Asynchronous and Load-Balanced Union-Find for Distributed and Parallel Scientific Data Visualization and Analysis

Article *in* IEEE Transactions on Visualization and Computer Graphics · April 2021

DOI: 10.1109/TVCG.2021.3074584

8 authors, including:

Jiayi Xu
The Ohio State University
17 PUBLICATIONS **116** CITATIONS

SEE PROFILE

Hanqi Guo
Argonne National Laboratory
56 PUBLICATIONS **773** CITATIONS

SEE PROFILE

Asynchronous and Load-Balanced Union-Find for Distributed and Parallel Scientific Data Visualization and Analysis

Jiayi Xu, Hanqi Guo, *Member, IEEE,* Han-Wei Shen, *Member, IEEE,* Mukund Raj, Xueyun Wang, Xueqiao Xu, Zhehui Wang, and Tom Peterka, *Member, IEEE*

Abstract—We present a novel distributed union-find algorithm that features asynchronous parallelism and k-d tree based load balancing for scalable visualization and analysis of scientific data. Applications of union-find include level set extraction and critical point tracking, but distributed union-find can suffer from high synchronization costs and imbalanced workloads across parallel processes. In this study, we prove that global synchronizations in existing distributed union-find can be eliminated without changing final results, allowing overlapped communications and computations for scalable processing. We also use a k-d tree decomposition to redistribute inputs, in order to improve workload balancing. We benchmark the scalability of our algorithm with up to 1,024 processes using both synthetic and application data. We demonstrate the use of our algorithm in critical point tracking and super-level set extraction with high-speed imaging experiments and fusion plasma simulations, respectively.

Index Terms—Union-find, disjoint set, connected component labeling, distributed and parallel processing, critical point, level set.

1 Introduction

As the scale of scientific data generated by experiments and simulations grows, it becomes a common practice to use High-Performance Computing (HPC) clusters to analyze and visualize data in parallel. In such distributed and parallel computing environments, data parallelism is a default paradigm; input data are partitioned into data blocks, which are distributed among parallel processes. With data-parallelism, intermediate results are produced from individual data blocks before they are merged into the final result.

This paper focuses on union-find, which is widely used in many scientific visualization algorithms [1–8] and plays a key role in merging disjoint sets. For example, in the extraction of super-level sets [7], regions with scalar values larger than a given threshold are extracted. Specifically, in 3D volumetric data, union-find merges neighboring voxels that are greater than the given threshold into connected components. In critical point tracking [9,10], union-find connects critical points detected in a spacetime gird, which allows effective tracking of time-dependent phenomena.

In the parallelization of the abovementioned visualization algorithms, we identify two scalability bottlenecks in distributed union-find: (1) high synchronization costs and (2) imbalanced workloads. First, because each disjoint set is usually distributed in a subgroup of processes, the use of global synchronizations may block the rest of the processes, causing busy waits in program execution. Up to date, distributed union-find [5,7,11–13] has been implemented with the bulk-synchronous parallel programming model [14], which manages parallel processes to alternate local computations and global synchronizations iteratively until distributed algorithms converge. In the context of distributed union-find, local computations consist of performing set operations on local data, and global synchronizations are used for merging and updating disjoint sets across processes. Second, imbalanced workloads between parallel processes lead to additional busy waits. The workload imbalance is caused by processes that possess imbalanced disjoint-set elements. For example, in super-level set extraction, data blocks in some processes may find more voxels above the given threshold than others. Likewise, in critical point tracking, critical points may be non-uniformly distributed in a domain. In this work, we present novel solutions to reduce synchronization costs and balance processes' workloads for distributed union-find, as described below.

First, we eliminate the use of global synchronizations for distributed union-find based on the fact that set unitings are order-independent. We prove that it is possible to eliminate global synchronizations by overlapping synchronizations with local computations and guarantee algorithm convergence and final-result correctness. It enables the use of asynchronous point-to-point communications to merge and update disjoint sets across processes in practice, which can

- *Jiayi Xu and Han-Wei Shen are with the Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, 43210, USA.*
  E-mail: {xu.2205, shen.94}@osu.edu
- *Hanqi Guo, Mukund Raj, and Tom Peterka are with the Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439, USA.*
  E-mail: {hguo, mraj, tpeterka}@anl.gov
- *Xueyun Wang is with the School of Physics, Peking University, Beijing, China 100871.*
  E-mail: wxy2015@pku.edu.cn
- *Xueqiao Xu is with the Physical and Life Sciences Directorate, Lawrence Livermore National Laboratory, Livermore, CA 94550, USA.*
  E-mail: xu2@llnl.gov
- *Zhehui Wang is with the Physics Division, Los Alamos National Laboratory, Los Alamos, NM 87545, USA.*
  E-mail: zwang@lanl.gov

be fully overlapped with local computations.

Second, we balance the disjoint-set elements in each process for workload balancing in distributed union-find because the time complexity of local computation is proportional to the number of set elements [15]. In our implementation, a k-d tree space decomposition is used to redistribute the set elements evenly among processes because k-d trees can be used to effectively balance spatial data, which is the common type for scientific datasets.

We demonstrate the scalability of our distributed union-find algorithm by measuring the performance with up to $1,024$ processes for both critical point tracking and super-level set extraction. Benchmark datasets include 3D spatial synthetic data and 2D time-varying application data output by high-speed imaging and fusion plasma simulations. We show that our algorithm achieves shorter execution time and better scalability than the existing distributed union-find methods in scientific datasets. In summary, the main contributions of this paper are twofold:

- We prove global synchronizations between processes can be eliminated in distributed union-find, and present a method that allows distributed union-find to overlap communications and local computations, which reduces processes' busy-waiting time.
- We redistribute the disjoint-set elements across processes evenly for load balancing of distributed union-find using a k-d tree decomposition scheme, which improves algorithm scalability in scientific applications.

2 Related Work

We distinguish shared-memory [16, 17] and distributed-memory [5, 7, 11–13] parallelization of union-find, and this paper is focused on the distributed-memory settings. Shared-memory parallel union-find focuses on computing environments that share the same memory space, such as multi- and many-core processors. Distributed parallel union-find is designed to perform computations in independent processes with distributed memory spaces.

2.1 Distributed Union-Find Algorithms

For data visualization and analysis, most distributed union-find algorithms [5, 7, 11–13] are implemented with the bulk synchronous parallelism [14], which has been the orthodox parallel programming model for distributed iterative algorithms since the 1980s. In the context of union-find, the bulk synchronous implementations alternate two stages: (1) each process performing serial union-find computations locally and (2) all processes synchronized to merge and update disjoint sets across processes; the two-stage iterations continue until convergence. The benefits of using bulk synchronous parallelism include: (1) ensuring disjoint sets are consistent among the processes at each iteration, (2) ensuring messages are exchanged in a predetermined order without causing deadlocks of communications, and (3) offering straightforward termination detection. The drawbacks include disallowing overlapped computations and communications.

Differences of existing distributed union-find algorithms mainly vary in two aspects: (1) data distribution and (2) communication patterns.

Data distribution: There are two ways to distribute data: full replication and data partitioning. Full replication refers to duplicating all set elements among participating processes [11], while data partitioning refers to subdividing the input data and redistributing the partitioned data among the processes. Harrison et al. [5] partitioned mesh data and balanced the mesh cells across the processes using a binary space partitioning approach.

Communication patterns: To merge and update disjoint sets across the processes, existing distributed union-find methods use global synchronizations with three different communication patterns: (1) master/slave, (2) parallel merging, and (3) neighbor exchange. First, with the master/slave pattern in [7], after all the processes finish local computations, all inter-process mergings of sets are sent to the master process to resolve. Then, the master process broadcasts the sets after the mergings to all processes. Second, with the parallel merging pattern, Cybenko et al. [11] built a tree of processes to merge disjoint sets in paired processes every time by sending all disjoint sets of a process to the process's partner. As a result, the root in the tree of processes produces the disjoint sets merged from all processes. Third, with the neighbor exchange pattern [5, 12, 13], disjoint sets spanning over adjacent processes are merged and updated at every iteration. The neighbor exchange can be implemented under Message Passing Interface (MPI) standards [18] by using either a collective synchronous operation `MPI_Alltoall` involving all processes [5] or point-to-point synchronous operations [12, 13].

