

CUBE: AN INFORMATION-OPTIMIZED PARALLEL COSMOLOGICAL N -BODY ALGORITHM

HAO-RAN YU^{1,2,3}, UE-LI PEN^{2,4,5,6}, XIN WANG²

¹Tsung-Dao Lee Institute, Shanghai Jiao Tong University, Shanghai, 200240, China

²Canadian Institute for Theoretical Astrophysics, University of Toronto, Toronto, ON M5H 3H8 Canada

³Department of Physics and Astronomy, Shanghai Jiao Tong University, Shanghai, 200240, China

⁴Dunlap Institute for Astronomy and Astrophysics, University of Toronto, Toronto, ON M5S 3H4, Canada

⁵Canadian Institute for Advanced Research, CIFAR Program in Gravitation and Cosmology, Toronto, ON M5G 1Z8, Canada

⁶Perimeter Institute for Theoretical Physics, Waterloo, ON N2L 2Y5, Canada

Draft version March 16, 2018

Abstract

Cosmological large scale structure N -body simulations are computation-light, memory-heavy problems in supercomputing. The considerable amount of memory is usually dominated by an inefficient way of storing more than sufficient phase space information of particles. We present a new parallel, information-optimized, particle-mesh based N -body code CUBE, in which information- and memory-efficiency is increased by nearly an order of magnitude. This is accomplished by storing particle's relative phase space coordinates instead of global values, and in the format of fixed point as light as 1-byte. The remaining information is given by complimentary density and velocity fields (negligible in memory space) and proper ordering of particles (no extra memory). Our numerical experiments show that this information-optimized N -body algorithm provides accurate results within the error of the particle-mesh algorithm. 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

The N -body simulation, a dynamical simulation of a group of particles, is a powerful tool in physics and astronomy (Hockney & Eastwood 1988). It is widely used in cosmology to model the large scale structure (LSS) of the Universe (Davis et al. 1985). Current percent and sub-percent level LSS measurements of cosmological parameters, via the matter power spectrum (Rimes & Hamilton 2005; Takahashi et al. 2011), baryonic acoustic oscillations (BAO) (Eisenstein et al. 2005; Takahashi et al. 2009), weak gravitational lensing (Vale & White 2003; Hilbert et al. 2009; Sato et al. 2009) etc., require understandings of nonlinear dynamics of the cosmic structure, and rely on high-resolution and high dynamic range N -body simulations.

When N is large, the brute force pairwise particle-particle (PP) force brings unaffordable $o(N^2)$ computations, so many algorithms were designed to alleviate it. Various fast multipole methods (Rokhlin 1985) improve the complexity to $o(N \log N)$ even $o(N)$, among which the most popular one is “tree”, like GADGET (Springel et al. 2001; Springel 2005) and its simulation “Millennium” (Springel et al. 2005; Angulo et al. 2012), TPM (Xu 1995) and GOTPM (Dubinski et al. 2004). Other methods include adaptive grid algorithms like HYDRA (Couchman et al. 1995) and RAMSES (Teyssier 2010), as well as mesh-refined codes (Couchman 1991) and moving adaptive particle-mesh (PM) codes (Pen 1995). The standard PM algorithm (Hockney & Eastwood 1988) is most memory and computational efficient if we focus on large cosmological scales. The load balancing problem is minimized as the matter distribution is rather homogeneous, and the speed benefits from the fast Fourier transform (FFT) libraries, such as FFTW3 (Frigo & Johnson 2005). PMFAST

(Merz et al. 2005) is first codes to use a 2-level PM algorithm, aiming to put PM codes toward speed, memory compactness and scalability. After subsequent developments on PMFAST, CUBEP3M (Harnois-Déraps et al. 2013) uses cubic spatial decomposition, and adds PP force and many other features.

Beside the methodology, the fast development of parallel supercomputers enable us 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” (Yu et al. 2017b; Emberson et al. 2017) run on the TianHe-2 supercomputer. With the code CUBEP3M adding neutrino modules, it uses 3×10^{12} particles to simulate the cold dark matter (CDM) and cosmic neutrino evolution through the cosmic age.

Relative to the optimized computation optimizations, N -body simulations use considerable amount of memory to store the information of particles. Their phase space coordinates (x, y, z, v_x, v_y, v_z) are stored as at least 6 single-precision floating point numbers (total of 24 bytes). 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, these computation-light but memory-heavy applications, compared to matrix multiplication and decomposition calculations, are currently 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 requirement 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 investigate how most amount of information of particles can be deducted whereas it still preserves the

accuracy of N -body simulations, in order that the total memory usage for given N is optimized.

In the following sections we present a new, information-optimized parallel cosmological N -body simulation code CUBE (Yu 2018), using as low as 6 byte per particle. It gives accurate results in cosmological LSS simulations – the error induced by information optimization is below the error from the PM algorithm. In §2 we show how the memory can be saved by using a “integer-based storage” and how the PM N -body algorithm is adapted with this storage format. In §3 we quantify the accuracy of this algorithm by groups of simulations by CUBEP3M and CUBE. Discussions and conclusions are in §4.

