17] and Harrison et al. [10] that split the stage for local computation and the stage for merging data structures across processors, Iverson et al. [13] proposed a round-by-round approach to fuse local computation and global communications to accomplish unions of disjoint sets. For each round of computation, processes complete parts of the local computation. Then, by using synchronous communications, processes exchange messages to perform updates for union-find data

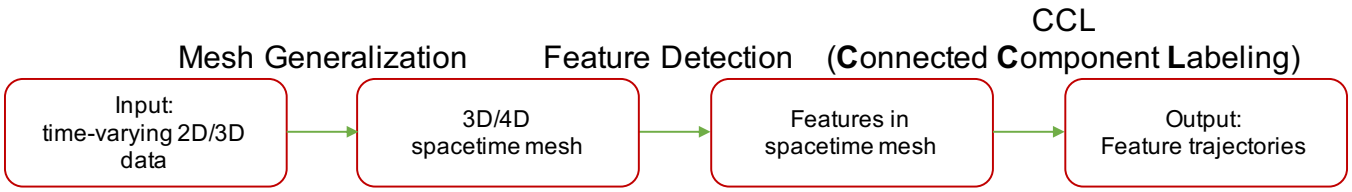


Figure 1: This figure shows the pipeline of feature tracking. Given the input time-varying data, we first generalize it into a spacetime mesh. We detect features on the spacetime mesh and perform connected component labeling to acquire trajectories of features.

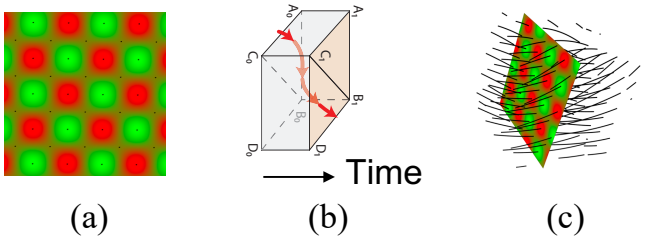


Figure 2: The figure illustrates the idea of tracking critical points. In (a), black dots are detected local maxima, local minima, and saddle points at a time slice. In (b), we relate critical points that are in the same spacetime mesh cell. (c) displays the resulting trajectories of the critical points.

structures across processors; however, the synchronous communications may lead to the waste of computational resources.

Our method is different from previous methods [10, 13, 17] in two aspects. First, by using asynchronous communications, our method overlaps local computations and communications and acquires good scalability on scientific datasets. Second, when processors have unbalanced numbers of elements, a kd-tree based load balancing method [19] is applied to obtain better scalability.

3 Feature Tracking Pipeline

Fig. 1 displays the pipeline for tracking local features. To parallelize methods of the pipeline, in our work, the mesh generation and feature detection are implemented by embarrassing parallelism. Specifically, we first decompose the input mesh evenly into regular data blocks, and assign one data block to each processor. Meantime, each processor generalizes the input of 2D or 3D time-varying data into a 3D or 4D spacetime mesh, respectively. Note that, in order to relate features across processors, each data block is included one layer of ghost cells for all dimensions in addition to the core cells.

Next, each processor detects features that are in both its core cells and ghost cells; but only the features in core cells belong to the processor. For tracking critical points, given scalar values at grid points, we locate the positions of critical points of each 2D mesh face by using the inverse interpolation. For tracking super level sets, we record which grid points have values that are larger than a specified threshold.

After the feature detection, we consider the features that share the same mesh cell to have local connections, and relate the mesh elements containing these features by edges. For example, for tracking critical points in Fig. 2b, when two critical points exist on faces of the same spacetime cell, we associate the faces by an edge. For tracking super level sets, if any two grid points of the same mesh cell have values that are larger than the threshold, we relate the two grid points. Till now, each processor only works for its data block, and no communications among processors are needed.

After that, the distributed CCL is performed to group features based on the edges and requires communications of processors. To implement the distributed CCL, we propose an asynchronous union-find algorithm. After performing the distributed union-find, features of the same trajectory are pointed to a 'root' feature, which has the smallest ID.

Afterwards, to obtain trajectories, we perform an all-to-all operation to gather all features with the same root to the processor of their root, and transform the set of features to a geometric trajectory such as a line segment (e.g., Fig. 2c) for tracking critical points and a connected volume for tracking super level sets. Finally, we output the resulting trajectories for analysis in situ or into files for post-analysis.

4 Distributed Union-Find

In our feature tracking pipeline, an important step is to acquire sets of connected mesh elements with features to obtain feature trajectories, which is solved by a new distributed union-find algorithm in this paper.

4.1 Data Structure

The union-find data structure maintains disjoint sets of elements. We assume that elements have unique IDs; also, their IDs are comparable, namely, there exists an ordered list for the IDs. Let U denote a collection of all elements; let S_i be a subset of U. Assuming we have k non-overlapping sets, we have $U = \bigcup_{i=1}^{k} S_i$; also, $S_i \cap S_j = \emptyset$ for any pairs of i and j as i is not equal to j.

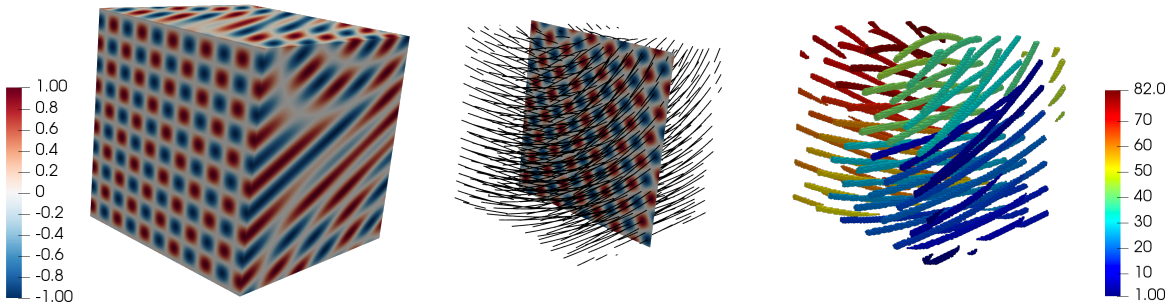
The elements in the same set S_i forms a tree structure, where the root is the element with the smallest ID within the set and represents the set. Each non-root element stores a pointer to its parent element; the parent element has a smaller ID than the element. Each root element points to itself.

