Draft version September 10, 2016
Preprint typeset using LaTeX style emulateapj v. 11/10/09

COMPACT PARTICLE-MESH N-BODY ALGORITHM

Authors
Affiliations
Draft version September 10, 2016

Abstract

N-body algorithms are computation-light but memory-heavy imbalanced problems in supercomputing. Traditional N-body simulation algorithms use at least 24-byte per particle in memory, where six 4-byte single precision real numbers keep track of each particle's phase space coordinate. This is usually the bottle neck of scaling-up of a N-body problem on modern supercomputers. Here we present a parallel, memory-lite Particle-Mesh based N-body algorithm, in which each particle can occupy as low as 9-byte memory. This is accomplished by storing only particles' relative location respect to a meshed grid, and their velocity relative to a predictable distribution function. The remaining information is given by the ordering of particles in memory space, and a complimentary density field, which is negligible in memory. Our numerical experiments show that this algorithm provides acceptable accuracy compared to traditional algorithm in cosmological simulations.

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 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. However `CUBEP3M` and TianNu did not fully use TianHe-2's computing power – 87% of the computing power on TianHe-2 are given by the Intel Xeon-Phi coprocessors while they occupy only 27% of the total memory. Although we accomplished preliminary works on developing native and offload modes of `CUBEP3M` on Xeon-Phi many integrated core (MIC) architecture, the memory problem is still the bottle neck of scaling up the N-body problem on the heterogeneous architectures. Lower the memory usage per particle will solve this problem, and will also help the N-body problem development on GPUs, which are even more massive parallelized and memory limited. In this paper we present a N-body algorithm which uses as low as 9 byte per particle (here after we use "bpp" referring to "byte per particle") and show that it can give accurate cosmological LSS simulations. The algorithm is presented in §2, and a comparison between this method and traditional method is shown in §3. Discussions and conclusions are in §4.

2. ALGORITHM

The most memory consuming part of a N-body simulation is usually the phase space coordinates of each N-body particles – 24 bpp (4 single precision real numbers) must be used to store each particles' position and velocity. A memory-lite parallel N-body code `CUBEP3M` can use as low as 40 bpp in sacrificing the speed. 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. We attempt to replace the coordinates and linked list 24+4=28 bpp memory usage with a integer based storage, described as follows.

haoran@cita.utoronto.ca

2.1. *Particle location storage*

Instead of storing the particle's global coordinates, we store its relative position respective to the coarse cell who contains the particle. We divide the coarse cell, in each dimension, evenly into $2^8 = 256$ bins, and we use a 1-byte integer (with range 0 to 255 or -128 to 127) to show which bin it locates in this dimension. Next, global locations of particles are given by cell-ordered format in memory space, and a complimentary number count of particle number in this coarse mesh (coarse density field) will give complete information of particle distribution in cells. Since the this field is usually coarser than particle distribution, it takes negligible memory. This coarse density field can be further compressed into 1-byte integer format, such that a 1-byte unsigned 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 $L/(256N_c)$, which is usually 10 times finer than the force resolution.

2.2. *Particle velocity storage*

Velocities, in each dimension, are compressed to 2-byte integers representing their index within a histogram containing 2^{16} equally spaced bins ranging from $-v_{\rm max}$ to $v_{\rm max}$, where $v_{\rm max}$ is the maximum absolute particle velocity. $v_{\rm max}$ can be derived from either computation or prediction. We prefer the latter case. In cosmological simulations, velocity distribution of matter fluids can be predicted by linear theory, given a model and redshift z (time of the simulation). Then by any time step, we have a cumulative distribution function (CDF) $F(|\boldsymbol{v}|)$ of

particles' velocity (see appendix A). We, instead, equally divide the y-axis of this CDF into 2^{16} bins, and the 2-byte integer shows which velocity bin the particle locates, in each dimension. This step does finer sampling on low-velocity end in velocity space and increases the simulation accuracy. This CDF may introduce velocity cutoffs, that particle cannot exceed some certain $\boldsymbol{v}_{\max}(z)$ given by the model. This process prevents the nonphysical scattering behaviours of particles, which do not exist in fluids. Practically, simulation results are very insensitive to the choice of velocity distribution. Even a top-hat distribution in range $-v_{\max}$ to $v_{\max}$ give accurate results. One can just assume a Gaussian distribution and whose standard deviation is given by the model and redshift $\sigma(\boldsymbol{\theta}, z)$. Proper choice of model only optimizes the velocity space sampling, but does not affect the physics.

2.3. *Spacial decomposition*

`CUBE` uses cubic decomposition structures. The global simulation volume is decomposed into $\texttt{nn}^3$ cubic sub-volumes, and each of these are assigned to a coarray *image*[1]. Inside of an image, the sub-volume is further decomposed into $\texttt{nnt}^3$ cubic *tiles*. Each tile is surrounded by a *buffer* region which is usually `ncb` coarse cells thick. The buffer is designed for two reasons: (1) computing the fine mesh force, whose cut-off $\texttt{nforce\_cutoff} \leq \texttt{ncb}$, and (2) collecting all possible particles travelling from a tiles buffer region to its center, *physical* region. Thus, a integer-based coarse mesh density is defined as

$$\texttt{integer(1) rho\_c(nex, nex, nex, nnt, nnt, nnt)[nn, nn, *]}$$

where $\texttt{nex} = \texttt{nt} + 2\texttt{ncb}$ covers the buffer region on both sides, `nnt` is the tile dimemsions, and `nn` is the image *codimensions*[2]. Particles' phase coordinates `x` and `v` are copied between tiles and images, when necessary.

```
program CUBE
  call initialize
  call particle_initialize
  sync all
  call buffer_density
  call buffer_x
  call buffer_v
  do
    call timestep
    call update_x
    call buffer_density
    call buffer_x
    call PM ! update_v
    call buffer_v
    if(checkpoint_step) then
       call checkpoint
       if (final_step) exit
    endif
  enddo
  call finalize
endprogram
```

FIG. 1.— Overall structure of `CUBE`.

