FDPS: Framework for Developing Particle Simulator

谷川衝、岩澤全規

目 次

1 change log

- 2014 年 5 月 xx 日 名前変更
  - GhostParticle から EssentialParticle に。
  - BHParticle から TreeParticle に。
  - BHParticle から TreeCell に。
  - 型名は大文字始まりキャメルスタイルに。
  - メンバ関数名は小文字始まりキャメルスタイルに。
  - メンバ変数名は小文字でアンダースコアでつなぎアンダスコアで終わらす。
- 2014 年 5 月 xx 日 EP 型を EPI と EPJ に分離。
- 2014 年 5 月 xx 日 ドキュメント、API と実装を分離。
- 2014 年 5 月 xx 日 領域の座標の表現を `vector` クラス二つから `rectangle` クラスに変更。
- 2014 年 5 月 xx 日 Nbody のサンプルコード、STDSPH、Nbody+SPH のサンプルコードを新しく。継続中。
- 2014 年 6 月 10 日 `Vector` 型、`Rectangle` 型の記述を加える。
- 2014 年 6 月 11 日 `Rectangle` 型から `Orthotope` 型へ。
- 2014 年 6 月 11 日 マクロ `PS_DIM` から `PARTICLE_SIMULATOR_DIMENSION` へ。
- 2014 年 6 月 11 日 `Vector` 型の説明更新。
- 2014 年 6 月 11 日 **??**更新。
- 2014 年 6 月 25 日 `TreeForForce` クラスのテンプレートパラメータに `search_mode` を追加。

- 2014 年 6 月 25 日 `TreeParticle` クラス、`TreeCell` クラスを変更。
- 2014 年 8 月 15 日 境界条件 API 追加。
- 2014 年 8 月 28 日 境界条件の仕様拡張に伴う仕様変更 (領域分割クラス、粒子群クラス、相互作用クラスの節のみ変更)
- 2014 年 8 月 28 日 周期境界条件などを用いる場合に現れる粒子のコピーを「イメージ粒子」と定義 (定義を文中に)
- 2014 年 8 月 28 日 粒子の実体が存在しうる領域を「ルートドメイン」と定義、イメージ粒子が存在する領域を「イメージドメイン」と定義、粒子の実体に力を及ぼすイメージ粒子が存在する領域を「ホライゾン」と定義 (定義を文中に)
- 2014 年 8 月 28 日 1 プロセスが担当する領域を「領域」から「ドメイン」に変更 (定義を文中に)
- 2014 年 8 月 29 日 境界条件指定方法を??に追加。
- 2014 年 8 月 29 日 相互作用クラスの相互作用関数 (`TreeForForce::calcForce()` 等) をテンプレート関数に変更し、関数オブジェクトも使える様に仕様変更。
- 2014 年 10 月 6 日 領域クラスの API 変更。
- 2014 年 12 月 19 日 treeforforce のテンプレート引数を変更。
- 2015 年 2 月 12 日 File I/O 周りの API 変更。

2 TO DO

- イメージ粒子に対する粒子の実体の名前を決定する。
- ネイバーリスト検討。

3 FDPS とは

FDPS は任意の粒子シミュレーションコードの開発を支援するフレームワークである。FDPS ユーザ (以下、ユーザ) が 1 粒子の持つ情報と粒子間の相互作用の形などを FDPS コードジェネレータ (以下、コードジェネレータ) に入力すると、粒子シミュレーション用のライブラリ (PS ライブラリ、以後 PSL) が出力される。ユーザは PSL を基に粒子シミュレーションコードを書くことができる。PSL は、任意の粒子と相互作用する粒子群の探査と、それらの相互作用の計算を、あらゆる階層の並列化 (マルチプロセス、マルチスレッド、SIMD 演算) を用いて高速に行う関数群を提供する。PSL 使用の最大の利点は、ユーザが粒子シミュレーションコードを高速化を意識せずに作成できることである。

PSL は C++で記述されている。ユーザは C++を用いて粒子シミュレーションコードを記述することが推奨される。

PSL は 2 粒子間相互作用の計算のみサポートする。3 粒子以上の相互作用はサポートしない。また、独立時間刻み法もサポートしない。ただし、近傍粒子を返す API を用意するので、ユーザが P^3T 法を用いて独立時間刻み法を実装することは可能である。

本文書では、PSL について記述する。粒子間相互作用の形の定義の仕方などは必要がない限り記述しない。コードジェネレータについては全く触れない。本文書の構成は以下の通りである。**??**節では、PSL で実行されることについて簡潔に記述する。**??**節では、PSL の API を示す。**??**節では、PSL を使用した粒子シミュレーションコードの例を示す。最後に**??**節では、FDPS 作成のロードマップを記述する。

4 PSL

本節では、PSL が実行すること、またそのアルゴリズムについて記述する。一般に、分散メモリ環境での粒子シミュレーションの手順は以下の 3 つに分けられる。

0. 各プロセスの役割分担を決定
1. i 粒子に対する j 粒子リストを作成
2. i 粒子に対する j 粒子の作用を計算し、i 粒子の物理量とその時間変化量を導出
3. i 粒子の物理量を時間積分

ここで、作用される粒子を i 粒子、作用する粒子を j 粒子と呼び、ある i 粒子に対する全 j 粒子を j 粒子リストと呼んだ。PSL は手順 0 から 2 を行う関数群を提供する。

