Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200050

A Hybrid MPI+Threads Approach to Particle Group Finding Using Union-Find

James S. WILLIS [a,1], Matthieu SCHALLER [a,b], Pedro GONNET [c] and John C. HELLY [a]

[a] *Institute for Computational Cosmology (ICC), Department of Physics, Durham University, Durham DH1 3LE, UK*
[b] *Leiden Observatory, PO Box 9513, NL-2300 RA Leiden, The Netherlands*
[c] *Google AI Switzerland GmbH, 8002 Zürich, Switzerland*

Abstract. The Friends-of-Friends (FoF) algorithm is a standard technique used in cosmological *N*-body simulations to identify structures. Its goal is to find clusters of particles (called groups) that are separated by at most a cut-off radius. *N*-body simulations typically use most of the memory present on a node, leaving very little free for a FoF algorithm to run on-the-fly. We propose a new method that utilises the common Union-Find data structure and a hybrid MPI+threads approach. The algorithm can also be expressed elegantly in a task-based formalism if such a framework is used in the rest of the application. We have implemented our algorithm in the open-source cosmological code, SWIFT. Our implementation displays excellent strong- and weak-scaling behaviour on realistic problems and compares favourably (speed-up of 18x) over other methods commonly used in the *N*-body community.

Keywords. Friends-of-Friends; Union-Find; MPI; Threads; Efficiency

1. Introduction

Over the last four decades cosmological simulations have been the main tool used by physicists to confront their theoretical predictions to observations. By creating more-and-more realistic universes they have been able to revolutionise our understanding of the cosmos and establish the current cosmological model. These simulations typically involve the evolution of large numbers of particles or resolution elements under the laws of gravity and hydrodynamics. Given the large volumes simulated and the ever-growing need for more details, these simulations are often at the forefront of research in HPC and require ever-increasing computing capabilities. For instance, the current record holder, the *Euclid flagship simulation* [1], evolved 8×10^{12} particles from the Big Bang to the present day and generated peta-bytes of data.

Putting aside the question of running such simulations, analysing these large volumes of data poses huge computational challenges as even the most basic operations require sizeable facilities to simply host the data in memory. One of the most-widely used post-processing tool for such simulations is the Friends-of-Friends (FoF) method [2],

[1] Corresponding Author; E-mail: james_willis@hotmail.co.uk.

which is designed to identify *groups* of particles that are within a certain *linking-length*, l_x, of each other. If the linking-length is chosen to be small enough then the method will identify groups that correspond to structures of particles that have formed due to gravity and hence capture information about the evolution of the Universe. More specifically two particles are in the same *group* if they are at a distance smaller than l_x of each other. Particles can be linked to multiple other particles and all particles linked in this way are in the same group [2]. The size of a group is later defined as the number of particles that are linked to each other by this criterion. Particles without any neighbours within l_x form a group of size one. Since producing catalogs of particle groups in post-processing can be prohibitively expensive, in terms of i/o at least, it is common practice to apply the FoF method on-the-fly at fixed time intervals over the course of the N-body simulation. This also allows the production of FoF outputs at a higher frequency. Over the years many dedicated stand-alone FoF packages have been implemented, recent examples used in production runs include [3–5]. Nevertheless, the challenge of efficiently distributing the method over large numbers of nodes on-the-fly, i.e. *whilst reusing the pre-existing data structures put in place for the N-body solver*, still remains.

In this paper we present a FoF implementation that exploits the hybrid shared/distributed parallelism built into the SWIFT cosmological code [3] [6, 7] to achieve excellent efficiency whilst also being able to run at regular intervals over the course of large cosmological simulations.

2. FoF using the Union-Find algorithm

FoF is related to the more general problem of Euclidean minimum spanning trees (here in 3 dimensions), which is a very well-studied problem (e.g. [8, 9]) with algorithms that are near-linear in the worst case, but differs crucially in that:

- The maximum Euclidean distance considered is limited, thus limiting the range of neighbours for each node, and
- We are not interested in the exact structure of the resulting minimum spanning tree (or set of trees), but only in which nodes belong to the same trees.

The problem is therefore equivalent to the *disjoint-set union (or* union-find) problem [10, 11], and the FoF method we have implemented is based on the approaches used for its solution in shared/distributed-memory parallel settings [12–15].

