FDPS Fortran Interface Specification

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

Particle Simulator Research Team, AICS, RIKEN

Contents

Chapter 1

About this document

1.1 Structure of the document

This document is the specification of FDPS (Framework for Developing Particle Simulator) Fortran interface, which supports the development of massively parallel particle simulation codes in Fortran language. This document is written by Daisuke Namekata, Masaki Iwasawa, Ataru Tanikawa, Keigo Nitadori, Takayuki Muranushi, Long Wang, Natsuki Hosono, and Junichiro Makino at RIKEN Advanced Institute for Computational Science.

This document is structured as follows.

In sections 2, 3, and 7, we present prerequisites for programing with FDPS Fortran interface. In section 2, we show the concept of FDPS. In section 3, we present the file configuration of FDPS and FDPS Fortran interface. In section 7, we describe how to compile simulation codes with FDPS.

In sections 4, 5, and 8, we present information to develop simulation codes with FDPS. In section 4, we described derived data types defined in FDPS Fortran interface. In section 5, we introduce user-defined types and user-defined functions required for developing codes with FDPS Fortran interface. In section 8, we describe APIs used to initialize and finalize FDPS. In section 9, we present modules in FDPS and their APIs.

In sections 9 and 10, we present troubleshooting information. In section 9, we describe error messages output by both FDPS and FDPS Fortran interface. In section 10, we describe the limitation of FDPS and FDPS Fortran interface.

Finally, we describe the change log of this document in section 11.

1.2 Lisence

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".

1.3 User support

Please contact us if you have problems in the development of your code using FDPS Fortran interface. The e-mail address is fdps-support<at>mail.jmlab.jp (Please replace `<at>` by @).

In the following cases, please follow our instructions described below.

1.3.1 When you cannot compile your codes

Please give us the following information:

- Information about the compiler and libraries used (e.g. their names and versions, and the compile options used)
- Error messages from the compiler
- Source codes if possible

1.3.2 When your code doesn't run as expected

Please give us the following information:

- Information about the execution environment (e.g. the name and the version of OS)
- Rumtime error messages
- Source codes if possible

1.3.3 Others

Please do not hesitate to contact us if you have a problem concerning optimization or other questions. We sincerely hope that you'll find FDPS useful for your research.

Chapter 2

FDPS

In this chapter, we present the design concept of FDPS: the purpose of its development, the basic concept, and the behavior of codes developed using FDPS.

2.1 The mission of FDPS

In the fields of science and engineering, particle method is used for a wide variety of simulations, such as gravitational N-body simulation, Smoothed Particle Hydrodynamics (SPH) simulation, vortex method, Moving Particle Semi-implicit (MPS) method, molecular dynamics simulation, and so on. We need high-performance particle simulation codes in order to follow physical phenomena with high spatial resolution, and for long timescales.

We cannot avoid parallelization in order to develop high performance particle simulation codes. For the parallelization, we need to implement the following procedures: dynamic domain decomposition for load balancing, exchange of particles between computing nodes, optimization of communication among nodes, effective use of cache memories and SIMD operation units, and support for accelerators. So far, individual research groups were trying to implement these procedures.

However, the above procedures are necessary for any particle simulation codes. The purpose of the development of FDPS is to provide numerical libraries for implementing these procedures, and reduce researchers' and programmers' burdens. We will be happy if researchers and programmers can use their time more creatively by using FDPS.

2.2 Basic concept

In this section, we describe the basic concept of FDPS.

2.2.1 Procedures of massively parallel particle simulations

First, we describe our model of massively parallel particle simulations on FDPS. In particle simulations, the set of ordinary differential equations,

$$\frac{d\boldsymbol{u}_i}{dt} = \sum_j f(\boldsymbol{u}_i, \boldsymbol{u}_j) + \sum_s g(\boldsymbol{u}_i, \boldsymbol{v}_s), \tag{2.1}$$

is numerically integrated, where $\boldsymbol{u}_i$ is the quantity vector of i-particle. This vector includes quantities of particle i, such as mass, position, and velocity. The function f specifies a force exerted by particle j on particle i. Hereafter, a particle receiving a force is called i-particle, and a particle exerting a force is called j-particle. The vector $\boldsymbol{v}_s$ is the quantity vector of a superparticle which represents a group of particles that are distant from i-particle. The function g specifies a force exerted by a superparticle on a particle. The second term in the left hand side of eq. (2.1) is a non-zero quantity in the case of long-range forces (e.g. gravity and Coulomb force), while it is zero in the case of short-range forces (e.g. pressure of fluid).

Massively parallel simulation codes integrate the above eq. (2.1) by taking the following steps (initialization and data I/O are omitted).

1. In the following two steps, we determine which MPI process handles which particles.
   (a) Decompose the whole domain into subdomains, and determine which MPI process handles which subdomains, in order to balance the calculation cost (domain decomposition).
   (b) MPI processes exchange their particles in order for each MPI process to have particles in its subdomain.
2. Each MPI process gathers quantity vectors of j-particles ($\boldsymbol{u}_j$) and superparticles ($\boldsymbol{v}_s$) required to calculate forces exerted on i-th i-particle (making interaction lists).
3. Each MPI process calculates the right hand of eq. (2.1) for all of its i-particle and obtains $d\boldsymbol{u}_i/dt$.
4. Each MPI process performs the time integration of its i-particles by using the quantity vectors of $\boldsymbol{u}_i$ and their derivatives $d\boldsymbol{u}_i/dt$.
5. Return to step 1.

2.2.2 Division of tasks between users and FDPS

FDPS handles tasks related to interaction calculation and efficient parallelization and the user-written code performs the rest. The actual function for interaction calculation is supplied by users. Thus, FDPS deals with domain decomposition and exchange of particles (step 1), and making interaction lists. On the other hand, the user code is responsible for actual calculation of forces (step 3), and time integration (step 4). Users can avoid the development of complicated codes necessary for realizing massively parallel program, by utilizing FDPS APIs.

2.2.3 Users' tasks

Users's tasks are as follows.

- Define a particle (Chap. 5). Users need to specify quantities of particles, *i.e.* the quantity vector $\boldsymbol{u}_i$ in eq. (2.1), which contains quantities such as position, velocity, acceleration, chemical composition, and particle size.

- Define interaction (Chap. 5). Users need to specify the interaction between particles, *i.e.* the function f and g in eq. (2.1), such as gravity, Coulomb force, and pressure.
- Call FDPS APIs (Chap. 8).
- Time integration of particles, diagnostic, output etc.

2.2.4 Complement

The right hand side of eq. (2.1), the particle-particle interactions, is strictly of two-body nature. FDPS APIs can not be used to implement three-particle interactions. However, for example, FDPS has APIs to return neighbor lists. Users can calculate three- or more-body interactions, using these neighbor lists.

Calculation steps in section 2.2.1 imply that all particles have one same timestep. FDPS APIs do not support individual timestep scheme. However, users can develop a particle simulation code with individual timestep scheme, using the Particle-Particle Particle Tree method.

2.3 The structure of a simulation code with FDPS

In this section, we first overview the structure of a simple simulation code written using FDPS, not FDPS Fortran interface. Then, we describe the structure of a code written using FDPS Fortran interface.

2.3.1 FDPS

In a code with FDPS, three FDPS-supplied classes and several user-defined classes are used.

- DomainInfo class. This class contains the information of all the subdomains, and APIs for domain decomposition.
- ParticleSystem class. This class contains the information of all particles in each MPI process, and APIs for the exchange of particles among MPI processes.
- TreeForForce class. This class contains tree structure made from particle distribution, and APIs for making interaction lists.
- User-defined classes. These classes include the definitions of particles and interactions.

These classes communicate with each other. This is illustrated in Fig. 2.1. The communication in this figure corresponds to steps 1 and 2, and to initialization (step 0).

0. Users give a user-defined particle class to ParticleSystem class, and a function object to TreeForForce class. These are not class inheritance. The particle class is used as a template argument of ParticleSystem class, and the function object is used as an argument of APIs in TreeForForce class.
1. Do load balancing in the following two steps.

(a) The user code calls APIs for domain decomposition in DomainInfo class. Particle information is transferred from ParticleSystem class to DomainInfo class (red text and arrows).

(b) The user code calls APIs for exchange of particles in ParticleSystem class. Information of subdomains is transferred from DomainInfo class to ParticleSystem class (blue text and arrows).

2. Do the force calculation in the following steps.

   (a) The user code calls force calculation API.

   (b) FDPS makes interaction lists in TreeForForce class. Information of subdomains and particles is transferrd from DomainInfo and ParticleSystem classes (green text and arrows).

3. FDPS calls an user-defined function object. This API is included in TreeForForce class. Interactions are calculated, and the results are transferred from TreeForForce class to ParticleSystem class (gray text and arrows).

2.3.2 FDPS Fortran interface

All the APIs described in the previous section are implemented in FDPS Fortran interface. Therefore, the structure of Fortran code is similar to that of C++ code. The complete list of the APIs in FDPS Fortran interface is given in Chap. 8.

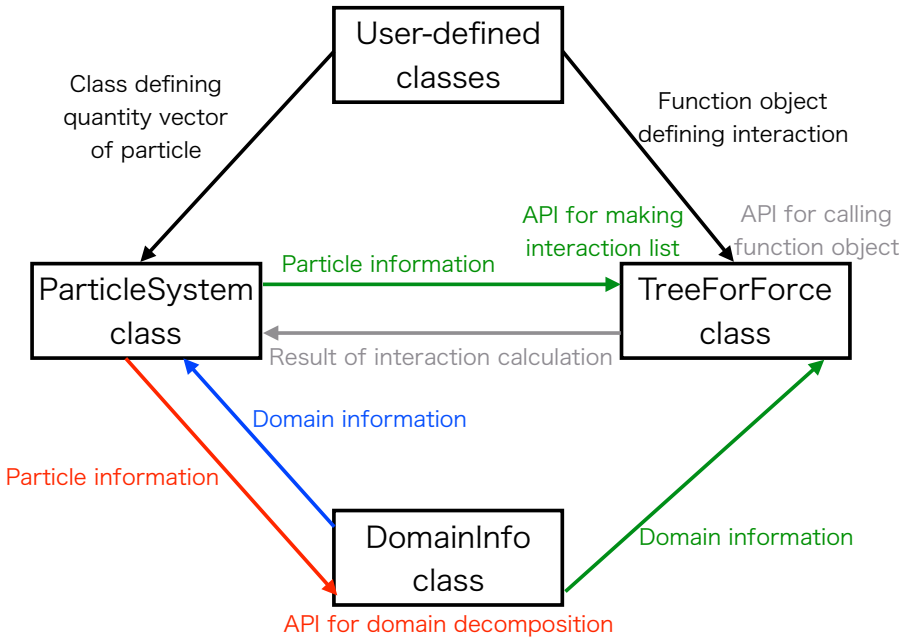

Figure 2.1: Illustration of module interface and data transfer.

Chapter 3

Fortran Interface

In this chapter, we first describe the file structure of FDPS Fortran interface and overview the way of code development using it. Then, we explain documents and sample codes related to FDPS Fortran interface.

3.1 File structure and overview

3.1.1 FDPS

Source files of FDPS are in the directory `src`. FDPS is written in C++ and all the source files related to the standard features are directly under the directory `src`. FDPS has extended features and, at present, there are two extended features: "Particle Mesh" and "Phantom-GRAPE for x86". The source files related to these features are in the directories `src/particle_mesh` and `src/phantom_GRAPE_x86`, respectively. These features are used from FDPS as external static libraries. Therefore, users have to build the static libraries when using the extended features. For more details, please see the specification document of FDPS itself (`doc/doc_specs_cpp_en.pdf`).

3.1.2 Fortran interface

Among the features described in previous section, the standard features (except for some low-level APIs) and the extended feature "Particle Mesh" are available in Fortran. Users can use these features via Fortran interface programs, which are manually created by the users by executing the script `gen_ftn_if.py` in the directory `scripts` (see Chap. 6 for the details of the script). This interface-generating script analyzes **user-defined types**, which are derived data types that the users must define when using FDPS Fortran interface (see Chap. 5), and generate the Fortran interface programs. Thus, **the first thing the users need to do is to implement the user-defined types**. The reason why the interface programs need to be generated is explained in § 3.1.4. Fortran source codes that are required to implement the user-defined types are in the directory `src/fortran_interface/modules` and files that are used as blueprint when generating the interface programs are in the directory `src/fortran_interface/blueprints`.

Figure 3.1 shows the file structure of the Fortran interface programs and their roles. Four

files enclosed by the dashed line (`FDPS_module.F90`, `FDPS_ftn_if.cpp`, `FDPS_Manipulators.cpp`, `main.cpp`) are the files to be generated by the script, and the file `f_main.F90` corresponds to the user's source codes. In the files enclosed by the dotted line (`FDPS_vector.F90`、`FDPS_matrix.F90`、`FDPS_super_particle.F90`, etc.), several derived data types are defined, which are needed to define the user-defined types and user-defined functions (see Chap. 2). In the following, we explain the role of each interface program.

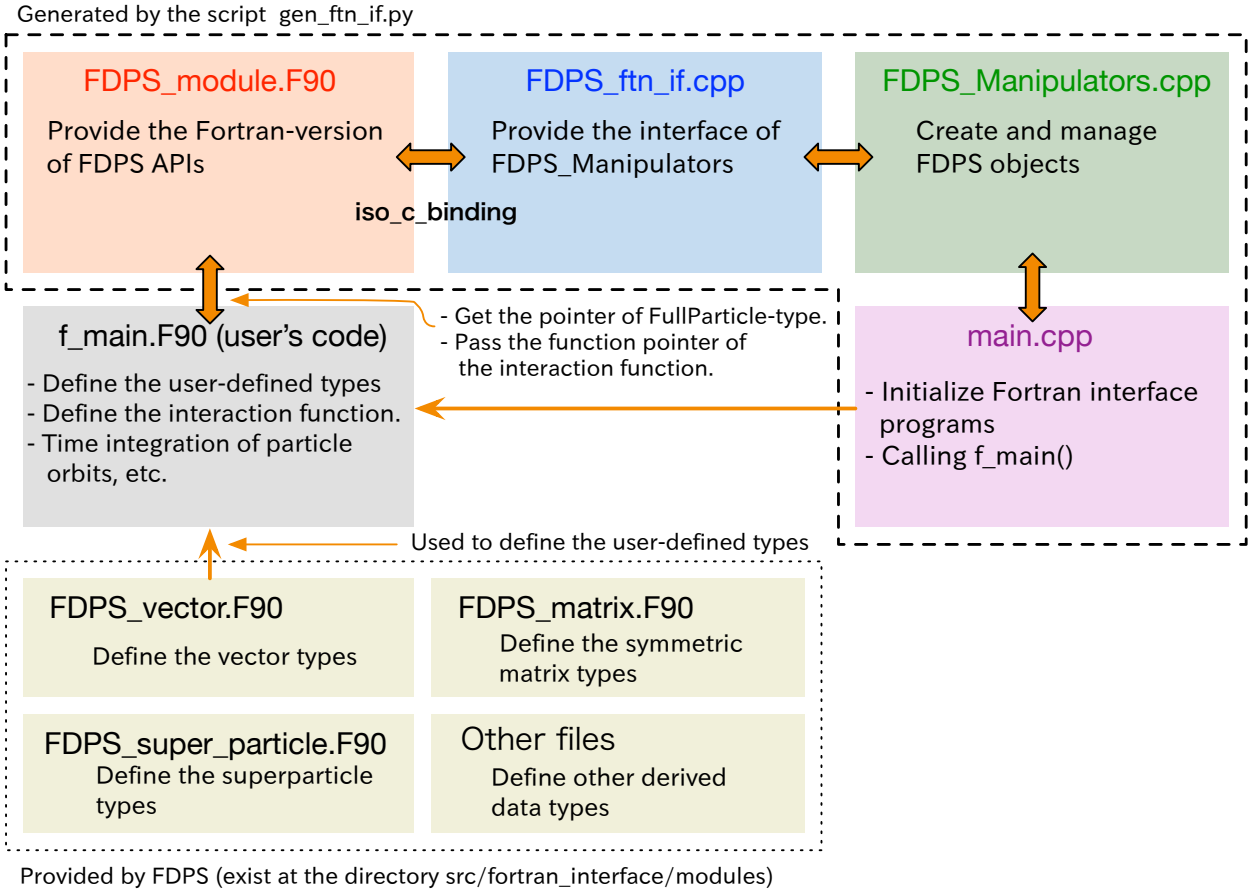

Figure 3.1: File structure of FDPS Fortran interface programs and its relation to user's code

At first, we explain the roles of `FDPS_Manipulators.cpp` and `main.cpp`. Because FDPS is written in C++, all of C++ objects of DomainInfo class, ParticleSystem class, and TreeForForce class described in Chap. 2 must be created and be managed in C++ codes. This task is performed by `FDPS_Manipulators.cpp`. By the same reason, we must place the `main` function of the user's code in C++ files. Thus, `main.cpp` is generated. It calls a Fortran subroutine named `f_main()`. Users should prepare a Fortran subroutine `f_main()` and must implement all parts of the simulation code inside `f_main()`. As described in Chap. 8, all of C++ objects created in `FDPS_Manipulators.cpp` are assigned to Fortran's integer variables. Hence, users also need to manage these objects using integer variables.

