FDPS Fortran interface Tutorial

Daisuke Namekata, Masaki Iwasawa, Keigo Nitadori, Ataru Tanikawa,
Takayuki Muranushi, Long Wang, Natsuki Hosono, and Jun-ichiro Makino

Particle Simulator Research Team, AICS, RIKEN

0 Contents

1 Change Log

- 2017/2/8
  - English version created.

2 Overview

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

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

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

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

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

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

3 Getting Started

In this section, we describe the first steps you need to do to start using FDPS and FDPS Fortran interface. 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 4.4.5 and K compiler version 1.2.0)
- A Fortran compiler that supports Fortran 2003 Standard and that are interoperable with the above C++ compiler (We have tested with gcc version 4.8.3).
- Python 2.7.5 or later, or, Python 3.4 or later (correct operation is not guaranteed for older Python versions)

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 4.4.5 and K compiler version 1.2.0)
- A Fortran compiler with OpenMP support (it must support Fortran 2003 Standard and be interoperable with the above C++ compiler. We have tested with gcc version 4.8.3).
- Python 2.7.5 or later, or, Python 3.4 or later (correct operation is not guaranteed for older Python versions)

3.2.1.2.2 MPI

- make
- A C++ compiler which supports MPI version 1.3 or later. (We have tested with Open MPI 1.8.1 and K compiler version 1.2.0)
- A Fortran compiler which supports MPI version 1.3 or later (it also must support Fortran 2003 Standard and be interoperable with the above C++ compiler. We have tested with OpenMPI 1.6.4).
- Python 2.7.5 or later, or, Python 3.4 or later (correct operation is not guaranteed for older Python versions)

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 1.8.1 and K compiler version 1.2.0)
- A Fortran compiler which supports OpenMP and MPI version 1.3 or later (it also must support Fortran 2003 Standard and be interoperable with the above C++ compiler. We have tested with OpenMPI 1.6.4).
- Python 2.7.5 or later, or, Python 3.4 or later (correct operation is not guaranteed for older Python versions)

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 1.8.1)
- 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

Because FDPS is a header library[*1)], you do not have to execute the `configure` command. All you need to do is to expand the archive of FDPS in some directory and to setup the include PATH when you compile your codes. An actual procedures can be found in Makefiles of the sample codes explained in § 3.4.

When using FDPS from Fortran, you first must create interface programs to FDPS based on user's codes. Its procedure is described in Chap. 6 of the specification document `doc_spec_ftn_en.pdf`. Makefiles of the sample codes are written so that the interface programs are automatically generated when `make` are running. We recommend that users use Makefiles of the sample codes as a reference when making your own Makefile.

[*1)] A library that consists of header files only.

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.

3.4.1.2 Move to the directory with the sample code

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

3.4.1.3 Edit Makefile

In the directory, there are two Makefiles: `Makefile` and `Makefile.intel`. The former is for GCC and the latter is for the Intel compilers. In this section, we mainly describe `Makefile` in detail and give an usage note on `Makefile.intel` at the end of this section.

First, we describe the default setting of `Makefile`. There are four Makefile variables that need to be set when compiling the sample code. They are the following. `FC` that stores the command to run a Fortran compiler, `CXX` that stores the command to run a C++ compiler, and `FCFLAGS` and `CXXFLAGS`, in which compiler options for both compilers are stored. The initial values of these variables are as follows:

```
FC=gfortran
CXX=g++
FCFLAGS = -std=f2003 -O3 -ffast-math -funroll-loops -finline-functions
CXXFLAGS = -O3 -ffast-math -funroll-loops $(FDPS_INC)
```

where `$(FDPS_INC)` is the variable storing the include PATH for FDPS. It is already set in this Makefile and you do not need to modify it here.

An executable file can be obtained by executing the `make` command after setting the above four Makefile variables appropriately. Edit `Makefile` according the following descriptions. The changes depend on if you use OpenMP and/or MPI.

- Without both OpenMP and MPI
  - Set the variable `FC` the command to run your Fortran compiler
  - Set the variable `CXX` the command to run your C++ compiler
- With OpenMP but not with MPI
  - Set the variable `FC` the command to run your Fortran compiler with OpenMP support
  - Set the variable `CXX` the command to run your C++ compiler with OpenMP support
  - Uncomment the line `FCFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`
  - Uncomment the line `CXXFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`
- With MPI but not with OpenMP
  - Set the variable `FC` the command to run your Fortran compiler that supports MPI
  - Set the variable `CXX` the command to run your C++ compiler that supports MPI
  - Uncomment the line `FCFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`
  - Uncomment the line `CXXFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`
- With both OpenMP and MPI
  - Set the variable `FC` the command to run your Fortran compiler that supports both OpenMP and MPI
  - Set the variable `CXX` the command to run your C++ compiler that supports both OpenMP and MPI
  - Uncomment the line `FCFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`
  - Uncomment the line `FCFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`
  - Uncomment the line `CXXFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`
  - Un comment the line `CXXFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`

Next, we describe useful information when users use this Makefile to compile users' codes. Most important variables when using this Makefile are `FDPS_LOC`, `SRC_USER_DEFINED_TYPE`, and `SRC_USER`. The variable `FDPS_LOC` is used to store the PATH of the top directory of FDPS. Based on the value of `FDPS_LOC`, this Makefile automatically sets a lot of variables related to FDPS, such as the PATH of the directory storing FDPS source files and the PATH of the Python script to generate Fortran interface. Thus, users should set appropriately. The variable `SRC_USER_DEFINED_TYPE` is used to store a list of names of Fortran files in which user-defined types are implemented, while the variable `SRC_USER` is used to store a list of names of Fortran files in which all the rest are implemented. The reason why we divide users'

source files as above is to avoid needless recompilation of FDPS (as a result, we can reduce time required to compile and link users' codes): Because FDPS Fortran interface programs are generated based on user-defined types, we need to recompile of FDPS only when files specified by `SRC_USER_DEFINED_TYPE` are modified. However, there is one thing users should be careful of. When there are dependencies between files specified by `SRC_USER_DEFINED_TYPE` or `SRC_USER`, users must describe these dependencies in Makefile. As for the way of describing dependencies in Makefile, please see the manual of GNU make, for example.

Finally, we describe the usage note for `Makefile.intel`. Except for the initial values of Makefile variables, `Makefile.intel` has the same structure as that of `Makefile`. Hence, users can make use of `Makefile.intel` in the same way as `Makefile` by modifying the values of the variables appropriately. The followings are things to keep in mind when editing Makefile:

- `/opt/intel/bin` should be replaced by the PATH of a directory that stores Intel compilers in your computer system.
- `/opt/intel/include` should be replaced by the PATH of a directory that stores header files used by Intel compilers.
- By default, the value of the variable `LDFLAGS` is `-L/opt/intel/lib/intel64 -L/usr/lib64 -lifport -lifcore -limf -lsvml -lm -lipgo -lirc -lirc_s`. Among them, the option `-lifcore` [*2)] is necessary for the Intel C++ compiler to link C++ objects and Fortran objects[*3)]. When the Intel compiler's libraries are not in the library PATH of the system, users need to specify libraries as `-L/opt/intel/lib/intel64 -L/usr/lib64 -lifport -limf -lsvml -lm -lipgo -lirc -lirc_s`, where `/opt/intel/lib/intel64` is the PATH of directory that stores the Intel compiler's libraries, `/usr/lib64` is the PATH of directory storing the library `libm`. These PATHs depend on the systems users use and therefore users must modify these appropriately. Note that libraries required to compile users' codes (`-l*`) may change depending on the version of Intel compilers and please confirm these.
- As of writing this (2016/12/26), the compile option that invokes OpenMP support is either `-openmp` or `-qopenmp` depending the version of Intel compilers. Recent compilers use the latter option (if the former is specified in this case, the compiler issues a waring of "deprecated").
- Depending on computer systems, all of the necessary settings except for the specification of the option `-lifcore` may be done by environment variables such as `PATH`, `CPATH`, `LD_LIBRARY_PATH`.

3.4.1.4 Run make

Type "make" to run `make`. In the process of `make`, Fortran interface programs are first generated and then they are compiled together with the sample codes.

[*2)] `libifcore` is an Intel compiler's Fortran runtime library.
[*3)] We have tested this with Intel compilers (ver. 17.0.0 20160721).

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:    9.5000000000E+000, energy error:   -3.8046534069E-003
time:    9.6250000000E+000, energy error:   -3.9711750200E-003
time:    9.7500000000E+000, energy error:   -3.8223429428E-003
time:    9.8750000000E+000, energy error:   -3.8843099298E-003
******** FDPS has successfully finished. ********
```

3.4.1.6 Analysis of the result

In the directory `result`, files "snap0000x-proc0000y.dat" have been created. These files store the distribution of particles. Here, x is an integer indicating time and y is an integer indicating MPI process number (y is always 0 if the program is executed without MPI). 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 increate the number of particles to 10000, set the value of the parameter variable `ntot` (defined in the subroutine `f_main()` in the file `f_main.F90`) to 10000, then recompile the sample codes, and run the executable file again.

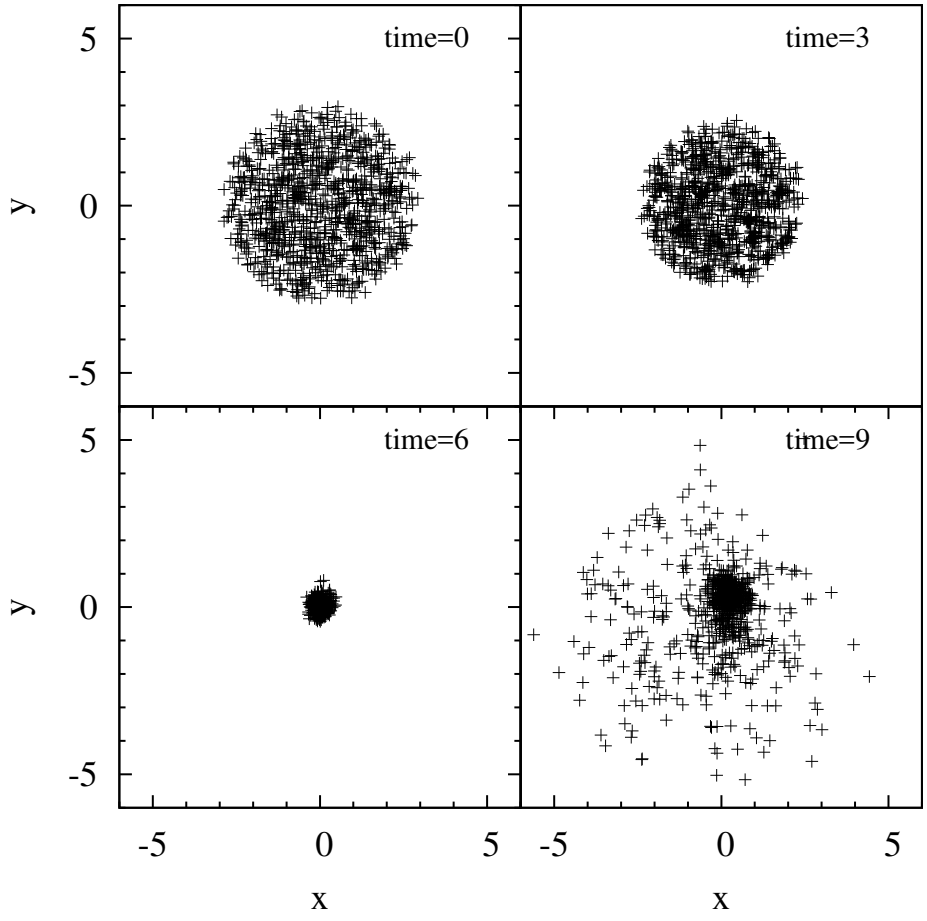

Figure 1:

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`. As in N-body sample code, in the process of `make`, Fortran interface programs are first generated. Then, they are compiled together with SPH sample codes.

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
```

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

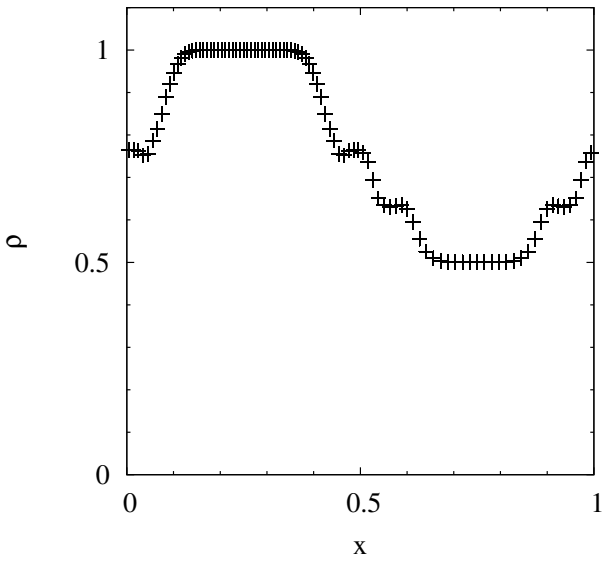

Figure 2:

3.4.2.6 Analysis of the result

In the directory `result`, files "snap0000x-proc0000y.dat" have been created. These files store the distribution of particles. Here, x and y are integers that indicate time and MPI process number, respectively. When executing the program without MPI, y is always 0. 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:

```
$ cd result
$ cat snap00040-proc* > snap00040.dat
$ gnuplot
> plot "snap00040.dat" 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 Fortran interface to 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/fortran/nbody`. The sample code consists of `user_defined.F90` where user-defined types are described, and `f_main.F90` where the other parts of N-body simulation are implemented. In addition to these, there are two Makefiles: `Makefile` (for GCC) and `Makefile.intel` (for Intel compilers).

4.1.2 User-defined types and user-defined functions

In this section, we describe the details of Fortran's derived data types and subroutines 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.F90`).

Listing 1: FullParticle type

```
1   !**** Full particle type
2   type, public, bind(c) :: full_particle !$fdps FP,EPI,EPJ,Force
3      !$fdps copyFromForce full_particle (pot,pot) (acc,acc)
4      !$fdps copyFromFP full_particle (id,id) (mass,mass) (eps,eps) (pos,
           pos)
5      !$fdps clear id=keep, mass=keep, eps=keep, pos=keep, vel=keep
6      integer(kind=c_long_long) :: id
7      real(kind=c_double)  mass !$fdps charge
8      real(kind=c_double) :: eps
9      type(fdps_f64vec) :: pos !$fdps position
10     type(fdps_f64vec) :: vel !$fdps velocity
11     real(kind=c_double) :: pot
12     type(fdps_f64vec) :: acc
13  end type full_particle
```

When developing a simulation code with FDPS Fortran interface, users must specify which user-defined type (FullParticle, EssentialParticleI, EssentialParticleJ, and Force types) a derived data type corresponds to. In FDPS Fortran interface, this is done by adding a

FDPS directive, which is a Fortran's comment text with a special format, to a derived data type. Because FullParticle type is used as EssentialParticleI type, EssentialParticleJ type, and Force type in this sample code, a FDPS directive specifying that the derived data type acts as any types of user-defined types is described:

```
type, public, bind(c) :: full_particle !$fdps FP,EPI,EPJ,Force
```

FDPS must know which member variable of FullParticle type corresponds to which necessary quantity, where **necessary quantities** are defined as the quantities that are necessary in any types of particle simulations (e.g. mass (or charge) and position of a particle), or that are necessary in particular types of particle simulations (e.g. size of a particle). This designation is also done by adding a comment text with a special format to each member variable. In this sample code, in order to specify that member variables `mass`, `pos`, `vel` correspond to mass, position, velocity of a particle, the following directives are described:

```
real(kind=c_double) :: mass !$fdps charge
type(fdps_f64vec) :: pos !$fdps position
type(fdps_f64vec) :: vel !$fdps velocity
```

Note that `velocity` in the directive `!$fdps velocity` is a just reserved keyword and it does not alter the operation of FDPS at the present moment (hence, the designation is arbitrary).

FDPS copies data from FullParticle type toEssentialParticleI type and EssentialParticleJ type, or from Force type to FullParticle type. Users must describe FDPS directives that specify how to copy data. In this sample code, the following directives are described:

```
!$fdps copyFromForce full_particle (pot,pot) (acc,acc)
!$fdps copyFromFP full_particle (id,id) (mass,mass) (eps,eps) (pos,pos)
```

where the FDPS directive with the keyword `copyFromForce` specifies which member variable of Force type is copied to which member variable of FullParticle type. Users **always have to** describe this directive in FullParticle type. The other directive with the keyword `copyFromFP` specifies how to copy data from FullParticle type to EssentialParticleI type and EssentialParticleJ type. This directive **must always** be described in EssentialParticleI type and EssentialParticleJ type. It is described here because FullParticle type in this sample code acts as EssentialParticleI type and EssentialParticleJ type.

FullParticle type also acts as Force type in this code. There is a FDPS directive that users must describe in Force type. It is the directive that specifies how to reset or initialize member variables of Force type before the calculation of interactions. In this code, the following directive is described to direct FDPS to zero-clear member variables corresponding to acceleration and potential only.

```
!$fdps clear id=keep, mass=keep, eps=keep, pos=keep, vel=keep
```

where the syntax *`mbr`*`=keep` to the right of the keyword `clear` is the syntax to direct FDPS not to change the value of member variable *`mbr`*.

Further details about the format of FDPS directive can be found in the specification document of FDPS Fortran interface, `doc_specs_ftn_en.pdf`.

4.1.2.2 calcForceEpEp

You must define a Fortran subroutine calcForceEpEp. It should contain actual code for the calculation of interaction between particles. Listing 2 shows the implementation of calcForceEpEp (see `user_defined.F90`).

Listing 2: calcForceEpEp

```
!**** Interaction function (particle-particle)
subroutine calc_gravity_pp(ep_i,n_ip,ep_j,n_jp,f) bind(c)
   integer(c_int), intent(in), value :: n_ip,n_jp
   type(full_particle), dimension(n_ip), intent(in) :: ep_i
   type(full_particle), dimension(n_jp), intent(in) :: ep_j
   type(full_particle), dimension(n_ip), intent(inout) :: f
   !* Local variables
   integer(c_int) :: i,j
   real(c_double) :: eps2,poti,r3_inv,r_inv
   type(fdps_f64vec) :: xi,ai,rij

   !* Compute force
   do i=1,n_ip
      eps2 = ep_i(i)%eps * ep_i(i)%eps
      xi = ep_i(i)%pos
      ai = 0.0d0
      poti = 0.0d0
      do j=1,n_jp
         rij%x  = xi%x - ep_j(j)%pos%x
         rij%y  = xi%y - ep_j(j)%pos%y
         rij%z  = xi%z - ep_j(j)%pos%z
         r3_inv = rij%x*rij%x &
                + rij%y*rij%y &
                + rij%z*rij%z &
                + eps2
         r_inv  = 1.0d0/sqrt(r3_inv)
         r3_inv = r_inv * r_inv
         r_inv  = r_inv * ep_j(j)%mass
         r3_inv = r3_inv * r_inv
         ai%x   = ai%x - r3_inv * rij%x
         ai%y   = ai%y - r3_inv * rij%y
         ai%z   = ai%z - r3_inv * rij%z
         poti   = poti - r_inv
         ! [IMPORTANT NOTE]
         !   In the innermost loop, we use the components of vectors
         !   directly for vector operations because of the following
         !   reasion. Except for intel compilers with `-ipo` option,
         !   most of Fortran compilers use function calls to perform
         !   vector operations like rij = x - ep_j(j)%pos.
         !   This significantly slow downs the speed of the code.
         !   By using the components of vector directly, we can avoid
         !   these function calls.
      end do
      f(i)%pot = f(i)%pot + poti
      f(i)%acc = f(i)%acc + ai
   end do

end subroutine calc_gravity_pp
```

In this sample code, it is implemented as the subroutine `calc_gravity_pp`. 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. Note that all the data types of the dummy arguments corresponding to user-defined types are `full_particle` type because FullParticle type acts as the other types of user-defined types in this sample code.

4.1.2.3 calcForceEpSp

You must defined a Fortran subroutine calcForceEpSp. It should contain actual code for the calculation of interaction between a particle and a superparticle. Listing 3 shows the implementation of calcForceEpSp (see `user_defined.F90`).

Listing 3: calcForceEpSp

```
1   !**** Interaction function (particle-super particle)
2   subroutine calc_gravity_psp(ep_i,n_ip,ep_j,n_jp,f) bind(c)
3      integer(c_int), intent(in), value :: n_ip,n_jp
4      type(full_particle), dimension(n_ip), intent(in) :: ep_i
5      type(fdps_spj_monopole), dimension(n_jp), intent(in) :: ep_j
6      type(full_particle), dimension(n_ip), intent(inout) :: f
7      !* Local variables
8      integer(c_int) :: i,j
9      real(c_double) :: eps2,poti,r3_inv,r_inv
10     type(fdps_f64vec) :: xi,ai,rij
11
12     do i=1,n_ip
13        eps2 = ep_i(i)%eps * ep_i(i)%eps
14        xi = ep_i(i)%pos
15        ai = 0.0d0
16        poti = 0.0d0
17        do j=1,n_jp
18           rij%x  = xi%x - ep_j(j)%pos%x
19           rij%y  = xi%y - ep_j(j)%pos%y
20           rij%z  = xi%z - ep_j(j)%pos%z
21           r3_inv = rij%x*rij%x &
22                  + rij%y*rij%y &
23                  + rij%z*rij%z &
24                  + eps2
25           r_inv  = 1.0d0/sqrt(r3_inv)
26           r3_inv = r_inv * r_inv
27           r_inv  = r_inv * ep_j(j)%mass
28           r3_inv = r3_inv * r_inv
29           ai%x   = ai%x - r3_inv * rij%x
30           ai%y   = ai%y - r3_inv * rij%y
31           ai%z   = ai%z - r3_inv * rij%z
32           poti   = poti - r_inv
33        end do
34        f(i)%pot = f(i)%pot + poti
35        f(i)%acc = f(i)%acc + ai
36     end do
37
38  end subroutine calc_gravity_psp
```

In this sample code, it is implemented as the subroutine `calc_gravity_psp`. Its dummy arguments are an array of EssentialParticleI type, the number of EssentialParticleI type variables, an array of superparticle type, the number of superparticle type variables, an array of Force type. Note that the data types of EssentialParticleI type and Force type are `full_particle` type because FullParticle type acts as these user-defined types in this sample code. Also note that the data type of superparticle type must be consistent with the type of a Tree object used in the calculation of interactions.

4.1.3 The main body of the user program

In this section, we describe the functions a user should write in a kind of main routine, `f_main()`, to implement gravitational N-body calculation using the FDPS Fortran interface. The reason why we do not use the term main routine clearly is as follows: If users use FDPS Fortran interface, the user code must be written in the subroutine `f_main()`. Thus the user code dose not include the main routine (main program). However, in practice, the `f_main()` plays the same role as a main routine. Thus here we use the term a kind of main routine. The term main routine is suitable for indicating the top level function of the user code. Hereafter, we call `f_main()` the main routine. The main routine of this sample is written in `f_main.F90`.

4.1.3.1 Creation of an object of type `fdps_controller`

In the FDPS Fortran interface, all APIs of FDPS are provided as member functions in the class `FDPS_controller`. This class is defined in the module `fdps_module` in `FDPS_module.F90`. Thus, in order to use APIs, the user must create an object of type `FDPS_controller`. In this sample, the object of type `FDPS_controller`, `fdps_ctrl`, is created in the main routine. Thus, in the following examples, APIs of FDPS are called as a member function of this object.

Listing 4: Creation of an object of type `fdps_controller`

```
1 subroutine f_main()
2    use fdps_module
3    implicit none
4    !* Local variables
5    type(fdps_controller) :: fdps_ctrl
6
7    ! Do something
8
9 end subroutine f_main
```

Note that the code shown above is an only necessary part from the sample code.

4.1.3.2 Initialization and Termination of FDPS

First, users must initialize FDPS by the following code.

Listing 5: Initialization of FDPS

```
1 call fdps_ctrl%PS_Initialize()
```

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 routine.

Listing 6: Termination of FDPS

```
1 call fdps_ctrl%PS_Finalize()
```

4.1.3.3 Creation and initialization of objects

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

4.1.3.3.1 Initialization of objects

In an N-body simulation, one needs to create objects of ParticleSystem type, DomainInfo type, and Tree type. In the Fortran interface, these objects can be handled by using identification number contained in integer type variables. Thus, at the beginning, you should create integer variables to contain the identification numbers. We will show an example bellow. These are written in the main routine `f_main.F90` in the sample code.

Listing 7: Creation of an object

```
1  subroutine f_main()
2     use fdps_module
3     use user_defined_types
4     implicit none
5     !* Local variables
6     integer :: psys_num,dinfo_num,tree_num
7
8     !* Create FDPS objects
9     call fdps_ctrl%create_dinfo(dinfo_num)
10    call fdps_ctrl%create_psys(psys_num,'full_particle')
11    call fdps_ctrl%create_tree(tree_num, &
12                               "Long,full_particle,full_particle,
                                       full_particle,Monopole")
13
14 end subroutine f_main
```

Here, the code shown is just a corresponding part of the sample code. As we can see above, to create the object of type ParticleSystem, you must give the string of the name of the derived data type corresponding to the type FullParticle. As in the case of type ParticleSystem, to create the object of type Tree, you must give the string which indicates the type of tree as an argument of the API. Note that, in both APIs, the name of the derived data type must be written in lower case.

4.1.3.3.2 Initailzation of an object of DomainInfo

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 `init_dinfo` to initialize the objects.

Listing 8: Initailzation of an object of DomainInfo