手順 0 は以下 2 つの手順に分割される。

0.1. 各プロセスの担当粒子を決定

0.2. 各プロセスにそれぞれの担当粒子を分配

PSL では、それぞれの手順は GreeM とほぼ同様のアルゴリズムを用いる。

手順 1 は、さらに以下のような手順に分割される。

1.1. 各プロセスが自分の担当粒子の探査を高速にするデータ構造を構築

1.2. 各プロセスが自分の担当粒子のうち別プロセスの担当粒子の j 粒子リストに入る可能性のある粒子を探査し、そうであればその粒子データを別プロセスへ送信

1.3. 各プロセスが自分の担当粒子と別プロセスから送信されてきた粒子の粒子探査を高速にするデータ構造を構築

1.4. 各プロセスが担当粒子の j 粒子リストを作成

PSL では、手順 1.1 と手順 1.3 でツリー構造が構築される。前者のツリーをローカルツリー、後者のツリーをグローバルツリーと呼ぶ。

手順 1.2 と手順 1.4 で行われる粒子探査は、扱う相互作用が短距離力か長距離力かで大きく異なる。ここで短距離力とは力の到達距離が有限である場合であり、長距離力とは力の到達距離が無限である場合である。短距離力において探査するのは粒子そのものである。一方長距離力においては、粒子そのものだけでなく、遠くの複数の粒子をまとめた超粒子も探査する。ただし、粒子に超粒子が作用しつつ力の到達距離が有限となる場合がある。例としては TreePM 法を用いた N 体シミュレーションである。PSL では、これは長距離力に分類される。

長距離力の場合の手順 1.2 のアルゴリズムは、GreeM に準じる。短距離力の場合の手順 1.2 のアルゴリズムは、相互作用の性質に応じて場合分けされる。相互作用の性質は以下の 4 種類が存在する。

- 力の到達距離が全粒子で等しい相互作用 (固定長モード)
- 力の到達距離が i 粒子のサイズで決まる相互作用 (収集モード)
- 力の到達距離が j 粒子のサイズで決まる相互作用 (散乱モード)
- 力の到達距離が ij 粒子の両方のサイズで決まる相互作用 (対称モード)

固定長モードのアルゴリズムはまだ決めていない。他のモードは ASURA に準じる。

PSL では手順 2 において SIMD 演算が用いられる。SIMD 演算を用いて計算性能を得るために、複数の i 粒子が共通の j 粒子リストを持つようにする。本文書では、PSL における SIMD 演算の実装については言及しない。

5 API

PSL は前節で示した手順 0、1、2 を実行するクラスを提供する。クラスは 3 つある。1 つ目は手順 0.1 を実行する領域分割クラス `DomainInfo`、2 つ目は手順 0.2 を実行する粒子群クラス `ParticleSystem`、3 つ目は手順 1、2 を実行する相互作用クラス `TreeforForce` である。まずこれらのクラスが属する名前空間について述べる。次に、初期化等の関数について触れる。次にシミュレーションを行う上でコアとなる上記 3 つのクラスそれぞれについて記述する。これらのクラス内部では、MPI、OpenMP、SIMD など様々な並列化手法が用いられている。最後にこれら並列化手法を最適化するためのパラメータ等を管理するコミュニケーションクラス `Comm` についてを述べる。

PSL では計算や通信の効率化の為、粒子情報やツリーの情報をいくつかの小さいサブクラスに持たしている。これらの定義は付録??節と??節に記されている。

5.1 インクルードファイル

ユーザーは PSL を使う為に `particle_simulator.h` というヘッダーファイルをインクルードしなければならない。このファイルをインクルードする事で、以下に記述される API を使

うことが出来る。記述されている API はすべて ParticleSimulator という名前空間に属する。これでは長いので、省略名 PS も使うことが出来る。ユーザーが ParticleSimulator と PS という名前の名前空間を定義する事は出来ない。

ユーザはコンパイル時に PARTICLE_SIMULATOR_TWO_DIMENSION を定義する事で、2 次元用の PSL を使うことが出来る。何も定義しない場合は 3 次元用の PSL を使うことが出来る。

particle_simulator.h は以下の様に記述される。

ソースコード 1: particle_simulator.h

```
1  namespace ParticleSimulator{
2  #ifdef PARTICLE_SIMULATOR_TOW_DIMENSION
3      static const int DIMENSION = 2;
4  #else
5      static const int DIMENSION = 3;
6  #endif
7  }
8
9  #include<ps_defs.hpp>
10 #include<domain_info.hpp>
11 #include<particle_system.hpp>
12 #include<tree_for_force.hpp>
13 namespace PS = ParticleSimulator;
```

domain_info.hpp, particle_system.hpp, tree_for_force.hpp はそれぞれ、領域クラス DomainInfo、粒子群クラス ParticleSystem、相互作用クラス TreeforForce について記述されているヘッダーファイルである。

すでにユーザー使っているライブラリなどで、これらの名前が使われている場合は、ユーザーは以下の様に PS を包含する名前空間の中で particle_simulator.h をインクルードする事で PS をネストさせ回避することが出来る。

ソースコード 2: 名前空間の衝突の回避方法

```
1  namespce hoge{
2      #include<particle_simulator.h>
3  }
```

これにより、ユーザーは PSL の API には hoge::PS:: という形で各種 API にアクセスできるようになる。

5.2 変数型の定義

PSL では以下のように変数型を提供する。以下に出てくる Vector2、Vector3 型については次節で説明する。

ソースコード 3: 変数型

```
1  namespace ParticleSimulator{
2      enum SEARCH_MODE{
3          SEARCH_MODE_LONG,
4          SEARCH_MODE_LONG_CUTOFF,
5          SEARCH_MODE_GATHER,
6          SEARCH_MODE_SCATTER,
7          SEARCH_MODE_SYMMETRY,
8          SEARCH_MODE_FIXED_LENGTH,
9      };
10
11     enum BOUNDARY_CONDITION{
12         BOUNDARY_CONDITION_PRRIODIC____,
13         BOUNDARY_CONDITION_PRRIODIC_X__,
14         BOUNDARY_CONDITION_PRRIODIC__Y_,
15         BOUNDARY_CONDITION_PRRIODIC___Z,
16         BOUNDARY_CONDITION_PRRIODIC_XY_,
17         BOUNDARY_CONDITION_PRRIODIC_X_Z,
18         BOUNDARY_CONDITION_PRRIODIC__YZ,
19         BOUNDARY_CONDITION_PRRIODIC_XYZ,
20         BOUNDARY_CONDITION_SHEARING_BOX,
21         BOUNDARY_CONDITION_USER_DEFINED,
22     };
23
24     typedef int S32;
25     typedef unsigned int U32;
26     typedef float F32;
27     typedef long S64;
28     typedef unsigned long U64;
29     typedef double F64;
30     typedef Vector2<F32> F32vec2;
31     typedef Vector3<F32> F32vec3;
32     typedef Vector2<F64> F64vec2;
33     typedef Vector3<F64> F64vec3;
34 #ifdef PARTICLE_SIMULATOR_TOW_DIMENSION
35     typedef F32vec2 F32vec;
36     typedef F64vec2 F64vec;
37     typedef F32ort2 F32ort;
38     typedef F64ort2 F64ort;
39     static const S32 N_CHILDREN = 4;
```

```
40 #else
41     typedef F32vec3 F32vec;
42     typedef F64vec3 F64vec;
43     typedef F32ort3 F32ort;
44     typedef F64ort3 F64ort;
45     static const S32 N_CHILDREN = 8;
46 #endif
47     MPI::Datatype MPI_F32VEC;
48     MPI::Datatype MPI_F64VEC;
49 }
```

