Efficient cosmological friends-of-friends computation with fine-grained optimizations

Ritvik Rao*, Laxmikant Kale*
* University of Illinois Urbana-Champaign, Urbana, IL, USA
Emails: {rsrao2, kale}@illinois.edu

***Abstract*—One computationally intensive part of post-processing cosmological simulations is identifying clusters. One method to do this is via the friends-of-friends (FoF) algorithm, which is functionally equivalent to finding incremental connected components. Optimizing FoF for large, imbalanced datasets requires high precision. In this paper, we present a parallel approach that integrates a tree traversal library with a union-find library. While this combination provided a foundational framework, we discovered that it fell short of meeting the rigorous performance demands of complex cosmological data. To overcome this, we designed and implemented a series of fine-grained optimizations tailored to the unique memory and processing patterns of the algorithm. We demonstrate that our refined approach achieves significant performance gains for highly imbalanced outputs without compromising cluster accuracy or quality. Finally, we analyze these results and identify remaining opportunities for further improvement in scalable cluster finding.**

***Index Terms*—HPC, irregular applications, cosmology**

I. Introduction

Cosmological simulations describe the movement of millions to trillions of particles, each representing stars, galaxies, and dark matter halos, often simulating this movement over a time span of billions of years. Accurate simulations handle gravitational forces, smooth particle hydrodynamics (SPH) and magnetic forces, among other factors.

Simulations are generally divided into multiple time steps. Periodically, the simulation must be briefly paused for analyses to gather useful information, including information that may influence settings for the rest of the simulation. One such computationally intensive analysis is the identification of clusters of bodies in the simulated volume. Cluster finding is useful for studying the formation of galaxies, to quickly compare simulations with observations, and simply to count the number of clusters in each size category at a given point in time.

One approach to cluster finding is spherical overdensity (SO) [14], which defines clusters as spheres around points in the densest regions of the bounding box. While SO benefits from producing well-defined spherical clusters, the focus of this work is the **friends-of-friends (FoF) method**. In FoF, any pair of particles within a certain linking length ($\ell\ell$) is considered to be a part of the same cluster. By iteratively merging particles within $\ell\ell$, clusters also merge until a steady state is reached where no cluster is within the linking length of another. This allows FoF to identify clusters of any arbitrary shape. Despite its simplicity, inherent inefficiencies of FoF include load imbalance (when particle merges are concentrated in certain areas of the bounding box) and communication overhead (since merging clusters may require sending many small messages across a dataset partitioned across multiple nodes). This makes implementing FoF in parallel a challenge from both an algorithmic and a software perspective.

Frameworks reduce the software development burden, especially for irregular problems on parallel computers. We used:

- The Parallel Tree Toolkit (ParaTreeT [8]) is a framework for implementing parallel algorithms for particles organized in a spatial tree structure, particularly for non-uniform data with density variations ranging over several orders of magnitude
- Union-Find is a graph algorithm library [11] that provides a distributed implementation of the well-known inverted-tree distributed union algorithm. FoF is functionally equivalent to the incremental connected components problem, where particles are equivalent to union-find vertices and clusters are equivalent to inverted trees.

Both of these libraries as well as the FoF application are written using the Charm++ parallel programming system, meant for dynamic and irregular applications.

This paper culminates in a highly efficient parallel implementation of the FoF algorithm. We present the algorithm as a series of optimizations to a baseline parallel algorithm implemented using both the above frameworks. It illustrates how the frameworks needed to be extended in order to accommodate the optimizations, which led to improved versions of each framework in the end. The optimizations themselves are instructive for the varied toolkit required to solve irregular problems at scale.

This paper makes the following contributions:

- A description of the ParaTreeT framework and Union-Find library in the next section, and the first naive version of the FoF implementation using them
- An analysis of this implementation's inefficiencies, leading to a modified design with an extension of the ParaTreeT framework known as ParaTreeT2

This research used the Anvil machine at the Rosen Center for Advanced Computing at Purdue University [20]. This research also used resources of the Oak Ridge Leadership Computing Facility, a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. Thank you to Tom Quinn at the University of Washington for providing data sets and details on how cosmologists use FoF.

- A series of optimizations demonstrated on scaling for two highly clustered datasets, with 80M and 2B particles, respectively, up to several thousand processors

II. Frameworks used and extended

A. Charm++

Charm++ [9] is a task-based parallel runtime system built on migratable objects called chares and on asynchronous, message-driven execution. Chares allow a program to be decomposed independently of the number of processors and ranks, which in turn allows dynamic load balancing by moving chares between processors based on load metrics; message-driven execution overlaps computation with communication automatically. Both properties matter for irregular, fine-grained applications (e.g. single-source shortest paths [17]).

B. ParaTreeT

The Charm N-Body Gravity Solver (ChaNGa) [13] is a long-standing application for N-Body simulations that has demonstrated near-perfect strong scaling up to 128K cores on imbalanced inputs. ChaNGa's tree representation and traversal code could be useful for any tree-based workloads, which motivated the creation of ParaTreeT [8], a tree traversal library designed for any tree-based workload, including N-body gravity. ParaTreeT also borrows innovations from ChaNGa, such as a shared software cache among threads in a process to reduce remote communication of tree pieces during traversal, and extends it to support a highly efficient atomics-based tree-cache that is critical for large shared-memory processes.

The current ParaTreeT implementation does not fully reach the ideal of solving diverse tree problems with optimal performance, in part due to design choices influenced by N-body gravity. The following sections describe the challenges presented by ParaTreeT's abstraction boundaries, our approach to addressing them, and the effects on FoF performance.

C. Union-Find

Union-Find (aka disjoint-set union) computes incremental connected components on an undirected, unweighted graph. Its vertices are partitioned into disjoint sets, each represented as an inverted tree: every vertex has a parent pointer to another vertex but no pointers to its children. As each edge is processed, a union operation finds the roots of the inverted trees to which its two endpoints belong and sets one root's parent pointer to the other, merging two previously disconnected components. FoF is an instance of this problem: two particles within distance $\ell\ell$ are an edge; merging their clusters is the same as merging the two components that the edge joins.

A parallel implementation of Union-Find has been developed with Charm++ [11]. Vertices are partitioned among chares in the Union-Find library, and inverted trees are merged in parallel and concurrently with other merge operations, with careful handling to avoid race conditions, and the use of `TramLib` [3] to aggregate short messages. The current Union-Find approach, while highly parallel, still requires some improvements. Some of these are identified and added to the library as a part of this paper.

