FDPS Tutorial

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

Contents

1 TODO

2 Change log

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

3 Overview

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

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

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

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

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

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

4 Getting Started

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

4.1 Environment

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

4.2 Necessary software

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

4.2.1 Standard functions

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

4.2.1.1 Single thread

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

4.2.1.2 Parallel processing

4.2.1.2.1 OpenMP

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

4.2.1.2.2 MPI

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

4.2.1.2.3 MPI+OpenMP

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

4.2.2 Extensions

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

4.2.2.1 Particle Mesh

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

4.3 Install

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

4.3.1 How to get the software

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

4.3.1.1 The latest version

You can use one of the following ways.

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

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

  - Using Git

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

4.3.1.2 Previous versions

You can get previous versions using browsers.

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

4.3.2 How to build

There is no need for configure or setup.

4.4 How to compile and run the sample codes

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

4.4.1 gravitational N-body simulation

4.4.1.1 Summary

Through the following steps one can use this sample.

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

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

4.4.1.2 Move to the directory with the sample code

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

4.4.1.3 Edit Makefile

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

- Without OpenMP or MPI
  - Set the variable "CC" the command to run your C++ compiler
- With OpenMP but not with MPI
  - Set the variable "CC" the command to run your C++ compiler
  - uncomment the line "CFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp". If you use Intel compiler, remove "-fopenmp"
- With MPI but not with OpenMP
  - Set the variable "CC" the command to run your MPI C++ compiler
  - uncomment the line "CFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL"
- With both OpenMP and MPI
  - Set the variable "CC" the command to run your MPI C++ compiler
  - uncomment the line "CFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp". If you use Intel compiler, remove "-fopenmp"
  - uncomment the line "CFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL"

4.4.1.4 run make

Type "make" to run make.

4.4.1.5 run the sample code

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

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

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

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

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

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

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

4.4.1.6 Analysis of the result

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

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

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

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

To increase the number of particles to 10,000, try: (without MPI)

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

4.4.1.7 To use Phantom-GRAPE for x86

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

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

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

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

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

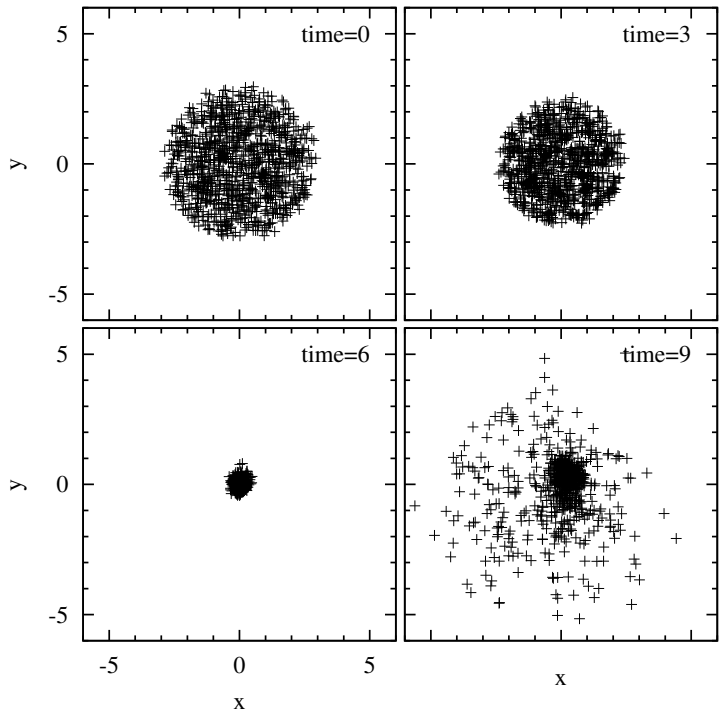

Figure 1:

4.4.1.8 To use NVIDIA GPUs

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

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

4.4.2 SPH simulation code

4.4.2.1 Summary

Through the following steps one can use this sample.

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

4.4.2.2 Move to the directory with the sample code

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

4.4.2.3 Edit Makefile

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

- Without OpenMP or MPI
  - Set the variable "CC" the command to run your C++ compiler
- With OpenMP but not with MPI
  - Set the variable "CC" the command to run your C++ compiler
  - uncomment the line "CFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp". If you use Intel compiler, remove "-fopenmp"
- With MPI but not with OpenMP
  - Set the variable "CC" the command to run your MPI C++ compiler
  - uncomment the line "CFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL"
- With both OpenMP and MPI
  - Set the variable "CC" the command to run your MPI C++ compiler
  - uncomment the line "CFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp". If you use Intel compiler, remove "-fopenmp"
  - uncomment the line "CFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL"

4.4.2.4 run make

Type "make" to run make.

4.4.2.5 run the sample code

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

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

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

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

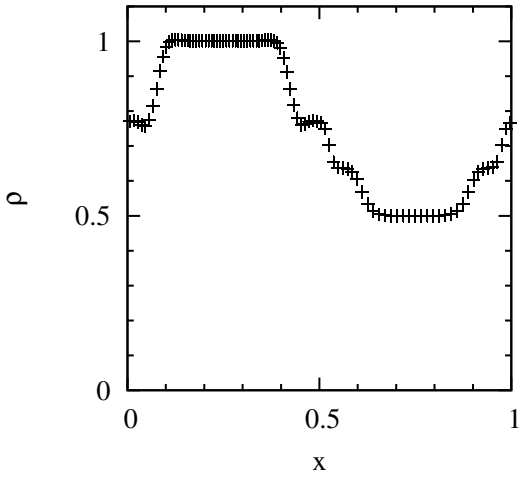

Figure 2:

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. ********
```

4.4.2.6 Analysis of the result

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

What is simulated is the three-dimensional shock-tube problem.

Using gnuplot, you can see the plot of the x-coordinate and density of particles at time=40:

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

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

5 How to Use

5.1 Compilation and execution of sample codes

5.2 Backgrounds

5.2.1 Vector Type

5.3 SPH simulation with fixed smoothing length

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

5.3.1 Working directory

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

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

5.3.2 Specifying include files

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

Listing 1: Include FDPS

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

5.3.3 User-defined classes

5.3.3.1 Overview

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

5.3.3.2 FullParticle type

You need to define the FullParticle type. FullParticle type should contain all physical quantities necessary for an SPH particles. It also should have member functions used to copy results from the Force type (discussed later). It should have member functions getCharge() (returns the particle mass), getPos() (returns the particle position), getRSearch() (returns the search radius for neighbours), and setPos() (set the position). In this tutorial, we also define member functions necessary to use file I/O functions of FDPS, writeAscii() and readAscii().

In addition to them, member function setPressure() is defined. This member function calculate the pressure from the equation of states. This function is not used by FDPS, but used within the user code.

The following is the code to define FullParticle type used here.

Listing 2: 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, "%ld\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, "%ld\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);
41          }
42          void setPressure(){
43                  const PS::F64 hcr = 1.4;
44                  pres = (hcr - 1.0) * dens * eng;
45                  snds = sqrt(hcr * pres / dens);
46          }
47  };
```

5.3.3.3 EssentialParticleI type

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

The following is the code to define EssentialParticleI type used here.

Listing 3: EssentialParticleI type

```
 1  struct EP{
 2          PS::F64vec pos;
 3          PS::F64vec vel;
 4          PS::F64    mass;
 5          PS::F64    smth;
 6          PS::F64    dens;
 7          PS::F64    pres;
 8          PS::F64    snds;
 9          void copyFromFP(const FP& rp){
10                  this->pos  = rp.pos;
11                  this->vel  = rp.vel;
12                  this->mass = rp.mass;
13                  this->smth = rp.smth;
14                  this->dens = rp.dens;
15                  this->pres = rp.pres;
16                  this->snds = rp.snds;
17          }
18          PS::F64vec getPos() const{
19                  return this->pos;
20          }
21          PS::F64 getRSearch() const{
22                  return kernelSupportRadius * this->smth;
```

```
23          }
24          void setPos(const PS::F64vec& pos){
25                  this->pos = pos;
26          }
27  };
```

5.3.3.4 Force type

You should define Force type. It should contain all information generated as the result of the calculation of Force. In this tutorial, there are two types of Force calculations, one for density and the other for actual hydrodynamic interaction. Thus, two Force types should be defined. A Force type should have member function clear(), which zero-clear member variables.

The following is the code to define Force types used here.

Listing 4: Force type

```
1  class Dens{
2          public:
3          PS::F64 dens;
4          PS::F64 smth;
5          void clear(){
6                  dens = 0;
7          }
8  };
9
10 class Hydro{
11         public:
12         PS::F64vec acc;
13         PS::F64 eng_dot;
14         PS::F64 dt;
15         void clear(){
16                 acc = 0;
17                 eng_dot = 0;
18         }
19 };
```

5.3.3.5 calcForceEpEp type

You should define calcForceEpEp type. It should contain actual code for the calculation of Force. It is implemented using Functor. The arguments of the Functor are an array of EssentialParticleI type, the number of EssentialParticleI type variables, an array of EssentialParticleJtype, the number of EssentialParticleJ variables, an array of Force type. Two Force classes, one for density and the other for actual hydrodynamic interaction, are used in this code. Thus, two calcForceEpEp types should be defined.

The following is the code to define calcForceEpEp types used here.

Listing 5: calcForceEpEp type

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

```
                                        ].smth));
28                                  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;
29                                  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;
30                              }
31                              hydro[i].dt = C_CFL * 2.0 * ep_i[i].
                                    smth / v_sig_max;
32                  }
33          }
34 };
```

5.3.4 The main body of the user program

5.3.4.1 Overview

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

5.3.4.2 Initialization and termination of FDPS

You should first initialize FDPS by the following code.

Listing 6: Initialization of FDPS

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

Once started, FDPS should be explicitly terminated. In this example, 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 7: Termination of FDPS

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

5.3.4.3 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.

5.3.4.3.1 Creation of necessary FDPS objects

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

Listing 8: Creation of FDPS Objects

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

5.3.4.3.2 Initialization of the DomainInfo object

FDPS objects created by a user code should be initialized. Here we describe the necessary initialization for a DomainInfo object. After the initialization, the type of the boundary and size of the simulation box should be set. In this code, we use the periodic boundary for all of x, y and z directions.

Listing 9: Initialization of DomainInfo

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

5.3.4.3.3 Initialization of the ParticleSystem object

The initialization of a ParticleSystem object can be done in the following single line of code

Listing 10: Initialization of PaticleSystem

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

5.3.4.3.4 Initialization of the TreeForForceShort objects

Finally, TreeForForceShort objects should be initialized. The initialization function for TreeForForceShort objects should be given the rough number of particles. In this example, we three times the total number of particles

Listing 11: Initialization of TreeForForceShort

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

5.3.4.4 Time integration loop

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

5.3.4.4.1 Domain Decomposition

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

Listing 12: Domain Decomposition

```
1 dinfo.decomposeDomain();
```

5.3.4.4.2 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 member function of the class ParticleSystem is called.