4.2 Algorithm for Distributed Unions of Disjoint Sets

In this paper, we propose an algorithm that supports using asynchronous communications to perform unions of disjoint sets for elements that are distributed among processors (or, cores). Previous researchers [10, 13, 17] focused on using synchronous communications to accomplish distributed unions of disjoint sets. To overlap the time of computation and communication, we propose a distributed asynchronous algorithm for unions of the disjoint sets.

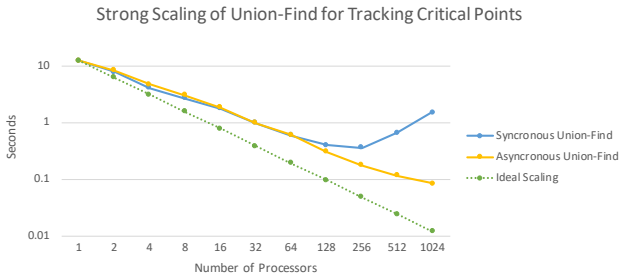
The input is consisting of two parts: (1) elements that are distributed among processors, and (2) edges between elements. We call *local elements* of each processor to be the elements that are assigned to the processor. Also, note that only one copy of each edge is stored; for an edge between two elements e_0 and e_1 that are in different processors, if $e_1 > e_0$, we store the edge in the processor of e_1; otherwise, we store the edge in the processor of e_0. In other words, we store each edge in one of its endpoints that has a larger ID.

The output is that each element is pointed to the root of its set. Namely, the output tree structure of each set has two layers, and elements except the root are pointed to the root.

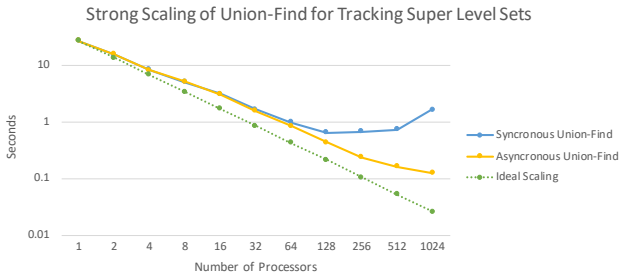


(a) Raw Volume (b) Trajectories of Critical Points (c) Trajectories of Super Level Sets

Figure 3: (a) displays raw scalar volume that ranges from -1 to 1 and has a 128^3 spacetime resolution. (b) displays trajectories of three types of critical points. (c) displays super level sets that are larger than the threshold 0.8. In (b), trajectories of the critical points are drawn by black lines. In (c), there are 82 connected components the super level sets totally, and each is assigned a unique hue.



(a) Critical Point



(b) Super Level Set

Figure 4: We compare the strong scaling between synchronous union-find and asynchronous union-find on synthetic data for (a) tracking critical points and (b) tracking super level sets. The scaling of synchronous union-find becomes worse than the asynchronous one when using no less than 128 cores.

4.2.1 Initialization

We initialize union-find data structures on processors. Each processor adds local elements into the union-find, and assigns the parents of local elements to be themselves initially. Each processor also adds edges of the local elements into the data structures; if an endpoint of a given edge is non-local (i.e., not in this processor), the processor records which the other processor the endpoint belongs.

4.2.2 Steps for Update of Union-Find Data Structure

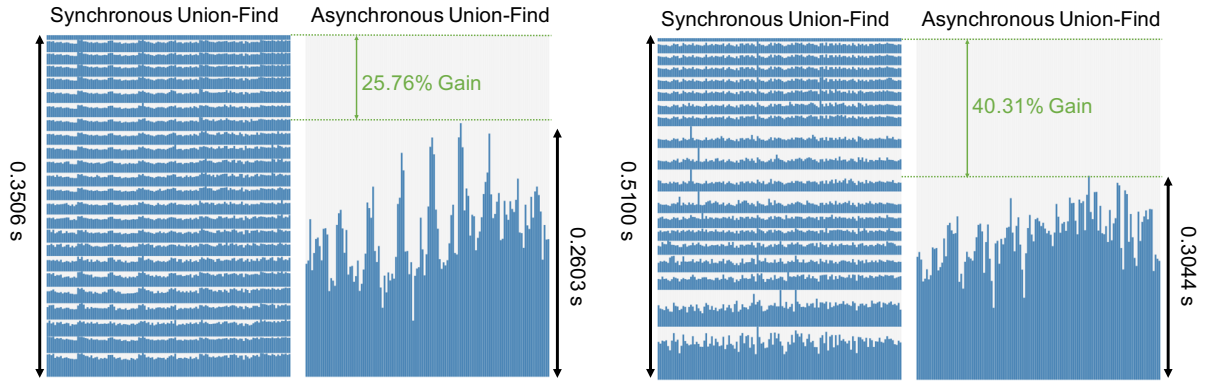
Each processor repeats updating its union-find data structure by the following rules round-by-round until termination conditions are reached. Let *temporary roots* be elements that currently have no parents when starting each round of updating. We define *hubs* of a processor as the local elements that have non-local parents; namely, their parents are in other processors. We use *ordinary elements* to represent the elements that are neither temporary roots nor hubs.

Update for ordinary elements. Each ordinary element asks its local parent about its grandparent. If its grandparent is local, the ordinary element diverts its parent pointer to its local grandparent; if its grandparent is non-local, which means its parent is a local hub, the ordinary element does **not** update its pointer for now.

Update for hubs. Given each local hub, it sends a message to the processor of its non-local parent to ask whether its parent has a parent, namely, whether the local hub has a grandparent. After the message has been sent, we record that the local hub has sent the message to avoid that the local hub sends multiple messages for the query of its grandparent. After the processor of its parent receives the message, its parent sends a feedback message to the local hub. If its parent is a temporary root, the feedback tells the local hub that its parent currently is a root, and to not send the query for grandparent again until its parent tells the local hub that the parent is no longer root. If its parent has a parent, namely, the local hub has a grandparent, its parent sends feedback that tells the local hub which is its grandparent, what is the processor that has its grandparent, and a flag indicates whether its parent knows its grandparent is a temporary root. If its parent knows its grandparent is a temporary root, the flag has a value of 'true'; otherwise, if its parent knows its grandparent is not a temporary root or does not has this information, the flag has a value of 'false'. When the local hub receives the feedback, it updates its pointer to its grandparent and records the processor ID of its grandparent; if the flag has the value of 'true', the processor of the local hub also records that its grandparent is a temporary root.

