


VELOCiraptor: A MPI+OpenMP data analysis tool for cosmological simulations

Pascal Jahan Elahi 

Pawsey Supercomputing Research Centre, Kensington, WA, Australia
`pascal.elahi@pawsey.org.au`

Abstract. We present VELOCiraptor, a massively parallel analysis tool for cosmological simulations designed to identify cosmic structures like galaxies. The code is written in C++17, and uses MPI and OpenMP API's for parallelisation. MPI decomposition uses space-filling Z-curves to minimize load imbalances and communication. Communication load is further reduced by dynamic repartition of MPI domains. The code also uses the HDF5 library for parallel IO, using MPI communicators to group MPI processes for reading and writing data, improving the parallel IO load balance. For each MPI process, nested OpenMP parallelism is implemented for all computationally intensive portions of the code. We also demonstrate the power of the VELOCiraptor (sub)halo finder, showing how it can identify subhalos, phase-space density peaks residing in larger so-called host halos, configuration space overdensities, deep within the host, where the density contrasts to between them and their host is negligible.

1 Introduction

A common approach to understanding the large-scale universe and the objects that form within it, like galaxies, is to run high-resolution, large-volume cosmological simulations. A critical step in extracting information from sophisticated, multi-billion particle simulations is the identification of structures, like dark matter halos and synthetic galaxies. Identifying (sub)structures is a non-trivial task and has led to the development of equally sophisticated structure finders [see 7, for an overview of (sub)halo/galaxy finding].

Here we present VELOCiraptor, a phase-space (sub)halo finder capable of identifying dark matter halos and galaxies (freely available github.com/pelahi/VELOCiraptor-STF). Documentation is found at velociraptor-stf.readthedocs.io). This C++17 code uses the CMAKE build system, and MPI [1] plus OpenMP [2] parallel APIs along with parallel HDF5 IO [6] to scale up to thousands of cores across many nodes. We focus on novel application of MPI and OpenMP APIs to improve performance of the code and provide only a brief summary of the clustering algorithms used in the code.

2 VELOCiraptor algorithm

The VELOCiraptor code can be called as a stand-alone executable or can be integrated into an existing cosmological code. This has been done for the SWIFTSIM Smooth-Particle Hydrodynamics code 9. The core algorithm can be broken down into a few key steps, listed in Fig. 1. We briefly describe some key algorithms here and for more details, we refer readers to [4].

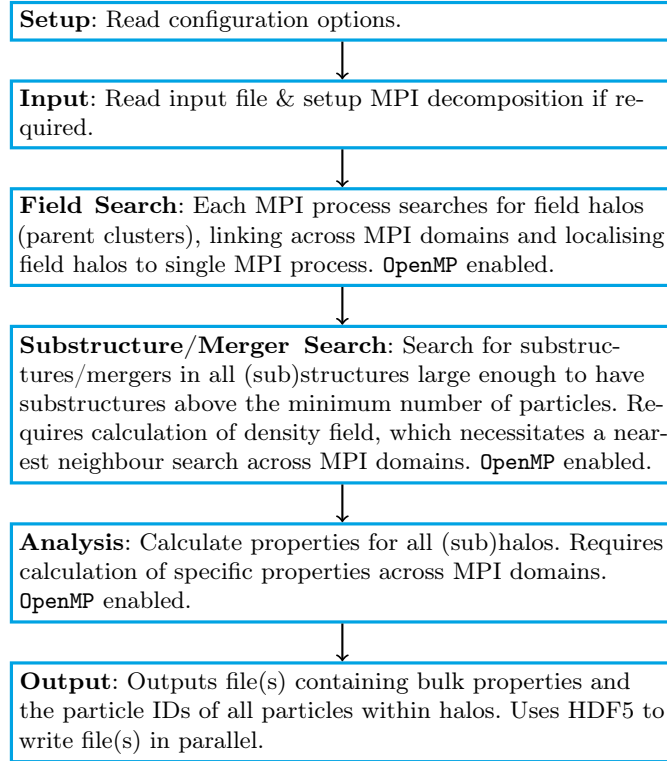


Fig. 1. Activity chart of VELOCIRAPTOR.