A disjoint-set data structure is the basis for the algorithm, which maintains a collection of dynamic non-overlapping sets consisting of N distinct elements. Each set is identified by a representative element (the *root*). It is widely used in the calculation of minimum spanning trees in graphs and the computation of connected components.

The Union-Find algorithm is designed around two operations: *Union*, which merges a pair of sets and *Find*, which identifies the set a given element resides in. The data structure is typically implemented using a *forest*, where each *tree* represents a connected set and the root of each tree identifies the set. Initially each set contains one element which

[2] More mathematically, the problem can be expressed as determining the connected components of a graph $\mathcal{G}$, based on a set of points P, where $\mathcal{G}$ is defined as $\mathcal{G} = (P, E)$ with the set $E = \{\{u, v\} : dist(u, v) \leqslant l_x\}$ and $u, v \in P$.

[3] See also www.swiftsim.com.

is the sole member of its set and its set's representative. Two sets containing elements that are within the linking-length distance, l_x, are merged using the *Union* operation[4].

There are several standard ways to optimise the Union-Find algorithm. The *Union* operation for example, can be implemented using *Union-by-size* which links smaller sets to larger ones and *Union-by-rank* that links sets with shorter trees to sets which have taller trees. However, we will use *Union-by-root* and make the larger root always point to the smaller root, where the initial root of each set is assigned by its offset in the array. This allows us to bypass the issues with parallelism (see below) reported by [15].

Another common optimisation technique is *path compression*. Each tree vertex traversed in a Find operation is set to point to the root of the set. This means that subsequent Find operations are quicker as most vertices will point directly to the root; reducing the rank of each particle and hence lowering the (theoretical) loss of performance using a Union-by-root approach over a Union-by-rank.

The Union-Find algorithm has been extensively parallelised in the literature for both shared and distributed memory machines: [12–14]. The novelty of our paper is the introduction of a hybrid shared/distributed memory algorithm that uses a task-based framework, which can be run on-the-fly within our N-body code that imposes a spatial decomposition.

3. Implementation in the SWIFT code

3.1. Serial implementation

In practice the Union-Find data structure is implemented using an array of length, N, where N is the total number of particles and each element represents a particle. The array is initialised so that each particle exists in its own group, i.e each element is set to the offset of the particle in the array. A neighbour search is then performed over the particles using the linking-length, l_x, as the search criterion. The Find operation is used on all particles that are neighbours to return their roots. Two groups are then merged using the Union operation, where the smaller of the two roots is used as the group label henceforth. For example:

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    if(i == j) continue; // Avoid self
    r = particle_dist(parts[i], parts[j]);
    if (r < l_x)
      // Find operation
      int root_i = fof_find(i, group_index);
      int root_j = fof_find(j, group_index);
      // Union operation
      if(root_i < root_j) group_index[root_j] = root_i;
      else group_index[root_i] = root_j;
```

Code 1: Union-Find with a simple iteration over neighbours.

[4]In the context of the FoF method we use the following terminology: a *set* is referred to as a *group* and an *element* is an individual *particle*.

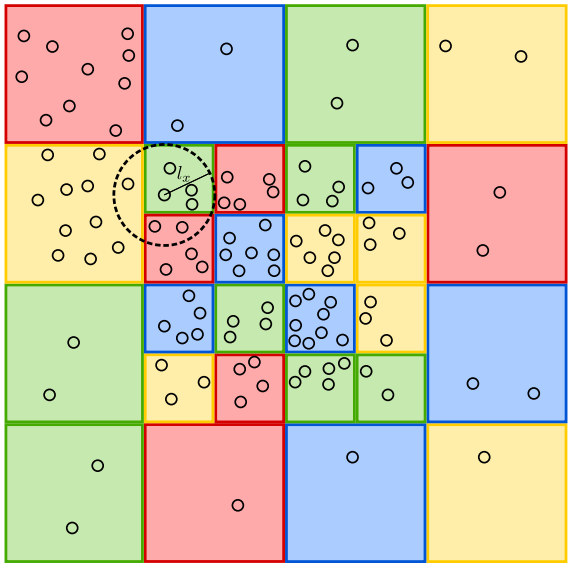

Figure 1. FoF Union-Find using task-based parallelism. Each coloured cell represents a single task. Self tasks are created for each cell and pair tasks are created between cells that lie within the cut-off radius, l_x, of each other. A self task performs a FoF search on particles in a single cell whereas a pair task carries out a FoF search between particles in neighbouring cells. Tasks are placed into a queue. A group of threads pick and execute tasks from the queue concurrently until there are none remaining.

where `parts` is the particle array and `group_index` is the array that represents the Union-Find data structure.

As in the case of minimum spanning tree problems, we make use of the octree (quadtree in 2D) present in SWIFT to significantly reduce the cost of the neighbour search, by only recursing on pairs of cells that are within the requested cut-off radius, l_x, of each other. We note, however, that the best performance is achieved when the size of the tree nodes matches the linking-length (see the technique of [4] or [5]), but that tailoring the octree node sizes would hinder the performance of the rest of the SWIFT code and is hence not an option. Once the tree has been setup, the problem becomes almost embarrassingly parallel and we split the workload evenly either between: (a) a group of threads, or equivalently (b) a set of tasks (see Fig. 1). We implement the latter in SWIFT using a variant of the QUICKSCHED tasking library [16].

3.2. Shared memory parallelism

In order to parallelise the algorithm a subtle issue needs to be taken care of, i.e. each thread must have a consistent view of the tree data. For example, consider two roots: r_i and r_j, we need to ensure that one thread does not find $r_i < r_j$ whilst another concludes $r_i > r_j$. One possibility would be to use locks when writing to the Union-Find data structure (`group_index`), but this would hinder scalability as more and more threads try to access the list. We instead solve this problem by checking that the value of r_i has not changed between being read and being found to be lower than r_j. If r_i has changed between these events the process is repeated until the value of r_i remains constant.

We implement the Union operation in a thread-safe manner by using the *Compare And Swap* (CAS) atomic, proposed by [12]:

```
1  int atomic_update_root(volatile size_t *address, size_t y) {
2
3    size_t *size_t_ptr = (size_t *)address;
4
5    size_t old_val = *address;
6    size_t test_val = old_val;
7    size_t new_val = y;
8
9    old_val = atomic_cas(size_t_ptr, test_val, new_val);
10
11   return (test_val == old_val);
12 }
13
14 void fof_union(size_t i, size_t j, size_t *group_index) {
15   int result = 0;
16   // Loop until the root can be set to a new value.
17   do {
18     size_t root_i = fof_find(i, group_index);
19     size_t root_j = fof_find(j, group_index);
20
21     if(root_j < root_i)
22       result = atomic_update_root(&group_index[root_i], root_j);
23     else
24       result = atomic_update_root(&group_index[root_j], root_i);
25
26   } while (!result);
27 }
```

Code 2: Using a CAS atomic operation to perform the *Union* of two groups in a thread-safe manner.

This ensures any update to `group_index` is lock-free, and hence avoids any performance penalties introduced by locks. A weakness of this method, however, is that the CAS operation can only update a single variable at a time[5]. Therefore, if a (formally more efficient) Union-by-size or Union-by-rank version of the algorithm were to be used, it would require a lock instead of an atomic to avoid data races. One solution to this problem is to adopt the approach by [15], where the Union is instead randomised. It avoids having to update two variables per Union as the size or rank of a group is not stored in addition to the root.

We also tested a version of our parallel algorithm using the randomisation technique proposed by [15]. This implementation showed similar times to solution compared to our basic approach. This is due to the fact that we only use the root of a group to perform a Union operation, and hence do not suffer from the weakness of the Anderson & Woll implementation [12]. Our specific workloads, where the *rank* of the elements added in the *Union* operations are typically small, are another reason why we did not see a noticeable increase in performance. For these reasons we chose to use our simpler solution and stick to the *Union-by-root* method.

[5]There has been an attempt by [17] to implement a multi-variable CAS operation, but their results show that in practice the performance of this approach is not superior to traditional locking techniques.

3.3. Distributed memory parallelism

For larger simulations, particles are distributed across multiple nodes (see Fig. 2). To address the problem of groups spanning multiple nodes, we follow the strategy outlined by [14] and improve upon it to handle the case where the number of groups is much larger than 10^2.

We first perform a multi-threaded local Union-Find on each node, as described in Section 3.2, followed by assigning unique group IDs across all nodes. This is done by computing an offset based upon the MPI rank of the node. Each rank, p, computes a sum of the total number of particles contained on every MPI rank lower than itself, $\sum_{i=0}^{i<p} N_i$, where N_i is the total number of particles present on rank i. The sum is then used to offset all group IDs on the local node. In practice this is done using `MPI_Scan`:

```
1    long long num_parts_cumulative;
2    long long num_parts_local = num_parts;
3    MPI_Scan(&num_parts_local, &num_parts_cumulative, 1, MPI_LONG_LONG,
4             MPI_SUM, MPI_COMM_WORLD);
5    size_t node_offset = num_parts_cumulative - num_parts_local;
```

Code 3: Computing the node offset with `MPI_Scan`.

Next, we identify links between groups that span at least two node domains, only communicating information for groups that are within the linking-length, l_x, of the domain boundaries. This greatly reduces the amount of data replication. The final step performs a global gather communication (`MPI_Allgatherv`) on the list of group links so that every node has access to the global list of group links (`global_group_links`). Each node then applies the Union-Find to `global_group_links`, only updating the roots of groups which are local to them. This ensures that all spanning groups are merged and each node agrees upon group ownership.

In order to apply the Union-Find on the global list we map each group ID to a number between 0 and the total number of group IDs that span node domains. The same group ID may appear multiple times in the list, therefore we need to search for the first occurrence of it and use the index as input to the Find operation. This ensures that the result of the Find operation is correct, as the group ID could have previously been updated from a group merger earlier in the list. See Fig. 3.

Naively one may think that each rank need only run the Union-Find on the group links that it shares with its neighbouring ranks. However, Fig. 2 shows a particle distribution that forms a group on rank 0 that is indirectly linked to the group on rank 2 via the groups on ranks 3 and 4. This group linkage will be overlooked if each rank only searches for links with its direct neighbours.

If we use the same Union strategy as the local FoF, the distribution of roots of spanning groups will be skewed towards the lower MPI ranks. This can lead to a load imbalance between nodes when assigning new local roots during Step 4. To address this problem we use Union-by-size when merging groups across MPI domains. This creates an even work load between ranks as Union-by-size will assign roots more arbitrarily and will only be based upon the domain decomposition.

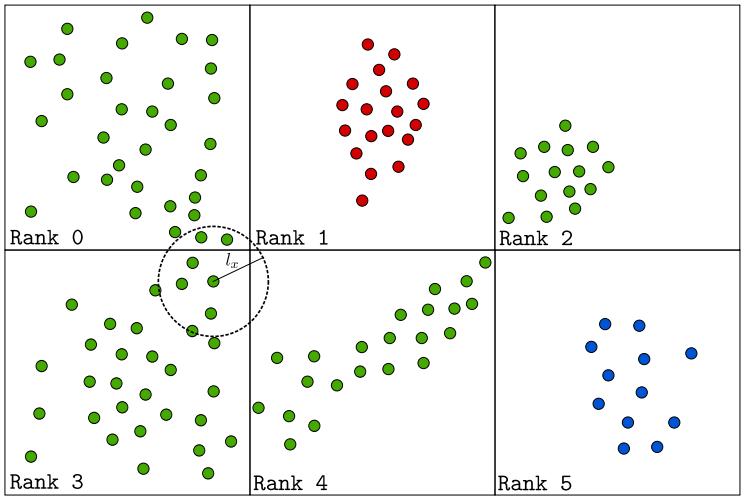

Figure 2. Distributed Union-Find over MPI. Particles are distributed across each MPI rank and the following steps are performed: 1) a local FoF is performed on each MPI rank; 2) relabel group IDs so that they are globally unique; 3) identify links between groups that span two MPI domains; 4) merge distributed groups and agree on ownership. The figure also illustrates an edge case that can occur. The group on rank 0 is indirectly linked to the group on rank 2 via the groups on ranks 3 and 4. If we were to only merge groups between MPI ranks that are direct neighbours in step 4), we would fail to take into account this subtlety and miss the indirect group links.

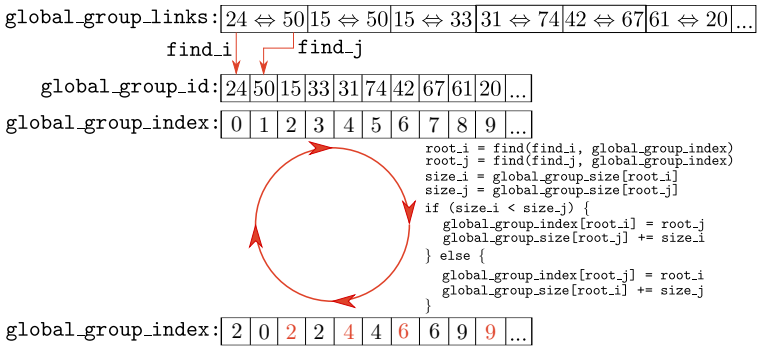

Figure 3. Step 4 in the distributed Union-Find method. Distributed groups are merged and each MPI rank agrees on group ownership. The `global_group_links` array stores all group links that span an MPI domain, each unique ID in the list is unpacked into `global_group_id` and mapped to a number between 0 and the total number of unique group IDs in the list (`global_group_index`). Find is applied to each pair of links in the list, where the group offset (`find_i` & `find_j`) into `global_group_id` is used as input. This ensures that Find returns the correct group ID in the case where it has been updated in an earlier group merger (Union). The pair of groups are then merged using the Union operation and `global_group_index` is updated.

3.4. Implementation details

3.4.1. Hash table

Performing the last step of the distributed FoF algorithm can become quite expensive, as the length of `global_group_links` scales with the node count. This is because searching for the index of a group ID into the list roughly takes $\mathcal{O}(N^2)$ operations. A

hash table on the other hand has constant look-up times, $\mathcal{O}(1)$. Therefore we construct a hash table of group IDs in the list and store their index into `global_group_links`.

We also make use of a hash table when calculating the group sizes in the local FoF. To find the group sizes in serial we loop through the `group_index` array and increment `group_size` indexed by the root of the group that each particle is in:

```
1    for (int i = 0; i < N; i++)
2      group_size[fof_find(i, group_index)]++;
```

Code 4: Group size calculation in serial.

In parallel we divide N by the number of threads and have each thread work on a section of `group_index`. We avoid race conditions between threads by protecting access to `group_size`. To do this we use a hash table to store the group sizes and root of each group. Once we have looped over `group_index`, we pull out each element of the hash table and write the intermediate group size to the global `group_size` array using an atomic addition (for instance GNU C's `__sync_fetch_and_add`).

3.4.2. Early elimination of small groups

The majority of groups in cosmological simulations are of lone particles. We were able to take advantage of this fact to lower the memory footprint significantly when calculating group sizes. When constructing the hash table only groups of size $\geqslant 2$ were stored. We achieved this by initialising each element of the `group_size` array to 1, which allowed us to exclude root particles in the hash table as their contribution to the group size was already accounted for.

3.4.3. Path compression optimisation

The Find operation is a tree traversal that retrieves the root of a group for a given particle. Hence, the execution time is dominated by the depth of the tree at each particle. To amortise the cost of this operation we have implemented path compression. But instead of compressing trees of all depths, we found it was quicker to only compress trees with a depth of at least 2.

4. Results

To test the performance of our FoF implementation we ran a number of different benchmarks. We measured the strong- and weak-scaling performance as well as the speed-up over another FoF application. All results were obtained on the COSMA-7 DiRAC 2.5x "Memory Intensive" System, located at the University of Durham [6]. The results are based

[6] The system consists of 452 nodes of 2 Intel Xeon Gold 5120 CPUs running at 2.2GHz (14 physical cores with AVX512 capability) with 512 GBytes of RAM. The nodes are connected using Mellanox EDR Infiniband in a 2:1 blocking configuration. The strong scaling results were obtained by running on the MAD02 machine at Durham with Turbo Boost disabled for the purposes of obtaining accurate measurements. It is a quad socket system each with an Intel Xeon Platinum 8180 CPU running at 2.5GHz (28 physical cores with AVX512 capability) with 1.5 TBytes of RAM. See https://dirac.ac.uk/resources/#MemoryIntensive for more details on each system.

on version 0.8.2 of SWIFT (`git` revision `f05bd301`), which implements the algorithm described in Section 3.

4.1. Measurement methodology

To get a realistic workload, all benchmarks were carried out using particle data from the flagship EAGLE simulations [18] at late times (redshift $z = 0.1$). The input data contains 4.25×10^8 particles split into $\sim 2 \times 10^5$ groups of length > 20. The workload is representative of an actual production run of SWIFT and nicely fits within a single node's memory. To create a weak-scaling test, we replicate the simulation volume periodically N times along each axis, creating problem sizes that are N^3 larger than the original volume.

We used the Intel compiler and MPI library v.18.0.2[7] as well as the GNU compiler v.9.1.0[8]. To obtain precise execution times we used the `RDTSC` cycle counter and converted the cycle counts to seconds using the clock-speed of the CPU. Each data point is the average time of 3 independent runs and the standard deviation is used to measure the uncertainty. For the weak-scaling tests, we use 4 MPI ranks per node (2 per NUMA region) and use the MPI version of the code even for the single-node data point in order to have the same MPI-related overheads throughout the test. The strong-scaling test does not use MPI and hence probes the efficiency of the shared memory algorithm.