Next, we explain the role of `FDPS_ftn_if.cpp`. Fortran programs cannot directly call C++ functions. However, a new feature introduced by Fortran 2003 standard (the feature provided by the Fortran module `iso_c_binding`) makes it possible that Fortran programs directly call C functions. So, under FDPS Fortran interface, manipulation of FDPS is

performed by calling the C interfaces of the functions defined in `FDPS_Manipulators.cpp` from a Fortran program. These C interface functions are implemented in `FDPS_ftn_if.cpp`.

Finally, we explain the role of `FDPS_module.F90`. `FDPS_module.F90` provides a derived data type `FDPS_controller` for users. This data type is used to call the C interface functions described above. `FDPS_controller` is actually a class in Fortran 2003 (a derived data type having member functions or member subroutines) and its member functions provide Fortran interface to FDPS. The list of member functions or Fortran interface is given in Chap. 8. The class `FDPS_controller` is defined in the file `FDPS_module.F90` as follows (Listing 3.1):

Listing 3.1: The structure of `FDPS_module.F90`

```
1  module FDPS_module
2     use, intrinsic :: iso_c_binding
3     implicit none
4
5     !**** FDPS controller
6     type, public :: FDPS_controller
7     contains
8        !
9        ! APIs are defined here.
10       !
11    end type FDPS_controller
12
13 end module FDPS_module
```

where we have omitted the declaration and implementation parts of the member functions for brevity. Actually, the declaration statements are described in a region between the strings `contains` and `end type FDPS_controller`. For this reason, users should use Fortran interface in the following procedure:

(1) Make accessible to the module `FDPS_module` by using the `use` statement.
(2) Create a object of the class `FDPS_controller`.
(3) Call a member function of the created `FDPS_controller` object.

A simplest example of user's code is shown in Listing 3.2:

Listing 3.2: A usage example of Fortran interface

```
1  subroutine f_main()
2     use FDPS_module ! Step (1)
3     implicit none
4     type(FDPS_controller) :: fdps_ctrl ! Step (2)
5
6     ! Call Fortran interface
7     call fdps_ctrl%PS_initialize() ! Step (3)
8
9  end subroutine f_main
```

where numbers shown in the comments correspond to the numbers of the procedure described above.

3.1.3 Code development flow with Fortran interface

In this section, we explain a flow of code development with FDPS Fortran interface. The following is a rough summary of the flow:

[1] **Define the user-defined types**
Implement the user-defined types to generate the interface programs as described in the previous section. The user-defined types must be implemented as Fortran's derived data types. The way to describe the user-defined types is explained in Chap. 5 in detail.

[2] **Generate the interface programs**
After implementing the user-defined types, generate the Fortran interface programs by executing the interface-generating script `gen_ftn_if.py`. If the generation is successfully completed, users can use Fortran APIs to FDPS in user's code. The system requirement of the script and its usage are explained in Chap. 6.

[3] **Define the user-defined functions**
Implement Fortran subroutine(s) that describe the details of interaction (the user-defined functions). The way to describe the user-defined functions is explained in Chap. 5.

[4] **Write the main part of user's code**
Write the main part of a particle simulation code using the user-defined types, the user-defined functions, and Fortran APIs to FDPS. Be aware the following points:

- Users' code must start and end within a Fortran subroutine `f_main()`.
- The Fortran APIs to FDPS are provided as the member functions of the Fortran class `FDPS_controller`. Hence, users must call the member functions to use the APIs.

For concrete examples of code written by using the Fortran interface, please see our sample codes placed in the directory `doc/sample/fortran` (see also Chap. 3.3).

[5] **Compile**
After completing the implementation of user's codes, compile the user's codes to obtain an executable file. As described in the previous sections, the Fortran interface consists of source codes written in C++ and Fortran. Therefore, a bit different way of compilation is needed when compared to the case that all the codes are written in a single language. For more information, see Chap. 7. In FDPS, some of the features should be specified by defining preprocessor macros at the compilation time. As for this, we explain this in Chap. 7. When using the extended feature "Particle Mesh", users must install libraries required by it and must use appropriate compilation options.

[6] **Execute**
Run the executable file according to the rule of computer system that a user uses.

3.1.4 The need for the generation of interface programs

As described in the previous sections, FDPS Fortran interface (Fortran APIs to FDPS) are provided as the form of source codes generated based on user's codes, not as libraries. In this section, we explain the need for generating the interface programs in detail.

As a preparation, we first overview the usage of FDPS in C++ codes. As described in Chap. 2 § 2.2.3, FDPS allows users to define a particle or an interaction freely. This feature allows FDPS to be applicable for various kinds of particle simulations. In order to realize

this feature, FDPS is implemented by using C++ templates. Roughly speaking, template is a feature of C++ that allows a function to receive data types as actual arguments in addition to usual arguments. Using templates, we can define functions using dummy data types in C++ (a dummy data type does not cause a problem if it is replaced by an actual data type at the compilation time). Also, FDPS is provided as the form of header file. Therefore, when a user uses FDPS in C++, a user first includes the header file of FDPS in the user's code and calls FDPS APIs with passing the particle class(es) defined by the user as the template arguments. During the compilation of the user's code, all the data types used in FDPS APIs are completely determined. Therefore a compiler can compile the code without any problem.

Fortran does not have a feature corresponding to templates in C++ and therefore we cannot implement functions or subroutines using dummy data types in Fortran. This is one reason why we cannot provide FDPS Fortran interface as a library. In order to allow users to define particles and interactions freely in Fortran, we adopt a system that generates Fortran interface programs based on the analysis of derived data types of particles that are implemented by users.

Another reason comes from the fact that we internally use FDPS written in C++ directly. In order to transfer data between FDPS written in C++ and Fortran programs, we need to prepare particle class(es) in the C++ side that are equivalent to the particle types defined by the user in Fortran. This also requires the generation of a C++ source code by analyzing derived data types defined by a user.

For the reasons stated above, FDPS Fortran interface is provided as the form of the generated source codes.

3.2 Documents

All the documents related to FDPS and FDPS Fortran interface are in the directory `doc`. `doc_tutorial_ftn_en.pdf` explains basic usage of FDPS Fortran interface with using sample codes (not yet available). `doc_specs_ftn_en.pdf` (this document) describes the specification of FDPS Fortran interface.

3.3 Sample codes

Sample codes are in the directory `sample/fortran`. At present, there are three sample codes: (1) Collision-less gravitational N-body simulation code (`sample/fortran/nbody`), (2) SPH (Smoothed Particle Hydrodynamics) code with fixed smoothing length (`sample/fortran/sph`), and (3) a sample code of P^3M (Particle-Particle Particle-Mesh) method (`sample/fortran/p3m`).

Chapter 4

Derived Data Types in FDPS

FDPS Fortran interface defines several derived data types including vector types, symmetric matrix types, superparticle types, time profile type, enum types. These data types are required to be used in user-defined types and user-defined functions (Chap. 5) or are used as returned values in some APIs (Chap. 8).

4.1 Vector types

FDPS Fortran interface defines two types of vector: `fdps_f32vec` and `fdps_f64vec`. They are defined in the file `src/fortran_interface/modules/FDPS_vector.F90` as follows (Listing 4.1). Each vector has 2 or 3 member variables depending on the spatial dimension of the simulation. By default, the spatial dimension of the simulation is assumed to be 3, but it is 2 if the macro `PARTICLE_SIMULATOR_TWO_DIMENSION` is defined at the compilation. The data type of the member variables is either 32-bit or 64-bit floating point numbers.

Listing 4.1: Vector types

```
1  module fdps_vector
2     use, intrinsic :: iso_c_binding
3     implicit none
4
5     type, public, bind(c) :: fdps_f32vec
6  #ifdef PARTICLE_SIMULATOR_TWO_DIMENSION
7        real(kind=c_float) :: x,y
8  #else
9        real(kind=c_float) :: x,y,z
10 #endif
11    end type fdps_f32vec
12
13    type, public, bind(c) :: fdps_f64vec
14 #ifdef PARTICLE_SIMULATOR_TWO_DIMENSION
15       real(kind=c_double) :: x,y
16 #else
17       real(kind=c_double) :: x,y,z
18 #endif
19    end type fdps_f64vec
20
21 end module fdps_vector
```

For the vector types, the definitions of the assignment (=) and the arithmetic operators (+,-,*,/) are extended as shown in Table. 4.1. For more details, see `FDPS_vector.F90`.

Symbol	Left-hand side	Right-hand side	Definition
=	vector	scalar†	Assign RHS to LHS♯. When RHS is scalar, it is assigned to all the components of LHS. When RHS is an array, each component of LHS is assigned by array element according to the memory ordering.
	vector	array of scalars‡	
	vector	vector	
+	vector	array of scalars	Addition of LHS and RHS. When one of the operands is an array, it is assumed that each element of the array corresponds to the component of the vector according to the memory ordering.
	array of scalars	vector	
	vector	vector	
	none	vector	Do nothing
-	vector	array of scalars	Subtraction RHS from LHS. When one of the operands is an array, it is assumed that each element of the array corresponds to the component of the vector according to the memory ordering.
	array of scalars	vector	
	vector	vector	
	none	vector	Inversion of the sign of the vector
*	vector	scalar	Scalar-vector product
	scalar	vector	
	vector	array of scalars	Inner product. When one of the operands is an array, it is assumed that each element of the array corresponds to the component of the vector according to the memory ordering.
	array of scalars	vector	
	vector	vector	
/	vector	scalar	Division LHS by RHS.

† The data type of scalar must be one of intrinsic data types in Fortran and be numeric.
‡ The size of array must be 2 if the macro `PARTICLE_SIMULATOR_TWO_DIMENSION` is defined at the compilation. Otherwise, it must be 3.
♯ LHS and RHS stand for the left- and the right-hand sides, respectively.

Table 4.1: The definitions of the assignment and the arithmetic operators extended for the vector types

4.2 Symmetric Matrix types

There are two types of symmetric matrix: `fdps_f32mat` and `fdps_f64mat`. These are defined in the file `src/fortran_interface/modules/FDPS_matrix.F90` as follows (Listing 4.2). Each symmetric matrix type has 3 or 6 member variables depending on the spatial dimension of the simulation. By default, the spatial dimension of the simulation is assumed to be 3, but it is 2 if the macro `PARTICLE_SIMULATOR_TWO_DIMENSION` is defined at the

compilation. The data type of the member variables is either 32-bit or 64-bit floating point numbers.

Listing 4.2: Symmetric Matrix types

```
1  module fdps_matrix
2     use, intrinsic :: iso_c_binding
3     implicit none
4
5     !**** PS::F32mat
6     type, public, bind(c) :: fdps_f32mat
7  #ifndef PARTICLE_SIMULATOR_TWO_DIMENSION
8        real(kind=c_float) :: xx,yy,zz,xy,xz,yz
9  #else
10       real(kind=c_float) :: xx,yy,xy
11 #endif
12    end type fdps_f32mat
13
14    !**** PS::F64mat
15    type, public, bind(c) :: fdps_f64mat
16 #ifndef PARTICLE_SIMULATOR_TWO_DIMENSION
17       real(kind=c_double) :: xx,yy,zz,xy,xz,yz
18 #else
19       real(kind=c_double) :: xx,yy,xy
20 #endif
21    end type fdps_f64mat
22
23 end module fdps_matrix
```

For the symmetric matrix types, the definitions of the assignment (=) and the arithmetic operators (+,-,*,-) are extended as shown in Table. 4.2. For more details, see `FDPS_matrix.F90`.

Symbol	Left-hand side	Right-hand side	Definition
=	matrix	scalar†	Assign RHS to LHS. When RHS is scalar, it is assigned to all the components of LHS.
	matrix	matrix	
+	matrix	matrix	Addition of LHS and RHS.
	none	matrix	Do nothing
-	matrix	matrix	Subtraction RHS from LHS.
	none	matrix	Sign inversion of all the components of the matrix
*	matrix	scalar	Scalar-matrix product
	scalar	matrix	
	matrix	matrix	Matrix product
/	matrix	scalar	Division LHS by RHS.

† The data type of scalar must be one of intrinsic data types in Fortran and be numeric.

Table 4.2: The definitions of the assignment and the arithmetic operators extended for the symmetric matrix types

4.3 Superparticle types

A superparticle is a virtual particle that represents a group of real particles, which are located far from the i-particle (a particle which we want to calculate the force acting on). Its properties are computed from the properties of these real particles via some way. Superparticle types are derived data types required to describe the interaction between a particle and a superparticle. They are used in user-defined functions to receive data of superparticles.

The superparticle types include `fdps_spj_monopole`,`fdps_spj_quadrupole`,`fdps_spj_monopole_geomcen`,`fdps_spj_dipole_geomcen`,`fdps_spj_quadrupole_geomcen`,`fdps_spj_monopole_scatter`,`fdps_spj_quadrupole_scatter`,`fdps_spj_monopole_cutoff`. They are defined in the file `src/fortran_interface/modules/FDPS_super_particle.F90` as follows (Listing 4.3). Note that the vector types and the symmetric matrix types are used to define the superparticle types.

Listing 4.3: Superparticle types

```
1  module fdps_super_particle
2     use, intrinsic :: iso_c_binding
3     use fdps_vector
4     use fdps_matrix
5     implicit none
6
7     !**** PS::SPJMonopole
8     type, public, bind(c) :: fdps_spj_monopole
9        real(kind=c_double) :: mass
10       type(fdps_f64vec) :: pos
11    end type fdps_spj_monopole
```

```
12
13    !**** PS::SPJQuadrupole
14    type, public, bind(c) :: fdps_spj_quadrupole
15       real(kind=c_double) :: mass
16       type(fdps_f64vec)  :: pos
17       type(fdps_f64mat)  :: quad
18    end type fdps_spj_quadrupole
19
20    !**** PS::SPJMonopoleGeometricCenter
21    type, public, bind(c) :: fdps_spj_monopole_geomcen
22       integer(kind=c_long_long) :: n_ptcl
23       real(kind=c_double) :: charge
24       type(fdps_f64vec) :: pos
25    end type fdps_spj_monopole_geomcen
26
27    !**** PS::SPJDipoleGeometricCenter
28    type, public, bind(c) :: fdps_spj_dipole_geomcen
29       integer(kind=c_long_long) :: n_ptcl
30       real(kind=c_double) :: charge
31       type(fdps_f64vec) :: pos
32       type(fdps_f64vec) :: dipole
33    end type fdps_spj_dipole_geomcen
34
35    !**** PS::SPJQuadrupoleGeometricCenter
36    type, public, bind(c) :: fdps_spj_quadrupole_geomcen
37       integer(kind=c_long_long) :: n_ptcl
38       real(kind=c_double) :: charge
39       type(fdps_f64vec) :: pos
40       type(fdps_f64vec) :: dipole
41       type(fdps_f64mat) :: quadrupole
42    end type fdps_spj_quadrupole_geomcen
43
44    !**** PS::SPJMonopoleScatter
45    type, public, bind(c) :: fdps_spj_monopole_scatter
46       real(kind=c_double) :: mass
47       type(fdps_f64vec) :: pos
48    end type fdps_spj_monopole_scatter
49
50    !**** PS::SPJQuadrupoleScatter
51    type, public, bind(c) :: fdps_spj_quadrupole_scatter
52       real(kind=c_double) :: mass
53       type(fdps_f64vec)  :: pos
54       type(fdps_f64mat)  :: quad
55    end type fdps_spj_quadrupole_scatter
56
57    !**** PS::SPJMonopoleCutoff
58    type, public, bind(c) :: fdps_spj_monopole_cutoff
59       real(kind=c_double) :: mass
60       type(fdps_f64vec) :: pos
61    end type fdps_spj_monopole_cutoff
62
63    ! [TODO]
64    !    PS::SPJMonopolePeriodic
65    !    PS::SPJMonopoleCutoffScatter
66
```

```
67 end module fdps_super_particle
```

There is an one-to-one correspondence between the superparticle types and the types of tree objects. Accordingly, users must use an appropriate superparticle type corresponding to the type of created tree object. The correspondence relationship between the types of tree objects and the superparticle types is shown in Table 4.3. Note that the types of tree object for short-range force are not shown in this table, because superparticle is only used in the case of long-range force. For the other types of tree object and the way to create tree objects, see Chap. 8 § 8.4.

Tree type	Highest-order of multipole moments†	Range of interaction	Superparticle type
Long-Monopole type	monopole(CoM)	Entire region	`fdps_spj_monopole`
Long-Quadrupole type	quadrupole(CoM)	Entire region	`fdps_spj_quadrupole`
Long-MonopoleGeometricCenter type	monopole(GC)	Entire region	`fdps_spj_monopole_geomcen`
Long-DipoleGeometricCenter type	dipole(GC)	Entire region	`fdps_spj_dipole_geomcen`
Long-QuadrupoleGeometricCenter type	quadrupole(GC)	Entire region	`fdps_spj_quadrupole_geomcen`
Long-MonopoleWithScatterSearch type‡	monopole(CoM)	Entire region	`fdps_spj_monopole_scatter`
Long-QuadrupoleWithScatterSearch type‡	quadrupole(CoM)	Entire region	`fdps_spj_quadrupole_scatter`
Long-MonopoleWithCutoff type	monopole(CoM)	Within cutoff radius	`fdps_spj_monopole_cutoff`

† CoM indicates that multipole moments are calculated assuming that the center-of-mass is the center of the expansion. GC indicates that multipole moments are calculated assuming that the geometric center is the center of the expansion.
‡ Scatter-mode neighbor search is possible.

