FDPS仕様書

谷川衝, 岩澤全規, 細野七月, 似鳥啓吾, 村主崇行, 行方大輔, 野村昴太郎, and 牧野淳一郎

理化学研究所 計算科学研究センター 粒子系シミュレータ研究チーム

目 次

1　この文書の概要

この文書は大規模並列粒子シミュレーションの開発を支援する Framework for Developing Particle Simulator (FDPS) の仕様書である。この文書は理化学研究所計算科学研究機構粒子系シミュレータ研究チームの谷川衝、岩澤全規、細野七月、似鳥啓吾、村主崇行、牧野淳一郎によって記述された。

この文書は以下のような構成となっている。

節 2、3、4 には、FDPS を使ってプログラムを書く際に前提となる情報が記述されている。節 2 には、FDPS の概要として、FDPS の基本的な考えかたや動作が記述されている。節 3 には、FDPS のファイル構成が記述されている。節 4 には、FDPS の API を使用したコードをコンパイルする時にどのようなマクロを用いればよいかが記述されている。

節 5、6、7、8、9 には、FDPS を使ってプログラムを書く際に必要となる情報が提供されている。節 5 には、FDPS 内での名前空間の構造についてが記述されている。節 6 には、FDPS で独自に定義されているデータ型が記述されている。節 7 には、FDPS の API を使用する際にユーザーが定義する必要があるクラスや関数オブジェクトについて記述されている。節 8 には、FDPS を開始するときと終了するときに呼ぶ必要のある API について記述されている。節 9 には、FDPS にあるモジュールとその API について記述されている。

節 10、11、12、13 には、FDPS の API を使用したコードを記述したがコードが思ったように動作しない場合に有用な情報が記載されている。節 10 にはエラーメッセージについてが記述されている。節 11 には、よく知られているバグについて記述されている。節 12 には、FDPS の限界について記述されている。節 13 には、ユーザーサポートに関する情報が記述されている。

最後に節 14 には FDPS のライセンスに関する情報が、節 15 にはこの文書の変更履歴が記述されている。

2 FDPS概要

この節ではFDPSの概要を記述する。FDPSの開発目的、FDPSの基本的な考えかた、FDPSを使用して作成したコードの動作について概説する。

2.1 開発目的

粒子シミュレーションは、重力N体シミュレーション、SPHシミュレーション、渦糸法、MPS法、分子動力学シミュレーションなど理学工学の様々な分野で使用されている。より大きい空間スケール、より高い空間分解能(または質量分解能)、より長い時間スケールの物理現象を追跡するために、高性能な粒子シミュレーションコードへの要請はますます強くなっている。

高性能な粒子シミュレーションコードを組むためには、シミュレーションコードの大規模並列化を避けることはできない。粒子シミュレーションコードの大規模並列化をする際には、ロードバランスのため動的領域分割、領域分割に合わせた粒子交換、ノード間通信の削減と最適化、キャッシュ利用効率の向上、SIMDユニット利用効率の向上、アクセラレータへの対応など、数多くの困難な処理を行う必要がある。現在、研究グループは個別にこれらの処理へ対応している。

しかし、上記の処理は粒子シミュレーション共通のものである。FDPSの開発目的は、これらの処理を高速に行うライブラリを提供し、大規模並列化への対応に追われていた研究者の負担を軽くすることである。FDPSを使うことで、研究者がよりクリエイティブな仕事に専念できるようになれば、幸いである。

2.2 基本的な考えかた

ここではFDPSの基本的な考えかたについて記述する。

2.2.1 大規模並列粒子シミュレーションの手順

まずFDPSにおいて、大規模並列粒子シミュレーションがどのような手順で行われることを想定しているかを記述する。粒子シミュレーションは、以下のような微分方程式を時間発展させるものである。

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

ここで$\boldsymbol{u}_i$は粒子iの物理量ベクトルであり、この物理量には質量、位置、速度など粒子が持つあらゆる物理量が含まれる。関数fは粒子jから粒子iへの作用を規定する。以後、作用を受ける粒子をi粒子、作用を与える粒子をj粒子と呼ぶことにする。$\boldsymbol{v}_s$はi粒子から十分遠方にある粒子を1つの粒子としてまとめた粒子(以後、この粒子を超粒子と呼ぶ)の物理量ベクトルである。関数gは超粒子からi粒子への作用を規定する。式(1)の第2項は、重力

やクーロン力など無限遠まで到達する長距離力の場合はゼロではない。しかし流体の圧力のような短距離力はゼロである。

大規模並列化された粒子シミュレーションコードは以下の手順で式 (1) を時間発展させる。ここではデータの入出力や初期化は省略している。

1. 以下の 2 段階の手順でどのプロセスがどの粒子の式 (1) を時間発展させるか決める。
   (a) プロセスの間でロードバランスを取れるように、シミュレーションで扱っている空間の領域を分割し、各プロセスの担当領域を決める (領域分割)。
   (b) 各プロセスが、自分の担当する領域に存在する全粒子の物理量ベクトル $\boldsymbol{u}_i$ を持つように、他のプロセスと物理量ベクトル $\boldsymbol{u}_i$ を交換する (粒子交換)。
2. 各プロセスは、自分の担当する全粒子の式 (1) の右辺を計算するのに必要な j 粒子の物理量ベクトル $\boldsymbol{u}_j$ と超粒子の物理量ベクトル $\boldsymbol{v}_s$ を他のプロセスと通信することで集めて、j 粒子のリストと超粒子のリスト (まとめて相互作用リストと呼ぶ) を作る (相互作用リストの作成)。
3. 各プロセスは自分の担当する全粒子に対して、式 (1) の右辺を計算し、$d\boldsymbol{u}_i/dt$ を求める (相互作用の計算)。
4. 各プロセスは、自分の担当する全粒子の物理量ベクトル $\boldsymbol{u}_i$ とその時間導関数 $d\boldsymbol{u}_i/dt$ を使って、全粒子の時間積分を実行し、次の時刻の物理量ベクトル $\boldsymbol{u}_i$ を求める (時間積分)。
5. 手順 1 に戻る。

2.2.2 ユーザーと FDPS の役割分担

FDPS は、プロセス間の通信が発生する処理は FDPS が担当し、プロセス間の通信の発生しない処理はユーザーが担当するという役割分担を基本としている。従って、前節に挙げた、領域分割・粒子交換 (項目 1)・相互作用リストの作成 (項目 2) を FDPS が、相互作用の計算 (項目 3)・時間積分 (項目 4) をユーザーが担当することになる。ユーザーは FDPS の API を呼び出すだけで、大規模並列化に関わる煩雑な処理を避けつつ、高性能な任意の相互作用の粒子シミュレーションコードを手に入れることができる。

2.2.3 ユーザーのやること

ユーザーが FDPS を使って粒子シミュレーションコードを作成するときにやることは以下の項目である。

- 粒子の定義 (節 7)。粒子の持つ物理量 (式 (1) で言えば $\boldsymbol{u}_i$) の指定。例えば質量、位置、速度、加速度、元素組成、粒子サイズ、など。

- 相互作用の定義 (節 7)。粒子間の相互作用 (式 (1) で言えばで関数 f, g) を指定。例えば、重力、クーロン力、圧力、など。
- FDPS の API の呼出 (節 8, 9)

2.2.4 補足

式 (1) の右辺は 2 粒子間相互作用の重ね合わせである。従って、FDPS の API を呼ぶだけでは、3 つ以上の粒子の間の相互作用の計算を行うことはできない。しかし、FDPS はネイバーリストを返す API を用意している。ネイバーリストを用いれば、ユーザーはプロセス間の通信の処理をすることなく、このような相互作用の計算をできる。

節 2.2.1 で示した手順は、全粒子が同じ時間刻みを持っている。そのため、FDPS の API を呼び出すだけでは、独立時間刻みで時間積分を効率的に行うことができない。しかし、上と同じくネイバーリストを返す API があるため、Particle Particle Particle Tree 法を用いて独立時間刻みを実装することは可能であろう。

2.3 コードの動作

ここでは FDPS を使用して作成したコードの動作の概略を記述する。このコードには、4 つのモジュールがあることになる。3 つは FDPS のモジュールで、1 つはユーザー定義のモジュールである。まとめると以下のようになる。

- 領域クラス：全プロセスが担当する領域の情報と、領域分割を行う API を持つ
- 粒子群クラス：全粒子の情報と、プロセスの間での粒子交換を行う API を持つ
- 相互作用ツリークラス：粒子分布から作られたツリー構造と、相互作用リストを作成する API を持つ
- ユーザー定義クラス：ある 1 粒子を定義するクラス、粒子間の相互作用を定義する関数オブジェクトを持つ

これら 4 つのモジュールの間で情報がやり取りされる。これは図 1 で概観できる。図 1 に示された情報のやりとりは、節 2.2.1 に記述された手順 1 から 3 と、これらの手順以前に行われる手順 (手順 0 とする) に対応する。以下はこれらの手順の詳細な記述である。

0. ユーザー定義クラスのうち 1 粒子を定義するクラスが粒子群クラスへ、粒子間の相互作用を定義する関数オブジェクトが相互作用ツリークラスへ渡される。これはクラスの継承ではなく、粒子を定義するクラスは粒子群クラスのテンプレート引数として、粒子間の相互作用を定義する関数オブジェクトは相互作用ツリークラスの API の引数として渡される
1. 以下の 2 段階でロードバランスを取る

(a) 領域クラスが持つ領域分割の API が呼ばれる。このとき粒子情報が粒子群クラスから領域クラスへ渡される (赤字と赤矢印)

(b) 粒子群クラスが持つ粒子交換の API が呼ばれる。このとき領域情報が領域クラスから粒子群クラスへ渡される (青字と青矢印)

2. 相互作用ツリークラスが持つ相互作用リストを作成する API が呼ばれる。このとき領域情報が領域クラスから相互作用ツリークラスへ、粒子情報が粒子群クラスから相互作用ツリークラスへ渡される (緑字と緑矢印)

3. 相互作用ツリーククラスが持つ相互作用を定義した関数オブジェクトを呼び出す API が呼ばれる。相互作用計算が実行され、相互作用計算の結果が相互作用ツリークラスから粒子群クラスへ渡される (灰色の字と灰色矢印)

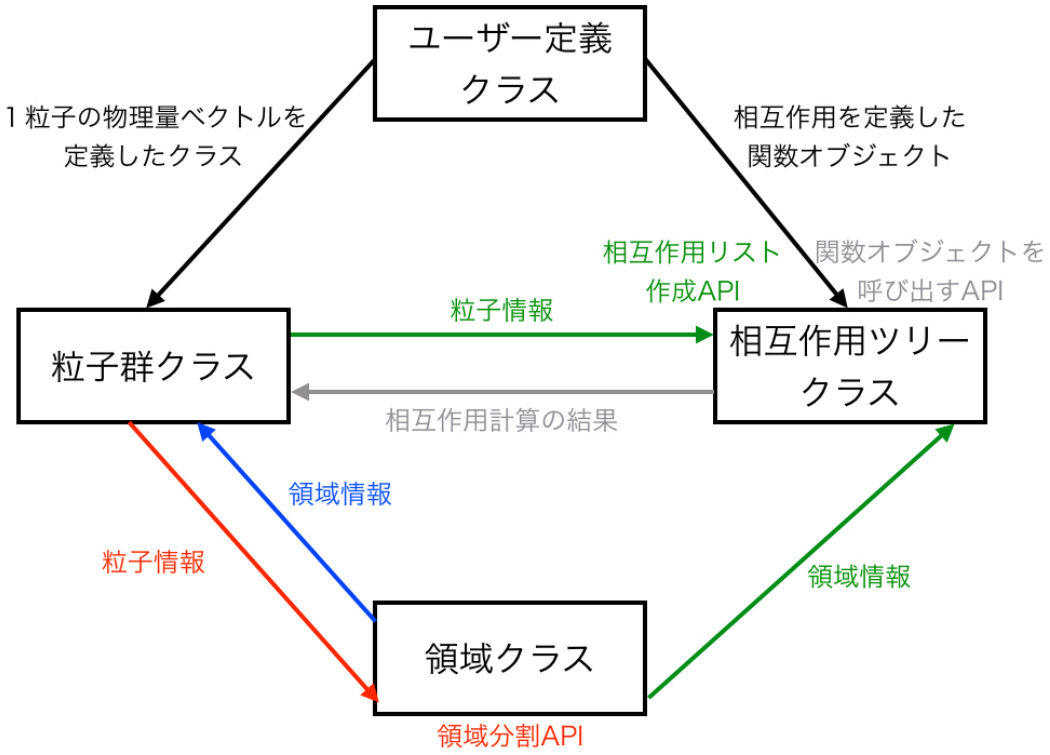

図 1: モジュールインターフェースと情報の流れの模式図。

3　ファイル構成

3.1　概要

ここではFDPSのファイル構成について記述する。ドキュメント、ソースファイル、テストコード、サンプルコードの順に記述する。

3.2　ドキュメント

ドキュメント関係のファイルはディレクトリdocの下にある。チュートリアルがdoc_tutorial.pdfであり、仕様書がdoc_specs.pdfである。

3.3　ソースファイル

ソースファイルはディレクトリsrcの下にある。標準機能関係のソースファイルはsrcの直下にある。ディレクトリsrcの直下にあるヘッダファイルparticle_simulator.hppをソースファイルにインクルードすれば、FDPSの標準機能を使用できるようになる。

3.3.1　拡張機能

拡張機能関係のソースファイルはディレクトリsrcの直下のディレクトリにそれぞれ入っている。拡張機能にはParticle Mesh、x86版Phantom-GRAPEがある。

3.3.1.1　Particle Mesh

Particle Meshのソースファイルはディレクトリsrc/particle_meshの下にある。ここでMakefileを編集して、makeを実行すると、ヘッダファイルparticle_mesh_class.hppとライブラリlibpm.aができる。このヘッダファイルをインクルードし、このライブラリをリンクづけすれば、Particle Meshの機能を使用できるようになる。

3.3.1.2　x86版Phantom-GRAPE

x86版Phantom-GRAPEのソースファイルはディレクトリsrc/phantom_GRAPE_x86の下にある。この下には低精度N体シミュレーション用、低精度カットオフ付き相互作用計算用、高精度N体シミュレーション用(ディレクトリG6/libavx)がある。それぞれについて述べる。

3.3.1.2.1 低精度 N 体シミュレーション用

これはディレクトリ src/phantom_GRAPE_x86/G5/newton/libpg5 にある。このディレクトリ内の Makefile を編集して、make を実行すると、ライブラリ libpg5.a ができる。このディレクトリ内のヘッダファイル gp5util.h をインクルードし、ライブラリ libpg5.a をリンクすると、この Phantom-GRAPE が使用可能になる。

3.3.1.2.2 低精度カットオフ付き相互作用計算用

これはディレクトリ src/phantom_GRAPE_x86/G5/table/にある。このディレクトリ内の Makefile を編集して、make を実行すると、ライブラリ libpg5.a ができる。このディレクトリ内のヘッダファイル gp5util.h をインクルードし、ライブラリ libpg5.a をリンクすると、この Phantom-GRAPE が使用可能になる。

3.3.1.2.3 高精度 N 体シミュレーション用

これはディレクトリ src/phantom_GRAPE_x86/G6/libavx/にある。このディレクトリ内の Makefile を編集して、make を実行すると、ライブラリ libg6avx.a ができる。このディレクトリ内のヘッダファイル gp6util.h をインクルードし、ライブラリ libg6avx.a をリンクすると、この Phantom-GRAPE が使用可能になる。

3.4 テストコード

テストコードはディレクトリ tests の下にある。ディレクトリ tests にカレントディレクトリを移し、make check を実行するとテストスィートが動作する。

3.5 サンプルコード

サンプルコードはディレクトリ sample の下にある。サンプルコードは 2 つ用意されており、重力 N 体シミュレーションと SPH シミュレーションである。

3.5.1 重力 N 体シミュレーション

ディレクトリ sample/nbody の下にソースファイルがある。サンプルコードの実行方法はチュートリアルを参照のこと。

3.5.2 SPH シミュレーション

ディレクトリ sample/sph の下にソースファイルがある。サンプルコードの実行方法はチュートリアルを参照のこと。

4　コンパイル時のマクロによる選択

4.1　概要

FDPS では、座標系や並列処理の有無、浮動小数点数型の精度、エラー検出等を選択できる。この選択はコンパイル時のマクロの定義によってなされる。以下、選択の方法について座標系、並列処理の有無、浮動小数点型の精度の順に記述する。

4.2　座標系

4.2.1　概要

座標系は直角座標系 3 次元と直角座標系 2 次元の選択ができる。以下、それらの選択方法について述べる。

4.2.2　直角座標系 3 次元

デフォルトは直角座標系 3 次元である。なにも行わなくても直角座標系 3 次元となる。

4.2.3　直角座標系 2 次元

コンパイル時に PARTICLE_SIMULATOR_TWO_DIMENSION をマクロ定義すると直交座標系 2 次元となる。

4.3　並列処理

4.3.1　概要

並列処理に関しては、OpenMP の使用／不使用、MPI の使用／不使用を選択できる。以下、選択の仕方について記述する。

4.3.2　OpenMP の使用

デフォルトは OpenMP 不使用である。使用する場合は、PARTICLE_SIMULATOR_THREAD_PARALLEL をマクロ定義すればよい。GCC コンパイラの場合はコンパイラオプションに-fopenmp をつける必要がある。

4.3.3　MPI の使用

デフォルトは MPI 不使用である。使用する場合は、PARTICLE_SIMULATOR_THREAD_PARALLEL をマクロ定義すればよい。

4.4 データ型の精度

4.4.1 概要

FDPS 側で用意した Moment クラス (第 7.5.2 節参照) と SuperParticleJ クラス (第 7.6.2 節参照) のデータ型の精度を選択できる。以下、選択の仕方について記述する。

4.4.2 既存の SuperParticleJ クラスと Moment クラスの精度

既存の SuperParticleJ クラスと Moment クラスのメンバ変数の精度はデフォルトで 64 ビットである。32 ビットにしたい場合、
PARTICLE_SIMULATOR_SPMOM_F32
をマクロ定義すればよい。

5 名前空間

5.1 概要

本節では、名前空間の構造について述べる。FDPS は ParticleSimulator という名前空間で囲まれている。以下では、ParticleSimulator 直下にある機能と、ParticleSimulator にネストされている名前空間について述べる。

5.2 ParticleSimulator

FDPS の標準機能すべては名前空間 ParticleSimulator の直下にある。

名前空間 ParticleSimulator は以下のように省略されており、この文書におけるあとの記述でもこの省略形を採用する。

```
namespace PS = ParticleSimulator;
```

名前空間 ParticleSimulator の下にはいくつかの名前空間が拡張機能毎にネストされている。拡張機能には ParticleMesh がある。以下では拡張機能の名前空間について記述する。

5.2.1 ParticleMesh

Particle Mesh の機能は名前空間 PartcielMesh に囲まれており、名前空間 ParticleMesh は名前空間 ParticleSimulator の直下にネストされている。また、ParticleMesh は PM と省略されている。これらをまとめると以下のようになっている。

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

以後、この文書では省略形の PM を用いて記述する。

6 データ型

6.1 概要

FDPS では独自の整数型、実数型、ベクトル型、対称行列型、PS::SEARCH_MODE 型、列挙型が定義されている。整数型、実数型、ベクトル型、対称行列型に関しては必ずしもここに挙げるものを用いる必要はないが、これらを用いることを推奨する。PS::SEARCH_MODE 型、列挙型は必ず用いる必要がある。以下、整数型、実数型、ベクトル型、対称行列型、PS::SEARCH_MODE 型、列挙型の順に記述する。

6.2 整数型

6.2.1 概要

整数型には PS::S32, PS::S64, PS::U32, PS::U64 がある。以下、順にこれらを記述する。

6.2.2 PS::S32

PS::S32 は以下のように定義されている。すなわち 32bit の符号付き整数である。

ソースコード 1: S32

```
1 namespace ParticleSimulator {
2     typedef int S32;
3 }
```

6.2.3 PS::S64

PS::S64 は以下のように定義されている。すなわち 64bit の符号付き整数である。

ソースコード 2: S64

```
1 namespace ParticleSimulator {
2     typedef long long int S64;
3 }
```

6.2.4 PS::U32

PS::U32 は以下のように定義されている。すなわち 32bit の符号なし整数である。

ソースコード 3: U32

```
1 namespace ParticleSimulator {
2     typedef unsigned int U32;
3 }
```

6.2.5 PS::U64

PS::U64 は以下のように定義されている。すなわち 64bit の符号なし整数である。

ソースコード 4: U64

```
1 namespace ParticleSimulator {
2     typedef unsigned long long int U64;
3 }
```

6.2.6 PS::Count_t

カウント数を表す為の型。現在は PS::U64 型として定義されている。

ソースコード 5: Count_t

```
1 namespace ParticleSimulator {
2     typedef U64 Count_t;
3 }
```

6.3 実数型

6.3.1 概要

実数型には PS::F32, PS::F64 がある。以下、順にこれらを記述する。

6.3.2 PS::F32

PS::F32 は以下のように定義されている。すなわち 32bit の浮動小数点数である。

ソースコード 6: F32

```
1 namespace ParticleSimulator {
2     typedef float F32;
3 }
```

6.3.3 PS::F64

PS::F64 は以下のように定義されている。すなわち 64bit の浮動小数点数である。

ソースコード 7: F64

```
1 namespace ParticleSimulator {
2     typedef double F64;
3 }
```

6.4 ベクトル型

6.4.1 概要

ベクトル型には2次元ベクトル型PS::Vector2と3次元ベクトル型PS::Vector3がある。まずこれら2つを記述する。最後にこれらベクトル型のラッパーについて記述する。

6.4.2 PS::Vector2

PS::Vector2はx, yの2要素を持つ。これらに対する様々なAPIや演算子を定義した。それらの宣言を以下に記述する。この節ではこれらについて詳しく記述する。

ソースコード 8: Vector2

```
1  namespace ParticleSimulator{
2      template <typename T>
3      class Vector2{
4      public:
5          //メンバ変数2要 素
6          T x, y;
7
8          //コンストラクタ
9          Vector2();
10         Vector2(const T _x, const T _y) : x(_x), y(_y) {}
11         Vector2(const T s) : x(s), y(s) {}
12         Vector2(const Vector2 & src) : x(src.x), y(src.y) {}
13
14         //代入演算子
15         const Vector2 & operator = (const Vector2 & rhs);
16
17         //[]演 算 子
18         const T & opertor[](const int i);
19         T & operator[](const int i);
20
21         //加減算
22         Vector2 operator + (const Vector2 & rhs) const;
23         const Vector2 & operator += (const Vector2 & rhs);
24         Vector2 operator - (const Vector2 & rhs) const;
25         const Vector2 & operator -= (const Vector2 & rhs);
26
27         //ベクトルスカラ積
28         Vector2 operator * (const T s) const;
29         const Vector2 & operator *= (const T s);
```

```
30          friend Vector2 operator * (const T s,
31                                     const Vector2 & v);
32          Vector2 operator / (const T s) const;
33          const Vector2 & operator /= (const T s);
34
35          //内積
36          T operator * (const Vector2 & rhs) const;
37
38          //外積(返り値はスカラ!!)
39          T operator ^ (const Vector2 & rhs) const;
40
41          //Vector2<U>への型変換
42          template <typename U>
43          operator Vector2<U> () const;
44      };
45  }
46  namespace PS = ParticleSimulator;
```

6.4.2.1 コンストラクタ

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

- **引数**

  なし。

- **機能**

  デフォルトコンストラクタ。メンバ x,y は 0 で初期化される。

```
template<typename T>
PS::Vector2<T>::Vector2(const T _x, const T _y)
```

- **引数**

  _x: 入力。const T 型。

  _y: 入力。const T 型。

- **機能**

  メンバ x、y をそれぞれ_x、_y で初期化する。

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

- **引数**

  s: 入力。const T 型。

- **機能**

  メンバ x、y を両方とも s の値で初期化する。

6.4.2.2 コピーコンストラクタ

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

- **引数**

  src: 入力。const PS::Vector2<T> &型。

- **機能**

  コピーコンストラクタ。src で初期化する。

6.4.2.3 メンバ変数

```
template<typename T>
T PS::Vector2<T>::x;

template<typename T>
T PS::Vector2<T>::y;
```

- **機能**

  メンバ x、y を直接操作出来る。

6.4.2.4 代入演算子

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

- **引数**

  rhs: 入力。const PS::Vector2<T> &型。

- **返り値**

  const PS::Vector2<T> &型。rhs の x,y の値を自身のメンバ x,y に代入し自身の参照を返す。代入演算子。

6.4.2.5 [] 演算子

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

- **引数**

  i: 入力。const int 型。

- **返り値**

  const <T> &型。ベクトルの i 成分を返す。

- **備考**

  直接メンバ変数を指定する場合に比べ、処理が遅くなることがある。

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

- **引数**

  i: 入力。const int 型。

- **返り値**

  <T> &型。ベクトルの i 成分を返す。

- **備考**

  直接メンバ変数を指定する場合に比べ、処理が遅くなることがある。

6.4.2.6 加減算

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

- **引数**

  rhs: 入力。const PS::Vector2<T> &型。

- **返り値**

  PS::Vector2<T> 型。rhs の x,y の値と自身のメンバ x,y の値の和を取った値を返す。

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

- **引数**

  rhs: 入力。const PS::Vector2<T> &型。

- **返り値**

  const PS::Vector2<T> &型。rhs の x,y の値を自身のメンバ x,y に足し、自身を返す。

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

- **引数**

  rhs: 入力。const PS::Vector2<T> &型。

- **返り値**

  PS::Vector2<T> 型。rhs の x,y の値と自身のメンバ x,y の値の差を取った値を返す。

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

- **引数**

  rhs: 入力。const PS::Vector2<T> &型。

- **返り値**

  const PS::Vector2<T> &型。自身のメンバ x,y から rhs の x,y を引き自身を返す。

6.4.2.7　ベクトルスカラ積

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

- **引数**

  s: 入力。const T 型。

- **返り値**

  PS::Vector2<T> 型。自身のメンバ x,y それぞれに s をかけた値を返す。

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

- **引数**

  rhs: 入力。const T 型。

- **返り値**

  const PS::Vector2<T> &型。自身のメンバ x,y それぞれに s をかけ自身を返す。

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

- **引数**

  s: 入力。const T 型。

- **返り値**

  PS::Vector2<T> 型。自身のメンバ x,y それぞれを s で割った値を返す。

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

- **引数**

  rhs: 入力。const T 型。

- **返り値**

  const PS::Vector2<T> &型。自身のメンバ x,y それぞれを s で割り自身を返す。

6.4.2.8 内積、外積

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

- **引数**

  rhs: 入力。const PS::Vector2<T> &型。

- **返り値**

  T 型。自身と rhs の内積を取った値を返す。

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

- **引数**

  rhs: 入力。const PS::Vector2<T> &型。

- **返り値**

  T 型。自身と rhs の外積を取った値を返す。

6.4.2.9 Vector2<U> への型変換

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

- **引数**

  なし。

- **返り値**

  const PS::Vector2<U> 型。

- **機能**

  const PS::Vector2<T> 型を const PS::Vector2<U> 型にキャストする。

6.4.3 PS::Vector3

PS::Vecotr3 は x, y, z の 3 要素を持つ。これらに対する様々な API や演算子を定義した。それらの宣言を以下に記述する。この節ではこれらについて詳しく記述する。

ソースコード 9: Vector3

```
1  namespace ParticleSimulator{
2      template <typename T>
3      class Vector3{
4      public:
5          //メンバ変数は以下の二つのみ。
6          T x, y, z;
7
8          //コンストラクタ
9          Vector3() : x(T(0)), y(T(0)), z(T(0)) {}
10         Vector3(const T _x, const T _y, const T _z) : x(_x), y(
               _y), z(_z) {}
11         Vector3(const T s) : x(s), y(s), z(s) {}
12         Vector3(const Vector3 & src) : x(src.x), y(src.y), z(
               src.z) {}
13
14         //代入演算子
15         const Vector3 & operator = (const Vector3 & rhs);
16
17         //[]演 算 子
18         const T & opertor[](const int i);
19         T & operator[](const int i);
20
21         //加減算
22         Vector3 operator + (const Vector3 & rhs) const;
23         const Vector3 & operator += (const Vector3 & rhs);
24         Vector3 operator - (const Vector3 & rhs) const;
25         const Vector3 & operator -= (const Vector3 & rhs);
26
27         //ベクトルスカラ積
28         Vector3 operator * (const T s) const;
29         const Vector3 & operator *= (const T s);
30         friend Vector3 operator * (const T s, const Vector3 & v
               );
31         Vector3 operator / (const T s) const;
32         const Vector3 & operator /= (const T s);
```

```
33
34          //内積
35          T operator * (const Vector3 & rhs) const;
36
37          //外積(返り値はスカラ!!)
38          T operator ^ (const Vector3 & rhs) const;
39
40          //Vector3<U>への型変換
41          template <typename U>
42          operator Vector3<U> () const;
43      };
44  }
```

6.4.3.1 コンストラクタ

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

- **引数**

  なし。

- **機能**

  デフォルトコンストラクタ。メンバ x,y は 0 で初期化される。

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

- **引数**

  _x: 入力。const T 型。

  _y: 入力。const T 型。

- **機能**

  メンバ x、y をそれぞれ_x、_y で初期化する。

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

- **引数**

  s: 入力。const T 型。

- **機能**

  メンバ x、y を両方とも s の値で初期化する。

6.4.3.2 コピーコンストラクタ

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

- **引数**

  src: 入力。const PS::Vector3<T> &型。

- **機能**

  コピーコンストラクタ。src で初期化する。

6.4.3.3 メンバ変数

```
template<typename T>
T PS::Vector3<T>::x;

template<typename T>
T PS::Vector3<T>::y;

template<typename T>
T PS::Vector3<T>::z;
```

- **機能**

  メンバ x、y、z を直接操作出来る。

6.4.3.4 代入演算子

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

- **引数**

  rhs: 入力。const PS::Vector3<T> &型。

- **返り値**

  const PS::Vector3<T> &型。rhs の x,y の値を自身のメンバ x,y に代入し自身の参照を返す。代入演算子。

6.4.3.5 [] 演算子

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

- **引数**

  i: 入力。const int 型。

- **返り値**

  const <T> &型。ベクトルの i 成分を返す。

- **備考**

  直接メンバ変数を指定する場合に比べ、処理が遅くなることがある。

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

- **引数**

  i: 入力。const int 型。

- **返り値**

  <T> &型。ベクトルの i 成分を返す。

- **備考**

  直接メンバ変数を指定する場合に比べ、処理が遅くなることがある。

6.4.3.6 加減算

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

- **引数**

  rhs: 入力。const PS::Vector3<T> &型。

- **返り値**

  PS::Vector3<T> 型。rhs の x,y の値と自身のメンバ x,y の値の和を取った値を返す。

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

- **引数**

  rhs: 入力。const PS::Vector3<T> &型。

- **返り値**

  const PS::Vector3<T> &型。rhs の x,y の値を自身のメンバ x,y に足し、自身を返す。

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

- **引数**

  rhs: 入力。const PS::Vector3<T> &型。

- **返り値**

  PS::Vector3<T> 型。rhs の x,y の値と自身のメンバ x,y の値の差を取った値を返す。

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

- **引数**

  rhs: 入力。const PS::Vector3<T> &型。

- **返り値**

  const PS::Vector3<T> &型。自身のメンバ x,y から rhs の x,y を引き自身を返す。

6.4.3.7 ベクトルスカラ積

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

- **引数**

  s: 入力。const T 型。

- **返り値**

  PS::Vector3<T> 型。自身のメンバ x,y それぞれに s をかけた値を返す。

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

- **引数**

  rhs: 入力。const T 型。

- **返り値**

  const PS::Vector3<T> &型。自身のメンバ x,y それぞれに s をかけ自身を返す。

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

- **引数**

  s: 入力。const T 型。

- **返り値**

  PS::Vector3<T> 型。自身のメンバ x,y それぞれを s で割った値を返す。

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

- **引数**

  rhs: 入力。const T 型。

- **返り値**

  const PS::Vector3<T> &型。自身のメンバ x,y それぞれを s で割り自身を返す。

6.4.3.8 内積、外積

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

- **引数**

  rhs: 入力。const PS::Vector3<T> &型。

- **返り値**

  T 型。自身と rhs の内積を取った値を返す。

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

- **引数**

  rhs: 入力。const PS::Vector3<T> &型。

- **返り値**

  T 型。自身と rhs の外積を取った値を返す。

6.4.3.9 Vector3<U> への型変換

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

- **引数**

  なし

- **返り値**

  const PS::Vector3<U> 型。

- **機能**

  const PS::Vector3<T> 型を const PS::Vector3<U> 型にキャストする。

6.4.4 ベクトル型のラッパー

ベクトル型のラッパーの定義を以下に示す。

ソースコード 10: vectorwrapper

```
1  namespace ParticleSimulator{
2      typedef Vector2<F32> F32vec2;
3      typedef Vector3<F32> F32vec3;
4      typedef Vector2<F64> F64vec2;
5      typedef Vector3<F64> F64vec3;
6  #ifdef PARTICLE_SIMULATOR_TWO_DIMENSION
7      typedef F32vec2 F32vec;
8      typedef F64vec2 F64vec;
9  #else
10     typedef F32vec3 F32vec;
11     typedef F64vec3 F64vec;
12 #endif
13 }
```

すなわち PS::F32vec2, PS::F32vec3, PS::F64vec2, PS::F64vec3 はそれぞれ単精度 2 次元ベクトル、倍精度 2 次元ベクトル、単精度 3 次元ベクトル、倍精度 3 次元ベクトルである。FDPS で扱う空間座標系を 2 次元とした場合、PS::F32vec と PS::F64vec はそれぞれ単精度 2 次元ベクトル、倍精度 2 次元ベクトルとなる。一方、FDPS で扱う空間座標系を 3 次元とした場合、PS::F32vec と PS::F64vec はそれぞれ単精度 3 次元ベクトル、倍精度 3 次元ベクトルとなる。

6.5 オルソトープ型

6.5.1 概要

オルソトープ型 (長方形 [空間 2 次元] や直方体 [空間 3 次元] を表すデータ型) には 2 次元オルソトープ型 `PS::Orthotope2` と 3 次元オルソトープ型 `PS::Orthotope3` がある。まずこれら 2 つを記述する。最後にこれらオルソトープ型のラッパーについて記述する。

6.5.2 PS::Orthotope2

PS::Orthotope2はPS::Vector2型のメンバ変数low_, high_を持つ。これらに対する様々なAPIや演算子を定義した。それらの宣言を以下に記述する。この節ではこれらについて詳しく記述する。

ソースコード 11: Orthotope2

```
1  namespace ParticleSimulator{
2      template<class T>
3      class Orthotope2{
4      public:
5          Vector2<T> low_;
6          Vector2<T> high_;
7
8          Orthotope2(): low_(9999.9), high_(-9999.9){}
9
10         Orthotope2(const Vector2<T> & _low, const Vector2<T> &
               _high)
11             : low_(_low), high_(_high){}
12
13         Orthotope2(const Orthotope2 & src) : low_(src.low_),
               high_(src.high_){}
14
15         Orthotope2(const Vector2<T> & center, const T length) :
16             low_(center-(Vector2<T>)(length)), high_(center+(
                   Vector2<T>)(length)) {
17         }
18
19         void initNegativeVolume(){
20             low_ = std::numeric_limits<float>::max() / 128;
21             high_ = -low_;
22         }
23
24         void init(){
25             initNegativeVolume();
26         }
27
28         void merge( const Orthotope2 & ort ){
29             this->high_.x = ( this->high_.x > ort.high_.x ) ?
                       this->high_.x : ort.high_.x;
30             this->high_.y = ( this->high_.y > ort.high_.y ) ?
```

```
                this->high_.y : ort.high_.y;
31              this->low_.x = ( this->low_.x <= ort.low_.x ) ?
                    this->low_.x : ort.low_.x;
32              this->low_.y = ( this->low_.y <= ort.low_.y ) ?
                    this->low_.y : ort.low_.y;
33          }
34
35          void merge( const Vector2<T> & vec ){
36              this->high_.x = ( this->high_.x > vec.x ) ? this->
                    high_.x : vec.x;
37              this->high_.y = ( this->high_.y > vec.y ) ? this->
                    high_.y : vec.y;
38              this->low_.x = ( this->low_.x <= vec.x ) ? this->
                    low_.x : vec.x;
39              this->low_.y = ( this->low_.y <= vec.y ) ? this->
                    low_.y : vec.y;
40          }
41
42          void merge( const Vector2<T> & vec, const T size){
43              this->high_.x = ( this->high_.x > vec.x + size ) ?
                    this->high_.x : vec.x + size;
44              this->high_.y = ( this->high_.y > vec.y + size ) ?
                    this->high_.y : vec.y + size;
45              this->low_.x = ( this->low_.x <= vec.x - size) ?
                    this->low_.x : vec.x - size;
46              this->low_.y = ( this->low_.y <= vec.y - size) ?
                    this->low_.y : vec.y - size;
47          }
48      };
49  }
50  namespace PS = ParticleSimulator;
```

