METROC++

A parallel code for the computation of merger trees in cosmological simulations

Edoardo Carlesi
Leibniz Institut für Astrophysik
Potsdam, Germany
ecarlesi@aip.de

January 17, 2019

Contents

1 Introduction

METROC++ is an acronym that stands for **ME**rger**TR**ees **O**n **C++**, and refers to the infamous **Metro C** subway line in Rome, where the author of the code grew up and lived for more than 20 years.

This legendary piece of infrastructure (which is still - in 2019 - largely under construction, though a shorter section of it is already operating) took something of the order of a few Giga-Years to be built, a time span which can be compared to the formation time of most dark matter halos. The name of the code is a tribute to this mythological pillar of the Roman public transportation system, which connects through space different points of the city just like a merger tree connects a Halo through different points in time.

The code is distributed under the terms of the GNU Public License, and can be freely downloaded from this `GitHub` repository.

METROC++ has been written in `C++` and relies on `MPI2.0 C`-bindings for the parallelization.

Since `C++14`-style syntax is used at some points, in particular for the manipulation of (ordered) maps, the source code needs to be compiled with a non-prehistoric version of the compiler. A series of python and `bash` scripts for the post processing, analysis and visualization of the merger trees are also provided together with the sources, inside the `scripts/` and `python/` subfolders. However, the post-processing scripts are very basic and might require a substantial tweak of the variables, so that the user may prefer to write his own post-processing routines instead. A basic description of the algorithm and design of the code is provided in section 4.1. In this user's guide, we explain the basics for the setup and running of `METROC++`

2 Basic setup

2.1 Compiling the code

The main folder contains a `Makefile.config` file with some options related to the performance of the code and the setup of the machine the code will be running on. Apart from a modern, $C++14$-compatible compiler and a working `MPI-2.0` installation, the code does not require the installation of any additional library. The python scripts which are provided for post-processing might depend on libraries such as `numpy` and `matplotlib`, however these are still largely being coded and are provided only for a very quick-and-dirty peek into the output.

`-DZOOM`: When this flag is switched on, the comparison algorithm is optimized for the comparison of the particle content of halos in zoom-in simulations, avoiding the buffer and overhead which is required by the full box comparison mode. Trees for zoom-in simulations can still be computed without switching this flag on, using one MPI task only.

`-DVERBOSE`: This flag controls the output of the program - when enabled, the code will dump a lot more of (boring) information at runtime, on inter-task communication, buffer sizes, number of particles and halos exchanged and so on.

`-DNPTYPES=N`: Here we control the number of particles being tracked separately. For reasons of internal consistency of the code, N needs to be set greater or equal to 2. Using the minimum amount of particle types required results in some marginal gains in the memory usage and code speed.

`-DNOPTYPE`: When this option is set, we disregard the type and do not differentiate between different particles, so that each particle type has the same weight when computing the merger trees.

2.2 Code operation modes

The code has two basic modes of operation: *tree-building* (`runMode=0`) and *post-processing* (`runMode=1`); `runMode=2` will execute the post-processing routine right after the tree-building is finished. **The post-processing mode is not yet fully implemented in the code at the moment so that run modes 1 and 2 are disabled for the moment.** The number N of cores on which the code should run on has to be chosen according to the number of `AHF` files each catalog is split into. If each AHF_halos file is divided into `nChunks` parts, the number N_{MPI} of `MPI` tasks should be smaller than `nChunks` and allow for to be equally divided among the tasks. For instance, if `nChunks`$= 100$ setting $N_{MPI} = 10$ each MPI task would read in 10 pieces of the halo catalog at any given redshift. Due to the fact that different tasks need to exchange buffers, an increase in the number of task does not necessarily mean an increase in the speed of the code, as this will also increase the size of the buffer data that needs to be exchanged. Ideally the number of

2.3 Configuration file

The configuration file templates can be found in the subfolder `config/` where a few examples are provided.

2.4 Temporary files

The code produces a number of temporary files (`.tmp` format extension, located in the `tmp/` folder). These files are needed at runtime to They can be produced manually or using the scripts (`find_z.sh`, `find_n.sh` located in the `scripts/` folder) and contain the list of files on which the halo finder will run on. This can be the full list of snapshots in one simulation, or can be edited to be only a subset of it.

3 Examples

3.1 Full box simulation

To properly run the code for full box simulations, the `-DZOOM` flag needs to be commented in the `Makefile.config` file. `mpiexec -n N ./bin/metroCPP config/zoom_test.cfg`

3.2 Zoom simulation

To properly run the code on zoom simulation, the `-DZOOM` flag needs to be switched on in the `Makefile.config` file. Zoom simulation can run on a single MPI task only, and the code needs to be executed as e.g.:

```
mpiexec -n 1 ./bin/metroCPP config/zoom_test.cfg
```

4 Advanced features

4.1 Code performance

The

4.2 Code structure

- `main.cpp` A wrapper for the functions determined elsewhere
- `utils.cpp` General functions and utilities are implemented here
- `spline.cpp` The spline class is used for interpolation
- `global_vars.cpp` A list of global variables accessible throughout the whole program

- `MergerTree.cpp` This file contains the merger tree class, that tracks the merging of the halos, and series of functions (FindProgenitors) that compute the trees themselves.
- `IOSetting.cpp` Input/Output settings
- `Cosmology.cpp` Everything related to cosmological calculations, gravity solver to reconstruct the orphan halo positions and velocities
- `Communication.cpp` Handles most of the communication among tasks - sending / receiving buffers and so on
- `Grid.cpp` This handles the grid on which halos are placed and the buffers
- `Halo.cpp` The halo class contains the main halo properties

4.3 Compatibility with other halo finders

Although `METROC++` was conceived and mainly tested using the `AHF` halo finder, it can be easily extended to support other software as well, as long as:

- Halo catalogues include informations about the number and types of particles, positions and velocities for each object
- Particle catalogs contain the (unique) IDs for each halo particles' content

To add