[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.

2.4. *Code overview*

By using the previous compact format storage, we modified the `CUBEP3M` and utilized it in the outputs TianNu simulation. On average, 9.125 bpp is used for all outputs and saves 62 % hard drive space on Tianhe-2. The compact version of `CUBEP3M`, named `CUBE`, accomplished the compact format in memory, and the code keeps only integer format for storing particles' phase space coordinates. `CUBE` is written in Coarray Fortran, where the coarray features are used to do MPI communications. Here we review the methodology of solving this integer based N-body problem. Like its predecessor `CUBEP3M`, `CUBE` uses two-level mesh grids for solving the Poisson equations. The advantages of this are described in Harnois-Déraps et al. (2013).

Figure 1 shows the overall structure of the code. 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 are also created or read in here. In `particle_initialize`, for each image, we read in initial positions and velocities in the compressed format. When this step is done by all images (synchronized by "`sync all`", which is equivalent to a `mpi_barrier`), the density fields `rho_c` are buffered between tiles (communications between images are needed) by subroutine `buffer_density`. Then, particles' phase space coordinates `x` and `v` are also buffered, by `buffer_x` and `buffer_v` respectively. In these steps, `x` and `v`'s ordering is changed according to the updated `rho_c`.

In each iteration of the main loop, we firstly call `timestep`, where a increment of time `dt` is controlled by particles' maximum velocities, accelerations, and cosmic expansions. According to `dt`, subroutine `update_x` updates the positions of particles in a "*gather*" algorithm. $\texttt{x} = \texttt{x} + \texttt{v} * \texttt{dt}$ is executed twice, first time to determine a updated density field on the tile, `rho_c_new`, and second time to generate a new particle list `x_new` and `v_new` on the tile. The reason for this repeated execution is the dependence of `x_new` and `v_new`'s value and ordering on both the old `x`, `v` and the new `rho_c_new`. However, $\texttt{x} = \texttt{x} + \texttt{v} * \texttt{dt}$ scales as $o(N)$ and is computational inexpensive. Although `x_new` and `v_new` arrays take extra memory, they do not appear simultaneously for multiple tiles, and compared to `x` and `v`, the extra memory overhead is by factor of $1/\texttt{nnt}^3$. This requires the "gather" algorithm to move particles[3] – as we synchronize the correct {`rho_c`, `x`, `v`} in the buffer region of each tile, and the particles in the extended (physical+buffer) region are a superset of particles locating in the physical region in the next time step, given that $\texttt{v} * \texttt{dt}$ is not greater than the buffer depth. In `update_x`, {`rho_c_new`, `x_new`, `v_new`} can be set to collect only the physical region of the tile, with the rest of this subset discarded. At the end, {`rho_c`, `x`, `v`} is replaced by {`rho_c_new`, `x_new`, `v_new`}, and the memory of latter is freed up.

Next, `buffer_density` and `buffer_x` are called again to synchronize the updated particle positions in order that tile-based local fine forces are computed correctly.

A standard particle-mesh scheme is then performed in the `PM` subroutine. First, by looping over tiles, fine force

[3] In contrast, `CUBEP3M` uses a "*scatter*" algorithm.

is calculated and velocities are updated on the physical region of each tile. The memory overhead of fine-mesh arrays are well controlled by $1/\texttt{nnt}^3$. Next, global coarse force is solved 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, upon which velocities are updated again. The memory usage of coarse arrays are negligible. An optional particle-particle (PP) force can be called to increase the force resolution and the velocities are updated last time according to this. As discussed in §2.2, updating velocities use model predicted velocity distribution, thus $\texttt{v} = \texttt{v} + \texttt{dv}$ is executed only once, scaling directly to the velocity distribution of the next time step $\texttt{d}+\texttt{dt}$. During each of these force calculations, maximum accelerations are collected serving for the `timestep` in the next iteration, controlling next `dt`.

By far, particle velocities in the physical regions of each tile are updated. Particle locations in the buffer regions has updated before `PM` and remained unchanged. So we simply call `buffer_v` again, such that the `update_x` in the next iteration will be done correctly.

If a desired redshift is reached, we call `checkpoint` and/or exit the main loop. The corresponding logical variables are controlled in `timestep`. Finally, in `finalize` subroutine we destroy all the FFT plans and finish up any timing or statistics taken in the simulation.

2.5. *Memory layout*

3. RESULTS

4. DISCUSSION AND CONCLUSION

APPENDIX

VELOCITY DISTRIBUTION FUNCTION

Thanks.

REFERENCES

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