Table 4.3: The correspondence relationship between the types of tree for long-range force and the superparticle types

4.4 Time profile types

Time profile types are used to obtain the elapsed times of various types of calculations performed in FDPS. At present, there is only one time profile type `fdps_time_profile`. It is defined in the file `src/fortran_interface/modules/FDPS_time_profile.F90` as follows (Listing 4.4). This derived data type is exclusively used by the APIs for time measurement (for detail, see Chap. 8).

Listing 4.4: Time profile types

```
1  module fdps_time_profile
2     use, intrinsic :: iso_c_binding
3     implicit none
4
5     !**** PS::TimeProfile
6     type, public, bind(c) :: fdps_time_prof
7        real(kind=c_double) :: collect_sample_particle
8        real(kind=c_double) :: decompose_domain
9        real(kind=c_double) :: exchange_particle
10       real(kind=c_double) :: set_particle_local_tree
11       real(kind=c_double) :: set_particle_global_tree
12       real(kind=c_double) :: make_local_tree
13       real(kind=c_double) :: make_global_tree
14       real(kind=c_double) :: set_root_cell
15       real(kind=c_double) :: calc_force
16       real(kind=c_double) :: calc_moment_local_tree
17       real(kind=c_double) :: calc_moment_global_tree
18       real(kind=c_double) :: make_LET_1st
19       real(kind=c_double) :: make_LET_2nd
20       real(kind=c_double) :: exchange_LET_1st
21       real(kind=c_double) :: exchange_LET_2nd
22
23       real(kind=c_double) :: morton_sort_local_tree
24       real(kind=c_double) :: link_cell_local_tree
25       real(kind=c_double) :: morton_sort_global_tree
26       real(kind=c_double) :: link_cell_global_tree
27
28       real(kind=c_double) :: make_local_tree_tot
29       ! = make_local_tree + calc_moment_local_tree
30       real(kind=c_double) :: make_global_tree_tot
31       real(kind=c_double) :: exchange_LET_tot
32       ! = make_LET_1st + make_LET_2nd + exchange_LET_1st +
               exchange_LET_2nd
33
34       real(kind=c_double) :: calc_force__core__walk_tree
35
36       real(kind=c_double) :: calc_force__make_ipgroup
37       real(kind=c_double) :: calc_force__core
38       real(kind=c_double) :: calc_force__copy_original_order
39
40       real(kind=c_double) :: exchange_particle__find_particle
41       real(kind=c_double) :: exchange_particle__exchange_particle
42
43       real(kind=c_double) :: decompose_domain__sort_particle_1st
44       real(kind=c_double) :: decompose_domain__sort_particle_2nd
```

```
45        real(kind=c_double) :: decompose_domain__sort_particle_3rd
46        real(kind=c_double) :: decompose_domain__gather_particle
47
48        real(kind=c_double) :: decompose_domain__setup
49        real(kind=c_double) :: decompose_domain__determine_coord_1st
50        real(kind=c_double) :: decompose_domain__migrae_particle_1st
51        real(kind=c_double) :: decompose_domain__determine_coord_2nd
52        real(kind=c_double) :: decompose_domain__determine_coord_3rd
53        real(kind=c_double) :: decompose_domain__exchange_pos_domain
54
55        real(kind=c_double) :: exchange_LET_1st__a2a_n
56        real(kind=c_double) :: exchange_LET_1st__icomm_sp
57        real(kind=c_double) :: exchange_LET_1st__a2a_sp
58        real(kind=c_double) :: exchange_LET_1st__icomm_ep
59        real(kind=c_double) :: exchange_LET_1st__a2a_ep
60    end type fdps_time_prof
61
62 end module fdps_time_profile
```

4.5 Enumerated types

In this section, we describe enumerated types defined in FDPS Fortran interface.

4.5.1 Boundary condition types

Boundary condition types are used in the API `set_boundary_condition` to specify the boundary condition of simulation (see § 8.3 "APIs for DomainInfo object" in Chap. 8). It is defined in the file `FDPS_module.F90` as follows.

Listing 4.5: Boundary condition types

```
1  module FDPS_module
2     use, intrinsic :: iso_c_binding
3     implicit none
4
5     !* Enum types
6     !**** PS::BOUNDARY_CONDITION
7     enum, bind(c)
8        enumerator :: fdps_bc_open
9        enumerator :: fdps_bc_periodic_x
10       enumerator :: fdps_bc_periodic_y
11       enumerator :: fdps_bc_periodic_z
12       enumerator :: fdps_bc_periodic_xy
13       enumerator :: fdps_bc_periodic_xz
14       enumerator :: fdps_bc_periodic_yz
15       enumerator :: fdps_bc_periodic_xyz
16       enumerator :: fdps_bc_shearing_box
17       enumerator :: fdps_bc_user_defined
18    end enum
19
20 end module FDPS_module
```

Table 4.4 shows the relations between the enumerators of the boundary condition types and boundary conditions.

Enumerator	Boundary condition
`fdps_bc_open`	The open boundary condition. **This is the default boundary condition in FDPS**.
`fdps_bc_periodic_x`	The periodic boundary condition in the direction of x-axis, and the open boundary condition in other directions. FDPS assume that the lower and upper boundaries of the computational box (or the interval) along x-axis are closed and open, respectively (i.e. `[)`). This is assumed for all periodic boundary conditions.
`fdps_bc_periodic_y`	The periodic boundary condition in the direction of y-axis, and the open boundary condition in other directions.
`fdps_bc_periodic_z`	The periodic boundary condition in the direction of z-axis, and the open boundary condition in other directions.
`fdps_bc_periodic_xy`	The periodic boundary condition in the directions of x- and y-axes, and the open boundary condition in the direction of z-axis.
`fdps_bc_periodic_xz`	The periodic boundary condition in the directions of x- and z-axes, and the open boundary condition in the direction of y-axis.
`fdps_bc_periodic_yz`	The periodic boundary condition in the directions of y- and z-axes, and the open boundary condition in the direction of x-axis.
`fdps_bc_periodic_xyz`	The periodic boundary condition in all three directions.
`fdps_bc_shearing_box`	The shearing-box boundary condition (Not implemented yet).
`fdps_bc_user_defined`	User-defined boundary condition (Not implemented yet).

Table 4.4: The correspondence relation between the enumerator of the boundary condition types and the boundary conditions

4.5.2 Interaction list mode types

Interaction list mode types are used to determine whether we reuse interaction lists at interaction calculations. These types are used as an argument in the APIs `calc_force_all_and_write_back` and `calc_force_all` (see § 8.4 ‘APIs for Tree object‘” in Chap. 8) and are defined in the file `FDPS_module.F90` as follows.

Listing 4.6: Interaction list mode types

```
1 module FDPS_module
2    use, intrinsic :: iso_c_binding
3    implicit none
4
```

```
5     !* Enum types
6     !**** PS::INTERACTION_LIST_MODE
7     enum, bind(c)
8        enumerator :: fdps_make_list
9        enumerator :: fdps_make_list_for_reuse
10       enumerator :: fdps_reuse_list
11    end enum
12
13 end module FDPS_module
```

Table 4.5 shows the relations between the enumerators of the interaction list mode types and operation mode of the APIs described above.

Enumerator	Operation mode
`fdps_make_list`	FDPS (re)makes interaction lists for each interaction calculation (each call of the APIs described above). In this case, we cannot reuse interaction list in the next interaction calculation because FDPS does not store the information of interaction list. **This is the default operation mode in FDPS**.
`fdps_make_list_for_reuse`	FDPS (re)makes interaction lists and stores them internally. Then, it performs interaction calculation. In this case, we can reuse these interaction lists in the next interaction calculation if we call the APIs with the flag `fdps_reuse_list`. The interaction lists memorized in FDPS are destroyed if we perform the interaction calculation with the flags `fdps_make_list_for_reuse` or `fdps_make_list`.
`fdps_reuse_list`	FDPS performs interaction calculation using the previously-created interaction lists, which are the lists that are created at the previous call of the APIs with the flag `fdps_make_list_for_reuse`. In this case, moment information in superparticles are automatically updated using the latest particle information.

Table 4.5: The correspondence relation between the enumerator of the interaction list mode types and the operation modes

Chapter 5

User-defined Types and User-defined Functions

In this chapter, we describe the details of **user-defined types** (derived data types that users must define) and **user-defined functions** (subroutines that users must define). The user-defined types include FullParticle type, EssentialParticleI type, EssentialParticleJ type, and Force type. The user-defined functions include calcForceEpEp defining particle-particle interaction and calcForceEpSp defining particle-superparticle interaction. This chapter describes how these types and functions should be implemented. FDPS requires that the user-defined types have quantities necessary to perform particle simulation such as the position of particle, etc. Users must tell FDPS the member variables of the user-defined types corresponding to the necessary quantities. FDPS internally performs data copy between these user-defined types according to the ways specified by users. Therefore, users must describe their ways in the users' codes. In FDPS Fortran interface, all the instructions to FDPS are done by describing special directives in users' codes (hereafter, we call them (**FDPS directives**). In the following, we first describe the user-defined types and then the user-defined functions are explained.

5.1 User-defined types

We first outline user-defined types. A FullParticle type is a derived data type that contains all informations of a particle and it is used to create ParticleSystem object (see step 0 in § 2.3 of Chap. 2). EssentialParticleI type, EssentialParticleJ type, and Force type are derived data types that support for defining the interactions between two particles. EssentialParticleI and EssentialParticleJ types contain the quantities of i- and j- particles used for the calculation of interactions. A Force type contains the quantities of an i-particle used to store the results of the calculations of interactions. Since these types contain part of information of FullParticle type, it is possible to use FullParticle in place of these types. However, FullParticle type may contain other values which are not used to evaluate the calculations of interactions. It is recommended to use these types when high performance is desirable.

In the following, we first describe common rules to be satisfied when users define user-defined types. Then, we explain how to implement user-defined types in order of FullParticle type, EssentialParticleI type, EssentialParticleJ type, and Force type.

5.1.1 Common rules

5.1.1.1 Requirements for Fortran syntax

In this section, we describe the minimum requirements for Fortran syntax that must be satisfied for a Fortran derived data type to be a user-defined type. As described in Chap. 3, the Fortran interface programs send data to FDPS or receive data from FDPS through the C interface of FDPS. Hence, all the user-defined types must be **interoperable** with C according to Fortran 2003 standard. More specifically, a derived data type that is to be a user-defined type must satisfy the following conditions:

(1) The derived data type has the `bind(c)` attribute.
(2) The data types of all member variables must be interoperable with C. A list of the data types interoperable with C in Fortran 2003 standard (ISO/IEC 1539-1:2004(E)) can be found at § 15 "Interoperability with C" of the specification document of Fortran 2003 standard[*1)], unofficial documents[*2)] introduced in the page GFortranStandards of the site GCC Wiki, and the online document of GNU gfortran. It is allowed that a derived data type that is interoperable with C becomes a member variable.
(3) All member variables do not have the `allocatable` attribute.
(4) All member variables do not have the `pointer` attribute.
(5) The derived data type does not have member functions or member subroutines.

In addition, FDPS Fortran interface requires the following conditions:

(6) The derived data type must be defined in a Fortran module.
(7) The derived data type must have the `public` attribute.
(8) Vector types and symmetric matrix types defined in Chap. 4 are the only derived data types that can be member variables of an user-defined type.
(9) The derived data type cannot have multidimensional arrays as member variables (this limitation will be removed in future).
(10) When specifying the shape of a (one-dimensional) member array, users must specify it **either** of by using the `dimension` statement or by adding a string `(size)` to the right of the name of the member variable, where *`size`* is the number of elements of the array.

These are the minimum requirements that a derived data type must satisfy to be a user-defined type. In addition to these, users must specify both the type of user-defined types (FullParticle, EssentialParticleI, EssentialParticleJ, and Force) and member variables corresponding to the necessary quantities using FDPS directives explained in the next section (§ 5.1.1.2).

5.1.1.2 FDPS directives usable for all the user-defined types

In this section, we present the summary of FDPS directives that can be used for all of the user-defined types and how to describe them. FDPS directives specific for each user-defined type is explained in § 5.1.2 - 5.1.5.

There are three kinds of FDPS directives:

[*1)] As of writing this, it seems that we can buy the specification document of Fortran 2008 Standard (ISO/IEC 1539-1:2010(E)) only from ISO (International Organization for Standardization).
[*2)] drafts for the documents of language specification

(a) The directive specifying the type of a user-defined type.
(b) The directives specifying which member variable corresponds to each of the necessary quantities.
(c) The directives specifying the ways of data copy between different user-defined types.

Among them, the first two items (a),(b) are explained as follows.

5.1.1.2.1 FDPS directive specifying the type of an user-defined type

In order to specify which user-defined type a derived data type *type_name* corresponds to, users must describe directive in the following format:

```
type, public, bind(c) :: type_name !$fdps keyword
end type [type_name]
```

or,

```
!$fdps keyword
type, public, bind(c) :: type_name
end type [type_name]
```

where `[]` are the symbols that represents that users can omit descriptions inside the brackets (users must not write the bracket symbols `[]` in practice). All the FDPS directives start with a string `!$fdps`, of which all the alphabetical characters must be lower case. As it can be seen from the fact that the string starts with !, FDPS directive is just a comment and it does not affect a Fortran program. Only the interface-generating script interprets it as a directive or an instruction by a user. *keyword*, which follows `!$fdps` with separated by one or more space character(s), is the keyword to specify the type of this user-defined type. Possible keywords include `FP`, `EPI`, `EPJ`, and `Force`, and all of these are case-sensitive. They correspond to FullParticle type, EssentialParticleI type, EssentialParticleJ type, and Force type, respectively. This directive must be described either of at the right of the name of the derived data type or at the previous line of the type definition. Users cannot start a new line in the directive. As described in § 5.1, EssentialParticleI type, EssentialParticleJ type, and Force type are subsets of FullParticle type. Hence, FullParticle type can serve as these types. In this case, please specify all relevant keywords with separating them by comma as shown in Listing 5.1:

Listing 5.1: An example of cases where FullParticle type serves as the others types

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

It is possible that FullParticle type serves as EssentialParticleI type only.

5.1.1.2.2 FDPS directives specifying the type of a member variable

Next, we explain the FDPS directives (b), specifying which necessary quantity a member variable corresponds to. The necessary quantities in FDPS include the charge (or mass) of

particle and the position of particle. In addition, depending on the type of particle simulation, the search radius is required. In order to specify which necessary quantity member variable *mbr_name* of derived data type *type_name* corresponds to, users must describe FDPS directive(s) in the following format:

```
type, public, bind(c) :: type_name
   data_type :: mbr_name !$fdps keyword
end type [type_name]
```

or,

```
type, public, bind(c) :: type_name
   !$fdps keyword
   data_type :: mbr_name
end type [type_name]
```

where we have omitted the directive specifying the type of an user-defined type for brevity. The directive starts with a string `!$fdps` and, after one or more space characters, *keyword* follows. Possible keywords include `id`, `charge`, `position`, `rsearch`, and `velocity`[*3)]. They correspond to the identification number of particle, the charge (mass) of particle, the position of particle, the search radius of particle, and the velocity of particle, respectively. The keywords must be lower-case. This directive must be matched with a single member variable and be described either of at the right of the name of a member variable or at the previous line of the variable declaration statement for a target member variable.

The data type of a member variable, *data_type*, must match that of the corresponding necessary quantity. In the following table, we summarize the data types of the necessary quantities in FDPS:

Name	Possible data types
Identification number	`integer(kind=c_long_long)`
Charge (mass) and search radius	`real(kind=c_float)` `real(kind=c_double)`

Position and velocity	`type(fdps_f32vec)`
	`real(kind=c_float), dimension(space_dim)`†
	`type(fdps_f64vec)`
	`real(kind=c_double), dimension(space_dim)`†

† `space_dim` is the spatial dimension of the simulation. If macro `PARTICLE_SIMULATOR_TWO_DIMENSION` is defined at the compilation, `space_dim` is 2. Otherwise, 3 (see Chap. 7).

Table 5.1: The data types of the necessary quantities

[*3)] Note that the keyword `velocity` is just a reserved word in the current version and it does not change the interface programs generated.

5.1.1.2.3 Example of writing FDPS directives

Finally, we show an example of the implementation of FullParticle type (Listing 5.2). Please check the usage of two FDPS directives, (a) and (b). Note that FDPS directive (c), which is not explained here, is used in this example. For this, see the following sections.

Listing 5.2: An example of the implementation of an user-defined type

```
1  module user_defined_types
2     use, intrinsic :: iso_c_binding
3     use :: fdps_vector
4     implicit none
5
6     !**** Full particle type
7     type, public, bind(c) :: full_particle !$fdps FP,EPI,EPJ,Force
8        !$fdps copyFromForce full_particle (pot,pot) (acc,acc)
9        !$fdps copyFromFP full_particle (id,id) (mass,mass) (eps,eps) (pos,
               pos)
10       integer(kind=c_long_long) :: id
11       real(kind=c_double) :: mass !$fdps charge
12       real(kind=c_double) :: eps
13       type(fdps_f64vec) :: pos !$fdps position
14       type(fdps_f64vec) :: vel !$fdps velocity
15       real(kind=c_double) :: pot
16       type(fdps_f64vec) :: acc
17    end type full_particle
18
19 end module user_defined
```

In the following, we explain how to define each user-defined type as well as FDPS directive specific for each.

