FDPS Tutorial

Ataru Tanikawa, Daisuke Namekata, Natsuki Hosono, Masaki Iwasawa, Keigo Nitadori, Takayuki Munanushi, and Junichiro Makino

Paricle Simulator Research Team, RIKEN Center for Computational Science, 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)
- 2018/08/22 Description of N-body/SPH sample code added (Sec. 7)
- 2019/07/19 Description of N-body/SPH sample code updated (Sec. 7)

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, $\mathrm{P}^3\mathrm{M}$, 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 .

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
```

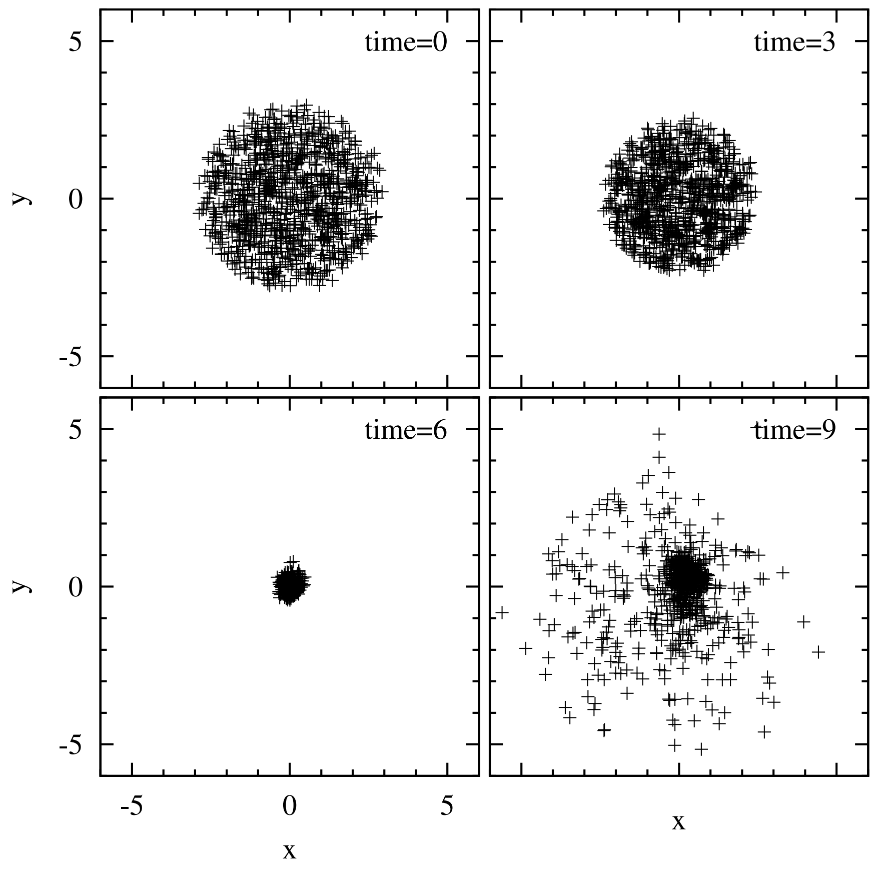

Figure 1:

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 `Makefile` in `$(FDPS)/sample/c++/nbody`and assign to CUDA_HOME in Makefile a value appropriate 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
```

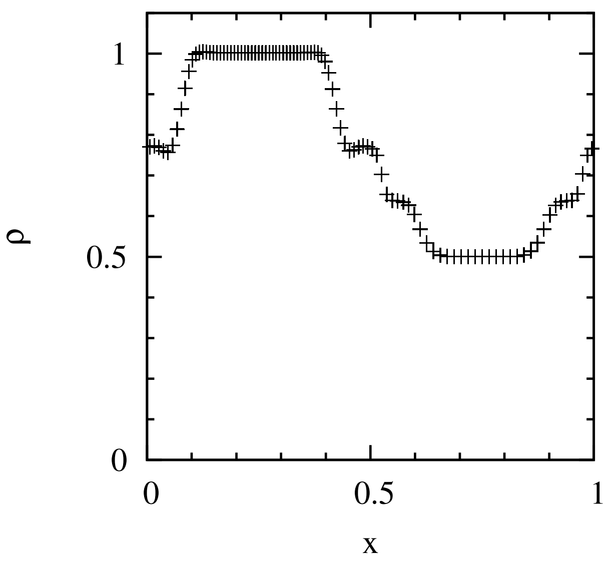

Figure 2:

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 describe the sample codes used in previous section (§ 3) in more detail. Especially, the explanation will focus mainly on derived data types that users must define (hereafter, **user-defined types**) and how to use APIs of FDPS. In order to avoid duplication of explanation, some matters are explained in § 4.1 only, where we explain the N-body sample code. Therefore, we recommend users who are interested in SPH simulation only to read § 4.1.

4.1 N-body simulation code

4.1.1 Location of source files and file structure

The source files of the sample code are in the directory `$(FDPS)/sample/c++/nbody`. The sample code consists of `user-defined.hpp` where user-defined types and interaction function are described, and `nbody.cpp` where the other parts of N-body simulation code are implemented. In addition to these, there is a Makefile for GCC, `Makefile`.

4.1.2 User-defined types and user-defined functions

In this section, we describe the details of classes and `void` functions that users must define when performing an N-body simulation with 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`).

Listing 1: FullParticle type

```
class FPGrav{
public:
    PS::S64    id;
    PS::F64    mass;
    PS::F64vec pos;
    PS::F64vec vel;
    PS::F64vec acc;
    PS::F64    pot;

    static PS::F64 eps;

    PS::F64vec getPos() const {
        return pos;
    }

    PS::F64 getCharge() const {
        return mass;
    }

    void copyFromFP(const FPGrav & fp){
        mass = fp.mass;
        pos  = fp.pos;
    }

    void copyFromForce(const FPGrav & force) {
        acc = force.acc;
        pot = force.pot;
    }

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

    void writeAscii(FILE* fp) const {
        fprintf(fp, "%lld\t%g\t%g\t%g\t%g\t%g\t%g\t%g\n",
                this->id, this->mass,
                this->pos.x, this->pos.y, this->pos.z,
                this->vel.x, this->vel.y, this->vel.z);
    }

    void readAscii(FILE* fp) {
        fscanf(fp, "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\n",
               &this->id, &this->mass,
               &this->pos.x, &this->pos.y, &this->pos.z,
               &this->vel.x, &this->vel.y, &this->vel.z);
        }

};
```

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.

4.1.2.2 calcForceEpEp

You must define an interaction function calcForceEpEp as `void` function in C++. It should contain actual code for the calculation of interaction between particles. Listing 2 shows the implementation of calcForceEpEp (see `user-defined.hpp`).

Listing 2: Function calcForceEpEp

```
template <class TParticleJ>
void CalcGravity(const FPGrav * ep_i,
                 const PS::S32 n_ip,
                 const TParticleJ * ep_j,
                 const PS::S32 n_jp,
                 FPGrav * force) {
    PS::F64 eps2 = FPGrav::eps * FPGrav::eps;
    for(PS::S32 i = 0; i < n_ip; i++){
        PS::F64vec xi = ep_i[i].getPos();
        PS::F64vec ai = 0.0;
        PS::F64 poti = 0.0;
        for(PS::S32 j = 0; j < n_jp; j++){
            PS::F64vec rij    = xi - ep_j[j].getPos();
            PS::F64    r3_inv = rij * rij + eps2;
            PS::F64    r_inv  = 1.0/sqrt(r3_inv);
            r3_inv  = r_inv * r_inv;
            r_inv  *= ep_j[j].getCharge();
            r3_inv *= r_inv;
            ai     -= r3_inv * rij;
            poti   -= r_inv;
        }
        force[i].acc += ai;
        force[i].pot += poti;
    }
}
```

Here, we show the implementation for the case that the code is executed on CPUs without the Phantom-GRAPE library.

In this sample, we implemented calcForceEpEp using function template[1]. Its dummy arguments are an array of EssentialParticleI type, the number of EssentialParticleI type variables, an array of EssentialParticleJ type, the number of EssentialParticleJ type variables, an array of Force type.

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 Including the header file of FDPS

To make the standard features of FDPS available, we must include header file `particle_simulator.hpp`.

Listing 3: Including header file `particle_simulator.hpp`

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

4.1.3.2 Initialization and Termination of FDPS

First, users must initialize FDPS by the following code.

Listing 4: Initialization of FDPS

[1] Function template is, as the name suggests, template of function, in which we can use variables of general data types listed in the angle brackets of `templete <...>` to define function. This characteristic is very different from normal functions in which we must use variables of specific data types such as `int`, `double`, `struct foo`, etc. All of general data types used in a function template is replaced by specific data types at the compile-time because function template must be called with the template argument in which we must describe a list of specific data types.

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

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

Listing 5: Termination of FDPS

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

4.1.3.3 Creation and initialization of FDPS objects

Once succeed the initialization, the user needs to create objects used to talk to FDPS. In this section, we describe how to create and initialize these objects.

4.1.3.3.1 Creation of FDPS objects

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

Listing 6: Creation of FDPS objects

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

4.1.3.3.2 Initialization of DomainInfo object

Once create the objects, user must initialize these objects. In this sample code, since the boundary condition is not periodic, users have only to call the API initialize to initialize the objects.

Listing 7: Initialization of DomainInfo object

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

Note that the first argument of API `initialize` is 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 the specification of FDPS (see § 9.1.2 in `doc_spec_cpp_en.pdf`).

4.1.3.3.3 Initialization of ParticleSystem object

Next, you must initialize a ParticleSystem object. This is done by calling the API `initialize`.

Listing 8: Initialization of ParticleSystem object

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

4.1.3.3.4 Initialization of Tree object

Next, we must initialize a Tree object. The initialization of a Tree object is done by calling the API `initialize` . This API should be given a rough number of particles. In this sample, we set the total number of particles `ntot`:

Listing 9: Initialization of Tree object

```
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.

- `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.

4.1.3.4 Time integration loop

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

4.1.3.4.1 Domain Decomposition

First, the computational domain is decomposed, using the current distribution of particles. In the sample, this is done by API decomposeDomainAll of the DomainInfo object:

Listing 10: Domain Decomposition

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

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

4.1.3.4.2 Particle Exchange

Then, particles are exchanged between processes so that they belong to the process for the domain of their coordinates. To do so, users can use API `exchangeParticle` of ParticleSystem object.

Listing 11: Particle Exchange

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

4.1.3.4.3 Interaction Calculation

After the domain decomposition and the particle exchange, an interaction calculation is done. To do so, users can use API `calcForceAllAndWriteBack` of Tree object.

Listing 12: Interaction Calculation

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

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

4.1.3.4.4 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 `void` 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 13: Calculation of $D(\Delta t)K(\frac{\Delta t}{2})$ operator

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

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

Listing 14: Calculation of $K(\frac{\Delta t}{2})$ operator

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

4.1.4 Log file

Once the calculation starts successfully, the time and the energy error are printed in the standard output. The first step is shown in the bellow example.

Listing 15: standard output

```
time:    0.0000000000E+000, energy error:   -0.0000000000E+000
```

4.2 SPH simulation code with fixed smoothing length

In this section, we describe the sample code used in the previous section (§ 3), a standard SPH code with fixed smoothing length, in detail.

4.2.1 Location of source files and file structure

The source files of the sample code are in the directory `$(FDPS)/sample/c++/sph`. The sample code consists of `main.cpp` and a Makefile for GCC, `Makefile`.

4.2.2 User-defined types and user-defined functions

In this section, we describe the derived data types and subroutines that users must define when performing SPH simulations by using of FDPS.

4.2.2.1 FullParticle type

Users must define a FullParticle type as a user-defined type. The FullParticle type must contain all physical quantities of an SPH particle necessary for the simulation. Listing 16 shows an example implementation of the `FullParticle` type in our sample code (see `main.cpp`).

Listing 16: FullParticle type

```
struct FP{
   PS::F64 mass;
   PS::F64vec pos;
   PS::F64vec vel;
   PS::F64vec acc;
   PS::F64 dens;
   PS::F64 eng;
   PS::F64 pres;
   PS::F64 smth;
   PS::F64 snds;
   PS::F64 eng_dot;
   PS::F64 dt;
   PS::S64 id;
   PS::F64vec vel_half;
   PS::F64 eng_half;
   void copyFromForce(const Dens& dens){
      this->dens = dens.dens;
   }
   void copyFromForce(const Hydro& force){
      this->acc     = force.acc;
      this->eng_dot = force.eng_dot;
      this->dt      = force.dt;
   }
   PS::F64 getCharge() const{
      return this->mass;
   }
   PS::F64vec getPos() const{
      return this->pos;
   }
   PS::F64 getRSearch() const{
      return kernelSupportRadius * this->smth;
   }
   void setPos(const PS::F64vec& pos){
      this->pos = pos;
   }
   void writeAscii(FILE* fp) const{
      fprintf(fp,
              "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t"
              "%lf\t%lf\t%lf\t%lf\t%lf\n",
              this->id, this->mass,
              this->pos.x, this->pos.y, this->pos.z,
                this->vel.x, this->vel.y, this->vel.z,
                this->dens, this->eng, this->pres);
    }
    void readAscii(FILE* fp){
        fscanf(fp,
               "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t"
               "%lf\t%lf\t%lf\t%lf\t%lf\n",
               &this->id, &this->mass,
               &this->pos.x, &this->pos.y, &this->pos.z,
               &this->vel.x, &this->vel.y, &this->vel.z,
               &this->dens, &this->eng, &this->pres);
    }
    void setPressure(){
        const PS::F64 hcr = 1.4;
        pres = (hcr - 1.0) * dens * eng;
        snds = sqrt(hcr * pres / dens);
    }
};
```

FullParticle type 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.

4.2.2.2 EssentialParticleI(J) type

Users must define an EssentialParticleI type. An EssentialParticleI type must contain all necessary physical quantities to compute the Force as an i-particle in its member variables. Moreover in this sample code, it also doubles as an EssentialParticleJ type and all necessary physical quantities as a j-particle as well need to be included in the member variables. Hereinafter, we simply call this EssentialParticle type. Listing 17 shows an example of EssentialParticle type of this sample code (see `main.cpp`):

Listing 17: EssentialParticle type

```
struct EP{
    PS::F64vec pos;
    PS::F64vec vel;
    PS::F64    mass;
    PS::F64    smth;
    PS::F64    dens;
    PS::F64    pres;
    PS::F64    snds;
    void copyFromFP(const FP& rp){
        this->pos  = rp.pos;
        this->vel  = rp.vel;
        this->mass = rp.mass;
        this->smth = rp.smth;
        this->dens = rp.dens;
```

```
      this->pres = rp.pres;
      this->snds = rp.snds;
   }
   PS::F64vec getPos() const{
      return this->pos;
   }
   PS::F64 getRSearch() const{
      return kernelSupportRadius * this->smth;
   }
   void setPos(const PS::F64vec& pos){
      this->pos = pos;
   }
};
```

EssentialParticle type should have member function `copyFromFP` to copy necessary quantities from FullParticle type described above. Also, it should have member functions `getPos()`, `getRSearch()`, and `setPos()`.

4.2.2.3 Force type

Users must define a Force type. A Force type must contain all the resultant physical quantities after performing the Force computations. In this sample code, we have 2 force computations; one for the density and the other for the fluid interactions. Thus, we have to define 2 different Force types. In Listing 18, we show an example of the Force types in this sample code.

Listing 18: Force type

```
class Dens{
   public:
   PS::F64 dens;
   PS::F64 smth;
   void clear(){
      dens = 0;
   }
};
class Hydro{
   public:
   PS::F64vec acc;
   PS::F64 eng_dot;
   PS::F64 dt;
   void clear(){
      acc = 0;
      eng_dot = 0;
   }
};
```

A Force type should have member function `clear()`, which zero-clears or initializes member variables that store the result of some accumulation operation.

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.

4.2.2.4 calcForceEpEp

Users must define a `void` function calcForceEpEp in C++ which specifies the interaction between particles. It should contain actual code for the calculation of interaction between particles. Listing 19 shows the implementation of calcForceEpEp (see `main.cppuser_defined.cuser_defined.F90` .

Listing 19: Function calcForceEpEp type

```
class CalcDensity{
   public:
   void operator () (const EP* const ep_i, const PS::S32 Nip,
                     const EP* const ep_j, const PS::S32 Njp,
                     Dens* const dens){
      for(PS::S32 i = 0 ; i < Nip ; ++i){
         dens[i].clear();
         for(PS::S32 j = 0 ; j < Njp ; ++j){
            const PS::F64vec dr = ep_j[j].pos - ep_i[i].pos;
            dens[i].dens += ep_j[j].mass * W(dr, ep_i[i].smth);
         }
      }
   }
};

class CalcHydroForce{
   public:
   void operator () (const EP* const ep_i, const PS::S32 Nip,
                     const EP* const ep_j, const PS::S32 Njp,
                     Hydro* const hydro){
      for(PS::S32 i = 0; i < Nip ; ++ i){
         hydro[i].clear();
         PS::F64 v_sig_max = 0.0;
         for(PS::S32 j = 0; j < Njp ; ++j){
            const PS::F64vec dr = ep_i[i].pos - ep_j[j].pos;
            const PS::F64vec dv = ep_i[i].vel - ep_j[j].vel;
            const PS::F64 w_ij = (dv * dr < 0) ? dv * dr / sqrt(dr * dr) :
                0;
            const PS::F64 v_sig = ep_i[i].snds + ep_j[j].snds - 3.0 * w_ij
               ;
            v_sig_max = std::max(v_sig_max, v_sig);
            const PS::F64 AV = - 0.5 * v_sig * w_ij / (0.5 * (ep_i[i].dens
                + ep_j[j].dens));
            const PS::F64vec gradW_ij = 0.5 * (gradW(dr, ep_i[i].smth) +
                gradW(dr, ep_j[j].smth));
            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;
            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;
          }
          hydro[i].dt = C_CFL * 2.0 * ep_i[i].smth / v_sig_max;
        }
    }
};
```

Users can define calcForceEpEp using a functor (function object). In this sample, we use a functor instead of `void` function. In any case, the arguments of calcForceEpEp 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 should be defined.

4.2.3 The main body of the user program

In this section, we describe `void` functions and functions to be called from the main function of the user program when a user want to do an SPH simulation using FDPS .

4.2.3.1 Including the header file of FDPS

To make the standard features of FDPS available, we must include header file `particle_simulator.hpp`.

Listing 20: Including header file `particle_simulator.hpp`

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

4.2.3.2 Initialization and termination of FDPS

You should first initialize FDPS by the following code.

Listing 21: Initialization of FDPS

```
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 22: Termination of FDPS

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

4.2.3.3 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.