```
1 call fdps_ctrl%init_dinfo(dinfo_num,coef_ema)
```

Note that the second argument of API `init_dinfo` 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.

4.1.3.3.3 Initialization of the ParticleSystem object

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

Listing 9: Initialization of the ParticleSystem object

```
1 call fdps_ctrl%init_psys(psys_num)
```

4.1.3.3.4 Initialization of the Tree objects

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

Listing 10: Initialization of the Tree objects

```
1 call fdps_ctrl%init_tree(tree_num,ntot,theta, &
2                          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 Initailzation of particle data

To initialize particle data, users must give the particle data to the ParticleSystem object. To do so, users can use APIs `set_nptcl_loc` and `get_psys_fptr` as follows:

Listing 11: Initailzation of particle data

```
1  subroutine foo(fdps_ctrl,psys_num)
2     use fdps_vector
3     use fdps_module
4     use user_defined_types
5     implicit none
6     type(fdps_controller), intent(IN) :: fdps_ctrl
7     integer, intent(IN) :: psys_num
8     !* Local variables
9     integer :: i,nptcl_loc
10    type(full_particle), dimension(:), pointer :: ptcl
11
```

```
12     !* Set # of local particles
13     call fdps_ctrl%set_nptcl_loc(psys_num,nptcl_loc)
14
15     !* Get the pointer to full particle data
16     call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
17
18     !* Initialize particle data
19     do i=1,nptcl_loc
20        ptcl(i)%pos = ! Do something
21     end do
22
23     !* Release the pointer
24     nullify(ptcl)
25
26 end subroutine foo
```

First, you must allocate the memory to store the particle data. To do so, you have only to call API `set_nptcl_loc`. This API sets the number of local particles (the number of particles assigned to the local process) and allocate enough memory to store the particles. To initialize particle data, the address of the allocated memory is needed. Users can obtain the address by using the API `get_psys_fptr`. Users must receive the address by a Fortran pointer. In the example above, the pointer is prepapred as follows:

```
type(full_particle), dimension(:), pointer :: ptcl
```

Once you sets the pointer by the API `get_psys_fptr`, you can use the pointer as an array. In the above example, after initialize particle data, the pointer is freed by the built-in function `nullify`.

4.1.3.5 Time integration loop

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

4.1.3.5.1 Domain Decomposition

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

Listing 12: Domain Decomposition

```
1 if (mod(num_loop,4) == 0) then
2    call fdps_ctrl%decompose_domain_all(dinfo_num,psys_num)
3 end if
```

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

4.1.3.5.2 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 `exchange_particle` of

ParticleSystem object.

Listing 13: Particle Exchange

```
call fdps_ctrl%exchange_particle(psys_num,dinfo_num)
```

4.1.3.5.3 Interaction Calculation

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

Listing 14: Interaction Calculation

```
subroutien f_main()
   use, intrinsic :: iso_c_binding
   use user_defined_types
   implicit none
   !* Local variables
   type(c_funptr) :: pfunc_ep_ep,pfunc_ep_sp

   ! Do somehting

   pfunc_ep_ep = c_funloc(calc_gravity_pp)
   pfunc_ep_sp = c_funloc(calc_gravity_psp)
   call fdps_ctrl%calc_force_all_and_write_back(tree_num,    &
                                                pfunc_ep_ep, &
                                                pfunc_ep_sp, &
                                                psys_num,    &
                                                dinfo_num)

   ! Do something

end subroutine f_main
```

Here, the second and the third arguments are functions pointers of calcForceEpEp and calcForceEpSp. The address of the function in C can be obtained using the built-in function `c_funloc`, which is introduced in Fortran 2003. This built-in function is provided by the module `iso_c_binding` and we use `use` statement to use this module. To store the address in C, we need the variables of derived data type `c_funptr`, which is also introduced in Fortran 2003. In this sample, we use variables of type `c_funptr`, `pfunc_ep_ep` and `pfunc_ep_sp`, to store the address in C of `calc_gravity_pp` and `calc_gravity_psp` and give them to the API.

4.1.3.5.4 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 $t + \Delta t$. In the sample code, these operators are implemented as the functions `kick` and `drift`.

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

Listing 15: $D(\Delta t)K(\frac{\Delta t}{2})$operator

```
!* Leapfrog: Kick-Drift
call kick(fdps_ctrl,psys_num,0.5d0*dt)
time_sys = time_sys + dt
call drift(fdps_ctrl,psys_num,dt)
```

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

Listing 16: $K(\frac{\Delta t}{2})$ operator

```
!* Leapfrog: Kick
call kick(fdps_ctrl,psys_num,0.5d0*dt)
```

4.1.3.6 Update of particle data

To update the data of particles in the subroutines such as `kick` or `drift`, you need to access the data of particles contained in the object of type ParticleSystem. To do so, the user can follow the same way described in section 4.1.3.4.

Listing 17: Update of particle data

```
subroutine foo(fdps_ctrl,psys_num)
   use fdps_vector
   use fdps_module
   use user_defined_types
   implicit none
   type(fdps_controller), intent(IN) :: fdps_ctrl
   integer, intent(IN) :: psys_num
   !* Local variables
   integer :: i,nptcl_loc
   type(full_particle), dimension(:), pointer :: ptcl

   !* Get # of local particles
   nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)

   !* Get the pointer to full particle data
   call fdps_ctrl%get_psys_fptr(psys_num,ptcl)

   !* Initialize or update particle data
   do i=1,nptcl_loc
      ptcl(i)%pos = ! Do something
   end do

   !* Release the pointer
   nullify(ptcl)

end subroutine foo
```

Using API `get_psys_fptr`, you can obtain the address of particle data contained in the object of ParticleSystem as a pointer. The pointer obtained here can be regarded as an array with the size of `nptcl_loc`. Thus user can update the particle data as array.

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 18: 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/fortran/sph`. The sample code consists of `user_defined.F90` where user-defined types are described, and `f_main.F90` where the main loop etc. of the SPH simulation are described. In addition, there are two Makefiles: `Makefile` (for GCC) and `Makefile.intel` (for Intel compilers).

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 19 shows an example implementation of the `FullParticle` type in our sample code (see `user_defined.F90`).

Listing 19: FullParticle type

```
   !**** Full particle type
   type, public, bind(c) :: full_particle !$fdps FP
      !$fdps copyFromForce dens_force (dens,dens)
      !$fdps copyFromForce hydro_force (acc,acc) (eng_dot,eng_dot) (dt,dt)
      real(kind=c_double) :: mass !$fdps charge
      type(fdps_f64vec) :: pos !$fdps position
      type(fdps_f64vec) :: vel
      type(fdps_f64vec) :: acc
      real(kind=c_double) :: dens
      real(kind=c_double) :: eng
      real(kind=c_double) :: pres
      real(kind=c_double) :: smth !$fdps rsearch
      real(kind=c_double) :: snds
      real(kind=c_double) :: eng_dot
      real(kind=c_double) :: dt
      integer(kind=c_long_long) :: id
      type(fdps_f64vec) :: vel_half
```

```
18      real(kind=c_double) :: eng_half
19    end type full_particle
```

Unlike the case of the N-body simulation sample code, the FullParticle type of the SPH simulation sample code does not double as other user-defined types. Thus, to specify that the derived data-type is a FullParticle type, we append the following directive.

```
type, public, bind(c) :: full_particle !$fdps FP
```

In the SPH simulations, the interaction type is short-range. Therefore, a search radius is an additional necessary physical quantity. Including the position and others, the correspondence between the member variable and the necessary physical quantities are specified by the following directive:

```
real(kind=c_double) :: mass !$fdps charge
type(fdps_f64vec) :: pos !$fdps position
real(kind=c_double) :: smth !$fdps rsearch
```

As described in the section of the N-boy simulation code, the keyword `velocity` to specify that a member corresponds to the velocity of a particle is mere a reserved word and not always necessary, we do not specify that in this sample code.

The FullParticle type copies data from a Force type. Users must specify how the data is copied by using of directives. As we will describe later, there are 2 Force types in this SPH sample code. Thus, for each Force type, users must write the directives. In this sample code, these are:

```
!$fdps copyFromForce dens_force (dens,dens)
!$fdps copyFromForce hydro_force (acc,acc) (eng_dot,eng_dot) (dt,dt)
```

4.2.2.2 EssentialParticleI 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. Listing 20 shows an example of EssentialParticleI type of this sample code (see `user_defined.F90`):

Listing 20: EssentialParticleI type

```
1     !**** Essential particle type
2     type, public, bind(c) :: essential_particle !$fdps EPI,EPJ
3       !$fdps copyFromFP full_particle (id,id) (pos,pos) (vel,vel) (mass,
              mass) (smth,smth) (dens,dens) (pres,pres) (snds,snds)
4       integer(kind=c_long_long) :: id
5       type(fdps_f64vec) :: pos !$fdps position
6       type(fdps_f64vec) :: vel
7       real(kind=c_double) :: mass !$fdps charge
8       real(kind=c_double) :: smth !$fdps rsearch
```

```
9       real(kind=c_double) :: dens
10      real(kind=c_double) :: pres
11      real(kind=c_double) :: snds
12   end type essential_particle
```

First, users must indicate to FDPS that this derived data type corresponds to both the EssentialParticleI type and EssentialParticleJ type by using the directives. This sample code describes that as follows:

```
type, public, bind(c) :: essential_particle !$fdps EPI,EPJ
```

Next, users must indicate the correspondence between the each of member variable in this derived data type and necessary physical quantity. For this SPH simulation, a search radius needs to be indicated as well. This sample code describes them as follows:

```
type(fdps_f64vec) :: pos !$fdps position
real(kind=c_double) :: mass !$fdps charge
real(kind=c_double) :: smth !$fdps rsearch
```

The EssentialParticleI and EssentialParticleJ types receive data from the FullParticle type. Users must specify the source member variables in the FullParticle type and the destination member variable in the EssentialParticle? type (?=I,J) that will be copied through the directives. This sample code describes them as follows:

```
!$fdps copyFromFP full_particle (id,id) (pos,pos) (vel,vel) (mass,mass)
(smth,smth) (dens,dens) (pres,pres) (snds,snds)
```

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 21, we show an example of the Force types in this sample code.

Listing 21: Force type

```
1    !**** Force types
2    type, public, bind(c) :: dens_force !$fdps Force
3       !$fdps clear smth=keep
4       real(kind=c_double) :: dens
5       real(kind=c_double) :: smth
6    end type dens_force
7
8    type, public, bind(c) :: hydro_force !$fdps Force
9       !$fdps clear
10      type(fdps_f64vec) :: acc
11      real(kind=c_double) :: eng_dot
12      real(kind=c_double) :: dt
13   end type hydro_force
```

First, users must indicate with directives that these derived data types correspond to the Force types. In this example, these writes:

```
type, public, bind(c) :: dens_force !$fdps Force
type, public, bind(c) :: hydro_force !$fdps Force
```

For these derived data types are Force types, users **must** indicate the initialization methods for the member variables that are accumulated during the interaction calculations. In this sample code, we indicate that only the accumulator variables — density, acceleration (from pressure gradient), time-derivative of energy, and time step to be zero-cleared.

```
!$fdps clear smth=keep
!$fdps clear
```

In this example the Force type `dens_force` includes a member variable `smth` that indicates the smoothing length. For a fixed length SPH, a member variable for the smoothing length in the Force type has nothing to do. We prepare this member variable for the future extension to the variable length SPH for some users. In one of the formulations of the variable length SPH in Springel [2005,MNRAS,364,1105], we need to calculate the smoothing length at the same time we calculate the density. To implement a formulation like that, a Force type need to contain a variable for the smoothing length as in this example. In this sample code for fixed length SPH, the member function `clear` will not zero-clear the variable `smth`, so as not to crush the next computation of the density.

4.2.2.4 Subroutine calcForceEpEp

Users must define a Fortran subroutine calcForceEpEp which specifies the interaction between particles. It should contain actual code for the calculation of interaction between particles. Listing 22 shows the implementation of calcForceEpEp (see `user_defined.F90`).