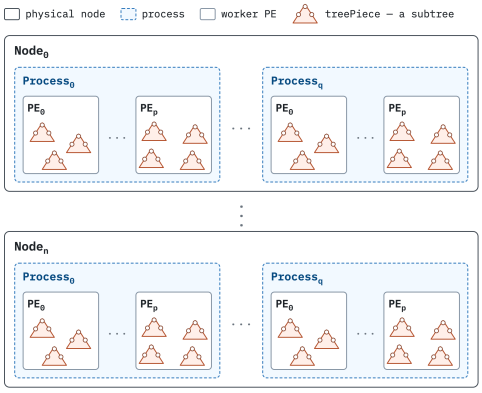

Fig. 1. Machine Hierarchy with Nodes, Processes, PEs, and TreePiece Chares

III. Algorithm design

We next illustrate our implementation of FoF using ParaTreeT, ParaTreeT2, and Union-Find. We implement a series of optimizations, each of which builds on top of the previous change, and demonstrate the effect of each change with results.

A. Machine model and decomposition

The main decomposition used is a collection of Charm++ objects (*chares*) of type TreePiece. Consistent with Charm++'s over-decomposition strategy, there are many TreePieces per worker core (called PEs in Charm++). Many PEs form a process, and there are typically multiple processes in a host (a physical compute node). These TreePieces are created by splitting particles with an oct-tree decomposition.

B. Basic Idea

A concrete specification of our Friends-of-Friends cluster finding algorithm is: given a set of particles in a periodic box, and given a linking distance $\ell\ell$, (a) label each particle with the component it belongs to and (b) count the number of *clusters* (defined below) in each bin of a histogram, where i'th bin holds counts of clusters whose size s fall between $2^i <= s < 2^{i+1}$. Two particles are said to be *friends* if the distance between them is less than $\ell\ell$. The *cluster* containing a particle p is the transitive closure of the *friends* relation starting at p. Every particle lies in exactly one cluster, and a particle with no friends is a cluster of size one.

Given the existing ParaTreeT and Union-Find parallel libraries, a parallel algorithm for cluster-finding suggests itself: use ParaTreeT to conduct, from each leaf (a leaf is also called a bucket and typically includes 12 or fewer particles in ParaTreeT), a walk that ascends to the root and descends again into every subtree the opening criterion admits, excluding any node whose box is too far away to contain a particle near one of the bucket's particles. For each pair of particles found to be within the linking distance, emit an edge to the distributed Union-Find library (i.e. call its `union(v,w)` function), which can concurrently run its union algorithm. When the tree walks and Union-Find are done (identified by

quiescence detection in Charm++), the cluster is formed as a distributed inverted tree. That is all, almost!

There is a little more work to actually create the histogram. Label each global root with a unique contiguous number (by doing a prefix sum over the counts of roots on each PE). Then, traverse the inverted tree so you can find the root label for every component, label every particle with the label of its root, do a reduction over components to get the size of each component at the root (typically PE 0), and then calculate the signature by binning the component sizes in logarithmic bins.

Given the libraries, the new code to be written is small: specifying the opening criteria in the traversal's visitor, and writing the `leaf()` function that ParaTreeT's traversal calls to allow the application to process interaction between particles of two leaves, to emit edges between particles of a pair of leaves with explicit distance calculations, in addition to the labeling and counting phases mentioned above.

This approach produces correct results, but the overall execution time is still disappointing. Not only does performance fall short of other FoF implementations [15], but the total execution time is slower than one iteration of an N-body gravity simulation on the same data set, meaning that the end-to-end execution time of a simulation is dominated by FoF. The next sections demonstrate how this performance can be greatly improved, but it requires a number of fairly sophisticated optimizations.

C. Analysis of ParaTreeT and Union-Find

An analysis of the code running on an 80M particle data set shows that the number of edges submitted to Union-Find globally is about 6.5 billion. Can it be reduced?

Consider what happens when 2 particles are found to be within linking distance. The Union-Find library finds the current roots of each associated vertex. This process can be shortened if we regularly compress at least PE-local paths. This optimization improved performance only a little bit, because following local chains was fast compared with messaging anyway. However, it reveals an insight: if there is a remote particle r that happens to be close to local particles x and y, and x and y belong to the same cluster locally, we still ask Union-Find to `union(`x`,`r`)` and `union(`y`,`r`)`, which lead to both following the same path to the root of z and the root of r. It is clearly redundant, but the library does not have a way of eliminating it by itself. If the FoF visitor follows local chains to the common local ancestor of x and y (say z), only submits edge (z, r) to Union-Find, and uses a hash-map to avoid duplicate submissions, the number of edges submitted can be substantially reduced. Strategies such as this to avoid duplicates are called *de-duplication*.

De-duplicating within a chare reduces the edge count somewhat. Extending this to PE-level tips (by crossing TreePiece boundaries when it is on the same PE while following parent pointers) only modestly reduces edge count. Process-level tips are the natural next step, since the PEs of a process access the process-local particles through shared memory. However, this introduces the possibility of data races, which is remedied with a 2-phase approach: traverse the edges crossing PE boundaries to collect each vertex's process-level tip into a map, then (after a barrier) have each PE point its particles' parents at that tip. This has to be done while no other activity concurrently modifies the parent pointer, so data coming from the cache must be paused.

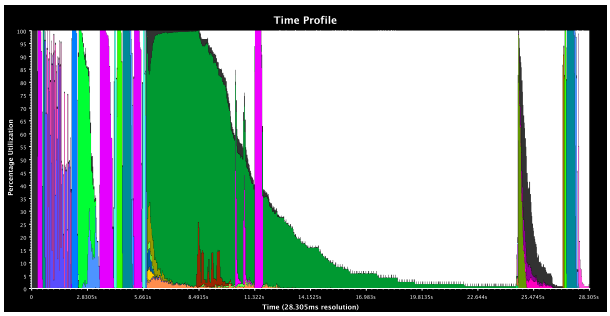

Fig. 2. Time profile with triggered stop-everything phases (pink) compressing process-local paths, inside the tree traversal (green), on 120 cores. White is idle, black system overhead

This stop-everything structure is forced by the framework. We would rather compress after process-local work is done and before remote work begins, but ParaTreeT only drives the computation through the visitor and leaf functions we supply, not pausing or reporting when the local portion has completed. We can only stop the traversal repeatedly at heuristically chosen intervals. Fig. 2 shows the result as a *time-profile* from the Projections [10] performance visualization tool. We use this view repeatedly, so we describe it once here.

