Draft version October 9, 2017
Preprint typeset using LaTeX style emulateapj v. 12/16/11

CUBE: A MEMORY-LITE COSMOLOGICAL N-BODY ALGORITHM

Hao-Ran Yu[1,2,3], Ue-Li Pen[2,4,5,6], Xin Wang[2]
[1]Tsung-Dao Lee Institute, Shanghai Jiao Tong University, Shanghai, 200240, China
[2]Canadian Institute for Theoretical Astrophysics, University of Toronto, Toronto, ON M5H 3H8 Canada
[3]Department of Physics and Astronomy, Shanghai Jiao Tong University, Shanghai, 200240, China
[4]Dunlap Institute for Astronomy and Astrophysics, University of Toronto, Toronto, ON M5S 3H4, Canada
[5]Canadian Institute for Advanced Research, CIFAR Program in Gravitation and Cosmology, Toronto, ON M5G 1Z8, Canada
[6]Perimeter Institute for Theoretical Physics, Waterloo, ON N2L 2Y5, Canada
Draft version October 9, 2017

Abstract

Cosmological large scale structure N-body simulations are computation-light, memory-heavy problems in supercomputing. Traditional N-body simulation algorithms use at least 24-byte memory per particle, of which six 4-byte single precision floating point numbers keep track phase space coordinates of each particle. Here we present the algorithm and accuracy of a new parallel, memory-lite, Particle- Mesh based N-body code, where each particle can occupy as low as 6-byte memory. This is accomplished by storing relative position and relative velocity of each particle, in the format of 1-byte-integer, respect to their averaged value of a mesh-grid. The remaining information is given by complimentary density and velocity fields, which are negligible in memory space, and proper ordering of particles, which gives no extra memory. Our numerical experiments show that this integer based N-body algorithm provides acceptable accuracy compared to traditional algorithm in cosmological simulations. This significant lowering of memory-to-computation ratio breaks the bottleneck of scaling up and speeding up large cosmological N-body simulations on multi-core and heterogenous computing systems.

1. INTRODUCTION

N-body simulation is a powerful tool to solve the highly nonlinear dynamic problems (Hockney & Eastwood 1988). It is widely used in cosmology to model the formation and evolution of the large scale structure (LSS). With the fast development of parallel supercomputers, we are able to simulate a system of more than a trillion (10^{12}) N-body particles. To date the largest N-body simulation in application is the "TianNu" simulation (Yu et al. 2017a; Emberson et al. 2017) run on the TianHe-2 supercomputer by cosmological simulation code `CUBEP3M` (Harnois-Déraps et al. 2013). It uses nearly 3×10^{12} particles to simulate the cold dark matter (CDM) and cosmic neutrino evolution through the cosmic age.

The N-body simulations use considerable amount of memory, because the phase space coordinates (x, y, z, v_x, v_y, v_z) of each N-body particle must be stored as, at least, six single precision floating point numbers (24 bytes). Contrarily, their computing workload can be alleviated by many algorithms, like Particle-Mesh [cite] and Tree [cite], scale as $o(N \log N)$ or even $o(N)$. On the other hand, modern supercomputer systems use multi cores, many integrated cores (MIC) and even densely parallelized GPUs, bringing orders of magnitude higher computing power, whereas these architectures usually have limited memory allocation. Thus, the computation-light but memory-heavy applications, compared to matrix multiplication and decomposition calculations, are less suitable for fully usage of the computing power of modern supercomputers. For example, although native and offload modes of `CUBEP3M` are able to run on the Intel Xeon-PHI MIC architectures, with the require ment of enough memory, TianNu simulation were done on TianHe-2 with only its CPUs – 73% of the total memory but only 13% of the total computing power.

We present a new N-body simulation code `CUBE`, using as low as 6 byte per particle (here after we use "bpp" referring to "byte per particle"). We show that it gives accurate results in cosmological LSS simulations. The method is presented in §2, and a comparison between this method and traditional method is shown in §3. Discussions and conclusions are in §4.

haoran@cita.utoronto.ca

2. METHOD

The most memory consuming part of a N-body simulation is usually the phase space coordinates of N-body particles – 24 bpp (4 single precision floating numbers) must be used to store each particles' 3D position and velocity vectors. `CUBEP3M`, an example of a memory-efficient parallel N-body code, can use as low as 40 bpp in sacrificing computing speed (Harnois-Déraps et al. 2013). This includes the phase coordinates (24 bpp) for particles in physical domain and buffered region, a linked list (4 bpp), and a global coarse mesh and local fine mesh. 4-byte real numbers are not necessarily adequate in representing the *global* coordinates in simulations. If the box size is many orders of magnitude larger than interactive distance between particles, especially in the field of re-simulation of dense subregions, double precision (8-byte) coordinates are needed to avoid truncation errors. Another solution is to record *relative* coordinates for both position and velocity. `CUBE` replaces the coordinates and linked list 24+4=28 bpp memory usage with an integer based storage, reduces the basic memory usage from 28 bpp down to 6 bpp, described as following 2.1 and 2.2. The algorithm is described in 2.3.

2.1. *Particle position storage*

We construct a uniform mesh throughout the space and each particle belongs to its parent cell of the mesh. Instead of storing global coordinates of each particle, we store its offset relative to its parent cell which contains the particle. We divide the cell, in each dimension d, evenly into $2^8 = 256$ bins, and use a 1-byte (8 bits) integer $\chi_d \in \{-128, -127, ..., 127\}$ to indicate which bin it locates in this dimension. The global locations of particles are given by cell-ordered format in memory space, and a complimentary number count of particle number in this mesh (density field) will give complete information of particle distribution in the mesh. Then the global coordinate in dth dimension x_d is given by $x_d = (n_c - 1) + (\chi_d + 128 + 1/2)/256$, where $n_c = 1, 2, ..., N_c$ is the index of the coarse grid. The mesh is chosen to be coarse enough such that the density field takes negligible memory. This coarse density field can be further compressed into 1-byte integer format, such that a 1-byte integer show the particle number in this coarse cell in range 0 to 255. In the densest cells (rarely happened) where there are ≥ 255 particles, we can just write 255, and write the actual number as a 4-byte integer in another file.

In a simulation with volume L^3 and N_c^3 coarse cells, particle positions are stored with a precision of $L/(256N_c)$. The force calculation (e.g. softening length) should be configured much finer than this resolution, discussed in later sections. On the other hand, particle position can also be stored as 2-byte (16 bits) integers to increase the resolution. In this case, each coarse cell is divided into $2^{16} = 65536$ bins and the position precision is $L/(65536N_c)$, precise enough compared to using 4-byte global coordinates, see later results. We denote this case "x2" and denote using 1-byte integers for positions "x1".

We collectively write the general position conversion formulae

$$\chi_d = \left[2^{8n_\chi}(x_d - [x_d])\right] - 2^{8n_\chi - 1}, \quad (1)$$

$$x_d = (n_c - 1) + 2^{-8n_\chi}\left(\chi_d + 2^{8n_\chi - 1} + 1/2\right), \quad (2)$$

where [ ] is the operator to take the integer part. $n_\chi \in \{1, 2\}$ is the number of bytes used for each integer, x_d and χ_d are floating and integer version of the coordinate. The velocity counterpart of them are $n_\nu = 1, 2$, v_d and ν_d. $n_c = 1, 2, ...$ is the coarse grid index, given by the ordering of particles and a integer based particle density field (see 2.3.1). The position resolution for n_χ-byte integer, "xn_χ", is $2^{-8n_\chi}L/N_c$.