6.5.2.1 メンバ変数

```
template<typename T>
PS::Vector2<T> PS::Orthotope2<T>::low_;

template<typename T>
PS::Vector2<T> PS::Orthotope2<T>::high_;
```

- **機能**

  メンバ変数 low_、high_を直接操作出来る。

6.5.2.2 コンストラクタ

```
template<typename T>
PS::Orthotope2<T>::Orthotope2();
```

- **引数**

  なし。ただし、テンプレート引数 T は PS::F32 または PS::F64 でなければならない。

- **機能**

  デフォルトコンストラクタ。メンバ変数 low_, high_は、それぞれ、(9999.9, 9999.9), (-9999.9, -9999.9) で初期化される。

```
template<typename T>
PS::Orthotope2<T>::Orthotope2(const Vector2<T> _low, const Vector2<T> _high);
```

- **引数**

  _low: 入力。const Vector2<T> 型。

  _high: 入力。const Vector2<T> 型。

  ここで、T は PS::F32 または PS::F64 でなければならない。

- **機能**

  メンバ変数 low_、high_を、それぞれ_low、_high で初期化する。

```
template<typename T>
PS::Orthotope2<T>::Orthotope2(const Vector2<T> & center, const T length);
```

- **引数**

  center: 入力。const Vector2<T> &

  length: 入力。const T 型。

- **機能**

  メンバ変数low_、high_を、それぞれ、center-(Vector2<T>)(length)、center+(Vector2<T>)(le
  で初期化する。

6.5.2.3 コピーコンストラクタ

```
template<typename T>
PS::Orthotope2<T>::Orthotope2(const Orthotope2<T> & src);
```

- **引数**

  src: 入力。const Orthotope2<T> &型。

- **機能**

  メンバ変数 low_、high_を、それぞれ、src.low_、src.high_で初期化する。

6.5.2.4 初期化

```
template<typename T>
PS::Orthotope2<T>::initNegativeVolume();
```

- **引数**

  なし。

- **機能**

  メンバ変数low_、high_を、それぞれ、(a, a)、$(\text{-}a, \text{-}a)$で初期化する。ここで、a=std::numeric_limits
  / 128 である。

```
template<typename T>
PS::Orthotope2<T>::init();
```

- **引数**

  なし。

- **機能**

  メンバ関数 initNegativeVolume と同じ機能を提供する。

6.5.2.5 結合操作

```
template<typename T>
void PS::Orthotope2<T>::merge( const Orthotope2 & ort )();
```

- **引数**

  ort: 入力。const Orthotope2 &型。

- **機能**

  メンバ変数 low_、high_を、当該長方形と長方形 ort を包含する最小の長方形を記述するように更新する。

```
template<typename T>
void PS::Orthotope2<T>::merge( const Vector2<T> & vec )();
```

- **引数**

  vec: 入力。const Vector2<T> &型。

- **機能**

  メンバ変数 low_、high_を、点 vec を包含するように更新する。

```
template<typename T>
void PS::Orthotope2<T>::merge( const Vector2<T> & vec, const T size )();
```

- **引数**

  vec: 入力。const Vector2<T> &型。

  size: 入力。const T 型。

- **機能**

  メンバ変数 low_、high_を、中心 vec、半径 size の円を包含するように更新する。

6.5.3 PS::Orthotope3

PS::Orthotope3 は PS::Vector3 型のメンバ変数 low_, high_を持つ。これらに対する様々な API や演算子を定義した。それらの宣言を以下に記述する。この節ではこれらについて詳しく記述する。

ソースコード 12: Orthotope3

```
1  namespace ParticleSimulator{
2      template<class T>
3      class Orthotope3{
4      public:
5          Vector3<T> low_;
6          Vector3<T> high_;
7
8          Orthotope3(): low_(9999.9), high_(-9999.9){}
9
10         Orthotope3(const Vector3<T> & _low, const Vector3<T> &
               _high)
11             : low_(_low), high_(_high) {}
12
13         Orthotope3(const Orthotope3 & src) : low_(src.low_),
               high_(src.high_){}
14
15         Orthotope3(const Vector3<T> & center, const T length) :
16             low_(center-(Vector3<T>)(length)), high_(center+(
                   Vector3<T>)(length)) {
17         }
18
19         void initNegativeVolume(){
20             low_ = std::numeric_limits<float>::max() / 128;
21             high_ = -low_;
22         }
23
24         void init(){
25             initNegativeVolume();
26         }
27
28         void merge( const Orthotope3 & ort ){
29             this->high_.x = ( this->high_.x > ort.high_.x ) ?
                       this->high_.x : ort.high_.x;
30             this->high_.y = ( this->high_.y > ort.high_.y ) ?
                       this->high_.y : ort.high_.y;
31             this->high_.z = ( this->high_.z > ort.high_.z ) ?
                       this->high_.z : ort.high_.z;
32             this->low_.x = ( this->low_.x <= ort.low_.x ) ?
                       this->low_.x : ort.low_.x;
```

```
33              this->low_.y = ( this->low_.y <= ort.low_.y ) ?
                      this->low_.y : ort.low_.y;
34              this->low_.z = ( this->low_.z <= ort.low_.z ) ?
                      this->low_.z : ort.low_.z;
35          }
36
37          void merge( const Vector3<T> & vec ){
38              this->high_.x = ( this->high_.x > vec.x ) ? this->
                      high_.x : vec.x;
39              this->high_.y = ( this->high_.y > vec.y ) ? this->
                      high_.y : vec.y;
40              this->high_.z = ( this->high_.z > vec.z ) ? this->
                      high_.z : vec.z;
41              this->low_.x = ( this->low_.x <= vec.x ) ? this->
                      low_.x : vec.x;
42              this->low_.y = ( this->low_.y <= vec.y ) ? this->
                      low_.y : vec.y;
43              this->low_.z = ( this->low_.z <= vec.z ) ? this->
                      low_.z : vec.z;
44          }
45
46          void merge( const Vector3<T> & vec, const T size){
47              this->high_.x = ( this->high_.x > vec.x + size ) ?
                      this->high_.x : vec.x + size;
48              this->high_.y = ( this->high_.y > vec.y + size ) ?
                      this->high_.y : vec.y + size;
49              this->high_.z = ( this->high_.z > vec.z + size) ?
                      this->high_.z : vec.z + size;
50              this->low_.x = ( this->low_.x <= vec.x - size) ?
                      this->low_.x : vec.x - size;
51              this->low_.y = ( this->low_.y <= vec.y - size) ?
                      this->low_.y : vec.y - size;
52              this->low_.z = ( this->low_.z <= vec.z - size) ?
                      this->low_.z : vec.z - size;
53          }
54      };
55  }
56  namespace PS = ParticleSimulator;
```

6.5.3.1 メンバ変数

```
template<typename T>
PS::Vector3<T> PS::Orthotope3<T>::low_;

template<typename T>
PS::Vector3<T> PS::Orthotope3<T>::high_;
```

- **機能**

  メンバ変数 low_、high_を直接操作出来る。

6.5.3.2 コンストラクタ

```
template<typename T>
PS::Orthotope3<T>::Orthotope3();
```

- **引数**

  なし。ただし、テンプレート引数 T は PS::F32 または PS::F64 でなければならない。

- **機能**

  デフォルトコンストラクタ。メンバ変数 low_, high_は、それぞれ、(9999.9, 9999.9, 9999.9), (-9999.9, -9999.9, -9999.9) で初期化される。

```
template<typename T>
PS::Orthotope3<T>::Orthotope3(const Vector3<T> _low, const Vector3<T> _high);
```

- **引数**

  _low: 入力。const Vector3<T> 型。

  _high: 入力。const Vector3<T> 型。

  ここで、T は PS::F32 または PS::F64 でなければならない。

- **機能**

  メンバ変数 low_、high_を、それぞれ_low、_high で初期化する。

```
template<typename T>
PS::Orthotope3<T>::Orthotope3(const Vector3<T> & center, const T length);
```

- **引数**

  center: 入力。const Vector3<T> &

  length: 入力。const T 型。

- **機能**

  メンバ変数low_、high_を、それぞれ、center-(Vector3<T>)(length)、center+(Vector3<T>)(le
  で初期化する。

6.5.3.3 コピーコンストラクタ

```
template<typename T>
PS::Orthotope3<T>::Orthotope3(const Orthotope3<T> & src);
```

- **引数**

  src: 入力。const Orthotope3<T> &型。

- **機能**

  メンバ変数 low_、high_を、それぞれ、src.low_、src.high_で初期化する。

6.5.3.4 初期化

```
template<typename T>
PS::Orthotope3<T>::initNegativeVolume();
```

- **引数**

  なし。

- **機能**

  メンバ変数 low_、high_を、それぞれ、(a, a, a)、$(-a, -a, -a)$ で初期化する。ここで、
  a=std::numeric_limits<float>::max() / 128 である。

```
template<typename T>
PS::Orthotope3<T>::init();
```

- **引数**

  なし。

- **機能**

  メンバ関数 initNegativeVolume と同じ機能を提供する。

6.5.3.5 結合操作

```
template<typename T>
void PS::Orthotope3<T>::merge( const Orthotope3 & ort )();
```

- **引数**

  ort: 入力。const Orthotope3 &型。

- **機能**

  メンバ変数 low_、high_を、当該長方形と長方形 ort を包含する最小の長方形を記述するように更新する。

```
template<typename T>
void PS::Orthotope3<T>::merge( const Vector3<T> & vec )();
```

- **引数**

  vec: 入力。const Vector3<T> &型。

- **機能**

  メンバ変数 low_、high_を、点 vec を包含するように更新する。

```
template<typename T>
void PS::Orthotope3<T>::merge( const Vector3<T> & vec, const T size )();
```

- **引数**

  vec: 入力。const Vector3<T> &型。

  size: 入力。const T 型。

- **機能**

  メンバ変数 low_、high_を、中心 vec、半径 size の球を包含するように更新する。

6.5.4 オルソトープ型のラッパー

オルソトープ型のラッパーの定義を以下に示す。

ソースコード 13: Orthotope Wrapper

```
1 namespace ParticleSimulator{
2     typedef Orthotope2<F32> F32ort2;
3     typedef Orthotope3<F32> F32ort3;
4     typedef Orthotope2<F64> F64ort2;
5     typedef Orthotope3<F64> F64ort3;
6 #ifdef PARTICLE_SIMULATOR_TWO_DIMENSION
7     typedef F32ort2 F32ort;
8     typedef F64ort2 F64ort;
9 #else
10    typedef F32ort3 F32ort;
11    typedef F64ort3 F64ort;
12 #endif
13 }
```

すなわち PS::F32ort2, PS::F32ort3, PS::F64ort2, PS::F64ort3 はそれぞれ単精度 2 次元オルソトープ、倍精度 2 次元オルソトープ、単精度 3 次元オルソトープ、倍精度 3 次元オルソトープである。FDPS で扱う空間座標系を 2 次元とした場合、PS::F32ort と PS::F64ort はそれぞれ単精度 2 次元オルソトープ、倍精度 2 次元オルソトープとなる。一方、FDPS で扱う空間座標系を 3 次元とした場合、PS::F32ort と PS::F64ort はそれぞれ単精度 3 次元オルソトープ、倍精度 3 次元オルソトープとなる。

6.6 対称行列型

6.6.1 概要

対称行列型には 2x2 対称行列型 PS::MatrixSym2 と 3x3 対称行列型 PS::MatrixSym3 がある。まずこれら 2 つを記述する。最後にこれら対称行列型のラッパーについて記述する。

6.6.2 PS::MatrixSym2

PS::MatrixSym2 は xx, yy, xy の 3 要素を持つ。これらに対する様々な API や演算子を定義した。それらの宣言を以下に記述する。この節ではこれらについて詳しく記述する。

ソースコード 14: MatrixSym2

```
1 namespace ParticleSimulator{
2     template<class T>
3     class MatrixSym2{
4     public:
5         // メンバ変数 3 要素
6         T xx, yy, xy;
```

```
 7
 8         // コンストラクタ
 9         MatrixSym2() : xx(T(0)), yy(T(0)), xy(T(0)) {}
10         MatrixSym2(const T _xx, const T _yy, const T _xy)
11             : xx(_xx), yy(_yy), xy(_xy) {}
12         MatrixSym2(const T s) : xx(s), yy(s), xy(s){}
13         MatrixSym2(const MatrixSym2 & src) : xx(src.xx), yy(src
               .yy), xy(src.xy) {}
14
15         // 代入演算子
16         const MatrixSym2 & operator = (const MatrixSym2 & rhs);
17
18         // 加減算
19         MatrixSym2 operator + (const MatrixSym2 & rhs) const;
20         const MatrixSym2 & operator += (const MatrixSym2 & rhs)
               const;
21         MatrixSym2 operator - (const MatrixSym2 & rhs) const;
22         const MatrixSym2 & operator -= (const MatrixSym2 & rhs)
               const;
23
24         // トレースの計算
25         T getTrace() const;
26
27         // MatrixSym2<U>への型変換
28         template <typename U>
29         operator MatrixSym2<U> () const;
30     }
31 }
32 namespace PS = ParticleSimulator;
```

6.6.2.1 コンストラクタ

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

- **引数**

  なし。

- **機能**

デフォルトコンストラクタ。メンバ xx,yy,xy は 0 で初期化される。

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

- **引数**

  _xx: 入力。const T 型。

  _yy: 入力。const T 型。

  _xy: 入力。const T 型。

- **機能**

  メンバ xx、yy、xy をそれぞれ_xx、_yy、_xy で初期化する。

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

- **引数**

  s: 入力。const T 型。

- **機能**

  メンバ xx、yy、xy すべてを s の値で初期化する。

6.6.2.2 コピーコンストラクタ

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

- **引数**

  src: 入力。const PS::MatrixSym2<T> &型。

- **機能**

  コピーコンストラクタ。src で初期化する。

6.6.2.3 代入演算子

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

- **引数**

  rhs: 入力。const PS::MatrixSym2<T> &型。

- **返り値**

  const PS::MatrixSym2<T> &型。rhs の xx,yy,xy の値を自身のメンバ xx,yy,xy に代入し自身の参照を返す。代入演算子。

6.6.2.4 加減算

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

- **引数**

  rhs: 入力。const PS::MatrixSym2<T> &型。

- **返り値**

  PS::MatrixSym2<T> 型。rhs の xx,yy,xy の値と自身のメンバ xx,yy,xy の値の和を取った値を返す。

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

- **引数**

  rhs: 入力。const PS::MatrixSym2<T> &型。

- **返り値**

  const PS::MatrixSym2<T> &型。rhs の xx,yy,xy の値を自身のメンバ xx,yy,xy に足し、自身を返す。

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

- **引数**

  rhs: 入力。const PS::MatrixSym2<T> &型。

- **返り値**

  PS::MatrixSym2<T> 型。rhs の xx,yy,xy の値と自身のメンバ xx,yy,xy の値の差を取った値を返す。

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

- **引数**

  rhs: 入力。const PS::MatrixSym2<T> &型。

- **返り値**

  const PS::MatrixSym2<T> &型。自身のメンバ xx,yy,xy から rhs の xx,yy,xy を引き自身を返す。

6.6.2.5 トレースの計算

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

- **引数**

  なし

- **返り値**

  T 型。

- **機能**

  トレースを計算し、その結果を返す。

6.6.2.6 MatrixSym2<U> への型変換

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

- **引数**

  なし。

- **返り値**

  const PS::MatrixSym2<U> 型。

- **機能**

  const PS::MatrixSym2<T> 型を const PS::MatrixSym2<U> 型にキャストする

6.6.3 PS::MatrixSym3

PS::MatrixSym3 は xx, yy, zz, xy, xz, yz の 6 要素を持つ。これらに対する様々な API や演算子を定義した。それらの宣言を以下に記述する。この節ではこれらについて詳しく記述する。

ソースコード 15: MatrixSym3

```
1  namespace ParticleSimulator{
2      template<class T>
3      class MatrixSym3{
4      public:
5          // メンバ変数 6 要素
6          T xx, yy, zz, xy, xz, yz;
7
8          // コンストラクタ
9          MatrixSym3() : xx(T(0)), yy(T(0)), zz(T(0)),
10                        xy(T(0)), xz(T(0)), yz(T(0)) {}
11         MatrixSym3(const T _xx, const T _yy, const T _zz,
12                    const T _xy, const T _xz, const T _yz )
13                        : xx(_xx), yy(_yy), zz(_zz),
14                        xy(_xy), xz(_xz), yz(_yz) {}
15         MatrixSym3(const T s) : xx(s), yy(s), zz(s),
16                                 xy(s), xz(s), yz(s) {}
17         MatrixSym3(const MatrixSym3 & src) :
18             xx(src.xx), yy(src.yy), zz(src.zz),
```

```
19              xy(src.xy), xz(src.xz), yz(src.yz) {}
20
21          // 代入演算子
22          const MatrixSym3 & operator = (const MatrixSym3 & rhs);
23
24          // 加減算
25          MatrixSym3 operator + (const MatrixSym3 & rhs) const;
26          const MatrixSym3 & operator += (const MatrixSym3 & rhs)
                const;
27          MatrixSym3 operator - (const MatrixSym3 & rhs) const;
28          const MatrixSym3 & operator -= (const MatrixSym3 & rhs)
                const;
29
30          // トレースを取る
31          T getTrace() const;
32
33          // MatrixSym3<U>への型変換
34          template <typename U>
35          operator MatrixSym3<U> () const;
36      }
37  }
38  namespace PS = ParticleSimulator;
```

6.6.3.1 コンストラクタ

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

- **引数**

  なし。

- **機能**

  デフォルトコンストラクタ。6要素は0で初期化される。

```
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);
```

- **引数**

  _xx: 入力。const T 型。

  _yy: 入力。const T 型。

  _zz: 入力。const T 型。

  _xy: 入力。const T 型。

  _xz: 入力。const T 型。

  _yz: 入力。const T 型。

- **機能**

  メンバ xx、yy、zz、xy、xz、yz をそれぞれ_xx、_yy、_zz、_xy、_xz、_yz で初期化する。

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

- **引数**

  s: 入力。const T 型。

- **機能**

  ６要素すべてを s の値で初期化する。

6.6.3.2 コピーコンストラクタ

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

- **引数**

  src: 入力。const PS::MatrixSym3<T> &型。

- **機能**

  コピーコンストラクタ。src で初期化する。

6.6.3.3 代入演算子

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

- **引数**

  rhs: 入力。const PS::MatrixSym3<T> &型。

- **返り値**

  const PS::MatrixSym3<T> &型。rhs の 6 要素それぞれの値を自身の 6 要素それぞれに代入し自身の参照を返す。代入演算子。

6.6.3.4 加減算

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

- **引数**

  rhs: 入力。const PS::MatrixSym3<T> &型。

- **返り値**

  PS::MatrixSym3<T> 型。rhs の 6 要素それぞれの値と自身の 6 要素の値の和を取った値を返す。

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

- **引数**

  rhs: 入力。const PS::MatrixSym3<T> &型。

- **返り値**

  const PS::MatrixSym3<T> &型。rhs の 6 要素それぞれの値を自身の 6 要素それぞれに足し、自身を返す。

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

- **引数**

  rhs: 入力。const PS::MatrixSym3<T> &型。

- **返り値**

  PS::MatrixSym3<T> 型。rhs の 6 要素それぞれの値と自身の 6 要素それぞれの値の差を取った値を返す。

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

- **引数**

  rhs: 入力。const PS::MatrixSym3<T> &型。

- **返り値**

  const PS::MatrixSym3<T> &型。自身の 6 要素それぞれから rhs の 6 要素それぞれを引き自身を返す。

6.6.3.5 トレースの計算

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

- **引数**

  なし

- **返り値**

  T 型。

- **機能**

  トレースを計算し、その結果を返す。

6.6.3.6 MatrixSym3<U> への型変換

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

- **引数**

  なし。

- **返り値**

  const PS::MatrixSym3<U> 型。

- **機能**

  const PS::MatrixSym3<T> 型を const PS::MatrixSym3<U> 型にキャストする

6.6.4 対称行列型のラッパー

対称行列型のラッパーの定義を以下に示す。

ソースコード 16: matrixsymwrapper

```
1  namespace ParticleSimulator{
2      typedef MatrixSym2<F32> F32mat2;
3      typedef MatrixSym3<F32> F32mat3;
4      typedef MatrixSym2<F64> F64mat2;
5      typedef MatrixSym3<F64> F64mat3;
6  #ifdef PARTICLE_SIMULATOR_TWO_DIMENSION
7      typedef F32mat2 F32mat;
8      typedef F64mat2 F64mat;
9  #else
10     typedef F32mat3 F32mat;
11     typedef F64mat3 F64mat;
12 #endif
13 }
14 namespace PS = ParticleSimulator;
```

すなわち PS::F32mat2, PS::F32mat3, PS::F64mat2, PS::F64mat3 はそれぞれ単精度 2x2 対称行列、倍精度 2x2 対称行列、単精度 3x3 対称行列、倍精度 3x3 対称行列である。FDPS で扱う空間座標系を 2 次元とした場合、PS::F32mat と PS::F64mat はそれぞれ単精度 2x2 対称行列、倍精度 2x2 対称行列となる。一方、FDPS で扱う空間座標系を 3 次元とした場合、PS::F32mat と PS::F64mat はそれぞれ単精度 3x3 対称行列、倍精度 3x3 対称行列となる。

6.7 PS::SEARCH_MODE 型

6.7.1 概要

本節では、PS::SEARCH_MODE 型について記述する。PS::SEARCH_MODE 型は相互作用ツリークラスのテンプレート引数としてのみ使用されるものである。この型によって、相互作用ツリークラスで計算する相互作用のモードを決定する。PS::SEARCH_MODE 型には以下がある:

- 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
- PS::SEARCH_MODE_LONG_SYMMETRY
- PS::SEARCH_MODE_LONG_CUTOFF_SCATTER

以下で、それぞれが対応する相互作用のモードについて記述する。

6.7.2 PS::SEARCH_MODE_LONG

この型を使用するのは、遠くの粒子からの寄与を複数の粒子にまとめた超粒子からの寄与として計算する場合である。開放境界条件における重力やクーロン力に適用できる (周期境界条件では使用不可能)。

6.7.3 PS::SEARCH_MODE_LONG_CUTOFF

この型を使用するのは、遠くの粒子からの寄与を複数の粒子にまとめた超粒子からの寄与として計算し、かつ有限の距離までの寄与しか計算しない場合である。周期境界条件における重力やクーロン力 (Particle Mesh 法の並用が必要) などに適用できる。

6.7.4 PS::SEARCH_MODE_GATHER

この型を使用するのは、相互作用の到達距離が有限でかつ、その到達距離が i 粒子の大きさで決まる場合である。

6.7.5 PS::SEARCH_MODE_SCATTER

この型を使用するのは、相互作用の到達距離が有限でかつ、その到達距離が j 粒子の大きさで決まる場合である。

6.7.6 PS::SEARCH_MODE_SYMMETRY

この型を使用するのは、相互作用の到達距離が有限でかつ、その到達距離が i, j 粒子のうち大きいほうのサイズで決まる場合である。

6.7.7 PS::SEARCH_MODE_LONG_SCATTER

基本的には SEARCH_MODE_LONG と同じであるが、i 粒子と j 粒子の距離が j 粒子の探査半径よりも短い場合は、その j 粒子は超粒子に含めない (超粒子としてではなく粒子として扱う)。

6.7.8 PS::SEARCH_MODE_LONG_SYMMETRY

基本的には SEARCH_MODE_LONG と同じであるが、i 粒子と j 粒子の距離が、i 粒子と j 粒子の探査半径のどちらか大きい方よりも短い場合は、その j 粒子は超粒子に含めない (超粒子としてではなく粒子として扱う)。

6.7.9 PS::SEARCH_MODE_LONG_CUTOFF_SCATTER

未実装。

6.8 列挙型

6.8.1 概要

本節では FDPS で定義されている列挙型について記述する。列挙型には BOUNDARY_CONDITION 型と INTERACTION_LIST_MODE 型が存在する。以下、各列挙型について記述する。

6.8.2 PS::BOUNDARY_CONDITION 型

6.8.2.1 概要

BOUNDARY_CONDITION 型は境界条件を指定するためのデータ型である。これは以下のように定義されている。

ソースコード 17: boundarycondition

```
1 namespace ParticleSimulator{
2     enum BOUNDARY_CONDITION{
3         BOUNDARY_CONDITION_OPEN,
4         BOUNDARY_CONDITION_PERIODIC_X,
5         BOUNDARY_CONDITION_PERIODIC_Y,
6         BOUNDARY_CONDITION_PERIODIC_Z,
```

```
 7           BOUNDARY_CONDITION_PERIODIC_XY,
 8           BOUNDARY_CONDITION_PERIODIC_XZ,
 9           BOUNDARY_CONDITION_PERIODIC_YZ,
10           BOUNDARY_CONDITION_PERIODIC_XYZ,
11           BOUNDARY_CONDITION_SHEARING_BOX,
12           BOUNDARY_CONDITION_USER_DEFINED,
13       };
14   }
```

以下にどの変数がどの境界条件に対応するかを記述する。

6.8.2.2 PS::BOUNDARY_CONDITION_OPEN

開放境界となる。

6.8.2.3 PS::BOUNDARY_CONDITION_PERIODIC_X

x 軸方向のみ周期境界、その他の軸方向は開放境界となる。周期の境界の下限は閉境界、上限は開境界となっている。この境界の規定はすべての軸方向にあてはまる。

6.8.2.4 PS::BOUNDARY_CONDITION_PERIODIC_Y

y 軸方向のみ周期境界、その他の軸方向は開放境界となる。

6.8.2.5 PS::BOUNDARY_CONDITION_PERIODIC_Z

z 軸方向のみ周期境界、その他の軸方向は開放境界となる。

6.8.2.6 PS::BOUNDARY_CONDITION_PERIODIC_XY

x, y 軸方向のみ周期境界、その他の軸方向は開放境界となる。

6.8.2.7 PS::BOUNDARY_CONDITION_PERIODIC_XZ

x, z 軸方向のみ周期境界、その他の軸方向は開放境界となる。

6.8.2.8 PS::BOUNDARY_CONDITION_PERIODIC_YZ

y, z 軸方向のみ周期境界、その他の軸方向は開放境界となる。

6.8.2.9 PS::BOUNDARY_CONDITION_PERIODIC_XYZ

x, y, z 軸方向すべてが周期境界となる。

6.8.2.10 PS::BOUNDARY_CONDITION_SHEARING_BOX

未実装。

6.8.2.11 PS::BOUNDARY_CONDITION_USER_DEFINED

未実装。

6.8.3 PS::INTERACTION_LIST_MODE 型

6.8.3.1 概要

INTERACTION_LIST_MODE 型は相互作用リストを再利用するかどうかを決定するためのデータ型である。これは以下のように定義されている。

ソースコード 18: boundarycondition

```
1 namespace ParticleSimulator{
2     enum INTERACTION_LIST_MODE{
3         MAKE_LIST,
4         MAKE_LIST_FOR_REUSE,
5         REUSE_LIST,
6     };
7 }
```

このデータ型は calcForceAllAndWriteBack() 等の関数の引数として使われる (詳しくはセクション 9.1.4.2.3 を参照)。

6.8.3.2 PS::MAKE_LIST

相互作用リストを毎回作り相互作用計算を行う場合に用いる。相互作用リストの再利用はできない。

6.8.3.3 PS::MAKE_LIST_FOR_REUSE

相互作用リストを再利用し相互作用計算を行いたい場合に用いる。このオプションを選択する事で FDPS は相互作用リストを作りそれを保持する。作成した相互作用リストは PS::MAKE_LIST_FOR_REUSE もしくは PS::MAKE_LIST を用いて相互作用計算を行った際に破棄される。

6.8.3.4 PS::REUSE_INTERACTION_LIST

相互作用リストを再利用し相互作用計算を行う。再利用される相互作用リストはPS::MAKE_LIST_FOR_R
を選択時に作成した相互作用リストである。

6.8.4 PS::EXCHANGE$_{L}ET_{M}ODE$ 型

6.8.4.1 概要

EXCHANGE_LET_MODE 型は LET 交換の方法を決定するためのデータ型である。これは以下のように定義されている。

ソースコード 19: boundarycondition

```
1 namespace ParticleSimulator{
2     enum EXCHANGE_LET_MODE{
3         EXCHANGE_LET_A2A,
4         EXCHANGE_LET_P2P_EXACT,
5         EXCHANGE_LET_P2P_FAST,
6     };
7 }
```

このデータ型は calcForceAllAndWriteBack() 等の関数の引数として使われる (詳しくはセクション 9.1.4.2.3 を参照)。

6.8.4.2 PS::EXCHANGE_LET_A2A

LET の交換に MPI_Alltoall を使用する。

6.8.4.3 PS::EXCHANGE_LET_P2P_EXACT

LET の交換に MPI_Alltoall を使用せず、MPI_Allgather と MPI_Isend/recv を使用する。MPI_Alltoall が効率的に動かない計算機では、こちらの方が速い場合がある。結果は丸め誤差の範囲で PS::EXCHANGE_LET_A2A を用いた場合と一致する。

6.8.4.4 EXCHANGE$_{L}ET_{P}2P_{F}AST$

LET の交換に MPI_Alltoall を使用せず、MPI_Allgather と MPI_Isend/recv を使用する。結果は PS::EXCHANGE_LET_P2P_EXACT を用いた場合と異なるが、より通信量が減っているため、高速に動作する可能性がある。

6.9 PS::TimeProfile

6.9.1 概要

本節では、PS::TimeProfile 型について記述する。PS::TimeProfile 型は FDPS で使われる３つのクラス、領域分割クラス、粒子群クラス、相互作用ツリークラス、各メソッドの計算時間を格納するクラスである。これら３つのクラスには`PS::TimeProfile getTimeProfile()`というメソッドが存在し、このメソッドをつかって、ユーザーは各メソッドの計算時間を取得出来る。

このクラスは以下のように記述されている。

ソースコード 20: TimeProfile

```
1  namespace ParticleSimulator{
2      class TimeProfile{
3      publid:
4          F64 collect_sample_particle;
5          F64 decompose_domain;
6          F64 exchange_particle;
7          F64 make_local_tree;
8          F64 make_global_tree;
9          F64 calc_force;
10         F64 calc_moment_local_tree;
11         F64 calc_moment_global_tree;
12         F64 make_LET_1st;
13         F64 make_LET_2nd;
14         F64 exchange_LET_1st;
15         F64 exchange_LET_2nd;
16     };
17 }
```

6.9.1.1 加算

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

- **引数**

  rhs: 入力。const TimeProfile &型。

- **返り値**

PS::TimeProfile 型。rhs のすべてのメンバ変数の値と自身のメンバ変数の値の和を取った値を返す。

6.9.1.2 縮約

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

- **引数**

  なし。

- **返り値**

  PS::F64 型。すべてのメンバ変数の値の和を返す。

6.9.1.3 初期化

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

- **引数**

  なし。

- **返り値**

  なし。

- **機能**

  すべてのメンバ変数に 0 を代入する。

7　ユーザー定義クラス・ユーザー定義関数オブジェクト

7.1　概要

本節では、ユーザーが定義するクラスとファンクタについて記述する。ユーザー定義クラスとなるのは、FullParticle クラス、EssentialParticleI クラス、EssentialParticleJ クラス、Moment クラス、SuperParticleJ クラス、Force クラス、ヘッダクラスである。またユーザー定義の関数オブジェクトには、関数オブジェクト calcForceEpEp、calcForceSpEp がある。

FullParticle クラスは、ある 1 粒子の情報すべてを持つクラスであり、粒子群クラスにテンプレート引数として渡されるものである (節 2.3 の手順 0)。

関数オブジェクト calcForceEpEp と calcForceSpEp は、それぞれ j 粒子から i 粒子への作用を計算する関数オブジェクトと超粒子から i 粒子への作用を計算する関数オブジェクトである。これらは相互作用ツリークラスの API の引数として渡されるものである (節 2.3 の手順 0)。超粒子を必要とする PS::SEARCH_MODE 型 (PS::SEARCH_MODE_LONG か PS::SEARCH_MODE_LONG_CUTOFF) 以外を使用する場合には、関数オブジェクト calcForceSpEp を定義する必要はない。

EssentialParticleI クラス、EssentialParticleJ クラス、Moment クラス、SuperParticleJ クラス、Force クラスは粒子間の相互作用の定義を補助するものである。これらのクラスのうち EssentialParticleI クラス、EssentialParticleJ クラス、Force クラスはそれぞれ相互作用を計算する際に i 粒子に必要な情報、相互作用を計算する際に j 粒子に必要な情報、相互作用の結果の情報を持つ。これらは FullParticle クラスのサブセットであるため、これらを FullParticle クラスで代用することも可能である。しかし、FullParticle クラスは相互作用の定義に必要のないデータを多く含む場合も考えられるため、計算コストを軽減したいならば、これらのクラスを使用することを検討するべきである。Moment クラスと SuperParticleJ クラスは、それぞれツリーセルのモーメント情報を持つクラスと超粒子に必要な情報を持つ SuperParticleJ クラスである。ユーザーが定義する必要があるのは、超粒子を使う必要がある場合、すなわち PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG か PS::SEARCH_MODE_LONG_CUTOFF を選んだ場合のみである。

ヘッダクラスは入出力ファイルのヘッダ情報を持つ。

この節で記述するのは、これらのクラスや関数オブジェクトを定義する際の規定である。ユーザーはこれらの間でのデータのやりとりや、関数オブジェクト内でのデータの加工についてコードに書く必要がある。これらは上に挙げたクラスのメンバ関数と関数オブジェクト内で行われる。以下、必要なメンバ関数とその規定について記述する。

7.2　FullParticle クラス

7.2.1　概要

FullParticle クラスは粒子情報すべてを持つクラスであり、節 2.3 の手順 0 で、粒子群クラスに渡されるユーザー定義クラスの 1 つである。ユーザーはこのクラスに対して、どのようなメンバ変数、メンバ関数を定義してもかまわない。ただし、FDPS から FullParticle クラ

スの情報にアクセスする ために、ユーザーはいくつかの決まった名前のメンバ関数を定義する必要がある。以下、この節の前提、常に必要なメンバ関数と、場合によっては必要なメンバ関数について記述する。

7.2.2 前提

この節の中では、以下のように、FullParticle クラスとして FP というクラスを一例とする。FP という名前は自由に変えることができる。

```
class FP;
```

7.2.3 必要なメンバ関数

7.2.3.1 概要

常に必要なメンバ関数は FP::getPos と FP::copyFromForce である。FP::getPos は FullParticle の位置情報を FDPS に読み込ませるための関数で、FP::copyFromForce は計算された相互作用の結果を FullParticle に書き戻す関数である。これらのメンバ関数の記述例と解説を以下に示す。