4.2.3.3.1 Creation of necessary FDPS objects

In an SPH simulation code, one needs to create objects for particles, for domain information, for interaction calculation of Gather type (for density calculation using gather type interaction), and for interaction calculation of Symmetry type (for hydrodynamic interaction calculation using symmetric type interaction). The following is the code to create to them.

Listing 23: Creation of FDPS Objects

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

4.2.3.3.2 Initialization of the domain information 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 API `initialize` of DomainInfo object. After the initialization of the object, the type of the boundary and the size of the simulation box should be set by calling APIs `setBoundaryCondition` and `setPosRootDomain` of DomainInfo object. In this code, we use the periodic boundary for all of x, y and z directions.

Listing 24: Initialization of DomainInfo object

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

4.2.3.3.3 Initialization of ParticleSystem object

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

Listing 25: Initialization of ParticleSystem object

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

4.2.3.3.4 Initialization of Tree objects

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

Listing 26: Initialization of tree objects

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

4.2.3.4 Time integration loop

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

4.2.3.4.1 Domain Decomposition

First, the computational domain is decomposed, using the current distribution of particles. To do so, the API `decomposeDomainAll` of DomainInfo object is called.

Listing 27: Domain Decomposition

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

4.2.3.4.2 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 API `exchangeParticle` of `ParticleSystem` object is used.

Listing 28: Particle Exchange

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

4.2.3.4.3 Interaction Calculation

After the domain decomposition and particle exchange, interaction calculation is done. To do so, the following API `calcForceAllAndWriteBack` of Tree object is used.

Listing 29: Interaction Calculation

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

4.2.4 Compilation of the program

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

```
$ make
```

4.2.5 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.

4.2.6 Log and output files

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

4.2.7 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 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`)

```
#pragma once
class FileHeader{
public:
    PS::S64 n_body;
    PS::F64 time;
    PS::S32 readAscii(FILE * fp) {
        fscanf(fp, "%lf\n", &time);
        fscanf(fp, "%lld\n", &n_body);
        return n_body;
    }
    void writeAscii(FILE* fp) const {
        fprintf(fp, "%e\n", time);
        fprintf(fp, "%lld\n", n_body);
    }
};

class FPGrav{
public:
    PS::S64    id;
    PS::F64    mass;
    PS::F64vec pos;
    PS::F64vec vel;
    PS::F64vec acc;
    PS::F64    pot;

    static PS::F64 eps;

    PS::F64vec getPos() const {
        return pos;
    }

    PS::F64 getCharge() const {
        return mass;
    }

    void copyFromFP(const FPGrav & fp){
        mass = fp.mass;
        pos  = fp.pos;
    }

    void copyFromForce(const FPGrav & force) {
        acc = force.acc;
        pot = force.pot;
    }

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

    void writeAscii(FILE* fp) const {
        fprintf(fp, "%lld\t%g\t%g\t%g\t%g\t%g\t%g\t%g\n",
                this->id, this->mass,
                this->pos.x, this->pos.y, this->pos.z,
                this->vel.x, this->vel.y, this->vel.z);
    }

    void readAscii(FILE* fp) {
        fscanf(fp, "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\n",
               &this->id, &this->mass,
               &this->pos.x, &this->pos.y, &this->pos.z,
               &this->vel.x, &this->vel.y, &this->vel.z);
        }

};


#ifdef ENABLE_PHANTOM_GRAPE_X86


template <class TParticleJ>
void CalcGravity(const FPGrav * iptcl,
                 const PS::S32 ni,
                 const TParticleJ * jptcl,
                 const PS::S32 nj,
                 FPGrav * force) {
    const PS::S32 nipipe = ni;
    const PS::S32 njpipe = nj;
    PS::F64 (*xi)[3] = (PS::F64 (*)[3])malloc(sizeof(PS::F64) * nipipe *
         PS::DIMENSION);
    PS::F64 (*ai)[3] = (PS::F64 (*)[3])malloc(sizeof(PS::F64) * nipipe *
         PS::DIMENSION);
    PS::F64  *pi     = (PS::F64  *    )malloc(sizeof(PS::F64) * nipipe);
    PS::F64 (*xj)[3] = (PS::F64 (*)[3])malloc(sizeof(PS::F64) * njpipe *
         PS::DIMENSION);
    PS::F64  *mj     = (PS::F64  *    )malloc(sizeof(PS::F64) * njpipe);
    for(PS::S32 i = 0; i < ni; i++) {
        xi[i][0] = iptcl[i].getPos()[0];
        xi[i][1] = iptcl[i].getPos()[1];
        xi[i][2] = iptcl[i].getPos()[2];
        ai[i][0] = 0.0;
        ai[i][1] = 0.0;
        ai[i][2] = 0.0;
        pi[i]    = 0.0;
    }
    for(PS::S32 j = 0; j < nj; j++) {
        xj[j][0] = jptcl[j].getPos()[0];
        xj[j][1] = jptcl[j].getPos()[1];
        xj[j][2] = jptcl[j].getPos()[2];
        mj[j]    = jptcl[j].getCharge();
        xj[j][0] = jptcl[j].pos[0];
        xj[j][1] = jptcl[j].pos[1];
        xj[j][2] = jptcl[j].pos[2];
        mj[j]    = jptcl[j].mass;
    }
    PS::S32 devid = PS::Comm::getThreadNum();
    g5_set_xmjMC(devid, 0, nj, xj, mj);
    g5_set_nMC(devid, nj);
    g5_calculate_force_on_xMC(devid, xi, ai, pi, ni);
    for(PS::S32 i = 0; i < ni; i++) {
        force[i].acc[0] += ai[i][0];
        force[i].acc[1] += ai[i][1];
        force[i].acc[2] += ai[i][2];
        force[i].pot    -= pi[i];
    }
    free(xi);
    free(ai);
    free(pi);
    free(xj);
    free(mj);
}

#else

template <class TParticleJ>
void CalcGravity(const FPGrav * ep_i,
                 const PS::S32 n_ip,
                 const TParticleJ * ep_j,
                 const PS::S32 n_jp,
                 FPGrav * force) {
    PS::F64 eps2 = FPGrav::eps * FPGrav::eps;
    for(PS::S32 i = 0; i < n_ip; i++){
        PS::F64vec xi = ep_i[i].getPos();
        PS::F64vec ai = 0.0;
        PS::F64 poti = 0.0;
        for(PS::S32 j = 0; j < n_jp; j++){
            PS::F64vec rij    = xi - ep_j[j].getPos();
            PS::F64    r3_inv = rij * rij + eps2;
            PS::F64    r_inv  = 1.0/sqrt(r3_inv);
            r3_inv  = r_inv * r_inv;
            r_inv  *= ep_j[j].getCharge();
            r3_inv *= r_inv;
            ai     -= r3_inv * rij;
            poti   -= r_inv;
        }
        force[i].acc += ai;
        force[i].pot += poti;
    }
}

#endif
```

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

```
#include<iostream>
#include<fstream>
```

```
#include<unistd.h>
#include<sys/stat.h>
#include<particle_simulator.hpp>
#ifdef ENABLE_PHANTOM_GRAPE_X86
#include <gp5util.h>
#endif
#ifdef ENABLE_GPU_CUDA
#define MULTI_WALK
#include"force_gpu_cuda.hpp"
#endif
#include "user-defined.hpp"

void makeColdUniformSphere(const PS::F64 mass_glb,
                           const PS::S64 n_glb,
                           const PS::S64 n_loc,
                           PS::F64 *& mass,
                           PS::F64vec *& pos,
                           PS::F64vec *& vel,
                           const PS::F64 eng = -0.25,
                           const PS::S32 seed = 0) {

    assert(eng < 0.0);
    {
        PS::MTTS mt;
        mt.init_genrand(0);
        for(PS::S32 i = 0; i < n_loc; i++){
            mass[i] = mass_glb / n_glb;
            const PS::F64 radius = 3.0;
            do {
                pos[i][0] = (2. * mt.genrand_res53() - 1.) * radius;
                pos[i][1] = (2. * mt.genrand_res53() - 1.) * radius;
                pos[i][2] = (2. * mt.genrand_res53() - 1.) * radius;
            }while(pos[i] * pos[i] >= radius * radius);
            vel[i][0] = 0.0;
            vel[i][1] = 0.0;
            vel[i][2] = 0.0;
        }
    }

    PS::F64vec cm_pos  = 0.0;
    PS::F64vec cm_vel  = 0.0;
    PS::F64    cm_mass = 0.0;
    for(PS::S32 i = 0; i < n_loc; i++){
        cm_pos  += mass[i] * pos[i];
        cm_vel  += mass[i] * vel[i];
        cm_mass += mass[i];
    }
    cm_pos /= cm_mass;
    cm_vel /= cm_mass;
    for(PS::S32 i = 0; i < n_loc; i++){
        pos[i] -= cm_pos;
        vel[i] -= cm_vel;
    }
}

```

```
template<class Tpsys>
void setParticlesColdUniformSphere(Tpsys & psys,
                                   const PS::S32 n_glb,
                                   PS::S32 & n_loc) {

    n_loc = n_glb;
    psys.setNumberOfParticleLocal(n_loc);

    PS::F64    * mass = new PS::F64[n_loc];
    PS::F64vec * pos  = new PS::F64vec[n_loc];
    PS::F64vec * vel  = new PS::F64vec[n_loc];
    const PS::F64 m_tot = 1.0;
    const PS::F64 eng   = -0.25;
    makeColdUniformSphere(m_tot, n_glb, n_loc, mass, pos, vel, eng);
    for(PS::S32 i = 0; i < n_loc; i++){
        psys[i].mass = mass[i];
        psys[i].pos  = pos[i];
        psys[i].vel  = vel[i];
        psys[i].id   = i;
    }
    delete [] mass;
    delete [] pos;
    delete [] vel;
}

template<class Tpsys>
void kick(Tpsys & system,
          const PS::F64 dt) {
    PS::S32 n = system.getNumberOfParticleLocal();
    for(PS::S32 i = 0; i < n; i++) {
        system[i].vel  += system[i].acc * dt;
    }
}

template<class Tpsys>
void drift(Tpsys & system,
           const PS::F64 dt) {
    PS::S32 n = system.getNumberOfParticleLocal();
    for(PS::S32 i = 0; i < n; i++) {
        system[i].pos  += system[i].vel * dt;
    }
}

template<class Tpsys>
void calcEnergy(const Tpsys & system,
                PS::F64 & etot,
                PS::F64 & ekin,
                PS::F64 & epot,
                const bool clear=true){
    if(clear){
        etot = ekin = epot = 0.0;
    }
    PS::F64 etot_loc = 0.0;
    PS::F64 ekin_loc = 0.0;
    PS::F64 epot_loc = 0.0;
    const PS::S32 nbody = system.getNumberOfParticleLocal();
    for(PS::S32 i = 0; i < nbody; i++){
        ekin_loc += system[i].mass * system[i].vel * system[i].vel;
        epot_loc += system[i].mass * (system[i].pot + system[i].mass /
               FPGrav::eps);
    }
    ekin_loc *= 0.5;
    epot_loc *= 0.5;
    etot_loc  = ekin_loc + epot_loc;
    etot = PS::Comm::getSum(etot_loc);
    epot = PS::Comm::getSum(epot_loc);
    ekin = PS::Comm::getSum(ekin_loc);
}

void printHelp() {
    std::cerr<<"o:␣dir␣name␣of␣output␣(default:␣./result)"<<std::endl;
    std::cerr<<"t:␣theta␣(default:␣0.5)"<<std::endl;
    std::cerr<<"T:␣time_end␣(default:␣10.0)"<<std::endl;
    std::cerr<<"s:␣time_step␣(default:␣1.0␣/␣128.0)"<<std::endl;
    std::cerr<<"d:␣dt_diag␣(default:␣1.0␣/␣8.0)"<<std::endl;
    std::cerr<<"D:␣dt_snap␣(default:␣1.0)"<<std::endl;
    std::cerr<<"l:␣n_leaf_limit␣(default:␣8)"<<std::endl;
    std::cerr<<"n:␣n_group_limit␣(default:␣64)"<<std::endl;
    std::cerr<<"N:␣n_tot␣(default:␣1024)"<<std::endl;
    std::cerr<<"h:␣help"<<std::endl;
}

void makeOutputDirectory(char * dir_name) {
    struct stat st;
    PS::S32 ret;
    if (PS::Comm::getRank() == 0) {
        if (stat(dir_name, &st) != 0) {
            ret = mkdir(dir_name, 0777);
        } else {
            ret = 0; // the directory named dir_name already exists.
        }
    }
    PS::Comm::broadcast(&ret, 1);
    if (ret == 0) {
        if (PS::Comm::getRank() == 0)
            fprintf(stderr, "Directory␣\"%s\"␣is␣successfully␣made.\n",
                    dir_name);
    } else {
        if (PS::Comm::getRank() == 0)
            fprintf(stderr, "Directory␣%s␣fails␣to␣be␣made.\n", dir_name);
        PS::Abort();
    }
}

PS::F64 FPGrav::eps = 1.0/32.0;

int main(int argc, char *argv[]) {
    std::cout<<std::setprecision(15);
    std::cerr<<std::setprecision(15);

    PS::Initialize(argc, argv);
    PS::F32 theta = 0.5;
    PS::S32 n_leaf_limit = 8;
    PS::S32 n_group_limit = 64;
    PS::F32 time_end = 10.0;
    PS::F32 dt = 1.0 / 128.0;
    PS::F32 dt_diag = 1.0 / 8.0;
    PS::F32 dt_snap = 1.0;
    char dir_name[1024];
    PS::S64 n_tot = 1024;
    PS::S32 c;
    sprintf(dir_name,"./result");
    opterr = 0;
    while((c=getopt(argc,argv,"i:o:d:D:t:T:l:n:N:hs:")) != -1){
        switch(c){
        case 'o':
            sprintf(dir_name,optarg);
            break;
        case 't':
            theta = atof(optarg);
            std::cerr << "theta␣=" << theta << std::endl;
            break;
        case 'T':
            time_end = atof(optarg);
            std::cerr << "time_end␣=␣" << time_end << std::endl;
            break;
        case 's':
            dt = atof(optarg);
            std::cerr << "time_step␣=␣" << dt << std::endl;
            break;
        case 'd':
            dt_diag = atof(optarg);
            std::cerr << "dt_diag␣=␣" << dt_diag << std::endl;
            break;
        case 'D':
            dt_snap = atof(optarg);
            std::cerr << "dt_snap␣=␣" << dt_snap << std::endl;
            break;
        case 'l':
            n_leaf_limit = atoi(optarg);
            std::cerr << "n_leaf_limit␣=␣" << n_leaf_limit << std::endl;
            break;
        case 'n':
            n_group_limit = atoi(optarg);
            std::cerr << "n_group_limit␣=␣" << n_group_limit << std::endl;
            break;
        case 'N':
            n_tot = atoi(optarg);
            std::cerr << "n_tot␣=␣" << n_tot << std::endl;
            break;
        case 'h':
            if(PS::Comm::getRank() == 0) {
                printHelp();
            }
            PS::Finalize();
```

```
                return 0;
            default:
                if(PS::Comm::getRank() == 0) {
                    std::cerr<<"No␣such␣option!␣Available␣options␣are␣here."<<
                        std::endl;
                    printHelp();
                }
                PS::Abort();
            }
        }

        makeOutputDirectory(dir_name);

        std::ofstream fout_eng;

        if(PS::Comm::getRank() == 0) {
            char sout_de[1024];
            sprintf(sout_de, "%s/t-de.dat", dir_name);
            fout_eng.open(sout_de);
            fprintf(stdout, "This␣is␣a␣sample␣program␣of␣N-body␣simulation␣on␣
                FDPS!\n");
            fprintf(stdout, "Number␣of␣processes:␣%d\n", PS::Comm::
                getNumberOfProc());
            fprintf(stdout, "Number␣of␣threads␣per␣process:␣%d\n", PS::Comm::
                getNumberOfThread());
        }

        PS::ParticleSystem<FPGrav> system_grav;
        system_grav.initialize();
        PS::S32 n_loc    = 0;
        PS::F32 time_sys = 0.0;
        if(PS::Comm::getRank() == 0) {
            setParticlesColdUniformSphere(system_grav, n_tot, n_loc);
        } else {
            system_grav.setNumberOfParticleLocal(n_loc);
        }

        const PS::F32 coef_ema = 0.3;
        PS::DomainInfo dinfo;
        dinfo.initialize(coef_ema);
        dinfo.decomposeDomainAll(system_grav);
        system_grav.exchangeParticle(dinfo);
        n_loc = system_grav.getNumberOfParticleLocal();

#ifdef ENABLE_PHANTOM_GRAPE_X86
        g5_open();
        g5_set_eps_to_all(FPGrav::eps);
#endif

        PS::TreeForForceLong<FPGrav, FPGrav, FPGrav>::Monopole tree_grav;
        tree_grav.initialize(n_tot, theta, n_leaf_limit, n_group_limit);
#ifdef MULTI_WALK
        const PS::S32 n_walk_limit = 200;
        const PS::S32 tag_max = 1;
        tree_grav.calcForceAllAndWriteBackMultiWalk(DispatchKernelWithSP,
                                                RetrieveKernel,
                                                tag_max,
                                                system_grav,
                                                dinfo,
                                                n_walk_limit);
#else
    tree_grav.calcForceAllAndWriteBack(CalcGravity<FPGrav>,
                                       CalcGravity<PS::SPJMonopole>,
                                       system_grav,
                                       dinfo);
#endif
    PS::F64 Epot0, Ekin0, Etot0, Epot1, Ekin1, Etot1;
    calcEnergy(system_grav, Etot0, Ekin0, Epot0);
    PS::F64 time_diag = 0.0;
    PS::F64 time_snap = 0.0;
    PS::S64 n_loop = 0;
    PS::S32 id_snap = 0;
    while(time_sys < time_end){
        if( (time_sys >= time_snap) || ( (time_sys + dt) - time_snap ) > ( time_snap - time_sys) ){
            char filename[256];
            sprintf(filename, "%s/%04d.dat", dir_name, id_snap++);
            FileHeader header;
            header.time   = time_sys;
            header.n_body = system_grav.getNumberOfParticleGlobal();
            system_grav.writeParticleAscii(filename, header);
            time_snap += dt_snap;
        }

        calcEnergy(system_grav, Etot1, Ekin1, Epot1);

        if(PS::Comm::getRank() == 0){
            if( (time_sys >= time_diag) || ( (time_sys + dt) - time_diag ) > (time_diag - time_sys) ){
                fout_eng << time_sys << "␣␣␣" << (Etot1 - Etot0) / Etot0 << std::endl;
                fprintf(stdout, "time:␣%10.7f␣energy␣error:␣%+e\n",
                        time_sys, (Etot1 - Etot0) / Etot0);
                time_diag += dt_diag;
            }
        }


        kick(system_grav, dt * 0.5);

        time_sys += dt;
        drift(system_grav, dt);

        if(n_loop % 4 == 0){
            dinfo.decomposeDomainAll(system_grav);
        }

        system_grav.exchangeParticle(dinfo);
#ifdef MULTI_WALK
        tree_grav.calcForceAllAndWriteBackMultiWalk(DispatchKernelWithSP,
                                                                  RetrieveKernel,
                                                                  tag_max,
                                                                  system_grav,
                                                                  dinfo,
                                                                  n_walk_limit,
                                                                  true);
#else
          tree_grav.calcForceAllAndWriteBack(CalcGravity<FPGrav>,
                                                       CalcGravity<PS::SPJMonopole>,
                                                       system_grav,
                                                       dinfo);
#endif

          kick(system_grav, dt * 0.5);

          n_loop++;
     }

#ifdef ENABLE_PHANTOM_GRAPE_X86
     g5_close();
#endif

     PS::Finalize();
     return 0;
}
```