A time-profile is a stacked bar-graph. Each bar is one time interval; each colored segment within it is one function (an entry method in Charm++), its height being that function's time summed over all processors and expressed as a percentage of the interval's duration. A full bar therefore means every processor was busy throughout the interval. White is idle time and black is system overhead such as scheduling and communication, so white and black are the quantities to read. Here green is the tree traversal, and the narrow pink bars inside it are the stop-everything phases. Utilization is high only initially before decaying into a long tail, indicating a severe load imbalance.

Since the pauses were expensive, we also tried splitting the work into two traversals, the first ignoring any across-process interaction in the leaf function so that process-local tips are complete before remote work begins. Fig. 3 shows the two traversals cleanly separated, but there is a clear slowdown with this change, from a 33.6s end-to-end time to 47.3s. This is because forcing local interactions by using the same traversal logic and ignoring remote interactions is wasteful and cancels out the benefit of a 2-phase approach.

Even after these changes, the number of edges decreased only modestly. Process-tip de-duplication helped, but it captured only half the available benefit: For a candidate pair (p, q) we de-duplicate on (p_{tip}, q), where p_{tip} is the process tip of the local particle p; the ancestor q_{tip} of q lives on a remote process

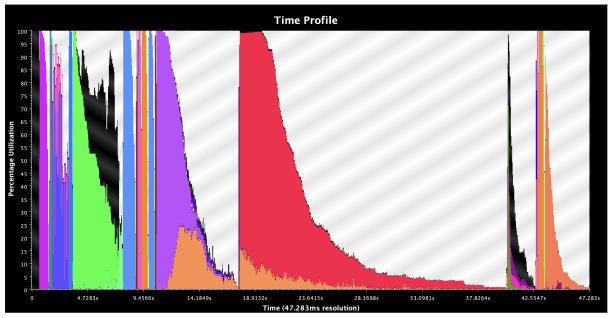

Fig. 3. Time profile of two traversals, local-only (green) then remote, on 120 cores. The phases separate cleanly, but the run is slower

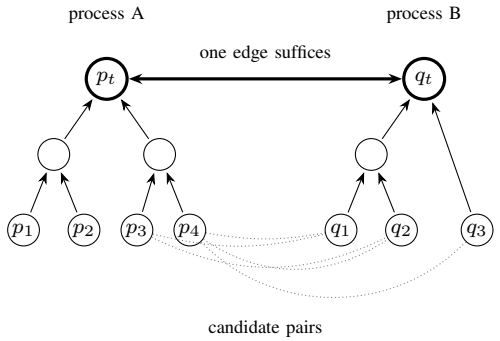

Fig. 4. Two inverted trees whose tips are on different processes. Every candidate pair found by the traversal (dotted) stands for the same edge between the tips (bold)

and is invisible to us. But every pair drawn from the local and remote inverted trees yields the same (p_{tip}, q_{tip}) edge, which needs to be inserted only once. De-duplicating on (p_{tip}, q) collapses one side of this cross-product, leaving one edge per remote particle. (p_{tip}, q_{tip}) would collapse both, leaving one edge per pair of trees. On clustered data, the difference is large. Fig. 4 illustrates the two collapses.

IV. ParaTreeT2

Eliminating most duplicate edges requires separating a process-local phase for calculating process-tips (and freezing them) followed by a distributed tree walk and doing distributed union-find for edges that cross process boundaries. This requires allowing the tree walk to ship an additional field (remote process tip, the q_{tip} in the example above) with each particle and to also support traversals confined to local process only.

We started with a new (forked) version of ParaTreeT called ParaTreeT2, which was templated on the client's datatype and supports local-only traversals. As in ParaTreeT, we use an oct-tree decomposition to build the particle tree.

The process-local phase (see Fig. 1) has 3 subphases:

- **Phase A** processes all the edges among vertices within the same PE with an interleaved sequential union-find, and computes PE-level "tips" for each vertex. This has 2 sub-phases: processing within a TreePiece, with a self-interaction walk, and processing pairs of TreePieces that both belong to the same PE. A final path-compression pass ensures that every vertex points to a PE-level tip.[1]
- **Phase B** handles pairs of TreePieces that belong to the same process but different PEs with a specialized tree-walk to skip both remote and within-PE pairs, with each pair assigned to a single PE. Phase B only *identifies* cross-PE edges: pairs of PE-level tips are accumulated in a list owned by the executing PE.
- **Process-level tips:** After Phase B, the edge-lists created by each PE are merged by one worker, which processes them sequentially (via sequential union-find) to identify *process level tips* for each process-level fragment (A *fragment* is a connected component or cluster, but confined within one process). Note that this union-find only changes parent pointers of vertices that are PE-level tips. Each PE then loops over all particles it owns, and makes them point to their process-level tip. The PE-level tips that were not involved in across-PE edges are naturally promoted to process level tips at this point.

The remote phase now begins with a global dual-tree walk. The TreeCache is initialized with the upper portion of the tree (i.e. all the nodes above the roots of TreePieces.) Whenever two particles (p_{local}, p_{remote}) are found to be within the $\ell\ell$, an edge between their process tips (p_{tip}, q_{tip}) is (if not duplicate) passed to the union-find library in batches. The distributed Union-Find library computes the global union and makes each process-level tip point to the global root of its component.

After this traversal, component sizes are binned in a log-scale histogram (so each component with size s, $2^i <= s < 2^{i+1}$, increments the ith bin of the histogram). The binning is done sequentially at the root. Finally, a unique number is assigned to the parent of each component and the labels are broadcast to all particles in the component.

This is a complete algorithm which implements full de-duplication. The idea of doing all the local unions first is not new to cluster finding; grid-type algorithms that exchange a ghost layer do it. What a ghost layer does not need, and a tree walk does, is a way to *name* the local component a remote particle should join: unless the walk ships each particle's tip alongside its position, it can collapse only the near side of the cross-product in Fig. 4, not both. Furthermore, we seek to illustrate the issues that arise from composing two parallel frameworks for irregular problems.

This formulation exposes and enables many optimizations that we describe next. For each optimization, we provide a description of the optimization followed by an analysis of its effect on performance. We conclude with a summary of the overall progression of these changes. We used multiple datasets to validate the optimizations. For convenience of illustration, we focus on how each optimization affects a highly clustered dataset of 80 million particles called lambb.00500. We also show scaling results for our final implementation on a 2 billion dataset called cosmo25cmb.768g2_dm.001024.

[1]Throughout, a vertex may be its own tip, and a component may be a single particle; we do not repeat these qualifications.