7.2.3.2 FP::getPos

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

- **引数**

  なし

- **返値**

  PS::F32vec 型または PS::F64vec 型。FP クラスのオブジェクトの位置情報を保持したメンバ変数。

- **機能**

  FP クラスのオブジェクトの位置情報を保持したメンバ変数を返す。

7.2.3.3 FP::copyFromForce

```
class Force;

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

- **引数**

  force: 入力。const Force &型。粒子の相互作用の計算結果を保持。

- **返値**

  なし。

- **機能**

  粒子の相互作用の計算結果を FP クラスへ書き戻す。

7.2.4 場合によっては必要なメンバ関数

7.2.4.1 概要

本節では、以下に示す場合に必要となるメンバ関数について記述する:

[1] 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合
[2] 粒子群クラスのファイル入出力 API を用いる場合
[3] 粒子群クラスの API である ParticleSystem::adjustPositionIntoRootDomain を用いる場合
[4] 拡張機能の Particle Mesh クラスを用いる場合

7.2.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合

7.2.4.2.1 FP::getRSearch

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

- **引数**

  なし

- **返値**

  PS::F32 型または PS::F64 型。 FP クラスのオブジェクトの近傍粒子を探す半径の大きさを保持したメンバ変数。

- **機能**

  FP クラスのオブジェクトの近傍粒子を探す半径の大きさを保持したメンバ変数を返す。

7.2.4.3 粒子群クラスのファイル入出力 API を用いる場合

粒子群クラスのファイル入出力 API である ParticleSystem::readParticleAscii, ParticleSystem::writeParticleAscii, ParticleSystem::readParticleBinary, ParticleSystem::writeParticleBinary を使用するときにそれぞれ readAscii, writeAscii, readBinary, writeBinary というメンバ関数が必要となる (readAscii, writeAscii, readBinary, writeAscii 以外の名前を使うことも可能。詳しくは節 9.1.3.2.3 を参照)。以下、readAscii と writeAscii, readBinary, writeBinary の規定について記述する。

7.2.4.3.1 FP::readAscii

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

- **引数**

  fp: FILE *型。粒子データの入力ファイルを指すファイルポインタ。

- **返値**

  なし。

- **機能**

  アスキー形式の粒子データの入力ファイル 1 行から FP クラス 1 個の情報を読み取り、メンバ変数に値を入れる。ここで 1 行とは、FP クラス 1 個のデータの末尾に**必ず唯一つの**改行コード (\n) がある状態の意味である。

7.2.4.3.2　FP::writeAscii

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

- **引数**

  fp: FILE *型。粒子データの出力ファイルを指すファイルポインタ。

- **返値**

  なし。

- **機能**

  粒子データの出力ファイルへFPクラス1個の情報を1行としてアスキー形式で書き出す。ここで1行とは、FPクラス1個のデータの末尾に**必ず唯一つの**改行コード (\n) がある状態の意味である。

7.2.4.3.3　FP::readBinary

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

- **引数**

  fp: FILE *型。粒子データの入力ファイルを指すファイルポインタ。

- **返値**

  なし。

- **機能**

  バイナリ形式の粒子データの入力ファイルからFPクラス1個の情報を読み取り、メンバ変数に値を入れる。

7.2.4.3.4　FP::writeBinary

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

- **引数**

  fp: FILE *型。粒子データの出力ファイルを指すファイルポインタ。

- **返値**

  なし。

- **機能**

  粒子データの出力ファイルへ FP クラス 1 個の情報をバイナリ形式で書き出す。

7.2.4.4　ParticleSystem::adjustPositionIntoRootDomain を用いる場合

7.2.4.4.1　FP::setPos

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

- **引数**

  pos_new: 入力。const PS::F32vec または const PS::F64vec 型。FDPS 側で修正した粒子の位置情報。

- **返値**

  なし。

- **機能**

  FDPS が修正した粒子の位置情報を FP クラスのオブジェクトの位置情報に書き込む。

7.2.4.5　Particle Mesh クラスを用いる場合

Particle Mesh クラスを用いる場合には、メンバ関数 FP::getChargeParticleMesh と FP::copyFromForceParticleMesh を用意する必要がある。以下にそれぞれの規定を記述する。

7.2.4.5.1　FP::getChargeParticleMesh

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

- **引数**

  なし。

- **返値**

  PS::F32 型または PS::F64 型。１つの粒子の質量または電荷の変数を返す。

- **機能**

  １つの粒子の質量または電荷を表すメンバ変数を返す。

7.2.4.5.2　FP::copyFromForceParticleMesh

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

- **引数**

  acc_pm: const PS::F32vec 型または const PS::F64vec 型。１つの粒子の Particle Mesh による力の計算結果。

- **返値**

  なし。

- **機能**

  １つの粒子の Particle Mesh による力の計算結果をこの粒子のメンバ変数に書き込む。

7.2.4.6　粒子交換時に粒子データをシリアライズして送る場合

粒子交換時に粒子データをシリアライズして送る場合には、メンバ関数に FP::pack と FP::unpack を用意する必要がある。以下にそれぞれの規定を記述する。

7.2.4.6.1　FP::pack

```
class FP {
public:
    static PS::S32 pack(const PS::S32 n_ptcl, const FP *ptcl[], char *buf,
                        size_t & packed_size, const size_t max_buf_size);
};
```

- **引数**

  n_ptcl: 粒子交換時に送る粒子の数。
  ptcl: 送る粒子へのポインタの配列。
  buf: 送信バッファーの先頭アドレス。
  packed_size: ユーザーがバッファーへ書き込むサイズ。単位はバイト。
  max_buf_size: 送信バッファーの書き込み可能な領域のサイズ。単位はバイト。

- **返値**

  PS::S32 型。packed_size が max_buf_size を超えた場合は-1 を返す。それ以外の場合は 0 を返す。

- **機能**

  粒子交換時に送信する粒子をシリアライズし、送信バッファーに書き込む。

7.2.4.6.2　FP::unPack

```
class FP {
public:
    static PS::S32 unPack(const PS::S32 n_ptcl, FP ptcl[], const char *buf);
};
```

- **引数**

  n_ptcl: 粒子交換時に受け取る粒子の数。
  ptcl: 受け取る粒子の配列。
  buf: 受信バッファーの先頭アドレス。

- **返値**

  PS::S32 型。デシリアライズに失敗した場合は-1 を返す。それ以外の場合は 0 を返す。

- **機能**

粒子交換時に受信する粒子をデシリアライズし、粒子配列に書き込む。詳しくはセクション 9.1.3.2.4.1 を参照。デシリアライズに失敗した場合は PS::Abort() が呼ばれプログラムは終了する。

7.3 EssentialParticleI クラス

7.3.1 概要

EssentialParticleI クラスは相互作用の計算に必要な i 粒子の情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。EssentialParticleI クラスは FullParticle クラス (節 7.2) のサブセットである。FDPS は、このクラスのデータにアクセスする必要がある。そのため、EssentialParticleI クラスはいくつかのメンバ関数を持つ必要がある。以下、この節の前提、常に必要なメンバ関数と、場合によっては必要なメンバ関数について記述する。

7.3.2 前提

この節の中では、EssentialParticleI クラスとして EPI というクラスを一例として使う。また、FullParticle クラスの一例として FP というクラスを使う。EPI, FP というクラス名は変更可能である。

EPI と FP の宣言は以下の通りである。

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

7.3.3 必要なメンバ関数

7.3.3.1 概要

常に必要なメンバ関数は EPI::getPos と EPI::copyfromFP である。EPI::getPos は EPI クラスの位置情報を FDPS に読み込ませるための関数で、EPI::copyFromFP は FP クラスの情報を EPI クラスに書きこむ関数である。これらのメンバ関数の記述例と解説を以下に示す。

7.3.3.2 EPI::getPos

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

- **引数**

  なし

- **返値**

  PS::F64vec 型。EPI クラスの位置情報を保持したメンバ変数。

- **機能**

  EPI クラスのオブジェクトの位置情報を保持したメンバ変数を返す。

7.3.3.3 EPI::copyFromFP

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

- **引数**

  fp: 入力。const FP &型。FP クラスの情報を持つ。

- **返値**

  なし。

- **機能**

  FP クラスの持つ１粒子の情報の一部を EssnetialParticleI クラスに書き込む。

7.3.4 場合によっては必要なメンバ関数

7.3.4.1 概要

本節では、場合によっては必要なメンバ関数について記述する。相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_GATHER または PS::SEARCH_MODE_SYMMETRY を用いる場合に必要となるメンバ関数ついて記述する。

7.3.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_GATHER または PS::SEARCH_MODE_SYMMETRY を用いる場合

7.3.4.2.1 EPI::getRSearch

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

- **引数**

  なし

- **返値**

  PS::F32 型または PS::F64 型。 EPI クラスの近傍粒子を探す半径の大きさを保持したメンバ変数。

- **機能**

  EPI クラスの近傍粒子を探す半径の大きさを保持したメンバ変数を返す。

7.4 EssentialParticleJ クラス

7.4.1 概要

EssentialParticleJ クラスは相互作用の計算に必要な j 粒子の情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。EssentialParticleJ クラスは FullParticle クラス (節 7.2) のサブセットである。FDPS は、このクラスのデータにアクセスする必要がある。このために、EssentialParticleJ クラスはいくつかのメンバ関数を持つ必要がある。以下、この節の前提、常に必要なメンバ関数と、場合によっては必要なメンバ関数について記述する。

7.4.2 前提

この節の中では、EssentialParticleJ クラスとして EPJ というクラスを一例として使う。また、FullParticle クラスの一例として FP というクラスを使う。EPJ, FP というクラス名は変更可能である。

EPJ と FP の宣言は以下の通りである。

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

7.4.3 必要なメンバ関数

7.4.3.1 概要

常に必要なメンバ関数はEPJ::getPosとEPJ::copyfromFPである。EPJ::getPosはEPJクラスの位置情報をFDPSに読み込ませるための関数で、EPJ::copyFromFPはFPクラスの情報をEPJクラスに書きこむ関数である。これらのメンバ関数の記述例と解説を以下に示す。

7.4.3.2 EPJ::getPos

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

- **引数**

  なし

- **返値**

  PS::F64vec型。EPJクラスの位置情報を保持したメンバ変数。

- **機能**

  EPJクラスの位置情報を保持したメンバ変数を返す。

7.4.3.3 EPJ::copyFromFP

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

- **引数**

  fp: 入力。const FP &型。FPクラスの情報を持つ。

- **返値**

  なし。

- **機能**

  FPクラスの持つ1粒子の情報の一部をEPJクラスに書き込む。

7.4.4 場合によっては必要なメンバ関数

7.4.4.1 概要

本節では、場合によっては必要なメンバ関数について記述する。相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合に必要なメンバ関数、列挙型の BOUNDARY_CONDITION 型に PS::BOUNDARY_CONDITION_OPEN 以外を選んだ場合に必要となるメンバ関数について記述する。なお、既存の Moment クラスや SuperParticleJ クラスを用いる際に必要となるメンバ変数はこれら既存のクラスの節を参照のこと。

7.4.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合

7.4.4.2.1 EPJ::getRSearch

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

- **引数**

  なし

- **返値**

  PS::F32 型または PS::F64 型。 EPJ クラスの近傍粒子を探す半径の大きさを保持したメンバ変数。

- **機能**

  EPJ クラスの近傍粒子を探す半径の大きさを保持したメンバ変数を返す。

  なお、PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG_CUTOFF を用いる場合、すべての EPJ が同じ値を返すように定義しなければならない。

7.4.4.3 BOUNDARY_CONDITION 型に PS::BOUNDARY_CONDITION_OPEN 以外を用いる場合

7.4.4.3.1 EPJ::setPos

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

- **引数**

  pos_new: 入力。const PS::F32vec または const PS::F64vec 型。FDPS 側で修正した粒子の位置情報。

- **返値**

  なし。

- **機能**

  FDPS が修正した粒子の位置情報を EPJ クラスの位置情報に書き込む。

7.4.4.4 粒子の id 番号から対応する EPJ を取得したい場合

7.4.4.4.1 EPJ::getId

```
class EPJ {
public:
    PS::S64 getId();
};
```

- **引数**

  なし。

- **返値**

  PS::S64 型。

- **機能**

  PS::TreeForForce::getEpjFromId() を使用する場合に必要。詳しくは 9.1.4.2.7 を参照。

7.4.4.5 LET 交換時に粒子データをシリアライズして送る場合

LET 交換時に粒子データをシリアライズして送る場合には、メンバ関数に EPJ::pack と EPJ::unpack を用意する必要がある。以下にそれぞれの規定を記述する。

7.4.4.5.1 EPJ::pack

```
class EPJ {
public:
    static PS::S32 pack(const PS::S32 n_ptcl, const EPJ *ptcl[], char *buf,
                        size_t & packed_size, const size_t max_buf_size);
};
```

- **引数**

  n_ptcl: LET 交換時に送る粒子の数。
  ptcl: 送る粒子へのポインタの配列。
  buf: 送信バッファーの先頭アドレス。
  packed_size: ユーザーがバッファーへ書き込むサイズ。単位はバイト。
  max_buf_size: 送信バッファーの書き込み可能な領域のサイズ。単位はバイト。

- **返値**

  PS::S32 型。packed_size が max_buf_size を超えた場合は-1 を返す。それ以外の場合は 0 を返す。

- **機能**

  LET 交換時に送信する粒子をシリアライズし、送信バッファーに書き込む。

7.4.4.5.2 EPJ::unPack

```
class EPJ {
public:
    static void unPack(const PS::S32 n_ptcl, EPJ ptcl[], const char *buf);
};
```

- **引数**

  n_ptcl: LET 交換時に受け取る粒子の数。
  ptcl: 受け取る粒子の配列。
  buf: 受信バッファーの先頭アドレス。

- **返値**

  なし。

- **機能**

  LET 交換時に受信する粒子をデシリアライズし、粒子配列に書き込む。詳しくはセクション 9.1.4.2.3 を参照。デシリアライズに失敗した場合は PS::Abort() が呼ばれプログラムは終了する。

7.5 Moment クラス

7.5.1 概要

Moment クラスは近い粒子同士でまとまった複数の粒子のモーメント情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。モーメント情報の例としては、複数粒子の単極子や双極子、さらにこれら粒子の持つ最大の大きさなど様々なものが考えられる。このクラスは、EssentialParticleJ クラスから SuperParticleJ クラスを作るための中間変数のような役割を果す。従って、このクラスが持つメンバ関数は、EssentialParticleJ クラスから情報を読み出してモーメントを計算するメンバ関数、少ない数の粒子のモーメントからそれらの粒子を含むより多くの粒子のモーメントを計算するメンバ関数などがある。

このようなモーメント情報にはある程度決っているものが多いので、それらについては FDPS 側で用意した。これら既存のクラスについてまず記述する。その後にユーザーがモーメントクラスを自作する際に必ず必要なメンバ関数、場合によっては必要になるメンバ関数について記述する。

7.5.2 既存のクラス

7.5.2.1 概要

FDPS はいくつかの Moment クラスを用意している。これらは相互作用ツリークラスで特定の PS::SEARCH_MODE 型を選んだ場合に有効である。以下、各 PS::SEARCH_MODE 型において選ぶことのできる Moment 型を記述する。PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY については Moment クラスを意識してコーディングする必要がないので、これらについては記述しない。

7.5.2.2 PS::SEARCH_MODE_LONG

7.5.2.2.1 PS::MomentMonopole

単極子までを情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

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

- クラス名 PS::MomentMonopole
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心
- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.2.2 PS::MomentQuadrupole

単極子と四重極子を情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の重心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentQuadrupole {
    public:
        F64    mass;
        F64vec pos;
        F64mat quad;
    };
}
```

- クラス名 PS::MomentQuadrupole
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量

  pos: 近傍でまとめた粒子の重心

  quad: 近傍でまとめた粒子の四重極子

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.2.3 PS::MomentMonopoleGeometricCenter

単極子までを情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の幾何中心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentMonopoleGeometricCenter {
    public:
        F64    charge;
        F64vec pos;
    };
}
```

- クラス名 PS::MomentMonopoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の幾何中心

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.2.4 PS::MomentDipoleGeometricCenter

双極子までを情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の幾何中心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentDipoleGeometricCenter {
    public:
        F64    charge;
        F64vec pos;
        F64vec dipole;
    };
}
```

- クラス名 PS::MomentDipoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の幾何中心

  dipole: 粒子の質量または電荷の双極子

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.2.5 PS::MomentQuadrupoleGeometricCenter

四重極子までを情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の幾何中心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentQuadrupoleGeometricCenter {
    public:
        F64    charge;
        F64vec pos;
        F64vec dipole;
        F64mat quadrupole;
    };
}
```

- クラス名 PS::MomentQuadrupoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の幾何中心

dipole: 粒子の質量または電荷の双極子

quadrupole: 粒子の質量または電荷の四重極子

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.3 PS::SEARCH_MODE_LONG_SCATTER

7.5.2.3.1 PS::MomentMonopoleScatter

単極子までを情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentMonopoleScatter {
    public:
        F64    mass;
        F64vec pos;
        F64ort vertex_out_;
        F64ort vertex_in_;
    };
}
```

- クラス名 PS::MomentMonopoleScatter

- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

  vertex_out_: 粒子を EssentialParticleJ::getRSearch の返り値を半径とする球とみなしたとき、近傍でまとめた粒子すべてを包含する最小の直方体の座標情報

  vertex_in_: 近傍でまとめた粒子すべての座標を包含する最小の直方体の座標情報

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge, EssentialParticleJ::getPos, EssentialParticleJ::getRSearch を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置、粒子の力の到達距離を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.3.2 PS::MomentQuadrupoleScatter

単極子と四重極子を情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentQuadrupoleScatter {
    public:
        F64    mass;
        F64vec pos;
        F64mat quad;
        F64ort vertex_out_;
        F64ort vertex_in_;
    };
}
```

- クラス名 PS::MomentQuadrupoleScatter
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

  quad: 近傍でまとめた粒子の四重極子

  vertex_out_: 粒子を EssentialParticleJ::getRSearch の返り値を半径とする球とみなしたとき、近傍でまとめた粒子すべてを包含する最小の直方体の座標情報

  vertex_in_: 近傍でまとめた粒子すべての座標を包含する最小の直方体の座標情報

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge, EssentialParticleJ::getPos, EssentialParticleJ::getRSearch を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置、粒子の力の到達距離を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.4 PS::SEARCH_MODE_LONG_SYMMETRY

7.5.2.4.1 PS::MomentMonopoleSymmetry

単極子までを情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentMonopoleSymmetry {
    public:
        F64    mass;
        F64vec pos;
        F64ort vertex_out_;
        F64ort vertex_in_;
    };
}
```

- クラス名 PS::MomentMonopoleSymmetry
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

  vertex_out_: 粒子を EssentialParticleJ::getRSearch の返り値を半径とする球とみなしたとき、近傍でまとめた粒子すべてを包含する最小の直方体の座標情報

  vertex_in_: 近傍でまとめた粒子すべての座標を包含する最小の直方体の座標情報

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge, EssentialParticleJ::getPos, EssentialParticleJ::getRSearch を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置、粒子の力の到達距離を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.4.2 PS::MomentQuadrupoleSymmetry

単極子と四重極子を情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentQuadrupoleSymmetry {
    public:
        F64    mass;
        F64vec pos;
        F64mat quad;
        F64ort vertex_out_;
        F64ort vertex_in_;
    };
}
```

- クラス名 PS::MomentQuadrupoleSymmetry
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

  quad: 近傍でまとめた粒子の四重極子

  vertex_out_: 粒子を EssentialParticleJ::getRSearch の返り値を半径とする球とみなしたとき、近傍でまとめた粒子すべてを包含する最小の直方体の座標情報

  vertex_in_: 近傍でまとめた粒子すべての座標を包含する最小の直方体の座標情報

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge, EssentialParticleJ::getPos, EssentialParticleJ::getRSearch を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置、粒子の力の到達距離を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.5 PS::SEARCH_MODE_LONG_CUTOFF

7.5.2.5.1 PS::MomentMonopoleCutoff

単極子までを情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

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

- クラス名 PS::MomentMonopoleCutoff
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心
- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge, EssentialParticleJ::getPos, EssentialParticleJ::getRSearch を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置、粒子の力の到達距離を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.3 必要なメンバ関数

7.5.3.1 概要

以下では Moment クラスを定義する際に、必要なメンバ関数を記述する。このとき Moment クラスのクラス名を Mom とする。これは変更自由である。

7.5.3.2 コンストラクタ

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

- **引数**

  なし
- **返値**

  なし

- **機能**

  Mom クラスのオブジェクトの初期化をする。

7.5.3.3 Mom::init

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

- **引数**

  なし

- **返値**

  なし

- **機能**

  Mom クラスのオブジェクトの初期化をする。

7.5.3.4 Mom::getPos

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

- **引数**

  なし

- **返値**

  PS::F32vec または PS::F64vec 型。Mom クラスのメンバ変数 pos。

- **機能**

  Mom クラスのメンバ変数 pos を返す。

7.5.3.5 Mom::getCharge

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

- **引数**

  なし

- **返値**

  PS::F32 または PS::F64 型。Mom クラスのメンバ変数 mass。

- **機能**

  Mom クラスのメンバ変数 mass を返す。

7.5.3.6 Mom::getVertexIn

```
class Mom {
public:
    PS::F32ort getVertexIn() const;
};
```

- **引数**

  なし

- **返値**

  PS::F32ort または PS::F64ort 型。

- **機能**

  Mom クラスに対応する (複数の) 粒子すべてを包含する直方体の

7.5.3.7 Mom::accumulateAtLeaf

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

- **引数**

  epj: 入力。const Tepj &型。Tepj のオブジェクト。

- **返値**

  なし。

- **機能**

  EssentialParticleJ クラスのオブジェクトからモーメントを計算する。

7.5.3.8 Mom::accumulate

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

- **引数**

  mom: 入力。const Mom &型。Mom クラスのオブジェクト。

- **返値**

  なし。

- **機能**

  Mom クラスのオブジェクトからさらに Mom クラスの情報を計算する。

7.5.3.9 Mom::set

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

- **引数**

  なし

- **返値**

  なし

- **機能**

  上記のメンバ関数 Mom::accumulateAtLeaf, Mom::accumulate ではモーメントの位置情報の規格化ができていない場合ので、ここで規格化する。

7.5.3.10 Mom::accumulateAtLeaf2

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

- **引数**

  epj: 入力。const Tepj &型。Tepj のオブジェクト。

- **返値**

  なし。

- **機能**

  EssentialParticleJ クラスのオブジェクトからモーメントを計算する。

7.5.3.11 Mom::accumulate2

```
class Mom {
public:
    void accumulate2(const Mom & mom);
};
```

- **引数**

  mom: 入力。const Mom &型。Mom クラスのオブジェクト。

- **返値**

  なし。

- **機能**

  Mom クラスのオブジェクトからさらに Mom クラスの情報を計算する。

7.5.4 場合によっては必要なメンバ関数

7.5.4.1 概要

以下では Moment クラスを定義する際に、場合によっては必要なメンバ関数を記述する。このとき Moment クラスのクラス名を Mom とする。これは変更自由である。

7.5.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG_CUTOFF、PS::SEARCH_MODE_LONG_SCATTER、PS::SEARCH_MODE_LONG_SYMMETRY のいずれかを用いる場合

7.5.4.3 Mom::getVertexIn

```
class Mom {
public:
    F64ort getVertexIn();
};
```

- **引数**

  なし。

- **返値**

  PS::F32ort または PS::F64ort 型。

- **機能**

  Mom クラスに対応する粒子すべてを包含する最小の直方体 (2 次元の場合には直方体) の座標をオルソトープ型で返す。

- **備考**

  上記の直方体の座標は、各ツリーセルで正しく計算される必要がある。そのためには、粒子情報からツリー構造の一番深いレベルのツリーセルのモーメント情報を計算するためのメンバ関数 `accumulateAtLeaf` と、複数のツリーセルからその親セルのモーメント情報を計算するためのメンバ関数 `accumulate` に直方体の座標計算を実装しなければならない。以下に実装例を示す。

```
class Mom {
public:
    F64ort vertex_in_; // 直方体の座標を格納するメンバ変数
    F64ort getVertexIn() const { return vertex_in_; }
    template<class Tepj>
    void accumulateAtLeaf(const Tepj & epj){
         // 他の演算は省略
        (this->vertex_in_).merge(epj.getPos());
    }
    void accumulate(const Mom & mom){
        // 他の演算は省略
        (this->vertex_in_).merge(mom.vertex_in_);
    }
};
```

上記実装例では、オルソトープ型のメンバ関数 merge を使用している。なお、メンバ変数 vertex_in_の名称は自由に変更可能である。

7.5.4.4 Mom::getVertexOut

```
class Mom {
public:
    F64ort getVertexOut();
};
```

- **引数**

  なし。

- **返値**

  PS::F32ort または PS::F64ort 型。

- **機能**

  Mom クラスに対応する各粒子を、中心を粒子座標、半径をその粒子の getRSearch() の返り値とする球 (2 次元の場合には円) に置き換えたとき、それらすべてを包含する最小の直方体 (2 次元の場合には直方体) の座標をオルソトープ型で返す。

- **備考**

  メンバ関数 getVertexIn のときと同様、上記の直方体の座標は、各ツリーセルで正しく計算される必要がある。そのためには、メンバ関数 accumulateAtLeaf とメンバ関数 accumulate に直方体の座標計算を実装しなければならない。以下に実装例を示す。

```
class Mom {
public:
    F64ort vertex_out_; // 直方体の座標を格納するメンバ変数
    F64ort getVertexOut() const { return vertex_out_; }
    template<class Tepj>
    void accumulateAtLeaf(const Tepj & epj){
         // 他の演算は省略
         (this->vertex_out_).merge(epj.getPos(), epj.getRSearch());
    }
    void accumulate(const Mom & mom){
        // 他の演算は省略
        (this->vertex_out_).merge(mom.vertex_out_);
    }
};
```

上記実装例では、オルソトープ型のメンバ関数 merge を使用している。なお、メンバ変数 vertex_out_の名称は自由に変更可能である。

7.6 SuperParticleJ クラス

7.6.1 概要

SuperParticleJ クラスは近い粒子同士でまとまった複数の粒子を代表してまとめた超粒子の情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。このクラスが必要となるのは PS::SEARCH_MODE に以下のいずれかを選択した場合だけである。

- PS::SEARCH_MODE_LONG
- PS::SEARCH_MODE_LONG_SCATTER
- PS::SEARCH_MODE_LONG_SYMMETRY
- PS::SEARCH_MODE_LONG_CUTOFF

このクラスのメンバ関数には、超粒子の位置情報を FDPS 側とやりとりするメンバ関数がある。また、超粒子の情報と Moment クラスの情報は対になるものである。従って、このクラスのメンバ関数には、Moment クラスからこのクラスへ情報を変換 (またはその逆変換) するメンバ関数がある。

SuperParticleJ クラスも Moment クラス同様、ある程度決っているものが多いので、それらについては FDPS 側で用意した。以下、既存のクラス、SuperParticleJ クラスを作るときに必要なメンバ関数、場合によっては必要なメンバ関数について記述する。

7.6.2 既存のクラス

FDPS はいくつかの SuperParticleJ クラスを用意している。以下、各 PS::SEARCH_MODE に対し選ぶことのできるクラスについて記述する。PS::SEARCH_MODE_LONG、PS::SEARCH_MODE_LO PS::SEARCH_MODE_LONG_SYMMETRY、PS::SEARCH_MODE_LONG_CUTOFF の場合をこの順序で記述する。その他の PS::SEARCH_MODE では超粒子を必要としない。

7.6.2.1 PS::SEARCH_MODE_LONG

7.6.2.1.1 PS::SPJMonopole

単極子までの情報を持つ Moment クラス PS::MomentMonopole と対になる SuperParticleJ クラス。以下、このクラスの概要を記述する。

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

- クラス名 PS::SPJMonopole
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心
- 使用条件

  Moment クラスである PS::MomentMonopole クラスの使用条件に準ずる。

7.6.2.1.2 PS::SPJQuadrupole

単極子と四重極子を情報を持つ Moment クラス PS::MomentQuadrupole と対になる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJQuadrupole {
    public:
        F64    mass;
        F64vec pos;
        F64mat quad;
    };
}
```

- クラス名 PS::SPJQuadrupole
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

  quad: 近傍でまとめた粒子の四重極子
- 使用条件

  Moment クラスである PS::MomentQuadrupole クラスの使用条件に準ずる。

7.6.2.1.3 PS::SPJMonopoleGeometricCenter

単極子までを情報として持つ (ただしモーメント計算の際の座標系の中心は粒子の幾何中心)Moment クラス PS::MomentMonopoleGeometricCenter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJMonopoleGeometricCenter {
    public:
        F64    charge;
        F64vec pos;
    };
}
```

- クラス名 PS::SPJMonopoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の幾何中心
- 使用条件

  PS::MomentMonopoleGeometricCenter の使用条件に準ずる。

7.6.2.1.4 PS::SPJDipoleGeometricCenter

双極子までを情報として持つ (ただしモーメント計算の際の座標系の中心は粒子の幾何中心)Moment クラス PS::MomentDipoleGeometricCenter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJDipoleGeometricCenter {
    public:
        F64    charge;
        F64vec pos;
        F64vec dipole;
    };
}
```

- クラス名 PS::SPJDipoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の幾何中心

  dipole: 粒子の質量または電荷の双極子
- 使用条件

  PS::MomentDipoleGeometricCenter の使用条件に準ずる。

7.6.2.1.5 PS::SPJQuadrupoleGeometricCenter

四重極子までを情報として持つ (ただしモーメント計算の際の座標系の中心は粒子の幾何中心)Moment クラス PS::MomentQuadrupoleGeometricCenter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJQuadrupoleGeometricCenter {
    public:
        F64    charge;
        F64vec pos;
        F64vec dipole;
        F64mat quadrupole;
    };
}
```

- クラス名 PS::SPJQuadrupoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の幾何中心

  dipole: 粒子の質量または電荷の双極子

  quadrupole: 粒子の質量または電荷の四重極子
- 使用条件

  PS::MomentQuadrupoleGeometricCenter の使用条件に準ずる。

7.6.2.2 PS::SEARCH_MODE_LONG_SCATTER

7.6.2.2.1 PS::SPJMonopoleScatter

単極子までを情報として持つ Moment クラス PS::MomentMonopoleScatter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

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

- クラス名 PS::SPJMonopoleScatter
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心
- 使用条件

  PS::MomentMonopoleScatter の使用条件に準ずる。

7.6.2.2.2 PS::SPJQuadrupoleScatter

単極子と四重極子を情報として持つ Moment クラス PS::MomentQuadrupoleScatter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJQuadrupolepoleScatter {
    public:
        F64    mass;
        F64vec pos;
        F64mat quad;
    };
}
```

- クラス名 PS::SPJQuadrupoleScatter
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

  quad: 近傍でまとめた粒子の四重極子
- 使用条件

  PS::MomentQuadrupoleScatter の使用条件に準ずる。

7.6.2.3 PS::SEARCH_MODE_LONG_SYMMETRY

7.6.2.3.1 PS::SPJMonopoleSymmetry

単極子までを情報として持つ Moment クラス PS::MomentMonopoleSymmetry と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

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

- クラス名 PS::SPJMonopoleSymmetry
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

- 使用条件

  PS::MomentMonopoleSymmetry の使用条件に準ずる。

7.6.2.3.2 PS::SPJQuadrupoleSymmetry

単極子と四重極子を情報として持つ Moment クラス PS::MomentQuadrupoleSymmetry と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJQuadrupolepoleSymmetry {
    public:
        F64    mass;
        F64vec pos;
        F64mat quad;
    };
}
```

- クラス名 PS::SPJQuadrupoleSymmetry
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

  quad: 近傍でまとめた粒子の四重極子
- 使用条件

  PS::MomentQuadrupoleSymmetry の使用条件に準ずる。

7.6.2.4 PS::SEARCH_MODE_LONG_CUTOFF

7.6.2.4.1 PS::SPJMonopoleCutoff

単極子までを情報として持つ Moment クラス PS::MomentMonopoleCutoff と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

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

- クラス名 PS::SPJMonopoleCutoff
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心
- 使用条件

  PS::MomentMonopoleCutoff の使用条件に準ずる。

7.6.3 必要なメンバ関数

7.6.3.1 概要

以下では SuperParticleJ クラスを作る際に必要なメンバ関数を記述する。このとき SuperParticleJ クラスのクラス名を SPJ とする。これは変更自由である。

7.6.3.2 SPJ::getPos

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

- **引数**

  なし
- **返値**

  PS::F32vec 型または PS::F64vec 型。SPJ クラスの位置情報を保持したメンバ変数。
- **機能**

  SPJ クラスの位置情報を保持したメンバ変数を返す。

7.6.3.3 SPJ::setPos

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

- **引数**

  pos_new: 入力。const PS::F32vec または const PS::F64vec 型。FDPS 側で修正した粒子の位置情報。

- **返値**

  なし。

- **機能**

  FDPS が修正した粒子の位置情報を SPJ クラスの位置情報に書き込む。

7.6.3.4 SPJ::copyFromMoment

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

- **引数**

  mom: 入力。const Mom &型。Mom にはユーザー定義または FDPS 側で用意した Moment クラスが入る。

- **返値**

  なし。

- **機能**

  Mom クラスの情報を SPJ クラスにコピーする。

7.6.3.5 SPJ::convertToMoment

```
class Mom;
class SPJ {
public:
    Mom convertToMoment() const;
};
```

- **引数**

  なし

- **返値**

  Mom 型。Mom クラスのコンストラクタ。

- **機能**

  超粒子をモーメントに変換し、その変換したものを Mom クラスのコンストラクタを返す。

7.6.3.6 SPJ::clear

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

- **引数**

  なし

- **返値**

  なし

- **機能**

  SPJ クラスのオブジェクトの情報をクリアする。

7.6.4 場合によっては必要なメンバ関数

7.6.4.1 概要

本節では、場合によっては必要なメンバ関数について記述する。

7.6.4.2 LET 交換時に粒子データをシリアライズして送る場合

LET 交換時に粒子データをシリアライズして送る場合には、メンバ関数に SPJ::pack と SPJ::unpack を用意する必要がある。以下にそれぞれの規定を記述する。

7.6.4.2.1　SPJ::pack

```
class SPJ {
public:
    static PS::S32 pack(const PS::S32 n_ptcl, const SPJ *ptcl[], char *buf,
                        size_t & packed_size, const size_t max_buf_size);
};
```

- **引数**

  n_ptcl: LET 交換時に送る粒子の数。

  ptcl: 送る粒子へのポインタの配列。

  buf: 送信バッファーの先頭アドレス。

  packed_size: ユーザーがバッファーへ書き込むサイズ。単位はバイト。

  max_buf_size: 送信バッファーの書き込み可能な領域のサイズ。単位はバイト。

- **返値**

  PS::S32 型。packed_size が max_buf_size を超えた場合は-1 を返す。それ以外の場合は 0 を返す。

- **機能**

  LET 交換時に送信する粒子をシリアライズし、送信バッファーに書き込む。

7.6.4.2.2　SPJ::unPack

```
class SPJ {
public:
    static void unPack(const PS::S32 n_ptcl, SPJ[], const char *buf);
};
```

- **引数**

  n_ptcl: LET 交換時に受け取る粒子の数。

  ptcl: 受け取る粒子の配列。

  buf: 受信バッファーの先頭アドレス。

- **返値**

  なし。

- **機能**

  LET 交換時に受信する粒子をデシリアライズし、粒子配列に書き込む。

7.7 Force クラス

7.7.1 概要

Force クラスは相互作用の結果を保持するクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。以下、この節の前提、常に必要なメンバ関数について記述する。

7.7.2 前提

この節で用いる例として Force クラスのクラス名を Result とする。このクラス名は変更自由である。

7.7.3 必要なメンバ関数

常に必要なメンバ関数は Result::clear である。この関数は相互作用の計算結果を初期化する。以下、Result::clear について記述する。

7.7.3.1 Result::clear

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

- **引数**

  なし

- **返値**

  なし。

- **機能**

  Result クラスのメンバ変数を初期化する。

7.8 ヘッダクラス

7.8.1 概要

ヘッダクラスは入出力ファイルのヘッダの形式を決めるクラスである。ヘッダクラスは FDPS が提供する粒子群クラスのファイル入出力 API を使用し、かつ入出力ファイルにヘッダを含ませたい場合に必要となるクラスである。粒子群クラスのファイル入出力 API とは、ParticleSystem::readParticleAscii, ParticleSystem::writeParticleAscii, ParticleSystem::readParticleBinary,

ParticleSystem::writeParticleBinary である。以下、この節における前提と、これらの API を使用する際に必要となるメンバ関数とその記述の規定を述べる。この節において、常に必要なメンバ関数というものは存在しない。

7.8.2 前提

この節では、ヘッダクラスのクラス名を Hdr とする。このクラス名は変更可能である。

7.8.3 場合によっては必要なメンバ関数

7.8.3.1 Hdr::readAscii

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

- **引数**

  fp: 入力。FILE *型。粒子データの入力ファイルを指すファイルポインタ。

- **返値**

  PS::S32 型。粒子数の情報を返す。ヘッダに粒子数の情報がない場合は-1 を返す。

- **機能**

  粒子データの入力ファイルからヘッダ情報を読みこむ。

7.8.3.2 Hdr::writeAscii

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

- **引数**

  fp: 入力。FILE *型。粒子データの出力ファイルを指すファイルポインタ。

- **返値**

  なし。

- **機能**

  粒子データの出力ファイルへヘッダ情報を書き込む。

7.8.3.3 Hdr::readBinary

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

- **引数**

  fp: 入力。FILE *型。粒子データの入力ファイルを指すファイルポインタ。

- **返値**

  PS::S32 型。粒子数の情報を返す。ヘッダに粒子数の情報がない場合は-1 を返す。

- **機能**

  粒子データの入力ファイルからヘッダ情報を読みこむ。

7.8.3.4 Hdr::writeBinary

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

- **引数**

  fp: 入力。FILE *型。粒子データの出力ファイルを指すファイルポインタ。

- **返値**

  なし。

- **機能**

  粒子データの出力ファイルへヘッダ情報を書き込む。

7.9 関数オブジェクト calcForceEpEp

7.9.1 概要

関数オブジェクト calcForceEpEp は粒子同士の相互作用を記述するものであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。以下、これの書き方の規定を記述する。

7.9.2 前提

ここで示すのは重力 N 体シミュレーションの粒子間相互作用の記述の仕方である。関数オブジェクト calcForceEpEp の名前は gravityEpEp とする。これは変更自由である。また、EssentialParitlceI クラスのクラス名を EPI, EssentialParitlceJ クラスのクラス名を EPJ, Force クラスのクラス名を Result とする。

7.9.3 gravityEpEp::operator ()

ソースコード 21: calcForceEpEp