5.3 Vector型

5.3.1 PS::Vector2型

x,yの2要素を持つ。それらの要素の型はPS::S32,PS::S64,PS::U32,PS::U64,PS::F32,PS::F64に限られる。外積演算も定義しており、結果はスカラーとして返す。このクラスは以下の様に記述される。

ソースコード 4: Vector2

```
1  namespace ParticleSimulator{
2      template <typename T>
3      class Vector2{
4      public:
5          //メンバ変数は以下の二つのみ。
6          T x, y;
7
8          //コンストラクタ
9          Vector2() : x(T(0)), y(T(0)) {}
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         //加減算
18         Vector2 operator + (const Vector2 & rhs) const;
19         const Vector2 & operator += (const Vector2 & rhs);
20         Vector2 operator - (const Vector2 & rhs) const;
```

```
21          const Vector2 & operator -= (const Vector2 & rhs);
22
23          //ベクトルスカラ積
24          Vector2 operator * (const T s) const;
25          const Vector2 & operator *= (const T s);
26          friend Vector2 operator * (const T s, const Vector2 & v
                );
27          Vector2 operator / (const T s) const;
28          const Vector2 & operator /= (const T s);
29
30          //内積
31          T operator * (const Vector2 & rhs) const;
32
33          //外積(返り値はスカラ!!)
34          T operator ^ (const Vector2 & rhs) const;
35
36          //Vector2<U>への型変換
37          template <typename U>
38          operator Vector2<U> () const;
39      };
40  }
```

5.3.1.1 コンストラクタ

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

- **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>(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>(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。

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

5.3.1.2 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。

5.3.1.3 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。

5.3.1.4 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。

5.3.1.5 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。

5.3.1.6 Vector2<U>への型変換

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

- **引数**

  rhs: 入力。const PS::Vector2<T> &型。

- **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。

5.3.2 PS::Vector3型

x,y,zの3要素を持つ。それらの要素の型はPS::S32,PS::S64,PS::U32,PS::U64,PS::F32,PS::F64に限られる。定義されているメソッド等はVector2型と同じであるが外積はVector2型と違いVector3型で返す。このクラスは以下の様に記述される。

ソースコード 5: 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;
7
8          //コンストラクタ
9          Vector3() : x(T(0)), y(T(0)) {}
10         Vector3(const T _x, const T _y) : x(_x), y(_y) {}
11         Vector3(const T s) : x(s), y(s) {}
12         Vector3(const Vector3 & src) : x(src.x), y(src.y) {}
13
14         //代入演算子
15         const Vector3 & operator = (const Vector3 & rhs);
16
17         //加減算
18         Vector3 operator + (const Vector3 & rhs) const;
19         const Vector3 & operator += (const Vector3 & rhs);
20         Vector3 operator - (const Vector3 & rhs) const;
21         const Vector3 & operator -= (const Vector3 & rhs);
22
23         //ベクトルスカラ積
24         Vector3 operator * (const T s) const;
25         const Vector3 & operator *= (const T s);
26         friend Vector3 operator * (const T s, const Vector3 & v
               );
27         Vector3 operator / (const T s) const;
28         const Vector3 & operator /= (const T s);
29
30         //内積
31         T operator * (const Vector3 & rhs) const;
32
33         //外積(返り値はスカラ!!)
```

```
34          T operator ^ (const Vector3 & rhs) const;
35
36          //Vector3<U>への型変換
37          template <typename U>
38          operator Vector3<U> () const;
39      };
40  }
```

5.3.2.1 コンストラクタ

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

- **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>(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>(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。

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

5.3.2.2 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。

5.3.2.3 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。

5.3.2.4 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。

5.3.2.5 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。

5.3.2.6 Vector3<U>への型変換

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

- **引数**

  const PS::Vector3<T>型。

- **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。

5.4 PSL 初期化、終了

5.4.1 概要

PSL を初期化又は終了するために必要な関数。

5.4.2 API

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

- **引数**

  argc: コマンドライン引数の総数

  argv: 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

- **返り値**

  なし。

- **機能**

  PSL の初期化を行う。PSL を使う前に読み出す必要がある。内部では MPI::Init を呼び出すため、引数 argc と argv が変わっている可能性がある。

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

- **引数**

  なし。

- **返り値**

  なし。

- **機能**

  PSL の終了処理を行う。

5.5 領域クラス (PS::DomainInfo)

5.5.1 概要

ルートドメインの情報を持ち、その分割を行うクラス。クラス内に全ドメインの境界と、ルートドメインの分割の際に使われるサンプル粒子の座標を持つ。ユーザー定義境界条件の場合、イメージドメインの情報も持つ。

5.5.1.1 ルートドメインの分割の方法

ルートドメインの分割を行う際、各ドメインから粒子の位置座標をサンプリングし、サンプリングされた粒子が各ドメインで均等になるように、ドメインの境界を決める。この際、サンプル粒子が少ないとポアソンノイズによる影響を受ける。この影響を減らすため、境界に指数移動平均を用いる。具体的にドメインの境界の決め方は以下の様になる。

$$X_{\mathrm{EMA,t+\Delta t}} = \alpha x_{t+\Delta t} + (1-\alpha) X_{\mathrm{EMA,t}} \quad (0 \leq \alpha \leq 1). \tag{1}$$

ここで、α は平滑化係数、$X_{\mathrm{EMA,t}}$ は時刻 t での平均化された境界、$x_{t+\Delta t}$ は時刻 $t+\Delta t$ での平均化する前の境界である。

領域クラス (DomainInfo) はルートドメインの分割の為に必要な平滑化係数と、サンプルする最大粒子数を内部の private メンバとして持つ。

領域クラス (DomainInfo) は以下のように記述される。

ソースコード 6: 領域クラス