Listing 13: Particle Exchange

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

5.3.4.4.3 Interaction Calculation

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

Listing 14: Interaction Calculation

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

5.3.5 Compilation of the program

run `make` at the working directory. You can use the Makefile attached to the tutorial.

```
$ make
```

5.3.6 Execution

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

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

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

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

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

5.3.7 Log and output files

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

5.3.8 Visualization

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

```
$ gnuplot
```

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

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

5.4 (Gravitational N-body simulation code

5.4.1 Working directory

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

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

5.4.2 User-defined classes

5.4.2.1 Overview

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

5.4.2.2 FullParticle type

You need to define the FullParticle type. FullParticle type should contain all physical quantities necessary for an N-body In addition, in our tutorial code, FullParticle type is used also as Essential Particle I and Essential Particle J types. It has necessary member functions to copy data, copyfromFP and copyFromForce. It should have member functions getCharge() (returns the particle mass), getPos() (returns the particle position), and setPos() (set the position). In this tutorial, we also define member functions necessary to use file I/O functions of FDPS, writeAscii() and readAscii(). Also the function clear, which zero-clear the

acceleration and potential, is necessary. The member functions for the leap-frog integrator, predict and correct, are defined. These are used within the user program, not within FDPS.

Listing 15: FullParticle type

```
1  class FPGrav{
2  public:
3      PS::F64    mass;
4      PS::F64vec pos;
5      PS::F64vec vel;
6      PS::F64vec acc;
7      PS::F64    pot;
8      PS::F64vec vel2;
9
10     static PS::F64 eps;
11
12     PS::F64vec getPos() const {
13         return pos;
14     }
15
16     PS::F64 getCharge() const {
17         return mass;
18     }
19
20     void copyFromFP(const FPGrav & fp){
21         mass = fp.mass;
22         pos  = fp.pos;
23     }
24
25     void copyFromForce(const FPGrav & force) {
26         acc = force.acc;
27         pot = force.pot;
28     }
29
30     void clear() {
31         acc = 0.0;
32         pot = 0.0;
33     }
34
35     void predict(PS::F32 dt) {
36         pos  = pos  +       vel * dt + 0.5 * acc * dt * dt;
37         vel2 = vel  + 0.5 * acc * dt;
38     }
39
40     void correct(PS::F32 dt) {
41         vel  = vel2 + 0.5 * acc * dt;
42     }
```

```
43 };
44
45 PS::F64 FPGrav::eps = 1.0 / 32.0;
```

5.4.2.3 calcForceEpEp

You should define calcForceEpEp type. It should contain actual code for the calculation of Force. It is implemented using Functor. The arguments of the Functor are an array of EssentialParticleI type, the number of EssentialParticleI type variables, an array of EssentialParticleJtype, the number of EssentialParticleJ variables, an array of Force type. The following is the code to define calcForceEpEp types used here.

Listing 16: calcForceEpEp type

```
1  template <class TParticleJ>
2  struct CalcGravity{
3      void operator () (const FPGrav * iptcl,
4                        const PS::S32 ni,
5                        const TParticleJ * jptcl,
6                        const PS::S32 nj,
7                        FPGrav * force) {
8
9          PS::F64 eps2 = FPGrav::eps * FPGrav::eps;
10
11         for(PS::S32 i = 0; i < ni; i++){
12
13             PS::F64vec posi = iptcl[i].pos;
14             PS::F64vec acci = 0.0;
15             PS::F64    poti = 0.0;
16
17             for(PS::S32 j = 0; j < nj; j++){
18                 PS::F64vec posj   = jptcl[j].pos;
19                 PS::F64    massj  = jptcl[j].mass;
20
21                 PS::F64vec drvec  = posi - posj;
22                 PS::F64    dr2    = drvec * drvec + eps2;
23                 PS::F64    drinv  = 1.0 / sqrt(dr2);
24                 PS::F64    mdrinv = drinv * massj;
25
26                 poti -= mdrinv;
27                 acci -= mdrinv * drinv * drinv * drvec;
28             }
29
30             force[i].acc += acci;
31             force[i].pot += poti;
32         }
33     }
```

```
34 };
```

5.4.3 The main body of the user program

5.4.3.1 Overview

In this section, we describe the functions a user should write to implement gravitational N-body calculation using FDPS.

5.4.3.2 Initialization and termination of FDPS

You should first initialize FDPS by the following code.

Listing 17: Initialization of FDPS

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

Once started, FDPS should be explicitly terminated. In this example, 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 18: Termination of FDPS

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

5.4.3.3 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.

5.4.3.3.1 Creation of necessary FDPS objects

In an SPH simulation code, one needs to create objects of ParticleSystem type, DomainInfo type, and TreeForForceLong type. The following is the code to create them.

Listing 19: Creation of FDPS Objects

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

5.4.3.3.2 Initialization of the DomainInfo object

FDPS objects created by a user code should be initialized. Here we describe the necessary initialization for a DomainInfo object. Since the open boundary is used in this example, the initialization is done by the following single call without arguments.

Listing 20: Initialization of DomainInfo

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

5.4.3.3.3 Initialization of the ParticleSystem object

The intialization of a ParticleSystem object can be done in the following single line of code

Listing 21: Initialization of PaticleSystem

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

5.4.3.3.4 Initialization of the TreeForForceShort objects

Finally, TreeForForceLong objects should be initialized. The initilization function for a TreeForForceLong object should be given the rough number of particles. In this example, we three times the total number of particles

Listing 22: Initialization of TreeForForceLong

```
1 tree_grav.initialize(ntot);
```

5.4.3.4 Time integration loop

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

5.4.3.4.1 Domain Decomposition

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

Listing 23: Domain Decomposition

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

5.4.3.4.2 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 member function of the class ParticleSystem is called.