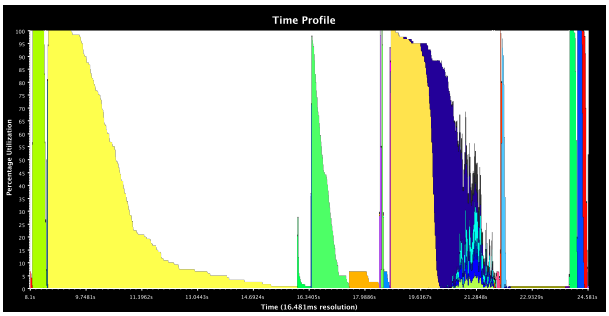

Fig. 5. Time profile of the initial version of ParaTreeT2 with two phases on 120 cores

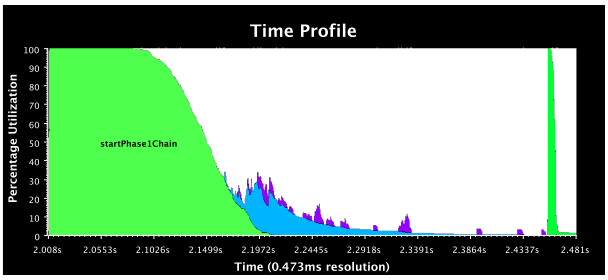

Fig. 6. Time profile of the local phase: phase A (green) is balanced, but phase B (blue) has a long tail. White is idle time

Most of the results were obtained with the use of Purdue Anvil. Each Anvil node has 128 cores and 256 GB of memory, and Anvil uses an Infiniband interconnect with 100 GB/s bandwidth. Scaling results for the 80 million particle set in Fig. 13 were obtained on Frontier, which has 64 cores and 128 SMT threads per node, with 512 GB of memory.

V. Optimizations

A. Dual Tree walks

The original formulation was based on the idea that the walk used for SPH (where one looks for k nearest neighbors for each particle) is adequate for the task of finding particles within a fixed distance ($\ell\ell$), which we need. Each bucket (which is the leaf of the tree that holds 12 or fewer particles) originates its own walk for this purpose. Although correct, this is far from optimal because it forgoes opportunities that arise when the distance is constant (unlike SPH). We therefore change this to a dual tree walk, originating with a TreePiece (rather than the root, because a TreePiece is guaranteed to reside fully on one PE). As we recursively descend to subtree pairs, any pair whose bounding boxes are sufficiently far apart can be pruned, removing the need to traverse further down the tree. On a single-process run, this led to a reduction by a factor of 20 overall, because it avoided a tree node interacting with each leaf. The speedup decreases with the number of processes, because multiple processes spread the same set of leaves across processors, thus reducing redundancy.

B. Reducing redundant work in the local phases

Several optimizations were developed that prune the search for edges during the PE-local and process-local dual-tree walks. These are enabled by removing the shackles that the old framework imposed (and by using PE-local and process-local data structures, which were not forbidden by old ParaTreeT, but there was no real reason to use them unless the walks could be separated into localized phases).

A quick list of optimizations: A walk over a pair of tree nodes whose (oct-tree, so cubic) boxes are farther apart than the linking length is discarded. If the boxes of a pair of nodes (or of a single leaf, a special case encountered during the self-walk) are fully contained in a cube of side $\ell\ell$, then all of their particles belong to one component and are made to point to a single representative in $O(1)$ time per particle. Both nodes are marked "uniform" (i.e. everything under them belongs to one component) and point to that representative. During the recursive descent of pairwise walks, if a node is encountered whose children are all uniform and share the same representative, the node is itself marked uniform and made to point to that representative. The recursive walk descends on one side only (the larger box) rather than both sides, i.e. 8 recursive calls instead of 64.

During pair-walks in phase B, when particles p and q are found to be within linking distance, their PE-level tips (p_{petip}, q_{petip}) form the new edge. It is de-duplicated by checking whether it already exists in a process-level table. Further, by tracking the representatives of the *uniform* nodes, recursive calls on node pairs can be discarded without having to traverse to the leaf level.

These (and a few more) optimizations cut the end-to-end time on 120 cores from 12.3 s to 5.2 s, the largest single step in this sequence, and reduced the max/average variance (load imbalance) as well.

C. Process-level Load Balancing

In the local phase, on 480 PEs, Fig. 6 shows that although phase A (green) has been optimized and its variance reduced, phase B (blue) has a long tail, indicating a straggler or load imbalance. A compacted timeline view (Fig. 7) confirms it. In that view, which we use for the rest of the paper, each horizontal line is one PE and each colored block is one entry method executing on it, with white for idle time. It is indeed a few straggler PEs (blue streaks) making everyone wait.

Local Phase B performs dual tree walks for each pair of TreePieces that resides on the process. The output is a list of PE-to-PE edges, without any side effects, so it does not matter which PE processes a pair. We therefore create a pool of pairs using Charm++'s `NodeGroup` construct and have each PE pull tree-piece pairs from it. As can be seen in Fig. 8, the thick blue lines (spanning 15 PEs of a process) indicate that work is being spread within a process. The few remaining stragglers suggest that some TreePieces are too large, which led to a modification that further splits subtree pairs into independent tasks, leading to a satisfactory outcome (Fig. 9). The total time for phases A and B thus decreased from 0.45s to 0.3s.

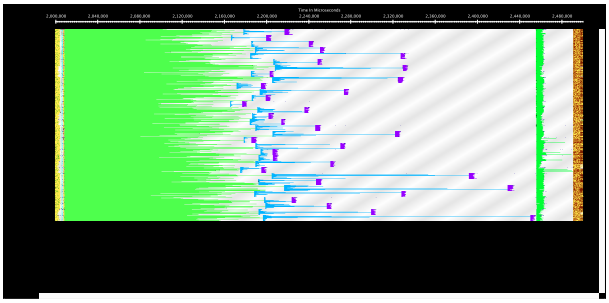

Fig. 7. Timeline of the local phase before balancing. Each row is one PE, each block an entry method, white is idle; phase B (blue) runs on a few PEs while the rest wait

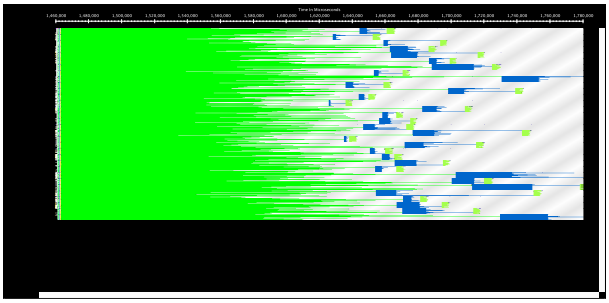

Fig. 8. The same view with the work pool: phase B (blue) now spans the 15 PEs of a process, but a few stragglers remain

D. Distributed dual tree walk and Union-Find

After the local phases, every connection within a process has been found and each particle points at a process-level tip, so the remote phase need only produce edges between tips on *different* processes. It is again a dual tree walk originating at each TreePiece, but the opposing side may be owned by any process and is descended through ParaTreeT2's software cache, which fetches a node on a miss and resumes the walk when it arrives. The attributes stored for each node (as described in the previous subsection, including uniformity and, for a uniform node, a pointer to the process tip signifying that all its children belong to the same fragment) carry over, and once one witness (a pair of particles within $\ell\ell$) has been found for a pair of tips, a process-level table suppresses every remaining search for that pair. Particles and nodes sent to the cache are slimmed to the two fields the walk reads, position and tip, twenty of their roughly 112 bytes.

