Specifications of FDPS

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

Contents

1 Concept of Document

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

This document is structured as follows.

In sections 2, 3, and 4, we present prerequisites for programing with FDPS. In section 2, we show the concept of FDPS. In section 3, we present the file configuration of FDPS. In section 4, we describe how to compile simulation codes with FDPS.

In sections 5, 6, 7, 8, and 9, we present necessary information to develop simulation codes with FDPS. In section 5, we describe the structure of namespaces unused in FDPS. In section 6, we described data types defined in FDPS. In section 7, we introduce user-defined classes and function objects necessary for developing codes with FDPS. 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 10, 11, 12, and 13, we present information useful for troubleshooting. In section 10, we describe error messages. In section 11, we present known bugs. In section 12, we describe the limitation of FDPS. In section 13, we present our current system for user supports.

Finally, we describe the license of FDPS in section 14, and the change log of this document in section 15.

2 Concept of FDPS

In this section, 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 Purpose of the development 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 making use of 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,

$$\frac{d\boldsymbol{u}_i}{dt} = \sum_j f(\boldsymbol{u}_i, \boldsymbol{u}_j) + \sum_s g(\boldsymbol{u}_i, \boldsymbol{v}_s), \tag{1}$$

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 particle j. The vector $\boldsymbol{v}_s$ is the quantity vector of a superparticle which is the representative particle of a group of particles 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. (1) is non-zero 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. (1) in 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. (1) for all of its i-particle and obtains $d\boldsymbol{u}_i/dt$.

4. Each MPI process performes the time integration of its i-particles by using 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 its 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 to realize massively parallel program, by utilizing FDPS APIs.

2.2.3 Users' tasks

Users's tasks are as follows.

- Define a particle (section 7). Users need to specify quantities of particles, *i.e.* the quantity vector $\boldsymbol{u}_i$ in eq. (1), which contains quantities such as position, velocity, acceleration, chemical composition, and particle size.
- Define interaction (section 7). Users need to specify the interaction between particles, *i.e.* the function f and g in eq. (1), such as gravity, Coulomb force, and pressure.
- Call FDPS APIs (setion 8 and 9).
- Time integration of particles, diagnostic, output etc.

2.2.4 Complement

The right hand side of eq. (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

We overview the structure of a simple simulation code written using 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. 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 transfered 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 transfered 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 transferd from DomainInfo and ParticleSystem classes (green text and arrows).

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

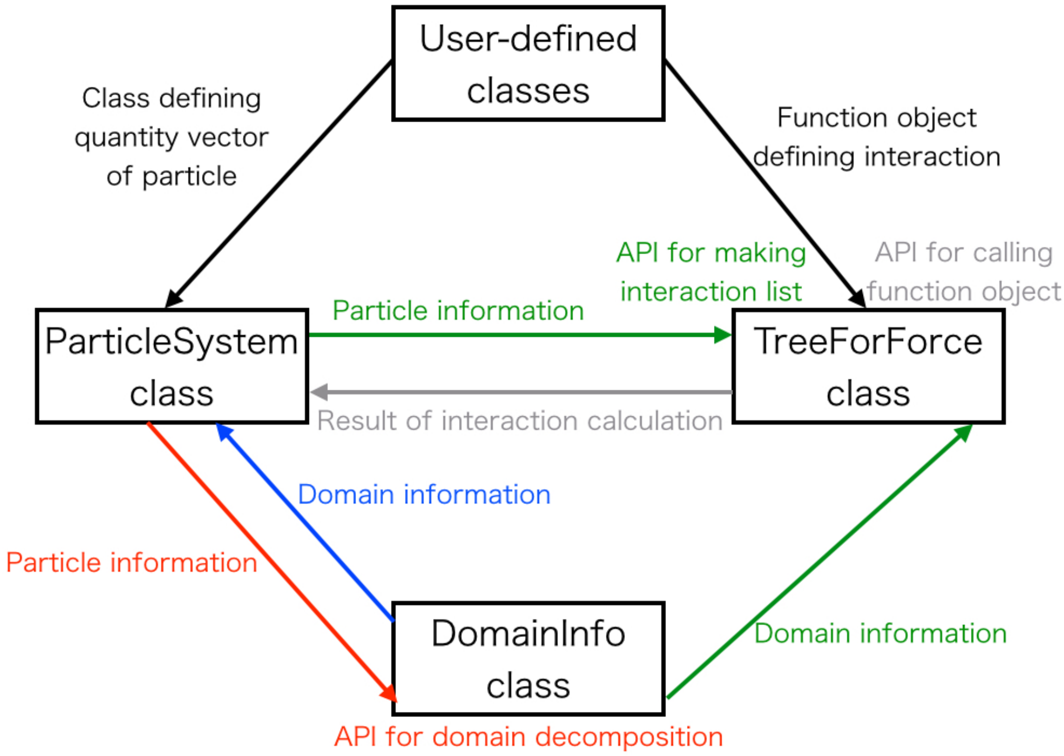

Figure 1: Illustration of module interface and data transfer.

3 Files and directries

3.1 Summary

We describe the directory strutcures and files of FDPS file distribution.

3.2 Documents

Document files are in directory `doc`. Files `doc_tutorial_e.pdf` and `doc_specs_en.pdf` are tutorial and specification, respectively.

3.3 Source files

Source files are in directory `src`. Files related to the standard features are directly under directory `src`. When users include header file `particle_simulator.hpp` in their source files, the standard features become available.

3.3.1 Extended features

Source files related to extended features are in directories in directory `src`. The extended features include Particle Mesh and Phantom-GRAPE version x86.

3.3.1.1 Particle Mesh scheme

Source files related to the Particle Mesh scheme are in directory `particle_mesh`. When users edit `Makefile`, and run `make`, they get header file `particle_mesh_class.hpp` and library `libpm.a`. By including this header file, and linking this library, they can use the features of the Particle Mesh scheme.

3.3.1.2 x86 version Phantom-GRAPE

Source files related to x86 version Phantom-GRAPE are in directory `src/phantom_GRAPE_x86`. This directory has directories containing Phantom-GRAPE libraries for low precision, for low precision with cutoff, and for high precision.

3.3.1.2.1 Low precision

Files are in directory `src/phantom_GRAPE_x86/G5/newton/libpg5`. Users should edit `Makefile` in this directory, and run `make`, to get library `libpg5.a`. Users include header file `gp5util.h` in this directory, and to link library `libpg5.a`, to use this Phantom-GRAPE.

3.3.1.2.2 Low precision with cutoff

Files are in directory `src/phantom_GRAPE_x86/G5/table`. Users should edit `Makefile` in this directory, and run `make`, to get library `libpg5.a`. Users include header file `gp5util.h` in this directory, and to link library `libpg5.a`, to use this Phantom-GRAPE.

3.3.1.2.3 High precision

Files are in directory `src/phantom_GRAPE_x86/G6/libavx`. Users should edit `Makefile` in this directory, and run `make`, to get library `libg6avx.a`. Users include header file `gp6util.h` in this directory, and to link library `libg6avx.a`, to use this Phantom-GRAPE.

3.4 Test codes

Test codes are in directory `tests`. When users run `make check` in this directory, the test suite are executed.

3.5 Sample codes

Sample codes are in directory `sample`. There are two sample codes: gravitational N-body simulation, and SPH simulation.

3.5.1 Gravitational N-body simulation

Source files are in directory `sample/nbody`. In file `doc_tutorial.pdf`, we describe how to use this code.

3.5.2 SPH Simulation

Source files are in directory `sample/sph`. In file `doc_tutorial.pdf`, we describe how to use this code.

4 Macros at compile time

4.1 Summary

Some of the features of FDPS should be supecified at the compile time. They are (a)coordinate sysmtem and (b)method of parallelization.

4.2 Coordinate system

4.2.1 Summary

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

4.2.2 3D Cartesian coordinate system

3D Cartesian coordinate system is used by default.

4.2.3 2D Cartesian coordinate system

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

4.3 Parallel processing

4.3.1 Summary

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

4.3.2 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.

4.3.3 MPI

MPI is disabled by default. If macro `PARTICLE_SIMULATOR_MPI_PARALLEL` is defined.

5 Namespace

5.1 Summary

In this section, we describe the namespaces used in FDPS. All the FDPS APIs are under namespace `ParticleSimulator`. In the following, we show APIs directly under `ParticleSimulator`, and namespaces nested under `ParticleSimulator`.

5.2 ParticleSimulator

The standard features of FDPS are in namespace `ParticleSimulator`.

Namespace `ParticleSimulator` is abbreviated to `PS` as follows.

```
namespace PS = ParticleSimulator;
```

In this document, we use this abbreviation.

Extended features of FDPS are grouped under nested namespaces. Currently, `ParticleMesh` features is under nested namespace `ParticleMesh`. In the following, we describe this namespace containing the extended features.

5.2.1 ParticleMesh

The features of Particle Mesh are under namespace `PartcielMesh`. This namespace is abbreviated to `PM` as follows.

```
ParticleSimulator {
    ParticleMesh {
    }
    namespace PM = ParticleMesh;
}
```

In this document, we use this abbreviation.

6 Data Type

6.1 Summary

In this chapter we define data types used in FDPS.The data types include integer types, floating point types, vector types, symmetric matrix types, PS::SEARCH_MODE type, and enumerated types. We recommend users to use these data types. The integer and floating point types can be replaced with data types available in C/C++ languages. PS::SEARCH_MODE and enumerated types must be used. In the following, we describe these data types.

6.2 Integer type

6.2.1 Overview

The integer types are PS::S32, PS::S64, PS::U32, and PS::U64. We describe these data types in this section.

6.2.2 PS::S32

PS::S32, which is 32-bit signed integer, is defined as follows.

Listing 1: S32

```
namespace ParticleSimulator {
    typedef int S32;
}
```

6.2.3 PS::S64

PS::S64, which is 64-bit signed integer, is defined as follows.

Listing 2: S64

```
namespace ParticleSimulator {
    typedef long long int S64;
}
```

6.2.4 PS::U32

PS::U32, which is 32-bit unsigned integer, is defined as follows.

Listing 3: U32

```
namespace ParticleSimulator {
    typedef unsigned int U32;
}
```

6.2.5 PS::U64

PS::U64, which is 64-bit unsigned integer, is defined as follows.

Listing 4: U64

```
namespace ParticleSimulator {
    typedef unsigned long long int U64;
}
```

6.3 Floating point type

6.3.1 Abstract

The floating point types are PS::F32 and PS::F64. We described these data types in this section.

6.3.2 PS::F32

PS::F32, which is 32-bit floating point number, is defined as follows.

Listing 5: F32

```
namespace ParticleSimulator {
    typedef float F32;
}
```

6.3.3 PS::F64

PS::F64, which is 64-bit floating point number, is defined as follows.

Listing 6: F64

```
namespace ParticleSimulator {
    typedef double F64;
}
```

6.4 Vector type

6.4.1 Abstract

The vector types are `PS::Vector2` (2D vector) and `PS::Vector3` (3D vector). We describe these vector types first. These are template classes, which can take basic datatypes as `F32` or `F64` as template arguments. We then present wrappers for these vector types.

6.4.2 PS::Vector2

`PS::Vector2` has two components: `x` and `y`. We define various APIs and operators for these components. In the following, we list them.

Listing 7: Vector2

```
namespace ParticleSimulator{
    template <typename T>
    class Vector2{
    public:
        // Two member variables
        T x, y;

        // Constructors
        Vector2();
        Vector2(const T _x, const T _y) : x(_x), y(_y) {}
        Vector2(const T s) : x(s), y(s) {}
        Vector2(const Vector2 & src) : x(src.x), y(src.y) {}

        // Assignment operator
        const Vector2 & operator = (const Vector2 & rhs);

        //[] operators
        const T & opertor[](const int i);
        T & operator[](const int i);

        // Addition and subtraction
        Vector2 operator + (const Vector2 & rhs) const;
        const Vector2 & operator += (const Vector2 & rhs);
        Vector2 operator - (const Vector2 & rhs) const;
        const Vector2 & operator -= (const Vector2 & rhs);

        // Scalar multiplication
        Vector2 operator * (const T s) const;
        const Vector2 & operator *= (const T s);
        friend Vector2 operator * (const T s,
                                   const Vector2 & v);
        Vector2 operator / (const T s) const;
        const Vector2 & operator /= (const T s);

        // Inner product (scalar product)
        T operator * (const Vector2 & rhs) const;

        // Outer product (vector product).
        // Caution: the return value is a scalar value.
        T operator ^ (const Vector2 & rhs) const;

        // Typecast to Vector2<U>
        template <typename U>
        operator Vector2<U> () const;
    };
```

```
}
namespace PS = ParticleSimulator;
```

6.4.2.1 Constructor

```
template<typename T>
PS::Vector2<T>::Vector2()
```

- **Argument**

  None.

- **Feature**

  Default constructor. The member variables `x` and `y` are initialized as 0.

```
template<typename T>
PS::Vector2<T>::Vector2(const T_x, const T_y)
```

- **Argument**

  _x: Input. Type `const T`.

  _y: Input. Type `const T`.

- **Feature**

  Values of members `x` and `y` are set to values of arguments `_x` and `_y`, respectively.

```
template<typename T>
PS::Vector2<T>::Vector2(const T s);
```

- **Argument**

  `s`: Input. Type `const T`.

- **Feature**

  Values of both of members `x` and `y` are set to the value of argument `s`.

6.4.2.2 Copy constructor

```
template<typename T>
PS::Vector2<T>::Vector2(const PS::Vector2<T> & src)
```

- **Argument**

  `src`: Input. Type `const PS::Vector2<T>`.

- **Feature**

  Copy constructor. The new variable will have the same value as `src`.

6.4.2.3 Assignment operator

```
template<typename T>
const PS::Vector2<T> & PS::Vector2<T>::operator =
                        (const PS::Vector2<T> & rhs);
```

- **Argument**

  `rhs`: Input. Type `const PS::Vector2<T> &`.

- **Return value**

  Type `const PS::Vector2<T> &`. Assigns the values of components of `rhs` to its components, and returns the vector itself.

6.4.2.4 [] operators

```
template<typename T>
const T & PS::Vector2<T>::operator[]
                        (const int i);
```

- **Argument**

  `i`: input. Type `const int`.

- **Return value**

  Type `const <T> &`. It returns i-component of the vector.

```
template<typename T>
T & PS::Vector2<T>::operator[]
                        (const int i);
```

- **Argument**

  `i`: input. Type `const int`.

- **Return value**

  Type `<T> &`. It returns i-component of the vector.

6.4.2.5 Addition and subtraction

```
template<typename T>
PS::Vector2<T> PS::Vector2<T>::operator +
               (const PS::Vector2<T> & rhs) const;
```

- **Argument**

  `rhs`: Input. Type `const PS::Vector2<T> &`.

- **Return value**

  Type `PS::Vector2<T>`. Add the components of `rhs` and its components, and return the results.

```
template<typename T>
const PS::Vector2<T> & PS::Vector2<T>::operator +=
                       (const PS::Vector2<T> & rhs);
```

- **Argument**

  `rhs`: Input. Type `const PS::Vector2<T> &`.

- **Return value**

  Type `const PS::Vector2<T> &`. Add the components of `rhs` to its components, and return itself (lhs is changed).

```
template<typename T>
PS::Vector2<T> PS::Vector2<T>::operator -
               (const PS::Vector2<T> & rhs) const;
```

- **Argument**

  `rhs`: Input. Type `const PS::Vector2<T> &`.

- **Return value**

  Type `PS::Vector2<T>`. Subtract the components of `rhs` from its components, and return the results.

```
template<typename T>
const PS::Vector2<T> & PS::Vector2<T>::operator -=
                       (const PS::Vector2<T> & rhs);
```

- **Argument**

  `rhs`: Input. Type `const PS::Vector2<T> &`.

- **Return value**

  Type `const PS::Vector2<T> &`. Subtract the components of `rhs` from its components, and return itself (lhs is changed).

6.4.2.6 Scalar multiplication

```
template<typename T>
PS::Vector2<T> PS::Vector2<T>::operator * (const T s) const;
```

- **Argument**

  `s`: Input. Type `const T`.

- **Return value**

  Type `PS::Vector2<T>`. Multiply its components by `s`, and return the results.

```
template<typename T>
const PS::Vector2<T> & PS::Vector2<T>::operator *= (const T s);
```

- **Argument**

  `rhs`: Input. Type `const T`.

- **Return value**

  Type `const PS::Vector2<T> &`. Multiply its components by `s`, and return itself (lhs is changed).

```
template<typename T>
PS::Vector2<T> PS::Vector2<T>::operator / (const T s) const;
```

- **Argument**

  `s`: Input. Type `const T`.

- **Return value**

  Type `PS::Vector2<T>`. Divide its components by `s`, and return the results.

```
template<typename T>
const PS::Vector2<T> & PS::Vector2<T>::operator /= (const T s);
```

- **Argument**

  `rhs`: Input. Type `const T`.

- **Return value**

  Type `const PS::Vector2<T> &`. Divide its components by `s`, and return itself (lhs is changed).

6.4.2.7 Inner and outer products

```
template<typename T>
T PS::Vector2<T>::operator * (const PS::Vector2<T> & rhs) const;
```

- **Argument**

  `rhs`: Input. Type `const PS::Vector2<T> &`.

- **Return value**

  Type `T`. Take inner product of it and `rhs`, and return the result.

```
template<typename T>
T PS::Vector2<T>::operator ^ (const PS::Vector2<T> & rhs) const;
```

- **Argument**

  `rhs`: Input. Type `const PS::Vector2<T> &`.

- **Return value**

  Type `T`. Take outer product of it and `rhs`, and return the results.

6.4.2.8 Typecast to Vector2<U>

```
template<typename T>
template <typename U>
PS::Vector2<T>::operator PS::Vector2<U> () const;
```

- **Argument**

  None.

- **Return value**

  Type `const PS::Vector2<U>`.

- **Feature**

  Typecast from type `const PS::Vector2<T>` to type `const PS::Vector2<U>`.

6.4.3 PS::Vector3

`PS::Vector3` has three components: `x`, `y`, and `z`. We define various APIs and operators for these components. In the following, we list them.

Listing 8: Vector3

```
namespace ParticleSimulator{
    template <typename T>
    class Vector3{
    public:
        // Three member variables
        T x, y, z;

        // Constructor
        Vector3() : x(T(0)), y(T(0)), z(T(0)) {}
        Vector3(const T _x, const T _y, const T _z) : x(_x), y(
            _y), z(_z) {}
        Vector3(const T s) : x(s), y(s), z(s) {}
        Vector3(const Vector3 & src) : x(src.x), y(src.y), z(
            src.z) {}

        // Assignment operator
        const Vector3 & operator = (const Vector3 & rhs);

        //[] operators
        const T & opertor[](const int i);
        T & operator[](const int i);

        // Addition and subtraction
        Vector3 operator + (const Vector3 & rhs) const;
        const Vector3 & operator += (const Vector3 & rhs);
        Vector3 operator - (const Vector3 & rhs) const;
        const Vector3 & operator -= (const Vector3 & rhs);

        // Scalar multiplication
        Vector3 operator * (const T s) const;
        const Vector3 & operator *= (const T s);
        friend Vector3 operator * (const T s, const Vector3 & v
            );
        Vector3 operator / (const T s) const;
        const Vector3 & operator /= (const T s);

        // Inner product (scalar product)
        T operator * (const Vector3 & rhs) const;

        // Outer product (vector product).
        Vector3 operator ^ (const Vector3 & rhs) const;

        // Typecast to Vector3<U>
        template <typename U>
        operator Vector3<U> () const;
```

```
    };
}
```

6.4.3.1 Constructor

```
template<typename T>
PS::Vector3<T>::Vector3()
```

- **Argument**

  None.

- **Feature**

  Default constructor. The member variables `x`, `y` and `z` are initialized as 0.

```
template<typename T>
PS::Vector3<T>::Vector3(const T _x, const T _y, const T _z)
```

- **Argument**

  _x: Input. Type `const T`.

  _y: Input. Type `const T`.

  _z: Input. Type `const T`.

- **Feature**

  Values of members `x`, `y` and `z` are set to values of arguments `_x`, `_y` and `_z`, respectively.

```
template<typename T>
PS::Vector3<T>::Vector3(const T s);
```

- **Argument**

  `s`: Input. Type `const T`.

- **Feature**

  Values of members `x`, `y` and `z` are set to the value of argument `s`.

6.4.3.2 Copy constructor

```
template<typename T>
PS::Vector3<T>::Vector3(const PS::Vector3<T> & src)
```

- **Argument**

  src: Input. Type `const PS::Vector3<T> &`.

- **Feature**

  Copy constructor. The new variable will have the same value as `src`.

6.4.3.3 Assignment operator

```
template<typename T>
const PS::Vector3<T> & PS::Vector3<T>::operator =
                       (const PS::Vector3<T> & rhs);
```

- **Argument**

  rhs: Input. Type `const PS::Vector3<T> &`.

- **Return value** Type `const PS::Vector3<T> &`. Assigns the values of components of `rhs` to its components, and returns the vector itself.

6.4.3.4 [] operators

```
template<typename T>
const T & PS::Vector3<T>::operator[]
                       (const int i);
```

- **Argument**

  i: input. Type `const int`.

- **Return value**

  Type `const <T> &`. It returns ith-component of the vector.

```
template<typename T>
T & PS::Vector3<T>::operator[]
                       (const int i);
```

- **Argument**

  `i`: input. Type `const int`.

- **Return value**

  Type <T> `&`. It returns ith-component of the vector.

6.4.3.5 Addition and subtraction

```
template<typename T>
PS::Vector3<T> PS::Vector3<T>::operator +
               (const PS::Vector3<T> & rhs) const;
```

- **Argument**

  `rhs`: Input. Type `const PS::Vector3<T> &`.

- **Return value**

  Type `PS::Vector3<T>`. Add the components of `rhs` and its components, and return the results.

```
template<typename T>
const PS::Vector3<T> & PS::Vector3<T>::operator +=
                       (const PS::Vector3<T> & rhs);
```

- **Argument**

  `rhs`: Input. Type `const PS::Vector3<T> &`.

- **Return value**

  Type `const PS::Vector3<T> &`. Add the components of `rhs` to its components, and return itself (lhs is changed).

```
template<typename T>
PS::Vector3<T> PS::Vector3<T>::operator -
               (const PS::Vector3<T> & rhs) const;
```

- **引数**

  `rhs`: Input. Type `const PS::Vector3<T> &`.

- **Return value**

  Type `PS::Vector3<T>`. Subtract the components of `rhs` from its components, and return the results.

```
template<typename T>
const PS::Vector3<T> & PS::Vector3<T>::operator -=
                         (const PS::Vector3<T> & rhs);
```

- **Argument**

  rhs: Input. Type const PS::Vector3<T> &.

- **Return value**

  Type const PS::Vector3<T> &. Subtract the components of rhs from its components, and return itself (lhs is changed).

6.4.3.6 Scalar multiplication

```
template<typename T>
PS::Vector3<T> PS::Vector3<T>::operator * (const T s) const;
```

- **Argument**

  s: Input. Type const T.

- **Return value**

  Type PS::Vector3<T>. Multiply its components by s, and return the results.

```
template<typename T>
const PS::Vector3<T> & PS::Vector3<T>::operator *= (const T s);
```

- **Argument**

  rhs: Input. Type const T.

- **Return value**

  Type const PS::Vector3<T> &. Multiply its components by s, and return itself (lhs is changed).

```
template<typename T>
PS::Vector3<T> PS::Vector3<T>::operator / (const T s) const;
```

- **Argument**

  s: Input. Type const T.

- **Return value**

  Type `PS::Vector3<T>`. Divide its components by `s`, and return the results.

```
template<typename T>
const PS::Vector3<T> & PS::Vector3<T>::operator /= (const T s);
```

- **Argument**

  `rhs`: Input. Type `const T`.

- **Return value**

  Type `const PS::Vector3<T> &`. Divide its components by `s`, and return itself (lhs is changed).

6.4.3.7 Inner and outer product

```
template<typename T>
T PS::Vector3<T>::operator * (const PS::Vector3<T> & rhs) const;
```

- **Argument**

  `rhs`: Input. Type `const PS::Vector3<T> &`.

- **Return value**

  Type `T`. Take inner product of it and `rhs`, and return the result.

```
template<typename T>
T PS::Vector3<T>::operator ^ (const PS::Vector3<T> & rhs) const;
```

- **Argument**

  `rhs`: Input. Type `const PS::Vector3<T> &`.

- **Return value**

  Type `T`. Take outer product of it and `rhs`, and return the results.

6.4.3.8 Typecast to Vector3<U>

```
template<typename T>
template <typename U>
PS::Vector3<T>::operator PS::Vector3<U> () const;
```

- **Argument**

  None.

- **Return value**

  Type `const PS::Vector3<U>`.

- **Feature**

  Typecast from type `const PS::Vector3<T>` to `const PS::Vector3<U>`.

6.4.4 Wrappers

The wrappers of vector types are defined as follows.

Listing 9: vectorwrapper

```
namespace ParticleSimulator{
    typedef Vector2<F32> F32vec2;
    typedef Vector3<F32> F32vec3;
    typedef Vector2<F64> F64vec2;
    typedef Vector3<F64> F64vec3;
#ifdef PARTICLE_SIMULATOR_TOW_DIMENSION
    typedef F32vec2 F32vec;
    typedef F64vec2 F64vec;
#else
    typedef F32vec3 F32vec;
    typedef F64vec3 F64vec;
#endif
}
```

PS::F32vec2, PS::F32vec3, PS::F64vec2, and PS::F64vec3 are, respectively, 2D vector in single precision, 3D vector in single presicion, 2D vector in double precision, and 3D vector in double precision. If users set 2D (3D) coordinate system, PS::F32vec and PS::F64vec is wrappers of PS::F32vec2 and PS::F64vec2 (PS::F32vec3 and PS::F64vec3).

6.5 Symmetric matrix type

6.5.1 Abstract

Symmetric matrix types are `PS::MatrixSym2` (2x2 matrix) and `PS::MatrixSym3` (3x3 matrix). We describe these matrix types first and then, we present wrappers for these matrix types.

6.5.2 PS::MatrixSym2

`PS::MatrixSym2` has three components: `xx`, `yy`, and `xy`. We define various APIs and operators for these components. In the following, we list them.