Listing 22: Subroutine calcForceEpEp

```
1    !**** Interaction function
2    subroutine calc_density(ep_i,n_ip,ep_j,n_jp,f) bind(c)
3       integer(kind=c_int), intent(in), value :: n_ip,n_jp
4       type(essential_particle), dimension(n_ip), intent(in) :: ep_i
5       type(essential_particle), dimension(n_jp), intent(in) :: ep_j
6       type(dens_force), dimension(n_ip), intent(inout) :: f
7       !* Local variables
8       integer(kind=c_int) :: i,j
9       type(fdps_f64vec) :: dr
10
11      do i=1,n_ip
12         f(i)%dens = 0.0d0
13         do j=1,n_jp
14            dr%x = ep_j(j)%pos%x - ep_i(i)%pos%x
15            dr%y = ep_j(j)%pos%y - ep_i(i)%pos%y
16            dr%z = ep_j(j)%pos%z - ep_i(i)%pos%z
17            f(i)%dens = f(i)%dens &
18                      + ep_j(j)%mass * W(dr,ep_i(i)%smth)
19         end do
```

```
20        end do
21
22    end subroutine calc_density
23
24    !**** Interaction function
25    subroutine calc_hydro_force(ep_i,n_ip,ep_j,n_jp,f) bind(c)
26       integer(kind=c_int), intent(in), value :: n_ip,n_jp
27       type(essential_particle), dimension(n_ip), intent(in) :: ep_i
28       type(essential_particle), dimension(n_jp), intent(in) :: ep_j
29       type(hydro_force), dimension(n_ip), intent(inout) :: f
30       !* Local parameters
31       real(kind=c_double), parameter :: C_CFL=0.3d0
32       !* Local variables
33       integer(kind=c_int) :: i,j
34       real(kind=c_double) :: mass_i,mass_j,smth_i,smth_j, &
35                              dens_i,dens_j,pres_i,pres_j, &
36                              snds_i,snds_j
37       real(kind=c_double) :: povrho2_i,povrho2_j, &
38                              v_sig_max,dr_dv,w_ij,v_sig,AV
39       type(fdps_f64vec) :: pos_i,pos_j,vel_i,vel_j, &
40                            dr,dv,gradW_ij
41
42       do i=1,n_ip
43          !* Zero-clear
44          v_sig_max = 0.0d0
45          !* Extract i-particle info.
46          pos_i = ep_i(i)%pos
47          vel_i = ep_i(i)%vel
48          mass_i  = ep_i(i)%mass
49          smth_i  = ep_i(i)%smth
50          dens_i  = ep_i(i)%dens
51          pres_i  = ep_i(i)%pres
52          snds_i  = ep_i(i)%snds
53          povrho2_i = pres_i/(dens_i*dens_i)
54          do j=1,n_jp
55             !* Extract j-particle info.
56             pos_j%x = ep_j(j)%pos%x
57             pos_j%y = ep_j(j)%pos%y
58             pos_j%z = ep_j(j)%pos%z
59             vel_j%x = ep_j(j)%vel%x
60             vel_j%y = ep_j(j)%vel%y
61             vel_j%z = ep_j(j)%vel%z
62             mass_j  = ep_j(j)%mass
63             smth_j  = ep_j(j)%smth
64             dens_j  = ep_j(j)%dens
65             pres_j  = ep_j(j)%pres
66             snds_j  = ep_j(j)%snds
67             povrho2_j = pres_j/(dens_j*dens_j)
68             !* Compute dr & dv
69             dr%x = pos_i%x - pos_j%x
70             dr%y = pos_i%y - pos_j%y
71             dr%z = pos_i%z - pos_j%z
72             dv%x = vel_i%x - vel_j%x
73             dv%y = vel_i%y - vel_j%y
74             dv%z = vel_i%z - vel_j%z
```

```
75              !* Compute the signal velocity
76              dr_dv = dr%x * dv%x + dr%y * dv%y + dr%z * dv%z
77              if (dr_dv < 0.0d0) then
78                 w_ij = dr_dv / sqrt(dr%x * dr%x + dr%y * dr%y + dr%z * dr%z
                        )
79              else
80                 w_ij = 0.0d0
81              end if
82              v_sig = snds_i + snds_j - 3.0d0 * w_ij
83              v_sig_max = max(v_sig_max, v_sig)
84              !* Compute the artificial viscosity
85              AV = - 0.5d0*v_sig*w_ij / (0.5d0*(dens_i+dens_j))
86              !* Compute the average of the gradients of kernel
87              gradW_ij = 0.5d0 * (gradW(dr,smth_i) + gradW(dr,smth_j))
88              !* Compute the acceleration and the heating rate
89              f(i)%acc%x = f(i)%acc%x - mass_j*(povrho2_i+povrho2_j+AV)*
                       gradW_ij%x
90              f(i)%acc%y = f(i)%acc%y - mass_j*(povrho2_i+povrho2_j+AV)*
                       gradW_ij%y
91              f(i)%acc%z = f(i)%acc%z - mass_j*(povrho2_i+povrho2_j+AV)*
                       gradW_ij%z
92              f(i)%eng_dot = f(i)%eng_dot &
93                           + mass_j * (povrho2_i + 0.5d0*AV) &
94                            *(dv%x * gradW_ij%x &
95                              +dv%y * gradW_ij%y &
96                              +dv%z * gradW_ij%z)
97           end do
98           f(i)%dt = C_CFL*2.0d0*smth_i/(v_sig_max*kernel_support_radius)
99        end do
100       ! [IMPORTANT NOTE]
101       !   In the innermost loop, we use the components of vectors
102       !   directly for vector operations because of the following
103       !   reasion. Except for intel compilers with '-ipo' option,
104       !   most of Fortran compilers use function calls to perform
105       !   vector operations like rij = x - ep_j(j)%pos.
106       !   This significantly slow downs the speed of the code.
107       !   By using the components of vector directly, we can avoid
108       !   these function calls.
109
110    end subroutine calc_hydro_force
```

This SPH simulation code include two different forms of interactions, and hence, two different implementations of calcForceEpEp subroutines are needed. In either case, the dummy arguments of the subroutine are, an array of EssentialParticleI, the number of EssentialParticleI, an array of EssentialParticleJ, the number of EssentialParticleJ, and an array of Force.

4.2.3 The main body of the user program

In this section, we describe the functions to be called from the main routine of the user program when a user want to do an SPH simulation using FDPS (for the meaning of "main routine" see section 4.1.3).

4.2.3.1 Creation of an object of type `fdps_controller`

In order to use APIs of FDPS, a user program should create an object of type `FDPS_controller`. In this sample code, `fdps_ctrl`, an object of type `FDPS_controller`, is created in the main routine.

Listing 23: Creation of an object of type `fdps_controller`

```
1 subroutine f_main()
2    use fdps_module
3    implicit none
4    !* Local variables
5    type(fdps_controller) :: fdps_ctrl
6
7    ! Do something
8
9 end subroutine f_main
```

Note that this code snippet only shows the necessary part of the code from the actual sample code. Also note that all FDPS APIs are called as member functions of this object because of the reason described above.

4.2.3.2 Initialization and termination of FDPS

You should first initialize FDPS by the following code.

Listing 24: Initialization of FDPS

```
1 call fdps_ctrl%PS_Initialize()
```

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 routine.

Listing 25: Termination of FDPS

```
1 call fdps_ctrl%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).

Listing 26: Creation of necessary FDPS objects

```
1 subroutine f_main()
```

```
2     use fdps_vector
3     use fdps_module
4     use user_defined_types
5     implicit none
6     !* Local variables
7     integer :: psys_num,dinfo_num
8     integer :: dens_tree_num,hydro_tree_num
9
10    !* Create FDPS objects
11    call fdps_ctrl%create_psys(psys_num,'full_particle')
12    call fdps_ctrl%create_dinfo(dinfo_num)
13    call fdps_ctrl%create_tree(dens_tree_num, &
14                               "Short,dens_force,essential_particle,
                                       essential_particle,Gather")
15    call fdps_ctrl%create_tree(hydro_tree_num, &
16                               "Short,hydro_force,essential_particle,
                                       essential_particle,Symmetry")
17
18 end subroutine f_main
```

Note that here again this code snippet only shows the necessary part of the code from the actual sample code.

API `create_psys` and `create_tree` should receive strings indicating particle type and tree type, respectively. **All derived type names in these strings should be in lower cases.**

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 domain object. After the initialization of the object, the type of the boundary and the 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 27: Initialization of the domain infomation object

```
1 call fdps_ctrl%init_dinfo(dinfo_num,coef_ema)
2 call fdps_ctrl%set_boundary_condition(dinfo_num,fdps_bc_periodic_xyz)
3 call fdps_ctrl%set_pos_root_domain(dinfo_num,pos_ll,pos_ul)
```

4.2.3.3.3 Initialization of the object for particles

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

Listing 28: Initialization of the object for particles

```
1 call fdps_ctrl%init_psys(psys_num)
```

4.2.3.3.4 Initialization of the tree objects

Finally, tree objects should be initialized. The initialization routine should be given the rough number of particles. In this sample, we set three times the total number of particles:

Listing 29: Initialization of tree objects

```
call fdps_ctrl%init_tree(dens_tree_num,3*ntot,theta, &
                         n_leaf_limit,n_group_limit)
call fdps_ctrl%init_tree(hydro_tree_num,3*ntot,theta, &
                         n_leaf_limit,n_group_limit)
```

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 `decompose_domain_all` of the domain information object is called.

Listing 30: Domain Decomposition

```
call fdps_ctrl%decompose_domain_all(dinfo_num,psys_num)
```

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 `exchange_particle` of the object for particles is used.

Listing 31: Particle Exchange

```
call fdps_ctrl%exchange_particle(psys_num,dinfo_num)
```

4.2.3.4.3 Interaction Calculation

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

Listing 32: Interaction Calculation

```
subroutine f_main()
   use, intrinsic :: iso_c_binding
   use user_defined_types
   implicit none
   !* Local variables
   type(c_funptr) :: pfunc_ep_ep

   ! Do something

   pfunc_ep_ep = c_funloc(calc_density)
   call fdps_ctrl%calc_force_all_and_write_back(dens_tree_num, &
                                                pfunc_ep_ep,   &
                                                psys_num,      &
                                                dinfo_num)
   call set_pressure(fdps_ctrl,psys_num)
```

```
16    pfunc_ep_ep = c_funloc(calc_hydro_force)
17    call fdps_ctrl%calc_force_all_and_write_back(hydro_tree_num, &
18                                                 pfunc_ep_ep,    &
19                                                 psys_num,       &
20                                                 dinfo_num)
21
22    ! Do something
23
24  end subroutine f_main
```

For the second argument of API, the function pointer (as in the C language) of function calcForceEpEp should be given.

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 code

In this section, we show a sample code for N-body simulation. This code is the same as what we described in § 3 and § 4. One can create a working code by cut and paste this code and compile and link the resulted source program.

Listing 33: A sample code of N-body simulation (`user_defined.F90`)

```
1  !================================
2  !   MODULE: User defined types
3  !================================
4  module user_defined_types
5     use, intrinsic :: iso_c_binding
6     use fdps_vector
7     use fdps_super_particle
8     implicit none
9
10    !**** Full particle type
11    type, public, bind(c) :: full_particle !$fdps FP,EPI,EPJ,Force
12       !$fdps copyFromForce full_particle (pot,pot) (acc,acc)
13       !$fdps copyFromFP full_particle (id,id) (mass,mass) (eps,eps) (pos,
               pos)
14       !$fdps clear id=keep, mass=keep, eps=keep, pos=keep, vel=keep
15       integer(kind=c_long_long) :: id
16       real(kind=c_double)  mass !$fdps charge
17       real(kind=c_double) :: eps
18       type(fdps_f64vec) :: pos !$fdps position
19       type(fdps_f64vec) :: vel !$fdps velocity
20       real(kind=c_double) :: pot
21       type(fdps_f64vec) :: acc
22    end type full_particle
23
24    contains
25
26    !**** Interaction function (particle-particle)
27    subroutine calc_gravity_pp(ep_i,n_ip,ep_j,n_jp,f) bind(c)
28       integer(c_int), intent(in), value :: n_ip,n_jp
29       type(full_particle), dimension(n_ip), intent(in) :: ep_i
30       type(full_particle), dimension(n_jp), intent(in) :: ep_j
31       type(full_particle), dimension(n_ip), intent(inout) :: f
32       !* Local variables
33       integer(c_int) :: i,j
34       real(c_double) :: eps2,poti,r3_inv,r_inv
35       type(fdps_f64vec) :: xi,ai,rij
36
37       !* Compute force
38       do i=1,n_ip
39          eps2 = ep_i(i)%eps * ep_i(i)%eps
40          xi = ep_i(i)%pos
41          ai = 0.0d0
42          poti = 0.0d0
43          do j=1,n_jp
44             rij%x  = xi%x - ep_j(j)%pos%x
```