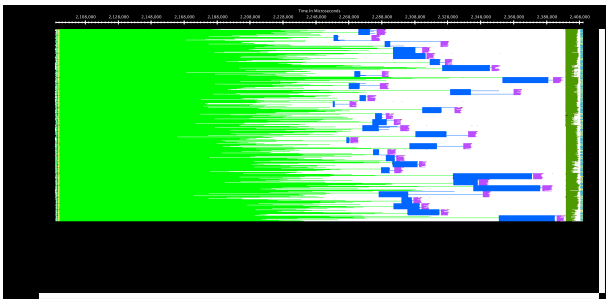

Fig. 9. The same view once subtree pairs are split into smaller tasks as well: phase B (blue) covers every PE

The two endpoints of an edge are by construction owned by different processes, so the distributed Union-Find must route a request for any tip to its owner. We make the identifier carry its owner: a tip becomes $(process \ll k) \mid order$, where $order$ is the global order of the fragment's minimum-order particle. Decoding becomes a shift, so routing is $O(1)$ arithmetic. The renumbering must happen before the walk rather than after, since the walk reads tips directly off cache-shipped copies, and a process holding an edge to a remote tip cannot otherwise learn how that tip's owner encoded it.

At this point, some empirical data is useful to motivate the design. For 80M particles, running on 240 cores organized in 16 processes, there were roughly 23.7 million process-level tips. Yet only 66,000 edges were inserted between these process-level tips; this was the power of process-level de-duplication. The remaining 23.6 million fragments were each completely contained within their own process.

The Union-Find library itself required each process to declare its vertices up front as a dense array, statically sized to hold one vertex for every tip that process owns. This meant that all 80 million vertices were allocated across the run, yet the vast majority were never included in one of the 66,000 edges, because local, within-process edges were never sent to the Union-Find library. We therefore added a lazy storage mode in which vertices live in a hash map, created on first touch and keyed by the decoded local part of the identifier. Every remaining cost in the distributed Union-Find then scales with the tips an edge reaches rather than with the tips that exist. To retain the integrity and utility of the Union-Find library for other applications with dense graphs, this "lazy" vertex scheme was made an opt-in option; dense storage is retained for clients whose vertices are mostly touched.

The same collapse in edge count removes the need for two mechanisms that the original composition depended on. Charm++ allows the Union-Find library to process edges, communication included, while the tree walk is still producing them, and with billions of edges this overlap was essential: without it, the edges would have been prohibitive to store, never mind to communicate. We retain it, releasing edges in batches of 4096, but at a few tens of thousands of edges it buys nothing measurable over simply waiting for the walk to finish. The same is true of `TramLib` [3], the message aggregation library that Union-Find uses to amortize the overhead of short messages at some cost in latency: it is now only marginally useful even at 2B particles, and we leave it as an option that our reported results do not use.

Together these changes take the end-to-end time from 4.6 s to 2.2 s, and the labeling phase from 2.1 s to 0.3 s.

E. Avoiding the local walk, and symmetry

Two further observations lead to optimizations that allow us to skip some work during the remote walk. Both are expressed in ParaTreeT2 as opt-in traits on the visitor, so a traversal that does not declare them is unaffected.

The first is that the remote dual walk should not be walking local data at all. The dual walk enumerates every pair of subtrees whose boxes are within $\ell\ell$, but if both subtrees are on the same process, they have already been dealt with by the local phases: every pair of local particles within $\ell\ell$ has been connected, so both carry the same tip. The traverser can therefore discard a pair consisting of locally owned nodes (in practice subtree roots, since it never descends below them).

The second is that the walk discovered every edge twice. A pair of TreePieces (A, B) on different processes is reachable from both ends during the dual walk, once on A's owner and once on B's, and since the edge a pair produces does not depend on which end processed it, both walks find it. Keeping each TreePiece-pair on one owner halves the pair work, the remote fetches, and the emitted edges together. The tie-break needs care: awarding every TreePiece pair to the lower-indexed piece is correct but leaves low-indexed pieces holding all of their counterparts and high-indexed ones none. We instead break the tie on the parity of the index sum, so each piece keeps about half of its counterpart set. Self pairs are never dropped.

F. Cross-Process Component Separation

When the distributed Union-Find finishes, a recursive backward pass propagates each cluster's label to the process-level tips and thence to every particle, and all that remains is to histogram the component sizes. This phase had become one of the more expensive in the run, partly because everything around it was by now well optimized, but also because the encoding that made the earlier phases fast made this one slow: component IDs are no longer contiguous integers, since they pack a process and a vertex number, so sizes cannot simply be added up by a vector reduction. Custom reductions combining hash maps at the interior nodes of the spanning tree, a `gather` of every local count to one processor, and materializing a contiguous vector were all expensive at the 23.7 million components of our exemplar 80M dataset.

The way out involved realizing that the vast majority of those components are contained within a process, and the lazy vertex scheme already marks which ones. Their sizes are known from process-tip formation, so each process bins them directly into a local log-scale histogram. Since there are only a handful of logarithmic bins, combining these histograms involves just a reduction over a short vector. Only the components that straddle processes need to be gathered at the root, where they are summed, binned, and added in. This (combined with the distributed dual tree walk changes) reduced the phase from several seconds to a few milliseconds.