```
1  namespace  ParticleSimulator{
2      class DomainInfo{
3      public:
4          void initialize(const S32 nsamplemax, const F32 alpha);
5          void initialize(std::string domainparam);
6          void setNProcMultiDim(const S32 nx, const S32 ny, const
                 S32 nz);
7          void setBoundaryCondition(enum BOUNDARY\_CONDITION);
8          void setPosOfRootDomain(const F32vec & low,
9                                  const F32vec & high);
10         void setNumberOfImageDomain(const S32 nimage);
11         void collectSample(const PS::ParticleSystem & psys,
12                            const bool clear=true)
13         void decomposeDomain();
14
15     };
16 }
```

5.5.1.2 境界条件

ユーザーは領域クラスのメソッド setBoundarConditino(enum) を使う事で、以下の境界条件のシミュレーションを行う事が出来る。具体的な引数については API のセクションを参照。

直方体型の周期境界条件

ユーザーは x,y,z, の中の任意の軸を周期境界、もしくは開放境界に設定する事が出来る。

シアリングボックス

ユーザーは x 軸を動径方向、y 軸を接戦方向とするシアリングボックスのシミュレーションを扱う事が出来る (y 軸方向は周期境界、x 軸方向のボックスがずれていく)。3 次元の場合 z 軸方向は開放境界となる。ボックスは y 軸方向にずれていくので、ユーザーは TreeForForce::setShiftY(const PS::F32) を使って y 軸方向のシフト量を設定する。この境界条件を扱う場合は短距離力でなくてはならず、PSL はカットオフ半径内にある粒子を全て見つける。

ユーザー定義境界条件

ユーザーはイメージ粒子を定義する関数を定義し (イメージマップ関数)、PS::DomainInfo::SetNumberOf PS::S32) を使って全イメージ数を設定する必要がある。その為、例えば、近距離力のシミュレーションでは、ユーザーは定義したイメージマップ関数が十分にシミュレーション領域を覆い尽くす様にしなければならない。もし、カットオフ半径内をユーザー定義イメージが十分に覆い尽くせていない場合も PS はエラーや、例外を送出する事はない。

5.5.2 API

5.5.2.1 初期化

```
void PS::DomainInfo::initialize(const PS::S32 nsamplemax,
                                const PS::F32 alpha=1.0)
```

- **引数**

  nsamplemax: 入力。const PS::S32 型。サンプル粒子数の最大値。

  alpha: 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。

- **返り値**なし。

- **機能**

  最大のサンプル粒子数と平滑化係数を設定し、サンプリングされる粒子の配列と全ドメインの境界の配列を確保する。実際にサンプルされる粒子数はここで設定した数と同じかわずかに小さくなる。最大のサンプル数が全粒子数より多い場合、もしくは全プロセス数より小さい場合は例外を送出する。

```
void PS::DomainInfo::initialize(std::string domainparam);
```

- **引数**

  domainparam: 入力。std::string 型。PS::DomainInfo に関するパラメータファイル名。

- **返り値**なし。

- **機能**

  最大のサンプル粒子数と平滑化係数を設定し、サンプリングされる粒子の配列と全ドメインの境界の配列を確保する。実際にサンプルされる粒子数はここで設定した数と同じがわずかに小さくなる。最大のサンプル数が全粒子数より多い場合、もしくは全プロセス数より小さい場合は例外を送出する。

  パラメータファイルの記述方法は以下のとおりである。

  一行に設定する値は一つとし、第一カラムで設定する変数を指定し、第二カラムにその値をいれる。カラムの間は空白で区切る。サンプル粒子の最大値を設定する場合は第一カラムに NSAMPLEMAX、平滑化係数を設定する場合は COEFFICIENT_OF_EMA。#で始まる行はコメントアウトされる。

  サンプル粒子の最大値が 10000 で、平滑化係数が 0.5 の時のファイルは以下の様に書く。

```
1 NSAMPLEMAX 10000
2 COEFFICIENT_OF_EMA  0.5
```

5.5.2.2 ルートドメイン分割数の設定

```
void PS::DomainInfo::setNProcMultiDim(const PS::S32 nx, const PS::S32 ny, const PS::S3
```

- **引数**

  nx: 入力。const PS::S32 型。x 軸方向のドメインの分割数。

  ny: 入力。const PS::S32 型。y 軸方向のドメインの分割数。

  nz: 入力。const PS::S32 型。z 軸方向のドメインの分割数。

- **返り値**

  なし。

- **機能**

  ルートドメインの分割数を設定する。nx,ny,nz(2 次元の場合は nx,ny) の積が全プロセス数にならない場合は例外を送出する。2 次元の場合は nz に任意の値を入れてよい。このメソッドを使わない場合は PSL が自動的に nx,ny,nz を決定する。

5.5.2.3 境界条件設定