2. METHOD

The most memory consuming part of a N -body simulation is usually the phase space coordinates of N -body particles – 24 byte per particle (here after we use “bpp” referring to “byte per particle”), which contains 4 single precision floating numbers, must be used to store each particles’ 3-dimensional 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. Sometimes 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 resimulation 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. This is similar with storing the quantities of nodes/clumps (structures of a tree in a tree code) relative to their parents (Appel 1985). We divide the cell, in each dimension d , evenly into $2^8 = 256$ bins, and use a 1-byte (8 bits) integer $\chi_d \in \{-128, -127, \dots, 127\}$ 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 d th dimension x_d is given by $x_d = (n_c - 1) + (\chi_d + 128 + 1/2)/256$, where $n_c = 1, 2, \dots, N_c$ 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

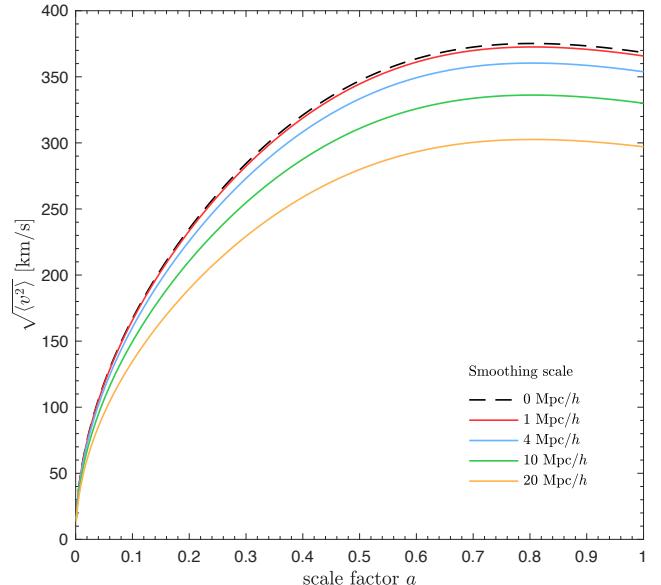


FIG. 1.— Variance of the velocity distribution given by Equation 4. The x -axis is the scale factor characterizing the cosmic evolution and the y -axis shows the σ_v in unit of km/s. The 5 curves, from top to bottom, shows the $\sigma_v(a)$ with increasing smoothing scale r from 0 to 20 Mpc/ h .

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 resolution 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 resolution 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

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

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

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 versions of the coordinate. The velocity counterpart of them are $n_\nu = 1, 2$, v_d and ν_d . The position resolution for n_χ -byte integer, “xn $_\chi$ ”, 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

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

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

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

2.2. Particle velocity storage

Similarly, actual velocity in d th 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 should not occupy considerable 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

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

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 velocity field, we use $\sigma_\Delta^2(a)/3$ according to the equipartition theorem. 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 Λ CDM (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

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



FIG. 2.— Spacial decomposition in CUBE in a 2D analogy. In this example, there are 2 images per dimension ($M_g = 2$), and 2 tiles per image per dimension ($M_t = 2$). The orange boxes show the overlapped extended(V_e)=physical(V_p)+buffer(V_b) tile regions. One physical region is indicated with green.

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

where $\lfloor \cdot \rfloor$ is the operator to take the nearest integer. Tangent functions are convenient and computing-efficient. Compared to error functions used in Gaussian case, they take the same variance at $\nu_d = 0$ but resolve high velocities relatively better. All possible choice of conversion formulae and σ_Δ are unbiased in the conversion Equations (6,7), however a proper choices optimize the velocity space sampling and can result in more precise results.

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. 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 replace MPI (Message Passing Interface) communications between computation nodes/images¹. The algorithm is described in this language.

2.3.1. Spacial decomposition

CUBE decomposes the global simulation volume into M_g^3 cubic sub-volumes with N_c coarse grids or $N_f = RN_c$ fine grids per side. The fine-mesh is usually $R = 4$ times

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

finer than the coarse mesh. Each of these sub-volumes is assigned to a coarray *image*. Inside of an image, the sub-volume is further decomposed into M_t^3 cubic *tiles* (defined as V_p , or *physical* region) with $N_c/(M_g M_t)$ coarse grids per side. Each V_p is surrounded by a *buffer* region V_b which is N_b coarse cells thick. We define extended tile region $V_e \equiv V_p + V_b$. V_e is designed for two reasons: (1) if the short fine-mesh force \mathbf{F}_f has a cut-off N_b , and is computed by ρ_c whose domain $D(\rho_f) = V_e$, then \mathbf{F}_f in V_p is guaranteed to be correct, i.e.,

$$\begin{aligned} \mathbf{F}_f(r > N_b) &= 0 \wedge D(\rho_c) = V_e \\ \Rightarrow \{\mathbf{F}_f(\rho_c) | \mathbf{F}_f \in V_p\} &\text{ is correct,} \end{aligned} \quad (8)$$

and (2) to collect all particles being able to travel to V_p , i.e.,

$$\mathbf{v} dt \leq N_b \Rightarrow \{\mathbf{x} | \mathbf{x} + \mathbf{v} dt \in V_p\} \in V_e. \quad (9)$$

Figure 2 shows the spacial decomposition in a 2-dimensional analogy, with $M_g = 2$ and $M_t = 2$.

According to this spatial decomposition, and as discussed in the last two subsections, we declare $\{\rho_c, v_c, \chi_d, \nu_d\}$ by using Fortran language:

```
integer(1) rho_c(Ne, Ne, Ne, Mt, Mt, Mt)[Mg, Mg, *]
real(4)      v_c(Ne, Ne, Ne, Mt, Mt, Mt)[Mg, Mg, *]
integer(n_chi) chi_d(3, P_max)[Mg, Mg, *]
integer(n_nu)  nu_d(3, P_max)[Mg, Mg, *]
```

where $N_e = N_t + 2N_b$ covers the buffer region on both sides, M_t is the tile dimensions, and M_g is the image *codimensions*². We denote actual number of particles in a given image P_{local} , and $P_{\text{max}} > P_{\text{local}}$ is a value (discussed in Section 2.4) large enough to store particles in V_e . χ_d and ν_d are required to be sorted according to the same memory layout of ρ_c , such that n_c and x_d can be obtained by equation (2). $\{\rho_c, v_c, \chi_d, \nu_d\}$ provides a complete information of positions and velocities of particles, and we call it a *checkpoint*.