Listing 24: Particle Exchange

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

5.4.3.4.3 Interaction Calculation

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

Listing 25: Interaction Calculation

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

5.4.3.4.4 Time Integration

5.4.3.4.5 predict

At the beginning of the timestep, positions and velocities of particles are updated using `predict`

Listing 26: predict

```
1 predict(system_grav, dtime);
```

5.4.3.4.6 correct

After the force calculation, velocities of particles are corrected.

Listing 27: correct

```
1 correct(system_grav, dtime);
```

5.4.4 Diagnostic output

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

Listing 28: Standard error output

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

6 Sample Codes

6.1 SPH simulation with fixed smoothing length

In this section, we show a sample code for the SPH simulation with fixed smoothing length. This code is the same as what we described in section 5. One can create a working code by cut and paste this code and compile and link the resulted source program.

Listing 29: Sample code of SPH simulation

```
1  #define SANITY_CHECK_REALLOCATABLE_ARRAY
2  //include the FDPS header
3  #include <particle_simulator.hpp>
4  //include the standard headers  used in the code
5  #include <cmath>
6  #include <cstdio>
7  #include <cstdlib>
8  #include <iostream>
9  #include <vector>
10
11 /*
12 Parameter
13 */
14 const short int Dim = 3;
15 const PS::F64 SMTH = 1.2;
16 const PS::U32 OUTPUT_INTERVAL = 10;
17 const PS::F64 C_CFL = 0.3;
18
19 /*
20 SPH  Kernel
21 */
22 const PS::F64 pi = atan(1.0) * 4.0;
23 const PS::F64 kernelSupportRadius = 2.5;
24
25 PS::F64 W(const PS::F64vec dr, const PS::F64 h){
26         const PS::F64 H = kernelSupportRadius * h;
27         const PS::F64 s = sqrt(dr * dr) / H;
28         const PS::F64 s1 = (1.0 - s < 0) ? 0 : 1.0 - s;
29         const PS::F64 s2 = (0.5 - s < 0) ? 0 : 0.5 - s;
30         PS::F64 r_value = pow(s1, 3) - 4.0 * pow(s2, 3);
31         //if # of dimension == 3
32         r_value *= 16.0 / pi / (H * H * H);
33         return r_value;
34 }
35
36 PS::F64vec gradW(const PS::F64vec dr, const PS::F64 h){
37         const PS::F64 H = kernelSupportRadius * h;
38         const PS::F64 s = sqrt(dr * dr) / H;
```

```
39          const PS::F64 s1 = (1.0 - s < 0) ? 0 : 1.0 - s;
40          const PS::F64 s2 = (0.5 - s < 0) ? 0 : 0.5 - s;
41          PS::F64 r_value = - 3.0 * pow(s1, 2) + 12.0 * pow(s2,
                2);
42          //if # of dimension == 3
43          r_value *= 16.0 / pi / (H * H * H);
44          return dr * r_value / (sqrt(dr * dr) * H + 1.0e-6 * h);
45  }
46
47  /*
48  classes
49  */
50  class Dens{
51          public:
52          PS::F64 dens;
53          PS::F64 smth;
54          void clear(){
55                  dens = 0;
56          }
57  };
58  class Hydro{
59          public:
60          PS::F64vec acc;
61          PS::F64 eng_dot;
62          PS::F64 dt;
63          void clear(){
64                  acc = 0;
65                  eng_dot = 0;
66          }
67  };
68
69  class RealPtcl{
70          public:
71          PS::F64 mass;
72          PS::F64vec pos;//POSition
73          PS::F64vec vel;//VELocity
74          PS::F64vec acc;//ACCeleration
75          PS::F64 dens;  //DENSity
76          PS::F64 eng;   //ENerGy
77          PS::F64 pres;  //PRESsure
78          PS::F64 smth;  //SMooTHing length
79          PS::F64 snds;  //SouND Speed
80          PS::F64 eng_dot;
81          PS::F64 dt;
82          PS::S64 id;
83          //half step
```

```
84          PS::F64vec vel_half;
85          PS::F64 eng_half;
86          //Copy functions
87          void copyFromForce(const Dens& dens){
88                  this->dens = dens.dens;
89                  this->smth = dens.smth;
90          }
91          void copyFromForce(const Hydro& force){
92                  this->acc     = force.acc;
93                  this->eng_dot = force.eng_dot;
94                  this->dt      = force.dt;
95          }
96          //Give necessary values to FDPS
97          PS::F64 getCharge() const{
98                  return this->mass;
99          }
100         PS::F64vec getPos() const{
101                 return this->pos;
102         }
103         PS::F64 getRSearch() const{
104                 return kernelSupportRadius * this->smth;
105         }
106         void setPos(const PS::F64vec& pos){
107                 this->pos = pos;
108         }
109         void writeAscii(FILE* fp) const{
110                 fprintf(fp, "%ld\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);
111         }
112         void readAscii(FILE* fp){
113                 fscanf(fp, "%ld\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);
114         }
115         void setPressure(){
116                 const PS::F64 hcr = 1.4;//heat capacity ratio
117                 pres = (hcr - 1.0) * dens * eng;
118                 snds = sqrt(hcr * pres / dens);
119         }
```

```
120 };
121
122 class EP{
123 public:
124         PS::F64vec pos;
125         PS::F64vec vel;
126         PS::F64    mass;
127         PS::F64    smth;
128         PS::F64    dens;
129         PS::F64    pres;
130         PS::F64    snds;
131         void copyFromFP(const RealPtcl& rp){
132                 this->pos  = rp.pos;
133                 this->vel  = rp.vel;
134                 this->mass = rp.mass;
135                 this->smth = rp.smth;
136                 this->dens = rp.dens;
137                 this->pres = rp.pres;
138                 this->snds = rp.snds;
139         }
140         PS::F64vec getPos() const{
141                 return this->pos;
142         }
143         PS::F64 getRSearch() const{
144                 return kernelSupportRadius * this->smth;
145         }
146         void setPos(const PS::F64vec& pos){
147                 this->pos = pos;
148         }
149 };
150
151 class FileHeader{
152         public:
153         int Nbody;
154         double time;
155         int readAscii(FILE* fp){
156                 fscanf(fp, "%e\n", &time);
157                 fscanf(fp, "%d\n", &Nbody);
158                 return Nbody;
159         }
160         void writeAscii(FILE* fp) const{
161                 fprintf(fp, "%e\n", time);
162                 fprintf(fp, "%d\n", Nbody);
163         }
164 };
165
```

```
166 struct boundary{
167         PS::F64 x, y, z;
168 };
169
170 /*
171 Force functor
172 */
173
174 class CalcDensity{
175         public:
176         void operator () (const EP* const ep_i, const PS::S32
              Nip, const EP* const ep_j, const PS::S32 Njp,
              Dens* const dens){
177                 for(PS::S32 i = 0 ; i < Nip ; ++ i){