Update for temporary roots. For each temporary root, processors enumerate its stored edges to unite with a connected element that has a smaller ID than the temporary root. If multiple connected elements have smaller IDs, we select one based on the following rule. If there existing local connected elements, we select the one with the smallest ID from the local elements; otherwise, we select the one with the smallest ID from the non-local connected elements.

When a temporary root, e, is no longer a root, it has a new parent e_{parent}. e notifies its descendants that it is not root anymore, and tell them who is its new parent e_{parent}, the processor ID of e_{parent}, and whether it knows e_{parent} is a temporary root or not. Note that, if previously a child e_{child} of e sent messages to e for the query of grandparent, e responded that e was a temporary root; that time, we would record which processor of e_{child} sent the message to e. Now, we can send the message to the processors of such children to let them know that e is no longer a temporary root. Also, if the processor of e_{child} has told other processors that e is a temporary root before, the processor of e_{child} would record these other processors; as the



(a) Critical Point

(b) Super Level Set

Figure 5: The figures compare the computation time of different processors between synchronous and asynchronous union-find when using 128 cores; (a) is for tracking critical points, and (b) is for tracking super level sets. The x-axis of each chart has 128 intervals, and each interval is corresponding to a core; the y-axis encodes time. The left charts are for the synchronous union-find; each row is corresponding to a round of computation with synchronous communications. Due to the synchronous communications, processors with less computational tasks become idle after finishing their work and wait for ones with more work for each round. However, for asynchronous union-find, each core works with its own pace and makes better usage of its computational resources.

processor of e_{child} receives the message that e is not a temporary root, the processor of e_{child} broadcasts this message to these other processors. Note that, hence, each processor needs to record that which elements are no longer roots. Each processor stores a set of elements $set_{nonroot}$ that it knows they are **not** temporary roots anymore, and the other set of elements set_{root} that it knows they are temporary roots currently. When inserting an element to set_{root}, we ensure the element is not in $set_{nonroot}$.

4.2.3 Transfer of Edges

Certain non-root elements transfer all of their edges to their parent for each round of computation; these elements are:

- Any local hub that knows its parent is a temporary root.
- Any ordinary element whose parent is a local hub.

These elements first change one endpoint of their edges to their parents. Next, since we only store one copy of a given edge at one of its endpoints, the elements send the edges to either their parents or the other endpoints by the following rule. Without loss of generality, for example, an element e_0 owns an edge that connects it to the other element e_1; now, e_0 needs to transfer this edge to its parent e_p. First, e_0 makes this edge connect its parent e_p with e_1. Next, e_0 needs to decide to send this edge to e_p or the other endpoint e_1 of this edge. If $e_p > e_1$, we send this edge to its parent e_p; if $e_p < e_1$, we pass this edge to e_1. Note that, when e_p is equal to e_1, we do not need to send this edge.

Note that, an element e_0 may receive multiple edges that let it connect to the same element noted as e_1; we only save one copy of the edges. Also, if e_0 previously had an edge that connects to e_1 and e_0 now receive this kind of edge again, e_0 will not save this edge to avoid duplicates.

When an edge is sent across processors, the edge needs to contain the information of processor IDs of its endpoints.

4.2.4 Termination Conditions

Our algorithm supports both synchronous and asynchronous communications. For the synchronous algorithm, all processors synchronize to send and receive messages after each round of computation. The algorithm terminates with that there are no changes for all processors. For the asynchronous algorithm, processors send and receive messages asynchronously. Each processor performs local computation round-by-round at their own pace, and ends with two conditions: (1) there are no changes locally, and (2) there are no incoming messages for this processor. When all processors end, the communicative parts of the asynchronous algorithm terminate.

4.2.5 Finalization

After the termination of the communicative parts mentioned above, we have a tree structure for each disjoint set: the root of each disjoint set is the element with the smallest ID, and the root is pointed by its local children and certain hubs in other processors; these hubs are pointed by their local children. The tree structure is at most three layers. Hence, the final update is that, for each child of such local hubs, the child now asks its hub, which is the root (i.e., the hub's parent), and directly points to the root. Note that, this final update requires no communications. After the finalization, this algorithm terminates.

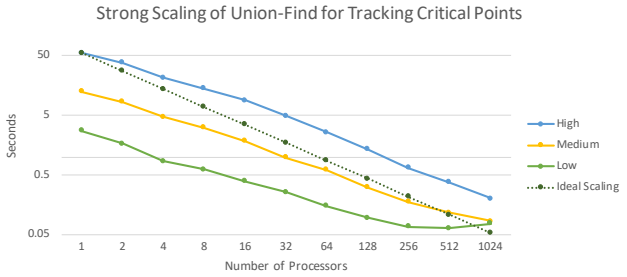
4.3 Load Balancing

Given detected features, we use a kd-tree based method [19] to decompose the mesh such that decomposed data blocks have similar numbers of features; each processor has one such data block. When the numbers of features on processors are unbalanced, the processors with few features usually finish their computation earlier than the ones with lots of features, and become idle afterwards, which is inefficient. The unbalanced numbers of features lead to the waist of computation resources. To remedy this, we balance the number of features of processors before we perform distributed unions of disjoint sets.

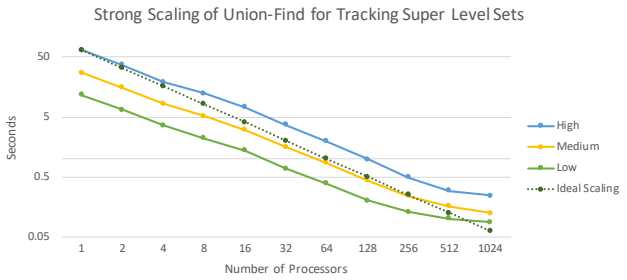
5 Experiments on Synthetic Data

We conduct multiple experiments to measure the performance of our approach for tracking features on synthetic data. The synthetic data are consisting of spiral and columnar shapes, e.g., Fig. 3a. We test two types of features: (1) critical points and (2) super level sets. For tracking critical points, we consider three types of critical points, (1) local maxima, (2) local minima, and (3) saddle points.