The code will first ingest a cosmological simulation data set that consists of particles or a mesh which is mapped to a particle representation. This read is MPI-enabled and will produce a load balanced decomposition (discussed in later sections). It then identifies candidate halos (large physical overdensities) using a 3DFOF algorithm [3D Friends-of-Friends in configuration space, see 3], linking particles together if

$$\frac{(\mathbf{x}_i - \mathbf{x}_j)^2}{\ell_x^2} < 1, \quad (1)$$

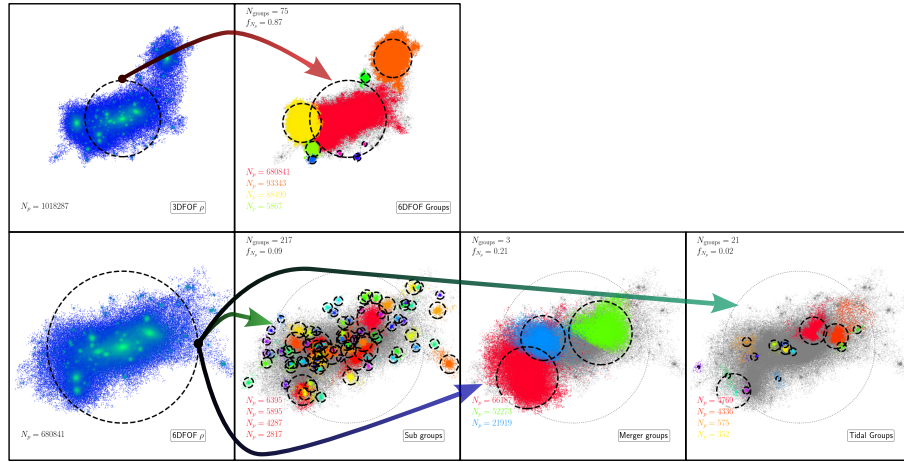


Fig. 2. Halo Decomposition: We show the process of running the routines that decompose an initial FOF halo candidate into phase-space FOF halos (top row), followed by the search for substructure. Here we emphasize three different classes of substructures, standard, major merger (where the substructure is $\gtrsim 10\%$ of the host’s particles), and loosely-bound tidal debris. For each object we show $R_{\Delta\rho_c}$, a size, by a dashed black circle. Particles are colour-coded according to the 3D density (blue to green for increasing density, left column), or by group membership (other columns). In these group sub-panels: we show only groups composed of $\gtrsim 100$ particles for clarity; list the total number of groups; the fraction of host in groups; the number of particles for the 4 largest such groups; and show the parent 3DFOF halo’s particles and $R_{\Delta\rho_c}$ with gray points and a gray circle respectively.

where \mathbf{x}_i is the i^{th} particle’s position, and ℓ_x is the linking length. This can be again processed with a phase-space 6DFOF algorithm if desired.

The code will then ensure that all large-scale FOF halos reside within a single MPI domain and then search for substructures (clusters-within-clusters) using a higher dimensional phase-space FOF algorithm on particles that appear to be *dynamically distinct* from the mean halo background. This portion is computationally intensive.

Once all clusters and the relation to other clusters have been identified, the code then does a significant amount of data analysis on the clusters, calculating a wide-variety of properties. This portion is computationally intensive.

It then writes the results to disk using the HDF5 parallel IO library. The main output data consists of the properties of clusters and the particle IDs belonging to the clusters. This output can be split into several files up to one file per MPI process or can produce a single large file.

An example of the halos and substructures found by VELOCiraptor is presented in Fig. 2. Here, this single object is composed of a million particles and a large simulation will contain many such clusters, hence the computational challenge.

3 MPI

The challenge in this data analysis is that simulations can contain billions to trillions of particles, each particle requiring ~ 100 bytes [see for instance 5, 10]. This necessitates the use of MPI to fit the problem of running and analysing such simulations in memory. Here we describe how VELOCiraptor uses MPI to ensure a load-balanced run with minimal communication and fast IO.

3.1 Domain decomposition

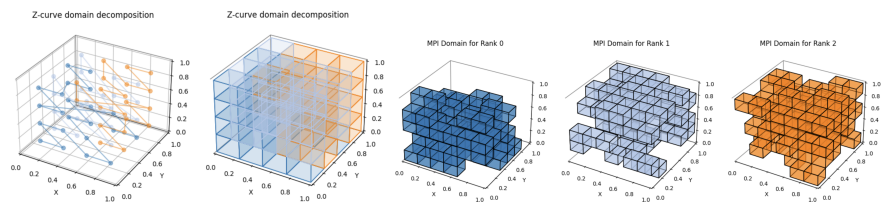


Fig. 3. MPI Decomposition: Decomposition of cubic volume using Z-curve (upper row) and a more representative decomposition between 3 MPI processes for a non-uniform particle distribution (bottom row).