178                         dens[i].clear();
179                         for(PS::S32 j = 0 ; j < Njp ; ++ j){
180                                 const PS::F64vec dr = ep_j[j].
                                        pos - ep_i[i].pos;
181                                 dens[i].dens += ep_j[j].mass *
                                        W(dr, ep_i[i].smth);
182                         }
183                         dens[i].smth = SMTH * pow(ep_i[i].mass
                                / dens[i].dens, 1.0/(PS::F64)(Dim
                                ));
184                 }
185         }
186 };
187
188 class CalcHydroForce{
189         public:
190         void operator () (const EP* const ep_i, const PS::S32
              Nip, const EP* const ep_j, const PS::S32 Njp,
              Hydro* const hydro){
191                 for(PS::S32 i = 0; i < Nip ; ++ i){
192                         hydro[i].clear();
193                         PS::F64 v_sig_max = 0.0;
194                         for(PS::S32 j = 0; j < Njp ; ++ j){
195                                 const PS::F64vec dr = ep_i[i].
                                        pos - ep_j[j].pos;
196                                 const PS::F64vec dv = ep_i[i].
                                        vel - ep_j[j].vel;
197                                 const PS::F64 w_ij = (dv * dr <
                                         0) ? dv * dr / sqrt(dr *
                                         dr) : 0;
198                                 const PS::F64 v_sig = ep_i[i].
                                        snds + ep_j[j].snds - 3.0
```

```
                                        * w_ij;
199                                 v_sig_max = std::max(v_sig_max,
                                        v_sig);
200                                 const PS::F64 AV = - 0.5 *
                                        v_sig * w_ij / (0.5 * (
                                        ep_i[i].dens + ep_j[j].
                                        dens));
201                                 const PS::F64vec gradW_ij = 0.5
                                        * (gradW(dr, ep_i[i].
                                        smth) + gradW(dr, ep_j[j
                                        ].smth));
202                                 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;
203                                 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;
204                         }
205                         hydro[i].dt = C_CFL * 2.0 * ep_i[i].
                                smth / v_sig_max;
206                 }
207         }
208 };
209
210 void SetupIC(PS::ParticleSystem<RealPtcl>& sph_system, PS::F64
        *end_time, boundary *box){
211         /////////
212         //place ptcls
213         /////////
214         std::vector<RealPtcl> ptcl;
215         const PS::F64 dx = 1.0 / 128.0;
216         box->x = 1.0;
217         box->y = box->z = box->x / 8.0;
218         PS::S32 i = 0;
219         for(PS::F64 x = 0 ; x < box->x * 0.5 ; x += dx){
220                 for(PS::F64 y = 0 ; y < box->y ; y += dx){
221                         for(PS::F64 z = 0 ; z < box->z ; z +=
                                dx){
222                                 RealPtcl ith;
223                                 ith.pos.x = x;
224                                 ith.pos.y = y;
```

```
225                                 ith.pos.z = z;
226                                 ith.dens = 1.0;
227                                 ith.mass = 0.75;
228                                 ith.eng  = 2.5;
229                                 ith.id   = i++;
230                                 ptcl.push_back(ith);
231                         }
232                 }
233         }
234         for(PS::F64 x = box->x * 0.5 ; x < box->x * 1.0 ; x +=
                dx * 2.0){
235                 for(PS::F64 y = 0 ; y < box->y ; y += dx){
236                         for(PS::F64 z = 0 ; z < box->z ; z +=
                                dx){
237                                 RealPtcl ith;
238                                 ith.pos.x = x;
239                                 ith.pos.y = y;
240                                 ith.pos.z = z;
241                                 ith.dens = 0.5;
242                                 ith.mass = 0.75;
243                                 ith.eng  = 2.5;
244                                 ith.id   = i++;
245                                 ptcl.push_back(ith);
246                         }
247                 }
248         }
249         for(PS::U32 i = 0 ; i < ptcl.size() ; ++ i){
250                 ptcl[i].mass = ptcl[i].mass * box->x * box->y *
                        box->z / (PS::F64)(ptcl.size());
251         }
252         std::cout << "#␣of␣ptcls␣is...␣" << ptcl.size() << std
                ::endl;
253         /////////
254         //scatter ptcls^^e2^^86^^b2
255         /////////^^e2^^86^^b2
256         assert(ptcl.size() % PS::Comm::getNumberOfProc() == 0);
257         const PS::S32 numPtclLocal = ptcl.size() / PS::Comm::
                getNumberOfProc();
258         sph_system.setNumberOfParticleLocal(numPtclLocal);
259         const PS::U32 i_head = numPtclLocal * PS::Comm::getRank
                ();
260         const PS::U32 i_tail = numPtclLocal * (PS::Comm::
                getRank() + 1);
261         for(PS::U32 i = 0 ; i < ptcl.size() ; ++ i){
262                 if(i_head <= i && i < i_tail){
263                         const PS::U32 ii = i - numPtclLocal *
```

```
                                PS::Comm::getRank();
264                         sph_system[ii] = ptcl[i];
265                 }
266         }
267         /////////
268         *end_time = 0.11;
269         //Fin.
270         std::cout << "setup..." << std::endl;
271 }
272
273 void Initialize(PS::ParticleSystem<RealPtcl>& sph_system){
274         for(PS::S32 i = 0 ; i < sph_system.
                getNumberOfParticleLocal() ; ++ i){
275                 sph_system[i].smth = SMTH * pow(sph_system[i].
                        mass / sph_system[i].dens, 1.0/(PS::F64)(
                        Dim));
276                 sph_system[i].setPressure();
277         }
278 }
279
280 PS::F64 getTimeStepGlobal(const PS::ParticleSystem<RealPtcl>&
        sph_system){
281         PS::F64 dt = 1.0e+30;//set VERY LARGE VALUE
282         for(PS::S32 i = 0 ; i < sph_system.
                getNumberOfParticleLocal() ; ++ i){
283                 dt = std::min(dt, sph_system[i].dt);
284         }
285         return PS::Comm::getMinValue(dt);
286 }
287
288 void InitialKick(PS::ParticleSystem<RealPtcl>& sph_system,
        const PS::F64 dt){
289         for(PS::S32 i = 0 ; i < sph_system.