```
1  class Result;
2  class EPI;
3  class EPJ;
4  struct gravityEpEp {
5      void operator () (const EPI *epi,
6                        const PS::S32 ni,
7                        const EPJ *epj,
8                        const PS::S32 nj,
9                        Result *result);
10 };
```

- **引数**

  epi: 入力。const EPI *型または EPI *型。i 粒子情報を持つ配列。

  ni: 入力。const PS::S32 型または PS::S32 型。i 粒子数。

  epj: 入力。const EPJ *型または EPJ *型。j 粒子情報を持つ配列。

  nj: 入力。const PS::S32 型または PS::S32 型。j 粒子数。

  result: 出力。Result *型。i 粒子の相互作用結果を返す配列。

- **返値**

  なし。

- **機能**

  j 粒子から i 粒子への作用を計算する。

7.10 関数オブジェクト calcForceSpEp

7.10.1 概要

関数オブジェクト calcForceSpEp は超粒子から粒子への作用を記述するものであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。以下、これの書き方の規定を記述する。

7.10.2 前提

ここで示すのは重力 N 体シミュレーションにおける超粒子から粒子への作用の記述の仕方である。超粒子は単極子までの情報で作られているものとする。関数オブジェクト calcForceSpEp の名前は gravitySpEp とする。これは変更自由である。また、EssentialParitlceI クラスのクラス名を EPI, SuperParitlceJ クラスのクラス名を SPJ, Force クラスのクラス名を Result とする。

7.10.3 gravitySpEp::operator ()

ソースコード 22: calcForceSpEp

```
1  class Result;
2  class EPI;
3  class SPJ;
4  struct gravitySpEp {
5      void operator () (const EPI *epi,
6                        const PS::S32 ni,
7                        const SPJ *spj,
8                        const PS::S32 nj,
9                        Result *result);
10 };
```

- **引数**

  epi: 入力。const EPI *型または EPI *型。i 粒子情報を持つ配列。

  ni: 入力。const PS::S32 型または PS::S32 型。i 粒子数。

  spj: 入力。const SPJ *型または SPJ *型。超粒子情報を持つ配列。

  nj: 入力。const PS::S32 型または PS::S32 型。超粒子数。

  result: 出力。Result *型。i 粒子の相互作用結果を返す配列。

- **返値**

  なし。

- **機能**

  超粒子から i 粒子への作用を計算する。

7.11 関数オブジェクト calcForceDispatch

7.11.1 概要

関数 calcForceDispatch は関数 calcForceRetrieve と合わせて粒子同士の相互作用を記述するものであり、calcForceSpEp や calcForceEpEp の代わりに相互作用の定義 (節 2.3 の手順 0) に使うことができる。calcForceSpEp や calcForceEpEp との違いは、calcForceDispatch は複数の相互作用リストと i 粒子リストを受け取ることである。これにより、GPGPU 等のアクセラレータを起動する回数を削減し、実行効率を向上させる。以下、これの書き方の規定を記述する。関数 calcForceDispatch の名前は GravityDispatch とする。これは変更自由である。また、EssentialParitlceI クラスのクラス名を EPI, EssentialParitlceJ クラスのクラス名を EPJ, SuperParitlceJ クラスのクラス名を SPJ とする。

7.11.2 短距離力の場合

ソースコード 23: calcForceDispatch

```
1 class EPI;
2 class EPJ;
3 PS::S32 HydroforceDispatch(const PS::S32  tag,
4                             const PS::S32  nwalk,
5                             const EPI**      epi,
6                             const PS::S32*  ni,
7                             const EPJ**      epj,
8                             const PS::S32*  nj_ep;
9 };
```

- **引数**

  tag: 入力。const PS::S32 型。tag の番号。発行される tag の番号は
  0 から関数 PS::TreeForForce::calcForceAllandWriteBackMultiWalk() の
  第三引数として設定された値から 1 引いた数までである。
  tag の番号は calcForceRetrieve() で設定する tag の番号と対応させる必要がある。

  nwalk: 入力。const PS::S32 型。walk の数。walk の数の最大値は
  PS::TreeForForce::calcForceAllandWriteBackMultiWalk() の第六引数の値である。

  epi: 入力。const EPI** 型。i 粒子情報を持つポインタのポインタ。

  ni: 入力。const PS::S32*型。i 粒子数のポインタ。

  epj: 入力。const EPJ** 型。j 粒子情報を持つポインタのポインタ。

  nj_ep: 入力。const PS::S32* 型。j 粒子数のポインタ。

- **返値**

  PS::S32 型。ユーザーは正常に実行された場合は 0 を、エラーが起こった場合は 0 以外の値を返すようにする。

- **機能**

  epi,epj の情報をアクセラレータに送り、相互作用カーネルを発行する。

7.11.3 長距離力の場合

ソースコード 24: calcForceDispatch

```
1  class EPI;
2  class EPJ;
3  class SPJ;
4  PS::S32 GravityDispatch(const PS::S32   tag,
5                          const PS::S32   nwalk,
6                          const EPI**     epi,
7                          const PS::S32*  ni,
8                          const EPJ**     epj,
9                          const PS::S32*  nj_ep,
10                         const SPJ**     spj,
11                         const PS::S32*  nj_sp);
12 };
```

- **引数**

  tag: 入力。const PS::S32 型。tag の番号。発行される tag の番号は 0 から関数 PS::TreeForForce::calcFor
  の第三引数として設定された値から 1 引いた数までである。tag の番号は calcForceRe-
  trieve() で設定する tag の番号と対応させる必要がある。

  nwalk: 入力。const PS::S32 型。walk の数。walk の数の最大値は PS::TreeForForce::calcForceAllandWr
  の第六引数の値である。

  epi: 入力。const EPI** 型。i 粒子情報を持つ配列の配列。

  ni: 入力。const PS::S32*型。i 粒子数の配列。

  epj: 入力。const EPJ** 型。j 粒子情報を持つ配列の配列。

  nj_ep: 入力。const PS::S32* 型。j 粒子数の配列。

  spj: 入力。const SPJ** 型。j 粒子情報を持つ配列の配列。

  nj_sp: 入力。const PS::S32* 型。j 粒子数の配列。

- **返値**

  PS::S32 型。ユーザーは正常に実行された場合は 0 を、エラーが起こった場合は 0 以外の値を返すようにする。

- **機能**

  epi,epj,spj の情報をアクセラレータに送り、相互作用カーネルを発行する。

7.12 関数オブジェクト calcForceRetrieve

7.12.1 概要

関数 calcForceRetrieve は関数 calcForceDispatch で行った相互作用の結果を回収する関数である。以下、これの書き方の規定を記述する。関数 calcForceRetrieve の名前は GravityRetrieve とする。これは変更自由である。また、Force クラスのクラス名を Result とする。

ソースコード 25: calcForceDispatch

```
1 class EPI;
2 class EPJ;
3 class Result;
4 PS::S32 GravityRetrieve(const PS::S32  tag,
5                          const PS::S32  nwalk,
6                          const PS::S32  ni     [],
7                          Result         result [][]);
8 };
```

- **引数**

  tag: 入力。const PS::S32 型。tag の番号。対応する calcForceDispatch の tag 番号と一致させる必要がある。

  nwalk: 入力。const PS::S32 型。walk の数。対応する calcForceDispatch に与えた nwalk の値と一致させる必要がある。

  ni: 入力。const PS::S32*型。i 粒子数の配列。

  result: 出力。Result *型。i 粒子の相互作用結果を返す配列の配列。

- **返値**

  PS::S32 型。ユーザーは正常に実行された場合は 0 を、エラーが起こった場合は 0 以外の値を返すようにする。

- **機能**

  同じ tag 番号を持つ関数 calcForceDispatch で行った相互作用の結果を回収する。

8 プログラムの開始と終了

8.1 概要

プログラムの開始と終了に必要な API などを記述する。

8.2 API

8.2.1 PS::Initialize

プログラムの開始を行うには以下の API を呼び出す必要がある。

```
void PS::Initialize
               (PS::S32 & argc,
                char ** & argv,
                const PS::S64 mpool_size=100000000);
```

- **引数**

  argc: 入力。PS::S32 型。コマンドライン引数の総数。

  argv: 入力。char ** &型。コマンドライン引数の文字列を指すポインタのポインタ。

  mpool_size: 入力。const PS::S64 型。FDPS が内部で使用するメモリプールのサイズ。単位はバイト。デフォルト値は 10^8 バイト。

- **返値**

  なし

- **機能**

  FDPS の初期化を行う。FDPS の API のうち最初に呼び出さなければならない。内部では MPI::Init を呼び出すため、引数 argc と argv が変っている可能性がある。

8.2.2 PS::Finalize

プログラムの終了するには以下の API を呼び出す必要がある。

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

- **引数**

  なし

- **返値**

  なし

- **機能**

  FDPS の終了処理を行う。

8.2.3 PS::Abort

プログラムを異常終了させる場合には以下の API を呼び出す必要がある。

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

- **引数**

  err: PS::S32 型。プログラムの終了ステータス。デフォルト-1。

- **返値**

  なし

- **機能**

  FDPS の異常終了処理を行う。引数はプログラムの終了ステータスである。この引数は、MPI を使用していない場合は exit 関数に渡され、MPI を使用している場合は MPI の Abort 関数に渡される。

8.2.4 PS::DisplayInfo

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

- **引数**

  なし

- **返値**

  なし

- **機能**

  FDPS のライセンス情報などを表示する。

9　モジュール

本節では、FDPS のモジュールについて記述する。最初に FDPS の標準機能について、次に FDPS の拡張機能について記述する。

9.1　標準機能

9.1.1　概要

本節では、FDPS の標準機能について記述する。標準機能には４つのモジュールがあり、領域クラス、粒子群クラス、相互作用ツリークラス、通信用データクラスがある。この４つのクラスについて順に記述する。

9.1.2　領域クラス

本節では、領域クラスについて記述する。このクラスは領域情報の保持や領域の分割を行うモジュールである。まずオブジェクトの生成方法を記述し、その後 API を記述する。

9.1.2.1　オブジェクトの生成

領域クラスは以下のように宣言されている。

ソースコード 26: DomainInfo0

```
1 namespace ParticleSimulator {
2     class DomainInfo;
3 }
```

領域クラスのオブジェクトの生成は以下のように行う。ここでは dinfo というオブジェクトを生成している。

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

9.1.2.2　API

領域クラスには初期設定関連の API、領域分割関連の API がある。以下、各節に分けて記述する。

9.1.2.2.1　初期設定

初期設定関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 27: DomainInfo1

```
1  namespace ParticleSimulator {
2      class DomainInfo{
3      public:
4          DomainInfo();
5          void initialize(const F32 coef_ema=1.0);
6          void setNumberOfDomainMultiDimension(const S32 nx,
7                                               const S32 ny,
8                                               const S32 nz=1);
9          void setDomain(const S32 nx,
10                        const S32 ny,
11                        const S32 nz=1);
12         void setBoundaryCondition(enum BOUNDARY_CONDITION bc);
13         S32 getBoundaryCondition();
14         void setPosRootDomain(const F32vec & low,
15                               const F32vec & high);
16     };
17 }
```

9.1.2.2.1.1 コンストラクタ

コンストラクタ

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

- **引数**

  なし

- **返値**

  なし

- **機能**

  領域クラスのオブジェクトを生成する。

9.1.2.2.1.2 PS::DomainInfo::initialize

PS::DomainInfo::initialize

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

- **引数**

  coef_ema: 入力。 const PS::F32 型。指数移動平均の平滑化係数。デフォルト 1.0

- **返値**

  なし

- **機能**

  領域クラスのオブジェクトを初期化し、指数移動平均の平滑化係数を設定する。この係数の許される値は 0 から 1 である。それ以外の値を入れた場合はエラーメッセージを送出しプログラムは終了する。大きくなるほど、最新の粒子分布の情報が領域分割に反映されやすい。1 の場合、最新の粒子分布の情報のみ反映される。0 の場合、最初の粒子分布の情報のみ反映される。１度は呼ぶ必要がある。過去の粒子分布の情報を領域分割に反映する必要がある理由については、Ishiyama, Fukushige & Makino (2009, Publications of the Astronomical Society of Japan, 61, 1319) を参照のこと。

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);
```

- **引数**

  nx: 入力。 const PS::S32 型。x 軸方向のルートドメインの分割数。

  ny: 入力。 const PS::S32 型。y 軸方向のルートドメインの分割数。

  nz: 入力。 const PS::S32 型。z 軸方向のルートドメインの分割数。デフォルト 1。

- **返値**

  なし

- **機能**

  計算領域の分割する方法を設定する。nx, ny, nz はそれぞれ x 軸、y 軸、z 軸方向の計算領域の分割数である。呼ばなければ自動的に nx, ny, nz が決まる。呼んだ場合に入力する nx, ny, nz の総積が MPI プロセス数と等しくなければ、FDPS はエラーメッセージを送り、プログラムを止める。

9.1.2.2.1.4　PS::DomainInfo::setDomain

PS::DomainInfo::setDomain

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

- **引数**

  nx: 入力。 const PS::S32 型。x 軸方向のルートドメインの分割数。

  ny: 入力。 const PS::S32 型。y 軸方向のルートドメインの分割数。

  nz: 入力。 const PS::S32 型。z 軸方向のルートドメインの分割数。デフォルト 1。

- **返値**

  なし

- **機能**

  PS::DomainInfo::setNumberOfDomainMultiDimension の別名であり、全く同じ動作をする。

9.1.2.2.1.5　PS::DomainInfo::setBoundaryCondition

PS::DomainInfo::setBoundaryCondition

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

- **引数**

  bc: 入力。 列挙型。境界条件。

- **返値**

  なし

- **機能**

  境界条件の設定をする。許される入力は、6.8.2 で挙げた列挙型のみ (ただし BOUNDARY_CONDITION_SHEARING_BOX, BOUNDARY_CONDITION_USER_DEFINED は未実装)。呼ばない場合は、開放境界となる。

9.1.2.2.1.6　PS::DomainInfo::getBoundaryCondition

PS::DomainInfo::getBoundaryCondition

```
S32 PS::DomainInfo::getBoundaryCondition();
```

- **引数**

  なし

- **返値**

  PS::S32 型。

- **機能**

  設定されている境界条件に対応する整数を返す。取りうる値はPS::BOUNDARY_CONDITION 型の各列挙子に割り当てられている整数のいずれかである。

9.1.2.2.1.7　PS::DomainInfo::setPosRootDomain

PS::DomainInfo::setPosRootDomain

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

- **引数**

  low: 入力。 PS::F32vec 型。計算領域の下限 (閉境界)。

  high: 入力。 PS::F32vec 型。計算領域の上限 (開境界)。

- **返値**

  なし

- **機能**

  計算領域の下限と上限を設定する。開放境界条件の場合は呼ぶ必要はない。それ以外の境界条件の場合は、呼ばなくても動作するが、その結果が正しいことは保証できない。high の座標の各値は low の対応する座標よりも大きくなければならない。そうでない場合は、FDPS はエラーメッセージを送出し、ユーザープログラムを終了させる。

9.1.2.2.2　領域分割

領域分割関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 28: DomainInfo2

```
1  namespace ParticleSimulator {
2      class DomainInfo{
3      public:
4          template<class Tpsys>
5          void collectSampleParticle(Tpsys & psys,
6                                     const bool clear,
7                                     const F32 weight);
8          template<class Tpsys>
9          void collectSampleParticle(Tpsys & psys,
10                                    const bool clear);
11         template<class Tpsys>
12         void collectSampleParticle(Tpsys & psys);
13
14         void decomposeDomain();
15
16         template<class Tpsys>
17         void decomposeDomainAll(Tpsys & psys,
18                                 const F32 weight);
19         template<class Tpsys>
20         void decomposeDomainAll(Tpsys & psys);
21     };
22 }
```

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);
```

- **引数**

  psys: 入力。 ParticleSystem &型。領域分割のためのサンプル粒子を提供する粒子群クラス。

  clear: 入力。 bool 型。前にサンプルされた粒子情報をクリアするかどうかを決定するフラグ。true でクリアする。

weight: 入力。 const PS::F32 型。領域分割のためのサンプル粒子数を決めるためのウェイト。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys から粒子をサンプルする。clear によってこれより前にサンプルした粒子の情報を消すかどうか決める。weight によってその MPI プロセスからサンプルする粒子の量を調整する (weight が大きいほどサンプル粒子数が多い)。具体的には、プロセス i の weight を w_i、粒子群クラスの API
  `setAverateTargetNumberOfSampleParticlePerProcess`
  で設定されたプロセスあたりのサンプル粒子数を $n_{\rm smpl}$、プロセス数を $n_{\rm proc}$ とすると、プロセス i からは $n_{\rm smpl} n_{\rm proc} (w_i / \sum_k w_k)$ 個の粒子数がサンプリングされる。通常、weight にはそのプロセスでの計算時間を反映する量を指定する。

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

- **引数**

  psys: 入力。 ParticleSystem &型。領域分割のためのサンプル粒子を提供する粒子群クラス。

  clear: 入力。 bool 型。前にサンプルされた粒子情報をクリアするかどうかを決定するフラグ。true でクリアする。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys から粒子をサンプルする。clear によってこれより前にサンプルした粒子の情報を消すかどうか決める。

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

- **引数**

  psys: 入力。 ParticleSystem &型。領域分割のためのサンプル粒子を提供する粒子群クラス。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys から粒子をサンプルする。

9.1.2.2.2.2 PS::DomainInfo::decomposeDomain

PS::DomainInfo::decomposeDomain

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

- **引数**

  なし

- **返値**

  なし

- **機能**

  計算領域の分割を実行する。

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);
```

- **引数**

  psys: 入力。 ParticleSystem &型。領域分割のためのサンプル粒子を提供する粒子群クラス。

  weight: 入力。 const PS::F32 型。領域分割のためのサンプル粒子数を決めるためのウェイト。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys から粒子をサンプルし、続けてルートドメインの分割を行う。PS::DomainInfo::collectSampleParticle と PS::DomainInfo::decomposeDomain が行うことを一度に行う。weight の意味は PS::DomainInfo::collectSampleParticle と同じ。

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

- **引数**

  psys: 入力。 ParticleSystem &型。領域分割のためのサンプル粒子を提供する粒子群クラスのオブジェクト。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys から粒子をサンプルし、続けてルートドメインの分割を行う。PS::DomainInfo::collectSampleParticle と PS::DomainInfo::decomposeDomain が行うことを一度に行う。

9.1.2.2.3 時間計測

クラス内の時間計測関連の API の宣言は以下のようになっている。自クラスの主要なメソッドを呼び出すとそれにかかった時間をプライベートメンバの time_profile_の該当メンバに書き込む。メソッド clearTimeProfile() を呼ばない限り時間は足しあわされていく。

ソースコード 29: DomainInfo3

```
1 namespace ParticleSimulator {
2     class DomainInfo{
3     public:
4         TimeProfile getTimeProfile();
5         void clearTimeProfile();
6     };
7 }
```

9.1.2.2.3.1　PS::DomainInfo::getTimeProfile

PS::DomainInfo::getTimeProfile

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

- **引数**

  なし。

- **返値**

  PS::TimeProfile 型。

- **機能**

  メンバ関数 collectSampleParticle と decomposeDomain にかかった時間（ミリ秒単位）を TimeProfile 型のメンバ変数 collect_sample_particles_と decompose_domain_に格納する。

9.1.2.2.3.2　PS::DomainInfo::clearTimeProfile

PS::DomainInfo::clearTimeProfile

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

- **引数**

  なし。

- **返値**

  なし。

- **機能**

  領域情報クラスの TimeProfile 型のプライベートメンバ変数のメンバ変数 collect_sample_particles_と decompose_domain_の値を 0 クリアする。

9.1.2.2.4　情報取得

クラス内の情報取得関連の API の宣言は以下のようになっている。

ソースコード 30: DomainInfo3

```
1 namespace ParticleSimulator {
2     class DomainInfo{
3     public:
```

```
4           TimeProfile getTimeProfile();
5           void clearTimeProfile();
6           S64 getUsedMemorySize();
7       };
8   }
```

9.1.2.2.4.1 PS::DomainInfo::getUsedMemorySize

PS::DomainInfo::getUsedMemorySize

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

- **引数**

  なし。

- **返値**

  PS::S64。

- **機能**

  対象のオブジェクトが使用しているメモリー量を Byte 単位で返す。

9.1.3 粒子群クラス

本節では、粒子群クラスについて記述する。このクラスは粒子情報の保持や MPI プロセス間で粒子情報の交換を行うモジュールである。まずオブジェクトの生成方法を記述し、その後 API を記述する。

9.1.3.1 オブジェクトの生成

粒子群クラスは以下のように宣言されている。

ソースコード 31: ParticleSystem0

```
1 namespace ParticleSimulator {
2     template<class Tptcl>
3     class ParticleSystem;
4 }
```

テンプレート引数 Tptcl はユーザー定義の FullParticle クラスである。

粒子群クラスのオブジェクトの生成は以下のように行う。ここでは system というオブジェクトを生成している。

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

テンプレート引数 FP はユーザー定義の FullParticle クラスの 1 例である FP クラスである。

9.1.3.2 API

このモジュールには初期設定関連の API、オブジェクト情報取得設定関連の API、ファイル入出力関連の API、粒子交換関連の API がある。以下、各節に分けて記述する。

9.1.3.2.1 初期設定

初期設定関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 32: ParticleSystem1

```
1  namespace ParticleSimulator {
2      template<class Tptcl>
3      class ParticleSystem{
4      public:
5          ParticleSystem();
6          void initialize();
7          void setAverageTargetNumberOfSampleParticlePerProcess
8                           (const S32 & nsampleperprocess);
9      };
10 }
```

9.1.3.2.1.1 コンストラクタ

コンストラクタ

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

- **引数**

  なし

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクトを生成する。

9.1.3.2.1.2　PS::ParticleSystem::initialize

PS::ParticleSystem::initialize

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

- **引数**

  なし

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクトを初期化する。１度は呼ぶ必要がある。

9.1.3.2.1.3　PS::ParticleSystem::
setAverateTargetNumberOfSampleParticlePerProcess

PS::ParticleSystem::
setAverateTargetNumberOfSampleParticlePerProcess

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

- **引数**

  nsampleperprocess: 入力。const PS::S32 &型。１つの MPI プロセスでサンプルする粒子数目標。

- **返値**

  なし

- **機能**

  １つの MPI プロセスでサンプルする粒子数の目標を設定する。呼び出さなくてもよいが、呼び出さないとこの目標数が 30 となる。

9.1.3.2.2　情報取得

オブジェクト情報取得関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 33: ParticleSystem2

```
1  namespace ParticleSimulator {
2      template<class Tptcl>
3      class ParticleSystem{
4      public:
5          Tptcl & operator [] (const S32 id);
6          S32 getNumberOfParticleLocal() const;
7          S32 getNumberOfParticleGlobal() const;
8          S64 getUsedMemorySize() const;
9      };
10 }
```

9.1.3.2.2.1 PS::ParticleSystem::operator []

PS::ParticleSystem::operator []

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

- **引数**

  id: 入力。const PS::S32 型。粒子配列のインデックス。

- **返値**

  FullParticle &型。Tptcl 型のオブジェクト。

- **機能**

  id 番目の FullParticle 型のオブジェクトの参照を返す。

9.1.3.2.2.2 PS::ParticleSystem::getNumberOfParticleLocal

PS::ParticleSystem::getNumberOfParticleLocal

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

- **引数**

  なし

- **返値**

  PS::S32 型。１つの MPI プロセスの持つ粒子数。

- **機能**

  １つの MPI プロセスの持つ粒子数を返す。

9.1.3.2.2.3　PS::ParticleSystem::getNumberOfParticleGlobal

PS::ParticleSystem::getNumberOfParticleGlobal

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

- **引数**

  なし

- **返値**

  PS::S32 型。全 MPI プロセスの持つ粒子数。

- **機能**

  全 MPI プロセスの持つ粒子数を返す。

9.1.3.2.2.4　PS::DomainInfo::getUsedMemorySize

PS::DomainInfo::getUsedMemorySize

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

- **引数**

  なし。

- **返値**

  PS::S64。

- **機能**

  対象のオブジェクトが使用しているメモリー量を Byte 単位で返す。

9.1.3.2.3 ファイル入出力

ファイル入出力関連の API の宣言は以下のようになっている。

ソースコード 34: ParticleSystem3

```
1  namespace ParticleSimulator {
2      template<class Tptcl>
3      class ParticleSystem{
4      public:
5          template <class Theader>
6          void readParticleAscii(const char * const filename,
7                                 const char * const format,
8                                 Theader & header);
9          void readParticleAscii(const char * const filename,
10                                const char * const format);
11         template <class Theader>
12         void readParticleAscii(const char * const filename,
13                                Theader & header);
14         void readParticleAscii(const char * const filename);
15         template <class Theader>
16         void readParticleAscii(const char * const filename,
17                                const char * const format,
18                                Theader & header,
19                                void (Tptcl::*pFunc)(FILE*));
20         void readParticleAscii(const char * const filename,
21                                const char * const format,
22                                void (Tptcl::*pFunc)(FILE*));
23         template <class Theader>
24         void readParticleAscii(const char * const filename,
25                                Theader & header,
26                                void (Tptcl::*pFunc)(FILE*));
27         void readParticleAscii(const char * const filename,
28                                void (Tptcl::*pFunc)(FILE*) );
29
30         template <class Theader>
31         void readParticleBinary(const char * const filename,
32                                const char * const format,
33                                Theader & header);
34         void readParticleBinary(const char * const filename,
35                                const char * const format);
36         template <class Theader>
```

```
37          void readParticleBinary(const char * const filename,
38                                  Theader & header);
39          void readParticleBinary(const char * const filename);
40          template <class Theader>
41          void readParticleBinary(const char * const filename,
42                                  const char * const format,
43                                  Theader & header,
44                                  void (Tptcl::*pFunc)(FILE*));
45          void readParticleBinary(const char * const filename,
46                                  const char * const format,
47                                  void (Tptcl::*pFunc)(FILE*));
48          template <class Theader>
49          void readParticleBinary(const char * const filename,
50                                  Theader & header,
51                                  void (Tptcl::*pFunc)(FILE*));
52          void readParticleBinary(const char * const filename,
53                                  void (Tptcl::*pFunc)(FILE*) );
54
55          template <class Theader>
56          void writeParticleAscii(const char * const filename,
57                                  const char * const format,
58                                  const Theader & header);
59          void writeParticleAscii(const char * const filename,
60                                  const char * format);
61          template <class Theader>
62          void writeParticleAscii(const char * const filename,
63                                  const Theader & header);
64          void writeParticleAscii(const char * const filename);
65          template <class Theader>
66          void writeParticleAscii(const char * const filename,
67                                  const char * const format,
68                                  const Theader & header,
69                                  void (Tptcl::*pFunc)(FILE*)
                                        const);
70          void writeParticleAscii(const char * const filename,
71                                  const char * format,
72                                  void (Tptcl::*pFunc)(FILE*)
                                        const);
73          template <class Theader>
74          void writeParticleAscii(const char * const filename,
```

```
75                                  const Theader & header,
76                                  void (Tptcl::*pFunc)(FILE*)
                                          const);
77          void writeParticleAscii(const char * const filename,
78                                  void (Tptcl::*pFunc)(FILE*)
                                          const);
79
80          template <class Theader>
81          void writeParticleBinary(const char * const filename,
82                                  const char * const format,
83                                  const Theader & header);
84          void writeParticleBinary(const char * const filename,
85                                  const char * format);
86          template <class Theader>
87          void writeParticleBinary(const char * const filename,
88                                  const Theader & header);
89          void writeParticleBinary(const char * const filename);
90          template <class Theader>
91          void writeParticleBinary(const char * const filename,
92                                  const char * const format,
93                                  const Theader & header,
94                                  void (Tptcl::*pFunc)(FILE*)
                                          const);
95          void writeParticleBinary(const char * const filename,
96                                  const char * format,
97                                  void (Tptcl::*pFunc)(FILE*)
                                          const);
98          template <class Theader>
99          void writeParticleBinary(const char * const filename,
100                                 const Theader & header,
101                                 void (Tptcl::*pFunc)(FILE*)
                                          const);
102         void writeParticleBinary(const char * const filename,
103                                 void (Tptcl::*pFunc)(FILE*)
                                          const);
104     };
105 }
```

上記に示すようにファイル入出力関連の API には readParticleAscii, readParticleBinary, writeParticleAscii, writeParticleBinary が存在する。これらはオーバーロードされた関数であり、それぞれ 8 種類の異なる引数仕様を持つ。この数は下記に示す引数の有無の組み合わ

せに依るものである (引数の定義については後述する)。

1) `template <class Theader> Theader & header`
2) `const char * const format`
3) `void (Tptcl::*pFunc)(FILE *)`

このあと、各 API について記述する。説明を簡潔にするため、オーバーロードされた関数群をまとめて記述する。

9.1.3.2.3.1 PS::ParticleSystem::readParticleAscii

PS::ParticleSystem::readParticleAscii

readParticleAscii は下記の 8 つの異なる引数仕様を持つ。¡parindent=0pt¿

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

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

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

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

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

```
                    void (Tptcl::*pFunc)(FILE*));

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

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

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

- **引数**

  filename: 入力。const char *型。分散ファイル読み込み時には入力ファイル名のベースとなる部分。単一ファイル読み込み時には入力ファイル名。

  format: 入力。const char *型。分散ファイルから粒子データを読み込む際のファイルフォーマット。

  header: 入力。Theader &型。ファイルのヘッダ情報。

  pFunc: 入力。 void (Tptcl::*)(FILE*) 型。FILE ポインタを引数にとり void を返す Tptcl のメンバ関数ポインタ。

- **返値**

  なし

- **機能**

  引数 format が存在する場合、各プロセスが filename と format の組によって指定される名前の入力ファイルから粒子データを読み出し、データを FullParticle クラスのオブジェクトに格納する (**分散ファイルの読み込み**)。一方、引数 format が存在しない場合、ルートプロセスが filename で指定された入力ファイルから粒子データを読み出し、データを FullParticle クラスのオブジェクトに格納した後、各プロセスに分配する (**単一ファイルの読み込み**)。いずれの場合においても、引数 header が存在しない場合、ファイルに保存されている粒子数を調べるため、1 回ファイルを読み込んで行数を取得した後、

もう一度ファイルを読み込み直し、粒子データを読み込む。すなわち、粒子数の推定は、1 粒子のデータが 1 行に記述されているという仮定の下行う。

分散ファイルを読み込む場合、filename で、分散しているファイルのベースとなる名前を指定する。format でファイル名のフォーマットを指定する。フォーマットの指定方法は標準 C ライブラリの関数 printf の第 1 引数と同じである。ただし変換指定は必ず 3 つであり、その指定子は 1 つめは文字列、残りはどちらも整数である。2 つ目の変換指定にはそのジョブの全プロセス数が、3 つ目の変換指定にはプロセス番号が入る。例えば、filename が nbody、format が%s_%03d_%03d.init ならば、全プロセス数 64 のジョブのプロセス番号 12 のプロセスは、nbody_064_012.init というファイルを読み込む。

1 粒子のデータを読み取る関数は引数 pFunc が存在すればそれを使用し、そうでない場合には FullParticle クラスのメンバ関数 readAscii を使用する。readAscii はユーザが定義しなければならない。readAscii の仕様については節 7.2.4.3.1 を、実装例については節 A.1.4.3 を参照のこと。pFunc も readAscii と同じ仕様を満たす必要がある。

ファイルのヘッダのデータを読み取る関数は Theader のメンバ関数 readAscii でユーザが定義する。仕様については節 7.8.3.1 を、実装例については節 A.7 を参照のこと。

ファイルはアスキーモードで開く。

9.1.3.2.3.2 PS::ParticleSystem::readParticleBinary

PS::ParticleSystem::readParticleBinary

readParticleBinary は下記の 8 つの異なる引数仕様を持つ。¡parindent=0pt¿

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

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

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

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

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

template <class Tptcl>
void PS::ParticleSystem<Tptcl>::readParticleBinary
            (const char * const filename,
             const char * const format,
             void (Tptcl::*pFunc)(FILE*));

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

template <class Tptcl>
void PS::ParticleSystem<Tptcl>::readParticleBinary
            (const char * const filename,
             void (Tptcl::*pFunc)(FILE*));

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

- **引数**

  filename: 入力。const char *型。分散ファイル読み込み時には入力ファイル名のベースとなる部分。単一ファイル読み込み時にはファイル名。

format: 入力。const char *型。分散ファイルから粒子データを読み込む際のファイルフォーマット。

header: 入力。Theader &型。ファイルのヘッダ情報。

pFunc: 入力。 void (Tptcl::*)(FILE*) 型。FILE ポインタを引数にとり void を返す Tptcl のメンバ関数ポインタ。

- **返値**

  なし

- **機能**

  readParticleAscii とほぼ同じ機能を提供するが、下記の点が異なる。

  – ファイルはバイナリーモードで開く。

  – 引数 header が存在しない場合の粒子数の推定は、実際に 1 粒子を読み込んだときのファイル位置指示子のバイト数を調べることによって行うため、1 粒子のデータの最後が改行文字 (\n) となっている必要はない。

9.1.3.2.3.3 PS::ParticleSystem::writeParticleAscii

PS::ParticleSystem::writeParticleAscii

writeParticleAscii は下記の 8 つの異なる引数仕様を持つ。¡parindent=0pt¿

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

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

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

template <class Tptcl>
```

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

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

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

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

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

- **引数**

  filename: 入力。const char * const 型。出力ファイル名のベースとなる部分。

  format: 入力。const char * const 型。分散ファイルに粒子データを書き込む際のファイルフォーマット。

  header: 入力。const Theader &型。ファイルのヘッダ情報。

  pFunc: 入力。void (Tptcl::*)(FILE*)const 型。FILE ポインタを引数にとり void を返す Tptcl のメンバ関数ポインタ。

- **返り値**

  なし。

- **機能**

  引数 format が存在する場合、各プロセスが filename と format の組によって指定される名前の出力ファイルに FullParticle クラスのオブジェクトのデータと、もし存在すれば Theader クラスのオブジェクトのデータを出力する (**分散ファイルへの書き出し**)。一方、引数 format が存在しない場合、各プロセスの FullParticle クラスのオブジェクトのデータをルートプロセスに集め、ルートプロセスが filename で指定された出力ファイルに集めたデータと、もし存在すれば Theader クラスのオブジェクトデータを出力する (**単一ファイルへの書き出し**)。

  分散ファイル書き出し時の出力ファイル名のフォーマットはメンバ関数 PS::ParticleSystem::readParticleAscii の場合と同様である。

  1 粒子のデータを書き込む関数は pFunc が存在する場合にはそれが使用され、そうでない場合には FullParticle クラスのメンバ関数 writeAscii が使用される。writeAscii はユーザが定義しなければならない。writeAscii の仕様については節 7.2.4.3.2 を、実装例については節 A.1.4.3 を参照のこと。

  ファイルのヘッダのデータを書き込む関数はヘッダクラスのメンバ関数 writeAscii でユーザが定義する。仕様については節 7.8.3.2 を、実装例については節 A.7 を参照のこと。

  ファイルはアスキーモードで開く。

9.1.3.2.3.4　PS::ParticleSystem::writeParticleBinary

PS::ParticleSystem::writeParticleBinary

writeParticleBinary は下記の 8 つの異なる引数仕様を持つ。¡parindent=0pt¿

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

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

template <class Tptcl>
template <class Theader>
void PS::ParticleSystem<Tptcl>::writeParticleBinary
            (const char * const filename,
```

```
             const Theader & header);

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

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

template <class Tptcl>
void PS::ParticleSystem<Tptcl>::writeParticleBinary
            (const char * const filename,
             const char * const format,
             void (Tptcl::*pFunc)(FILE*)const);