5.1.2 FullParticle type

The FullParticle type contains all information of a particle and it is required to create a ParticleSystem object (see step 0 in § 2.3 of Chap. 2). Users can define arbitrary member variables. However, users must specify member variables corresponding to the necessary quantities and the way of data copy between FullParticle type and the other user-defined types using FDPS directives. Below, we first describe FDPS directives always required. Then, FDPS directives required in some specific cases are described.

5.1.2.1 FDPS directives always required and how to describe them

FDPS directives always required are as follows:

- Directive specifying a member variable corresponding to the charge (mass) of particle.
- Directive specifying a member variable corresponding to the position of particle.
- Directive specifying the way of data copy from Force type, which store the results of the calculation of interactions, to FullParticle type.

The first two can be described by the method explained in § 5.1.1.2. As for the third directive, users must describe it in the following format:

```
type, public, bind(c) :: FP
   !$fdps copyFromForce force (src_mbr,dst_mbr) (src_mbr,dst_mbr) ...
end type FP
```

The directive starts with a string `!$fdps`. After one or more space characters follows it, users must describe the keyword `copyFromForce`. This keyword tells the interface-generating script that this directive specifies the way of data copy from Force type to FullParticle type. After the keyword `copyFromForce`, users must describe the name of the derived data type corresponding to Force type, *force*. One or more space characters are needed between `copyFromForce` and *force*. Then, one or more pairs of member variable names (*src_mbr*,*dst_mbr*) are described. The delimiter is space characters. Each pair specifies the names of source and destination variables of copy operation. *src_mbr* and *dst_mbr* are the names of member variables of Force type and FullParticle type, respectively. Users cannot start a new line in the directive.

In some particle simulations, users may have to define multiple interactions, hence multiple corresponding Force types for a single FullParticle type. In such cases, users must describe this FDPS directive for each Force type.

An example of this FDPS directive is shown in Listing 5.2 and please check it.

5.1.2.2 FDPS directives required in specific cases and how to describe them

In this section, we describe FDPS directives in the following cases:

(i) Cases where the following types of Tree objects are used:

- Long-MonopoleWithScatterSearch type
- Long-QuadrupoleWithScatterSearch type
- Long-MonopoleWithCutoff type
- All Short types

(ii) Case that users use the extended feature "Particle Mesh"

(iii) Case that FullParticle type serves as other user-defined types

In case (i), users must specify a member variable corresponding to the search radius (for the types of Tree objects, see Chap. 8). This can be specified by the method described in § 5.1.1.2.

In case (ii), users must specify the way of copying the results of force calculation performed in the Particle Mesh module of FDPS to FullParticle type. This can be done by the following FDPS directive:

```
type, public, bind(c) :: FP
   !$fdps copyFromForcePM mbr_name
end type FP
```

The FDPS directive begins with a string `!$fdps`, which is followed by the keyword `copyFromForcePM` after the lapse of one or more space characters. This keyword tells FDPS that this directive specifies the way of copying data from the Particle Mesh module of FDPS to

FullParticle type. Next to the keyword, users must describe the name of a member variable of FullParticle type, *mbr_name*, to receive data from the Particle Mesh module. One or more space characters should be needed between `copyFromForcePM` and *mbr_name*. The data type of this member variable must be vector type defined Chap. 4. Users cannot start a new line in the FDPS directive.

In case (iii), users must describe all FDPS directives that are required in other user-defined types. Please see the sections of the corresponding user-defined types on these points.

5.1.3 EssentialParticleI type

The EssentialParticleI type should contain all information of an i-particle and it is necessary to define subroutines calculating interaction and to create Tree objects. EssentialParticleI type is a subset of FullParticle type (§ 5.1.2). Users must specify member variables corresponding to the necessary quantities and the way of copying data from FullParticle type to EssentialParticleI type. Below, we first describe FDPS directives always required. Then, FDPS directives required in some specific cases are described.

5.1.3.1 FDPS directives always required and how to describe them

FDPS directives always required are as follows:

- Directive specifying a member variable corresponding to the charge (mass) of particle.
- Directive specifying a member variable corresponding to position of particle.
- Directive specifying the way of data copy from FullParticle type to EssentialParticleI type.

The first two can be described by the method explained in § 5.1.1.2. As for the third directive, users must describe it in the following format:

```
type, public, bind(c) :: EPI
   !$fdps copyFromFP fp (src_mbr,dst_mbr) (src_mbr,dst_mbr) ...
end type EPI
```

The format of this directive is the same as that of the `copyFromForce` directive described in § 5.1.2.1 except (i) that the keyword following a string `!$fdps` is `copyFromFP` and (ii) that *fp* is the name of a derived data type corresponding to FullParticle type. Note that *src_mbr* is the name of a member variable of FullParticle type in this case.

5.1.3.2 FDPS directives required in specific cases and how to describe them

In this section, we describe FDPS directives in the following cases:

(i) Cases where the following types of Tree objects are used:

- All Short types

(ii) Case that EssentialParticleI type serves as other user-defined types

In case (i), users must specify a member variable corresponding to the search radius (for the types of Tree objects, see Chap. 8). This can be done by the method explained in § 5.1.1.2.

In case (ii), users must describe all FDPS directives that are required in other user-defined types. Please see the sections of the corresponding user-defined types on these points.

5.1.4 EssentialParticleJ type

The EssentialParticleJ type should contain all information of an j-particle and it is necessary to define subroutines calculating interaction and to create Tree objects. EssentialParticleJ type is a subset of FullParticle type (§ 5.1.2). Users must specify member variables corresponding to the necessary quantities and the way of copying data from FullParticle type to EssentialParticleJ type. Below, we first describe FDPS directives always required. Then, FDPS directives required in some cases are described.

5.1.4.1 FDPS directives always required and how to describe them

FDPS directives always required are as follows:

- Directive specifying a member variable corresponding to the charge (mass) of particle.
- Directive specifying a member variable corresponding to position of particle.
- Directive specifying the way of data copy from FullParticle type to EssentialParticleJ type.

The first two can be described by the method explained in § 5.1.1.2. The third directive must be described by the `copyFromFP` directive explained in § 5.1.3.1.

5.1.4.2 FDPS directives required in specific cases and how to describe them

In this section, we describe FDPS directives in the following cases:

(i) Cases where the following types of Tree objects are used:

- Long-MonopoleWithScatterSearch type
- Long-QuadrupoleWithScatterSearch type
- Long-MonopoleWithCutoff type
- All Short types

(ii) Case that EssentialParticleJ serves as other user-defined types

In case (i), users must specify a member variable corresponding to the search radius (for the types of Tree objects, see Chap. 8). This can be described by the method explained in § 5.1.1.2.

In case (ii), users must describe all FDPS directives that are required in other user-defined types. Please see the sections of the corresponding user-defined types on these points.

5.1.5 Force type

The Force type contains the results of the calculation of interactions and it is necessary to define subroutines calculating interaction and to create Tree objects. In this section we describe the member functions in any case required. Below, we first describe FDPS directives always required. Then, FDPS directives required in some cases are described.

5.1.5.1 FDPS directives always required and their description methods

FDPS directive always required is the directive specifying how to initialize Force type before the calculation of interactions. There are three ways to specify the way of initialization. Users must specify the initialization method by one method. Below, we explain each format.

(1) **Case that one wants to initialize all member variables by the default initialization of FDPS**
If users do not describe anything about the initialization, FDPS Fortran interface automatically applies the FDPS's default initialization, which sets integers and floating point numbers to 0, logical variables to `.false.`, and all the components of vector types and symmetric matrix types defined in Chap. 4 to 0.

(2) **Case that one wants to initialize member variables individually**
Users can specify the initial values of member variables individually by the following directive:

```
type, public, bind(c) :: Force
   !$fdps clear [mbr=val , mbr=keep, ...]
end type Force
```

where `Force` is the name of a derived data type corresponding to Force type. For brevity, we have omitted FDPS directive specifying that this derived data type is Force type. A string `!$fdps` shows the beginning of the directive. After one or more space characters, the keyword `clear` follows. This keyword tells FDPS that this directive specifies the way of initialization of Force type. Brackets `[]` after the keyword `clear` is the symbols that represents that users can omit descriptions inside of the brackets. Do not describe the bracket symbols `[]` in actual users' codes.

The way of the initialization of each member variable is described after the keyword `clear`. FDPS Fortran interface automatically applies the FDPS's default initialization for the member variables not specified here. There are two syntaxes to specify the way to initialize. In the following, we explain them.

In order to set member variable *`mbr`* to value *`val`*, users must use the *`mbr=val`* syntax, where we can insert one or more space characters before or after the symbol `=`. The data type of *`val`* must be consistent with that of *`mbr`*. Furthermore, *`val`* must be described according to the language specification of Fortran. For instance, if *`mbr`* is logical type, *`val`* must be either `.true.` or `.false.`. If *`mbr`* is a vector or a symmetric matrix, only the initialization that sets all the components to the same value is allowed, and *`val`* must be scalar value. If you want to set each component of vector or symmetric matrix to a different value, please use the FDPS directive explained in the next item.

To avoid the initialization of member variable *mbr*, users should use the *mbr*=keep syntax. The word keep in the right-hand side instructs FDPS to skip the initialization of a member variable written in the left-hand side. Again, we can insert one or more space characters before or after the symbol =.

Users can specify the ways of initialization of several member variables. In that case, please describe the syntaxes in a line with separating them by comma.

(3) **Case that one wants to initialize in a more complex way**
A more complex initialization can be performed by using a Fortran subroutine. In this case, users must describe the following directive:

```
type, public, bind(c) :: Force
   !$fdps clear subroutine subroutine_name
end type Force
```

where *subroutine_name* is the name of a Fortran subroutine used to initialize Force type. This subroutine must be defined in the global region or the global namespace. In other words, this subroutine must not be defined in a Fortran module or as an internal procedure. It must have the following interface:

```
subroutine subroutine_name(f) bind(c)
   use, intrinsic :: iso_c_binding
   implicit none
   type(Force), intent(inout) :: f

   ! Initialize Force

end subroutine [subroutine_name]
```

where [] is the same symbols used in the previous item.

5.1.5.2 FDPS directives required in specific cases and how to describe them

None.

5.2 User-defined functions

We first outline user-defined functions. Functions calcForceEpEp and calcForceSpEp are the functions that calculate action from j-particles to i-particle and action from superparticles to i-particle, respectively. The function pointers of these functions are passed to APIs for Tree objects as arguments. If the type of interaction is short-range force, superparticle is not required. In this case, users do not need to define calcForceEpSp.

5.2.1 Common rules

5.2.1.1 Requirements for Fortran syntax

In this section, we describe the minimum requirements for Fortran syntax that must be satisfied for a Fortran subroutine to be a user-defined function. To explain this, we first summarize the procedures to be followed when users perform the calculation of interaction using FDPS Fortran interface. Assuming that the interface programs are successfully generated, the procedures are as follows:

(I) Implement Fortran subroutine(s) that define interaction between particles.
(II) Prepare variables to store the C addresses of Fortran subroutines in user's codes. The data type of them must be the derived data type `type(c_funloc)`, which is defined in the module `iso_c_binding` introduced in Fortran 2003 standard.
(III) Get the C addresses of the Fortran subroutines that are used in the interaction calculation by using the function `c_funloc`, which is also provided by the module `iso_c_binding`, and store them in the variables prepared in the step (II).
(IV) Call a Fortran API to perform the interaction calculation, where the variables storing the C addresses (see the step (III)) are passed to the API as arguments.
(V) FDPS interprets or regards the Fortran subroutines specified by the C addresses as C functions and performs the interaction calculation.

In the step (III), the Fortran subroutines must be interoperable with C in order to get their C addresses by using the function `c_funloc`. More specifically, a Fortran subroutine can be a user-defined function if the following conditions are satisfied:

(1) a Fortran subroutine must have the `bind(c)` attribute.
(2) All the data types of dummy arguments must be the data types that are interoperable with C. More information for such data types can be found in § 5.1.1.

5.2.1.2 Requirements from FDPS

In addition to the conditions described in the previous section, there are some requirements due to the the specification of FDPS itself. They are as follows:

(3) The dummy arguments corresponding to the numbers of i-particles and j-particles must have the `value` attribute, which is an attribute introduced in Fortran 2003 standard and which specifies that an argument is a pass-by-value argument.

These are the minimum requirements that a user-defined function must satisfy. To help users to understand, we show the implementation of the user-defined function used in the N-body sample code in Listing 5.3 as an example. Because we have not yet explained the detail of the method of describing user-defined functions, please confirm the locations of the `bind(c)` attribute and the `value` attribute only for now.

Listing 5.3: An example of the implementation of an user-defined function defining particle-particle interaction in N-body simulation

```
1 subroutine calc_gravity_pp(ep_i,n_ip,ep_j,n_jp,f) bind(c)
2    integer(c_int), intent(in), value :: n_ip,n_jp
3    type(full_particle), dimension(n_ip), intent(in) :: ep_i
```

```
4     type(full_particle), dimension(n_jp), intent(in) :: ep_j
5     type(full_particle), dimension(n_ip), intent(inout) :: f
6     !* Local variables
7     integer(c_int) :: i,j
8     real(c_double) :: eps2,poti,r3_inv,r_inv
9     type(fdps_f64vec) :: xi,ai,rij
10
11    do i=1,n_ip
12       eps2 = ep_i(i)%eps * ep_i(i)%eps
13       xi%x = ep_i(i)%pos%x
14       xi%y = ep_i(i)%pos%y
15       xi%z = ep_i(i)%pos%z
16       ai%x = 0.0d0
17       ai%y = 0.0d0
18       ai%z = 0.0d0
19       poti = 0.0d0
20       do j=1,n_jp
21          rij%x  = xi%x - ep_j(j)%pos%x
22          rij%y  = xi%y - ep_j(j)%pos%y
23          rij%z  = xi%z - ep_j(j)%pos%z
24          r3_inv = rij%x*rij%x &
25                 + rij%y*rij%y &
26                 + rij%z*rij%z &
27                 + eps2
28          r_inv  = 1.0d0/sqrt(r3_inv)
29          r3_inv = r_inv * r_inv
30          r_inv  = r_inv * ep_j(j)%mass
31          r3_inv = r3_inv * r_inv
32          ai%x   = ai%x - r3_inv * rij%x
33          ai%y   = ai%y - r3_inv * rij%y
34          ai%z   = ai%z - r3_inv * rij%z
35          poti   = poti - r_inv
36       end do
37       f(i)%pot   = f(i)%pot   + poti
38       f(i)%acc%x = f(i)%acc%x + ai%x
39       f(i)%acc%y = f(i)%acc%y + ai%y
40       f(i)%acc%z = f(i)%acc%z + ai%z
41    end do
42
43 end subroutine calc_gravity_pp
```

5.2.2 calcForceEpEp

Function calcForceEpEp defines the interaction between two particles and it is required for the calculation of interactions. Users must define function calcForceEpEp according to the following format:

```
subroutine calc_force_ep_ep(ep_i,n_ip,ep_j,n_jp,f) bind(c)
   use, intrinsic :: iso_c_binding
   implicit none
   integer(kind=c_int), intent(in), value :: n_ip,n_jp
   type(essential_particle_i), dimension(n_ip), intent(in) :: ep_i
   type(essential_particle_j), dimension(n_jp), intent(in) :: ep_j
   type(force), dimension(n_ip), intent(inout) :: f

end subroutine calc_force_ep_ep
```

Dummy argument specification

Name	Data type	I/O characteristics	Definition
n_ip	integer(kind=c_int)	Input	The number of i-particles.
n_jp	integer(kind=c_int)	Input	The number of j-particles.
ep_i	essential_particle_i type†	Input	Array of i-particles.
ep_j	essential_particle_j type†	Input	Array of j-particles.
f	force type†	Input and Output	Array of the results of interaction of i-particles.

† The names of derived data types of EssentialParticleI type, EssentialParticleJ type, and Force types, respectively. If these derived data types are defined in Fortran modules different from the module that defines calcForceEpEp, users must `use` these modules in calcForceEpEp.

Returned value

None.

Function

Calculates the interaction to i-particle from j-particle.

5.2.3 calcForceEpSp

Function calcForceEpSp defines the interaction to a particle from superparticle and it is required for the calculation of interactions. Users must define function calcForceEpSp according to the following format:

```
subroutine calc_force_ep_sp(ep_i,n_ip,ep_j,n_jp,f) bind(c)
   use, intrinsic :: iso_c_binding
   use :: fdps_super_particle
   implicit none
   integer(kind=c_int), intent(in), value :: n_ip,n_jp
   type(essential_particle_i), dimension(n_ip), intent(in) :: ep_i
   type(super_particle_j), dimension(n_jp), intent(in) :: ep_j
   type(force), dimension(n_ip), intent(inout) :: f

end subroutine calc_force_ep_sp
```

Dummy argument specification

Name	Data type	I/O characteristics	Definition
`n_ip`	integer(kind=c_int)	Input	The number of i-particles.
`n_jp`	integer(kind=c_int)	Input	The number of superparticles.
`ep_i`	essential_particle_i type†	Input	Array of i-particles.
`ep_j`	super_particle_j type‡	Input	Array of superparticles.
`f`	force type†	Input and Output	Array of the results of interaction of i-particles.