                getNumberOfParticleLocal() ; ++ i){
290                 sph_system[i].vel_half = sph_system[i].vel +
                        0.5 * dt * sph_system[i].acc;
291                 sph_system[i].eng_half = sph_system[i].eng +
                        0.5 * dt * sph_system[i].eng_dot;
292         }
293 }
294
295 void FullDrift(PS::ParticleSystem<RealPtcl>& sph_system, const
        PS::F64 dt){
296         //time becomes t + dt;
297         for(PS::S32 i = 0 ; i < sph_system.
                getNumberOfParticleLocal() ; ++ i){
```

```
298                 sph_system[i].pos += dt * sph_system[i].
                        vel_half;
299         }
300 }
301
302 void Predict(PS::ParticleSystem<RealPtcl>& sph_system, const PS
        ::F64 dt){
303         for(PS::S32 i = 0 ; i < sph_system.
                getNumberOfParticleLocal() ; ++ i){
304                 sph_system[i].vel += dt * sph_system[i].acc;
305                 sph_system[i].eng += dt * sph_system[i].eng_dot
                        ;
306         }
307 }
308
309 void FinalKick(PS::ParticleSystem<RealPtcl>& sph_system, const
        PS::F64 dt){
310         for(PS::S32 i = 0 ; i < sph_system.
                getNumberOfParticleLocal() ; ++ i){
311                 sph_system[i].vel = sph_system[i].vel_half +
                        0.5 * dt * sph_system[i].acc;
312                 sph_system[i].eng = sph_system[i].eng_half +
                        0.5 * dt * sph_system[i].eng_dot;
313         }
314 }
315
316 void setPressure(PS::ParticleSystem<RealPtcl>& sph_system){
317         for(PS::S32 i = 0 ; i < sph_system.
                getNumberOfParticleLocal() ; ++ i){
318                 sph_system[i].setPressure();
319         }
320 }
321
322 int main(int argc, char* argv[]){
323         //Initialize FDPS
324         PS::Initialize(argc, argv);
325         //send particle data to FDPS and initialize
326         PS::ParticleSystem<RealPtcl> sph_system;
327         sph_system.initialize();
328         //variable definition
329         PS::F64 dt, end_time;
330         boundary box;
331         //setup the initial condition
332         SetupIC(sph_system, &end_time, &box);
333         Initialize(sph_system);
334         //setup the domain info and initialize
```

```
335         PS::DomainInfo dinfo;
336         dinfo.initialize();
337         //set boundary type and size to domain info
338         dinfo.setBoundaryCondition(PS::
                BOUNDARY_CONDITION_PERIODIC_XYZ);
339         dinfo.setPosRootDomain(PS::F64vec(0.0, 0.0, 0.0), PS::
                F64vec(box.x, box.y, box.z));
340         //domain decomposition
341         dinfo.decomposeDomain();
342         //particle exchange
343         sph_system.exchangeParticle(dinfo);
344         //create and initialize density tree and interaction
                tree
345         PS::TreeForForceShort<Dens, EP, EP>::Gather dens_tree;
346         dens_tree.initialize(3 * sph_system.
                getNumberOfParticleGlobal());
347
348         PS::TreeForForceShort<Hydro, EP, EP>::Symmetry
                hydr_tree;
349         hydr_tree.initialize(3 * sph_system.
                getNumberOfParticleGlobal());
350         //calculation of density, pressuerm acceleration
351         dens_tree.calcForceAllAndWriteBack(CalcDensity(),
                sph_system, dinfo);
352         setPressure(sph_system);
353         hydr_tree.calcForceAllAndWriteBack(CalcHydroForce(),
                sph_system, dinfo);
354         //get  timestep
355         dt = getTimeStepGlobal(sph_system);
356         //start time integration loop
357         PS::S32 step = 0;
358         for(PS::F64 time = 0 ; time < end_time ; time += dt, ++
                step){
359           //Leap frog: Initial Kick & Full Drift
360           InitialKick(sph_system, dt);
361           FullDrift(sph_system, dt);
362           //correct positions of particle outside the box
363           sph_system.adjustPositionIntoRootDomain(dinfo);
364           //Leap frog: Predict
365           Predict(sph_system, dt);
366           //update domain decomposition
367           dinfo.decomposeDomain();
368           //exchange particles
369           sph_system.exchangeParticle(dinfo);
370           //calculation of density, pressuerm
                  acceleration
```

```
371                 dens_tree.calcForceAllAndWriteBack(CalcDensity
                        (), sph_system, dinfo);
372                 setPressure(sph_system);
373                 hydr_tree.calcForceAllAndWriteBack(
                        CalcHydroForce(), sph_system, dinfo);
374                 //get  timestep
375                 dt = getTimeStepGlobal(sph_system);
376                 //Leap frog: Final Kick
377                 FinalKick(sph_system, dt);
378                 //Output result files
379                 if(step % OUTPUT_INTERVAL == 0){
380                         FileHeader header;
381                         header.time = time;
382                         header.Nbody = sph_system.
                                getNumberOfParticleGlobal();
383                         char filename[256];
384                         sprintf(filename, "result/%04d.txt",
                                step);
385                         sph_system.writeParticleAscii(filename,
                                header);
386                         if(PS::Comm::getRank() == 0){
387                                 std::cout << "
                                        //================================
                                        " << std::endl;
388                                 std::cout << "output␣" <<
                                        filename << "." << std::
                                        endl;
389                                 std::cout << "
                                        //================================
                                        " << std::endl;
390                         }
391                 }
392                 //write diags to stdout
393                 if(PS::Comm::getRank() == 0){
394                         std::cout << "
                                //================================
                                " << std::endl;
395                         std::cout << "time␣=␣" << time << std::
                                endl;
396                         std::cout << "step␣=␣" << step << std::
                                endl;
397                         std::cout << "
                                //================================
                                " << std::endl;
398                 }
399         }
```

```
400         //terminate FDPS
401         PS::Finalize();
402         return 0;
403 }
```