As a $n_\chi = 1$, 1D ($d = 1$), 4-coarse-cell ($N_c = 4$) example, if

$$\chi_1 = (-128, 127, 0, 60),$$

particle number density

$$\rho_c^{\rm 1D} = (1, 0, 2, 1),$$

then in unit of coarse cells, the accurate positions of these four particles are

$$x_1 = (0.001953125, 2.998046875, 2.501953125, 3.736328125).$$

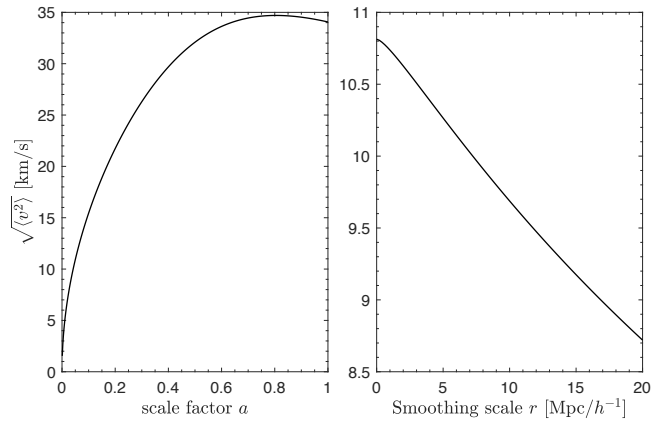

FIG. 1.— Xin, could you please check the units of both y-axis?

2.2. *Particle velocity storage*

Similarly, actual velocity in dth dimension v_d is decomposed into an averaged velocity field on the same coarse grid v_c and a residual Δv relative to this field:

$$v_d = v_c + \Delta v. \quad (3)$$

v_c is always recorded and kept updated, and occupies negligible memory. We then divide velocity space Δv into uneven bins, and use a n_ν-byte integer to indicate which Δv bin the particle is located.

The reason why we use uneven bins is that, slower particles are more abundant compared to faster ones, and one should better resolve slower particles tracing at least linear evolution. On the other hand, there could be extreme scattering particles (in case of particle-particle force), and we can safely ignore or less resolve those non-physical particles. One of the solution is that, if we know the probability distribution function (PDF) $f(\Delta v)$ we divide its cumulative distribution function (CDF) $F(\Delta v) \in (0, 1)$ into 2^{8n_ν} bins to determine the boundary of Δv bins, and particles should evenly distribute in the corresponding uneven Δv bins. Practically we find that either $f(v_d)$ or $f(\Delta v)$ is close to Gaussian, so we can use Gaussian CDF, or any convenient analytic functions which are close to Gaussian, to convert velocity between real numbers and integers.

The essential parameter of the velocity distribution is its variance. On non-linear scale, the velocity distribution function is non-Gaussian. However, to the first order approximation, we simply assume it as Gaussian and characterized by the variance

$$\sigma_v(a, k) = \frac{1}{3}(aHfD)^2 \int_0^k d^3q \frac{P_L(q)}{q^2}, \quad (4)$$

where $a(z)$ is the scale factor, $H(z)$ the Hubble parameter, D is the linear growth factor, $f = d\ln D/d\ln a$, and $P_L(k)$ is the linear power spectrum of density contrast at redshift zero. $\sigma_v(a, k)$ is a function of cosmic evolution a and a smoothing scale k, or r (see Figure 1). Δv is the velocity dispersion relative to the coarse grid, so we approximate its variance as

$$\sigma_\Delta^2(a) = \sigma_v^2(a, r_c) - \sigma_v^2(a, r_p), \quad (5)$$

where r_c is the scale of coarse grid, and r_p is the scale of average particle separation. In each dimension of 3D

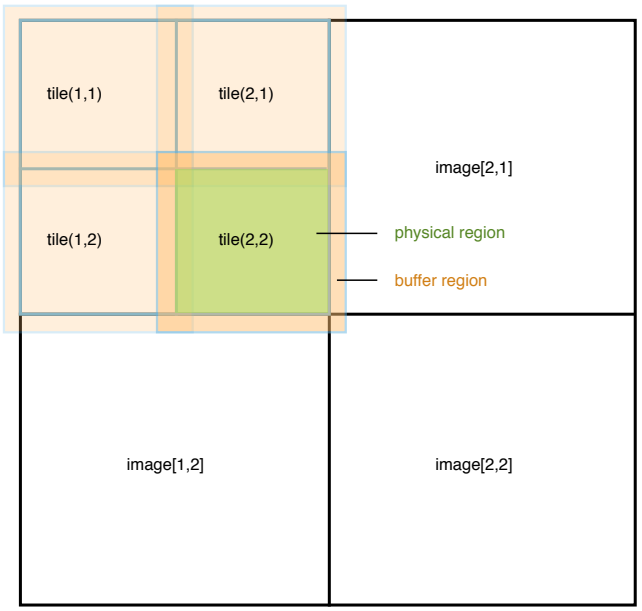

FIG. 2.— Spacial decomposition in CUBE in a 2D analogy. In this example, there are 2 images per dimension (nn = 2), and 2 tiles per image per dimension (nnt = 2). The orange boxes show the overlapped physical+buffer=extended tile regions, inside of which one physical regions is indicated with green.

velocity field, we use $\sigma_\Delta^2(a)/3$ according to the equipartition therorem. On different scales, we measure the statistics of v_d, v_c and Δv and find good agreement with the above model.

The simulation results are very insensitive if we manually tune the variance of the model σ_Δ within an order of magnitude. However, in $n_\nu = 1$ case, the method of using uneven bins gets much better results than simply using equal bins between minimum and maximum values $[\min(\Delta v), \max(\Delta v)]$. So, one can safely use a standard ΛCDM (cold dark matter with a cosmological constant as dark energy) for slightly different cosmological models, in equation (4). In CUBE, the velocity conversion takes the formula

$$\nu_d = \left\lfloor (2^{8n_\nu} - 1)\pi^{-1} \tan^{-1}\left((v_d - v_c)\sqrt{\pi/2\sigma_\Delta^2}\right)\right\rfloor, \quad (6)$$

$$v_d = v_c + \tan\left(\frac{\pi\nu_d}{2^{8n_\nu} - 1}\right)\sqrt{2\sigma_\Delta^2/\pi}, \quad (7)$$

where $\lfloor\ \rfloor$ is the operator to take the nearest integer. Tangent functions are convenient and compute very fast. Compared to error functions used in Gaussian case, they take the same variance at $\nu_d = 0$ but resolve high velocities relatively better. Note again that proper choice of conversion formulae and σ_Δ only optimizes the velocity space sampling, but does not affect the physics.

Initially, particles are generated by initial condition generator, at a higher redshift. The coarse grid density field v_c is also generated at this step by averaging all particles in the coarse cell. A global σ_Δ is calculated by equation (5), where linear approximation is hold. Then velocities are stored by equation (6). During the simulation, v_c is updated every time step, and a nonlinear σ_Δ is measured directly from the simulation, and can be simply used in the next time step, after scaled by the ratio of growth factors between two adjacent time steps. More details see section 2.3.3.