G. Capstone Results

Fig. 11 summarizes the cumulative effect of the sequence on one Anvil node (120 cores in 8 processes, leaving 8 cores out for OS): the end-to-end time for 80M particles falls from 15.1 s to 1.18 s. The two largest steps are the redundant-work prunes of Section V-B and the distributed walk with lazy Union-Find

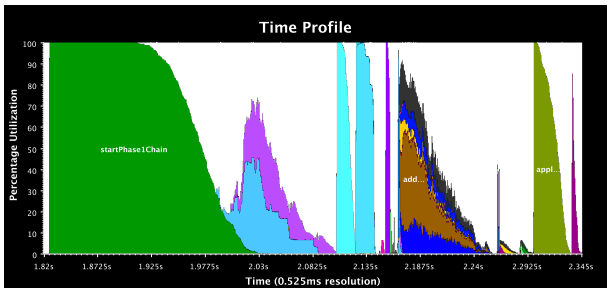

Fig. 10. Time profile of the entire cluster finding algorithm for 80 million particles on 480 cores

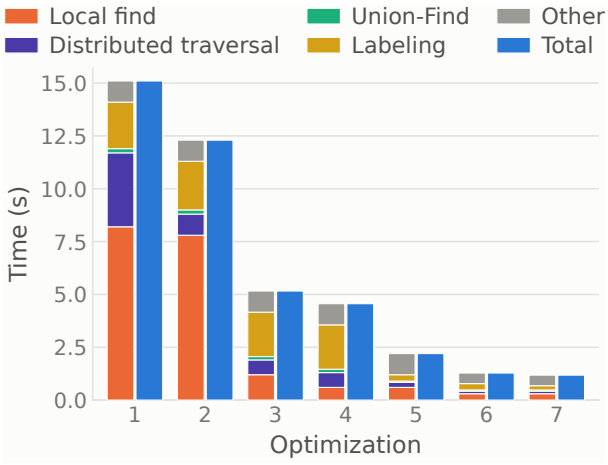

Fig. 11. Breakdown of how optimizations affect FoF on 80M dataset. Key phases (focuses of optimizations) are separated out. All runs are on 1 node (120 PEs) on Anvil. 1=from section IV, 2=V-A, 3=V-B, 4=V-C, 5=V-D, 6=V-E, 7=V-F

of Section V-D. The *Other* segment is work the measured phases do not cover and none of the optimizations touch it.

The final time profile for the 80M lambb.00500 dataset on 480 cores (Fig. 10) shows the whole run taking about half a second. On our challenge dataset of 2B particles (cosmo25), we obtained 12.3 seconds on 16 nodes and 1,792 cores of *Frontier* (Fig. 12). Phase B (light blue in the middle) dominates: one process has far more PE-to-PE tips within $\ell\ell$ than the others, due to imbalance in the particle decomposition. The remaining phases are balanced, and the tree traversal and Union-Find phases (26.6 s to 27.6 s) are short as well, reflecting the traversal optimizations.

Fig. 13 shows how lambb.00500 scales from 1 to 16 physical nodes (112 cores used per node). For 1 to 4 nodes, scaling is almost perfect since Phase A dominates the execution time. At higher node counts, scaling is slower as phase B and the remote tree traversal constitute more of the execution time. The remote walk also scales well from 1 to 4 nodes before settling at 150 ms; we believe that this is due to the reduced amount of available parallelism in the data set at nearly 2K cores. Phase B scales little, due primarily to imbalance: its balancing is largely within-process, so further scaling requires global load balancing in this step.

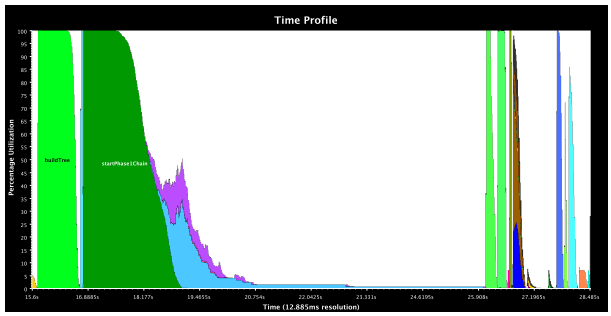

Fig. 12. Time profile of 16 node run (1792 cores) on *Frontier*, 2B particles 8 processes per node

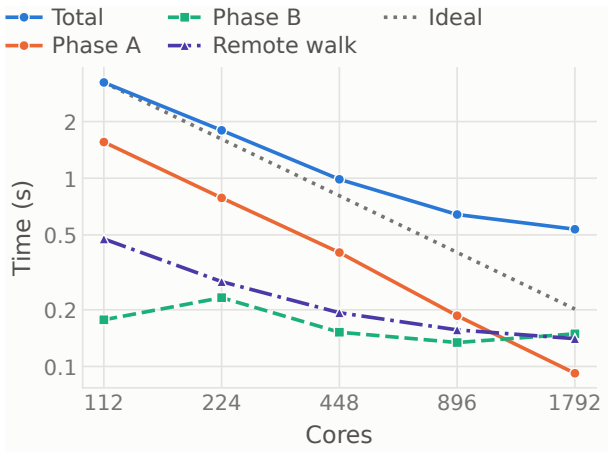

Fig. 13. Strong scaling of the 80M particle dataset on *Frontier*, 8 processes per node. Ideal scaling is anchored at the 8-node total.

VI. Related Work

Cluster finding, also known as halo finding, has many approaches. Beyond FoF and the spherical overdensity methods mentioned in Section I, of which ASOHF [14] is a representative adaptive implementation, density-peak methods such as HOP [4] and its parallel successor [19] grow clusters by hopping along a density gradient toward local maxima, while SUBFIND [22] refines an existing halo into gravitationally bound subhaloes and ROCKSTAR [2] clusters in six-dimensional phase space to separate structures that overlap in position alone. These methods recover richer information than FoF, but they are also more expensive, and the halo-finder comparison project [12] found broad agreement on bulk halo properties across all of them. Recent finders continue to combine techniques [1]. FoF therefore remains the standard first pass, and is the operation whose cost dominates postprocessing.

Most parallel FoF implementations share a common structure: partition the volume spatially, replicate a boundary region of width at least the linking length into each neighboring partition, run a fast serial FoF locally, and reconcile the clusters that straddle partitions afterward. pFoF [18] follows this pattern with a cubical decomposition and an explicit merge step, and the halo finder used in HACC [6], [23] overloads each rank with a sufficiently wide border that the merge is reduced to eliminating duplicates, confining communication to a single phase. GADGET-4 [21] likewise couples a distributed FoF pass to SUBFIND. The appeal of this structure is that it isolates communication, but it also fixes the granularity of work at the partition, which is a poor fit for the density variations of clustered data: the partition containing a large halo determines the runtime, and the replicated borders grow with the linking length.