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

2.3.2. Initial conditions

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 displacement potential Φ . At an initial redshift z_i , we generate a linear density fluctuation $\delta_L(\mathbf{q})$ on Lagrangian grid \mathbf{q} by multiplying a Gaussian random field with the transfer function $T(k)$ (given by the assumed cosmological model) in Fourier space. Then we solve for the potential $\Phi(\mathbf{q}, z_i)$ of a curl-less displacement field $\Psi(\mathbf{q})$ by Poisson equation $-\nabla^2 \Phi = \delta_L$ in Fourier space.

The second step to generate particles and displace them by Zel'dovich approximation (ZA) (Zel'dovich

```
program Initial_Condition_Generator_for_CUBE
  calculate  $\Phi$  in Fourier space
  do (each tile)
    do (each particle at  $\mathbf{q} \in V_e$ )
       $\Psi(\mathbf{q}) = \nabla \Phi(\mathbf{q})$ 
       $\{\mathbf{q}, \Psi(\mathbf{q})\} \rightarrow \{\mathbf{x}, \mathbf{v}\}$ 
       $\{\mathbf{x}, \mathbf{v}\} \rightarrow \{\rho_c, v_c\}$ 
    enddo
    do (each particle at  $\mathbf{q} \in V_e$ )
       $\Psi(\mathbf{q}) = \nabla \Phi(\mathbf{q})$ 
       $\{\mathbf{q}, \Psi(\mathbf{q})\} \rightarrow \{\mathbf{x}, \mathbf{v}\}$ 
      calculate particle's index  $i$  according to  $\rho_c$ 
       $\mathbf{x} \rightarrow \chi_d(:, i); \{\mathbf{v}, v_c\} \rightarrow \nu_d(:, i)$ 
      if (PID_flag) create  $I_P(i)$ 
    enddo
    do (each coarse grid  $\in V_e$ )
      delete particles  $\in V_b$ 
    enddo
    write  $\{\rho_c, v_c, \chi_d, \nu_d\}$  to disk
    sum up  $P_{\text{local}}$ 
  enddo
  sync all
  sum up  $P_{\text{global}}$ 
end
```

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

1970), where the displacement field is obtained by differentiating Φ , $\Psi(\mathbf{q}) = \nabla \Phi(\mathbf{q})$ in real space. This step is done on V_e of each tile. Similar to Equation (9), we have

$$\max(\Psi_d) \leq N_b \Rightarrow \{\mathbf{q} | \mathbf{q} + \Psi(\mathbf{q}) \in V_p\} \in V_e. \quad (10)$$

We iterate twice over particles' \mathbf{q} in V_e . The first iteration calculates particles' \mathbf{x} and \mathbf{v} by ZA and obtains ρ_c and v_c on the coarse grid. The second iteration, \mathbf{x} and \mathbf{v} is calculated again and are converted to χ_d and ν_d by Equations (1,6) and place them in a certain order according to ρ_c . Lastly, we delete particles in V_b , and resort the ones in V_p and write $\{\rho_c, v_c, \chi_d, \nu_d, I_P(\text{optional})\}$ of this tile to disk. The above is similar to `update_xp` of Section 2.3.3. If PIDs are needed, they are also generated here. After working on all tiles, we sum up P_{local} and P_{global} . We summarize the above steps into a pseudocode in Figure 3. During this step, the only major memory usage is Φ on the fine mesh. If the number of particles per fine grid $P_f = 1$, the memory consumption of this in-place FFT is 4 bpp.

2.3.3. Algorithm

Figure 4 shows the overall structure of the main code. `initialize` – creates fine-mesh and coarse mesh 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 K_c , K_f are also computed or loaded.

`read_particles` – from disk, reads in checkpoint $\{\rho_c, v_c, \chi_d, \nu_d, I_P(\text{optional})\}$ for each image. Because they exist only in V_p of every tile, they are *disjoint*; and they provide *complete* information of the whole simulation volume – we call it “*disjoint state*”. In this state, ρ_c , v_c 's values in buffer regions, and $\chi_d(:, P_{\text{local}} + 1 :)$ and $\nu_d(:, P_{\text{local}} + 1 :)$ are 0's. Because I_P is generated and manipulated together with ν_d , so we do not explicitly mention I_P in the followings.

`buffer_density`, `buffer_x` and `buffer_v` – converts “*disjoint*” state to “*buffered state*”. In `buffer_density`, V_b regions of ρ_c is synchronized between tiles and images. By `buffer_x`, χ_d is updated to contain common, buffered particles, and they are sorted according to the buffered ρ_c . `buffer_v` deals with ν_d in a same manner.

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

```

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.

```

subroutine update_xp
  do (each physical tile)
    do (each particle)
       $\{\rho_c, \chi_d\} \rightarrow \mathbf{x}; \{v_c, v_d\} \rightarrow \mathbf{v}$ 
       $\mathbf{x} = \mathbf{x} + \mathbf{v} dt$ 
      update  $\rho_c^*, v_c^*$  according to  $\mathbf{x}$ 
    enddo
    do (each particle)
       $\{\rho_c, \chi_d\} \rightarrow \mathbf{x}; \{v_c, v_d\} \rightarrow \mathbf{v}$ 
       $\mathbf{x} = \mathbf{x} + \mathbf{v} dt$ 
      calculate particle's index  $i$  according to  $\rho_c^*$ 
       $\mathbf{x} \rightarrow \chi_d^*(\cdot, i); \{\mathbf{v}, v_c\} \rightarrow v_d^*(\cdot, i)$ 
    enddo
    do (each coarse grid)
      discard buffer information
    enddo
    replace  $\{\rho_c, v_c, \chi_d, v_d\}$  with  $\{\rho_c^*, v_c^*, \chi_d^*, v_d^*\}$ 
  enddo
  sync all
  update velocity dispersion  $\sigma_\Delta^2$ 
  sum up  $P_{\text{local}}, P_{\text{global}}$ 
end

```

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

timestep – Runge-Kutta 2 method is used in the time integration. i.e., for n time steps, we update positions (D =drift) and velocities (K =kick) at interleaved half time-steps by operator splitting; the operation $(DKKD)^n$ is 2nd order accurate. The actual simulation applies varied time steps by `timestep`, where a time increment dt is constrained by particles' maximum velocities, accelerations, cosmic expansion and any other desired conditions.

update_xp – According to dt , `update_xp` updates particle positions (drift D) in a “gather” algorithm tile by tile. For each particle, χ_d and v_d are converted to x_d and v_d by Equations(2,7). Due to Condition (9), we are able to collect all particles to travel to V_p .

In order to keep particles ordered, for each tile, we first do $x_d = x_d + v_d dt$ on all particles to obtain an updated density and velocity field on the tile, ρ_c^* and v_c^* . Then, this calculation is done on the same tile again³ to gener-

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

CUBE