5.2 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 32: Sample code of SPH simulation

```
// Include FDPS header
#include <particle_simulator.hpp>
// Include the standard C++ headers
#include <cmath>
#include <cstdio>
#include <iostream>
#include <vector>
#include <sys/stat.h>

/* Parameters */
const short int Dim = 3;
const PS::F64 SMTH = 1.2;
const PS::U32 OUTPUT_INTERVAL = 10;
const PS::F64 C_CFL = 0.3;

/* Kernel Function */
const PS::F64 pi = atan(1.0) * 4.0;
const PS::F64 kernelSupportRadius = 2.5;

PS::F64 W(const PS::F64vec dr, const PS::F64 h){
   const PS::F64 H = kernelSupportRadius * h;
   const PS::F64 s = sqrt(dr * dr) / H;
   const PS::F64 s1 = (1.0 - s < 0) ? 0 : 1.0 - s;
   const PS::F64 s2 = (0.5 - s < 0) ? 0 : 0.5 - s;
   PS::F64 r_value = pow(s1, 3) - 4.0 * pow(s2, 3);
   //if # of dimension == 3
   r_value *= 16.0 / pi / (H * H * H);
   return r_value;
}

PS::F64vec gradW(const PS::F64vec dr, const PS::F64 h){
   const PS::F64 H = kernelSupportRadius * h;
   const PS::F64 s = sqrt(dr * dr) / H;
   const PS::F64 s1 = (1.0 - s < 0) ? 0 : 1.0 - s;
   const PS::F64 s2 = (0.5 - s < 0) ? 0 : 0.5 - s;
   PS::F64 r_value = - 3.0 * pow(s1, 2) + 12.0 * pow(s2, 2);
   //if # of dimension == 3
   r_value *= 16.0 / pi / (H * H * H);
   return dr * r_value / (sqrt(dr * dr) * H + 1.0e-6 * h);
}

/* Class Definitions */
//** Force Class (Result Class)
class Dens{
   public:
   PS::F64 dens;
   PS::F64 smth;
   void clear(){
      dens = 0;
   }
};
class Hydro{
   public:
   PS::F64vec acc;
   PS::F64 eng_dot;
   PS::F64 dt;
   void clear(){
      acc = 0;
      eng_dot = 0;
   }
};

//** Full Particle Class
struct FP{
   PS::F64 mass;
   PS::F64vec pos;
   PS::F64vec vel;
   PS::F64vec acc;
   PS::F64 dens;
   PS::F64 eng;
   PS::F64 pres;
   PS::F64 smth;
   PS::F64 snds;
   PS::F64 eng_dot;
   PS::F64 dt;
   PS::S64 id;
   PS::F64vec vel_half;
   PS::F64 eng_half;
   void copyFromForce(const Dens& dens){
      this->dens = dens.dens;
   }
   void copyFromForce(const Hydro& force){
      this->acc     = force.acc;
      this->eng_dot = force.eng_dot;
      this->dt      = force.dt;
   }
   PS::F64 getCharge() const{
      return this->mass;
   }
   PS::F64vec getPos() const{
      return this->pos;
   }
   PS::F64 getRSearch() const{
      return kernelSupportRadius * this->smth;
   }
   void setPos(const PS::F64vec& pos){
      this->pos = pos;
   }
   void writeAscii(FILE* fp) const{
      fprintf(fp,
              "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t"
              "%lf\t%lf\t%lf\t%lf\t%lf\n",
              this->id, this->mass,
              this->pos.x, this->pos.y, this->pos.z,
              this->vel.x, this->vel.y, this->vel.z,
              this->dens, this->eng, this->pres);
   }
   void readAscii(FILE* fp){
      fscanf(fp,
             "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t"
             "%lf\t%lf\t%lf\t%lf\t%lf\n",
             &this->id, &this->mass,
             &this->pos.x, &this->pos.y, &this->pos.z,
             &this->vel.x, &this->vel.y, &this->vel.z,
             &this->dens, &this->eng, &this->pres);
   }
   void setPressure(){
      const PS::F64 hcr = 1.4;
      pres = (hcr - 1.0) * dens * eng;
      snds = sqrt(hcr * pres / dens);
   }
};

//** Essential Particle Class
struct EP{
   PS::F64vec pos;
   PS::F64vec vel;
   PS::F64    mass;
   PS::F64    smth;
   PS::F64    dens;
   PS::F64    pres;
   PS::F64    snds;
   void copyFromFP(const FP& rp){
      this->pos  = rp.pos;
      this->vel  = rp.vel;
      this->mass = rp.mass;
      this->smth = rp.smth;
      this->dens = rp.dens;
      this->pres = rp.pres;
      this->snds = rp.snds;
   }
   PS::F64vec getPos() const{
      return this->pos;
   }
   PS::F64 getRSearch() const{
      return kernelSupportRadius * this->smth;
   }
   void setPos(const PS::F64vec& pos){
      this->pos = pos;
   }
};

class FileHeader{
   public:
   PS::S32 Nbody;
   PS::F64 time;
   int readAscii(FILE* fp){
      fscanf(fp, "%lf\n", &time);
      fscanf(fp, "%d\n", &Nbody);
      return Nbody;
   }
   void writeAscii(FILE* fp) const{
      fprintf(fp, "%e\n", time);
      fprintf(fp, "%d\n", Nbody);
   }
};

struct boundary{
   PS::F64 x, y, z;
};


/* Force Functors */
class CalcDensity{
   public:
   void operator () (const EP* const ep_i, const PS::S32 Nip,
                     const EP* const ep_j, const PS::S32 Njp,
                     Dens* const dens){
      for(PS::S32 i = 0 ; i < Nip ; ++i){
         dens[i].clear();
         for(PS::S32 j = 0 ; j < Njp ; ++j){
            const PS::F64vec dr = ep_j[j].pos - ep_i[i].pos;
            dens[i].dens += ep_j[j].mass * W(dr, ep_i[i].smth);
         }
      }
   }
};

class CalcHydroForce{
   public:
   void operator () (const EP* const ep_i, const PS::S32 Nip,
                     const EP* const ep_j, const PS::S32 Njp,
                     Hydro* const hydro){
      for(PS::S32 i = 0; i < Nip ; ++ i){
         hydro[i].clear();
         PS::F64 v_sig_max = 0.0;
         for(PS::S32 j = 0; j < Njp ; ++j){
            const PS::F64vec dr = ep_i[i].pos - ep_j[j].pos;
            const PS::F64vec dv = ep_i[i].vel - ep_j[j].vel;
            const PS::F64 w_ij = (dv * dr < 0) ? dv * dr / sqrt(dr * dr) : 0;
            const PS::F64 v_sig = ep_i[i].snds + ep_j[j].snds - 3.0 * w_ij;
            v_sig_max = std::max(v_sig_max, v_sig);
            const PS::F64 AV = - 0.5 * v_sig * w_ij / (0.5 * (ep_i[i].dens + ep_j[j].dens));
            const PS::F64vec gradW_ij = 0.5 * (gradW(dr, ep_i[i].smth) + gradW(dr, ep_j[j].smth));
            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;
            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;
         }
         hydro[i].dt = C_CFL * 2.0 * ep_i[i].smth / v_sig_max;
      }
   }
};

void makeOutputDirectory(char * dir_name) {
    struct stat st;
    PS::S32 ret;
    if (PS::Comm::getRank() == 0) {
        if (stat(dir_name, &st) != 0) {
            ret = mkdir(dir_name, 0777);
        } else {
            ret = 0; // the directory named dir_name already exists.
        }
    }
    PS::Comm::broadcast(&ret, 1);
    if (ret == 0) {
        if (PS::Comm::getRank() == 0)
            fprintf(stderr, "Directory␣\"%s\"␣is␣successfully␣made.\n", dir_name);
    } else {
        if (PS::Comm::getRank() == 0)
            fprintf(stderr, "Directory␣%s␣fails␣to␣be␣made.\n", dir_name);
        PS::Abort();
    }
}
```

```

void SetupIC(PS::ParticleSystem<FP>& sph_system, PS::F64 *end_time,
      boundary *box){
   // Place SPH particles
   std::vector<FP> ptcl;
   const PS::F64 dx = 1.0 / 128.0;
   box->x = 1.0;
   box->y = box->z = box->x / 8.0;
   PS::S32 i = 0;
   for(PS::F64 x = 0 ; x < box->x * 0.5 ; x += dx){
      for(PS::F64 y = 0 ; y < box->y ; y += dx){
         for(PS::F64 z = 0 ; z < box->z ; z += dx){
            FP ith;
            ith.pos.x = x;
            ith.pos.y = y;
            ith.pos.z = z;
            ith.dens = 1.0;
            ith.mass = 0.75;
            ith.eng  = 2.5;
            ith.id   = i++;
            ith.smth = 0.012;
            ptcl.push_back(ith);
         }
      }
   }
   for(PS::F64 x = box->x * 0.5 ; x < box->x * 1.0 ; x += dx * 2.0){
      for(PS::F64 y = 0 ; y < box->y ; y += dx){
         for(PS::F64 z = 0 ; z < box->z ; z += dx){
            FP ith;
            ith.pos.x = x;
            ith.pos.y = y;
            ith.pos.z = z;
            ith.dens = 0.5;
            ith.mass = 0.75;
            ith.eng  = 2.5;
            ith.id   = i++;
            ith.smth = 0.012;
            ptcl.push_back(ith);
         }
      }
   }
   for(PS::U32 i = 0 ; i < ptcl.size() ; ++ i){
      ptcl[i].mass = ptcl[i].mass * box->x * box->y * box->z / (PS::F64)(
            ptcl.size());
   }
   std::cout << "#␣of␣ptcls␣is...␣" << ptcl.size() << std::endl;
   // Scatter SPH particles
   assert(ptcl.size() % PS::Comm::getNumberOfProc() == 0);
   const PS::S32 numPtclLocal = ptcl.size() / PS::Comm::getNumberOfProc();
   sph_system.setNumberOfParticleLocal(numPtclLocal);
   const PS::U32 i_head = numPtclLocal * PS::Comm::getRank();
   const PS::U32 i_tail = numPtclLocal * (PS::Comm::getRank() + 1);
   for(PS::U32 i = 0 ; i < ptcl.size() ; ++ i){
      if(i_head <= i && i < i_tail){
         const PS::U32 ii = i - numPtclLocal * PS::Comm::getRank();
```

```
            sph_system[ii] = ptcl[i];
        }
    }
    // Set the end time
    *end_time = 0.12;
    // Fin.
    std::cout << "setup..." << std::endl;
}

void Initialize(PS::ParticleSystem<FP>& sph_system){
    for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
        sph_system[i].setPressure();
    }
}

PS::F64 getTimeStepGlobal(const PS::ParticleSystem<FP>& sph_system){
    PS::F64 dt = 1.0e+30; //set VERY LARGE VALUE
    for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
        dt = std::min(dt, sph_system[i].dt);
    }
    return PS::Comm::getMinValue(dt);
}

void InitialKick(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
    for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
        sph_system[i].vel_half = sph_system[i].vel + 0.5 * dt * sph_system[i
                ].acc;
        sph_system[i].eng_half = sph_system[i].eng + 0.5 * dt * sph_system[i
                ].eng_dot;
    }
}

void FullDrift(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
    // time becomes t + dt;
    for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
        sph_system[i].pos += dt * sph_system[i].vel_half;
    }
}

void Predict(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
    for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
        sph_system[i].vel += dt * sph_system[i].acc;
        sph_system[i].eng += dt * sph_system[i].eng_dot;
    }
}

void FinalKick(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
    for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
        sph_system[i].vel = sph_system[i].vel_half + 0.5 * dt * sph_system[i
                ].acc;
        sph_system[i].eng = sph_system[i].eng_half + 0.5 * dt * sph_system[i
                ].eng_dot;
    }
}

