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 TODO

2 Change log

- 2015/03/17 English version created
- 2015/06/04 Spell-checked complete version
- 2016/01/18 Desctription of GPU version added (section 4.4.1.8

3 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.

4 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.

4.1 Environment

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

4.2 Necessary software

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

4.2.1 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.

4.2.1.1 Single thread

- make
- A C++ compiler (We have tested with gcc version 4.4.5 and K compiler version 1.2.0)

4.2.1.2 Parallel processing

4.2.1.2.1 OpenMP

- make
- A C++ compiler with OpenMP support (We have tested with gcc version 4.4.5 and K compiler version 1.2.0)

4.2.1.2.2 MPI

- make
- A C++ compiler which supports MPI version 1.3 or later. (We have tested with Open MPI 1.8.1 and K compiler version 1.2.0)

4.2.1.2.3 MPI+OpenMP

- make
- A C++ compiler which supports OpenMP and MPI version 1.3 or later. (We have tested with Open MPI 1.8.1 and K compiler version 1.2.0)

4.2.2 Extensions

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

4.2.2.1 Particle Mesh

- make
- A C++ compiler which supports OpenMP and MPI version 1.3 or later. (We have tested with Open MPI 1.8.1)
- FFTW 3.3 or later

4.3 Install

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

4.3.1 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.

4.3.1.1 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
  - 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
```

4.3.1.2 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.

4.3.2 How to build

There is no need for configure or setup.

4.4 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.

4.4.1 gravitational N-body simulation

4.4.1.1 Summary

Through the following steps one can use this sample.

- Move to the directory $(FDPS)/sample/nbody. Here, $(FDPS) denotes the highest-level directory for FDPS. It is not necessary d to set environmental variable FDPS. 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/nbody)
- run 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.

4.4.1.2 Move to the directory with the sample code

Move to $(FDPS)/sample/nbody using chdir.

4.4.1.3 Edit Makefile

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

- Without OpenMP or 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, remove "-fopenmp"
- 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, remove "-fopenmp"
  - uncomment the line "CFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL"

4.4.1.4 run make

Type "make" to run make.

4.4.1.5 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. ********
```

4.4.1.6 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 10,000, try: (without MPI)

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

4.4.1.7 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/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

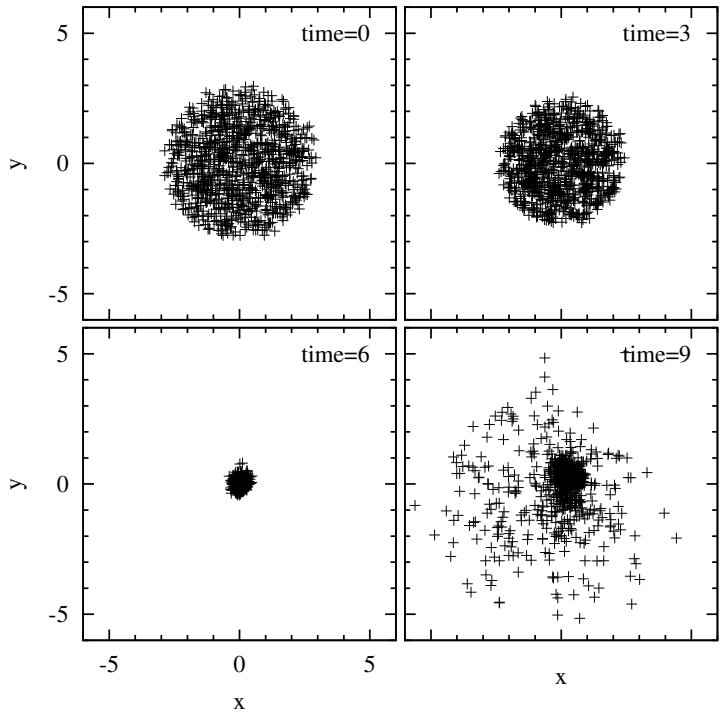

Figure 1:

without Phantom GRAPE by a factor a bit less than five. The following is the sample command line:

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

4.4.1.8 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/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.

4.4.2 SPH simulation code

4.4.2.1 Summary

Through the following steps one can use this sample.