Listing 10: MatrixSym2

```
namespace ParticleSimulator{
    template<class T>
    class MatrixSym2{
    public:
        // Three member variables
        T xx, yy, xy;

        // Constructors
        MatrixSym2() : xx(T(0)), yy(T(0)), xy(T(0)) {}
        MatrixSym2(const T _xx, const T _yy, const T _xy)
            : xx(_xx), yy(_yy), xy(_xy) {}
        MatrixSym2(const T s) : xx(s), yy(s), xy(s){}
        MatrixSym2(const MatrixSym2 & src) : xx(src.xx), yy(src
             .yy), xy(src.xy) {}

        // Assignment operator
        const MatrixSym2 & operator = (const MatrixSym2 & rhs);

        // Addition and subtraction
        MatrixSym2 operator + (const MatrixSym2 & rhs) const;
        const MatrixSym2 & operator += (const MatrixSym2 & rhs)
             const;
        MatrixSym2 operator - (const MatrixSym2 & rhs) const;
        const MatrixSym2 & operator -= (const MatrixSym2 & rhs)
             const;

        // Trace
        T getTrace() const;

        // Typecast to MatrixSym2<U>
        template <typename U>
        operator MatrixSym2<U> () const;
    }
}
namespace PS = ParticleSimulator;
```

6.5.2.1 Constructor

```
template<typename T>
PS::MatrixSym2<T>::MatrixSym2();
```

- **Argument**

  None.

- **Feature**

  Default constructor. The member variables `xx`, `yy` and `xy` are initialized as 0.

```
template<typename T>
PS::MatrixSym2<T>::MatrixSym2
            (const T _xx,
             const T _yy,
             const T _xy);
```

- **Argument**

  `_xx`: Input. Type `const T`.

  `_yy`: Input. Type `const T`.

  `_xy`: Input. Type `const T`.

- **Feature**

  Values of members `xx`, `yy` and `xy` are set to values of arguments `_xx`, `_yy` and `_xy`, respectively.

```
template<typename T>
PS::MatrixSym2<T>::MatrixSym2(const T s);
```

- **Argument**

  `s`: Input. Type `const T`.

- **Feature**

  Values of members `xx`, `yy` and `xy` are set to the value of argument `s`.

6.5.2.2 Copy constructor

```
template<typename T>
PS::MatrixSym2<T>::MatrixSym2(const PS::MatrixSym2<T> & src)
```

- **Argument**

  `src`: Input. Type `const PS::MatrixSym2<T> &`.

- **Feature**

  Copy constructor. The new variable will have the same value as `src`.

6.5.2.3 Assignment operator

```
template<typename T>
const PS::MatrixSym2<T> & PS::MatrixSym2<T>::operator =
                       (const PS::MatrixSym2<T> & rhs);
```

- **Argument**

  `rhs`: Input. Type `const PS::MatrixSym2<T> &`.

- **Return value**

  Type `const PS::MatrixSym2<T> &`. Assigns the values of components of `rhs` to its components, and returns the vector itself.

6.5.2.4 Addition and subtraction

```
template<typename T>
PS::MatrixSym2<T> PS::MatrixSym2<T>::operator +
               (const PS::MatrixSym2<T> & rhs) const;
```

- **Argument**

  `rhs`: Input. Type `const PS::MatrixSym2<T> &`.

- **Return value**

  Type `PS::MatrixSym2<T>`. Add the components of `rhs` and its components, and return the results.

```
template<typename T>
const PS::MatrixSym2<T> & PS::MatrixSym2<T>::operator +=
                       (const PS::MatrixSym2<T> & rhs);
```

- **Argument**

  `rhs`: Input. Type `const PS::MatrixSym2<T> &`.

- **Return value**

  Type `const PS::MatrixSym2<T> &`. Add the components of `rhs` to its components, and return itself (lhs is changed).

```
template<typename T>
PS::MatrixSym2<T> PS::MatrixSym2<T>::operator -
               (const PS::MatrixSym2<T> & rhs) const;
```

- **Argument**

  rhs: Input. Type const PS::MatrixSym2<T> &.

- **Return value**

  Type PS::MatrixSym2<T>. Subtract the components of rhs from its components, and return the results.

```
template<typename T>
const PS::MatrixSym2<T> & PS::MatrixSym2<T>::operator -=
                        (const PS::MatrixSym2<T> & rhs);
```

- **Argument**

  rhs: Input. Type const PS::MatrixSym2<T> &.

- **Return value**

  Type const PS::MatrixSym2<T> &. Subtract the components of rhs from its components, and return itself (lhs is changed).

6.5.2.5 Trace

```
template<typename T>
T PS::MatrixSym2<T>::getTrace() const;
```

- **Argument**

  None.

- **Return value**

  Type T.

- **Feature**

  Calculate the trace, and return the result.

6.5.2.6 Typecast to MatrixSym2<U>

```
template<typename T>
template<typename U>
PS::MatrixSym2<T>::operator PS::MatrixSym2<U> () const;
```

- **Argument**

  None.

- **Return value**

  Type `const PS::MatrixSym2<U>`.

- **Feature**

  Typecast from type `const PS::MatrixSym2<T>` to `const PS::MatrixSym2<U>`.

6.5.3 PS::MatrixSym3

`PS::MatrixSym3` has six components: `xx`, `yy`, `zz`, `xy`, `xz`, and `yz`. We define various APIs and operators for these components. In the following, we list them.

Listing 11: MatrixSym3

```
namespace ParticleSimulator{
    template<class T>
    class MatrixSym3{
    public:
        // Six member variables
        T xx, yy, zz, xy, xz, yz;

        // Constructors
        MatrixSym3() : xx(T(0)), yy(T(0)), zz(T(0)),
                       xy(T(0)), xz(T(0)), yz(T(0)) {}
        MatrixSym3(const T _xx, const T _yy, const T _zz,
                   const T _xy, const T _xz, const T _yz )
                       : xx(_xx), yy(_yy), zz(_zz),
                       xy(_xy), xz(_xz), yz(_yz) {}
        MatrixSym3(const T s) : xx(s), yy(s), zz(s),
                                xy(s), xz(s), yz(s) {}
        MatrixSym3(const MatrixSym3 & src) :
            xx(src.xx), yy(src.yy), zz(src.zz),
            xy(src.xy), xz(src.xz), yz(src.yz) {}

        // Assignment operator
        const MatrixSym3 & operator = (const MatrixSym3 & rhs);

        // Addition and subtraction
        MatrixSym3 operator + (const MatrixSym3 & rhs) const;
        const MatrixSym3 & operator += (const MatrixSym3 & rhs)
              const;
        MatrixSym3 operator - (const MatrixSym3 & rhs) const;
        const MatrixSym3 & operator -= (const MatrixSym3 & rhs)
              const;

        // Trace
        T getTrace() const;

```

```
        // Typecast to MatrixSym3<U>
        template <typename U>
        operator MatrixSym3<U> () const;
    }
}
namespace PS = ParticleSimulator;
```

6.5.3.1 Constructor

```
template<typename T>
PS::MatrixSym3<T>::MatrixSym3();
```

- **Argument**

  None.

- **Feature**

  Default constructor. All the member variables are initialized as 0.

```
template<typename T>
PS::MatrixSym3<T>::MatrixSym3(const T _xx,
                              const T _yy,
                              const T _zz,
                              const T _xy,
                              const T _xz,
                              const T _yz);
```

- **Argument**

  _xx: Input. Type `const T`.

  _yy: Input. Type `const T`.

  _zz: Input. Type `const T`.

  _xy: Input. Type `const T`.

  _xz: Input. Type `const T`.

  _yz: Input. Type `const T`.

- **Feature**

  Values of members `xx`, `yy`, `zz`, `xy`, `xz` and `yz` are set to values of arguments _xx, _yy, _zz, _xy, `xz` and _yz, respectively.

```
template<typename T>
PS::MatrixSym3<T>::MatrixSym3(const T s);
```

- **Argument**

  s: Input. Type `const T`.

- **Feature**

  Values of members are set to the value of argument `s`.

6.5.3.2 Copy constructor

```
template<typename T>
PS::MatrixSym3<T>::MatrixSym3(const PS::MatrixSym3<T> & src)
```

- **Argument**

  `src`: Input. Type const `PS::MatrixSym3<T> &`.

- **Feature**

  Copy constructor. The new variable will have the same value as `src`.

6.5.3.3 Assignment operator

```
template<typename T>
const PS::MatrixSym3<T> & PS::MatrixSym3<T>::operator =
                        (const PS::MatrixSym3<T> & rhs);
```

- **Argument**

  `rhs`: Input. Type `const PS::MatrixSym3<T> &`.

- **Return value**

  Type `const PS::MatrixSym3<T> &`. Assigns the values of components of `rhs` to its components, and returns the vector itself.

6.5.3.4 Addition and subtraction

```
template<typename T>
PS::MatrixSym3<T> PS::MatrixSym3<T>::operator +
                (const PS::MatrixSym3<T> & rhs) const;
```

- **Argument**

  `rhs`: Input. Type `const PS::MatrixSym3<T> &`.

- **Return value**

  Type `PS::MatrixSym3<T>`. Add the components of `rhs` and its components, and return the results.

```
template<typename T>
const PS::MatrixSym3<T> & PS::MatrixSym3<T>::operator +=
                        (const PS::MatrixSym3<T> & rhs);
```

- **Argument**

  `rhs`: Input. Type `const PS::MatrixSym3<T> &`.

- **Return value**

  Type `const PS::MatrixSym3<T> &`. Add the components of `rhs` to its components, and return itself (lhs is changed).

```
template<typename T>
PS::MatrixSym3<T> PS::MatrixSym3<T>::operator -
                (const PS::MatrixSym3<T> & rhs) const;
```

- **Argument**

  `rhs`: Input. Type `const PS::MatrixSym3<T> &`.

- **Return value**

  Type `PS::MatrixSym3<T>`. Subtract the components of `rhs` from its components, and return the results.

```
template<typename T>
const PS::MatrixSym3<T> & PS::MatrixSym3<T>::operator -=
                        (const PS::MatrixSym3<T> & rhs);
```

- **Argument**

  `rhs`: Input. Type `const PS::MatrixSym3<T> &`.

- **Return value**

  Type `const PS::MatrixSym3<T> &`. Subtract the components of `rhs` from its components, and return itself (lhs is changed).

6.5.3.5 Trace

```
template<typename T>
T PS::MatrixSym3<T>::getTrace() const;
```

- **Argument**

  None.

- **Return value**

  Type `T`.

- **Feature**

  Calculate the trace, and return the result.

6.5.3.6 Typecast to MatrixSym3<U>

```
template<typename T>
template<typename U>
PS::MatrixSym3<T>::operator PS::MatrixSym3<U> () const;
```

- **Argument**

  None.

- **Return value**

  Type `const PS::MatrixSym3<U>`.

- **Feature**

  Typecast from type `const PS::MatrixSym3<T>` to `const PS::MatrixSym3<U>`.

6.5.4 Wrappers

The wrappers of symmetric matrix types are defined as follows.

Listing 12: matrixsymwrapper

```
namespace ParticleSimulator{
    typedef MatrixSym2<F32> F32mat2;
    typedef MatrixSym3<F32> F32mat3;
    typedef MatrixSym2<F64> F64mat2;
    typedef MatrixSym3<F64> F64mat3;
#ifdef PARTICLE_SIMULATOR_TOW_DIMENSION
    typedef F32mat2 F32mat;
    typedef F64mat2 F64mat;
#else
```

```
    typedef F32mat3 F32mat;
    typedef F64mat3 F64mat;
#endif
}
namespace PS = ParticleSimulator;
```

PS::F32mat2, PS::F32mat3, PS::F64mat2, and PS::F64mat3 are, respectively, 2x2 symmetric matrix in single precision, 3x3 symmetric matrix in single presicion, 2x2 symmetric matrix in double precision, and 3x3 symmetric matrix in double precision. If users set 2D (3D) coordinate system, PS::F32mat and PS::F64mat is wrappers of PS::F32mat2 and PS::F64mat2 (PS::F32mat3 and PS::F64mat3).

6.6 PS::SEARCH_MODE type

6.6.1 Summary

In this section, we describe data type PS::SEARCH_MODE. It is used only as template arguments of class PS::TreeForForce. This data type determines the interaction mode of the class. PS::SEARCH_MODE can take the following five values; PS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOFF, PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY, PS::SEARCH_MODE_LONG_SCATTER and PS::SEARCH_MODE_LONG_CUTOFF_SCATTER, and each of them corresponds to a mode for interaction calculation. In the following, we describe them.

6.6.2 PS::SEARCH_MODE_LONG

This type is used when a group of distant particles is regarded as a superparticle as in the standard Barnes-Hut tree code. This type is for gravitational force and Coulomb's force under open boundary condition.

6.6.3 PS::SEARCH_MODE_LONG_CUTOFF

This type is used when a group of distant particles is regarded as a superparticle, and when its force does not reach to infinity. This type is for gravitational force and Coulomb's force under periodic boundary condition.

6.6.4 PS::SEARCH_MODE_GATHER

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

6.6.5 PS::SEARCH_MODE_SCATTER

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

6.6.6 PS::SEARCH_MODE_SYMMETRY

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 sizes of i- and j-particles.

6.6.7 PS::SEARCH_MODE_LONG_SCATTER

Almost the same as `SEARCH_MODE_LONG`, but if the distance between i- and j-particles is smaller than the search radius of j-particle, the j-particle is not included in superparticle.

6.6.8 PS::SEARCH_MODE_LONG_CUTOFF_SCATTER

Not implemented yet.

6.7 Enumerated type

6.7.1 Summary

In this section, we describe enumerated types defined in FDPS. Currently, there is just one datatype. We describe it below.

6.7.2 PS::BOUNDARY_CONDITION 型

6.7.2.1 Summary

Type BOUNDARY_CONDITION specifies boundary conditions. The definition is as follows.

Listing 13: boundarycondition

```
namespace ParticleSimulator{
    enum BOUNDARY_CONDITION{
        BOUNDARY_CONDITION_OPEN,
        BOUNDARY_CONDITION_PERIODIC_X,
        BOUNDARY_CONDITION_PERIODIC_Y,
        BOUNDARY_CONDITION_PERIODIC_Z,
        BOUNDARY_CONDITION_PERIODIC_XY,
        BOUNDARY_CONDITION_PERIODIC_XZ,
        BOUNDARY_CONDITION_PERIODIC_YZ,
        BOUNDARY_CONDITION_PERIODIC_XYZ,
        BOUNDARY_CONDITION_SHEARING_BOX,
        BOUNDARY_CONDITION_USER_DEFINED,
    };
}
```

We explain each value below.

6.7.2.2 PS::BOUNDARY_CONDITION_OPEN

This specifies the open boundary condition.

6.7.2.3 PS::BOUNDARY_CONDITION_PERIODIC_X

This specifies the periodic boundary condition in the direction of x-axis, and open boundary condition in other directions. The interval is left-bounded and right-unbounded. This is true for all periodic boundary conditions.

6.7.2.4 PS::BOUNDARY_CONDITION_PERIODIC_Y

This specifies the periodic boundary condition in the direction of y-axis, and open boundary condition in other directions.

6.7.2.5 PS::BOUNDARY_CONDITION_PERIODIC_Z

This specifies the periodic boundary condition in the direction of z-axis, and open boundary condition in other directions.

6.7.2.6 PS::BOUNDARY_CONDITION_PERIODIC_XY

This specifies the periodic boundary condition in the directions of x- and y-axes, and open boundary condition in the direction of z-axis.

6.7.2.7 PS::BOUNDARY_CONDITION_PERIODIC_XZ

This specifies the periodic boundary condition in the directions of x- and z-axes, and open boundary condition in the direction of y-axis.

6.7.2.8 PS::BOUNDARY_CONDITION_PERIODIC_YZ

This specifies the periodic boundary condition in the directions of y- and z-axes, and open boundary condition in the direction of x-axis.

6.7.2.9 PS::BOUNDARY_CONDITION_PERIODIC_XYZ

This specifies the periodic boundary condition in all three directions.

6.7.2.10 PS::BOUNDARY_CONDITION_SHEARING_BOX

Not implemented yet.

6.7.2.11 PS::BOUNDARY_CONDITION_USER_DEFINED

Not implemented yet.

6.8 PS::TimeProfile type

6.8.1 Abstract

In this section, we describe data type PS::TimeProfile. This data type is class to store calculation time for each function, and is used for three classes: DomainInfo, ParticleSystem, and TreeForForce. These three classes have "PS::TimeProfile getTimeProfile()". Users get the calculation time of each function by using the function "getTimeProfile()".

This class is described as follows.

Listing 14: TimeProfile

```
namespace ParticleSimulator{
    class TimeProfile{
    publid:
        F64 collect_sample_particle;
        F64 decompose_domain;
        F64 exchange_particle;
        F64 make_local_tree;
        F64 make_global_tree;
        F64 calc_force;
        F64 calc_moment_local_tree;
        F64 calc_moment_global_tree;
        F64 make_LET_1st;
        F64 make_LET_2nd;
        F64 exchange_LET_1st;
        F64 exchange_LET_2nd;
    };
}
```

6.8.1.1 Addition

```
PS::TimeProfile PS::TimeProfile::operator +
             (const PS::TimeProfile & rhs) const;
```

- **Argument**

  rhs: Input. Type const TimeProfile &.

- **Return value**

  Type PS::TimeProfile. Add the components of rhs to its own components, and return the results.

6.8.1.2 Reduction

```
PS::F64 PS::TimeProfile::getTotalTime() const;
```

- **Argument**

  None.

- **Return value**

  Type PS::F64. Return values of all the member variables.

6.8.1.3 Initialize

```
void PS::TimeProfile::clear();
```

- **Argument**

  None.

- **Return value**

  None.

- **Feature**

  Assign 0 to all the member variables.

7 Classes and Functors User-Defined

7.1 Summary

In this section we provide details of user-defined classes and functors. The user-defined classes are `FullParticle`, `EssentialParticleI`, `EssentialParticleJ`, `Moment`, `SuperParticleJ`, `Force` and `Header` classes. The functors are `calcForceEpEp` and `calcForceSpEp`.

A `FullParticle` class contains all informations of a particle and is a template argument of FDPS-defined classes (see step 0 in Sec. 2.3).

Functors `calcForceEpEp` and `calcForceSpEp` define the interaction between two `EssentialParticle` classes and `SuperParticles` class and `EssentialParticle`, respectively. These functors are argument of FDPS-defined `TreeForForce` (see step 0 in Sec. 2.3). If neither `PS::SEARCH_MODE_LONG` nor `PS::SEARCH_MODE_LONG_CUTOFF` are used, users do not need to define `calcForceSpEp`.

Classes `EssentialParticleI`, `EssentialParticleJ`, `Moment`, `SuperParticleJ` and `Force` are support classes for the interactions between two particles. Classes `EssentialParticleI` and `EssentialParticleJ` contain the quantities of $i-$ and $j-$ particles used for the calculation of interactions. A `Force` class contains the quantities of an $i-$ particle used to store the results of the calculations of interactions. Since these classes contain part of information of `FullParticle`, it is possible to use `FullParticle` in place of these classes. However, `FullParticle` may contain other values which are not used to evaluate the calculations of interactions. It is recommended to use these classes when high performance is desirable. Classes `Moment` and `SuperParticleJ` contain the information of moment and super particle, respectively. These classes are necessary if users require the super particle, *i.e.*, when `PS::SEARCH_MODE_LONG` or `PS::SEARCH_MODE_LONG_CUTOFF` are used. A class `Header` contains the header informations of input/output file.

The rest of this chapter describes how these classes and functors should be implemented. Users must write data transfer functions between these classes and interaction calculation in functor, which are used in the above member functions and functors.

7.2 FullParticle

7.2.1 Summary

The `FullParticle` class contains all information of a particle and is one of the template parameters of FDPS-defined `ParticleSystem` (see step 0 in Sec. 2.3). Users can define arbitrary member variables and member functions, as far as required member functions are defined. Below, we describe the required member functions.

7.2.2 Premise

Let us take `FP` class as an example of `FullParticle` as below. Users can use an arbitrary name in place of `FP`.

```
class FP;
```

7.2.3 Required member functions

7.2.3.1 Summary

The member functions `FP::getPos` and `FP::copyFromForce` are required. `FP::getPos` returns the position of a particle. Function `FP::copyFromForce` copies the results of calculation back to `FullParticle`. The examples and descriptions for these member functions are listed below.

7.2.3.2 FP::getPos

```
class FP {
public:
    PS::F64vec getPos() const;
};
```

- **Arguments** None.
- **Returns**

  `PS::F32vec` or `PS::F64vec`. Returns the position of `FP` class.
- **Behaviour**

  Returns the member variables which contains the position of a particle.

7.2.3.3 FP::copyFromForce

```
class Force;

class FP {
public:
    void copyFromForce(const Force & force);
};
```

- **Arguments**

  `force`: Input. `const Force` type.
- **Returns**

  None.
- **Behaviour**

  Copies back the results of calculation to `FP` class.

7.2.4 Required member functions for specific cases

7.2.4.1 Summary

In this section we describe the member functions for specific cases listed below;

- Modes other than `PS::SEARCH_MODE_LONG` for `PS::SEARCH_MODE` are used.
- APIs for file I/O in `ParticleSystem` are used.
- `ParticleSystem::adjustPositionIntoRootDomain` API is used.
- Particle Mesh class, which is an extension of FDPS, is used.

7.2.4.2 Modes other than `PS::SEARCH_MODE_LONG` for `PS::SEARCH_MODE` are used

7.2.4.2.1 FP::getRSearch

```
class FP {
public:
    PS::F64 getRSearch() const;
};
```

- **Arguments**

  None.

- **Returns**

  `PS::F32vec` or `PS::F64vec`. Returns the value of the member variable which contains neighbor search radius in `FP`.

- **Behaviour**

  Returns the value of the member variable which contains neighbor search radius in `FP`.

7.2.4.3 APIs for file I/O in `ParticleSystem` are used

The member functions `readAscii` and `writeAscii` are necessary, if users use `ParticleSystem::readPa` and `ParticleSystem::writeParticleAscii`. In this section we describe the rules for `readAscii` and `writeAscii`.

7.2.4.3.1 FP::readAscii

```
class FP {
public:
    void readAscii(FILE *fp);
};
```

- **Arguments**

  `fp`: `FILE *` type. A file pointer of input file.

- **Returns**

  None.

- **Behaviour**

  Inputs the data, `id`, `mass` and `pos` in `FP` class from an input file.

7.2.4.3.2 FP::writeAscii

```
class FP {
public:
    void writeAscii(FILE *fp);
};
```

- **Arguments**

  `fp`: `FILE *` type. A file pointer of output file.

- **Returns**

  None.

- **Behaviour**

  Inputs the data, `id`, `mass` and `pos` in `FP` class to an output file.

7.2.4.4 `ParticleSystem::adjustPositionIntoRootDomain` API is used

7.2.4.4.1 FP::setPos

```
class FP {
public:
    void setPos(const PS::F64vec pos_new);
};
```

- **Arguments**

  `pos_new`: Input. `const PS::F32vec` or `const PS::F64vec`. Modified positions of particle by FDPS.

- **Returns**

  None.

- **Behaviour**

  Replaces the positions in `FP` class by those modified by FDPS.

7.2.4.5 Particle Mesh class, which is an extension of FDPS, is used

When `Particle Mesh` class is used, `FP::getChargeParticleMesh` and `FP::copyFromForceParticleM`
must be defined. Below, the rules for these functions are described.

7.2.4.5.1 FP::getChargeParticleMesh

```
class FP {
public:
    PS::F64 getChargeParticleMesh() const;
};
```

- **Arguments**

  None.

- **Returns**

  `PS::F32` or `PS::F64`. Returns the mass or the electric charge of a particle.

7.2.4.5.2 FP::copyFromForceParticleMesh

```
class FP {
public:
    void copyFromForceParticleMesh(const PS::F32vec & acc_pm);
};
```

- **Arguments**

  acc_pm: `const PS::F32vec` or `const PS::F64vec`. Returns the resulting force by Particle Mesh.

- **Returns**

  None.

- **Behaviour**

  Writes back the resulting force by Particle Mesh to a particle.

7.3 EssentialParticleI

7.3.1 Summary

The `EssentialParticleI` class should contain all information of an $i-$ particle which is necessary to calculate interaction (see step 0 in Sec. 2.3). Class `EssentialParticleI` should have required member functions with specific names, as described below.

7.3.2 Premise

Let us take EPI and FP classes as examples of EssentialParticleI and FullParticle as below. Users can use arbitrary names in place of EPI and FP.

```
class FP;
class EPI;
```

7.3.3 Required member functions

7.3.3.1 Summary

The member functions EPI::getPos and EPI::copyFromForce are required. EPI::getPos returns the position of a particle to FDPS. EPI::copyFromFP copies the information necessary for the interaction calculation from FullParticle. The examples and descriptions for these member functions are listed below.

7.3.3.2 EPI::getPos

```
class EPI {
public:
    PS::F64vec getPos() const;
};
```

- **Arguments**

  None.

- **Returns**

  PS::F64vec. Returns the position of a particle of EPI class.

7.3.3.3 EPI::copyFromFP

```
class FP;
class EPI {
public:
    void copyFromFP(const FP & fp);
};
```

- **Arguments**

  fp: Input. const FP & type.

- **Returns**

  None.

- **Behaviour**

  Copies the part of information of `FP` to `EssnetialParticleI`.

7.3.4 Required member functions for specific case

7.3.4.1 Summary

In this section we describe the member functions in the case that `PS::SEARCH_MODE_GATHER` or `PS::SEARCH_MODE_SYMMETRY` are used.

7.3.4.2 Modes other than `PS::SEARCH_MODE_LONG` as `PS::SEARCH_MODE` are used

7.3.4.2.1 EPI::getRSearch

```
class EPI {
public:
    PS::F64 getRSearch() const;
};
```

- **Arguments**

  None.

- **Returns**

  `PS::F32vec` or `PS::F64vec`. Returns the value of the neighbour search radius in `EPI`.

7.4 EssentialParticleJ

7.4.1 Summary

The `EssentialParticleJ` class should contain all information of a $j-$ particle which is necessary to calculate interaction (see step 0 in Sec. 2.3). This class is a subset of `FullParticle` (see, Sec. 7.2). Class `EssentialParticleJ` should have required member functions with specific names, as described below.