2.3. *Code overview*

CUBE uses a 2-level PM force calculation, same as CUBEP3M, However, in order to apply the integer based format to the N-body simulation, substantial structural changes need to be done. CUBE is written in Coarray Fortran, where Coarray features implement MPI communication between computation nodes/images[1]. The algorithm is described in this language.

2.3.1. *Spacial decomposition*

CUBE uses cubic decomposition structures. The global simulation volume is decomposed into $\texttt{nn}^3$ cubic subvolumes with nc coarse grids per side, and each of these are assigned to a coarray *image*. Inside of an image, the sub-volume is further decomposed into $\texttt{nnt}^3$ cubic *tiles*, with nt=nc/nnt coarse grids per side, which is a essential unit in the calculations. Each tile is surrounded by a *buffer* region which is ncb coarse cells thick. The buffer is designed for two reasons: (1) computing the fine mesh force, whose cut-off length nforce_cutoff $\leq$ ncb, and (2) collecting all possible particles travelling from a tile's buffer region to its center, *physical* region. Figure 2 shows the spacial decomposition in a 2-dimensional analogy, with nn = 2 and nnt = 2.

According to this spatial decomposition, and as discussed in the last two subsections, we declare $\{\rho_c, \chi_d, v_c, \nu_d\}$, or in variable expression {rho_c, xp, vfield, vp}, as follows:

integer(1) rho_c(nex, nex, nex, nnt, nnt, nnt)[nn, nn, *]
integer(n_χ) xp(npmax)[nn, nn, *]
real(4) vfield(nex, nex, nex, nnt, nnt, nnt)[nn, nn, *]
integer(n_ν) vp(npmax)[nn, nn, *]

where nex = nt + 2 × ncb covers the buffer region on both sides, nnt is the tile dimemsions, and nn is the image *codimensions*[2]. We denote actual number of particles in a given image nplocal, and npmax > nplocal is a value (discussed in Section 2.4) large enough to store particles. The particle position and velocity arrays xp and vp (equivalent to χ_d and ν_d in 2.1 and 2.2 respectively) are required to be sorted according to the same memory layout of rho_c, such that n_c and thus global positions of particles x_d can be obtained by equation (2). {rho_c, xp, vfield, vp} provides a complete information of positions and velocities of particles, and we call it a snapshot, or checkpoint.

An additional particle-ID (PID) array, PID, can also be declared to differentiate particle types (e.g. CDM and neutrino particles) or differentiate every particle, by using 1-byte, 4-byte or 8-byte integer per particle. If PID is turned on, the array PID is also included in the checkpoints, and the ordering of PID is same as xp and vp.

2.3.2. *Initial conditions*

[1] Images are the concept of computing nodes or MPI tasks in Coarray Fortran. We use this terminology in this paper.

[2] Coarray Fortran concept. Codimensions can do communications between images.

```
program Initial_Condition_Generator_for_CUBE
  Get random seed
  Generate Gaussian random field
  Multiply by power spectrum in Fourier space
  Get linear density field
  Solve Poisson equation
  Get displacement potential
  do (each tile) ! create particles per tile
    do (each fine grid in the extended tile region)
      calculate particle's coarse grid position
      update rho_c_new
    enddo
    do (each fine grid in the extended tile region)
      calculate particle's new accurate position
      calculate praticle's initial velocity
      calculate particle's index
      create xp_new(index) & vp_new(index)
      if (PID_flag) create PID(index)
    enddo
    do (each coarse grid)
      discard buffer information
    enddo
    write to disk
    sum up nplocal
  enddo
  sync all
  sum up npglobal
  write to headers of the checkpoint
end
```

FIG. 3.— Pseudocode for the initial condition generator.

The cosmological initial condition generator is compiled and run separately from the main N-body code. Here we briefly describe it for completeness.

The first step is the calculation of the potential field. At a sufficient high initial redshift z_i, we first generate a linear density fluctuation $\delta_L(\boldsymbol{q})$ on Lagrangian grid $\boldsymbol{q}$ by multiplying a Gaussian random field with the transfer function $T(k)$ in Fourier space, where $T(k)$ is given by the assumed cosmological model. Then we solve for the potential $\Phi(\boldsymbol{q}, z_i)$ of a curl-less displacement field $\boldsymbol{\Psi}(\boldsymbol{q})$ by Poisson equation $-\nabla^2\Phi = \delta_L$. At this step, global Pencil FFTs are applied on the fine mesh to transform the initial random Gaussian field to the displacement potential Φ. The initial condition generator is applied only once, and in principle, at some costs of speed, one can use only one scalar field on the fine mesh, and FFTs can be done in-place. If the number of particles equals the number of fine grids, then the memory consumption is dominated by the single precision fine mesh field, or 4 bpp.

The second step is to generate particles on Lagrangian coordinates $\boldsymbol{q}$ and displace them with Zel'dovich approximation (ZA) (Zel'dovich 1970), by locally differentiating Φ in real space. This step is done tile by tile. For each tile, we iterate twice over particles' Lagrangian coordinates $\boldsymbol{q}$ (usually at the center of every fine grid). The first iteration calculates particles' new, coarse grid positions by ZA and get a coarse density field on tile. The second iteration, ZA is applied again to calculate the accurate positions of particles and their velocities, and convert them into integer based ones by Equations (1,6) and place them in a certain order according to the coarse density field generated by the first iteration. Lastly, we discard those particles out of the physical region of the tile, and write only particles in physical region on to the disk. As we will see in Section 2.3.3, `update_xp` paragraph, it is very similar to the "drift" step in the main N-body code, where an "buffered" state of particles is

```
program CUBE
  call initialize
  call read_particles
  call buffer_density
  call buffer_xp
  call buffer_vp
  do
    call timestep
    call update_xp
    call buffer_density
    call buffer_xp
    call update_vp
    call buffer_vp
    if(checkpoint_step) then
       call update_xp
       call checkpoint
       if (final_step) exit
       call buffer_density
       call buffer_xp
       call buffer_vp
    endif
  enddo
  call finalize
end
```

FIG. 4.— Overall structure of `CUBE`. Sections of the code are grouped into Fortran subroutines, which are described in paragraphs of Section 2.3.3.

firstly created by two iterations of particles, and the particles in the buffer region are discarded to achieve a "disjoint" state. If PIDs are needed, they are also generated at the second iteration and are manipulated similarly as integer based velocities. Because particles are generated and saved to disk tile by tile, locally, after working on certain tile, the arrays for positions, velocities and PIDs (if any) are freed up and can be reused for the next tile. During this step, the only major memory usage is still dominated by the fine grid potential Φ.

After working on all tiles, we sum up the total particle number on this image, `nplocal`, and write it in the header of the checkpoint. Finally, {`rho_c`, `xp`, `vfield`, `vp`, `PID`(optional)} is rewritten to the disk as the checkpoint of $z = z_i$. We summarize the above steps into a pseudocode in Figure 3, where the "`do`" loop to create particles per tile is very similar to the following `update_xp` subroutine in the main N-body code. The corresponding arrays are also discussed below.

2.3.3. *Algorithm*

Figure 4 shows the overall structure of the code and these subroutines are described in following paragraphs.