```
void PS::DomainInfo::setBoundaryCondition(enum BOUNDARY\_CONDITION)
```

- **引数**

  BOUNDARY_CONDITION: 入力。enum 型。境界条件。

- **返り値**なし。

- **機能**

  引数に対応する、境界条件を設定する。このメソッドを呼ばない場合は自動的に開放境界条件となる。

BOUNDARY_CONDITION	=	PS::BOUNDARY_CONDITION_OPEN	開放境界条件。デフォルト。
	=	PS::BOUNDARY_CONDITION_PERIODIC_X	x 軸方向周期境界。y,z 軸開放境界
	=	PS::BOUNDARY_CONDITION_PERIODIC_Y	y 軸方向周期境界。x,z 軸開放境界
	=	PS::BOUNDARY_CONDITION_PERIODIC_Z	z 軸方向周期境界。x,y 軸開放境界
	=	PS::BOUNDARY_CONDITION_PERIODIC_XY	x,y 軸方向周期境界。z 軸開放境界
	=	PS::BOUNDARY_CONDITION_PERIODIC_XZ	x,z 軸方向周期境界。y 軸開放境界
	=	PS::BOUNDARY_CONDITION_PERIODIC_YZ	y,z 軸方向周期境界。x 軸開放境界
	=	PS::BOUNDARY_CONDITION_PERIODIC_XYZ	x,y,z 軸方向周期境界。
	=	PS::BOUNDARY_CONDITION_SHEARING_BOX	シアリングボックス。
	=	PS::BOUNDARY_CONDITION_USER_DEFINED	ユーザー指定境界条件。

5.5.2.4 非開放境界用ルートドメイン設定

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

- **引数**

  low: 入力。const PS::F32vec 型。ルートドメインの座標値が小さい側の頂点の座標。

  high: 入力。const PS::F32vec 型。ルートドメインの座標値が大きい側の頂点の座標。

- **返り値**なし。

- **機能**

  ルートドメインを設定する。開放境界の場合、このメソッドを使う必要はなく、PS がルートドメインの設定を自動で行う。また、軸方向に 0 を与えた場合も PS がドメインの設定を行う。

5.5.2.5 ユーザー定義境界条件用イメージドメイン数の設定

```
void PS::DomainInfo::setNumberOfImageDomain(const PS::S32 nimage)
```

- **引数**

  nimage: 入力。const PS::S32 型。イメージドメインの数。

- **返り値**なし。

- **機能**

  ユーザー指定境界条件の場合のイメージドメインの数を設定する。ユーザー指定境界条件以外の境界条件の場合に呼び出されると、例外を送出する。

5.5.2.6 サンプル粒子回収

```
void PS::DomainInfo::collectSample(const PS::ParticleSystem & psys,
                                   const PS::F32 wgh = 1.0,
                                   const bool clear=true)
```

- **引数**

  psys: 入力。const PS::ParticleSystem &型。

  wgh: 入力。PS::F32 型。サンプル数調整用重み。

  clear: 入力。const bool 型。クリアフラグ。デフォルト true。

- **返り値**

  なし。

- **機能**

  各プロセスが自分のドメインからいくつかの粒子の座標をサンプルし、そのサンプルをルートプロセスに送る。clear でサンプルする前にサンプル粒子の位置データを保持したメモリをクリアするかどうか決める。wgh で自分のドメインからサンプルする粒子の数を調整する。

  サンプル数の調整方法

  各ドメインから供出されるサンプル粒子の数が、1/wgh の比になるように、粒子サンプルを行う。例えば、計算時間でサンプル数の調整を行う場合、wgh に各プロセスでかかった計算時間を設定することで、時間のかかったプロセス程、担当する i 粒子が少なくなり、ロードバランスがとりやすくなる。

5.5.2.7 領域分割

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

- **引数**

  なし。

- **返り値**

  なし。

- **機能**

  全プロセスのドメインの境界を決定し、その座標を領域クラス内に格納する。

5.6 粒子群クラス (PS::ParticleSystem)

5.6.1 概要

FullParticle の配列を持ち、粒子の交換等を行うクラス。粒子の読み込みや書き込み等はこのクラスを通して行う。粒子種の数だけインスタンスを作る必要がある。

粒子群クラス (PS::ParticleSystem) は以下のように記述される。

ソースコード 7: 粒子群クラス

```
1  namespace ParticleSimulator{
2      template<class Tptcl>
3      class ParticleSystem{
4      public:
5
6          template <class Theader>
7          void readParticleAscii(const char * const filename,
8                                 const char * const format,
9                                 const Theader& header);
10         template <class Theader>
11         void readParticleAscii(const char * const filename,
12                                const Theader& header);
13         void readParticleAscii(const char * const filename,
14                                const char * const format);
15         void readParticleAscii(const char * const filename);
16
17         template <class Theader>
18         void writeParticleAscii(const char * const filename,
19                                 const char * const format,
20                                 Theader& header);
21         template <class Theader>
22         void writeParticleAscii(const char * const filename,
23                                 Theader& header);
24         void writeParticleAscii(const char * const filename,
25                                 const char * const format);
26         void writeParticleAscii(const char * const filename);
27
28
29         void createParticleArray(const S32 array_size);
30
31         template<class Tdinfo>
32         void exchangeParticle(const Tdinfo & dinfo);
33         Tptcl & operator [] (const S32 id);
```

```
34          const Tptcl & operator [] (const S32 id) const;
35          Tptcl * getParticlePointer(const S32 id=0);
36          S32 getNumberOfParticleLocal() const;
37          S64 getNumberOfParticleTotal() const;
38          S32 getSizeOfParticle() const;
39      };
40  }
```

5.6.2 API

5.6.2.1 ファイル入出力

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

- **引数**

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

  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。

- **返り値**

  なし。

- **機能**

  各プロセスが filename と format 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 として格納する。

  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 FullParticle のメンバ関数でユーザが定義する。名前は readAscii であり、引数は FILE*型である。例えば 3 次元の重力計算の場合、以下のように定義できる。読み込むべき 1 粒子のデータは、質量 (PS::F64 mass)、位置

(PS::F64vec pos)、速度 (PS::F64vec vel) であり、そのフォーマットが 10 進数アスキーであるとする。