7.4.2 Premise

Let us take `EPJ` and `FP` classes as examples of `EssentialParticleJ` and `FullParticle` as below. Users can use arbitrary names in place of `EPJ` and `FP`.

```
class FP;
class EPJ;
```

7.4.3 Required member functions

7.4.3.1 Summary

The member functions `EPJ::getPos` and `EPJ::copyFromForce` are required. `EPJ::getPos` returns the position of a particle to FDPS. `EPJ::copyFromFP` copies the information necessary for the interaction calculation from `FullParticle`. The examples and descriptions for these member functions are listed below.

7.4.3.2 EPJ::getPos

```
class EPJ {
public:
    PS::F64vec getPos() const;
};
```

- **Arguments**

  None.

- **Returns**

  `PS::F64vec`. Returns the position of a particle of `EPJ` class.

7.4.3.3 EPJ::copyFromFP

```
class FP;
class EPJ {
public:
    void copyFromFP(const FP & fp);
};
```

- **Arguments**

  `fp`: Input. `const FP &` type.

- **Returns**

  None.

- **Behaviour**

  Copies the part of information of `FP` to `EPJ`.

7.4.4 Required member functions for specific cases

7.4.4.1 Summary

In this section we describe the member functions in the case that the other mode than `PS::SEARCH_MODE_LONG` as `PS::SEARCH_MODE` or `PS::BOUNDARY_CONDITION_OPEN` as `BOUNDARY_CONDITION` are used. For the detailed description about pre-defined `Moment` and `SuperParticleJ` classes, see corresponding section.

7.4.4.2 Modes other than `PS::SEARCH_MODE_LONG` as `PS::SEARCH_MODE` are used

7.4.4.2.1 EPJ::getRSearch

```
class EPJ {
public:
    PS::F64 getRSearch() const;
};
```

- **Arguments**

  None.

- **Returns**

  `PS::F32vec` or `PS::F64vec`. Returns the value of the member variable which contains neighbor search radius in `EPJ`.

7.4.4.3 Modes other than `PS::BOUNDARY_CONDITION_OPEN` are used `BOUNDARY_CONDITION`

7.4.4.3.1 EPJ::setPos

```
class EPJ {
public:
    void setPos(const PS::F64vec pos_new);
};
```

- **Arguments**

  `pos_new`: Input. `const PS::F32vec` or `const PS::F64vec`. Modified positions of particle by FDPS.

- **Returns**

  None.

- **Behaviour**

  Replaces the positions in `EPJ` class by those modified by FDPS.

7.5 Moment

7.5.1 Summary

The class `Moment` encapsulates the values of moment and related quantities of a set of grouped particles (see, step 0 in Sec. 2.3). The examples of moment are monopole, dipole and radius of the largest particle and so on. This class is used for intermediate variables to make `SuperParticleJ` from `EssentialParticleJ`. Thus, this class has a member function that calculates the moment from `EssentialParticleJ` or `SuperParticleJ`.

Since these processes have prescribed form, FDPS has pre-defined classes. In this section we first describe the pre-defined class and then describe the rules in the case users define it.

7.5.2 Pre-defined class

7.5.2.1 Summary

FDPS has several pre-defined `Moment` classes, which are used if specific case are chosen as `PS::SEARCH_MODE` in `PS::TreeForForce`. Below, we describe about the `Moment` class which can be chosen in each `PS::SEARCH_MODE`.

7.5.2.2 PS::SEARCH_MODE_LONG

7.5.2.2.1 PS::MomentMonopole

This class encapsulates the monopole moment. The reference point for the calculation of monopole is set to center of mass or center of charge.

```
namespace ParticleSimulator {
    class MomentMonopole {
    public:
        F32    mass;
        F32vec pos;
    };
}
```

- Name of the class PS::MomentMonopole
- Members and their information mass: accumulated mass or electron charge.

  pos: center of mass or center of charge.
- Terms of use

  The class `EssentialParticleJ` (see, Sec. 7.4) has `EssentialParticleJ::getCharge` and `EssentialParticleJ::getPos` and returns mass/electron charge and position. Users can use an arbitrary name instead of `EssentialParticleJ`.

7.5.2.2.2 PS::MomentQuadrupole

This class encapsulates the monopole and quadrupole moment. The reference point for the calculation of moments is set to the center of mass.

```
namespace ParticleSimulator {
    class MomentQuadrupole {
    public:
        F32    mass;
        F32vec pos;
        F32mat quad;
    };
}
```

- Name of the class `PS::MomentQuadrupole`
- Members and their information

  mass: accumulated mass.

  pos: center of mass.

  quad: accumulated quadrupole.
- Terms of use

  The class `EssentialParticleJ` (see, Sec. 7.4) has `EssentialParticleJ::getCharge` and `EssentialParticleJ::getPos` and returns mass/electron charge and position. Users can use an arbitrary name instead of `EssentialParticleJ`.

7.5.2.2.3 PS::MomentMonopoleGeometricCenter

This class encapsulates the monopole moment. The reference point for the calculation of monopole is set to the geometric center.

```
namespace ParticleSimulator {
    class MomentMonopoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
    };
}
```

- Name of the class `PS::MomentMonopoleGeometricCenter`
- Members and their information

  `charge`: accumulated mass/charge.

  `pos`: geometric center.
- Terms of use

  The class `EssentialParticleJ` (see, Sec. 7.4) has `EssentialParticleJ::getCharge` and `EssentialParticleJ::getPos` and returns mass/electron charge and position. Users can use an arbitrary name instead of `EssentialParticleJ`.

7.5.2.2.4 PS::MomentDipoleGeometricCenter

This class encapsulates the moments up to dipole. The reference point for the calculation of moments is set to geometric center.

```
namespace ParticleSimulator {
    class MomentDipoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
        F32vec dipole;
    };
}
```

- Name of the class `PS::MomentDipoleGeometricCenter`
- Members and their information

  `charge`: accumulated mass/charge.

  `pos`: geometric center.

  `dipole`: dipole of the particle mass or electron charge.
- Terms of use

  The class `EssentialParticleJ` (see, Sec. 7.4) has `EssentialParticleJ::getCharge` and `EssentialParticleJ::getPos` and returns mass/electron charge and position. Users can use an arbitrary name instead of `EssentialParticleJ`.

7.5.2.2.5 PS::MomentQuadrupoleGeometricCenter

This class encapsulates the moments up to quadrupole. The reference point for the calculation of moments is set to geometric center.

```
namespace ParticleSimulator {
    class MomentQuadrupoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
        F32vec dipole;
        F32mat quadrupole;
    };
}
```

- Name of the class `PS::MomentQuadrupoleGeometricCenter`
- Members and their information

  `charge`: accumulated mass/charge.

`pos`: geometric center.

`dipole`: dipole of the particle mass or electron charge.

`quadrupole`: quadrupole of the particle mass or electron charge.

- Terms of use

  The class `EssentialParticleJ` (see, Sec. 7.4) has `EssentialParticleJ::getCharge` and `EssentialParticleJ::getPos` and returns mass/electron charge and position. Users can use an arbitrary name instead of `EssentialParticleJ`.

7.5.2.3 PS::SEARCH_MODE_LONG_CUTOFF

7.5.2.3.1 PS::MomentMonopoleCutoff

This class encapsulates the monopole moment. The reference point for the calculation of monopole is set to center of mass or center of charge.

```
namespace ParticleSimulator {
    class MomentMonopoleCutoff {
    public:
        F32    mass;
        F32vec pos;
    };
}
```

- Name of the class `PS::MomentMonopoleCutoff`

- Members and their information

  `mass`: accumulated mass or electron charge.

  `pos`: center of mass or center of charge.

- Terms of use

  The class `EssentialParticleJ` (see, Sec. 7.4) has `EssentialParticleJ::getCharge`, `EssentialParticleJ::getPos` and `EssentialParticleJ::getRSearch` and returns mass/electron charge, position and cutoff length. Users can use an arbitrary name instead of the name of member functions of `EssentialParticleJ`.

7.5.3 Required member functions

7.5.3.1 Summary

Below, the required member functions of `Moment` class are described. In this section we use the name `Mom` as a `Moment` class.

7.5.3.2 Constructor

```
class Mom {
public:
    Mom ();
};
```

- **Arguments**

  None.

- **Returns**

  None.

- **Behaviour**

  Initialize the instance of `Mom` class.

7.5.3.3 Mom::init

```
class Mom {
public:
    void init();
};
```

- **Arguments**

  None.

- **Returns**

  None.

- **Behaviour**

  Initialize the instance of `Mom` class.

7.5.3.4 Mom::getPos

```
class Mom {
public:
    PS::F32vec getPos();
};
```

- **Arguments**

  None.

- **Returns**

  `PS::F32vec` or `PS::F64vec`. Returns the position of a variable of class.

7.5.3.5 Mom::getCharge

```
class Mom {
public:
    PS::F32 getCharge() const;
};
```

- **Arguments**

  None.

- **Returns**

  `PS::F32` or `PS::F64`. Returns the mass or charge of a variable of class `Mom`.

7.5.3.6 Mom::accumulateAtLeaf

```
class Mom {
public:
    template <class Tepj>
    void accumulateAtLeaf(const Tepj & epj);
};
```

- **Arguments**

  `epj`: Input. `const Tepj &` type. The object of `Tepj`.

- **Returns**

  None.

- **Behaviour**

  Accumulate the multipole moment from `EssentialParticleJ`.

7.5.3.7 Mom::accumulate

```
class Mom {
public:
    void accumulate(const Mom & mom);
};
```

- **Arguments**

  `mom`: Input. `const Mom &` type. The instance of `Mom` class.

- **Returns**

  None.

- **Behaviour**

  Accumulate the multipole moments in `Mom` class from `Mom` class objects.

7.5.3.8 Mom::set

```
class Mom {
public:
    void set();
};
```

- **Arguments**

  None.

- **Returns**

  None.

- **Behaviour**

  Normalize the multipole moments, since the member functions `Mom::accumulateAtLeaf` and `Mom::accumulate` do not normalize the multipole moment.

7.5.3.9 Mom::accumulateAtLeaf2

```
class Mom {
public:
    template <class Tepj>
    void accumulateAtLeaf2(const Tepj & epj);
};
```

- **Arguments**

  `epj`: Input. `const Tepj &` type. The instance of `Teps`.

- **Returns**

  None.

- **Behaviour**

  Accumulate the multipole moments in `Mom` class from `EssentialParticleJ` class objects.

7.5.3.10 Mom::accumulate2

```
class Mom {
public:
    void accumulate(const Mom & mom);
};
```

- **Arguments**

  `mom`: Input. `const Mom &` type. The instance of `Mom` class.

- **Returns**

  None.

- **Behaviour**

  Accumulate the multipole moments in `Mom` class from `Mom` class objects.

7.6 SuperParticleJ

7.6.1 概要

The class `SuperParticleJ` encapsulates the informations of a grouped particles (see, step 0 in Sec. 2.3). This class is required if `PS::SEARCH_MODE_LONG` or `PS::SEARCH_MODE_LONG_CUTOFF` are employed as `PS::SEARCH_MODE`. This class has a member function which is used for data-transfer between FDPS. This class is related to `Moment` class. Thus, this class has member functions which cast the `Moment` class to this class (and vice versa).

Similar to `Moment` class, since this class has prescribed rules, FDPS has pre-defined classes. This section aims to describe the specific member functions both always and in specific case required for `SuperParticleJ`.

7.6.2 Pre-defined class

FDPS `has` some pre-defined `SuperParticleJ` class. Below, the available classes for each `PS::SEARCH_MODE` are described. First, we described classes which are available for `PS::SEARCH_MODE_LONG` and then for `PS::SEARCH_MODE_LONG_CUTOFF`. In other case `PS::SEARCH_MODE` does not require super particle.

7.6.2.1 PS::SEARCH_MODE_LONG

7.6.2.1.1 PS::SPJMonopole

A `SuperParticleJ` class which is related to `PS::MomentMonopole` class. It contains up to monopole moment.

```
namespace ParticleSimulator {
    class SPJMonopole {
    public:
        F64    mass;
        F64vec pos;
    };
}
```

- Name of the class `PS::SPJMonopole`
- Members and their information

  `mass`: accumulated mass or electron charge.

  `pos`: center of mass or center of charge.
- Terms of use

  The same as `PS::MomentMonopole` class.

7.6.2.1.2 PS::SPJQuadrupole

A `SuperParticleJ` class which is related to `PS::MomentQuadrupole` class. It contains up to quadrupole moment.

```
namespace ParticleSimulator {
    class SPJQuadrupole {
    public:
        F32    mass;
        F32vec pos;
        F32mat quad;
    };
}
```

- Name of the class `PS::SPJQuadrupole`
- Members and their information

  `mass`: accumulated mass.

  `pos`: center of mass.

  `quad`: accumulated quadrupole.
- Terms of use

  The same as `PS::MomentQuadrupole` class.

7.6.2.1.3 PS::SPJMonopoleGeometricCenter

A `SuperParticleJ` class which is related to `PS::MomentMonopoleGeometricCenter` class. It contains up to quadrupole moment (the reference point is set to geometric center of grouped particles).

```
namespace ParticleSimulator {
    class SPJMonopoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
    };
}
```

- Name of the class `PS::SPJMonopoleGeometricCenter`
- Members and their information

  `charge`: accumulated mass/charge.

  `pos`: geometric center.
- Terms of use

  The same as `PS::MomentMonopoleGeometricCenter` class.

7.6.2.1.4 PS::SPJDipoleGeometricCenter

A `SuperParticleJ` class which is related to `PS::MomentDipoleGeometricCenter` class. It contains up to dipole moment (the reference point is set to geometric center of grouped particles).

```
namespace ParticleSimulator {
    class SPJDipoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
        F32vec dipole;
    };
}
```

- Name of the class `PS::SPJDipoleGeometricCenter`
- Members and their information

  `charge`: accumulated mass/charge.

  `pos`: geometric center.

  `dipole`: dipole of the particle mass or electron charge.

- Terms of use

  The same as `PS::MomentDipoleGeometricCenter` class.

7.6.2.1.5 *PS::SPJQuadrupoleGeometricCenter*

A `SuperParticleJ` class which is related to `PS::MomentQuadrupoleGeometricCenter` class. It contains up to quadrupole moment (the reference point is set to geometric center of grouped particles).

```
namespace ParticleSimulator {
    class SPJQuadrupoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
        F32vec dipole;
        F32mat quadrupole;
    };
}
```

- Name of the class `PS::SPJQuadrupoleGeometricCenter`
- Members and their information

  `charge`: accumulated mass/charge.

  `pos`: geometric center.

  `dipole`: dipole of the particle mass or electron charge.

  `quadrupole`: quadrupole of the particle mass or electron charge.
- Terms of use

  The same as `PS::MomentQuadrupoleGeometricCenter` class.

7.6.2.2 PS::SEARCH_MODE_LONG_CUTOFF

7.6.2.2.1 *PS::SPJMonopoleCutoff*

A `SuperParticleJ` class which is related to `PS::MomentMonopoleCutoff` class. It contains up to monopole moment.

```
namespace ParticleSimulator {
    class SPJMonopoleCutoff {
    public:
        F32    mass;
        F32vec pos;
    };
}
```

- Name of the class `PS::SPJMonopoleCutoff`
- Members and their information

  `mass`: accumulated mass or electron charge.

  `pos`: center of mass or center of charge.
- Terms of use

  The same as `PS::MomentMonopoleCutoff` class.

7.6.3 Required member functions

7.6.3.1 Summary

Below, the required member functions of `SuperParticleJ` class are described. In this section we use the name `SPJ` as a `SuperParticleJ` class.

7.6.3.2 SPJ::getPos

```
class SPJ {
public:
    PS::F64vec getPos() const;
};
```

- **Arguments**

  None.
- **Returns**

  `PS::F32vec` or `PS::F64vec`. Returns the position pf a super-particle of class `SPJ`.

7.6.3.3 SPJ::setPos

```
class SPJ {
public:
    void setPos(const PS::F64vec pos_new);
};
```

- **Arguments**

  `pos_new`: Input. `const PS::F32vec` or `const PS::F64vec`. Modified positions of particle by FDPS.
- **Returns**

  None.
- **Behaviour**

  Replaces the positions in `SPJ` class by those modified by FDPS.

7.6.3.4 SPJ::copyFromMoment

```
class Mom;
class SPJ {
public:
    void copyFromMoment(const Mom & mom);
};
```

- **Arguments**

  mom: Input. const Mom & type. Mom can be both user defined and pre-defined Moment class.

- **Returns**

  None.

- **Behaviour**

  Copies the informations of Mom class to SPJ.

7.6.3.5 SPJ::convertToMoment

```
class Mom {
public:
    Mom(const PS::F32 m,
        const PS::F32vec & p);
}
class SPJ {
public:
    Mom convertToMoment() const;
};
```

- **Arguments**

  None.

- **Returns**

  Mom type. The constructor of Mom class.

- **Behaviour**

  Returns the constructor of Mom class.

7.6.3.6 SPJ::clear

```
class SPJ {
public:
    void clear();
};
```

- **Arguments**

  None.

- **Returns**

  None.

- **Behaviour**

  Clears the information of `SPJ` class.

7.7 Force

7.7.1 Summary

The `Force` class contains the results of the calculation of interactions (see, step 0 in Sec. 2.3). In this section we describe the member functions in any case required.

7.7.2 Premise

Let us take `Result` class as an example of `Force` as below. Users can use an arbitrary name instead of `Result`.

7.7.3 Required member functions

The member function `Result::clear` is in any case needed. This function initialize the result on interaction calculations.

7.7.3.1 Result::clear

```
class Result {
public:
    void clear();
};
```

- **Arguments**

  None.

- **Returns**

  None.

- **Behaviour**

  Initializes the result of interaction calculations.

7.8 Header

7.8.1 Summary

`Header` class defines the format of header of input/output files. This class is required if users use file input/output API provided by FDPS (`ParticleSystem::readParticleAscii` and `ParticleSystem::writeParticleAscii`) and involves header information to input/output file. Below, we describe the descriptions and rules for the member functions required from provided input/output API. There is no functions which is always required.

7.8.2 Premise

Let us take `Hdr` class as an example of `Header` as below. Users can use an arbitrary name in place of `Hdr`.

7.8.3 Required member functions for specific case

7.8.3.1 Hdr::readAscii

```
class Hdr {
public:
    PS::S32 readAscii(FILE *fp);
};
```

- **Arguments**

  `fp`: Input. `FILE *` type. The file pointer of the input file.

- **Returns**

  `PS::S32` type. Returns the value of number of particles. Returns -1 if header does not contain the number of particle.

- **Behaviour**

  Read the header data from the input file.

7.8.3.2 Hdr::writeAscii

```
class Hdr {
public:
    void writeAscii(FILE *fp);
};
```

- **Arguments**

  fp: Input. FILE * type. The file pointer of the output file.

- **Returns**

  None.

- **Behaviour**

  Copies the header information to the output file.

7.9 Functor calcForceEpEp

7.9.1 Summary

Functor calcForceEpEp defines the interaction between two particles. This functor is required for the calculation of interactions (see, step 0 in Sec. 2.3).

7.9.2 Premise

Here one example of gravitational N-body problems are shown. The name of functor calcForceEpEp is gravityEpEp, which is an arbitrary. The class name of EssentialParitlceI, EssentialParitlceJ and Force are EPI, EPJ and Result.

7.9.3 gravityEpEp::operator ()

Listing 15: calcForceEpEp

```
class Result;
class EPI;
class EPJ;
struct gravityEpEp {
    static PS::F32 eps2;
    void operator () (const EPI *epi,
                      const PS::S32 ni,
                      const EPJ *epj,
                      const PS::S32 nj,
                      Result *result)
    }
};
PS::F32 gravityEpEp::eps2 = 9.765625e-4;
```

- **Arguments**

  epi: Input. const EPI * type or EPI * type. Array of i particles.

  ni: Input. const PS::S32 type or PS::S32. The number of i particles.

  epj: Input. const EPJ * type or EPJ * type. Array of j particles.

  nj: Input. const PS::S32 type or PS::S32 type. Number of j particles.

  result: Output. Result * type. Array of the results of interaction.

- **Returns**

  None.

- **Behaviour**

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

7.10 Functor calcForceSpEp

7.10.1 Summary

Functor `calcForceSpEp` defines the interaction to a particle from super particle. This functor is required for the calculation of interactions (see, step 0 in Sec. 2.3).

7.10.2 Premise

Here one example of gravitational N-body problems are shown. The superparticles are regarded to be constructed from up to monopole. The name of functor `calcForceSpEp` is `gravitySpEp`, which is an arbitrary. The class name of `EssentialParitlceI`, `SuperParitlceJ` and `Force` are `EPI`, `SPJ` and `Result`.

7.10.3 gravitySpEp::operator ()

Listing 16: calcForceSpEp

```
class Result;
class EPI;
class SPJ;
struct gravitySpEp {
    static PS::F32 eps2;
    void operator () (const EPI *epi,
                      const PS::S32 ni,
                      const SPJ *spj,
                      const PS::S32 nj,
                      Result *result);
};
```

- **Arguments**

  `epi`: Input. `const EPI *` type or `EPI *` type. Array of i particles.

  `ni`: Input. `const PS::S32` type or `PS::S32`. The number of i particles.

  `spj`: Input. `const SPJ *` type or `SPJ *` type. Array of super particle.

  `nj`: Input. `const PS::S32` type or `PS::S32` type. Number of super particles.

  `result`: Output. `Result *` type. Array of the results of interaction.

- **Returns**

  None.

- **Behaviour**

  Calculate interactions from super particle to i-th particle.

7.11 Functor calcForceDispatch

7.11.1 Summary

Functor `calcForceDispatch`, paired with functor `calcForceRetrieve`, defines the interaction between two particles. This functor can be used for the calculation of interactions (see, step 0 in Sec. 2.3), instead of `calcForceSpEp` and `calcForceEpEp`. The difference from `calcForceSpEp` and `calcForceEpEp` is that `calcForceDispatch` receives multiple interactions lists and i-particle lists. By doing so, it reduces the numbere of kernel calls to acclerators such as GPGPU, and improves the efficiency. The name of the functor `calcForceDispatch` is `GravityDispatch`. The name for classes `EssentialParitlceI`, `EssentialParitlceJ` and `SuperParitlceJ` are `EPI`, `EPJ`, and `SPI`, respectively. These names can be changed to any name legal.

7.11.2 The case of short-range interactions

Listing 17: calcForceDispatch

```
class EPI;
class EPJ;
PS::S32 HydroforceDispatch(const PS::S32  tag,
                           const PS::S32  nwalk,
                           const EPI**      epi,
                           const PS::S32*  ni,
                           const EPJ**      epj,
                           const PS::S32*  nj_ep;
};
```

- **Arguments**

  `tag`: input. Type `const PS::S3`. The value should be non-negative and less than the value specified by the third argument of function `PS::TreeForForce::calcForceAllandWriteBackMultiWalk()`. Corresponding call to `CalcForceRetrieve()` should use the same value for `tag`.

  `nwalk`: input.Type `const PS::S32`. The number of interaction lists. The value should not exceed the value of the sixth argument of `PS::TreeForForce::calcForceAllandWriteBackMultiWalk()`.

  `epi`: input. Type `const EPI**`. Array of the array of i-particles.

  `ni`: input. Type `const PS::S32*`. Array of the numbers of i-particles.

  `epj`: input. Type `const EPJ**`. Array of the array of j-particles.

  `nj`: input. Type `const PS::S32*`. Array of the number of j-paricles.

- **Return value**

  Type `PS::S32`. Should return 0 upon normal completion, and otherwize non-zero value.

- **Function**

  Send `epi` and `epj` to the accelerator and let the accelerator do the interaction calculation.

7.11.3 Case of long-range interaction

Listing 18: calcForceDispatch

```
class EPI;
class EPJ;
class SPJ;
PS::S32 GravityDispatch(const PS::S32   tag,
                        const PS::S32   nwalk,
                        const EPI**     epi,
                        const PS::S32*  ni,
                        const EPJ**     epj,
                        const PS::S32*  nj_ep,
                        const SPJ**     spj,
                        const PS::S32*  nj_sp);
};
```

- **Arguments**

  `tag`: input. Type `const PS::S3`. The value should be non-negative and less than the value specified by the third argument of function `PS::TreeForForce::calcForceAllandWriteBackMultiWalk()`. Corresponding call to `CalcForceRetrieve()` should use the same value for `tag`.

  `nwalk`: input.Type `const PS::S32`. The number of interaction lists. The value should not exceed the value of the sixth argument of `PS::TreeForForce::calcForceAllandWriteBackMultiWalk()`.

  `epi`: input. Type `const EPI**`. Array of the array of i-particles.

  `ni`: input. Type `const PS::S32*`. Array of the numbers of i-particles.

  `epj`: input. Type `const EPJ**`. Array of the array of j-particles.

  `nj`: input. Type `const PS::S32*`. Array of the number of j-paricles.

  `spj`: input. Type `const SPJ**`. Array of the array of superparticles.

  `nj_sp`: input. Type `const PS::S32*`. Array of the number of superparicles.

- **Return value**

  Type `PS::S32`. Should return 0 upon normal completion, and otherwize non-zero value.

- **Function**

  Send `epi` and `epj` to the accelerator and let the accelerator do the interaction calculation.

7.12 Functor calcForceRetrieve

7.12.1 Summary

Functor `calcForceRetrieve` retrieves the result of interaction calculation dispatched by functor `calcForceDispatch`.

The name of functor `calcForceRetrieve` is `GravityRetrieve` and the name of `Force` class is `Result`

Listing 19: calcForceDispatch

```
class EPI;
class EPJ;
class Result;
PS::S32 GravityRetrieve(const PS::S32  tag,
                        const PS::S32  nwalk,
                        const PS::S32  ni     [],
                        Result         result [][]);
};
```

- **Arguments**

  `tag`: input. Type `const PS::S32`. Should be the same as that for the corresponding call to `CalcForceDispatch()`.

  `nwalk`: input. Type `const PS::S32`. The number of interaction lists. Should be the same as that for the corresponding call to `CalcForceDispatch()`.

  `ni:` input. Type `const PS::S32*` or `PS::S32*`. Array of the numbers of i-particles.

  `result:` output. Type `Result**`.

