FDPS Tutorial

Ataru Tanikawa, Masaki Iwasawa, Natsuki Hosono, Keigo Nitadori,
Takayuki Munanushi, Daisuke Namekata, and Junichiro Makino
Particle Simulator Research Team, AICS, RIKEN

0 Contents

1 Change log

- 2015/03/17 English version created
- 2015/06/04 Spell-checked complete version
- 2016/01/18 Description of GPU version added (Sec. 3.4.1.8)
- 2018/07/11
  - Typographical error correction in Section 3:
    - Update the information of compilers tested (using FDPS ver. 4.1a)
    - The extension of output file of the SPH sample code is wrong
  - Typographical error correction in Section 4:
    - The section number of DomainInfo class in the specification document is wrong (Sec. 4.1)
    - The arguments of calcForceAllAndWriteBack() in the document is not consistent with the source code (Sec. 4.1)
  - Typographical error in Section 6:
    - Description of the PP part is not consistent with the source code (Sec. 6.1)
    - Description of the file structure is not consistent with the sample code (Sec. 6.2)
    - The arguments of decomposeDomainAll() in the document is not consistent with the source code (Sec. 6.2)

2 Overview

In this section, we present the overview of Framework for Developing Particle Simulator (FDPS). FDPS is an application-development framework which helps the application programmers and researchers to develop simulation codes for particle systems. What FDPS does are calculation of the particle-particle interactions and all of the necessary works to parallelize that part on distributed-memory parallel computers wit near-ideal load balancing, using hybrid parallel programming model (uses both MPI and OpenMP). Low-cost part of the simulation program, such as the integration of the orbits of particles using the calculated interaction, is taken care by the user-written part of the code.

FDPS support two- and three-dimensional Cartesian coordinates. Supported boundary conditions are open and periodic. For each coordinate, the user can select open or periodic boundary.

The user should specify the functional form of the particle-particle interaction. FDPS divides the interactions into two categories: long-range and short-range. The difference between two categories is that if the grouping of distant particles is used to speedup calculation (long-range) or not (short range).

The long-range force is further divided into two subcategories: with and without a cutoff scale. The long range force without cutoff is what is used for gravitational N-body simulations with open boundary. For periodic boundary, one would usually use TreePM, P^3M, PME or other variant, for which the long-range force with cutoff can be used.

The short-range force is divided to four subcategories. By definition, the short-range force has some cutoff length. If the cutoff length is a constant which does not depend on the identity of particles, the force belongs to "constant" class. If the cutoff depends on the source or receiver of the force, it is of "scatter" or "gather" classes. Finally, if the cutoff depends on both the source and receiver in the symmetric way, its class is "symmetric". Example of a "constant" interaction is the Lennard-Jones potential. Other interactions appear, for example, SPH calculation with adaptive kernel size.

The user writes the code for particle-particle interaction kernel and orbital integration using C++ language. We are studying the possibility to allow users to write their code in traditional Fortran language.

3 Getting Started

In this section, we describe the first steps you need to do to start using FDPS. We explain the environment (the supported operating systems), the necessary software (compilers etc), and how to compile and run the sample codes.

3.1 Environment

FDPS works on Linux, Mac OS X, Windows (with Cygwin).

3.2 Necessary software

In this section, we describe software necessary to use FDPS, first for standard functions, and then for extensions.

3.2.1 Standard functions

we describe software necessary to use standard functions of FDPS. First for the case of single-thread execution, then for multithread, then for multi-nodes.

3.2.1.1 Single thread

- make
- A C++ compiler (We have tested with gcc version 4.8.3 and K compiler version 1.2.0)

3.2.1.2 Parallel processing

3.2.1.2.1 OpenMP

- make
- A C++ compiler with OpenMP support (We have tested with gcc version 4.8.3 and K compiler version 1.2.0)

3.2.1.2.2 MPI

- make
- A C++ compiler which supports MPI version 1.3 or later. (We have tested with Open MPI 1.6.4 and K compiler version 1.2.0)

3.2.1.2.3 MPI+OpenMP

- make
- A C++ compiler which supports OpenMP and MPI version 1.3 or later. (We have tested with Open MPI 1.6.4 and K compiler version 1.2.0)

3.2.2 Extensions

Current extension for FDPS is the "Particle Mesh" module. We describe the necessary software for it below.

3.2.2.1 Particle Mesh

- make
- A C++ compiler which supports OpenMP and MPI version 1.3 or later. (We have tested with Open MPI 1.6.4)
- FFTW 3.3 or later

3.3 Install

In this section we describe how to get the FDPS software and how to build it.

3.3.1 How to get the software

We first describe how to get the latest version, and then previous versions. We recommend to use the latest version.

3.3.1.1 The latest version

You can use one of the following ways.

- Using browsers
  1. Click "Download ZIP" in https://github.com/FDPS/FDPS to download `FDPS-master.zip`
  2. Move the zip file to the directory under which you want to install FDPS and unzip the file (or place the files using some GUI).
- Using CLI (Command line interface)
  - Using Subversion:

```
$ svn co --depth empty https://github.com/FDPS/FDPS
$ cd FDPS
$ svn up trunk
```

  - Using Git

```
$ git clone git://github.com/FDPS/FDPS.git
```

3.3.1.2 Previous versions

You can get previous versions using browsers.

- Previous versions are listed in https://github.com/FDPS/FDPS/releases. Click the version you want to download it.
- Extract the files under the directory you want.

3.3.2 How to install

There is no need for configure or setup.

3.4 How to compile and run the sample codes

We provide two samples: one for gravitational N-body simulation and the other for SPH. We first describe gravitational N-body simulation and then SPH. Sample codes do not use extensions.

3.4.1 Gravitational N-body simulation

3.4.1.1 Summary

Through the following steps one can use this sample.

- Move to the directory `$(FDPS)/sample/c++/nbody`. Here, `$(FDPS)` denotes the highest-level directory for FDPS (Note that `FDPS` is not an environmental variable). The actual value of `$(FDPS)` depends on the way you acquire the software. If you used the browser, the last part is "FDPS-master". If you used Subversion or Git, it is "trunk" or "FDPS", respectively.
- Edit `Makefile` in the current directory (`$(FDPS)/sample/c++/nbody`).
- Run the `make` command to create the executable `nbody.out`.
- Run `nbody.out`
- Check the output.

In addition, we describe the way to use Phantom-GRAPE for x86.

3.4.1.2 Move to the directory with the sample code

Move to `$(FDPS)/sample/c++/nbody`.

3.4.1.3 Edit Makefile

Edit Makefile following the description below. The changes depend on if you use OpenMP and/or MPI.

- Without both OpenMP and MPI

  - Set the variable `CC` the command to run your C++ compiler

- With OpenMP but not with MPI
  - Set the variable `CC` the command to run your C++ compiler
  - Uncomment the line `CFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`. If you use Intel compiler, replace `-fopenmp` by `-qopenmp` or `-openmp` depending on the version of the compiler.
- With MPI but not with OpenMP
  - Set the variable `CC` the command to run your MPI C++ compiler
  - Uncomment the line `CFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`
- With both OpenMP and MPI
  - Set the variable `CC` the command to run your MPI C++ compiler
  - Uncomment the line `CFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`. If you use Intel compiler, replace `-fopenmp` by `-qopenmp` or `-openmp` depending on the version of the compiler.
  - Uncomment the line `CFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`

3.4.1.4 Run make

Type "make" to run `make`.

3.4.1.5 Run the sample code

- If you are not using MPI, run the following in CLI (terminal)

```
$ ./nbody.out
```

- If you are using MPI, run the following in CLI (terminal)

```
$ MPIRUN -np NPROC ./nbody.out
```

  Here, `MPIRUN` should be `mpirun` or `mpiexec` depending on your MPI configuration, and `NPROC` is the number of processes you will use.

  Upon normal completion, the following output log should appear in stderr. The exact value of the energy error may depend on the system, but it is okay if its absolute value is of the order of 1×10^{-3}.

```
time:  9.6250000 energy error: -4.512836e-03
time:  9.7500000 energy error: -4.440746e-03
time:  9.8750000 energy error: -4.652358e-03
time: 10.0000000 energy error: -4.605855e-03
******** FDPS has successfully finished. ********
```

3.4.1.6 Analysis of the result

In the directory `result`, files "000x.dat" have been created. These files store the distribution of particles. Here, x is an integer (from 0 to 9) and it indicates time. The output file format is that in each line, index of particle, mass, position (x, y, z) and velocity (vx, vy, vz) are listed.

What is simulated with the default sample is the cold collapse of an uniform sphere with radius three expressed using 1024 particles. Using gnuplot, you can see the particle distribution in the xy plane at time=9:

```
$ gnuplot
$ plot "result/0009.dat" using 3:4
```

By plotting the particle distributions at other times, you can see how the initially uniform sphere contracts and then expands again. (Figure 1).

To increase the number of particles to 10000, try: (without MPI)

```
$ ./nbody.out -N 10000
```

3.4.1.7 To use Phantom-GRAPE for x86

If you are using a computer with Intel or AMD x86 CPU, you can use Phantom-GRAPE for x86.

Move to the directory `$(FDPS)/src/phantom_grape_x86/G5/newton/libpg5`, edit the Makefile there (if necessary), and run make to build the Phantom-GRAPE library `libpg5.a`.

Then go back to directory `$(FDPS)/sample/c++/nbody`, edit Makefile and remove "#" at the top of the line
"#use_phantom_grape_x86 = yes", and (after removing the existing executable) run make again. (Same for with and without OpenMP or MPI). You can run the executable in the same way as that for the executable without Phantom GRAPE.

The performance test on a machine with Intel Core i5-3210M CPU @2.50GHz (2 cores, 4 threads) indicates that, for N=8192, the code with Phantom GRAPE is faster than that without Phantom GRAPE by a factor a bit less than five. The following is the sample command line:

```
$ ./nbody.out -N 8192 -n 256
```

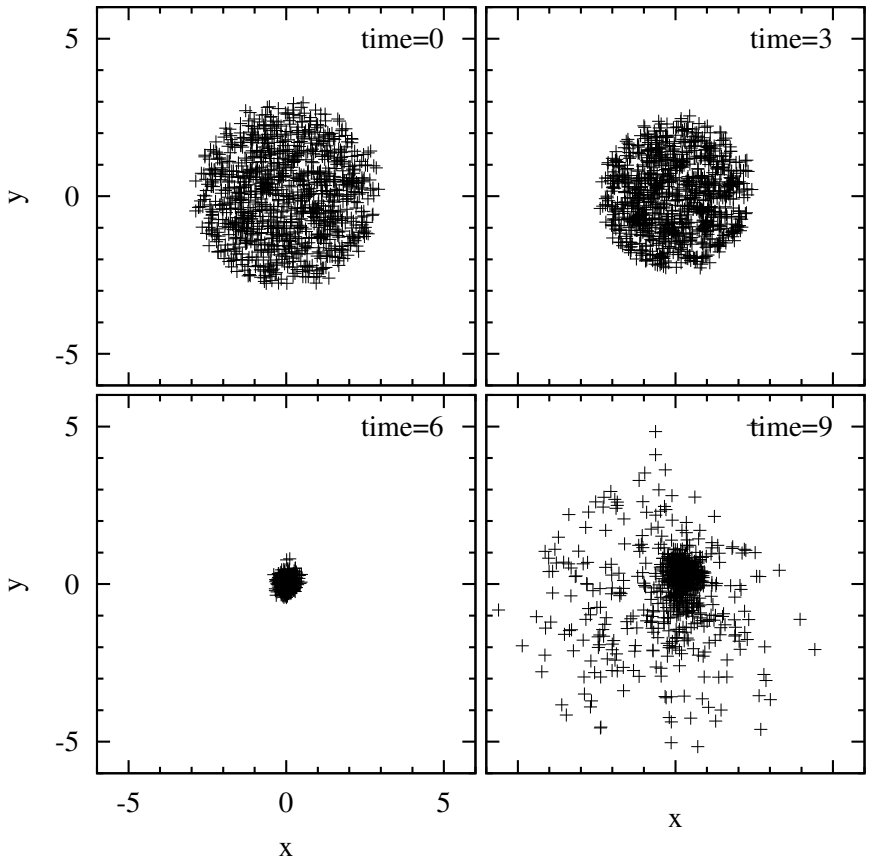

Figure 1:

3.4.1.8 To use NVIDIA GPUs

The sample program includes the interaction kernel written in Cuda for NVIDIA GPUs.

Uncomment the line "#use_cuda_gpu = yes" in file $(FDPS)/sample/c++/nbody/Makefile and assign to CUDA_HOME in Makefile a value approporiate to your environment. You can then run make to obtain the executable (OpenMP and MPI are also supported). The executable can be tested in the same way as the non-GPU version.

3.4.2 SPH simulation code

3.4.2.1 Summary

Through the following steps one can use this sample.

- Move to the directory `$(FDPS)/sample/c++/sph`.
- Edit Makefile in the current directory (`$(FDPS)/sample/c++/sph`).
- Run make command to create the executable `sph.out`.
- Run `sph.out`.
- Check the output.

3.4.2.2 Move to the directory with the sample code

Move to `$(FDPS)/sample/c++/sph`.

3.4.2.3 Edit Makefile

Edit Makefile following the same description described in § 3.4.1.3.

3.4.2.4 Run make

Type "make" to run `make`.

3.4.2.5 Run the sample code

- If you are not using MPI, run the following in CLI (terminal)

```
$ ./sph.out
```

- If you are using MPI, run the following in CLI (terminal)

```
$ MPIRUN -np NPROC ./sph.out
```

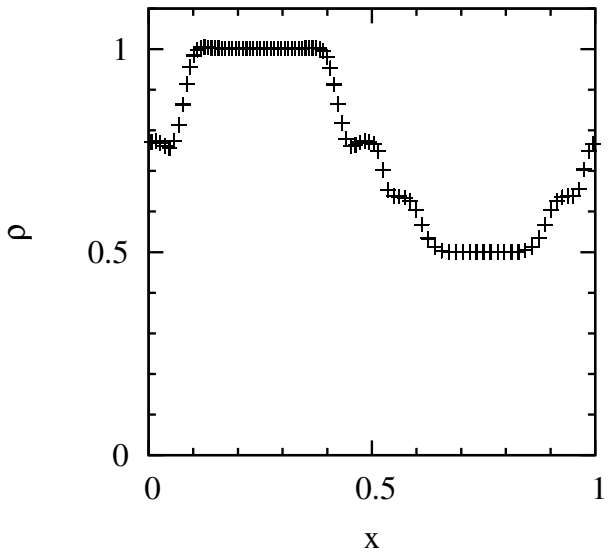

Figure 2:

Here, `MPIRUN` should be `mpirun` or `mpiexec` depending on your MPI configuration, and `NPROC` is the number of processes you will use.

Upon normal completion, the following output log should appear in stderr.

```
******** FDPS has successfully finished. ********
```

3.4.2.6 Analysis of the result

In the directory `result`, files "000x.txt" have been created. These files store the distribution of particles. Here, x is an integer (from 0 to 9) and it indicates time. The output file format is that in each line, index of particle, mass, position (x, y, z), velocity (vx, vy, vz), density, internal energy and pressure are listed.

What is simulated is the three-dimensional shock-tube problem. Using gnuplot, you can see the plot of the x-coordinate and density of particles at time=40:

```
$ gnuplot
$ plot "result/0040.txt" using 3:9
```

When the sample worked correctly, a figure similar to Figure 2 should appear.

4 How to Use

In this section, we explain in detail the contents of the sample codes shown in previous section (§ 3). Especially, we focus on classes that need to be defined by the users and how to use various APIs of FDPS.

4.1 Gravitational N-body simulation code

4.1.1 Working directory

We use `$(FDPS)/tutorial/nbody` as the working directory. First, change directory to there.

```
$ cd (FDPS)/tutorial/nbody
```

4.1.2 User-defined classes

In this section, we describe the classes which you need to define in order to perform gravitational N-body simulations using FDPS.

