CUBE 2.0 Manual

Contents

1 Introduction

CUBE is an open-source, parallel N-body simulation code for cosmological large scale structures. It is written in Modern Fortran, where Coarray features replaces the MPI communication between computing nodes. Although Fortran is usually considered as an old language, it is efficient and is designed for scientific computing. In addition, usually minimal programming skills are required from scientists, and it has automatic optimizations for fast computation. Modern Fortran language is concise, especially for the multi-dimensional array computation and coarray communications. The code suffixes "`*.f90`" indicate that the code is written in Fortran free-form instead of fixed-form. From 2008 and later standards support Coarray Fortran (CAF) but I still name them as "`*.f90`".

CUBE is a two-level particle-mesh (PMPM) based code with particle-particle (PP) force interaction configurations available (from version 2.0). The initial code CUBE (CUBE 1.0) available at `https://github.com/yuhaoran/CUBE` and described by paper [**?**]. This manual describes CUBE 2.0. Based on version 1.0, it adds OpenMP, runtime halo-finder, PP force and additional memory

optimizations. Version 2.1 includes cosmological neutrino components modelled by N-body particles, dark matter halo angular momentum analysis, etc.

CUBE is designed to run on the massively parallel modern supercomputers. For N-body problems we usually consider the weak-scaling – how the performance varies with the number of processors for a fixed problem size per processor. This could be challenging for computing efficiencies, total memory, load balance, and communications.

CUBE uses integer-1 or integer-2 for particles' locations and velocities, so the memory usage per particle can be squeezed to 6 bytes. For the PM version of the code we demonstrated that storing particles using integer-1's does not affect the power spectrum [**?**], and using integer-2's (12 byte per particle) CUBE reproduces the exact result from the traditional single-precision floating number based algorithms.

The PMPM algorithm decomposes gravity into long and short forces. The coarse, global PM solver is based on a 4-times-coarser grid, so the global FFT computation and communication cost is $4^3 = 64$ times less. This feature keeps the fine-grid FFT fixed, and makes the computational complexity $o(N)$.

CUBE uses its own `pencil_fft` modules to do global distributed Fast Fourier Transform (FFT), and the only external module required is FFT (e.g. fftw3, MKL).

The PP force can be configured to be extended to multiple adjacent fine grid cells. With PP turned on/off, CUBE is one of the most-accurate/fastest N-body codes on the market.

CUBE 2.0 parallelizes in two layers, MPI (coarray) and OpenMP. The OpenMP keeps strong-scaling for more than 100 shared memory multi-processors. Communication is minimized by the two-level PM algorithm.

1.1 Pipeline

Refer to §2.3.2 and §2.3.3 of [**?**].

1.2 Requirements

CUBE needs a Fortran compiler and a FFT library. Fortran 2008 standard starts to support Coarray features, which is essential for CUBE. FFT is the key step to solve Poisson equations in gravity solver. A free library is FFTW (`fftw.org`). CUBE supports FFTW 3.

2 Benchmark

Here is an example of testing the code on my MacBook, where MacOS is a Unix based system.

2.1 Install FFTW

First let's install FFTW library. Download the `fftw-3.3.8.tar.gz` or similar versions from the website. Open a terminal and unzip it:

```
> tar -xvf fftw-3.3.8.tar
```
or
```
> tar -xzvf fftw-3.3.8.tar.gz
```

and go into the directory and configure and install:

```
> ./configure --enable-float --enable-threads --enable-openmp
   --prefix=/Users/haoran/opt/
> make
> make install
```

Check the output from screen carefully if there are errors and warnings. It will take a few minutes for each step. Note that we will need single precision (which is accurate enough) FFT routines installed. Depending on the system, the OpenMP might or might not work. If you have limited permission on the system you are working on, install FFTW on your local path by using "`--prefix=`". For the complete description of FFTW configuration see `http://fftw.org/fftw3_doc/Installation-on-Unix.html#Installation-on-Unix`.

If successful, we will able to see `include` and `lib` directories and FFTW files inside.

```
> ls /Users/haoran/opt/
bin    include    lib    share
```

Reference them in the in the CUBE configurations below.

2.2 Install CUBE

Go to the directory where you want to install the code, and clone the code from github:

```
> cd /Users/haoran/cloud/
> git clone https://github.com/yuhaoran/CUBE_XMU.git
```

then you will find a new directory named `CUBE_XMU`. There are initially four folders in it.

```
> cd CUBE_XMU/CUBE_v2.0/
> ls
kernels    tf    work    velocity_conversion
```

They stand for gravitational force kernels for PMPM algorithm, cosmological transfer functions, working directory, and velocity dispersion statistics for CUBE checkpoint format conversions, respectively. Go to the directory `work`,

```
> cd work/
> ls
main    utilities
```

The `main` directory stores codes for the main N-body simulation. The `utilities` directory has

codes generating initial conditions of the simulation, and analysis tools for the simulation results.