- **return value** Returns 0 upon normal completion. Otherwize non-zero values are returned.

- **Function**

  Store the results calculated by the call to `CalcForceDispatch()` with the same tag value to the array `force`.

8 Initialization and Finalization of FDPS

8.1 General

This section describes the APIs of initialization and finalization of FDPS.

8.2 API

8.2.1 PS::Initialize

This API must be called to initialize FDPS.

```
void PS::Initialize
            (PS::S32 & argc,
             char ** & argv);
```

- **arguments**

  argc: input. Type `PS::S32 &`. The total number of arguments on the command line.

  argv: input. Type `char ** &`. The pointer to pointer to strings on the command line.

- **returned value**

  void.

- **function**

  Initialize FDPS. This API must be called before other APIs of FDPS are called. Note, since MPI::Init is called, that inside this API arguments "argc" and "argv" may change.

8.2.2 PS::Finalize

To finalize FDPS, this API must be called.

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

- **arguments**

  void.

- **return value**

  void.

- **function**

  Finalize FDPS.

8.2.3 PS::Abort

```
void PS::Abort(const PS::S32 err = -1);
```

- **arguments**

  err: Input. Type const `PS::S32`. The termination status of a program. Default value is -1.

- **returned value**

  void.

- **function**

  Terminate the user program abnormally. The argument is the termination status of the program. Under MPI environment, MPI::Abort() is called, otherwise exit() is called.

8.2.4 PS::DisplayInfo

```
void PS::DisplayInfo();
```

- **arguments**

  void.

- **returned value**

  void.

- **function**

  Show the license and other information of FDPS.

9 Classes defined in FDPS and their APIs

This section describes the classes defined in FDPS and their APIs. The following subsections describe the standard classes and the extention classes of FDPS.

9.1 Standard Classes

9.1.1 Summary

The following subsections describe the four standard classes of FDPS: `DomainInfo` class, `ParticleSystem` class, `TreeForForce` class and `Comm` class.

9.1.2 DomainInfo Class

This section describes `DomainInfo` class. This class handles the decomposition of computational domains and keeps their data.

9.1.2.1 Creation of Object

`DomainInfo` class is declared as below.

Listing 20: DomainInfo0

```
namespace ParticleSimulator {
    class DomainInfo;
}
```

Next example shows how to create an object of `DomainInfo` class. Here, the object is named “dinfo”.

```
PS::DomainInfo dinfo;
```

9.1.2.2 API

`DomainInfo` class has the APIs for initialization and decomposition of domain. The following subsections describe them.

9.1.2.2.1 Initial Setup

The APIs for the initial setup of `DomainInfo` class are declared below.

Listing 21: DomainInfo1

```
namespace ParticleSimulator {
    class DomainInfo{
    public:
        DomainInfo();
        void initialize(const F32 coef_ema=1.0);
        void setNumberOfDomainMultiDimension(const S32 nx,
```

```
                                                    const S32 ny,
                                                    const S32 nz=1);
        void setBoundaryCondition(enum BOUNDARY_CONDITION bc);
        void setPosRootDomain(const F32vec & low,
                              const F32vec & high);
    };
}
```

9.1.2.2.1.1 Constructor

Constructor

```
void PS::DomainInfo::DomainInfo();
```

- **arguments**

  void.

- **returned value**

  void.

- **function**

  Create an object of `DomainInfo` class.

9.1.2.2.1.2 PS::DomainInfo::initialize

PS::DomainInfo::initialize

```
void PS::DomainInfo::initialize(const PS::F32 coef_ema=1.0);
```

- **arguments**

  `coef_ema`: input. Type const `PS::F32`. The smoothing factor of an exponential moving average.

- **returned value**

  void.

- **function**

  Initialize an object of the domain information class. 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)

9.1.2.2.1.3 PS::DomainInfo::setNumberOfDomainMultiDimension

PS::DomainInfo::setNumberOfDomainMultiDimension

```
void PS::DomainInfo::setNumberOfDomainMultiDimension
                 (const PS::S32 nx,
                  const PS::S32 ny,
                  const PS::S32 nz=1);
```

- **arguments**

  `nx`: Input. Type const `PS::S32`. The number of subdomains along x direction.

  `ny`: Input. Type const `PS::S32`. The number of subdomains along y direction.

  `nz`: Input. Type const `PS::S32`. The number of subdomains along z direction. The default value is 1.

- **returned value**

  void.

- **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.

9.1.2.2.1.4 PS::DomainInfo::setBoundaryCondition

PS::DomainInfo::setBoundaryCondition

```
void PS::DomainInfo::setBoundaryCondition
                 (enum PS::BOUNDARY_CONDITION bc);
```

- **arguments**

  `bc`: input. Type enum `PS::BOUNDARY_CONDITION`. Boundary conditions.

- **returned value**

  void.

- **function**

  Set the boundary condition. FDPS allows boundary conditions defined in Section 6.7.2 (BOUNDARY_CONDITION_SHEARING_BOX and BOUNDARY_CONDITION_USER_DEFINED have not been implemented yet). If the API is not called, the open boundary is used.

9.1.2.2.1.5 PS::DomainInfo::setPosRootDomain

PS::DomainInfo::setPosRootDomain

```
void PS::DomainInfo::setPosRootDomain
                 (const PS::F32vec & low,
                  const PS::F32vec & high);
```

- **arguments**

  `low`: input. Type const `PS::F32vec`. Top vertex of the boundary (inclusive).

  `high`: input. Type const `PS::F32vec`. Bottom vertex of the boundary (exclusive).

- **returned value**

  void.

- **function**

  Set positions of vertexes of top and bottom of root domain. The API do 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.

9.1.2.2.2 Decomposition of Domain

The APIs of decomposition of domain are declared below:

Listing 22: DomainInfo2

```
namespace ParticleSimulator {
    class DomainInfo{
    public:
        template<class Tpsys>
        void collectSampleParticle(Tpsys & psys,
                                   const bool clear,
                                   const F32 weight);
        template<class Tpsys>
        void collectSampleParticle(Tpsys & psys,
                                   const bool clear);
        template<class Tpsys>
        void collectSampleParticle(Tpsys & psys);

        void decomposeDomain();

        template<class Tpsys>
        void decomposeDomainAll(Tpsys & psys,
                                const F32 weight);
        template<class Tpsys>
        void decomposeDomainAll(Tpsys & psys);
```

```
    };
}
```

9.1.2.2.2.1 PS::DomainInfo::collectSampleParticle
PS::DomainInfo::collectSampleParticle

```
template<class Tpsys>
void PS::DomainInfo::collectSampleParticle
                   (Tpsys & psys,
                    const bool clear,
                    const PS::F32 weight);
```

- **arguments**

  `psys`: input. Type `ParticleSystem &`. `ParticleSystem` class for giving sample particles to decompose domain.

  `clear`: input. Type `const bool`. A flag to clear the data of sample particles, if set to `true`.

  `weight`: input. Type const `PS::F32`. A weight to determine the number of sample particles.

- **returned value**

  void.

- **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.

```
template<class Tpsys>
void PS::DomainInfo::collectSampleParticle
                   (Tpsys & psys,
                    const bool clear);
```

- **arguments**

  `psys`: input. Type `ParticleSystem &`. An object of `ParticleSystem` class for giving sample particles to decompose domain.

  `clear`: input. Type `const bool`. A flag to clear the data of particles, if set to 'true'.

- **returned value**

  void.

- **function**

  Sample particles from `psys`. If `clear` is true, the data are cleared.

```
template<class Tpsys>
void PS::DomainInfo::collectSampleParticle
                    (Tpsys & psys);
```

- **arguments**

  `psys`: input. Type `ParticleSystem &`. An object of `ParticleSystem` class for giving sample particles to decompose domain.

- **returned value**

  void.

- **functions**

  Sample particles from `psys`.

9.1.2.2.2.2 PS::DomainInfo::decomposeDomain

PS::DomainInfo::decomposeDomain

```
void PS::DomainInfo::decomposeDomain();
```

- **arguments**

  void.

- **returned value**

  void.

- **function**

  Decompose calculation domains.

9.1.2.2.2.3 PS::DomainInfo::decomposeDomainAll

PS::DomainInfo::decomposeDomainAll

```
template<class Tpsys>
void PS::DomainInfo::decomposeDomainAll
                    (Tpsys & psys,
                     const PS::F32 weight);
```

- **arguments**

  `psys`: input. Type `ParticleSystem &`. The particle system class giving sample particles for the domain decomposition.

  `weight`: input. Type `const PS::F32`. The sampling weight.

- **returned value**

  void.

- **function**

  Sample particles from `psys` and decompose domains. This API is the combination of `PS::DomainInfo::collectSampleParticle` and `PS::DomainInfo::decomposeDomain`.

```
template<class Tpsys>
void PS::DomainInfo::decomposeDomainAll
                (Tpsys & psys);
```

- **arguments**

  psys: input. Type `ParticleSystem` &. The particle system class for giving samples.

- **returned value**

  void.

- **function**

  Sample particles from "psys" and decompose domains. This API is the combination of `PS::DomainInfo::collectSampleParticle` and `PS::DomainInfo::decomposeDomain`.

9.1.2.2.3 Time Measurment

The member functions of collecting information of an object are declared below. When a member function is called, its execution time is set to the private member of `TimeProfile` class `time_profile_`. Until the method `clearTimeProfile()` is called, execution times are accumulated.

Listing 23: DomainInfo3

```
namespace ParticleSimulator {
    class DomainInfo{
    public:
        TimeProfile getTimeProfile();
        void clearTimeProfile();
    };
}
```

9.1.2.2.3.1 PS::DomainInfo::getTimeProfile

PS::DomainInfo::getTimeProfile

```
PS::TimeProfile PS::DomainInfo::getTimeProfile();
```

- **arguments**

  void.

- **returned value**

  Type `PS::TimeProfile`.

- **function**

  Return the copy of the member variable `time_profile_`.

9.1.2.2.3.2 PS::DomainInfo::clearTimeProfile

PS::DomainInfo::clearTimeProfile

```
void PS::DomainInfo::clearTimeProfile();
```

- **arguments**

  void.

- **returned value**

  void.

- **function**

  Set all member variables of `time_profile_` to 0.

9.1.2.2.4 Obtain Information

The APIs of obtaining information is deleared bellow.

Listing 24: DomainInfo3

```
namespace ParticleSimulator {
    class DomainInfo{
    public:
        S64 getMemSizeUsed();
    };
}
```

9.1.2.2.4.1 PS::DomainInfo::getUsedMemorySize

PS::DomainInfo::getUsedMemorySize

```
PS::S64 PS::DomainInfo::getUsedMemorySize();
```

- **arguments**

  void.

- **returned value**

  Type `PS::S64`. Size of used memory of the calling object in the unit of byte.

- **function**

  Return the size of used memory of the calling object in the unit of byte.

9.1.3 ParticleSystem Class

This section describes `ParticleSystem` class. This class is used to keep the information of particles for the exchange of particles.

9.1.3.1 Creation of Object

The particle system class is declared below.

Listing 25: ParticleSystem0

```
namespace ParticleSimulator {
    template<class Tptcl>
    class ParticleSystem;
}
```

The template argument is user-defined `FullParticle` class

Next example shows how to create an object of `ParticleSystem` class. Here, the object is named `system`.

```
PS::ParticleSystem<FP> system;
```

9.1.3.2 API

`DomainInfo` class has APIs for initialization, obtaining information, file I/O and exchanging particles. The following subsections describe them.

9.1.3.2.1 Initial Setup

The APIs of initialization are declared below.

Listing 26: ParticleSystem1

```
namespace ParticleSimulator {
    template<class Tptcl>
    class ParticleSystem{
    public:
        ParticleSystem();
        void initialize();
        void setAverageTargetNumberOfSampleParticlePerProcess
                                (const S32 & nsampleperprocess);
    };
}
```

9.1.3.2.1.1 Constructor

Constructor

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::ParticleSystem();
```

- **arguments**

  void.

- **returned value**

  void.

- **function**

  Create an object of `ParticleSystem` class.

9.1.3.2.1.2 PS::ParticleSystem::initialize

PS::ParticleSystem::initialize

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::initialize();
```

- **arguments**

  void.

- **returned value**

  void.

- **function**

  Initialize an object of `ParticleSystem` class. Users must call this API once.

9.1.3.2.1.3 PS::ParticleSystem:: setAverageTargetNumberOfSampleParticlePerProcess

PS::ParticleSystem::
setAverageTargetNumberOfSampleParticlePerProcess

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::setAverageTargetNumberOfSampleParticlePerProcess
                 (const PS::S32 & nsampleperprocess);
```

- **arguments**

  `nsampleperprocess`: input. Type `const PS::S32 &`. The average number of sample particles per MPI process.

- **returned value**

  void.

- **function**

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

9.1.3.2.2 Obtain Information

The APIs of obtaining information are declared below.

Listing 27: ParticleSystem2

```
namespace ParticleSimulator {
    template<class Tptcl>
    class ParticleSystem{
    public:
        Tptcl & operator [] (const S32 id);
        S32 getNumberOfParticleLocal() const;
        S32 getNumberOfParticleGlobal() const;
        S64 getMemSizeUsed() const;
    };
}
```

9.1.3.2.2.1 PS::ParticleSystem::operator []

PS::ParticleSystem::operator []

```
template <class Tptcl>
Tptcl & PS::ParticleSystem<Tptcl>::operator []
             (const PS::S32 id);
```

- **arguments**

  `id`: input. Type `const PS::S32`. Index of the particle array in the calling object.

- **returned value**

  Type `FullParticle &`.

- **function**

  Return the reference of the `id`-th object of `FullParticle` class.

9.1.3.2.2.2 PS::ParticleSystem::getNumberOfParticleLocal
PS::ParticleSystem::getNumberOfParticleLocal

```
template <class Tptcl>
PS::S32 PS::ParticleSystem<Tptcl>::getNumberOfParticleLocal();
```

- **arguments**

  void.

- **returned value**

  Type `PS::S32`. The number of particles of the calling process.

- **function**

  Return the number of particles of the calling process.

9.1.3.2.2.3 PS::ParticleSystem::getNumberOfParticleGlobal
PS::ParticleSystem::getNumberOfParticleGlobal

```
template <class Tptcl>
PS::S32 PS::ParticleSystem<Tptcl>::getNumberOfParticleGlobal();
```

- **arguments**

  void.

- **returned value**

  Type PS::S32. The total number of particles of all processes.

- **function**

  Return the total number of particles of all processes.

9.1.3.2.2.4 PS::DomainInfo::getUsedMemorySize

PS::DomainInfo::getUsedMemorySize

```
PS::S64 PS::DomainInfo::getUsedMemorySize();
```

- **arguments**

  void.

- **returned value**

  PS::S64. The size of used memory of the calling object in the unit of byte.

- **function**

  Return the size of used memory of the calling object in the unit of byte.

9.1.3.2.3 File I/O

The APIs of file I/O are declared below.

Listing 28: ParticleSystem3

```
namespace ParticleSimulator {
    template<class Tptcl>
    class ParticleSystem{
    public:
        template <class Theader>
        void readParticleAscii(const char * const filename,
                               const char * const format,
                               Theader & header);
        void readParticleAscii(const char * const filename,
                               const char * const format);
        template <class Theader>
        void readParticleAscii(const char * const filename,
                               Theader & header);
        void readParticleAscii(const char * const filename);
        template <class Theader>
        void writeParticleAscii(const char * const filename,
                                const char * const format,
                                const Theader & header);
        void writeParticleAscii(const char * const filename,
                                const char * format);
        template <class Theader>
        void writeParticleAscii(const char * const filename,
                                const Theader & header);
        void writeParticleAscii(const char * const filename);
    };
}
```

9.1.3.2.3.1 PS::ParticleSystem::readParticleAscii

PS::ParticleSystem::readParticleAscii

```
template <class Tptcl>
template <class Theader>
void PS::ParticleSystem<Tptlc>::readParticleAscii
                 (const char * const filename,
                  const char * const format,
                  Theader & header);
```

- **arguments**

  `filename`: input. Type `const char *`. Prefix of a name of an input file.

  `format`: input. Type `const char *`. File format.

  `header`: input. Type Hdr &. Header information of a file.

- **returned value**

  void.

- **function**

  Each process reads the input file defined by `filename` and `format` and set the information of the particles to `FullParticle` object.

  Format specifier is the same as that of the standard C library. There must be three format specifiers. The first is string type and both the second and third are integer. The first string is set to be `filename`. The second and third integers are the number of the total processes and the rank id of the calling process, respectively. For instance, if `filename` is `nbody`, `format` is `%s_%03d_%03d.init` and the number of total process is 64, the process with rank id of 12 reads the file, `nbody_064_012.init`.

  If users use this API, users must define two functions: one is a method of `FullParticle`, the other is a method of `Hdr`. For more information, please refer to sections A.1.4.3 and A.7.

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::readParticleAscii
            (const char * const filename,
             const char * const format);
```

- **arguments**

  `filename`: input. Type `const char *`. Name of an input file.

  `format`: input. Type `const char *`. File format.

- **returned value**

  void.

- **function**

  Each process reads the input file defined by `filename` and `format` and set the information of the particles to `FullParticle`-type objects.

  Format specifier is the same as that of the standard C library. There must be three format specifiers. The first is string type and both the second and third are integer. The first string is set to be `filename`. The second and third integers are the number of the total processes and the rank id of the calling process, respectively. For instance, if `filename` is `nbody`, `format` is `%s_%03d_%03d.init` and the number of total process is 64, the process with rank id of 12 reads the file, `nbody_064_012.init`.

  If users use this API, users must define two functions: one is a method of `FullParticle` class. For more information, please refer to sections A.1.4.3.

```
template <class Tptcl>
template <class Theader>
void PS::ParticleSystem<Tptcl>::readParticleAscii
            (const char * const filename,
             Theader & header);
```

- **arguments**

  `filename`: input. Type `const char *`. Name of an input file.

  `header`: input. Type `Hdr &`. Header information of an input file.

- **returned value**

  void.

- **function**

  The root process reads the information of particles from the input file named `filename` and send their information to other processes. If a user use this API, the user must define the methods `FullParticle::writeAscii` and `Hdr::writeAscii`. For more information, please refer to sections A.1.4.3.

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::readParticleAscii
            (const char * const filename);
```

- **arguments**

  `filename`: input. Type `const char * const`. Name of input file.

- **returned value**

  void.

- **function**

  The root process reads the information of particles from the input file named `filename` and send their information to other processes. If users use this API, the users must define the methods `PS::FullParticle::writeAscii`. For more information, please refer to sections A.1.4.3.

9.1.3.2.3.2 PS::ParticleSystem::readParticleBinary
PS::ParticleSystem::readParticleBinary

Not implemented yet.

9.1.3.2.3.3 PS::ParticleSystem::writeParticlAscii
PS::ParticleSystem::writeParticlAscii

```
template <class Tptcl>
template <class Theader>
void PS::ParticleSystem<Tptcl>::writeParticleAscii
            (const char * const filename,
             const char * const format,
             const Theader & header);
```

- **arguments**

  `filename`: input. Type `const char *`. Prefix of a name of an input file.

  `format`: input. Type `const char *`. File format.

  `header`: input. Type `Hdr &`. Header information of an input file.

- **returned value**

  void.

- **function**

  Each process writes the information of the `FullParticle` class in the output file defined by `filename` and `format`. The format is the same as that in the method `PS::ParticleSystem::readParticleAscii`. If users use this API, the users must define the two methods: `FullParticle::writeAscii` and `Hdr::writeAscii`. For more information, please refer to sections A.1.4.3.

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::writeParticleAscii
            (const char * const filename,
             const char * const format);
```

- **arguments**

  filename: input. Type const char * const. Prefix of a name of an output filename.

  format: input. Type const char * const. File format.

- **returned value**

  void.

- **function**

  Each process writes the information of the FullParticle class in the output file defined by filename and format. The format is the same as that in the method PS::ParticleSystem::readParticleAscii. If users use this API, the users must define the methods PS::FullParticle::writeAscii. For more information, please refer to sections A.1.4.3.

```
template <class Tptcl>
template <class Theader>
void PS::ParticleSystem<Tptcl>::writeParticleAscii
            (const char * const filename,
             const Theader & header);
```

- **arguments**

  filename: input. Type const char *. Name of an output file.

  header: input. Type const Hdr &. Object of Hdr class.

- **returned value**

  void.

- **function**

  Write the data of the FullParticle class in the calling object and header information defined by header to the output file named filename. If users use this API, the users must define the two methods: FullParticle::writeAscii and Hdr::writeAscii. For more information, please refer to sections A.1.4.3.

```
void PS::ParticleSystem<Tptcl>::writeParticleAscii
            (const char * const filename);
```

- **arguments**

  filename: output. Type const char *. Name of an output file.

- **returned value**

  void.

- **function**

  Write the data of the `FullParticle` class in the calling object to the output file named `filename`. If users use this API, the users must define the method `PS::FullParticle::writeAscii`. For more information, please refer to sections A.1.4.3.

9.1.3.2.3.4 PS::ParticleSystem::writeParticleBinary
PS::ParticleSystem::writeParticleBinary

Not implemented yet.

9.1.3.2.4 Exchange Particles

The APIs of exchange particle are declared below.

Listing 29: ParticleSystem4

```
namespace ParticleSimulator {
    template<class Tptcl>
    class ParticleSystem{
    public:
        template<class Tdinfo>
        void exchangeParticle(Tdinfo & dinfo);
    };
}
```

9.1.3.2.4.1 PS::ParticleSystem::exchangeParticle
PS::ParticleSystem::exchangeParticle

```
template <class Tptcl>
template <class Tdinfo>
void PS::ParticleSystem<Tptcl>::exchangeParticle
                  (Tdinfo & dinfo);
```

- **arguments**

  `dinfo`: input. Type `DomainInfo &`.

- **returned value**

  void.

- **function**

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

9.1.3.2.5 Time Measurement

The APIs of Time Measurement are given below. When a member function is called, its execution time is set to the private member `time_profile_`. Until the method, `clearTimeProfile()`, execution times are accumulated.

Listing 30: ParticleSystem3

```
namespace ParticleSimulator {
    template<class Tptcl>
    class ParticleSystem{
    private:
        TimeProfile time_profile_;
    public:
        TimeProfile getTimeProfile();
        void clearTimeProfile();

    };
}
```

9.1.3.2.5.1 PS::ParticleSystem::getTimeProfile

PS::ParticleSystem::getTimeProfile

```
PS::TimeProfile PS::ParticleSystem::getTimeProfile();
```

- **arguments**

  void.

- **returned value**

  Type `PS::TimeProfile`.

- **function**

  Return the copy of the member variable `time_profile_`.

9.1.3.2.5.2 PS::ParticleSystem::clearTimeProfile

PS::ParticleSystem::clearTimeProfile

```
void PS::ParticleSystem::clearTimeProfile();
```

- **arguments**

  void.

- **returned value**

  void.

- **function**

  Set all member variables of `time_profile_` to 0.

9.1.3.2.6 Others

Other APIs are declared below.

Listing 31: ParticleSystem4

```
namespace ParticleSimulator {
    template<class Tptcl>
    class ParticleSystem{
    public:
        template<class Tdinfo>
        void adjustPositionIntoRootDomain
                    (const Tdinfo & dinfo);
        void setNumberOfParticleLocal(const S32 n);
    };
}
```

9.1.3.2.6.1 PS::ParticleSystem::adjustPositionIntoRootDomain

PS::ParticleSystem::adjustPositionIntoRootDomain

```
template <class Tptcl>
template <class Tdinfo>
void ParticleSystem<Tptcl>::adjustPositionIntoRootDomain
          (const Tdinfo & dinfo);
```

- **arguments**

  `dinfo`: input. Type `const Tdinfo &`. Object of the domain information class.

- **returned value**

  void.

- **function**

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

9.1.3.2.6.2 PS::ParticleSystem::setNumberOfParticleLocal

PS::ParticleSystem::setNumberOfParticleLocal

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::setNumberOfParticleLocal
              (const PS::S32 n);
```

- **arguments**

  `n`: input. Type `const PS::S32`. The number of particles of the calling process.

- **returned value**

  void.

- **function**

  Set the number of particles of the calling process.

9.1.4 TreeForForce Class

This section describes `TreeForForce` class. This class is a module to calculate interactions between particles.

9.1.4.1 Creation of Object

This class is declared below.

Listing 32: TreeForForce0

```
namespace ParticleSimulator {
    template<class TSearchMode,
             class TResult,
             class TEpi,
             class TEpj,
             class TMomLocal,
             class TMomGlobal,
             class TSpj>
    class TreeForForce;
}
```

The types of template arguments are `PS::SEARCH_MODE class` (user selected), `Force` class (user defined), `EssentialParticleI` class (user defined), `EssentialParticleJ` class (user defined), `Moment` class for local tree (user defined), `Moment` class for global tree (user defined) and `SuperParticleJ` class (user defined).

Depending on `PS::SEARCH_MODE`, the wrapper classes to reduce the template arguments are available. Using these wrapper classes is highly recommended.

The following sections describe how to create objects by using the wrappers.

9.1.4.1.1 PS::SEARCH_MODE_LONG

PS::SEARCH_MODE_LONG_SCATTER is similar.

This section describes the wrapper class for `PS::SEARCH_MODE_LONG`.

Next example shows how to create an object of `PS::SEARCH_MODE_LONG` class. Here, the object is named `system`.

```
PS::TreeForForceLong<TResult, TEpi, TEpj, TMom, TSpj>::Normal system;
```

The template arguments are `Force` class, `EssentailParticleI` class, `EssentailParticleJ` class, `Moment` class for both local and global trees and `SuperParticleJ` class. FDPS prepare some useful classes of the Moment and the SPJ (see section 7.5, 7.6). Users can use them.

More specialized wrappers are also prepared for some cases.

The next wrapper is for the case that the multipole expansion is up to the monopole and the expansion center is the barycenter of particles.

```
PS::TreeForForceLong<TResult, TEpi, TEpj>::Monopole system;
```

The next wrapper is for the case that the multipole expansion is up to the quadrupole and the expansion center is the barycenter of particles.

```
PS::TreeForForceLong<TResult, TEpi, TEpj>::Quadrupole system;
```

The next wrapper is for the case that the multipole expansion is up to the monopole and the expansion center is the geometric center of particles (not charge-weighted).