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

template <class Tptcl>
void PS::ParticleSystem<Tptcl>::writeParticleBinary
            (const char * const filename,
             void (Tptcl::*pFunc)(FILE*)const));
```

- **引数**

  filename: 入力。const char * const 型。分散ファイルへの書き込み時には出力ファイル名のベースとなる部分。単一ファイルへの書き込み時にはファイル名。

  format: 入力。const char * const 型。分散ファイルに粒子データを書き込む際のファイルフォーマット。

  header: 入力。const Theader &型。ファイルのヘッダ情報。

  pFunc: 入力。void (Tptcl::*)(FILE*)const 型。FILE ポインタを引数にとり void を返す Tptcl のメンバ関数ポインタ。

- **返り値**

  なし。

- **機能**

  writeParticleAscii とほぼ同じ機能を提供するが、下記の点が異なる。

  – ファイルはバイナリーモードで開く。

9.1.3.2.4 粒子交換

粒子交換関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 35: ParticleSystem4

```
1 namespace ParticleSimulator {
2     template<class Tptcl>
3     class ParticleSystem{
4     public:
5         template<class Tdinfo>
6         void exchangeParticle(Tdinfo & dinfo, const bool
              flag_serialize=false);
7     };
8 }
```

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, const bool flag_serialize=false);
```

- **引数**

  dinfo: 入力。DomainInfo & 型。領域クラスのオブジェクト。

  flag_serialize: 入力。const bool 型。粒子情報をシリアライズして送信するかを決定するフラグ。true で粒子をシリアライズする。デフォルトは false。

- **返値**

  なし

- **機能**

  粒子が適切なドメインに配置されるように、粒子の交換を行う。粒子をシリアライズして送信する場合は FP にメンバ関数 pack と unPack を定義し (詳しくはセクション 7.2.4.6)、flag_serialize を true にする (シリアライズ送信の対応は未実装)。

9.1.3.2.5 粒子の追加、削除

粒子の追加もしくは削除関連の API は以下の様に宣言されている。

9.1.3.2.5.1 PS::ParticleSystem::addOneParticle()

PS::ParticleSystem::addOneParticle()

```
void PS::ParticleSystem::addOneParticle(const FullPartilce & fp);
```

- **引数**

  fp: 入力。const FullParticle &型。追加する粒子の FullParticle。

- **返値**

  なし。

- **機能**

  追加された粒子を粒子配列の末尾に追加する。

9.1.3.2.5.2 PS::ParticleSystem::removeParticle()

PS::ParticleSystem::removeParticle()

```
void PS::ParticleSystem::removeParticle(const PS::S32 * idx,
                                        const PS::S32 n);
```

- **引数**

  idx: 入力。const PS::S32 * 型。消去する粒子の配列インデックスの配列。

  n: 入力。const PS::S32 型。配列 idx のサイズ。

- **返値**

  なし。

- **機能**

  配列 idx[] に格納されているインデックスの粒子を削除する。この関数を呼ぶ前後で、粒子の配列インデックスが同じである事は保証されない。

9.1.3.2.6 時間計測

クラス内の情報取得関連の API の宣言は以下のようになっている。自クラスの主要なメソッドを呼び出すとそれにかかった時間をプライベートメンバの time_profile_の該当メンバに書き込む。メソッド clearTimeProfile() を呼ばない限り時間は足しあわされていく。

ソースコード 36: ParticleSystem3

```
1  namespace ParticleSimulator {
2      template<class Tptcl>
3      class ParticleSystem{
4      private:
5          TimeProfile time_profile_;
6      public:
7          TimeProfile getTimeProfile();
8          void clearTimeProfile();
9      };
10 }
```

9.1.3.2.6.1 PS::ParticleSystem::getTimeProfile

PS::ParticleSystem::getTimeProfile

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

- **引数**

  なし。

- **返値**

  PS::TimeProfile 型。

- **機能**

  メンバ関数 exchangeParticle にかかった時間（ミリ秒単位）を TimeProfile 型のメンバ変数 exchange_particles_に格納し、返す。

9.1.3.2.6.2 PS::ParticleSystem::clearTimeProfile

PS::ParticleSystem::clearTimeProfile

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

- **引数**

  なし。

- **返値**

  なし。

- **機能**

  領域情報クラスのTimeProfile型のプライベートメンバ変数のメンバ変数exchange_particles_の値を0クリアする。

9.1.3.2.7 その他

その他のAPIの宣言は以下のようになっている。このあと各APIについて記述する。

ソースコード 37: ParticleSystem4

```
1  namespace ParticleSimulator {
2      template<class Tptcl>
3      class ParticleSystem{
4      public:
5          template<class Tdinfo>
6          void adjustPositionIntoRootDomain
7                      (const Tdinfo & dinfo);
8          void setNumberOfParticleLocal(const S32 n);
9          tamplate<class Tcomp>
10         void sortParticle(Tcomp comp);
11     };
12 }
```

9.1.3.2.7.1 PS::ParticleSystem::adjustPositionIntoRootDomain

PS::ParticleSystem::adjustPositionIntoRootDomain

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

- **引数**

  dinfo: 入力。Tdinfo型。領域クラスのオブジェクト。

- **返値**

  なし

- **機能**

  周期境界条件の場合に、計算領域からはみ出した粒子を計算領域に適切に戻す。

9.1.3.2.7.2　PS::ParticleSystem::setNumberOfParticleLocal

PS::ParticleSystem::setNumberOfParticleLocal

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

- **引数**

  n: 入力。const PS::S32 型。粒子数。

- **返値**

  なし

- **機能**

  １つの MPI プロセスの持つ粒子数を設定する。

9.1.3.2.7.3　PS::ParticleSystem::sortParticle

PS::ParticleSystem::sortParticle

```
tamplate<class Tcomp>
void sortParticle(Tcomp comp);
```

- **引数**

  comp: 入力。Tcomp 型。比較関数。比較関数は返り値を bool 型とし、引数は const FullParticle & を２つ。例として以下に FullParticle がメンバ変数 id を持っており id の昇順に並べ替える場合を示す。

  ```
  bool comp(const FP & left, const FP & right){
      return left.id < right.id;
  }
  ```

- **返値**

  なし

- **機能**

  粒子群クラスが保持する FullParticle の配列を引数 comp で指示したように並べ替える。

9.1.4 相互作用ツリークラス

本節では、相互作用ツリークラスについて記述する。このクラスは粒子間相互作用の計算を行うモジュールである。まずオブジェクトの生成方法を記述し、その後 API を記述する。

9.1.4.1 オブジェクトの生成

このクラスは以下のように宣言されている。

ソースコード 38: TreeForForce0

```
1  namespace ParticleSimulator {
2      template<class TSearchMode,
3               class TResult,
4               class TEpi,
5               class TEpj,
6               class TMomLocal,
7               class TMomGlobal,
8               class TSpj>
9      class TreeForForce;
10 }
```

テンプレート引数は順に、PS::SEARCH_MODE 型 (ユーザー選択)、Force クラス (ユーザー定義)、EssentialParticleI クラス (ユーザー定義)、EssentialParticleJ 型 (ユーザー定義)、ローカルツリーの Moment 型 (ユーザー定義)、グローバルツリーの Moment 型 (ユーザー定義)、SuperParticleJ 型 (ユーザー定義) である。

PS::SEARCH_MODE 型に応じてラッパーを用意した。これらのラッパーを使えば入力するテンプレート引数の数が減るので、こちらのラッパーを用いることを推奨する。以下、各 PS::SEARCH_MODE 型の場合のオブジェクトの生成方法を記述する。

9.1.4.1.1 PS::SEARCH_MODE_LONG

以下のようにオブジェクト system を生成する。

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

テンプレート引数は順に、Force クラス (ユーザー定義)、EssentialParticleI クラス (ユーザー定義)、EssentialParticleJ クラス (ユーザー定義)、ローカルツリー及びグローバルツリーの Moment クラス (ユーザー定義)、SuperParticleJ クラス (ユーザー定義) である。

Moment クラスと SuperParticleJ クラスに FDPS が提供する Moment クラスと SuperParticleJ クラスを指定した型も用意した。これらはモーメントの計算方法別に 6 種類ある。以下、粒子の重心を中心とした場合の単極子まで、四重極子までのモーメント計算、粒子の幾何中心を中心とした場合の単極子まで、双極子まで、四重極子までのモーメント計算、のオブジェクトの生成方法をこの順で記述する。ここでは、すべて system というオブジェクトを生成している。

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

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

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

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

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

各相互作用ツリークラスで使用している Moment クラスと SuperParticleJ クラスは、上記の順で、

- PS::MomentMonopole と PS::SPJMonopole (定義は 7.5.2.2.1, 7.6.2.1.1 参照)
- PS::MomentQuadrupole と PS::SPJQuadrupole (定義は 7.5.2.2.2, 7.6.2.1.2 参照)
- PS::MomentMonopoleGeometricCenter と PS::SPJMonopoleGeometricCenter (定義は 7.5.2.2.3, 7.6.2.1.3 参照)
- PS::MomentDipoleGeometricCenter と PS::SPJDipoleGeometricCenter (定義は 7.5.2.2.4, 7.6.2.1.4 参照)
- PS::MomentQuadrupoleGeometricCenter と PS::SPJQuadrupoleGeometricCenter (定義は 7.5.2.2.5, 7.6.2.1.5 参照)

である。

テンプレート引数は PS::SEARCH_MODE_LONG の場合と同じである。

9.1.4.1.2 PS::SEARCH_MODE_LONG_SCATTER

以下のようにオブジェクト system を生成する。

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

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラス、ローカルツリー及びグローバルツリーの Moment クラス、SuperParticleJ クラスである。

Moment クラスと SuperParticleJ クラスに FDPS が提供する Moment クラスと SuperParticle J クラスを指定したクラスも用意した。これらはモーメントの計算方法別に 2 種類ある。以下、粒子の重心を中心とした場合の単極子まで、四重極子までのモーメント計算のオブジェクトの生成方法をこの順で記述する。ここでは、すべて system というオブジェクトを生成している。

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

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

各相互作用ツリークラスで使用している Moment クラスと SuperParticleJ クラスは、上記の順で、PS::MomentMonopoleScatter と PS::SPJMonopoleScatter (定義は 7.5.2.3.1, 7.6.2.2.1 参照)、PS::MomentQuadrupoleScatter と PS::SPJQuadrupoleScatter (定義は 7.5.2.3.2, 7.6.2.2.2 参照) である。

テンプレート引数は PS::SEARCH_MODE_LONG の場合と同じである。

9.1.4.1.3 PS::SEARCH_MODE_LONG_SYMMETRY

以下のようにオブジェクト system を生成する。

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

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラス、ローカルツリー及びグローバルツリーの Moment クラス、SuperParticleJ クラスである。

Moment クラスと SuperParticleJ クラスに FDPS が提供する Moment クラスと SuperParticleJ クラスを指定したクラスも用意した。これらはモーメントの計算方法別に 2 種類ある。以下、粒子の重心を中心とした場合の単極子まで、四重極子までのモーメント計算のオブジェクトの生成方法をこの順で記述する。ここでは、すべて system というオブジェクトを生成している。

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

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

各相互作用ツリークラスで使用している Moment クラスと SuperParticleJ クラスは、上記の順で、PS::MomentMonopoleSymmetry と PS::SPJMonopoleSymmetry (定義は 7.5.2.4.1, 7.6.2.3.1 参照)、PS::MomentQuadrupoleSymmetry と PS::SPJQuadrupoleSymmetry (定義は 7.5.2.4.2, 7.6.2.3.2 参照) である。

テンプレート引数は PS::SEARCH_MODE_LONG の場合と同じである。

9.1.4.1.4 PS::SEARCH_MODE_LONG_CUTOFF

以下のようにオブジェクト system を生成する。

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

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラス、ローカルツリー及びグローバルツリーの Moment クラス、SuperParticleJ クラスである。

Moment クラスと SuperParticleJ クラスに FDPS が提供する PS::MomentMonopoleCutoff と PS::SPJMonopoleCutoff (定義は、それぞれ 7.5.2.5.1 節と 7.6.2.4.1 節) を使う場合には、以下の宣言で system を生成できる。これは、モーメント計算の中心を粒子の重心とした場合に、単極子まで計算するものである。

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

テンプレート引数は PS::SEARCH_MODE_LONG の場合と同じである。

9.1.4.1.5 PS::SEARCH_MODE_GATHER

以下のようにオブジェクト system を生成する。

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

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラスである。

9.1.4.1.6 PS::SEARCH_MODE_SCATTER

以下のようにオブジェクト system を生成する。

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

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラスである。

9.1.4.1.7 PS::SEARCH_MODE_SYMMETRY

以下のようにオブジェクト system を生成する。

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

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラスである。

9.1.4.2 API

このモジュールには初期設定関連の API、相互作用計算関連の低レベル API、相互作用計算関連の高レベル API、ネイバーリスト関連の API がある。以下、各節に分けて記述する。

本節の中の API の宣言ではテンプレートクラスのテンプレート引数は省略する。すなわち、本来ならば以下のように記述するべきであるが、

```
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);
```

冗長であるので、以下のように省略する。

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

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

9.1.4.2.1 初期設定

初期設定関連のAPIの宣言は以下のようになっている。このあと各APIについて記述する。

ソースコード 39: TreeForForce1

```
1  namespace ParticleSimulator {
2      template<class TSearchMode,
3               class TResult,
4               class TEpi,
5               class TEpj,
6               class TMomLocal,
7               class TMomGlobal,
8               class TSpj>
9      class TreeForForce{
10     public:
11     void TreeForForce();
12     void initialize(const U64 n_glb_tot,
13                     const F32 theta=0.7,
14                     const U32 n_leaf_limit=8,
15                     const U32 n_group_limit=64);
16     };
17 }
```

9.1.4.2.1.1 コンストラクタ

コンストラクタ

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

- **引数**

  なし

- **返値**

  なし

- **機能**

  相互作用ツリークラスのオブジェクトを生成する。

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);
```

- **引数**

  n_glb_tot: 入力。const PS::U64 型。粒子配列の上限。

  theta: 入力。const PS::F32 型。見こみ角に対する基準。デフォルト 0.7。

  n_leaf_limit。const PS::U32 型。ツリーを切るのをやめる粒子数の上限。デフォルト 8。

  n_group_limit。const PS::U32 型。相互作用リストを共有する粒子数の上限。デフォルト 64。

- **返値**

  なし

- **機能**

  相互作用ツリークラスのオブジェクトを初期化する。

9.1.4.2.2 低レベル関数

相互作用計算関連の低レベル API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 40: TreeForForce1

```
1  namespace ParticleSimulator {
2      template<class TSearchMode,
3               class TResult,
4               class TEpi,
5               class TEpj,
6               class TMomLocal,
7               class TMomGlobal,
8               class TSpj>
9      class TreeForForce{
10     public:
11         template<class Tpsys>
12         void setParticleLocalTree(const Tpsys & psys,
13                                   const bool clear=true);
```

```
14          template<class Tdinfo>
15          void makeLocalTree(const Tdinfo & dinfo);
16          void makeLocalTree(const F32 l,
17                             const F32vec & c = F32vec(0.0));
18          template<class Tdinfo>
19          void makeGlobalTree(const Tdinfo & dinfo);
20          void calcMomentGlobalTree();
21          template<class Tfunc_ep_ep>
22          void calcForce(Tfunc_ep_ep pfunc_ep_ep,
23                         const bool clear=true);
24          template<class Tfunc_ep_ep, class Tfunc_sp_ep>
25          void calcForce(Tfunc_ep_ep pfunc_ep_ep,
26                         Tfunc_sp_ep pfunc_sp_ep,
27                         const bool clear=true);
28          Tforce getForce(const S32 i);
29      };
30  }
```

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);
```

- **引数**

  psys: 入力。const Tpsys & 型。ローカルツリーを構成する粒子群クラスのオブジェクト。

  clear: 入力。const bool 型。前に読込んだ粒子をクリアするかどうか決定するフラグ。true でクリアする。デフォルト true。

- **返値**

  なし

- **機能**

  相互作用ツリークラスのオブジェクトに粒子群クラスのオブジェクトの粒子を読み込む。clear が true ならば前に読込んだ粒子情報をクリアし、false ならクリアしない。

9.1.4.2.2.2　PS::TreeForForce::makeLocalTree

PS::TreeForForce::makeLocalTree

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

- **引数**

  dinfo: 入力。const Tdinfo &型。領域クラスのオブジェクト。

- **返値**

  なし

- **機能**

  ローカルツリーを作る。領域クラスのオブジェクトから扱うべきルートドメインを読み取り、ツリーのルートセルを決定する。

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

- **引数**

  l: 入力。const PS::F32 型。ツリーのルートセルの大きさ。

  c: 入力。const PS::F32vec &型。ツリーの中心の座標。デフォルトは座標原点。

- **返値**

  なし

- **機能**

  ローカルツリーを作る。ツリーのルートセルを２つの引数で決定する。ツリーのルートセルは全プロセスで共通でなければならない。共通でない場合の動作の正しさは保証しない。

9.1.4.2.2.3　PS::TreeForForce::makeGlobalTree

PS::TreeForForce::makeGlobalTree

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

- **引数**

  dinfo: 入力。const Tdinfo & 型。領域クラスのオブジェクト。

- **返値**

  なし

- **機能**

  グローバルツリーを作る。

9.1.4.2.2.4 PS::TreeForForce::calcMomentGlobalTree

PS::TreeForForce::calcMomentGlobalTree

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

- **引数**

  なし

- **返値**

  なし

- **機能**

  グローバルツリーの各々のセルのモーメントを計算する。

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);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト (節 7.9 参照)。関数の引数は第 1 引数から順に (const) TEpi *型、const PS::S32 型、(const) TEpj *型、const PS::S32 型、TResult *型。

clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

- **返値**

  なし

- **機能**

  このオブジェクトに読み込まれた粒子すべての粒子間相互作用を計算する。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY の場合に限る。

```
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);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、const PS::S32 型、(const) TEpj *型、const PS::S32 型、TResult *型。

  pfunc_sp_ep: 入力。返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、const PS::S32 型、(const) TSpj *型、const PS::S32 型、TResult *型。

  clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

- **返値**

  なし

- **機能**

  このオブジェクトに読み込まれた粒子すべての粒子間相互作用を計算する。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOFF の場合に限る。

9.1.4.2.2.6 PS::TreeForForce::getForce

PS::TreeForForce::getForce

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

- **引数**

  i: 入力。const PS::S32 型。粒子配列のインデックス。

- **返値**

  TResult 型。PS::TreeForForce::setParticleLocalTree で i 番目に読み込まれた粒子の受ける作用。

- **機能**

  PS::TreeForForce::setParticleLocalTree で i 番目に読み込まれた粒子の受ける作用を返す。

9.1.4.2.2.7 PS::TreeForForce::copyLocalTreeStructure

PS::TreeForForce::copyLocalTreeStructure

今後、追加する。

9.1.4.2.2.8 PS::TreeForForce::repeatLocalCalcForce

PS::TreeForForce::repeatLocalCalcForce

今後、追加する。

9.1.4.2.3 高レベル関数

相互作用計算関連の高レベル API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 41: TreeForForce1

```
1  namespace ParticleSimulator {
2      template<class TSearchMode,
3               class TResult,
4               class TEpi,
5               class TEpj,
6               class TMomLocal,
7               class TMomGlobal,
8               class TSpj>
9      class TreeForForce{
10     public:
11         template<class Tfunc_ep_ep,
12                  class Tpsys>
13         void calcForceAllAndWriteBack
14                     (Tfunc_ep_ep pfunc_ep_ep,
```

```
15                  Tpsys & psys,
16                  DomainInfo & dinfo,
17                  const bool clear_force = true,
18                  const INTERACTION_LIST_MODE list_mode =
                        MAKE_LIST,
19                  const bool flag_serialize=false);
20
21          template<class Tfunc_ep_ep,
22                   class Tfunc_sp_ep,
23                   class Tpsys>
24          void calcForceAllAndWriteBack
25                      (Tfunc_ep_ep pfunc_ep_ep,
26                       Tfunc_sp_ep pfunc_sp_ep,
27                       Tpsys & psys,
28                       DomainInfo & dinfo,
29                       const bool clear_force=true,
30                       const INTERACTION_LIST_MODE list_mode =
                             MAKE_LIST,
31                       const bool flag_serialize=false);
32
33
34          template<class Tfunc_dispatch,
35                   class Tfunc_retrieve,
36                   class Tpsys>
37          void calcForceAllandWriteBackMultiWalk
38                      (Tfunc_dispatch pfunc_dispatch,
39                       Tfunc_retrieve pfunc_retrieve,
40                       const S32 tag_max,
41                       Tpsys & psys,
42                       DomainInfo & dinfo,
43                       const S32 n_walk_limit,
44                       const bool clear=true,
45                       const INTERACTION_LIST_MODE list_mode =
                             MAKE_LIST,
46                       const bool flag_serialize=false);
47
48          template<class Tfunc_dispatch,
49                   class Tfunc_retrieve,
50                   class Tpsys>
51          void calcForceAllandWriteBackMultiWalkIndex
```

```
52                  (Tfunc_dispatch pfunc_dispatch,
53                   Tfunc_retrieve pfunc_retrieve,
54                   const S32 tag_max,
55                   Tpsys & psys,
56                   DomainInfo & dinfo,
57                   const S32 n_walk_limit,
58                   const bool clear=true,
59                   const INTERACTION_LIST_MODE list_mode =
                          MAKE_LIST,
60                   const bool flag_serialize=false);
61
62          template<class Tfunc_ep_ep,
63                   class Tpsys>
64          void calcForceAll
65                  (Tfunc_ep_ep pfunc_ep_ep,
66                   Tpsys & psys,
67                   DomainInfo & dinfo,
68                   const bool clear_force=true,
69                   const INTERACTION_LIST_MODE list_mode =
                          MAKE_LIST,
70                   const bool flag_serialize=false);
71
72          template<class Tfunc_ep_ep,
73                   class Tfunc_sp_ep,
74                   class Tpsys>
75          void calcForceAll
76                  (Tfunc_ep_ep pfunc_ep_ep,
77                   Tfunc_sp_ep pfunc_sp_ep,
78                   Tpsys & psys,
79                   DomainInfo & dinfo,
80                   const bool clear_force=true,
81                   const INTERACTION_LIST_MODE list_mode =
                          MAKE_LIST,
82                   const bool flag_serialize=false);
83
84
85          template<class Tfunc_ep_ep>
86          void calcForceMakeingTree
87                  (Tfunc_ep_ep pfunc_ep_ep,
88                   DomainInfo & dinfo,
```

```
89                      const bool clear_force=true,
90                      const INTERACTION_LIST_MODE list_mode =
                            MAKE_LIST,
91                      const bool flag_serialize=false);
92
93          template<class Tfunc_ep_ep,
94                   class Tfunc_sp_ep>
95          void calcForceMakingTree
96                      (Tfunc_ep_ep pfunc_ep_ep,
97                       Tfunc_sp_ep pfunc_sp_ep,
98                       DomainInfo & dinfo,
99                       const bool clear_force=true,
100                      const INTERACTION_LIST_MODE list_mode =
                            MAKE_LIST,
101                      const bool flag_serialize=false);
102
103         template<class Tfunc_ep_ep,
104                  class Tpsys>
105         void calcForceAndWriteBack
106                     (Tfunc_ep_ep pfunc_ep_ep,
107                      Tpsys & psys,
108                      const bool clear=true,
109                      const INTERACTION_LIST_MODE list_mode =
                            MAKE_LIST,
110                      const bool flag_serialize=false);
111
112         template<class Tfunc_ep_ep,
113                  class Tfunc_sp_ep,
114                  class Tpsys>
115         void calcForceAndWriteBack
116                     (Tfunc_ep_ep pfunc_ep_ep,
117                      Tfunc_sp_ep pfunc_sp_ep,
118                      Tpsys & psys,
119                      const bool clear=true,
120                      const INTERACTION_LIST_MODE list_mode =
                            MAKE_LIST,
121                      const bool flag_serialize=false);
122     };
123 }
124 namespace PS = ParticleSimulator;
```

9.1.4.2.3.1 PS::TreeForForce::calcForceAllAndWriteBack

PS::TreeForForce::calcForceAllAndWriteBack

```
template<class Tfunc_ep_ep,
         class Tpsys>
void PS::TreeForForce::calcForceAllAndWriteBack
             (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                      const PS::S32,
                                      TEpj *,
                                      const PS::S32,
                                      TResult *),
              Tpsys & psys,
              DomainInfo & dinfo
              const bool clear=true,
              const PS::INTERACTION_LIST_MODE list_mode = PS::MAKE_LIST,
              const bool flag_serialize=false);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TEpj *型、PS::S32 型、TResult *型。

  psys: 入力。Tpsys &型。相互作用を計算したい粒子群クラスのオブジェクト。

  dinfo: 入力。DomainInfo &型。領域クラスのオブジェクト。

  clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

  list_mode: 入力。PS::INTERACTION_LIST_MODE 型。相互作用リストを作成し相互作用計算を行うか、前回作成した相互作用リストを再利用し相互作用計算を行うかを決定するフラグ。PS::MAKE_LIST ならば新たに相互作用リストを作成する。この場合、次の相互作用計算時に相互作用リストの再利用はできず、新たに相互作用リストを作成する必要がある。PS::MAKE_LIST_FOR_REUSE は新たに相互作用リストを作成し相互作用計算を行う。この場合、次回の相互作用計算時に、今回作った相互作用リストを再利用し相互作用計算ができる。PS::REUSE_LIST ならば、前回 PS::MAKE_LIST_FOR_REUSE を選んだ際に作成した相互作用リストを再利用し相互作用計算を行う。デフォルトは PS::MAKE_LIST。PS::INTERACTION_LIST_MODE の詳細はセクション 6.8.3 を参照。

  flag_serialize: 入力。const bool 型。粒子情報をシリアライズして送信するかを決定するフラグ。true で粒子をシリアライズする。デフォルトは false。

- **返値**

なし

- **機能**

  粒子群クラスのオブジェクト psys の粒子すべての相互作用を計算し、その計算結果を psys に書き戻す。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY の場合に限る。

  PS::INTERACTION_LIST_MODE により相互作用リストを新たに作るか再利用するかを選ぶことができる。相互作用リストの再利用している間は PS::ParticleSystem::exchangeParticle() によって粒子の交換を行ってはならない。また、粒子データの消去や粒子配列の順番の並び替えを行ってはならない。

  LET をシリアライズして送信する場合は EPJ にメンバ関数 pack と unPack を定義し (詳しくはセクション 7.4.4.5)、flag_serialize を true にする (この機能は未実装)。

```
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,
              DomainInfo & dinfo
              const bool clear=true,
              const PS::INTERACTION_LIST_MODE list_mode = PS::MAKE_LIST,
              const bool flag_serialize=false);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TEpj *型、PS::S32 型、TResult *型。

  pfunc_sp_ep: 入力。返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TSpj *型、PS::S32 型、TResult *型。

psys: 入力。Tpsys &型。相互作用を計算したい粒子群クラスのオブジェクト。

dinfo: 入力。DomainInfo &型。領域クラスのオブジェクト。

clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

list_mode: 入力。PS::INTERACTION_LIST_MODE 型。相互作用リストを作成し相互作用計算を行うか、前回作成した相互作用リストを再利用し相互作用計算を行うかを決定するフラグ。PS::MAKE_LIST ならば新たに相互作用リストを作成する。この場合、次の相互作用計算時に相互作用リストの再利用はできず、新たに相互作用リストを作成する必要がある。PS::MAKE_LIST_FOR_REUSE は新たに相互作用リストを作成し相互作用計算を行う。この場合、次回の相互作用計算時に、今回作った相互作用リストを再利用し相互作用計算ができる。PS::REUSE_LIST ならば、前回 PS::MAKE_LIST_FOR_REUSE を選んだ際に作成した相互作用リストを再利用し相互作用計算を行う。デフォルトは PS::MAKE_LIST。PS::INTERACTION_LIST_MODE の詳細はセクション 6.8.3 を参照。

flag_serialize: 入力。const bool 型。粒子情報をシリアライズして送信するかを決定するフラグ。true で粒子をシリアライズする。デフォルトは false。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys の粒子すべての相互作用を計算し、その計算結果を psys に書き戻す。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOFF の場合に限る。

  PS::INTERACTION_LIST_MODE により相互作用リストを新たに作るか再利用するかを選ぶことができる。相互作用リストの再利用している間は PS::ParticleSystem::exchangeParticle() によって粒子の交換を行ってはならない。また、粒子データの消去や粒子配列の順番の並び替えを行ってはならない。

  LET をシリアライズして送信する場合は EPJ と SPJ にメンバ関数 pack と unPack を定義し (詳しくはセクション 7.4.4.5,7.6.4.2)、flag_serialize を true にする (この機能は未実装)。

9.1.4.2.3.2 PS::TreeForForce::calcForceAllAndWriteBackMultiWalk

PS::TreeForForce::calcForceAllAndWriteBackMultiWalk

```
template<class Tfunc_dispatch,
         class Tfunc_retrieve,
         class Tpsys>
void PS::TreeForForce::calcForceAllAndWriteBackMultiWalk
             (Tfunc_dispatch pfunc_dispatch,
              Tfunc_retrieve pfunc_retrieve,
              const PS::S32 tag_max,
              Tpsys & psys,
              DomainInfo & dinfo,
              const PS::S32 n_walk_limit,
              const bool clear=true,
              const PS::INTERACTION_LIST_MODE list_mode = PS::MAKE_LIST,
              const bool flag_serialize=false);
```

- **引数**

  pfunc_dispatch: 入力。返値が void 型で、EssentialParticleI の配列と EssentialParticleJ の配列 (と SuperParticleJ の配列) をアクセラレータに転送し、カーネルを発行させる関数。PS::SEARCH_MODE 型が PS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOFF の場合にこの関数は以下の形ををしている必要がある。

  ```
  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);
  ```

  関数の引数は第 1 引数から順に const PS::S32 型、const PS::S32 型、const TEpi**型、PS::S32*型、const TEpj**型、PS::S32*型、const TSpj**型、PS::S32*型。返り値は PS::S32 型とし、正常に終了した場合は 0 を返す。

  PS::SEARCH_MODE 型が PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY の場合にこの関数は以下の形ををしている必要がある。

  ```
  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);
```

関数の引数は第１引数から順に const PS::S32 型、const PS::S32 型、const TEpi**型、PS::S32*型、const TEpj**型、PS::S32*型。返り値は PS::S32 型とし、正常に終了した場合は 0 を返す。

pfunc_retrieve: 入力。返値が void 型で、pfunc_dispatch で転送したデータの結果を回収する関数。この関数は以下の形ををしている必要がある。

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

関数の引数は第１引数から順に const PS::S32 型、PS::S32*型、TResult**型。引数 tag は pfunc_dispatch() と pfunc_retrieve() を関係させるもので、pfunc_dispatch で計算された結果は同じ tag の値を持つ pfunc_retrieve() で回収される。

tag_max: 入力。const PS::S32 型。発行される tag の数の最大値。扱う tag の番号は 0 から tag_max-1 までとなる。0 以下の整数を指定した場合はエラーを出力する。現バージョンでは、1 の場合に正常に動作し、1 を超える値を指定した場合には tag の値は０のみである。

psys: 入力。Tpsys &型。相互作用を計算したい粒子群クラスのオブジェクト。

dinfo: 入力。DomainInfo &型。領域クラスのオブジェクト。

n_walk_limit: 入力。const PS::S32 型。1 度にアクセラレータに送る相互作用リストの数の最大値。

clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

list_mode: 入力。PS::INTERACTION_LIST_MODE 型。相互作用リストを作成し相互作用計算を行うか、前回作成した相互作用リストを再利用し相互作用計算を行うかを決定するフラグ。PS::MAKE_LIST ならば新たに相互作用リストを作成する。この場合、次の相互作用計算時に相互作用リストの再利用はできず、新たに相互作用リストを作成する必要がある。PS::MAKE_LIST_FOR_REUSE は新たに相互作用リストを作成し相互作用計算を行う。この場合、次回の相互作用計算時に、今回作った相互作用リストを再利用し相互作用計算ができる。PS::REUSE_LIST ならば、前回 PS::MAKE_LIST_FOR_REUSE を選んだ際に作成した相互作用リストを再利用し相互作用計算を行う。デフォルトは PS::MAKE_LIST。PS::INTERACTION_LIST_MODE の詳細はセクション 6.8.3 を参照。

flag_serialize: 入力。const bool 型。粒子情報をシリアライズして送信するかを決定するフラグ。true で粒子をシリアライズする。デフォルトは false。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys の粒子すべての相互作用を計算し、その計算結果を psys に書き戻す。相互作用リストを作る際にマルチウォーク法を用い、一度に複数の相互作用リストを作成する。一度に作成する相互作用リストの数の最大値は n_walk_limit である。

  PS::INTERACTION_LIST_MODE により相互作用リストを新たに作るか再利用するかを選ぶことができる。相互作用リストの再利用している間は PS::ParticleSystem::exchangeParticle() によって粒子の交換を行ってはならない。また、粒子データの消去や粒子配列の順番の並び替えを行ってはならない。

  LET をシリアライズして送信する場合は EPJ(長距離力の場合は SPJ も) にメンバ関数 pack と unPack を定義し (詳しくはセクション 7.4.4.5,7.6.4.2)、flag_serialize を true にする (この機能は未実装)。

9.1.4.2.3.3 PS::TreeForForce::calcForceAllAndWriteBackMultiWalkIndex

PS::TreeForForce::calcForceAllAndWriteBackMultiWalkIndex

```
template<class Tfunc_dispatch,
         class Tfunc_retrieve,
         class Tpsys>
void PS::TreeForForce::calcForceAllAndWriteBackMultiWalkIndex
             (Tfunc_dispatch pfunc_dispatch,
              Tfunc_retrieve pfunc_retrieve,
              const PS::S32 tag_max,
              Tpsys & psys,
              DomainInfo & dinfo,
              const PS::S32 n_walk_limit,
              const bool clear=true,
              const PS::INTERACTION_LIST_MODE list_mode = PS::MAKE_LIST,
              const bool flag_serialize=false);
```

- **引数**

  pfunc_dispatch: 入力。返値が void 型で、EssentialParticleI の配列と EssentialParticleJ の配列 (と SuperParticleJ の配列) をアクセラレータに転送し、カーネルを発行させる関

数。PS::SEARCH_MODE 型が PS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOF
の場合にこの関数は以下の形ををしている必要がある。