2.2 Distributed and Parallel Visualization and Analysis

Distributed union-find is mostly used for connected component labeling (CCL) in scientific visualization and analysis algorithms, including distributed percolation analysis [7], interval volume extraction [5], statistical analysis of connected regions [6], contour tree computation [2, 4], and merge tree computation [3, 8]. Although the implementation of CCL [19] can be either union-find or breadth-first search [20–24], union-find is proved to be more scalable in distributed and parallel settings [13, 19] because union-find organizes elements of connected components using tree structures, explained in Section 3.1, which can efficiently synchronize labels of elements in the connected components across processes. The rest of this section samples typical scientific visualization and analysis applications with distributed union-find.

Region extraction: Visualizing and analyzing spatial regions in scientific data usually offer important insights to scientists, where union-find can play the role of grouping voxels or mesh cells into connected regions. In distributed percolation analysis for turbulent flows [7], the percolation function requires quantifying the volume of regions with values higher than a threshold, where the regions are extracted using distributed union-find. Interval volume, the volume between two isosurfaces, is important for distributed analysis of supernova simulation, and can be extracted using distributed union-find [5]. In the distributed statistical analysis of fluid flows [6], connected phase regions are first extracted using the existing distributed union-find [5, 13]; then, each compute node launches asynchronous threads to compute statistics on the extracted regions.

Critical point tracking: In critical point tracking, union-find is used to connect critical points sharing the same spacetime mesh cells. Tricoche et al. [9] and Garth et al. [10] generalized the spatial mesh of the input data into spacetime mesh, with time being the additional dimension. By assuming continuities in the spacetime domain, critical points can be tracked based on spacetime mesh connectivities.

Scalar field topology: Distributed union-find has been used in the distributed computation of merge tree [3, 8] and contour tree [2, 4]. Given scalar values defined on mesh vertices, union-find is used to group vertices on the mesh paths where vertices' values on the paths are increasing.

3 Preliminaries

This section reviews the serial union-find and the bulk synchronous parallelism based distributed union-find methods.

3.1 Serial Union-Find

In general, the input of union-find algorithms is a graph, $G =< V, E >$, where V is the collection of all elements and E is the collection of all edges between elements. The output consists of disjoint sets between which no connecting edges exist. Union-find has two basic operations: `Union`(v, v') and `Find`(v), where `Union` unites disjoint sets that two edge-connected elements belong to and `Find` returns the representative element of the set containing the given element. A `Union` operation is performed on every two edge-connected elements to output final disjoint sets. To manage disjoint sets efficiently, the internal implementation of union-find uses a tree data structure [15, 25] as described below.

Disjoint-set trees: Each disjoint-set tree corresponds to a disjoint set and has a *root* element, which is the representative element and the identifier of the set. Every element has a parent pointer pointing to a *parent* element and is a *child* of that parent. Every root's parent is itself.

`Union` and `Find` with disjoint-set trees: A `Find`(v) operation is implemented by following the parent pointers starting from the given element to identify the root. A `Union`(v, v') operation is performed on a pair of edge-connected elements using two sub-operations: (1) *set uniting* and (2) *edge passing*. If an edge connects two roots, a set uniting is performed to point one root to the other to form a single disjoint-set tree. Otherwise, the edge is passed through parent pointers of the given elements to `Find` the roots; after the edge is passed to the roots, a set uniting will be invoked so that the sets of the given elements are merged. The set uniting can follow either *uniting by rank* [5, 12, 15, 26] or *uniting by size* [15] rule, which unites one set into the other with a higher rank or a greater size.

Path compression: Path compression is a process to shorten the paths from elements to roots in disjoint-set trees by pointing elements to either (1) their grandparents [27, 28] or (2) their roots [29], where the two choices were proved to have the same amortized time complexity in [15]. Path compression can make `Find` and the edge passing operations identify elements' roots with fewer iterations, hence, more efficient.

3.2 Distributed Bulk-Synchronous Union-Find

Distributed bulk-synchronous union-find [5, 8, 11–13] iterates over local computations, synchronous communications, and synchronous termination detection.

Local computations: Each process is responsible for handling local work, including processing incoming messages, performing `Union` operations and path compressions for local elements stored in the process, and queuing outgoing messages.

Communications: Processes use synchronous communications to perform inter-process (1) set uniting, (2) edge passing, and/or (3) path compression. First, the set uniting following uniting by rank or uniting by size rule requires processes' synchronizations to exchange ranks or sizes of sets to be united and avoid creating cycles in resulting disjoint-set trees. Second, processes may pass edges of local elements to the processes of elements' parents. The element parents' processes receive the edges and continue passing edges until reaching roots for future set unitings. Third, for path compression, processes may issue queries about elements' grandparents to elements' parents. Then, the processes of elements' parents send the information of grandparents back after receiving the queries.

Termination detection: A *collective synchronous communication*, such as `MPI_Allreduce` used in [13], is applied to check whether all processes have completed assigned work periodically for iterations to terminate. The "collective" means all available processes participate in the communication. The "synchronous" means the communication blocks the participating processes for an agreement of termination and will not return until all the participating processes respond, ensuring all the participating processes agree when it is ready to terminate.

4 Overview and Design Considerations

Each process's input consists of elements, a subset of V, and the input elements' edges. Elements in different processes are not overlapping. Every element has a unique ordinal *identifier (ID)* and numeric coordinates. The output consists of disjoint-set trees distributed among the processes.

Algorithm overview: We give examples for involved operations in Fig. 1. The algorithm initialization includes redistributing input elements and edges for the load balancing (Section 6) and creating a disjoint set for every element. After the initialization, each process iterates over local computations (Algorithm 1), asynchronous communications, and asynchronous termination detection (Section 5.4); each process remains in the iterations as long as there is still remaining local work or incoming new message. In the local computations, each process first consumes incoming messages and then repeats set uniting (Section 5.2), path compression (Section 5.3), and edge passing (Section 5.2) until local work is complete. The outgoing messages are immediately sent after they have been produced, and the incoming messages promptly drive a new iteration of operations after the incoming messages have been delivered. After all processes terminate iterations, we have acquired correct disjoint sets. For scientific visualization and analysis, a final local path compression (Section 5.3) is performed to label all elements in the same set by the same identifier,

i.e., the ID of the set root. As not all distributed iterative algorithms can necessarily be designed to overlap communications and local computations, we prove our algorithm's convergence and correctness in Section 5.5.

Algorithm design considerations: We explain four considerations for the design of distributed asynchronous union-find with the challenges when global synchronizations are eliminated.

First, we adopt an asynchronous parallelism pipeline with asynchronous communications and asynchronous termination detection to overlap communications and local computations. Because asynchronous communications do not block participating processes, processes can continue doing local computations without waiting for communications to complete.

Second, we use a *uniting by identifier (ID)* rule for the set uniting by uniting a set into the other with a smaller root ID to support overlapping inter-process and local merging of sets. Although uniting by rank rule and uniting by size rule are frequently used in the literature, when multiple sets with the same rank or size are united, the two rules may cause cycles in resulting disjoint-set trees and lead to deadlocks if processes are not synchronized [17]. Hence, in shared-memory asynchronous union-find [17], when the sets to be united have equal ranks or sizes, Anderson and Woll used the set records in the shared memory to prevent cycles such that one set is united into the other with a greater record index. Because distributed-memory processes do not have such shared records of sets, elements' identifiers are used instead in our algorithm.

Third, we create local-tree data structures to reduce communications in contrast to the existing distributed union-find [5, 11–13] without such local trees. In our algorithm, only path compression for local-tree roots require communications with the set roots' processes, and the path compression for non-root elements does not involve communications. For example, in Fig. 1d, local root 6 represents the subset consisting of 6, 7, and 10. After 2 unites with 0, only the local root 6 involves communications to point to the new set root 0 for path compression using our algorithm; however, without local trees, process 0 may exchange messages with all the three elements (6, 7, and 10) for path compression, which involves more communications.

Fourth, we consider boundary cases in path compression to solve deadlocks of communications when the asynchronous termination detection is used. After local roots already point to set roots, the local roots may still need to communicate with the set roots for path compression because the set roots may point to other elements after applied union operations. For example, in Fig. 1d, local root 6 may need to keep communicating with set root 2 for path compression because 2 may point to another element, i.e., 0 in this example later. A communication deadlock happens when multiple processes with local roots may keep performing such communications with each other even though correct disjoint sets have been built, leading to one of the processes that may always be active so that iterations may never finish when using asynchronous termination detection. We present a solution in Section 5.3.

5 Distributed Asynchronous Union-Find

We detail our distributed asynchronous union-find.

5.1 Distributed Disjoint-Set Trees

We extend serial disjoint-set trees to create additional data structures for our distributed union-find algorithm.