```
PS::TreeForForceLong<TResult, TEpi, TEpj>::MonopoleGeometricCenter system;
```

The next wrapper is for the case that the multipole expansion is up to the dipole and the expansion center is the geometric center of particles (not charge-weighted).

```
PS::TreeForForceLong<TResult, TEpi, TEpj>::DipoleGeometricCenter system;
```

The next wrapper is for the case that the multipole expansion is up to the quadrupole and the expansion center is the geometric center of particles (not charge-weighted).

```
PS::TreeForForceLong<TResult, TEpi, TEpj>::QuadrupoleGeometricCenter system;
```

For `PS::SEARCH_MODE_LONG_SCATTER`, two types in which the `Moment` class and `SuperParticleJ` are specified are provided. They are the following two types corresponding to `MomentMonopoleScatter` and `MomentQuadruopoleScatter`

```
PS::TreeForForceLong<TResult, TEpi, TEpj>::MonopoleWithScatterSearch system;
```

```
PS::TreeForForceLong<TResult, TEpi, TEpj>::QuadrupoleWithScatterSearch system;
```

The template arguments are the same as those for `PS::SEARCH_MODE_LONG`.

9.1.4.1.2 PS::SEARCH_MODE_LONG_CUTOFF

This section describes the wrapper class for PS::SEARCH_MODE_LONG_CUTOFF.

Next example shows how to create an object of `PS::SEARCH_MODE_LONG_CUTOFF` class. Here, the object is named `system`.

```
PS::TreeForForceLong<TResult, TEpi, TEpj, TMom, TSpj>::WithCutoff system;
```

The template arguments are `Force` class, `EssentialParticleI` class, `EssentialParticleJ` class, `Moment` class for both local and global trees and `SuperParticleJ` class. FDPS prepare some useful classes of `Moment` and `SuperParticleJ` (see section 7.5, 7.6). Users can use them.

There is the more specialized wrapper for the case that the multipole expansion is up to the monopole and the expansion cetner is the barycenter of particles.

```
PS::TreeForForceLong<TResult, TEpi, TEpj>::MonopoleWithCutoff system;
```

9.1.4.1.3 PS::SEARCH_MODE_GATHER

This wrapper is for the gather mode search. The template arguments are `Force` class, `EssentialParticleI` class and `EssentialParticleJ` class.

Next example shows how to create an object of `PS::SEARCH_MODE_GATHER` class. Here, the object is named `system`.

```
PS::TreeForForceShort<TResult, TEpi, TEpj>::Gather system;
```

9.1.4.1.4 PS::SEARCH_MODE_SCATTER

This wrapper is for the scatter mode search. The template arguments are `Force` class, `EssentialParticleI` class and `EssentialParticleJ` class.

Next example shows how to create an object of `PS::SEARCH_MODE_SCATTER` class. Here, the object is named `system`.

```
PS::TreeForForceShort<TResult, TEpi, TEpj>::Scatter system;
```

9.1.4.1.5 PS::SEARCH_MODE_SYMMETRY

This wrapper is for the symmetry mode search. The template arguments are `Force` class, `EssentialParticleI` class and `EssentialParticleJ` class.

Next example shows how to create an object of `PS::SEARCH_MODE_SYMMETRY` class. Here, the object is named `system`.

```
PS::TreeForForceShort<TResult, TEpi, TEpj>::Symmetry system;
```

9.1.4.2 API

This module has APIs for initial setup, calculation of interaction and searching neighbors. The following subsections describe them.

Next examples are declarations of member functions of the TreeForForce class.

```
template <class TSearchMode,
          class TResult,
          class TEpi,
          class TEpj,
          class TMomLocal,
          class TMomGlobal,
          class TSpj>
void PS::TreeForForce<TSearchMode,
                      TEpi,
                      TEpj,
                      TMomLocal,
                      TMomGlobal,
                      TSpj>::MemberFunction1();

template <class TSearchMode,
          class TResult,
          class TEpi,
          class TEpj,
          class TMomLocal,
          class TMomGlobal,
          class TSpj>
template <class TTT>
void PS::TreeForForce<TSearchMode,
                      TEpi,
                      TEpj,
                      TMomLocal,
                      TMomGlobal,
                      TSpj>::MemberFunction2(TTT arg1);
```

In this section, to simplify notation, the template arguments of the TreeForForce class are omitted as follows.

```
void PS::TreeForForce::MemberFunction1();

template <class TTT>
void PS::TreeForForce::MemberFunction2(TTT arg1);
```

9.1.4.2.1 Initial Setup

The APIs of initial setup of `TreeForForce` class are declared below.

Listing 33: TreeForForce1

```
namespace ParticleSimulator {
    template<class TSearchMode,
             class TResult,
```

```
            class TEpi,
            class TEpj,
            class TMomLocal,
            class TMomGlobal,
            class TSpj>
    class TreeForForce{
    public:
    void TreeForForce();
    void initialize(const U64 n_glb_tot,
                    const F32 theta=0.7,
                    const U32 n_leaf_limit=8,
                    const U32 n_group_limit=64);
    };
}
```

9.1.4.2.1.1 Constructor

Constructor

```
void PS::TreeForForce::TreeForForce();
```

- **arguments**

  void.

- **returned value**

  void.

- **function**

  Create an object of `TreeForForce` class.

9.1.4.2.1.2 PS::TreeForForce::initialize

PS::TreeForForce::initialize

```
void PS::TreeForForce::initialize
            (const PS::U64 n_glb_tot,
             const PS::F32 theta=0.7,
             const PS::U32 n_leaf_limit=8,
             const PS::U32 n_group_limit=64);
```

- **arguments**

  `n_glb_tot`: input. Type `const PS::U64`. Total number of particles for all processes.

  `theta`: input. Type `const PS::F32`. Opening criterion of tree.

n_leaf_limit: input. Type `const PS::U32`. The maximum number of particles in leaf cell.

n_group_limit: input. Type `const PS::U32`. The maximum number of particles which share the same interactions list.

- **returned value**

  void.

- **function**

  Initialize an object of `PS::TreeForForce` class.

9.1.4.2.2 Low Level APIs

The low-level APIs for calculating forces are declared below.

Listing 34: TreeForForce1

```
namespace ParticleSimulator {
    template<class TSearchMode,
             class TResult,
             class TEpi,
             class TEpj,
             class TMomLocal,
             class TMomGlobal,
             class TSpj>
    class TreeForForce{
    public:
        template<class Tpsys>
        void setParticleLocalTree(const Tpsys & psys,
                                  const bool clear=true);
        template<class Tdinfo>
        void makeLocalTree(const Tdinfo & dinfo);
        void makeLocalTree(const F32 l,
                           const F32vec & c = F32vec(0.0));
        template<class Tdinfo>
        void makeGlobalTree(const Tdinfo & dinfo);
        void calcMomentGlobalTree();
        template<class Tfunc_ep_ep>
        void calcForce(Tfunc_ep_ep pfunc_ep_ep,
                       const bool clear=true);
        template<class Tfunc_ep_ep, class Tfunc_sp_ep>
        void calcForce(Tfunc_ep_ep pfunc_ep_ep,
                       Tfunc_sp_ep pfunc_sp_ep,
                       const bool clear=true);
        Tforce getForce(const S32 i);
    };
}
```

9.1.4.2.2.1 PS::TreeForForce::setParticleLocalTree

PS::TreeForForce::setParticleLocalTree

```
template<class Tpsys>
void PS::TreeForForce::setParticleLocalTree
            (const Tpsys & psys,
             const bool clear = true);
```

- **arguments**

  psys: input. Type `const ParticleSystem`. Object of `ParticleSystem` class

  clear: input. Type `const bool`. A flag to clear the data of particles, if set to `true`.

- **returned value**

  void.

- **function**

  Read the data of particles from `psys` and write them to the calling object. If `clear` is `true`, the data recorded are cleared.

9.1.4.2.2.2 PS::TreeForForce::makeLocalTree

PS::TreeForForce::makeLocalTree

```
template<class Tdinfo>
void PS::TreeForForce::makeLocalTree
            (const Tdinfo & dinfo);
```

- **arguments**

  dinfo: input. Type `const DomainInfo &`. Object of `DomainInfo` class.

- **returned value**

  void.

- **function**

  Make local tree. The root cell of the local tree is calculated by using `dinfo`.

```
template<class Tdinfo>
void PS::TreeForForce::makeLocalTree
            (const PS::F32 l,
             const PS::F32vec & c = PS::F32vec(0.0));
```

- **arguments**

  `l`: input. Type `const PS::F32`. Length of a side of root cell of local tree.

  `c`: input. Type `PS::F32vec`. Center coordinate of root cell of local tree.

- **returned value**

  void.

- **function**

  Make local tree. The two arguments must be the same for all processes. Otherwise, the results can be wrong.

9.1.4.2.2.3 PS::TreeForForce::makeGlobalTree

PS::TreeForForce::makeGlobalTree

```
template<class Tdinfo>
void PS::TreeForForce::makeGlobalTree
            (const Tdinfo & dinfo);
```

- **arguments**

  `dinfo`: input. Type `const DomainInfo &`. An object of `DomainInfo` class.

- **returned value**

  void.

- **function**

  Make global tree.

9.1.4.2.2.4 PS::TreeForForce::calcMomentGlobalTree

PS::TreeForForce::calcMomentGlobalTree

```
void PS::TreeForForce::calcMomentGlobalTree();
```

- **arguments**

  void.

- **returned value**

  void.

- **function**

  Calculate moments of cells of global tree.

9.1.4.2.2.5 PS::TreeForForce::calcForce

PS::TreeForForce::calcForce

```
template<class Tfunc_ep_ep>
void PS::TreeForForce::calcForce
            (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                     const PS::S32,
                                     TEpj *,
                                     const PS::S32,
                                     TResult *),
             const bool clear=true);
```

- **arguments**

  `pfunc_ep_ep`: input. A Functor of which a returned value is void and of which arguments are const `EssentialParticleI *` type, `const PS::S32` type, `const EssentailParticleJ *` type, `const PS::S32` type and `Result *` type.

  `clear`: input. Type `const bool`. A flag to clear the data of particles, if set to `true`.

- **returned value**

  void.

- **function**

  Calculate forces of all particles of the calling object. This function can be used only if `PS::SEARCH_MODE` is `PS::SEARCH_MODE_GATHER`, `PS::SEARCH_MODE_SCATTER` and `PS::SEARCH_MODE_SYMMETRY`.

```
template<class Tfunc_ep_ep, class Tfunc_sp_ep>
void PS::TreeForForce::calcForce
            (Tfunc_ep_ep pfunc_ep_ep,
             Tfunc_ep_ep pfunc_sp_ep,
             const bool clear=true);
```

- **arguments**

  `pfunc_ep_ep`: input. A Functor of which a returned value is void and of which arguments are `const EssentialParticleI *` type, `const PS::S32` type, `const EssentialParticleJ *` type, `const PS::S32` type and `Result` * type.

  `pfunc_sp_ep`: input. A Functor of which a returned value is void and of which arguments are `const EssentailParticleI *` type, `const PS::S32` type, `const SuperParticleJ *` type, `const PS::S32` type and `Result *` type.

  `clear`: input. Type `const bool`. A flag to clear the data of forces, if set to `true`.

- **returned value**

  void.

- **function**

  Calculate forces of all particles loaded in the calling object. This function can be used only if `PS::SEARCH_MODE` is `PS::SEARCH_MODE_GATHER`, `PS::SEARCH_MODE_SCATTER` or `PS::SEARCH_MODE_SYMMETRY`.

9.1.4.2.2.6 PS::TreeForForce::getForce

PS::TreeForForce::getForce

```
TResult PS::TreeForForce::getForce(const PS::S32 i);
```

- **arguments**

  `i`: input. Type `const PS::S32`. Index of array of particles.

- **returned value**

  Type `Force`. Force on the i-th particle.

- **function**

  Return the force on the i-th particle.

9.1.4.2.3 High Level APIs

The high level APIs for interactions are declared below.

Listing 35: TreeForForce1

```
namespace ParticleSimulator {
    template<class TSearchMode,
             class TResult,
             class TEpi,
             class TEpj,
             class TMomLocal,
             class TMomGlobal,
             class TSpj>
    class TreeForForce{
    public:
        template<class Tfunc_ep_ep,
                 class Tpsys,
                 class Tdinfo>
        void calcForceAllAndWriteBack
                    (Tfunc_ep_ep pfunc_ep_ep,
                     Tpsys & psys,
                     Tdinfo & dinfo,
                     const bool clear_force = true);
        template<class Tfunc_ep_ep,
                 class Tfunc_sp_ep,
```

```
              class Tpsys,
              class Tdinfo>
    void calcForceAllAndWriteBack
                (Tfunc_ep_ep pfunc_ep_ep,
                 Tfunc_sp_ep pfunc_sp_ep,
                 Tpsys & psys,
                 TDinfo & dinfo,
                 const bool clear_force=true);
    template<class Tfunc_dispatch,
             class Tfunc_retrieve,
             class Tpsys,
             class Tdinfo>
    void TreeForForce::calcForceAllandWriteBackMultiWalk
         (Tfunc_dispatch pfunc_dispatch,
          Tfunc_retrieve pfunc_retrieve,
          Tpsys & psys,
          Tdinfo & dinfo,
          const PS::S32 nwalk,
          const bool clear=true);

    template<class Tfunc_ep_ep,
             class Tfunc_sp_ep,
             class Tpsys,
             class Tdinfo>
    void calcForceAll
                (Tfunc_ep_ep pfunc_ep_ep,
                 Tfunc_sp_ep pfunc_sp_ep,
                 Tpsys & psys,
                 Tdinfo & dinfo,
                 const bool clear_force=true);
    template<class Tfunc_ep_ep,
             class Tfunc_sp_ep,
             class Tpsys,
             class Tdinfo>
    void calcForceAll(Tfunc_ep_ep pfunc_ep_ep,
                      Tfunc_sp_ep pfunc_sp_ep,
                      Tpsys & psys,
                      Tdinfo & dinfo,
                      const bool clear_force=true);

    template<class Tfunc_ep_ep,
             class Tdinfo>
    void calcForceMakeingTree
                (Tfunc_ep_ep pfunc_ep_ep,
                 Tdinfo & dinfo,
                 const bool clear_force=true);
```

```
        template<class Tfunc_ep_ep,
                 class Tfunc_sp_ep,
                 class Tdinfo>
        void calcForceMakingTree
                    (Tfunc_ep_ep pfunc_ep_ep,
                     Tfunc_sp_ep pfunc_sp_ep,
                     Tdinfo & dinfo,
                     const bool clear_force=true);

        template<class Tfunc_ep_ep,
                 class Tpsys>
        void calcForceAndWriteBack
                    (Tfunc_ep_ep pfunc_ep_ep,
                     Tpsys & psys,
                     const bool clear=true);
        template<class Tfunc_ep_ep,
                 class Tfunc_sp_ep,
                 class Tpsys>
        void calcForceAndWriteBack
                    (Tfunc_ep_ep pfunc_ep_ep,
                     Tfunc_sp_ep pfunc_sp_ep,
                     Tpsys & psys,
                     const bool clear=true);
    };
}
namespace PS = ParticleSimulator;
```

9.1.4.2.3.1 PS::TreeForForce::calcForceAllAndWriteBack

PS::TreeForForce::calcForceAllAndWriteBack

```
template<class Tfunc_ep_ep,
         class Tpsys,
         class Tdinfo>
void PS::TreeForForce::calcForceAllandWriteBack
             (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                      const PS::S32,
                                      TEpj *,
                                      const PS::S32,
                                      TResult *),
              Tpsys & psys,
              Tdinfo & dinfo
              const bool clear=true);
```

- **arguments**

pfunc_ep_ep: input. A Functor of which a returned value is void and of which arguments are const EssentialParticleI * type, const PS::S32 type, const EssentialParticleJ * type, const PS::S32 type and Result * type.

psys: input and output. Type ParticleSystem &. Object of ParticleSystem class.

dinfo: input. Type DomainInfo &. Object of DomainInfo class

clear: input. Type const bool. A flag to clear the data of forces if set to true.

- **returned value**

  void.

- **function**

  Calculate forces of all particles in psys and write back these forces to psys. This function can be used only if PS::SEARCH_MODE is PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER and PS::SEARCH_MODE_SYMMETRY.

```
template<class Tfunc_ep_ep,
         class Tfunc_sp_ep,
         class Tpsys,
         class Tdinfo>
void PS::TreeForForce::calcForceAllandWriteBack
            (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                     const PS::S32,
                                     TEpj *,
                                     const PS::S32,
                                     TResult *),
             Tfunc_sp_ep pfunc_sp_ep(TEpi *,
                                     const PS::S32,
                                     TSpj *,
                                     const PS::S32,
                                     TResult *),
             Tpsys & psys,
             Tdinfo & dinfo
             const bool clear=true);
```

- **arguments**

  pfunc_ep_ep: input. A Functor of which a returned value is void and of which arguments are const EssentialParticleI * type, const PS::S32 type, const EssentialParticleJ * type, const PS::S32 type and Result * type.

  pfunc_sp_ep: input. A Functor of which a returned value is void and of which arguments are const EssentialParticleI * type, const PS::S32 type, const SuperParticleJ * type, const PS::S32 type and Result * type.

  psys: input and output. Type ParticleSystem &. An object of ParticleSystem class.

`dinfo`: input. Type `DomainInfo &`. An object of `DomainInfo` class.

`clear`: input. Type `const bool`. A flag to clear the data of forces, if set to `true`.

- **arguments**

  void.

- **function**

  Calculate forces of particles in `psys` and write back these forces to `psys`. This function can be used only if `PS::SEARCH_MODE` is `PS::SEARCH_MODE_LONG` or `PS::SEARCH_MODE_LONG_CUTOFF`.

```
template<class Tfunc_dispatch,
         class Tfunc_retrieve,
         class Tpsys,
         class Tdinfo>
void PS::TreeForForce::calcForceAllandWriteBackMultiWalk
             (Tfunc_dispatch pfunc_dispatch,
              Tfunc_retrieve pfunc_retrieve,
              const PS::S32 tag_max,
              Tpsys & psys,
              Tdinfo & dinfo,
              const PS::S32 nwalk,
              const bool clear=true);
```

- **Arguments**

  `pfunc_dispatch`: Input. Returns void. A functor which receives the arrays of `EssentialParticleI`, `EssentialParticleJ` (and `SuperParticleJ`) and dispatch the interaction calculation. When `PS::SEARCH_MODE` is `PS::SEARCH_MODE_LONG` or `PS::SEARCH_MODE_LONG_CUTOFF`, this function should has the following form:

```
PS::S32 pfunc_dispatch(const PS::S32  tag,
                       const PS::S32  nwalk,
                       const TEpi**   iptcl,
                       const PS::S32* ni,
                       const TEpj**   jptcl_ep,
                       const PS::S32  nj_ep,
                       const TSpj**   jptcl_sp,
                       const PS::S32* nj_sp);
```

  This function returns zero upon normal completion.

  When `PS::SEARCH_MODE` is `PS::SEARCH_MODE_GATHER` or `PS::SEARCH_MODE_SCATTER`, or `PS::SEARCH_MODE_SYMMETRY`, this function should has the following form:

```
PS::S32 pfunc_dispatch(const PS::S32  tag,
                       const PS::S32  nwalk,
                       const TEpi**   iptcl,
                       const PS::S32* ni,
                       const TEpj**   jptcl_ep,
                       const PS::S32* nj_ep);
```

This function returns zero upon normal completion.

`pfunc_retrieve`: Input. A functor of type void. Returns the result calculated by the call to `pfunc_dispatch`. This function should have the following form:

```
void pfunc_retrieve(const PS::S32  tag,
                    const PS::S32  nwalk,
                    const PS::S32* ni,
                    TResult**      force);
```

Here, `tag` connects the call to `pfunc_dispatch()` and that to `pfunc_retrieve()`. The result of calculation by a call to `pfunc_dispatch` is retrieved by the call to `pfunc_retrieve()` with the same value of `tag`.

`tag_max`: Input. Type `const PS::S32`. The maximum number of tags. Tag values beteween zero and `tag_max -1` are allowed. Non-positive value causes error. The current version ignores the value greater than unity and uses tag vale of zero only.

`psys`: input. Type `Tpsys &` An object of `ParticleSystem` class.

`dinfo`: input. Type `DomainInfo &`. An object of `DomainInfo` class.

`nwalk` :input. Type `const PS::S32` The maximum number of interaction list to be sent by a single call to dispatch/retrive.

`clear`: inpuyt. Type `const bool`. If true, the interaction result array is cleared before starting the calculation. Default is true.

- **Return value**

  None

- **Function**

  Calculate all interactions between all particles in `psys` and store the result to `psys`. The multiwalk method is used for the interaction calculation. The maximum number of the interaction list is `nwalk`.

9.1.4.2.3.2 PS::TreeForForce::calcForceAll

PS::TreeForForce::calcForceAll

```
template<class Tfunc_ep_ep,
         class Tpsys,
         class Tdinfo>
void PS::TreeForForce::calcForceAll
             (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                      const PS::S32,
                                      TEpj *,
                                      const PS::S32,
                                      TResult *),
              Tpsys & psys,
              Tdinfo & dinfo
              const bool clear=true);
```

- **arguments**

  `pfunc_ep_ep`: input. A Functor of which a returned value is void and of which arguments are `const EssentialParticleI *` type, `const PS::S32` type, `const EssentialParticleJ *` type, `const PS::S32` type and `Result *` type.

  `psys`: input and output. Type `ParticleSystem &`. An object of `ParticleSystem` class.

  `dinfo`: input. Type `DomainInfo &`. An object of `DomainInfo` class.

  `clear`: input. Type `const bool`. A flag to clear the data of forces, if set to `true`.

- **returned value**

  void.

- **function**

  Calculate forces of particles in `psys`. This function can be used only if `PS::SEARCH_MODE` is `PS::SEARCH_MODE_SCATTER` or `PS::SEARCH_MODE_SYMMETRY`. This API works similar as that of `PS::TreeForForce::calcForceAllAndWriteBack` without writing back forces to `psys`.

```
template<class Tfunc_ep_ep,
         class Tfunc_sp_ep,
         class Tpsys,
         class Tdinfo>
void PS::TreeForForce::calcForceAll
             (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                      const PS::S32,
                                      TEpj *,
                                      const PS::S32,
                                      TResult *),
              Tfunc_sp_ep pfunc_sp_ep(TEpi *,
                                      const PS::S32,
                                      TSpj *,
                                      const PS::S32,
                                      TResult *),
              Tpsys & psys,
              Tdinfo & dinfo
              const bool clear=true);
```

- **arguments**

  `pfunc_ep_ep`: input. A Functor of which a returned value is void and of which arguments are `const EssentialParticleI *` type, `const PS::S32 type`, `const EssentialParticleJ *` type, `const PS::S32` type and `Result *` type.

  `pfunc_sp_ep`: input. A Functor of which a returned value is void and of which arguments are `const EssentialParticleI *` type, `const PS::S32` type, const `SuperParticleJ *` type, `const PS::S32` type and `Result *` type.

  `psys`: input. Type `ParticleSystem &`. An object of `ParticleSystem` class

  `dinfo`: input. Type `DomainInfo &`. An object of `DomainInfo` class

  `clear`: input. Type `const bool`. A flag to clear the data of forces, if set to `true`.

- **returned value**

  void.

- **function**

  Calculate forces of particles in `psys`. This function can be used only if `PS::SEARCH_MODE` is `PS::SEARCH_MODE_LONG` or `PS::SEARCH_MODE_LONG_CUTOFF`. This API works similar as that of `PS::TreeForForce::calcForceAllAndWriteBack` without writing back forces to `psys`.

9.1.4.2.3.3 PS::TreeForForce::calcForceAndWriteBack

PS::TreeForForce::calcForceAndWriteBack

```
template<class Tfunc_ep_ep,
         class Tpsys>
void PS::TreeForForce::calcForceAndWriteBack
             (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                      const PS::S32,
                                      TEpj *,
                                      const PS::S32,
                                      TResult *),
              Tpsys & psys,
              const bool clear=true);
```

- **arguments**

  pfunc_ep_ep: input. A Functor of which a returned value is void and of which arguments are const `EssentialParticleI *` type, `const PS::S32` type, `const EssentialParticleJ *` type, `const PS::S32` type and `Result *` type.

  psys: input. Type `ParticleSystem &`.

  clear: input. Type `const bool`. A flag to clear the data of forces, if set to `true`.

- **returned value**

  void.

- **function**

  Calculate forces of particles in `psys` by using the global tree created before. This function can be used only if `PS::SEARCH_MODE` is `PS::SEARCH_MODE_GATHER`, `PS::SEARCH_MODE_SCATTER` or `PS::SEARCH_MODE_SYMMETRY`. This API works similar as that of `PS::TreeForForce::calcForceAl` without reading particle from `psys`, making the local tree, making global tree and calculate momentum.