4.1.2.1 FullParticle type

You must define a FullParticle type. FullParticle type should contain all physical quantities necessary for an N-body simulation. Listing 1 shows the implementation of FullParticle type in our sample code (see `user-defined.hpp`). Note that FullParticle type is used as EssentialParticleI type, EssentialParticleJ type, and Force type in this sample code. FullParticle type must have member functions `copyfromFP()` and `copyFromForce()` to copy data. It should have member functions `getCharge()` (returns the particle mass), `getPos()` (returns the particle position), and `setPos()` (sets the particle position). In this code, we also define member functions `writeAscii()` and `readAscii()`, which are necessary to use file I/O functions of FDPS. The member function `clear()` is also necessary, which zero-clear the acceleration and potential.

Listing 1: FullParticle type

```
1  class FPGrav{
2  public:
3      PS::S64    id;
4      PS::F64    mass;
5      PS::F64vec pos;
6      PS::F64vec vel;
7      PS::F64vec acc;
8      PS::F64    pot;
9
10     static PS::F64 eps;
11
12     PS::F64vec getPos() const {
13         return pos;
14     }
15
16     PS::F64 getCharge() const {
```

```
17          return mass;
18      }
19
20      void copyFromFP(const FPGrav & fp){
21          mass = fp.mass;
22          pos  = fp.pos;
23      }
24
25      void copyFromForce(const FPGrav & force) {
26          acc = force.acc;
27          pot = force.pot;
28      }
29
30      void clear() {
31          acc = 0.0;
32          pot = 0.0;
33      }
34
35      void writeAscii(FILE* fp) const {
36          fprintf(fp, "%lld\t%g\t%g\t%g\t%g\t%g\t%g\t%g\n",
37                  this->id, this->mass,
38                  this->pos.x, this->pos.y, this->pos.z,
39                  this->vel.x, this->vel.y, this->vel.z);
40      }
41
42      void readAscii(FILE* fp) {
43          fscanf(fp, "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\n",
44                 &this->id, &this->mass,
45                 &this->pos.x, &this->pos.y, &this->pos.z,
46                 &this->vel.x, &this->vel.y, &this->vel.z);
47          }
48
49  };
```

4.1.2.2 calcForceEpEp

You must define a calcForceEpEp type. It should contain actual code for the calculation of Force. Listing 2 shows the implementation of calcForceEpEp type in our sample code for the case that the code is executed on CPUs without the Phantom-GRAPE library (see `user-defined.hpp`).

In this sample code, it is implemented as a template function. Its arguments are an array of EssentialParticleI type, the number of EssentialParticleI type variables, an array of EssentialParticleJ type, the number of EssentialParticleJ variables, and an array of Force type.

Listing 2: calcForceEpEp type

```
1  template <class TParticleJ>
2  void CalcGravity(const FPGrav * ep_i,
3                   const PS::S32 n_ip,
4                   const TParticleJ * ep_j,
5                   const PS::S32 n_jp,
6                   FPGrav * force) {
7      PS::F64 eps2 = FPGrav::eps * FPGrav::eps;
```

```
 8     for(PS::S32 i = 0; i < n_ip; i++){
 9         PS::F64vec xi = ep_i[i].getPos();
10         PS::F64vec ai = 0.0;
11         PS::F64 poti = 0.0;
12         for(PS::S32 j = 0; j < n_jp; j++){
13             PS::F64vec rij    = xi - ep_j[j].getPos();
14             PS::F64    r3_inv = rij * rij + eps2;
15             PS::F64    r_inv  = 1.0/sqrt(r3_inv);
16             r3_inv  = r_inv * r_inv;
17             r_inv  *= ep_j[j].getCharge();
18             r3_inv *= r_inv;
19             ai     -= r3_inv * rij;
20             poti   -= r_inv;
21         }
22         force[i].acc += ai;
23         force[i].pot += poti;
24     }
25 }
```

4.1.3 The main body of the user program

In this section, we describe the functions a user should write to implement gravitational N-body calculation using FDPS. The main function is described in the file `nbody.cpp`.

4.1.3.1 Initialization and termination of FDPS

You should first initialize FDPS by the following code.

Listing 3: Initialization of FDPS

```
1 PS::Initialize(argc, argv);
```

Once started, FDPS should be explicitly terminated. In the sample code, FDPS is terminated just before the termination of the program. To achieve this, you write the following code at the end of the main function.

Listing 4: Termination of FDPS

```
1 PS::Finalize();
```

4.1.3.2 Creation and initialization of FDPS objects

After the initialization of FDPS, a user need to create the objects used to talk to FDPS. In this section, we describe how to create and initialize these objects.

4.1.3.2.1 Creation of necessary FDPS objects

In an N-body simulation, one needs to create objects of ParticleSystem type, DomainInfo type, and TreeForForceLong type. The following is the code to create them (see the main function in `nbody.cpp`).

Listing 5: Creation of FDPS Objects

```
1 PS::DomainInfo dinfo;
2 PS::ParticleSystem<FPGrav> system_grav;
3 PS::TreeForForceLong<FPGrav, FPGrav, FPGrav>::Monopole tree_grav;
```

4.1.3.2.2 Initialization of the DomainInfo object

FDPS objects created by a user code should be initialized. Since the open boundary is used in this example, the initialization of a DomainInfo object is done simply by calling the `initialize` method:

Listing 6: Initialization of DomainInfo

```
1 const PS::F32 coef_ema = 0.3;
2 dinfo.initialize(coef_ema);
```

Note that the argument of `initialize` method represents a smoothing factor of an exponential moving average operation that is performed in the domain decomposition procedure. The definition of this factor is described in § 9.1.2 of the specification of FDPS.

4.1.3.2.3 Initialization of the ParticleSystem object

Next, we must initialize a ParticleSystem object. This is done by calling the `initialize` method without any function arguments:

Listing 7: Initialization of PaticleSystem

```
1 system_grav.initialize();
```

4.1.3.2.4 Initialization of the TreeForForceShort objects

Finally, we must initialize a TreeForForceLong object. The initialization of a TreeForForceLong object is done by calling the `initialize` method. This method should be given a rough number of particles. In this sample, we set the total number of particles `n_tot`:

Listing 8: Initialization of TreeForForceLong

```
1 tree_grav.initialize(n_tot , theta , n_leaf_limit , n_group_limit);
```

The `initialize` method has three optional arguments. Here, we pass these arguments explicitly. The meanings of these optional arguments are as follows (see also § 9.1.4 of the specification):

- `theta` — the so-called opening angle criterion for the tree method.
- `n_leaf_limit` — the upper limit for the number of particles in the leaf nodes.
- `n_group_limit` — the upper limit for the number of particles with which the particles use the same interaction list for the force calculation.

4.1.3.3 Time integration loop

In this section we describe the structure of the time integration loop.

4.1.3.3.1 Domain Decomposition

First, the computational domain is decomposed, using the current distribution of particles. In the sample, this is done by calling the `decomposeDomainAll` method of the DomainInfo class:

Listing 9: Domain Decomposition

```
1 if (n_loop % 4 == 0){
2     dinfo.decomposeDomainAll(system_grav);
3 }
```

In this sample code, we perform domain decomposition once in 4 main loops in order to reduce the computational cost.

4.1.3.3.2 Particle Exchange

Then, particles are exchanged between processes so that they belong to the process for the domain of their coordinates. To do so, the following member function of the ParticleSystem class is called.

Listing 10: Particle Exchange

```
1 system_grav.exchangeParticle(dinfo);
```

4.1.3.3.3 Interaction Calculation

After the domain decomposition and particle exchange, an interaction calculation is done. To do so, the following member functions of the TreeForForceLong class is called.

Listing 11: Interaction Calculation

```
1 tree_grav.calcForceAllAndWriteBack(CalcGravity<FPGrav>,
2                                    CalcGravity<PS::SPJMonopole>,
3                                    system_grav,
4                                    dinfo);
```

Note that the content of the description `<...>` in the arguments of this method represents a template argument.

4.1.3.3.4 Time Integration

In this sample code, we use the Leapfrog method to integrate the particle system in time. In this method, the time evolution operator can be expressed as $K(\frac{\Delta t}{2})D(\Delta t)K(\frac{\Delta t}{2})$, where Δt is the timestep, $K(\Delta t)$ is the 'kick' operator that integrates the velocities of particles from t to $t + \Delta t$, $D(\Delta t)$ is the 'drift' operator that integrates the positions of particles from t to $t + \Delta t$ (e.g. see Springel [2005,MNRAS,364,1105]). In the sample code, these operators are implemented as the functions `kick` and `drift`.

At the beginning of the main loop, the positions and the velocities of the particles are updated by the operator $D(\Delta t)K(\frac{\Delta t}{2})$:

Listing 12: Calculation of $D(\Delta t)K(\frac{\Delta t}{2})$ operator

```
1 kick(system_grav, dt * 0.5);
2 drift(system_grav, dt);
```

After the force calculation, the velocities of the particles are updated by the operator $K(\frac{\Delta t}{2})$:

Listing 13: Calculation of $K(\frac{\Delta t}{2})$ operator

```
1 kick(system_grav, dt * 0.5);
```

4.1.4 Diagnostic output

After the calculation started correctly, the time, the total energy of the system and the energy error are written to the standard error output. The following is the example of the output of the first step.

Listing 14: Standard error output

```
1 time:  0.0000000 energy: -1.974890e-01 energy error: +0.000000e+00
```

4.2 SPH simulation with fixed smoothing length

In this section, we describe how to implement the standard SPH scheme with a fixed smoothing length using FDPS. In the code discussed in this section, the initial condition for the 3D shock tube problem is generated and integrated.

4.2.1 Working directory

We use `$(FDPS)/sample/c++/sph` as the working directory. First, change directory to there.

```
$ cd (FDPS)/sample/c++/sph
```

4.2.2 Specifying include files

Since FDPS is realized as header files, you can use all functionalities of FDPS by including `particle_simulator.hpp` to your source program.

Listing 15: Include FDPS

```
1 #include <particle_simulator.hpp>
```

4.2.3 User-defined classes

In this section, we describe the classes which you need to define in order to perform SPH simulations using FDPS.

4.2.3.1 FullParticle type

You must define a FullParticle type. FullParticle type must contain all physical quantities that a SPH particle should have in order to perform a SPH simulation. It also must have a member function `copyFromForce` to copy the results from the Force type (explained later). It should have member functions `getCharge()` (returns the particle mass), `getPos()` (returns the particle position), `getRSearch()` (returns the search radius for neighbor particles), and `setPos()` (sets the position). In this sample code, we make use of file I/O functions of FDPS, which requires a user to define member functions `writeAscii()` and `readAscii()`. In addition to them, member function `setPressure()` is defined. This member function calculates the pressure from the equation of states. This function is not used by FDPS, but used within the user code. The following is the implementation of FullParticle type in the sample code.

Listing 16: FullParticle type

```
1  struct FP{
2     PS::F64 mass;
3     PS::F64vec pos;
4     PS::F64vec vel;
5     PS::F64vec acc;
6     PS::F64 dens;
7     PS::F64 eng;
8     PS::F64 pres;
9     PS::F64 smth;
10    PS::F64 snds;
11    PS::F64 eng_dot;
12    PS::F64 dt;
13    PS::S64 id;
14    PS::F64vec vel_half;
15    PS::F64 eng_half;
16    void copyFromForce(const Dens& dens){
17       this->dens = dens.dens;
18    }
19    void copyFromForce(const Hydro& force){
20       this->acc     = force.acc;
21       this->eng_dot = force.eng_dot;
22       this->dt      = force.dt;
23    }
24    PS::F64 getCharge() const{
25       return this->mass;
26    }
27    PS::F64vec getPos() const{
28       return this->pos;
29    }
30    PS::F64 getRSearch() const{
31       return kernelSupportRadius * this->smth;
32    }
33    void setPos(const PS::F64vec& pos){
34       this->pos = pos;
35    }
36    void writeAscii(FILE* fp) const{
37       fprintf(fp,
38               "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t"
```

```
39                "%lf\t%lf\t%lf\t%lf\t%lf\n",
40                this->id, this->mass,
41                this->pos.x, this->pos.y, this->pos.z,
42                this->vel.x, this->vel.y, this->vel.z,
43                this->dens, this->eng, this->pres);
44    }
45    void readAscii(FILE* fp){
46       fscanf(fp,
47              "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t"
48              "%lf\t%lf\t%lf\t%lf\t%lf\n",
49              &this->id, &this->mass,
50              &this->pos.x, &this->pos.y, &this->pos.z,
51              &this->vel.x, &this->vel.y, &this->vel.z,
52              &this->dens, &this->eng, &this->pres);
53    }
54    void setPressure(){
55       const PS::F64 hcr = 1.4;
56       pres = (hcr - 1.0) * dens * eng;
57       snds = sqrt(hcr * pres / dens);
58    }
59 };
```

4.2.3.2 EssentialParticleI type

You must define a EssentialParticleI type. It should have all information necessary for an i particle to do the calculation of Force. In this sample code, it is also used as EssentialParticleJ type. Therefore, it should have all information necessary for a j particle to do the calculation of Force. It should have member function `copyFromFP` to copy necessary quantities from FullParticle type described above. It should have member functions `getPos()`, `getRSearch()`, and `setPos()`. Their functions are the same as those for FullParticle type.

The following is the implementation of EssentialParticleI type in the sample code.

Listing 17: EssentialParticleI type

```
1  struct EP{
2     PS::F64vec pos;
3     PS::F64vec vel;
4     PS::F64    mass;
5     PS::F64    smth;
6     PS::F64    dens;
7     PS::F64    pres;
8     PS::F64    snds;
9     void copyFromFP(const FP& rp){
10       this->pos  = rp.pos;
11       this->vel  = rp.vel;
12       this->mass = rp.mass;
13       this->smth = rp.smth;
14       this->dens = rp.dens;
15       this->pres = rp.pres;
16       this->snds = rp.snds;
17    }
18    PS::F64vec getPos() const{
19       return this->pos;
```

```
20     }
21     PS::F64 getRSearch() const{
22        return kernelSupportRadius * this->smth;
23     }
24     void setPos(const PS::F64vec& pos){
25        this->pos = pos;
26     }
27  };
```

4.2.3.3 Force type

You must define a Force type. It must have member variables that store the result of the Force calculation. In this sample code, there are two types of Force calculations, one for density and the other for actual hydrodynamic interaction. Thus, two Force types should be defined. A Force type should have member function `clear()`, which zero-clears or initializes member variables that store the result of some accumulation operation. The following is the implementation of Force types in the sample code.

In this sample, the `Dens` class has a member variable `smth` that stands for the smoothing length of a SPH particle, which is actually unnecessary for a SPH simulation with a *fixed* smoothing length. However, we leave it in the sample code because it would be useful for a user to extend this sample code to a SPH simulation code with *variable* smoothing length. In the formulation by Springel [2005,MNRAS,364,1105] (one of the most popular formulation of SPH with variable smoothing length), it is required to calculate the mass density and the smoothing length simultaneously. If you adopt this formulation, you need to let Force type have a member variable that represents smoothing length as in this sample code. The member function `clear` in the `Dens` class does not zero-clear `smth` because this sample code assume a fixed smoothing length (the density calculation will fail if `smth` is zero-cleared!).

The `Hydro` class has a member variable `dt` that stands for a timestep of each particle. In this sample, `dt` is not zero-cleared because `dt` is not a quantity that stores the result of some accumulation operation and therefore zero-clear is unnecessary.

Listing 18: Force type

```
1  class Dens{
2     public:
3     PS::F64 dens;
4     PS::F64 smth;
5     void clear(){
6        dens = 0;
7     }
8  };
9  class Hydro{
10    public:
11    PS::F64vec acc;
12    PS::F64 eng_dot;
13    PS::F64 dt;
14    void clear(){
15       acc = 0;
16       eng_dot = 0;
17    }
18 };
```

4.2.3.4 calcForceEpEp type

You must define a calcForceEpEp type. It should contain actual code for the calculation of Force. In the sample code, it is implemented using Functor (function object). The arguments of the Functor are an array of EssentialParticleI type, the number of EssentialParticleI type variables, an array of EssentialParticleJ type, the number of EssentialParticleJ variables, an array of Force type. As described above, two Force classes, one for density and the other for actual hydrodynamic interaction, are used in this code. Thus, two calcForceEpEp types should be defined.

The following is the implementation of calcForceEpEp types in the sample code.