```
PS::S32 pfunc_dispatch(const PS::S32   tag,
                       const PS::S32   nwalk,
                       const TEpi**    iptcl,
                       const PS::S32*  ni,
                       const PS::S32** id_jptcl_ep,
                       const PS::S32*  nj_ep,
                       const PS::S32** id_jptcl_sp,
                       const PS::S32*  nj_sp,
                       const TEpj*     jptcl_ep,
                       const PS::S32   n_send_ep,
                       const TSpj*     jptcl_sp,
                       const PS::S32   n_send_sp,
                       const bool send_ptcl);
```

関数の引数は第 1 引数から順に const PS::S32 型、const PS::S32 型、const TEpi**型、PS::S32*型、const PS::S32**型、const PS::S32*型、const PS::S32**型、const PS::S32*型、const TEpj*型、PS::S32 型、const TSpj*型、PS::S32 型、const bool 型。返り値は PS::S32 型とし、正常に終了した場合は 0 を返す。

PS::SEARCH_MODE 型が PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY の場合にこの関数は以下の形ををしている必要がある。

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

関数の引数は第 1 引数から順に const PS::S32 型、const PS::S32 型、const TEpi**型、PS::S32*型、const PS::S32**型、const PS::S32*型、const TEpj* 型、PS::S32 型、const TSpj*型、PS::S32 型、const bool 型。返り値は PS::S32 型とし、正常に終了した場合は 0 を返す。

pfunc_retrieve: 入力。返値が void 型で、pfunc_dispatch で転送したデータの結果を回収する関数。この関数は以下の形ををしている必要がある。

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

関数の引数は第１引数から順に const PS::S32 型、const TEpi *型、PS::S32 型、const TSpj *型、PS::S32 型、TResult *型。引数 tag は pfunc_dispatch() と pfunc_retrieve() を関係させるもので、pfunc_dispatch で計算された結果は同じ tag の値を持つ pfunc_retrieve() で回収される。

tag_max: 入力。const PS::S32 型。発行される tag の数の最大値。扱う tag の番号は 0 から tag_max-1 までとなる。0 以下の整数を指定した場合はエラーを出力する。現バージョンでは、1 の場合に正常に動作し、1 を超える値を指定した場合には tag の値は０のみである。

psys: 入力。Tpsys &型。相互作用を計算したい粒子群クラスのオブジェクト。

dinfo: 入力。DomainInfo &型。領域クラスのオブジェクト。

n_walk_limit: 入力。const PS::S32 型。1 度にアクセラレータに送る相互作用リストの数の最大値。

clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

list_mode: 入力。PS::INTERACTION_LIST_MODE 型。相互作用リストを作成し相互作用計算を行うか、前回作成した相互作用リストを再利用し相互作用計算を行うかを決定するフラグ。PS::MAKE_LIST ならば新たに相互作用リストを作成する。この場合、次の相互作用計算時に相互作用リストの再利用はできず、新たに相互作用リストを作成する必要がある。PS::MAKE_LIST_FOR_REUSE は新たに相互作用リストを作成し相互作用計算を行う。この場合、次回の相互作用計算時に、今回作った相互作用リストを再利用し相互作用計算ができる。PS::REUSE_LIST ならば、前回 PS::MAKE_LIST_FOR_REUSE を選んだ際に作成した相互作用リストを再利用し相互作用計算を行う。デフォルトは PS::MAKE_LIST。PS::INTERACTION_LIST_MODE の詳細はセクション 6.8.3 を参照。

flag_serialize: 入力。const bool 型。粒子情報をシリアライズして送信するかを決定するフラグ。true で粒子をシリアライズする。デフォルトは false。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys の粒子すべての相互作用を計算し、その計算結果を psys に書き戻す。初期に全ての EPJ と SPJ を device メモリ上に転送し、FDPS はデバイス上のインデクスについての相互作用リストを作る。この際にマルチウォーク法を

用い、一度に複数の相互作用リストを作成する。一度に作成する相互作用リストの数の最大値は n_walk_limit である。

PS::INTERACTION_LIST_MODE により相互作用リストを新たに作るか再利用するかを選ぶことができる。相互作用リストの再利用している間は PS::ParticleSystem::exchangeParticle() によって粒子の交換を行ってはならない。また、粒子データの消去や粒子配列の順番の並び替えを行ってはならない。

LET をシリアライズして送信する場合は EPJ(長距離力の場合は SPJ も) にメンバ関数 pack と unPack を定義し (詳しくはセクション 7.4.4.5,7.6.4.2)、flag_serialize を true にする (この機能は未実装)。

9.1.4.2.3.4 PS::TreeForForce::calcForceAll

PS::TreeForForce::calcForceAll

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

- **引数**

pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TEpj *型、PS::S32 型、TResult *型。

psys: 入力。Tpsys &型。相互作用を計算したい粒子群クラスのオブジェクト。

dinfo: 入力。DomainInfo &型。領域クラスのオブジェクト。

clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

list_mode: 入力。PS::INTERACTION_LIST_MODE 型。相互作用リストを作成し相互作用計算を行うか、前回作成した相互作用リストを再利用し相互作用計算を行うかを決定するフラグ。PS::MAKE_LIST ならば新たに相互作用リストを作成する。こ

の場合、次の相互作用計算時に相互作用リストの再利用はできず、新たに相互作用リストを作成する必要がある。PS::MAKE_LIST_FOR_REUSE は新たに相互作用リストを作成し相互作用計算を行う。この場合、次回の相互作用計算時に、今回作った相互作用リストを再利用し相互作用計算ができる。PS::REUSE_LIST ならば、前回 PS::MAKE_LIST_FOR_REUSE を選んだ際に作成した相互作用リストを再利用し相互作用計算を行う。デフォルトは PS::MAKE_LIST。PS::INTERACTION_LIST_MODE の詳細はセクション 6.8.3 を参照。

flag_serialize: 入力。const bool 型。粒子情報をシリアライズして送信するかを決定するフラグ。true で粒子をシリアライズする。デフォルトは false。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys の粒子すべての相互作用を計算する。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY の場合に限る。PS::TreeForForce::calcForceAllAndWriteBack から計算結果の書き戻しがなくなったもの。

  PS::INTERACTION_LIST_MODE により相互作用リストを新たに作るか再利用するかを選ぶことができる。相互作用リストの再利用している間は PS::ParticleSystem::exchangeParticle() によって粒子の交換を行ってはならない。また、粒子データの消去や粒子配列の順番の並び替えを行ってはならない。

  LET をシリアライズして送信する場合は EPJ(長距離力の場合は SPJ も) にメンバ関数 pack と unPack を定義し (詳しくはセクション 7.4.4.5,7.6.4.2)、flag_serialize を true にする (この機能は未実装)。

```
template<class Tfunc_ep_ep,
         class Tfunc_sp_ep,
         class Tpsys>
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,
              DomainInfo & dinfo
              const bool clear=true,
              const PS::INTERACTION_LIST_MODE list_mode = PS::MAKE_LIST,
              const bool flag_serialize=false);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TEpj *型、PS::S32 型、TResult *型。

  pfunc_sp_ep: 入力。返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TSpj *型、PS::S32 型、TResult *型。

  psys: 入力。Tpsys &型。相互作用を計算したい粒子群クラスのオブジェクト。

  dinfo: 入力。DomainInfo &型。領域クラスのオブジェクト。

  clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

  list_mode: 入力。PS::INTERACTION_LIST_MODE 型。相互作用リストを作成し相互作用計算を行うか、前回作成した相互作用リストを再利用し相互作用計算を行うかを決定するフラグ。PS::MAKE_LIST ならば新たに相互作用リストを作成する。この場合、次の相互作用計算時に相互作用リストの再利用はできず、新たに相互作用リストを作成する必要がある。PS::MAKE_LIST_FOR_REUSE は新たに相互作用リストを作成し相互作用計算を行う。この場合、次回の相互作用計算時に、今回作った相互作用リストを再利用し相互作用計算ができる。PS::REUSE_LIST ならば、前回

PS::MAKE_LIST_FOR_REUSE を選んだ際に作成した相互作用リストを再利用し相互作用計算を行う。デフォルトは PS::MAKE_LIST。PS::INTERACTION_LIST_MODE の詳細はセクション 6.8.3 を参照。

flag_serialize: 入力。const bool 型。粒子情報をシリアライズして送信するかを決定するフラグ。true で粒子をシリアライズする。デフォルトは false。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys の粒子すべての相互作用を計算する。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOFF の場合に限る。PS::TreeForForce::calcForceAllAndWriteBack から計算結果の書き戻しがなくなったもの。

  PS::INTERACTION_LIST_MODE により相互作用リストを新たに作るか再利用するかを選ぶことができる。相互作用リストの再利用している間は PS::ParticleSystem::exchangeParticle() によって粒子の交換を行ってはならない。また、粒子データの消去や粒子配列の順番の並び替えを行ってはならない。

  LET をシリアライズして送信する場合は EPJ(長距離力の場合は SPJ も) にメンバ関数 pack と unPack を定義し (詳しくはセクション 7.4.4.5,7.6.4.2)、flag_serialize を true にする (この機能は未実装)。

9.1.4.2.3.5 PS::TreeForForce::calcForceMakingTree

PS::TreeForForce::calcForceMakingTree

```
template<class Tfunc_ep_ep>
void PS::TreeForForce::calcForceMakingTree
             (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                      const PS::S32,
                                      TEpj *,
                                      const PS::S32,
                                      TResult *),
              DomainInfo & dinfo
              const bool clear=true,
              const PS::INTERACTION_LIST_MODE list_mode = PS::MAKE_LIST,
              const bool flag_serialize=false);
```

- **引数**

pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TEpj *型、PS::S32 型、TResult *型。

dinfo: 入力。DomainInfo &型。領域クラスのオブジェクト。

clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

list_mode: 入力。PS::INTERACTION_LIST_MODE 型。相互作用リストを作成し相互作用計算を行うか、前回作成した相互作用リストを再利用し相互作用計算を行うかを決定するフラグ。PS::MAKE_LIST ならば新たに相互作用リストを作成する。この場合、次の相互作用計算時に相互作用リストの再利用はできず、新たに相互作用リストを作成する必要がある。PS::MAKE_LIST_FOR_REUSE は新たに相互作用リストを作成し相互作用計算を行う。この場合、次回の相互作用計算時に、今回作った相互作用リストを再利用し相互作用計算ができる。PS::REUSE_LIST ならば、前回 PS::MAKE_LIST_FOR_REUSE を選んだ際に作成した相互作用リストを再利用し相互作用計算を行う。デフォルトは PS::MAKE_LIST。PS::INTERACTION_LIST_MODE の詳細はセクション 6.8.3 を参照。

flag_serialize: 入力。const bool 型。粒子情報をシリアライズして送信するかを決定するフラグ。true で粒子をシリアライズする。デフォルトは false。

- **返値**

  なし

- **機能**

  これより前に相互作用ツリークラスのオブジェクトに読み込まれた粒子群クラスのオブジェクトの粒子すべての相互作用を計算する。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY の場合に限る。PS::TreeForForce::calcForceAllAndWriteBack から粒子群クラスのオブジェクトの読込と計算結果の書き戻しがなくなったもの。

  PS::INTERACTION_LIST_MODE により相互作用リストを新たに作るか再利用するかを選ぶことができる。相互作用リストの再利用している間は PS::ParticleSystem::exchangeParticle() によって粒子の交換を行ってはならない。また、粒子データの消去や粒子配列の順番の並び替えを行ってはならない。

  LET をシリアライズして送信する場合は EPJ にメンバ関数 pack と unPack を定義し (詳しくはセクション 7.4.4.5)、flag_serialize を true にする (この機能は未実装)。

```
template<class Tfunc_ep_ep,
         class Tfunc_sp_ep>
void PS::TreeForForce::calcForceMakingTree
             (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 *),
              DomainInfo & dinfo
              const bool clear=true,
              const PS::INTERACTION_LIST_MODE list_mode = PS::MAKE_LIST,
              const bool flag_serialize=false);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TEpj *型、PS::S32 型、TResult *型。

  pfunc_sp_ep: 入力。返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TSpj *型、PS::S32 型、TResult *型。

  dinfo: 入力。DomainInfo &型。領域クラスのオブジェクト。

  clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

  list_mode: 入力。PS::INTERACTION_LIST_MODE 型。相互作用リストを作成し相互作用計算を行うか、前回作成した相互作用リストを再利用し相互作用計算を行うかを決定するフラグ。PS::MAKE_LIST ならば新たに相互作用リストを作成する。この場合、次の相互作用計算時に相互作用リストの再利用はできず、新たに相互作用リストを作成する必要がある。PS::MAKE_LIST_FOR_REUSE は新たに相互作用リストを作成し相互作用計算を行う。この場合、次回の相互作用計算時に、今回作った相互作用リストを再利用し相互作用計算ができる。PS::REUSE_LIST ならば、前回 PS::MAKE_LIST_FOR_REUSE を選んだ際に作成した相互作用リストを再利用し相互作用計算を行う。デフォルトは PS::MAKE_LIST。PS::INTERACTION_LIST_MODE の詳細はセクション 6.8.3 を参照。

flag_serialize: 入力。const bool 型。粒子情報をシリアライズして送信するかを決定するフラグ。true で粒子をシリアライズする。デフォルトは false。

- **返値**

  なし

- **機能**

  これより前に相互作用ツリークラスのオブジェクトに読み込まれた粒子群クラスのオブジェクトの粒子すべての相互作用を計算する。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOFF の場合に限る。PS::TreeForForce::calcForceAllAndWriteBack から粒子群クラスのオブジェクトの読込と計算結果の書き戻しがなくなったもの。

  PS::INTERACTION_LIST_MODE により相互作用リストを新たに作るか再利用するかを選ぶことができる。相互作用リストの再利用している間は PS::ParticleSystem::exchangeParticle() によって粒子の交換を行ってはならない。また、粒子データの消去や粒子配列の順番の並び替えを行ってはならない。

  LET をシリアライズして送信する場合は EPJ と SPJ にメンバ関数 pack と unPack を定義し (詳しくはセクション 7.4.4.5,7.6.4.2)、flag_serialize を true にする (この機能は未実装)。

9.1.4.2.3.6 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,
              const PS::INTERACTION_LIST_MODE list_mode = PS::MAKE_LIST,
              const bool flag_serialize=false);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TEpj *型、PS::S32 型、TResult *型。

psys: 入力。Tpsys &型。相互作用の計算結果を書き戻したい粒子群クラスのオブジェクト。

clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

list_mode: 入力。PS::INTERACTION_LIST_MODE 型。相互作用リストを作成し相互作用計算を行うか、前回作成した相互作用リストを再利用し相互作用計算を行うかを決定するフラグ。PS::MAKE_LIST ならば新たに相互作用リストを作成する。この場合、次の相互作用計算時に相互作用リストの再利用はできず、新たに相互作用リストを作成する必要がある。PS::MAKE_LIST_FOR_REUSE は新たに相互作用リストを作成し相互作用計算を行う。この場合、次回の相互作用計算時に、今回作った相互作用リストを再利用し相互作用計算ができる。PS::REUSE_LIST ならば、前回 PS::MAKE_LIST_FOR_REUSE を選んだ際に作成した相互作用リストを再利用し相互作用計算を行う。デフォルトは PS::MAKE_LIST。PS::INTERACTION_LIST_MODE の詳細はセクション 6.8.3 を参照。

flag_serialize: 入力。const bool 型。粒子情報をシリアライズして送信するかを決定するフラグ。true で粒子をシリアライズする。デフォルトは false。

- **返値**

  なし

- **機能**

  これより前に相互作用ツリークラスのオブジェクトに構築されたグローバルツリーとそのモーメントをもとに、相互作用ツリークラスのオブジェクトに属する粒子すべての相互作用が計算され、さらにその結果が粒子群クラスのオブジェクト psys に書き戻される。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY の場合に限る。PS::TreeForForce::calcForceAllAndWriteBack から粒子群クラスのオブジェクトの読込、ローカルツリーの構築、グローバルツリーの構築、グローバルツリーのモーメントの計算がなくなったもの。

  PS::INTERACTION_LIST_MODE により相互作用リストを新たに作るか再利用するかを選ぶことができる。相互作用リストの再利用している間は PS::ParticleSystem::exchangeParticle() によって粒子の交換を行ってはならない。また、粒子データの消去や粒子配列の順番の並び替えを行ってはならない。

  LET をシリアライズして送信する場合は EPJ にメンバ関数 pack と unPack を定義し (詳しくはセクション 7.4.4.5)、flag_serialize を true にする (この機能は未実装)。

```
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,
              const PS::INTERACTION_LIST_MODE list_mode = PS::MAKE_LIST,
              const bool flag_serialize=false);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TEpj *型、PS::S32 型、TResult *型。

  pfunc_sp_ep: 入力。返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TSpj *型、PS::S32 型、TResult *型。

  psys: 入力。Tpsys &型。相互作用の計算結果を書き戻したい粒子群クラスのオブジェクト。

  clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

  list_mode: 入力。PS::INTERACTION_LIST_MODE 型。相互作用リストを作成し相互作用計算を行うか、前回作成した相互作用リストを再利用し相互作用計算を行うかを決定するフラグ。PS::MAKE_LIST ならば新たに相互作用リストを作成する。この場合、次の相互作用計算時に相互作用リストの再利用はできず、新たに相互作用リストを作成する必要がある。PS::MAKE_LIST_FOR_REUSE は新たに相互作用リストを作成し相互作用計算を行う。この場合、次回の相互作用計算時に、今回作った相互作用リストを再利用し相互作用計算ができる。PS::REUSE_LIST ならば、前回 PS::MAKE_LIST_FOR_REUSE を選んだ際に作成した相互作用リストを再利用し相互

作用計算を行う。デフォルトは PS::MAKE_LIST。PS::INTERACTION_LIST_MODE の詳細はセクション 6.8.3 を参照。

flag_serialize: 入力。const bool 型。粒子情報をシリアライズして送信するかを決定するフラグ。true で粒子をシリアライズする。デフォルトは false。

- **返値**

  なし

- **機能**

  これより前に相互作用ツリークラスのオブジェクトに構築されたグローバルツリーとそのモーメントをもとに、相互作用ツリークラスのオブジェクトに属する粒子すべての相互作用が計算され、さらにその結果が粒子群クラスのオブジェクト psys に書き戻される。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOFF の場合に限る。
  PS::TreeForForce::calcForceAllAndWriteBack から粒子群クラスのオブジェクトの読込、ローカルツリーの構築、グローバルツリーの構築、グローバルツリーのモーメントの計算がなくなったもの。

  PS::INTERACTION_LIST_MODE により相互作用リストを新たに作るか再利用するかを選ぶことができる。相互作用リストの再利用している間は PS::ParticleSystem::exchangeParticle() によって粒子の交換を行ってはならない。また、粒子データの消去や粒子配列の順番の並び替えを行ってはならない。

  LET をシリアライズして送信する場合は EPJ と SPJ にメンバ関数 pack と unPack を定義し (詳しくはセクション 7.4.4.5,7.6.4.2)、flag_serialize を true にする (この機能は未実装)。

9.1.4.2.4 ネイバーリスト

9.1.4.2.4.1 getNeighborListOneParticle

getNeighborListOneParticle

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

- **引数**

  ptcl: 入力。Tptcl &型。近傍粒子を求めたい粒子。

  epj: 出力。EPJ * &型。近傍粒子の配列の先頭ポインタ。

- **返値**

  PS::S32 &型。近傍粒子の個数。

- **機能**

  呼び出し元のツリー構造を使って、ptcl の近傍粒子の配列の先頭ポインタを epj に与え、近傍粒子数を返す。epj は EPJ 型の粒子データの配列 (粒子へのポインタの配列ではない) の先頭へのポインタであり、FDPS が内部にもっているバッファ領域を指す。このため、ユーザはこのポインタに対して free() や delete() をしてはならない。この関数はスレッドセーフである。すなわち、スレッド毎に別のバッファ領域をもっている。なお、この領域はスレッド毎に 1 つしかないため、同一スレッドでこの関数を呼び出すたびに上書きされる。

  これを使うのは PS::SEARCH_MODE 型が

  - PS::SEARCH_MODE_GATHER
  - PS::SEARCH_MODE_SCATTER
  - PS::SEARCH_MODE_SYMMETRY
  - PS::SEARCH_MODE_LONG_SCATTER
  - PS::SEARCH_MODE_LONG_SYMMETRY
  - PS::SEARCH_MODE_LONG_GATHER (未実装)
  - PS::SEARCH_MODE_LONG_CUTOFF_GATHER (未実装)
  - PS::SEARCH_MODE_LONG_CUTOFF_SCATTER (未実装)
  - PS::SEARCH_MODE_LONG_CUTOFF_SYMMETRY (未実装)

  の場合に限る。ptcl のメンバ関数には FP と同様に PS::F64vec getPos() が必要である。PS::SEARCH_MODE 型が

  - PS::SEARCH_MODE_GATHER
  - PS::SEARCH_MODE_SYMMETRY
  - PS::SEARCH_MODE_LONG_SYMMETRY
  - PS::SEARCH_MODE_LONG_GATHER (未実装)
  - PS::SEARCH_MODE_LONG_CUTOFF_GATHER (未実装)
  - PS::SEARCH_MODE_LONG_CUTOFF_SYMMETRY (未実装)

  の場合にはさらに、ptcl の探査半径を返すメンバ関数 PS::F64 getRSearch() も必要となる。

9.1.4.2.5 時間計測

クラス内の時間計測関連の API の宣言は以下のようになっている。自クラスの主要なメソッドを呼び出すとそれにかかった時間をプライベートメンバの time_profile_の該当メンバに書き込む。メソッド clearTimeProfile() を呼ばない限り時間は足しあわされていく。

ソースコード 42: TreeForForce2

```
1 namespace ParticleSimulator {
2     template<class TSearchMode,
3               class TResult,
4               class TEpi,
```

```
5                class TEpj,
6                class TMomLocal,
7                class TMomGlobal,
8                class TSpj>
9      class TreeForForce{
10     public:
11         TimeProfile getTimeProfile();
12         void clearTimeProfile();
13     };
14 }
```

9.1.4.2.5.1 PS::TreeForForce::getTimeProfile

PS::TreeForForce::getTimeProfile

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

- **引数**

  なし。

- **返値**

  PS::TimeProfile 型。

- **機能**

  ローカルツリー構築、グローバルツリー構築、力の計算 (walk 込)、ローカルツリーのモーメント計算、グローバルツリーのモーメント計算、LET 構築、LET 交換にかかった時間（ミリ秒単位）を TimeProfile 型のメンバ変数の該当部分 make_local_tree, make_global_tree_, calc_force_, calc_moment_local_tree_, calc_moment_global_tree_, make_LET_1st_, make_LET_2nd_, exchange_LET_1st_, exchange_LET_2nd_に格納する。長距離力や散乱モードの様に LET 交換が 1 段階通信の場合は make_LET_2nd_, exchange_LET_2nd_に値は格納されない。

9.1.4.2.5.2 PS::TreeForForce::clearTimeProfile

PS::TreeForForce::clearTimeProfile

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

- **引数**

  なし。

- **返値**

  なし。

- **機能**

  相互作用ツリークラスのTimeProfile型のプライベートメンバ変数のメンバ変数make_local_tree, make_global_tree_, calc_force_, calc_moment_local_tree_, calc_moment_global_tree_, make_LET_1st_, make_LET_2nd_, exchange_LET_1st_, exchange_LET_2nd_の値を0クリアする。

9.1.4.2.6 情報取得

クラス内の情報取得関連のAPIの宣言は以下のようになっている。自クラスの主要なメソッドを呼び出すとそれにかかった時間をプライベートメンバのtime_profile_の該当メンバに書き込む。メソッドclearTimeProfile()を呼ばない限り時間は足しあわされていく。

ソースコード 43: TreeForForce2

```
1  namespace ParticleSimulator {
2      template<class TSearchMode,
3               class TResult,
4               class TEpi,
5               class TEpj,
6               class TMomLocal,
7               class TMomGlobal,
8               class TSpj>
9      class TreeForForce{
10     public:
11         TimeProfile getTimeProfile();
12         void clearTimeProfile();
13         Count_t getNumberOfInteractionEPEPLocal();
14         Count_t getNumberOfInteractionEPSPLocal();
15         Count_t getNumberOfInteractionEPEPGlobal();
16         Count_t getNumberOfInteractionEPSPGlobal();
17         void clearNumberOfInteraction();
18         S64 getUsedMemorySizeTotal();
19     };
20 }
```

9.1.4.2.6.1 PS::TreeForForce::getNumberOfInteractionEPEPLocal

PS::TreeForForce::getNumberOfInteractionEPEPLocal

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

- **引数**

  なし。

- **返値**

  PS::Count_t 型。

- **機能**

  自プロセス内で計算した EPI と EPJ の相互作用数を返す。

9.1.4.2.6.2　PS::TreeForForce::getNumberOfInteractionEPEPGlobal

PS::TreeForForce::getNumberOfInteractionEPEPGlobal

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

- **引数**

  なし。

- **返値**

  PS::Count_t 型。

- **機能**

  全プロセス内で計算した EPI と EPJ の相互作用数を返す。

9.1.4.2.6.3　PS::TreeForForce::getNumberOfInteractionEPSPLocal

PS::TreeForForce::getNumberOfInteractionEPSPLocal

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

- **引数**

  なし。

- **返値**

  PS::Count_t 型。

- **機能**

  自プロセス内で計算した EPI と SPJ の相互作用数を返す。

9.1.4.2.6.4　PS::TreeForForce::getNumberOfInteractionEPSPGlobal

PS::TreeForForce::getNumberOfInteractionEPSPGlobal

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

- **引数**

  なし。

- **返値**

  PS::Count_t 型。

- **機能**

  全プロセスで計算した EPI と SPJ の相互作用数を返す。

9.1.4.2.6.5　PS::TreeForForce::clearNumberOfInteraction

PS::TreeForForce::clearNumberOfInteraction

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

- **引数**

  なし。

- **返値**

  なし。

- **機能**

  EP-EP,EP-SP の local,global の相互作用数を 0 クリアする。

9.1.4.2.6.6　PS::TreeForForce::getNumberOfWalkLocal

PS::TreeForForce::getNumberOfWalkLocal

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

- **引数**

  なし。

- **返値**

  PS::Count_t 型。

- **機能**

  自プロセスでの相互作用計算時の tree walk 数を返す。

9.1.4.2.6.7　PS::TreeForForce::getNumberOfWalkGlobal

PS::TreeForForce::getNumberOfWalkGlobal

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

- **引数**

  なし。

- **返値**

  PS::S64。

- **機能**

  全プロセスでの相互作用計算時の tree walk 数を返す。

9.1.4.2.6.8　PS::TreeForForce::getUsedMemorySize

PS::TreeForForce::getUsedMemorySize

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

- **引数**

  なし。

- **返値**

  PS::S64。

- **機能**

  対象のオブジェクトが使用しているメモリー量を Byte 単位で返す。

9.1.4.2.7　粒子 id から EPJ を取得する

9.1.4.2.7.1　getEpjFromId

getEpjFromId

```
EPJ * PS::TreeForForce::getEpjFromId(const PS::S64 id)
```

- **引数**

  id: 入力。const PS::S64 型。取得したい粒子の id(EPJ のメンバ関数 getId() で返す値)。

- **返値**

  EPJ *型:id に対応する EPJ の粒子ポインター。

- **機能**

  EPJ がメンバ関数 getId() を持つ場合に使用可能。引数 id と getId() で返した値が同じ EPJ のポインタを返す。対応する EPJ がない場合は NULL を返す。また、複数の EPJ が同じ id を持つ場合結果は保証されない。getId() についてはセクション 7.4.4.4.1 を参照。

9.1.5 通信用データクラス

本節では、通信用データクラスについて記述する。このクラスはノード間通信のための情報の保持や実際の通信を行うモジュールである。このクラスはシングルトンパターンとして管理されており、オブジェクトの生成は必要としない。ここではこのモジュールの API を記述する。

9.1.5.1 API

このモジュールの API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 44: Communication

```
1  namespace ParticleSimulator {
2      class Comm{
3      public:
4          static S32 getRank();
5          static S32 getNumberOfProc();
6          static S32 getRankMultiDim(const S32 id);
7          static S32 getNumberOfProcMultiDim(const S32 id);
8          static bool synchronizeConditionalBranchAND
9                      (const bool local);
10         static bool synchronizeConditionalBranchOR
11                     (const bool local);
12         template<class T>
13         static T getMinValue(const T val);
14         template<class Tfloat, class Tint>
15         static void getMinValue(const Tfloat f_in,
16                                 const Tint i_in,
```

```
17                                  Tfloat & f_out,
18                                  Tint & i_out);
19          template<class T>
20          static T getMaxValue(const T val);
21          template<class Tfloat, class Tint>
22          static void getMaxValue(const Tfloat f_in,
23                                  const Tint i_in,
24                                  Tfloat & f_out,
25                                  Tint & i_out );
26          template<class T>
27          static T getSum(const T val);
28          template<class T>
29          static void broadcast(T * val,
30                                const S32 n,
31                                const S32 src=0);
32      };
33  }
```

9.1.5.1.1 PS::Comm::getRank

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

- **引数**

  なし。

- **返り値**

  PS::S32 型。全プロセス中でのランクを返す。

9.1.5.1.2 PS::Comm::getNumberOfProc

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

- **引数**

  なし。

- **返り値**

  PS::S32 型。全プロセス数を返す。

9.1.5.1.3　PS::Comm::getRankMultiDim

```
static PS::S32 PS::Comm::getRankMultiDim(const PS::S32 id);
```

- **引数**

  id: 入力。const PS::S32 型。軸の番号。x 軸:0, y 軸:1, z 軸:2。

- **返り値**

  PS::S32 型。id 番目の軸でのランクを返す。2 次元の場合、id=2 は 1 を返す。

9.1.5.1.4　PS::Comm::getNumberOfProcMultiDim

```
static PS::S32 PS::Comm::getNumberOfProcMultiDim(const PS::S32 id);
```

- **引数**

  id: 入力。const PS::S32 型。軸の番号。x 軸:0, y 軸:1, z 軸:2。

- **返り値**

  PS::S32 型。id 番目の軸のプロセス数を返す。2 次元の場合、id=2 は 1 を返す。

9.1.5.1.5　PS::Comm::synchronizeConditionalBranchAND

```
static bool PS::Comm::synchronizeConditionalBranchAND(const bool local)
```

- **引数**

  local: 入力。const bool 型。

- **返り値**

  bool 型。全プロセスで local の論理積を取り、結果を返す。

9.1.5.1.6　PS::Comm::synchronizeConditionalBranchOR

```
static bool PS::Comm::synchronizeConditionalBranchOR(const bool local);
```

- **引数**

  local: 入力。const bool 型。

- **返り値**

  bool 型。全プロセスで local の論理和を取り、結果を返す。

9.1.5.1.7　PS::Comm::getMinValue

```
template <class T>
static T PS::Comm::getMinValue(const T val);
```

- **引数**

  val: 入力。const T 型。

- **返り値**

  T 型。全プロセスで val の最小値を取り、結果を返す。

```
template <class Tfloat, class Tint>
static void PS::Comm::getMinValue(const Tfloat f_in,
                                  const Tint i_in,
                                  Tfloat & f_out,
                                  Tint & i_out);
```

- **引数**

  f_in: 入力。const Tfloat 型。

  i_in: 入力。const Tint 型。

  f_out: 出力。Tfloat 型。全プロセスで f_in の最小値を取り、結果を返す。

  i_out: 出力。Tint 型。f_out に伴う ID を返す。

- **返り値**

  なし。

9.1.5.1.8　PS::Comm::getMaxValue

```
template <class T>
static T PS::Comm::getMaxValue(const T val);
```

- **引数**

  val: 入力。const T 型。

- **返り値**

  T 型。全プロセスで val の最大値を取り、結果を返す。

```
template <class Tfloat, class Tint>
static void PS::Comm::getMaxValue(const Tfloat f_in,
                                  const Tint i_in,
                                  Tfloat & f_out,
                                  Tint & i_out);
```

- **引数**

  f_in: 入力。const Tfloat 型。

  i_in: 入力。const Tint 型。

  f_out: 出力。Tfloat 型。全プロセスで f_in の最大値を取り、結果を返す。

  i_out: 出力。Tint 型。f_out に伴う ID を返す。

- **返り値**

  なし。

9.1.5.1.9　PS::Comm::getSum

```
template <class T>
static T PS::Comm::getSum(const T val);
```

- **引数**

  val: 入力。const T 型。

- **返り値**

  T 型。全プロセスで val の総和を取り、結果を返す。

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);
```

- **引数**

  val: 入力。T *型。

  n: 入力。const PS::S32 型。T 型変数の数。

  src: 入力。const PS::S32 型。放送するプロセスランク。デフォルトのランクは 0。

- **返り値**

  なし。

- **機能**

  プロセスランク src のプロセスが n 個の T 型変数を全プロセスに放送する。

9.1.6 その他関数

本節では、名前空間 ParticleSimulator 以下に直に定義されている関数について述べる。

9.1.6.1 時間計測

9.1.6.1.1 PS::GetWtime

```
inline PS::F64 PS::GetWtime();
```

- **引数**

  なし。

- **返り値**

  PS::F64 型。ウォールクロックタイムを返す。単位は秒。

9.2 拡張機能

9.2.1 概要

本節では、FDPS の拡張機能について記述する。拡張機能には 2 つのモジュールがあり、Particle Mesh クラス、x86 版 Phantom-GRAPE がある。この 2 つのモジュールについて記述する。

9.2.2 Particle Mesh クラス

本節では、Particle Mesh クラスについて記述する。このクラスは Particle Mesh 法を用いて粒子の相互作用を計算するモジュールである。メッシュからの力は S-2 型の関数によってカットオフされており、カットオフ半径はメッシュ間隔の 3 倍で固定されている。カットオフ関数径を変更する事は出来ない。また、Particle Mesh クラスに送る粒子の座標は 0 以上 1 未満の値に規格化して置かなければならない。以下に、オブジェクトの生成方法、API、使用済マクロ、使いかたについて記述する。

9.2.2.1 オブジェクトの生成

Particle Mesh クラスは以下のように宣言されている。

ソースコード 45: ParticleMesh0

```
1 namespace ParticleSimulator {
2     namespace ParticleMesh {
3         class ParticleMesh;
4     }
5 }
```

Particle Mesh クラスのオブジェクトの生成は以下のように行う。ここでは pm というオブジェクトを生成している。

```
PS::PM::ParticleMesh pm;
```

9.2.2.2 API

Particle Mesh クラスには初期設定関連の API、低レベル API、高レベル API がある。以下、各節に分けて記述する。

9.2.2.2.1 初期設定

初期設定関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 46: ParticleMesh1

```
1 namespace ParticleSimulator {
2     namespace ParticleMesh {
3         class ParticleMesh{
4             ParticleMesh();
5         };
6     }
7 }
```

9.2.2.2.1.1 コンストラクタ

コンストラクタ

```
void PS::PM::ParticleMesh::ParticleMesh();
```

- **引数**

  なし

- **返値**

  なし

- **機能**

  Particle Mesh クラスのオブジェクトを生成する。

9.2.2.2.2 低レベル *API*

低レベル API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 47: ParticleMesh1

```
1  namespace ParticleSimulator {
2      namespace ParticleMesh {
3          class ParticleMesh{
4              template<class Tdinfo>
5              void setDomainInfoParticleMesh
6                          (const Tdinfo & dinfo);
7              template<class Tpsys>
8              void setParticleParticleMesh
9                          (const Tpsys & psys,
10                          const bool clear=true);
11             void calcMeshForceOnly();
12             F32vec getForce(F32vec pos);
13             F32 getPotential(F32vec pos);
14         };
15     }
16 }
```

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);
```

- **引数**

  dinfo: 入力。Tdinfo &型。領域クラスのオブジェクト。

- **返値**

  なし

- **機能**

  領域情報を読み込む。

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);
```

- **引数**

  psys: 入力。Tpsys & 型。粒子群クラスのオブジェクト。

  clear: 入力。const bool 型。これまで読込んだ粒子情報をクリアするかどうか決定するフラグ。true ならばクリアする。デフォルトは true。

- **返値**

  なし

- **機能**

  粒子情報を粒子群クラスのオブジェクトから読み込む。

9.2.2.2.2.3 PS::PM::ParticleMesh::calcMeshForceOnly

PS::PM::ParticleMesh::calcMeshForceOnly

```
void PS::PM::ParticleMesh::calcMeshForceOnly();
```

- **引数**

  なし

- **返値**

  なし

- **機能**

  メッシュ上の力を計算する。

9.2.2.2.2.4 PS::PM::ParticleMesh::getForce

PS::PM::ParticleMesh::getForce

```
PS::F32vec PS::PM::ParticleMesh::getForce(PS::F32vec pos);
```

- **引数**

  pos: 入力。PS::F32vec 型。メッシュからの力を計算したい位置。

- **返値**

  PS::F32vec 型。位置 pos におけるメッシュからの力。

- **機能**

  位置 pos でのメッシュからの力を返す。この関数は thread-safe である。

9.2.2.2.2.5 PS::PM::ParticleMesh::getPotential

PS::PM::ParticleMesh::getPotential

```
PS::F32 PS::PM::ParticleMesh::getPotential(PS::F32vec pos);
```

- **引数**

  pos: 入力。PS::F32vec 型。メッシュからのポテンシャルを計算したい位置。

- **返値**

  PS::F32 型。位置 pos におけるメッシュからポテンシャル。

- **機能**

  位置 pos でのメッシュからのポテンシャルを返す。この関数は thread-safe である。

9.2.2.2.3 高レベル API

高レベル API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 48: ParticleMesh1

```
1  namespace ParticleSimulator {
2      namespace ParticleMesh {
3          class ParticleMesh{
4              template<class Tpsys,
5                       class Tdinfo>
6              void calcForceAllAndWriteBack
7                           (Tpsys & psys,
8                            const Tdinfo & dinfo);
9          };
10     }
11 }
```

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);
```

- **引数**

  psys: 入力であり出力。Tpsys & 型。粒子群クラスのオブジェクト。

  dinfo: 入力。const Tdinfo &型。領域クラスのオブジェクト。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys に含まれる粒子間のメッシュ力を計算し、その結果を psys に返す。

9.2.2.3 使用済マクロ

このモジュールでは多くのマクロを使っている。これらを別のマクロとして使用した場合にプログラムが正しく動作する保証はない。ここでは使用されているマクロをアルファベティカルに列挙する。

- 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 Particle Mesh クラスの使いかた

Particle Mesh クラスを使うには以下の４つのことを行う必要がある。

1. Particle Mesh クラスのコンパイル
2. Particle Mesh クラスを使った FDPS コードの記述
3. FDPS コードのコンパイル

以下、詳細に記述する。

9.2.2.4.1 Particle Mesh クラスのコンパイル

以下のように行う。ディレクトリ src の下のディレクトリ particle_mesh の Makefile を適切に編集して make する。編集すべきことは以下の 2 点である。

- INCLUDE_FFTW に FFTW のヘッダファイルがあるディレクトリを記述する
- param_fdps.h の中の SIZE_OF_MESH (1 次元方向のメッシュの数) を設定。推奨値は $N^{1/3}/2$(N は粒子数)。