We run experiments on an HPC cluster, and each of the nodes in the cluster has 32 cores and 128 GB memory; the CPU is Intel Xeon

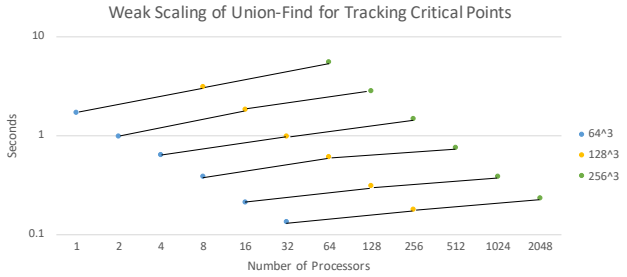


(a) Critical Point

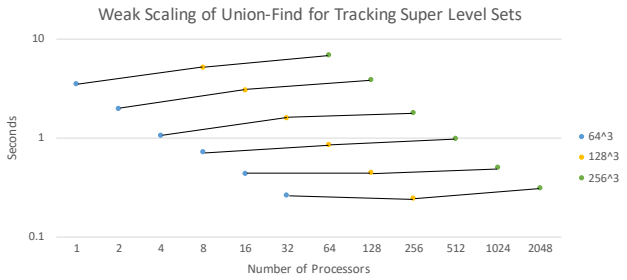


(b) Super Level Set

Figure 6: The figures show the strong scaling of union-find on synthetic data with three levels of feature density: high, medium, and low for (a) tracking critical points and (b) tracking super level sets. The synthetic data has a fixed 128^3 resolution and varying numbers of features. For (a) tracking critical points, the high feature density case has $377,620$ features in space and time, the medium one has $94,510$ features, and the low one has $23,326$ features. For (b) tracking super level sets, the high feature density case has $298,774$ features in space and time, the medium one has $140,365$ features, and the low one has $68,229$ features.



(a) Critical Point



(b) Super Level Set

Figure 7: The figures show the weak scaling of union-find on synthetic data with three data sizes: 64^3, 128^3, and 256^3 for (a) tracking critical points and (b) tracking super level sets.

E5-2695v4. Message Passing Interface (MPI) is supported by the Intel MPI library.

5.1 Experiment One: Comparison between Synchronous and Asynchronous Union-Find

We conduct the experiment to compare the performance of constructing union-find by using synchronous communications and by using asynchronous communications on a synthetic data with 128^3 resolution. From Fig. 4, by compared with the synchronous union-find, the asynchronous one is more scalable when using no less than 128 cores. To investigate why that happens, we list the time of all processors in Fig. 5 when using 128 cores. For synchronous union-find since the workload of different cores is unbalanced, when making synchronization after finishing one round of computation, the cores with less work need to wait for the cores with more work; that causes waste of computational resources. While, for asynchronous union-find, processors perform computation as their own paces since the processors communicate asynchronously, which leads to the gain of computational time.

5.2 Experiment Two: Effect of Feature Density

We hypothesize that, when the data has the same resolution, increasing feature densities lead to better strong scaling performance. The reasons are as follows. When there is higher feature density, processors need more time to complete the local computation; since we overlap the time of communications and local computations by using asynchronous communications, more local computational time covers more communication time. Hence, we may acquire a better strong scaling when the feature density is higher for a fixed data resolution. We compare the performance of asynchronous union-find on three levels of feature density for the two types of features on a synthetic data with 128^3 resolution yet different numbers of features. The results are shown in Fig. 6, and support our hypothesis.

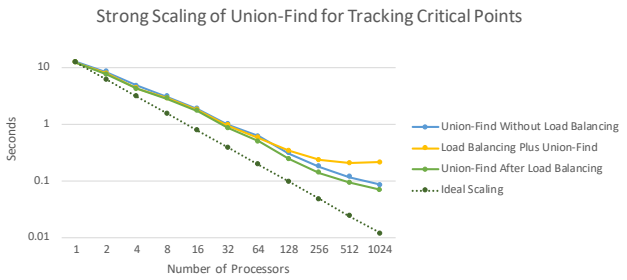
5.3 Experiment Three: Weak Scaling

We measure the weak scaling of the asynchronous union-find on three data resolutions 64^3, 128^3, 256^3 with similar feature densities. The results for tracking critical points and super level sets are displayed in Fig. 7.

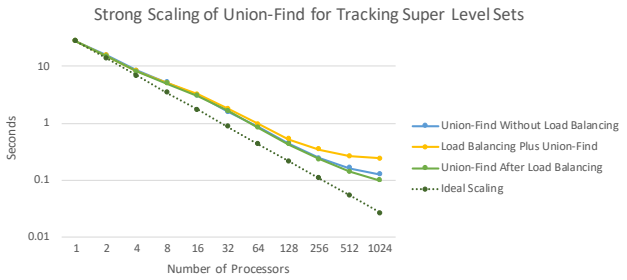
5.4 Experiment Four: Effect of Load Balancing

We hypothesize that, when processors have unbalanced features, the kd-tree based load balancing improves the strong scaling of asynchronous union-find; when processors have balanced features, the load balancing may introduce additional time overhead. In this experiment, we compare amongst the time of union-find without performing load balancing, the time of load balancing plus the time of union-find, and the time of union-find only after completing the load balancing.

We first conduct this experiment on a synthetic data of 128^3 resolution; the features of synthetic data are highly balanced on different processors. The results are shown in Fig. 8, where the strong scaling curves of union-find without or after load balancing are quite similar. Since the construction of kd-tree and transferring data for load balancing introduce more time overhead, the load-balancing plus union-find has a worse strong scaling than the union-find without load balancing. However, for many real-world scientific datasets, the features are quite unbalanced on different processors; hence, the load balancing pre-processing works well on these scientific datasets including the exploding wire experiments and BOUT++ fusion plasma simulations, as shown in Fig. 10a and Fig. 13a. In general, the results support our hypothesis.