Listing 19: calcForceEpEp type

```
1  class CalcDensity{
2     public:
3     void operator () (const EP* const ep_i, const PS::S32 Nip,
4                       const EP* const ep_j, const PS::S32 Njp,
5                       Dens* const dens){
6        for(PS::S32 i = 0 ; i < Nip ; ++i){
7           dens[i].clear();
8           for(PS::S32 j = 0 ; j < Njp ; ++j){
9              const PS::F64vec dr = ep_j[j].pos - ep_i[i].pos;
10             dens[i].dens += ep_j[j].mass * W(dr, ep_i[i].smth);
11          }
12       }
13    }
14 };
15
16 class CalcHydroForce{
17    public:
18    void operator () (const EP* const ep_i, const PS::S32 Nip,
19                      const EP* const ep_j, const PS::S32 Njp,
20                      Hydro* const hydro){
21       for(PS::S32 i = 0; i < Nip ; ++ i){
22          hydro[i].clear();
23          PS::F64 v_sig_max = 0.0;
24          for(PS::S32 j = 0; j < Njp ; ++j){
25             const PS::F64vec dr = ep_i[i].pos - ep_j[j].pos;
26             const PS::F64vec dv = ep_i[i].vel - ep_j[j].vel;
27             const PS::F64 w_ij = (dv * dr < 0) ? dv * dr / sqrt(dr * dr) :
                      0;
28             const PS::F64 v_sig = ep_i[i].snds + ep_j[j].snds - 3.0 * w_ij
                      ;
29             v_sig_max = std::max(v_sig_max, v_sig);
30             const PS::F64 AV = - 0.5 * v_sig * w_ij / (0.5 * (ep_i[i].dens
                       + ep_j[j].dens));
31             const PS::F64vec gradW_ij = 0.5 * (gradW(dr, ep_i[i].smth) +
                      gradW(dr, ep_j[j].smth));
32             hydro[i].acc     -= ep_j[j].mass * (ep_i[i].pres / (ep_i[i].
                      dens * ep_i[i].dens) + ep_j[j].pres / (ep_j[j].dens *
                      ep_j[j].dens) + AV) * gradW_ij;
33             hydro[i].eng_dot += ep_j[j].mass * (ep_i[i].pres / (ep_i[i].
                      dens * ep_i[i].dens) + 0.5 * AV) * dv * gradW_ij;
34          }
35          hydro[i].dt = C_CFL * 2.0 * ep_i[i].smth / v_sig_max;
```

```
36         }
37     }
38 };
```

4.2.4 The main body of the user program

In this section, we describe the functions a user should write to implement SPH calculation using FDPS.

4.2.4.1 Initialization and termination of FDPS

You should first initialize FDPS by the following code.

Listing 20: Initialization of FDPS

```
1 PS::Initialize(argc, argv);
```

Once started, FDPS should be explicitly terminated. In this sample, FDPS is terminated just before the termination of the program. To achieve this, you write the following code at the end of the main function.

Listing 21: Termination of FDPS

```
1 PS::Finalize();
```

4.2.4.2 Creation and initialization of FDPS objects

After the initialization of FDPS, a user need to create the objects used to talk to FDPS. In this section we describe how to create and initialize these objects.

4.2.4.2.1 Creation of necessary FDPS objects

In an SPH simulation code, one needs to create objects of ParticleSystem type, DomainInfo type, and TreeForForceShort type (for density calculation using gather type interaction), and one more object of TreeForForceShort type (for hydrodynamic interaction calculation using symmetric type interaction).

The following is the code to create them.

Listing 22: Creation of FDPS Objects

```
1 PS::ParticleSystem<FP> sph_system;
2 PS::DomainInfo dinfo;
3 PS::TreeForForceShort<Dens, EP, EP>::Gather dens_tree;
4 PS::TreeForForceShort<Hydro, EP, EP>::Symmetry hydr_tree;
```

4.2.4.2.2 Initialization of the DomainInfo object

FDPS objects created by a user code should be initialized. Here, we describe the necessary procedures required to initialize a DomainInfo object. First, we need to call the `initialize` method. Then, the type of the boundary and the size of the simulation box should be

set because we do not use the open boundary (FDPS adopts the open boundary as the default boundary condition). In this code, we use the periodic boundary for all of x, y and z directions.

Listing 23: Initialization of DomainInfo

```
1 dinfo.initialize();
2 dinfo.setBoundaryCondition(PS::BOUNDARY_CONDITION_PERIODIC_XYZ);
3 dinfo.setPosRootDomain(PS::F64vec(0.0, 0.0, 0.0),
4                        PS::F64vec(box.x, box.y, box.z));
```

4.2.4.2.3 Initialization of the ParticleSystem object

Next, we need to initialize a ParticleSystem object. This is done by the following single line of code:

Listing 24: Initialization of PaticleSystem

```
1 sph_system.initialize();
```

4.2.4.2.4 Initialization of the TreeForForceShort objects

Finally, TreeForForceShort objects should be initialized. This is done by calling the `initialize` method. This function should be given the rough number of particles. In this sample, we set three times the total number of particles:

Listing 25: Initialization of TreeForForceShort

```
1 dens_tree.initialize(3 * sph_system.getNumberOfParticleGlobal());
2 hydr_tree.initialize(3 * sph_system.getNumberOfParticleGlobal());
```

4.2.4.3 Time integration loop

In this section we describe the structure of the time integration loop.

4.2.4.3.1 Domain Decomposition

First, the computational domain is decomposed, using the current distribution of particles. To do so, the following member function of the DomainInfo class is called.

Listing 26: Domain Decomposition

```
1 dinfo.decomposeDomainAll(sph_system);
```

4.2.4.3.2 Particle Exchange

Then particles are exchanged between processes so that they belong to the process for the domain of their coordinates. To do so, the following member function of the ParticleSystem class is called.

Listing 27: Particle Exchange

```
1 sph_system.exchangeParticle(dinfo);
```

4.2.4.3.3 Interaction Calculation

After the domain decomposition and particle exchange, interaction calculation is done. To do so, the following member functions of the TreeForForceShorts class are called.

Listing 28: Interaction Calculation

```
1 dens_tree.calcForceAllAndWriteBack(CalcDensity(), sph_system, dinfo);
2 hydr_tree.calcForceAllAndWriteBack(CalcHydroForce(), sph_system, dinfo);
```

4.2.5 Compilation of the program

Run `make` at the working directory. You can use the Makefile attached to the sample code.

```
$ make
```

4.2.6 Execution

To run the code without MPI, you should execute the following command in the command shell.

```
$ ./sph.out
```

To run the code using MPI, you should execute the following command in the command shell, or follow the document of your system.

```
$ MPIRUN -np NPROC ./sph.out
```

Here,"MPIRUN" represents the command to run your program using MPI such as mpirun or mpiexec, and "NPROC" is the number of MPI processes.

4.2.7 Log and output files

Log and output files are created under `result` directory.

4.2.8 Visualization

In this section, we describe how to visualize the calculation result using gnuplot. To enter the interactive mode of gnuplot, execute the following command.

```
$ gnuplot
```

In the interactive mode, you can visualize the result. In the following example, using the 40th snapshot file, we create the plot in which the abscissa is the x coordinate of particles and the ordinate is the density of particles.

```
gnuplot> plot "result/0040.txt" u 3:9
```

5 Sample Codes

5.1 SPH simulation with fixed smoothing length

In this section, we show a sample code for the SPH simulation with fixed smoothing length. This code is the same as what we described in section 4. One can create a working code by cut and paste this code and compile and link the resulted source program.