`initialize` and `read_particles` – The subroutine `initialize` creates necessary FFT plans and read in configuration files telling the program at which redshifts we need to do checkpoints, halofinds, or stop the simulation. Force kernels `kern_c`, `kern_f` are also created or read in here. In `read_particles`, for each image, we read in all particles in *physical* regions of every tile (indicated in Figure 2), i.e., {`rho_c`, `xp`, `vfield`, `vp`, `PID`(if any)}. These are obtained by the initial condition generator (Section 2.3.2). Because physical regions of tiles are *complete* and *disjoint* in space, particles at this stage are also complete and disjoint. We call it "disjoint state". At this stage, `rho_c`'s values in buffer regions are 0's, and the elements of `xp` and `vp` beyond physical number – `xp(nplocal+1:)` and `vp(nplocal+1:)` can be arbitrary and are not used.

`buffer_density`, `buffer_x` and `buffer_v` – In order to use the integer based format, `xp` and `vp` must al-

ways be ordered, and their number density field `rho_c` must always be present. In updating arrays of `xp` and `vp` (not simultaneously) of a local tile, the buffer region of the tile is also needed. First, buffer regions of `rho_c` is synchronized between tiles and images by subroutine `buffer_density`. Then, by subroutines `buffer_x` and `buffer_v` respectively, `xp` and `vp` are updated to contain common, buffered particles, in an order according to the new, buffered `rho_c`. If there are PIDs used, then `PID` is also updated in subroutine `update_v`. We call this stage "buffered state". After `particle_initialize` is done by all images (synchronized by a "`sync all`", which is equivalent to a `mpi_barrier`), these three subroutines are called and particles are converted from disjoint state to buffered state.

`timestep` – We operate a Runge-Kutta 2 method for time integration. i.e., we update position (D=drift) and velocity (K=kick) every half time step. For n time steps, the operation would be $(DKKD)^n$ which is 2nd order accurate. The actual simulation applies varied time steps. In each iteration of the main loop, we firstly call `timestep`, where a increment of time `dt` is controlled by particles' maximum velocities, accelerations, cosmic expansion and any other desired conditions.

`update_xp` – According to `dt`, subroutine `update_xp` updates the positions of particles in a "*gather*" algorithm (in contrast, `CUBEP3M` uses "scatter" algorithm) tile by tile. For each particle, `xp` and `vp` are converted to x_d and v_d by Equations(2,7). Because each tile is in the buffered state with buffer depth `ncb`, we are able to collect all possible particles whose $v_d \times$ `dt` $<$ `ncb` from physical+buffer region to its physical region.

In order to keep particles ordered, for each tile, we first do $x_d = x_d + v_d \times$ `dt` on all particles to obtain an updated density and velocity field on the tile, `rho_c_new` and `vfield_new`. Then, this calculation in done on the same tile again[3] to generate a new, local particle list `xp_new` and `vp_new` (also `PID_new` if needed) on the tile by Equations(1,6). Here, the ordering of particles depends on `rho_c_new`, and `vfield_new` is used as v_c in Equation(6). Then, another iteration is done on this tile to discard buffer regions of {`rho_c_new`, `xp_new`, `vfield_new`, `vp_new`, `PID_new`(if any)}, converting buffer state to disjoint state, and it replaces corresponding part of {`rho_c`, `xp`, `vfield`, `vp`, `PID`(if any)}. When all tiles are iterated, the entire image is in disjoint state, and `nplocal` is updated. Then all images is synchronized by `sync all` and we sum up `nplocal` over images to check if total number of particles `npglobal` is conserved. These steps are summarized in Figure 5.

`update_vp` – The particle-mesh (PM) or particle-particle particle-mesh (P^3M) algorithm is applied in this subroutine to update particles's velocities. After `update_xp`, or drift D, we call `buffer_density` and `buffer_xp` in order that particle positions are in buffered state, based on which, we update velocities (kick K) of particles in physical region of each tile. `CUBE` uses a 2-level particle mesh scheme (Harnois-Déraps et al. 2013). Local fine forces have a force cutoff `nforce_cutoff` $\leq$ `ncb`, i.e., $F_{\rm fine}(r >$ `ncb`$) = 0$. So, if we apply a fine-grid particle-mesh on an extended tile with buffered-depth `ncb` (see Figure 2 and regard it periodic), the force on physical regions does not depend on the false periodic boundary assumption, and the resulting fine force $F_{\rm fine}$ and velocity update on physical region is correct.

The compensating coarse grid force $F_{\rm coarse}$ is globally computed by using a coarser (usually by factor of 4) mesh by dimensional splitting – a distributed-memory cubic decomposed 3D coarse density field is interpolated by particles, and we Fourier transform data in consecutive three dimensions with global transposition in between (known as the pencil decomposition). After the multiplication of force kernels, the inverse transform takes place to get the cubic distributed coarse force field $F_{\rm coarse}$, upon which velocities are updated again.

An optional particle-particle (PP) force $F_{\rm pp}$ can be called to increase the force resolution and the velocities are updated again. The collective operations of velocity update is Equation(7), $v_d = v_d + F_{\rm total}$ and Equation(6).

The maximum accelerations $\max(\dot{v}_{\rm fine})$, $\max(\dot{v}_{\rm coarse})$, $\max(\dot{v}_{\rm pp})$ and maximum of velocities $\max(v_d)$ are collected controlling `dt` for the `timestep` in the next iteration. We also update σ^2_Δ according to the new $v_d - v_c$.

```
subroutine update_xp
  do (each tile)
    do (each coarse grid)
      do (each particle)
        calculate particle's new coarse grid position
        update rho_c_new
        update vfield_new
      enddo
    enddo
    do (each coarse grid)
      do (each particle)
        calculate particle's new accurate position
        calculate particle's index
        update xp_new(index) & vp_new(index)
      enddo
    enddo
    do (each coarse grid)
      discard buffer information
      replace xp and vp for this tile
    enddo
  enddo
  sync all
  update velocity dispersion
  check nplocal & npglobal
end
```

FIG. 5.— Pseudocode for subroutine `update_xp`.

```
subroutine update_vp
  do (each tile)
    calculate fine mesh density
    calculate fine mesh force: local FFT
    update fine mesh velocity
    update maximum acceleration
    if (PP force) then
      PP calculation & velocity update
      update maximum acceleration
    endif
  enddo
  sync all
  calculate coarse mesh density
  calculate coarse mesh force: global pencil FFT
  update coarse mesh velocity
  update maximum acceleration
  update maximum velocity
end
```

FIG. 6.— Pseudocode for subroutine `update_vp`.

[3] This repetition scales as $o(N)$ and is computational inexpensive.

These steps are summarized in Figure 6.

By far, vp in physical regions are correctly updated. Remember that particle locations in the buffer regions has updated before PM and remained unchanged. So, in the main code we simply call buffer_v again to bring vp and PID (if any) into buffered state, such that the update_x in the next iteration will be done correctly.

checkpoint – If a desired redshift is reached, we execute the last drift step in the $(DKKD)^n$ operation by update_xp, and call checkpoint to save the disjoint state of {xp, vp, rho_c, vfield} on disk. Related operations, like run-time halo finder, projections are also done at this point. If final desired redshift is reached, final_step let us exit the main loop. These corresponding logical variables are controlled in timestep.

finalize – Finally, in finalize subroutine we destroy all the FFT plans and finish up any timing or statistics taken in the simulation.

2.4. *Memory layout*

Here we list the memory-consuming arrays and how they scale with different configurations of the simulation. We classify them into (1) arrays of particles, (2) coarse mesh arrays and (3) fine mesh arrays.