(a) Critical Point



(b) Super Level Set

Figure 8: We show the strong scaling for three conditions: (1) time of union-find without performing load balancing, (2) time of load balancing plus time of union-find, and (3) time of union-find only after performing load balancing.

6 Applications and Results

We evaluate our approach on two scientific datasets: (1) exploding wire experimental data and (2) BOUT++ fusion plasma simulation data.

6.1 Exploding Wire Experiments

6.1.1 Application Background

Interests in understanding particles with different sizes in magnetic fusion grow recently in part due to advances in imaging and image analysis techniques. The interactions of plasmas with materials generate various particles including electrons, ions, atoms, hot and melton dust, or microparticles [33]. By using an exploding-wire apparatus, scientists can generate many high-temperature microparticles of different types. High-speed imaging cameras can capture the movement of these particles well and produce high-resolution images. By tracking these particles on the images help researchers understand properties of microparticles and high-temperature plasmas, study their interactions, and develop advanced techniques on, for example, plasma fueling. Moreover, the findings on experimental imaging data can aid scientists to enhance theoretical models for simulations.

6.1.2 Tracking Critical Points

We track particles on each frame of the exploding wire data; a frame is shown in Fig. 9a. The tested data is of 384×384 spatial resolution and 4745 timesteps. Particles, on the images of the exploding wire data, usually have higher intensities locally by compared with the background such as Fig. 9a; hence, we model particles as local maximum points on each frame of the temporal images. Then, we identify the movement of particles by tracking the detected local maximum points on the time-varying 2D images. This exploding wire data contains $3,197,333$ features and $3,186,046$ edges totally; the average edge degree of features is 1.99. The resulting trajectories are shown in Fig. 9b, which pass through the particles on the image.

Fig. 10 shows the strong scaling performance of our approach for tracking the critical points on the exploding wire experiments. We obtain 19.16% strong scaling efficiency for the total time of the load balancing and the distributed union-find when using 512 processors; we also acquire 63.95% strong scaling efficiency for the whole tracking critical points on this dataset. Fig. 11 displays the time percentage breakdown of all steps of the whole pipeline.

6.2 BOUT++ Fusion Plasma Simulations

6.2.1 Application Background

Fusion energy has attracted interest for decades as a promising candidate for fossil energy substitutes in the future. Tokamak, a kind of torus device with a strong helical magnetic field, is the mainstream fusion reactor to confine plasma in order to achieve fusion energy production magnetically. The turbulent transport from the edge plasma usually takes the ubiquitous form of filaments, defined as density-enhancement coherent structures, also referred to as blobs. As blobs moving radially outwards, they carry a large amount of heat and particles to the first wall of the device and may cause great loss of plasma and serious damage to the wall. The blob movement has been, therefore, subject to intensive research [14, 20, 27]. BOUT++ is a three-dimensional electromagnetic fluid code that provides a flexible framework to study the edge physics in tokamaks [5, 36]. By self-consistently solving fluid equations, BOUT++ outputs the spatial-temporal evolution of plasma variables. In this work, we present the ion density fluctuation data from the BOUT++ simulation of a future fusion reactor named ITER. By applying feature tracking, we obtain the trajectories of blobs that are observed in the simulation, which helps the plasma scientists to gain more insight on blob propagation and to investigate further the transport level contributed by blobs.

6.2.2 Tracking Super Level Sets

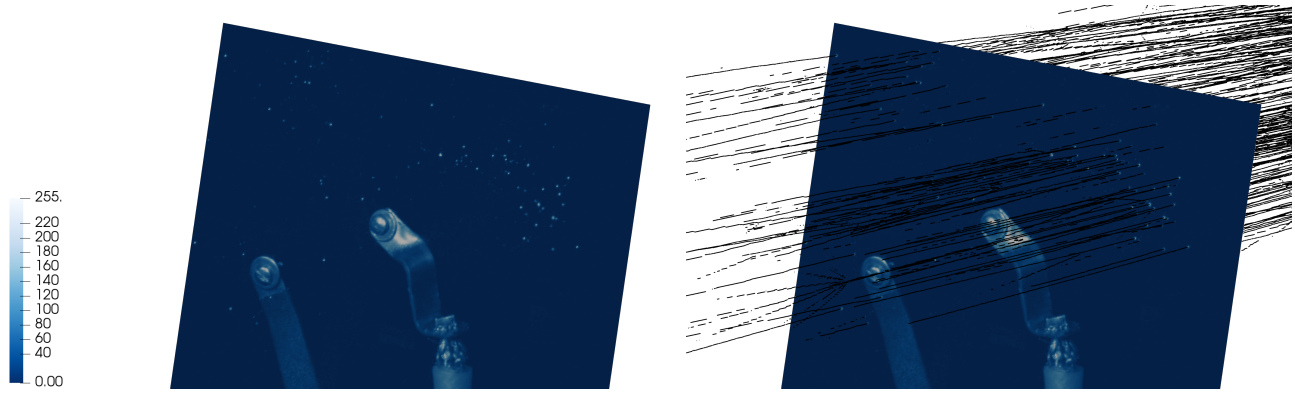
We track blobs on a 2D separatrix slice of the 3D torus simulation domain; a separatrix slice is shown in Fig. 12a. The tested data is of 425×880 spatial resolution and 701 timesteps. Blobs usually are the regions with high ion density in the scalar field. Hence, following the work [21], we model blobs at a timestep as the regions that have densities that are larger than 2.5 standard deviation than the average density; we standardize the density field at each timestep, and acquire such regions by extracting the super level sets that are larger than 2.5. Afterwards, we track the movement of blobs by tracking the detected super level sets of different timesteps. This BOUT++ fusion plasma data contains $1,708,341$ features and $10,093,696$ edges totally; the average edge degree of features is 11.82. The resulting trajectories are shown in Fig. 12bc.

Fig. 13 shows the strong scaling performance of our approach for tracking the super level sets on the BOUT++ fusion plasma simulations. We obtain 20.20% strong scaling efficiency for the total time of the load balancing and the distributed union-find when using 512 processors; the strong scaling efficiency for the whole super level sets tracking on this dataset is 5.77%, which is explained in Sect. 7. Fig. 14 displays the time percentage breakdown of all steps of the whole pipeline.