Listing 29: Sample code of SPH simulation

```
1  // Include FDPS header
2  #include <particle_simulator.hpp>
3  // Include the standard C++ headers
4  #include <cmath>
5  #include <cstdio>
6  #include <iostream>
7  #include <vector>
8  #include <sys/stat.h>
9
10 /* Parameters */
11 const short int Dim = 3;
12 const PS::F64 SMTH = 1.2;
13 const PS::U32 OUTPUT_INTERVAL = 10;
14 const PS::F64 C_CFL = 0.3;
15
16 /* Kernel Function */
17 const PS::F64 pi = atan(1.0) * 4.0;
18 const PS::F64 kernelSupportRadius = 2.5;
19
20 PS::F64 W(const PS::F64vec dr, const PS::F64 h){
21    const PS::F64 H = kernelSupportRadius * h;
22    const PS::F64 s = sqrt(dr * dr) / H;
23    const PS::F64 s1 = (1.0 - s < 0) ? 0 : 1.0 - s;
24    const PS::F64 s2 = (0.5 - s < 0) ? 0 : 0.5 - s;
25    PS::F64 r_value = pow(s1, 3) - 4.0 * pow(s2, 3);
26    //if # of dimension == 3
27    r_value *= 16.0 / pi / (H * H * H);
28    return r_value;
29 }
30
31 PS::F64vec gradW(const PS::F64vec dr, const PS::F64 h){
32    const PS::F64 H = kernelSupportRadius * h;
33    const PS::F64 s = sqrt(dr * dr) / H;
34    const PS::F64 s1 = (1.0 - s < 0) ? 0 : 1.0 - s;
35    const PS::F64 s2 = (0.5 - s < 0) ? 0 : 0.5 - s;
36    PS::F64 r_value = - 3.0 * pow(s1, 2) + 12.0 * pow(s2, 2);
37    //if # of dimension == 3
38    r_value *= 16.0 / pi / (H * H * H);
39    return dr * r_value / (sqrt(dr * dr) * H + 1.0e-6 * h);
40 }
41
42 /* Class Definitions */
43 //** Force Class (Result Class)
44 class Dens{
45    public:
```

```
46     PS::F64 dens;
47     PS::F64 smth;
48     void clear(){
49        dens = 0;
50     }
51  };
52  class Hydro{
53     public:
54     PS::F64vec acc;
55     PS::F64 eng_dot;
56     PS::F64 dt;
57     void clear(){
58        acc = 0;
59        eng_dot = 0;
60     }
61  };
62
63  //** Full Particle Class
64  struct FP{
65     PS::F64 mass;
66     PS::F64vec pos;
67     PS::F64vec vel;
68     PS::F64vec acc;
69     PS::F64 dens;
70     PS::F64 eng;
71     PS::F64 pres;
72     PS::F64 smth;
73     PS::F64 snds;
74     PS::F64 eng_dot;
75     PS::F64 dt;
76     PS::S64 id;
77     PS::F64vec vel_half;
78     PS::F64 eng_half;
79     void copyFromForce(const Dens& dens){
80        this->dens = dens.dens;
81     }
82     void copyFromForce(const Hydro& force){
83        this->acc     = force.acc;
84        this->eng_dot = force.eng_dot;
85        this->dt      = force.dt;
86     }
87     PS::F64 getCharge() const{
88        return this->mass;
89     }
90     PS::F64vec getPos() const{
91        return this->pos;
92     }
93     PS::F64 getRSearch() const{
94        return kernelSupportRadius * this->smth;
95     }
96     void setPos(const PS::F64vec& pos){
97        this->pos = pos;
98     }
99     void writeAscii(FILE* fp) const{
100       fprintf(fp,
```

```
101                 "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t"
102                 "%lf\t%lf\t%lf\t%lf\t%lf\n",
103                 this->id, this->mass,
104                 this->pos.x, this->pos.y, this->pos.z,
105                 this->vel.x, this->vel.y, this->vel.z,
106                 this->dens, this->eng, this->pres);
107     }
108     void readAscii(FILE* fp){
109        fscanf(fp,
110               "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t"
111               "%lf\t%lf\t%lf\t%lf\t%lf\n",
112               &this->id, &this->mass,
113               &this->pos.x, &this->pos.y, &this->pos.z,
114               &this->vel.x, &this->vel.y, &this->vel.z,
115               &this->dens, &this->eng, &this->pres);
116     }
117     void setPressure(){
118        const PS::F64 hcr = 1.4;
119        pres = (hcr - 1.0) * dens * eng;
120        snds = sqrt(hcr * pres / dens);
121     }
122 };
123
124 //** Essential Particle Class
125 struct EP{
126     PS::F64vec pos;
127     PS::F64vec vel;
128     PS::F64    mass;
129     PS::F64    smth;
130     PS::F64    dens;
131     PS::F64    pres;
132     PS::F64    snds;
133     void copyFromFP(const FP& rp){
134        this->pos  = rp.pos;
135        this->vel  = rp.vel;
136        this->mass = rp.mass;
137        this->smth = rp.smth;
138        this->dens = rp.dens;
139        this->pres = rp.pres;
140        this->snds = rp.snds;
141     }
142     PS::F64vec getPos() const{
143        return this->pos;
144     }
145     PS::F64 getRSearch() const{
146        return kernelSupportRadius * this->smth;
147     }
148     void setPos(const PS::F64vec& pos){
149        this->pos = pos;
150     }
151 };
152
153 class FileHeader{
154     public:
155     PS::S32 Nbody;
```

```
156     PS::F64 time;
157     int readAscii(FILE* fp){
158        fscanf(fp, "%lf\n", &time);
159        fscanf(fp, "%d\n", &Nbody);
160        return Nbody;
161     }
162     void writeAscii(FILE* fp) const{
163        fprintf(fp, "%e\n", time);
164        fprintf(fp, "%d\n", Nbody);
165     }
166 };
167
168 struct boundary{
169     PS::F64 x, y, z;
170 };
171
172
173 /* Force Functors */
174 class CalcDensity{
175     public:
176     void operator () (const EP* const ep_i, const PS::S32 Nip,
177                       const EP* const ep_j, const PS::S32 Njp,
178                       Dens* const dens){
179        for(PS::S32 i = 0 ; i < Nip ; ++i){
180           dens[i].clear();
181           for(PS::S32 j = 0 ; j < Njp ; ++j){
182              const PS::F64vec dr = ep_j[j].pos - ep_i[i].pos;
183              dens[i].dens += ep_j[j].mass * W(dr, ep_i[i].smth);
184           }
185        }
186     }
187 };
188
189 class CalcHydroForce{
190     public:
191     void operator () (const EP* const ep_i, const PS::S32 Nip,
192                       const EP* const ep_j, const PS::S32 Njp,
193                       Hydro* const hydro){
194        for(PS::S32 i = 0; i < Nip ; ++ i){
195           hydro[i].clear();
196           PS::F64 v_sig_max = 0.0;
197           for(PS::S32 j = 0; j < Njp ; ++j){
198              const PS::F64vec dr = ep_i[i].pos - ep_j[j].pos;
199              const PS::F64vec dv = ep_i[i].vel - ep_j[j].vel;
200              const PS::F64 w_ij = (dv * dr < 0) ? dv * dr / sqrt(dr * dr) :
                        0;
201              const PS::F64 v_sig = ep_i[i].snds + ep_j[j].snds - 3.0 * w_ij
                        ;
202              v_sig_max = std::max(v_sig_max, v_sig);
203              const PS::F64 AV = - 0.5 * v_sig * w_ij / (0.5 * (ep_i[i].dens
                         + ep_j[j].dens));
204              const PS::F64vec gradW_ij = 0.5 * (gradW(dr, ep_i[i].smth) +
                       gradW(dr, ep_j[j].smth));
205              hydro[i].acc     -= ep_j[j].mass * (ep_i[i].pres / (ep_i[i].
                       dens * ep_i[i].dens) + ep_j[j].pres / (ep_j[j].dens *
```

```
                    ep_j[j].dens) + AV) * gradW_ij;
206             hydro[i].eng_dot += ep_j[j].mass * (ep_i[i].pres / (ep_i[i].
                    dens * ep_i[i].dens) + 0.5 * AV) * dv * gradW_ij;
207         }
208         hydro[i].dt = C_CFL * 2.0 * ep_i[i].smth / v_sig_max;
209       }
210    }
211 };
212
213 void makeOutputDirectory(char * dir_name) {
214     struct stat st;
215     PS::S32 ret;
216     if (PS::Comm::getRank() == 0) {
217         if (stat(dir_name, &st) != 0) {
218             ret = mkdir(dir_name, 0777);
219         } else {
220             ret = 0; // the directory named dir_name already exists.
221         }
222     }
223     PS::Comm::broadcast(&ret, 1);
224     if (ret == 0) {
225         if (PS::Comm::getRank() == 0)
226             fprintf(stderr, "Directory␣\"%s\"␣is␣successfully␣made.\n",
                    dir_name);
227     } else {
228         if (PS::Comm::getRank() == 0)
229             fprintf(stderr, "Directory␣%s␣fails␣to␣be␣made.\n", dir_name);
230         PS::Abort();
231     }
232 }
233
234 void SetupIC(PS::ParticleSystem<FP>& sph_system, PS::F64 *end_time,
        boundary *box){
235    // Place SPH particles
236    std::vector<FP> ptcl;
237    const PS::F64 dx = 1.0 / 128.0;
238    box->x = 1.0;
239    box->y = box->z = box->x / 8.0;
240    PS::S32 i = 0;
241    for(PS::F64 x = 0 ; x < box->x * 0.5 ; x += dx){
242       for(PS::F64 y = 0 ; y < box->y ; y += dx){
243          for(PS::F64 z = 0 ; z < box->z ; z += dx){
244             FP ith;
245             ith.pos.x = x;
246             ith.pos.y = y;
247             ith.pos.z = z;
248             ith.dens = 1.0;
249             ith.mass = 0.75;
250             ith.eng  = 2.5;
251             ith.id   = i++;
252             ith.smth = 0.012;
253             ptcl.push_back(ith);
254          }
255       }
256    }
```

```
257     for(PS::F64 x = box->x * 0.5 ; x < box->x * 1.0 ; x += dx * 2.0){
258        for(PS::F64 y = 0 ; y < box->y ; y += dx){
259           for(PS::F64 z = 0 ; z < box->z ; z += dx){
260              FP ith;
261              ith.pos.x = x;
262              ith.pos.y = y;
263              ith.pos.z = z;
264              ith.dens = 0.5;
265              ith.mass = 0.75;
266              ith.eng  = 2.5;
267              ith.id   = i++;
268              ith.smth = 0.012;
269              ptcl.push_back(ith);
270           }
271        }
272     }
273     for(PS::U32 i = 0 ; i < ptcl.size() ; ++ i){
274        ptcl[i].mass = ptcl[i].mass * box->x * box->y * box->z / (PS::F64)(
                ptcl.size());
275     }
276     std::cout << "# of ptcls is... " << ptcl.size() << std::endl;
277     // Scatter SPH particles
278     assert(ptcl.size() % PS::Comm::getNumberOfProc() == 0);
279     const PS::S32 numPtclLocal = ptcl.size() / PS::Comm::getNumberOfProc();
280     sph_system.setNumberOfParticleLocal(numPtclLocal);
281     const PS::U32 i_head = numPtclLocal * PS::Comm::getRank();
282     const PS::U32 i_tail = numPtclLocal * (PS::Comm::getRank() + 1);
283     for(PS::U32 i = 0 ; i < ptcl.size() ; ++ i){
284        if(i_head <= i && i < i_tail){
285           const PS::U32 ii = i - numPtclLocal * PS::Comm::getRank();
286           sph_system[ii] = ptcl[i];
287        }
288     }
289     // Set the end time
290     *end_time = 0.12;
291     // Fin.
292     std::cout << "setup..." << std::endl;
293 }
294
295 void Initialize(PS::ParticleSystem<FP>& sph_system){
296     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
297        sph_system[i].setPressure();
298     }
299 }
300
301 PS::F64 getTimeStepGlobal(const PS::ParticleSystem<FP>& sph_system){
302     PS::F64 dt = 1.0e+30; //set VERY LARGE VALUE
303     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
304        dt = std::min(dt, sph_system[i].dt);
305     }
306     return PS::Comm::getMinValue(dt);
307 }
308
309 void InitialKick(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
310     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
```

```
311         sph_system[i].vel_half = sph_system[i].vel + 0.5 * dt * sph_system[i
                ].acc;
312         sph_system[i].eng_half = sph_system[i].eng + 0.5 * dt * sph_system[i
                ].eng_dot;
313     }
314 }
315
316 void FullDrift(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
317     // time becomes t + dt;
318     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
319         sph_system[i].pos += dt * sph_system[i].vel_half;
320     }
321 }
322
323 void Predict(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
324     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
325         sph_system[i].vel += dt * sph_system[i].acc;
326         sph_system[i].eng += dt * sph_system[i].eng_dot;
327     }
328 }
329
330 void FinalKick(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
331     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
332         sph_system[i].vel = sph_system[i].vel_half + 0.5 * dt * sph_system[i
                ].acc;
333         sph_system[i].eng = sph_system[i].eng_half + 0.5 * dt * sph_system[i
                ].eng_dot;
334     }
335 }
336
337 void setPressure(PS::ParticleSystem<FP>& sph_system){
338     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
339         sph_system[i].setPressure();
340     }
341 }
342
343 void CheckConservativeVariables(const PS::ParticleSystem<FP>& sph_system){
344     PS::F64vec Mom=0.0; // total momentum
345     PS::F64    Eng=0.0; // total enegry
346     for(PS::S32 i = 0; i < sph_system.getNumberOfParticleLocal(); ++ i){
347         Mom += sph_system[i].vel * sph_system[i].mass;
348         Eng += (sph_system[i].eng + 0.5 * sph_system[i].vel * sph_system[i].
                vel)
349               * sph_system[i].mass;
350     }
351     Eng = PS::Comm::getSum(Eng);
352     Mom = PS::Comm::getSum(Mom);
353      if(PS::Comm::getRank() == 0){
354          printf("%.16e\n", Eng);
355          printf("%.16e\n", Mom.x);
356          printf("%.16e\n", Mom.y);
357          printf("%.16e\n", Mom.z);
358      }
359 }
360
```

```
361 int main(int argc, char* argv[]){
362    // Initialize FDPS
363    PS::Initialize(argc, argv);
364    // Make a directory
365    char dir_name[1024];
366    sprintf(dir_name,"./result");
367    makeOutputDirectory(dir_name);
368    // Display # of MPI processes and threads
369    PS::S32 nprocs = PS::Comm::getNumberOfProc();
370    PS::S32 nthrds = PS::Comm::getNumberOfThread();
371    std::cout << "=========================================" << std::endl
372              << "␣This␣is␣a␣sample␣program␣of␣"             << std::endl
373              << "␣Smoothed␣Particle␣Hydrodynamics␣on␣FDPS!"  << std::endl
374              << "␣#␣of␣processes␣is␣" << nprocs              << std::endl
375              << "␣#␣of␣thread␣is␣␣␣␣" << nthrds              << std::endl
376              << "=========================================" << std::endl
                    ;
377    // Make an instance of ParticleSystem and initialize it
378    PS::ParticleSystem<FP> sph_system;
379    sph_system.initialize();
380    // Define local variables
381    PS::F64 dt, end_time;
382    boundary box;
383    // Make an initial condition and initialize the particle system
384    SetupIC(sph_system, &end_time, &box);
385    Initialize(sph_system);
386    // Make an instance of DomainInfo and initialize it
387    PS::DomainInfo dinfo;
388    dinfo.initialize();
389    // Set the boundary condition
390    dinfo.setBoundaryCondition(PS::BOUNDARY_CONDITION_PERIODIC_XYZ);
391    dinfo.setPosRootDomain(PS::F64vec(0.0, 0.0, 0.0),
392                           PS::F64vec(box.x, box.y, box.z));
393    // Perform domain decomposition
394    dinfo.decomposeDomainAll(sph_system);
395    // Exchange the SPH particles between the (MPI) processes
396    sph_system.exchangeParticle(dinfo);
397    // Make two tree structures
398    // (one is for the density calculation and
399    //  another is for the force calculation.)
400    PS::TreeForForceShort<Dens, EP, EP>::Gather dens_tree;
401    dens_tree.initialize(3 * sph_system.getNumberOfParticleGlobal());
402
403    PS::TreeForForceShort<Hydro, EP, EP>::Symmetry hydr_tree;
404    hydr_tree.initialize(3 * sph_system.getNumberOfParticleGlobal());
405    // Compute density, pressure, acceleration due to pressure gradient
406    dens_tree.calcForceAllAndWriteBack(CalcDensity(), sph_system, dinfo);
407    setPressure(sph_system);
408    hydr_tree.calcForceAllAndWriteBack(CalcHydroForce(), sph_system, dinfo)
           ;
409    // Get timestep
410    dt = getTimeStepGlobal(sph_system);
411    // Main loop for time integration
412    PS::S32 step = 0;
413    for(PS::F64 time = 0 ; time < end_time ; time += dt, ++ step){
```

```
414         // Leap frog: Initial Kick & Full Drift
415         InitialKick(sph_system, dt);
416         FullDrift(sph_system, dt);
417         // Adjust the positions of the SPH particles that run over
418         // the computational boundaries.
419         sph_system.adjustPositionIntoRootDomain(dinfo);
420         // Leap frog: Predict
421         Predict(sph_system, dt);
422         // Perform domain decomposition again
423         dinfo.decomposeDomainAll(sph_system);
424         // Exchange the SPH particles between the (MPI) processes
425         sph_system.exchangeParticle(dinfo);
426         // Compute density, pressure, acceleration due to pressure gradient
427         dens_tree.calcForceAllAndWriteBack(CalcDensity(), sph_system, dinfo)
                ;
428         setPressure(sph_system);
429         hydr_tree.calcForceAllAndWriteBack(CalcHydroForce(), sph_system,
                dinfo);
430         // Get a new timestep
431         dt = getTimeStepGlobal(sph_system);
432         // Leap frog: Final Kick
433         FinalKick(sph_system, dt);
434         // Output result files
435         if(step % OUTPUT_INTERVAL == 0){
436            FileHeader header;
437            header.time = time;
438            header.Nbody = sph_system.getNumberOfParticleGlobal();
439            char filename[256];
440            sprintf(filename, "result/%04d.txt", step);
441            sph_system.writeParticleAscii(filename, header);
442            if (PS::Comm::getRank() == 0){
443               std::cout << "================================" << std::endl;
444               std::cout << "output␣" << filename << "." << std::endl;
445               std::cout << "================================" << std::endl;
446            }
447         }
448         // Output information to STDOUT
449         if (PS::Comm::getRank() == 0){
450            std::cout << "================================" << std::endl;
451            std::cout << "time␣=␣" << time << std::endl;
452            std::cout << "step␣=␣" << step << std::endl;
453            std::cout << "================================" << std::endl;
454         }
455         CheckConservativeVariables(sph_system);
456      }
457      // Finalize FDPS
458      PS::Finalize();
459      return 0;
460  }
```

5.2 N-body simulation

In this section, we show a sample code for the N-body simulation. This code is the same as what we described in section 4. One can create a working code by cut and paste this code

and compile and link the resulted source program.

Listing 30: Sample code of N-body simulation (`user-defined.hpp`)

```
1  #pragma once
2  class FileHeader{
3  public:
4      PS::S64 n_body;
5      PS::F64 time;
6      PS::S32 readAscii(FILE * fp) {
7          fscanf(fp, "%lf\n", &time);
8          fscanf(fp, "%lld\n", &n_body);
9          return n_body;
10     }
11     void writeAscii(FILE* fp) const {
12         fprintf(fp, "%e\n", time);
13         fprintf(fp, "%lld\n", n_body);
14     }
15 };
16
17 class FPGrav{
18 public:
19     PS::S64    id;
20     PS::F64    mass;
21     PS::F64vec pos;
22     PS::F64vec vel;
23     PS::F64vec acc;
24     PS::F64    pot;
25
26     static PS::F64 eps;
27
28     PS::F64vec getPos() const {
29         return pos;
30     }
31
32     PS::F64 getCharge() const {
33         return mass;
34     }
35
36     void copyFromFP(const FPGrav & fp){
37         mass = fp.mass;
38         pos  = fp.pos;
39     }
40
41     void copyFromForce(const FPGrav & force) {
42         acc = force.acc;
43         pot = force.pot;
44     }
45
46     void clear() {
47         acc = 0.0;
48         pot = 0.0;
49     }
50
51     void writeAscii(FILE* fp) const {
52         fprintf(fp, "%lld\t%g\t%g\t%g\t%g\t%g\t%g\t%g\n",
```

```
53                  this->id, this->mass,
54                  this->pos.x, this->pos.y, this->pos.z,
55                  this->vel.x, this->vel.y, this->vel.z);
56      }
57
58      void readAscii(FILE* fp) {
59          fscanf(fp, "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\n",
60                 &this->id, &this->mass,
61                 &this->pos.x, &this->pos.y, &this->pos.z,
62                 &this->vel.x, &this->vel.y, &this->vel.z);
63          }
64
65  };
66
67
68  #ifdef ENABLE_PHANTOM_GRAPE_X86
69
70
71  template <class TParticleJ>
72  void CalcGravity(const FPGrav * iptcl,
73                   const PS::S32 ni,
74                   const TParticleJ * jptcl,
75                   const PS::S32 nj,
76                   FPGrav * force) {
77      const PS::S32 nipipe = ni;
78      const PS::S32 njpipe = nj;
79      PS::F64 (*xi)[3] = (PS::F64 (*)[3])malloc(sizeof(PS::F64) * nipipe *
            PS::DIMENSION);
80      PS::F64 (*ai)[3] = (PS::F64 (*)[3])malloc(sizeof(PS::F64) * nipipe *
            PS::DIMENSION);
81      PS::F64  *pi     = (PS::F64  *    )malloc(sizeof(PS::F64) * nipipe);
82      PS::F64 (*xj)[3] = (PS::F64 (*)[3])malloc(sizeof(PS::F64) * njpipe *
            PS::DIMENSION);
83      PS::F64  *mj     = (PS::F64  *    )malloc(sizeof(PS::F64) * njpipe);
84      for(PS::S32 i = 0; i < ni; i++) {
85          xi[i][0] = iptcl[i].getPos()[0];
86          xi[i][1] = iptcl[i].getPos()[1];
87          xi[i][2] = iptcl[i].getPos()[2];
88          ai[i][0] = 0.0;
89          ai[i][1] = 0.0;
90          ai[i][2] = 0.0;
91          pi[i]    = 0.0;
92      }
93      for(PS::S32 j = 0; j < nj; j++) {
94          xj[j][0] = jptcl[j].getPos()[0];
95          xj[j][1] = jptcl[j].getPos()[1];
96          xj[j][2] = jptcl[j].getPos()[2];
97          mj[j]    = jptcl[j].getCharge();
98          xj[j][0] = jptcl[j].pos[0];
99          xj[j][1] = jptcl[j].pos[1];
100         xj[j][2] = jptcl[j].pos[2];
101         mj[j]    = jptcl[j].mass;
102     }
103     PS::S32 devid = PS::Comm::getThreadNum();
104     g5_set_xmjMC(devid, 0, nj, xj, mj);
```

```
105     g5_set_nMC(devid, nj);
106     g5_calculate_force_on_xMC(devid, xi, ai, pi, ni);
107     for(PS::S32 i = 0; i < ni; i++) {
108         force[i].acc[0] += ai[i][0];
109         force[i].acc[1] += ai[i][1];
110         force[i].acc[2] += ai[i][2];
111         force[i].pot    -= pi[i];
112     }
113     free(xi);
114     free(ai);
115     free(pi);
116     free(xj);
117     free(mj);
118 }
119
120 #else
121
122 template <class TParticleJ>
123 void CalcGravity(const FPGrav * ep_i,
124                  const PS::S32 n_ip,
125                  const TParticleJ * ep_j,
126                  const PS::S32 n_jp,
127                  FPGrav * force) {
128     PS::F64 eps2 = FPGrav::eps * FPGrav::eps;
129     for(PS::S32 i = 0; i < n_ip; i++){
130         PS::F64vec xi = ep_i[i].getPos();
131         PS::F64vec ai = 0.0;
132         PS::F64 poti = 0.0;
133         for(PS::S32 j = 0; j < n_jp; j++){
134             PS::F64vec rij    = xi - ep_j[j].getPos();
135             PS::F64    r3_inv = rij * rij + eps2;
136             PS::F64    r_inv  = 1.0/sqrt(r3_inv);
137             r3_inv  = r_inv * r_inv;
138             r_inv  *= ep_j[j].getCharge();
139             r3_inv *= r_inv;
140             ai     -= r3_inv * rij;
141             poti   -= r_inv;
142         }
143         force[i].acc += ai;
144         force[i].pot += poti;
145     }
146 }
147
148 #endif
```

