METROC++ v0.1

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

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

May 2, 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:

https://github.com/EdoardoCarlesi/MetroCPP.

METROC++ has been written in C++ and relies on MPI2.0 C-bindings for the parallelization; there are also a number of bash scripts used to automatically and consistently read-in large number of halo and particle catalogs.

Since C++11/14-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.

To address the issue of the so-called *orphan* halos, i.e. halos (usually subhalos) unable to be linked to their parent halo for one or more time steps, the code keeps track of their particle content for a number of snapshots proportional to the halo mass. In the usual mode of operation, though, it does not attempt to reconstruct the missing halos' positions and velocity, a feature which is provided in the post processing mode of operation of the code.

A basic description of the algorithm and design of the code is provided in section 4. In this user's guide, we explain the basics for the setup and running of METROC++.

2 Compiling and running 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, $C++11$-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.

Compile-time options: The `Makefile.config` file contains a few options that need to be switched on at runtime to optimize and speed-up the execution of the code

`-DZOOM`: When this flag is switched on, the comparison algorithm is optimized for the 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.

Once the adequate compile-time options are set, just type `make` in the root directory, and a binary executable named `MetroCPP` will be placed in the `bin/` subfolder.

Execution: To run the code, simply type:

```
mpiexec -n N_MPI ./bin/MetroCPP config/your_configuration_file.cfg
```

from the terminal in the installation directory of the code, using a properly setup for the configuration file. The number N_{MPI} of tasks 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.

2.1 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 post-processing routines will be smoothing out the mass accretion histories of the halos properly taking into account temporary fly-by of subhalos, bound satellites partially orbiting outside of the viral radius of their host (giving rise to large mass fluctuations) and reconstruction of the orbits of the untracked halos.

2.2 Configuration file

The configuration file templates can be found in the subfolder `config/`. For the moment, these files are designed for the `AHF` format only, but can be easily extended to other halo finders as well. The expected structure of the input file is `prefix_XXX.NNNN.zZ.ZZZ.suffix` (for those produced with the MPI version of `AHF`) or `prefix_XXX.zZ.ZZZ.suffix` (for the serial AHF); where the `prefix` and `suffix` are specified by the `haloPrefix`, `haloSuffix` for the halo catalogs and `partPrefix` and `partSuffix` for the particle lists. Then we need to set the installation path of the program with the `pathMetroCpp` parameter, the path storing the input files (`pathInput`) and the path where the output files will be printed to, `pathOutput`.

2.3 Temporary files

The code produces a number of temporary files (`.tmp` format extension, located in the `tmp/` folder), containing some information about the input files (such as their numbering and their redshift); these files are produced at each run but have to be manually removed after each run. Otherwise the code will not attempt to write new files but rather read those already existing inside the `tmp/` folder. 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. These scripts produce three `.tmp` files in the `tmp/` subfolder called: `output_id.tmp` (which contains the snapshot unique number, usually in the form of `XXX` e.g. `001`, `023 256` etc.); `output_n.tmp` (that simply contains the total number of snapshots found, for consistency check) `output_z.tmp` (which stores the redshift corresponding to each snapshot). These files can be also produced manually e.g. if the user wants to select a subset of snapshots for the comparison without using them at all the timesteps. These files are read in at the beginning of each execution of the code; they are not needed for the subsequent run and should be deleted. They are not automatically erased each time in order to give to the users the opportunity to edit or write their own list of catalogues to be compared.

2.4 Input format

Halo catalogs: At the time being, `METROC++` supports `AHF` ASCII halo catalogs, of the form:

```
#ID(1)  hostHalo(2)     numSubStruct(3) Mvir(4) npart(5)
Xc(6)   Yc(7)   Zc(8)   VXc(9)  VYc(10) VZc(11) Rvir(12) [...]
```

The reader can be easily modified for alternative halo catalog formats.

Particle lists: Particle files must contain a list of the unique particle IDs that belong to each halo, and possibly their type (gas, dark matter, stars...). The first row of the file should contain the total number of halos stored in the file; while the following lines should be structured as follows:

```
HaloID (1)  Particle Number (2)
Particle ID (1) Particle Type (2)
```

i.e. after specifying the halo ID and the number of particles it contains an actual list of particle IDs and types should follow. When switching on the `-DNOPTYPE` option, only the particle ID will be read and the rest of the line will be ignored.

2.5 Output format

The code produces a series of `ASCII` files in the `.mtree` format as well as a logfile with the `.txt` extension. The `.mtree` files contain basic information about the descendant-progenitor halos at each step, specified by the header file printed by the master MPI task:

```
# ID host(1) N particles host(2) Num. progenitors(3)
Orphan[0=no, 1=yes](4)
# Common DM particles (1) ID progenitor(2) Num. particles(3)
```

In the case of orphan halos, which are signaled by the 1 on the fourth column, the information about the dumped about the progenitor is a copy of the parent halo, keeping the same halo ID and number of particles.

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.

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
```

This configuration of the code assumes that all the important halos are located in the same

4 Code properties

4.1 Algorithms

When dealing with cosmological full box simulations, the code reads in the halo catalogs among different tasks, so that each task will only hold a fraction of the total halo and particle content of the simulation. To ease the access and communication of data, a grid is placed on the simulation box on which each halo is assigned to its nearest grid point (NGP). A node can be shared among different tasks. Then, each task sends and receives the information for halos and particles at the edge of the subvolume of the box it contains. This buffer is needed to consistently compare the particle content of objects which may be close but located on different tasks at different steps.

4.2 Code structure

The `src/` subfolder of the code contains the main source files:

- `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 of e.g. $a(t)$ and $z(t)$, which are read from tabulated quantities.

- `global_vars.cpp` A list of global variables accessible throughout the whole program
- `MergerTree.cpp` This file contains the merger tree class (tracking the pairwise connections of halos in different catalogs) as well as functions (e.g. `FindProgenitors()`) that compute the themselves.
- `IOSetting.cpp` Input and output routines and 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 class handles the grid on which halos are placed, and computes the buffer zones that need to be communicated among tasks.
- `Halo.cpp` The Halo class contains the basic halo properties and functions.

The `python/` subfolder contains some useful phyton libraries used to post process and store the data in a standard `SQL` database, to ease the access to the fully reconstructed merging histories querying the halo ID value of $z = 0$ objects.

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