Since input data consists of highly clustered data with a simulation volume, load balancing is critical. This load balancing is not just about ensuring similar

memory footprints per MPI process but also reducing the amount of point-to-point communication between MPI processes. VELOCiraptor does this by using a mesh placed across the input domain and then assigns cells in the mesh to a given MPI process using a space-filling Z-curve. The space-filling curve, an example of which is shown in Fig. 3, satisfies both requirements, assigning cells to a given MPI process as it goes along with curve and ensuring most of the volume assign to a MPI process is spatially clustered.

This decomposition is not static as clusters are localised to individual MPI processes, resulting in cells within the mesh occasionally assigned to multiple MPI processes. Clusters split across MPI processes are set to the MPI process with the fewest particles in its MPI domain, ensuring load balancing is maintained.

3.2 Communication

The cluster identification process is spatially localised, however clusters can span several MPI domains, each containing some fraction of the cluster. These means that cluster identification process is dominated by point-to-point communication, where each MPI process i does not know *a priori* what other MPI processes contain relevant particles, nor what other processes might require from it.

VELOCiraptor reduces the amount of communication using a combination of approaches. All processes know to which processes a cell in the MPI mesh belongs. Thus, as a MPI processes its particles is quickly able to determine if a particle's search window intersects the volume of a cell that belongs to other MPI processes using a fast binary tree. It uses this information to construct a communication work queue containing all communicating pairs and then iterates over the pair:

Listing 1.1. MPI communication queue

```

//NProcs = total number of Process in MPI_COMM_WORLD
std::vector<tuple<int, int>> MPIGenerateCommPairs(unsigned long long
    *send_info)
{
    std::vector<std::tuple<int, int>> commpair;
    for(auto task1 = 0; task1 < NProcs; task1++)
    {
        for(auto task2 = task1+1; task2 < NProcs; task2++)
        {
            if (send_info[task1 * NProcs + task2] == 0 && send_info[task2
                * NProcs + task1] == 0) continue;
            commpair.push_back(make_tuple(task1, task2));
        }
    }
    // ensure rank = 0 doesn't dominate communication pairs by
    randomizing
    unsigned seed = 4322;
    std::shuffle(commpair.begin(), commpair.end(),
        std::default_random_engine(seed));
    return commpair;
}

```

```

}
...

//sample code calling comm pairs
//now send the data.
auto commpair = MPIGenerateCommPairs(mpi_nsend);
for(auto [task1, task2]:commpair)
{
    if (rank != task1 && rank != task2) continue;
    auto [sendTask,recvTask] = MPISetSendRecvTask(task1, task2);
    // now proceed to point-to-point communication
    ...
}

```

This communication pattern is not just present in the first clustering process but is also present when estimating the density field. At this point, particles belonging to clusters have been localised, resulting in the MPI mesh with cells shared between MPI processes. Estimating the density field proceeds by using a fast binary tree on local particles to identify nearest neighbours. This nearest-neighbour list is incomplete and the radius of the distant nearest-neighbour of each particle is used to identify which MPI processes need to communicate. These particles are then exported to appropriate MPI processes to identify on those particles which particles need to be sent back. Again, this requires lots of point-to-point communication.

Some properties of clusters, such as the total mass enclosed in a spherical window centred on the cluster, also use a similar search of other MPI domains.

3.3 IO

VELOCiraptor uses the HDF5 library for writing parallel IO. However, it does not simply have all MPI processes read input and write output, which could produce significant amounts of MPI communication.

Input can be a single file or split between many files. VELOCiraptor can read this data with multiple MPI processes, where the number of reading MPI processes need not have any relation to the number of input files if the input is HDF5 input. Other raw binary data requires a single MPI process per input file. The number of reading MPI processes can be a subset of the total number of MPI processes and is set at runtime.

While reading input data, the code initialises two sets of communicators, one for MPI processes that will read input data and determine where to send this data and another that contains all the non-reading processes waiting to receive data. It spreads the processes reading across the `MPI_COMM_WORLD`, as shown in Fig. 5. MPI processes of similar rank are neighbours in the MPI domain decomposition (see Section 3.1) so to minimise the amount of internode communication, reading MPI processes are spread across the `MPI_COMM_WORLD`. This only happens if there is some spatial clustering in the input data, which is often the case.

Similarly, output is aggregated using the initialisation of MPI communicators, one for each file to be produced. However, in this case, MPI processes are close to each other in the `MPI_COMM_WORLD`, thereby reducing internode communication when coordinating file writes as shown in Fig. 5. The number of files produced can also be set at runtime.