```

```
void setPressure(PS::ParticleSystem<FP>& sph_system){
   for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
      sph_system[i].setPressure();
   }
}

void CheckConservativeVariables(const PS::ParticleSystem<FP>& sph_system){
   PS::F64vec Mom=0.0; // total momentum
   PS::F64    Eng=0.0; // total enegry
   for(PS::S32 i = 0; i < sph_system.getNumberOfParticleLocal(); ++ i){
      Mom += sph_system[i].vel * sph_system[i].mass;
      Eng += (sph_system[i].eng + 0.5 * sph_system[i].vel * sph_system[i].vel)
            * sph_system[i].mass;
   }
   Eng = PS::Comm::getSum(Eng);
   Mom = PS::Comm::getSum(Mom);
    if(PS::Comm::getRank() == 0){
        printf("%.16e\n", Eng);
        printf("%.16e\n", Mom.x);
        printf("%.16e\n", Mom.y);
        printf("%.16e\n", Mom.z);
    }
}

int main(int argc, char* argv[]){
   // Initialize FDPS
   PS::Initialize(argc, argv);
   // Make a directory
   char dir_name[1024];
   sprintf(dir_name,"./result");
   makeOutputDirectory(dir_name);
   // Display # of MPI processes and threads
   PS::S32 nprocs = PS::Comm::getNumberOfProc();
   PS::S32 nthrds = PS::Comm::getNumberOfThread();
   std::cout << "===========================================" << std::endl
             << "␣This␣is␣a␣sample␣program␣of␣"               << std::endl
             << "␣Smoothed␣Particle␣Hydrodynamics␣on␣FDPS!"   << std::endl
             << "␣#␣of␣processes␣is␣" << nprocs               << std::endl
             << "␣#␣of␣thread␣is␣␣␣␣" << nthrds               << std::endl
             << "===========================================" << std::endl;
   // Make an instance of ParticleSystem and initialize it
   PS::ParticleSystem<FP> sph_system;
   sph_system.initialize();
   // Define local variables
   PS::F64 dt, end_time;
   boundary box;
   // Make an initial condition and initialize the particle system
   SetupIC(sph_system, &end_time, &box);
   Initialize(sph_system);
   // Make an instance of DomainInfo and initialize it
   PS::DomainInfo dinfo;
   dinfo.initialize();
   // Set the boundary condition
```

```
    dinfo.setBoundaryCondition(PS::BOUNDARY_CONDITION_PERIODIC_XYZ);
    dinfo.setPosRootDomain(PS::F64vec(0.0, 0.0, 0.0),
                           PS::F64vec(box.x, box.y, box.z));
    // Perform domain decomposition
    dinfo.decomposeDomainAll(sph_system);
    // Exchange the SPH particles between the (MPI) processes
    sph_system.exchangeParticle(dinfo);
    // Make two tree structures
    // (one is for the density calculation and
    //  another is for the force calculation.)
    PS::TreeForForceShort<Dens, EP, EP>::Gather dens_tree;
    dens_tree.initialize(3 * sph_system.getNumberOfParticleGlobal());

    PS::TreeForForceShort<Hydro, EP, EP>::Symmetry hydr_tree;
    hydr_tree.initialize(3 * sph_system.getNumberOfParticleGlobal());
    // Compute density, pressure, acceleration due to pressure gradient
    dens_tree.calcForceAllAndWriteBack(CalcDensity(), sph_system, dinfo);
    setPressure(sph_system);
    hydr_tree.calcForceAllAndWriteBack(CalcHydroForce(), sph_system, dinfo)
          ;
    // Get timestep
    dt = getTimeStepGlobal(sph_system);
    // Main loop for time integration
    PS::S32 step = 0;
    for(PS::F64 time = 0 ; time < end_time ; time += dt, ++ step){
       // Leap frog: Initial Kick & Full Drift
       InitialKick(sph_system, dt);
       FullDrift(sph_system, dt);
       // Adjust the positions of the SPH particles that run over
       // the computational boundaries.
       sph_system.adjustPositionIntoRootDomain(dinfo);
       // Leap frog: Predict
       Predict(sph_system, dt);
       // Perform domain decomposition again
       dinfo.decomposeDomainAll(sph_system);
       // Exchange the SPH particles between the (MPI) processes
       sph_system.exchangeParticle(dinfo);
       // Compute density, pressure, acceleration due to pressure gradient
       dens_tree.calcForceAllAndWriteBack(CalcDensity(), sph_system, dinfo)
             ;
       setPressure(sph_system);
       hydr_tree.calcForceAllAndWriteBack(CalcHydroForce(), sph_system,
             dinfo);
       // Get a new timestep
       dt = getTimeStepGlobal(sph_system);
       // Leap frog: Final Kick
       FinalKick(sph_system, dt);
       // Output result files
       if(step % OUTPUT_INTERVAL == 0){
          FileHeader header;
          header.time = time;
          header.Nbody = sph_system.getNumberOfParticleGlobal();
          char filename[256];
          sprintf(filename, "result/%04d.txt", step);
          sph_system.writeParticleAscii(filename, header);
```

```
         if (PS::Comm::getRank() == 0){
            std::cout << "================================" << std::endl;
            std::cout << "output␣" << filename << "." << std::endl;
            std::cout << "================================" << std::endl;
         }
      }
      // Output information to STDOUT
      if (PS::Comm::getRank() == 0){
         std::cout << "================================" << std::endl;
         std::cout << "time␣=␣" << time << std::endl;
         std::cout << "step␣=␣" << step << std::endl;
         std::cout << "================================" << std::endl;
      }
      CheckConservativeVariables(sph_system);
   }
   // Finalize FDPS
   PS::Finalize();
   return 0;
}
```

6 Extensions

6.1 P^{3}M code

In this section, we explain the usage of a FDPS extension "Particle Mesh" (hereafter PM) using a sample program for P^{3}M(Particle-Particle-Particle-Mesh) method. The sample code calculates the crystal energy of sodium chloride (NaCl) crystal using the P^{3}M method and compares the result with the analytical solution. In the P^{3}M 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.

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

The sample code consists of `main.cpp` and a Makefile for GCC, `Makefile`.

6.1.2 User-defined types

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

6.1.2.1 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^{3}M method. 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

```
class Nbody_FP
{
   public:
      PS::S64 id;
      PS::F64 m;
      PS::F64 rc;
      PS::F64vec x;
      PS::F64vec v,v_half;
      PS::F64vec agrv;
      PS::F64 pot;
      // Member functions required by FDPS
      PS::F64 getCharge() const {
         return m;
      };
      PS::F64 getChargeParticleMesh() 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 copyFromForce(const Nbody_PP_Results& result) {};
      void copyFromForceParticleMesh(const PS::F64 apm) {};
};
```

6.1.2.2 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;
      };
};
```

6.1.2.3 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;
      PS::F64vec agrv;
      void clear() {
         pot  = 0.0;
         agrv = 0.0;
      }
};
```

6.1.2.4 calcForceEpEp

You must define an interaction function calcForceEpEp. calcForceEpEp must contain actual code for the PP part of the Force calculation and must be implemented as `void` function or functor (function object) . Its arguments 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 shows the implementation of calcForceEpEp in this sample code. In the code, it is implemented as a functor.

Listing 36: Interaction function calcForceEpEp

```
class Calc_force_ep_ep{
   public:
      void operator () (const Nbody_EP* const ep_i,
                        const PS::S32 Nip,
                        const Nbody_EP* const ep_j,
                        const PS::S32 Njp,
                        Nbody_PP_Results* const result) {
         for (PS::S32 i=0; i<Nip; i++) {
            for (PS::S32 j=0; j<Njp; j++) {
               PS::F64vec dx = ep_i[i].x - ep_j[j].x;
               PS::F64 rij = std::sqrt(dx * dx);
               if ((ep_i[i].id == ep_j[j].id) && (rij == 0.0)) continue;
               PS::F64 rinv = 1.0/rij;
               PS::F64 rinv3 = rinv*rinv*rinv;
               PS::F64 xi = 2.0*rij/ep_i[i].rc;
               result[i].pot  += ep_j[j].m * S2_pcut(xi) * rinv;
               result[i].agrv += ep_j[j].m * S2_fcut(xi) * rinv3 * dx;
            }
            //* Self-interaction term
            result[i].pot -= ep_i[i].m * (208.0/(70.0*ep_i[i].rc));
         }

      };
};
```

The PP part in the P^{3}M 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

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

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:

```
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.

6.1.2.5 calcForceEpSp

You must define an interaction function calcForceEpSp[2)]. calcForceEpSp must contain actual code for particle-superparticle interaction and must be implemented as `void` function or functor (function object) . Its arguments is an array of EssentialParticleI objects, the number of EssentialParticleI objects, an array of SuperParticleJ objects, the number of SuperParticleJ objects, and an array of Force objects. Listing 37 shows the implementation of calcForceEpSp in the sample code.

Listing 37: Interaction function calcForceEpSp

```
class Calc_force_ep_sp{
  public:
    void operator () (const Nbody_EP* const ep_i,
                      const PS::S32 Nip,
                      const PS::SPJMonopoleCutoff* const ep_j,
                      const PS::S32 Njp,
                      Nbody_PP_Results* const result) {
      for (PS::S32 i=0; i<Nip; i++) {
        for (PS::S32 j=0; j<Njp; j++) {
          PS::F64vec dx = ep_i[i].x - ep_j[j].pos;
          PS::F64 rij = std::sqrt(dx * dx);
          PS::F64 rinv = 1.0/rij;
          PS::F64 rinv3 = rinv*rinv*rinv;
          PS::F64 xi = 2.0*rij/ep_i[i].rc;
          result[i].pot  += ep_j[j].mass * S2_pcut(xi) * rinv;
          result[i].agrv += ep_j[j].mass * S2_fcut(xi) * rinv3 * dx;
        }
```

[2)] As describe at the beginning of this section, the sample code uses P^3M for the calculation of interaction. In order to realize it using FDPS, we perform the calculation of interaction with the opening angle criterion θ of 0. Hence, particle-superparticle interaction should not occur. However, API `calcForceAllAndWriteBack` requires a function pointer of a `void` function that calculates particle-superparticle interaction. Therefore, we defined calcForceEpSp here.

```
        }

    };
};
```

6.1.3 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^3M 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 (in `void` function `NaCl_IC()`)
(3) Compute the potential energy of each particle by the P^3M method (in `void` function `Nbody_objs.calc_gravity()`)
(4) Compute the total energy of the crystal and compare it with the analytical solution (`void` function `calc_energy_error()`)
(5) Repeat (2)-(4)

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

6.1.4 Include 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 38: Include FDPS's header files

```
#include <particle_simulator.hpp>
#include <particle_mesh.hpp>
#include <param.h>
#include <param_fdps.h>
```

6.1.4.1 Initialization and Termination of FDPS

First, you must initialize FDPS by the following code.

Listing 39: Initialization of FDPS

```
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 40: Termination of FDPS

```
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^{3}M 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 41: `Nbody_Objects` class

```
class Nbody_Objects {
   public:
      PS::ParticleSystem<Nbody_FP> system;
      PS::DomainInfo dinfo;
      PS::TreeForForceLong<Nbody_PP_Results, Nbody_EP, Nbody_EP>::
              MonopoleWithCutoff pp_tree;
      PS::PM::ParticleMesh pm;
}
```

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

Listing 42: Creation of a `Nbody_Objects`-class object

```
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 43: Initialization of a ParticleSystem object

```
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 44: Initialization of a DomainInfo object

```
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 APIs `setBoundaryCondition` and `setPosRootDomain` . In the sample code, these procedures are performed in `void` function `NaCl_IC()` that sets up the distribution of particles:

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

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

Listing 45: Initialization of a TreeForForceLong object

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

You need to give a rough number of particles to this API as the first argument. Here, we set three times the number of local particles at the time of calling. The second argument of this API 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).

```
if (is_tree_initialized == false) {
   Nbody_objs.init_tree();
   is_tree_initialized = true;
}
```

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 `void` 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 $(0, 0, 0)$, `void` function `NaCl_IC` makes a three-dimensional uniform grid of particles. These parameters are specified through an object of class `Crystal_Parameters` , `NaCl_params`:

```
class Crystal_Parameters
{
   public:
      PS::S32 numPtcl_per_side;
      PS::F64vec pos_vertex;
};
/* In main function */
Crystal_Parameters NaCl_params;
NaCl_IC(Nbody_objs.system,
        Nbody_objs.dinfo,
        NaCl_params);
```

In the first half of `void` 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

API `decomposeDomainAll` of the DomainInfo object is used to perform domain decomposition based on the current distribution of particles:

Listing 46: Domain Decomposition

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

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

6.1.4.3.2 Particle Exchange

API `exchangeParticle` of the ParticleSystem object is used to exchange particles based on the current decomposed domains:

Listing 47: Particle Exchange

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

Note that this API 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 48: Execution of interaction calculation

```
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 49: Interaction calculation

```
void calc_gravity() {
   //* Local variables
   PS::S32 numPtclLocal = system.getNumberOfParticleLocal();

   //* Reset potential and accelerations
   for (PS::S32 i=0; i<numPtclLocal; i++) {
      system[i].pot  = 0.0;
      system[i].agrv = 0.0;
   }

   //=================
   //* [1] PM part
   //=================
   pm.setDomainInfoParticleMesh(dinfo);
   pm.setParticleParticleMesh(system);
   pm.calcMeshForceOnly();
   for (PS::S32 i=0; i<numPtclLocal; i++) {
      PS::F32vec x32 = system[i].x;
      system[i].pot  -= pm.getPotential(x32);
      system[i].agrv -= pm.getForce(x32);
   }

   //=================
   //* [2] PP part
   //=================
   pp_tree.calcForceAll(Calc_force_ep_ep(),
                        Calc_force_ep_sp(),
                        system, dinfo);
   for (PS::S32 i=0; i<numPtclLocal; i++) {
      Nbody_PP_Results result = pp_tree.getForce(i);
      system[i].pot   += result.pot;
      system[i].agrv += result.agrv;
   }
};
```

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 50: PM part of Force calculation

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

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 51: PP part of Force calculation

```
pp_tree.calcForceAll(Calc_force_ep_ep(),
                     Calc_force_ep_sp(),
                     system, dinfo);
for (PS::S32 i=0; i<numPtclLocal; i++) {
   Nbody_PP_Results result = pp_tree.getForce(i);
   system[i].pot   += result.pot;
    system[i].agrv += result.agrv;
}
```

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.

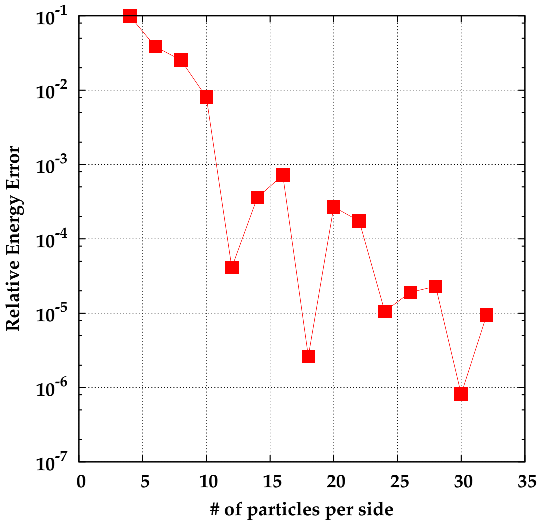

Figure 3: The relative error of the crystal energy as a function of the number of particles per side, where we assume that the number of the PM grids is 16^3 and the cutoff radius is 3/16.

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^{3}M method. Therefore, functions of FDPS used in the sample code is almost the same as the sample code for the P^{3}M method. The difference between the two is that the PP part is computed by the Tree method in the TreePM method while the P^{3}M 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 52: Include FDPS