Configure some environment variables

```
> vim utilities/module_load_mac.sh
export FC='gfortran'
export FFTFLAG='-I/Users/haoran/opt/include/ -L/Users/haoran/opt/lib/
-lfftw3f -lfftw3 -lfftw3_omp -lfftw3f_omp -lm -ldl'
export OMP_NUM_THREADS=4
```

Here we set the Fortran compiler name, FFT flags, and number of OpenMP threads (could be the number of available cores of the system). Remember to change the FFTW paths based on your FFTW installations. For intel compiler, use "`-mkl`" instead of FFTW flags can usually achieve a faster computing speed. If FFTW is installed without OpenMP, remove the flags "`-lfftw3_omp -lfftw3f_omp`".

With these configured, go to directory `main` and run the test script:

```
> cd main
> source run.sh
```

It runs everything from end to end. This script contains the initial setup by "`source ../utilities/module_load_mac.sh`" and can be expanded as the following steps for compiling and running the code.

3 Code overview

3.1 Basic parameters

`main/parameters.f90` sets the most basic parameters of the simulation. If the above `run.sh` is successfully completed, it actually simulates $N_p = 128^3$ N-body particles, representing cold dark matter (CDM), in a $L = 50$ Mpc/h per side cubic box, with periodic boundary conditions. The initial redshift is set to be $z = 99$ where the universe is $a = 1/(1 + z) = 0.01$ of the current size, where a is the scale factor. The final redshift is $z = 0$ ($a = 1$). These, and optionally intermediate redshifts, can be set in `main/z_checkpoint.txt`.

3.2 Initial conditions

The initial condition generator is written in `utilities/initial_conditions.f90`. Again, set apply the environmental variables in `module_load_mac.sh` first, and compile the initial condition code. In the directory `utilities/`:

```
> source module_load_mac.sh
> make ic.x
```

then it will generate the executable `ic.x`. To view the details, check `Makefile` in the same directory for the dependencies of `ic.x`. Beside `initial_conditions.f90`, it uses `../main/parameters.f90`,

`../main/variables.f90` and `../main/pencil_fft.f90`. In the `Makefile` it also declares some preprocessors. For example, "`OPTIONS=-DFFTFINE`" defines the macro "`FFTFINE`", which allows the global, distributed FFT work on a fine grid. "`OPTIONS+=-DPID`" defines the macro "`PID`", which enables the "particle identification (PID)" then each N-body particle has a "name". On the screen it displays the detailed steps of the compilation, and if there is no errors, type "`ls -lrt ic.x`" to check if a new executable is indeed generated.

To run the executable, in the same directory, type

```
> ./ic.x
```

and you will see the program running with some output on your screen. By the current default configuration the program is running on one computing *node* (in CAF language, *image*) with multiple processors. When CUBE is configured to run on multiple nodes (by setting `nn` > 1 in `main/parameters.f90`), it might use a different command to run the executable, for example, `mpirun`, and potentially environment variables are required to set accordingly, e.g. `FOR_COARRAY_NUM_IMAGES`. These heavily depend on systems so I do not enumerate here.

If `ic.x` is finished successfully, it automatically creates the output directories respectively for the data and code. So type

```
> ls -lrth ../../output/universe1/image1/

 total 118016
 -rw-r--r--  1 haoran  staff    48B 26 May 13:10 seed_1.bin
 -rw-r--r--  1 haoran  staff   8.0M 26 May 13:10 noise_1.bin
 -rw-r--r--  1 haoran  staff   8.0M 26 May 13:10 delta_L_1.bin
 -rw-r--r--  1 haoran  staff    64K 26 May 13:10 delta_L_proj_1.bin
 -rw-r--r--  1 haoran  staff   8.0M 26 May 13:11 99.000_phi1_1.bin
 -rw-r--r--  1 haoran  staff   8.0M 26 May 13:11 99.000_id_1.bin
 -rw-r--r--  1 haoran  staff    12M 26 May 13:11 99.000_xp_1.bin
 -rw-r--r--  1 haoran  staff    12M 26 May 13:11 99.000_vp_1.bin
 -rw-r--r--  1 haoran  staff   128K 26 May 13:11 99.000_np_1.bin
 -rw-r--r--  1 haoran  staff   384K 26 May 13:11 99.000_vc_1.bin
 -rw-r--r--  1 haoran  staff   224B 26 May 13:11 99.000_info_1.bin
```

to check the data are generated there, and type

```
> ls -lrth ../../output/universe1/code/
```

to review that the codes are automatically backed up together with the data. If the program is run on multiple images (usually on multiple computing nodes), there will be more directories named like `image2`, containing `delta_L_2.bin`, etc.