† The names of derived data types of EssentialParticleI type and Force type. If these derived data types are defined in Fortran modules different from the module that defines calcForceEpSp, user must `use` these modules in calcForceEpSp.
‡ `super_particle_j` type must be one of superparticle types defined in § 4.3 of Chap. 4.

Returned value

None.

Function

Calculate interactions from superparticle to i-particle.

Chapter 6

Generating Fortran Interface

In this chapter, we describe the system requirements and the usage of the Fortran interface-generating script.

6.1 System requirements for the script

This section describes the system requirements for the interface-generating script. The script is placed in the directory `scripts` and it is implemented by the programing language Python. In order for the script to work, Python 2.7.5 or later, or, Python 3.4 or later must be installed in your system. Before using the script, please modify the following first line of the script according to your computational environment:

```
#!/usr/bin/env python
```

Things to be checked are the PATH of the `env` command and the name of a Python interpreter (there is a case where the name of an interpreter is not `python`, but `python2.7` or `python3.4` [the numbers indicate the versions of the interpreters], depending on the system). If a Python interpreter is not in the environment variable `PATH`, please add the PATH of the interpreter to `PATH` or specify the interpreter by its absolute PATH as follows:

```
#!/path/to/python
```

In addition, user's codes must satisfy the following conditions for the script to work:

- All the user's codes input to the script must be written by Fortran 2003 standard (ISO/IEC 1539-1:2004(E)). This script does not have function to identify the programing language used in user's codes and an advanced syntax checker. Hence, the script might show an unexpected behavior if there are syntax errors in users codes.

6.2 Usage of the script

To generate FDPS Fortran interface programs, run the script as follows:

```
$ gen_ftn_if.py -o output_directory user1.F90 user2.F90...
```

where we assumed that the directory `scripts` is added to the environment variable `PATH`.

Otherwise, you must run the script by its relative PATH or its absolute PATH. The script `gen_ftn_if.py` accepts Fortran source codes as arguments and it requires that the user-defined types must be defined in one of the input files. If the number of the input files is more than 1, you must separate them by at least one space in no particular order.

The option `-o` can be used to specify the directory where the interface programs are output. You can use the options `--output` or `--output_dir` instead of `-o`. If not specified, the interface programs are output in the current directory.

The option `-DPARTICLE_SIMULATOR_TWO_DIMENSION` is used to indicate that the spatial dimension of the simulation is 2. This option does not have an argument. If this option is not specified, the script assume that the spatial dimension of the simulation is 3. The spatial dimension of the simulation is used to check the data types of the member variables that correspond to position and velocity. If the macro `PARTICLE_SIMULATOR_TWO_DIMENSION` is defined at the compilation of user's codes, you must specify this option when generating the interface programs (otherwise, the interface programs do not work as expected).

The usage of the script can be checked by executing the script with the options `-h` or `--help`:

```
[user@hostname somedir]$ gen_ftn_if.py -h
[namekata@jenever0 scripts]$ ./gen_ftn_if.py --help
usage: gen_ftn_if.py [-h] [-o DIRECTORY] [-DPARTICLE_SIMULATOR_TWO_DIMENSION]
                     FILE [FILE ...]

Analyze user's Fortran codes and generate C++/Fortran source files
required to use FDPS from the user's Fortran code.

positional arguments:
  FILE                  The PATHs of input Fortran files

optional arguments:
  -h, --help            show this help message and exit
  -o DIRECTORY, --output DIRECTORY, --output_dir DIRECTORY
                        The PATH of output directory
  -DPARTICLE_SIMULATOR_TWO_DIMENSION
                        Indicate that simulation is performed
                        in the 2-dimensional space (equivalent
                        to define the macro
                        PARTICLE_SIMULATOR_TWO_DIMENSION)
```

You can obtain the execution file by compiling interface programs generated and user's codes. Next chapter (Chap. 7) explain how to compile your codes.

Chapter 7

Compling Fortran Interface

In previous chapters, we have described the necessary information to write a simulation code using FDPS Fortran interface and to generate the Fortran interface programs. In this chapter, we cover topic related to the compilation of user's code together with the interface programs. As shown in Fig. 3.1 in Chap. 3, the interface programs consist of C++ source codes and Fortran source codes. The first half of this chapter describes how to compile the interface programs. Some of the features of FDPS such as the type of coordinate used in a simulation and the method of parallelization should be specified at the compile time. Therefore, the last half of this chapter explains about the macros that are available in FDPS Fortran interface.

7.1 Compilation

In this section, we describe how to compile user's codes together with the interface programs. Firstly, we describe a general procedure that does not depend on compilers. Then, we explain the compilation method for the case of GCC (The GNU Compiler Collection) as an example.

7.1.1 Basic procedure of the compilation

Here, we explain basic procedure of the compilation independent of the type of compiler.

In order to obtain an executable file from source codes described both in C++ and Fortran, we must prepare a C++ compiler, a C++ linker, and a Fortran compiler that are interoperable with each other. Today, most of C++ compilers work as a C++ linker. Hence, we only have to prepare a C++ compiler and a Fortran compiler that are interoperable with each other. Fortran compiler must support Fortran 2003 standard (ISO/IEC 1539-1:2004(E)). In addition, C++ compiler must support C++03 standard (ISO/IEC 14882:2003) to compile FDPS itself.

As described in Chap. 3, the so-called `main` function exists in the C++ side of user's codes. Therefore, to get the executable file, we must use a C++ linker to link the object files that are created from the source codes by using the both compilers. More specifically, the compilation is performed as follows:

[1] **Compiling Fortran source codes**
Compile all the user's Fortran source codes, `FDPS_module.F90` (one of the interface pro-

grams), all the Fortran source codes provided by us (`src/fortran_interface/modules/*.F90`) to create the Fortran object files. You can obtain the object files by compiling with the compilation option `-c` in most cases.

One thing you should be careful of is the order of the files passed to the Fortran compiler. In most of Fortran compilers, when the module `foo` is used in the file `bar.F90`, the file that defines the module `foo` must be compiled before compiling the file `bar.F90`. Because the compiler processes the files in the same order as the argument, we must first pass the files in which independent modules are defined and then we need to pass the remaining files according to their dependency relation. To be more precise, we have to compile as follows:

```
$ FC -c \
  FDPS_time_profile.F90 \
  FDPS_vector.F90 \
  FDPS_matrix.F90 \
  FDPS_super_particle.F90 \
  user_defined_1.F90 ... user_defined_n.F90 \
  FDPS_module.F90 \
  user_code_1.F90 ... user_code_n.F90
```

where `FC` is a Fortran compiler. The symbol "\" shows that the command-line is continued to the next line. This symbol is introduced due to space limitation and it is unnecessary in practice. Here, we assumed that the subroutine `f_main()` is implemented in one of the user's codes (`user_code_*.F90`). The dependency relationship of the files in this example is as follows:

- `FDPS_super_particle.F90` depends on both `FDPS_vector.F90` and `FDPS_matrix.F90`.
- `FDPS_module.F90` depends on the n files `user_defined_`i`.F90` (i = 1-n) where the user-defined types are defined.
- the n files `user_code_`i`.F90` (i = 1-n) where the main part of the simulation code are implemented depend on `FDPS_module.F90`.

[2] **Compiling C++ source codes**
Compile all the C++ files in the interface programs (`main.cpp`, `FDPS_Manipulators.cpp`, `FDPS_ftn_if.cpp`) to create the C++ object files. Because of the existence of header files, you do not need to worry about the order of the files in the C++ case. Hence, we compile as follows:

```
$ CXX -c FDPS_Manipulators.cpp FDPS_ftn_if.cpp main.cpp
```

where `CXX` is a C++ compiler.

[3] **Linking the object files**
Using a C++ linker (actually a C++ compiler), Link the object files (`*.o`) created in the steps [1] and [2] to obtain the executable file. Depending the type of compiler, the compiler may require a special compilation option to link C++ objects with Fortran objects. Assuming that this option is `LDFLAGS`, the link is performed as follows:

```
$ CXX *.o [LDFLAGS]
```

where the symbols [] shows that the inside of it can be omitted and they should not be described in practice. If succeed in linking, the executable file will be created.

In the procedure described above, we have omitted other compilation options such as the option that specifies the language specification. Also, we have omitted the options specifying libraries needed for parallel computation or the extended feature "ParticleMesh". The way of specifying these things depend on the types of both compiler and system that users use. Users must specify them at the compilation accordingly.

7.1.2 Compilation with GCC

In this section, we describe how to compile user's codes with GCC (ver. 4.8.3 or later) as an example. Throughout this section, we assume the following things: (i) the C++ and Fortran compilers are `g++` and `gfortran`, respectively, (ii) the compilers that support MPI are `mpic++` and `mpif90`, (iii) the MPI library used is OpenMPI (ver. 1.6.4 or later). In the following, we explain separately the cases with and without the use of MPI.

7.1.2.1 The case without MPI

In GCC, we need the compilation option `-std=f2003` to compile Fortran source codes according to Fortran 2003 standard. Also, we need the link option `-lgfortran` to link C++ object files to Fortran object files. Therefore, set the variables `FC`, `CXX`, and `LDFLAGS` in the procedure described in § 7.1.1 as follows:

```
FC      = gfortran -std=f2003
CXX     = g++
LDFLAGS = -lgfortran
```

7.1.2.2 The case with MPI

When using MPI, attention should be paid if users use MPI in the user's codes. In this case, we need link the MPI library for Fortran in addition to that for C++. Assuming that the names of both libraries are `libmpi` and `libmpi_f90` respectively, the compilation will succeed if you set the variables `FC`, `CXX`, and `LDFLAGS` in the procedure described in § 7.1.1 as follows:

```
FC      = mpif90 -std=f2003
CXX     = mpic++
LDFLAGS = -lgfortran -LPATH -lmpi -lmpi_f90
```

where *PATH* is the absolute PATH of the directory where the MPI libraries are installed.

The names of MPI libraries will be different depending on the system users use. Regarding this point, please inquire the administrator of the computer system users use.

7.2 Macro at the compilation

7.2.1 Coordinate system

Users have alternatives of 2D and 3D Cartesian coordinate systems.

7.2.1.1 3D Cartesian coordinate system

3D Cartesian coordinate system is used by default.

7.2.1.2 2D Cartesian coordinate system

2D Cartesian coordinate system can be used by defining `PARTICLE_SIMULATOR_TWO_DIMENSION` as macro.

7.2.2 Parallel processing

Users choose whether OpenMP is used or not, and whether MPI is used or not.

7.2.2.1 OpenMP

OpenMP is disabled by default. If macro `PARTICLE_SIMULATOR_THREAD_PARALLEL` is defined, OpenMP becomes enabled. Compiler option `-fopenmp` is required for GCC compiler.

7.2.2.2 MPI

MPI is disabled by default. If macro `PARTICLE_SIMULATOR_MPI_PARALLEL` is defined, MPI becomes enabled.

7.2.3 Accuracy of data types

Users can change the accuracy of data types in Superparticle types.

7.2.3.1 Accuracy of data types in superparticle types

All the member variables in Superparticle types are 64 bit accuracy. They becomes 32 bit accuracy if macro `PARTICLE_SIMULATOR_SPMOM_F32` is defined at the compile time.

7.2.4 The extended feature "Particle Mesh"

The extended feature "Particle Mesh" is disabled by default. If macro `PARTICLE_SIMULATOR_USE_PM_MODULE` is defined, this feature becomes enabled.

7.2.5 Log output for debugging

If macro `PARTICLE_SIMULATOR_DEBUG_PRINT` is defined, FDPS output detailed log. This may be useful for debugging.

Chapter 8

List of API specifications

In this chapter, we describe the specifications of all the APIs in FDPS Fortran interface. As described in Chap. 3, all the APIs are implemented as the member functions of an object of the Fortran 2003 class `fdps_controller`. In the following, we assume that the name of an instance of this class is `fdps_ctrl`.

8.1 APIs for initialization/finalization

In this section, we describe APIs to initialize or finalize FDPS.

8.1.1 PS_initialize

```
subroutine fdps_ctrl%ps_initialize()
```

Dummy argument specification

None.

Returned value

None.

Function

Initialize FDPS. This API must be called before other APIs of FDPS are called.

8.1.2 PS_finalize

```
subroutine fdps_ctrl%ps_finalize()
```

Dummy argument specification

None.

Returned value

None.

Function

Finalize FDPS.

8.1.3 PS_abort

```
subroutine fdps_ctrl%ps_abort(err_num)
```

Dummy argument specification

Name	Data type	I/O characteristics	Definition
err_num	integer(kind=c_int)	Input	Variable giving the termination status of a program. This argument is optional and the default value is -1.

Returned value

None.

Function

Terminate the user program abnormally. The argument is the termination status of the program and it is passed to the function `MPI::Abort()` under MPI environment, otherwise it is passed to the C++ function `std::exit()`.

8.2 APIs for ParticleSystem object

In this section, we describe the specifications of APIs related to an object of ParticleSystem class in FDPS (see Chap. 2; hereafter, we call it **ParticleSystem object**). In FDPS, ParticleSystem object has all information of a particle as described in FullParticle type and provides an API to exchange particles between MPI processes. Users must manage particles via this ParticleSystem object. In FDPS Fortran interface, this object is managed by an identification number.

Here is the list of APIs to manipulate ParticleSystem object:

```
create_psys
delete_psys
init_psys
get_psys_info
get_psys_memsize
get_psys_time_prof
clear_psys_time_prof
set_nptcl_smpl
set_nptcl_loc
get_nptcl_loc
get_nptcl_glb
get_psys_fptr
exchange_particle
add_particle
remove_particle
adjust_pos_into_root_domain
sort_particle
```

In the following, we describe the specification of each API in the order shown above.

8.2.1 create_psys

```
subroutine fdps_ctrl%create_psys(psys_num,psys_info_in)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
psys_num	integer(kind=c_int)	Input and Output	Variable receiving the identification number of a ParticleSystem object.
psys_info_in	character (len=*,kind=c_char)	Input	The name of a derived data type corresponding to FullParticle type.

Returned value

None.

Function

Create an ParticleSystem object class and return its identification number. Users must specify the name of a derived data type in lower-case .

8.2.2 delete_psys

```
subroutine fdps_ctrl%delete_psys(psys_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.

Returned value

None.

Function

Delete a ParticleSystem object indicated by the identification number.

8.2.3 init_psys

```
subroutine fdps_ctrl%init_psys(psys_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.

Returned value

None.

Function

Initialize an ParticleSystem object indicated by the identification number.

8.2.4 get_psys_info

```
subroutine fdps_ctrl%get_psys_info(psys_num,psys_info)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.
psys_info	character (len=*,kind=c_char)	Input and Output	Characters to receive the name of a FullParticle type corresponding to a ParticleSystem object indicated by the identification number.

Returned value

None.

Function

Obtain the name of FullParticle type corresponding to ParticleSystem object indicated by the identification number.

8.2.5 get_psys_memsize

```
function fdps_ctrl%get_psys_memsize(psys_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.

Returned value

Type integer(kind=c_long_long).

Function

Return the size of the memory used in the ParticleSystem object.

8.2.6 get_psys_time_prof

```
subroutine fdps_ctrl%get_psys_time_prof(psys_num,prof)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.
prof	type(fdps_time_prof)	Input and Output	Variable giving the identification number of a ParticleSystem object.

Returned value

None.

Function

Store the execution time (in milliseconds) of the API `exchange_particle` to the member variable `exchange_particles` of a `time_profile` object.

8.2.7 clear_psys_time_prof

```
subroutine fdps_ctrl%clear_psys_time_prof(psys_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.

Returned value

None.

Function

ParticleSystem object has a private variable of TimeProfile class, which is a C++ class corresponding to a derived data type `fdps_time_prof` in FDPS Fortran interface (for details, see the specification document of FDPS, `doc_specs_cpp_en.pdf`). This API sets the the member variable `exchange_particles` of this private variable of the ParticleSystem object indicated by `psys_num` to 0. Usually, this API is used to reset time measurement.

8.2.8 set_nptcl_smpl

```
subroutine fdps_ctrl%set_nptcl_smpl(psys_num,nptcl)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.
nptcl	integer(kind=c_int)	Input	The average number of sample particles per MPI process.

Returned value

None.

Function

Set the average number of sample particles per MPI process. If this function is not called, the average number is 30.

8.2.9 set_nptcl_loc

```
subroutine fdps_ctrl%set_nptcl_loc(psys_num,nptcl)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.
nptcl	integer(kind=c_int)	Input	The number of particles of the MPI process calling this API.

Returned value

None.

Function

Set the number of particles of the MPI process calling this API.

8.2.10 get_nptcl_loc

```
function fdps_ctrl%get_nptcl_loc(psys_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
psys_num	integer(kind=c_int)	Input	The number of particles of the MPI process calling this API.

Returned value

Type integer(kind=c_int).

Function

Return the number of particles of the MPI process calling this API.