Local disjoint-set trees are local subtrees formed by local elements of processes, where the *local elements* of a process are elements stored in the process. *Local roots* represent the roots of such local subtrees. Each process can identify local roots by examining local elements that have non-local parents. For example, in Fig. 1c, 6, 7, and 10 form a subtree in process 2, and 6 is the local root of the subtree.

A *distributed disjoint-set tree* is formed by all elements of the corresponding disjoint set and may consist of multiple local trees. The roots of the distributed disjoint-set trees are called *disjoint-set tree roots*, or *set root* in short hereafter. Each process can identify the set roots by checking local elements that point to themselves.

Non-root elements represent the elements that are neither local roots nor set roots.

5.2 Distributed Union Operations

Similar to the serial union-find, a distributed `Union` operation is performed on a pair of edge-connected elements with two sub-operations: (1) distributed set uniting and (2) distributed edge passing. Distributed edges are passed to set roots following paths in distributed disjoint-set trees so as to be used for set unitings.

Distributed set uniting: The set uniting is based on a *uniting by ID* rule, where two disjoint sets are united by setting the parent of the root of one set to an element with a smaller ID in the other set. For example, from Fig. 1 (b) to (c), element 6 sets its parent pointer to element 2; however, 2 will not point to 3 because 3 is bigger than 2, and hence, 2 becomes a (temporary) set root. As a result, a parent has a strictly smaller ID than its children in disjoint-set trees, and each set root is the smallest element in a disjoint set, which ensures no cycles.

The set uniting does not produce outgoing messages. In our implementation, each process points every stored set root to the smallest neighbor element if the process stores or receives any edge connecting the set root with a smaller element. Examples are illustrated in Fig. 1c.

To make elements be aware of smaller neighbor elements for set unitings, we store each edge in the same process of its larger endpoint after the data redistribution (Section 6) and during the following distributed edge passings. For example, in Fig. 1b, the edge between element 5 and element 9 is stored in process 3 so that 9 is aware of 5 and can point to 5 for the set uniting.

Distributed edge passing: Edge-passing transports edges of elements to set roots so that the roots are aware of the edges connecting to other disjoint sets for following set unitings.

Edges are passed through endpoints' parent pointers as follows repeatedly. We replace an edge's larger endpoint with the endpoint's parent. For example, an edge between element 3 and element 2 in Fig. 1c is updated so that

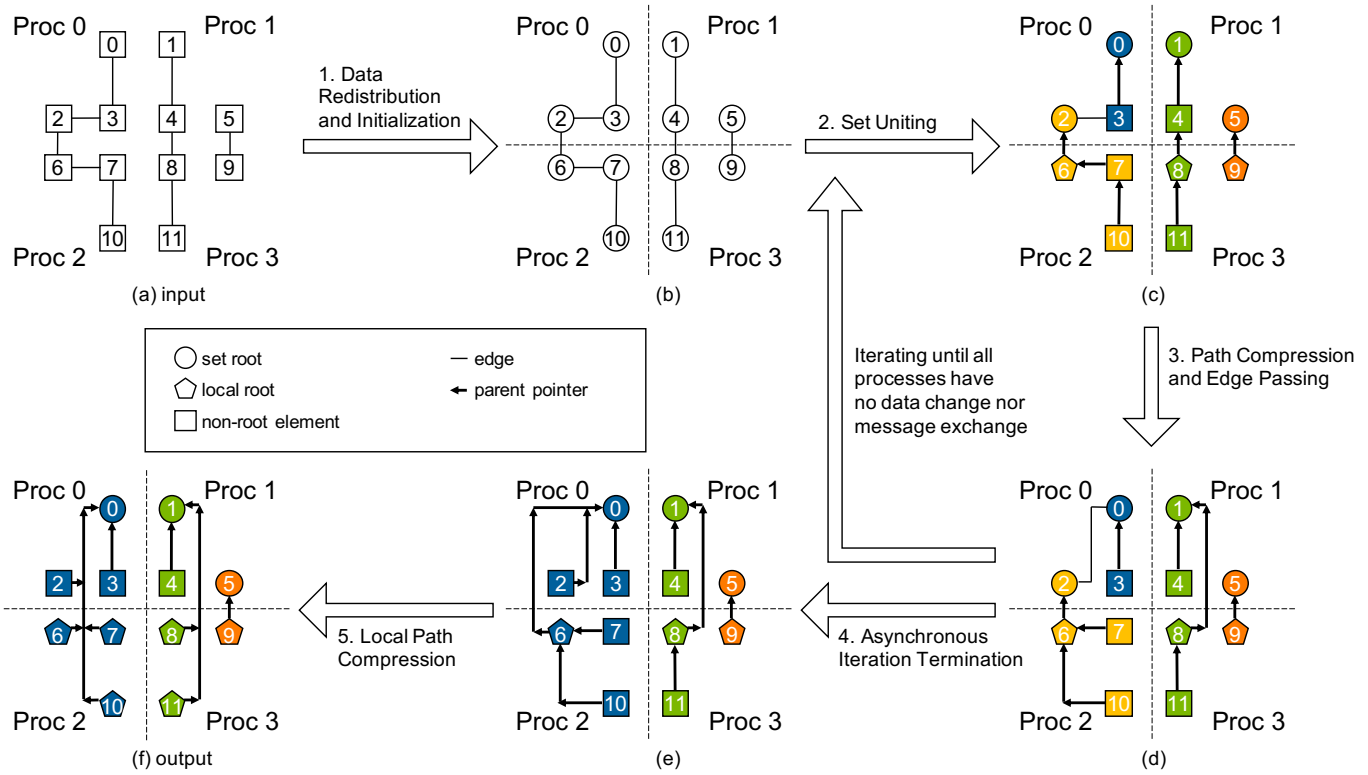


Fig. 1: Examples of different operations. (a): An input graph consists of twelve elements. The IDs of the elements are from 0 to 11. (b): Elements and edges are redistributed evenly across processes with disjoint sets initialized. (c): Disjoint sets are united in parallel. Colors represent (temporary) disjoint sets. (d): Paths from elements to roots are compressed from (c) to (d), such as the path between 8 and 1 and the path between 10 and 6. Remaining edges are passed toward set roots; for example, an edge between 3 and 2 in (c) is passed to connect 0 and 2 in (d). Each process performs the path compression and edge passing independently without blocking other processes. (e): After processes terminate iterations asynchronously, correct disjoint sets are acquired. Certain non-root elements (e.g., 7, 10, and 11) point to local roots rather than set roots. (f): After all non-root elements point to set roots using a local path compression, three disjoint sets are produced.

connecting 0 and 2 in Fig. 1d after the replacement. We deprecate the edge if its current endpoints have the same ID; otherwise, the edge is sent to and stored in the process of its current larger endpoint for a set uniting or additional passings.

5.3 Distributed Path Compression

Path compression makes edge passing more efficient by shortening the paths between elements and those connected in the disjoint-set trees with lower IDs until each element's parent pointer points to either a set root or a local root. We detail the path compression algorithm for different types of elements as below.

Non-root elements: Path compression for non-root elements does not produce outgoing messages either. At every iteration, non-root elements modify their parent pointers from their parents to the parents' parents, i.e., their grandparents, if there exist grandparents in the same processes. For example, non-root element 10 changes its parent pointer from 7 to 6 in Fig. 1d and remains pointing to 6 until iterations finish.

After the iterations terminate, each process points all the non-root elements to the current set roots, as illustrated in Fig. 1f; as a result, all elements within the same set have a common label: the ID of the set root.

Local roots: At every iteration, each local root communicates to its parent, which is in a different process, to query its grandparent. The parent who receives the query sends the queried information back to the local root for it to update its parent pointer. For example, in Fig. 1c, after element 8 queries its grandparent, element 4 return element 1 to element 8. When the local root receives the feedback about its grandparent, the local root updates its parent pointer accordingly.

A boundary case is when a local root has a set root parent, the set root records the local root's process ID after receiving a grandparent query from the local root and sends feedback indicating not to request the grandparent again, which reduces future communications and avoids possible deadlocks of communications due to the use of asynchronous termination detection. If the set root later points to another element, the set root notifies its non-local children, which are guaranteed to be local roots in other processes, for path compression. After the notification, the local roots are allowed to send additional grandparent queries if they do not point to set roots. For example, in Fig. 1d, set root 2 records the process ID of local root 6 after receiving the grandparent query from 6. When set root 2 later unites with 0, 2 notifies 6 of 0 for path compression.

5.4 Asynchronous Nonblocking Communications and Termination Detection

Processes exchange messages using asynchronous communications, which do not block participating processes.