```
45              rij%y  = xi%y - ep_j(j)%pos%y
46              rij%z  = xi%z - ep_j(j)%pos%z
47              r3_inv = rij%x*rij%x &
48                     + rij%y*rij%y &
49                     + rij%z*rij%z &
50                     + eps2
51              r_inv  = 1.0d0/sqrt(r3_inv)
52              r3_inv = r_inv * r_inv
53              r_inv  = r_inv * ep_j(j)%mass
54              r3_inv = r3_inv * r_inv
55              ai%x   = ai%x - r3_inv * rij%x
56              ai%y   = ai%y - r3_inv * rij%y
57              ai%z   = ai%z - r3_inv * rij%z
58              poti   = poti - r_inv
59              ! [IMPORTANT NOTE]
60              !   In the innermost loop, we use the components of vectors
61              !   directly for vector operations because of the following
62              !   reasion. Except for intel compilers with `-ipo` option,
63              !   most of Fortran compilers use function calls to perform
64              !   vector operations like rij = x - ep_j(j)%pos.
65              !   This significantly slow downs the speed of the code.
66              !   By using the components of vector directly, we can avoid
67              !   these function calls.
68           end do
69           f(i)%pot = f(i)%pot + poti
70           f(i)%acc = f(i)%acc + ai
71        end do
72
73     end subroutine calc_gravity_pp
74
75     !**** Interaction function (particle-super particle)
76     subroutine calc_gravity_psp(ep_i,n_ip,ep_j,n_jp,f) bind(c)
77        integer(c_int), intent(in), value :: n_ip,n_jp
78        type(full_particle), dimension(n_ip), intent(in) :: ep_i
79        type(fdps_spj_monopole), dimension(n_jp), intent(in) :: ep_j
80        type(full_particle), dimension(n_ip), intent(inout) :: f
81        !* Local variables
82        integer(c_int) :: i,j
83        real(c_double) :: eps2,poti,r3_inv,r_inv
84        type(fdps_f64vec) :: xi,ai,rij
85
86        do i=1,n_ip
87           eps2 = ep_i(i)%eps * ep_i(i)%eps
88           xi = ep_i(i)%pos
89           ai = 0.0d0
90           poti = 0.0d0
91           do j=1,n_jp
92              rij%x  = xi%x - ep_j(j)%pos%x
93              rij%y  = xi%y - ep_j(j)%pos%y
94              rij%z  = xi%z - ep_j(j)%pos%z
95              r3_inv = rij%x*rij%x &
96                     + rij%y*rij%y &
97                     + rij%z*rij%z &
98                     + eps2
99              r_inv  = 1.0d0/sqrt(r3_inv)
```

```
100             r3_inv = r_inv * r_inv
101             r_inv  = r_inv * ep_j(j)%mass
102             r3_inv = r3_inv * r_inv
103             ai%x   = ai%x - r3_inv * rij%x
104             ai%y   = ai%y - r3_inv * rij%y
105             ai%z   = ai%z - r3_inv * rij%z
106             poti   = poti - r_inv
107          end do
108          f(i)%pot = f(i)%pot + poti
109          f(i)%acc = f(i)%acc + ai
110       end do
111
112    end subroutine calc_gravity_psp
113
114 end module user_defined_types
```

Listing 34: A sample code of N-body simulation (`f_main.F90`)

```
 1 !-------------------------------------------------------------------------
 2 !///////////////////// < M A I N  R O U T I N E > ////////////////////
 3 !-------------------------------------------------------------------------
 4 subroutine f_main()
 5    use fdps_module
 6    use user_defined_types
 7    implicit none
 8    !* Local parameters
 9    integer, parameter :: ntot=2**10
10    !-(force parameters)
11    real, parameter :: theta = 0.5
12    integer, parameter :: n_leaf_limit = 8
13    integer, parameter :: n_group_limit = 64
14    !-(domain decomposition)
15    real, parameter :: coef_ema=0.3
16    !-(timing parameters)
17    double precision, parameter :: time_end = 10.0d0
18    double precision, parameter :: dt = 1.0d0/128.0d0
19    double precision, parameter :: dt_diag = 1.0d0/8.0d0
20    double precision, parameter :: dt_snap = 1.0d0
21    !* Local variables
22    integer :: i,j,k,num_loop,ierr
23    integer :: psys_num,dinfo_num,tree_num
24    integer :: nloc
25    logical :: clear
26    double precision :: ekin0,epot0,etot0
27    double precision :: ekin1,epot1,etot1
28    double precision :: time_diag,time_snap,time_sys
29    double precision :: r,acc
30    type(fdps_controller) :: fdps_ctrl
31    type(full_particle), dimension(:), pointer :: ptcl
32    type(c_funptr) :: pfunc_ep_ep,pfunc_ep_sp
33    !-(IO)
34    character(len=64) :: fname
35
36    !* Initialize FDPS
37    call fdps_ctrl%PS_Initialize()
38
```

```
39     !* Create domain info object
40     call fdps_ctrl%create_dinfo(dinfo_num)
41     call fdps_ctrl%init_dinfo(dinfo_num,coef_ema)
42
43     !* Create particle system object
44     call fdps_ctrl%create_psys(psys_num,'full_particle')
45     call fdps_ctrl%init_psys(psys_num)
46
47     !* Create tree object
48     call fdps_ctrl%create_tree(tree_num, &
49                                "Long,full_particle,full_particle,
                                       full_particle,Monopole")
50     call fdps_ctrl%init_tree(tree_num,ntot,theta, &
51                              n_leaf_limit,n_group_limit)
52
53     !* Make an initial condition
54     call setup_IC(fdps_ctrl,psys_num,ntot)
55
56     !* Domain decomposition and exchange particle
57     call fdps_ctrl%decompose_domain_all(dinfo_num,psys_num)
58     call fdps_ctrl%exchange_particle(psys_num,dinfo_num)
59
60     !* Compute force at the initial time
61     pfunc_ep_ep = c_funloc(calc_gravity_pp)
62     pfunc_ep_sp = c_funloc(calc_gravity_psp)
63     call fdps_ctrl%calc_force_all_and_write_back(tree_num,    &
64                                                  pfunc_ep_ep, &
65                                                  pfunc_ep_sp, &
66                                                  psys_num,    &
67                                                  dinfo_num)
68     !* Compute energies at the initial time
69     clear = .true.
70     call calc_energy(fdps_ctrl,psys_num,etot0,ekin0,epot0,clear)
71
72     !* Time integration
73     time_diag = 0.0d0
74     time_snap = 0.0d0
75     time_sys  = 0.0d0
76     num_loop = 0
77     do
78        !* Output
79       !if (fdps_ctrl%get_rank() == 0) then
80       !   write(*,50)num_loop,time_sys
81       !   50 format('(num_loop, time_sys) = ',i5,1x,1es25.16e3)
82       !end if
83        if ( (time_sys >= time_snap) .or. &
84             (((time_sys + dt) - time_snap) > (time_snap - time_sys)) ) then
85           call output(fdps_ctrl,psys_num)
86           time_snap = time_snap + dt_snap
87        end if
88
89        !* Compute energies and output the results
90        clear = .true.
91        call calc_energy(fdps_ctrl,psys_num,etot1,ekin1,epot1,clear)
92        if (fdps_ctrl%get_rank() == 0) then
```

```
93          if ( (time_sys >= time_diag) .or. &
94               (((time_sys + dt) - time_diag) > (time_diag - time_sys)) )
                    then
95             write(*,100)time_sys,(etot1-etot0)/etot0
96             100 format("time: ",1es20.10e3,", energy error: ",1es20.10e3)
97             time_diag = time_diag + dt_diag
98          end if
99       end if
100
101      !* Leapfrog: Kick-Drift
102      call kick(fdps_ctrl,psys_num,0.5d0*dt)
103      time_sys = time_sys + dt
104      call drift(fdps_ctrl,psys_num,dt)
105
106      !* Domain decomposition & exchange particle
107      if (mod(num_loop,4) == 0) then
108         call fdps_ctrl%decompose_domain_all(dinfo_num,psys_num)
109      end if
110      call fdps_ctrl%exchange_particle(psys_num,dinfo_num)
111
112      !* Force calculation
113      pfunc_ep_ep = c_funloc(calc_gravity_pp)
114      pfunc_ep_sp = c_funloc(calc_gravity_psp)
115      call fdps_ctrl%calc_force_all_and_write_back(tree_num,    &
116                                                   pfunc_ep_ep, &
117                                                   pfunc_ep_sp, &
118                                                   psys_num,    &
119                                                   dinfo_num)
120      !* Leapfrog: Kick
121      call kick(fdps_ctrl,psys_num,0.5d0*dt)
122
123      !* Update num_loop
124      num_loop = num_loop + 1
125
126      !* Termination
127      if (time_sys >= time_end) then
128         exit
129      end if
130   end do
131
132   !* Finalize FDPS
133   call fdps_ctrl%PS_Finalize()
134
135 end subroutine f_main
136
137 !-------------------------------------------------------------------------
138 !////////////////////////// S U B R O U T I N E ////////////////////////////
139 !////////////////////////// < S E T U P _ I C > ////////////////////////////
140 !-------------------------------------------------------------------------
141 subroutine setup_IC(fdps_ctrl,psys_num,nptcl_glb)
142    use fdps_vector
143    use fdps_module
144    use user_defined_types
145    implicit none
146    type(fdps_controller), intent(IN) :: fdps_ctrl
```

```
147     integer, intent(IN) :: psys_num,nptcl_glb
148     !* Local parameters
149     double precision, parameter :: m_tot=1.0d0
150     double precision, parameter :: rmax=3.0d0,r2max=rmax*rmax
151     !* Local variables
152     integer :: i,j,k,ierr
153     integer :: nprocs,myrank
154     double precision :: r2,cm_mass
155     type(fdps_f64vec) :: cm_pos,cm_vel,pos
156     type(full_particle), dimension(:), pointer :: ptcl
157     character(len=64) :: fname
158
159     !* Get # of MPI processes and rank number
160     nprocs = fdps_ctrl%get_num_procs()
161     myrank = fdps_ctrl%get_rank()
162
163     !* Make an initial condition at RANK 0
164     if (myrank == 0) then
165        !* Set # of local particles
166        call fdps_ctrl%set_nptcl_loc(psys_num,nptcl_glb)
167
168        !* Create an uniform sphere of particles
169        !** get the pointer to full particle data
170        call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
171        !** initialize Mersenne twister
172        call fdps_ctrl%MT_init_genrand(0)
173        do i=1,nptcl_glb
174           ptcl(i)%id   = i
175           ptcl(i)%mass = m_tot/nptcl_glb
176           do
177              ptcl(i)%pos%x = (2.0d0*fdps_ctrl%MT_genrand_res53()-1.0d0) *
                    rmax
178              ptcl(i)%pos%y = (2.0d0*fdps_ctrl%MT_genrand_res53()-1.0d0) *
                    rmax
179              ptcl(i)%pos%z = (2.0d0*fdps_ctrl%MT_genrand_res53()-1.0d0) *
                    rmax
180              r2 = ptcl(i)%pos*ptcl(i)%pos
181              if ( r2 < r2max ) exit
182           end do
183           ptcl(i)%vel = 0.0d0
184           ptcl(i)%eps = 1.0d0/32.0d0
185        end do
186
187        !* Correction
188        cm_pos  = 0.0d0
189        cm_vel  = 0.0d0
190        cm_mass = 0.0d0
191        do i=1,nptcl_glb
192           cm_pos  = cm_pos  + ptcl(i)%mass * ptcl(i)%pos
193           cm_vel  = cm_vel  + ptcl(i)%mass * ptcl(i)%vel
194           cm_mass = cm_mass + ptcl(i)%mass
195        end do
196        cm_pos = cm_pos/cm_mass
197        cm_vel = cm_vel/cm_mass
198        do i=1,nptcl_glb
```

```
199          ptcl(i)%pos = ptcl(i)%pos - cm_pos
200          ptcl(i)%vel = ptcl(i)%vel - cm_vel
201       end do
202
203       !* Output
204      !fname = 'initial.dat'
205      !open(unit=9,file=trim(fname),action='write',status='replace', &
206      !     form='unformatted',access='stream')
207      !open(unit=9,file=trim(fname),action='write',status='replace')
208      !   do i=1,nptcl_glb
209      !     !write(9)ptcl(i)%pos%x,ptcl(i)%pos%y,ptcl(i)%pos%z
210      !      write(9,'(3es25.16e3)')ptcl(i)%pos%x,ptcl(i)%pos%y,ptcl(i)%pos
               %z
211      !   end do
212      !close(unit=9)
213
214       !* Release the pointer
215       nullify( ptcl )
216
217    else
218       call fdps_ctrl%set_nptcl_loc(psys_num,0)
219    end if
220
221 end subroutine setup_IC
222
223 !-------------------------------------------------------------------------
224 !////////////////////////// S U B R O U T I N E ////////////////////////////
225 !//////////////////////////     < K I C K >     ////////////////////////////
226 !-------------------------------------------------------------------------
227 subroutine kick(fdps_ctrl,psys_num,dt)
228    use fdps_vector
229    use fdps_module
230    use user_defined_types
231    implicit none
232    type(fdps_controller), intent(IN) :: fdps_ctrl
233    integer, intent(IN) :: psys_num
234    double precision, intent(IN) :: dt
235    !* Local variables
236    integer :: i,nptcl_loc
237    type(full_particle), dimension(:), pointer :: ptcl
238
239    !* Get # of local particles
240    nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
241
242    !* Get the pointer to full particle data
243    call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
244    do i=1,nptcl_loc
245       ptcl(i)%vel = ptcl(i)%vel + ptcl(i)%acc * dt
246    end do
247    nullify(ptcl)
248
249 end subroutine kick
250
251 !-------------------------------------------------------------------------
252 !////////////////////////// S U B R O U T I N E ////////////////////////////
```