6.2 N-body simulation

In this section, we show a sanple code for the N-body simulation. This code is the same as what we described in section 5. One can create a working code by cut and paste this code and compile and link the resulted source program.

Listing 30: Sample code of N-body simulation

```
1  #include <particle_simulator.hpp>
2
3  class FPGrav{
4  public:
5      PS::F64    mass;
6      PS::F64vec pos;
7      PS::F64vec vel;
8      PS::F64vec acc;
9      PS::F64    pot;
10     PS::F64vec vel2;
11
12     static PS::F64 eps;
13
14     PS::F64vec getPos() const {
15         return pos;
16     }
17
18     PS::F64 getCharge() const {
19         return mass;
20     }
21
22     void copyFromFP(const FPGrav & fp){
23         mass = fp.mass;
24         pos  = fp.pos;
25     }
26
27     void copyFromForce(const FPGrav & force) {
28         acc = force.acc;
29         pot = force.pot;
30     }
31
32     void clear() {
33         acc = 0.0;
34         pot = 0.0;
```

```
35      }
36
37      void predict(PS::F32 dt) {
38          pos  = pos  +       vel * dt + 0.5 * acc * dt * dt;
39          vel2 = vel  + 0.5 * acc * dt;
40      }
41
42      void correct(PS::F32 dt) {
43          vel  = vel2 + 0.5 * acc * dt;
44      }
45
46  };
47
48  PS::F64 FPGrav::eps = 1.0 / 32.0;
49
50  template <class TParticleJ>
51  struct CalcGravity{
52      void operator () (const FPGrav * iptcl,
53                        const PS::S32 ni,
54                        const TParticleJ * jptcl,
55                        const PS::S32 nj,
56                        FPGrav * force) {
57
58          PS::F64 eps2 = FPGrav::eps * FPGrav::eps;
59
60          for(PS::S32 i = 0; i < ni; i++){
61
62              PS::F64vec posi = iptcl[i].pos;
63              PS::F64vec acci = 0.0;
64              PS::F64    poti = 0.0;
65
66              for(PS::S32 j = 0; j < nj; j++){
67                  PS::F64vec posj   = jptcl[j].pos;
68                  PS::F64    massj  = jptcl[j].mass;
69
70                  PS::F64vec drvec  = posi - posj;
71                  PS::F64    dr2    = drvec * drvec + eps2;
72                  PS::F64    drinv  = 1.0 / sqrt(dr2);
73                  PS::F64    mdrinv = drinv * massj;
74
75                  poti -= mdrinv;
76                  acci -= mdrinv * drinv * drinv * drvec;
77              }
78
79              force[i].acc += acci;
80              force[i].pot += poti;
```

```
81          }
82      }
83  };
84
85  template <class Tpsys>
86  void setParticleColdUniformSphere(Tpsys & psys,
87                                    const PS::S32 n_glb) {
88
89      PS::S32 rank  = PS::Comm::getRank();
90      PS::S32 n_loc = (rank == 0) ? n_glb : 0;
91      psys.setNumberOfParticleLocal(n_loc);
92
93      PS::MT::init_genrand(rank);
94      for(PS::S32 i = 0; i < n_loc; i++) {
95          psys[i].mass = 1.0 / (PS::F32)n_glb;
96          const PS::F64 radius = 3.0;
97          do {
98              psys[i].pos[0] = (2. * PS::MT::genrand_res53()
99                                - 1.) * radius;
100             psys[i].pos[1] = (2. * PS::MT::genrand_res53()
101                               - 1.) * radius;
102             psys[i].pos[2] = (2. * PS::MT::genrand_res53()
103                               - 1.) * radius;
104         }while(psys[i].pos * psys[i].pos >= radius * radius);
105         psys[i].vel[0] = 0.0;
106         psys[i].vel[1] = 0.0;
107         psys[i].vel[2] = 0.0;
108     }
109 }
110
111 template<class Tpsys>
112 void predict(Tpsys & system,
113              const PS::F64 dt) {
114     PS::S32 n_loc = system.getNumberOfParticleLocal();
115     for(PS::S32 i = 0; i < n_loc; i++) {
116         system[i].predict(dt);
117     }
118 }
119
120 template<class Tpsys>
121 void correct(Tpsys & system,
122              const PS::F64 dt) {