```
#include <particle_simulator.hpp>
#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 53 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 53: FullParticle type

```
class FPtreepm {
private:
    template<class T>
    T reverseEndian(T value){
        char * first = reinterpret_cast<char*>(&value);
        char * last = first + sizeof(T);
        std::reverse(first, last);
        return value;
    }
```

```
public:
    PS::S64    id;
    PS::F32    mass;
    PS::F32    eps;
    PS::F64vec pos;
    PS::F64vec vel;
    PS::F64vec acc;
    PS::F64vec acc_pm;

    //static PS::F64vec low_boundary;
    //static PS::F64vec high_boundary;
    //static PS::F64 unit_l;
    //static PS::F64 unit_m;
    static PS::F64 H0;
    static PS::F64 Lbnd;

    PS::F64vec getPos() const {
        return pos;
    }

    PS::F64 getCharge() const {
        return mass;
    }

    void copyFromForce(const Result_treepm & force) {
        this->acc = force.acc;
    }

    PS::F64 getRSearch() const {
        PS::F64 rcut = 3.0/SIZE_OF_MESH;
        return rcut;
    }

    void setPos(const PS::F64vec pos_new) {
        this->pos = pos_new;
    }

    PS::F64 getChargeParticleMesh() const {
        return this->mass;
    }

    void copyFromForceParticleMesh(const PS::F64vec & acc_pm) {
        this->acc_pm = acc_pm;
    }

    /*
    void writeParticleBinary(FILE *fp) {
        int count;
        count = 0;

        count += fwrite(&mass,   sizeof(PS::F32),1,fp);
        count += fwrite(&eps,    sizeof(PS::F32),1,fp);
        count += fwrite(&pos[0], sizeof(PS::F64),1,fp);
        count += fwrite(&pos[1], sizeof(PS::F64),1,fp);
        count += fwrite(&pos[2], sizeof(PS::F64),1,fp);
        count += fwrite(&vel[0], sizeof(PS::F64),1,fp);
        count += fwrite(&vel[1], sizeof(PS::F64),1,fp);
        count += fwrite(&vel[2], sizeof(PS::F64),1,fp);
    }
    */
    /*
    int readParticleBinary(FILE *fp) {
        int count;
        count = 0;

        count += fread(&mass,   sizeof(PS::F32),1,fp);
        count += fread(&eps,    sizeof(PS::F32),1,fp);
        count += fread(&pos[0], sizeof(PS::F64),1,fp);
        count += fread(&pos[1], sizeof(PS::F64),1,fp);
        count += fread(&pos[2], sizeof(PS::F64),1,fp);
        count += fread(&vel[0], sizeof(PS::F64),1,fp);
        count += fread(&vel[1], sizeof(PS::F64),1,fp);
        count += fread(&vel[2], sizeof(PS::F64),1,fp);

        return count;
    }
    */

    void writeParticleBinary(FILE *fp) {
        PS::F32 x = pos[0];
        PS::F32 y = pos[1];
        PS::F32 z = pos[2];
        PS::F32 vx = vel[0];
        PS::F32 vy = vel[1];
        PS::F32 vz = vel[2];
        PS::S32 i = id;
        PS::S32 m = mass;
        fwrite(&x, sizeof(PS::F32),1,fp);
        fwrite(&vx, sizeof(PS::F32),1,fp);
        fwrite(&y, sizeof(PS::F32),1,fp);
        fwrite(&vy, sizeof(PS::F32),1,fp);
        fwrite(&z, sizeof(PS::F32),1,fp);
        fwrite(&vz, sizeof(PS::F32),1,fp);
        //fwrite(&mass,   sizeof(PS::F32),1,fp);
        fwrite(&m,   sizeof(PS::F32),1,fp);
        fwrite(&i,   sizeof(PS::F32),1,fp);
        //fwrite(&id,   sizeof(PS::F32),1,fp);
    }


    // for API of FDPS
    // in snapshot, L unit is Mpc/h, M unit is Msun, v unit is km/s
    void readBinary(FILE *fp){
        static PS::S32 ONE = 1;
        static bool is_little_endian = *reinterpret_cast<char*>(&ONE) ==
              ONE;
        static const PS::F64 Mpc_m = 3.08567e22; // unit is m
        static const PS::F64 Mpc_km = 3.08567e19; // unit is km
        static const PS::F64 Msun_kg = 1.9884e30; // unit is kg
        static const PS::F64 G = 6.67428e-11; // m^3*kg^-1*s^-2
        static const PS::F64 Cl = 1.0 / FPtreepm::Lbnd;
        static const PS::F64 Cv = 1.0 / (FPtreepm::Lbnd * FPtreepm::H0);
        static const PS::F64 Cm = 1.0 / (pow(Mpc_m*FPtreepm::Lbnd, 3.0) /
              pow(Mpc_km/FPtreepm::H0, 2.0) / G / Msun_kg);
        PS::F32 x, y, z, vx, vy, vz, m;
        PS::S32 i;
        fread(&x, 4, 1, fp);
        fread(&vx, 4, 1, fp);
        fread(&y, 4, 1, fp);
        fread(&vy, 4, 1, fp);
        fread(&z, 4, 1, fp);
        fread(&vz, 4, 1, fp);
        fread(&m,   4, 1, fp);
        fread(&i,   4, 1, fp);
        if( is_little_endian){
            pos.x = x * Cl;
            pos.y = y * Cl;
            pos.z = z * Cl;
            vel.x = vx * Cv;
            vel.y = vy * Cv;
            vel.z = vz * Cv;
            mass = m * Cm;
            //mass = m / 1.524e17;
            id = i;
        }
        else{
            pos.x = reverseEndian(x) * Cl;
            pos.y = reverseEndian(y) * Cl;
            pos.z = reverseEndian(z) * Cl;
            vel.x = reverseEndian(vx) * Cv;
            vel.y = reverseEndian(vy) * Cv;
            vel.z = reverseEndian(vz) * Cv;
            mass = reverseEndian(m) * Cm;
            //mass = reverseEndian(m) / 1.524e17;
            id = reverseEndian(i);
        }
    }

    // for API of FDPS
    void writeBinary(FILE *fp){
        static const PS::F64 Mpc_m = 3.08567e22; // unit is m
        static const PS::F64 Mpc_km = 3.08567e19; // unit is km
        static const PS::F64 Msun_kg = 1.9884e30; // unit is kg
        static const PS::F64 G = 6.67428e-11; // m^3*kg^-1*s^-2
        static const PS::F64 Cl = FPtreepm::Lbnd;
        static const PS::F64 Cv = (FPtreepm::Lbnd * FPtreepm::H0);
        static const PS::F64 Cm = (pow(Mpc_m*FPtreepm::Lbnd, 3.0) / pow(
              Mpc_km/FPtreepm::H0, 2.0) / G / Msun_kg);
        PS::F32vec x = pos * Cl;
        PS::F32vec v = vel * Cv;
        PS::F32 m = mass * Cm;
        PS::S32 i = id;
        fwrite(&x.x,   sizeof(PS::F32), 1, fp);
        fwrite(&v.x,   sizeof(PS::F32), 1, fp);
        fwrite(&x.y,   sizeof(PS::F32), 1, fp);
        fwrite(&v.y,   sizeof(PS::F32), 1, fp);
        fwrite(&x.z,   sizeof(PS::F32), 1, fp);
        fwrite(&v.z,   sizeof(PS::F32), 1, fp);
        fwrite(&m,   sizeof(PS::F32), 1, fp);
        fwrite(&i,   sizeof(PS::S32), 1, fp);
    }

    PS::F64 calcDtime(run_param &this_run) {
    PS::F64 dtime_v, dtime_a, dtime;
    PS::F64 vnorm, anorm;
    vnorm = sqrt(SQR(this->vel))+TINY;
    anorm = sqrt(SQR(this->acc+this->acc_pm))+TINY;

    dtime_v = this->eps/vnorm;
    dtime_a = sqrt(this->eps/anorm)*CUBE(this_run.anow);

    dtime = fmin(0.5*dtime_v, dtime_a);

    return dtime;
  }
};
```

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 54 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 54: EssentialParticleI type

```
class EPItreepm {
public:
  PS::S64    id;
  PS::F32    eps;
  PS::F64vec pos;

  PS::F64vec getPos() const {
    return this->pos;
  }

  void copyFromFP(const FPtreepm & fp) {
    this->id = fp.id;
    this->eps = fp.eps;
    this->pos = fp.pos;
  }

};
```

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^{3}M method (see § 6.1). Listing 55 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 55: EssentialParticleJ type

```
class EPJtreepm {
public:
  PS::S64    id;
  PS::F64vec pos;
  PS::F64    mass;
  //  PS::F64    rcut;

  PS::F64vec getPos() const {
    return this->pos;
  }

  PS::F64 getCharge() const {
    return this->mass;
  }

  void copyFromFP(const FPtreepm & fp) {
    this->id = fp.id;
    this->mass = fp.mass;
    this->pos = fp.pos;
  }

  PS::F64 getRSearch() const {
    PS::F64 rcut = 3.0/SIZE_OF_MESH;
    return rcut;
  }

  void setPos(const PS::F64vec pos_new) {
    this->pos = pos_new;
  }
};
```

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 56 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 56: 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 57 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 57: 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;
            }
        }
    }
};
```

The PP part of the TreePM method is a two-body interaction with cutoff as in the P^3M method. Hence, a cutoff function is involved in the calculation of gravitational acceleration. As explained in § 6.1.2.4, 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:

```
#ifdef ENABLE_PHANTOM_GRAPE_X86
    //g5_open();
    pg5_gen_s2_force_table(EPS_FOR_PP, 3.0/SIZE_OF_MESH);
#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 58: Initialization of FDPS

```
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 59: Termination of FDPS

```
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 60: Creation of FDPS objects

```
int main(int argc, char **argv)
{
   PS::PM::ParticleMesh pm;
   PS::ParticleSystem<FPtreepm> ptcl;
   PS::DomainInfo domain_info;
   PS::TreeForForceLong<Result_treepm, EPItreepm, EPJtreepm>::
          MonopoleWithCutoff treepm_tree;

}
```

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 61: Initialization of FDPS objects

```
int main(int argc, char **argv)
{
   // Initialize ParticleSystem
   ptcl.initialize();

   // Initialize DomainInfo
   domain_info.initialize();
   domain_info.setBoundaryCondition(PS::BOUNDARY_CONDITION_PERIODIC_XYZ);
   domain_info.setPosRootDomain(PS::F64vec(0.0, 0.0, 0.0),
                                PS::F64vec(1.0, 1.0, 1.0));

   // Initialize Tree
   treepm_tree.initialize(3*ptcl.getNumberOfParticleGlobal(),
                          this_run.theta);
}
```

- 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 `setBound-aryCondition` 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:

```
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 `de-composeDomainAll` method (see the `main` function):

Listing 62: Domain Decomposition

```
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 63: Particle Exchange

```
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 64: Interaction Calculation

```
//* PP part
treepm_tree.calcForceAllAndWriteBack
    (calc_pp_force<EPJtreepm>(),
     calc_pp_force<PS::SPJMonopoleCutoff>(),
     ptcl,
     domain_info);

//* PM part
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 § 4.1.3.4.4). 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.

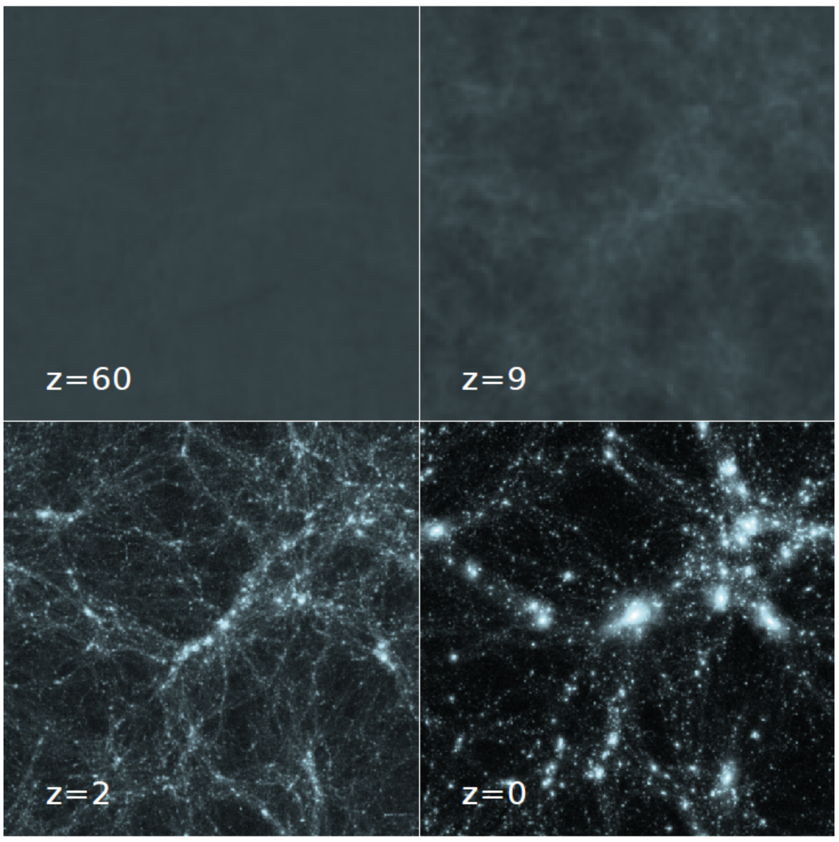

Figure 4: Time evolution of particle density of Santa Barbara Cluster Comparison test (the number of particles is 256^3)

7 Practical Applications

In previous sections, we have explained fundamental features of FDPS using relatively simple application codes. However, we need to develop a more complex application in actual research, in which for example we need to treat different types of particles. In this section, we will explain advanced features of FDPS using practical applications. To keep the expla nations short and simple, we require the readers understand the contents of the previous se ctions in this document.

7.1 N-body/SPH code

In this section, we explain the accompanying sample code for N-body/SPH simulation of a disk galaxy. In this code, dark matter and stars, which perform gravitational interaction only, are represented by N-body particles, while interstellar gas, which performs both gravitational and hydrodynamic interactions, is represented by SPH particles. The tree method is used for the gravity calculation. The SPH scheme adopted in this code is the one proposed by Springel & Hernquist [2002, MNRAS, 333, 649] and Springel [2005, MNRAS, 364, 1105] (hereafter, we call it Springel's SPH scheme). The readers can understand how to treat different types of particles using FDPS by reading this section.

Below, we first explain the usage of the code. Next, we give a brief explanation of the Springel's SPH scheme. Then, we explain the contents of the sample source codes in detail.

7.1.1 How to run the sample code

As we described, this code simulates the dynamical evolution of a disk galaxy. This code sets the initial distributions of dark matter and stars by reading a file created by MAGI (Miki & Umemura [2018, MNRAS, 475, 2269]), which is a software to make an initial condition of a galaxy simulation. On the other hand, the initial gas distribution is set inside the code. Therefore, the following procedures are required to use the code.

- Move to directory `$(FDPS)/sample/c++/nbody+sph`
- Edit `Makefile` in the current directory
- Create particle data using MAGI and place it under directory `./magi_data/dat`
- Run the `make` command to create the executable `nbodysph.out`
- Run `nbodysph.out`
- Check the output

Below, we explain each procedure.

7.1.1.1 Move to the directory the sample code is placed

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

7.1.1.2 File structure of the sample code

The following is the file structure of the sample code.

```
$ ls | awk '{print $0}'
Makefile
Makefile.K
Makefile.ofp
ic.hpp
job.K.sh
job.ofp.sh
leapfrog.hpp
macro_defs.hpp
magi_data/
main.cpp
mathematical_constants.cpp
mathematical_constants.h
physical_constants.cpp
physical_constants.h
test.py*
user_defined.hpp
```

We explain briefly the content of each source file. In `ic.hpp`, functions to create initial conditions are implemented. Users can choose an initial condition other than that for a disk galaxy (described later). In `leapfrog.hpp`, we implement functions necessary to integrate the orbits of particles based on the Leapfrog method. In `macro_defs.hpp`, we define macros that are used to control numerical simulation. In `main.cpp`, the main routine is implemented. In `mathematical_constants.h` and `mathematical_constants.cpp`, we define some mathematical constants. In `physical_constants.h` and `physical_constants.cpp`, we define some physical constants. In `user_defined.hpp`, we define user-defined classes and interaction functions.

Directory `magi_data` stores a parameter file input to the software MAGI (`magi_data/cfg/*`) and a script file used to run MAGI (`magi_data/sh/run.sh`).

7.1.1.3 Edit Makefile

Edit `Makefile` following the description below.

- Set the variable `CXX` the command to run your C++ compiler.
- Set the variable `CXXFLAGS` compile options of the C++ compiler.
- In this code, several macros are used to control numerical simulations. Table 1 lists the names of the macros and their definitions. In addition, there are macros whose states (i.e. value or defined/undefined states) are automatically set according to the value of macro `INITIAL_CONDITION`. Generally, users do not have to change them. Please see `macro_defs.h` directly for detail.

- Phantom-GRAPE library for x86 can be used for the gravity calculation. To use it, set the variable `use_phantom_grape_x86 yes`.

As for the way to specify the use/non-use of OpenMP and MPI, see § 3.

Macro name	Defintion
`INITIAL_CONDITION`	It specifies the type of initial condition or the operation mode of the code. It must take a value from 0 to 3. According to its value, the code operates as follows. 0: an initial condition for a disk galaxy is used, 1: an initial condition for cold collapse test problem is used, 2: an initial condition for Evrard test is used, 3: the code operates in the mode to make a glass-like distribution of SPH particles.
`ENABLE_VARIABLE_SMOOTHING_LENGTH`	It specifies that smoothing length of SPH particles is variable or not. If it is defined, variable smoothing length is used and the SPH calculation is performed according to the Springel's SPH scheme. If it is not defined, the fixed smoothing length is used and the SPH calculation is done in almost the same way as the sample code described in § 3-4.
`USE_ENTROPY`	It specifies whether to use entropy or specific internal energy as an independent variable to describe the thermodynamic state of SPH particle. If defined, entropy is used. But, if macro `ISOTHERMAL_EOS` described below is defined, specific internal energy is forcibly used (specific internal energy is used to calculate pressure).
`USE_BALSARA_SWITCH`	It specifies whether Balsara switch (Balsara [1995, JCP, 121, 357]) is used or not. If defined, the Balsara switch is used.
`USE_PRESCR_OF_THOMAS_COUCHMAN_1992`	It specifies whether a simple prescription proposed by Thomas & Couchman [1992, MNRAS,257, 11] to prevent the tensile instability is used or not. If defined, this prescription is used.
`ISOTHERMAL_EOS`	It specifies whether isothermal process is assumed or not. If defined, isothermal process is assumed (specific internal energy is assumed to be constant). If not defined, the code solve the entropy equation or the internal energy equation.
`READ_DATA_WITH_BYTESWAP`	It specifies whether the program reads particle data with performing byte swap (byte swap is applied for each variable of basic data type). If defined, byte swap is performed.

Table 1: Compile-time macros and their definitions

7.1.1.4 Create particle data using MAGI

As described earlier, users need to create particle data using the software MAGI before simulation according to the procedures described below. For users who cannot use `MAGI` for some reasons, we prepared sample particle data in web sites described below. In the following, we explain each case in detail.

Create particle data using MAGI Create particle data as follows.