Serial performance has been attacked separately. Feng and Modi [5] reformulate FoF as pair enumeration over a dual KD-tree and prune subtree pairs that are already known to be connected, reducing the operation count by close to two orders of magnitude relative to a linked-list implementation. This dual-tree formulation is the same one we adopt in ParaTreeT2, though we apply it in a distributed setting rather than within a partition. A separate line of work trades exactness for speed, using relaxed linking lengths, cubical rather than spherical neighborhoods, or stochastic subsampling of the particle set [5]; jFoF [7] goes further and relaxes the discrete cluster assignment itself so that gradients can be propagated through it. Our work computes exact FoF, so these approximations are complementary rather than competing.

On GPUs, the prevailing approach is to treat FoF as a specialized version of DBSCAN. ArborX [15], [16] fuses the neighbor search with cluster construction over a bounding volume hierarchy, treats the merging as a union-find over the resulting edges, and is used for in-situ halo finding in HACC at exascale. jFoF [7] is GPU-native throughout, avoiding host-device transfers entirely. Our approach is similar to ArborX and jFOF in using Union-Find combined with a tree traversal, but we focus on particularly imbalanced and clustered data where cross-partition communication is inevitable. We believe that for our inputs, it is best to use CPUs through fine-grained asynchronous over-decomposition and efficient communication rather than through bulk data parallelism.

VII. Summary and Future Work

We introduced a new friends-of-friends application using ParaTreet2, Union-Find, and Charm++. We showed how several optimizations enabled by these software packages speed up FoF for clustered datasets. Parallel frameworks for irregular problems can be fragile. They need to be extended, templated, and parameterized to allow for their use in multiple settings as intended. This work has also ended up extending ParaTreeT2 and Union-Find libraries.

Our next steps are to further improve on load-balanced optimizations by dynamically re-balancing the data after initial decomposition using Charm++. That is where the load imbalance remaining in the 2B particle run on 16 nodes is most likely to be addressed. The tradeoff between balancing and data migration costs over the course of a whole N-Body simulation is worth exploring. We also plan on experimenting with the effect of using GPUs to replace process-level Union-Find operations, which can provide speedups but may increase the costs of data migration to and from device. Finally, we

want to expand our implementation to include sub-cluster identification, where a second, smaller linking length is defined and sub-clusters are identified with this new parameter. That approach requires further dividing our algorithm into multiple (possibly recursive) phases.

REFERENCES

[1] Kirk S. S. Barrow, Thinh Huu Nguyen, and Edward C. Skrabacz. Solving Milky Way-sized systems with Haskap Pie: A halo finding algorithm with efficient sampling, k-means clustering, tree-assembly, particle tracking, python modules, inter-code applicability, and energy solving. *arXiv preprint arXiv:2505.22709*, 2025.

[2] Peter S. Behroozi, Risa H. Wechsler, and Hao-Yi Wu. The ROCKSTAR phase-space temporal halo finder and the velocity offsets of cluster cores. *The Astrophysical Journal*, 762(2):109, 2013.

[3] Kavitha Chandrasekar and Laxmikant Kale. Shared memory-aware latency-sensitive message aggregation for fine-grained communication. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 682–687. IEEE, 2024.

[4] Daniel J. Eisenstein and Piet Hut. HOP: A new group-finding algorithm for N-body simulations. *The Astrophysical Journal*, 498(1):137–142, 1998.

[5] Yu Feng and Chirag Modi. A fast algorithm for identifying friends-of-friends halos. *Astronomy and Computing*, 20:44–51, 2017.

[6] Salman Habib, Adrian Pope, Hal Finkel, Nicholas Frontiere, Katrin Heitmann, David Daniel, Patricia Fasel, Vitali Morozov, George Zagaris, Tom Peterka, et al. HACC: Simulating sky surveys on state-of-the-art supercomputing architectures. *New Astronomy*, 42:49–65, 2016.

[7] Benjamin Horowitz and Adrian E. Bayer. jFoF: GPU cluster finding with gradient propagation. *arXiv preprint arXiv:2510.26851*, 2025.

[8] Joseph Hutter, Justin Szaday, Jaemin Choi, Simeng Liu, Laxmikant Kale, Spencer Wallace, and Thomas Quinn. Paratreet: A fast, general framework for spatial tree traversal. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 762–772. IEEE, 2022.

[9] Laxmikant V Kale and Sanjeev Krishnan. Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, 1993.

[10] L.V. Kalé and Amitabh Sinha. Projections: A preliminary performance tool for charm. In *Parallel Systems Fair, International Parallel Processing Symposium*, pages 108–114, Newport Beach, CA, April 1993.

[11] Raghavendra Kanakagiri, Ritvik Rao, Laxmikant Kale, and Karthik Senthil. UnionFindLib, October 2018.

[12] Alexander Knebe, Steffen R. Knollmann, Stuart I. Muldrew, Frazer R. Pearce, Miguel Angel Aragon-Calvo, Yago Ascasibar, Peter S. Behroozi, et al. Haloes gone MAD: The halo-finder comparison project. *Monthly Notices of the Royal Astronomical Society*, 415(3):2293–2318, 2011.

[13] Harshitha Menon, Lukasz Wesolowski, Gengbin Zheng, Pritish Jetley, Laxmikant Kale, Thomas Quinn, and Fabio Governato. Adaptive techniques for clustered n-body cosmological simulations. *Computational Astrophysics and Cosmology*, 2(1):1–16, 2015.

[14] Susana Planelles and Vicent Quilis. ASOHF: a new adaptive spherical overdensity halo finder. *Astronomy & Astrophysics*, 519:A94, 2010.

[15] Andrey Prokopenko, Daniel Arndt, Damien Lebrun-Grandié, Bruno Turcksin, Nicholas Frontiere, J. D. Emberson, and Michael Buehlmann. Advances in ArborX to support exascale applications. *The International Journal of High Performance Computing Applications*, 39(1):167–176, 2025.

[16] Andrey Prokopenko, Damien Lebrun-Grandié, and Daniel Arndt. Fast tree-based algorithms for DBSCAN for low-dimensional data on GPUs. In *Proceedings of the 52nd International Conference on Parallel Processing (ICPP)*, pages 503–512, 2023.

[17] Ritvik Rao, Kavitha Chandrasekar, and Laxmikant Kale. An adaptive asynchronous approach for the single-source shortest paths problem. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 697–702. IEEE, 2024.

[18] Fabrice Roy, Vincent R. Bouillot, and Yann Rasera. pFoF: a highly scalable halo-finder for large cosmological data sets. *Astronomy & Astrophysics*, 564:A13, 2014.