Here let's briefly view the data. The file names begin with "`99.000`" indicates that the redshift of the data is $z = 99$. The last number "`1`" in "`_1.bin`" shows the image number. Because CUBE uses a specially compressed format to save disk space (during computation, the memory space is also saved in this way) of the data, the checkpoint of one redshift contains several files. Their suffixes

are `*.bin` to indicate that they are in binary formats. The detailed descriptions of the data format are in later sections.

Some special files are generated by the initial condition generator. `seed_1.bin` contains the seeds (a series of integers) for random number generator. By default these seeds are generated according to the system clock and are unique. From the same seeds, and by using the same system, we could reproduce the same initial noise field. The noise field, `noise_1.bin`, is a series of single precision floating numbers (`real*4`) which follow a standard Gaussian distribution. They are further converted to the initial density fluctuation of the universe by a scale dependent *transfer function* $T(k)$, where k is the Fourier wavenumber scalar $k = \sqrt{k_x^2 + k_y^2 + k_z^2}$. $T(k)$ depends on the cosmological model. Here, we define *overdensity*

$$\delta \equiv \rho / \langle \rho \rangle - 1, \tag{1}$$

a dimensionless quantity. `delta_L_1.bin` stores the initial overdensity extrapolated to redshift 0, i.e.,

$$\delta_L \equiv \frac{D(0)}{D(z_i)} \delta(z = z_i), \tag{2}$$

where $D(z)$ is the *growth factor* as a function of redshift z. δ_L is considered as the underlying initial condition field and contains valuable information. In reality δ_L is unobservable and is expected to be estimated from various observations.

Usually we simulate the model either by a Eulerian or a Lagrangian method. N-body simulation adopts the latter method so we have to use N-body particles representing the underlying δ_L and its subsequent evolution. Therefore, from δ_L we generate particles and displace them using the Zel'dovich Approximation (ZA) [**?**]. ZA is usually referred to 1st-order Lagrangian perturbation theory (1LPT). nLPT ($n > 1$) is more accurate but computationally more expensive. When N-body simulations are fast, we can safely use 1LPT at a higher redshift to generate initial conditions, where the system is very linear and 1LPT is a good approximation. The resulting particles are generated by the CUBE format, located in the same folder. Besides, `99.000_info_1.bin` is a short binary file storing the basic parameters of the simulation. If more details of the above is required, please dig into `initial_conditions.f90`. However quick access of simulation information is available if you are familiar with the format. For example, type the following:

```
> od -t d4 -N 4 ../../output/universe1/image1/99.000_info_1.bin
0000000    2097152
0000004
```

It tells you the first 4 byte of the binary file `99.000_info_1.bin` in the format of 4-byte integers. The answer `2097152` actually shows that there are $128^3 = 2097152$ in this checkpoint.

3.3 Main code

Have initial condition generated, we go to `main` directory and compile the main N-body code.

```
> cd ../main/
> make
```

It would also be helpful to check `Makefile` in the `main` directory for according compiling dependencies. By default `PID` is defined so the particle-IDs are evolved associated with particles, and `HALOFIND` is defined so the run-time halo-finder is activated to find *dark matter halos* at redshifts listed in `z_halofind.txt`.

Then, type "`ls -lrt main.x`" to check if the executable is generated. Run the N-body simulation by

```
> ./main.x
```

We expect to see some outputs on the screen indicating the progress of the evolution of your simulated universe.

When the simulation is done, check the output directory, there should be new files "`0.000_*.bin`".

```
> ls ../../output/universe1/image1/0.000_*
0.000_halo_1.bin     0.000_id_1.bin     0.000_vc_1.bin
0.000_halo_pid_1.bin 0.000_info_1.bin   0.000_vp_1.bin
0.000_hid_1.bin      0.000_np_1.bin     0.000_xp_1.bin
```

These files are named `[redshift]_[content]_[image].bin` where `[redshift]` shows the redshift z of the checkpoint, `[content]` shows the information contained in the file, and `[image]` shows the Coarray-image number, starting from 1 (Fortran convention). The suffix `.bin` indicates that these files are written in binary format.

Alphabetically, `[content]`s have the following meanings. `[halo]`=halo catalog, `[halo_pid]`=particle list (by there particle ID, with `PID` enabled) in each halo, `[hid]`=halo number (starting form 1) for each particle (0 if the particle in isolated, belonging to none of the halos), `[id]`=particle id at the checkpoint, `[info]`=header file for the basic information of the checkpoint, `[np]`=coarse grid based particle number density field, `[vc]`=coarse grid based velocity field, `[vp]`=velocities of particles, and `[xp]`=positions of particles. Before going further, check if the particle number is conserved:

```
> od -t d4 -N 4 ../../output/universe1/image1/0.000_info_1.bin
0000000    2097152
0000004
```

4 Output format

Refer to §2.1, §2.2 and §2.3.1 of [**?**].