1. Download the source file of MAGI from the web side https://bitbucket.org/ymiki/magi and install it in appropriate PATH according to the descriptions in Section "How to compile MAGI" in the above web side. But, our N-body/SPH sample code supports TIPSY file format only. Therefore, please build MAGI with `USE_TIPSY_FORMAT=ON`.
2. Edit `./magi_data/sh/run.sh` and set the variable `MAGI_INSTALL_DIR` the PATH of the directory where the `magi` command is stored. Also, set the variable `NTOT` the number of N-body particles (MAGI automatically assigns the numbers of dark matter particles and star particles).
3. Edit `./magi_data/cfg/*` to specify a galaxy model. For detail of the format of input file for MAGI, please see the web side above or Section 2.4 in the original paper Miki & Umemura [2018, MNRAS, 475, 2269]. In the default, galaxy model consists of the following four components (hereafter, we call this **default galaxy model**):
   (i) Dark matter halo (NFW profile, $M = 10^{12}$ M$_\odot$, $r_s = 21.5$ kpc, $r_c = 200$ kpc, $\Delta_c = 10$ kpc)
   (ii) Stellar bulge (King model, $M = 5 \times 10^{10}$ M$_\odot$, $r_s = 0.7$ kpc, $W_0 = 5$)
   (iii) Thick stellar disk (Sérsic profile, $M = 2.5 \times 10^{10}$ M$_\odot$, $r_s = 3.5$ kpc, $n = 1.5$, $z_d = 1$ kpc, $Q_{T,\mathrm{min}} = 1.0$)
   (iv) Thin stellar disk (exponential disk, $M = 2.5 \times 10^{10}$ M$_\odot$, $r_s = 3.5$ kpc, $z_d = 0.5$ kpc, $Q_{T,\mathrm{min}} = 1.0$)

   In the default galaxy model, two stellar disks are marginally unstable to a bar-mode in view of the Ostriker-Peebles criterion. Therefore, a simulated galaxy is expected to evolve into a spiral galaxy having a weak bar. In the latest release of MAGI (version 1.1.1 [as of July 19th, 2019]), its default operation mode is changed from previous releases. With this demand, we have replaced parameter f in thick and thin disks by $Q_{T,\mathrm{min}}$, where f is a parameter controlling the velocity dispersion of disk and is used in the previous releases of MAGI to specify the stability of a disk component. $Q_{T,\mathrm{min}}$ is the minimum of Toomre Q value in the disk. (In the sample code in FDPS 5.0d or earlier, we used $f = 0.125$).
4. Move to directory `magi_data` and run the following command:

```
$ ./sh/run.sh
```

5. If MAGI stops successfully, particle data whose extension is `tipsy` will be created in directory `magi_data/dat`.

Download sample particle data form our web sites Download a particle data file from one of the following URLs and place it under directory `./magi_data/dat/`. All of particle data is made with the default galaxy model. Only the number of particles is different for each data.

- $N = 2^{21}$: http://particle.riken.jp/~fdps/magi_data/Galaxy/21/Galaxy.tipsy
- $N = 2^{22}$: http://particle.riken.jp/~fdps/magi_data/Galaxy/22/Galaxy.tipsy
- $N = 2^{23}$: http://particle.riken.jp/~fdps/magi_data/Galaxy/23/Galaxy.tipsy
- $N = 2^{24}$: http://particle.riken.jp/~fdps/magi_data/Galaxy/24/Galaxy.tipsy

7.1.1.5 Run make

Type "make" to run the `make` command.

7.1.1.6 Run the sample code

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

```
$ ./nbodysph.out
```

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

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

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

7.1.1.7 Analysis of the result

In the directory `result`, data of N-body and SPH particles are output as files "nbody0000x.dat" and "sph0000x.dat", where x is an integer representing time. The output file format of N-body particle data is that in each line, index of particle, mass, position (x, y, z), velocity (vx, vy, vz) are listed. The output file format of SPH particle data is that in each line, index of particle, mass, positon (x, y, z), velocity (vx, vy, vz), density, specific internal energy, entropy, pressure are listed.

Figure 5 shows the distribution of star and SPH particles at $T = 0.46$ for a disk galaxy simulation with the number of N-body particles is 2^{21} and the number of SPH particles is 2^{18}.

Below, we briefly explain the Springel's SPH scheme and then explain the implementation of the sample code.

7.1.2 Springel's SPH scheme

Springel & Hernquist [2002, MNRAS, 333, 649] proposed a formulation of SPH (actually, equation of motion[EoM]) where the total energy and entropy of a system are conserved even

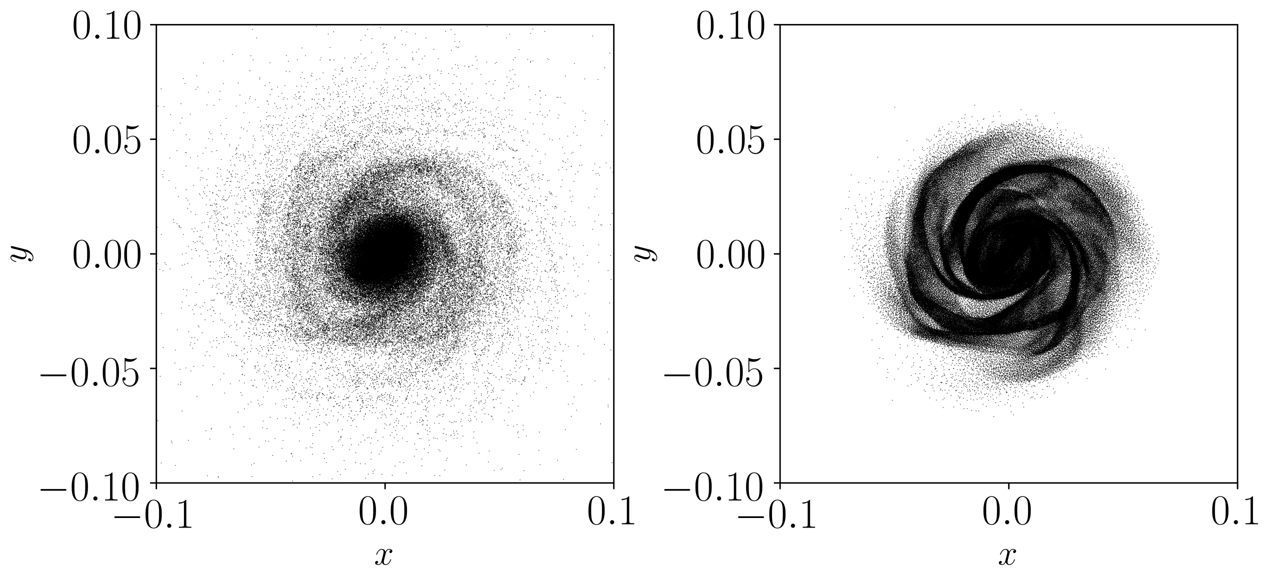

Figure 5: Face-on view of distributions of stars (left) and gas (right) (simulation configuration: the simulation is performed the number of N-body particles is 2^{21}, the number of SPH particles is 2^{18}, isothermal, gas temperature is 10^4 K, mean molecular weight to the mass of hydrogen $\mu = 0.5$)

if smoothing length changes with time. In this section, we briefly explain their formulation. The outline of the derivation is as follows. Construct a Lagrangian of the system assuming that smoothing length is also independent variable, then solve the Euler-Lagrange equations under N constraints, where N is the number of particles.

More specifically, they consider the Lagrangian

$$L(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \frac{1}{2}\sum_{i=1}^{N} m_i \dot{\boldsymbol{r}}_i^2 - \frac{1}{\gamma - 1}\sum_{i=1}^{N} m_i A_i \rho_i^{\gamma-1} \tag{7}$$

where $\boldsymbol{q} = (\boldsymbol{r}_1, ..., \boldsymbol{r}_N, h_1, ...h_N)$ is the generalized coordinates (the subscripts represent the indice of particles), $\boldsymbol{r}_i$ is the position, h_i is smoothing length, m_i is mass, γ is the ratio of specific heats, ρ_i is density, A_i is called entropy function and it is related with specific internal energy u_i and ρ_i through the equation

$$u_i = \frac{A_i}{\gamma - 1}\rho_i^{\gamma-1} \tag{8}$$

The first and second terms of Eq.(7) represents the kinetic energy and the internal energy of the system, respectively. Because solving the Euler-Lagrangian equation directly using this Lagrangian results in $4N$ equations, which is not undesirable, they introduce the following N constraints.

$$\phi_i = \frac{4\pi}{3} h_i^3 \rho_i - \overline{m} N_{\rm neigh} = 0 \tag{9}$$

where $\overline{m}$ is the average mass of SPH particles[3)], $N_{\rm neigh}$ is the number of neighbor particles (constant). Under these constraints, using the method of Lagrange multiplier, they solve

[3)]This must be treated as constant.

the Euler-Lagrange equations to obtain the following equations of motion:

$$\frac{\mathrm{d}\boldsymbol{v}_i}{\mathrm{d}t} = -\sum_{j=1}^{N} m_j \left[f_i \frac{P_i}{\rho_i^2} \nabla_i W(r_{ij}, h_i) + f_j \frac{P_j}{\rho_j^2} \nabla_i W(r_{ij}, h_j) \right] \tag{10}$$

where P_i is pressure, $r_{ij} = |\boldsymbol{r}_i - \boldsymbol{r}_j|$, W is the kernel function, f_i is the so-called ∇h term, defined by

$$f_i = \left(1 + \frac{h_i}{3\rho_i} \frac{\partial \rho_i}{\partial h_i} \right)^{-1} \tag{11}$$

The thermodynamic state of the system is described by the independent variable A_i, the entropy. If the flow is adiabatic, the entropy is constant along the flow except for locations of shock waves where the entropy is increased. Springel [2005, MNRAS, 364, 1105] modeled the increase of the entropy by passing shock waves using the method of artificial viscosity:

$$\frac{\mathrm{d}A_i}{\mathrm{d}t} = \frac{1}{2} \frac{\gamma - 1}{\rho_i^{\gamma-1}} \sum_{j=1}^{N} m_j \Pi_{ij} \boldsymbol{v}_{ij} \cdot \nabla_i \overline{W}_{ij} \tag{12}$$

$$\left. \frac{\mathrm{d}\boldsymbol{v}_i}{\mathrm{d}t} \right|_{\mathrm{visc}} = -\sum_{j=1}^{N} m_j \Pi_{ij} \nabla_i \overline{W}_{ij} \tag{13}$$

where $\boldsymbol{v}_{ij} = \boldsymbol{v}_i - \boldsymbol{v}_j$, $\boldsymbol{v}_i$ is velocity, $\overline{W}_{ij} = \frac{1}{2}(W(r_{ij}, h_i) + W(r_{ij}, h_j))$. For Π_{ij}, please see the original papers.

The procedures of SPH calculation is summarized as follows:

(1) Solve Eq.(9) and the following equation self-consistently to determine the density ρ_i and the smoothing length h_i.

$$\rho_i = \sum_{j=1}^{N} m_j W(r_{ij}, h_i) \tag{14}$$

(2) Calculate ∇h term defined by Eq.(11).
(3) Calculate the right-hand side of Eqs.(10), (12)、(13).
(4) Update the positions, velocities, entropies of SPH particles.

In the remaining sections, we first explain the implementations of user-defined classes and interaction functions. Then, we explain the implementation of the main routine where we explain how to treat different types of particles in FDPS.

7.1.3 User-defined types

All user-defined types are defined in `user_defined.hpp` . Here, we explain the types of user-defined types used in this code. As described earlier, this code use two types of particles, N-body and SPH particles. Thus, this code defines **two FullParticle types** (`FP_nbody` class for N-body particles and `FP_sph` class for SPH particles). The number of types of *physical* interactions are two, the gravitational and hydrodynamic interactions. But, as explained in

§ 4, we need to perform (at least) two interaction calculations (for density and acceleration) in SPH calculations. Therefore, the code defines **three** Force types (`Force_grav` class for the gravity calculation, `Force_dens` class for the density calculation, and `Force_hydro` class for the calculation of acceleration due to pressure gradient (hereafter we call it pressure-gradient acceleration for simplicity)). For simplicity, this code uses one class for both EssentialParticleI type and EssentialParticleJ type (hereafter, we call them together EssentialParticle type). Also this code uses the same EssentialParticle type for the calculations of density and pressure-gradient acceleration. Therefore, the number of types of EssentialParticle types is **two** (`EP_grav` class for the gravity calculation and `EP_hydro` class for SPH calculation).

Below, we explain the implementation of each user defined type.

7.1.3.1 FullParticle type

First, we explain `FP_nbody` class , which is used to store the information of N-body particles. This data type contains all physical quantities that a N-body particle should have as member variables. Listing 65 shows the implementation of `FP_nbody` class . The definitions of the member variables and member functions are almost the same as those of N-body sample code introduced in § 3-4. Thus, please see the corresponding section for detail.

Listing 65: FullParticle type (`FP_nbody` class)

```
class FP_nbody{
public:
    PS::S64     id;
    PS::F64     mass;
    PS::F64vec pos;
    PS::F64vec vel;
    PS::F64vec acc;
    PS::F64     pot;

    PS::F64vec getPos() const {
        return pos;
    }
    PS::F64 getCharge() const {
        return mass;
    }
    void setPos(const PS::F64vec& pos){
       this->pos = pos;
    }
    void copyFromForce(const Force_grav & f) {
        this->acc = f.acc;
        this->pot = f.pot;
    }
    void writeAscii(FILE* fp) const {
        fprintf(fp, "%lld\t%g\t%g\t%g\t%g\t%g\t%g\t%g\n",
                this->id, this->mass,
                this->pos.x, this->pos.y, this->pos.z,
                this->vel.x, this->vel.y, this->vel.z);
    }
    void readAscii(FILE* fp) {
        fscanf(fp, "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\n",
               &this->id, &this->mass,
                  &this->pos.x, &this->pos.y, &this->pos.z,
                  &this->vel.x, &this->vel.y, &this->vel.z);
    }
};
```

Next, we explain `FP_sph` class , which is used to store the information of SPH particles. This data type contains all physical quantities that a SPH particle should have as member variables. Listing 66 shows the implementation of `FP_sph` class . The definitions of main member variables are as follows: `id` (identification number), `mass` (mass), `pos` (position[$\boldsymbol{r}_i$]), `vel` (velocity[$\boldsymbol{v}_i$]), `acc_grav` (gravitational acceleration), `pot_grav` (gravitational potential), `acc_hydro` (pressure-gradient acceleration), `dens` (density[ρ_i]), `eng` (specific internel energy[u_i]), `ent` (entropy function [hereafter, entropy][A_i]), `pres` (pressure[P_i]), `smth` (smoothing length[4][h_i]), `gradh` (∇h term[f_i]), `divv` ($(\nabla \cdot \boldsymbol{v})_i$, where the subscript i means that the derivative is performed at particle position), `rotv` ($(\nabla \times \boldsymbol{v})_i$), `BalSW` (coefficient for Balsara switch and its definition is the same as $f(a)$ in Balsara [1995, JCP, 121, 357]), `snds` (sound speed), `eng_dot` (time rate of change of `eng`), `ent_dot` (time rate of change of `ent`), `dt` (the maximum allowable time step to integrate the orbit of this particle).

The configuration of member functions are almost the same as those of the SPH sample code introduced in § 3-4, but there are the following differences:

- SPH particles are involved with three types of interaction calculations (gravity, density, pressure-gradient acceleration). Thus, **three** types of member function `copyFromForce` are defined.
- The existence of member function `writeBinaryPos`. This function is only used in the case of `INITIAL_CONDITION=3`.
- The existence of member function `setEntropy`. This function is used to set the initial value of entropy.

For the other member functions, please see § 3-4.

Listing 66: FullParticle type (`FP_sph` class)

```
class FP_sph {
public:
    PS::S64    id;
    PS::F64    mass;
    PS::F64vec pos;
    PS::F64vec vel;
    PS::F64vec acc_grav; // gravitational acceleration
    PS::F64    pot_grav; // gravitational potential
    PS::F64vec acc_hydro; // acceleration due to pressure-gradient
    PS::S32    flag;
    PS::F64    dens; // mass density
    PS::F64    eng; // specific internal energy
    PS::F64    ent; // entropy
    PS::F64    pres; // pressure
    PS::F64    smth; // smoothing length
    PS::F64    gradh; // grad-h term
    PS::F64    divv; // divergence of velocity
```

[4]It is defined as the distance from the center of a particle where the value of the SPH kernel function is 0.

```
    PS::F64vec rotv; // rotation of velocity
    PS::F64    BalSW; // Balsara switch
    PS::F64    snds; // sound speed
    PS::F64    eng_dot; // time rate of change of ‘eng‘
    PS::F64    ent_dot; // time rate of change of ‘ent‘
    PS::F64    dt; // hydrodynamic time step for this particle
    PS::F64vec vel_half;
    PS::F64    eng_half;
    PS::F64    ent_half;

    void copyFromForce(const Force_grav& f) {
        this->acc_grav = f.acc;
        this->pot_grav = f.pot;
    }
    void copyFromForce(const Force_dens& f){
        this->flag  = f.flag;
        this->dens  = f.dens;
        this->smth  = f.smth;
        this->gradh = f.gradh;
        this->divv  = f.divv;
        this->rotv  = f.rotv;

    }
    void copyFromForce(const Force_hydro& f){
        this->acc_hydro = f.acc;
        this->eng_dot   = f.eng_dot;
        this->ent_dot   = f.ent_dot;
        this->dt        = f.dt;
    }
    PS::F64 getCharge() const{
        return this->mass;
    }
    PS::F64vec getPos() const{
        return this->pos;
    }
    PS::F64 getRSearch() const{
        return this->smth;
    }
    void setPos(const PS::F64vec& pos){
       this->pos = pos;
    }
    void writeAscii(FILE* fp) const{
        fprintf(fp,
                "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t"
                "%lf\t%lf\t%lf\t%lf\t%lf\t%lf\n",
                this->id, this->mass,
                this->pos.x, this->pos.y, this->pos.z,
                this->vel.x, this->vel.y, this->vel.z,
                this->dens, this->eng, this->ent, this->pres);
    }
    void readAscii(FILE* fp){
        fscanf(fp,
               "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t"
               "%lf\t%lf\t%lf\t%lf\t%lf\t%lf\n",
               &this->id, &this->mass,
                &this->pos.x, &this->pos.y, &this->pos.z,
                &this->vel.x, &this->vel.y, &this->vel.z,
                &this->dens, &this->eng, &this->ent, &this->pres);
    }
    void writeBinaryPos(FILE* fp) const {
        fwrite(&this->pos, sizeof(this->pos), 1, fp);
    }
    void setEntropy(){
        ent = (specific_heat_ratio - 1.0) * eng / std::pow(dens,
              specific_heat_ratio - 1.0);
    }
    void setPressure(){
#if defined(ISOTHERMAL_EOS)
        // In this case, eng = const.
        pres = (specific_heat_ratio - 1.0) * dens * eng;
        ent  = pres / std::pow(dens, specific_heat_ratio);
#else
#if defined(USE_ENTROPY)
        pres = ent * std::pow(dens, specific_heat_ratio);
        eng  = pres / ((specific_heat_ratio - 1.0) * dens);
#else
        pres = (specific_heat_ratio - 1.0) * dens * eng;
        ent  = pres / std::pow(dens, specific_heat_ratio);
#endif
#endif
        snds = std::sqrt(specific_heat_ratio * pres / dens);
#if defined(USE_BALSARA_SWITCH)
        BalSW = std::fabs(divv) / (std::fabs(divv) + std::sqrt(rotv * rotv
              ) + 1.0e-4 * snds / smth);
#else
        BalSW = 1.0;
#endif
    }
};
```

7.1.3.2 EssentialParticle type

First, we explain `EP_grav` class , which is used for the gravity calculation. This data type has all physical quantities that i- and j-particles should have in order to perform gravity calculation as member variables. Listing 67 shows the implementation of `EP_grav` class . EssentialParticle type should have member function(s) `copyFromFP()` to copy data from a FullParticle type. In this code, there are two FullParticle types and hence **two** `copyFromFP` functions are defined.