2.4.1. *Arrays of particles*

The arrays of particles contains xvp(6,npmax) and a temporary xvp_new(6,nptile)[4]. npmax is the number of particles xvp can store:

$$\texttt{npmax} = \langle\texttt{nplocal}\rangle \left(1 + \frac{2\times\texttt{ncb}}{\texttt{nt}}\right)^3 (1+\epsilon_{\rm image}), \quad (8)$$

where $\langle$nplocal$\rangle$ is the average number of particles per image. The second term above let us store particles in buffer regions, and the third term $1+\epsilon_{\rm image}$ takes into account the inhomogeneity of nplocal on different images. When each image models smaller physical scales, $\epsilon_{\rm image}$ should be set larger.

xvp_new stores temporary particles only on tiles:

$$\texttt{nptile} = \langle\texttt{nplocal}\rangle \left(\frac{1}{\texttt{nnt}}\right)^3 (1+\epsilon_{\rm tile}), \quad (9)$$

where, recall that nnt is the number of tiles per image per dimension, and similarly $\epsilon_{\rm tile}$ controls the inhomogeneity on scale of tiles. Larger nnt causes more inhomogeneity on smaller tiles, and $\epsilon_{\rm tile}$ can be much larger than $\epsilon_{\rm image}$, however the term $\texttt{nnt}^{-3}$ decreases much faster. So, the majority memory is occupied by xvp.

2.4.2. *Coarse mesh arrays*

On coarse mesh, rho_c, vfield and force kernel kern_c should always be kept. They are usually configured to be 4 times coarser than fine grids and particle number density. In this case, each coarse grid, 3D vfield takes only 12 bytes, and 64 particles will at least take 384 bytes ($n_\chi = n_\nu = 1$ case). Similarly, pencil-FFT arrays, coarse force arrays etc. are memory-light and/or temporary.

[4] We use "xvp" to represent {xp, vp}, and use "xvp_new" to represent {xp_new, vp_new}.

TABLE 1
Memory layout for a certain configuration

		Memory usage		
Type	Array	/GB	/bpp	Percentage
Particles	xvp	29.9	8.24	83.8%
	xvp_new	<u>3.16</u>	<u>0.872</u>	<u>8.87%</u>
	PID	0	0	0%
	Subtotal	33.0	9.12	92.7%
Coarse mesh	rho_c	0.296	0.0818	0.831%
	vfield	0.889	0.245	2.49%
	kern_c	0.340	0.0939	0.954%
	force_c	(0.690)	(0.190)	(1.94%)
	rho_c_new	0.0140	0.00388	0.0394%
	Pencil-FFT	(0.454)	(0.125)	(1.27%)
	Subtotal	1.55	0.429	4.36%
Fine mesh	kern_f	1.06	0.292	2.97%
	force_f	(1.63)	(0.450)	(4.57%)
	Fine-FFT	(1.41)	(0.389)	(3.96%)
	Subtotal	1.06	0.292	2.97 %
Total		35.6	9.84	100%
Optimal limit		21.7	6	61.0%

2.4.3. *Fine mesh arrays*

On the local fine mesh, only a force kernel array needs to be kept. The fine mesh density arrays, force field arrays are temporary. Since xvp_new, coarse force arrays, pencil-FFT arrays are also temporary and are not used in any calculation simultaneously, they can be overlapped in memory by using equivalent statements.

2.4.4. *Compare with traditional algorithms*

To illustrate the improvement of memory usage, we give an example of TianNu simulation (Yu et al. 2017a) on the Tianhe-2 supercomputer, which used a traditional CUBEP3M code. TianNu's particles number is limited by memory per computing node – per computing node, an average of 576^3 neutrino particles and 288^3 CDM particles are used, and consumes about 40 GB of memory[5], or about 186 bpp. A memory efficient case of CUBEP3M by using large physical scales and at costs of speed, still uses about 40 bpp.

By using CUBE, with parameters $n_\chi = n_\nu = 1$, nc = 384, nnt = 3, ncb = 6, $\epsilon_{\rm image} = 5\%$, $\epsilon_{\rm tile} = 200\%$ and $\langle$nplocal$\rangle = 1536^3$, we use about 35.6 GB of memory, corresponding to 9.84 bpp. This can be done on most of the supercomputers, even modern laptops.

Table 1 shows the memory consumption for this test simulation. The memory-consuming arrays are listed and classified into the three types above mentioned, and their memory usages are in unit of GB (10^9 bytes), byte per particle (bpp), and their percentage to the total memory usage. The parenthesized numbers show overlapped memory, which is saved by equivalencing them with the underscored numbers. There are other, unlisted variables which are memory-light, and can also be equivalenced with listed variables. For this 1536^3 per image simulation, using $\texttt{nnt}^3 = 27$ tiles per image optimizes the memory usage, because it well controls the balance between the second terms in Equations (8,9). A great amount of temporary memory in coarse and fine mesh force calculations is equivalenced to xvp_new.

[5] Additional memory is used for OpenMP parallelization and for particle IDs to differentiate different particle spices.