```

subroutine update_vp
  do (each extended tile)
    calculate  $\rho_f$  and solve  $\mathbf{F}_f$  in Fourier space
    do (each particle in physical region)
       $\nu_d \rightarrow \mathbf{v}, \mathbf{v} = \mathbf{v} + \mathbf{F}_f dt, \mathbf{v} \rightarrow \nu_d$ 
    enddo
    update max(| $\dot{\mathbf{v}}$ |)
    (PP force)
  enddo
  calculate  $\rho_c$  and solve  $\mathbf{F}_c$  in Fourier space
  do (each particle)
     $\nu_d \rightarrow \mathbf{v}, \mathbf{v} = \mathbf{v} + \mathbf{F}_c dt, \mathbf{v} \rightarrow \nu_d$ 
  enddo
  update max( $v_d$ ), max(| $\dot{\mathbf{v}}$ |)
end

```

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

ate a new, local particle list χ_d^* and ν_d^* by Equations(1,6). Here, the ordering of χ_d^* and ν_d^* relies on ρ_c^* . Then, the third iteration is done on this tile to delete buffer regions of $\{\rho_c^*, v_c^*, \chi_d^*, \nu_d^*\}$. Then the disjoint state of $\{\rho_c^*, v_c^*, \chi_d^*, \nu_d^*\}$ replaces the old $\{\rho_c, v_c, \chi_d, \nu_d\}$. Finally P_{local} and $P_{\text{global}} = \sum P_{\text{local}}$ are updated. These steps are summarized in Figure 5.

update_vp – the PM or particle-particle particle-mesh (P³M) algorithm is applied in this subroutine to update particles's velocities (kick K). We first call `buffer_density` and `buffer_xp` in order that particle positions are in buffered state. Then according to the particle distributions ρ_f on V_e we calculate the fine-mesh force \mathbf{F}_f and update the particle velocities in the V_p . Condition (8) guarantees the correctness of this step. An optional particle-particle (PP) force \mathbf{F}_{pp} can be called to increase the force resolution.

The compensating coarse grid force \mathbf{F}_c is globally computed by using a coarser- (usually by factor of $R = 4$) mesh by dimensional splitting – the cubic distributed coarse density field ρ_c is transposed (inter-image) and Fourier transformed (inner-image) in consecutive three dimensions. After the multiplication of memory-distributed force kernel K_c , the inverse transform takes place to get the cubic distributed coarse force field \mathbf{F}_c , upon which velocities are updated again.

For each type of velocity updates, the collective operations are Equation(7), $\mathbf{v} = \mathbf{v} + \mathbf{F}_{\text{total}}$ and Equation(6). We also update σ_Δ^2 according to the new $v_d - v_c$. These steps are summarized in Figure 6.

By far, ν_d in physical regions are correctly updated. We simply call `buffer_v` again to bring ν_d 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 $\{\rho_c, v_c, \chi_d, \nu_d\}$ on disk. Related operations, like runtime halo finder, projections are also done at this point.

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.

sive.

2.4.1. Arrays of particles

These are the majority memory usage, contains checkpoint arrays χ_d, ν_d, I_P , and temporary arrays χ_d^*, ν_d^*, I_P^* . The former uses memory $\mathcal{M} = (3n_\chi + 3n_\nu + n_I)P_{\max}$ byte, where

$$P_{\max} = \langle P_{\text{local}} \rangle \left(1 + \frac{2N_b}{N_t}\right)^3 (1 + \epsilon_{\text{image}}), \quad (11)$$

and $\langle P_{\text{local}} \rangle$ is the average number of particles per image. The second term, proportional to V_e/V_p , let us store additional particles in V_b , and the third term $1 + \epsilon_{\text{image}}$ takes into account the inhomogeneity of P_{local} on different images. When each image models smaller physical scales, ϵ_{image} should be set larger.

Temporary χ_d^*, ν_d^*, I_P^* store particles only on tiles, and the particle number is set to be

$$P_{\max}^* = \langle P_{\text{local}} \rangle \left(\frac{1}{M_t}\right)^3 (1 + \epsilon_{\text{tile}}), \quad (12)$$

