The:

CubeP^{3}M $\rightarrow$ Chunked data $\rightarrow$ AHF $\rightarrow$ Haloes

Pipeline

William Watson

w.watson@sussex.ac.uk

1 Overview

The post processing of data from today's largest cosmological N-body simulations is a significant challenge. Vast (>TBs) amounts of data need to be read and analysed just for one timeslice of a large simulation to be processed. Where possible today's cosmological simulation codes produce reduced data on-the-fly so as to avoid the difficulties of post processing the simulation output. Nevertheless there are frequent occurrences where the need to post process the data is unavoidable.

This documentation outlines a pipeline to achieve one such post processing step: that of running a halofinder on the particle data from an N-body simulation. Typically this is a trivial task as the majority of simulations being run today are small enough to enable halofinding codes to be run directly on their particle data. For very large simulations however (and in this case the cut-off between trivial and large is somewhere around the 30 billion particle mark) the problem becomes much harder due to issues with code scalability of the halofinders in question.

Here we discuss the specific pipeline constructed to run the Amiga Halo Finder (AHF) on data from the CubeP^{3}M N-body code. AHF can run out-of-the-box on an entire CubeP^{3}M simulation[1] provided the particle count is fewer than about 30 billion (and there is a suitably large machine available for the analysis). For simulations with more particles in them it is necessary to take a different approach.

We make use of the fact that halofinding is a local process (i.e. to find a halo in a particular location in a simulation we only need to analyse the particles in the region around that halo) and split the simulation box up into sub-regions called 'chunks'. In each chunk we can then run individual instances of AHF as if they were being run on individual cosmological simulations. One subtlety makes this process slightly more complicated: each chunk needs to contain a buffer zone around it containing extra particles from the simulation. This is necessary to correctly identify haloes on the boundaries of each chunk and allocate their properties accurately. Figure 1 illustrates how the chunks relate to a CubeP^{3}M simulation's standard domain decomposition.

The entire procedure is summarised as follows:

1) take a CubeP^{3}M simulation output timeslice of particle data
2) run the `chunk_cubep3m.f90` code on it to split it into chunks
3) run AHF on the individual chunks
4) combine the AHF data into one coherent halo catalogue.

2 `chunk_cubep3m.f90`

The code `chunk_cubep3m.f90` is a fortran 90 code implemented with an MPI parellisation. To successfully run the code on an output timeslice of particle data from a CubeP^{3}M simulation a number of steps need to be implemented:

[1]We discuss briefly this case in the AHF section below

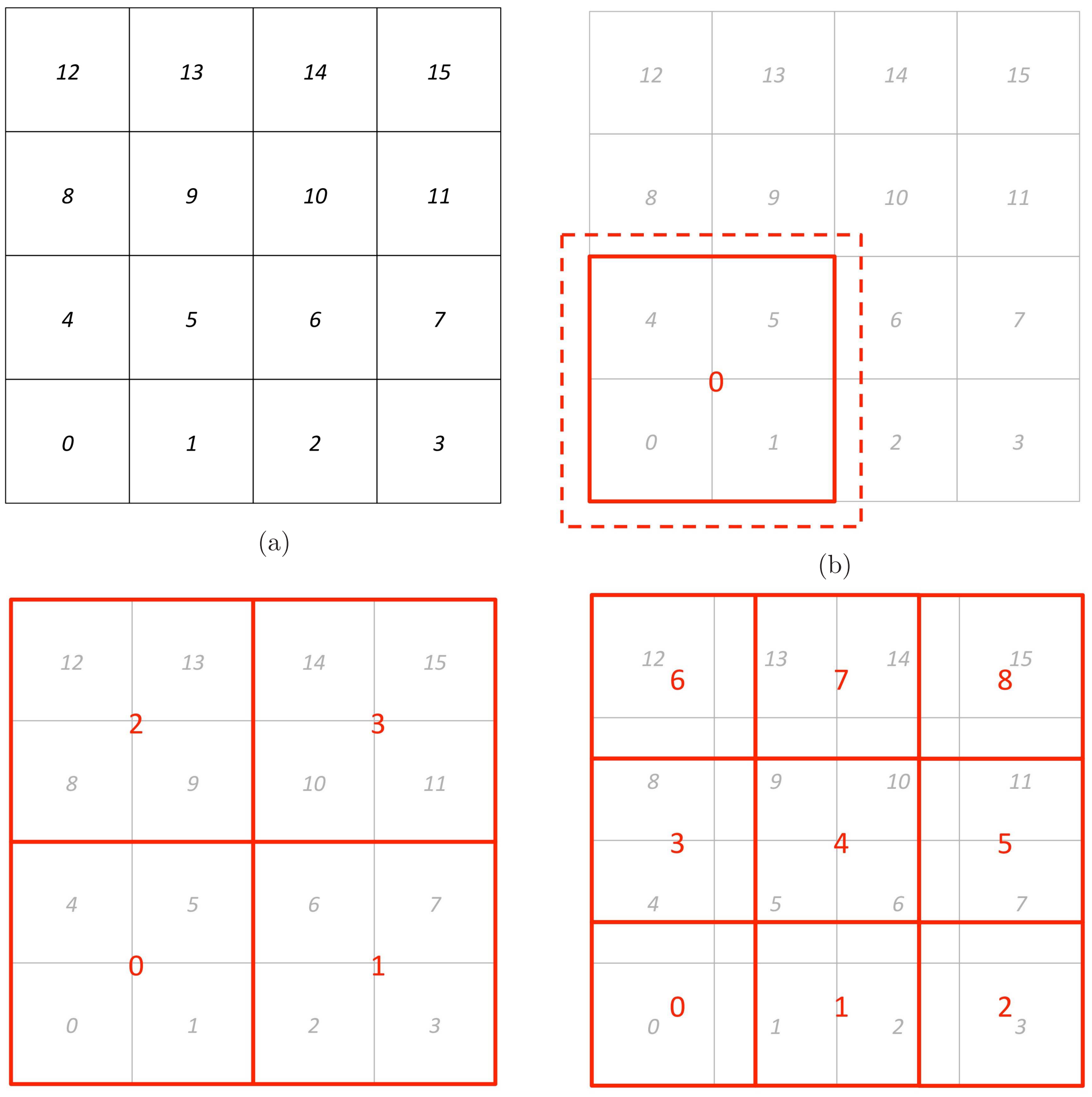

Figure 1: Schematic of the relationship between the CubeP^{3}M domain decomposition and the chunks it is split up into by `chunk_cubep3m.f90`. The first x-y plane layer of nodes from a CubeP^{3}M simulation (with nodes per dimension = 4) is shown in black in panel a). In panel b) we show in red an example of one chunk and its buffer zone, in c) we show all the chunks that would be laid down in this particular plane. In panel d) we illustrate that the chunks, whilst being cubical themselves, typically do not need to align with the boundaries of the CubeP^{3}M nodes (although this may be desirable).

1) The code needs to be compiled. Currently the code supports using either the Intel Fortran compiler (`ifort`) or the Gnu Fortran compiler (`gfortran`).

2) The appropriate directory structure for the output data needs to be constructed. The final product of the chunking code is a collection of files for each chunk that need to reside in their own directory.

3) An appropriate parameter file needs to be constructed. This file is read in by the code at runtime and contains important parameters that relate both to the details of the CubeP^{3}M simulation and to the desired chunking scheme.

4) The code needs to be executed. A machine-dependent batch file needs to written (if necessary) to run the code in the correct MPI configuration.

2.1 Compilation

The current version of the chunking code is v1.5. The current filename for this version is `chunk_cubep3m_v1.5.f90`. The relevant compilation commands for the GNU and Intel compilers are:

GNU Fortran:

```
mpif90 -cpp -O3 chunk_cubep3m_v1.5.f90 -o chunk -DGFORTRAN
```

Intel Fortran:

```
mpif90 -fpp -O3 chunk_cubep3m_v1.5.f90 -o chunk -DBINARY
```

Note that the '`-cpp`' flag in `gfortran` sometimes does not work on certain machines with certain versions of the GNU compiler. In this case try using '`-x f95-cpp-input`' instead.

2.2 Directory Structure

The `chunk_cubep3m.f90` code will read data from a directory designated fully in the parameter file (see below) and write data to a group of sub directories within an output directory designated in the parameter file (see below). As such, the directory structure for the output needs to be fully created on the machine that is performing the chunking before the code is run.

The output directory path is set in the parameter file (see below), for example as:

```
/path/to/output/data/
```

Within this directory sub directories need to be created for each redshift output the `chunk_cubep3m.f90` code is being run on. These take the syntax

```
z_<redshift>
```

with `<redshift>` being a number with three decimal places. For example, if the `chunk_cubep3m.f90` code is run on outputs at redshifts 0, 0.5 and 1.125 then the directories would need to be called `z_0.000`, `z_0.500`, and `z_1.125`, giving a directory structure that looks like:

```
/path/to/output/data/z_0.000/
/path/to/output/data/z_0.500/
/path/to/output/data/z_0.125/
```

Within each redshift sub directory a subdirectory then needs to exist for each of the chunks that the data is being to split up into. These take the syntax

```
chunk_<chunk_num>
```

with `<chunk_num>` taking an integer value from 0 to the number of chunks – 1. For example, if we were splitting the simulation into 8 chunks (i.e. a 2 by 2 by 2 decomposition) then, with the same redshifts as before, the complete directory structure would need to be:

```
/path/to/output/data/z_0.000/chunk_0/
/path/to/output/data/z_0.000/chunk_1/
/path/to/output/data/z_0.000/chunk_2/
/path/to/output/data/z_0.000/chunk_3/
/path/to/output/data/z_0.000/chunk_4/
/path/to/output/data/z_0.000/chunk_5/
/path/to/output/data/z_0.000/chunk_6/
/path/to/output/data/z_0.000/chunk_7/
/path/to/output/data/z_0.500/chunk_0/
...
/path/to/output/data/z_0.500/chunk_7/
/path/to/output/data/z_1.125/chunk_0/
...
/path/to/output/data/z_1.125/chunk_7/
```