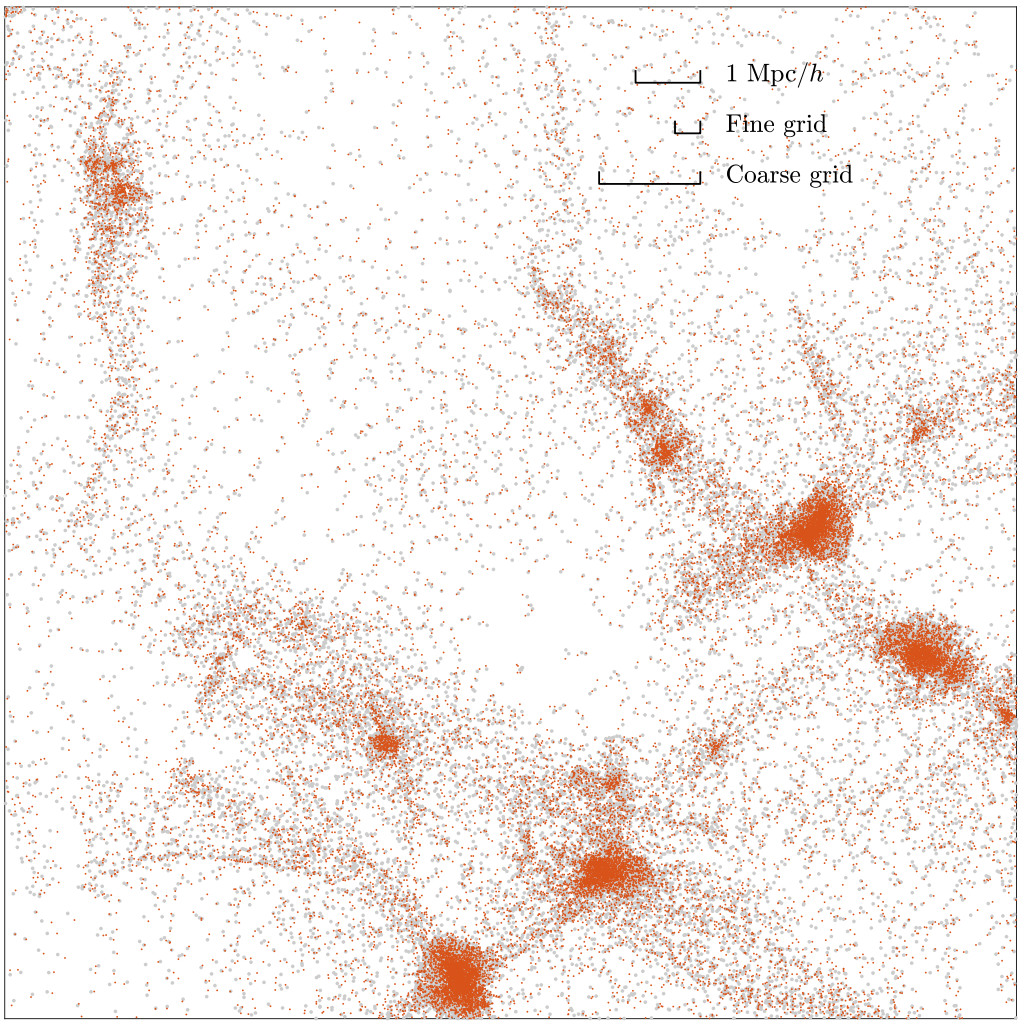

FIG. 7.— Offset in the particle distribution induced by integer-1 based algorithm (x1v1). In S512's subregion of volume $15.625^2 \times 3.90625$ $(\mathrm{Mpc}/h)^3$, particles from `CUBEP3M` (bigger grey dots in the background) and `CUBE`-x1v1 (smaller red dots) are projected onto the plane of $(15.625\ \mathrm{Mpc}/h)^2$. The comparing rules show 1 Mpc/h, fine grid and coarse grid respectively, and position resolution of x1v1 is 1/64 of a fine grid.

In the bottom of Table 1 we stress that the optimal memory usage is 6 bpp, or 21.7 GB, 61% of the actual memory usage in this test simulation. The most prominent departure from this limit is that `xvp` already uses 8.24 bpp, and all other variables occupy only additional 8%. Of coarse, on modern super computers, the memory per computing node is usually much larger, by scaling up the number of particles per node, the buffer ratio `ncb`/`nt` will be lowered and we can approach closer to the 6 bpp limit.

3. ACCURACY

We run a group of simulations to test the accuracy of `CUBE`. We use same seeds to generate same Gaussian random fields in the initial condition generators of `CUBEP3M` and `CUBE`, and then they produce initial conditions of their own format respectively. Then the main N-body codes run their own initial conditions to redshift $z = 0$.

First, by using different configurations – different number of computing nodes, box sizes, particle resolutions, different number of tiles per node/image etc., we find that by using 2-byte integers for both positions and velocities ($n_\chi = n_\nu = 2$, or x2v2) `CUBE` gives exact results compared to `CUBEP3M`. So, if memory usage is not a problem, one can always double the memory usage to get exact results, and the optimal memory limit of this case is 12 bpp, still much lower than traditional methods. Next, we focus on the accuracy of the other three cases – x1v2, x2v1 and x1v1.

We list the names and configurations of simulations used in Table 2, where N_{node}, L, z_i, z_{ckpt}, N_p and m_p are respectively the number of computing nodes used, length of the side of the box, initial redshift, redshift where we do checkpoint, total number of particles and particle mass. These configurations are run by `CUBEP3M`, x2v2, x1v2, x2v1 and x1v1 versions of `CUBE`. Using different number of tiles per image gives exact same results. We also list the configurations for TianNu and TianZero (Yu et al. 2017a; Emberson et al. 2017) simulations as a reference.

3.1. *Displacement of particles*

In S512, we zoom into a small region of $15.625^2 \times 3.90625$ $(\mathrm{Mpc}/h)^3$ and compare the particle distribution between `CUBEP3M` and `CUBE`-x1v1 in Figure 7. For clarity, `CUBEP3M` particles are marked with bigger, grey dots,

TABLE 2
SIMULATION CONFIGURATIONS

	Configurations					
Name	$N_{\rm node}$	$L/({\rm Mpc}/h)$	z_i	$z_{\rm ckpt}$	N_p	$m_p/M_\odot$
S512	8	200	49	0	512^3	7.5×10^9
S256	1	80	49	0	256^3	3.8×10^9
S2048S	64	400	49	1,0	2048^3	9.4×10^8
S2048L	64	1200	49	1,0	2048^3	2.5×10^{10}
TianNu	13824	1200	100	0.01	6912^3	6.9×10^8
			5	0.01	13824^3	3.2×10^5
TianZero	13824	1200	100	0.01	6912^3	7.0×10^8

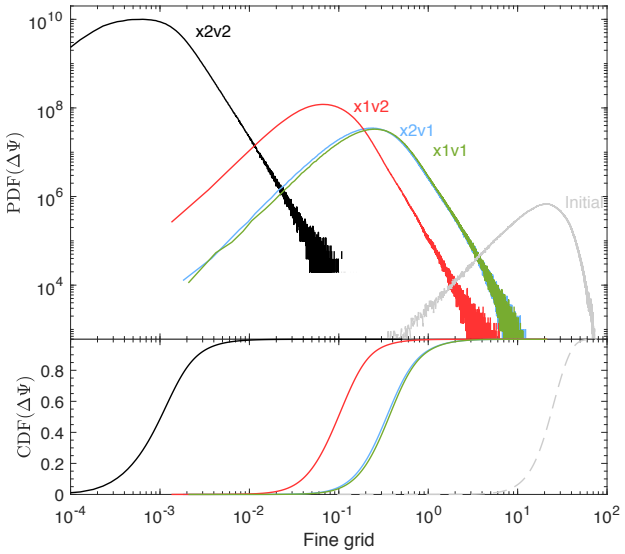

FIG. 8.— Statistics of the error in particle displacement $\Delta\boldsymbol{\Psi}$ induced by integer based algorithms in CUBE. The PDF and CDF of $|\Delta\boldsymbol{\Psi}|$ from S256 are shown in the upper and lower axes, as functions of fine grids. Black, red, blue and green correspond to x2v2, x1v2, x2v1 and x1v1 respectively. The grey lines, marked with "initial", shows the distribution of actual displacement of particles in CUBEP3M, $|\boldsymbol{\Psi}_0|$, which is orders of magnitudes larger than $|\Delta\boldsymbol{\Psi}|$.

whereas smaller red dots are CUBE-x1v1 particles overplotted onto them. One can see the difference between them. The 1 Mpc/h, fine grid and coarse grid scales are shown in the figure. The position resolution of particles in CUBE-x1v1 is 1/256 of a coarse grid, or 1/64 of a fine grid.

To quantify the offset in the final particle distributions, we use PIDs to track the displacement $\boldsymbol{\Psi}(\boldsymbol{q})\equiv\boldsymbol{x}-\boldsymbol{q}$ of every particle (Yu et al. 2017b), where $\boldsymbol{x}$ and $\boldsymbol{q}$ are Eulerian and Lagrangian coordinates of the particle. Then we calculate the absolute value of the offset vector

$$\Delta\Psi\equiv|\boldsymbol{\Psi}_i-\boldsymbol{\Psi}_0|. \tag{10}$$

Here, $\boldsymbol{\Psi}_0$ stands for CUBEP3M and subscript $_i$ can stand for x2v2, x1v2, x2v1 or x1v1. The probability distribution functions (PDFs) and cumulative distribution functions (CDFs) of $\Delta\boldsymbol{\Psi}$ in S256 are shown in Figure 8. Results from x2v2, x1v2, x2v1 or x1v1 are in black, red, blue and green respectively. The results from absolute displacement of particles (by replacing $\boldsymbol{\Psi}_i$ with $\boldsymbol{q}$ in Equation (10)) are shown in grey for comparison.