where ϵ_{tile} controls the inhomogeneity on scale of tiles. Larger M_t causes more inhomogeneity on smaller tiles, and ϵ_{tile} can be much larger than ϵ_{image} , however the term M_t^{-3} decreases much faster. Practically, the majority memory is occupied by χ_d, ν_d, I_P .

Summarizing above we obtain the memory usage per particle (byte per particle) $\mathcal{M}_P \equiv \mathcal{M}/\langle P_{\text{local}} \rangle$ /byte, given $n_\chi = n_\nu = 1$ and $n_I = 0$, is

$$\begin{aligned} \mathcal{M}_P^{\text{particle}} = 6 & \left[(1 + 2N_b N_t^{-1})^3 (1 + \epsilon_{\text{image}}) \right. \\ & \left. + M_t^{-3} (1 + \epsilon_{\text{tile}}) \right]. \end{aligned} \quad (13)$$

Because $N_t = N_c/(M_g M_t)$, we can minimize Equation (13) by tuning M_t .

2.4.2. Coarse mesh arrays

On coarse mesh, ρ_c (4-byte integers), v_c and force kernel K_c should always be kept. They are usually configured to be $R = 4$ times coarser than fine grids and particle number density. They have use memory $(1+3) \times 4 \times (N_e M_t)^3 + 3 \times 4 \times (N_c/M_g)^3/2$ bytes per image, or

$$\mathcal{M}_P^{\text{coarse}} = P_c^{-1} \left[16 \left(1 + \frac{2N_b}{N_t}\right)^3 + 6 \right], \quad (14)$$

where P_c is the average number of particles per coarse cell. Other coarse grid based arrays are ρ_c^* , F_c pencil-FFT arrays. ρ_c^* exists only on tiles; F_c and pencil-FFT arrays can be equivalenced with other temporary arrays. Thus the major memory usage from coarse FFT arrays are from Equation (14).

2.4.3. Fine mesh arrays

On the local fine mesh, only a force kernel array K_f needs to be kept. It has memory of $3 \times 4 \times (R N_e M_t)^3/2$ per image, or

$$\begin{aligned} \mathcal{M}_P^{\text{fine}} &= 6P_f^{-1} N_t^{-3} \left(1 + \frac{2N_b}{N_t}\right)^3 \\ &= 6R^3 P_c^{-1} N_t^{-3} \left(1 + \frac{2N_b}{N_t}\right)^3, \end{aligned} \quad (15)$$

TABLE 1
MEMORY LAYOUT FOR A CERTAIN CONFIGURATION

Type	Array	Memory usage		
		/GB	/bpp	Percentage
Particles	χ_d, ν_d	29.9	8.24	83.8%
	χ_d^*, ν_d^*	3.16	0.872	8.87%
	I_P, I_P^*	0	0	0%
	Subtotal	33.0	9.12	92.7%
Coarse mesh	ρ_c	0.296	0.0818	0.831%
	v_c	0.889	0.245	2.49%
	K_c	0.340	0.0939	0.954%
	F_c	(0.690)	(0.190)	(1.94%)
	ρ_c^*	0.0140	0.00388	0.0394%
	Pencil-FFT	(0.454)	(0.125)	(1.27%)
	Subtotal	1.55	0.429	4.36%
Fine mesh	K_f	1.06	0.292	2.97%
	F_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%

where P_f is the average number of particles per fine cell. The fine mesh density arrays, force field arrays are temporary. Since χ_d^*, ν_d^* , 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 refer to the TianNu simulation (Yu et al. 2017b) run on the Tianhe-2 supercomputer, which used a traditional N -body code CUBEP3M. TianNu's particles number is limited by memory per computing node – for each computing node, an average of 576^3 neutrino particles and 288^3 CDM particles are used, and consumes $\mathcal{M} = 40$ GB⁴, or about $\mathcal{M}_P = 186$. A memory efficient configuration of CUBEP3M by using large physical scales and at costs of speed, still uses about $\mathcal{M}_P = 40$.

If the same amount of memory is allocated to CUBE, we can set parameters as $n_\chi = n_\nu = 1$, $N_c/M_g = 384$, $N_b = 6$, $\epsilon_{\text{image}} = 5\%$, $\epsilon_{\text{tile}} = 200\%$ and thus $\langle P_{\text{local}} \rangle = 1536^3$. Setting $M_t = 3$ (27 tiles per image) minimizes \mathcal{M} and uses about $\mathcal{M} = 35.6$ GB, corresponding to $\mathcal{M}_P = 9.84$. 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 byte), 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.

In the bottom of Table 1 we stress that the optimal memory usage is 6 bpp, or 21.7 GB, 61% of the actual \mathcal{M} . The dominating departure from this limit is that χ_d and ν_d already occupy 8.24 bpp, which come from the $(1 + 2N_b/N_t)^3$ term of Equation (13). All other variables