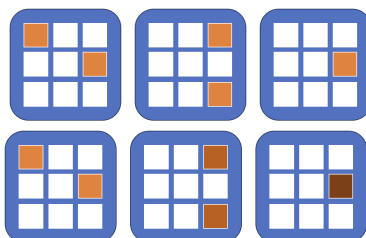


Fig. 4. MPI write process distribution: Example of 5 MPI reading processes in a 27 MPI process comm world reading single input file (top) and 3 files (middle). Here, 3 nodes each have 9 cores (indicated by blue region encompassing 9 squares). Cores with MPI processes reading from/writing to the file are coloured, with colour depending on file.



Fig. 5. MPI read process distribution: Example of 8 files being written by a 27 MPI process comm world. Figure is similar to Fig. 4.

4 OpenMP

VELOCiraptor makes extensive use of OpenMP parallelism throughout the code. A key part of the code is its fast, purpose-built binary tree library [see 8, for discussion of binary trees]. Building an N-dimensional binary tree can be time consuming, particularly if the space spanned by the tree requires non-Euclidean distances to determine the distance between particles. To speed-up tree construction, the code uses recursive task parallelism and a thread pool class to ensure that the nested parallelism does not overproduce threads:

Listing 1.2. Recursive OpenMP-enabled Binary Tree build.

```
//thread pool class to manage recursive parallelism
struct KDTreeOMPThreadPool{
```

```

    unsigned int nthreads;
    unsigned int nactivethreads;
    vector<unsigned int> activethreadids;
};
//build tree
KDTree::KDTree()
{
    // set some parameters
    ...
    // then set a thread pool
    KDTreeOMPThreadPool otp = OMPInitThreadPool();
    root=BuildNodes(0,numparts, otp);
    BuildNodeIDs();
}
// recursive node building
Node *KDTree::BuildNodes(int start, int end, KDTreeOMPThreadPool &otp)
{
    // if a leaf node is found return leaf node, return leaf node
    if (isleafflag) return new LeafNode(id, start, end, bnd, ND);
    // otherwise recursively call buildnodes
    // split input data
    auto splitindex = start + (size - 1) / 2;
    auto [splitvalue, splitdim] = SortData(...);
    //run the node construction in parallel
    if (otp.nactivethreads > 1) {
        //note that if OpenMP not defined then ibuildinparallel is false
        Node *left, *right;
        vector<KDTreeOMPThreadPool> newotp = OMPSPplitThreadPool(otp);
        #pragma omp parallel default(shared) num_threads(2)
        #pragma omp single
        {
            #pragma omp task
            left = BuildNodes(start, splitindex+1, newotp[0]);
            #pragma omp task
            right = BuildNodes(splitindex+1, end, newotp[1]);
            #pragma omp taskwait
        }
        return new SplitNode(splitdim, splitvalue, size, start, end,
            left, right);
    }
    // if not enough threads available, then just proceed with serial
    // recursive calls
    else {
        return new SplitNode(splitdim, splitvalue, size, start, end,
            BuildNodes(start, splitindex+1, otp),
            BuildNodes(splitindex+1, end, otp));
    }
}
}

```

The other use of OpenMP is in the initial FOF cluster finding algorithm. Each OpenMP thread is given a subset of particles to link in a given subvolume. The linking across these subvolumes follows the same approach as the linking across MPI domains. Initially, a thread private group association is determined for particles in the subvolume. Then particles with search regions extending to other OpenMP volumes are “exported” to the appropriate thread and linked to relevant particles, with particle group associations iteratively updated. The approach of mirroring an MPI approach is not common but this approach allows the code to operate in a mode with minimal MPI decomposition, with each MPI process having large number of particles and large number of threads and still perform efficiently.

OpenMP is also used to parallelise some computationally heavy for loops. The substructure search involves processing each parent structure independently. Structures can vary greatly in size, with many consisting of a few tens of particles, to others, such as in Fig. 2 composed of a million particles and containing hundreds to thousands of substructures. Hence, we heavily use dynamic scheduling with a chunk size of 1. A similar approach is taken when analysing the properties of each structure. A large number of properties are calculated but most are independent of any other structure.

5 Summary

We have presented VELOCIraptor, a C++17 MPI+OpenMP code and highlighted specific novel applications of MPI and OpenMP APIs to improve code performance and scaling. Future work will involve more extensive use of OpenMP task parallelism and the addition of GPU offloading. GPU offloading is of particular interest given the available compute but provides a significant challenge due to current structure of processing structures independently and most clusters being small. The required restructuring, though extensive, could provide a significant performance boost.