For x2v2, almost all particles are accurate up to 1/100 of a fine grid, and the worst particle is $\sim$ 1/10 of a fine grid away from its counterpart in CUBEP3M. The difference is caused by truncation errors and is negligible in physical and cosmological applications. The accuracy of x1v2 is between x2v2 and x1v1, and x2v1 gives only minor improvement from x1v1. We also run a simulation with same number of particles but with $L=600$ Mpc/h ($m_p=1.2\times10^{12}M_\odot$), and find that the accuracy of x1v2 is in turn between x2v1 and x1v1. We interpret that, in this latter case, particles have lower mass resolution, so they move slower and need higher position resolution but need lower velocity resolution, then x2v1 outperforms x1v2.

3.2. *Power spectrum*

In S2048S and S2048L are two simulations with 2048^3 particles on small ($L=400$ Mpc/h) and large ($L=1200$ Mpc/h) box sizes. We compare their accuracy by their power spectra and their cross correlations with CUBEP3M at two checkpoints $z=1$ and $z=0$. The physical scale of S2048S is designed such that the particle mass resolution m_p is comparable to TianNu and TianZero simulations (their parameters are also listed in Table 2) to study halo properties. On the other hand, S2048L focuses on larger structures, on which scale one can study weak gravitational lensing, reconstruction of baryonic acoustic oscillation (BAO) (Wang et al. 2017) etc.

For each simulation the particles are firstly cloud-in-cell (CIC) interpolated onto the fine mesh grid of simulation, and from the density field ρ we define the density contrast $\delta\equiv\rho/\langle\rho\rangle-1$. More generally we define the cross power spectrum $P_{\alpha\beta}(k)$ between two fields δ_α and δ_β ($\delta_\alpha=\delta_\beta$ for auto power spectrum) in Fourier space as

$$\left\langle\delta^\dagger_\alpha(\boldsymbol{k})\delta_\beta(\boldsymbol{k}')\right\rangle=(2\pi)^3P_{\alpha\beta}(k)\boldsymbol{\delta}_{\rm 3D}(\boldsymbol{k}-\boldsymbol{k}'), \tag{11}$$

where $\boldsymbol{\delta}_{\rm 3D}$ is the three-dimensional Dirac delta function. In cosmology we usually consider the dimensionless power spectrum $\Delta^2_{\alpha\beta}(k)\equiv k^3P_{\alpha\beta}(k)/(2\pi^2)$. The cross correlation coefficient is defined as

$$\xi(k)\equiv P_{\alpha\beta}/\sqrt{P_{\alpha\alpha}P_{\beta\beta}}. \tag{12}$$

We study the fractional power spectrum deviations $\Delta^2(k)/\Delta^2_0(k)-1$ (where $\Delta^2_0(k)$ denotes the CUBEP3M power spectrum) and the cross correlation coefficient functions $\xi(k)$ between CUBE and CUBEP3M, and show the results in Figure 9. For the four panels, the left shows S2048S and right shows S2048L; the upper shows $z=1$ and lower shows $z=0$. We label $k=0.2\ k_{\rm Nyquist}$ as dashed lines, where $k_{\rm Nyquist}$ is the fine mesh scale, and the average particle separation scale. On the scale $k=0.2\ k_{\rm Nyquist}$, the power spectrum is suppressed by $\sim$ 20% compared to higher resolution simulations by Poisson noise of particles and we do not consider $k>0.2\ k_{\rm Nyquist}$.

The upper parts of the four panels show the fractional power spectrum deviations. Because x2v2 gives exact results as CUBEP3M, $\Delta^2_{\rm x2v2}(k)/\Delta^2_0(k)-1=0$ for both simulations and redshifts, shown by the four black straight lines. Similar to the results form Figure 8, x1v2 (red) curves give better results than x2v1 (blue) and x1v1 (green), and x2v1 and x1v1 give nearly same results. Another trend is that higher redshifts and/or larger box sizes gives much better results. The worst case is S2048S,