4.2. Strong- and weak-scaling results

The strong scaling results are shown in the left hand panel of Fig. 4. We stress that these results were obtained starting from one core and keeping the problem size constant. Turbo Boost was also disabled on the node for the purposes of obtaining accurate measurements. We display very good strong scaling and maintain a high parallel efficiency, achieving 77% on 112 cores. Only dropping in efficiency when hyper-threads are used, but this can be explained by resource contention between competing threads. This is a result of our shared memory strategy: effective load balancing between threads using an octree and task-based parallelism; and a lock-less implementation of the parallel Union-Find algorithm.

The right-hand panel displays the weak-scaling performance, where we achieve good scaling up to 10,206 cores despite the overhead costs of MPI communication. The last data point corresponds to a simulation with 3×10^{11} particles. The jump from ½ a node to 4 nodes is a result of the MPI communication being performed over the network, as opposed to on a single node. Additionally, since that data point only uses half the available cores on the node, a better memory throughput is achieved and the cores are running at a slightly higher clock speed (2.9 vs. 2.6 GHz) thanks to Turbo Boost. We hence only consider the results starting from the next data point (4 nodes) where all the cores are busy on each node. From that point onwards, the gradual increase in runtime is a result of the network, as it has a greater effect at higher node counts and becomes the limiting factor. The loss in performance running on 10,206 with ICC is 34%. Starting from the second data point (where the nodes are now using all cores and do not suffer from the caveats mentioned above), we obtain a significant improvement only losing 15% going from 4 ($= 2^3$ the original problem size) to 364½ nodes ($= 9^3$ the original problem size).

[7] with the flags `-O3 -xCORE-AVX512`.

[8] with the flags `-O3 -ffast-math -march=skylake-avx512 -mavx512dq`.

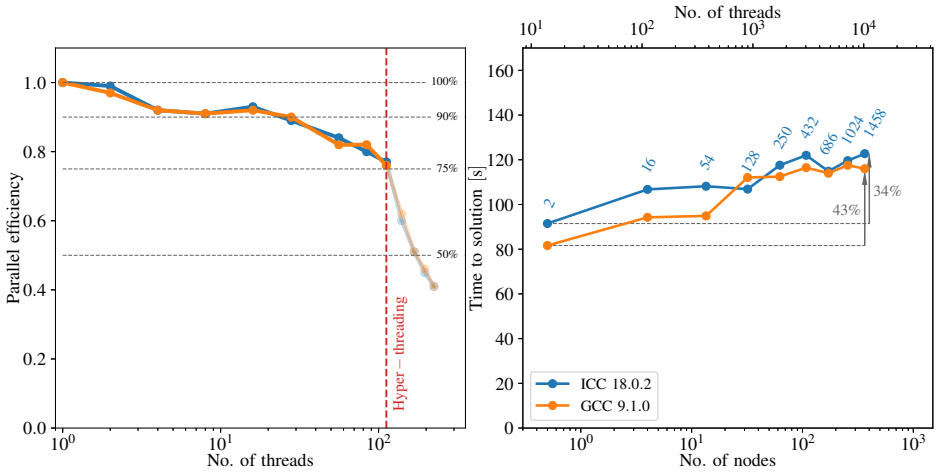

Figure 4. SWIFT FoF scaling results on a representative cosmological problem. The particle data is taken from the EAGLE simulations [18] from a snapshot at redshift $z = 0.1$, i.e. near the end of the calculation when the distribution of particles is far from uniform. (*Left*) Strong scaling results. The particle load was kept constant at 4.25×10^8 whilst the number of cores was increased. As the benchmark was performed on one node, the non-MPI version of the algorithm was used. We maintain very good strong scaling performance and obtain 77% parallel efficiency on 112 cores. The efficiency drops when running with hyper threads due to resource contention between threads. (*Right*) Weak-scaling results. The number of particles per core is kept constant at 3×10^7, as we increase the core count. We use 4 MPI ranks per node (2 ranks *per socket*). For convenience, the total number of MPI ranks used is indicated by the labels above the data points. The vertical arrow displays the percentage loss in performance running on 10,206 cores, which was 43% for GCC and 34% for ICC. We achieve good weak scaling from ½ a node to 364½ nodes (a factor of 729 increase in the number of particles and number of cores) despite the overhead costs of MPI communication. For both panels, the standard deviation of each measurement is smaller than the symbol size.