⁴ Additional memory is used for OpenMP parallelization and for particle IDs to differentiate different particle species.

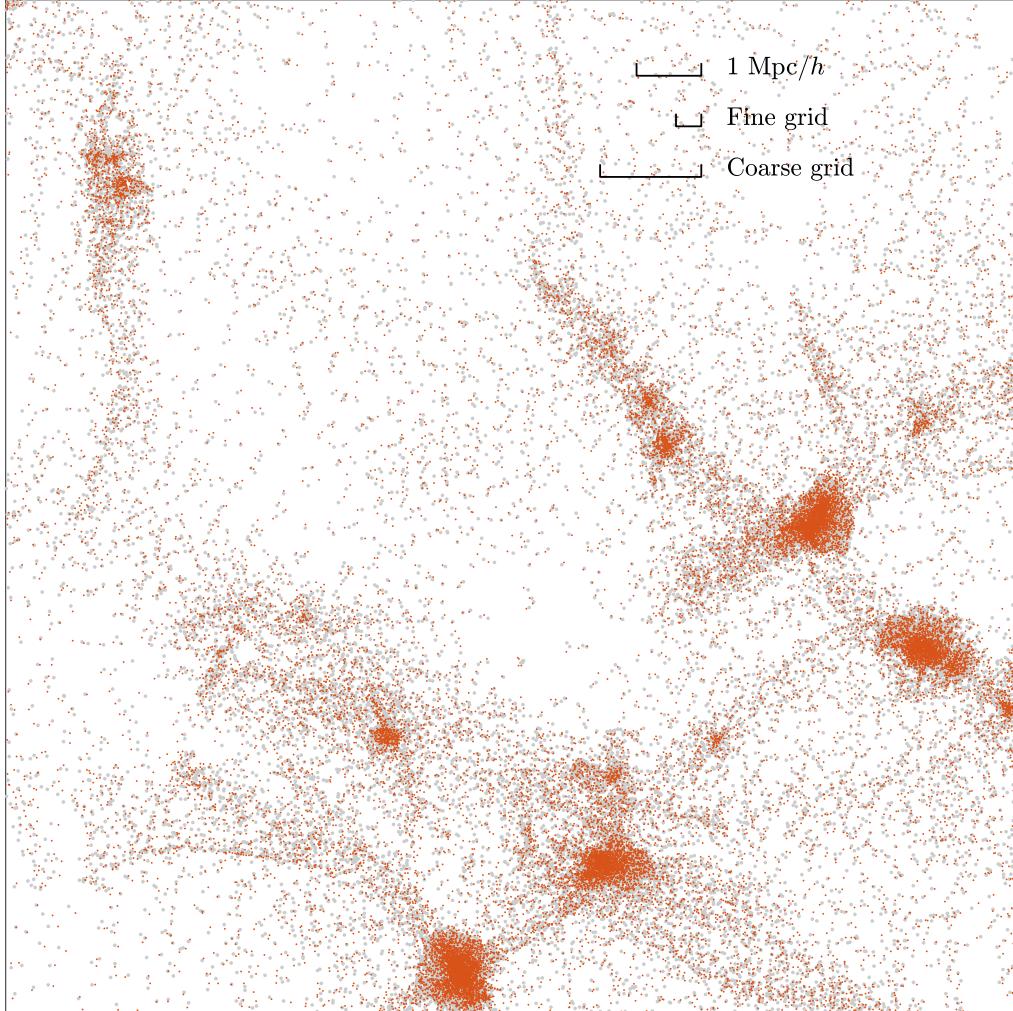


FIG. 7.— Offset in the particle distribution induced by integer-1 based algorithm (x1v1). In S512’s subregion of volume $15.625^2 \times 3.90625$ (Mpc/h) 3 , particles from CUBEp3M (bigger grey dots in the background) and CUBE-x1v1 (smaller red dots) are projected onto the plane of $(15.625 \text{ Mpc}/h)^2$. The comparing rules show $1 \text{ Mpc}/h$, fine grid and coarse grid respectively, and position resolution of x1v1 is $1/64$ of a fine grid.

TABLE 2
SIMULATION CONFIGURATIONS

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

occupy additional 8%. On modern super computers, the memory per computing node is usually much larger, and by scaling up the number of particles per node, the buffer ratio N_b/N_t 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 $z = 0$. We use the same force kernels as CUBEp3M without PP force. Note that near find grid scales there are possibilities to enhance the force kernel to better match the nonlinear power spectrum predictions, however we use the conservative mode of CUBEp3M in this paper. An extended PP force and an unbiased force matching algorithm will be added to CUBE. The power spectrum studies are presented in Harnois-Déraps et al. (2013), and here we focus on the cross correlations between different integer based methods.

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 sufficient memory is provided, one can always use x2v2 to get exact results as CUBEp3M, 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.

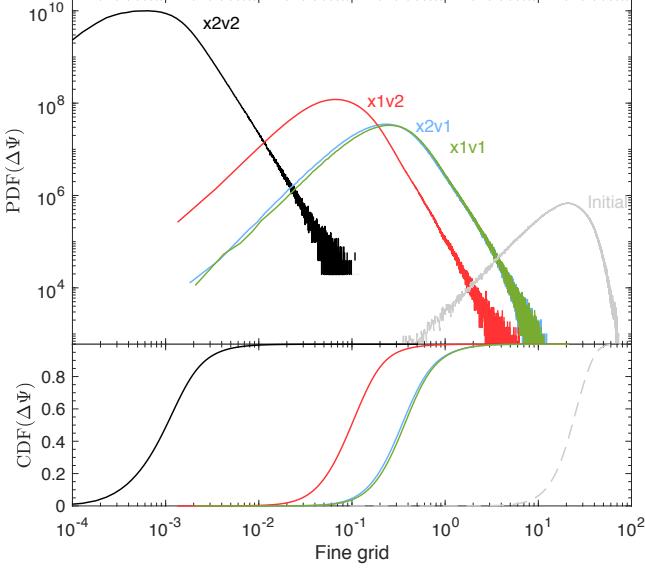


FIG. 8.— Statistics of the error in particle displacement $\Delta\Psi$ induced by integer based algorithms in CUBE. The PDF and CDF of $|\Delta\Psi|$ from S256 are shown in the upper and lower axes, as functions of fine grids. Black, red, blue and green correspond to x2v2, x1v2, x2v1 and x1v1 respectively. The grey lines, marked with “initial”, shows the distribution of actual displacement of particles in CUBEPM, $|\Psi_0|$, which is orders of magnitudes larger than $|\Delta\Psi|$.

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

3.1. Power spectrum

3.2. Displacement of particles

In S512, we zoom into a small region of $15.625^2 \times 3.90625$ (Mpc/h)³ and compare the particle distribution between CUBEPM and CUBE-x1v1 in Figure 7. For clarity, CUBEPM particles are marked with bigger, grey dots, whereas smaller red dots are CUBE-x1v1 particles overplotted onto them. One can see the difference between them. The $1 \text{ Mpc}/h$, 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 $\Psi(\mathbf{q}) \equiv \mathbf{x} - \mathbf{q}$ of every particle (Yu et al. 2017a), where \mathbf{x} and \mathbf{q} are Eulerian and Lagrangian coordinates of the particle. Then we calculate the absolute value of the offset vector

$$\Delta\Psi \equiv |\Psi_i - \Psi_0|. \quad (16)$$

Here, Ψ_0 stands for CUBEPM and subscript i can stand for x2v2, x1v2, x2v1 or x1v1. The probability distribution functions (PDFs) and cumulative distribution functions (CDFs) of $\Delta\Psi$ 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 dis-

placement of particles (by replacing Ψ_i with \mathbf{q} in Equation (16)) 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 CUBEPM. 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 $L = 600 \text{ Mpc}/h$ ($m_p = 1.2 \times 10^{12} M_\odot$), 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 x1v1.