Listing 31: Sample code of N-body simulation (`nbody.cpp`)

```
1 #include<iostream>
2 #include<fstream>
3 #include<unistd.h>
4 #include<sys/stat.h>
5 #include<particle_simulator.hpp>
6 #ifdef ENABLE_PHANTOM_GRAPE_X86
7 #include <gp5util.h>
8 #endif
9 #ifdef ENABLE_GPU_CUDA
```

```
10 #define MULTI_WALK
11 #include"force_gpu_cuda.hpp"
12 #endif
13 #include "user-defined.hpp"
14
15 void makeColdUniformSphere(const PS::F64 mass_glb,
16                            const PS::S64 n_glb,
17                            const PS::S64 n_loc,
18                            PS::F64 *& mass,
19                            PS::F64vec *& pos,
20                            PS::F64vec *& vel,
21                            const PS::F64 eng = -0.25,
22                            const PS::S32 seed = 0) {
23
24     assert(eng < 0.0);
25     {
26         PS::MTTS mt;
27         mt.init_genrand(0);
28         for(PS::S32 i = 0; i < n_loc; i++){
29             mass[i] = mass_glb / n_glb;
30             const PS::F64 radius = 3.0;
31             do {
32                 pos[i][0] = (2. * mt.genrand_res53() - 1.) * radius;
33                 pos[i][1] = (2. * mt.genrand_res53() - 1.) * radius;
34                 pos[i][2] = (2. * mt.genrand_res53() - 1.) * radius;
35             }while(pos[i] * pos[i] >= radius * radius);
36             vel[i][0] = 0.0;
37             vel[i][1] = 0.0;
38             vel[i][2] = 0.0;
39         }
40     }
41
42     PS::F64vec cm_pos  = 0.0;
43     PS::F64vec cm_vel  = 0.0;
44     PS::F64    cm_mass = 0.0;
45     for(PS::S32 i = 0; i < n_loc; i++){
46         cm_pos  += mass[i] * pos[i];
47         cm_vel  += mass[i] * vel[i];
48         cm_mass += mass[i];
49     }
50     cm_pos /= cm_mass;
51     cm_vel /= cm_mass;
52     for(PS::S32 i = 0; i < n_loc; i++){
53         pos[i] -= cm_pos;
54         vel[i] -= cm_vel;
55     }
56 }
57
58 template<class Tpsys>
59 void setParticlesColdUniformSphere(Tpsys & psys,
60                                    const PS::S32 n_glb,
61                                    PS::S32 & n_loc) {
62
63     n_loc = n_glb;
64     psys.setNumberOfParticleLocal(n_loc);
```

```
65
66      PS::F64    * mass = new PS::F64[n_loc];
67      PS::F64vec * pos  = new PS::F64vec[n_loc];
68      PS::F64vec * vel  = new PS::F64vec[n_loc];
69      const PS::F64 m_tot = 1.0;
70      const PS::F64 eng   = -0.25;
71      makeColdUniformSphere(m_tot, n_glb, n_loc, mass, pos, vel, eng);
72      for(PS::S32 i = 0; i < n_loc; i++){
73          psys[i].mass = mass[i];
74          psys[i].pos  = pos[i];
75          psys[i].vel  = vel[i];
76          psys[i].id   = i;
77      }
78      delete [] mass;
79      delete [] pos;
80      delete [] vel;
81  }
82
83  template<class Tpsys>
84  void kick(Tpsys & system,
85            const PS::F64 dt) {
86      PS::S32 n = system.getNumberOfParticleLocal();
87      for(PS::S32 i = 0; i < n; i++) {
88          system[i].vel  += system[i].acc * dt;
89      }
90  }
91
92  template<class Tpsys>
93  void drift(Tpsys & system,
94             const PS::F64 dt) {
95      PS::S32 n = system.getNumberOfParticleLocal();
96      for(PS::S32 i = 0; i < n; i++) {
97          system[i].pos  += system[i].vel * dt;
98      }
99  }
100
101 template<class Tpsys>
102 void calcEnergy(const Tpsys & system,
103                 PS::F64 & etot,
104                 PS::F64 & ekin,
105                 PS::F64 & epot,
106                 const bool clear=true){
107     if(clear){
108         etot = ekin = epot = 0.0;
109     }
110     PS::F64 etot_loc = 0.0;
111     PS::F64 ekin_loc = 0.0;
112     PS::F64 epot_loc = 0.0;
113     const PS::S32 nbody = system.getNumberOfParticleLocal();
114     for(PS::S32 i = 0; i < nbody; i++){
115         ekin_loc += system[i].mass * system[i].vel * system[i].vel;
116         epot_loc += system[i].mass * (system[i].pot + system[i].mass /
                FPGrav::eps);
117     }
118     ekin_loc *= 0.5;
```

```
119     epot_loc *= 0.5;
120     etot_loc  = ekin_loc + epot_loc;
121     etot = PS::Comm::getSum(etot_loc);
122     epot = PS::Comm::getSum(epot_loc);
123     ekin = PS::Comm::getSum(ekin_loc);
124 }
125
126 void printHelp() {
127     std::cerr<<"o: dir name of output (default: ./result)"<<std::endl;
128     std::cerr<<"t: theta (default: 0.5)"<<std::endl;
129     std::cerr<<"T: time_end (default: 10.0)"<<std::endl;
130     std::cerr<<"s: time_step (default: 1.0 / 128.0)"<<std::endl;
131     std::cerr<<"d: dt_diag (default: 1.0 / 8.0)"<<std::endl;
132     std::cerr<<"D: dt_snap (default: 1.0)"<<std::endl;
133     std::cerr<<"l: n_leaf_limit (default: 8)"<<std::endl;
134     std::cerr<<"n: n_group_limit (default: 64)"<<std::endl;
135     std::cerr<<"N: n_tot (default: 1024)"<<std::endl;
136     std::cerr<<"h: help"<<std::endl;
137 }
138
139 void makeOutputDirectory(char * dir_name) {
140     struct stat st;
141     PS::S32 ret;
142     if (PS::Comm::getRank() == 0) {
143         if (stat(dir_name, &st) != 0) {
144             ret = mkdir(dir_name, 0777);
145         } else {
146             ret = 0; // the directory named dir_name already exists.
147         }
148     }
149     PS::Comm::broadcast(&ret, 1);
150     if (ret == 0) {
151         if (PS::Comm::getRank() == 0)
152             fprintf(stderr, "Directory \"%s\" is successfully made.\n",
                    dir_name);
153     } else {
154         if (PS::Comm::getRank() == 0)
155             fprintf(stderr, "Directory %s fails to be made.\n", dir_name);
156         PS::Abort();
157     }
158 }
159
160 PS::F64 FPGrav::eps = 1.0/32.0;
161
162 int main(int argc, char *argv[]) {
163     std::cout<<std::setprecision(15);
164     std::cerr<<std::setprecision(15);
165
166     PS::Initialize(argc, argv);
167     PS::F32 theta = 0.5;
168     PS::S32 n_leaf_limit = 8;
169     PS::S32 n_group_limit = 64;
170     PS::F32 time_end = 10.0;
171     PS::F32 dt = 1.0 / 128.0;
172     PS::F32 dt_diag = 1.0 / 8.0;
```

```
173     PS::F32 dt_snap = 1.0;
174     char dir_name[1024];
175     PS::S64 n_tot = 1024;
176     PS::S32 c;
177     sprintf(dir_name,"./result");
178     opterr = 0;
179     while((c=getopt(argc,argv,"i:o:d:D:t:T:l:n:N:hs:")) != -1){
180         switch(c){
181         case 'o':
182             sprintf(dir_name,optarg);
183             break;
184         case 't':
185             theta = atof(optarg);
186             std::cerr << "theta = " << theta << std::endl;
187             break;
188         case 'T':
189             time_end = atof(optarg);
190             std::cerr << "time_end = " << time_end << std::endl;
191             break;
192         case 's':
193             dt = atof(optarg);
194             std::cerr << "time_step = " << dt << std::endl;
195             break;
196         case 'd':
197             dt_diag = atof(optarg);
198             std::cerr << "dt_diag = " << dt_diag << std::endl;
199             break;
200         case 'D':
201             dt_snap = atof(optarg);
202             std::cerr << "dt_snap = " << dt_snap << std::endl;
203             break;
204         case 'l':
205             n_leaf_limit = atoi(optarg);
206             std::cerr << "n_leaf_limit = " << n_leaf_limit << std::endl;
207             break;
208         case 'n':
209             n_group_limit = atoi(optarg);
210             std::cerr << "n_group_limit = " << n_group_limit << std::endl;
211             break;
212         case 'N':
213             n_tot = atoi(optarg);
214             std::cerr << "n_tot = " << n_tot << std::endl;
215             break;
216         case 'h':
217             if(PS::Comm::getRank() == 0) {
218                 printHelp();
219             }
220             PS::Finalize();
221             return 0;
222         default:
223             if(PS::Comm::getRank() == 0) {
224                 std::cerr<<"No such option! Available options are here."<<
                        std::endl;
225                 printHelp();
226             }
```

```
227             PS::Abort();
228         }
229     }
230
231     makeOutputDirectory(dir_name);
232
233     std::ofstream fout_eng;
234
235     if(PS::Comm::getRank() == 0) {
236         char sout_de[1024];
237         sprintf(sout_de, "%s/t-de.dat", dir_name);
238         fout_eng.open(sout_de);
239         fprintf(stdout, "This␣is␣a␣sample␣program␣of␣N-body␣simulation␣on␣
                FDPS!\n");
240         fprintf(stdout, "Number␣of␣processes:␣%d\n", PS::Comm::
                getNumberOfProc());
241         fprintf(stdout, "Number␣of␣threads␣per␣process:␣%d\n", PS::Comm::
                getNumberOfThread());
242     }
243
244     PS::ParticleSystem<FPGrav> system_grav;
245     system_grav.initialize();
246     PS::S32 n_loc    = 0;
247     PS::F32 time_sys = 0.0;
248     if(PS::Comm::getRank() == 0) {
249         setParticlesColdUniformSphere(system_grav, n_tot, n_loc);
250     } else {
251         system_grav.setNumberOfParticleLocal(n_loc);
252     }
253
254     const PS::F32 coef_ema = 0.3;
255     PS::DomainInfo dinfo;
256     dinfo.initialize(coef_ema);
257     dinfo.decomposeDomainAll(system_grav);
258     system_grav.exchangeParticle(dinfo);
259     n_loc = system_grav.getNumberOfParticleLocal();
260
261 #ifdef ENABLE_PHANTOM_GRAPE_X86
262     g5_open();
263     g5_set_eps_to_all(FPGrav::eps);
264 #endif
265
266     PS::TreeForForceLong<FPGrav, FPGrav, FPGrav>::Monopole tree_grav;
267     tree_grav.initialize(n_tot, theta, n_leaf_limit, n_group_limit);
268 #ifdef MULTI_WALK
269     const PS::S32 n_walk_limit = 200;
270     const PS::S32 tag_max = 1;
271     tree_grav.calcForceAllAndWriteBackMultiWalk(DispatchKernelWithSP,
272                                                 RetrieveKernel,
273                                                 tag_max,
274                                                 system_grav,
275                                                 dinfo,
276                                                 n_walk_limit);
277 #else
278     tree_grav.calcForceAllAndWriteBack(CalcGravity<FPGrav>,
```

```
279                                         CalcGravity<PS::SPJMonopole>,
280                                         system_grav,
281                                         dinfo);
282 #endif
283     PS::F64 Epot0, Ekin0, Etot0, Epot1, Ekin1, Etot1;
284     calcEnergy(system_grav, Etot0, Ekin0, Epot0);
285     PS::F64 time_diag = 0.0;
286     PS::F64 time_snap = 0.0;
287     PS::S64 n_loop = 0;
288     PS::S32 id_snap = 0;
289     while(time_sys < time_end){
290         if( (time_sys >= time_snap) || ( (time_sys + dt) - time_snap ) > (
                time_snap - time_sys) ){
291             char filename[256];
292             sprintf(filename, "%s/%04d.dat", dir_name, id_snap++);
293             FileHeader header;
294             header.time   = time_sys;
295             header.n_body = system_grav.getNumberOfParticleGlobal();
296             system_grav.writeParticleAscii(filename, header);
297             time_snap += dt_snap;
298         }
299
300         calcEnergy(system_grav, Etot1, Ekin1, Epot1);
301
302         if(PS::Comm::getRank() == 0){
303             if( (time_sys >= time_diag) || ( (time_sys + dt) - time_diag )
                    > (time_diag - time_sys) ){
304                 fout_eng << time_sys << "␣␣␣" << (Etot1 - Etot0) / Etot0
                        << std::endl;
305                 fprintf(stdout, "time:␣%10.7f␣energy␣error:␣%+e\n",
306                         time_sys, (Etot1 - Etot0) / Etot0);
307                 time_diag += dt_diag;
308             }
309         }
310
311
312         kick(system_grav, dt * 0.5);
313
314         time_sys += dt;
315         drift(system_grav, dt);
316
317         if(n_loop % 4 == 0){
318             dinfo.decomposeDomainAll(system_grav);
319         }
320
321         system_grav.exchangeParticle(dinfo);
322 #ifdef MULTI_WALK
323         tree_grav.calcForceAllAndWriteBackMultiWalk(DispatchKernelWithSP,
324                                                     RetrieveKernel,
325                                                     tag_max,
326                                                     system_grav,
327                                                     dinfo,
328                                                     n_walk_limit,
329                                                     true);
330 #else
```

```
331         tree_grav.calcForceAllAndWriteBack(CalcGravity<FPGrav>,
332                                            CalcGravity<PS::SPJMonopole>,
333                                            system_grav,
334                                            dinfo);
335 #endif
336
337         kick(system_grav, dt * 0.5);
338
339         n_loop++;
340     }
341
342 #ifdef ENABLE_PHANTOM_GRAPE_X86
343     g5_close();
344 #endif
345
346     PS::Finalize();
347     return 0;
348 }
```

6 Extensions

6.1 P^3M code

In this section, we explain the usage of a FDPS extension "Particle Mesh" (hereafter PM) using a sample program for P^3M(Particle-Particle-Particle-Mesh) method. The sample code calculates the crystal energy of sodium chloride (NaCl) crystal using the P^3M method and compares the result with the analytical solution. In the P^3M method, the calculation of force and potential energy is performed by splitting into Particle-Particle(PP) part and Particle-Mesh(PM) part. In this sample code, the PP part is calculated by using FDPS standard features and the PM part is computed by using a FDPS extension "PM". Note that the detail of the extension "PM" is described in § 9.2 of the specification of FDPS and please see it for detail.

6.1.1 Location of sample code and working directory

The sample code is placed at `$(FDPS)/sample/c++/p3m`. Change the current directory to there. `main.cpp` in this directory is the sample code.

```
$ cd (FDPS)/sample/c++/p3m
```

6.1.2 Required header files

In order to use the FDPS extension "PM", we must include the header files `particle_mesh.hpp` and `param_fdps.h` as well as `particle_simulator.hpp` (the last one is needed to use FDPS standard features). In addition, `param.h` is included because the sample code accesses a non-public constant `CUTOFF_RADIUS`.

Listing 32: Include FDPS

```
1 #include <particle_simulator.hpp>
2 #include <particle_mesh.hpp>
3 #include <param.h>
4 #include <param_fdps.h>
```

6.1.3 User-defined classes

In this section, we describe classes that you need to define in order to perform P^3M calculation using FDPS.

6.1.3.1 FullParticle type

You must define a FullParticle type. Listing 33 shows the implementation of FullParticle type in the sample code. FullParticle type must have all physical quantities required to perform a calculation with P^3M method and it must have the following member functions:

`getCharge()`
: required for FDPS to get the charge of particles

`getChargeParticleMesh()`
: required for the PM module of FDPS to get the charge of particles

`getPos()`
: required for FDPS to get the position of particles

`getRSearch()`
: required for FDPS to get the cutoff radius

`setPos()`
: required for FDPS to write the positions of particles recorded in FullParticle object

`copyFromForce()`
: required for FDPS to copy data form Force object

`copyFromForceParticleMesh()`
: required for the PM module to write the result of Force calculation to FullParticle object

Note that `copyFromForce()` and `copyFromForceParticleMesh()` are empty functions in this sample code. This is because the sample code explicitly copy data from Force objects to FullParticle object using APIs such as `getForce()` (explained later).