Communication protocols: Three types of messages are exchanged across processes for edge passing and path compression:

Transferred edge message is used to transfer the data of an edge. This message contains element IDs and process IDs of the two endpoints of the edge. This message is used in (1) edge redistribution after load balancing and (2) edge passing across processes.

Grandparent query message is used when a local root requests for its grandparent. This message contains the element ID and process ID of the local root. This message is sent from the process of the local root to the process of its parent.

Grandparent message is used to answer a grandparent query issued from a local root. This message either contains the element ID and process ID of the corresponding grandparent or indicates the local root's current parent is a set root and inhibit the local root from sending grandparent queries again. This message is sent from the process of the local root's parent to the process of the local root.

Asynchronous termination detection: In our asynchronous union-find algorithm, each process exits from iterations when all processes finish local work and no messages are being exchanged. The iteration termination detection is straightforward in bulk-synchronous parallelism but requires careful designs for distributed asynchronous algorithms to ensure each process knows all other processes have finished asynchronously and no messages are being transferred. We follow the `iexchange` module of DIY library [30, 31] and the nonblocking termination detection mentioned in [24]. Each process undergoes four states: (1) *active*, (2) *idle*, (3) *ready-to-terminate*, and (4) *terminate*, and exchanges their states using asynchronous communications to achieve a consensus for correct termination. We refer to [24] for details about the transitions between the states, which are also summarized in the supplementary appendices.

Algorithm 1: local_computations(comm, in_msgs, elements, edges) // The "comm" is a communicator, such as `MPI_COMM_WORLD`. The "isend" is a point-to-point asynchronous operation for message sending, such as `MPI_Isend`.

```
   /* consume incoming messages                                  */
 1 for each msg in in_msgs do
 2     if "transferred edge" in msg then
 3         add_local_edge(msg, edges)
 4     end
 5     else if "grandparent query" in msg then
 6         grandparent ← retrieve_grandparent(msg,
             elements)
 7         comm.isend(grandparent)
 8     end
 9     else if "grandparent" in msg then
           /* pass compression of local roots                   */
10         point_to_grandparent(msg, elements)
11     end
12 end
   /* perform distributed asynchronous union-find
      operations                                                 */
13 while have local work do
14     set_uniting(elements, edges) ▷Section 5.2
15     grandparent_queries, grandparents ←
         path_compression(elements) ▷Section 5.3
16     comm.isend(grandparent_queries)
17     comm.isend(grandparents)
18     transferred_edges ← edge_passing(elements,
         edges) ▷Section 5.2
19     comm.isend(transferred_edges)
20 end
```

5.5 Convergence and Correctness

We prove our algorithm converges to the correct result with a finite number of iterations in each process. Our proof has three assumptions. First, input data contain a finite number of elements and edges. Second, every message can be delivered, but the delivery order of messages is not guaranteed, for example, when MPI asynchronous communications are used. Third, the asynchronous termination detection can inform each process to exit iterations when all processes finish local work and no message is being transferred.

5.5.1 Converging in Finite Iterations

We explain our algorithm converges within a finite number of total iterations of processes, where the convergence is achieved when (1) all edges are consumed and (2) disjoint-set trees have no further changes.

(1) All input edges are consumed within finite total iterations.

Proof. At each iteration of a process, if a set root has edges connecting with smaller elements, at least one edge is consumed for set uniting; all other edges are passed to a new set root. When passing edges, any edge is either (1) deprecated because the endpoints already belong to the same set or (2) passed to a new set root within finite such passes because each disjoint set tree has no cycles, guaranteed by the uniting by ID rule, and contains finite elements. Each pass is either a local operation completed in one iteration or using a message delivered within finite iterations. Because set uniting and edge passing keep reducing the number of edges, which is finite, all input edges are consumed within finite total iterations. □

(2) Disjoint-set trees converge to trees with at most two layers within finite iterations, and no messages are exchanged after the convergence of disjoint-set trees.

Proof. In the loop of iterations, disjoint-set trees converge when all non-root elements point to (local/set) roots and all local roots point to set roots. Non-root elements point to local/set roots within finite local computations given that, at every iteration, non-root elements of a process will point to their local grandparents for path compression. Each local root points to a set root within a finite number of message deliveries, given that the local root points to its grandparent after one grandparent query process. Because every message

can be delivered within finite iterations, the local root points to a set root within finite iterations. Additionally, local roots do not send grandparent queries again after pointing to set roots because the boundary cases in Section 5.3 are considered, ensuring no messages are exchanged after disjoint-set trees converge and iterations can properly terminate when asynchronous termination detection is used. After the final local path compression, which is illustrated in Fig. 1f and forms one local iteration at each process, all elements point to set roots, leading to trees with at most two layers. □

5.5.2 Outputting Correct Disjoint-Set Trees

The correctness is guaranteed by (1) ensuring connected elements in input belong to the same sets in the output and (2) producing unique disjoint set trees; both are independent of the order of set unitings.

(1) Every two edge-connected elements in an input graph belong to the same disjoint set in output.

Proof. At the first iteration of each process, if an edge is used for a set uniting, then the two elements connected by the edge are united to the same set; every other edge of the input graph undergoes distributed edge passing. Although we cannot guarantee the order of edge passings, we have shown that any edge is consumed within finite iterations. If a passed edge is applied for a union operation, the original two elements of the edge are guaranteed to belong to the same set due to the tree topology. If the edge has never been considered for a union operation and is deprecated during the edge passing, the only reason is the original two endpoints of the edge have already belonged to the same set before the edge is passed to set roots. Therefore, any two elements connected in an input graph belong to the same set in the output. □

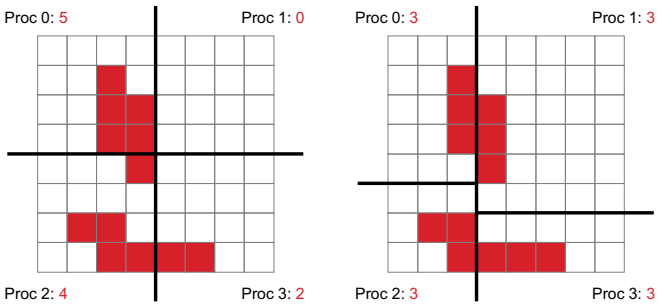
(2) The output consists of ideal disjoint-set trees, where elements, within each set, point to the set's smallest element.

Proof. We have shown that, the disjoint-set trees have at most two layers after the convergence, and all elements within a set point to a set root, regardless of the execution order of union operations. Because the uniting by ID rule guarantees each disjoint-set tree's root to be the smallest element within each set, our algorithm can produce the ideal disjoint-set trees after convergence. □

6 Load Balancing Using K-D Tree based Element Redistribution

We redistribute elements to balance the number of elements in each process for workload balancing in distributed union-find, as illustrated in Fig. 2b. We follow the k-d tree space decomposition method proposed by Morozov and Peterka [32] for distributed computing environments. Modified pseudocode is presented in Algorithm 2 and described below to support the load balancing in distributed union-find.

Given distributed elements with numeric coordinates, the decomposition approach partitions the domain so that each process is assigned a partition that contains a similar number of elements. Initially, all processes belong to a single group, which represents the whole domain. The domain is then decomposed based on medians of the coordinates of



(a) balancing mesh cells (b) balancing set elements

Fig. 2: Two possible load balancing schemes for distributed union-find in scientific visualization and analysis. We color set elements by red, and the element counts assigned to processes are indicated in respective corners. (a): Balancing the number of mesh cells in each process using binary space partitioning [5]. However, balancing cells may not solve workload imbalance effectively. For example, Process 1 has zero elements, and hence has no work to do. (b): Balancing the number of elements in each process based on a k-d tree decomposition. In this example, each process is assigned three elements and attains a balanced workload.