There is also a noticeable difference in runtime between the Intel and GNU compilers for the first three data points, with GNU showing a speed-up of ∼13% over Intel. A similar discrepancy is also seen in the strong scaling results.

This is a combination of a highly efficient parallel Union-Find algorithm within a single node and a scalable distributed memory strategy between nodes. The domain decomposition implemented in SWIFT also plays a role so as to keep the work load balanced between MPI ranks (see [6] and [19]).

4.3. Comparison to other software

As another performance test we compared our implementation against VELOCIRAPTOR [3], a FoF application commonly used in the literature. We used the same setup as in the strong-scaling test and ran on the MAD02 machine using the Intel compiler and MPI library v.18.0.2[9]. We ran the non-MPI version of our code and the MPI version of VELOCIRAPTOR with 1 rank per core. Our FoF took 13.2s to run to completion and VELOCIRAPTOR took 242s, leading to a net speed-up of 18.3x[10]. Both codes yield the same answer. Given the large difference in run time on one node and the good weak-

[9] with the flags `-O3 -xCORE-AVX512`

[10] Note that we used the MPI version of VELOCIRAPTOR as it was significantly faster than its shared-memory (OpenMP) version which took 1882s running with 112 threads on the same setup.

scaling displayed by our implementation, we decided not to compare our performance with VELOCIRAPTOR at scale.

5. Conclusions

We presented an efficient and scalable new implementation of the FoF method that is commonly used to identify structure in cosmological simulations. The Union-Find data structure was used to create a *forest* of particles, where each *tree* contains a set of particles that share the same group. A hybrid approach was adopted using threads and MPI, which allows it to optimally utilise both shared and distributed memory machines. We made use of atomics to update the list of particle groups which ensures our implementation remains lock-free. The neighbour search over particles was sped up using the octree present in the SWIFT code. A hash table was used in both the group size calculation and group merging across MPI domains to lower the memory footprint and improve the time to solution.

When implemented in the SWIFT code our FoF algorithm achieves good weak-scaling from 14 to 10,206 cores and displays good strong-scaling performance, maintaining 77% parallel efficiency running on 112 cores. We also compare favourably with the commonly used FoF application VELOCIRAPTOR, obtaining a speed-up of 18x over it. Together with the weak-scaling performance displayed up to 10^4 cores this speed-up should allow for an efficient run time when used on-the-fly in production simulations using $\gtrsim 10^5$ cores.

Acknowledgements

We thank the anonymous referees for their comments that greatly helped improve the paper. This work would not have been possible without Lydia Heck, Peter Draper, Richard Regan and Alastair Basden's help and expertise running on the cosma systems; as well as the SWIFT team for their help and input on this project. This work is supported by INTEL through establishment of the ICC as an INTEL parallel computing centre (IPCC). MS is additionally supported by the NWO VENI grant 639.041.749. This work used the DiRAC@Durham facility managed by the Institute for Computational Cosmology on behalf of the STFC DiRAC HPC Facility (www.dirac.ac.uk). The equipment was funded by BEIS capital funding via STFC capital grants ST/K00042X/1, ST/P002293/1, ST/R002371/1 and ST/S002502/1, Durham University and STFC operations grant ST/R000832/1. DiRAC is part of the National e-Infrastructure.