Listing 33: FullParticle type

```
1  class Nbody_FP
2  {
3     public:
4        PS::S64 id;
5        PS::F64 m;
6        PS::F64 rc;
7        PS::F64vec x;
8        PS::F64vec v,v_half;
9        PS::F64vec agrv;
10       PS::F64 pot;
11       // Member functions required by FDPS
12       PS::F64 getCharge() const {
13          return m;
14       };
15       PS::F64 getChargeParticleMesh() const {
16          return m;
17       };
18       PS::F64vec getPos() const {
19          return x;
20       };
21       PS::F64 getRSearch() const {
22          return rc;
23       };
24       void setPos(const PS::F64vec& x) {
25          this->x = x;
26       };
27       void copyFromForce(const Nbody_PP_Results& result) {};
28       void copyFromForceParticleMesh(const PS::F64 apm) {};
29 };
```

6.1.3.2 EssentialParticleI type

You must define a EssentialParticleI type. EssentialParticleI type must have member variables that store all physical quantities necessary for an i particle to perform the PP part of

the Force calculation. In the sample code, it is also used as EssentialParticleJ type. Therefore, it should have member variables that store all physical quantities necessary for a j particle to perform the PP part of the Force calculation. Listing 34 shows the implementation of EssentialParticleI type in the sample code. EssentialParticleI type needs to have member function `copyFromFP()` to copy data from FullParticle object described above. In addition, it must have `getCharge()` (returns the charges of particles), `getPos()` (returns the positions of particles), `getRSearch()` (returns the cutoff radius of particles), and `setPos()` (sets the positions of particles).

Listing 34: EssentialParticleI type

```
class Nbody_EP
{
   public:
      PS::S64 id;
      PS::F64 m;
      PS::F64 rc;
      PS::F64vec x;
      // Member functions required by FDPS
      PS::F64 getCharge() const {
         return m;
      };
      PS::F64vec getPos() const {
         return x;
      };
      PS::F64 getRSearch() const {
         return rc;
      };
      void setPos(const PS::F64vec& x) {
         this->x = x;
      };
      void copyFromFP(const Nbody_FP& FP) {
         id  = FP.id;
         m   = FP.m;
         rc  = FP.rc;
         x   = FP.x;
      };
};
```

6.1.3.3 Force type

You must define a Force type. Force type must have member variables that store the results of the PP part of the Force calculation. Listing 35 shows the implementation of Force type in this sample code. Because we consider Coulomb interaction only, one Force type is defined. Force type needs to have member function `clear()` to zero-clear or initialize member variables that store the results of accumulation operation.

Listing 35: Force type

```
class Nbody_PP_Results
{
   public:
      PS::F64 pot;
```

```
5       PS::F64vec agrv;
6       void clear() {
7          pot  = 0.0;
8          agrv = 0.0;
9       }
10 };
```

6.1.3.4 calcForceEpEp type

You must define a calcForceEpEp type. calcForceEpE type must contain actual code for the PP part of the Force calculation. Listing 36 shows the implementation of calcForceEpEp type in this sample code. In the code, it is implemented as a Functor (function object). The arguments of the Functor is an array of EssentialParticleI objects, the number of EssentialParticleI objects, an array of EssentialParticleJ objects, the number of EssentialParticleJ objects, and an array of Force objects.

Listing 36: calcForceEpEp 型

```
1  class Calc_force_ep_ep{
2     public:
3        void operator () (const Nbody_EP* const ep_i,
4                          const PS::S32 Nip,
5                          const Nbody_EP* const ep_j,
6                          const PS::S32 Njp,
7                          Nbody_PP_Results* const result) {
8           for (PS::S32 i=0; i<Nip; i++) {
9              for (PS::S32 j=0; j<Njp; j++) {
10                PS::F64vec dx = ep_i[i].x - ep_j[j].x;
11                PS::F64 rij = std::sqrt(dx * dx);
12                if ((ep_i[i].id == ep_j[j].id) && (rij == 0.0)) continue;
13                PS::F64 rinv = 1.0/rij;
14                PS::F64 rinv3 = rinv*rinv*rinv;
15                PS::F64 xi = 2.0*rij/ep_i[i].rc;
16                result[i].pot  += ep_j[j].m * S2_pcut(xi) * rinv;
17                result[i].agrv += ep_j[j].m * S2_fcut(xi) * rinv3 * dx;
18             }
19             //* Self-interaction term
20             result[i].pot -= ep_i[i].m * (208.0/(70.0*ep_i[i].rc));
21          }
22
23       };
24 };
```

The PP part in the P^3M method is a two-body interaction with cutoff (i.e. the interaction is truncated if the distance between the particles is larger than the cutoff distance). Hence, cutoff functions (`S2_pcut()`, `S2_fcut()`) appears in the calculations of potential and acceleration. These cutoff functions must be the ones that are constructed assuming that the particle shape function is $S2(r)$, which is introduced by Hockney & Eastwood (1988)(Eq.(8.3)) and takes the form of

$$S2(r) = \begin{cases} \frac{48}{\pi a^4}\left(\frac{a}{2} - r\right) & r < a/2, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

where r is the distance from the center of a particle, a is the scale length of the shape function. When assuming this shape function, the charge density distribution due to a particle, $\rho(r)$, is expressed as $\rho(r) = q\,S2(r)$, where q is the charge of the particle. Thus, $S2(r)$ shape function gives linear density distribution. The reason why we have to use the cutoff functions that correspond to $S2(r)$ shape function is that the cutoff functions used in the PM part also assumes the $S2(r)$ shape function (the cutoff functions in the PM and PP parts should be consistent with each other).

The cutoff functions must be defined by a user. Possible implementations for `S2_pcut()` and `S2_fcut()` are given at the beginning of the sample code (see the lines 22-72 in `main.cpp`). In these examples, we used Eqs.(8-72) and (8-75) in Hockney & Eastwood (1988) and we define them such that the PP interaction takes of the form:

$$\Phi_{\rm PP}(\boldsymbol{r}) = \frac{m}{|\boldsymbol{r}-\boldsymbol{r}'|}\texttt{S2\_pcut}(\xi) \tag{2}$$

$$\boldsymbol{f}_{\rm PP}(\boldsymbol{r}) = \frac{m(\boldsymbol{r}-\boldsymbol{r}')}{|\boldsymbol{r}-\boldsymbol{r}'|^3}\texttt{S2\_fcut}(\xi) \tag{3}$$

where $\xi = 2|\boldsymbol{r}-\boldsymbol{r}'|/a$. In this sample code, a is expressed as a variable `rc`.

As is clear from Eq.(8-75) in Hockney & Eastwood (1988), the mesh potential ϕ^m has a finite value at $r = 0$ (we omit a factor $1/4\pi\varepsilon_0$ here):

$$\phi^m(0) = \frac{208}{70a} \tag{4}$$

This term is taken into account the last line in the i-particle loop:

```
1 result[i].pot -= ep_i[i].m * (208.0/(70.0*ep_i[i].rc));
```

Note that this term is necessary to match the numerical result with the analytical solution.

6.1.4 Main body of the sample code

In this section, we explain the main body of the sample code. Before going into details, we first give a simple explanation about the content and the structure of the sample code. As described in § 6.1, this code computes the crystal energy of NaCl crystal using the P^{3}M method and compares the result with the analytical solution. The NaCl crystal is expressed as an uniform grid of particles in this sample code. Na and Cl are placed in the staggered layout. Particles corresponding to Na has a positive charge, while those corresponding to Cl has a negative charge. We place a crystal expressed as an grid of charged particles into a periodic computational box of the sizes $[0, 1]^3$ and calculates the crystal energy. The computational accuracy of the crystal energy should depend on the number of particles and the configuration of particles (to the grid used in the PM calculation). Hence, in the sample code, we measure the relative energy errors for a different set of these parameters and output the result of the comparisons into a file.

The structure of the sample code is as follows:

(1) Create and initialize FDPS objects
(2) Create a NaCl crystal for given number of particles and configuration (the function `NaCl_IC()`)

(3) Compute the potential energy of each particle by the P^3M method (the function `Nbody_objs.calc_gravity()`)
(4) Compute the total energy of the crystal and compare it with the analytical solution (the function `calc_energy_error()`)
(5) Repeat (2)-(4)

In the following, we explain in detail each steps described above.

6.1.4.1 Initialization and Termination of FDPS

First, you must initialize FDPS by the following code.

Listing 37: Initialization of FDPS

```
1 PS::Initialize(argc, argv);
```

Once started, FDPS should be terminated explicitly. In this sample, FDPS is terminated just before the termination of the program. Hence, you need to write the following code at the end of the main function.

Listing 38: Termination of FDPS

```
1 PS::Finalize();
```

6.1.4.2 Creation and initialization of FDPS objects

After the initialization of FDPS, a user need to create the objects used to talk to FDPS. In this section, we describe how to create and initialize these objects.

6.1.4.2.1 Creation of necessary FDPS objects

In the calculation using the P^3M method, we must create objects of the ParticleSystem class and the DomainInfo class. In addition, objects of the TreeForForceLong class and the ParticleMesh class are need to calculate the PP and PM parts of the Force calculation. In this sample code, these objects are grouped into `Nbody_Objects` class. The following code is the implementation of the `Nbody_Objects` class.

Listing 39: `Nbody_Objects` class

```
1 class Nbody_Objects {
2    public:
3       PS::ParticleSystem<Nbody_FP> system;
4       PS::DomainInfo dinfo;
5       PS::TreeForForceLong<Nbody_PP_Results, Nbody_EP, Nbody_EP>::
              MonopoleWithCutoff pp_tree;
6       PS::PM::ParticleMesh pm;
7 }
```

In this sample, a object of the `Nbody_Objects` class is created as a local variable in the main function:

Listing 40: Creation of a `Nbody_Objects`-class object

```
1 Nbody_Objects Nbody_objs;
```

6.1.4.2.2 Initialization of FDPS objects

After the creation of FDPS objects, you must initialize these objects before you use them in a user code. In the following, we explain how to initialize each object.

(i) Initialization of a ParticleSystem object A ParticleSystem object is initialized as follows:

Listing 41: Initialization of a ParticleSystem object

```
1 Nbody_objs.system.initialize();
```

This is done in the main function in the sample code.

(ii) Initialization of a DomainInfo object A DomainInfo object is initialized as follows:

Listing 42: Initialization of a DomainInfo object

```
1 Nbody_objs.dinfo.initialize();
```

This is done in the main function in the sample code.

After the initialization, you need to specify the boundary condition and the size of the simulation box through the `setBoundaryCondition` and `setPosRootDomain` methods. In the sample code, these procedures are performed in the function `NaCl_IC()` that sets up the distribution of particles:

```
1 dinfo.setBoundaryCondition(PS::BOUNDARY_CONDITION_PERIODIC_XYZ);
2 dinfo.setPosRootDomain(PS::F64vec(0.0,0.0,0.0),
3                        PS::F64vec(1.0,1.0,1.0));
```

(iii) Initialization of a TreeForForceLong object A TreeForForceLong is initialized by the `initialize` method:

Listing 43: Initialization of a TreeForForceLong object

```
1 void init_tree() {
2    PS::S32 numPtclLobal = system.getNumberOfParticleLobal();
3    PS::U64 ntot = 3 * numPtclLobal;
4    pp_tree.initialize(ntot,0.0);
5 };
```

You need to give a rough number of particles to this method as the first argument. Here, we set three times the number of local particles at the time of calling. The second argument of this method is an optional argument and represents the opening angle criterion θ for the tree method. In the sample, we do not use the tree method in the PP part of the Force calculation. Therefore, we set $\theta = 0$.

In this sample code, a TreeForForceLong object is initialized within the function `Nbody_objs.init_tree()` (see the main function).

```
1 if (is_tree_initialized == false) {
2    Nbody_objs.init_tree();
3    is_tree_initialized = true;
4 }
```

where the `if` statement above is necessary because the initialization should be done only once in the program (otherwise, the program will fail).

(iv) Initialization of a ParticleMesh object No explicit initialization is needed.

6.1.4.3 Generation of a distribution of particles

In this section, we explain the function `NaCl_IC` that generates a distribution of particles, and FDPS APIs called within it. Given the number of particles per one space dimension and the position of the particle that is nearest to the origin $(0, 0, 0)$, the function `NaCl_IC` makes a three-dimensional uniform grid of particles. These parameters are specified through a object of the `Crystal_Parameters` class, `NaCl_params`:

```
1  class Crystal_Parameters
2  {
3     public:
4        PS::S32 numPtcl_per_side;
5        PS::F64vec pos_vertex;
6  };
7  /* In main function */
8  Crystal_Parameters NaCl_params;
9  NaCl_IC(Nbody_objs.system,
10           Nbody_objs.dinfo,
11           NaCl_params);
```

In the first half of the function `NaCl_IC`, it makes an uniform grid of particles based on the value of `NaCl_params`. In this process, we scale the particle charge m to satisfy the relation

$$\frac{2Nm^2}{R_0} = 1, \tag{5}$$

where N is the total number of molecules (the total number of atomic particles is $2N$) and R_0 is the distance to the nearest particle. This scaling is introduced just for convenience: The crystal energy can be written analytically as

$$E = -\frac{N\alpha m^2}{R_0}, \tag{6}$$

where α is the Madelung constant and $\alpha \approx 1.747565$ for the NaCl crystal (e.g. see Kittel (2004) "Introduction to Solid State Physics"). Thus, the crystal energy depends on the total number of particles. This is inconvenient when comparing the calculation result with the analytical solution. By scaling the particle charge as described above, the crystal energy becomes independent from N.

After generating a particle distribution, this function performs domain decomposition and particle exchange using FDPS APIs. In the following, we explain these APIs.

6.1.4.3.1 Domain Decomposition

The `decomposeDomainAll` method of the DomainInfo class is used to perform domain decomposition based on the current distribution of particles:

Listing 44: Domain Decomposition

```
1 dinfo.decomposeDomainAll(system);
```

Note that this method needs a ParticleSystem object as the argument to get the information of particle distribution.

6.1.4.3.2 Particle Exchange

The `exchangeParticle` method of the ParticleSystem class is used to exchange particles based on the current decomposed domains:

Listing 45: Particle Exchange

```
1 system.exchangeParticle(dinfo);
```

Note that this method needs a DomainInfo object as the argument to get the domain information.

6.1.4.4 Interaction Calculation

After these procedures are completed, we must perform the interaction calculation. In the sample code, it is performed in the main function by calling the function `Nbody_objs.calc_gravity()`:

Listing 46: Execution of interaction calculation

```
1 Nbody_objs.calc_gravity();
```

The function `Nbody_objs.calc_gravity()` consists of (i) zero-clear of potential energy and acceleration of each particle, (ii) calculation of the PM part, and (iii) calculation of the PP part:

Listing 47: Interaction calculation

```
1  void calc_gravity() {
2     //* Local variables
3     PS::S32 numPtclLocal = system.getNumberOfParticleLocal();
4
5     //* Reset potential and accelerations
6     for (PS::S32 i=0; i<numPtclLocal; i++) {
7        system[i].pot  = 0.0;
8        system[i].agrv = 0.0;
9     }
10
11    //=================
12    //* [1] PM part
13    //=================
14    pm.setDomainInfoParticleMesh(dinfo);
15    pm.setParticleParticleMesh(system);
16    pm.calcMeshForceOnly();
17    for (PS::S32 i=0; i<numPtclLocal; i++) {
18       PS::F32vec x32 = system[i].x;
19       system[i].pot  -= pm.getPotential(x32);
20       system[i].agrv -= pm.getForce(x32);
```

```
21    }
22
23    //=================
24    //* [2] PP part
25    //=================
26    pp_tree.calcForceAll(Calc_force_ep_ep(),
27                         Calc_force_ep_sp(),
28                         system, dinfo);
29    for (PS::S32 i=0; i<numPtclLocal; i++) {
30       Nbody_PP_Results result = pp_tree.getForce(i);
31       system[i].pot  += result.pot;
32       system[i].agrv += result.agrv;
33    }
34 };
```

The code shown below is the PM part of the Force calculation (hereafter PM calculation). In order to perform the PM calculation, the ParticleMesh object `pm` must have information about the domain and the particles in advance. Therefore, `setDomainInfoParticleMesh` and `setParticleParticleMesh` methods are first called to give these information to the object `pm`. Now, the object `pm` are ready to perform the PM calculation. In the sample code, the PM calculation is performed by `calcMeshForceOnly` method. Then, `getPotential` and `getForce` methods are called to obtain the potential and acceleration at the particle position. They are stored to the FullParticle type object `system`. ***Note that the accumulation operation is done by the operator*** `-=`. The reason why we use `-=` instead of `+=` is that the FDPS extension "PM" computes the potential energy assuming gravity. In other words, the FDPS extension "PM" treats a charge with $m(>0)$ creates negative potential. Hence, ***we need to invert the signs of potential energy and acceleration*** in order to use the FDPS extension "PM" for the Coulomb interaction calculation.