Listing 67: EssentialParticle type (`EP_grav` class)

```
class EP_grav {
public:
    PS::S64 id;
    PS::F64 mass;
    PS::F64vec pos;

    void copyFromFP(const FP_nbody& fp) {
        this->id   = fp.id;
        this->mass = fp.mass;
        this->pos  = fp.pos;
    }
    void copyFromFP(const FP_sph& fp) {
        this->id   = fp.id;
        this->mass = fp.mass;
        this->pos  = fp.pos;
    }
    PS::F64 getCharge() const {
        return this->mass;
    }
    PS::F64vec getPos() const {
        return this->pos;
    }
};
```

Next, we explain `EP_hydro` class , which is used for the calculations of density and pressure-gradient acceleration. This data type has all physical quantities that i- and j-partiles should have in order to perform the calculations of density and pressure-gradient acceleration. Listing 68 shows the implementation of `EP_hydro` class . Note that member function `getRSearch` returns `smth` multiplied by a coefficient `SCF_smth` (it has a value larger than 1, but nearly equal to 1) instead of `smth` itself. This gimmick is introduced to perform the density calculation efficiently. For detail, please see § 7.1.4.2.

Listing 68: EssentialParticle type (`EP_hydro` class)

```
class EP_hydro {
public:
   PS::S64    id;
   PS::F64vec pos;
   PS::F64vec vel;
   PS::F64    mass;
   PS::F64    smth;
   PS::F64    dens;
   PS::F64    pres;
   PS::F64    gradh;
   PS::F64    snds;
   PS::F64    BalSW;

   void copyFromFP(const FP_sph& fp){
      this->id    = fp.id;
      this->pos   = fp.pos;
      this->vel   = fp.vel;
      this->mass  = fp.mass;
      this->smth  = fp.smth;
      this->dens  = fp.dens;
      this->pres  = fp.pres;
      this->gradh = fp.gradh;
      this->snds  = fp.snds;
      this->BalSW = fp.BalSW;
   }
   PS::F64vec getPos() const{
      return this->pos;
   }
   PS::F64 getRSearch() const{
        return SCF_smth * this->smth;
    }
    void setPos(const PS::F64vec& pos){
        this->pos = pos;
    }
};
```

7.1.3.3 Force type

First, we explain `Force_grav` class , which is a Force type used for the gravity calculation. This data type must have all physical quantities that are obtained as the result of the gravity calculation. Listing 69 shows the implementation of `Force_grav` class .

Listing 69: Force type (`Force_grav` class)

```
class Force_grav {
public:
    PS::F64vec acc;
    PS::F64 pot;
    void clear() {
        acc = 0.0;
        pot = 0.0;
    }
};
```

Next, we explain `Force_dens` class , which is a Force type used for the density calculation. This data type must have all physical quantities that are obtained as the result of the density calculation. Listing 70 shows the implementation of `Force_dens` class . In the Springel's SPH scheme, the smoothing length h_i changes depending on the density at the position of a particle, ρ_i. In other words, h_i is also updated with ρ_i. Therefore, there is member variable `smth` to store updated smoothing length. In this code, we calculate ∇h term, $(\nabla \cdot \boldsymbol{v})_i$ $(\nabla \times \boldsymbol{v})_i$ at the same time (if `USE_BALSARA_SWITCH` is defined). Thus, there are member variables `gradh`, `divv`, `rotv` to store them. Member variable `flag` is used to store the result of iteration calculation of ρ_i and h_i (for detail, see § 7.1.4.2)。

Listing 70: Force type (`Force_dens` class)

```
class Force_dens{
public:
    PS::S32 flag;
    PS::F64 dens;
    PS::F64 smth;
    PS::F64 gradh;
    PS::F64 divv;
    PS::F64vec rotv;
    void clear(){
        flag  = 0;
        dens  = 0.0;
        gradh = 0.0;
        divv  = 0.0;
        rotv  = 0.0;
    }
};
```

Finally, we explain `Force_hydro` class , which is a Force type used for the calculation of pressure-gradient acceleration. This data type must have all physical quantities that are obtained as the result of the calculation of pressure-gradient acceleration. Listing 71 shows the implementation of `Force_hydro` class .

Listing 71: Force type (`Force_hydro` class)

```
class Force_hydro{
public:
   PS::F64vec acc;
   PS::F64 eng_dot;
   PS::F64 ent_dot;
   PS::F64 dt;
   void clear(){
      acc = 0.0;
      eng_dot = 0.0;
      ent_dot = 0.0;
   }
};
```

7.1.4 Interaction functions

All interaction functions are implemented in `user_defined.hpp` . There are **three** types of interaction functions. Below, we explain them.

7.1.4.1 Interaction function for the gravity calculation

Interaction functions for the gravity calculation are implemented as function template `CalcGravity` . Listing 72 shows the implementation. The implementation is almost the same as that of the N-body sample code introduced in § 3-4. For detail, please the corresponding section.

Listing 72: Interaction function for the gravity calculation

```
#if defined(ENABLE_PHANTOM_GRAPE_X86)
template <class TParticleJ>
void CalcGravity(const EP_grav * iptcl,
                 const PS::S32 ni,
                 const TParticleJ * jptcl,
                 const PS::S32 nj,
                 Force_grav * force) {
    const PS::S32 nipipe = ni;
    const PS::S32 njpipe = nj;
    PS::F64 (*xi)[3] = (PS::F64 (*)[3])malloc(sizeof(PS::F64) * nipipe *
          PS::DIMENSION);
    PS::F64 (*ai)[3] = (PS::F64 (*)[3])malloc(sizeof(PS::F64) * nipipe *
          PS::DIMENSION);
    PS::F64  *pi     = (PS::F64  *    )malloc(sizeof(PS::F64) * nipipe);
    PS::F64 (*xj)[3] = (PS::F64 (*)[3])malloc(sizeof(PS::F64) * njpipe *
          PS::DIMENSION);
    PS::F64  *mj     = (PS::F64  *    )malloc(sizeof(PS::F64) * njpipe);
    for(PS::S32 i = 0; i < ni; i++) {
        xi[i][0] = iptcl[i].getPos()[0];
        xi[i][1] = iptcl[i].getPos()[1];
        xi[i][2] = iptcl[i].getPos()[2];
        ai[i][0] = 0.0;
        ai[i][1] = 0.0;
        ai[i][2] = 0.0;
        pi[i]    = 0.0;
    }
    for(PS::S32 j = 0; j < nj; j++) {
        xj[j][0] = jptcl[j].getPos()[0];
        xj[j][1] = jptcl[j].getPos()[1];
        xj[j][2] = jptcl[j].getPos()[2];
        mj[j]    = jptcl[j].getCharge();
    }
    PS::S32 devid = PS::Comm::getThreadNum();
    g5_set_xmjMC(devid, 0, nj, xj, mj);
    g5_set_nMC(devid, nj);
    g5_calculate_force_on_xMC(devid, xi, ai, pi, ni);
    for(PS::S32 i = 0; i < ni; i++) {
        force[i].acc[0] += ai[i][0];
        force[i].acc[1] += ai[i][1];
        force[i].acc[2] += ai[i][2];
        force[i].pot    -= pi[i];
    }
    free(xi);
    free(ai);
    free(pi);
    free(xj);
    free(mj);
}
#else
template <class TParticleJ>
void CalcGravity (const EP_grav * ep_i,
                  const PS::S32 n_ip,
                  const TParticleJ * ep_j,
                  const PS::S32 n_jp,
                  Force_grav * force) {
    const PS::F64 eps2 = eps_grav * eps_grav;
    for(PS::S32 i = 0; i < n_ip; i++){
        PS::F64vec xi = ep_i[i].getPos();
        PS::F64vec ai = 0.0;
        PS::F64 poti = 0.0;
        for(PS::S32 j = 0; j < n_jp; j++){
            PS::F64vec rij    = xi - ep_j[j].getPos();
            PS::F64    r3_inv = rij * rij + eps2;
            PS::F64    r_inv  = 1.0/sqrt(r3_inv);
            r3_inv  = r_inv * r_inv;
            r_inv  *= ep_j[j].getCharge();
            r3_inv *= r_inv;
            ai     -= r3_inv * rij;
            poti   -= r_inv;
        }
        force[i].acc += ai;
        force[i].pot += poti;
    }
}
```

```
#endif
```

7.1.4.2 Interaction function for the density calculation

Interaction function for the density calculation is implemented as function object `CalcDensity` . Listing 73 shows its implementation. The implementation actually used differs depending on the state of macro `ENABLE_VARIABLE_SMOOTHING_LENGTH`. If this macro is not defined, an implementation for fixed smoothing length is used. Its source code is almost the same as the interaction function for the density calculation of the SPH sample code described in § 3-4. Thus, we omit explanation for this case. Below, we explain an implementation used for the case that the above macro is defined.

As described in § 7.1.2, we need to determine the density ρ_i and smoothing length h_i at the same time by solving Eqs.(14) and (9) self-consistently. For this, we need to perform an iterative calculation. This calculation is performed in the infinite `for` loop in the code. As you'll see by reading the source code of member function `getRSearch() of EP_hydro class` , this sample code performs the density calculation after multiplying the smoothing lengths of all particles by a constant `SCF_smth` in order to make the density calculation efficiently. By this, we can change h_i between 0 and $h_{\mathrm{max,alw}} \equiv$ `SCF_smth` $\times h_{i,0}$, during the iteration, where $h_{i,0}$ is the value of the smoothing length of particle i before we multiply by `SCF_smth`. This is because all of particles that is eligible to be j-particles are contained in the current j-particle list (`ep_j`). If the iteration does not converge for some particle i, we cannot determine ρ_i and h_i for this particle by using the current j particle list because the value of the smoothing length we want to obtain will be larger than $h_{\mathrm{max,alw}}$. In this case, we need to perform the density calculation again after increasing $h_{i,0}$. This "outer" iteration is performed in `void` function `calcDensity`(note that the first letter of the function name is lower case) in `main.cpp` . We will describe this `void` function in § 7.1.5.

After the infinite `for` loop, this `void` function performs the calculations of ∇h, $(\nabla \cdot \boldsymbol{v})_i$, and $(\nabla \times \boldsymbol{v})_i$.

Listing 73: Interaction function for the density calculation

```
class CalcDensity{
public:
    void operator () (const EP_hydro * ep_i,
                      const PS::S32 n_ip,
                      const EP_hydro * ep_j,
                      const PS::S32 n_jp,
                      Force_dens * force){
#if defined(ENABLE_VARIABLE_SMOOTHING_LENGTH)
        const PS::F64 eps = 1.0e-6;
        const PS::F64 M_trgt = mass_avg * N_neighbor;
        for (PS::S32 i = 0; i < n_ip; i++) {
            PS::F64 dens = 0.0;
            PS::F64 h = ep_i[i].smth;
            const PS::F64 h_max_alw = SCF_smth * h; // maximum allowance
            PS::F64 h_L = 0.0;
            PS::F64 h_U = h_max_alw;
            PS::F64 dh_prev = 0.0;
            PS::S32 n_unchanged = 0;
            // Software caches
            PS::F64 * mj  = (PS::F64 *)malloc(sizeof(PS::F64) * n_jp);
            PS::F64 * rij = (PS::F64 *)malloc(sizeof(PS::F64) * n_jp);
            for (PS::S32 j = 0; j < n_jp; j++) {
                mj[j] = ep_j[j].mass;
                const PS::F64vec dr = ep_j[j].pos - ep_i[i].pos;
                rij[j] = std::sqrt(dr * dr);
            }
            for (;;) {
                // Calculate density
                dens = 0.0;
                for (PS::S32 j = 0; j < n_jp; j++) {
                   dens += mj[j] * W(rij[j], h);
                }
                // Check if the current value of the smoohting length satisfies
                // Eq.(5) in Springel (2005).
                const PS::F64 M = 4.0 * math_const::pi * h * h * h * dens / 3.0;
                if ((h < h_max_alw) && (std::abs(M/M_trgt - 1.0) < eps)) {
                    // In this case, Eq.(5) holds within a specified accuracy.
                    force[i].flag = 1;
                    force[i].dens = dens;
                    force[i].smth = h;
                    break;
                }
                if (((h == h_max_alw) && (M < M_trgt)) || (n_unchanged == 4)) {
                    // In this case, we skip this particle forcibly.
                    // In order to determine consistently the density
                    // and the smoohting length for this particle,
                    // we must re-perform calcForceAllAndWriteBack().
                    force[i].flag = 0;
                    force[i].dens = dens;
                    force[i].smth = h_max_alw;
                    break;
                }
                // Update h_L & h_U
                if (M < M_trgt) {
                   if (h_L < h) h_L = h;
                }
                else if (M_trgt < M) {
                   if (h < h_U) h_U = h;
                }
                const PS::F64 dh = h_U - h_L;
                if (dh == dh_prev) {
                   n_unchanged++;
                }
                else {
                   dh_prev = dh;
                   n_unchanged = 0;
                }
                // Update smoothing length
                h = std::pow((3.0 * M_trgt)/(4.0 * math_const::pi * dens), 1.0/3.0);
```

```
                if ((h <= h_L) || (h == h_U)) {
                   // In this case, we switch to the bisection search.
                   // The inclusion of '=' in the if statement is very
                   // important to escape a limit cycle.
                   h = 0.5 * (h_L + h_U);
                }
                else if (h_U < h) {
                   h = h_U;
                }
            }
            // Calculate grad-h term
            if (force[i].flag == 1) {
                PS::F64 drho_dh = 0.0;
                for (PS::S32 j = 0; j < n_jp; j++) {
                   drho_dh += mj[j] * dWdh(rij[j], h);
                }
                force[i].gradh = 1.0 / (1.0 + (h * drho_dh) / (3.0 * dens)
                      );
            }
            else {
                force[i].gradh = 1.0; // dummy value
            }
#if defined(USE_BALSARA_SWITCH)
            // Compute \div v & \rot v for Balsara switch
            for (PS::S32 j = 0; j < n_jp; j++) {
               const PS::F64vec dr = ep_i[i].pos - ep_j[j].pos;
               const PS::F64vec dv = ep_i[i].vel - ep_j[j].vel;
               force[i].divv -= mj[j] * dv * gradW(dr, force[i].smth);
               force[i].rotv -= mj[j] * dv ^ gradW(dr, force[i].smth);
            }
            force[i].divv /= force[i].dens;
            force[i].rotv /= force[i].dens;
#endif
            // Release memory
            free(mj);
            free(rij);
        }
#else
        for (PS::S32 i = 0; i < n_ip ; i++){
            // Compute density
            for (PS::S32 j = 0; j < n_jp; j++){
               const PS::F64vec dr = ep_j[j].pos - ep_i[i].pos;
               const PS::F64 rij = std::sqrt(dr * dr);
               force[i].dens += ep_j[j].mass * W(rij, ep_i[i].smth);
            }
            force[i].smth = ep_i[i].smth;
            force[i].gradh = 1.0;
#if defined(USE_BALSARA_SWITCH)
            // Compute \div v & \rot v for Balsara switch
            for (PS::S32 j = 0; j < n_jp; j++) {
               const PS::F64vec dr = ep_i[i].pos - ep_j[j].pos;
               const PS::F64vec dv = ep_i[i].vel - ep_j[j].vel;
               force[i].divv -= ep_j[j].mass * dv * gradW(dr, force[i].
                     smth);
               force[i].rotv -= ep_j[j].mass * dv ^ gradW(dr, force[i].
                        smth);
                }
                force[i].divv /= force[i].dens;
                force[i].rotv /= force[i].dens;
#endif
            }
#endif
        }
};
```

7.1.4.3 Interaction function for the calculation of pressure-gradient acceleration

Interaction function for the calculation of pressure-gradient acceleration is implemented as function object `CalcHydroForce` . Listing 74 shows its implementation. This performs the calculations of the right hand sides of Eqs.(10), (12), and (13), and `dt` according to Eq.(16) in Springel [2005, MNRAS, 364, 1105] (for `dt`, see the definition of `FP_sph` class)。