- Move to the directory $(FDPS)/sample/sph
- Edit Makefile in the current directory ($(FDPS)/sample/sph)
- run make command to create the executable "sph.out"

- run sph.out
- Check the output

4.4.2.2 Move to the directory with the sample code

Move to $(FDPS)/sample/sph using chdir.

4.4.2.3 Edit Makefile

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

- Without OpenMP or 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, remove "-fopenmp"
- 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, remove "-fopenmp"
  - uncomment the line "CFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL"

4.4.2.4 run make

Type "make" to run make.

4.4.2.5 run the sample code

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

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

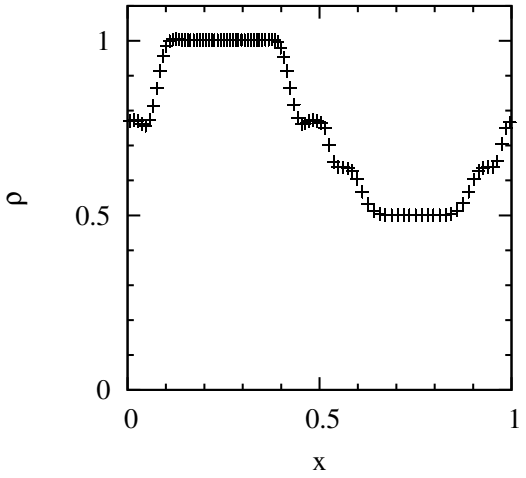

Figure 2:

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

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

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

4.4.2.6 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), 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.dat" using 3:9
```

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

5 How to Use

In this section, we explain in detail the contents of the sample codes shown in previous section (§ 4). Especially, we focus on classes that need to be defined by the users and how to use various APIs of FDPS.

5.1 Gravitational N-body simulation code

5.1.1 Working directory

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

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

5.1.2 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.

5.1.2.1 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  };
```

5.1.2.2 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 }
```

5.1.3 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`.

5.1.3.1 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();
```

5.1.3.2 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.

5.1.3.2.1 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;
```

5.1.3.2.2 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 § 9.2.1 of the specification of FDPS.

5.1.3.2.3 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();
```

5.1.3.2.4 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.

5.1.3.3 Time integration loop

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

5.1.3.3.1 *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.

5.1.3.3.2 *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);
```

5.1.3.3.3 *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.

5.1.3.3.4 *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);
```

5.1.4 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
```

5.2 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.

5.2.1 Working directory

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

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

5.2.2 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>
```

5.2.3 User-defined classes

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

5.2.3.1 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               "%ld\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              "%ld\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 };
```

5.2.3.2 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  };
```

5.2.3.3 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 };
```

5.2.3.4 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 };
```

5.2.4 The main body of the user program

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

5.2.4.1 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();
```

5.2.4.2 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.

5.2.4.2.1 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;
```

5.2.4.2.2 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));
```

5.2.4.2.3 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();
```

5.2.4.2.4 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());
```

5.2.4.3 Time integration loop

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

5.2.4.3.1 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);
```

5.2.4.3.2 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);
```

5.2.4.3.3 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);
```

5.2.5 Compilation of the program

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

```
$ make
```

5.2.6 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.

5.2.7 Log and output files

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

5.2.8 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
```

6 Sample Codes

6.1 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 5. 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
9  /* Parameters */
10 const short int Dim = 3;
11 const PS::F64 SMTH = 1.2;
12 const PS::U32 OUTPUT_INTERVAL = 10;
13 const PS::F64 C_CFL = 0.3;
14
15 /* Kernel Function */
16 const PS::F64 pi = atan(1.0) * 4.0;
17 const PS::F64 kernelSupportRadius = 2.5;
18
19 PS::F64 W(const PS::F64vec dr, const PS::F64 h){
20    const PS::F64 H = kernelSupportRadius * h;
21    const PS::F64 s = sqrt(dr * dr) / H;
22    const PS::F64 s1 = (1.0 - s < 0) ? 0 : 1.0 - s;
23    const PS::F64 s2 = (0.5 - s < 0) ? 0 : 0.5 - s;
24    PS::F64 r_value = pow(s1, 3) - 4.0 * pow(s2, 3);
25    //if # of dimension == 3
26    r_value *= 16.0 / pi / (H * H * H);
27    return r_value;
28 }
29
30 PS::F64vec gradW(const PS::F64vec dr, const PS::F64 h){
31    const PS::F64 H = kernelSupportRadius * h;
32    const PS::F64 s = sqrt(dr * dr) / H;
33    const PS::F64 s1 = (1.0 - s < 0) ? 0 : 1.0 - s;
34    const PS::F64 s2 = (0.5 - s < 0) ? 0 : 0.5 - s;
35    PS::F64 r_value = - 3.0 * pow(s1, 2) + 12.0 * pow(s2, 2);
36    //if # of dimension == 3
37    r_value *= 16.0 / pi / (H * H * H);
38    return dr * r_value / (sqrt(dr * dr) * H + 1.0e-6 * h);
39 }
40
41 /* Class Definitions */
42 //** Force Class (Result Class)
43 class Dens{
44    public:
45    PS::F64 dens;
```