```
template<class Tfunc_ep_ep,
         class Tfunc_sp_ep,
         class Tpsys>
void PS::TreeForForce::calcForceAllandWriteBack
             (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                      const PS::S32,
                                      TEpj *,
                                      const PS::S32,
                                      TResult *),
              Tfunc_sp_ep pfunc_sp_ep(TEpi *,
                                      const PS::S32,
                                      TSpj *,
                                      const PS::S32,
                                      TResult *),
              Tpsys & psys,
              const bool clear=true);
```

- **arguments**

  `pfunc_ep_ep`: input. A Functor of which a returned value is void and of which arguments are `const EssentialParticleI *` type, `const PS::S32` type, `const EssentialParticleJ *` type, `const PS::S32` type and `Result *` type.

  `pfunc_sp_ep`: input. A Functor of which a returned value is void and of which arguments are `const EssentialParticleI *` type, `const PS::S32` type, `const SuperParticleJ *` type, `const PS::S32` type and `Result *` type.

  `psys`: output. Type `ParticleSystem &`.

  `clear`: input. Type `const bool`. A flag to clear the data of forces, if set to `true`.

- **returned value**

  void.

- **function**

  Calculate forces of particles in `psys` by using the global tree created previously. This function can be used only if `PS::SEARCH_MODE` is `PS::SEARCH_MODE_LONG` or `PS::SEARCH_MODE_LONG_C` This API works similar as that of `PS::TreeForForce::calcForceAllAndWriteBack` without reading particle from `psys`, making the local tree, making global tree and calculate momentum.

9.1.4.2.4 Neighbor List

9.1.4.2.4.1 getNeighborListOneParticle

getNeighborListOneParticle

```
template<class Tptcl>
PS::S32 PS::TreeForForce::getNeighborListOneParticle(const Tptcl & ptcl,
                                                     EPJ * & epj);
```

- **arguments**

  `ptcl`: input. Type `Tptcl &`. A particle to obtain its neighbor particles.

  `epj`: output. Type `EssentailParticleJ * &`. The pointer to the beginning of the array of neighbor particles of `ptcl`.

- **returned value**

  Type `PS::S32 &`. The number of neighbor particles of `ptcl`.

- **function**

  By using the global tree of the calling object, search the neighbors of `ptcl` and set the pointer to the beginning of the array of neighbors to `epj` and return the number of neighbors. Note that `epj` is the pointer to the first element of the array of the particle data of type `EPJ` (not the array of pointers to particles), and points to the internal buffer maintained by FDPS. Therefore, users must not call `free()` or `delete()` to `epj`. This function is thread safe. In other words, each thread has its own buffer. Since each thread has only one buffer, the content of the buffer is overwritten at each call to this function.

  This function is available only if `PS::SEARCH_MODE` is `PS::SEARCH_MODE_LONG_SCATTER`, `PS::SEARCH_MODE_LONG_CUTOFF_SCATTER` (not implemented yet), `PS::SEARCH_MODE_LONG_GATHER` (not implemented yet), `PS::SEARCH_MODE_LONG_CUTOFF_GATHER` (not implemented yet), `PS::SEARCH_MODE_LONG_SYMMETRY` (not implemented yet), `PS::SEARCH_MODE_LONG_CUTOFF_SYMMETRY` (not implemented yet). `ptcl` needs a member function `PS::F64vec getPos()` like `FullParticle` class. If `PS::SEARCH_MODE` is `PS::SEARCH_MODE_LONG_GATHER`, `PS::SEARCH_MODE_LONG_CUTOFF_GATHER`, `PS::SEARCH_MODE_LONG_SYMMETRY`, `PS::SEARCH_MODE_LONG_CUTOFF_SYMMETRY`, `ptcl` also needs a member function `PS::F64 getRSearch()`.

9.1.4.2.5 Time Measurment

The APIs for time measurment are declared below. When a member function is called, its execution time is set to the private member `time_profile_`. Until the method `PS::TreeForForce::clearTimeProfile()` is called, execution times are accumulated.

Listing 36: TreeForForce2

```
namespace ParticleSimulator {
    template<class TSearchMode,
             class TResult,
             class TEpi,
```

```
            class TEpj,
            class TMomLocal,
            class TMomGlobal,
            class TSpj>
    class TreeForForce{
    public:
        TimeProfile getTimeProfile();
        void clearTimeProfile();
    };
}
```

9.1.4.2.5.1 PS::TreeForForce::getTimeProfile

PS::TreeForForce::getTimeProfile

```
PS::TimeProfile PS::TreeForForce::getTimeProfile();
```

- **arguments**

  void.

- **returned value**

  Type `PS::TimeProfile`.

- **function**

  The execution time 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 are recorded appropriate members of the private member variable `time_profile_`, `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_`.

9.1.4.2.5.2 PS::TreeForForce::clearTimeProfile

PS::TreeForForce::clearTimeProfile

```
void PS::TreeForForce::clearTimeProfile();
```

- **arguments**

  void.

- **returned value**

  void.

- **function**

  Set all member variables of `time_profile_` to 0.

9.1.4.2.6 Obtain Information

The member functions of obtaining information are declared below.

Listing 37: TreeForForce2

```
namespace ParticleSimulator {
    template<class TSearchMode,
             class TResult,
             class TEpi,
             class TEpj,
             class TMomLocal,
             class TMomGlobal,
             class TSpj>
    class TreeForForce{
    public:
        Count_t getNumberOfInteractionEPEPLocal();
        Count_t getNumberOfInteractionEPSPLocal();
        Count_t getNumberOfInteractionEPEPGlobal();
        Count_t getNumberOfInteractionEPSPGlobal();
        void clearNumberOfInteraction();
        S64 getMemSizeTotalUsed();
    };
}
```

9.1.4.2.6.1 PS::TreeForForce::getNumberOfInteractionEPEPLocal

PS::TreeForForce::getNumberOfInteractionEPEPLocal

```
PS::CountT PS::TreeForForce::getNumberOfInteractionEPEPLocal();
```

- **arguments**

  void.

- **returned value**

  Type `PS::CountT`.

- **function**

  Return the number of interactions between `EssentialParticleI` and `EssentialParticleJ` in the calling process.

9.1.4.2.6.2 PS::TreeForForce::getNumberOfInteractionEPEPGlobal

PS::TreeForForce::getNumberOfInteractionEPEPGlobal

```
PS::CountT PS::TreeForForce::getNumberOfInteractionEPEPGlobal();
```

- **arguments**

  void.

- **returned value**

  Type `PS::CountT`.

- **function**

  Return the total number of interactions between `EssentialParticleI` and `EssentialParticleJ`, evaluated in all processes.

9.1.4.2.6.3 PS::TreeForForce::getNumberOfInteractionEPSPLocal

PS::TreeForForce::getNumberOfInteractionEPSPLocal

```
PS::Count\_t PS::TreeForForce::getNumberOfInteractionEPSPLocal();
```

- **arguments**

  void.

- **returned value**

  Type `PS::CountT`.

- **function**

  Return the number of interactions between `EssentialParticleI` and `SuperParticleJ` in the calling process.

9.1.4.2.6.4 PS::TreeForForce::getNumberOfInteractionEPSPGlobal

PS::TreeForForce::getNumberOfInteractionEPSPGlobal

```
PS::S64 PS::TreeForForce::getNumberOfInteractionEPSPGlobal();
```

- **arguments**

  void.

- **returned value**

  Type `PS::CountT`.

- **function**

  Return the total number of interactions between `EssentialParticleI` and SPJ, evaluated in all processes.

9.1.4.2.6.5 PS::TreeForForce::clearNumberOfInteraction

PS::TreeForForce::clearNumberOfInteraction

```
void PS::TreeForForce::clearNumberOfInteraction();
```

- **arguments**

  void.

- **returned value**

  void.

- **function**

  The counter for recording the number of interactions is set to 0.

9.1.4.2.6.6 PS::TreeForForce::getNumberOfWalkLocal

PS::TreeForForce::getNumberOfWalkLocal

```
PS::Count\_t PS::TreeForForce::getNumberOfWalkLocal();
```

- **arguments**

  void.

- **returned value**

  Type `PS::CountT`.

- **function**

  Return the number of tree traverse for the calling process.

9.1.4.2.6.7 PS::TreeForForce::getNumberOfWalkGlobal

PS::TreeForForce::getNumberOfWalkGlobal

```
PS::Count\_t PS::TreeForForce::getNumberOfWalkGlobal();
```

- **arguments**

  void.

- **returned value**

  Type `PS::S64`.

- **function**

  Return the total number of tree traverse for all processes.

9.1.4.2.6.8 PS::TreeForForce::getMemSizeUsed
PS::TreeForForce::getMemSizeUsed

```
PS::S64 PS::TreeForForce::getMemSizeUsed();
```

- **arguments**

  void.

- **returned value**

  Type `PS::S64`.

- **function**

  Return the size of used memory of an object in the unit of byte.

9.1.5 Comm Class

This section describes `PS::Comm` class which is a module used to communicate data among processes and keep the data for communication. Since the singleton pattern is applied in this class, users do not need to create an object.

9.1.5.1 API

The APIs are declared below.

Listing 38: Communication

```
namespace ParticleSimulator {
    class Comm{
    public:
        static S32 getRank();
        static S32 getNumberOfProc();
        static S32 getRankMultiDim(const S32 id);
        static S32 getNumberOfProcMultiDim(const S32 id);
        static bool synchronizeConditionalBranchAND
                    (const bool local);
        static bool synchronizeConditionalBranchOR
                    (const bool local);
        template<class T>
        static T getMinValue(const T val);
        template<class Tfloat, class Tint>
        static void getMinValue(const Tfloat f_in,
                                const Tint i_in,
                                Tfloat & f_out,
                                Tint & i_out);
        template<class T>
        static T getMaxValue(const T val);
```

```
        template<class Tfloat, class Tint>
        static void getMaxValue(const Tfloat f_in,
                                const Tint i_in,
                                Tfloat & f_out,
                                Tint & i_out );
        template<class T>
        static T getSum(const T val);
        template<class T>
        static void broadcast(T * val,
                              const S32 n,
                              const S32 src=0);
    };
}
```

9.1.5.1.1 PS::Comm::getRank

```
static PS::S32 PS::Comm::getRank();
```

- **arguments**

  void.

- **returned value**

  Type `PS::S32`.

- **function**

  Return the rank of the calling process.

9.1.5.1.2 PS::Comm::getNumberOfProc

```
static PS::S32 PS::Comm::getNumberOfProc();
```

- **arguments**

  void.

- **returned value**

  Type `PS::S32`. The total number of processes.

- **function**

  Return the total number of processes.

9.1.5.1.3 PS::Comm::getRankMultiDim

```
static PS::S32 PS::Comm::getRankMultiDim(const PS::S32 id);
```

- **arguments**

  `id`: input. Type `const PS::S32`. Id of axes. x-axis:0, y-axis:1, z-axis:2.

- **returned value**

  Type `PS::S32`. The rank of the calling process along `id`-th axis. In the case of two dimensional simulations, FDPS returns 0 for `id`=2.

9.1.5.1.4 PS::Comm::getNumberOfProcMultiDim

```
static PS::S32 PS::Comm::getNumberOfProcMultiDim(const PS::S32 id);
```

- `id`: input. Type const `PS::S32`. id of axes. x-axis:0, y-axis:1, z-axis:2.

- **returned value**

  Type `PS::S32`. The number of processes along `id`-th axis. In the case of two dimensional simulations, FDPS returns 1 for `id`=2.

9.1.5.1.5 PS::Comm::synchronizeConditionalBranchAND

```
static bool PS::Comm::synchronizeConditionalBranchAND(const bool local)
```

- **arguments**

  `local`: input. Type `const bool`.

- **returned value**

  `bool` type. Logical product of `local` over all processes.

- **function**

  Return logical product of `local` over all processes.

9.1.5.1.6 PS::Comm::synchronizeConditionalBranchOR

```
static bool PS::Comm::synchronizeConditionalBranchOR(const bool local);
```

- **arguments**

  `local`: input. Type `const bool`.

- **returned value**

  Type `bool`.

- **function**

  Return logical sum of `local` over all processes.

9.1.5.1.7 PS::Comm::getMinValue

```
template <class T>
static T PS::Comm::getMinValue(const T val);
```

- **arguments**

  `val`: input. Type `const T`. Type `T` is allowed to be `PS::F32`, `PS::F64`, `PS::S32`, `PS::S64`, `PS::U32` and `PS::U64`

- **returned value**

  Type `T`. The minimum value of val of all processes.

```
template <class Tfloat, class Tint>
static void PS::Comm::getMinValue(const Tfloat f_in,
                                  const Tint i_in,
                                  Tfloat & f_out,
                                  Tint & i_out);
```

- **arguments**

  `f_in`: input. Type `const Tfloat`.

  `i_in`: input. Type `const Tint`.

  `f_out`: output. Type `Tfloat &`. `Tfloat` must be `PS::F64` or `PS::F32`. The minimum value of `f_in` among all processes.

  `i_out`: output. Type `Tint &`. `Tint` must be `PS::S64`, `PS::S32`, `PS::U64` or `PS::U32`. The rank of the process of which `f_in` is the minimum.

- **returned value**

  void.

9.1.5.1.8 PS::Comm::getMaxValue

```
template <class T>
static T PS::Comm::getMaxValue(const T val);
```

- **arguments**

  `val`: input. Type `const T`.

- **returned value**

  Type `T`. The maximum value of `val` of all processes.

```
template <class Tfloat, class Tint>
static void PS::Comm::getMaxValue(const Tfloat f_in,
                                  const Tint i_in,
                                  Tfloat & f_out,
                                  Tint & i_out);
```

- **arguments**

  `f_in`: input. Type `const Tfloat`. Type `Tfloat` is `PS::F32` or `PS::F64`.

  `i_in`: input. Type `const Tint`. Type `Tint` is `PS::S32`.

  `f_out`: output. Type `Tfloat &`. The maximum value of `f_in` among all processes.

  `i_out`: output. Type `Tint &`. The rank of the process of which `f_in` is the maximum.

- **returned value**

  void.

9.1.5.1.9 PS::Comm::getSum

```
template <class T>
static T PS::Comm::getSum(const T val);
```

- **arguments**

  `val`: input. Type `const T`. Type T is allowed to be `PS::S32`, `PS::S64`, `PS::F32`, `PS::F64`, `PS::U32`, `PS::U64`, `PS::F32vec` and `F64vec`.

- **returned value**

  Type `T`. Return the global sum of `val`.

9.1.5.1.10 PS::Comm::broadcast

```
template <class T>
static void PS::Comm::broadcast(T * val,
                                const PS::S32 n,
                                const PS::S32 src=0);
```

- **arguments**

  `val`: input. Type `T *`. Type `T` is allowed to be any types.

  `n`: input. Type `const PS::S32`. The number of variables.

  `src`: input. Type `const PS::S32`. The rank of source process.

- **returned value**

  void.

- **function**

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

9.1.6 Other functions

In this section, functions defined directly in the namespace `ParticleSimulator` are described.

9.1.6.1 Timing functon

9.1.6.1.1 PS::GetWtime

```
inline PS::F64 PS::GetWtime();
```

- **Arguments**

  None.

- **Return value**

  Type `PS::F64`. Wall-clock time in seconds.

9.2 Extended Classes

9.2.1 Summary

This section describes two extended modules: one is the class for the Particle Mesh scheme and the other is the class for Phantom-GRAPE.

9.2.2 Particle Mesh Class

This section describes `ParticleMesh` class which is for the evaluation of interactions between particles using the Particle Mesh method. The cutoff of the S-2 type function is applied to the mesh force, and the cutoff radius is fixed to three times the mesh size. The coordinates of particles sent to `ParticleMesh` class should be in the range of [0,1). The following subsections describe how to create the object for Particle Mesh method, APIs, predefined macros and how to use the class.

9.2.2.1 Creation of Object

`ParticleMesh` class is declared bellow.

Listing 39: ParticleMesh0

```
namespace ParticleSimulator {
    namespace ParticleMesh {
        class ParticleMesh;
    }
}
```

Next example shows how to create an object of `ParticleMesh` class. Here, the object is named `pm`.

```
PS::PM::ParticleMesh pm;
```

9.2.2.2 API

`ParticleMesh` class has APIs for initialization and force evaluation. The following subsections describe them.

9.2.2.2.1 Initial Setup

Listing 40: ParticleMesh1

```
namespace ParticleSimulator {
    namespace ParticleMesh {
        class ParticleMesh{
            ParticleMesh();
        };
    }
}
```

9.2.2.2.1.1 Constructor

Constructor

```
void PS::PM::ParticleMesh::ParticleMesh();
```

- **arguments**

  void.

- **returned value**

  void.

- **function**

  Create an object of `ParticleMesh` class.

9.2.2.2.2 Low Level API

The low Level APIs are declared below.

Listing 41: ParticleMesh1

```
namespace ParticleSimulator {
    namespace ParticleMesh {
        class ParticleMesh{
            template<class Tdinfo>
            void setDomainInfoParticleMesh
                        (const Tdinfo & dinfo);
            template<class Tpsys>
            void setParticleParticleMesh
                        (const Tpsys & psys,
                         const bool clear=true);
            void calcMeshForceOnly();
            F32vec getForce(F32vec pos);
        };
    }
}
```

9.2.2.2.2.1 PS::PM::ParticleMesh::setDomainInfoParticleMesh

PS::PM::ParticleMesh::setDomainInfoParticleMesh

```
template<class Tdinfo>
void PS::PM::ParticleMesh::setDomainInfoParticleMesh
          (const Tdinfo & dinfo);
```

- **arguments**

  `dinfo`: input. Type `const DomainInfo &`. An object of `DomainInfo` class.

- **returned value**

  void.

- **function**

  Read the data from `dinfo`.

9.2.2.2.2.2 PS::PM::ParticleMesh::setParticleParticleMesh

PS::PM::ParticleMesh::setParticleParticleMesh

```
template<class Tpsys>
void PS::PM::ParticleMesh::setParticleParticleMesh
            (const Tpsys & psys,
             const bool clear=true);
```

- **arguments**

  `psys`: input. Type `const ParticleSystem &`. An object of `ParticleSystem` class.

  `clear`: input. Type `const bool`. A flag to determine if previous information of particles loaded is cleared.

- **returned value**

  void.

- **function**

  Read data of particles from `psys`.

9.2.2.2.2.3 PS::PM::ParticleMesh::calcMeshForceOnly
PS::PM::ParticleMesh::calcMeshForceOnly

```
void PS::PM::ParticleMesh::calcMeshForceOnly();
```

- **arguments**

  void.

- **returned value**

  void.

- **function**

  Calculate forces on grid points.

9.2.2.2.2.4 PS::PM::ParticleMesh::getForce
PS::PM::ParticleMesh::getForce

```
PS::F32vec PS::PM::ParticleMesh::getForce
             (PS::F32vec pos);
```

- **arguments**

  `pos`: input. Type `PS::F32vec`.

- **returned value**

  Type `PS::F32vec`. PM force at the position `pos`.

- **function**

  Return a PM force exerted at the position “pos”

9.2.2.2.3 High Level API

High level APIs are declared below.

Listing 42: ParticleMesh1

```
namespace ParticleSimulator {
    namespace ParticleMesh {
        class ParticleMesh{
            template<class Tpsys,
                     class Tdinfo>
            void calcForceAllAndWriteBack
                        (Tpsys & psys,
                         const Tdinfo & dinfo);
        };
    }
}
```

9.2.2.2.3.1 PS::PM::ParticleMesh::calcForceAllAndWriteBack

PS::PM::ParticleMesh::calcForceAllAndWriteBack

```
template<class Tpsys,
         class Tdinfo>
void PS::PM::ParticleMesh::calcForceAllAndWriteBack
            (Tpsys & psys,
             const Tdinfo & dinfo);
```

- **arguments**

  `psys`: input and output. Type `ParticleSystem &`. An object of the `ParticleSystem` class.

  `dinfo`: input. Type `const DomainInfo &`. An object of the `DomainInfo` class.

- **returned value**

  void.

- **function**

  Calculate forces between particles of `psys` and write back the forces to `psys`.

9.2.2.3 Predefined Macros

There are many macros predefined in this module and users must not define the same name macros. The predefined macros are listed bellow in alphabetical order.

- BINARY_BOUNDARY
- BOUNDARY_COMM_NONBLOCKING
- BOUNDARY_SMOOTHING
- BUFFER_FOR_TREE
- CALCPOT
- CLEAN_BOUNDARY_PARTICLE
- CONSTANT_TIMESTEP
- EXCHANGE_COMM_NONBLOCKING
- FFT3D
- FFTW3_PARALLEL
- FFTW_DOUBLE
- FIX_FFTNODE
- GADGET_IO
- GRAPE_OFF
- KCOMPUTER
- LONG_ID
- MAKE_LIST_PROF
- MERGE_SNAPSHOT
- MULTI_TIMESTEP
- MY_MPI_BARRIER
- N128_2H
- N256_2H
- N256_H
- N32_2H
- N512_2H

- NEW_DECOMPOSITION
- NOACC
- NPART_DIFFERENT_DUMP
- OMP_SCHDULE_DISABLE
- PRINT_TANIKAWA
- REVERSE_ENDIAN_INPUT
- REVERSE_ENDIAN_OUTPUT
- RMM_PM
- SHIFT_INITIAL_BOUNDARY
- STATIC_ARRAY
- TREE2
- TREECONSTRUCTION_PARALLEL
- TREE_PARTICLE_CACHE
- UNIFORM
- UNSTABLE
- USING_MPI_PARTICLE
- VERBOSE_MODE
- VERBOSE_MODE2。

9.2.2.4 How To Use Particle Mesh Class

Through the following steps, one can use this `ParticleMesh` class.

1. Compile ParticleMesh class.
2. Write a program including FDPS.
3. Compile the program.

The following subsections describe these steps.

9.2.2.4.1 Compile of Particle Mesh Class

Users need to edit the `Makefile` in `$(FDPS)/src/particle_mesh` appropriately as follows.

- Set the macro `INCLUDE_FFTW` (the directory in which the header file of FFTW is) in Makefile.
- Set the macro `SIZE_OF_MESH` (the number of meshes along one direction) in `param_fdps.h`. Recommended value is $N^{1/3}/2$ (N is the number of the particle).

After editing, users run make command.
If successful, the library `libpm.a` and the header file `particle_mesh.hpp` are created.

9.2.2.4.2 Writing Source Code

Next, users write programs as follows.

- Include the header files created above.
- Add member functions described below to `FullParticle` class (here, this particle class called FP).
  - void `FP::copyFromForceParticleMesh(const PS::F32vec & force)`. In this function, copy `force` to arbitrary member variables in FP.
  - `PS::F64 FP::getChargeParticleMesh()`. This function returns mass.

9.2.2.4.3 Compile of Source Code

To compile the programs written above, users need to follow the following steps.

- Specify the path to the header file `particle_mesh.hpp`.
- Link the library `libpm.a`.
- Specify the path to the directory in which the header files of FFTW are.
- Link the FFTW libraries.

9.2.2.4.4 Note

`ParticleMesh` class does not work if the total number of processes is one.

9.2.3 Phantom-GRAPE for X86

To use the low precision Phantom-GRAPE, refer to Tanikawa et al.(2013, New Astronomy, 19, 74). In this library, interactions with cutoff is also available. To use the high precision Phantom-GRAPE, refer to Tanikawa et al.(2012, New Astronomy, 17, 82).

10 Detection of Errors

10.1 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.

10.2 Compile time errors

(Not written yet)

10.3 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.

10.3.1 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"

10.3.2 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”

10.3.3 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.

10.3.4 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”

10.3.5 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”

10.3.6 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"
```

10.3.7 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.

10.3.8 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.

10.3.9 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"
```

10.3.10 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"
```

10.3.11 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"

11 Known Bugs

12 Limitation

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

13 User Support

14 License

15 Change Log

- 2015/03/13
  - Correct `getRsearch` to `getRSearch`. (section 7).
  - Add a function `PS::Abort` (section 8).
- 2015/03/17
  - Release version 1.0
- 2015/03/18
  - Modify the licence related to class Particle Mesh.
- 2015/03/20
  - Add the description of `PS::Comm::broadcast`.
- 2015/04/01
  - Add a caution that class Particle Mesh does not work in the case of 1 MPI process.
- 2015/10/07
  - Add the description of PM method. Section 9.2.2。
- 2015/12/01
  - Add the description of Multiwalk method. Sections A.10、A.11、9.1.4.2.3.1、7.11、7.12。
- 2016/02/19
  - Added descriptions for `getNeighborListOneParticle` in 9.1.4.2.4.
  - In accordance with the above change, added the descriptions of `PS:: MomentMonopoleScatter` and `PS::MomentQuadrupoleScatter` in section A.4.2.2.
  - Added the descriptions of their wrappers, `MonopoleWithScatter` and `QuadrupoleWithScatter`, to section 9.1.4.1.1.

A Examples of implementation using use-defined classes

A.1 Class FullParticle

A.1.1 Summary

Class FullParticle has all the data of a particle, and a user-defined class given to FDPS with procedure 0 in section 2.3. Users can attach any member variables and functions to this class. However, they need to define several member functions with fixed names. This is because FDPS accesses information of class FullParticle, using these member functions. In what follows, we describe presumptions in this section, necessary member functions, and member functions necessary for some situations.

A.1.2 Presumptions

In this section, we use class FP as an example of class FullParticle. Users can change the name of FP as they like.

```
class FP;
```

A.1.3 Necessary member funcitons

A.1.3.1 Summary

Necessary member functions are functions FP::getPos and FP::copyFromForce. Using the function FP::getPos, FDPS reads a position from class FullParticle. Using the function FP::copyFromForce, FDPS writes results of interaction calculation to class FullParticle. We present description examples of these functions.

A.1.3.2 FP::getPos

```
class FP {
public:
    PS::F64vec getPos() const;
};
```

- **Arguments**

  None

- **Return vale**

  Types PS::F32vec or PS::F64vec. A member variable keeping a position of class FP object.

- **Behavior**

  Return a member variable keeping a position of class FP object.

A.1.3.3 FP::copyFromForce

```
class Force {
public:
    PS::F64vec acc;
    PS::F64    pot;
};
class FP {
public:
    PS::F64vec acceleration;
    PS::F64    potential;
    void copyFromForce(const Force & force) {
        this->acceleration = force.acc;
        this->potential    = force.pot;
    }
};
```

- **Presumptions**

  Class Force has results of interaction calculation of a particle.

- **Arguments**

  force: input. Type const Force. The variable force has results of interaction calculation of a particle.

- **Return value**

  なし。

- **Behavior**

  Write results of interaction calculation of a particle to class FP object. The member variables acc and pot in class Force correspond to the member variables of acceleration and potential in class FP, respectively.

- **Remarks**

  The names of "Force" and its member variables are mutable. The names of member variables of class FP are also mutable. The argument name of member function FP::copyFromForce is mutable.

A.1.4 Member functions necessary for some situations

A.1.4.1 Summary

We describe member functions necessary for some situations. These situations are where all the types of PS::SEARCH_MODE but type PS::SEARCH_MODE_LONG are used, where file I/O APIs of class ParticleSystem are used, where ParticleSystem::adjustPositionIntoRootDomain is used, and where class Particle Mesh is used.

A.1.4.2 The cases where all the types of PS::SEARCH_MODE but type PS::SEARCH_MODE_LONG are used

A.1.4.2.1 FP::getRSearch

```
class FP {
public:
    PS::F64 search_radius;
    PS::F64 getRSearch() const {
        return this->search_radius;
    }
};
```

- **Presumptions**

  A member variable search_radius is a radius of a particle which is used to search for neighbor particles of a particle. The data type of the search_radius is types PS::F32 or PS::F64.