Algorithm 2: balancing_elements(comm, elements, edges)

```
1  rank ← comm.rank() // ID of this process
2  nproc ← comm.size() // process count
3  ndim ← domain dimensionality
4  dim ← 0 // current splitting dimension
5  group ← {0, 1, ..., nproc−1} // all processes
     belong to the same group initially
6  for i ∈ [0, log2(nproc)) do
7  |   m ← select_median(comm, elements, group,
   |     dim)
8  |   i_bit ← 1 << i
9  |   group ← {x ∈ group | x AND i_bit = rank AND
   |     i_bit}
10 |   partner ← rank XOR i_bit
11 |   elements ← swap_elements(comm, partner,
   |     elements, m)
12 |   dim ← (dim+1) MOD ndim
13 end
14 edges ← redistribute_edges(comm, elements, edges)
15 return elements, edges
```

elements. After a median is selected, the group of processes is split into two subgroups: the first subgroup contains all processes whose IDs' least significant bit is 0, and the second subgroup contains all processes whose IDs' least significant bit is 1. Elements are then exchanged between the two subgroups so that the first subgroup corresponds to the lower half of the domain and the second subgroup corresponds to the upper half of the domain. Each process in the first subgroup sends local elements that lie above the selected median to a partner process in the second subgroup; the partner process shares the same bits in the ID as the first group process except that the least significant bit is 1. Also, the partner process sends the first group process

the elements below the selected median. In the following iterations, the domain decomposition happens within every subgroup based on the second, third, and so on significant bits of process IDs until each subgroup contains one process.

After domain decomposition, we redistribute edges such that every edge is stored in its larger endpoint's process.

7 Algorithm Evaluation

We evaluate distributed union-find algorithms under a scientific feature extraction and tracking framework consisting of four stages: (1) domain partitioning, (2) feature detection, (3) connected component labeling (CCL) using distributed union-find, and (4) finalization, following existing scientific studies in the Feature Tracking Kit (FTK) [33, 34].

First, we evenly partition the input spatiotemporal data domain into spacetime blocks, which are then distributed over the participating processes. The spacetime blocks are constructed using spacetime meshing [9, 10] to support detecting scientific features with time continuity [9] and capturing topological events [9, 35]. Each spacetime block includes one layer of ghost cells in both space and time dimensions to associate spatiotemporal features across blocks.

Second, each process independently detects features and connections between the features. For example, in super-level set extraction, we detect spacetime cells with values higher than a specified threshold. In critical point tracking, we follow Tricoche et al. [9] to detect critical points on the faces of spacetime cells: we estimate derivatives at mesh grid points using central difference and locate critical points on the faces using inverse interpolation. The connections exist between features in adjacent spacetime cells or faces.

Third, we perform CCL using a distributed union-find algorithm. CCL involves detecting the connectivity between features and labeling each connected component by a unique identifier. We regard spatiotemporal features as elements in disjoint sets and connections between the features as edges between the elements. As a result of the distributed union-find algorithm, features within the same connected component are merged into a single set and labeled by a common identifier.

Fourth, we finalize feature extraction and tracking by gathering features within the same connected component to the same process for further visualization, analysis, and storage.

Our study below focuses on measuring the performance of CCL using distributed union-find algorithms.

Baseline: We implement the distributed union-find (DUF) method of Iverson et al. [13] load-balanced by balancing mesh cells [5] as the baseline for comparison. The study of [13] evaluates five state-of-the-art distributed approaches on the CCL task and indicates that, except the breadth-first search based label propagation exhibiting considerably worse results, the other four approaches, including the DUF, have similar strong scaling efficiency in each test data.

Compared with the baseline, the improvement caused by overlapping communications with computations using asynchronous parallelism is demonstrated on synthetic data (Section 7.1). The benefit of redistributing elements for load balancing is highlighted in both experimental and simulation datasets due to which have non-uniformly distributed features (Section 7.2).

Computing platform: We run experiments on an HPC cluster, which has 664 compute nodes. Every compute node has Intel Xeon E5-2695v4 CPUs with 32 cores and 128 GB memory. The HPC cluster uses an Intel Omni-Path interconnect network. Message passing is supported by the Intel MPI library. The file storage system of the cluster is IBM General Parallel File System.

7.1 Benchmark on Synthetic Data

We begin by measuring the influence of overlapping communications and computations using asynchronous parallelism with asynchronous communications and asynchronous termination detection for distributed union-find. In the experiments, we track and extract two types of features, including (1) critical points (local maxima, local minima, and saddle points) and (2) super-level sets, on synthetic data. Synthesizing data allows us to control data resolution and the number of features to support different evaluations. An example of the synthetic data we use is visualized in Fig. 3.

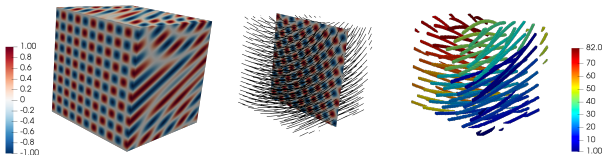


Fig. 3: Visualizations of a synthetic case. (a): A synthetic volume with scalar values ranging from -1 to 1. We track critical points and extract super-level sets on the same synthetic volume. (b): Trajectories of the critical points represented by black lines. (c): Super-level sets with a threshold of 0.8. 82 connected components are labeled, and each is assigned a unique hue.

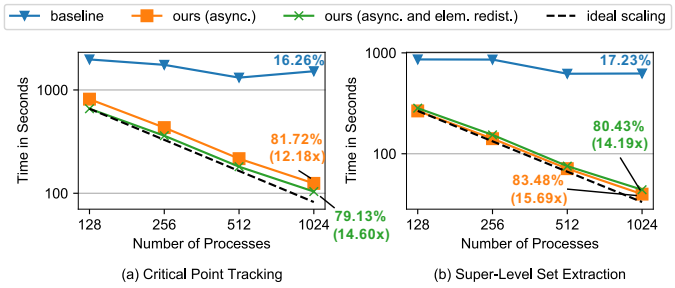


Fig. 4: Strong scaling of distributed union-find on $1,024^3$ synthetic data using 128 to 1,024 processes for (a) tracking critical points and (b) extracting super-level sets. Both axes are log scales. The baseline is the distributed union-find (DUF) of Iverson et al. [13] with balanced mesh cells [5]. Our methods consist of the distributed asynchronous (async.) union-find without/with the k-d tree based element redistribution (elem. redist.).

7.1.1 *Strong Scalability Study*

We conduct a strong scaling experiment using synthetic data with a $1,024^3$ resolution, which has 161,338,942 critical points and 72,097,212 voxels with values larger than 0.8.

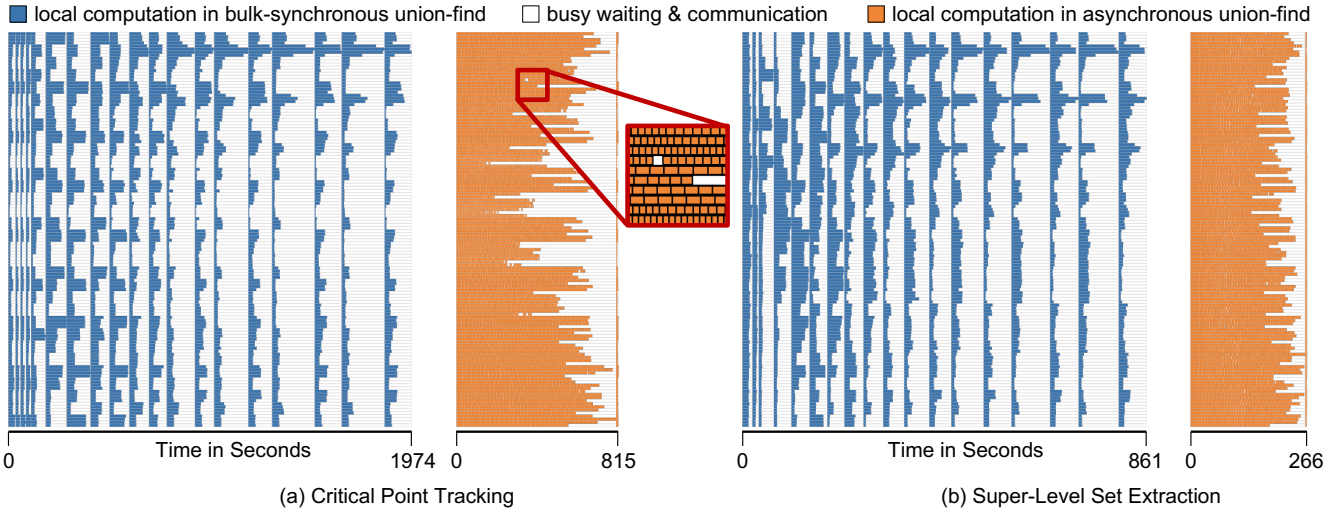


Fig. 5: Gantt charts of bulk-synchronous baseline [13] and our asynchronous algorithm using $1,024^3$ synthetic data distributed among 128 processes. The horizontal axis encodes time. Each row corresponds to a process.