7 Discussions

A limitation for the feature type. Currently, the feature detection of our distributed feature tracking pipeline only supports local features since we only apply embarrassing parallelism for feature detection; the detection of a global feature may require communications for the detection of features.

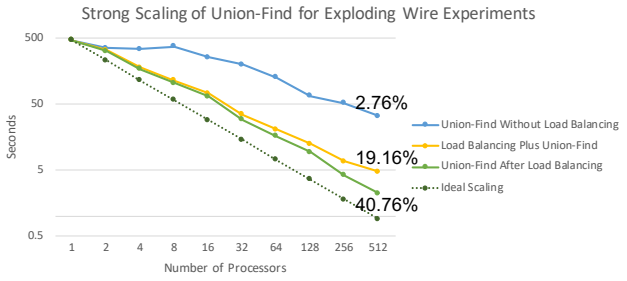
A limitation caused by ghost cells. For our approach, we require each processor to include ghost cells of one additional mesh layer to relate features across processors. However, when the number



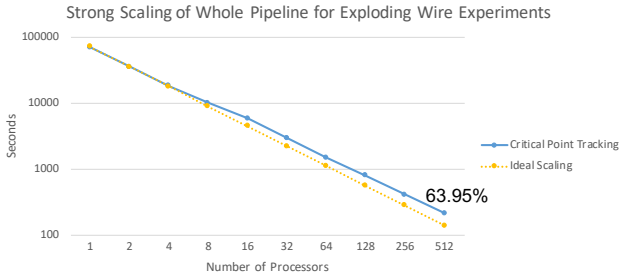
(a) Raw Image

(b) Trajectories of Critical Points

Figure 9: The figures show an example of tracking critical points on exploding wire experimental data; the intensity value ranges from 0 to 255. (a) shows one frame of the time-varying images. (b) shows the generated trajectories by black lines that pass through the particles on the frame.

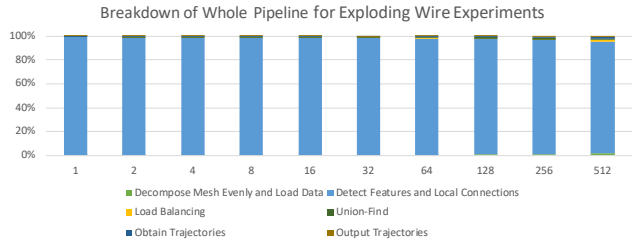


(a) Union-Find

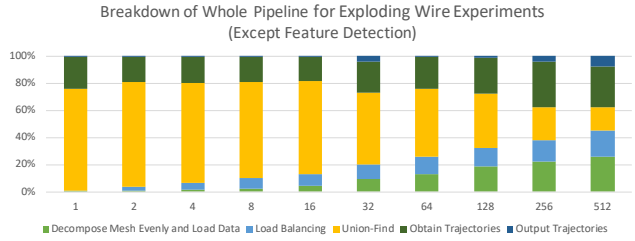


(b) Whole Pipeline

Figure 10: The figures show the strong scaling performance of our approach for exploding wire experiments. (a) shows the strong scaling of the union-find. (b) shows the strong scaling of the whole pipeline for tracking critical points. The labeled percentages are the scaling efficiencies of different methods when using 512 cores.



(a) All Steps of Pipeline



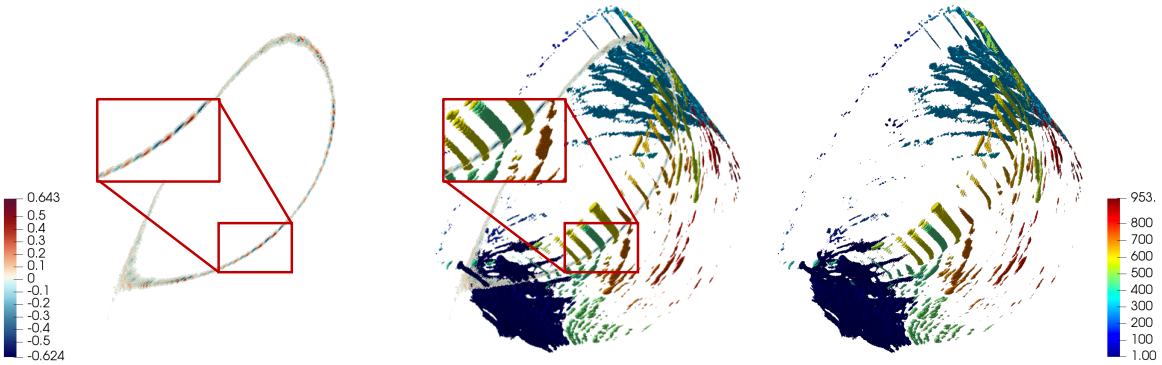
(b) All Steps of Pipeline Except Feature Detection

Figure 11: (a) shows the percentage breakdown of time for all steps of the whole pipeline when tracking critical points on exploding wire experimental data. In (b), we exclude feature detection and show others.

of processors that are used increases, the ratio of ghost cells to core cells increases as well, which hampers the scaling efficiency.

A limitation for the load balancing. The load balancing works well when features are unbalanced on processors; however, when features are balanced, the load balancing introduces additional time overhead. Currently, we manually decide whether to use load balancing or not. In the future, we may devise a metric to measure the degree of unbalanced features after the detection of features, and use the metric to determine whether to perform the load balancing or not afterwards.

Discussions for the percentage breakdown and the strong scaling performance of applications. The percentage breakdown in Fig. 11 and Fig. 14 indicate the time percentages of different steps within the whole pipeline when we use different numbers of cores. As shown in Fig. 11a and Fig. 14a, the detection of features usually occupies the most of the time. To display other steps clearly, we show the percentage breakdown except the detection of features in Fig. 11b and Fig. 14b. In Fig. 11b and Fig. 14b, as compared with other steps, the time percentage of the asynchronous union-find decreases as we use more cores, which shows that the asynchronous union-find algorithm has a good scaling. A limitation is that, in Fig. 14b, the step, obtaining trajectories, has a bad scaling, which hampers the scaling of the whole pipeline in Fig. 13b; the reason is that to acquire trajectories, we need to gather distributed elements of the same sets to their roots which is an all-to-all communication and is hard to scale well when each set has a large number of elements.