- **Arguments**

  None

- **Return value**

  Types PS::F32 or PS::F64. A member variable of class FP object which has a radius of a particle to search for neighbor particles.

- **Behavior**

  Return the member variable of class FP object which has a radius of a particle to search for its neighbor particles.

- **Remarks**

  The name of member variable search_radius is mutable.

A.1.4.3 The case where file I/O APIs of class ParticleSystem is used

Member function readAscii and writeAscii are necessary when ParticleSystem::readParticleAscii and ParticleSystem::writeParticleAscii are used, respectively. In what follows, we describe how to describe the function readAscii and writeAscii.

A.1.4.3.1 FP::readAscii

```
class FP {
public:
    PS::S32 id;
    PS::F64 mass;
    PS::F64vec pos;
    void readAscii(FILE *fp) {
        fscanf(fp, "%d%lf%lf%lf%lf", &this->id, &this->mass,
               &this->pos[0], &this->pos[1], &this->pos[2]);
    }
};
```

- **Presumptions**

  In the input file of particle data, the first, second, third, fourth, and fifth columns contain particle IDs, masses, and three components of positions, respectively. File format is Ascii. 3D Cartesian coordinate system is adopted.

- **Arguments**

  fp: type FILE *. File pointer which points to an input file with particle data.

- **Return value**

  None.

- **Behavior**

  Read information of id, mass, and pos of class FP from an input file.

- **Remarks**

  None.

A.1.4.3.2 FP::writeAscii

```
class FP {
public:
    PS::S32 id;
    PS::F64 mass;
    PS::F64vec pos;
    void writeAscii(FILE *fp) {
        fscanf(fp, "%d %lf %lf %lf %lf", this->id, this->mass,
               this->pos[0], this->pos[1], this->pos[2]);
    }
};
```

- **Presumptions**

  In the output file of particle data, the first, second, third, fourth, and fifth columns contain particle IDs, masses, and three components of positions, respectively. File format is Ascii. 3D Cartesian coordinate system is adopted.

- **Arguments**

  fp: type FILE *. File pointer which points to an output file with particle data.

- **Return value**

  None.

- **Behavior**

  Write information of id, mass, and pos of class FP to an output file.

- **Remarks**

  None.

A.1.4.4 The case where ParticleSystem::adjustPositionIntoRootDomain is used

A.1.4.4.1 FP::setPos

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

- **Presumptions**

  The member variable pos is a position of a particle. Its data type is PS::F32vec or PS::F64vec.

- **Arguments**

  pos_new: input. Type const PS::F32vec or const PS::F64vec. Position data modified by FDPS.

- **Return value**

  None.

- **Behavior**

  Write position data modified by FDPS to class FP object.

- **Remarks**

  The name of member variable pos is mutable. The argument name of pos_new is mutable. If the data types of pos and pos_new are different, it may not work well.

A.1.4.5 The case where class Particle Mesh is used

When class Particle Mesh is used, member functions FP::getChargeParticleMesh and FP::copyFromForceParticleMesh are necessary. In the following, we describe how to describe them.

A.1.4.5.1 FP::getChargeParticleMesh

```
class FP {
public:
    PS::F64 mass;
    PS::F64 getChargeParticleMesh() const {
        return this->mass;
    }
};
```

- **Presumptions**

  The member variable mass has particle mass or charge. Its data type is PS::F32 or PS::F64.

- **Arguments**

  None.

- **Return value**

  Types PS::F32 or PS::F64. Return particle mass or charge.

- **Behavior**

  Return a member variable keeping particle mass or charge.

- **Remarks**

  The variable name of mass is mutable.

A.1.4.5.2 FP::copyFromForceParticleMesh

```
class FP {
public:
    PS::F64vec accelerationFromPM;
    void copyFromForceParticleMesh(const PS::F32vec & acc_pm) {
        this->accelerationFromPM = acc_pm;
    }
};
```

- **Presumptions**

  The member variable accelerationFromPM_pm has force data exerted on a particle. Its data type is PS::F32vec or PS::F64vec.

- **Arguments**

  acc_pm: type const PS::F32vec or const PS::F64vec. Results of class Particle Mesh.

- **Return value**

  None.

- **Behavior**

  Write a force on a particle from class Particle Mesh to the member variable.

- **Remarks**

  The variable name of acc_pm is mutable. The argument name acc_pm is mutable.

A.2 Class EssentialParticleI

A.2.1 Summary

Class EssentialParticleI has i-particle data, and is necessary when users define particle-particle interaction (procedure 0 in section 2.3). The data set of class EssentialParticleI is a subset of the data set of class FullParticle. FDPS accesses the data variables of class EssentialParticleI. In what follows, we describe the presumptions in this section, necessary member functions, and member functions necessary for some situations.

A.2.2 Presumptions

In this section, we use class EPI and FP as examples of classes EssentialParticleI and FullParticle, respectively. The names EPI and FP are mutable.

The classes EPI and FP are declared as follows.

```
class FP;
class EPI;
```

A.2.3 Necessary member functions

A.2.3.1 Summary

The member functions EPI::getPos and EPI::copyfromFP are necessary. FDPS uses the former to read position data in class EPI, and the latter to write class FP data to class EPI. We present how to describe these functions.

A.2.3.2 EPI::getPos

```
class EPI {
public:
    PS::F64vec pos;
    PS::F64vec getPos() const {
        return this->pos;
    }
};
```

- **Presumptions**

  The member variable pos is particle position. Its data type is PS::F64vec.

- **Arguments**

  None.

- **Return value**

  Type PS::F64vec. Member variable keeping position data of class EPI.

- **Behavior**

  Return a memeber variable with position data of class EPI object.

- **Remarks**

  The variable name pos is mutable.

A.2.3.3 EPI::copyFromFP

```
class FP {
public:
    PS::S64    identity;
    PS::F64    mass;
    PS::F64vec position;
    PS::F64vec velocity;
    PS::F64vec acceleration;
    PS::F64    potential;
};
class EPI {
public:
    PS::S64    id;
    PS::F64vec pos;
    void copyFromFP(const FP & fp) {
        this->id  = fp.identity;
        this->pos = fp.position;
    }
};
```

- **Presumptions**

  The member variable identity and position in class FP correspond to the member variables id and pos in class EPI.

- **Arguments**

  fp: input. Type const FP &. Information of class FP object.

- **Return value**

  None.

- **Behavior**

  Write a part of particle data of class FP to class EPI object.

- **Remarks**

  The member variables of classes FP and EPI are mutable. The argument name of EPI::copyFromFP is mutable. The particle data of class EPI is a subset of the particle data of class FP.

A.2.4 Member functions necessary for some situations

A.2.4.1 Summary

In this section, we describe member functions necessary for some situations. The situations are where PS::SEARCH_MODE_GATHER and PS::SEARCH_MODE_SYMMETRY are adopted for type PS::SEARCH_MODE.

A.2.4.2 The case where PS::SEARCH_MODE_GATHER and PS::SEARCH_MODE_SYMMETRY are adopted for type PS::SEARCH_MODE

A.2.4.2.1 EPI::getRSearch

```
class EPI {
public:
    PS::F64 search_radius;
    PS::F64 getRSearch() const {
        return this->search_radius;
    }
};
```

- **Presumptions**

  A member variable search_radius is a radius of a particle which is used to search for neighbor particles of a particle. The data type of the search_radius is types PS::F32 or PS::F64.

- **Arguments**

  None.

- **Return value**

  Types PS::F32 or PS::F64. A member variable of class EPI object which has a radius of a particle to search for neighbor particles.

- **Behavior**

  Return the member variable of class EPI object which has a radius of a particle to search for its neighbor particles.

- **Remarks**

  The name of member variable search_radius is mutable.

A.3 Class EssentialParticleJ

A.3.1 Summary

Class EssentialParticleJ has j-particle data necessary for interaction calculations, and is necessary for the definition of interactions (procedure 0 in section 2.3). The data of class EssentialParticleJ is a subset of the data of class FullParticle (section 7.2). FDPS accesses the data of this class. For this purpose, class EssntialParticleJ must have some member functions. In the following, we describe the presumptions of this section, necessary member functions, and member functions necessary for some situations.

A.3.2 Presumptions

In this section, we use class EPJ and FP as examples of classes EssentialParticleJ and FullParticle, respectively. The class names of EPJ and FP are mutable.

The declarations of EPJ and FP are as follows.

```
class FP;
class EPJ;
```

A.3.3 Necessary member functions

A.3.3.1 Summary

Necessary member functions are EPJ::getPos and EPJ::copyfromFP. Using the former function, FDPS reads the position data of class FP, and write it to class EPJ. Using the latter funciton, FDPS reads the various data of class FP, and write them to class EPJ. We describe these functions in this section.

A.3.3.2 EPJ::getPos

```
class EPJ {
public:
    PS::F64vec pos;
    PS::F64vec getPos() const {
        return this->pos;
    }
};
```

- **Presumptions**

  Member variable pos is the position data of one particle. Its data type is PS::F64vec.

- **Arguments**

  None.

- **Return value**

  Type PS::F64vec. Member variable keeping position data.

- **Behavior**

  Return position data of class EPJ object.

- **Remarks**

  The member variable name of pos is mutable.

A.3.3.3 EPJ::copyFromFP

```
class FP {
public:
    PS::S64    identity;
    PS::F64    mass;
    PS::F64vec position;
    PS::F64vec velocity;
    PS::F64vec acceleration;
    PS::F64    potential;
};
class EPJ {
public:
    PS::S64    id;
    PS::F64    m;
    PS::F64vec pos;
    void copyFromFP(const FP & fp) {
        this->id  = fp.identity;
        this->m   = fp.mass;
        this->pos = fp.position;
    }
};
```

- **Presumptions**

  Member variables of identity, mass, and position in class FP correspond to those of id, m, and pos in class EPJ.

- **Arguments**

  fp: input. Type const FP &. The variable keeping the data of class FP object.

- **Return value**

  None.

- **Behavior**

  Read a part of the data of class FP object, and write them to class EPJ object.

- **Remarks**

  The member variable names of classes FP and EPJ are mutable. The argument name of a member function EPJ::copyFromFP is mutable.

A.3.4 Member functions necessary for some situations

A.3.4.1 Summary

In this section, we describe member functions necessary for some situations. The situations are where users adopt all but PS::SEARCH_MODE_LONG for data type PS::SEARCH_MODE, and where they adopt all but PS::BOUNDARY_CONDITION_OPEN for data type BOUNDARY_CONDITION. We do not describe member functions necessary for situations where pre-existing Moment and SuperParticleJ classes. These member functions are explained in sections 7.5 and 7.6.

A.3.4.2 The cases where all but PS::SEARCH_MODE_LONG are used for data type PS::SEARCH_MODE

A.3.4.2.1 EPJ::getRSearch

```
class EPJ {
public:
    PS::F64 search_radius;
    PS::F64 getRSearch() const {
        return this->search_radius;
    }
};
```

- **Presumptions**

  The member variable search_radius is a radius of a particle. Neighbor particles of the particle are defined as those inside the particle. The data type of search_radius may be PS::F32 or PS::F64.

- **Arguments**

  None.

- **Return value**

  Data types PS::F32 or PS::F64. A radius of a particle.

- **Behavior**

  Return a radius of particle.

- **Remarks**

  The member variable name of search_radius is mutable.

A.3.4.3 The cases where all but PS::BOUNDARY_CONDITION_OPEN are used for data type BOUNDARY_CONDITION

A.3.4.3.1 EPJ::setPos

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

- **Presumptions**

  The member variable pos is a position data of a particle. The data type of pos may be PS::F32vec or PS::F64vec. The original data of the variable pos is a member variable of class FP, position. This data type may be PS::F32vec or PS::F64vec.

- **Arguments**

  pos_new: Input. Type const PS::F32vec or const PS::F64vec. Position data modified by FDPS.

- **Return value**

  None.

- **Behavior**

  FDPS writes modified position data to a position data of class EPJ.

- **Remarks**

  The member variable name of pos is mutable. The argument name of pos_new is mutable.

A.4 Class Moment

A.4.1 Summary

Class Moment has moment data of several particles close to each other, and is necessary for the definition of interaction (procedure 0 in section 2.3). Examples of moment data are monopole and dipole of several particles, and the maximum size of these particles. This class plays a role like intermediate variables between classes EssentialParticleJ and SuperParticleJ. Necessary member functions of this class are, for example, a member function which reads some data from class EssnetialParticleJ, and calculate moment of particles.

Some classes are pre-existing. We first describe these classes. Next, we describe how to make class Moment, and what member functions are needed.

A.4.2 Pre-existing classes

A.4.2.1 Summary

There are several pre-existing classes. These classes are available when some PS::SEARCH_MODE types are adopted. In the following, we describe class Moment available when PS::SEARCH_MODE_LONG and PS::SEARCH_MODE_LONG_CUTOFF are adopted. When the other PS::SEARCH_MODE are adopted, users need not to program, considering class Moment.

A.4.2.2 PS::SEARCH_MODE_LONG

In the `Moment` classes described below, for `MomentMonopole` and `MomentQuadrupole`, their equivalents are defined for `PS::SEARCH_MODE_LONG_SCATTER` which can be used for the neighbor search. They are `MomentMonopoleScatter` and `MomentQuadrupoleScatter`, respectively.

A.4.2.2.1 PS::MomentMonopole

This class has moment up to monopole. When this moment is calculated, the center of the coordinate is taken to the center of mass or charge. In the following, we describe an overview of this class.

```
namespace ParticleSimulator {
    class MomentMonopole {
    public:
        F32    mass;
        F32vec pos;
    };
}
```

- Class name PS::MomentMonopole
- Member variables and their information

  mass: the total mass or charge of particles close to each other.

  pos: the center of mass or charge of particles close to each other.
- Use conditions

  Class EssentialParticleJ (section 7.4) has member functions EssentialParticleJ::getCharge and EssentialParticleJ::getPos.

  In the case of `MomentMonopoleScatter`, class `EssentialParticleJ` has the member function `getRSearch()`

A.4.2.2.2 PS::MomentQuadrupole

This class has moment up to quadrupole. When this moment is calculated, the center of the coordinate is taken to the center of mass or charge. In the following, we describe an overview of this class.

```
namespace ParticleSimulator {
    class MomentQuadrupole {
    public:
        F32    mass;
        F32vec pos;
        F32mat quad;
    };
}
```

- Class name PS::MomentQuadrupole
- Member variables and their information

  mass: the total mass of particles close to each other.

  pos: the center of mass of particles close to each other.

  quad: the quadrupole of mass of particles close to each other.
- Use conditions

  Class EssentialParticleJ (section 7.4) has member functions EssentialParticleJ::getCharge and EssentialParticleJ::getPos.

  In the case of `MomentQuadlepoleScatter`, class `EssentialParticleJ` has the member function `getRSearch()`

A.4.2.2.3 PS::MomentMonopoleGeometricCenter

This class has moment up to monopole. When this moment is calculated, the center of the coordinate is taken to the geometric center of particles. In the following, we describe an overview of this class.

```
namespace ParticleSimulator {
    class MomentMonopoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
    };
}
```

- Class name PS::MomentMonopoleGeometricCenter

- Member variables and their information

  charge: the total mass or charge of particles close to each other.

  pos: the geometric center of particles close to each other.

- Use conditions

  Class EssentialParticleJ (section 7.4) has member functions EssentialParticleJ::getCharge and EssentialParticleJ::getPos.

A.4.2.2.4 PS::MomentDipoleGeometricCenter

This class has moment up to dipole. When this moment is calculated, the center of the coordinate is taken to the geometric center of particles. In the following, we describe an overview of this class.

```
namespace ParticleSimulator {
    class MomentDipoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
        F32vec dipole;
    };
}
```

- Class name

  PS::MomentDipoleGeometricCenter

- Member variables and their information

  charge: the total mass or charge of particles close to each other.

  pos: the geometric center of particles close to each other.

  dipole: dipole of particle masses or charges.

- Use conditions

  Class EssentialParticleJ (section 7.4) has member functions EssentialParticleJ::getCharge and EssentialParticleJ::getPos.

A.4.2.2.5 PS::MomentQuadrupoleGeometricCenter

This class has moment up to quadrupole. When this moment is calculated, the center of the coordinate is taken to the geometric center of particles. In the following, we describe an overview of this class.

```
namespace ParticleSimulator {
    class MomentQuadrupoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
        F32vec dipole;
        F32mat quadrupole;
    };
}
```

- Class name

  PS::MomentQuadrupoleGeometricCenter

- Member variables and their information

  charge: the total mass or charge of particles close to each other.

  pos: the geometric center of particles close to each other.

  dipole: dipole of particle masses or charges.

  quadrupole: quadrupole of particle masses or charges.

- Use conditions

  Class EssentialParticleJ (section 7.4) has member functions EssentialParticleJ::getCharge and EssentialParticleJ::getPos.

A.4.2.3 PS::SEARCH_MODE_LONG_CUTOFF

In the `Moment` classes described below, for `MomentMonopoleCutoff`, its equivalent is defined for `PS::SEARCH_MODE_LONG_CUTOFF_CATTER` which can be used for the neighbor search. It is `MomentMonopoleCutoffScatter`.

A.4.2.3.1 PS::MomentMonopoleCutoff

This class has moment up to monopole. When this moment is calculated, the center of the coordinate is taken to the center of mass or charge. In the following, we describe an overview of this class.

```
namespace ParticleSimulator {
    class MomentMonopoleCutoff {
    public:
        F32    mass;
        F32vec pos;
    };
}
```

- Class name

  PS::MomentMonopoleCutoff

- Member variables and their information

  mass: the total mass or charge of particles close to each other.

  pos: the center of mass or charge of particles close to each other.

- Use conditions

  Class EssentialParticleJ (section 7.4) has member functions EssentialParticleJ::getCharge and EssentialParticleJ::getPos.

  In the case of `MomentMonopoleCutoffScatter`, class `EssentialParticleJ` has the member function `getRSearch()`

A.4.3 Necessary member functions

A.4.3.1 Summary

In the following, we describe necessary member functions when users define class Moment. As an example, we use Mom for class Moment name. This name is mutable.

A.4.3.2 Constructor

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    Mom () {
        mass = 0.0;
        pos  = 0.0;
    }
};
```

- **Presumptions**

  Member variables mass and pos are mass and position information of class Mom, respectively.

- **Arguments**

  None.

- **Return value**

  None.

- **Behavior**

  Initialize class Mom object.

- **Remarks**

  The member variable names are mutable. Some member variables can be added.

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    Mom(const PS::F32 m,
        const PS::F32vec & p) {
        mass = m;
        pos  = p;
    }
};
```

- **Presumptions**

  Member variables mass and pos are mass and position information of class Mom, respectively.

- **Arguments**

  m: input. Type const PS::F32. Mass.

  p: Input. Type const PS::F32vec &. Position.

- **Return value**

  None.

- **Behavior**

  Initialize class Mom object.

- **Remarks**

  The member variable names are mutable. Some member variables can be added.

A.4.3.3 Mom::init

```
class Mom {
public:
    void init();
};
```

- **Presumptions**

  None.

- **Arguments**

  None.

- **Return value**

  None.

- **Behavior**

  Initialize class Mom object.

- **Remarks**

  None.

A.4.3.4 Mom::getPos

```
class Mom {
public:
    PS::F32vec pos;
    PS::F32vec getPos() const {
        return pos;
    }
};
```

- **Presumptions**

  A member variable pos is a representative position of particles close to each other. This data type is PS::F32vec or PS::F64vec.

- **Arguments**

  None.

- **Return value**

  PS::F32vec or PS::F64vec. The member variable pos.

- **Behavior**

  Return the member variable pos.

- **Remarks**

  The member variable name is mutable.

A.4.3.5 Mom::getCharge

```
class Mom {
public:
    PS::F32 mass;
    PS::F32 getCharge() const {
        return mass;
    }
};
```

- **Presumptions**

  A member variable mass is the total mass or charge of particles close to each other. This data type is PS::F32 or PS::F64.

- **Arguments**

  None.

- **Return value**

  Type PS::F32 or PS::F64. The member variable mass.

- **Behavior**

  Return the member variable mass.

- **Remarks**

  The member variable name is mutable.

A.4.3.6 Mom::accumulateAtLeaf

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    template <class Tepj>
    void accumulateAtLeaf(const Tepj & epj) {
        mass += epj.getCharge();
        pos  += epj.getPos();
    }
};
```

- **Presumptions**

  A member variable mass is the total mass or charge of particles close to each other. This data type is PS::F32 or PS::F64. A member variable pos is a representative position of particles close to each other. This data type is PS::F32vec or PS::F64vec. Class EssentialParticleJ is inserted into the template argument Tepj, and has member functions getCharge and getPos.

- **Arguments**

  epj: input. Type const Tepj &. An object of class Tpj.

- **Return value**

  None.

- **Behavior**

  Calculate moment from an object of class EssentialParticleJ.

- **Remarks**

  The member variable names mass and pos are mutable. The argument name epj is mutable.

A.4.3.7 Mom::accumulate

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    void accumulate(const Mom & mom) {
        mass += mom.mass;
        pos  += mom.mass * mom.pos;
    }
};
```

- **Presumptions**

  A member variable mass is the total mass or charge of particles close to each other. This data type is PS::F32 or PS::F64. A member variable pos is the center of mass of particles close to each other. This data type is PS::F32vec or PS::F64vec. Class EssentialParticleJ is inserted into the template argument Tepj, and has member functions getCharge and getPos.

- **Arguments**

  mom: input. Type const Mom &. An object of class Mom.

- **Return value**

  None.

- **Behavior**

  Calculate information of class Mom from an object of class Mom.

- **Remarks**

  The member variable names mass and pos are mutable. The argument name epj is mutable.

A.4.3.8 Mom::set

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    void set() {
        pos = pos / mass;
    }
};
```

- **Presumptions**

  A member variable mass is the total mass or charge of particles close to each other. This data type is PS::F32 or PS::F64. A member variable pos is the center of mass of particles close to each other. This data type is PS::F32vec or PS::F64vec. Class EssentialParticleJ is inserted into the template argument Tepj, and has member functions getCharge and getPos.

- **Arguments**

  None.

- **Return value**

  None.

- **Behavior**

  Normalize position data of moment, since they are not normalized in the member functions in Mom::accumulateAtLeaf and Mom::accumulate.

- **Remarks**

  The member variable names mass and pos are mutable. The argument name epj is mutable.

A.4.3.9 Mom::accumulateAtLeaf2

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    PS::F32mat quad;
    template <class Tepj>
    void accumulateAtLeaf2(const Tepj & epj) {
        PS::F64 ctmp    = epj.getCharge();
        PS::F64vec ptmp = epj.getPos() - pos;
        PS::F64 cx = ctmp * ptmp.x;
        PS::F64 cy = ctmp * ptmp.y;
        PS::F64 cz = ctmp * ptmp.z;
        quad.xx += cx * ptmp.x;
        quad.yy += cy * ptmp.y;
        quad.zz += cz * ptmp.z;
        quad.xy += cx * ptmp.y;
        quad.xz += cx * ptmp.z;
        quad.yz += cy * ptmp.z;
    }
};
```

- **Presumptions**

  A member variable mass is the total mass or charge of particles close to each other. This data type is PS::F32 or PS::F64. A member variable pos is a representative position of particles close to each other. This data type is PS::F32vec or PS::F64vec. This has been already obtained in the member function Mom::accumulateAtLeaf. A member variable quad is quadrupole of particles close to each other. This data type is PS::F32mat or PS::F64mat. Class EssentialParticleJ is inserted into the template argument Tepj, and has member functions getCharge and getPos.

- **Arguments**

  epj: input. Type const Tepj &. An object of class Tepj.

- **Return value**

  None.

- **Behavior**

  Calculate moment from an object of class EssentialParticleJ.

- **Remarks**

  The member variable names mass, pos and quad are mutable. The argument name epj is mutable.

A.4.3.10 Mom::accumulate2

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    PS::F32mat quad;
    void accumulate(const Mom & mom) {
        PS::F64 mtmp    = mom.mass;
        PS::F64vec ptmp = mom.pos - pos;
        PS::F64 cx = mtmp * ptmp.x;
        PS::F64 cy = mtmp * ptmp.y;
        PS::F64 cz = mtmp * ptmp.z;
        quad.xx += cx * ptmp.x + mom.quad.xx;
        quad.yy += cy * ptmp.y + mom.quad.yy;
        quad.zz += cz * ptmp.z + mom.quad.zz;
        quad.xy += cx * ptmp.y + mom.quad.xy;
        quad.xz += cx * ptmp.z + mom.quad.xz;
        quad.yz += cy * ptmp.z + mom.quad.yz;
    }
};
```

- **Presumptions**

  A member variable mass is the total mass or charge of particles close to each other. This data type is PS::F32 or PS::F64. A member variable pos is a representative position of particles close to each other. This data type is PS::F32vec or PS::F64vec. This has been already obtained in the member function Mom::accumulate. A member variable quad is quadrupole of particles close to each other. This data type is PS::F32mat or PS::F64mat.

- **Arguments**

  mom: input. Type const Mom &. An object of class Mom.

- **Return value**

  None.

- **Behavior**

  Calculate information of class Mom from an object of class Mom

- **Remarks**

  The member variable names mass, pos and quad are mutable.

A.5 Class SuperParticleJ

A.5.1 Summary

Class SuperPrticleJ has information of superparticle which is a representative of particles close to each other. This is required for the definition of interaction (procedure 0 in section 2.3). This class is necessary only when PS::SEARCH_MODE_LONG and PS::SEARCH_MODE_LONG_CUTOFF are adopted for type PS::SEARCH_MODE. This class has member functions communicating position data with FDPS. Information of classes SuperParticleJ and Moment pairs up. Therefore, this class has member functions which transform from data of class Moment to that of class SuperParticleJ, and vice versa.

Some classes are pre-existing. We first describe these classes. Next, we describe how to make class SuperParticleJ, and what member functions are needed.

A.5.2 Pre-existing classes

There are several pre-existing classes. These classes are available when some PS::SEARCH_MODE types are adopted. In the following, we describe class SuperParticleJ available when PS::SEARCH_MODE_LONG and PS::SEARCH_MODE_LONG_CUTOFF are adopted. When the other PS::SEARCH_MODE are adopted, users need not to program, considering class SuperParticleJ.

A.5.2.1 PS::SEARCH_MODE_LONG

A.5.2.1.1 PS::SPJMonopole

This class has moment up to monopole, and is paired with class PS::MomentMonopole. We overview this class as follows.