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

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

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

```
205             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;
206          }
207          hydro[i].dt = C_CFL * 2.0 * ep_i[i].smth / v_sig_max;
208       }
209    }
210 };
211
212 void SetupIC(PS::ParticleSystem<FP>& sph_system, PS::F64 *end_time,
       boundary *box){
213    // Place SPH particles
214    std::vector<FP> ptcl;
215    const PS::F64 dx = 1.0 / 128.0;
216    box->x = 1.0;
217    box->y = box->z = box->x / 8.0;
218    PS::S32 i = 0;
219    for(PS::F64 x = 0 ; x < box->x * 0.5 ; x += dx){
220       for(PS::F64 y = 0 ; y < box->y ; y += dx){
221          for(PS::F64 z = 0 ; z < box->z ; z += dx){
222             FP ith;
223             ith.pos.x = x;
224             ith.pos.y = y;
225             ith.pos.z = z;
226             ith.dens = 1.0;
227             ith.mass = 0.75;
228             ith.eng  = 2.5;
229             ith.id   = i++;
230             ith.smth = 0.012;
231             ptcl.push_back(ith);
232          }
233       }
234    }
235    for(PS::F64 x = box->x * 0.5 ; x < box->x * 1.0 ; x += dx * 2.0){
236       for(PS::F64 y = 0 ; y < box->y ; y += dx){
237          for(PS::F64 z = 0 ; z < box->z ; z += dx){
238             FP ith;
239             ith.pos.x = x;
240             ith.pos.y = y;
241             ith.pos.z = z;
242             ith.dens = 0.5;
243             ith.mass = 0.75;
244             ith.eng  = 2.5;
245             ith.id   = i++;
246             ith.smth = 0.012;
247             ptcl.push_back(ith);
248          }
249       }
250    }
251    for(PS::U32 i = 0 ; i < ptcl.size() ; ++ i){
252       ptcl[i].mass = ptcl[i].mass * box->x * box->y * box->z / (PS::F64)(
              ptcl.size());
253    }
254    std::cout << "#␣of␣ptcls␣is...␣" << ptcl.size() << std::endl;
255    // Scatter SPH particles
256    assert(ptcl.size() % PS::Comm::getNumberOfProc() == 0);
```

```
257     const PS::S32 numPtclLocal = ptcl.size() / PS::Comm::getNumberOfProc();
258     sph_system.setNumberOfParticleLocal(numPtclLocal);
259     const PS::U32 i_head = numPtclLocal * PS::Comm::getRank();
260     const PS::U32 i_tail = numPtclLocal * (PS::Comm::getRank() + 1);
261     for(PS::U32 i = 0 ; i < ptcl.size() ; ++ i){
262        if(i_head <= i && i < i_tail){
263           const PS::U32 ii = i - numPtclLocal * PS::Comm::getRank();
264           sph_system[ii] = ptcl[i];
265        }
266     }
267     // Set the end time
268     *end_time = 0.12;
269     // Fin.
270     std::cout << "setup..." << std::endl;
271 }
272
273 void Initialize(PS::ParticleSystem<FP>& sph_system){
274     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
275        sph_system[i].setPressure();
276     }
277 }
278
279 PS::F64 getTimeStepGlobal(const PS::ParticleSystem<FP>& sph_system){
280     PS::F64 dt = 1.0e+30; //set VERY LARGE VALUE
281     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
282        dt = std::min(dt, sph_system[i].dt);
283     }
284     return PS::Comm::getMinValue(dt);
285 }
286
287 void InitialKick(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
288     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
289        sph_system[i].vel_half = sph_system[i].vel + 0.5 * dt * sph_system[i
               ].acc;
290        sph_system[i].eng_half = sph_system[i].eng + 0.5 * dt * sph_system[i
               ].eng_dot;
291     }
292 }
293
294 void FullDrift(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
295     // time becomes t + dt;
296     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
297        sph_system[i].pos += dt * sph_system[i].vel_half;
298     }
299 }
300
301 void Predict(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
302     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
303        sph_system[i].vel += dt * sph_system[i].acc;
304        sph_system[i].eng += dt * sph_system[i].eng_dot;
305     }
306 }
307
308 void FinalKick(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
309     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
```

```
310         sph_system[i].vel = sph_system[i].vel_half + 0.5 * dt * sph_system[i
                ].acc;
311         sph_system[i].eng = sph_system[i].eng_half + 0.5 * dt * sph_system[i
                ].eng_dot;
312     }
313 }
314
315 void setPressure(PS::ParticleSystem<FP>& sph_system){
316     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
317         sph_system[i].setPressure();
318     }
319 }
320
321 void CheckConservativeVariables(const PS::ParticleSystem<FP>& sph_system){
322     PS::F64vec Mom=0.0; // total momentum
323     PS::F64    Eng=0.0; // total enegry
324     for(PS::S32 i = 0; i < sph_system.getNumberOfParticleLocal(); ++ i){
325         Mom += sph_system[i].vel * sph_system[i].mass;
326         Eng += (sph_system[i].eng + 0.5 * sph_system[i].vel * sph_system[i].
                vel)
327               * sph_system[i].mass;
328     }
329     Eng = PS::Comm::getSum(Eng);
330     Mom = PS::Comm::getSum(Mom);