(a) Raw Density Field (b) Trajectories of Super Level Sets (c) Labeled Connected Components

Figure 12: The figures show an example of tracking super level sets on BOUT++ fusion plasma simulation data. (a) shows the 2D density field at a timestep; blobs are treated as regions with high densities. (b) shows the detected super level sets which pass through the high-density regions. In (c), there are 953 connected components of the super level sets in total, and each is assigned a unique hue.

8 Conclusion

In this paper, we present a novel asynchronous union-find algorithm for distributed-memory parallelism; when elements are unbalanced on processors, a kd-tree based load balancing is applied and improves the scalability. We incorporate the asynchronous union-find algorithm into a distributed feature tracking pipeline and evaluate its performance on synthetic data and two real-world scientific datasets including the exploding wire experimental data and the BOUT++ fusion plasm simulation data. Our experiments demonstrate that our algorithm has good scaling characteristics when used to track scientific features including critical points and super level sets.

References

[1] W. Bai, X. Zhou, J. Zhu, L. Ji, and S. T. Wong. Tracking of migrating glioma cells in feature space. In *2007 4th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pp. 272–275. IEEE, 2007.

[2] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[3] L. Buš and P. Tvrdík. A parallel algorithm for connected components on distributed memory machines. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pp. 280–287. Springer, 2001.

[4] G. Cybenko, T. G. Allen, and J. Polito. Practical parallel union-find algorithms for transitive closure and clustering. *International journal of parallel programming*, 17(5):403–423, 1988. doi: 10.1007/BF01383882

[5] B. Dudson, M. Umansky, X. Xu, P. Snyder, and H. Wilson. Bout++: A framework for parallel plasma fluid simulations. *Computer Physics Communications*, 180(9):1467 – 1480, 2009. doi: 10.1016/j.cpc.2009.03.008

[6] K. P. Gaither, H. Childs, K. W. Schulz, C. Harrison, W. Barth, D. Donzis, and P.-K. Yeung. Visual analytics for finding critical structures in massive time-varying turbulent-flow simulations. *IEEE computer graphics and applications*, 32(4):34–45, 2012.

[7] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys (CSUR)*, 23(3):319–344, 1991.

[8] H. Guo. Ftk: Feature tracking kit, Oct. 2019. `https://github.com/hguo/ftk`.

[9] H. Guo, T. Peterka, and A. Glatz. In situ magnetic flux vortex visualization in time-dependent ginzburg-landau superconductor simulations. In *2017 IEEE Pacific Visualization Symposium (PacificVis)*, pp. 71–80. IEEE, 2017.

[10] C. Harrison, J. Weiler, R. Bleile, K. Gaither, and H. Childs. A distributed-memory algorithm for connected components labeling of simulation data. In *Topological and Statistical Methods for Complex Data*, pp. 3–19. 2015. doi: 10.1007/978-3-662-44900-4_1

[11] L. He, Y. Chao, K. Suzuki, and K. Wu. Fast connected-component labeling. *Pattern recognition*, 42(9):1977–1987, 2009.

[12] L. He, X. Ren, Q. Gao, X. Zhao, B. Yao, and Y. Chao. The connected-component labeling problem: A review of state-of-the-art algorithms. *Pattern Recognition*, 70:25–43, 2017.

[13] J. Iverson, C. Kamath, and G. Karypis. Evaluation of connected-component labeling algorithms for distributed-memory systems. *Parallel Computing*, 44:53–68, 2015. doi: 10.1016/j.parco.2015.02.005

[14] S. Krasheninnikov. On scrape off layer plasma transport. *Physics Letters A*, 283(5):368 – 370, 2001. doi: 10.1016/S0375-9601(01)00252-3

[15] A. G. Landge, V. Pascucci, A. Gyulassy, J. C. Bennett, H. Kolla, J. Chen, and P.-T. Bremer. In-situ feature extraction of large scale combustion simulations using segmented merge trees. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1020–1031. IEEE, 2014.

[16] T. Lu, Q. Liu, X. He, H. Luo, E. Suchyta, J. Choi, N. Podhorszki, S. Klasky, M. Wolf, T. Liu, et al. Understanding and modeling lossy compression schemes on hpc scientific data. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 348–357. IEEE, 2018.

[17] F. Manne and M. M. A. Patwary. A scalable parallel union-find algorithm for distributed memory computers. In *International Conference on Parallel Processing and Applied Mathematics*, pp. 186–195, 2009. doi: 10.1007/978-3-642-14390-8_20

[18] J. Martín-Herrero. Hybrid object labelling in digital images. *Machine Vision and Applications*, 18(1):1–15, 2007.

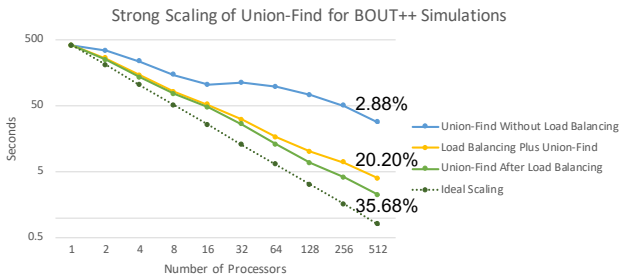
[19] D. Morozov and T. Peterka. Efficient delaunay tessellation through kd tree decomposition. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 728–738, 2016. doi: 10.1109/SC.2016.61

[20] F. Nespoli, I. Furno, B. Labit, P. Ricci, F. Avino, F. D. Halpern, F. Musil, and F. Riva. Blob properties in full-turbulence simulations of the TCV scrape-off layer. *Plasma Physics and Controlled Fusion*, 59(5):055009, mar 2017. doi: 10.1088/1361-6587/aa6276

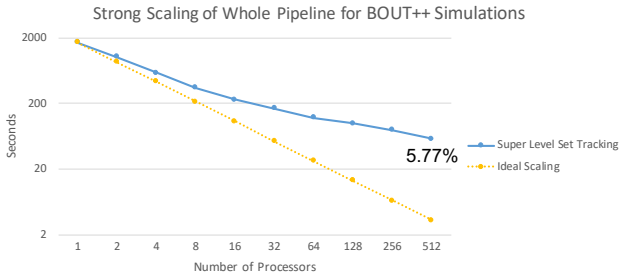
[21] F. Nespoli, P. Tamain, N. Fedorczak, G. Ciraolo, D. Galassi, R. Tatali, E. Serre, Y. Marandet, H. Bufferand, and P. Ghendrih. 3d structure and dynamics of filaments in turbulence simulations of west diverted plasmas. *Nuclear Fusion*, 2019. doi: 10.1088/1741-4326/ab2813

[22] J.-S. Par, J.-H. Yoon, and C. Kim. Stable 2d feature tracking for long video sequences. *International Journal of Signal Processing, Image Processing and Pattern Recognition*, 1(1):39–46.