```
void FullParticle::readAscii(FILE *fp){
    fscanf(fp, "%lf%lf%lf%lf%lf%lf%lf",
                  &this->mass,
                  &this->pos[0], &this->pos[1], &this->pos[2],
                  &this->vel[0], &this->vel[1], &this->vel[2]);
    return;
}
```

ユーザがこの関数を定義するに当って、以下の制限がある。

- 返値の型が void。
- 引数にファイルポインタを取り、このファイルポインタを入力先に指定すること

ファイルのヘッダのデータを読み取る関数は Theader のメンバ関数でユーザが定義する。名前は readAscii であり、引数は FILE*型である。もし、ヘッダーに粒子数が含まれていれば、この関数は粒子数を返さなければならない。この時、返ってきた粒子数分、ファイルから粒子データを読み込む。ヘッダーに粒子数が含まれていない場合は-1 以下の整数を返さなければならない。この時、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なおし、粒子データを読み込む。例えばヘッダーに粒子数 (nbody)、時刻 (time) が入っている場合、以下の様になる。

```
S32 Theader::readAscii(FILE *fp) {
    fscanf(fp, "%d%lf", &nbody, &time);
    return nbody;
}
```

ユーザがこの関数を定義するに当って、以下の制限がある。

- 返値の型が PS::S32。
- 引数にファイルポインタを取り、このファイルポインタを入力先に指定すること

```
void PS::ParticleSystem::readParticleAscii(const char * const filename,
                                           const char * const format);
```

- **引数**

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

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

- **返り値**

  なし。

- **機能**

  各プロセスが filename と format 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 として格納する。この時、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なおし、粒子データを読み込む。

  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。format の指定の仕方は、Theader が存在する場合の時と同様である。

  1 粒子のデータを読み取る関数は FullParticle のメンバ関数でユーザが定義する。このメンバ関数の書式と規約は、分散ファイルから読み出す場合と同様である。

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

- **引数**

  filename: 入力。const char * const 型。入力ファイル名。

  header: 入力。Theader&型。ファイルのヘッダ情報。

- **返り値**

  なし。

- **機能**

  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 として格納した後、各プロセスに分配する。

  1 粒子のデータを読み取る関数は FullParticle のメンバ関数でユーザが定義する。ファイルのヘッダのデータを読み取る関数は Theader のメンバ関数でユーザが定義する。これら 2 つのメンバ関数の書式と規約は、分散ファイルから読み出す場合と同様である。

```
void PS::ParticleSystem::readParticleAscii(const char * const filename);
```

- **引数**

  filename: 入力。const char * const 型。入力ファイル名。

- **返り値**

  なし。

- **機能**

  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 として格納した後、各プロセスに分配する。この時、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なおし、粒子データを読み込む。

  1 粒子のデータを読み取る関数は FullParticle のメンバ関数でユーザが定義する。このメンバ関数の書式と規約は、分散ファイルから読み出す場合と同様である。

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

- **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。

- **返り値**

  なし。

- **機能**

  各プロセスが filename と format で指定された出力ファイルに FullParticle 型の粒子データと、Theader 型のヘッダー情報を出力する。出力ファイルのフォーマットはメンバ関数 PS::ParticleSystem::readParticleAscii と同様である。

  1 粒子のデータを書き込む関数は FullParticle のメンバ関数でユーザが定義する。名前は writeAscii であり、引数は FILE*型である。例えば重力計算の場合、以下のように定義できる。読み込むべき 1 粒子のデータは、質量 (PS::F64 mass)、位置 (PS::F64vec pos)、速度 (PS::F64vec vel) であり、そのフォーマットが 10 進数アスキーであるとする。

```
void FullParticle::writeAscii(FILE *fp) const{
    fprintf(fp, "%lf%lf%lf%lf%lf%lf%lf",
                this->mass,
                this->pos[0], this->pos[1], this->pos[2],
                this->vel[0], this->vel[1], this->vel[2]);
}
```

  ユーザがこの関数を定義するに当って、以下の制限がある。

- 返値の型が void。
- 引数にファイルポインタを取り、このファイルポインタを入力先に指定すること。
- const メンバ関数であること。

ファイルのヘッダのデータを書き込む関数は Theader のメンバ関数でユーザが定義する。名前は writeAscii であり、引数は FILE*型である。例えばヘッダーに粒子数 (nbody)、時刻 (time) が入っている場合、以下の様になる。

```
void Theader::writeAscii(FILE *fp) const{
     fprintf(fp, "%d%lf", nbody, time);
}
```

ユーザがこの関数を定義するに当って、以下の制限がある。

- 返値の型が void。
- 引数にファイルポインタを取り、このファイルポインタを入力先に指定すること。
- const メンバ関数であること。

```
void PS::ParticleSystem::void writeParticleAscii(const char * const filename,
                                                 const char * const format);
```

- **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。

- **返り値**

  なし。

- **機能**

  各プロセスが filename と format で指定された出力ファイルに FullParticle 型の粒子データを出力する。出力ファイルのフォーマットはメンバ関数 PS::ParticleSystem::readParticleAs と同様である。

  1 粒子のデータを書き込む関数は FullParticle のメンバ関数でユーザが定義する。このメンバ関数の書式と規約は、分散ファイルに書き込む場合と同様である。

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

- **引数**

  filename: 入力。const char * const 型。出力ファイル名。

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

- **返り値**

  なし。

- **機能**

  各プロセスが filename で指定された出力ファイルに FullParticle 型の粒子データと、Theader 型のヘッダー情報を出力する。

  1 粒子のデータを書き込む関数は FullParticle のメンバ関数でユーザが定義する。ファイルのヘッダのデータを書き込む関数は Theader のメンバ関数でユーザが定義する。これら 2 つのメンバ関数の書式と規約は、分散ファイルに書き込む場合と同様である。

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

- **引数**

  filename: 入力。const char * const 型。出力ファイル名。

- **返り値**

  なし。

- **機能**

  各プロセスが filename で指定された出力ファイルに FullParticle 型の粒子データを出力する。

  1 粒子のデータを書き込む関数は FullParticle のメンバ関数でユーザが定義する。このメンバ関数の書式と規約は、分散ファイルに書き込む場合と同様である。

5.6.2.2 粒子交換