Listing 74: Interaction function for the calculation of pressure-gradient acceleration

```
class CalcHydroForce{
public:
    void operator () (const EP_hydro * ep_i,
                      const PS::S32 n_ip,
                      const EP_hydro * ep_j,
                      const PS::S32 n_jp,
                      Force_hydro * force){
        for (PS::S32 i = 0; i < n_ip; i++){
            const PS::F64vec pos_i = ep_i[i].pos;
            const PS::F64vec vel_i = ep_i[i].vel;
            const PS::F64 smth_i   = ep_i[i].smth;
            const PS::F64 dens_i   = ep_i[i].dens;
            const PS::F64 pres_i   = ep_i[i].pres;
            const PS::F64 f_i      = ep_i[i].gradh;
            const PS::F64 snds_i   = ep_i[i].snds;
            const PS::F64 povrho2_i = pres_i / (dens_i * dens_i);
            PS::F64 v_sig_max = 0.0;
            for (PS::S32 j = 0; j < n_jp; j++){
                const PS::F64vec dr = pos_i - ep_j[j].pos;
                const PS::F64vec dv = vel_i - ep_j[j].vel;
                const PS::F64 w_ij = (dv * dr < 0) ? dv * dr / std::sqrt(dr
                        * dr) : 0;
                const PS::F64 v_sig = snds_i + ep_j[j].snds - 3.0 * w_ij;
                v_sig_max = std::max(v_sig_max, v_sig);
                const PS::F64 AV = - 0.5 * alpha_AV * v_sig * w_ij / (0.5 *
                        (dens_i + ep_j[j].dens))
                                    * 0.5 * (ep_i[i].BalSW + ep_j[j].BalSW);
                const PS::F64vec gradW_i  = gradW(dr, smth_i);
                const PS::F64vec gradW_j  = gradW(dr, ep_j[j].smth);
                const PS::F64vec gradW_ij = 0.5 * (gradW_i + gradW_j);
                const PS::F64 povrho2_j = ep_j[j].pres / (ep_j[j].dens *
                        ep_j[j].dens);
                const PS::F64 f_j = ep_j[j].gradh;
                force[i].acc     -= ep_j[j].mass * (f_i * povrho2_i *
                        gradW_i
```

```
                                                     +f_j * povrho2_j * gradW_j
                                                     +AV * gradW_ij);
                force[i].eng_dot += ep_j[j].mass * (f_i * povrho2_i * gradW_i
                                                     +0.5 * AV * gradW_ij) * dv;
                force[i].ent_dot += 0.5 * ep_j[j].mass * AV * gradW_ij * dv;
            }
            const PS::F64 p = specific_heat_ratio - 1.0;
            force[i].ent_dot *= p/std::pow(dens_i, p);
            force[i].dt = CFL_hydro * 2.0 * ep_i[i].smth / v_sig_max;
        }
    }
};
```

7.1.5 Main body of the sample code

In this section, we describe the main body of the sample code implemented mainly in `main.cpp`. Before entering a detailed explanation, we describe here the overall structure of the code. As described in the beginning of § 7.1, this code performs a N-body/SPH simulation of a disk galaxy. Thus, in the default, the code sets an initial condition for a disk galaxy. But, initial conditions for simple test calculations are also prepared in the code. More specifically, the code supports the following four types of initial conditions:

(a) Initial condition for a disk galaxy simulation. It is selected when `-DINITIAL_CONDITION=0` is specified at the compile-time. The initial condition is created in `void` function `GalaxyIC` in `ic.hpp` . The initial distributions of dark matter and star particles are set by reading a file created by MAGI. The initial distribution of gas (SPH) particles is determined in the subroutine. In the default, an exponential disk ($M = 10^{10}$ $M_\odot$, $R_s = 7$ kpc [scale radius], $R_t = 12.5$ kpc [truncation radius], $z_d = 0.4$ kpc [scale height], $z_t = 1$ kpc [truncation height]) is created with the number of SPH particles of 2^{18}.

(b) Initial condition for cold collapse test. It is selected when `-DINITIAL_CONDITION=1` is specified at the compile-time. The initial condition is created in `void` function `ColdCollapseTestIC` in `ic.hpp` .

(c) Initial condition for the Evrard test (§ 3.3 in Evrard [1988,MNRAS,235,911]). It is selected when `-DINITIAL_CONDITION=2` is specified at the compile-time. This initial condition is created in `void` function `EvrardTestIC` in `ic.hpp` . There are two options for the way of creating an initial condition. We can specify the way by manually set the value of the last argument of the function 0 or 1. If 0 is given, the function creates the density profile of the Evrard gas sphere by rescaling the positions of particles which are placed in a grid. If 1 is specified, it creates the density profile by rescaling the positions of particles which are distributed glass-like. In order to use the second option, we have to create particle data by executing the code with the mode described in the next item.

(d) Operation mode to create a glass-like distribution of SPH particles in a box of $[-1, 1)^3$. This mode is selected when `-DINITIAL_CONDITION=3` is specified at the compile-time. The initial condition is created in `void` function `MakeGlassIC` in `ic.hpp` .

The structure of the sample code is as follows:

(1) Create and initialize FDPS objects
(2) Initialize the Phantom-GRAPE library for x86 if needed
(3) Read a data file of N-body particles and make an initial condition
(4) Calculate the motions of particles until the end time we specify

Below, we explain each item in detail.

7.1.5.1 Include the header file of FDPS

In order to use the features of FDPS, `particle_simulator.hpp` is included in the beginning part of `main.cpp`.

Listing 75: Include the header file of FDPS

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

7.1.5.2 Initialization and and termination of FDPS

We need first to initialize FDPS by calling API `Initialize` :

Listing 76: Initialize FDPS

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

Once started, FDPS should be explicitly terminated by calling API `Finalize` . This sample code terminates FDPS just before the termination of the program. You can find the following code at the last part of `main.cpp`.

Listing 77: Finalize FDPS

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

7.1.5.3 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.

7.1.5.3.1 Creation and initialization of ParticleSystem objects

This sample code uses different ParticleSystem objects to manage N-body and SPH particles. More specifically, the code uses objects of names of `psys_nbody` and `psys_sph` for N-body and SPH particles, respectively. The creation and the initialization of these objects are done as follows.

Listing 78: Creation and initialization of ParticleSystem objects

```
PS::ParticleSystem<FP_nbody> psys_nbody;
PS::ParticleSystem<FP_sph> psys_sph;
psys_nbody.initialize();
psys_sph.initialize();
```

7.1.5.3.2 Creation and initialization of DomainInfo object

This sample code decomposes the computational domain so that the *total* (N-body + SPH) particle distribution is divided equally. In this case, we need one DomainInfo object. Thus, the creation and initialization of DomainInfo object are performed as follows.

Listing 79: Creation and initialization of DomainInfo object

```
PS::DomainInfo dinfo;
dinfo.initialize();
```

7.1.5.3.3 Creation and initialization of TreeForForce objects

The code uses three types of TreeForForce objects and they are used for the gravity calculation, the density calculation, and the calculation of pressure-gradient acceleration. When initializing a TreeForForce object, we must pass a typical number of particles used in the interaction calculation as the first argument of API `initialize` . For TreeForForce object `tree_grav` , the value that is three times of the number of local particles (N-body + SPH) is passed. On the other hand, for TreeForForce objects `tree_dens` and `tree_hydro` , the value that is three times of the number of local SPH particles is passed.

Listing 80: Creation and initialization of TreeForForce objects

```
const PS::S64 numPtclSPH = std::max(psys_sph.getNumberOfParticleGlobal()
      ,1);
const PS::S64 numPtclAll = psys_nbody.getNumberOfParticleGlobal()
                         + numPtclSPH;

const PS::F32 theta_grav = 0.5;
PS::TreeForForceLong<Force_grav, EP_grav, EP_grav>::Monopole tree_grav;
tree_grav.initialize(3 * numPtclAll, theta_grav);

PS::TreeForForceShort<Force_dens, EP_hydro, EP_hydro>::Gather tree_dens;
tree_dens.initialize(3 * numPtclSPH);

PS::TreeForForceShort<Force_hydro, EP_hydro, EP_hydro>::Symmetry
      tree_hydro;
tree_hydro.initialize(3 * numPtclSPH);
```

7.1.5.4 Setting initial condition

The initial condition is set in `void` function `setupIC` , which internally calls a different `void` function depending on the value of macro `INITIAL_CONDITION`. The correspondence relation between the name of a internally-called `void` function and the value of the macro has been described already in the beginning part of § 7.1.5. The arguments `time_dump`, `dt_dump`, `time_end` represents the initial time of data output, the time interval of data output, and the end time of the simulation, respectively. These must be set in this `void` function. Also, the boundary condition, the gravitational softening (`eps_grav`), the maximum allowable time step of the system (`dt_max`) are set in this `void` function (a user does not necessarily set `dt_max`).

Listing 81: Setting initial condition

```
setupIC(psys_nbody, psys_sph, dinfo, time_dump, dt_dump, time_end);
```

In what follows, we describe some of points to remember for `void` function `GalaxyIC` .

- MAGI outputs particle data in its code unit. The information about the MAGI's code unit is described in file `./magi_data/doc/unit.txt` (see section "Computational unit"). This file is created when executing MAGI. The variables `magi_unit_mass`, `magi_unit_leng`, `magi_unit_time` in the `void` function must be consistent with the MAGI's code unit.
- The `void` function reads particle data from file of the name of `./magi_data/dat/Galaxy.tipsy` in the default. If you make the code read a different file, please change the source code manually.
- The `void` function generates an initial gas distribution which has exponential profile along both R ($\equiv \sqrt{x^2 + y^2}$) and z directions. The variables `Rs` and `zd` represents the scale lengths. The variables `Rt` and `zt` represents the truncation (cutoff) lengths.
- The initial thermodynamic state is specified by both the initial gas temperature `temp` and the mean molecular weight relative to the mass of hydrogen atom `mu`. Regardless of the state of the macro `USE_ENTROPY`, a user must specify the thermodynamic state of SPH particles via the specific internal energy (member variable `eng` in `FP_sph` class)[the sample code automatically does this]. If the macro `USE_ENTROPY` is defined, the initial value of the entropy is automatically set by `void` function `setEntropy` called in the `void` function `main()`, using the initial value of the specific internal energy and the calculated density. On the other hand, if the macro is not defined, the value of `eng` set in the `void` function `GalaxyIC` is treated as the initial value of the specific internal energy.

7.1.5.5 Domain decomposition

When there are different types of ParticleSystem objects, the domain decomposition based on the combined distribution of particles can be realized by using APIs `collectSampleParticle` and `decomposeDomain` . First, a user have to collect sample particles from each ParticleSystem object using API `collectSampleParticle` . **Here, we must pass `.false.` to the second argument of this API for the second or later ParticleSystem object because the previous information is cleared without this**. After collecting sample particles from all of ParticleSystem objects, call API `decomposeDomain` to perform domain decomposition.

Listing 82: Domain decomposition

```
dinfo.collectSampleParticle(psys_nbody);
dinfo.collectSampleParticle(psys_sph,false);
dinfo.decomposeDomain();
```

7.1.5.6 Particle exchange

In order to perform particle exchange based on the previous-calculated domain information, it is only necessary to call API `exchangeParticle`.

Listing 83: Particle exchange

```
psys_nbody.exchangeParticle(dinfo);
psys_sph.exchangeParticle(dinfo);
```

7.1.5.7 Interaction calculations

After the domain decomposition and particle exchange, interaction calculations are done. Below, we show the implementation of the interaction calculations just after setting the initial condition. At first, the code performs the gravity calculation. Then, it performs the calculations of density and pressure-gradient acceleration.

Listing 84: Interaction calculations

```
    //- Gravity calculations
#if defined(ENABLE_GRAVITY_INTERACT)
    tree_grav.setParticleLocalTree(psys_nbody);
    tree_grav.setParticleLocalTree(psys_sph,false);
    tree_grav.calcForceMakingTree(CalcGravity<EP_grav>,
                                  CalcGravity<PS::SPJMonopole>,
                                  dinfo);
    for (PS::S32 i = 0; i < psys_nbody.getNumberOfParticleLocal(); i++) {
        psys_nbody[i].copyFromForce(tree_grav.getForce(i));
    }
    const PS::S32 offset = psys_nbody.getNumberOfParticleLocal();
    for (PS::S32 i = 0; i < psys_sph.getNumberOfParticleLocal(); i++) {
        psys_sph[i].copyFromForce(tree_grav.getForce(i+offset));
    }
#endif

    //- SPH calculations
#if defined(ENABLE_HYDRO_INTERACT)
    calcDensity(psys_sph, dinfo, tree_dens);
#if defined(USE_ENTROPY)
    setEntropy(psys_sph);
#endif
    setPressure(psys_sph);
    tree_hydro.calcForceAllAndWriteBack(CalcHydroForce(), psys_sph, dinfo)
         ;
#endif
```

First, we explain the part of the implementation for the gravity calculation. In the gravity calculation, both N-body and SPH particles are involved. In order to perform an interaction calculation between different types of particles, we must use in combination TreeForForce object's APIs `setParticleLocalTree` and `calcForceMakingTree`. We first pass the particle information stored in each ParticleSystem object to a TreeForForce object using API `setParticleLocalTree`. **Here, we must pass `false` to the second argument of this API for the second or later ParticleSystem objects because all**

of the previously-passed information is cleared without this. After finishing calling this API for all of ParticleSystem objects that are involved in the gravity calculation, call API `calcForceMakingTree` to perform the interaction calculation. In order to obtain the result of the interaction calculation, we need to use API `getForce`. This API takes an integral argument i, and it returns the force of the ith particle read by API `setParticleLocalTree`. Hence, we must use appropriate offset to obtain the results of the interaction calculation of the second or later ParticleSystem.

Next, we explain the part of the implementation for the calculations of density and pressure-gradient acceleration. These interaction calculations involves only single type of particles, SPH particles. Therefore, we can use API `calcForceAllAndWriteBack`, which is frequently used in the sample code introduced in this document. For the calculation of pressure-gradient acceleration, the code performs this API in the `void` function `main()`. On the other hand, we need to handle the case that the iteration calculation of ρ_i and h_i does not converge for some particles as described in § 7.1.4. This handling is done in the `void` function `calcDensity`. The implementation of this `void` function is shown below. The implementation actually used differs depending on the state of the macro `ENABLE_VARIABLE_SMOOTHING_LENGTH`. If it is not defined, the code calls API `calcForceAllAndWriteBack` only once because in this case the code performs SPH calculation as the fixed smoothing length SPH code. If the macro is defined, the code calls the API repeatedly until ρ_i and h_i of all the particles are self-consistently determined. The member variable `flag` stores the result of the iteration calculation and the value of 1 means that the iteration converges successfully. So, the code stops the infinite `for` loop when the number of SPH particles whose `flag` has the value of 1 agrees with the total number of SPH particles.

Listing 85: Function `calcDensity`

```
void calcDensity(PS::ParticleSystem<FP_sph> & psys,
                 PS::DomainInfo & dinfo,
                 PS::TreeForForceShort<Force_dens, EP_hydro, EP_hydro>::
                        Gather & tree) {
#if defined(ENABLE_VARIABLE_SMOOTHING_LENGTH)
    const PS::S32 n_loc = psys.getNumberOfParticleLocal();
    const PS::S64 n_glb = psys.getNumberOfParticleGlobal();
    // Determine the density and the smoothing length so that Eq.(6) in
          Springel (2005)
    // holds within a specified accuracy.
    SCF_smth = 1.25;
    PS::S32 iter = 0;
    for (;;) {
        iter++;
        if (PS::Comm::getRank() == 0) std::cout << "iter␣=␣" << iter <<
              std::endl;
        // Compute density, etc.
        tree.calcForceAllAndWriteBack(CalcDensity(), psys, dinfo);
        // Check convergence
        PS::S32 n_compl_loc = 0;
        for (PS::S32 i = 0; i < n_loc; i++) {
            if (psys[i].flag == 1) n_compl_loc++;
        }
        const PS::S64 n_compl = PS::Comm::getSum(n_compl_loc);
        if (n_compl == n_glb) break;
    }
    // Reset SCF_smth
    SCF_smth = 1.0;
#else
    SCF_smth = 1.0;
    tree.calcForceAllAndWriteBack(CalcDensity(), psys, dinfo);
#endif
}
```

`void` function `setEntropy` is called only once just after setting an initial condition. As described earlier, this `void` function is used to set the initial value of the entropy. Because we need the initial density to set the initial value of the entropy using Eq. (8), this `void` function is placed just after `void` function `calcDensity`. After this, the entropy becomes the independent variable to describe the thermodynamic state of gas if the macro `USE_ENTROPY` is defined.

7.1.5.8 Time integration

This code performs the time integration using the Leapfrog method (see § 4.1.3.4.4 for this method). In this code, $D(\cdot)$ operator is implemented as the `void` function `full_drift`, while $K(\cdot)$ operator is implemented as `void` functions `initial_kick` and `final_kick`.

8 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.

8.1 Compile-time problem

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

8.2 Run-time problem

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

8.3 Other cases

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

9 License

This software is MIT licensed. Please cite Iwasawa et al. (2016, Publications of the Astronomical Society of Japan, 68, 54) and Namekata et al. (2018, Publications of the Astronomical Society of Japan, 70, 70) 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".