[23] C. L. Phillips, H. Guo, T. Peterka, D. Karpeyev, and A. Glatz. Tracking vortices in superconductors: Extracting singularities from a discretized complex scalar field evolving in time. *Physical Review E*, 93(2):023305, 2016.

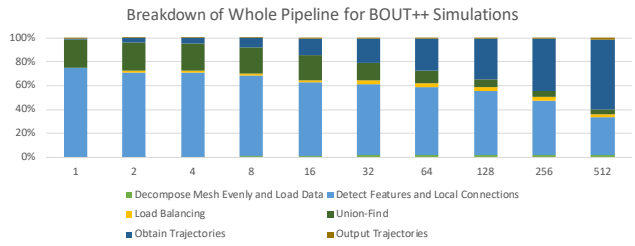


(a) Union-Find

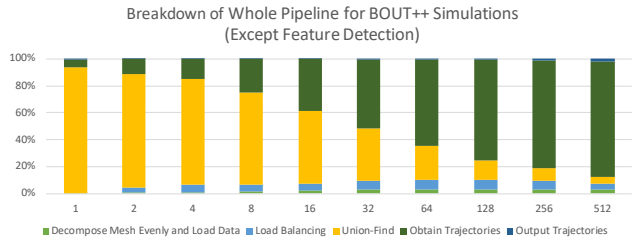


(b) Whole Pipeline

Figure 13: The figures show the strong scaling performance of our approach for BOUT++ fusion plasm simulations. (a) shows the strong scaling of the union-find. (b) shows the strong scaling of the whole pipeline for tracking super level sets. The labeled percentages are the scaling efficiencies of different methods when using 512 cores.



(a) All Steps of Pipeline



(b) All Steps of Pipeline Except Feature Detection

Figure 14: (a) shows the percentage breakdown of time for all steps of the whole pipeline when tracking super level sets on BOUT++ simulation data. In (b), we exclude feature detection and show others.

[24] B. Pirat, D. S. Khoury, C. J. Hartley, L. Tiller, L. Rao, D. G. Schulz, S. F. Nagueh, and W. A. Zoghbi. A novel feature-tracking echocardiographic method for the quantitation of regional myocardial function: validation in an animal model of ischemia-reperfusion. *Journal of the American College of Cardiology*, 51(6):651–659, 2008.

[25] Y. Qi, S. Zhang, L. Qin, H. Yao, Q. Huang, J. Lim, and M.-H. Yang. Hedged deep tracking. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[26] A. Rosenfeld and A. C. Kak. *Digital Picture Processing*. Academic Press, Inc., Orlando, FL, USA, 2nd ed., 1982.

[27] D. A. Russell, D. A. D'Ippolito, J. R. Myra, W. M. Nevins, and X. Q. Xu. Blob dynamics in 3d bout simulations of tokamak edge turbulence. *Phys. Rev. Lett.*, 93:265001, Dec 2004. doi: 10.1103/PhysRevLett.93.265001

[28] N. Saunier and T. Sayed. A feature-based tracking algorithm for vehicles in intersections. In *The 3rd Canadian Conference on Computer and Robot Vision (CRV'06)*, pp. 59–59. IEEE, 2006.

[29] J. Takle, D. Silver, E. Kovacs, and K. Heitmann. Visualization of multivariate dark matter halos in cosmology simulations. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, pp. 131–132. IEEE, 2013.

[30] H. Theisel and H.-P. Seidel. Feature flow fields. In *Proceedings of the Symposium on Data Visualisation 2003*, VISSYM '03, pp. 141–148. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2003.

[31] J. Tierny, A. Gyulassy, E. Simon, and V. Pascucci. Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1177–1184, Nov 2009. doi: 10.1109/TVCG.2009.163

[32] S. Vasuhi, B. Haripriya, and V. Vaidehi. Object detection and tracking using statistical and stochastic techniques. In *2015 International Conference on Industrial Instrumentation and Control (ICIC)*, pp. 1115–1119. IEEE, 2015.

[33] Z. Wang, Q. Liu, W. Waganaar, J. Fontanese, D. James, and T. Munsat. Four-dimensional (4d) tracking of high-temperature microparticles. *Review of Scientific Instruments*, 87(11):11D601, 2016. doi: 10.1063/1.4955280

[34] K. Wu, E. Otoo, and K. Suzuki. Optimizing two-pass connected-component labeling algorithms. *Pattern Analysis and Applications*, 12(2):117–135, 2009.

[35] L. Wu, K. J. Wu, A. Sim, M. Churchill, J. Y. Choi, A. Stathopoulos, C.-S. Chang, and S. Klasky. Towards real-time detection and tracking of spatio-temporal features: Blob-filaments in fusion plasma. *IEEE Transactions on Big Data*, 2(3):262–275, 2016.

[36] X. Q. Xu, R. H. Cohen, T. D. Rognlien, and J. R. Myra. Low-to-high confinement transition simulations in divertor geometry. *Physics of Plasmas*, 7(5):1951–1958, 2000. doi: 10.1063/1.874044

[37] X. Zou, K. Wu, D. A. Boyuka, D. F. Martin, S. Byna, H. Tang, K. Bansal, T. J. Ligocki, H. Johansen, and N. F. Samatova. Parallel in situ detection of connected components in adaptive mesh refinement data. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 302–312. IEEE, 2015.