うまく行けば、同じディレクトリにライブラリ libpm.a とヘッダファイル particle_mesh.hpp ができる。

9.2.2.4.2 FDPS コードを記述

以下のように行う。

- 上でできたヘッダファイルを include する
- PM を計算したい粒子クラスに以下のメンバ関数を加える (この粒子クラスのクラス名を FP とする)
  - void FP::copyFromForceParticleMesh(const PS::F32vec & force)。この中で force を好きなメンバ変数にセットする。
  - PS::F64 FP::getChargeParticleMesh()。この中で質量を返す。
- このクラスのオブジェクトを生成するときに、PS::PM::ParticleMesh とする

9.2.2.4.3 FDPS コードのコンパイル

上で記述した FDPS コードをコンパイルするには以下のことを行う必要がある。

- ヘッダファイル particle_mesh.hpp のあるディレクトリへのパスを指定する
- ライブラリ libpm.a とリンクする
- FFTW のヘッダファイルがあるディレクトリへのパスを指定する
- FFTW のライブラリとリンクする

9.2.2.4.4 注意事項

Particle Mesh クラスはプロセス数が 2 以上でないと、動かないことに注意。

9.2.3 x86 版 phantom-GRAPE

低精度N体シミュレーション用、低精度カットオフ付き相互作用計算用のPhantom-GRAPEについてはTanikawa et al.(2012, New Astronomy, 19, 74)を、高精度N体シミュレーション用のPhantom-GRAPEについてはTanikawa et al.(2012, New Astronomy, 17, 82)を参照のこと。

10 エラー検出

10.1 概要

FDPSではのコンパイル時もしくは実行時のエラー検出機能を備えている。ここでは、FDPSで検出可能なエラーとその場合の対処について記述する。ただし、ここに記述されていないエラーも起こる可能性がある。（その場合は開発者に報告していただけると助かります。）

10.2 コンパイル時のエラー

10.3 実行時のエラー

FDPSが実行時エラーを検出すると標準エラー出力に以下のような書式でメッセージを出力し、PS::Abort(-1) によってプログラムを終了する。

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

- *ERROR MESSAGE*

  エラーメッセージ

- *FUNCTION NAME*

  エラーが起こった関数の名前

- *LINE NUMBER*

  エラーが起こった行番号

- *FILE NAME*

  エラーが起こったファイルの名前

以下、FDPSで用意されている実行時エラーメッセージを列挙していく。

10.3.1 PS_ERROR: can not open input file

ユーザーがFDPSのファイル入力関数を使っており、ユーザーが指定した入力ファイルがなかった場合に表示される。
エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

input file: “入力ファイル名”

10.3.2 PS_ERROR: can not open output file

ユーザーが FDPS のファイル出力関数を使っており、ユーザーが指定した出力ファイルがなかった場合に表示される。

エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

output file: “出力ファイル名”

10.3.3 PS_ERROR: Do not initialize the tree twice

同一のツリーオブジェクトに対して関数 PS::TreeForForce::initialize(...) を 2 度呼び出した場合に表示される。同一のツリーオブジェクトに対して PS::TreeForForce::initialize(...) の呼び出しを一回にする。

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

長距離力モードでツリーのオープニングクライテリオンに負の値が入力された場合に表示される。関数 PS::TreeForForce::initialize(...) を使ってオープニングクライテリオンに 0 以上の値を指定する必要がある。

エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

theta_= “入力されたオープニングクライテリオンの値”
SEARCH_MODE: “対象となるツリーのサーチモードの型名”
Force: “対象となるツリーのフォースの型名”
EPI: “対象となるツリーの EPI の型名”
EPJ: “対象となるツリーの EPJ の型名”
SPJ: “対象となるツリーの SPJ の型名”

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

長距離力モードでツリーのリーフセルの最大粒子数に負の値が入力された場合に表示される。関数 PS::TreeForForce::initialize(...) を使ってリーフセルの最大粒子数に正の整数を指定する必要がある。

エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

n_leaf_limit_=" 入力されたリーフセルの最大粒子数"
SEARCH_MODE: "対象となるツリーのサーチモードの型名"
Force: "対象となるツリーのフォースの型名"
EPI: "対象となるツリーの EPI の型名"
EPJ: "対象となるツリーの EPJ の型名"
SPJ: "対象となるツリーの SPJ の型名"

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

長距離力モードでツリーのリーフセルの最大粒子数が i 粒子グループの粒子の最大数より大きかった場合に表示される。関数 PS::TreeForForce::initialize(...) を使って i 粒子グループの最大粒子数をリーフセルの最大粒子数以上にする必要がある。

エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

n_leaf_limit_=" 入力されたリーフセルの最大粒子数"
n_grp_limit_=" 入力された i 粒子グループの内の最大粒子数"
SEARCH_MODE: "対象となるツリーのサーチモードの型名"
Force: "対象となるツリーのフォースの型名"
EPI: "対象となるツリーの EPI の型名"
EPJ: "対象となるツリーの EPJ の型名"
SPJ: "対象となるツリーの SPJ の型名"

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

FDPS では 1 プロセスあたりに扱える粒子数は 2G(G=2^{30}) であり、それ以上の粒子を確保しようとした場合に表示される。この場合、プロセス数を増やすなどして、1 プロセスあたりの粒子数を減らす必要がある。

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

開放境界以外の条件下でカットオフなし長距離力を設定した場合に表示される。カットオフなし長距離力の計算では必ず、開放境界条件を使う。無限遠までの粒子からの力を計算したい場合はカットオフあり長距離力の計算を FDPS で行い、カットオフ外からの力の計算は外部モジュールである Particle Mesh を使う事ができる。

10.3.9 PS_ERROR: A particle is out of root domain

ユーザーが *PS::DomainInfo::setRootDomain(...)* 関数を用いてルートドメインを設定しており、粒子がそのルートドメインからはみ出していた場合に表示される。周期境界条件の場合はユーザーは粒子をルートドメイン内に収まるように位置座標をシフトする必要がある。FDPS では粒子をルートドメイン内にシフトする関数
PS::ParticleSystem::adjustPositionIntoRootDomain(...) を用意しており、それを使うこともできる。

エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

position of the particle=” 粒子の座標”
position of the root domain=” ルートドメインの座標”

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

ユーザーが PS::DomainInfo::initialize(...) 関数を用いて平滑化係数に 0 未満もしくは 1 を超える値を設定した場合に表示される。エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

The smoothing factor of an exponential moving average=” 平滑化係数の値”

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

ユーザーが PS::DomainInfo::setPosRootDomain(...) 関数を用いてルートドメインを設定した時に、ユーザーが設定した小さい側の頂点の座標の任意の成分が大きい側の頂点の対応する座標の値よりも大きかった場合に表示される。エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

The coordinate of the low vertex of the rood domain=” 小さい側の頂点の座標”
The coordinate of the high vertex of the rood domain=” 大きい側の頂点の座標”

10.3.12 PS_ERROR: Vector invalid accesse

`Vector` 型の `[]` 演算子で定義されている範囲外の成分にアクセスを行った場合に表示される。エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

Vector element=” 指定した成分” is not valid

11　よく知られているバグ

12 限界

- FDPS 独自の整数型を用いる場合、GCC コンパイラと K コンパイラでのみ正常に動作することが保証されている。

13　ユーザーサポート

FDPSを使用したコード開発に関する相談は以下のメールアドレスfdps-support@mail.jmlab.jpで受け付けています。以下のような場合は各項目毎の対応をお願いします。

13.1　コンパイルできない場合

ユーザーには以下の情報提供をお願いします。

- コンパイル環境
- コンパイル時に出力されるエラーメッセージ
- ソースコード (可能ならば)

13.2　コードがうまく動かない場合

ユーザーには以下の情報提供をお願いします。

- 実行環境
- 実行時に出力されるエラーメッセージ
- ソースコード (可能ならば)

13.3　その他

思い通りの性能がでない場合やその他の相談なども、上のメールアドレスにお知らせください。

14　ライセンス

MIT ライセンスに準ずる。標準機能のみ使用する場合は、Iwasawa et al. (PASJ, 68, 54)、Namekata et al. (in prep) の引用をお願いします。

拡張機能の Particle Mesh クラスは GreeM コード (開発者:石山智明、似鳥啓吾)　(Ishiyama, Fukushige & Makino 2009, Publications of the Astronomical　Society of Japan, 61, 1319; Ishiyama, Nitadori & Makino, 2012 SC'12　Proceedings of the International Conference on High Performance　Computing, Networking Stroage and Analysis, No. 5) のモジュールを使用している。GreeM コードは Yoshikawa & Fukushige (2005, Publications of the　Astronomical Society of Japan, 57, 849) で書かれたコードをベースとしている。Particle Mesh クラスを使用している場合は、上記 3 つの文献の引用をお願いします。

拡張機能のうち x86 版 Phantom-GRAPE を使用する場合は Tanikawa et al.(2012, New Astronomy, 17, 82) と Tanikawa et al.(2012, New Astronomy, 19, 74) の引用をお願いします。

15 変更履歴

- 2015.03.13
  - 様々なユーザー定義クラスのメンバ関数で getRsearch となっていたものを getRSearch と訂正 (節 7)
  - 異常終了させる関数 PS::Abort についての記述を追加 (節 8)
- 2015.03.17
  - バージョン 1.0 リリース
- 2015.03.18
  - Particle Mesh クラス関連のライセンス事項を修正。
- 2015.03.20
  - PS::Comm::broadcast の記述が抜けていたので追加。
- 2015.04.01
  - Particle Mesh クラスはプロセス数が 2 以上でないと動かないという注意事項を追加。
- 2015.10.07
  - PM 法についての記述を追加。セクション 9.2.2。
- 2015.12.01
  - Multi Walk についての記述を追加。セクション A.10、A.11、9.1.4.2.3.1、7.11、7.12。
- 2016.02.07
  - getNeighborListOneParticle についての記述を追加。セクション 9.1.4.2.4。
  - それに伴い、PS:: MomentMonopoleScatter、PS::MomentQuadrupoleScatter の記述を追加。セクション A.4.2.2。
  - さらに、これらのラッパーである MonopoleWithScatter および QuadrupoleWithScatter の記述をセクション 9.1.4.1.1 に追加。
- 2016.09.13
  - 粒子の追加、削除の API をセクション 9.1.3.2.5 に追加。
  - Vector 型の範囲外アクセスについての記述をセクション 10.3.12 に追加。

- 2016.10.11
  - Vector 型の演算子"[]" の実装変更。それによる注意事項を追加セクション 6.4.4 に追加。
- 2016.11.04
  - ファイル読み込み API をセクション 9.1.3.2.3.1,9.1.3.2.3.2,9.1.3.2.3.3,9.1.3.2.3.4 に追加
- 2017.07.11
  - 相互作用リストの再利用に関する記述をセクション 9.1.4.2.3 に追加。
- 2017.09.06
  - 粒子群クラスの粒子配列の並び替えに関する記述をセクション 9.1.3.2.7.3 に追加。
  - 粒子の id 番号から EPJ の情報を取り出す関数に関する記述をセクション 7.4.4.4.1,9.1.4.2.7 に追加。
- 2017.11.08
  - FDPS 4.0 リリース
- 2018.01.16
  - FDPS 4.0b リリース
    - メモリ割付に関するバグを修正。
    - TreeForForce クラス等にデストラクタを追加。
- 2018.01.23
  - 粒子交換時に粒子データをシリアライズする方法をセクション 7.2.4.6, 7.4.4.5, 7.6.4.2, 9.1.3.2.4.1, 9.1.4.2.3 に追加。
- 2018.05.25
  - FDPS 4.0c リリース
    - Particle Mesh 拡張の実装から MPI の C++ biding を削除。
- 2018.06.13
  - FDPS 4.1 リリース
    - Tree 法で $\theta = 0$ の場合にも超粒子が発生しうる場合があるバグを修正。
    - オルソトープ型の記述を第 6.5.4 節に追加。

    * Moment クラスのセクションに、場合によっては必要なメンバ関数についての記述を追加。

- 2018.06.20
  - FDPS 4.1a リリース
    * 仕様書 (本書) の変更履歴のセクションの記載もれを修正。

- 2018.08.2
  - 新しい API `getBoundaryCondition` を追加
  - API `collectSampleParticle` の引数 `weight` の意味についての説明を追加

- 2018.12.7
  - FDPS 5.0a リリース
    * Moment クラス 及び SuperParticleJ クラスの節 (第 7.5,7.6 節) に、
      PS::SEARCH_MODE_LONG_SCATTER
      PS::SEARCH_MODE_LONG_SYMMETRY
      に対して用意されている既存のクラスの説明を追加。
    * FullParticle クラス 及び ヘッダクラスの節 (第 7.2, 7.8 節) に、ファイル入出力 API を使う場合に必要となるメンバ関数 readBinary 及び writeBinary についての説明を追加。同様、これらの実装例を、それぞれ第 A.1, A.7 節に追加。

- 2018.12.9
  - FDPS 5.0b リリース
    * 仕様書の API getNeighborListOneParticle の記述が不正確であった問題を修正 (ソースコードには変更ありません)。

- 2019.1.25
  - FDPS 5.0c リリース
    * 周期境界条件で、かつ、特定の粒子分布が与えられた場合に FDPS 内部の関数 LinkCell でエラーが発生する問題を修正。

- 2019.3.1
  - FDPS 5.0d リリース
    * 孤立境界条件及び全方向周期境界条件以外の境界条件で、ツリーのルートセルの設定が正しく行われない問題を修正。

    * FDPS バージョン5.0から5.0cにおいて、一部のSEARCH_MODEで、PS::TimeProfileのメンバ変数make_LET_1st, make_LET_2nd, exchange_LET_1st, exchange_LET_2ndに値が設定されない問題を修正。時間測定の範囲の変更のため、個々の変数の値はFDPS バージョン 4.1 以前のものと互換性がないが、これらの和については互換性がある。
    * FDPS が内部で使用するメモリプールのサイズが不足していた場合、セグメンテーション違反によりコードが停止してしまう問題を修正。
    * 本リリースからFDPS内でC++11の機能を使用している。そのため、C++コンパイラ、及び、CUDAを利用する場合にはCUDAコンパイラに適切なオプション (GNU gccであれば-std=c++11) をつける必要がある。

- 2019.3.7
  - TreeForForceLong⟨,,⟩::MonopoleWithCutoff に関する記述を改善。
- 2019.9.6
  - FDPS 5.0f リリース
    * コンパイル時にマクロPARTICLE_SIMULATOR_TWO_DIMENSIONを定義した場合、コンパイルエラーになる問題を修正。
- 2019.9.10
  - FDPS 5.0g リリース
    * コンパイル時にマクロPARTICLE_SIMULATOR_TWO_DIMENSIONを定義した場合、実行時エラーになる問題を修正。
- 2020.8.16
  - FDPS 6.0 リリース
    * PIKG を導入

A　ユーザー定義クラスの実装例

A.1　FullParticle クラス

A.1.1　概要

FullParticle クラスは粒子情報すべてを持つクラスであり、節 2.3 の手順 0 で、粒子群クラスに渡されるユーザー定義クラスの 1 つである。ユーザーはこのクラスに対して、どのようなメンバ変数、メンバ関数を定義してもかまわない。ただし、FDPS から FullParticle クラスの情報にアクセスする ために、ユーザーはいくつかの決まった名前のメンバ関数を定義する必要がある。以下、この節の前提、常に必要なメンバ関数と、場合によっては必要なメンバ関数について記述する。

A.1.2　前提

この節の中では、以下のように、FullParticle クラスとして FP というクラスを一例とする。FP という名前は自由に変えることができる。

```
class FP;
```

A.1.3　必要なメンバ関数

A.1.3.1　概要

常に必要なメンバ関数は FP::getPos と FP::copyFromForce である。FP::getPos は FullParticle の位置情報を FDPS に読み込ませるための関数で、FP::copyFromForce は計算された相互作用の結果を FullParticle に書き戻す関数である。これらのメンバ関数の記述例と解説を以下に示す。

A.1.3.2　FP::getPos

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

- **引数**

  なし

- **返値**

  PS::F32vec 型または PS::F64vec 型。FP クラスのオブジェクトの位置情報を保持したメンバ変数。

- **機能**

  FP クラスのオブジェクトの位置情報を保持したメンバ変数を返す。

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;
    }
};
```

- **前提**

  Force クラスは粒子の相互作用の計算結果を保持するクラス。

- **引数**

  force: 入力。const Force &型。粒子の相互作用の計算結果を保持。

- **返値**

  なし。

- **機能**

  粒子の相互作用の計算結果を FP クラスへ書き戻す。Force クラスのメンバ変数 acc, pot がそれぞれ FP クラスのメンバ変数 acceleration, potential に対応。

- **備考**

  Force クラスというクラス名とそのメンバ変数名は変更可能。FP のメンバ変数名は変更可能。メンバ関数 FP::copyFromForce の引数名は変更可能。

A.1.4 場合によっては必要なメンバ関数

A.1.4.1 概要

本節では、場合によっては必要なメンバ関数について記述する。相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合、粒子群クラスのファイル入出力 API を用いる場合、粒子群クラスの API である ParticleSystem::adjustPositionIntoRootDomain を用いる場合、拡張機能の Particle Mesh クラスを用いる場合について必要となるメンバ関数を記述する。

A.1.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合

A.1.4.2.1 FP::getRSearch

```
class FP {
public:
    PS::F64 search_radius;
    PS::F64 getRSearch() const {
        return this->search_radius;
    }
};
```

- **前提**

  FP クラスのメンバ変数 search_radius はある 1 つの粒子の近傍粒子を探す半径の大きさ。この search_radius のデータ型は PS::F32 型または PS::F64 型。

- **引数**

  なし

- **返値**

  PS::F32 型または PS::F64 型。 FP クラスのオブジェクトの近傍粒子を探す半径の大きさを保持したメンバ変数。

- **機能**

  FP クラスのオブジェクトの近傍粒子を探す半径の大きさを保持したメンバ変数を返す。

- **備考**

  FP クラスのメンバ変数 search_radius の変数名は変更可能。

A.1.4.3　粒子群クラスのファイル入出力 API を用いる場合

粒子群クラスのファイル入出力 API である ParticleSystem::readParticleAscii, ParticleSystem::writeParticleAscii, ParticleSystem::readParticleBinary, ParticleSystem::writeParticleBinary を使用するときにそれぞれ readAscii, writeAscii, readBinary, writeBinary というメンバ関数が必要となる (readAscii, writeAscii, readBinary, writeBinary 以外の名前を使うことも可能。詳しくは節 9.1.3.2.3 を参照)。以下、readAscii, writeAscii, readBinary, writeBinary の実装例について記述する。

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]);
    }
};
```

- **前提**

  粒子データの入力ファイルの１列目には FP クラスのメンバ変数 id を表すデータが、２列目にはメンバ変数 mass を表すデータが、３、４、５列めにはメンバ変数 pos の第１、２、３要素が、それ以降には列がないとする。ファイルの形式はアスキー形式とする。３次元直交座標系を選択したとする。

- **引数**

  fp: FILE *型。粒子データの入力ファイルを指すファイルポインタ。

- **返値**

  なし。

- **機能**

  粒子データの入力ファイルから FP クラスの id、mass、pos の情報を読み取る。

- **備考**

  なし。

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]);
    }
};
```

- **前提**

  粒子データの出力ファイルの１列目にはFPクラスのメンバ変数idを表すデータが、２列目にはメンバ変数massを表すデータが、３、４、５列めにはメンバ変数posの第１、２、３要素が、それ以降には列がないとする。ファイルの形式はアスキー形式とする。３次元直交座標系を選択したとする。

- **引数**

  fp: FILE *型。粒子データの出力ファイルを指すファイルポインタ。

- **返値**

  なし。

- **機能**

  粒子データの出力ファイルへFPクラスのメンバ変数id、mass、posの情報を書き出す。

- **備考**

  なし。

A.1.4.3.3　FP::readBinary

```
class FP {
public:
    PS::S32 id;
    PS::F64 mass;
    PS::F64vec pos;
    void readBinary(FILE *fp) {
        fread(&this->id, sizeof(PS::S32), 1, fp);
        fread(&this->mass, sizeof(PS::F64), 1, fp);
        fread(&this->pos, sizeof(PS::F64vec), 1, fp);
    }
};
```

- **前提**

  粒子データの入力ファイルには (複数の)FP クラスのデータが連続して、すなわち、任意の 2 つの FP クラスのデータ間に余分な隙間なく、格納されているとする。FP クラス 1 個分のデータは、メンバ変数 id, mass, pos の第 1、2、3 要素がこの順に連続して並んでいるものとし、fp の現在位置はある FP クラスのデータの先頭を指しているとする。ファイルの形式はバイナリー形式とする。3 次元直交座標系を選択したとする。

- **引数**

  fp: FILE *型。粒子データの入力ファイルを指すファイルポインタ。

- **返値**

  なし。

- **機能**

  粒子データの入力ファイルから FP クラスの id、mass、pos の情報を読み取る。

- **備考**

  なし。

A.1.4.3.4　FP::writeBinary

```
class FP {
public:
    PS::S32 id;
    PS::F64 mass;
    PS::F64vec pos;
    void writeBinary(FILE *fp) {
        fwrite(&this->id, sizeof(PS::S32), 1 , fp);
        fwrite(&this->mass, sizeof(PS::F64), 1, fp);
        fwrite(&this->pos, sizeof(PS::F64vec), 1, fp);
    }
};
```

- **前提**

  粒子データの出力ファイルには (複数の)FP クラスのデータが連続して、すなわち、任意の 2 つの FP クラスのデータ間に余分な隙間なく、格納されているとする。引数 fp が指す現在位置は最後に出力された FP クラスのデータの末尾を指しているとする。今、FP クラス 1 個分のデータを、メンバ変数 id, mass, pos の第 1、2、3 要素がこの順に連続して並ぶ形で出力したい。ファイルの形式はバイナリー形式とする。3 次元直交座標系を選択したとする。

- **引数**

  fp: FILE *型。粒子データの出力ファイルを指すファイルポインタ。

- **返値**

  なし。

- **機能**

  粒子データの出力ファイルへ FP クラスのメンバ変数 id、mass、pos の情報を書き出す。

- **備考**

  なし。

A.1.4.4 ParticleSystem::adjustPositionIntoRootDomain を用いる場合

A.1.4.4.1 FP::setPos

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

- **前提**

  FP クラスのメンバ変数 pos は 1 つの粒子の位置情報。この pos のデータ型は PS::F32vec または PS::F64vec。

- **引数**

  pos_new: 入力。const PS::F32vec または const PS::F64vec 型。FDPS 側で修正した粒子の位置情報。

- **返値**

  なし。

- **機能**

  FDPS が修正した粒子の位置情報を FP クラスのオブジェクトの位置情報に書き込む。

- **備考**

  FP クラスのメンバ変数 pos の変数名は変更可能。メンバ関数 FP::setPos の引数名 pos_new は変更可能。pos と pos_new のデータ型が異なる場合の動作は保証しない。

A.1.4.5 Particle Mesh クラスを用いる場合

Particle Mesh クラスを用いる場合には、メンバ関数 FP::getChargeParticleMesh と FP::copyFromForceParticleMesh を用意する必要がある。以下にそれぞれの規定を記述する。

A.1.4.5.1　FP::getChargeParticleMesh

```
class FP {
public:
    PS::F64 mass;
    PS::F64 getChargeParticleMesh() const {
        return this->mass;
    }
};
```

- **前提**

  FP クラスのメンバ変数 mass は１つの粒子の質量または電荷の情報を持つ変数。データ型は PS::F32 または PS::F64 型。

- **引数**

  なし。

- **返値**

  PS::F32 型または PS::F64 型。１つの粒子の質量または電荷の変数を返す。

- **機能**

  １つの粒子の質量または電荷を表すメンバ変数を返す。

- **備考**

  FP クラスのメンバ変数 mass の変数名は変更可能。

A.1.4.5.2　FP::copyFromForceParticleMesh

```
class FP {
public:
    PS::F64vec accelerationFromPM;
    void copyFromForceParticleMesh(const PS::F32vec & acc_pm) {
        this->accelerationFromPM = acc_pm;
    }
};
```

- **前提**

  FP クラスのメンバ変数 accelerationFromPM_pm は１つの粒子の Particle Mesh による力の情報を保持する変数。この accelerationFromPM_pm のデータ型は PS::F32vec または PS::F64vec。

- **引数**

  acc_pm: const PS::F32vec 型または const PS::F64vec 型。１つの粒子の Particle Mesh による力の計算結果。

- **返値**

  なし。

- **機能**

  １つの粒子の Particle Mesh による力の計算結果をこの粒子のメンバ変数に書き込む。

- **備考**

  FP クラスのメンバ変数 acc_pm の変数名は変更可能。メンバ関数 FP::copyFromForceParticleMesh の引数 acc_pm の引数名は変更可能。

A.2 EssentialParticleI クラス

A.2.1 概要

EssentialParticleI クラスは相互作用の計算に必要な i 粒子の情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。EssentialParticleI クラスは FullParticle クラス (節 7.2) のサブセットである。FDPS は、このクラスのデータにアクセスする必要がある。そのため、EssentialParticleI クラスはいくつかのメンバ関数を持つ必要がある。以下、この節の前提、常に必要なメンバ関数と、場合によっては必要なメンバ関数について記述する。

A.2.2 前提

この節の中では、EssentialParticleI クラスとして EPI というクラスを一例として使う。また、FullParticle クラスの一例として FP というクラスを使う。EPI, FP というクラス名は変更可能である。

EPI と FP の宣言は以下の通りである。

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

A.2.3 必要なメンバ関数

A.2.3.1 概要

常に必要なメンバ関数は EPI::getPos と EPI::copyfromFP である。EPI::getPos は EPI クラスの位置情報を FDPS に読み込ませるための関数で、EPI::copyFromFP は FP クラスの情報を EPI クラスに書きこむ関数である。これらのメンバ関数の記述例と解説を以下に示す。

A.2.3.2 EPI::getPos

```
class EPI {
public:
    PS::F64vec pos;
    PS::F64vec getPos() const {
        return this->pos;
    }
};
```

- **前提**

  EPI のメンバ変数 pos はある 1 つの粒子の位置情報。この pos のデータ型は PS::F64vec 型。

- **引数**

  なし

- **返値**

  PS::F64vec 型。EPI クラスの位置情報を保持したメンバ変数。

- **機能**

  EPI クラスのオブジェクトの位置情報を保持したメンバ変数を返す。

- **備考**

  EPI クラスのメンバ変数 pos の変数名は変更可能。

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;
    }
};
```

- **前提**

  FP クラスのメンバ変数 identity, position と EPI クラスのメンバ変数 id, pos はそれぞれ対応する情報を持つ。

- **引数**

  fp: 入力。const FP &型。FP クラスの情報を持つ。

- **返値**

  なし。

- **機能**

  FP クラスの持つ 1 粒子の情報の一部を EssnetialParticleI クラスに書き込む。

- **備考**

  FP クラスのメンバ変数の変数名、EPI クラスのメンバ変数の変数名は変更可能。メンバ関数 EPI::copyFromFP の引数名は変更可能。EPI クラスの粒子情報は FP クラスの粒子情報のサブセット。対応する情報を持つメンバ変数同士のデータ型が一致している必要はないが、実数型とベクトル型 (または整数型とベクトル型) という違いがある場合に正しく動作する保証はない。

A.2.4 場合によっては必要なメンバ関数

A.2.4.1 概要

本節では、場合によっては必要なメンバ関数について記述する。相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_GATHER または PS::SEARCH_MODE_SYMMETRY を用いる場合に必要となるメンバ関数ついて記述する。

A.2.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_GATHER または PS::SEARCH_MODE_SYMMETRY を用いる場合

A.2.4.2.1 EPI::getRSearch

```
class EPI {
public:
    PS::F64 search_radius;
    PS::F64 getRSearch() const {
        return this->search_radius;
    }
};
```

- **前提**

  EPI クラスのメンバ変数 search_radius はある 1 つの粒子の近傍粒子を探す半径の大きさ。この search_radius のデータ型は PS::F32 型または PS::F64 型。

- **引数**

  なし

- **返値**

  PS::F32 型または PS::F64 型。 EPI クラスの近傍粒子を探す半径の大きさを保持したメンバ変数。

- **機能**

  EPI クラスの近傍粒子を探す半径の大きさを保持したメンバ変数を返す。

- **備考**

  EPI クラスのメンバ変数 search_radius の変数名は変更可能。

A.3 EssentialParticleJ クラス

A.3.1 概要

EssentialParticleJ クラスは相互作用の計算に必要な j 粒子の情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。EssentialParticleJ クラスは FullParticle クラス (節 7.2) のサブセットである。FDPS は、このクラスのデータにアクセスする必要がある。このために、EssentialParticleJ クラスはいくつかのメンバ関数を持つ必要がある。以下、この節の前提、常に必要なメンバ関数と、場合によっては必要なメンバ関数について記述する。

A.3.2 前提

この節の中では、EssentialParticleJ クラスとして EPJ というクラスを一例として使う。また、FullParticle クラスの一例として FP というクラスを使う。EPJ, FP というクラス名は変更可能である。

EPJ と FP の宣言は以下の通りである。

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

A.3.3 必要なメンバ関数

A.3.3.1 概要

常に必要なメンバ関数は EPJ::getPos と EPJ::copyfromFP である。EPJ::getPos は EPJ クラスの位置情報を FDPS に読み込ませるための関数で、EPJ::copyFromFP は FP クラスの情報を EPJ クラスに書きこむ関数である。これらのメンバ関数の記述例と解説を以下に示す。

A.3.3.2 EPJ::getPos

```
class EPJ {
public:
    PS::F64vec pos;
    PS::F64vec getPos() const {
        return this->pos;
    }
};
```

- **前提**

  EPJ のメンバ変数 pos はある 1 つの粒子の位置情報。この pos のデータ型は PS::F64vec 型。

- **引数**

  なし

- **返値**

  PS::F64vec 型。EPJ クラスの位置情報を保持したメンバ変数。

- **機能**

  EPJ クラスの位置情報を保持したメンバ変数を返す。

- **備考**

  EPJ クラスのメンバ変数 pos の変数名は変更可能。

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;
    }
};
```

- **前提**

  FP クラスのメンバ変数 identity, mass, position と EPJ クラスのメンバ変数 id, m, pos はそれぞれ対応する情報を持つ。

- **引数**

  fp: 入力。const FP &型。FP クラスの情報を持つ。

- **返値**

  なし。

- **機能**

  FP クラスの持つ 1 粒子の情報の一部を EPJ クラスに書き込む。

- **備考**

  FP クラスのメンバ変数の変数名、EPJ クラスのメンバ変数の変数名は変更可能。メンバ関数 EPJ::copyFromFP の引数名は変更可能。対応する情報を持つメンバ変数同士のデータ型が一致している必要はないが、実数型とベクトル型 (または整数型とベクトル型) という違いがある場合に正しく動作する保証はない。

A.3.4 場合によっては必要なメンバ関数

A.3.4.1 概要

本節では、場合によっては必要なメンバ関数について記述する。相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合に必要なメンバ関数、列挙型の BOUNDARY_CONDITION 型に PS::BOUNDARY_CONDITION_OPEN 以外を選んだ場合に必要となるメンバ関数について記述する。なお、既存の Moment クラスや SuperParticleJ クラスを用いる際に必要となるメンバ変数はこれら既存のクラスの節を参照のこと。

A.3.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合

A.3.4.2.1 EPJ::getRSearch

```
class EPJ {
public:
    PS::F64 search_radius;
    PS::F64 getRSearch() const {
        return this->search_radius;
    }
};
```

- **前提**

EPJ クラスのメンバ変数 search_radius はある 1 つの粒子の近傍粒子を探す半径の大きさ。この search_radius のデータ型は PS::F32 型または PS::F64 型。

- **引数**

  なし

- **返値**

  PS::F32 型または PS::F64 型。 EPJ クラスの近傍粒子を探す半径の大きさを保持したメンバ変数。

- **機能**

  EPJ クラスの近傍粒子を探す半径の大きさを保持したメンバ変数を返す。

- **備考**

  EPJ クラスのメンバ変数 search_radius の変数名は変更可能。

A.3.4.3 BOUNDARY_CONDITION 型に PS::BOUNDARY_CONDITION_OPEN 以外を用いる場合

A.3.4.3.1 EPJ::setPos

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

- **前提**

  EPJ クラスのメンバ変数 pos は 1 つの粒子の位置情報。この pos のデータ型は PS::F32vec または PS::F64vec。EPJ クラスのメンバ変数 pos の元データとなっているのは FP クラスのメンバ変数 position。このデータ型は PS::F32vec または PS::F64vec。

- **引数**

  pos_new: 入力。const PS::F32vec または const PS::F64vec 型。FDPS 側で修正した粒子の位置情報。

- **返値**

  なし。

- **機能**

  FDPS が修正した粒子の位置情報を EPJ クラスの位置情報に書き込む。

- **備考**

  EPJ クラスのメンバ変数 pos の変数名は変更可能。メンバ関数 EPJ::setPos の引数名 pos_new は変更可能。pos と pos_new のデータ型が異なる場合の動作は保証しない。

A.4 Moment クラス

A.4.1 概要

Moment クラスは近い粒子同士でまとまった複数の粒子のモーメント情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。モーメント情報の例としては、複数粒子の単極子や双極子、さらにこれら粒子の持つ最大の大きさなど様々なものが考えられる。このクラスは、EssentialParticleJ クラスから SuperParticleJ クラスを作るための中間変数のような役割を果す。従って、このクラスが持つメンバ関数は、EssentialParticleJ クラスから情報を読み出してモーメントを計算するメンバ関数、少ない数の粒子のモーメントからそれらの粒子を含むより多くの粒子のモーメントを計算するメンバ関数などがある。

このようなモーメント情報にはある程度決っているものが多いので、それらについては FDPS 側で用意した。これら既存のクラスについてまず記述する。その後にユーザーがモーメントクラスを自作する際に必ず必要なメンバ関数、場合によっては必要になるメンバ関数について記述する。

A.4.2 既存のクラス

A.4.2.1 概要

FDPS はいくつかの Moment クラスを用意している。これらは相互作用ツリークラスで特定の PS::SEARCH_MODE 型を選んだ場合に有効である。以下、各 PS::SEARCH_MODE 型において選ぶことのできる Moment 型を記述する。PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY については Moment クラスを意識してコーディングする必要がないので、これらについては記述しない。

A.4.2.2 PS::SEARCH_MODE_LONG

以下に述べる Moment クラスのうち、MomentMonopole 及び MomentQuadrupole については、近接粒子探索を行うことができる PS::SEARCH_MODE_LONG_SCATTER で使うものも同様に定義されている。それぞれ MomentMonopoleScatter および MomentQuadrupoleScatter である。

A.4.2.2.1 PS::MomentMonopole

単極子までを情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

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

- クラス名 PS::MomentMonopole
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心
- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

  MomentMonopoleScatter の場合、EssentialParticleJ クラスはメンバ関数 getRSearch() をもつこと。

A.4.2.2.2 PS::MomentQuadrupole

単極子と四重極子を情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の重心を取る。以下、このクラスの概要を記述する。

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

- クラス名 PS::MomentQuadrupole

- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量

  pos: 近傍でまとめた粒子の重心

  quad: 近傍でまとめた粒子の四重極子

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

  MomentQuadrupoleScatter の場合、EssentialParticleJ クラスはメンバ関数 getRSearch() をもつこと。

A.4.2.2.3 PS::MomentMonopoleGeometricCenter

単極子までを情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の幾何中心を取る。以下、このクラスの概要を記述する。

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

- クラス名 PS::MomentMonopoleGeometricCenter

- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の幾何中心

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

A.4.2.2.4 PS::MomentDipoleGeometricCenter

双極子までを情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の幾何中心を取る。以下、このクラスの概要を記述する。

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

- クラス名 PS::MomentDipoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の幾何中心

  dipole: 粒子の質量または電荷の双極子

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

A.4.2.2.5 PS::MomentQuadrupoleGeometricCenter

四重極子までを情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の幾何中心を取る。以下、このクラスの概要を記述する。

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

- クラス名 PS::MomentQuadrupoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の幾何中心

dipole: 粒子の質量または電荷の双極子

quadrupole: 粒子の質量または電荷の四重極子

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

A.4.2.3 PS::SEARCH_MODE_LONG_CUTOFF

以下に述べる Moment クラスのうち、MomentMonopoleCutoff については、近接粒子探索を行うことができる PS::SEARCH_MODE_LONG_CUTOFF_SCATTER で使うものも同様に定義されている。MomentMonopoleCutoffScatter である。

A.4.2.3.1 PS::MomentMonopoleCutoff

単極子までを情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

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

- クラス名 PS::MomentMonopoleCutoff
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge, EssentialParticleJ::getPos, EssentialParticleJ::getRSearch を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置、粒子の力の到達距離を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

  MomentMonopoleCutoffScatter の場合、EssentialParticleJ クラスはメンバ関数 getRSearch() をもつこと。

A.4.3 必要なメンバ関数

A.4.3.1 概要

以下では Moment クラスを定義する際に、必要なメンバ関数を記述する。このとき Moment クラスのクラス名を Mom とする。これは変更自由である。

A.4.3.2 コンストラクタ

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    Mom () {
        mass = 0.0;
        pos  = 0.0;
    }
};
```