Compared with the baseline using the bulk-synchronous parallelism, the use of asynchronous parallelism leads to significant improvement in strong scaling efficiency. Strong scaling results using up to $1,024$ processes are shown in Fig. 4. The baseline attains a strong scaling efficiency of 16.26% in the critical point tracking benchmark and 17.23% in the super-level set extraction benchmark. Our algorithm with the asynchronous parallelism attains 81.72% and 83.48% for the two benchmarks, respectively, with a speedup of 12.18x and 15.69x over the baseline.

To investigate why overlapping communications and computations using asynchronous parallelism is better than the distributed bulk-synchronous parallelism in detail, we list processes' computation time at each iteration in Fig. 5. In the distributed bulk-synchronous baseline, a global synchronization is performed at the end of each iteration to merge and update sets across the processes and detect iterations' termination. Due to the use of global synchronizations, processes with less computational work become busy-waiting for processes with more work at each iteration round. After overlapping communications and computations using asynchronous communications and asynchronous termination detection, each process performs computations at their own pace without being blocked by other processes, leading to reduced waiting time and improved computational resource usage. After the end of the iterations, our algorithm performs an additional local path compression, however, which occupies only a small portion of the total execution time.

We also evaluate the influence of using asynchronous parallelism on the number of iterations per process. Results in Fig. 6 illustrate that our distributed asynchronous approach has approximately doubled iterations compared with the bulk-synchronous baseline in both the critical point tracking and super-level set extraction. Because asynchronous communications are nonblocking, processes can iterate without waiting for other processes. As long as new messages come, processes can immediately come to the next iteration and handle new work. In contrast, the bulk-synchronous parallelism synchronizes processes between

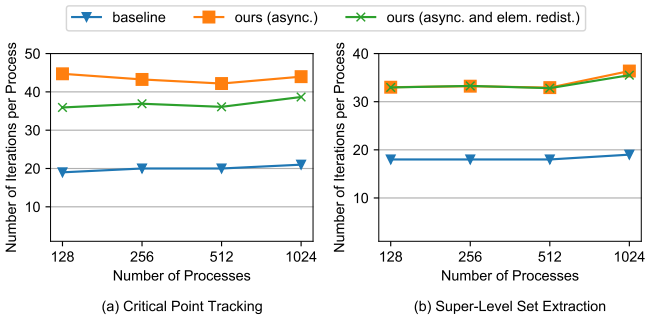


Fig. 6: Iteration count per process of distributed union-find on $1,024^3$ synthetic data. The horizontal axis is a log scale.

iterations. Hence, each process may receive work from all other processes after a synchronization, leading to more work to do during the next iteration and requiring fewer iterations. Although using asynchronous parallelism results in more iterations, our algorithm's total execution time is less than the bulk-synchronous baseline, as seen in Fig. 4.

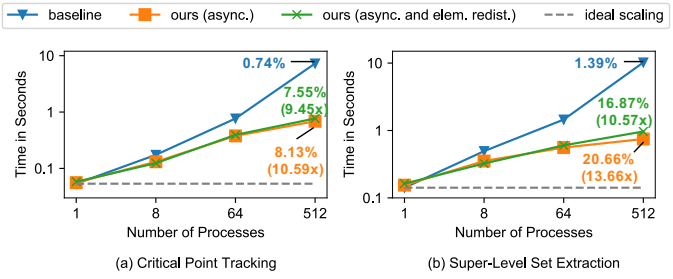


Fig. 7: Weak scaling of distributed union-find on synthetic data. We use four combinations of data resolutions and process counts: 32^3 with 1 process, 64^3 with 8 processes, 128^3 with 64 processes, and 256^3 with 512 processes. Both axes are log scales.

7.1.2 Weak Scalability Study

We measure the influence of using asynchronous parallelism on weak scaling of distributed union-find. The weak scaling evaluates the performance when each process is assigned a constant-size problem as the number of processes increases. In this study, each process is assigned with a 32^3 mesh grid with a constant feature density. Test data with sizes of 32^3, 64^3, 128^3, and 256^3 are processed using 1, 8, 64, and 512 processes respectively. The ideal weak scaling is when the execution time is constant in all the runs.

Compared with the baseline using bulk-synchronous parallelism, the asynchronous parallelism leads to significant enhancement in weak scaling efficiency. The results are displayed in Fig. 7. When using 512 processes, the baseline achieves weak scaling efficiency of 0.74% in the critical point tracking benchmark and 1.39% in the super-level set extraction benchmark. Our distributed union-find with the asynchronous parallelism attains 8.13% and 20.66% efficiency in the two benchmarks, respectively, with a speedup of 10.59x and 13.66x over the baseline.

7.2 Scientific Applications

We evaluate distributed union-find, especially the load balancing performance, in two scientific applications: (1) tracking critical points in exploding wire experimental data and (2) extracting super-level sets in fusion plasma simulation data. Both of the scientific data are time-varying.

7.2.1 Application Background and Benchmark Setting

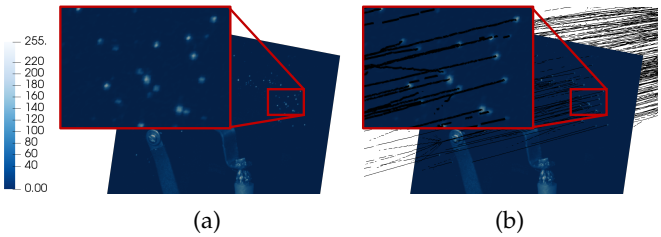
We give background and benchmark settings for the two scientific applications.

Exploding wire experiments: Scientists can use an exploding-wire apparatus to generate many high-temperature microparticles [36, 37]. High-speed imaging cameras can capture the movement of these particles and produce high-resolution images. Tracking these particles on the images helps physicists understand the particles' physical properties and helps computational scientists enhance theoretical models for simulation development.

We model particles as local maximum points in each frame and track the points' movement across frames of the exploding wire imaging data. A frame of the time-varying data is shown in Fig. 8a. The test data have a 384×384 spatial resolution and $4,745$ timesteps. $3,197,333$ maximum points are detected from the data. The maximum points' trajectories are shown in Fig. 8b, which pass through the particles on the image.

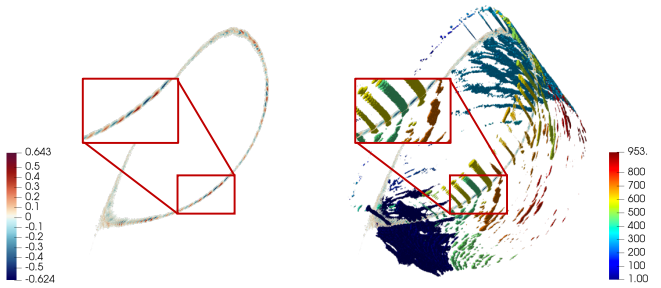
Fusion plasma simulations: A Tokamak torus device is the mainstream fusion reactor to confine plasma magnetically in order to achieve fusion energy production magnetically. The turbulent transport from the edge plasma usually takes a ubiquitous form of filaments, defined as density-enhancement coherent structures, also referred to as blobs. The blob movement may cause loss of plasma and severe damage to the device, and hence, has been subject to intensive researches [38–40].

We track blobs in ion density fluctuation data produced by a BOUT++ electromagnetic fluid simulation [41, 42] for the fusion reactor. A separatrix slice of the data is shown in Fig. 9a. Blobs usually are the regions of high ion density.



(a) (b)

Fig. 8: Tracking critical points in the exploding wire experimental data. The intensity value ranges from 0 to 255. (a): One image frame, where bright particles are detected as maximum points. (b): Trajectories of maximum points represented by black lines.



(a) (b)

Fig. 9: Tracking super-level sets in fusion plasma simulation data. (a): 2D density field at a timestep, where blobs are high-density regions and are detected as super-level sets. (b): Extracted super-level sets having 953 labeled connected components colored by unique hues.

Hence, following the work of [40], we model blobs as regions with densities larger than 2.5 standard deviation than the average density and track super-level sets across timesteps. The test data have 425×880 spatial resolution and 701 timesteps. $1,708,341$ high-density voxels are extracted from the data. The results are shown in Fig. 9b.