8.2.11 get_nptcl_glb

```
function fdps_ctrl%get_nptcl_glb(psys_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
psys_num	integer(kind=c_int)	Input	The total number of particles of all processes.

Returned value

Type integer(kind=c_int).

Function

Return the total number of particles of all processes.

8.2.12 get_psys_fptr

```
subroutine fdps_ctrl%get_psys_fptr(psys_num,fptr_to_FP)
```

Dummy argument specification

Name	Data type	I/O characteristics	Definition
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.
fptr_to_FP	FullParticle type, dimension(:), pointer	Input and Output	The pointer to the array of particles of FullParticle type stored in the ParticleSystem object.

Returned value

None.

Function

Get the pointer to the array of particles of FullParticle type stored in the ParticleSystem object indicated by the identification number psys_num. The size of the array is set to the number of the local particles, which is the returned value of the API get_nptcl_loc. Users can validly access the array elements fptr_to_FP(i) (i = 1-$n_{\mathrm{ptcl,loc}}$, where $n_{\mathrm{ptcl,loc}}$ is the number of local particles). This API provides only the way to access the array of particles stored in a ParticleSystem object. Below is an usage example of this API. In this example, one first gets the pointer of the array of particles of FullParticle type full_particle and then set some values to these particles:

Listing 8.1: An usage example of API get_psys_fptr

```
1  !* Local variables
2  type(full_particle), dimension(:), pointer :: ptcl
3  !* Get the pointer to full particle data
4  call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
5  !* Set particle data
6  do i=1,nptcl_loc
7     ptcl(i)%mass =  ! do something
8     ptcl(i)%pos%x = ! do something
9     ptcl(i)%pos%y = ! do something
10    ptcl(i)%pos%z = ! do something
11 end do
```

8.2.13 exchange_particle

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

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.
dinfo_num	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.

Returned value

None.

Function

Redistribute particles among MPI processes so that the particles are in appropriate domains.

8.2.14 add_particle

```
subroutine fdps_ctrl%add_particle(psys_num,ptcl)
```

Dummy argument specification

Name	Data type	I/O characteristics	Definition
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.
ptcl	FullParticle type	Input	Data of particle of FullParticle type.

Returned value

None.

Function

Add particle ptcl to the end of the array of particles of FullParticle type stored in the ParticleSystem object indicated by the identification number psys_num.

8.2.15 remove_particle

```
subroutine fdps_ctrl%remove_particle(psys_num,nptcl,ptcl_indx)
```

Dummy argument specification

Name	Data type	I/O characteristics	Definition
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.
nptcl	integer(kind=c_int)	Input	The number of particles to be removed.
ptcl_indx	integer(kind=c_int), dimension(nptcl)	Input	An array of the index of the particles to be removed.

Returned value

None.

Function

Remove the particles with the indice in the array idx. After calling this API, the order of the array of the particles would be changed.

8.2.16 adjust_pos_into_root_domain

```
subroutine fdps_ctrl%adjust_pos_into_root_domain(psys_num,dinfo_num)
```

Dummy argument specification

Name	Data type	I/O characteristics	Defintion
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.
dinfo_num	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.

Returned value

None.

Function

Under the periodic boundary condition, the particles outside the calculation domain move to appropriate positions.

8.2.17 sort_particle

```
subroutine fdps_ctrl%sort_particle(psys_num,pfunc_comp)
```

Dummy argument specification

Name	Data type	I/O characteristics	Definition
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.
pfunc_comp	type(c_funptr)	Input	Pointer to a function which returns .true. if FullParticle in the first argument is less than the other one in the second argument.

Returned value

None.

Function

This API sorts an array of FullParticles stored in the ParticleSystem object specified by psys_num in the order determined by a comparison function comp (function pointer to which is pfunc_comp). The returned value of comp must be logical(kind=c_bool) type and it must take two arguments of FullParticles. Note that the data types of two arguments of comp must be FullParticle type that is used in the creation of the ParticleSystem object specified by psys_num; Otherwise, the API does not work correctly. The following is an example of comp to sort FullParticles in ascending order of particle ID.

Listing 8.2: An example of comparison function

```
1 function comp(left, right) bind(c)
2    use, intrinsic :: iso_c_binding
3    use user_defined_types
4    implicit none
5    logical(kind=c_bool) :: comp
6    type(full_particle), intent(in) :: left, right
7    comp = (left%id < right%id)
8 end function comp
```

where we assume that derived data type full_particle is defined in the module user_defined_types.

8.3 APIs for DomainInfo object

In this section, we describe the specifications of APIs related to an object of DomainInfo class in FDPS (see Chap. 2; hereafter, we call it **DomainInfo object**). In FDPS, DomainInfo object has all information about the size of computational domain and decomposed domains and it provides an API to perform domain decomposition. In FDPS Fortran interface, this object is managed by an identification number.

This is the list of APIs to manipulate DomainInfo object:

```
create_dinfo
delete_dinfo
init_dinfo
get_dinfo_time_prof
clear_dinfo_time_prof
set_nums_domain
set_boundary_condition
set_pos_root_domain
collect_sample_particle
decompose_domain
decompose_domain_all
```

In the following, we describe the specification of each API in the order shown above.

8.3.1 create_dinfo

```
subroutine fdps_ctrl%create_dinfo(dinfo_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
dinfo_num	integer(kind=c_int)	Input and output	Variable receiving the identification number of a DomainInfo object.

Returned value

None.

Function

Create a DomainInfo object, and return its identification number.

8.3.2 delete_dinfo

```
subroutine fdps_ctrl%delete_dinfo(dinfo_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`dinfo_num`	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.

Returned value

None.

Function

Erase a DomainInfo object indicated by the identification number.

8.3.3 init_dinfo

```
subroutine fdps_ctrl%init_dinfo(dinfo_num,coef_ema)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`dinfo_num`	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.
`coef_ema`	real(kind=c_float)	Input	The smoothing factor of an exponential moving average (default: 1.0).

Returned value

void.

Function

Initialize an DomainInfo object. The argument `coef_ema` is the smoothing factor of exponential moving average and is a constant real value between 0 and 1. If other values are chosen, FDPS sends an error message and terminates the user program. A larger `coef_ema` weighs newer values rather than older values. In the case of unity, the domains are determined by using the newest values only and in the case of zero, they are determined by using the initial values only. Users call this API only once. The details of this function are described in the paper by Ishiyama, Fukushige & Makino (2009, Publications of the Astronomical Society of Japan, 61, 1319)

8.3.4 get_dinfo_time_prof

```
subroutine fdps_ctrl%get_dinfo_time_prof(dinfo_num,prof)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
dinfo_num	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.
prof	type(fdps_time_prof)	Input and output	Variable receiving time spent by DomainInfo APIs.

Returned value

None.

Function

Set the time spent by DomainInfo APIs collect_sample_particle and decompose_domain to collect_sample_particles and decompose_domain, respectively, which are member variables of derived data type fdps_time_prof.

8.3.5 clear_dinfo_time_prof

```
subroutine fdps_ctrl%clear_dinfo_time_prof(dinfo_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`dinfo_num`	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.

Returned value

None.

Function

DomainInfo object has a private variable of TimeProfile class, which is a C++ class corresponding to a derived data type `fdps_time_prof` in FDPS Fortran interface (for details, see the specification document of FDPS, `doc_specs_cpp_en.pdf`). This API sets the the member variables `collect_sample_particles` and `decompose_domain` of this private variable of the DomainInfo object indicated by `psys_num` to 0. Usually, this API is used to reset time measurement.

8.3.6 set_nums_domain

```
subroutine fdps_ctrl%set_nums_domain(dinfo_num,nx,ny,nz)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
dinfo_num	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.
nx	integer(kind=c_int)	Input	The number of subdomains along x direction.
ny	integer(kind=c_int)	Input	The number of subdomains along y direction.
nz	integer(kind=c_int)	Input	The number of subdomains along x direction (default: 1).。

Returned value

None.

Function

Set the numbers of subdomains. If the API is not called: nx, ny, and nz are determined automatically. If the product of nx, ny, and nz is not equal to the total number of MPI processes, FDPS sends an error message and terminates the user program.

8.3.7 set_boundary_condition

```
subroutine fdps_ctrl%set_boundary_condition(dinfo_num,bc)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
dinfo_num	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.
bc	integer(kind=c_int)	Input	Boundary conditions.

Returned value

None.

Function

Set the boundary condition. Input value must be one of the boundary condition types described in § 4.5 of Chap. 4. Namely, we must choose among fdps_bc_open, fdps_bc_periodic_x, fdps_bc_periodic_y, fdps_bc_periodic_z, fdps_bc_periodic_xy, fdps_bc_periodic_xz, fdps_bc_periodic_yz, fdps_bc_periodic_xyz, fdps_bc_shearing_box, and fdps_bc_user_defined (fdps_bc_shearing_box and fdps_bc_user_defined have not been implemented yet). If the API is not called, the open boundary is used.

8.3.8 set_pos_root_domain

```
subroutine fdps_ctrl%set_pos_root_domain(dinfo_num,low,high)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`dinfo_num`	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.
`low`	real(kind=c_float), dimension(space_dim) real(kind=c_double), dimension(space_dim) type(fdps_f32vec) type(fdps_f64vec)	Input	Top vertex of the boundary (inclusive).
`high`	the same type as `low`	Input	Bottom vertex of the boundary (exclusive).

Returned value

None.

Function

Set positions of vertexes of top and bottom of root domain. The API does not need to be called under open boundary condition. Every coordinate of `high` must be greater than the corresponding coordinate of `low`. Otherwise, FDPS sends a error message and terminates the user program.

8.3.9 collect_sample_particle

```
subroutine fdps_ctrl%collect_sample_particle(dinfo_num, &
                                             psys_num,  &
                                             clear,     &
                                             weight)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`dinfo_num`	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.
`psys_num`	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object whose particles are sampled for domain decomposition.
`clear`	logical(kind=c_bool)	Input	A flag whether previously sampled particles are cleared or not (default: `.true.`).
`weight`	real(kind=c_float)	Input	A weight to determine the number of sampled particles for domain decomposition (default: 1).

Returned value

None.

Function

Sample particles from an object of `ParticleSystem` class. If `clear` is true, the data of the samples collected before is cleared. Larger `weight` leads to give more sample particles.

8.3.10 decompose_domain

```
subroutine fdps_ctrl%decompose_domain(dinfo_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
dinfo_num	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.

Returned value

None.

Function

Decompose calculation domains.

8.3.11 decompose_domain_all

```
subroutine fdps_ctrl%decompose_domain_all(dinfo_num,psys_num,weight)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
dinfo_num	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object whose particles are sampled for domain decomposition.
weight	real(kind=c_float)	Input	A weight to determine the number of sampled particles for domain decomposition (default: 1).

Returned value

None.

Function

Sample particles from psys_num and decompose domains. This API is the combination of collect_sample_particle and decompose_domain.

8.4 APIs for Tree object

In this section, we describe the specifications of APIs related to an object of TreeForForce class in FDPS (see Chap. 2; hereafter, we call it **Tree object** simply). In FDPS, Tree object provides APIs to perform the calculation of interactions. In FDPS Fortran interface, this object is managed by an identification number.

This is the list of APIs to manipulate Tree object:

```
create_tree
delete_tree
init_tree
get_tree_info
get_tree_memsize
get_tree_time_prof
clear_tree_time_prof
get_num_interact_ep_ep_loc
get_num_interact_ep_sp_loc
get_num_interact_ep_ep_glb
get_num_interact_ep_sp_glb
clear_num_interact
get_num_tree_walk_loc
get_num_tree_walk_glb
calc_force_all_and_write_back
calc_force_all
calc_force_making_tree
calc_force_and_write_back
get_neighbor_list
get_epj_from_id
```

In the following, we first explain the types of Tree object and then we describe the specification of each API in the order shown above.

8.4.1 Types of Tree objects

In this section, we explain the types of Tree objects and their definitions. Almost all physical interactions in the nature can be classified into long-range force and short-range force. Based in this fact, FDPS uses different types of Tree object for the calculations of long- and short-range forces. For simplicity, we call these two types **Long-type** and **Short-type**, respectively. In FDPS Fortran interface, these two types of Tree objects are further classified into subtypes. In the following, we explain them.

8.4.1.1 Subtypes of Long-type

Long-type is classified into six subtypes depending on the way of the calculation of moments. A Tree object is called **Monopole type** if it computes only monopole moments of tree nodes taking their centers of mass as the centers of multipole expansions. If a Tree object computes up to quadrupole moments of tree nodes in the same way, we call it **Quadrupole type**. Multipole moments can be calculated taking the geometric centers of tree nodes as the expansion centers. To support this case, we prepare the following three subtypes of Tree object: **MonopoleGeometricCenter type**, **DipoleGeometricCenter type**, and **QuadrupoleGeometricCenter type**, where their names indicate the highest order of multipole moments calculated by these Tree objects. In some force calculation methods such as P^3T method[*1)], it is required for users to perform neighbor search. In order to support this, we prepare the following two subtypes of Tree object: **MonopoleWithScatterSearch type** and **QuadrupoleWithScatterSearch type**, by using which we can perform neighbor search. The names of these also indicate the highest order of multipole moments calculated in these Tree objects. In P^3M[*2)] and TreePM methods, the calculation of interactions is performed combining the direct-summation method or the tree method with the particle mesh method. In such cases, we can apply some optimization because we only have to take into account tree structure within the so-called cutoff radius, which is the distance from a particle to split force. **MonopoleWithCutoff type** is the subtype obtained by applying such optimization to Monopole-type.

These are all the subtypes of Long-type in FDPS Fortran interface. The list is available in Table. 4.3 in § 4.3.

8.4.1.2 Subtypes of Short-type

Short-type is classified into the following three subtypes depending on the type of interaction:

1. **Gather type**
   This type is used when its force decays to zero at a finite distance, and when the distance is determined by the search radius of i-particle.

2. **Scatter type**
   This type is used when its force decays to zero at a finite distance, and when the distance is determined by the search radius of j-particle.

[*1)] The abbreviation of Particle-Particle Particle-Tree.
[*2)] The abbreviation of Particle-Particle Particle-Mesh.

3. **Symmetry type**
   This type is used when its force decays to zero at a finite distance, and when the distance is determined by the larger of the search radii of i- and j-particles.

8.4.2 create_tree

```
subroutine fdps_ctrl%create_tree(tree_num,tree_info_in)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input and output	Variable receiving the identification number of a Tree object.
`tree_info_in`	character (len=*,kind=c_char)	Input	String to specify the type of a Tree object.

Returned value

None.

Function

Create a Tree object and returns its identifier. The type of the Tree object is indicated by a string `tree_info_in`. To create a Tree object for long-range force, the string must obey the following format:

```
"Long,<force_type>,<epi_type>,<epj_type>,<tree_mode>"
```

where `<tree_mode>` should be selected from Monopole, Quadrupole, MonopoleGeometricCenter, DipoleGeometricCenter, QuadrupoleGeometricCenter, MonopoleWithScatterSearch, QuadrupoleWithScatterSearch, MonopoleWithCutoff. Note that all the above keywords including `Long` are case-sensitive and users should not write the angle brackets `<>`.

To create a Tree object for short-range force, the string should be written as

```
"Short,<force_type>,<epi_type>,<epj_type>,<search_mode>"
```

where `<search_mode>` must be chosen from Gather, Scatter, or Symmetry.

For both cases, `<force_type>`,`<epi_type>`,`<epj_type>` are user-defined types. Users cannot insert space characters before or after commas and must specify the names of user-defined types in lower-case .

8.4.3 delete_tree

```
subroutine fdps_ctrl%delete_tree(tree_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.

Returned value

None.

Function

Delete a Tree object indicated by the identification number.

8.4.4 init_tree

```
subroutine fdps_ctrl%init_tree(tree_num,     &
                               nptcl,theta,  &
                               n_leaf_limit, &
                               n_group_limit)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Default	Definition
tree_num	integer(kind=c_int)	Input		Variable giving the identification number of a Tree object.
nptcl	integer(kind=c_int)	Input		Upper limit for the total number of particles.
theta	real(kind=c_float)	Input	0.7	Opening criterion for the tree.
n_leaf_limit	integer(kind=c_int)	Input	8	Maximum number of particles in a leaf cell.
n_group_limit	integer(kind=c_int)	Input	64	Maximum number of particles which share the same interaction list.

Returned value

None.

Function

Initialize a Tree object indicated by the identification number.

8.4.5 get_tree_info

```
subroutine fdps_ctrl%get_tree_info(tree_num,tree_info)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.
`tree_info`	character (len=*,kind=c_char)	Input and output	String variable receiving a string showing the type of the Tree object.

Returned value

None.

Function

Obtain a string showing the type of the Tree object indicated by the identification number.

8.4.6 get_tree_memsize

```
function fdps_ctrl%get_tree_memsize(tree_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.

Returned value

Type `integer(kind=c_long_long)`.

Function

Return the size of memory used in the Tree object in bytes.

8.4.7 get_tree_time_prof

```
subroutine fdps_ctrl%get_tree_time_prof(tree_num,prof)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.
`prof`	type(fdps_time_prof)	Input and output	Structure to receive the time spent in tree APIs.

Returned value

None.