331      if(PS::Comm::getRank() == 0){
332          printf("%.16e\n", Eng);
333          printf("%.16e\n", Mom.x);
334          printf("%.16e\n", Mom.y);
335          printf("%.16e\n", Mom.z);
336      }
337 }
338
339 int main(int argc, char* argv[]){
340     // Initialize FDPS
341     PS::Initialize(argc, argv);
342     // Display # of MPI processes and threads
343     PS::S32 nprocs = PS::Comm::getNumberOfProc();
344     PS::S32 nthrds = PS::Comm::getNumberOfThread();
345     std::cout << "========================================" << std::endl
346               << "␣This␣is␣a␣sample␣program␣of␣"             << std::endl
347               << "␣Smoothed␣Particle␣Hydrodynamics␣on␣FDPS!"  << std::endl
348               << "␣#␣of␣processes␣is␣" << nprocs              << std::endl
349               << "␣#␣of␣thread␣is␣␣␣␣" << nthrds              << std::endl
350               << "========================================" << std::endl
                      ;
351     // Make an instance of ParticleSystem and initialize it
352     PS::ParticleSystem<FP> sph_system;
353     sph_system.initialize();
354     // Define local variables
355     PS::F64 dt, end_time;
356     boundary box;
357     // Make an initial condition and initialize the particle system
358     SetupIC(sph_system, &end_time, &box);
359     Initialize(sph_system);
360     // Make an instance of DomainInfo and initialize it
```

```
361     PS::DomainInfo dinfo;
362     dinfo.initialize();
363     // Set the boundary condition
364     dinfo.setBoundaryCondition(PS::BOUNDARY_CONDITION_PERIODIC_XYZ);
365     dinfo.setPosRootDomain(PS::F64vec(0.0, 0.0, 0.0),
366                            PS::F64vec(box.x, box.y, box.z));
367     // Perform domain decomposition
368     dinfo.decomposeDomainAll(sph_system);
369     // Exchange the SPH particles between the (MPI) processes
370     sph_system.exchangeParticle(dinfo);
371     // Make two tree structures
372     // (one is for the density calculation and
373     //  another is for the force calculation.)
374     PS::TreeForForceShort<Dens, EP, EP>::Gather dens_tree;
375     dens_tree.initialize(3 * sph_system.getNumberOfParticleGlobal());
376
377     PS::TreeForForceShort<Hydro, EP, EP>::Symmetry hydr_tree;
378     hydr_tree.initialize(3 * sph_system.getNumberOfParticleGlobal());
379     // Compute density, pressure, acceleration due to pressure gradient
380     dens_tree.calcForceAllAndWriteBack(CalcDensity(), sph_system, dinfo);
381     setPressure(sph_system);
382     hydr_tree.calcForceAllAndWriteBack(CalcHydroForce(), sph_system, dinfo)
            ;
383     // Get timestep
384     dt = getTimeStepGlobal(sph_system);
385     // Main loop for time integration
386     PS::S32 step = 0;
387     for(PS::F64 time = 0 ; time < end_time ; time += dt, ++ step){
388        // Leap frog: Initial Kick & Full Drift
389        InitialKick(sph_system, dt);
390        FullDrift(sph_system, dt);
391        // Adjust the positions of the SPH particles that run over
392        // the computational boundaries.
393        sph_system.adjustPositionIntoRootDomain(dinfo);
394        // Leap frog: Predict
395        Predict(sph_system, dt);
396        // Perform domain decomposition again
397        dinfo.decomposeDomainAll(sph_system);
398        // Exchange the SPH particles between the (MPI) processes
399        sph_system.exchangeParticle(dinfo);
400        // Compute density, pressure, acceleration due to pressure gradient
401        dens_tree.calcForceAllAndWriteBack(CalcDensity(), sph_system, dinfo)
               ;
402        setPressure(sph_system);
403        hydr_tree.calcForceAllAndWriteBack(CalcHydroForce(), sph_system,
               dinfo);
404        // Get a new timestep
405        dt = getTimeStepGlobal(sph_system);
406        // Leap frog: Final Kick
407        FinalKick(sph_system, dt);
408        // Output result files
409        if(step % OUTPUT_INTERVAL == 0){
410           FileHeader header;
411           header.time = time;
412           header.Nbody = sph_system.getNumberOfParticleGlobal();
```

```
413             char filename[256];
414             sprintf(filename, "result/%04d.txt", step);
415             sph_system.writeParticleAscii(filename, header);
416             if (PS::Comm::getRank() == 0){
417                 std::cout << "================================" << std::endl;
418                 std::cout << "output " << filename << "." << std::endl;
419                 std::cout << "================================" << std::endl;
420             }
421         }
422         // Output information to STDOUT
423         if (PS::Comm::getRank() == 0){
424             std::cout << "================================" << std::endl;
425             std::cout << "time = " << time << std::endl;
426             std::cout << "step = " << step << std::endl;
427             std::cout << "================================" << std::endl;
428         }
429         CheckConservativeVariables(sph_system);
430     }
431     // Finalize FDPS
432     PS::Finalize();
433     return 0;
434 }
```