```
253 !//////////////////////    < D R I F T >    //////////////////////
254 !-------------------------------------------------------------------
255 subroutine drift(fdps_ctrl,psys_num,dt)
256    use fdps_vector
257    use fdps_module
258    use user_defined_types
259    implicit none
260    type(fdps_controller), intent(IN) :: fdps_ctrl
261    integer, intent(IN) :: psys_num
262    double precision, intent(IN) :: dt
263    !* Local variables
264    integer :: i,nptcl_loc
265    type(full_particle), dimension(:), pointer :: ptcl
266
267    !* Get # of local particles
268    nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
269
270    !* Get the pointer to full particle data
271    call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
272    do i=1,nptcl_loc
273       ptcl(i)%pos = ptcl(i)%pos + ptcl(i)%vel * dt
274    end do
275    nullify(ptcl)
276
277 end subroutine drift
278
279 !-------------------------------------------------------------------
280 !//////////////////////    S U B R O U T I N E    //////////////////////
281 !////////////////////// < C A L C _ E N E R G Y > //////////////////////
282 !-------------------------------------------------------------------
283 subroutine calc_energy(fdps_ctrl,psys_num,etot,ekin,epot,clear)
284    use fdps_vector
285    use fdps_module
286    use user_defined_types
287    implicit none
288    type(fdps_controller), intent(IN) :: fdps_ctrl
289    integer, intent(IN) :: psys_num
290    double precision, intent(INOUT) :: etot,ekin,epot
291    logical, intent(IN) :: clear
292    !* Local variables
293    integer :: i,nptcl_loc
294    double precision :: etot_loc,ekin_loc,epot_loc
295    type(full_particle), dimension(:), pointer :: ptcl
296
297    !* Clear energies
298    if (clear .eqv. .true.) then
299       etot = 0.0d0
300       ekin = 0.0d0
301       epot = 0.0d0
302    end if
303
304    !* Get # of local particles
305    nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
306    call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
307
```

```
308     !* Compute energies
309     ekin_loc = 0.0d0
310     epot_loc = 0.0d0
311     do i=1,nptcl_loc
312        ekin_loc = ekin_loc + ptcl(i)%mass * ptcl(i)%vel * ptcl(i)%vel
313        epot_loc = epot_loc + ptcl(i)%mass * (ptcl(i)%pot + ptcl(i)%mass/
                 ptcl(i)%eps)
314     end do
315     ekin_loc = ekin_loc * 0.5d0
316     epot_loc = epot_loc * 0.5d0
317     etot_loc = ekin_loc + epot_loc
318     call fdps_ctrl%get_sum(ekin_loc,ekin)
319     call fdps_ctrl%get_sum(epot_loc,epot)
320     call fdps_ctrl%get_sum(etot_loc,etot)
321
322     !* Release the pointer
323     nullify(ptcl)
324
325 end subroutine calc_energy
326
327 !-------------------------------------------------------------------------
328 !////////////////////////// S U B R O U T I N E ////////////////////////////
329 !//////////////////////////   < O U T P U T >   ////////////////////////////
330 !-------------------------------------------------------------------------
331 subroutine output(fdps_ctrl,psys_num)
332     use fdps_vector
333     use fdps_module
334     use user_defined_types
335     implicit none
336     type(fdps_controller), intent(IN) :: fdps_ctrl
337     integer, intent(IN) :: psys_num
338     !* Local parameters
339     character(len=16), parameter :: root_dir="result"
340     character(len=16), parameter :: file_prefix_1st="snap"
341     character(len=16), parameter :: file_prefix_2nd="proc"
342     !* Local variables
343     integer :: i,nptcl_loc
344     integer :: myrank
345     character(len=5) :: file_num,proc_num
346     character(len=64) :: cmd,sub_dir,fname
347     type(full_particle), dimension(:), pointer :: ptcl
348     !* Static variables
349     integer, save :: snap_num=0
350
351     !* Get the rank number
352     myrank = fdps_ctrl%get_rank()
353
354     !* Get # of local particles
355     nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
356
357     !* Get the pointer to full particle data
358     call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
359
360     !* Output
361     write(file_num,"(i5.5)")snap_num
```

```
362     write(proc_num,"(i5.5)")myrank
363     fname =  trim(root_dir) // "/" &
364           // trim(file_prefix_1st) // file_num // "-" &
365           // trim(file_prefix_2nd) // proc_num // ".dat"
366     open(unit=9,file=trim(fname),action='write',status='replace')
367        do i=1,nptcl_loc
368           write(9,100)ptcl(i)%id,ptcl(i)%mass, &
369                       ptcl(i)%pos%x,ptcl(i)%pos%y,ptcl(i)%pos%z, &
370                       ptcl(i)%vel%x,ptcl(i)%vel%y,ptcl(i)%vel%z
371           100 format(i8,1x,7e25.16e3)
372        end do
373     close(unit=9)
374     nullify(ptcl)
375
376     !* Update snap_num
377     snap_num = snap_num + 1
378
379 end subroutine output
```

5.2 SPH simulation with fixed smoothing length

In this section, we show a sample code for SPH simulation with fixed smoothing length. This code is the same as what we described in § 3 and § 4. One can create a working code by cut and paste this code and compile and link the resulted source program.

Listing 35: A sample code of SPH simulation with fixed smoothing length (`user_defined.F90`)

```
1  !================================
2  !   MODULE: User defined types
3  !================================
4  module user_defined_types
5     use, intrinsic :: iso_c_binding
6     use fdps_vector
7     implicit none
8
9     !* Private parameters
10    real(kind=c_double), parameter, private :: pi=datan(1.0d0)*4.0d0
11    !* Public parameters
12    real(kind=c_double), parameter, public :: kernel_support_radius=2.5d0
13
14    !**** Force types
15    type, public, bind(c) :: dens_force !$fdps Force
16       !$fdps clear smth=keep
17       real(kind=c_double) :: dens
18       real(kind=c_double) :: smth
19    end type dens_force
20
21    type, public, bind(c) :: hydro_force !$fdps Force
22       !$fdps clear
23       type(fdps_f64vec) :: acc
24       real(kind=c_double) :: eng_dot
25       real(kind=c_double) :: dt
26    end type hydro_force
```

```
27
28    !**** Full particle type
29    type, public, bind(c) :: full_particle !$fdps FP
30       !$fdps copyFromForce dens_force (dens,dens)
31       !$fdps copyFromForce hydro_force (acc,acc) (eng_dot,eng_dot) (dt,dt)
32       real(kind=c_double) :: mass !$fdps charge
33       type(fdps_f64vec) :: pos !$fdps position
34       type(fdps_f64vec) :: vel
35       type(fdps_f64vec) :: acc
36       real(kind=c_double) :: dens
37       real(kind=c_double) :: eng
38       real(kind=c_double) :: pres
39       real(kind=c_double) :: smth !$fdps rsearch
40       real(kind=c_double) :: snds
41       real(kind=c_double) :: eng_dot
42       real(kind=c_double) :: dt
43       integer(kind=c_long_long) :: id
44       type(fdps_f64vec) :: vel_half
45       real(kind=c_double) :: eng_half
46    end type full_particle
47
48    !**** Essential particle type
49    type, public, bind(c) :: essential_particle !$fdps EPI,EPJ
50       !$fdps copyFromFP full_particle (id,id) (pos,pos) (vel,vel) (mass,
             mass) (smth,smth) (dens,dens) (pres,pres) (snds,snds)
51       integer(kind=c_long_long) :: id
52       type(fdps_f64vec) :: pos !$fdps position
53       type(fdps_f64vec) :: vel
54       real(kind=c_double) :: mass !$fdps charge
55       real(kind=c_double) :: smth !$fdps rsearch
56       real(kind=c_double) :: dens
57       real(kind=c_double) :: pres
58       real(kind=c_double) :: snds
59    end type essential_particle
60
61    !* Public routines
62    public :: W
63    public :: gradW
64    public :: calc_density
65    public :: calc_hydro_force
66
67    contains
68
69    !-----------------------------------------------------------------------
70    pure function W(dr,h)
71       implicit none
72       real(kind=c_double) :: W
73       type(fdps_f64vec), intent(in) :: dr
74       real(kind=c_double), intent(in) :: h
75       !* Local variables
76       real(kind=c_double) :: s,s1,s2
77
78       s = dsqrt(dr%x*dr%x &
79                 +dr%y*dr%y &
80                 +dr%z*dr%z)/h
```

```
81        s1 = 1.0d0 - s
82        if (s1 < 0.0d0) s1 = 0.0d0
83        s2 = 0.5d0 - s
84        if (s2 < 0.0d0) s2 = 0.0d0
85        W = (s1*s1*s1) - 4.0d0*(s2*s2*s2)
86        W = W * 16.0d0/(pi*h*h*h)
87
88     end function W
89
90     !-------------------------------------------------------------------
91     pure function gradW(dr,h)
92        implicit none
93        type(fdps_f64vec) :: gradW
94        type(fdps_f64vec), intent(in) :: dr
95        real(kind=c_double), intent(in) :: h
96        !* Local variables
97        real(kind=c_double) :: dr_abs,s,s1,s2,coef
98
99        dr_abs = dsqrt(dr%x*dr%x &
100                      +dr%y*dr%y &
101                      +dr%z*dr%z)
102       s = dr_abs/h
103       s1 = 1.0d0 - s
104       if (s1 < 0.0d0) s1 = 0.0d0
105       s2 = 0.5d0 - s
106       if (s2 < 0.0d0) s2 = 0.0d0
107       coef = - 3.0d0*(s1*s1) + 12.0d0*(s2*s2)
108       coef = coef * 16.0d0/(pi*h*h*h)
109       coef = coef / (dr_abs*h + 1.0d-6*h)
110       gradW%x = dr%x * coef
111       gradW%y = dr%y * coef
112       gradW%z = dr%z * coef
113
114    end function gradW
115
116    !**** Interaction function
117    subroutine calc_density(ep_i,n_ip,ep_j,n_jp,f) bind(c)
118       integer(kind=c_int), intent(in), value :: n_ip,n_jp
119       type(essential_particle), dimension(n_ip), intent(in) :: ep_i
120       type(essential_particle), dimension(n_jp), intent(in) :: ep_j
121       type(dens_force), dimension(n_ip), intent(inout) :: f
122       !* Local variables
123       integer(kind=c_int) :: i,j
124       type(fdps_f64vec) :: dr
125
126       do i=1,n_ip
127          f(i)%dens = 0.0d0
128          do j=1,n_jp
129             dr%x = ep_j(j)%pos%x - ep_i(i)%pos%x
130             dr%y = ep_j(j)%pos%y - ep_i(i)%pos%y
131             dr%z = ep_j(j)%pos%z - ep_i(i)%pos%z
132             f(i)%dens = f(i)%dens &
133                       + ep_j(j)%mass * W(dr,ep_i(i)%smth)
134          end do
135       end do
```

```
136
137     end subroutine calc_density
138
139     !**** Interaction function
140     subroutine calc_hydro_force(ep_i,n_ip,ep_j,n_jp,f) bind(c)
141        integer(kind=c_int), intent(in), value :: n_ip,n_jp
142        type(essential_particle), dimension(n_ip), intent(in) :: ep_i
143        type(essential_particle), dimension(n_jp), intent(in) :: ep_j
144        type(hydro_force), dimension(n_ip), intent(inout) :: f
145        !* Local parameters
146        real(kind=c_double), parameter :: C_CFL=0.3d0
147        !* Local variables
148        integer(kind=c_int) :: i,j
149        real(kind=c_double) :: mass_i,mass_j,smth_i,smth_j, &
150                               dens_i,dens_j,pres_i,pres_j, &
151                               snds_i,snds_j
152        real(kind=c_double) :: povrho2_i,povrho2_j, &
153                               v_sig_max,dr_dv,w_ij,v_sig,AV
154        type(fdps_f64vec) :: pos_i,pos_j,vel_i,vel_j, &
155                             dr,dv,gradW_ij
156
157        do i=1,n_ip
158           !* Zero-clear
159           v_sig_max = 0.0d0
160           !* Extract i-particle info.
161           pos_i = ep_i(i)%pos
162           vel_i = ep_i(i)%vel
163           mass_i  = ep_i(i)%mass
164           smth_i  = ep_i(i)%smth
165           dens_i  = ep_i(i)%dens
166           pres_i  = ep_i(i)%pres
167           snds_i  = ep_i(i)%snds
168           povrho2_i = pres_i/(dens_i*dens_i)
169           do j=1,n_jp
170              !* Extract j-particle info.
171              pos_j%x = ep_j(j)%pos%x
172              pos_j%y = ep_j(j)%pos%y
173              pos_j%z = ep_j(j)%pos%z
174              vel_j%x = ep_j(j)%vel%x
175              vel_j%y = ep_j(j)%vel%y
176              vel_j%z = ep_j(j)%vel%z
177              mass_j  = ep_j(j)%mass
178              smth_j  = ep_j(j)%smth
179              dens_j  = ep_j(j)%dens
180              pres_j  = ep_j(j)%pres
181              snds_j  = ep_j(j)%snds
182              povrho2_j = pres_j/(dens_j*dens_j)
183              !* Compute dr & dv
184              dr%x = pos_i%x - pos_j%x
185              dr%y = pos_i%y - pos_j%y
186              dr%z = pos_i%z - pos_j%z
187              dv%x = vel_i%x - vel_j%x
188              dv%y = vel_i%y - vel_j%y
189              dv%z = vel_i%z - vel_j%z
190              !* Compute the signal velocity
```