Function

Store the execution time recorded in milliseconds for creation of local tree, creation of global tree, evaluation of force, evaluation of momenta of local tree, evaluation of momenta of global tree, creation of LET, and exchange LET to appropriate private members of `type(fdps_time_prof)`, `make_local_tree`, `make_global_tree`, `calc_force`, `calc_moment_local_tree`, `calc_moment_global_tree`, `make_LET_1st_`, `make_LET_2nd`, `exchange_LET_1st`, `exchange_LET_2nd`.

When the LET exchanging is one-step — that in a long-range tree or a scatter mode — the fields `make_LET_2nd` and `exchange_LET_2nd` are not stored.

8.4.8 clear_tree_time_prof

```
subroutine fdps_ctrl%clear_tree_time_prof(tree_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
tree_num	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.

Returned value

None.

Function

Tree object has a private variable of TimeProfile class, which is a C++ class corresponding to a derived data type fdps_time_prof in FDPS Fortran interface (for details, see the specification document of FDPS, doc_specs_cpp_en.pdf). This API sets the the member variables make_local_tree, make_global_tree, calc_force, calc_moment_local_tree, calc_moment_global_tree, make_LET_1st, make_LET_2nd, exchange_LET_1st, and exchange_LET_2nd of this private variable of the Tree object indicated by tree_num to 0. Usually, this API is used to reset time measurement.

8.4.9 get_num_interact_ep_ep_loc

```
function fdps_ctrl%get_num_interact_ep_ep_loc(tree_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.

Returned value

Type integer(kind=c_long_long).

Function

Return the number of interactions between EPI and EPJ in the MPI process calling this API.

8.4.10 get_num_interact_ep_sp_loc

```
function fdps_ctrl%get_num_interact_ep_sp_loc(tree_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.

Returned value

Type integer(kind=c_long_long).

Function

Return the number of interactions between EPI and SPJ in the MPI process calling this API.

8.4.11 get_num_interact_ep_ep_glb

```
function fdps_ctrl%get_num_interact_ep_ep_glb(tree_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.

Returned value

Type integer(kind=c_long_long).

Function

Return the number of interactions between EPI and EPJ, evaluated in all the processes.

8.4.12 get_num_interact_ep_sp_glb

```
function fdps_ctrl%get_num_interact_ep_sp_glb(tree_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.

Returned value

Type integer(kind=c_long_long).

Function

Return the number of interactions between EPI and SPJ, evaluated in all the processes.

8.4.13 clear_num_interact

```
subroutine fdps_ctrl%clear_num_interact(tree_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.

Returned value

None.

Function

Zero clear the all EP-EP/EP-SP and local/global interaction counters.

8.4.14 get_num_tree_walk_loc

```
function fdps_ctrl%get_num_tree_walk_loc(tree_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.

Returned value

Type integer(kind=c_long_long).

Function

Return the number of tree traverses for the MPI process calling this API.

8.4.15 get_num_tree_walk_glb

```
function fdps_ctrl%get_num_tree_walk_glb(tree_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.

Returned value

Type integer(kind=c_long_long).

Function

Return the number of tree traverses for all the processes.

8.4.16 calc_force_all_and_write_back

```
subroutine fdps_ctrl%calc_force_all_and_write_back(tree_num,    &
                                                   pfunc_ep_ep, &
                                                   psys_num,    &
                                                   dinfo_num,   &
                                                   list_mode)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.
`pfunc_ep_ep`	type(c_funptr)	Input	Pointer to a function calculating the interactions between EPI and EPJ.
`psys_num`	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.
`dinfo_num`	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.
`list_mode`	integer(kind=c_int)	Input	Variable determining whether interaction lists are reused (see below for details).

Returned value

None.

Function

This is an API for short-range force. It calculates all the interactions over all the particles in the ParticleSystem object and write back the result to the object. The function passed to this API must have the interface described in § 5.2.2.

The data type of the argument `list_mode` is interaction list mode type explained in § 4.5.2. This argument controls the behavior of the API with respect to the reuse of interaction lists. The value must be either of `fdps_make_list`, `fdps_make_list_for_reuse`, or `fdps_reuse_list`. The action of the API is not determined for other values. If `fdps_make_list` is given, FDPS makes interaction lists newly and performs interaction calculations using them. FDPS does not store these interaction lists. Hence, we cannot reuse these lists in

the next interaction calculation (in the next call of the API). If `fdps_make_list_for_reuse` is given, FDPS makes interaction lists newly and stores them internally for future reuse. Then, FDPS performs interaction list calculation. Therefore, we can reuse the interaction lists in the next interaction calculation. When `fdps_reuse_list` is given, FDPS performs interaction calculation using the interaction lists created previously with `fdps_make_list_for_reuse`. If `list_mode` is omitted, FDPS acts as if `fdps_make_list` is given.

```
subroutine fdps_ctrl%calc_force_all_and_write_back(tree_num,    &
                                                   pfunc_ep_ep, &
                                                   pfunc_ep_sp, &
                                                   psys_num,    &
                                                   dinfo_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.
`pfunc_ep_ep`	type(c_funptr)	Input	Pointer to a function calculating the interactions between EPI and EPJ.
`pfunc_ep_sp`	type(c_funptr)	Input	Pointer to a function calculating the interactions between EPI and SPJ.
`psys_num`	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.
`dinfo_num`	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.

Returned value

None.

Function

This is an API for long-range force and is identical to the short-range version except it takes an additional function pointer for particle-superparticle interactions.

8.4.17 calc_force_all

```
subroutine fdps_ctrl%calc_force_all(tree_num,    &
                                    pfunc_ep_ep, &
                                    psys_num,    &
                                    dinfo_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
tree_num	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.
pfunc_ep_ep	type(c_funptr)	Input	Pointer to a function calculating the interactions between EPI and EPJ.
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.
dinfo_num	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.

Returned value

None.

Function

This is an API for short-range force. This API works similarly as the API calc_force_all_and_write_back, but without writing back the result.

```
subroutine fdps_ctrl%calc_force_all(tree_num,    &
                                    pfunc_ep_ep, &
                                    pfunc_ep_sp, &
                                    psys_num,    &
                                    dinfo_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.
`pfunc_ep_ep`	type(c_funptr)	Input	Pointer to a function calculating the interactions between EPI and EPJ.
`pfunc_ep_sp`	type(c_funptr)	Input	Pointer to a function calculating the interactions between EPI and SPJ.
`psys_num`	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.
`dinfo_num`	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.

Returned value

None.

Function

This is an API for long-range force and it is identical to the short-range version except for the second function pointer in the arguments.

8.4.18 calc_force_making_tree

```
subroutine fdps_ctrl%calc_force_making_tree(tree_num,    &
                                            pfunc_ep_ep, &
                                            dinfo_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.
`pfunc_ep_ep`	type(c_funptr)	Input	Pointer to a function calculating the interactions between EPI and EPJ.
`pfunc_ep_sp`	type(c_funptr)	Input	Pointer to a function calculating the interactions between EPI and SPJ.
`dinfo_num`	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.

Returned value

None.

Function

This is an API for short-range force. This API works similarly as the API `calc_force_all_and_write_back`, but without reading the particles from the ParticleSystem object, neither writing back the result to it. This API does not take an identification number for a ParticleSystem object.

```
subroutine fdps_ctrl%calc_force_making_tree(tree_num,    &
                                            pfunc_ep_ep, &
                                            pfunc_ep_sp, &
                                            dinfo_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.
`pfunc_ep_ep`	type(c_funptr)	Input	Pointer to a function calculating the interactions between EPI and EPJ.
`pfunc_ep_sp`	type(c_funptr)	Input	Pointer to a function calculating the interactions between EPI and SPJ.
`dinfo_num`	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object.

Returned value

None.

Function

This is an API for long-range force and it is identical to the short-range version except for the second function pointer in the arguments.

8.4.19 calc_force_and_write_back

```
subroutine fdps_ctrl%calc_force_and_write_back(tree_num,   &
                                               func_ep_ep, &
                                               psys_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
tree_num	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.
pfunc_ep_ep	type(c_funptr)	Input	Pointer to a function calculating the interactions between EPI and EPJ.
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.

Returned value

None.

Function

This is an API for short-range force. This API works similarly as the API calc_force_all_and_write_back, without reading particle from a particle-system object, making the local tree, making global tree, and calculating moments of global tree. This API does not take an identification number of a DomainInfo object.

```
subroutine fdps_ctrl%calc_force_and_write_back(tree_num,    &
                                               pfunc_ep_ep, &
                                               pfunc_ep_sp, &
                                               psys_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.
`pfunc_ep_ep`	type(c_funptr)	Input	Pointer to a function calculating the interactions between EPI and EPJ.
`pfunc_ep_sp`	type(c_funptr)	Input	Pointer to a function calculating the interactions between EPI and SPJ.
`psys_num`	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object.

Returned value

None.

Function

This is an API for long-range force and it is identical to the short-range version except for the second function pointer in the arguments.

8.4.20 get_neighbor_list

```
subroutine fdps_ctrl%get_neighbor_list(tree_num, &
                                       pos,      &
                                       r_search, &
                                       num_epj,  &
                                       fptr_to_EPJ)
```

Dummy argument specificaiton

Name	Data type	I/O characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.
`pos`	type(fdps_f64vec)	Input	The position of a particle for which neighbor particles are searched.
`r_search`	real(kind=c_double)	Input	The search radius of a particle for which neighbor particles are searched.
`num_epj`	integer(kind=c_int)	Input and Output	The number of neighbor particles.
`fptr_to_EPJ`	EssentialParticleJ type, dimension(:), pointer	Input and Output	The pointer to the array of neighbor particles of EssentialParticleJ type.

Returned value

None.

Function

Using the Tree object indicated by `tree_num`, search the neighbor particles of a particle whose position and search radius are given by `pos` and `r_search`, and store the number of the neighbor particles and the pointer to the array of particles of EssentialParticleJ type. This EssentialParticleJ type must be the same as that used to create this Tree object
.

8.4.21 get_epj_from_id

```
subroutine fdps_ctrl%get_epj_from_id(tree_num, &
                                     id,       &
                                     fptr_to_EPJ)
```

Dummy argument specification

Name	Data type	I/O characteristics	Definition
`tree_num`	integer(kind=c_int)	Input	Variable giving the identification number of a Tree object.
`id`	type(kind=c_long_long)	Input	Identification number of particle you want to get.
`fptr_to_EPJ`	EssentialParticleJ type, pointer	Input and output	Pointer to a EssentialParticleJ variable.

Returned value

None.

Function

This API is usable *only when* EssentialParticleJ type has a member variable representing particle ID (a member variable corresponding to particle ID must be specified through a FDPS directive; see § 5.1.4). Note that this EssentialParticleJ type must be the same data type that was specified as an argument at the creation of tree object `tree_num` (see the description of API `create_tree`). This API sets the pointer to a EPJ whose particle ID is `id` to `fptr_to_EPJ`. If `id` is not in the list of EPJ, `fptr_to_EPJ` is set to `NULL()`(you can check the status of `fptr_to_EPJ` using the intrinsic function `associated`). The action of the API is not determined for the case that EPJ more than one have the same ID. The following is an example:

Listing 8.3: Example

```
1  integer(kind=c_long_long) :: id
2  type(essential_particle_j), pointer :: epj
3
4  call fdps_ctrl%get_epj_from_id(tree_num,id,epj)
5  if (associated(epj)) then
6     ! Do something using epj
7     write(*,*)'id␣=␣',epj%id
8  else
9     write(*,*)'epj␣is␣NULL'
10 end if
```

8.5 APIs for communication

In this section, we describe the specifications of APIs to perform MPI communications. The list of the APIs explained here is shown below:

```
get_rank
get_rank_multi_dim
get_num_procs
get_num_procs_multi_dim
get_logical_and
get_logical_or
get_min_value
get_max_value
get_sum
broadcast
get_wtime
```

In the following, we describe the specification of each API in the order above.

8.5.1 get_rank

```
integer(kind=c_int) fdps_ctrl%get_rank()
```

Dummy argument specification

None.

Returned value

Type integer(kind=c_int). Returns the rank of the calling process.

Function

Returns the rank of the calling process.

8.5.2 get_rank_multi_dim

```
integer(kind=c_int) fdps_ctrl%get_rank_multi_dim(id)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
id	integer(kind=c_int)	Input	Id of axes. x-axis:0, y-axis:1, z-axis:2.

Returned value

Type integer(kind=c_int).

Function

The rank of the calling process along id-th axis. In the case of two dimensional simulations, FDPS returns 1 for id=2.

8.5.3 get_num_procs

```
integer(kind=c_int) fdps_ctrl%get_num_procs()
```

Dummy argument specification

None.

Returned value

Type integer(kind=c_int).

Function

Returns the total number of processes.

8.5.4 get_num_procs_multi_dim

```
integer(kind=c_int) fdps_ctrl%get_num_procs_multi_dim(id)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
id	integer(kind=c_int)	Input	Id of axes. x-axis:0, y-axis:1, z-axis:2.

Returned value

Type integer(kind=c_int).

Function

The rank of the calling process along id-th axis. In the case of two dimensional simulations, FDPS returns 1 for id=2.

8.5.5 get_logical_and

```
subroutine fdps_ctrl%get_logical_and(f_in, &
                                     f_out)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
f_in	logical(kind=c_bool)	Input	Input logical value
f_out	logical(kind=c_bool)	inout	Output logical value

Returned value

None.

Function

Returns logical product of f_in over all processes.

8.5.6 get_logical_or

```
subroutine fdps_ctrl%get_logical_or(f_in, &
                                    f_out)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
f_in	logical(kind=c_bool)	Input	Input logical value
f_out	logical(kind=c_bool)	inout	Output logical value

Returned value

None.

Function

Returns logical sum of f_in over all processes.

8.5.7 get_min_value

```
subroutine fdps_ctrl%get_min_value(f_in, &
                                   f_out)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
f_in	integer(kind=c_int) integer(kind=c_long_long) real(kind=c_float) real(kind=c_double)	Input	Input value
f_out	Same as f_in	inout	Output value

Returned value

None.

Function

The minimum value of f_in of all processes is stored to f_out.

This function has alternative API in which the index associated to the minimum value is also returned. It is as follows:

```
subroutine fdps_ctrl%get_min_value(f_in, &
                                   i_in, &
                                   f_out,&
                                   i_out)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
f_in	real(kind=c_float) real(kind=c_double)	Input	Input value
i_in	integer(kind=c_int)	Input	Index associated to Input value
f_out	Same as f_in	inout	Output value
i_out	integer(kind=c_int)	inout	Index associated to Output value

Returned value

None.

Function

The minimum value of `f_in` of all processes is stored to `f_out`. In addition, the value of `i_in` corresponding to the minimum is stored to `i_out`.

8.5.8 get_max_value

```
subroutine fdps_ctrl%get_max_value(f_in, &
                                   f_out)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
f_in	integer(kind=c_int) integer(kind=c_long_long) real(kind=c_float) real(kind=c_double)	Input	Input value
f_out	Same as f_in	inout	Output value

Returned value

None.

Function

The maximum value of f_in of all processes is stored to f_out.

This function has alternative API in which the index associated to the maximum value is also returned. It is as follows:

```
subroutine fdps_ctrl%get_max_value(f_in, &
                                   i_in, &
                                   f_out,&
                                   i_out)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
f_in	real(kind=c_float) real(kind=c_double)	Input	Input value
i_in	integer(kind=c_int)	Input	Index associated to Input value
f_out	Same as f_in	inout	Output value
i_out	integer(kind=c_int)	inout	Index associated to Output value

Returned value

None.

Function

The maximum value of `f_in` of all processes is stored to `f_out`. In addition, the value of `i_in` corresponding to the maximum is stored to `i_out`.

8.5.9 get_sum

```
subroutine fdps_ctrl%get_sum(f_in, &
                             f_out)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
f_in	integer(kind=c_int) integer(kind=c_long_long) real(kind=c_float) real(kind=c_double)	Input	Input value
f_out	Same as f_in	inout	Output value

Returned value

None.

Function

Returns the sum of f_in over all processes.

8.5.10 broadcast

```
subroutine fdps_ctrl%broadcast(val, &
                               n,   &
                               src)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
`val`	integer(kind=c_int) integer(kind=c_long_long) real(kind=c_float) real(kind=c_double) or an array of them	Input	Input value
`n`	integer(kind=c_int)	Input	Number of input data
`src`	integer(kind=c_int)	Input	Process Rank of the source

Returned value

None.

Function

Broadcast `val` for the `src`-th process.

8.5.11 get_wtime

```
real(kind=c_double) fdps_ctrl%get_wtime()
```

Dummy argument specification

None.

Returned value

Type real(kind=c_double).

Function

Returns wall-clock time in seconds.

8.6 APIs for ParticleMesh object

In this section, we describe the specifications of APIs to use the extended feature "Particle Mesh". In FDPS, all data required to perform particle mesh calculation are stored in ParticleMesh object (hereafter, we call it **PM object** for simplicity). Similar to other FDPS objects, this object is managed by an identification number in FDPS Fortran interface.

This is the list of APIs to manipulate PM object:

```
create_pm
delete_pm
get_pm_mesh_num
get_pm_cutoff_radius
set_dinfo_of_pm
set_psys_of_pm
get_pm_force
get_pm_potential
calc_pm_force_only
calc_pm_force_all_and_write_back
```

In the following, we describe the specification of each API in the order above.