2.3 Parameter File

The `chunk_cubep3m.f90` code reads in a parameter file at run time that controls the characteristics of the chunking procedure. An example parameter file would look something like:

```
# Redshift to be chunked (set to -1 to use an input file):
2.000
#
# Redshift input file string (if -1 is set above.  File needs to be ascii
and contain redshift on first line):
input_redshift
#
# Particle data path:
/scratch/particle_data/cubepm_110526_3_216_20Mpc/output/
#
# Chunk output path:
/scratch/chunk_example/cubepm_110526_3_216_20Mpc/chunked_output/
#
# CUBEP3M box length (in Mpc/h):
20
#
# CUBEP3M Nodes per dimension:
3
# CUBEP3M fine mesh cells per dimension:
432
#
# PID flag.  Set to 1 if PIDs are available or 0 if not:
0
#
# Buffer size in Mpc/h:
1.0
#
# Chunking dimensions (number of chunks in x,y,z):
2
2
2
```

Any line beginning with a hash symbol (`#`) will be treated as a comment line. The code parses this parameter file in such a way that the order of the parameters in the file needs to be maintained as in the example above. The individual entries are as follows.

Redshift to be chunked

This is the redshift of the output timeslice of particle data to be split into chunks. This redshift must correspond both to the redshift that prefixes the CubeP^{3}M output particle data (for example, `6.000xv23.dat` would require a redshift of 6.000) and the redshift in the output directory structure detailed above.

It may be preferable to have the redshift to be chunked read in from another file (for example if multiple redshifts are being processed in a batch job it is simpler for a shell script to update a file containing just the input redshift rather than the entire parameter file). To do this enter a redshift of -1 in the parameter file and provide the name of the file to be read in on the next line of the parameter file.

Redshift input file string

Regardless of whether the redshift is set as -1 or not the code requires that this variable be in the parameter file, even if it is just a dummy string. If -1 has been set as the redshift then this variable in the parameter file is taken to be the name of a file (or path to a file as well) that contains the redshift of the output timeslice to be chunked. The file needs to be an ASCII formatted file and contain the input redshift on only one line at the head of the file.

Path to particle data

This needs to contain the full path to the particle data. Note that the path needs to be terminated with an '/'.

Path to chunk output data

This is the path to the directory containing the chunked data. The directory itself needs to have a number of subdirectories inside it, as discussed in the section above. Note that this path needs to be terminated with an '/'.

CubeP^{3}M box length

This the length of cubep3m simulation box in Mpc/h. This can be simply taken from the CubeP^{3}M naming string, so, for example, for a simulation with the naming string:

```
cubepm_120701_10_6000_3Gpc
```

the box length would be 3000 Mpc/h.

CubeP^{3}M nodes per dimension

This is the number of nodes the CubeP^{3}M simulation was run on. For example the above simulation was run on 10 nodes, as given in the naming string:

```
cubepm_120701_10_6000_3Gpc
```

CubeP^{3}M fine mesh cells per dimension

This is the number of fine mesh cells per dimension in the simulation. This can be taken as twice the number of particles per dimension in the simulation, so, for example, the above simulation would have had 6000 particles per dimension and therefore the number of fine mesh cells per dimension would be 12000, as given in the naming string:

```
cubepm_120701_10_6000_3Gpc
```

Particle ID flag

Flag for particle IDs. Set to 1 if particle IDs are to be processed in the pipeline and 0 if not. Note if particle IDs are used then this needs to be reflected in the compilation flags in AHF and the chunk catalogue making code that stitches the chunks together as the final step in the process.

Buffer size

This sets the size of the buffer zone that will be placed around each chunk (see Figure 1b) in Mpc/h. Set this with care as if it is too large the process can become very inefficient. If it is too small then large haloes near the boundaries of the chunk may not have correct properties assigned to them as they may be missing particles. The recommended choice for the buffer size is a value of the order of twice the virial radius of the largest halo expected to form at a given redshift.

Chunking dimensions

These three parameters set the number of chunks the simulation will be split up into in the x, y and z directions. It is possible, but not advisable, to split the simulation up arbitrarily in the different dimensions. For cubical chunks (which are recommended) these three values should all be the same. In the example parameter script above the simulation will be split up into $2 \times 2 \times 2 = 8$ chunks as per Figure 1c (which shows only the bottom four chunks).

2.4 Code Execution

When the code has been compiled it can be run, for example, by an execution command such as:

```
mpiexec -n <number of tasks> ./chunk parameterfile
```

It is very important that the number of tasks used be equal to the number of nodes that the CubeP^{3}M simulation was run on. If not the code will return an error. The code is very memory light so will run on most machines even if each core on the machine only has a small amount of memory available to it.

3 AHF

4 Catalogue Making