```
191             dr_dv = dr%x * dv%x + dr%y * dv%y + dr%z * dv%z
192             if (dr_dv < 0.0d0) then
193                w_ij = dr_dv / sqrt(dr%x * dr%x + dr%y * dr%y + dr%z * dr%z
                       )
194             else
195                w_ij = 0.0d0
196             end if
197             v_sig = snds_i + snds_j - 3.0d0 * w_ij
198             v_sig_max = max(v_sig_max, v_sig)
199             !* Compute the artificial viscosity
200             AV = - 0.5d0*v_sig*w_ij / (0.5d0*(dens_i+dens_j))
201             !* Compute the average of the gradients of kernel
202             gradW_ij = 0.5d0 * (gradW(dr,smth_i) + gradW(dr,smth_j))
203             !* Compute the acceleration and the heating rate
204             f(i)%acc%x = f(i)%acc%x - mass_j*(povrho2_i+povrho2_j+AV)*
                      gradW_ij%x
205             f(i)%acc%y = f(i)%acc%y - mass_j*(povrho2_i+povrho2_j+AV)*
                      gradW_ij%y
206             f(i)%acc%z = f(i)%acc%z - mass_j*(povrho2_i+povrho2_j+AV)*
                      gradW_ij%z
207             f(i)%eng_dot = f(i)%eng_dot &
208                          + mass_j * (povrho2_i + 0.5d0*AV) &
209                           *(dv%x * gradW_ij%x &
210                            +dv%y * gradW_ij%y &
211                            +dv%z * gradW_ij%z)
212          end do
213          f(i)%dt = C_CFL*2.0d0*smth_i/(v_sig_max*kernel_support_radius)
214       end do
215       ! [IMPORTANT NOTE]
216       !   In the innermost loop, we use the components of vectors
217       !   directly for vector operations because of the following
218       !   reasion. Except for intel compilers with '-ipo' option,
219       !   most of Fortran compilers use function calls to perform
220       !   vector operations like rij = x - ep_j(j)%pos.
221       !   This significantly slow downs the speed of the code.
222       !   By using the components of vector directly, we can avoid
223       !   these function calls.
224
225    end subroutine calc_hydro_force
226
227 end module user_defined_types
```

Listing 36: A sample code of SPH simulation with fixed smoothing length (`f_main.F90`)

```
1  !-------------------------------------------------------------------------
2  !////////////////////  < M A I N   R O U T I N E >  ////////////////////
3  !-------------------------------------------------------------------------
4  subroutine f_main()
5     use fdps_vector
6     use fdps_module
7     use user_defined_types
8     implicit none
9     !* Local parameters
10    !-(force parameters)
11    real, parameter :: theta = 0.5
12    integer, parameter :: n_leaf_limit = 8
```

```
13    integer, parameter :: n_group_limit = 64
14    !-(domain decomposition)
15    real, parameter :: coef_ema=0.3
16    !-(IO)
17    integer, parameter :: output_interval=10
18    !* Local variables
19    integer :: i,j,k,ierr
20    integer :: nstep
21    integer :: psys_num,dinfo_num
22    integer :: dens_tree_num,hydro_tree_num
23    integer :: ntot,nloc
24    logical :: clear
25    double precision :: time,dt,end_time
26    type(fdps_f64vec) :: pos_ll,pos_ul
27    type(fdps_controller) :: fdps_ctrl
28    type(full_particle), dimension(:), pointer :: ptcl
29    type(c_funptr) :: pfunc_ep_ep
30    !-(IO)
31    character(len=64) :: filename
32    !* External routines
33    double precision, external :: get_timestep
34
35    !* Initialize FDPS
36    call fdps_ctrl%PS_Initialize()
37
38    !* Make an instance of ParticleSystem and initialize it
39    call fdps_ctrl%create_psys(psys_num,'full_particle')
40    call fdps_ctrl%init_psys(psys_num)
41
42    !* Make an initial condition and initialize the particle system
43    call setup_IC(fdps_ctrl,psys_num,end_time,pos_ll,pos_ul)
44
45    !* Make an instance of DomainInfo and initialize it
46    call fdps_ctrl%create_dinfo(dinfo_num)
47    call fdps_ctrl%init_dinfo(dinfo_num,coef_ema)
48    call fdps_ctrl%set_boundary_condition(dinfo_num,fdps_bc_periodic_xyz)
49    call fdps_ctrl%set_pos_root_domain(dinfo_num,pos_ll,pos_ul)
50
51    !* Perform domain decomposition and exchange particles
52    call fdps_ctrl%decompose_domain_all(dinfo_num,psys_num)
53    call fdps_ctrl%exchange_particle(psys_num,dinfo_num)
54
55    !* Make two tree structures
56    ntot = fdps_ctrl%get_nptcl_glb(psys_num)
57    !** dens_tree (used for the density calculation)
58    call fdps_ctrl%create_tree(dens_tree_num, &
59                               "Short,dens_force,essential_particle,
                                     essential_particle,Gather")
60    call fdps_ctrl%init_tree(dens_tree_num,3*ntot,theta, &
61                             n_leaf_limit,n_group_limit)
62
63    !** hydro_tree (used for the force calculation)
64    call fdps_ctrl%create_tree(hydro_tree_num, &
65                               "Short,hydro_force,essential_particle,
                                     essential_particle,Symmetry")
```

```
66    call fdps_ctrl%init_tree(hydro_tree_num,3*ntot,theta, &
67                             n_leaf_limit,n_group_limit)
68
69    !* Compute density, pressure, acceleration due to pressure gradient
70    pfunc_ep_ep = c_funloc(calc_density)
71    call fdps_ctrl%calc_force_all_and_write_back(dens_tree_num, &
72                                                 pfunc_ep_ep,   &
73                                                 psys_num,      &
74                                                 dinfo_num)
75    call set_pressure(fdps_ctrl,psys_num)
76    pfunc_ep_ep = c_funloc(calc_hydro_force)
77    call fdps_ctrl%calc_force_all_and_write_back(hydro_tree_num, &
78                                                 pfunc_ep_ep,   &
79                                                 psys_num,      &
80                                                 dinfo_num)
81    !* Get timestep
82    dt = get_timestep(fdps_ctrl,psys_num)
83
84    !* Main loop for time integration
85    nstep = 0; time = 0.0d0
86    do
87       !* Leap frog: Initial Kick & Full Drift
88       call initial_kick(fdps_ctrl,psys_num,dt)
89       call full_drift(fdps_ctrl,psys_num,dt)
90
91       !* Adjust the positions of the SPH particles that run over
92       !  the computational boundaries.
93       call fdps_ctrl%adjust_pos_into_root_domain(psys_num,dinfo_num)
94
95       !* Leap frog: Predict
96       call predict(fdps_ctrl,psys_num,dt)
97
98       !* Perform domain decomposition and exchange particles again
99       call fdps_ctrl%decompose_domain_all(dinfo_num,psys_num)
100      call fdps_ctrl%exchange_particle(psys_num,dinfo_num)
101
102      !* Compute density, pressure, acceleration due to pressure gradient
103      pfunc_ep_ep = c_funloc(calc_density)
104      call fdps_ctrl%calc_force_all_and_write_back(dens_tree_num, &
105                                                   pfunc_ep_ep,   &
106                                                   psys_num,      &
107                                                   dinfo_num)
108      call set_pressure(fdps_ctrl,psys_num)
109      pfunc_ep_ep = c_funloc(calc_hydro_force)
110      call fdps_ctrl%calc_force_all_and_write_back(hydro_tree_num, &
111                                                   pfunc_ep_ep,   &
112                                                   psys_num,      &
113                                                   dinfo_num)
114
115      !* Get a new timestep
116      dt = get_timestep(fdps_ctrl,psys_num)
117
118      !* Leap frog: Final Kick
119      call final_kick(fdps_ctrl,psys_num,dt)
120
```

```
121       !* Output result files
122       if (mod(nstep,output_interval) == 0) then
123          call output(fdps_ctrl,psys_num,nstep)
124          call check_cnsrvd_vars(fdps_ctrl,psys_num)
125       end if
126
127       !* Output information to STDOUT
128       if (fdps_ctrl%get_rank() == 0) then
129          write(*,200)time,nstep
130          200 format("================================"/ &
131                     "time  = ",1es25.16e3/               &
132                     "nstep = ",i6/                       &
133                     "================================")
134       end if
135
136       !* Termination condition
137       if (time >= end_time) exit
138
139       !* Update time & step
140       time  = time + dt
141       nstep = nstep + 1
142
143    end do
144    call fdps_ctrl%ps_finalize()
145    stop 0
146
147    !* Finalize FDPS
148    call fdps_ctrl%PS_Finalize()
149
150 end subroutine f_main
151
152 !-------------------------------------------------------------------------
153 !////////////////////////// S U B R O U T I N E ////////////////////////////
154 !////////////////////////// < S E T U P _ I C > ////////////////////////////
155 !-------------------------------------------------------------------------
156 subroutine setup_IC(fdps_ctrl,psys_num,end_time,pos_ll,pos_ul)
157    use fdps_vector
158    use fdps_module
159    use user_defined_types
160    implicit none
161    type(fdps_controller), intent(IN) :: fdps_ctrl
162    integer, intent(IN) :: psys_num
163    double precision, intent(inout) :: end_time
164    type(fdps_f64vec) :: pos_ll,pos_ul
165    !* Local variables
166    integer :: i
167    integer :: nprocs,myrank
168    integer :: nptcl_glb
169    double precision :: dens_L,dens_R,eng_L,eng_R
170    double precision :: x,y,z,dx,dy,dz
171    double precision :: dx_tgt,dy_tgt,dz_tgt
172    type(full_particle), dimension(:), pointer :: ptcl
173    character(len=64) :: fname
174
175    !* Get # of MPI processes and rank number
```

```
176     nprocs = fdps_ctrl%get_num_procs()
177     myrank = fdps_ctrl%get_rank()
178
179     !* Set the box size
180     pos_ll%x = 0.0d0
181     pos_ll%y = 0.0d0
182     pos_ll%z = 0.0d0
183     pos_ul%x = 1.0d0
184     pos_ul%y = pos_ul%x / 8.0d0
185     pos_ul%z = pos_ul%x / 8.0d0
186
187     !* Make an initial condition at RANK 0
188     if (myrank == 0) then
189        !* Set the left and right states
190        dens_L = 1.0d0
191        eng_L  = 2.5d0
192        dens_R = 0.5d0
193        eng_R  = 2.5d0
194        !* Set the separation of particle of the left state
195        dx = 1.0d0 / 128.0d0
196        dy = dx
197        dz = dx
198        !* Set the number of local particles
199        nptcl_glb = 0
200        !** (1) Left-half
201        x = 0.0d0
202        do
203           y = 0.0d0
204           do
205              z = 0.0d0
206              do
207                 nptcl_glb = nptcl_glb + 1
208                 z = z + dz
209                 if (z >= pos_ul%z) exit
210              end do
211              y = y + dy
212              if (y >= pos_ul%y) exit
213           end do
214           x = x + dx
215           if (x >= 0.5d0*pos_ul%x) exit
216        end do
217        write(*,*)'nptcl_glb(L)␣␣␣=␣',nptcl_glb
218        !** (2) Right-half
219        x = 0.5d0*pos_ul%x
220        do
221           y = 0.0d0
222           do
223              z = 0.0d0
224              do
225                 nptcl_glb = nptcl_glb + 1
226                 z = z + dz
227                 if (z >= pos_ul%z) exit
228              end do
229              y = y + dy
230              if (y >= pos_ul%y) exit
```

