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 parallelisation. 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

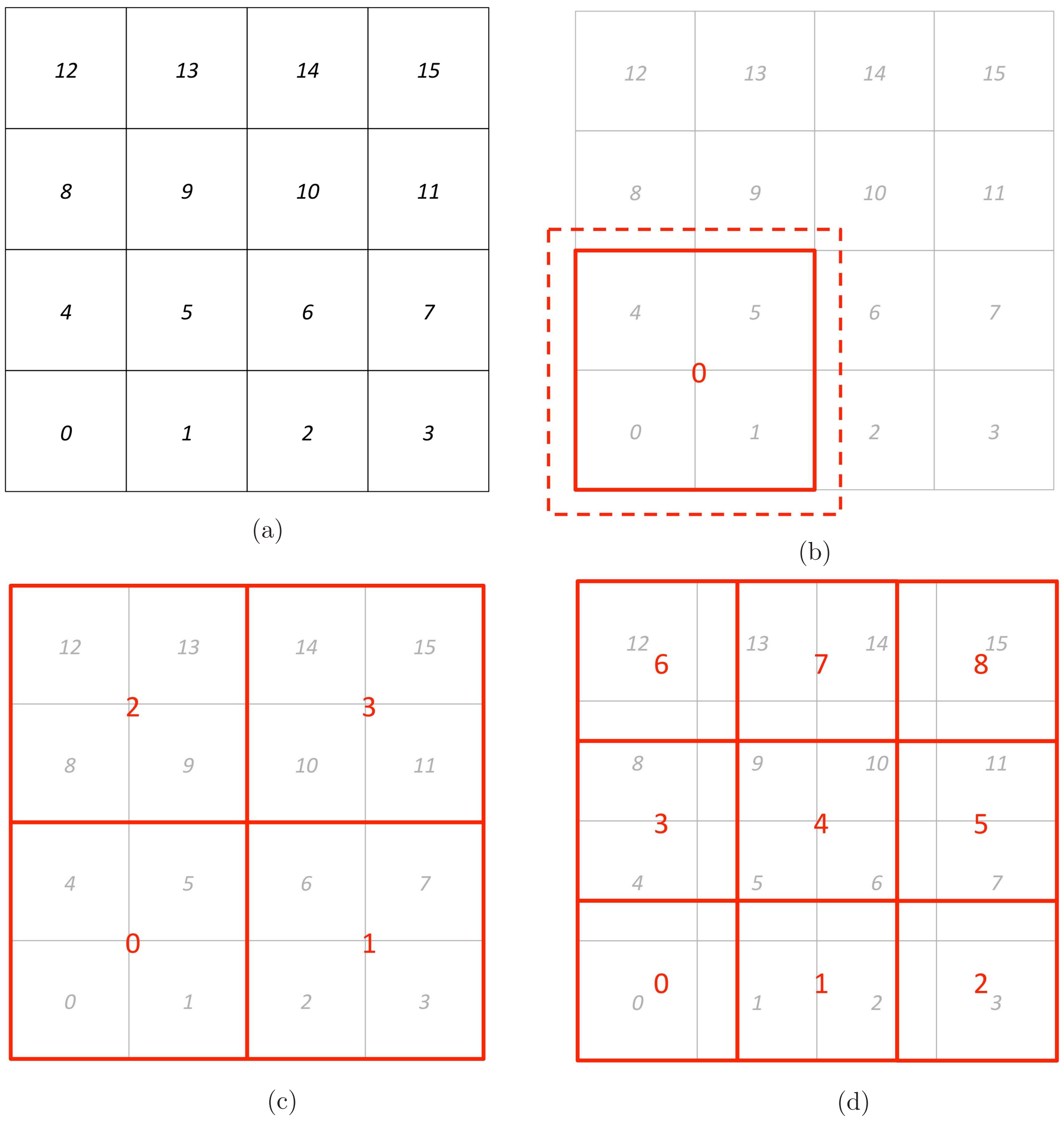

Figure 1: Schematic of the relationship between the CubeP^{3}M domain decomposition and the chunks it is split up into by `chunk_cubep3m.f90`. The first x-y plane layer of nodes from a CubeP^{3}M simulation (with nodes per dimension = 4) is shown in black in panel a). In panel b) we show in red an example 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.

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. Also, due to a peculiarity as yet undetermined, the parameter file must be invoked on the command line without using an absolute path to the file – i.e. one beginning with a '/'.

3 AHF

The AHF code is fully documented in its own right[2]. In this document we simply cover elements of running AHF that are specifically relevant to using it with CubeP^{3}M in general, and chunked CubeP^{3}M data in particular.