6.2 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 5. 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 #ifdef PARTICLE_SIMULATOR_MPI_PARALLEL
122     etot = PS::Comm::getSum(etot_loc);
123     epot = PS::Comm::getSum(epot_loc);
124     ekin = PS::Comm::getSum(ekin_loc);
125 #else
126     etot = etot_loc;
127     epot = epot_loc;
128     ekin = ekin_loc;
129 #endif
130 }
131
132 void printHelp() {
133     std::cerr<<"o: dir name of output (default: ./result)"<<std::endl;
134     std::cerr<<"t: theta (default: 0.5)"<<std::endl;
135     std::cerr<<"T: time_end (default: 10.0)"<<std::endl;
136     std::cerr<<"s: time_step (default: 1.0 / 128.0)"<<std::endl;
137     std::cerr<<"d: dt_diag (default: 1.0 / 8.0)"<<std::endl;
138     std::cerr<<"D: dt_snap (default: 1.0)"<<std::endl;
139     std::cerr<<"l: n_leaf_limit (default: 8)"<<std::endl;
140     std::cerr<<"n: n_group_limit (default: 64)"<<std::endl;
141     std::cerr<<"N: n_tot (default: 1024)"<<std::endl;
142     std::cerr<<"h: help"<<std::endl;
143 }
144
```