```
231             end do
232             x = x + (dens_L/dens_R)*dx
233             if (x >= pos_ul%x) exit
234          end do
235          write(*,*)'nptcl_glb(L+R) = ',nptcl_glb
236          !* Place SPH particles
237          call fdps_ctrl%set_nptcl_loc(psys_num,nptcl_glb)
238          call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
239          i = 0
240          !** (1) Left-half
241          x = 0.0d0
242          do
243             y = 0.0d0
244             do
245                z = 0.0d0
246                do
247                   i = i + 1
248                   ptcl(i)%id    = i
249                   ptcl(i)%pos%x = x
250                   ptcl(i)%pos%y = y
251                   ptcl(i)%pos%z = z
252                   ptcl(i)%dens  = dens_L
253                   ptcl(i)%eng   = eng_L
254                   z = z + dz
255                   if (z >= pos_ul%z) exit
256                end do
257                y = y + dy
258                if (y >= pos_ul%y) exit
259             end do
260             x = x + dx
261             if (x >= 0.5d0*pos_ul%x) exit
262          end do
263          write(*,*)'nptcl(L)    = ',i
264          !** (2) Right-half
265          x = 0.5d0*pos_ul%x
266          do
267             y = 0.0d0
268             do
269                z = 0.0d0
270                do
271                   i = i + 1
272                   ptcl(i)%id    = i
273                   ptcl(i)%pos%x = x
274                   ptcl(i)%pos%y = y
275                   ptcl(i)%pos%z = z
276                   ptcl(i)%dens  = dens_R
277                   ptcl(i)%eng   = eng_R
278                   z = z + dz
279                   if (z >= pos_ul%z) exit
280                end do
281                y = y + dy
282                if (y >= pos_ul%y) exit
283             end do
284             x = x + (dens_L/dens_R)*dx
285             if (x >= pos_ul%x) exit
```

```
286         end do
287         write(*,*)'nptcl(L+R) = ',i
288         !* Set particle mass and smoothing length
289         do i=1,nptcl_glb
290            ptcl(i)%mass = 0.5d0*(dens_L+dens_R)        &
291                         * (pos_ul%x*pos_ul%y*pos_ul%z) &
292                         / nptcl_glb
293            ptcl(i)%smth = kernel_support_radius * 0.012d0
294         end do
295
296         !* Check the initial distribution
297        !fname = "initial.dat"
298        !open(unit=9,file=trim(fname),action='write',status='replace')
299        !    do i=1,nptcl_glb
300        !       write(9,'(3es25.16e3)')ptcl(i)%pos%x, &
301        !                              ptcl(i)%pos%y, &
302        !                              ptcl(i)%pos%z
303        !    end do
304        !close(unit=9)
305
306      else
307         call fdps_ctrl%set_nptcl_loc(psys_num,0)
308      end if
309
310      !* Set the end time
311      end_time = 0.12d0
312
313      !* Inform to STDOUT
314      if (fdps_ctrl%get_rank() == 0) then
315         write(*,*)"setup..."
316      end if
317    !call fdps_ctrl%ps_finalize()
318    !stop 0
319
320  end subroutine setup_IC
321
322  !------------------------------------------------------------------------
323  !//////////////////////     S U B R O U T I N E     ////////////////////
324  !////////////////////// < G E T _ T I M E S T E P > ////////////////////
325  !------------------------------------------------------------------------
326  function get_timestep(fdps_ctrl,psys_num)
327     use fdps_vector
328     use fdps_module
329     use user_defined_types
330     implicit none
331     real(kind=c_double) :: get_timestep
332     type(fdps_controller), intent(in) :: fdps_ctrl
333     integer, intent(in) :: psys_num
334     !* Local variables
335     integer :: i,nptcl_loc
336     type(full_particle), dimension(:), pointer :: ptcl
337     real(kind=c_double) :: dt_loc
338
339     !* Get # of local particles
340     nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
```

```
341
342     !* Get the pointer to full particle data
343     call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
344     dt_loc = 1.0d30
345     do i=1,nptcl_loc
346        dt_loc = min(dt_loc, ptcl(i)%dt)
347     end do
348     nullify(ptcl)
349
350     !* Reduction
351     call fdps_ctrl%get_min_value(dt_loc,get_timestep)
352
353 end function get_timestep
354
355 !-------------------------------------------------------------------------
356 !//////////////////////     S U B R O U T I N E     ////////////////////
357 !////////////////////// < I N I T I A L _ K I C K > ////////////////////
358 !-------------------------------------------------------------------------
359 subroutine initial_kick(fdps_ctrl,psys_num,dt)
360     use fdps_vector
361     use fdps_module
362     use user_defined_types
363     implicit none
364     type(fdps_controller), intent(in) :: fdps_ctrl
365     integer, intent(in) :: psys_num
366     double precision, intent(in) :: dt
367     !* Local variables
368     integer :: i,nptcl_loc
369     type(full_particle), dimension(:), pointer :: ptcl
370
371     !* Get # of local particles
372     nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
373
374     !* Get the pointer to full particle data
375     call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
376     do i=1,nptcl_loc
377        ptcl(i)%vel_half = ptcl(i)%vel + 0.5d0 * dt * ptcl(i)%acc
378        ptcl(i)%eng_half = ptcl(i)%eng + 0.5d0 * dt * ptcl(i)%eng_dot
379     end do
380     nullify(ptcl)
381
382 end subroutine initial_kick
383
384 !-------------------------------------------------------------------------
385 !////////////////////////   S U B R O U T I N E   ////////////////////////
386 !//////////////////////// < F U L L _ D R I F T > ////////////////////////
387 !-------------------------------------------------------------------------
388 subroutine full_drift(fdps_ctrl,psys_num,dt)
389     use fdps_vector
390     use fdps_module
391     use user_defined_types
392     implicit none
393     type(fdps_controller), intent(in) :: fdps_ctrl
394     integer, intent(in) :: psys_num
395     double precision, intent(in) :: dt
```

```
396     !* Local variables
397     integer :: i,nptcl_loc
398     type(full_particle), dimension(:), pointer :: ptcl
399
400     !* Get # of local particles
401     nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
402
403     !* Get the pointer to full particle data
404     call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
405     do i=1,nptcl_loc
406        ptcl(i)%pos = ptcl(i)%pos + dt * ptcl(i)%vel_half
407     end do
408     nullify(ptcl)
409
410 end subroutine full_drift
411
412 !-------------------------------------------------------------------------
413 !////////////////////////// S U B R O U T I N E ////////////////////////////
414 !//////////////////////////  < P R E D I C T >  ////////////////////////////
415 !-------------------------------------------------------------------------
416 subroutine predict(fdps_ctrl,psys_num,dt)
417     use fdps_vector
418     use fdps_module
419     use user_defined_types
420     implicit none
421     type(fdps_controller), intent(in) :: fdps_ctrl
422     integer, intent(in) :: psys_num
423     double precision, intent(in) :: dt
424     !* Local variables
425     integer :: i,nptcl_loc
426     type(full_particle), dimension(:), pointer :: ptcl
427
428     !* Get # of local particles
429     nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
430
431     !* Get the pointer to full particle data
432     call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
433     do i=1,nptcl_loc
434        ptcl(i)%vel = ptcl(i)%vel + dt * ptcl(i)%acc
435        ptcl(i)%eng = ptcl(i)%eng + dt * ptcl(i)%eng_dot
436     end do
437     nullify(ptcl)
438
439 end subroutine predict
440
441 !-------------------------------------------------------------------------
442 !////////////////////////   S U B R O U T I N E   ////////////////////////
443 !////////////////////////// < F I N A L _ K I C K > ////////////////////////
444 !-------------------------------------------------------------------------
445 subroutine final_kick(fdps_ctrl,psys_num,dt)
446     use fdps_vector
447     use fdps_module
448     use user_defined_types
449     implicit none
450     type(fdps_controller), intent(in) :: fdps_ctrl
```

```
451     integer, intent(in) :: psys_num
452     double precision, intent(in) :: dt
453     !* Local variables
454     integer :: i,nptcl_loc
455     type(full_particle), dimension(:), pointer :: ptcl
456
457     !* Get # of local particles
458     nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
459
460     !* Get the pointer to full particle data
461     call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
462     do i=1,nptcl_loc
463        ptcl(i)%vel = ptcl(i)%vel_half + 0.5d0 * dt * ptcl(i)%acc
464        ptcl(i)%eng = ptcl(i)%eng_half + 0.5d0 * dt * ptcl(i)%eng_dot
465     end do
466     nullify(ptcl)
467
468 end subroutine final_kick
469
470 !-------------------------------------------------------------------------
471 !//////////////////////     S U B R O U T I N E     ////////////////////
472 !////////////////////// < S E T _ P R E S S U R E > ////////////////////
473 !-------------------------------------------------------------------------
474 subroutine set_pressure(fdps_ctrl,psys_num)
475     use fdps_vector
476     use fdps_module
477     use user_defined_types
478     implicit none
479     type(fdps_controller), intent(in) :: fdps_ctrl
480     integer, intent(in) :: psys_num
481     !* Local parameters
482     double precision, parameter :: hcr=1.4d0
483     !* Local variables
484     integer :: i,nptcl_loc
485     type(full_particle), dimension(:), pointer :: ptcl
486
487     !* Get # of local particles
488     nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
489
490     !* Get the pointer to full particle data
491     call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
492     do i=1,nptcl_loc
493        ptcl(i)%pres = (hcr - 1.0d0) * ptcl(i)%dens * ptcl(i)%eng
494        ptcl(i)%snds = dsqrt(hcr * ptcl(i)%pres / ptcl(i)%dens)
495     end do
496     nullify(ptcl)
497
498 end subroutine set_pressure
499
500 !-------------------------------------------------------------------------
501 !////////////////////////// S U B R O U T I N E ////////////////////////
502 !//////////////////////////   < O U T P U T >   ////////////////////////
503 !-------------------------------------------------------------------------
504 subroutine output(fdps_ctrl,psys_num,nstep)
505     use fdps_vector
```

```
506     use fdps_module
507     use user_defined_types
508     implicit none
509     type(fdps_controller), intent(IN) :: fdps_ctrl
510     integer, intent(IN) :: psys_num
511     integer, intent(IN) :: nstep
512     !* Local parameters
513     character(len=16), parameter :: root_dir="result"
514     character(len=16), parameter :: file_prefix_1st="snap"
515     character(len=16), parameter :: file_prefix_2nd="proc"
516     !* Local variables
517     integer :: i,nptcl_loc
518     integer :: myrank
519     character(len=5) :: file_num,proc_num
520     character(len=64) :: cmd,sub_dir,fname
521     type(full_particle), dimension(:), pointer :: ptcl
522
523     !* Get the rank number
524     myrank = fdps_ctrl%get_rank()
525
526     !* Get # of local particles
527     nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
528
529     !* Get the pointer to full particle data
530     call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
531
532     !* Output
533     write(file_num,"(i5.5)")nstep
534     write(proc_num,"(i5.5)")myrank
535     fname =  trim(root_dir) // "/" &
536           // trim(file_prefix_1st) // file_num // "-" &
537           // trim(file_prefix_2nd) // proc_num // ".dat"
538     open(unit=9,file=trim(fname),action='write',status='replace')
539        do i=1,nptcl_loc
540           write(9,100)ptcl(i)%id,ptcl(i)%mass, &
541                       ptcl(i)%pos%x,ptcl(i)%pos%y,ptcl(i)%pos%z, &
542                       ptcl(i)%vel%x,ptcl(i)%vel%y,ptcl(i)%vel%z, &
543                       ptcl(i)%dens,ptcl(i)%eng,ptcl(i)%pres
544           100 format(i8,1x,10e25.16e3)
545        end do
546     close(unit=9)
547     nullify(ptcl)
548
549 end subroutine output
550
551 !-------------------------------------------------------------------------
552 !////////////////          S U B R O U T I N E          ////////////////
553 !//////////////// < C H E C K _ C N S R V D _ V A R S > ////////////////
554 !-------------------------------------------------------------------------
555 subroutine check_cnsrvd_vars(fdps_ctrl,psys_num)
556     use fdps_vector
557     use fdps_module
558     use user_defined_types
559     implicit none
560     type(fdps_controller), intent(in) :: fdps_ctrl
```

```
561     integer, intent(in) :: psys_num
562     !* Local variables
563     integer :: i,nptcl_loc
564     type(full_particle), dimension(:), pointer :: ptcl
565     type(fdps_f64vec) :: mom_loc,mom
566     real(kind=c_double) :: eng_loc,eng
567
568     !* Get # of local particles
569     nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
570
571     !* Get the pointer to full particle data
572     call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
573     mom_loc = 0.0d0; eng_loc = 0.0d0
574     do i=1,nptcl_loc
575        mom_loc = mom_loc + ptcl(i)%vel * ptcl(i)%mass
576        eng_loc = eng_loc + ptcl(i)%mass &
577                            *(ptcl(i)%eng &
578                             +0.5d0*ptcl(i)%vel*ptcl(i)%vel)
579     end do
580     nullify(ptcl)
581
582     !* Reduction & output
583     call fdps_ctrl%get_sum(eng_loc,eng)
584     call fdps_ctrl%get_sum(mom_loc%x,mom%x)
585     call fdps_ctrl%get_sum(mom_loc%y,mom%y)
586     call fdps_ctrl%get_sum(mom_loc%z,mom%z)
587     if (fdps_ctrl%get_rank() == 0) then
588        write(*,100)eng
589        write(*,100)mom%x
590        write(*,100)mom%y
591        write(*,100)mom%z
592        100 format(1es25.16e3)
593     end if
594
595 end subroutine check_cnsrvd_vars
```

6 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.

6.1 Compile-time problem

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

6.2 Run-time problem

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

6.3 Other cases

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

7 License

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

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

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