S2048S and S2048L are two simulations with 2048^3 particles in small ($L = 400 \text{ Mpc}/h$) and large ($L = 1200 \text{ Mpc}/h$) box sizes. We compare their accuracy by their power spectra and their cross correlations with CUBEPM at $z = 0$. 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). On the other hand, S2048L focuses on larger structures, on which scale one can study weak gravitational lensing, baryonic acoustic oscillation (BAO) (Eisenstein et al. 2005) and its reconstruction (Eisenstein et al. 2007; Wang et al. 2017) etc.

For each simulation the particles are firstly cloud-in-cell (CIC) interpolated onto the fine mesh grid, and from the density field ρ we define the density contrast $\delta \equiv \rho/\langle\rho\rangle - 1$. 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

$$\langle \delta_\alpha^\dagger(\mathbf{k}) \delta_\beta(\mathbf{k}') \rangle = (2\pi)^3 P_{\alpha\beta}(k) \delta_{3D}(\mathbf{k} - \mathbf{k}'), \quad (17)$$

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

$$\xi(k) \equiv P_{\alpha\beta}/\sqrt{P_{\alpha\alpha}P_{\beta\beta}}. \quad (18)$$

In the upper two panels of figure 9 we show the power spectra of CUBE. In both plots of S2048S and S2048L the four solid curves of different colors show the results of x2v2, x1v2, x2v1 and x1v1, and they almost overlapped with each other. The dashed curves are the nonlinear prediction of matter power spectrum by CLASS (Blas et al. 2011).

We label $k = 0.2 k_{\text{Nyquist}}$ as vertical dashed lines, where k_{Nyquist} is the scale of fine mesh grids, and the scale of average particle separations. On this scale, the power spectra are offset from nonlinear predictions by at least 20%. This error is from the PM algorithm and one has to increase the resolution of the simulation to correct these offsets. We do not plot CUBEPM because we examined that CUBEPM and x2v2 of CUBE produce same results. Thus, the differences between CUBEPM and different integer formats of CUBE are negligible compared to the error of the PM algorithm.

In the lower parts of these four panels study the cross correlations. We compare everything with CUBE-x2v2, to emphasize the decorrelation by different integer formats.

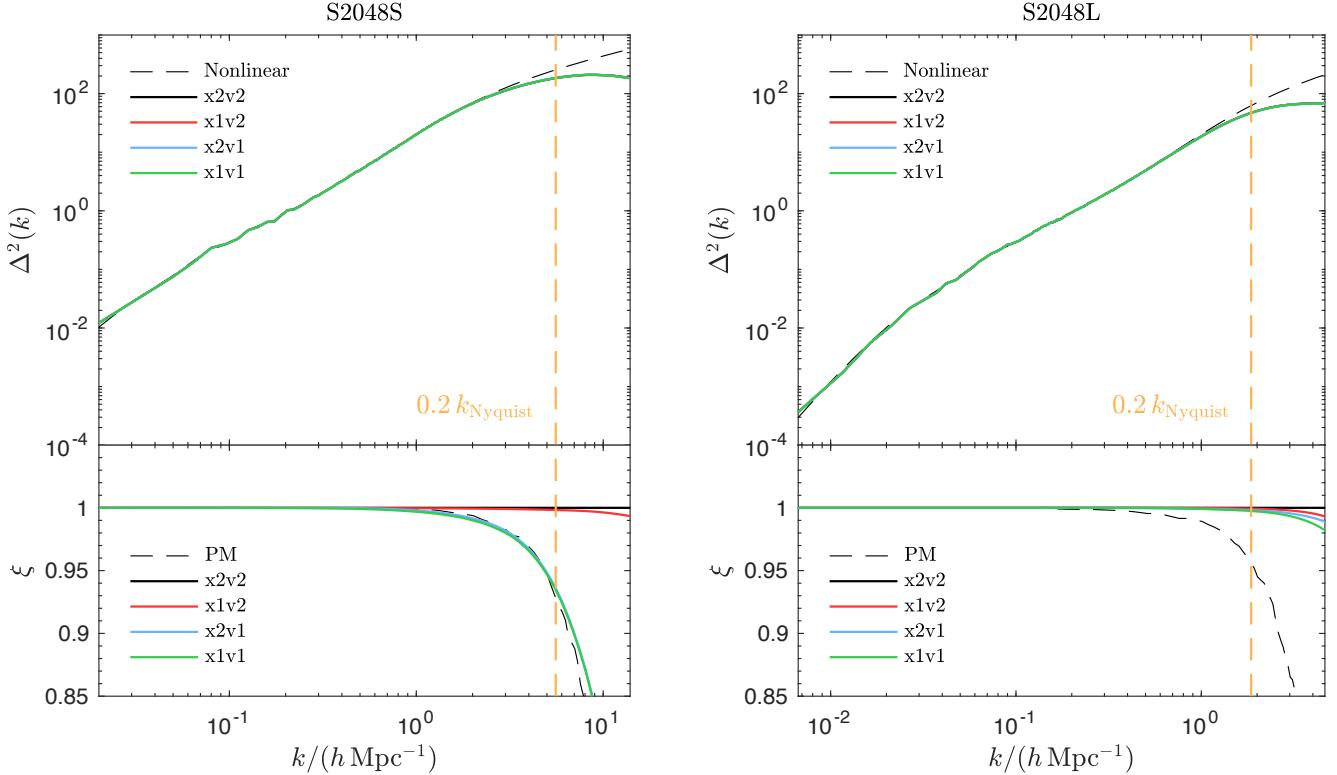


FIG. 9.— Dimensionless power spectra $\Delta^2(k)$ and decorrelations $\xi(k)$ (lower axes) induced by the integer-based algorithms in simulations S2048S and S2048L. The four solid lines show the results from $\{x_2v_2, x_1v_2, x_2v_1, x_1v_1\}$. In lower panels, the dashed curves show the decorrelations given by the PM algorithm, measured by cross correlating different resolutions of CUBE-x2v2. The vertical orange dashed lines show the scale $k = 0.2 k_{\text{Nyquist}}$.

Note that x2v2 is perfectly correlated with CUBEP3M. In the lower two panels of figure 9 the solid curves show the decorrelation ξ of x1v2, x2v1 and x1v1. Higher resolution (S2048S) in general comes with more decorrelations, and x1v2 cross correlate with x2v2 better than the other two cases. In order to quantify the PM error in terms of decorrelations, we run x2v2 simulations with same initial conditions, but 8 times more particles and cells, and measure the decorrelation caused by coarser resolutions. These results are shown in dashed curves (labeled “PM”). We conclude that cross correlation of x1v1, in all cases, is not worse than the PM errors.

To summarize from Figure 9, in terms of either power spectrum deviation or cross correlation, the error induced by information optimization (even for x1v1) is lower than the error from the PM algorithm, and we can safely use x1v1 for most of the LSS studies.

4. DISCUSSION AND CONCLUSION