References

[1] D. Potter, J. Stadel, and R. Teyssier, "Pkdgrav3: beyond trillion particle cosmological simulations for the next era of galaxy surveys," *Computational Astrophysics and Cosmology*, vol. 4, p. 2, May 2017.

[2] M. Davis, G. Efstathiou, C. S. Frenk, and S. White, "The evolution of large-scale structure in a universe dominated by cold dark matter," *The Astrophysical Journal*, vol. 292, 06 1985.

[3] P. J. Elahi *et al.*, "Hunting for galaxies and halos in simulations with velociraptor," *Publications of the Astronomical Society of Australia*, vol. 36, p. e021, 2019.

[4] P. E. Creasey, "Tree-less 3d Friends-of-Friends using Spatial Hashing," *Astron. Comput.*, vol. 25, pp. 159–167, 2018.

[5] Y. Kwon *et al.*, "Scalable clustering algorithm for n-body simulations in a shared-nothing cluster," in *Scientific and Statistical Database Management*, (Berlin, Heidelberg), pp. 132–150, Springer Berlin Heidelberg, 2010.

[6] M. Schaller *et al.*, "Swift: Using task-based parallelism, fully asynchronous communication, and graph partition-based domain decomposition for strong scaling on more than 100,000 cores," in *Proceedings of the PASC Conference*, PASC 16, (New York, USA), pp. 2:1–2:10, ACM, 2016.

[7] P. Gonnet, "Efficient and scalable algorithms for smoothed particle hydrodynamics on hybrid shared/distributed-memory architectures," *SIAM Journal on Scientific Computing*, vol. 37, no. 1, pp. C95–C121, 2015.

[8] S. Arya and D. M. Mount, "A fast and simple algorithm for computing approximate euclidean minimum spanning trees," in *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pp. 1220–1233, 2016.

[9] M. Connor and P. Kumar, "Fast construction of k-nearest neighbor graphs for point clouds," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, pp. 599–608, July 2010.

[10] 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.

[11] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. ACM*, vol. 22, pp. 215–225, Apr. 1975.

[12] R. J. Anderson and H. Woll, "Wait-free parallel algorithms for the union-find problem," in *In Proc. 23rd ACM Symposium on Theory of Computing*, pp. 370–380, 1994.

[13] F. Manne and M. M. A. Patwary, "A scalable parallel union-find algorithm for distributed memory computers," in *Parallel Processing and Applied Mathematics* (R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, eds.), (Berlin, Heidelberg), pp. 186–195, Springer Berlin Heidelberg, 2010.

[14] C. Harrison, H. Childs, and K. P. Gaither, "Data-parallel mesh connected components labeling and analysis," in *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '11, (Aire-la-Ville, Switzerland, Switzerland), pp. 131–140, Eurographics Association, 2011.

[15] S. V. Jayanti and R. E. Tarjan, "A randomized concurrent algorithm for disjoint set union," in *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, (New York, NY, USA), pp. 75–82, ACM, 2016.

[16] P. Gonnet, A. B. G. Chalk, and M. Schaller, "QuickSched: Task-based parallelism with dependencies and conflicts," *arXiv e-prints*, p. arXiv:1601.05384, Jan 2016.

[17] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," in *Distributed Computing* (D. Malkhi, ed.), (Berlin, Heidelberg), pp. 265–279, Springer Berlin Heidelberg, 2002.

[18] J. Schaye *et al.*, "The EAGLE project: simulating the evolution and assembly of galaxies and their environments," *Monthly Notices of the Royal Astronomical Society*, vol. 446, pp. 521–554, 11 2014.

[19] J. Borrow *et al.*, "SWIFT: Maintaining weak-scalability with a dynamic range of 10^4 in time-step size to harness extreme adaptivity," in *Proceedings of the 13th SPHERIC International Workshop*, SPHERIC 13, pp. 44 – 51, Jun 2018.