3.1 Compilation

To edit system specific compilation options AHF's `Makefile.config` file needs to be edited. For use on CubeP^{3}M data it is recommended that the system configuration is set using a hybrid OpenMP and MPI configuration. This can be set by ensuring the following line is uncommented:

```
SYSTEM = "Standard MPI+OpenMP"
```

Finally, certain specific define flags that are relevant to processing CubeP^{3}M data that may be of use. These are `-DAHFbinary` and `-DCUBEP3M_WITH_PIDS`. The `-DAHFbinary` flag turns on the AHF binary output option. With this option set the data AHF outputs will need to be read in by the final halo cataloguing code in binary format – see below for more details on this. The `-DCUBEP3M_WITH_PIDS` allows AHF to handle CubeP^{3}M particle IDs. If this is not turned on then AHF will assign random IDs to the particles.

3.2 Non-chunked CubeP^{3}M Execution

In order to run AHF on CubeP^{3}M data two specific details need to be observed. Firstly, the `ic_filetype` parameter in the AHF parameters file needs to be set to 21 in order for AHF to recognise that it needs to run on CubeP^{3}M data. Secondly, in the directory from which AHF is being run, an information file needs to be created that contains the details of the CubeP^{3}M simulation for AHF to read in. This file needs to be called '`cubep3m.info`' and should contain the lines:

```
0.27 CUBEP3M_OMEGA0
0.73 CUBEP3M_LAMBDA0
64 CUBEP3M_BOXSIZE [Mpc/h]
4096 CUBEP3M_NGRID [2x NPART]
```

[2] See http://popia.ft.uam.es/AHF/Documentation.html

```
8 CUBEP3M_NODES_DIM
```

where each of the entries should be taken from the details of the CubeP^{3}M simulation in question.

The specific configuration that AHF needs to be run in will depend on both the machine in question and the details of the simulation. A typical run may have 4 OpenMP threads per MPI task, as this generally gives a good payoff between memory depth and speed on machines with around 2-4 Gb of RAM available per core. AHF is typically executed as follows:

```
mpiexec -n 32 ./ahf_executable input.file
```

3.3 Chunked CubeP^{3}M Execution

For execution on chunked data AHF should be set up in precisely the same way, as it recognises chunked versus non-chunked data at runtime. Typically, however, a number of AHF instances will need to be run that equal the number of chunks the simulation has been split up into. As this is the case it is useful to create a different AHF input parameter file for each chunk. In each parameter file paths to the chunked data and to the output directory AHF needs to write to should be specified. As such it is therefore necessary to create a directory structure for the AHF output that is similar to the structure of the `chunk_cubep3m.f90` output directories. An example AHF input file for chunked data input might look like:

```
ic_filename = /scratch/cubepm_110708_8_2048_64Mpc/chunked_output/z_2.000/ch
unk_12/2.000xv_chunk_12_

ic_filetype = 21

outfile_prefix = /scratch/cubepm_110708_8_2048_64Mpc/AHF_output/z_2.000/chu
nk_12/2.000xv

LgridDomain = 128
LgridMax = 16777216
NperDomCell = 5.0
NperRefCell = 5.0
VescTune = 1.5
NminPerHalo = 20
RhoVir = 1
Dvir = 178
MaxGatherRad = 1.6
LevelDomainDecomp = 9
```

```
NcpuReading = 8
de_filename = my_dark_energy_table.txt
```

Note the implied directory structure of the AHF output. A description of the other parameters can be found in the AHF documentation.

4 Catalogue Making

The final step in the process is to remove data from the AHF output files that pertain to haloes that were detected in the buffer zones of the chunks and then to shift the halo coordinates into the global simulation coordinates. Each chunk is considered by AHF to be a separate simulation. In order for AHF to run correctly there is a zone of empty space as wide as the buffer parameter around the particle data. The particle data also contains a zone of extra particles in a buffer as wide as the buffer parameter (as illustrated in Figure 1b). This means that in order to piece everything back together again a) the haloes in the buffer region need to be discarded, b) each AHF halo needs to have its position shifted by twice the buffer size and c) each AHF halo then needs to be returned to its place in the global coordinate system of the simulation.

A code that handles all these steps has been created, `AHF_chunk_catalogue_maker.f90`. The latest version of the code is `AHF_chunk_catalogue_maker_v2.0.f90`. The code runs in serial and takes a parameter file as an input with various options detailed below. The code itself has a number of preprocessor options to handle various different instances that are detailed below.

4.1 Compilation

The code can be compiled using either GNU Fortran or Intel Fortran. A simple compilation line for each instance would look like:

GNU Fortran:

```
gfortran -cpp AHF_chunk_catalogue_maker_v2.0.f90 -o chunk_cat -DGFORTRAN
```

Intel Fortran:

```
ifort -fpp AHF_chunk_catalogue_maker_v2.0.f90 -o chunk_cat -DBINARY
```

However there are a number of extra preprocessor options that can be added at compilation. These are:

a) `-DPROFILES`:

This option will also analyse the AHF profile data and create output files containing the radial profile information of haloes in the simulation.

b) `-DAHFBINARY`:

This option needs to be set if AHF was run with the `-DAHFbinary` flag turned on. This means that the AHF halo finder outputted its data in a binary format.

c) `-DPIDS`:

This compilation option must be set if particle IDs from the haloes need to be processed along with the halo information.

d) `-DCUBEPM_DOMAINS`:

This option will create haloes in files that match the CubeP^{3}M code's domain decomposition (i.e. Figure 1a). If this is not set then haloes will be produced in files that match the chunked domains (i.e. Figure 1c).

e) `-DOUTPUT_BINARY`:

This option will produce all outputs in a binary format. This is useful for large datasets. The format matches the AHF binary output format. All output files will have a '`_bin`' suffix added to their filenames.

f) `-DWITH_BUFFER`:

The use of this option is not recommended. This will not remove haloes that are in the buffer zone of the chunks. The output in this case will be in a binary format.