5 Additional Utilities

The above initial condition generator and the main code are the most basic parts of CUBE. In reality we need more functions to do science. Actually, some additional function are by default

turned on, including particle ID, run-time halo finder, particle-ID recorded in each halo, halo-ID recorded for each particle, etc. Below are some more to verify the simulation results visually and statistically.

5.1 Visualization

When the main code is executed successfully we go back to the `utilities` directory, compile and run the `cicpower.x` code:

```
> cd ../utilities/
> make cicpower.x
> ./cicpower.x
```

This interpolates all N-body particles onto the fine grid by cloud-in-cell (CIC) interpolation method, to produce 3D overdensity fields $\delta_{\rm CDM}$. The 3D density field is further averaged over the third direction to output a 2D column overdensity field (a slice). In `cicpower.f90` you can edit the second line of below

```
open(11,file=output_name("delta_c_proj"),status="replace",access="stream")
write(11) sum(rho_c(:,:,:),dim=3)/ng ! sum over the 3rd dimension and average
close(11)
```

to change the projection direction and the thickness (in unit of cells) of the slice. One quick check of the results is using the `od` command to dump the first 32 bytes (8 4-byte floating numbers) of these files:

```
> od -f -N 32 ../../output/universe1/image1/0.000_delta_c_1.bin
0000000    -9.862825e-01  -1.000000e+00  -1.000000e+00  -1.000000e+00
0000020    -1.000000e+00  -7.731423e-01  -2.080493e-01  -6.806935e-01
0000040
> od -f -N 32 ../../output/universe1/image1/0.000_delta_c_1.bin
0000000    -2.374918e-01  -2.071878e-01   6.052342e-01   5.310116e-01
0000020    -2.323187e-01  -4.816008e-01  -4.958048e-01  -5.290655e-01
0000040
```

It is reasonable to see that in `0.000_delta_c_1.bin` the minimum value is $\delta_{\rm CDM} = -1$.

It is helpful to visualize the 3D or 2D density fields. Here is a simple MATLAB script to plot the 2D slice:

```
ng=128; % set number of grids
file_id=fopen('0.000_delta_c_proj_1.bin'); % open file
delta_c=fread(file_id,[ng,ng],'real*4'); % open projection file
fclose(file_id); % close file
figure(1); imagesc(delta_c') % plot 2D array by intrinsic function "imagesc"
hold on
```

It is also helpful to read the halo catalog file and overplot the halos onto the density projection:

```
ninfo=42; % set number of information for each halo
file_id=fopen('0.000_halo_1.bin'); % open halo catalog file
nhalo_tot=fread(file_id,1,'integer*4')'; % read total number of halos
nhalo=fread(file_id,1,'integer*4')'; % read number of halos on image 1
halo_odc=fread(file_id,1,'real*4')'; % read overdensity threshold
haloinfo=fread(file_id,[ninfo,nhalo],'real*4'); % read all halo information
fclose(file_id); % close the file
plot(haloinfo(7,:),haloinfo(8,:),'r.') % plot halo positions on x-y plane
```

The above format of the halo catalog can be found in `../main/halofind.f90`. Note that, in CUBE, the length units are in fine cells. Multiply the length unit by $L/N_g =$ `box/ng` to convert to Mpc/h. These can be found in `../main/parameters.f90`.

So far we have some basic visualizations at redshift $z = 0$ of your simulated universe. Of course you can repeat the above for other redshifts similarly. An animation can be made if the frames are ordered by time sequence, i.e., by decreasing the redshift.

5.2 Power spectrum

From the 3D density fields generated above `cicpower.f90` computes the power spectrum of $\delta_{\rm CDM}$. This is a more statistical way of checking the simulation results. We will discuss about it in the next update.

5.3 Displacement field and clustering of matter

By using particle-IDs we can trace each particle to its initial positions, and even their Lagrangian coordinates $\boldsymbol{q}$ (at $z \to \infty$). This can tell us the displacement (shift, offset) of matter from initial Lagrangian space to final Eulerian space. This displacement field contains valuable cosmological information. We will discuss about it (`displacement.f90`) in next updates.

5.4 Angular momentum correlation

With the help of particle IDs, we can study the properties of objects (halos) in Lagrangian space and cross-correlate these information with the properties of current time. One interesting example is the angular momentum direction (spin). We will discuss about it (`ang_mom_corr.f90`) in next updates.