```
void PS::ParticleSystem::exchangeParticle(const PS::DomainInfo & dinfo)
```

- **引数**

  dinfo: 入力。const PS::DomainInfo &型。

- **返り値**

  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。どのドメインにも属さない粒子が現れた場合、例外を送出する。

5.6.2.3 その他

```
FullPtcl & PS::ParticleSystem::operator [] (const PS::S32 id)
```

- **引数**

  id: const PS::S32 型。

- **返り値**

  FullPtcl &型。id 番目の粒子の参照を返す。

```
FullPtcl * PS::ParticleSystem::getParticlePointer(const PS::S32 id)
```

- **引数**

  id: 入力。const PS::S32 型。

- **返り値**

  FullPtcl *型。id 番目の粒子へのポインタを返す。

```
PS::S32 PS::ParticleSystem::getNumberOfParticleLocal()
```

- **引数**

  なし。

- **返り値**

  PS::S32 型。自身が担当する粒子数を返す。

```
PS::S64 PS::ParticleSystem::getNumberOfParticleGlobal()
```

- **引数**

  なし。

- **返り値**

  PS::S64 型。全プロセスでの粒子の総数を返す。

```
PS::S32 PS::ParticleSystem::getSizeOfParticle()
```

- **引数**

  なし。

- **返り値**

  PS::S32 型。自身が持つ粒子配列のサイズを返す。

5.7 相互作用クラス (PS::TreeForForce)

5.7.1 概要

PS::ParticleSystemから粒子の情報を読み取り、内部でツリーを構築し相互作用の計算を行い、結果を格納するクラス。PS::ParticleSystemに結果を書き戻す事も出来る。内部には相互作用演算を効率的に行う為の3種類の粒子型EssentialParticleI型、EssentialParticleJ型、SuperParticleJ型、また、相互作用の結果を格納するForce型、がありさらに、ツリー構造を実現するためのTreeParticle型、TreeCell型という、全6種類のクラスを配列として持っている。これらの配列はprvate空間に存在し、ユーザーが直接アクセスする事は出来ない。

相互作用クラス (TreeForForce) は以下のように記述される。

ソースコード 8: 相互作用クラス

```
1     template<int SEARCH_MODE, class Tforce, class Tepi,
2         class Tepj, class Tmomloc, class Tmomglb, class Tspj>
3     class TreeForForce{
4     public:
5         void initialize(const S64 ntot,
6                         const F32 theta,
7                         const S32 nleafmax,
8                         const S32 ngroupmax);
9
10        %void initializeLocalTree(const F32 len_half);
11
12        template<class Tpsys>
13        void setParticleLocalTree(const Tpsys & psys,
14                                  const bool clear=true);
15
16        void makeLocalTree(const bool cancel=false);
17
18        void makeGlobalTree(const bool cancel=false);
19
20        void exchangeLocalEssentialTree(const DomainInfo &
              dinfo);
21
22        void setLocalEssentialTreeToGlobalTree();
23
24        Tforce & getForce(const S32 id);
25
26        template<class Tfunc_ep_ep, class Tpsys>
27        void calcForceAll(Tfunc_ep_ep pfunc_ep_ep,
```

```
28                          Tpsys & psys,
29                          DomainInfo & dinfo,
30                          const bool clear_force = true);
31
32          template<class Tfunc_ep_ep, class Tpsys>
33          void calcForceAllAndWriteBack(Tfunc_ep_ep pfunc_ep_ep,
34                                        Tpsys & psys,
35                                        DomainInfo & dinfo,
36                                        const bool clear_force =
                                              true);
37
38          template<class Tfunc_ep_ep, class Tfunc_ep_sp, class
                Tpsys>
39          void calcForceAll(Tfunc_ep_ep pfunc_ep_ep,
40                          Tfunc_ep_sp pfunc_ep_sp,
41                          Tpsys & psys,
42                          DomainInfo & dinfo,
43                          const bool clear_force=true);
44
45          template<class Tfunc_ep_ep, class Tfunc_ep_sp, class
                Tpsys>
46          void calcForceAllAndWriteBack(Tfunc_ep_ep pfunc_ep_ep,
47                                        Tfunc_ep_sp pfunc_ep_sp,
48                                        Tpsys & psys,
49                                        DomainInfo & dinfo,
50                                        const bool clear_force=
                                              true);
51
52          template<class Tfunc_ep_ep, class Tpsys>
53          void makeTreeAndCalcForce(Tfunc_ep_ep pfunc_ep_ep,
54                                    DomainInfo & dinfo,
55                                    const bool clear_force = true
                                          );
56
57          template<class Tfunc_ep_ep, class Tfunc_ep_sp, class
                Tpsys>
58          void makeTreeAndCalcForce(Tfunc_ep_ep pfunc_ep_ep,
59                                    Tfunc_ep_sp pfunc_ep_sp,
60                                    DomainInfo & dinfo,
61                                    const bool clear_force=true){
```

```
62
63
64
65          template<class Tfunc_ep_ep, class Tfunc_ep_sp, class
                Tpsys>
66          void calcForceAll(Tfunc_ep_ep pfunc_ep_ep,
67                            Tfunc_ep_sp pfunc_ep_sp,
68                            DomainInfo & dinfo,
69                            const bool clear_force=true);
70      };
71  }
```

テンプレートパラメータの第一引数である SEARCH_MODE により、ツリーウォークの方法を選択できる。

SEARCH_MODE	=	PS::SEARCH_MODE_LONG	cutoff なし長距離力モード。
	=	PS::SEARCH_MODE_LONG_CUTOFF	cutoff あり長距離力モード。
	=	PS::SEARCH_MODE_GATHER	収集モード。
	=	PS::SEARCH_MODE_SCATTER	散乱モード。
	=	PS::SEARCH_MODE_SYMMETRY	対称モード。

テンプレートパラメータの第二引数は FORCE クラス、第三引数は EPI クラス、第四引数は EPJ クラス。第五、六引数はそれぞれローカルツリー、グローバルツリーの持つ MOMENT クラス。第七引数は SPJ クラスである。

TreeForForce はテンプレートパラメータが多いため、以下の様なラッパークラスが用意されており、ユーザーはこちらを使う事が推奨される。長距離力の場合はテンプレート引数が FORCE, EPI, EPJ, MOMENT, SPJ の 5 つ。短距離量の場合はテンプレート引数が FORCE, EPI, EPJ の 3 つである。長距離力の場合はさらに、四重極モーメント迄の計算については MOMENT, SPJ クラスが用意されており、それに対応したラッパーも存在する。