```
namespace ParticleSimulator {
    class SPJMonopole {
    public:
        F64    mass;
        F64vec pos;
    };
}
```

- Class name PS::SPJMonopole
- Member variables and their information

  mass: the total mass or charge of particles close to each other.

  pos: the center of mass or charge of particles close to each other.
- Use conditions

  The same as class PS::MomentMonopole.

A.5.2.1.2 PS::SPJQuadrupole

This class has monopole and quadrupole moments, and is paired with class PS::MomentQuadrupole. In the following, we describe an overview of this class.

```
namespace ParticleSimulator {
    class SPJQuadrupole {
    public:
        F32    mass;
        F32vec pos;
        F32mat quad;
    };
}
```

- Class name PS::SPJQuadrupole
- Member variables and their information

  mass: the total mass of particles close to each other.

  pos: the center of mass of particles close to each other.

  quad: the quadrupole of mass of particles close to each other.
- Use conditions

  The same as class PS::MomentQuadrupole.

A.5.2.1.3 PS::SPJMonopoleGeometricCenter

This class has moment up to monopole, and is paired with class PS::MomentMonpoleGeometricCenter. In the following, we describe an overview of this class.

```
namespace ParticleSimulator {
    class SPJMonopoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
    };
}
```

- Class name

  PS::SPJMonopoleGeometricCenter
- Member variables and their information

  charge: the total mass or charge of particles close to each other.

  pos: the geometric center of particles close to each other.

- Use conditions

  The same as class PS::MomentMonopoleGeometricCenter.

A.5.2.1.4 PS::SPJDipoleGeometricCenter

This class has moment up to dipole, and is paired with class PS::MomentDipoleGeometricCenter. In the following, we describe an overview of this class.

```
namespace ParticleSimulator {
    class SPJDipoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
        F32vec dipole;
    };
}
```

- Class name

  PS::SPJDipoleGeometricCenter

- Member variables and their information

  charge: the total mass or charge of particles close to each other.

  pos: the geometric center of particles close to each other.

  dipole: dipole of particle masses or charges.

- Use conditions

  The same as class PS::MomentDipoleGeometricCenter.

A.5.2.1.5 PS::SPJQuadrupoleGeometricCenter

This class has moment up to quadrupole, and is paired with class PS::MomentQuadrupoleGeometricCenter. In the following, we describe an overview of this class.

```
namespace ParticleSimulator {
    class SPJQuadrupoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
        F32vec dipole;
        F32mat quadrupole;
    };
}
```

- Class name

  PS::SPJQuadrupoleGeometricCenter

- Member variables and their information

  charge: the total mass or charge of particles close to each other.

  pos: the geometric center of particles close to each other.

  dipole: dipole of particle masses or charges.

  quadrupole: quadrupole of particle masses or charges.

- Use conditions

  The same as class PS::MomentQuadrupoleGeometricCenter.

A.5.2.2 PS::SEARCH_MODE_LONG_CUTOFF

A.5.2.2.1 PS::SPJMonopoleCutoff

This class has moment up to monopole, and is paired with class PS::MomentMonopoleCutoff. In the following, we describe an overview of this class.

```
namespace ParticleSimulator {
    class SPJMonopoleCutoff {
    public:
        F32    mass;
        F32vec pos;
    };
}
```

- Class name

  PS::SPJMonopoleCutoff

- Member variables and their information

  mass: the total mass or charge of particles close to each other.

  pos: the center of mass or charge of particles close to each other.

- Use conditions

  The same as class PS::MomentMonopoleCutoff.

A.5.3 Necessary member functions

A.5.3.1 Summary

We describe necessary member functions when users make class SuperParticleJ. We use class SPJ for an example of class SuperParticleJ. This name is mutable.

A.5.3.2 SPJ::getPos

```
class SPJ {
public:
    PS::F64vec pos;
    PS::F64vec getPos() const {
        return this->pos;
    }
};
```

- **Presumptions**

  Member variables pos is position information of this class. This data type is PS::F32vec or PS::F64vec.

- **Arguments**

  None.

- **Return value**

  Type PS::F32vec or PS::F64vec. Member variable keeping position data of class SPJ.

- **Behavior**

  Return position data of an object of class SPJ.

- **Remarks**

  The member variable name is mutable.

A.5.3.3 SPJ::setPos

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

- **Presumptions**

  A member variable pos is position data of a particle. This data type is PS::F32vec or PS::F64vec.

- **Arguments**

  pos_new: input. Type const PS::F32vec or const PS::F64vec. Position modified by FDPS.

- **Return value**

  None.

- **Behavior**

  Write position data modified by FDPS to position data of an object of class SPJ.

- **Remarks**

  The member variable name is mutable. The argument name is mutable.

A.5.3.4 SPJ::copyFromMoment

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
}
class SPJ {
public:
    PS::F32    mass;
    PS::F32vec pos;
    void copyFromMoment(const Mom & mom) {
        mass = mom.mass;
        pos  = mom.pos;
    }
};
```

- **Presumptions**

  None.

- **Arguments**

  mom: input. Type const Mom &. User-defined class Moment.

- **Return value**

  None.

- **Behavior**

  Copy an object of class Mom to that of class SPJ.

- **Remarks**

  Class name Mom is mutable. Member variables of classes Mom and SPJ are mutable. The argument name of the function SPJ::copyFromMoment is mutable.

A.5.3.5 SPJ::convertToMoment

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    Mom(const PS::F32 m,
        const PS::F32vec & p) {
        mass = m;
        pos  = p;
    }
}
class SPJ {
public:
    PS::F32    mass;
    PS::F32vec pos;
    Mom convertToMoment() const {
        return Mom(mass, pos);
    }
};
```

- **Presumptions**

  None.

- **Arguments**

  None.

- **Return value**

  Type Mom. Constructor of class Mom.

- **Behavior**

  Return the constructor of class Mom.

- **Remarks**

  Class name Mom is mutable. Member variable names of classes Mom and SPJ are mutable. The argument name of the function SPJ::copyFromMoment is mutable. Predefinition of the constructor of class Mom is necessary.

A.5.3.6 SPJ::clear

```
class SPJ {
public:
    PS::F32    mass;
    PS::F32vec pos;
    void clear() {
        mass = 0.0;
        pos  = 0.0;
    }
};
```

- **Presumptions**

  None.

- **Arguments**

  None.

- **Return value**

  None.

- **Behavior**

  Clear information of an object of class SPJ.

- **Remarks**

  Member variable names are mutable.

A.6 Class Force

A.6.1 Summary

This class has results of interaction calculations, and is necessary for the definition of interaction (procedure 0 in section 2.3). We describe presumptions in this section, and necessary member functions.

A.6.2 Presumptions

We use class Result as an example of class Force. This name is mutable.

A.6.3 Necessary member functions

A necessary member function is Result::clear. This function initializes results of interaction calculations.

A.6.3.1 Result::clear

```
class Result {
public:
    PS::F32vec acc;
    PS::F32    pot;
    void clear() {
        acc = 0.0;
        pot = 0.0;
    }
};
```

- **Presumptions**

  Member variables are acc and pot.

- **Arguments**

  None.

- **Return value**

  None.

- **Behavior**

  Initialize member variables.

- **Remarks**

  The member variable names are mutable.

A.7 Class Header

A.7.1 Summary

Class Header defines a format of a header of I/O files. This class is used in file I/O APIs of class ParticleSystem. These APIs are ParticleSystem::readParticleAscii and ParticleSystem::writeParticleAscii. We present presumptions in this section and member functions necessary when users use these APIs.

A.7.2 Presumptions

We use class Hdr as an example of class Header. This name is mutable.

A.7.3 Member functions necessary for some situations

A.7.3.1 Hdr::readAscii

```
class Hdr {
public:
    PS::S32 nparticle;
    PS::F64 time;
    PS::S32 readAscii(FILE *fp) {
        fscanf(fp, "%d%lf", &this->nparticle, &this->time);
        return this->nparticle;
    }
};
```

- **Presumptions**

  This header has the number of particles “nparticle” and time “time”.

- **Arguments**

  fp: input. Type FILE *. File pointer to an input file.

- **Return value**

  Type PS::S32. Return the number of particles, and return -1 unless there is the number of particles in the header.

- **Behavior**

  Read the header information from the input file of particle data.

- **Remarks**

  The member variable names are mutable. It is undefined when the return value is neither the number of particles and -1.

A.7.3.2 Hdr::writeAscii

```
class Hdr {
public:
    PS::S32 nparticle;
    PS::F64 time;
    void writeAscii(FILE *fp) {
        fprintf(fp, "%d %lf", this->nparticle, this->time);
    }
};
```

- **Presumptions**

  This header has the number of particles “nparticle” and time “time”.

- **Arguments**

  fp: input. Type FILE *. File pointer to an output file.

- **Return value**

  None.

- **Behavior**

  Write the header information to the output file.

- **Remarks**

  The member variable names are mutable.

A.8 Function object calcForceEpEp

A.8.1 Summary

Function object calcForceEpEp presents an interaction, and is necessary for the definition of the interaction (procedure 0 in section 2.3). In what follows, we describe how to describe this.

A.8.2 Presumptions

We show the function object for calculating gravitational N-body simulation. The object name is gravityEpEp, and is mutable. We use classes EPI and EPJ for classes EssentialParitlceI and EssentialParitlceJ, respectively.

A.8.3 gravityEpEp::operator ()

Listing 43: calcForceEpEp

```
class Result {
public:
    PS::F32vec acc;
};
class EPI {
public:
    PS::S32    id;
    PS::F32vec pos;
};
class EPJ {
public:
    PS::S32    id;
    PS::F32    mass;
    PS::F32vec pos;
};
struct gravityEpEp {
    static PS::F32 eps2;
```

```
    void operator () (const EPI *epi,
                      const PS::S32 ni,
                      const EPJ *epj,
                      const PS::S32 nj,
                      Result *result) {

        for(PS::S32 i = 0; i < ni; i++) {
            PS::S32    ii = epi[i].id;
            PS::F32vec xi = epi[i].pos;
            PS::F32vec ai = 0.0;
            for(PS::S32 j = 0; j < nj; j++) {
                PS::S32    jj = epj[j].id;
                PS::F32    mj = epj[j].mass;
                PS::F32vec xj = epj[j].pos;

                PS::F32vec dx   = xi - xj;
                PS::F32    r2   = dx * dx + eps2;
                PS::F32    rinv = (ii != jj) ? 1. / sqrt(r2)
                                             : 0.0;

                ai += mj * rinv * rinv * rinv * dx;
            }
            result.acc = ai;
        }
    }
};
PS::F32 gravityEpEp::eps2 = 9.765625e-4;
```

- **Presumptions**

  Member functions necessary for classes Result, EPI, and EPJ are omitted. A member variable acc in class Result is gravitational acceleration from j-particle to i-particle. Member variables id and pos in classes EPI and EPJ are particle ID and position, respectively. A member variable mass in class EPJ is mass of j-particle. A member variable eps2 in function object gravityEpEp is square of gravitational softening. Users need not to consider multi-threading, since multi-threading is defined outside this function object.

- **Arguments**

  epi: input. Type const EPI * or EPI *. Array of i-particle data.

  ni: input. Type const PS::S32 or PS::S32. The number of i-particles.

  epj: input. Type const EPJ * or EPJ *. Array of j-particle data.

  nj: input. Type const PS::S32 or PS::S32. The number of j-particles.

  result: output. Type Result *. Array of results of interactions of i-particles.

- **Return value**

  None.

- **Behavior**

  Calculate forces exerted by j-particles on i-particles.

- **Remarks**

  All the argument names are mutable. The contents of the function object are mutable.

A.9 Function object calcForceSpEp

A.9.1 Summary

Function object calcForceSpEp presents forces exerted by superparticles on particles, and is necessary for the definition of the interaction (procedure 0 in section 2.3). In what follows, we describe how to describe this.

A.9.2 Presumptions

We show the function object for calculating gravitational N-body simulation. The object name is gravitySpEp, and is mutable. We use classes EPI and SPJ for classes EssentialParitlceI and SuperParitlceJ, respectively.

A.9.3 gravitySpEp::operator ()

Listing 44: calcForceSpEp

```
class Result {
public:
    PS::F32vec accfromspj;
};
class EPI {
public:
    PS::S32    id;
    PS::F32vec pos;
};
class SPJ {
public:
    PS::F32    mass;
    PS::F32vec pos;
};
struct gravitySpEp {
    static PS::F32 eps2;
    void operator () (const EPI *epi,
                      const PS::S32 ni,
                      const SPJ *spj,
                      const PS::S32 nj,
```

```
                              Result *result) {

        for(PS::S32 i = 0; i < ni; i++) {
            PS::F32vec xi = epi[i].pos;
            PS::F32vec ai = 0.0;
            for(PS::S32 j = 0; j < nj; j++) {
                PS::F32    mj = spj[j].mass;
                PS::F32vec xj = spj[j].pos;

                PS::F32vec dx   = xi - xj;
                PS::F32    r2   = dx * dx + eps2;
                PS::F32    rinv = 1. / sqrt(r2);

                ai += mj * rinv * rinv * rinv * dx;
            }
            result.accfromspj = ai;
        }
    }
};
PS::F32 gravitySpEp::eps2 = 9.765625e-4;
```

- **Presumptions**

  Member functions necessary for classes Result, EPI, and SPJ are omitted. A member variable accfromspj in class Result is gravitational acceleration from superparticle to i-particle. Member variables pos in classes EPI and SPJ are particle position. A member variable mass in class SPJ is mass of superparticle. A member variable eps2 in function object gravitySpEp is square of gravitational softening. Users need not to consider multi-threading, since multi-threading is defined outside this function object.

- **Arguments**

  epi: input. Type const EPI * or EPI *. Array of i-particle data.

  ni: input. Type const PS::S32 or PS::S32. The number of i-particles.

  spj: input. Type const SPJ * or SPJ *. Array of superparticle data.

  nj: input. Type const PS::S32 or PS::S32. The number of superparticles.

  result: output. Type Result *. Array of results of interactions of i-particles.

- **Return value**

  None.

- **Behavior**

  Calculate forces exerted by superparticles on i-particles.

- **Remarks**

  All the argument names are mutable. The contents of the function object are mutable.

A.10 Functor calcForceDispatch

A.10.1 Summary

Functor `calcForceDispatch` is used in the case when some accelerator hardware is used for the interaction calculation. It sends the particles to the acclerator and startup the interaction calculation kernel on the accelator. In the following, we present an example implementattion of this function.

A.10.2 Premises

Here we present one example of using Cuda to implement gravitational interaction between particles. The name of functor `calcForceDispatch` is `CalcForceDispatch`. The names for classes `EssentialParitlceI`, `SuperParitlceJ`, and `Force` are `EPI`, `SPJ`, and `Result`, respectively. These name can be changed to any names legal within C++ grammer.

A.10.3 The example

Listing 45: calcForceDispatch

```

class EpiGPU{
public:
    float2 pos[3];
    int id_walk;
};

class EpjGPU{
public:
    float mass;
    float2 pos[3];
};

class ForceGPU{
public:
    float2 acc[3];
    float2 pot;
};

__global__ void ForceKernel(const EpiGPU * epi,
                            const EpjGPU * epj,
                            const int    * nj_disp,
                            ForceGPU     * force,
                            const float eps2){
    int id_i = blockDim.x * blockIdx.x + threadIdx.x;
    const EpiGPU & ip = epi[id_i];
    float2 poti;
    float2 acci[3];
```

```
    poti = acci[0] = acci[1] = acci[2] = make_float2(0.0, 0.0);
    const int j_head = nj_disp[ip.id_walk];
    const int j_tail = nj_disp[ip.id_walk+1];
    const int nj = j_tail - j_head;
    for(int j=j_head; j<j_tail; j++){
        EpjGPU jp = epj[j];
        const float dx = (jp.pos[0].x - ip.pos[0].x) + (jp.pos
              [0].y - ip.pos[0].y);
        const float dy = (jp.pos[1].x - ip.pos[1].x) + (jp.pos
              [1].y - ip.pos[1].y);
        const float dz = (jp.pos[2].x - ip.pos[2].x) + (jp.pos
              [2].y - ip.pos[2].y);
        const float r2 = ((eps2 + dx*dx) + dy*dy) + dz*dz;
        const float r_inv = rsqrtf(r2);
        const float pij = jp.mass * r_inv * (r2 > eps2);
        const float r2_inv = r_inv * r_inv;
        const float pij_r3_inv = pij * r2_inv;
        const float ax = pij_r3_inv * dx;
        const float ay = pij_r3_inv * dy;
        const float az = pij_r3_inv * dz;
        poti = float2_accum(poti, pij);
        acci[0] = float2_accum(acci[0], ax);
        acci[1] = float2_accum(acci[1], ay);
        acci[2] = float2_accum(acci[2], az);
    }
    poti = float2_regularize(poti);
    acci[0] = float2_regularize(acci[0]);
    acci[1] = float2_regularize(acci[1]);
    acci[2] = float2_regularize(acci[2]);
    force[id_i].pot = poti;
    force[id_i].acc[0] = acci[0];
    force[id_i].acc[1] = acci[1];
    force[id_i].acc[2] = acci[2];
}

static ForceGPU * force_d;
static ForceGPU * force_h;
static EpiGPU * epi_d;
static EpiGPU * epi_h;
static EpjGPU * epj_d;
static EpjGPU * epj_h;
static int * ni_disp_h;
static int * nj_disp_d;
static int * nj_disp_h;

int DispatchKernelWithSP(const PS::S32 tag,
```

```
                             const int    n_walk,
                             const EPIGrav ** epi,
                             const int  *  n_epi,
                             const EPJGrav ** epj,
                             const int  *  n_epj,
                             const PS::SPJMonopole ** spj,
                             const int  *  n_spj){
    static bool first = true;
    assert(n_walk <= N_WALK_LIMIT);
    if(first){
        CUDA_SAFE_CALL( cudaMalloc(     (void**)&nj_disp_d,  (N_WALK_LIMIT+1)*sizeof(int) ) );
        CUDA_SAFE_CALL( cudaMallocHost( (void**)&ni_disp_h,  (N_WALK_LIMIT+1)*sizeof(int) ) );
        CUDA_SAFE_CALL( cudaMallocHost( (void**)&nj_disp_h,  (N_WALK_LIMIT+1)*sizeof(int) ) );
        CUDA_SAFE_CALL( cudaMalloc( (void**)&epi_d, NI_LIMIT*sizeof(EpiGPU) ) );
        CUDA_SAFE_CALL( cudaMalloc( (void**)&epj_d, NJ_LIMIT*sizeof(EpjGPU) ) );
        CUDA_SAFE_CALL( cudaMalloc( (void**)&force_d, NI_LIMIT*sizeof(ForceGPU) ) );
        CUDA_SAFE_CALL( cudaMallocHost( (void**)&epi_h, NI_LIMIT*sizeof(EpiGPU) ) );
        CUDA_SAFE_CALL( cudaMallocHost( (void**)&epj_h, NJ_LIMIT*sizeof(EpjGPU) ) );
        CUDA_SAFE_CALL( cudaMallocHost( (void**)&force_h, NI_LIMIT*sizeof(ForceGPU) ) );
        first = false;
    }
    const float eps2 = EPIGrav::eps * EPIGrav::eps;
    ni_disp_h[0] = nj_disp_h[0] = 0;
    for(int i=0; i<n_walk; i++){
        ni_disp_h[i+1] = ni_disp_h[i] + n_epi[i];
        nj_disp_h[i+1] = nj_disp_h[i] + n_epj[i] + n_spj[i];
    }
    int ni_tot = ni_disp_h[n_walk];
    const int ni_tot_reg = ni_disp_h[n_walk] + ( (ni_tot%N_THREAD_GPU != 0) ? (N_THREAD_GPU - (ni_tot%N_THREAD_GPU)) : 0);
    assert(ni_tot_reg <= NI_LIMIT);
    assert(nj_disp_h[n_walk] <= NJ_LIMIT);
    ni_tot = 0;
    int nj_tot = 0;
    for(int iw=0; iw<n_walk; iw++){
        for(int ip=0; ip<n_epi[iw]; ip++){
```

```
            epi_h[ni_tot].pos[0]  = float2_split(epi[iw][ip].pos.x);
            epi_h[ni_tot].pos[1]  = float2_split(epi[iw][ip].pos.y);
            epi_h[ni_tot].pos[2]  = float2_split(epi[iw][ip].pos.z);
            epi_h[ni_tot].id_walk = iw;
            force_h[ni_tot].acc[0] = force_h[ni_tot].acc[1]
                = force_h[ni_tot].acc[2] = force_h[ni_tot].pot = make_float2(0.0, 0.0);
            ni_tot++;
        }
        for(int jp=0; jp<n_epj[iw]; jp++){
            epj_h[nj_tot].mass    = epj[iw][jp].mass;
            epj_h[nj_tot].pos[0]  = float2_split(epj[iw][jp].pos.x);
            epj_h[nj_tot].pos[1]  = float2_split(epj[iw][jp].pos.y);
            epj_h[nj_tot].pos[2]  = float2_split(epj[iw][jp].pos.z);
            nj_tot++;
        }
        for(int jp=0; jp<n_spj[iw]; jp++){
            epj_h[nj_tot].mass    = spj[iw][jp].getCharge();
            epj_h[nj_tot].pos[0]  = float2_split(spj[iw][jp].getPos().x);
            epj_h[nj_tot].pos[1]  = float2_split(spj[iw][jp].getPos().y);
            epj_h[nj_tot].pos[2]  = float2_split(spj[iw][jp].getPos().z);
            nj_tot++;
        }
    }
    for(int ip=ni_tot; ip<ni_tot_reg; ip++){
        epi_h[ni_tot].pos[0]  = epi_h[ni_tot].pos[1]  = epi_h[ni_tot].pos[2]  = make_float2(0.0, 0.0);
        epi_h[ni_tot].id_walk = 0;
        force_h[ni_tot].acc[0] = force_h[ni_tot].acc[1]
            = force_h[ni_tot].acc[2] = force_h[ni_tot].pot = make_float2(0.0, 0.0);
    }
    CUDA_SAFE_CALL( cudaMemcpy(epi_d, epi_h, ni_tot_reg*sizeof(EpiGPU), cudaMemcpyHostToDevice) );
    CUDA_SAFE_CALL( cudaMemcpy(epj_d, epj_h, nj_tot*sizeof(EpjGPU), cudaMemcpyHostToDevice) );
```

```
    CUDA_SAFE_CALL( cudaMemcpy(nj_disp_d, nj_disp_h, (n_walk+1)*sizeof(int), cudaMemcpyHostToDevice) );
    const int n_grid = ni_tot_reg/N_THREAD_GPU + ((ni_tot_reg%N_THREAD_GPU == 0) ? 0 : 1);
    dim3 size_grid(n_grid, 1, 1);
    dim3 size_thread(N_THREAD_GPU, 1, 1);
    ForceKernel<<<size_grid, size_thread>>> (epi_d, epj_d, nj_disp_d, force_d, float(eps2));

    return 0;
}
```

- **Arguments**

  `tag`: input. Type `const PS::S32`. Corresponding call to `CalcForceRetrieve()` should use the same value for `tag`

  `nwalk`: input.Type `const PS::S32`. The number of interaction lists.

  `epi`: input. Type `const EPI**` or `EPI**`. Array of the array of i-particles.

  `ni`: input. Type `const PS::S32*` or `PS::S32*`. Array of the numbers of i-particles.

  `spj`: input. Type `const EPJ**` or `EPJ**`. Array of the array of j-particles.

  `nj`: input. Type `const PS::S32*` or `PS::S32*`. Array of the number of j-paricles.

- **return value** Returns 0 upon normal completion. Otherwize non-zero values are returned.

- **Function**

  Send `epi` and `epj` to the accelerator and let the accelerator do the interaction calculation.

- **Remarks**

  All the argument names are mutable. The contents of the function object are mutable.

A.11 Functor calcForceRetrieve

A.11.1 Summary

Functor `calcForceRetrieve` retrieves the results calculated on the accelerator. In the following, we present an example implementattion of this function.

A.11.2 Premises

Here we present one example of using Cuda to implement gravitational interaction between particles. The name of functor `calcForceRetrieve` is `RetieveKernel`. The name for the class `Force` is `ForceGrav`. These name can be changed to any names legal within C++ grammer.

Listing 46: calcForceRetrieve

```
int RetrieveKernel(const PS::S32 tag,
                   const PS::S32    n_walk,
                   const PS::S32 *  ni,
                   ForceGrav     ** force){
    int ni_tot = 0;
    for(int i=0; i<n_walk; i++){
        ni_tot += ni[i];
    }
    CUDA_SAFE_CALL( cudaMemcpy(force_h, force_d,      ni_tot*sizeof(ForceGPU), cudaMemcpyDeviceToHost) );
    int n_cnt = 0;
    for(int iw=0; iw<n_walk; iw++){
        for(int ip=0; ip<ni[iw]; ip++){
            force[iw][ip].acc.x = (double)force_h[n_cnt].acc[0].x + (double)force_h[n_cnt].acc[0].y;
            force[iw][ip].acc.y = (double)force_h[n_cnt].acc[1].x + (double)force_h[n_cnt].acc[1].y;
            force[iw][ip].acc.z = (double)force_h[n_cnt].acc[2].x + (double)force_h[n_cnt].acc[2].y;
            force[iw][ip].pot   = (double)force_h[n_cnt].pot.x + (double)force_h[n_cnt].pot.y;
            force[iw][ip].pot *= -1.0;
            n_cnt++;
        }
    }
    return 0;
}
```

- **Arguments**

  `tag`: input. Type `const PS::S32`. Corresponding call to `CalcForceDispatch()` should use the same value for `tag`

  `nwalk`: input.Type `const PS::S32`. The number of interaction lists.

  `ni`: input. Type `const PS::S32*` or `PS::S32*`. Array of the numbers of i-particles.

  `force`: output. Type `Result**`.

- **return value** Returns 0 upon normal completion. Otherwize non-zero values are returned.

- **Function**

  Store the results calculated on the accielerator to the array `force`.

- **Remarks**

  All the argument names are mutable. The contents of the function object are mutable.