We present an parallel, information-optimized N -body algorithm. This open-source code CUBE, has recently been used in many studies of LSS, e.g. Yu et al. (2017a); 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 code is 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. Note that the results are exactly same in that they not only pro-

duce same physical statistics of LSS, but also the same error (not physical) from the PM algorithm near Nyquist frequencies. In another word, the positions and velocities of each particle are exact. In practice, we only require that the errors from information optimization is much lower than the errors from PM algorithm. In Figure 9 we see that it is the case even for the most memory efficient configuration x1v1. This shows that in most LSS studies, when our scales of interest are smaller than $k \simeq 0.2 k_{\text{Nyquist}}$, six 1-byte fixed point numbers contains sufficient information of every N -body particle.

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 is designed to optimize the efficiency of information and the memory usage in N -body simulations. CUBE is written in Coarray Fortran – the concise Coarray features are used instead of complicated MPI, and the code itself is much more concise than CUBEP3M for future mainte-

nance and development. 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.

We acknowledge funding from NSERC. H.R.Y. thanks Derek Inman for many helpful discussions. The simulations were performed on the GPC supercomputer at the SciNet HPC Consortium. The code is publicly available on github.com/yuhaoran/CUBE.

REFERENCES

- Angulo, R. E., Springel, V., White, S. D. M., et al. 2012, MNRAS, 426, 2046
- Appel, A. W. 1985, SIAM Journal on Scientific and Statistical Computing, vol. 6, no. 1, January 1985, p. 85-103., 6, 85
- Blas, D., Lesgourgues, J., & Tram, T. 2011, J. Cosmology Astropart. Phys., 7, 034
- Couchman, H. M. P. 1991, ApJ, 368, L23
- Couchman, H. M. P., Thomas, P. A., & Pearce, F. R. 1995, ApJ, 452, 797
- Davis, M., Efstathiou, G., Frenk, C. S., & White, S. D. M. 1985, ApJ, 292, 371
- Dubinski, J., Kim, J., Park, C., & Humble, R. 2004, New A, 9, 111
- Eisenstein, D. J., Seo, H.-J., Sirk, E., & Spergel, D. N. 2007, ApJ, 664, 675
- Eisenstein, D. J., Zehavi, I., Hogg, D. W., et al. 2005, ApJ, 633, 560
- Emberson, J. D., Yu, H.-R., Inman, D., et al. 2017, Research in Astronomy and Astrophysics, 17, 085
- Frigo, M., & Johnson, S. G. 2005, in PROCEEDINGS OF THE IEEE, 216–231
- Harnois-Déraps, J., Pen, U.-L., Iliev, I. T., et al. 2013, MNRAS, 436, 540
- Hilbert, S., Hartlap, J., White, S. D. M., & Schneider, P. 2009, A&A, 499, 31
- Hockney, R. W., & Eastwood, J. W. 1988, Computer simulation using particles
- Merz, H., Pen, U.-L., & Trac, H. 2005, New A, 10, 393
- Pan, Q., Pen, U.-L., Inman, D., & Yu, H.-R. 2017, MNRAS, 469, 1968
- Pen, U.-L. 1995, ApJS, 100, 269
- Rimes, C. D., & Hamilton, A. J. S. 2005, MNRAS, 360, L82
- Rokhlin, V. 1985, Journal of Computational Physics, 60, 187
- Sato, M., Hamana, T., Takahashi, R., et al. 2009, ApJ, 701, 945
- Springel, V. 2005, MNRAS, 364, 1105
- Springel, V., Yoshida, N., & White, S. D. M. 2001, New A, 6, 79
- Springel, V., White, S. D. M., Jenkins, A., et al. 2005, Nature, 435, 629
- Takahashi, R., Yoshida, N., Takada, M., et al. 2009, ApJ, 700, 479
- . 2011, ApJ, 726, 7
- Teyssier, R. 2010, RAMSES: A new N-body and hydrodynamical code, Astrophysics Source Code Library, , ascl:1011.007
- Vale, C., & White, M. 2003, ApJ, 592, 699
- Wang, X., Yu, H.-R., Zhu, H.-M., et al. 2017, ApJ, 841, L29
- Xu, G. 1995, ApJS, 98, 355
- Yu, H. R. 2018, CUBE: Cosmological N-body simulation code, v.1.0.0, Zenodo, doi:10.5281/zenodo.1198909
- Yu, H.-R., Pen, U.-L., & Zhu, H.-M. 2017a, Phys. Rev. D, 95, 043501
- Yu, H.-R., Emberson, J. D., Inman, D., et al. 2017b, Nature Astronomy, 1, 0143
- Zel'dovich, Y. B. 1970, A&A, 5, 84