```
145 void makeOutputDirectory(char * dir_name) {
146     struct stat st;
147     if(stat(dir_name, &st) != 0) {
148         PS::S32 ret_loc = 0;
149         PS::S32 ret     = 0;
150         if(PS::Comm::getRank() == 0)
151             ret_loc = mkdir(dir_name, 0777);
152         PS::Comm::broadcast(&ret_loc, ret);
153         if(ret == 0) {
154             if(PS::Comm::getRank() == 0)
155                 fprintf(stderr, "Directory␣\"%s\"␣is␣successfully␣made.\n"
                            , dir_name);
156         } else {
157             fprintf(stderr, "Directory␣%s␣fails␣to␣be␣made.\n", dir_name);
158             PS::Abort();
159         }
160     }
161 }
162
163 PS::F64 FPGrav::eps = 1.0/32.0;
164
165 int main(int argc, char *argv[]) {
166     std::cout<<std::setprecision(15);
167     std::cerr<<std::setprecision(15);
168
169     PS::Initialize(argc, argv);
170     PS::F32 theta = 0.5;
171     PS::S32 n_leaf_limit = 8;
172     PS::S32 n_group_limit = 64;
173     PS::F32 time_end = 10.0;
174     PS::F32 dt = 1.0 / 128.0;
175     PS::F32 dt_diag = 1.0 / 8.0;
176     PS::F32 dt_snap = 1.0;
177     char dir_name[1024];
178     PS::S64 n_tot = 1024;
179     PS::S32 c;
180     sprintf(dir_name,"./result");
181     opterr = 0;
182     while((c=getopt(argc,argv,"i:o:d:D:t:T:l:n:N:hs:")) != -1){
183         switch(c){
184         case 'o':
185             sprintf(dir_name,optarg);
186             break;
187         case 't':
188             theta = atof(optarg);
189             std::cerr << "theta␣=" << theta << std::endl;
190             break;
191         case 'T':
192             time_end = atof(optarg);
193             std::cerr << "time_end␣=␣" << time_end << std::endl;
194             break;
195         case 's':
196             dt = atof(optarg);
197             std::cerr << "time_step␣=␣" << dt << std::endl;
198             break;
```

```
199         case 'd':
200             dt_diag = atof(optarg);
201             std::cerr << "dt_diag = " << dt_diag << std::endl;
202             break;
203         case 'D':
204             dt_snap = atof(optarg);
205             std::cerr << "dt_snap = " << dt_snap << std::endl;
206             break;
207         case 'l':
208             n_leaf_limit = atoi(optarg);
209             std::cerr << "n_leaf_limit = " << n_leaf_limit << std::endl;
210             break;
211         case 'n':
212             n_group_limit = atoi(optarg);
213             std::cerr << "n_group_limit = " << n_group_limit << std::endl;
214             break;
215         case 'N':
216             n_tot = atoi(optarg);
217             std::cerr << "n_tot = " << n_tot << std::endl;
218             break;
219         case 'h':
220             if(PS::Comm::getRank() == 0) {
221                 printHelp();
222             }
223             PS::Finalize();
224             return 0;
225         default:
226             if(PS::Comm::getRank() == 0) {
227                 std::cerr<<"No such option! Available options are here."<<
                        std::endl;
228                 printHelp();
229             }
230             PS::Abort();
231         }
232     }
233
234     makeOutputDirectory(dir_name);
235
236     std::ofstream fout_eng;
237
238     if(PS::Comm::getRank() == 0) {
239         char sout_de[1024];
240         sprintf(sout_de, "%s/t-de.dat", dir_name);
241         fout_eng.open(sout_de);
242         fprintf(stdout, "This is a sample program of N-body simulation on
                FDPS!\n");
243         fprintf(stdout, "Number of processes: %d\n", PS::Comm::
                getNumberOfProc());
244         fprintf(stdout, "Number of threads per process: %d\n", PS::Comm::
                getNumberOfThread());
245     }
246
247     PS::ParticleSystem<FPGrav> system_grav;
248     system_grav.initialize();
249     PS::S32 n_loc    = 0;
```