4.2 Parameter File

As with the `chunk_cubep3m.f90` code the `AHF_chunk_catalogue_maker.f90` code is run with a parameter file. The contents of the parameter file are similar to, but not identical to the parameter file used for the `chunk_cubep3m.f90` code. An example parameter file would be:

```
# Chunk start (which chunk to begin processing):
0
# Chunk end (which chunk to process before stopping):
7
# Redshift:
2.000
```

```
#
# Redshift input file string (if -1 is set above.  File needs to be ascii
and contain redshift on first line):
input_redshift
#
# AHF data path:
/scratch/cubepm_110526_3_216_20Mpc/AHF_output/
#
# Catalogue output path:
/scratch/cubepm_110526_3_216_20Mpc/output/
#
# CubeP3M boxlength (in Mpc/h):
20
#
# CubeP3M Nodes per dimension:
3
#
# CubeP3M fine mesh cells per dimension:
432
#
# Buffer size in Mpc/h:
1.0
#
# Number of chunks in total:
8
#
# Number of AHF files per chunk:
2
```

As with the `chunk_cubep3m.f90` parameters file any line beginning with a hash symbol (#) will be ignored. The order of the parameters needs to be preserved as in the example above. The details are as follows.

Chunk Start and Chunk End

This selects the chunks that the catalogue code will run on. As the code is serial it can take some time to process very large simulations. However the nature of the code is such that it is easy to run in a parallel job script with multiple instances working on only a subset of the chunks. For the code to work on just one chunk simply set the start and end chunk numbers to be the same.

A word of caution is required here though. If the output of the code is desired to be in the CubeP^{3}M domain decomposition (i.e. the `-DCUBEPM_DOMAINS` flag has been set in the compilation of the code) then it is probable that two separate instances of the code, if running side-by-side, may want to write halo data into the same CubeP^{3}M domain.

If this is the case then there is a problem as the separate instances will have no way of knowing which one is to write into the CubeP^{3}M domain file at a given time and this will result in data corruption. Ways to avoid this include choosing the chunk schema to stride an integer number of CubeP^{3}M domains (as in Figure 1c), or avoiding running the code on chunks that will clash with each other (either by splitting up the chunks spatially – for example in a chess-board tiling – or temporally, for example processing the same chunk across different redshifts).

Redshift and Redshift Input File String

This is as per the `chunk_cubep3m.f90` code parameter file.

AHF Data Path and Catalogue Output Path

These are the paths to the AHF data and to the directory that the final catalogues should be written to. The AHF data path should be to the directory that contains the redshift folders. So for the example above, where the AHF parameter `outfile_prefix` is set as:

```
outfile_prefix = /scratch/cubepm_110708_8_2048_64Mpc/AHF_output/z_2.000/chu
nk_12/2.000xv
```

the AHF data path should be set as:

```
outfile_prefix = /scratch/cubepm_110708_8_2048_64Mpc/AHF_output/
```

CubeP^{3}M Boxlength, Nodes and Fine Mesh Cells

This is as per the `chunk_cubep3m.f90` code parameter file.

Buffer Size

This needs to be set to the same value as in the `chunk_cubep3m.f90` code parameter file.

Number of Chunks

This is the number of chunks that the data was split into. Note that the `AHF_chunk_-catalogue_maker.f90` code makes the tacit assumption that the chunks are cubical. As such this code is not appropriate to use with exotic choices of chunk dimensions.

Number of AHF Files

This is the number of AHF files produced for each chunk. This is equal to the number of MPI tasks that AHF was run with.

4.3 Execution

Execution is very simple as this is a serial code. A typical execute command would be:

```
./chunk_cat parameterfile
```

As with the parameter file input for the `chunk_cubep3m.f90` code if a path is used to the file it should not be an absolute path (i.e. one commencing with a '/') as this creates an error.

5 Example Scripts

To analyse a very large simulation on a large machine with a queuing system for parallel jobs it is necessary to use a submission script to run the pipeline. As halofinding is a method of data reduction the output from the AHF code will be much smaller in size than the input particle data. As such it is useful to run the pipeline all in one job in the following manner:

a) chunk a redshift
b) produce AHF haloes
c) delete redshift chunked data (so as to avoid clogging up storage)
d) goto a) and repeat for the next redshift.

Once the AHF data has been produced for all redshifts the catalogue creation can then be run as a final step.

Example scripts are included along with this documentation that show examples of scripts used on different super computers.