Listing 48: PM part of Force calculation

```
1 pm.setDomainInfoParticleMesh(dinfo);
2 pm.setParticleParticleMesh(system);
3 pm.calcMeshForceOnly();
4 for (PS::S32 i=0; i<numPtclLocal; i++) {
5    PS::F32vec x32 = system[i].x;
6    system[i].pot  -= pm.getPotential(x32);
7    system[i].agrv -= pm.getForce(x32);
8 }
```

Next, we shows the PP part of the Force calculation in the following. The `calcForceAll` method of the TreeForForceLong class is used to calculation the PP part (we do not use the `calcForceAllAndWriteBack` method because this method zero-clears the results of the PM calculation stored the FullParticle object). Then, the `getForce` method is used to obtain the potential energy and the acceleration at the particle position and they are accumulated to the FullParticle-type object `system`.

Listing 49: PP part of Force calculation

```
1 pp_tree.calcForceAll(Calc_force_ep_ep(),
2                      Calc_force_ep_sp(),
3                      system, dinfo);
4 for (PS::S32 i=0; i<numPtclLocal; i++) {
5    Nbody_PP_Results result = pp_tree.getForce(i);
```

```
6     system[i].pot  += result.pot;
7      system[i].agrv += result.agrv;
8  }
```

6.1.4.5 Calculation of relative energy error

The relative error of the crystal energy is computed in the function `calc_energy_error()`, where we assume that the analytical solution is $E_0 \equiv 2E = -1.7475645946332$, which is numerically evaluated by the PM^3(Particle-Mesh Multipole Method).

6.1.5 Compile

Before compiling your program, you need to install the FFTW(Fast Fourier Transform in the West) library. Then, edit the file `Makefile` in the working directory to set the PATHs of the locations of FFTW and FDPS to the variables `FFTW_LOC` and `FDPS_LOC`. After that, run `make`.

```
$ make
```

The execution file `p3m.x` will be created in the directory `work` if the compilation is succeeded.

6.1.6 Run

You must run your program using MPI with the number of MPI processes is equal to or greater than 2, because of the specification of FDPS extensions. Therefore, you should run the following command:

```
$ MPIRUN -np NPROC ./p3m.x
```

where "MPIRUN" represents the command to run your program using MPI such as `mpirun` or `mpiexec`, and "NPROC" is the number of MPI processes.

6.1.7 Check the result

After the program ended, a file that records the relative error of the crystal energy is output in the directory `work`. Figure 3 shows the dependency of the relative error on the number of particles used.

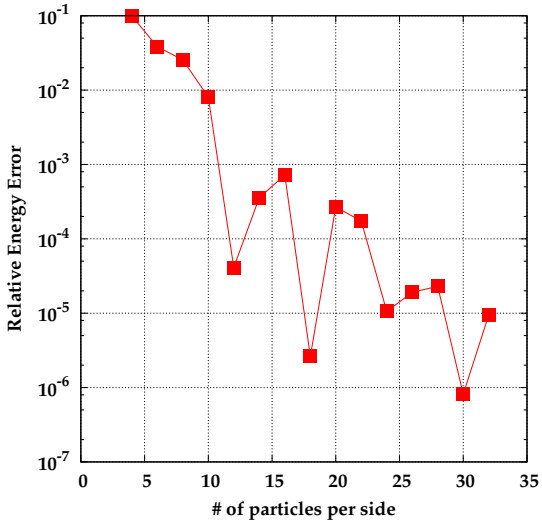

Figure 3: The relative error of the crystal energy as a function of the number of particles per side, where we assume that the number of the PM grids is 16^3 and the cutoff radius is 3/16.

6.2 TreePM code

In this section, we explain the usage of a FDPS extension "Particle Mesh" (hereafter PM) using a sample program for TreePM(Tree-Particle-Mesh) method. This sample code performs cosmological N-body simulation using the TreePM method. In the TreePM method, the calculation of gravity is performed by splitting into the PP part and the PM part as in the P^3M method. Therefore, functions of FDPS used in the sample code is almost the same as the sample code for the P^3M method. The difference between the two is that the PP part is computed by the Tree method in the TreePM method while the P^3M method uses the direct summation for the PP part.

6.2.1 Location of the sample code and the working directory

The sample code is placed at `$(FDPS)/sample/c++/treepm`. Change the current directory to there. As shown below, the sample code consists of several source files, of which the main body of the program is implemented in the files `treepm.hpp` and `treepm.cpp`.

```
$ cd (FDPS)/sample/c++/treepm
$ ls | awk '{print $0}'
IC/
Makefile
README_en.txt
README_ja.txt
constants.hpp
cosmology.hpp
fig/
make_directory.c
param_file_for_test.txt
prototype.h
result/
run_param.hpp
test.py*
timing.c
treepm.cpp
treepm.hpp
utils/
```

6.2.2 Required header files

In order to use the FDPS extension "PM", we must include the header file `particle_mesh.hpp` as well as `particle_simulator.hpp`. These are described in the file `treepm.cpp`:

Listing 50: Include FDPS

```
1 #include <particle_simulator.hpp>
2 #include <particle_mesh.hpp>
```

6.2.3 User-defined classes

In this section, we describe classes that you need to define in order to perform TreePM calculation using FDPS.

6.2.3.1 FullParticle type

You must define a FullParticle type. FullParticle type must have all physical quantities required to perform a calculation with the TreePM method. Listing 51 shows the implementation of FullParticle type in the sample code. In the code, it has member variables necessary for usual N-body simulations (`id`, `mass`, `eps`, `pos`, `vel`, `acc`). In addition, it has the following member variables: `acc_pm` (stores the acceleration of the PM part), `H0` (stores the Hubble constant)、and `Lbnd` (stores the size of the simulation box in the unit of Mpc h^{-1}). FullParticle type must have the following member functions to use the FDPS standard functions and the FDPS extension "PM":

`getCharge()`
: required for FDPS to get the mass of particle

`getChargeParticleMesh()`
: required for the PM module to get the mass of particle

`getPos()`
: required for FDPS to get the position of particle

`getRSearch()`
: required for FDPS to get the cutoff radius

`setPos()`
: required for FDPS to write the position of particle recorded in FullParticle object

`copyFromForce()`
: required for FDPS to copy data from a Force object

`copyFromForceParticleMesh()`
: required for the PM module to write the result of Force calculation to FullParticle object

In addition, the sample uses file I/O functions of FDPS, which requires a user to define the following member functions:

- `readBinary()`
- `writeBinary()`

Note that the use of these file I/O functions is not necessary and user-defined I/O can be used.

Listing 51: FullParticle type

```
1 class FPtreepm {
2 private:
3     template<class T>
4     T reverseEndian(T value){
5         char * first = reinterpret_cast<char*>(&value);
6         char * last = first + sizeof(T);
7         std::reverse(first, last);
8         return value;
9     }
```

```
10 public:
11     PS::S64    id;
12     PS::F32    mass;
13     PS::F32    eps;
14     PS::F64vec pos;
15     PS::F64vec vel;
16     PS::F64vec acc;
17     PS::F64vec acc_pm;
18
19     //static PS::F64vec low_boundary;
20     //static PS::F64vec high_boundary;
21     //static PS::F64 unit_l;
22     //static PS::F64 unit_m;
23     static PS::F64 H0;
24     static PS::F64 Lbnd;
25
26     PS::F64vec getPos() const {
27         return pos;
28     }
29
30     PS::F64 getCharge() const {
31         return mass;
32     }
33
34     void copyFromForce(const Result_treepm & force) {
35         this->acc = force.acc;
36     }
37
38     PS::F64 getRSearch() const {
39         PS::F64 rcut = 3.0/SIZE_OF_MESH;
40         return rcut;
41     }
42
43     void setPos(const PS::F64vec pos_new) {
44         this->pos = pos_new;
45     }
46
47     PS::F64 getChargeParticleMesh() const {
48         return this->mass;
49     }
50
51     void copyFromForceParticleMesh(const PS::F64vec & acc_pm) {
52         this->acc_pm = acc_pm;
53     }
54
55     /*
56     void writeParticleBinary(FILE *fp) {
57         int count;
58         count = 0;
59
60         count += fwrite(&mass,   sizeof(PS::F32),1,fp);
61         count += fwrite(&eps,    sizeof(PS::F32),1,fp);
62         count += fwrite(&pos[0], sizeof(PS::F64),1,fp);
63         count += fwrite(&pos[1], sizeof(PS::F64),1,fp);
64         count += fwrite(&pos[2], sizeof(PS::F64),1,fp);
```

```
65          count += fwrite(&vel[0], sizeof(PS::F64),1,fp);
66          count += fwrite(&vel[1], sizeof(PS::F64),1,fp);
67          count += fwrite(&vel[2], sizeof(PS::F64),1,fp);
68      }
69      */
70      /*
71      int readParticleBinary(FILE *fp) {
72          int count;
73          count = 0;
74
75          count += fread(&mass,   sizeof(PS::F32),1,fp);
76          count += fread(&eps,    sizeof(PS::F32),1,fp);
77          count += fread(&pos[0], sizeof(PS::F64),1,fp);
78          count += fread(&pos[1], sizeof(PS::F64),1,fp);
79          count += fread(&pos[2], sizeof(PS::F64),1,fp);
80          count += fread(&vel[0], sizeof(PS::F64),1,fp);
81          count += fread(&vel[1], sizeof(PS::F64),1,fp);
82          count += fread(&vel[2], sizeof(PS::F64),1,fp);
83
84          return count;
85      }
86      */
87
88      void writeParticleBinary(FILE *fp) {
89          PS::F32 x = pos[0];
90          PS::F32 y = pos[1];
91          PS::F32 z = pos[2];
92          PS::F32 vx = vel[0];
93          PS::F32 vy = vel[1];
94          PS::F32 vz = vel[2];
95          PS::S32 i = id;
96          PS::S32 m = mass;
97          fwrite(&x, sizeof(PS::F32),1,fp);
98          fwrite(&vx, sizeof(PS::F32),1,fp);
99          fwrite(&y, sizeof(PS::F32),1,fp);
100         fwrite(&vy, sizeof(PS::F32),1,fp);
101         fwrite(&z, sizeof(PS::F32),1,fp);
102         fwrite(&vz, sizeof(PS::F32),1,fp);
103         //fwrite(&mass,   sizeof(PS::F32),1,fp);
104         fwrite(&m,    sizeof(PS::F32),1,fp);
105         fwrite(&i,    sizeof(PS::F32),1,fp);
106         //fwrite(&id,   sizeof(PS::F32),1,fp);
107     }
108
109
110     // for API of FDPS
111     // in snapshot, L unit is Mpc/h, M unit is Msun, v unit is km/s
112     void readBinary(FILE *fp){
113         static PS::S32 ONE = 1;
114         static bool is_little_endian = *reinterpret_cast<char*>(&ONE) ==
                ONE;
115         static const PS::F64 Mpc_m = 3.08567e22; // unit is m
116         static const PS::F64 Mpc_km = 3.08567e19; // unit is km
117         static const PS::F64 Msun_kg = 1.9884e30; // unit is kg
118         static const PS::F64 G = 6.67428e-11; // m^3*kg^-1*s^-2
```

```
119         static const PS::F64 Cl = 1.0 / FPtreepm::Lbnd;
120         static const PS::F64 Cv = 1.0 / (FPtreepm::Lbnd * FPtreepm::H0);
121         static const PS::F64 Cm = 1.0 / (pow(Mpc_m*FPtreepm::Lbnd, 3.0) /
                pow(Mpc_km/FPtreepm::H0, 2.0) / G / Msun_kg);
122         PS::F32 x, y, z, vx, vy, vz, m;
123         PS::S32 i;
124         fread(&x, 4, 1, fp);
125         fread(&vx, 4, 1, fp);
126         fread(&y, 4, 1, fp);
127         fread(&vy, 4, 1, fp);
128         fread(&z, 4, 1, fp);
129         fread(&vz, 4, 1, fp);
130         fread(&m,  4, 1, fp);
131         fread(&i,  4, 1, fp);
132         if( is_little_endian){
133             pos.x = x * Cl;
134             pos.y = y * Cl;
135             pos.z = z * Cl;
136             vel.x = vx * Cv;
137             vel.y = vy * Cv;
138             vel.z = vz * Cv;
139             mass = m * Cm;
140             //mass = m / 1.524e17;
141             id = i;
142         }
143         else{
144             pos.x = reverseEndian(x) * Cl;
145             pos.y = reverseEndian(y) * Cl;
146             pos.z = reverseEndian(z) * Cl;
147             vel.x = reverseEndian(vx) * Cv;
148             vel.y = reverseEndian(vy) * Cv;
149             vel.z = reverseEndian(vz) * Cv;
150             mass = reverseEndian(m) * Cm;
151             //mass = reverseEndian(m) / 1.524e17;
152             id = reverseEndian(i);
153         }
154     }
155
156     // for API of FDPS
157     void writeBinary(FILE *fp){
158         static const PS::F64 Mpc_m = 3.08567e22; // unit is m
159         static const PS::F64 Mpc_km = 3.08567e19; // unit is km
160         static const PS::F64 Msun_kg = 1.9884e30; // unit is kg
161         static const PS::F64 G = 6.67428e-11; // m^3*kg^-1*s^-2
162         static const PS::F64 Cl = FPtreepm::Lbnd;
163         static const PS::F64 Cv = (FPtreepm::Lbnd * FPtreepm::H0);
164         static const PS::F64 Cm = (pow(Mpc_m*FPtreepm::Lbnd, 3.0) / pow(
                Mpc_km/FPtreepm::H0, 2.0) / G / Msun_kg);
165         PS::F32vec x = pos * Cl;
166         PS::F32vec v = vel * Cv;
167         PS::F32 m = mass * Cm;
168         PS::S32 i = id;
169         fwrite(&x.x,   sizeof(PS::F32), 1, fp);
170         fwrite(&v.x,   sizeof(PS::F32), 1, fp);
171         fwrite(&x.y,   sizeof(PS::F32), 1, fp);
```

```
172         fwrite(&v.y,    sizeof(PS::F32), 1, fp);
173         fwrite(&x.z,    sizeof(PS::F32), 1, fp);
174         fwrite(&v.z,    sizeof(PS::F32), 1, fp);
175         fwrite(&m,    sizeof(PS::F32), 1, fp);
176         fwrite(&i,    sizeof(PS::S32), 1, fp);
177     }
178
179     PS::F64 calcDtime(run_param &this_run) {
180     PS::F64 dtime_v, dtime_a, dtime;
181     PS::F64 vnorm, anorm;
182     vnorm = sqrt(SQR(this->vel))+TINY;
183     anorm = sqrt(SQR(this->acc+this->acc_pm))+TINY;
184
185     dtime_v = this->eps/vnorm;
186     dtime_a = sqrt(this->eps/anorm)*CUBE(this_run.anow);
187
188     dtime = fmin(0.5*dtime_v, dtime_a);
189
190     return dtime;
191   }
192 };
```

6.2.3.2 EssentialParticleI type

You must define a EssentialParticleI type and it must have all physical quantities as member variables that i particle should have. Listing 52 shows the implementation of EssentialParticleI type in the sample code. It must have member functions `copyFromFP()` (to copy data from a FullParticle object described above) and `getPos()` (to get the position of particle).