```
250     PS::F32 time_sys = 0.0;
251     if(PS::Comm::getRank() == 0) {
252         setParticlesColdUniformSphere(system_grav, n_tot, n_loc);
253     } else {
254         system_grav.setNumberOfParticleLocal(n_loc);
255     }
256
257     const PS::F32 coef_ema = 0.3;
258     PS::DomainInfo dinfo;
259     dinfo.initialize(coef_ema);
260     dinfo.decomposeDomainAll(system_grav);
261     system_grav.exchangeParticle(dinfo);
262     n_loc = system_grav.getNumberOfParticleLocal();
263
264 #ifdef ENABLE_PHANTOM_GRAPE_X86
265     g5_open();
266     g5_set_eps_to_all(FPGrav::eps);
267 #endif
268
269     PS::TreeForForceLong<FPGrav, FPGrav, FPGrav>::Monopole tree_grav;
270     tree_grav.initialize(n_tot, theta, n_leaf_limit, n_group_limit);
271 #ifdef MULTI_WALK
272     const PS::S32 n_walk_limit = 200;
273     const PS::S32 tag_max = 1;
274     tree_grav.calcForceAllAndWriteBackMultiWalk(DispatchKernelWithSP,
275                                                 RetrieveKernel,
276                                                 tag_max,
277                                                 system_grav,
278                                                 dinfo,
279                                                 n_walk_limit);
280 #else
281     tree_grav.calcForceAllAndWriteBack(CalcGravity<FPGrav>,
282                                        CalcGravity<PS::SPJMonopole>,
283                                        system_grav,
284                                        dinfo);
285 #endif
286     PS::F64 Epot0, Ekin0, Etot0, Epot1, Ekin1, Etot1;
287     calcEnergy(system_grav, Etot0, Ekin0, Epot0);
288     PS::F64 time_diag = 0.0;
289     PS::F64 time_snap = 0.0;
290     PS::S64 n_loop = 0;
291     PS::S32 id_snap = 0;
292     while(time_sys < time_end){
293         if( (time_sys >= time_snap) || ( (time_sys + dt) - time_snap ) > (
                time_snap - time_sys) ){
294             char filename[256];
295             sprintf(filename, "%s/%04d.dat", dir_name, id_snap++);
296             FileHeader header;
297             header.time   = time_sys;
298             header.n_body = system_grav.getNumberOfParticleGlobal();
299             system_grav.writeParticleAscii(filename, header);
300             time_snap += dt_snap;
301         }
302
303         calcEnergy(system_grav, Etot1, Ekin1, Epot1);
```

```
304
305         if(PS::Comm::getRank() == 0){
306             if( (time_sys >= time_diag) || ( (time_sys + dt) - time_diag )
                       > (time_diag - time_sys) ){
307                 fout_eng << time_sys << "␣␣␣" << (Etot1 - Etot0) / Etot0
                           << std::endl;
308                 fprintf(stdout, "time:␣%10.7f␣energy␣error:␣%+e\n",
309                         time_sys, (Etot1 - Etot0) / Etot0);
310                 time_diag += dt_diag;
311             }
312         }
313
314
315         kick(system_grav, dt * 0.5);
316
317         time_sys += dt;
318         drift(system_grav, dt);
319
320         if(n_loop % 4 == 0){
321             dinfo.decomposeDomainAll(system_grav);
322         }
323
324         system_grav.exchangeParticle(dinfo);
325 #ifdef MULTI_WALK
326         tree_grav.calcForceAllAndWriteBackMultiWalk(DispatchKernelWithSP,
327                                                     RetrieveKernel,
328                                                     tag_max,
329                                                     system_grav,
330                                                     dinfo,
331                                                     n_walk_limit,
332                                                     true);
333 #else
334         tree_grav.calcForceAllAndWriteBack(CalcGravity<FPGrav>,
335                                            CalcGravity<PS::SPJMonopole>,
336                                            system_grav,
337                                            dinfo);
338 #endif
339
340         kick(system_grav, dt * 0.5);
341
342         n_loop++;
343     }
344
345 #ifdef ENABLE_PHANTOM_GRAPE_X86
346     g5_close();
347 #endif
348
349     PS::Finalize();
350     return 0;
351 }
```

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

The MIT license is applied to the FDPS software. Any work which used only the standard function of FDPS should cite Iwasawa et al. (2015 in prep), Tanikawa et al. (2016 in prep).

When Particle Mesh class is used, 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 Storage and Analysis, No. 5) should also be cited.

When Phantom-GRAPE for x86 is used, Tanikawa et al.(2012, New Astronomy, 17, 82) と Tanikawa et al.(2012, New Astronomy, 19, 74) should be cited.