- **前提**

  Mom クラスのメンバ変数 mass, pos は Mom の質量と位置。

- **引数**

  なし

- **返値**

  なし

- **機能**

  Mom クラスのオブジェクトの初期化をする。

- **備考**

  メンバ変数名の変更可能。メンバ変数を加えることも可能。

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    Mom(const PS::F32 m,
        const PS::F32vec & p) {
        mass = m;
        pos  = p;
    }
};
```

- **前提**

  Mom クラスのメンバ変数 mass, pos は Mom の質量と位置。

- **引数**

  m: 入力。const PS::F32 型。質量

  p: 入力。const PS::F32vec &型。位置。

- **返値**

  なし

- **機能**

  Mom クラスのオブジェクトの初期化をする。

- **備考**

  メンバ変数名の変更可能。メンバ変数を加えることも可能。

A.4.3.3 Mom::init

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

- **前提**

  なし

- **引数**

  なし

- **返値**

  なし

- **機能**

  Mom クラスのオブジェクトの初期化をする。

- **備考**

  なし

A.4.3.4 Mom::getPos

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

- **前提**

  Mom のメンバ変数 pos は近傍でまとめた粒子の代表位置。この pos のデータ型は PS::F32vec または PS::F64vec 型。

- **引数**

  なし

- **返値**

  PS::F32vec または PS::F64vec 型。Mom クラスのメンバ変数 pos。

- **機能**

  Mom クラスのメンバ変数 pos を返す。

- **備考**

  Mom クラスのメンバ変数 pos の変数名は変更自由。

A.4.3.5 Mom::getCharge

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

- **前提**

  Mom のメンバ変数 mass は近傍でまとめた粒子の全質量または全電荷。この mass のデータ型は PS::F32 または PS::F64 型。

- **引数**

  なし

- **返値**

  PS::F32 または PS::F64 型。Mom クラスのメンバ変数 mass。

- **機能**

  Mom クラスのメンバ変数 mass を返す。

- **備考**

  Mom クラスのメンバ変数 mass の変数名は変更自由。

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();
    }
};
```

- **前提**

  Mom のメンバ変数 mass は近傍でまとめた粒子の全質量または全電荷。この mass のデータ型は PS::F32 または PS::F64 型。Mom のメンバ変数 pos は近傍でまとめた粒子の代表位置。この pos のデータ型は PS::F32vec または PS::F64vec 型。テンプレート引数 Tepj には EssentialParticleJ クラスが入り、クラスはメンバ関数 getCharge と getPos を持つ。

- **引数**

  epj: 入力。const Tepj &型。Tepj のオブジェクト。

- **返値**

  なし。

- **機能**

  EssentialParticleJ クラスのオブジェクトからモーメントを計算する。

- **備考**

  Mom クラスのメンバ変数 mass, pos の変数名は変更自由。引数 epj の引数名は変更自由。その他の変数を加えるのも可能。

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;
    }
};
```

- **前提**

  Mom のメンバ変数 mass は近傍でまとめた粒子の全質量または全電荷。この mass のデータ型は PS::F32 または PS::F64 型。Mom のメンバ変数 pos は近傍でまとめた粒子の重心または電荷の重心。この pos のデータ型は PS::F32vec または PS::F64vec 型。

- **引数**

  mom: 入力。const Mom &型。Mom クラスのオブジェクト。

- **返値**

  なし。

- **機能**

  Mom クラスのオブジェクトからさらに Mom クラスの情報を計算する。

- **備考**

  Mom クラスのメンバ変数 mass, pos の変数名は変更自由。引数 epj の引数名は変更自由。その他の変数を加えるのも可能。

A.4.3.8 Mom::set

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    void set() {
        pos = pos / mass;
    }
};
```

- **前提**

  Mom のメンバ変数 mass は近傍でまとめた粒子の全質量または全電荷。この mass のデータ型は PS::F32 または PS::F64 型。Mom のメンバ変数 pos は近傍でまとめた粒子の重心または電荷の重心。この pos のデータ型は PS::F32vec または PS::F64vec 型。

- **引数**

  なし

- **返値**

  なし

- **機能**

  上記のメンバ関数 Mom::accumulateAtLeaf, Mom::accumulate ではモーメントの位置情報の規格化ができていない場合ので、ここで規格化する。

- **備考**

  Mom クラスのメンバ変数 mass, pos の変数名は変更自由。引数 epj の引数名は変更自由。

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;
    }
};
```

- **前提**

  Mom のメンバ変数 mass は近傍でまとめた粒子の全質量または全電荷。この mass のデータ型は PS::F32 または PS::F64 型。Mom のメンバ変数 pos は近傍でまとめた粒子の代表位置。この pos のデータ型は PS::F32vec または PS::F64vec 型。この pos は Mom::accumulateAtLeaf ですでに求めている。Mom のメンバ変数 quad は近傍でまとめた粒子の四重極子。この quad のデータ型は PS::F32mat または PS::F64mat 型。テンプレート引数 Tepj には EssentialParticleJ クラスが入り、クラスはメンバ関数 getCharge と getPos を持つ。

- **引数**

  epj: 入力。const Tepj &型。Tepj のオブジェクト。

- **返値**

  なし。

- **機能**

  EssentialParticleJ クラスのオブジェクトからモーメントを計算する。

- **備考**

  Mom クラスのメンバ変数 mass, pos, quad の変数名は変更自由。引数 epj の引数名は変更自由。その他の変数を加えるのも可能。

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;
    }
};
```

- **前提**

  Mom のメンバ変数 mass は近傍でまとめた粒子の全質量または全電荷。この mass のデータ型は PS::F32 または PS::F64 型。Mom のメンバ変数 pos は近傍でまとめた粒子の代表位置。この pos のデータ型は PS::F32vec または PS::F64vec 型。この pos は Mom::accumulate ですでに求めている。Mom のメンバ変数 quad は近傍でまとめた粒子の四重極子。この quad のデータ型は PS::F32mat または PS::F64mat 型。

- **引数**

  mom: 入力。const Mom &型。Mom クラスのオブジェクト。

- **返値**

  なし。

- **機能**

Mom クラスのオブジェクトからさらに Mom クラスの情報を計算する。

- **備考**

  Mom クラスのメンバ変数 mass, pos, quad の変数名は変更自由。引数 epj の引数名は変更自由。その他の変数を加えるのも可能。

A.5 SuperParticleJ クラス

A.5.1 概要

SuperParticleJ クラスは近い粒子同士でまとまった複数の粒子を代表してまとめた超粒子の情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。このクラスが必要となるのは PS::SEARCH_MODE に PS::SEARCH_MODE_LONG または PS::SEARCH_MODE_LONG_CUTOFF を選んだ場合だけである。このクラスのメンバ関数には、超粒子の位置情報を FDPS 側とやりとりするメンバ関数がある。また、超粒子の情報と Moment クラスの情報は対になるものである。従って、このクラスのメンバ関数には、Moment クラスからこのクラスへ情報を変換 (またはその逆変換) するメンバ関数がある。

SuperParticleJ クラスも Moment クラス同様、ある程度決っているものが多いので、それらについては FDPS 側で用意した。以下、既存のクラス、SuperParticleJ クラスを作るときに必要なメンバ関数、場合によっては必要なメンバ関数について記述する。

A.5.2 既存のクラス

FDPS はいくつかの SuperParticleJ クラスを用意している。以下、各 PS::SEARCH_MODE に対し選ぶことのできるクラスについて記述する。まず、PS::SEARCH_MODE_LONG の場合、次に PS::SEARCH_MODE_LONG_CUTOFF の場合について記述する。その他の PS::SEARCH_MODE では超粒子を必要としない。

A.5.2.1 PS::SEARCH_MODE_LONG

A.5.2.1.1 PS::SPJMonopole

単極子までの情報を持つ Moment クラス PS::MomentMonopole と対になる SuperParticleJ クラス。以下、このクラスの概要を記述する。

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

- クラス名 PS::SPJMonopole
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心
- 使用条件

  Moment クラスである PS::MomentMonopole クラスの使用条件に準ずる。

A.5.2.1.2 PS::SPJQuadrupole

単極子と四重極子を情報を持つ Moment クラス PS::MomentQuadrupole と対になる SuperParticleJ クラス。以下、このクラスの概要を記述する。

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

- クラス名 PS::SPJQuadrupole
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

  quad: 近傍でまとめた粒子の四重極子
- 使用条件

  Moment クラスである PS::MomentQuadrupole クラスの使用条件に準ずる。

A.5.2.1.3 PS::SPJMonopoleGeometricCenter

単極子までを情報として持つ (ただしモーメント計算の際の座標系の中心は粒子の幾何中心)Moment クラス PS::MomentMonopoleGeometricCenter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

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

- クラス名 PS::SPJMonopoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の幾何中心
- 使用条件

  PS::MomentMonopoleGeometricCenter の使用条件に準ずる。

A.5.2.1.4 PS::SPJDipoleGeometricCenter

双極子までを情報として持つ (ただしモーメント計算の際の座標系の中心は粒子の幾何中心)Moment クラス PS::MomentDipoleGeometricCenter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

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

- クラス名 PS::SPJDipoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の幾何中心

  dipole: 粒子の質量または電荷の双極子
- 使用条件

  PS::MomentDipoleGeometricCenter の使用条件に準ずる。

A.5.2.1.5　PS::SPJQuadrupoleGeometricCenter

四重極子までを情報として持つ (ただしモーメント計算の際の座標系の中心は粒子の幾何中心)Moment クラス PS::MomentQuadrupoleGeometricCenter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

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

- クラス名 PS::SPJQuadrupoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の幾何中心

  dipole: 粒子の質量または電荷の双極子

  quadrupole: 粒子の質量または電荷の四重極子
- 使用条件

  PS::MomentQuadrupoleGeometricCenter の使用条件に準ずる。

A.5.2.2　PS::SEARCH_MODE_LONG_CUTOFF

A.5.2.2.1　PS::SPJMonopoleCutoff

単極子までを情報として持つクラス Moment クラス PS::MomentMonopoleCutoff と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

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

- クラス名 PS::SPJMonopoleCutoff
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心
- 使用条件

  PS::MomentMonopoleCutoff の使用条件に準ずる。

A.5.3 必要なメンバ関数

A.5.3.1 概要

以下では SuperParticleJ クラスを作る際に必要なメンバ関数を記述する。このとき SuperParticleJ クラスのクラス名を SPJ とする。これは変更自由である。

A.5.3.2 SPJ::getPos

```
class SPJ {
public:
    PS::F64vec pos;
    PS::F64vec getPos() const {
        return this->pos;
    }
};
```

- **前提**

  SPJ のメンバ変数 pos はある 1 つの超粒子の位置情報。この pos のデータ型は PS::F32vec または PS::F64vec 型。
- **引数**

  なし
- **返値**

  PS::F32vec 型または PS::F64vec 型。SPJ クラスの位置情報を保持したメンバ変数。
- **機能**

  SPJ クラスの位置情報を保持したメンバ変数を返す。
- **備考**

  SPJ クラスのメンバ変数 pos の変数名は変更可能。

A.5.3.3 SPJ::setPos

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

- **前提**

  SPJ クラスのメンバ変数 pos は 1 つの粒子の位置情報。この pos のデータ型は PS::F32vec または PS::F64vec。

- **引数**

  pos_new: 入力。const PS::F32vec または const PS::F64vec 型。FDPS 側で修正した粒子の位置情報。

- **返値**

  なし。

- **機能**

  FDPS が修正した粒子の位置情報を SPJ クラスの位置情報に書き込む。

- **備考**

  SPJ クラスのメンバ変数 pos の変数名は変更可能。メンバ関数 SPJ::setPos の引数名 pos_new は変更可能。pos と pos_new のデータ型が異なる場合の動作は保証しない。

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;
    }
};
```

- **前提**

  なし

- **引数**

  mom: 入力。const Mom &型。Mom にはユーザー定義または FDPS 側で用意した Moment クラスが入る。

- **返値**

  なし。

- **機能**

  Mom クラスの情報を SPJ クラスにコピーする。

- **備考**

  Mom クラスのクラス名は変更可能。Mom クラスと SPJ クラスのメンバ変数名は変更可能。メンバ関数 SPJ::copyFromMoment の引数名は変更可能。

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);
    }
};
```

- **前提**

  なし

- **引数**

  なし

- **返値**

  Mom 型。Mom クラスのコンストラクタ。

- **機能**

  Mom クラスのコンストラクタを返す。

- **備考**

  Mom クラスのクラス名は変更可能。Mom クラスと SPJ クラスのメンバ変数名は変更可能。メンバ関数SPJ::copyFromMoment の引数名は変更可能。メンバ関数SPJ::convertToMoment で使用される Mom クラスのコンストラクタが定義されている必要がある。

A.5.3.6 SPJ::clear

```
class SPJ {
public:
    PS::F32    mass;
    PS::F32vec pos;
    void clear() {
        mass = 0.0;
        pos  = 0.0;
    }
};
```

- **前提**

  なし

- **引数**

  なし

- **返値**

  なし

- **機能**

  SPJ クラスのオブジェクトの情報をクリアする。

- **備考**

  メンバ変数名は変更可能。

A.6 Force クラス

A.6.1 概要

Force クラスは相互作用の結果を保持するクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。以下、この節の前提、常に必要なメンバ関数について記述する。

A.6.2 前提

この節で用いる例として Force クラスのクラス名を Result とする。このクラス名は変更自由である。

A.6.3 必要なメンバ関数

常に必要なメンバ関数は Result::clear である。この関数は相互作用の計算結果を初期化する。以下、Result::clear について記述する。

A.6.3.1 Result::clear

```
class Result {
public:
    PS::F32vec acc;
    PS::F32    pot;
    void clear() {
        acc = 0.0;
        pot = 0.0;
    }
};
```

- **前提**

  Result クラスのメンバ変数は acc と pot。

- **引数**

  なし

- **返値**

  なし。

- **機能**

  Result クラスのメンバ変数を初期化する。

- **備考**

  Result クラスのメンバ変数 acc, pot の変数名は変更可能。他のメンバ変数を加えることも可能。

A.7 ヘッダクラス

A.7.1 概要

ヘッダクラスは入出力ファイルのヘッダの形式を決めるクラスである。ヘッダクラスは FDPS が提供する粒子群クラスのファイル入出力 API を使用し、かつ入出力ファイルにヘッダを含ませたい場合に必要となるクラスである。粒子群クラスのファイル入出力 API とは、ParticleSystem::readParticleAscii, ParticleSystem::writeParticleAscii, ParticleSystem::readParticleBinary,

ParticleSystem::writeParticleBinary である。以下、この節における前提と、これらの API を使用する際に必要となるメンバ関数とその記述の規定を述べる。この節において、常に必要なメンバ関数というものは存在しない。

A.7.2 前提

この節では、ヘッダクラスのクラス名を Hdr とする。このクラス名は変更可能である。

A.7.3 場合によっては必要なメンバ関数

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;
    }
};
```

- **前提**

  このヘッダは粒子数、時刻の情報を持つ。これらのメンバ変数はそれぞれ nparticle と time である。

- **引数**

  fp: 入力。FILE *型。粒子データの入力ファイルを指すファイルポインタ。

- **返値**

  PS::S32 型。粒子数の情報を返す。ヘッダに粒子数の情報がない場合は-1 を返す。

- **機能**

  粒子データの入力ファイルからヘッダ情報を読みこむ。

- **備考**

  メンバ変数名は入力ファイルに合わせて変更可能。返値に粒子数の情報を指定しない場合、または-1 を指定しない場合の動作は保証しない。

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);
    }
};
```

- **前提**

  このヘッダは粒子数、時刻の情報を持つ。これらのメンバ変数はそれぞれ nparticle と time である。

- **引数**

  fp: 入力。FILE *型。粒子データの出力ファイルを指すファイルポインタ。

- **返値**

  なし。

- **機能**

  粒子データの出力ファイルへヘッダ情報を書き込む。

- **備考**

  メンバ変数名は出力ファイルに合わせて変更可能。

A.7.3.3 Hdr::readBinary

```
class Hdr {
public:
    PS::S32 nparticle;
    PS::F64 time;
    PS::S32 readBinary(FILE *fp) {
        fread(&this->nparticle, sizeof(PS::S32), 1, fp);
        fread(&this->time, sizeof(PS::F64), 1, fp);
        return this->nparticle;
    }
};
```

- **前提**

  このヘッダは粒子数、時刻の情報を持つ。これらのメンバ変数はそれぞれ nparticle と time である。

- **引数**

  fp: 入力。FILE *型。粒子データの入力ファイルを指すファイルポインタ。

- **返値**

  PS::S32 型。粒子数の情報を返す。ヘッダに粒子数の情報がない場合は-1 を返す。

- **機能**

  粒子データの入力ファイルからヘッダ情報を読みこむ。

- **備考**

  メンバ変数名は入力ファイルに合わせて変更可能。返値に粒子数の情報を指定しない場合、または-1 を指定しない場合の動作は保証しない。

A.7.3.4 Hdr::writeBinary

```
class Hdr {
public:
    PS::S32 nparticle;
    PS::F64 time;
    void writeBinary(FILE *fp) {
        fwrite(&this->nparticle, sizeof(PS::S32), 1, fp);
        fwrite(&this->time, sizeof(PS::F64), 1, fp);
    }
};
```

- **前提**

  このヘッダは粒子数、時刻の情報を持つ。これらのメンバ変数はそれぞれ nparticle と time である。

- **引数**

  fp: 入力。FILE *型。粒子データの出力ファイルを指すファイルポインタ。

- **返値**

  なし。

- **機能**

  粒子データの出力ファイルへヘッダ情報を書き込む。

- **備考**

  メンバ変数名は出力ファイルに合わせて変更可能。

A.8 関数オブジェクト calcForceEpEp

A.8.1 概要

関数オブジェクト calcForceEpEp は粒子同士の相互作用を記述するものであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。以下、これの書き方の規定を記述する。

A.8.2 前提

ここで示すのは重力 N 体シミュレーションの粒子間相互作用の記述の仕方である。関数オブジェクト calcForceEpEp の名前は gravityEpEp とする。これは変更自由である。また、EssentialParitlceI クラスのクラス名を EPI, EssentialParitlceJ クラスのクラス名を EPJ, Force クラスのクラス名を Result とする。

A.8.3 gravityEpEp::operator ()

ソースコード 49: calcForceEpEp

```
1  class Result {
2  public:
3      PS::F32vec acc;
4  };
5  class EPI {
6  public:
7      PS::S32    id;
8      PS::F32vec pos;
9  };
10 class EPJ {
11 public:
12     PS::S32    id;
13     PS::F32    mass;
14     PS::F32vec pos;
15 };
16 struct gravityEpEp {
17     static PS::F32 eps2;
```

```
18      void operator () (const EPI *epi,
19                        const PS::S32 ni,
20                        const EPJ *epj,
21                        const PS::S32 nj,
22                        Result *result) {
23
24          for(PS::S32 i = 0; i < ni; i++) {
25              PS::S32    ii = epi[i].id;
26              PS::F32vec xi = epi[i].pos;
27              PS::F32vec ai = 0.0;
28              for(PS::S32 j = 0; j < nj; j++) {
29                  PS::S32    jj = epj[j].id;
30                  PS::F32    mj = epj[j].mass;
31                  PS::F32vec xj = epj[j].pos;
32
33                  PS::F32vec dx   = xi - xj;
34                  PS::F32    r2   = dx * dx + eps2;
35                  PS::F32    rinv = (ii != jj) ? 1. / sqrt(r2)
36                                               : 0.0;
37
38                  ai += mj * rinv * rinv * rinv * dx;
39              }
40              result.acc = ai;
41          }
42      }
43  };
44  PS::F32 gravityEpEp::eps2 = 9.765625e-4;
```

- **前提**

  クラス Result, EPI, EPJ に必要なメンバ関数は省略した。クラス Result のメンバ変数 acc は i 粒子が j 粒子から受ける重力加速度である。クラス EPI と EPJ のメンバ変数 id と pos はそれぞれの粒子 ID と粒子位置である。クラス EPJ のメンバ変数 mass は j 粒子の質量である。関数オブジェクト gravityEpEp のメンバ変数 eps2 は重力ソフトニングの 2 乗である。ここの外側でスレッド並列になっているため、ここで OpenMP を記述する必要はない。

- **引数**

  epi: 入力。const EPI *型または EPI *型。i 粒子情報を持つ配列。

  ni: 入力。const PS::S32 型または PS::S32 型。i 粒子数。

  epj: 入力。const EPJ *型または EPJ *型。j 粒子情報を持つ配列。

nj: 入力。const PS::S32 型または PS::S32 型。j 粒子数。

result: 出力。Result *型。i 粒子の相互作用結果を返す配列。

- **返値**

  なし。

- **機能**

  j 粒子から i 粒子への作用を計算する。

- **備考**

  引数名すべて変更可能。関数オブジェクトの内容などはすべて変更可能。

A.9 関数オブジェクト calcForceSpEp

A.9.1 概要

関数オブジェクト calcForceSpEp は超粒子から粒子への作用を記述するものであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。以下、これの書き方の規定を記述する。

A.9.2 前提

ここで示すのは重力 N 体シミュレーションにおける超粒子から粒子への作用の記述の仕方である。超粒子は単極子までの情報で作られているものとする。関数オブジェクト calcForceSpEp の名前は gravitySpEp とする。これは変更自由である。また、EssentialParitlceI クラスのクラス名を EPI, SuperParitlceJ クラスのクラス名を SPJ, Force クラスのクラス名を Result とする。

A.9.3 gravitySpEp::operator ()

ソースコード 50: calcForceSpEp

```
1  class Result {
2  public:
3      PS::F32vec accfromspj;
4  };
5  class EPI {
6  public:
7      PS::S32    id;
8      PS::F32vec pos;
9  };
10 class SPJ {
```

```
11 public:
12     PS::F32    mass;
13     PS::F32vec pos;
14 };
15 struct gravitySpEp {
16     static PS::F32 eps2;
17     void operator () (const EPI *epi,
18                       const PS::S32 ni,
19                       const SPJ *spj,
20                       const PS::S32 nj,
21                       Result *result) {
22
23         for(PS::S32 i = 0; i < ni; i++) {
24             PS::F32vec xi = epi[i].pos;
25             PS::F32vec ai = 0.0;
26             for(PS::S32 j = 0; j < nj; j++) {
27                 PS::F32    mj = spj[j].mass;
28                 PS::F32vec xj = spj[j].pos;
29
30                 PS::F32vec dx   = xi - xj;
31                 PS::F32    r2   = dx * dx + eps2;
32                 PS::F32    rinv = 1. / sqrt(r2);
33
34                 ai += mj * rinv * rinv * rinv * dx;
35             }
36             result.accfromspj = ai;
37         }
38     }
39 };
40 PS::F32 gravitySpEp::eps2 = 9.765625e-4;
```

- **前提**

  クラス Result, EPI, SPJ に必要なメンバ関数は省略した。クラス Result のメンバ変数 accfromspj は i 粒子が超粒子から受ける重力加速度である。クラス EPI と SPJ のメンバ変数 pos はそれぞれの粒子位置である。クラス SPJ のメンバ変数 mass は超粒子の質量である。ファンクタ gravitySpEp のメンバ変数 eps2 は重力ソフトニングの 2 乗である。ここの外側でスレッド並列になっているため、ここで OpenMP を記述する必要はない。

- **引数**

epi: 入力。const EPI *型または EPI *型。i 粒子情報を持つ配列。

ni: 入力。const PS::S32 型または PS::S32 型。i 粒子数。

spj: 入力。const SPJ *型または SPJ *型。超粒子情報を持つ配列。

nj: 入力。const PS::S32 型または PS::S32 型。超粒子数。

result: 出力。Result *型。i 粒子の相互作用結果を返す配列。

- **返値**

  なし。

- **機能**

  超粒子から i 粒子への作用を計算する。

- **備考**

  引数名すべて変更可能。関数オブジェクトの内容などはすべて変更可能。

A.10 関数オブジェクト calcForceDispatch

A.10.1 概要

関数オブジェクト calcForceDispatch は相互作用計算にアクセラレータを使う場合に用いられ、粒子をアクセラレータに送り、相互作用カーネルを発行する。以下、これの書き方の規定を記述する。

A.10.2 前提

ここで示すのは重力 N 体シミュレーションにおける相互作用を cuda を用いて記述する方法である。関数オブジェクト calcForceDispatch の名前は CalcForceDispatch とする。これは変更自由である。また、EssentialParitlceI クラスのクラス名を EPI, SuperParitlceJ クラスのクラス名を SPJ, Force クラスのクラス名を Result とする。

A.10.3 例

ソースコード 51: calcForceDispatch

```
1
2 class EpiGPU{
3 public:
4     float2 pos[3];
5     int id_walk;
6 };
```

```
 7
 8 class EpjGPU{
 9 public:
10     float mass;
11     float2 pos[3];
12 };
13
14 class ForceGPU{
15 public:
16     float2 acc[3];
17     float2 pot;
18 };
19
20 __global__ void ForceKernel(const EpiGPU * epi,
21                             const EpjGPU * epj,
22                             const int    * nj_disp,
23                             ForceGPU     * force,
24                             const float eps2){
25     int id_i = blockDim.x * blockIdx.x + threadIdx.x;
26     const EpiGPU & ip = epi[id_i];
27     float2 poti;
28     float2 acci[3];
29     poti = acci[0] = acci[1] = acci[2] = make_float2(0.0, 0.0);
30     const int j_head = nj_disp[ip.id_walk];
31     const int j_tail = nj_disp[ip.id_walk+1];
32     const int nj = j_tail - j_head;
33     for(int j=j_head; j<j_tail; j++){
34         EpjGPU jp = epj[j];
35         const float dx = (jp.pos[0].x - ip.pos[0].x) + (jp.pos
               [0].y - ip.pos[0].y);
36         const float dy = (jp.pos[1].x - ip.pos[1].x) + (jp.pos
               [1].y - ip.pos[1].y);
37         const float dz = (jp.pos[2].x - ip.pos[2].x) + (jp.pos
               [2].y - ip.pos[2].y);
38         const float r2 = ((eps2 + dx*dx) + dy*dy) + dz*dz;
39         const float r_inv = rsqrtf(r2);
40         const float pij = jp.mass * r_inv * (r2 > eps2);
41         const float r2_inv = r_inv * r_inv;
42         const float pij_r3_inv = pij * r2_inv;
43         const float ax = pij_r3_inv * dx;
```

```
44          const float ay = pij_r3_inv * dy;
45          const float az = pij_r3_inv * dz;
46          poti = float2_accum(poti, pij);
47          acci[0] = float2_accum(acci[0], ax);
48          acci[1] = float2_accum(acci[1], ay);
49          acci[2] = float2_accum(acci[2], az);
50      }
51      poti = float2_regularize(poti);
52      acci[0] = float2_regularize(acci[0]);
53      acci[1] = float2_regularize(acci[1]);
54      acci[2] = float2_regularize(acci[2]);
55      force[id_i].pot = poti;
56      force[id_i].acc[0] = acci[0];
57      force[id_i].acc[1] = acci[1];
58      force[id_i].acc[2] = acci[2];
59  }
60
61  static ForceGPU * force_d;
62  static ForceGPU * force_h;
63  static EpiGPU * epi_d;
64  static EpiGPU * epi_h;
65  static EpjGPU * epj_d;
66  static EpjGPU * epj_h;
67  static int * ni_disp_h;
68  static int * nj_disp_d;
69  static int * nj_disp_h;
70
71  int DispatchKernelWithSP(const PS::S32 tag,
72                           const int    n_walk,
73                           const EPIGrav ** epi,
74                           const int  *  n_epi,
75                           const EPJGrav ** epj,
76                           const int  *  n_epj,
77                           const PS::SPJMonopole ** spj,
78                           const int  *  n_spj){
79      static bool first = true;
80      assert(n_walk <= N_WALK_LIMIT);
81      if(first){
82          CUDA_SAFE_CALL( cudaMalloc(     (void**)&nj_disp_d,  (
                N_WALK_LIMIT+1)*sizeof(int) ) );
```

```
83          CUDA_SAFE_CALL( cudaMallocHost( (void**)&ni_disp_h,  (
                N_WALK_LIMIT+1)*sizeof(int) ) );
84          CUDA_SAFE_CALL( cudaMallocHost( (void**)&nj_disp_h,  (
                N_WALK_LIMIT+1)*sizeof(int) ) );
85          CUDA_SAFE_CALL( cudaMalloc( (void**)&epi_d,
                NI_LIMIT*sizeof(EpiGPU) ) );
86          CUDA_SAFE_CALL( cudaMalloc( (void**)&epj_d,
                NJ_LIMIT*sizeof(EpjGPU) ) );
87          CUDA_SAFE_CALL( cudaMalloc( (void**)&force_d,
                NI_LIMIT*sizeof(ForceGPU) ) );
88          CUDA_SAFE_CALL( cudaMallocHost( (void**)&epi_h,
                NI_LIMIT*sizeof(EpiGPU) ) );
89          CUDA_SAFE_CALL( cudaMallocHost( (void**)&epj_h,
                NJ_LIMIT*sizeof(EpjGPU) ) );
90          CUDA_SAFE_CALL( cudaMallocHost( (void**)&force_h,
                NI_LIMIT*sizeof(ForceGPU) ) );
91          first = false;
92      }
93      const float eps2 = EPIGrav::eps * EPIGrav::eps;
94      ni_disp_h[0] = nj_disp_h[0] = 0;
95      for(int i=0; i<n_walk; i++){
96          ni_disp_h[i+1] = ni_disp_h[i] + n_epi[i];
97          nj_disp_h[i+1] = nj_disp_h[i] + n_epj[i] + n_spj[i];
98      }
99      int ni_tot = ni_disp_h[n_walk];
100     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);
101     assert(ni_tot_reg <= NI_LIMIT);
102     assert(nj_disp_h[n_walk] <= NJ_LIMIT);
103     ni_tot = 0;
104     int nj_tot = 0;
105     for(int iw=0; iw<n_walk; iw++){
106         for(int ip=0; ip<n_epi[iw]; ip++){
107             epi_h[ni_tot].pos[0]  = float2_split(epi[iw][ip].
                    pos.x);
108             epi_h[ni_tot].pos[1]  = float2_split(epi[iw][ip].
                    pos.y);
109             epi_h[ni_tot].pos[2]  = float2_split(epi[iw][ip].
                    pos.z);
```

```
110             epi_h[ni_tot].id_walk = iw;
111             force_h[ni_tot].acc[0] = force_h[ni_tot].acc[1]
112                 = force_h[ni_tot].acc[2] = force_h[ni_tot].pot
                        = make_float2(0.0, 0.0);
113             ni_tot++;
114         }
115         for(int jp=0; jp<n_epj[iw]; jp++){
116             epj_h[nj_tot].mass    = epj[iw][jp].mass;
117             epj_h[nj_tot].pos[0]  = float2_split(epj[iw][jp].
                      pos.x);
118             epj_h[nj_tot].pos[1]  = float2_split(epj[iw][jp].
                      pos.y);
119             epj_h[nj_tot].pos[2]  = float2_split(epj[iw][jp].
                      pos.z);
120             nj_tot++;
121         }
122         for(int jp=0; jp<n_spj[iw]; jp++){
123             epj_h[nj_tot].mass    = spj[iw][jp].getCharge();
124             epj_h[nj_tot].pos[0]  = float2_split(spj[iw][jp].
                      getPos().x);
125             epj_h[nj_tot].pos[1]  = float2_split(spj[iw][jp].
                      getPos().y);
126             epj_h[nj_tot].pos[2]  = float2_split(spj[iw][jp].
                      getPos().z);
127             nj_tot++;
128         }
129     }
130     for(int ip=ni_tot; ip<ni_tot_reg; ip++){
131         epi_h[ni_tot].pos[0]  = epi_h[ni_tot].pos[1]  = epi_h[
              ni_tot].pos[2]  = make_float2(0.0, 0.0);
132         epi_h[ni_tot].id_walk = 0;
133         force_h[ni_tot].acc[0] = force_h[ni_tot].acc[1]
134             = force_h[ni_tot].acc[2] = force_h[ni_tot].pot =
                      make_float2(0.0, 0.0);
135     }
136     CUDA_SAFE_CALL( cudaMemcpy(epi_d, epi_h, ni_tot_reg*sizeof(
          EpiGPU), cudaMemcpyHostToDevice) );
137     CUDA_SAFE_CALL( cudaMemcpy(epj_d, epj_h, nj_tot*sizeof(
          EpjGPU), cudaMemcpyHostToDevice) );
```

```
138     CUDA_SAFE_CALL( cudaMemcpy(nj_disp_d, nj_disp_h, (n_walk
            +1)*sizeof(int), cudaMemcpyHostToDevice) );
139     const int n_grid = ni_tot_reg/N_THREAD_GPU + ((ni_tot_reg%
            N_THREAD_GPU == 0) ? 0 : 1);
140     dim3 size_grid(n_grid, 1, 1);
141     dim3 size_thread(N_THREAD_GPU, 1, 1);
142     ForceKernel<<<size_grid, size_thread>>> (epi_d, epj_d,
            nj_disp_d, force_d, float(eps2));
143
144     return 0;
145 }
```

- **引数**

  tag: 入力。const PS::S32 型。対応する CalcForceRetrieve() の tag 番号と一致させる必要がある。

  nwalk: 入力。const PS::S32 型。マルチウォークにより作成する相互作用リストの数。

  epi: 入力。const EPI** 型または EPI** 型。i 粒子情報を持つ配列。

  ni: 入力。const PS::S32* 型または PS::S32* 型。i 粒子数。

  spj: 入力。const EPJ** 型または EPJ** 型。超粒子情報を持つ配列。

  nj: 入力。const PS::S32* 型または PS::S32* 型。超粒子数。

- **返値**

  正常終了ならば 0、それ以外の場合は 0 以外の値を返す。

- **機能**

  epi,epj をアクセラレータに送り、アクセラレータ上で相互作用計算を行わせる。

- **備考**

  引数名すべて変更可能。関数オブジェクトの内容などはすべて変更可能。

A.11 関数オブジェクト calcForceRetrieve

A.11.1 概要

関数オブジェクト calcForceRetrieve はアクセラレータで計算された結果を回収する関数である。以下、これの書き方の規定を記述する。

A.11.2 前提

ここで示すのは重力 N 体シミュレーションにおける相互作用を cuda を用いて記述する方法である。関数オブジェクト calcForceRetrieve の名前は RetieveKernel とする。これは変更自由である。また、Force クラスのクラス名を ForceGrav とする。

ソースコード 52: calcForceRetrieve

```
1  int RetrieveKernel(const PS::S32 tag,
2                     const PS::S32    n_walk,
3                     const PS::S32 *  ni,
4                     ForceGrav     ** force){
5      int ni_tot = 0;
6      for(int i=0; i<n_walk; i++){
7          ni_tot += ni[i];
8      }
9      CUDA_SAFE_CALL( cudaMemcpy(force_h, force_d,       ni_tot*
           sizeof(ForceGPU), cudaMemcpyDeviceToHost) );
10     int n_cnt = 0;
11     for(int iw=0; iw<n_walk; iw++){
12         for(int ip=0; ip<ni[iw]; ip++){
13             force[iw][ip].acc.x = (double)force_h[n_cnt].acc
                   [0].x + (double)force_h[n_cnt].acc[0].y;
14             force[iw][ip].acc.y = (double)force_h[n_cnt].acc
                   [1].x + (double)force_h[n_cnt].acc[1].y;
15             force[iw][ip].acc.z = (double)force_h[n_cnt].acc
                   [2].x + (double)force_h[n_cnt].acc[2].y;
16             force[iw][ip].pot   = (double)force_h[n_cnt].pot.x
                      + (double)force_h[n_cnt].pot.y;
17             force[iw][ip].pot *= -1.0;
18             n_cnt++;
19         }
20     }
21     return 0;
22 }
```

- **引数**

  tag: 入力。const PS::S32 型。対応する CalcForceDispatch() の tag 番号と一致させる必要がある。

  nwalk: 入力。const PS::S32 型。マルチウォークにより作成する相互作用リストの数。

  ni: 入力。const PS::S32* 型または PS::S32* 型。i 粒子数。

force: 入力。Result** 型。

- **返値**

  正常終了ならば 0、それ以外の場合は 0 以外の値を返す。

- **機能**

  calcForceDispach で計算された結果を force に格納する。

- **備考**

  引数名すべて変更可能。関数オブジェクトの内容などはすべて変更可能。