Listing 52: EssentialParticleI type

```
1  class EPItreepm {
2  public:
3    PS::S64    id;
4    PS::F32    eps;
5    PS::F64vec pos;
6
7    PS::F64vec getPos() const {
8      return this->pos;
9    }
10
11   void copyFromFP(const FPtreepm & fp) {
12     this->id = fp.id;
13     this->eps = fp.eps;
14     this->pos = fp.pos;
15   }
16
17 };
```

6.2.3.3 EssentialParticleJ type

You must define a EssentialParticleJ type and it must have all physical quantities as member variables that j particle should have when the PP part of Force calculation is performed. Note that it is possible to define EssentialParticleI type so that it operates as EssentialParticleJ type as in the sample code for P^3M method (see § 6.1). Listing 53 shows the implementation of EssentialParticleJ type in this sample code. EssentialParticleJ type should have the following member functions:

`getPos()`
: required for FDPS to get the position of particle

`getCharge()`
: required for FDPS to get the mass of particle

`copyFromFP()`
: required for FDPS to copy data from FullParticle type to EssentialParticleJ type

`getRSearch()`
: required for FDPS to get the cutoff radius

`setPos()`
: required for FDPS to write the position of particle

Listing 53: EssentialParticleJ type

```
1  class EPJtreepm {
2  public:
3    PS::S64    id;
4    PS::F64vec pos;
5    PS::F64    mass;
6    //  PS::F64    rcut;
7
8    PS::F64vec getPos() const {
9      return this->pos;
10   }
11
12   PS::F64 getCharge() const {
13     return this->mass;
14   }
15
16   void copyFromFP(const FPtreepm & fp) {
17     this->id = fp.id;
18     this->mass = fp.mass;
19     this->pos = fp.pos;
20   }
21
22   PS::F64 getRSearch() const {
23     PS::F64 rcut = 3.0/SIZE_OF_MESH;
24     return rcut;
25   }
26
27   void setPos(const PS::F64vec pos_new) {
28     this->pos = pos_new;
29   }
30 };
```

6.2.3.4 Force type

You must define a Force type and it must have all physical quantities that obtained as the results of the PP part of Force calculation. Listing 54 shows the implementation of Force type. It must have a member function `clear()` in order to initialize or zero-clear the member variables that stored the results of accumulation operations.

Listing 54: Force type

```
class Result_treepm {
public:
  PS::F32vec acc;
  PS::F32    pot;

  void clear() {
    acc = 0.0;
    pot = 0.0;
  }
};
```

6.2.3.5 calcForceEpEp type

You must define a calcForceEpEp type and it must contain actual code for the PP part of Force calculation. Listing 55 shows the implementation of calcForceEpEp type. In the sample code, calcForceEpEp type is implemented as a template function. Depending on the value of the macro `ENABLE_PHANTOM_GRAPE_X86`, which determines whether to use the Phantom-GRAPE library, a different template function is used. In both cases, the arguments of the functions is an array of EssentialParticleI variables, the number of EssentialParticleI variables, an array of EssentialParticleJ variables, the number of EssentialParticleJ variables, an array of Force variables.

Listing 55: calcForceEpEp type

```
template <class TPJ>
class calc_pp_force {
public:
    void operator () (EPItreepm *iptcl,
                      const PS::S32 ni,
                      TPJ *jptcl,
                      const PS::S32 nj,
                      Result_treepm *ppforce) {
        for (PS::S32 i=0;i < ni;i++) {
            PS::F64 eps2 = SQR(iptcl[i].eps);
            for (PS::S32 j=0; j < nj;j++) {
                PS::F64vec dr = iptcl[i].pos - jptcl[j].pos;
                PS::F64 rsq = dr*dr;
                PS::F64 rad = sqrt(rsq+eps2);
                PS::F64 gfact = gfactor_S2(rad, 3.0/SIZE_OF_MESH);
                PS::F64 rinv  = 1.0/rad;
                PS::F64 mrinv3 = jptcl[j].mass*CUBE(rinv);
                ppforce[i].acc -= dr*gfact*mrinv3;
            }
        }
    }
```

```
22 };
```

The PP part of the TreePM method is a two-body interaction with cutoff as in the P^{3}M method. Hence, a cutoff function is involved in the calculation of gravitational acceleration. As explained in § 6.1.3.4, the cutoff function must be the one that is constructed assuming that the particle shape function is $S2(r)$ (Hockney & Eastwood 1988). The cutoff function is implemented as the function `gfactor_S2()` for the case where the Phantom-GRAPE library is not used. In using the Phantom-GRAPE library, you do not have to implement the cutoff function because the library computes the interaction taking into account cutoff. In this case, you must call the library's API `pg5_gen_s2_force_table()` before the Force calculation in order to give the value of the cutoff radius to the library. In the sample code, the call of the API is performed at the main function:

```
1 #ifdef ENABLE_PHANTOM_GRAPE_X86
2     //g5_open();
3     pg5_gen_s2_force_table(EPS_FOR_PP, 3.0/SIZE_OF_MESH);
4 #endif
```

6.2.4 Main body of the program

In this section, we explain in detail the main body of the program. Before going into details, we first give a simple explanation about the content and the structure of the sample code. As described in the beginning of § 6.2, this code performs a cosmological N-body simulation using the TreePM method. The code supports three different types of initial condition files:

(a) Initial condition files used in the Santa Barbara Cluster Comparison Test (Frenk et al.[1999, ApJ, 525, 554]). These initial condition files are available at http://particle.riken.jp/~fdps/data/sb/ic_sb128.tar ($N = 128^3$) and http://particle.riken.jp/~fdps/data/sb/ic_sb256.tar) ($N = 256^3$)
(b) Initial condition files described in the same format as the above test
(c) Random distribution of particles

You must pass the absolute PATH of an initial condition file as the runtime command-line argument. Then, the program reads the given initial condition file and automatically identifies the type of initial condition ((a)-(c)). After setting the initial condition, the code numerically integrates the motions of particles to the finish time (specified by the redshift z) described in the file using the TreePM method. For the details of the format of initial condition file, please see `$(FDPS)/sample/c++/treepm/README_en.txt`. Note that an example of the initial condition file for the case (a) is given at `$(FDPS)/sample/c++/treepm/result/input.para`.

The structure of the sample code is as follows::

(1) Create and initialize FDPS objects
(2) Initialize the Phantom-GRAPE library (if needed)
(3) Read the initial condition file
(4) Integrate the motions of particles in time to the finish time

In the followings, we explain in detail each step described above.

6.2.4.1 Initialization and Termination of FDPS

First, you must initialize FDPS by the following code.

Listing 56: Initialization of FDPS

```
1 PS::Initialize(argc, argv);
```

Once started, FDPS should be terminated explicitly. In this sample, FDPS is terminated just before the termination of the program. Hence, you need to write the following code at the end of the main function.

Listing 57: Termination of FDPS

```
1 PS::Finalize();
```

6.2.4.2 Creation and Initialization of FDPS objects

After the initialization of FDPS, a user need to create the objects used to talk to FDPS. In this section, we describe how to create and initialize these objects.

6.2.4.2.1 Creation of necessary FDPS objects

In the calculation using the TreePM method, we must create objects of the FullParticle class, the DomainInfo class, the TreeForForceLong class (used in the PP part), and the ParticleMesh class (used in the PM part). In the sample code, these objects are created in the main function described in `treepm.cpp`:

Listing 58: Creation of FDPS objects

```
1 int main(int argc, char **argv)
2 {
3    PS::PM::ParticleMesh pm;
4    PS::ParticleSystem<FPtreepm> ptcl;
5    PS::DomainInfo domain_info;
6    PS::TreeForForceLong<Result_treepm, EPItreepm, EPJtreepm>::
           MonopoleWithCutoff treepm_tree;
7
8 }
```

Note that the above code is the one that is constructed by collecting the parts of object creation.

6.2.4.2.2 Initialization of FDPS objects

Almost all of FDPS objects must be initialized before they are used in a user code. Of four objects described in previous section, the ParticleMesh object is the only object that does not require an explicit initialization. The initialization of the other objects is done by the `initialize` method. In the following, we show an excerpt of the sample code where the initializations are performed:

Listing 59: Initialization of FDPS objects

```
1  int main(int argc, char **argv)
2  {
3     // Initialize ParticleSystem
4     ptcl.initialize();
5
6     // Initialize DomainInfo
7     domain_info.initialize();
8     domain_info.setBoundaryCondition(PS::BOUNDARY_CONDITION_PERIODIC_XYZ);
9     domain_info.setPosRootDomain(PS::F64vec(0.0, 0.0, 0.0),
10                                 PS::F64vec(1.0, 1.0, 1.0));
11
12    // Initialize Tree
13    treepm_tree.initialize(3*ptcl.getNumberOfParticleGlobal(),
14                           this_run.theta);
15 }
```

- The initialization of a ParticleSystem object is simply done by calling `initialize` method without any arguments.
- As for a DomainInfo object, we must set the boundary condition and the size of the simulation box after calling `initialize` method. These are done by calling `setBoundaryCondition` and `setPosRootDomain` methods.
- We must pass a rough number of particles to the `initialize` method of a TreeForForceLong object as the first argument. In this sample code, we pass a value three times larger than the total number of the particles. We can specify the value of the opening angle criterion θ used in the force calculation of the tree method via the second argument. Note that the object `this_run` is used to store a set of parameters such as θ that control the simulation.

6.2.4.3 Initial Condition

An parameter file that specifies an initial condition is read in the function `read_param_file()` described in the main function:

```
1  read_param_file(ptcl, this_run, argv[1]);
```

This function reads the parameter file specified by the command line arguments and sets the particle information such as mass and position to the ParticleSystem object based on the parameter file. After that, the sample code performs domain decomposition and particle exchange using FDPS APIs. In the following, we explain these APIs in detail.

6.2.4.3.1 Domain Decomposition

The sample code first perform domain decomposition, which is done by calling the `decomposeDomainAll` method (see the `main` function):

Listing 60: Domain Decomposition

```
1  domain_info.decomposeDomainAll(ptcl);
```

This method divides the entire of the domain based on a given particle distribution. Hence, we need to pass a ParticleSystem object to this method.

6.2.4.3.2 Particle Exchange

Then, the code performs particle exchange, which is done by calling the `exchangeParticle` method (see the `main` function):

Listing 61: Particle Exchange

```
1 ptcl.exchangeParticle(domain_info);
```

where we pass a DomainInfo object to this method because the method needs to know the information of domain decomposition in advance.

6.2.4.4 Interaction Calculation

After that, the code performs interaction calculation to determine the accelerations at the initial time. In the following, we show our implementation of interaction calculation. In this code, the `calcForceAllAndWriteBack` method of the TreeForForceLong object is used to calculate the PP part of the force calculation. This method automatically stores the results into the member variable `acc` of the ParticleSystem object. As for the PM part of the force calculation, we use the `calcForceAllAndWriteBack` method of the ParticleMesh object. Likewise, this method automatically stores the results of the PM part into the member variable `acc_pm` of the ParticleSystem object.

Listing 62: Interaction Calculation

```
1 //* PP part
2 treepm_tree.calcForceAllAndWriteBack
3     (calc_pp_force<EPJtreepm>(),
4      calc_pp_force<PS::SPJMonopoleCutoff>(),
5      ptcl,
6      domain_info);
7
8 //* PM part
9 pm.calcForceAllAndWriteBack(ptcl, domain_info);
```

6.2.4.5 Time Integration

The sample code uses the Leapfrog time integrator to perform time integration (the details of the method is described in § 4.1.3.3.4). The $D(\cdot)$ operator, which integrates the positions of particles in time, is implemented as the function `drift_ptcl`, while the $K(\cdot)$ operator, which integrates the velocities of particles in time, is implemented as the function `kick_ptcl`. The effects of cosmic expansion is taken into account in the function `kick_ptcl`. The time evolution of the scale factor and the Hubble parameter is done by the `update_expansion` method of the `this_run` object.

6.2.5 Compile

As explained in README.txt, you must edit `Makefile` in `src` directory appropriately to adapt to your computer environment. Then, run `make` command to compile the sample code. Note that this code uses FFTW library and therefore you have to install it in advance. The execution file `treepm` will be created if the compilation is succeeded.

6.2.6 Execution

We must run the program using MPI with the number of MPI processes is equal to or greater than 2, because of the specification of FDPS extension "ParticleMesh" module. Therefore, you should run the following command:

```
$ MPIRUN -np NPROC ./treepm
```

where "MPIRUN" represents the command to run a program using MPI such as `mpirun` or `mpiexec`, and "NPROC" is the number of MPI processes.

6.2.7 Confirmation of the result

After the simulation is completed, the results will be output at the directory specified in the parameter file. Figure 4 shows the time evolution of column density distribution of dark matter in a Santa Barbara Cluster Comparison test with 256^3 particles.

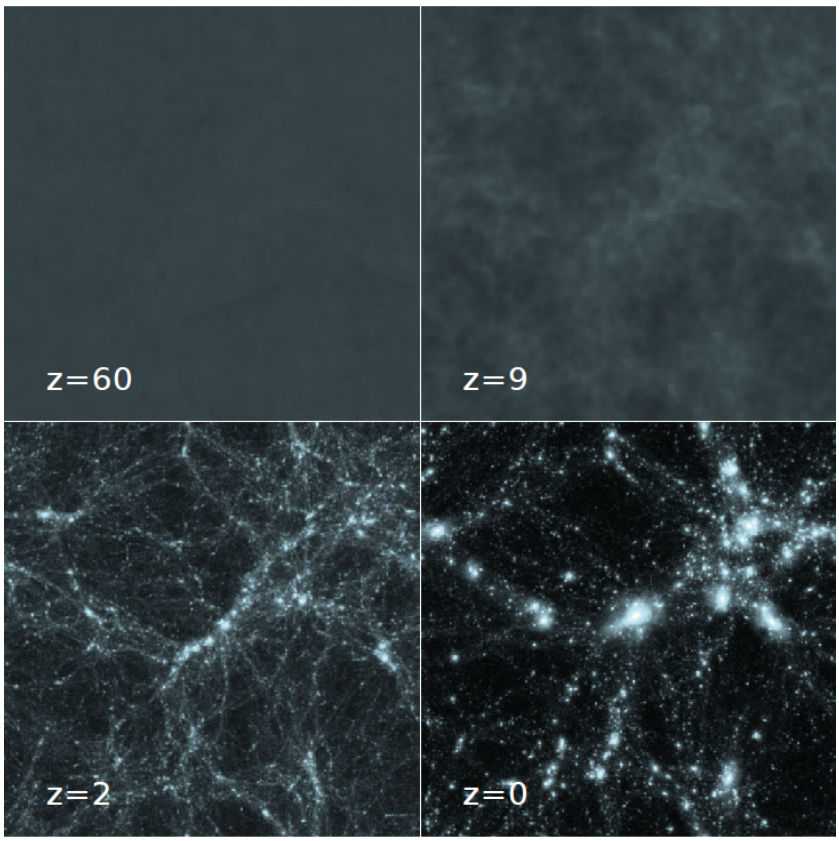

Figure 4: Time evolution of particle density of Santa Barbara Cluster Comparison test (the number of particles is 256^3)

7 User Supports

We accept questions and comments on FDPS at the following mail address:
fdps-support@mail.jmlab.jp
Please provide us with the following information.

7.1 Compile-time problem

- Compiler environment (version of the compiler, compile options etc)
- Error message at the compile time
- (if possible) the source code

7.2 Run-time problem

- Run-time environment
- Run-time error message
- (if possible) the source code

7.3 Other cases

For other problems, please do not hesitate to contact us. We sincerely hope that you'll find FDPS useful for your research.

8 License

This software is MIT licensed. Please cite Iwasawa et al. (2016, Publications of the Astronomical Society of Japan, 68, 54) if you use the standard functions only.

The extended feature "Particle Mesh" is implemented by using a module of GreeM code (Developers: Tomoaki Ishiyama and Keigo Nitadori) (Ishiyama, Fukushige & Makino 2009, Publications of the Astronomical Society of Japan, 61, 1319; Ishiyama, Nitadori & Makino, 2012 SC'12 Proceedings of the International Conference on High Performance Computing, Networking Stroage and Analysis, No. 5). GreeM code is developed based on the code in Yoshikawa & Fukushige (2005, Publications of the Astronomical Society of Japan, 57, 849). Please cite these three literatures if you use the extended feature "Particle Mesh".

Please cite Tanikawa et al.(2012, New Astronomy, 17, 82) and Tanikawa et al.(2012, New Astronomy, 19, 74) if you use the extended feature "Phantom-GRAPE for x86".