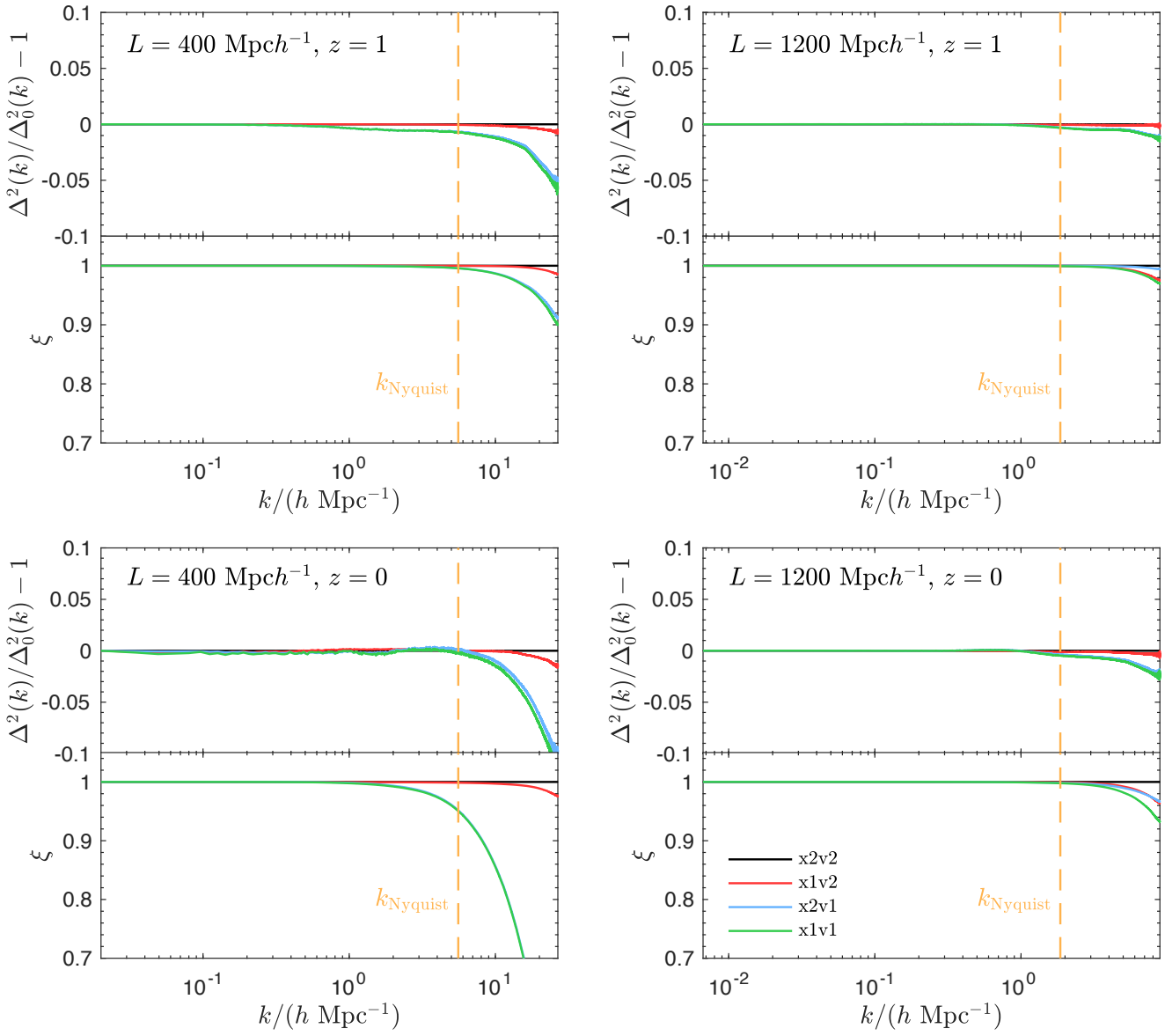

FIG. 9.— Fractional deviations in the power spectrum $\Delta^2(k)/\Delta_0^2(k) - 1$ (upper axes) and decorrelation ξ (lower axes) induced by the integer-based algorithms in CUBE. The four panels show the results of two redshifts in S2048S and S2048L: $\{z = 1, z = 0\} \times \{L = 400{\rm Mpc}/h, L = 1200{\rm Mpc}/h\}$. The four solid lines show the results from {x2v2,x1v2,x2v1,x1v1}, whose colors are shown in the legends in the last panel. The orange dashed lines show the scale $k = 0.2\ k_{\rm Nyquist}$, and smaller scales are beyond our interest.

$z = 0$ case – There is a $< 2\%$ deviation in the power spectrum.

The lower parts of the four panels show the cross correlation coefficient function between CUBE and CUBEP3M. Again, x2v2 performs perfect in all the four cases. For x1v2, $\xi(k) > 99.5\%$ for all cases within our scale of interest. For x1v2 and x1v1, $\xi(k)$ are very close, and they perform worst in S2048S, $z = 0$ case – on scale $k = 0.2\ k_{\rm Nyquist}$, the correlation drops to 90%.

4. DISCUSSION AND CONCLUSION

We present an parallel integer-based N-body algorithm. This open-source code CUBE, has recently been used in many studies of LSS, e.g. Yu et al. (2017b); Wang et al. (2017); Pan et al. (2017). It needs very low memory usage, approaching 6 byte per particle (6 bpp). The accuracy of this 6pp configuration is acceptable on most scales of our interest, e.g. BAO. The accuracy is also adjustable in that we can choose 1-byte/2-byte integers separately for positions and velocities of particles. In the case of using 2-byte integers for both positions and velocities ("x2v2", and memory can be 12 bpp), the algorithm gives exact results given by traditional N-body algorithms.

We analyze the deviation of power spectrum and decorrelation caused by integer-1 based algorithms in two simulations S2048S and S2048L. They have comparable number of particles per computing as the TianNu simulation. S2048S focuses on mass resolution – comparable to TianNu and TianZero simulations, whereas S2048L focuses on simulation volume and have higher particle mass. In the power spectra and cross correlations, S2048L performs better, and we conclude that in

high-m_p cases, we can safely use x1v1 in cosmological simulations.

Another benefit for this algorithm is LSS simulations with neutrinos, although the neutrino modules of `CUBE` is in development. Neutrinos have a high velocity dispersion and move much faster than CDM, and their small scale errors are dominated by their Poisson noise. We expect that, compared to CDM, x1v1 gives less power spectrum deviation to neutrinos, as they behave more Gaussian and less clustered. LSS-neutrino simulations, like TianNu, can contain much more (for TianNu, 8 times more) neutrino N-body particles than CDM, which dominates the memory. For a TianNu-like simulation, one can safely use x1v2 or x2v2 for CDM and x1v1 for neutrinos, and the memory usage can approach 6 bpp. This allows much more particles in the simulation and can lower the Poisson noise of neutrinos prominently. A 1-byte PID per particle increases minor memory usage and can differentiate 8 kinds of particles, or we can store different particles in different arrays without using PID.

We did not include PP force in `CUBE` and `CUBEP3M` in this paper. If one wants to focus on smaller scales, like halo masses and profiles, an extended PP forces, which act up to adjacent fine cells, should take into account. The memory consumption for PP force is only local and is negligible compared to particles. In these cases, even x2v2 is used, a 12 bpp memory usage is still much lower than traditional N-body algorithms.

Traditional N-body codes consumes considerable memory while doing a relatively light computations. `CUBE` prominently reduces the memory usage in N-body simulations and approaches toward a better balance between memory and computation. The next steps are optimization of the code and adapting it for various kinds of heterogeneous computing systems, e.g. MIC and GPUs. Optimizing the velocity storage may further improve the accuracy of x1v1, and whose effects on neutrino-LSS simulation is going to be discovered.

REFERENCES

Emberson, J. D. et al. 2017, Research in Astronomy and Astrophysics, 17, 085, 1611.01545

Harnois-Déraps, J., Pen, U.-L., Iliev, I. T., Merz, H., Emberson, J. D., & Desjacques, V. 2013, MNRAS, 436, 540, 1208.5098

Hockney, R. W., & Eastwood, J. W. 1988, Computer simulation using particles

Pan, Q., Pen, U.-L., Inman, D., & Yu, H.-R. 2017, MNRAS, 469, 1968, 1611.10013

Wang, X., Yu, H.-R., Zhu, H.-M., Yu, Y., Pan, Q., & Pen, U.-L. 2017, ApJ, 841, L29, 1703.09742

Yu, H.-R. et al. 2017a, Nature Astronomy, 1, 0143, 1609.08968

Yu, H.-R., Pen, U.-L., & Zhu, H.-M. 2017b, Phys. Rev. D, 95, 043501, 1610.07112

Zel'dovich, Y. B. 1970, A&A, 5, 84

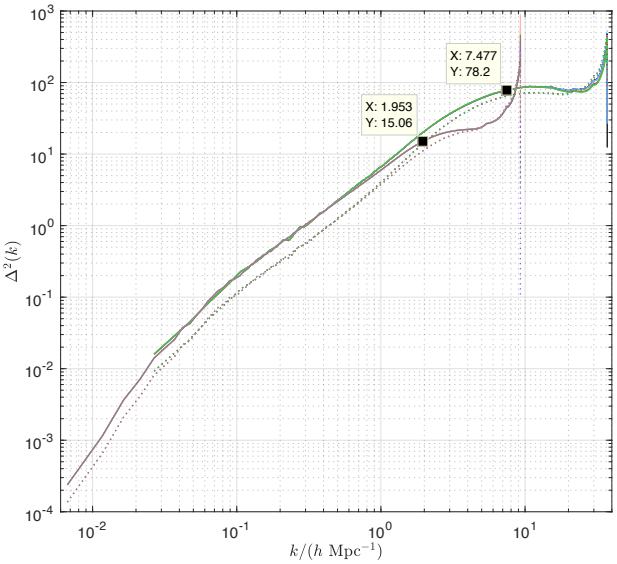

FIG. 10.— We don't need this figure in submission. However we need to compare the results with nonlinear power spectrum by HALOFIT. The solid lines show the power spectra of S2048L and S2048XS ($L = 300$ Mpc/h) at $z = 0$, and dotted lines show the $z = 1$ counterparts. For $z = 0$, the two data tips are at $k = 0.2\ k_{\mathrm{Nyquist}}$, where the power spectrum is suppressed by more than 20% compared to higher resolution simulations.