[19] Stephen Skory, Matthew J. Turk, Michael L. Norman, and Alison L. Coil. Parallel HOP: A scalable halo finder for massive cosmological data sets. *The Astrophysical Journal Supplement Series*, 191(1):43–57, 2010.

[20] X Carol Song, Preston Smith, Rajesh Kalyanam, Xiao Zhu, Eric Adams, Kevin Colby, Patrick Finnegan, Erik Gough, Elizabett Hillery, Rick Irvine, et al. Anvil-system architecture and experiences from deployment and early user operations. *PEARC'22: Practice and Experience in Advanced Research Computing*, 2022.

[21] Volker Springel, Rüdiger Pakmor, Oliver Zier, and Martin Reinecke. Simulating cosmic structure formation with the GADGET-4 code. *Monthly Notices of the Royal Astronomical Society*, 506(2):2871–2949, 2021.

[22] Volker Springel, Simon D. M. White, Giuseppe Tormen, and Guinevere Kauffmann. Populating a cluster of galaxies – I. results at $z = 0$. *Monthly Notices of the Royal Astronomical Society*, 328(3):726–750, 2001.

[23] Jonathan Woodring, Katrin Heitmann, James Ahrens, Patricia Fasel, Chung-Hsing Hsu, Salman Habib, and Adrian Pope. Analyzing and visualizing cosmological simulations with ParaView. *The Astrophysical Journal Supplement Series*, 195(1):11, 2011.

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

C_1 A parallel FoF cluster finder composing the existing ParaTreeT and Union-Find libraries, and an analysis showing it is limited by redundant edge submission that does not shrink with processor count.

C_2 ParaTreeT2 , a restructured framework admitting local-only traversals, node annotations recomputed after tree build, and visitors with mutable process-level state; and a Union-Find library with owner-encoded identifiers and lazy vertex storage.

C_3 An optimization sequence — dual-tree walks, uniformity certificates, a process-level work pool, ownership and symmetry pruning, split component counting — scaling 80M- and 2B-particle clustered datasets to several thousand cores.

B. Computational Artifacts

A_1 https://github.com/UIUC-PPL/paratreet2

A_1 is the FoF application and the ParaTreeT2 framework; it builds against four further public repositories (see Software) and supports C_1 (Figures 2–3), C_2 (Figure 5) and C_3 (Figures 6–12). Archival DOIs will be minted for the camera-ready version.

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

A_1 contains every implementation the paper measures. The `add-friends-of-friends` branch of ParaTreeT is the C_1 baseline; ParaTreeT2 carries C_2 and the C_3 sequence, each step recorded with its commit in `design/optimization-sequence.md`.

Expected Results

Component counts must be identical across node counts and between the two union-find modes: 23,707,197 on 80M and 424,897,832 on 2B. Phase A should scale near-linearly while phase B stays flat, so phase B comes to dominate. Time excluding I/O should be about 0.5 s for 80M on 240 cores and 12 s for 2B on 1,792 cores.

Expected Reproduction Time (in Minutes)

Setup about 60 minutes, of which Charm++ is 30. The correctness matrix takes under 5 minutes on a laptop; an 80M sweep (1–16 nodes) about 20 node-minutes, a 2B sweep about 400. Analysis under 5 minutes.

Artifact Setup (incl. Inputs)

Hardware: A Linux cluster. Reported runs used Frontier (AMD EPYC 7A53, 64 cores/node, Slingshot-11) at 8 processes per node, 14 PEs per process. Nothing is Frontier-specific, though LCI/CXI needs a Slingshot fabric; 2B needs about 8 nodes of memory.

Software: Five public repositories, in build order (default branch unless noted):

1) Charm++, branch `reconverse-specific-build`: github.com/charmplusplus/charm
2) htram: github.com/UIUC-PPL/htram
3) Union-Find, branch `fof_with_aggregation`: github.com/UIUC-PPL/unionfind
4) ParaTreeT2 (A_1): github.com/UIUC-PPL/paratreet2
5) ParaTreeT (C_1 baseline only), branch `add-friends-of-friends`: github.com/paratreet/paratreet

Also a C++17 compiler, CMake 3.20+, and Python 3 with matplotlib; LCI is fetched by the Charm++ build.

Datasets / Inputs: The reported datasets are the N-BodyShop tipsy outputs `lambb.00500` (80,621,568 particles) and `cosmo25cmb.768g2_dm.001024` (1.98B), from the ChaNGa collaboration on request. Not being redistributable, A_1 ships deterministic generators for the same regimes: `inputgen/plummer` (clustered) and `inputgen/uniform`, plus checked-in inputs of 100–10k particles.

Installation and Deployment: Build Charm++ first; the `reconverse-` triplets are always SMP and enable Reconverse and LCI in one step:

```
git clone -b reconverse-specific-build \
  https://github.com/charmplusplus/charm
cd charm && ./build charm++ reconverse-linux-x86_64 -j8
export CHARM_HOME=$PWD/reconverse-linux-x86_64
```

Extra CMake settings go through `--with-cmake-args`; a bare `-DFOO=BAR` is forwarded to the compiler instead. Check out htram, unionfind and paratreet2 as siblings, then:

```
cd htram && make && cd ../unionfind/prefixLib
make && cd .. && make AGGREGATION= PROFILE=
cd ../paratreet2 && git submodule update --init
cd utility/structures && ./configure && make
cd ../../src && make && cd ../fof && make
cd ../examples/fof3 && make && cd ../../inputgen && make
```

Union-Find is built aggregation-off, matching the reported configuration: htram is linked but dormant. `make PROJECTIONS=1` gives the traced binary for the time-profile figures.

Artifact Execution

Four tasks, $T_1 \to T_2 \to T_3 \to T_4$: T_1 stages or generates a dataset; T_2 runs `FoF3`, printing per-stage timings and the component histogram on `FOF3STAT` lines; T_3 collects those across node counts; T_4 renders the figures.

Run `make test` in `examples/fof3` first: a 12-run matrix over particle counts and process/PE layouts, each printing `FOF3 TEST PASSED`. Scaling runs sweep node count in powers of two at the layout above, with $b = 0.2$. Above 20M particles the harness selects statistics mode automatically, the $O(n^2)$ cross-check being infeasible; determinism across node counts is then the correctness criterion. Times are medians of four repetitions.

Artifact Analysis (incl. Outputs)

Correctness is the component histogram on the `FOF3STAT` lines, which must match bit for bit across node counts and union-find modes, and below 20M particles an exact serial reference. Performance comes from the per-stage times, which `plot_fof_scaling.py` turns into the scaling figure.