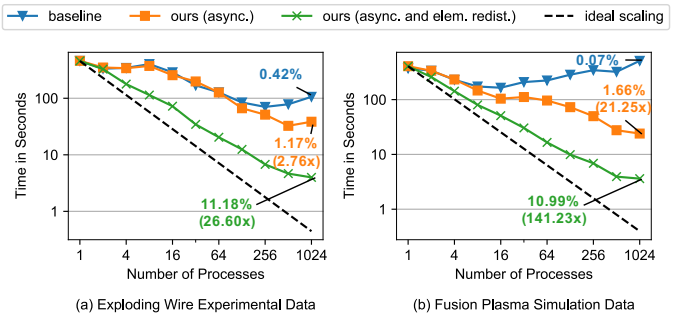


Fig. 10: Strong scaling of distributed union-find for tracking and extracting features in two application datasets: (a) exploding wire experimental data and (b) fusion plasma simulation data. Both axes are log scales. We compare a baseline (distributed union-find of Iverson et al. [13] with balanced mesh cells [5]) with our distributed asynchronous union-find without/with the redistribution of feature elements.

7.2.2 Benchmark in Scientific Applications

We demonstrate the performance of distributed union-find in the two scientific applications. Fig. 10 displays the strong scaling results. When $1,024$ processes are used, the bulk-synchronous baseline approach achieves strong scaling efficiency of 0.42% in the exploding wire data and 0.07% in the fusion plasma data. The asynchronous parallelism improves the efficiency to 1.17% and 1.66% with a speedup of 2.76x and 21.25x over the baseline, respectively. However, the scalability is still limited by the imbalanced features.

Evaluation of load balancing: We evaluate the load balancing of distributed union-find using the k-d tree based element redistribution in the two scientific applications. As visualized in Fig. 8 and Fig. 9, the features are not uniformly distributed in domain for both of the scientific applications. Hence, balancing mesh cells may not be effective enough for these cases, and balancing feature elements is expected to improve the distributed union-find performance. After redistributing feature elements to balance the number of features in each process, the strong scaling efficiency increases significantly to 11.18% and 10.99%, respectively. Compared with the baseline, our distributed asynchronous union-find with the feature element redistribution attains 26.60x speedup for tracking critical points in the exploding wire data and 141.23x speedup for tracking super-level sets in the fusion plasma data. Fig. 11 displays the cost breakdown of our distributed asynchronous union-find with the feature element redistribution.

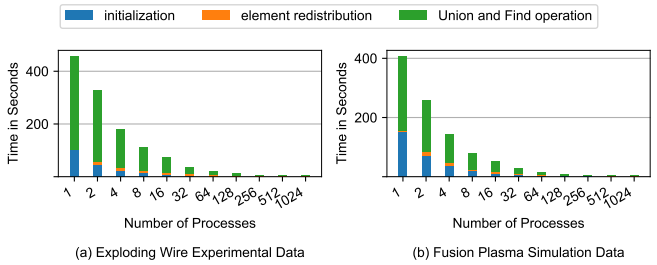


Fig. 11: Breakdown of the time cost of our distributed union-find algorithm with both the asynchronous parallelism and the element redistribution in two application datasets. The horizontal axis is a log scale. The "initialization" includes the initial assignment of element IDs and the initialization of data structures.

8 Discussions

We discuss limitations of our distributed union-find.

Limitation of element identifier: Our distributed asynchronous union-find assumes elements have unique and sortable IDs, which may not be the case for all datasets. In scientific applications using mesh data, we use IDs of mesh cells or mesh faces as elements' IDs, where the cost of such ID assignment is included in "initialization" of Fig. 11. However, for a graph structure without element IDs as input, an additional preprocessing for element ID assignment is required. A possible way to remedy this could be to collect element counts of all processes, compute a numeric ID range for each process, and each process assigns IDs to local elements in parallel within the ID range, which introduces additional cost.

Limitation of memory capacity: Out-of-core algorithms may be needed if the memory capacity cannot hold a single data block. A possible solution to remedy the limitation is described in the following. First, we decompose the data further into smaller data blocks with ghost layers such that the memory of each process is able to hold a single data block. Each process then loads a smaller data block. Second, each process extracts elements and edges of interest in the loaded data block to perform distributed union-find. Third, processes release current blocks and load new unprocessed blocks. We repeat the second and the third step until all data blocks are processed.

9 Conclusion and Future Work

This paper presents a novel distributed union-find algorithm that (1) overlaps communications with computations using asynchronous parallelism to reduce synchronization costs and (2) redistributes set elements using a distributed k-d tree decomposition to balance processes' workloads for scalable scientific visualization and analysis. Our algorithm demonstrated improved scaling characteristics than existing distributed union-find methods in the scientific applications of critical point tracking and super-level set extraction. Benchmark datasets included synthetic data, exploding wire experimental data, and fusion plasma simulation data.

In the future, first, we will evaluate our algorithm's performance in 4D (3D in space and 1D in time) scientific data. Distributed union-find algorithms can be extended to 4D seamlessly because the union-find algorithms accept elements and edges between elements as input, independent of the dimensionality of elements. Also, the element redistribution using distributed k-d trees can deal with elements with higher dimensions. Second, we will integrate our algorithm into other visualization and analysis applications, such as in situ visualization and graph/network data analysis.

Acknowledgments

The authors would like to thank Eric Brugger from Lawrence Livermore National Laboratory for providing the help of preprocessing fusion plasm simulation data. This work is supported in part by National Science Foundation Division of Information and Intelligent Systems-1955764, U.S. Department of Energy Los Alamos National Laboratory contract 47145, and UT-Battelle LLC contract 4000159447 program manager Laura Biven. The research is also supported by the Exascale Computing Project (ECP), project number 17-SC-20-SC, a collaborative effort of Department of Energy Office of Science and the National Nuclear Security Administration, as part of the Co-design center for Online Data Analysis and Reduction (CODAR) [43]. It is also supported by the U.S. Department of Energy, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program, and by Laboratory Directed Research and Development (LDRD) funding from Argonne National Laboratory, provided by the Director, Office of Science, of the U.S. Department of

Energy under Contract No. DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

REFERENCES

[1] H. Carr, J. Snoeyink, and U. Axen, "Computing contour trees in all dimensions," *Computational Geometry*, vol. 24, no. 2, pp. 75–94, 2003.

[2] V. Pascucci and K. Cole-McLaughlin, "Parallel computation of the topology of level sets," *Algorithmica*, vol. 38, no. 1, pp. 249–268, 2004.

[3] D. Morozov and G. Weber, "Distributed merge trees," in *Proc. ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 93–102.

[4] D. Morozov and G. H. Weber, "Distributed contour trees," in *Topological Methods in Data Analysis and Visualization III*, P.-T. Bremer, I. Hotz, V. Pascucci, and R. Peikert, Eds. Springer, 2014, pp. 89–102.

[5] 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*, 2015, pp. 3–19.

[6] J. E. McClure, M. A. Berrill, J. F. Prins, and C. T. Miller, "Asynchronous in situ connected-components analysis for complex fluid flows," in *Proc. Second Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, 2016, pp. 12–17.

[7] A. Friederici, W. Köpp, M. Atzori, R. Vinuesa, P. Schlatter, and T. Weinkauf, "Distributed percolation analysis for turbulent flows," in *Proc. IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, 2019, pp. 42–51.

[8] A. Nigmetov and D. Morozov, "Local-global merge tree computation with local exchanges," in *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 60:1–13.

[9] X. Tricoche, T. Wischgoll, G. Scheuermann, and H. Hagen, "Topology tracking for the visualization of time-dependent two-dimensional flows," *Computers & Graphics*, vol. 26, no. 2, pp. 249–257, 2002.

[10] C. Garth, X. Tricoche, and G. Scheuermann, "Tracking of vector field singularities in unstructured 3D time-dependent datasets," in *Proc. IEEE Visualization*, 2004, pp. 329–336.

[11] G. Cybenko, T. G. Allen, and J. Polito, "Practical parallel union-find algorithms for transitive closure and clustering," *International journal of parallel programming*, vol. 17, no. 5, pp. 403–423, 1988.

[12] F. Manne and M. M. A. Patwary, "A scalable parallel union-find algorithm for distributed memory computers," in *Proc. International Conference on Parallel Processing and Applied Mathematics*, 2009, pp. 186–195.

[13] J. Iverson, C. Kamath, and G. Karypis, "Evaluation of connected-component labeling algorithms for distributed-memory systems," *Parallel Computing*, vol. 44, pp. 53–68, 2015.

[14] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[15] Z. Galil and G. F. Italiano, "Data structures and algorithms for disjoint set union problems," *ACM Computing Surveys (CSUR)*, vol. 23, no. 3, pp. 319–344, 1991.

[16] Y. Shiloach and U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57–67, 1982.

[17] R. J. Anderson and H. Woll, "Wait-free parallel algorithms for the union-find problem," in *Proc. of the twenty-third annual ACM symposium on Theory of computing*, 1991, pp. 370–380.