Acknowledgements

We would like to acknowledge the Whadjuk people of the Noongar nation as the traditional custodians of this country, where the Pawsey Supercomputing Research Centre is located and where we live and work. We pay our respects to Noongar elders past, present, and emerging. This work was supported by resources provided by the Pawsey Supercomputing Research Centre with funding from the Australian Government and the Government of Western Australia. PJE would like to thank all the developers and authors of the original VELOCIraptor paper.

Bibliography

- [1] Message passing interface forum, mpi: A message-passing interface standard. <https://www.mpi-forum.org/>, <https://hpc.nmsu.edu/discovery/mpi/introduction/> (Mar 1994), accessed: 2022-12-12
- [2] Dagum, L., Menon, R.: **OpenMP**: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE* **5**(1), 46–55 (1998)
- [3] Davis, M., Efstathiou, G., Frenk, C.S., White, S.D.M.: The evolution of large-scale structure in a universe dominated by cold dark matter. *The Astrophysical Journal* **292**, 371–394 (May 1985). <https://doi.org/10.1086/163168>
- [4] Elahi, P.J., *et al.*: Hunting for galaxies and halos in simulations with VELLOCraptor. *Publications of the Astronomical Society of Australia* **36**, e021 (May 2019). <https://doi.org/10.1017/pasa.2019.12>
- [5] Elahi, P.J., Welker, C., Power, C., Lagos, C.d.P., Robotham, A.S.G., Cañas, R., Poulton, R.: SURFS: Riding the waves with Synthetic UniveRses For Surveys. *Monthly Notices of the Royal Astronomical Society* **475**(4), 5338–5359 (Apr 2018). <https://doi.org/10.1093/mnras/sty061>
- [6] Group, T.H.: Hierarchical data format, version 5. <https://www.hdfgroup.org/HDF5>
- [7] Knebe, A., Pearce, F.R., Lux, H., Ascasibar, Y., Behroozi, P., Casado, J., Moran, C.C., Diemand, J., Dolag, K., Dominguez-Tenreiro, R., Elahi, P., Falck, B., Gottlöber, S., Han, J., Klypin, A., Lukić, Z., Maciejewski, M., McBride, C.K., Merchán, M.E., Muldrew, S.I., Neyrinck, M., Onions, J., Planelles, S., Potter, D., Quilis, V., Rasera, Y., Ricker, P.M., Roy, F., Ruiz, A.N., Sgró, M.A., Springel, V., Stadel, J., Sutter, P.M., Tweed, D., Zemp, M.: Structure finding in cosmological simulations: the state of affairs. *Monthly Notices of the Royal Astronomical Society* **435**(2), 1618–1658 (Oct 2013). <https://doi.org/10.1093/mnras/stt1403>
- [8] Knuth, D.E.: *The art of computer programming. Vol.1: Fundamental algorithms* (1978)
- [9] Schaller, M., Borrow, J., Draper, P.W., Ivkovic, M., McAlpine, S., Vandenbroucke, B., Bahé, Y., Chaikin, E., Chalk, A.B.G., Chan, T.K., Correa, C., van Daalen, M., Elbers, W., Gonnet, P., Hausammann, L., Helly, J., Huško, F., Kegerreis, J.A., Nobels, F.S.J., Ploekinger, S., Revaz, Y., Roper, W.J., Ruiz-Bonilla, S., Sandnes, T.D., Uyttenhove, Y., Willis, J.S., Xiang, Z.: SWIFT: A modern highly-parallel gravity and smoothed particle hydrodynamics solver for astrophysical and cosmological applications. *Monthly Notices of the Royal Astronomical Society* **530**(2), 2378–2419 (May 2024). <https://doi.org/10.1093/mnras/stae922>
- [10] Schaye, J., Kugel, R., Schaller, M., Helly, J.C., Braspenning, J., Elbers, W., McCarthy, I.G., van Daalen, M.P., Vandenbroucke, B., Frenk, C.S., Kwan, J., Salcido, J., Bahé, Y.M., Borrow, J., Chaikin, E., Hahn, O., Huško, F., Jenkins, A., Lacey, C.G., Nobels, F.S.J.: The FLAMINGO project:

cosmological hydrodynamical simulations for large-scale structure and galaxy cluster surveys. *Monthly Notices of the Royal Astronomical Society* **526**(4), 4978–5020 (Dec 2023). <https://doi.org/10.1093/mnras/stad2419>