8.6.1 create_pm

```
subroutine fdps_ctrl%create_pm(pm_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
pm_num	integer(kind=c_int)	Input and Output	Variable to receive the identification number of a PM object.

Returned value

None.

Function

Create an PM object and return its identification number.

8.6.2 delete_pm

```
subroutine fdps_ctrl%delete_pm(pm_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
pm_num	integer(kind=c_int)	Input	Variable giving the identification number of a PM object.

Returned value

None.

Function

Delete a PM object indicated by the identification number.

8.6.3 get_pm_mesh_num

```
integer(kind=c_int) fdps_ctrl%get_pm_mesh_num()
```

Dummy argument specification

None.

Returned value

The number of the mesh per spatial dimension. Type integer(kind=c_int).

Function

Return the number of the mesh per spatial dimension used in the particle mesh calculation.

8.6.4 get_pm_cutoff_radius

```
real(kind=c_double) fdps_ctrl%get_pm_cutoff_radius()
```

Dummy argument specification

None.

Returned value

The size of cutoff radius used in the particle mesh calculation. Note that the cutoff radius is normalized by the size of the mesh interval. Type real(kind=c_double).

Function

Return the size of cutoff radius used in the particle mesh calculation.

8.6.5 set_dinfo_of_pm

```
subroutine fdps_ctrl%set_dinfo_of_pm(pm_num,dinfo_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
pm_num	integer(kind=c_int)	Input	Variable giving the identification number of a PM object.
dinfo_num	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object that is related to a ParticleSystem object for which the particle mesh calculation is performed.

Returned value

None.

Function

Set the identification number of a DomainInfo object stored in the PM object indicated by the identification number pm_num. FDPS uses this DomainInfo object to get the information on domain decomposition. Therefore, it should be the one that is related to a ParticleSystem object for which the particle mesh calculation is performed.

8.6.6 set_psys_of_pm

```
subroutine fdps_ctrl%set_psys_of_pm(pm_num,psys_num,clear)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
pm_num	integer(kind=c_int)	Input	Variable giving the identification number of a PM object.
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object for which the particle mesh calculation is performed.
clear	logical(kind=c_bool)	Input	A optional flag to determine if the previous information of particles loaded is cleared. If it is .true., the API performs clear (default: .true.).

Returned value

None.

Function

Set the identification number of a ParticleSystem object stored in the PM object indicated by the identification number pm_num. FDPS performs particle mesh calculation using this ParticleSystem object.

8.6.7 get_pm_force

```
subroutine fdps_ctrl%get_pm_force(pm_num,pos,f)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
pm_num	integer(kind=c_int)	Input	Variable giving the identification number of a PM object.
pos	real(kind=c_float), dimension(space_dim) real(kind=c_double), dimension(space_dim) type(fdps_f32vec) type(fdps_f64vec)	Input	Array or vector giving the position used to evaluate the mesh force.
f	the same data type as pos	Input and Output	Variable to receive the mesh force at the position pos.

If the macro PARTICLE_SIMULATOR_TWO_DIMENSION is defined at the compilation, space_dim is equal to 2. Otherwise, 3.

Returned value

None.

Function

Return the mesh force at the position pos. This function is thread-safe. Before calling this API, an user must perform particle mesh calculation at least once using APIs calc_pm_force_only or calc_pm_force_all_and_write_back with the same PM object.

8.6.8 get_pm_potential

```
subroutine fdps_ctrl%get_pm_potential(pm_num,pos,f)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
pm_num	integer(kind=c_int)	Input	Variable giving the identification number of a PM object.
pos	real(kind=c_float), dimension(space_dim) real(kind=c_double), dimension(space_dim) type(fdps_f32vec) type(fdps_f64vec)	Input	Array or vector giving the position used to evaluate the mesh potential.
f	real(kind=c_float)	Input and Output	Variable to receive the mesh potential at the position pos.

If the macro PARTICLE_SIMULATOR_TWO_DIMENSION is defined at the compilation, space_dim is equal to 2. Otherwise, 3.

Returned value

None.

Function

Return the mesh potential at the position pos. This function is thread-safe. Before calling this API, an user must perform Particle Mesh calculation at least once using APIs calc_pm_force_only or calc_pm_force_all_and_write_back with the same PM object.

8.6.9 calc_pm_force_only

```
subroutine fdps_ctrl%calc_pm_force_only(pm_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
pm_num	integer(kind=c_int)	Input	Variable giving the identification number of a PM object.

Returned value

None.

Function

Perform a particle mesh calculation using the PM object indicated by the identification number pm_num. In order for it to work properly, the identification numbers of ParticleSystem and DomainInfo objects must be set in advance.

8.6.10 calc_pm_force_all_and_write_back

```
subroutine fdps_ctrl%calc_pm_force_all_and_write_back(pm_num,   &
                                                      psys_num, &
                                                      dinfo_num)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
pm_num	integer(kind=c_int)	Input	Variable giving the identification number of a PM object.
psys_num	integer(kind=c_int)	Input	Variable giving the identification number of a ParticleSystem object that is used to particle mesh calculation.
dinfo_num	integer(kind=c_int)	Input	Variable giving the identification number of a DomainInfo object that is related to the ParticleSystem object above.

Returned value

None.

Function

Perform a Particle Mesh calculation using the PM, ParticleSystem, and DomainInfo objects indicated respectively by the arguments pm_num, psys_num, and dinfo_num.

8.7 Other APIs

In this section, we describe the specifications of other APIs. The list of the APIs explained here is shown below:

```
MT_init_genrand
MT_genrand_int31
MT_genrand_real1
MT_genrand_real2
MT_genrand_res53
```

In this list, APIs whose name starts with `MT` are the APIs to manipulate the pseudorandom number generator Mersenne twister.

In the following, we describe the specification of each API in the order above.

8.7.1 MT_init_genrand

```
subroutine fdps_ctrl%MT_init_genrand(s)
```

Dummy argument specification

Name	Data type	I/O Characteristics	Definition
s	integer(kind=c_int)	Input and Output	a seed used to generate pseudorandom number.

Returned value

None.

Function

Create an object for the pseudorandom number generator "Mersenne twister" and initialize it.

8.7.2 MT_genrand_int31

```
function fdps_ctrl%MT_genrand_int31()
```

Dummy argument specification

None.

Returned value

Type integer(kind=c_int).

Function

Using the pseudorandom number generator "Mersenne twister", generate an uniform integer random number on [0,0x7fffffff]-interval.

8.7.3 MT_genrand_real1

```
function fdps_ctrl%MT_genrand_real1()
```

Dummy argument specification

None.

Returned value

Type real(kind=c_double).

Function

Using the pseudorandom number generator "Mersenne twister", generate a floating-point random number on [0,1]-interval.

8.7.4 MT_genrand_real2

```
function fdps_ctrl%MT_genrand_real2()
```

Dummy argument specification

None.

Returned value

Type real(kind=c_double).

Function

Using the pseudorandom number generator "Mersenne twister", generate a floating-point random number on [0,1)-interval.

8.7.5 MT_genrand_real3

```
function fdps_ctrl%MT_genrand_real3()
```

Dummy argument specification

None.

Returned value

Type real(kind=c_double).

Function

Using the pseudorandom number generator "Mersenne twister", generate a floating-point random number on (0,1)-interval.

8.7.6 MT_genrand_res53

```
function fdps_ctrl%MT_genrand_res53()
```

Dummy argument specification

None.

Returned value

Type real(kind=c_double).

Function

Using the pseudorandom number generator "Mersenne twister", generate a floating-point random number on [0,1)-interval with 53 bit resolution. Note that the prescribed APIs `MT\_genrand\_real`x (x=1-3) use a 32-bit integer to generate a floating-point random number.

Chapter 9

Error messages

In this chapter, we describe the error messages that are output when you execute a program developed using FDPS Fortran interface. FDPS Fortran interface uses FDPS. Therefore, we first describe the error messages output by FDPS itself. Then, we describe those specific to FDPS Fortran interface.

9.1 FDPS

In this section, we describe the error messages output by FDPS itself. Please be aware the following points :

- The names of data types, functions, APIs defined in FDPS are used.
- Technical terms of C++ language are used.
- The errors that are not occurred in as far as FDPS Fortran interface is used are also described.

9.1.1 Abstract

FDPS equips features to detect compile time and run time errors. We describe errors detectable by FDPS, and how to deal with these errors. Note that unknown errors possibly happen. At that time, please inform us.

9.1.2 Compile time errors

(Not written yet)

9.1.3 Run time errors

In run time error, FDPS outputs error messages in the following format, and terminates the program by PS::Abort(-1).

PS_ERROR: *ERROR MESSAGE*
function: *FUNCTION NAME*, line: *LINE NUMBER*, file: *FILE NAME*

- *ERROR MESSAGE*

  Error message.

- *FUNCTION NAME*

  Function name in which an error happens.

- *LINE NUMBER*

  Line number in which an error happens.

- *FILE NAME*

  File name in which an error happens.

We list run time errors below.

9.1.3.1 PS_ERROR: can not open input file

This message indicates that there is no input file specified by users, when the file input functions in FDPS are called.

The following message follows the error message.

```
input file: “input file name”
```

9.1.3.2 PS_ERROR: can not open output file

This message indiates that there is no output file specified by users, when the file output functions in FDPS are called.

The following message follows the error message.

```
output file: “output file name”
```

9.1.3.3 PS_ERROR: Do not initialize the tree twice

This message indicates that PS::TreeForForce::initialize(...) is called twice for the same tree object. Users can call this function only once.

9.1.3.4 PS_ERROR: The opening criterion of the tree must be >= 0.0

This message indicates that a negative value is input into opening criterion of tree for long-distance force modes.

The following message follows the above message to the standard error.

theta_= “input value for opening criterion”
SEARCH_MODE: “data type for search mode”
Force: “Type name for tree force”
EPI: “type name for EPI”
EPJ: “type name for EPJ”
SPJ: “type name for SPJ”

9.1.3.5 PS_ERROR: The limit number of the particles in the leaf cell must be > 0

This message indicates that a negative value is input into maximum particle number for a tree leaf cell.

The following message follows the above message to the standard error.

n_leaf_limit_=”input value for the maximum particle number in tree leaf cell”
SEARCH_MODE: “data type for search mode”
Force: “Type name for tree force”
EPI: “type name for EPI”
EPJ: “type name for EPJ”
SPJ: “type name for SPJ”

9.1.3.6 PS_ERROR: The limit number of particles in ip groups msut be >= that in leaf cells

This message indicates that the maximum particle number for a leaf cell is more than the maximum particle number for a i-group particle number, and when long-distant force modes are chosen.

The following message follows the above message to the standard error.

n_leaf_limit_=”Input the maximum particle number in a leaf cell”
n_grp_limit_=”Input the maximum particle number in a i-group particle number”
SEARCH_MODE: “data type for search mode”
Force: “Type name for tree force”
EPI: “type name for EPI”
EPJ: “type name for EPJ”
SPJ: “type name for SPJ”

9.1.3.7 PS_ERROR: The number of particles of this process is beyond the FDPS limit number

This message indicates that users deal with more than 2G(G=2^{30}) particles per MPI process.

9.1.3.8 PS_ERROR: The forces w/o cutoff can be evaluated only under the open boundary condition

Thie message indicates that users set long-distance force without cutoff under periodic boundary condition.

9.1.3.9 PS_ERROR: A particle is out of root domain

This message indicates that when the user program set the root domain by using function PS::DomainInfo::setRootDomain(...), any particle is outside the root domain. Particularly under periodic boudnary condition, the user program should shift particles from outside of the root domain to inside. For this purpose, we recommend the use of function `PS::ParticleSystem::adjustPositionIntoRootDomain(...)`.

The following message follows the above message to the standard error.

position of the particle="position of particles outside"
position of the root domain="coordinates of the root domain"

9.1.3.10 PS_ERROR: The smoothing factor of an exponential moving average is must between 0 and 1.

This message indicates that users set the value which is less than 0 or grater than 1 as the smoothing factor of an exponential moving average by using function
`PS::DomainInfo::initialize(...)`.

The following message follows the above message to the standard error.

The smoothing factor of an exponential moving average="The smoothing factor"

9.1.3.11 PS_ERROR: The coodinate of the root domain is inconsistent.

This message indicates that users set the coordinate of the lower vertex to be greater than the corresponding coordinate of the higher vertex by using function
`PS::DomainInfo::setPosRootDomain(...)`.

The following message follows the above message for the standard error.

The coordinate of the low vertex of the rood domain="The coordinate of the lower vertex" The coordinate of the high vertex of the rood domain="The coordinate of the higher vertex"

9.1.3.12 PS_ERROR: Vector invalid accesse

This message indicates that users refer to an invalid element in the Vector class by using the operator "[]".

The following message follows the above message for the standard error.

Vector element=”element user reffered” is not valid

9.2 FDPS Fortran interface

In this section, we describe the error messages specific to FDPS Fortran interface.

9.2.1 Compile time errors

FDPS Fortran interface does not have a function that detects compile errors.

9.2.2 Runtime errors

In run time error, FDPS Fortran interface outputs error messages in the following format, and terminates the program by `PS_abort(-1)`.

```
*** PS_FTN_IF_ERROR ***
message:  error_message
function: function_name
file:     file_name
```

where

Name	Definition
error_message	Error message
function_name	The name of subroutine or function that detects an error.
file_name	The name of the file that defines the subroutine or function above.

We list runtime errors below.

9.2.2.1 FullParticle ‘*Name of FullParticle-type*‘ does not exist

This message indicates that a string of characters that are not any name of FullParticle types is passed to the API `create_psys`.

9.2.2.2 An invalid ParticleSystem number is received

This message indicates that an invalid identification number of ParticleSystem object is passed to APIs.

9.2.2.3 cannot create Tree ‘*Type of Tree object*‘

This message indicates that an invalid type is specified in the API `create_tree`. This error occurs, for example, when users try to create a Tree object for short-range force using EssentialParticleJ-type that have not the search radius.

9.2.2.4 An invalid Tree number is received

This message indicates that an invalid identification number of Tree object is passed to APIs.

9.2.2.5 The combination psys_num and tree_num is invalid

This messages indicates that the following cases are detected in the APIs `calc_force_all_and_write_back`,`calc_force_all`, and `calc_force_and_write_back`:

- The combination of identification numbers of ParticleSystem object and Tree objects is invalid.
- There are no ParticleSystem object and/or Tree object specified by the identification numbers passed by users.

9.2.2.6 tree_num passed is invalid

This message indicates that an invalid identification number of Tree object is passed to APIs.

9.2.2.7 EssentialParticleJ specified does not have a member variable representing the search radius or Tree specified does not support neighbor search

This messages indicates that the following cases are detected in the API `get_neighbor_list`:

- EssentialParticleJ not having the search radius was used when the Tree object specified by users was created.
- The Tree object does not support neighbor search.

The following message follows the message above.

```
Please check the definitions of EssentialParticleJ
and tree object:
 - EssentialParticleJ: EPJ_name
 - TreeInfo: tree_info
```

where

Name	Definition
EPJ_name	The name of EssentialParticleJ type used in the creation of the Tree object.
tree_info	The type of the Tree object (see Chap. 8 § 8.4)

9.2.2.8 Unknown boundary condition is specified

This message indicates that an invalid enumerator is passed to the API `set_boundary_condition`.

Chapter 10

Limitation

In this chapter, we describe the limitation and the restriction of FDPS and FDPS Fortran interface. Because FDPS Fortran interface are unconditionally restricted by the specification of FDPS, we first summarize the limitation of FDPS. Then, we describe the limitation specific to FDPS Fortran interface.

10.1 FDPS

- Safe performance of integer types unique to FDPS is ensured, only when users adopt GCC and K compilers.

10.2 FDPS Fortran interface

As of writing this, FDPS Fortran interface has the following limitations and restrictions:

- Some low-level APIs and APIs about I/O implemented in FDPS are not supported in Fortran interface.
- Execution on GPUs (Graphics Processing Units) is not supported yet.
- When user uses FDPS from C++ codes, the user can freely customize or design the moment information of superparticle, where the moment information is defined as the quantities that are required in the calculation of particle-superparticle interaction and that are calculated from physical quantities of the particles that consist of that superparticle. Examples of the moment information include monopole moment, dipole moment, and high-order multipole moments, etc. FDPS Fortran interface only supports the moment informations prepared by FDPS (see Chap. 4 § 4.3 and Chap. 8 § 8.4).

Chapter 11

Change Log

- 2016/12/26
  - Initial release of FDPS Fortran interface (as FDPS version 3.0)
- 2017/08/23
  - The default accuracy of superparticle types is changed to 64 bit (FDPS 3.0a)。
- 2017/11/01
  - A new API `sort_particle` is added. This API sorts an array of FullParticle stored in a ParticleSystem object using a given comparison function.
  - An optional flag that makes FDPS reuse interaction lists is added to API `calc_force_all_and_write_back` and `calc_force_all`.
  - A new API `get_epj_from_id` is added. This API gets the pointer to a EPJ corresponding to a given particle ID.
- 2017/11/08
  - Release FDPS 4.0
- 2017/11/17
  - A bug in API broadcast is fixed (FDPS 4.0a).