[18] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using advanced MPI: Modern features of the message-passing interface*. MIT Press, 2014.

[19] 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*, vol. 70, pp. 25–43, 2017.

[20] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 456–471.

[21] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[22] D. Nguyen, A. Lenharth, and K. Pingali, *Distributed implementation of connected components in Galois system*, accessed July 2020, https://iss.oden.utexas.edu/?p=projects/galois/analytics/dist-cc.

[23] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 752–768.

[24] R. Dathathri, G. Gill, L. Hoang, V. Jatala, K. Pingali, V. K. Nandivada, H.-V. Dang, and M. Snir, "Gluon-async: A bulk-asynchronous system for distributed and heterogeneous graph analytics," in *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019, pp. 15–28.

[25] B. A. Galler and M. J. Fisher, "An improved equivalence algorithm," *Communications of the ACM*, vol. 7, no. 5, pp. 301–303, 1964.

[26] R. E. Tarjan and J. Van Leeuwen, "Worst-case analysis of set union algorithms," *Journal of the ACM (JACM)*, vol. 31, no. 2, pp. 245–281, 1984.

[27] J. van Leeuwen and R. van der Weide, *Alternative path compression techniques*. Unknown Publisher, 1977, vol. 77.

[28] T. P. Weide, "Datastructures: An axiomatic approach and the use of binomial trees in developing and analyzing algorithms," Ph.D. dissertation, Leiden University, 1980.

[29] J. E. Hopcroft and J. D. Ullman, "Set merging algorithms," *SIAM Journal on Computing*, vol. 2, no. 4, pp. 294–303, 1973.

[30] D. Morozov and T. Peterka, "Block-parallel data analysis with DIY2," in *Proc. IEEE Symposium on Large Data Analysis and Visualization (LDAV)*. IEEE, 2016, pp. 29–36.

[31] ——, *DIY: data-parallel out-of-core library*, accessed January 2021, https://github.com/diatomic/diy.

[32] ——, "Efficient delaunay tessellation through kd tree decomposition," in *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 728–738.

[33] H. Guo, D. Lenz, J. Xu, X. Liang, W. He, I. R. Grindeanu, H.-W. Shen, T. Peterka, T. Munson, and I. Foster, "FTK: A high-dimensional simplicial meshing framework for robust and scalable feature tracking," *arXiv preprint arXiv:2011.08697*, 2020.

[34] H. Guo, "FTK: Feature Tracking Kit," Oct. 2019, https://github.com/hguo/ftk.

[35] G. Ji, H.-W. Shen, and R. Wenger, "Volume tracking using higher dimensional isosurfacing," in *Proc. IEEE Visualization*, 2003, pp. 209–216.

[36] 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*, vol. 87, no. 11, p. 11D601, 2016.

[37] Z. Wang, J. Xu, Y. E. Kovach, B. T. Wolfe, E. Thomas Jr, H. Guo, J. E. Foster, and H.-W. Shen, "Microparticle cloud imaging and tracking for data-driven plasma science," *Physics of Plasmas*, vol. 27, no. 3, p. 033703, 2020.

[38] S. I. Krasheninnikov, "On scrape off layer plasma transport," *Physics Letters A*, vol. 283, no. 5-6, pp. 368–370, 2001.

[39] 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.*, vol. 93, p. 265001, Dec 2004. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.93.265001

[40] 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.

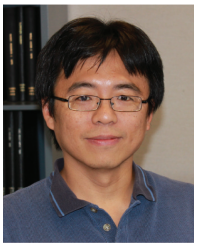
[41] X. Q. Xu, R. H. Cohen, T. D. Rognlien, and J. R. Myra, "Low-to-high confinement transition simulations in divertor geometry," *Physics of Plasmas*, vol. 7, no. 5, pp. 1951–1958, 2000.

[42] B. Dudson, M. Umansky, X. Xu, P. Snyder, and H. Wilson, "BOUT++: A framework for parallel plasma fluid simulations," *Computer Physics Communications*, vol. 180, no. 9, pp. 1467–1480, 2009.

[43] I. Foster, M. Ainsworth, J. Bessac, F. Cappello, J. Choi, S. Di, A. M. Gok, H. Guo, K. A. Huck, C. Kelly, S. Klasky, K. Kleese van Dam, X. Liang, K. Mehta, M. Parashar, T. Peterka, L. Pouchard, T. Shu, H. van Dam, J. M. Wozniak, M. Wolf, W. Xu, I. Yakushin, S. Yoo, and T. Munson, "Online data analysis and reduction: An important co-design motif for extreme-scale computers," *International Journal of High-Performance Computing Applications*, vol. in press, 2020.

Jiayi Xu is a Ph.D. candidate in the Department of Computer Science and Engineering at the Ohio State University. His research interests include graph visualization and scientific feature tracking. Xu received his B.S. degree in computer science and technology from Chu Kochen Honors College at Zhejiang University in 2014.

Hanqi Guo is an assistant computer scientist at Argonne National Laboratory, scientist at the University of Chicago Consortium for Advanced Science and Engineering (CASE), and fellow of the Northwestern Argonne Institute for Science and Engineering (NAISE). His research interests include data analysis, visualization, and machine learning for scientific data. He has published more than 40 research papers in top visualization journals and conferences including IEEE VIS, IEEE TVCG, and IEEE TPDS. He is also the recipient of the best paper award in IEEE VIS 2019 and the winner of the 2017 Postdoctoral Performance Award in Basic Research in Argonne National Laboratory. He received his Ph.D. degree in computer science from Peking University in 2014 and his B.S. degree in mathematics and applied mathematics from Beijing University of Posts and Telecommunications in 2009.



Han-Wei Shen is a full professor at the Ohio State University. He received his B.S. degree from the Department of Computer Science and Information Engineering at National Taiwan University in 1988, his M.S. degree in computer science from the State University of New York at Stony Brook in 1992, and his Ph.D. degree in computer science from the University of Utah in 1998. From 1996 to 1999, he was a research scientist at NASA Ames Research Center in Mountain View California. His primary research interests are scientific visualization and computer graphics. He is a winner of the National Science Foundation's CAREER award and U.S. Department of Energy's Early Career Principal Investigator Award. He also won the Outstanding Teaching award twice in the Department of Computer Science and Engineering at the Ohio State University.

Mukund Raj is a postdoctoral appointee in the Mathematics and Computer Science Division, Argonne National Laboratory. His research interests include ensemble visualization, scientific visualization, and in situ analysis. Raj received his PhD in computing from the University of Utah in 2018.

Xueyun Wang Xueyun Wang received her B.S. degree in 2015 and Ph.D. degree in 2020 from School of Physics, Peking University, Beijing, China. Her major interests include fluid simulation of plasma physics.

Xueqiao Xu is a principal physicist at Lawrence Livermore National Laboratory. His research interests include plasma physics and controlled nuclear fusion, plasma theory and large scale simulations, and machine learning for moment closures of kinetic equations. He has published more than 160 research papers in top plasma physics journals that include PRL, Nuclear Fusion, POP, PPCF, JCP, CPC, CiCP. Xu received his Ph.D. in physics from the University of Texas at Austin in 1990.

Zhehui Wang a physicist at Los Alamos National Laboratory (LANL). He received his B.S. degree from the Department of the Earth and Space Sciences at the University of Science and Technology of China in 1992, his M.S. and Ph. D degrees in Astrophysical Sciences (Plasma Physics) from Princeton in 1994 and1998 respectively. He joined LANL as a postdoc in 1998, and has been a staff member since 2001. One of his recent research interests is applied data science to problems in imaging of physical systems and particle tracking. He has authored and coauthored more than 100 peer-reviewed papers and holds six US patents in plasma accelerator, velocimetry, neutron detectors and X-ray cameras. He currently leads the high-speed imaging team funded by several DOE and internal LANL programs.

Tom Peterka is a computer scientist at Argonne National Laboratory, a scientist at the University of Chicago Consortium for Advanced Science and Engineering (CASE), an adjunct assistant professor at the University of Illinois at Chicago, and a fellow of the Northwestern Argonne Institute for Science and Engineering (NAISE). His research interests are in large-scale parallel in situ analysis of scientific data. Recipient of the 2017 DOE Early Career Award and four best paper awards, Peterka has published over 100 peer-reviewed papers in conferences and journals that include ACM/IEEE SC, IEEE IPDPS, IEEE VIS, IEEE TVCG, and ACM SIGGRAPH. Peterka received his Ph.D. in computer science from the University of Illinois at Chicago in 2007, and he currently leads several DOE- and NSF-funded projects.