123     PS::S32 n_loc = system.getNumberOfParticleLocal();
124     for(PS::S32 i = 0; i < n_loc; i++) {
125         system[i].correct(dt);
126     }
```

```
127 }
128
129 template<class Tpsys>
130 PS::F64 calcEnergy(const Tpsys & system) {
131
132     PS::F64 etot = 0.0;
133     PS::F64 etot_loc = 0.0;
134     PS::F64 ekin_loc = 0.0;
135     PS::F64 epot_loc = 0.0;
136
137     const PS::S32 n_loc = system.getNumberOfParticleLocal();
138     for(PS::S32 i = 0; i < n_loc; i++){
139         ekin_loc += system[i].mass *
140                         system[i].vel * system[i].vel;
141         epot_loc += system[i].mass *
142                         (system[i].pot
143                             + system[i].mass / FPGrav::eps);
144     }
145     ekin_loc *= 0.5;
146     epot_loc *= 0.5;
147     etot_loc  = ekin_loc + epot_loc;
148 #ifdef PARTICLE_SIMULATOR_MPI_PARALLEL
149     etot = PS::Comm::getSum(etot_loc);
150 #else
151     etot = etot_loc;
152 #endif
153
154     return etot;
155 }
156
157 int main(int argc, char *argv[]) {
158     PS::F32 time  = 0.0;
159     PS::F32 tend  = 10.0;
160     PS::F32 dtime = 1.0 / 128.0;
161     PS::F32 dtout = 1.0 / 8.0;
162     PS::S64 ntot  = 1024;
163
164     PS::Initialize(argc, argv);
165
166     PS::DomainInfo dinfo;
167     dinfo.initialize();
168
169     PS::ParticleSystem<FPGrav> system_grav;
170     system_grav.initialize();
171
172     PS::TreeForForceLong<FPGrav, FPGrav, FPGrav>::
```

```
173         Monopole tree_grav;
174     tree_grav.initialize(ntot);
175
176     setParticleColdUniformSphere(system_grav, ntot);
177
178     dinfo.decomposeDomainAll(system_grav);
179
180     system_grav.exchangeParticle(dinfo);
181
182     tree_grav.calcForceAllAndWriteBack
183         (CalcGravity<FPGrav>(),
184          CalcGravity<PS::SPJMonopole>(),
185          system_grav,
186          dinfo);
187
188     PS::F64 etot0 = calcEnergy(system_grav);
189     if(PS::Comm::getRank() == 0) {
190         fprintf(stderr,
191                 "time: %10.7f energy: %+e energy error: %+e\n",
192                 time, etot0, (etot0 - etot0) / etot0);
193     }
194
195     while(time < tend) {
196
197         predict(system_grav, dtime);
198         dinfo.decomposeDomainAll(system_grav);
199         system_grav.exchangeParticle(dinfo);
200         tree_grav.calcForceAllAndWriteBack
201             (CalcGravity<FPGrav>(),
202              CalcGravity<PS::SPJMonopole>(),
203              system_grav,
204              dinfo);
205         correct(system_grav, dtime);
206
207         time += dtime;
208         PS::F64 etot1 = calcEnergy(system_grav);
209         if(fmod(time, dtout) == 0.0 &&
210            PS::Comm::getRank() == 0) {
211                 fprintf
212                     (stderr,
213                      "time: %10.7f energy: %+e error: %+e\n",
214                      time, etot1, (etot1 - etot0) / etot0);
215         }
216
217     }
218
```

```
219     PS::Finalize();
220
221     return 0;
222 }
```

7 User Supports

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

7.1 Compile-time problem

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

7.2 Run-time problem

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

7.3 Other cases

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

8 License

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

When Particle Mesh class is used, Ishiyama, Fukushige & Makino (2009, Publications of the Astronomical Society of Japan, 61, 1319), Ishiyama, Nitadori & Makino (2012 SC'12 Proceedings of the International Conference on High Performance Computing, Networking Storage and Analysis, No. 5) should also be cited.

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

Copyright (c) <year> <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.