```
1
2  namespace ParticleSimulator{
3      template<class Tforce, class Tepi, class Tepj, class Tmom=
           void, class Tsp=void>
4      class TreeForForceLong{
5      public:
6          typedef TreeForForce
7          <SEARCH_MODE_LONG,
8           Tforce, Tepi, Tepj,
9           Tmom, Tmom, Tsp> Normal; // cutoffなし長距離力モード
10
```

```
11          typedef TreeForForce
12          <SEARCH_MODE_LONG_CUTOFF,
13           Tforce, Tepi, Tepj,
14           Tmom, Tmom, Tsp> WithCutoff; //
                 cutoffあり長距離力モード
15      };
16
17      template<class Tforce, class Tepi, class Tepj>
18      class TreeForForceLong<Tforce, Tepi, Tepj, void, void>{
19      public:
20          typedef TreeForForce
21          <SEARCH_MODE_LONG,
22           Tforce, Tepi, Tepj,
23           MomentMonopole,
24           MomentMonopole,
25           SPJMonoPole> Monopole;
26
27          typedef TreeForForce
28          <SEARCH_MODE_LONG,
29           Tforce, Tepi, Tepj,
30           MomentDipole,
31           MomentDipole,
32           SPJMonoPole> Dipole;
33
34          typedef TreeForForce
35          <SEARCH_MODE_LONG,
36           Tforce, Tepi, Tepj,
37           MomentQuadrupole,
38           MomentQuadrupole,
39           SPJMonoPole> Quadrupole;
40      };
41
42      template<class Tforce, class Tepi, class Tepj>
43      class TreeForForceShort{
44      public:
45          typedef TreeForForce
46          <SEARCH_MODE_SYMMETRY,
47           Tforce, Tepi, Tepj,
48           MomentSearchInAndOut,
49           MomentSearchInAndOut,
```

```
50          SuperParticleBase> Symmetry; // 短距離、対称モード
51
52          typedef TreeForForce
53          <SEARCH_MODE_GATHER,
54           Tforce, Tepi, Tepj,
55           MomentSearchInAndOut,
56           MomentSearchInOnly,
57           SuperParticleBase> Gather; // 短距離、収集モード
58
59          typedef TreeForForce
60          <SEARCH_MODE_SCATTER,
61           Tforce, Tepi, Tepj,
62           MomentSearchInOnly,
63           MomentSearchInAndOut,
64           SuperParticleBase> Scatter; // 短距離、散乱モード
65      };
66  }
```

5.7.2 API

5.7.2.1 初期化

```
void PS::TreeForForce::initialize(const PS::S64 ntot,
                                  const PS::F32 theta=0.5,
                                  const PS::S32 nleafmax=8,
                                  const PS::S32 ngroupmax=64)
```

- **引数**

  ntot: 入力。const PS::S64 型。相互作用にかかわる全粒子数。

  theta: 入力。const PS::F32 型。ツリーのオープニングクライテリオン。長距離相互作用において、あるツリーセルからある粒子 (実際には i 粒子グループ) への力を計算する時に見込角が theta 以下の時はツリーセルの多重極モーメントを用いて相互作用を評価する。近距離力では任意の値でよい。デフォルト 0.5。

  nleafmax: 入力。const PS::S32 型。ツリーリーフセル内の最大粒子数。あるセル内にこの数を超える粒子が入っていた場合、そのセルはさらに分割される。デフォルト 8。

  ngroupmax: 入力。const PS::S32 型。i 粒子グループ内の最大粒子数。i 粒子グループとは相互作用リスト（EssentialParticleJ 型、SuperParticleJ 型の配列）を共有する i 粒子のグループである。ツリー構造にそってグルーピングする為、グループ内の粒

子は固まって存在している。グループの粒子数はngroupmaxを超えないようにグルーピングが行われる。デフォルト64。

- **返り値**

  なし。

- **機能**

  相互作用ツリーの初期設定を行い、EssentialParticleI型、EssentialParticleJ型、SuperParticleJ型、Force型、PS::TreeParticle型、PS::TreeCell型のメンバの配列を確保する。配列のサイズを推定するのにntotを使う。自クラス内にi粒子グループ内の最大粒子数とツリーリーフセル内の最大粒子数をセットする。thetaは長距離力用のツリーのオープニングクライテリオンであり、短距離力の場合は値は何でもよい。

半分の長さ。

5.7.2.2 粒子読み込み

```
void PS::TreeForForce::setParticleLocalTree(
                        const PS::ParticleSystem & psys,
                        const bool clear=true);
```

- **引数**

  psys: 入力。const PS::ParticleSystem &型。

  clear: 入力。const bool clear型。クリアフラグ。デフォルトtrue。

- **返り値**

  なし。

- **機能**

  psysからFullParticleを読み込み、メンバのPS::TreeParticle型、EssentailParticleI、EssentailParticleJ型の配列にツリー生成や力の計算に必要な情報をコピーする。

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

- **引数**

- **返り値**

  なし。

- **機能**

  exchangeLocalEssentialTreeで送られて来た粒子をTreeParticle型に変換する。

5.7.2.3 ツリー生成

```
void PS::TreeForForce::makeLocalTree(const bool cancel=false);
```

- **引数**

  cancel: 入力。const bool 型。キャンセルフラグ。デフォルト false。

- **返り値**

  なし。

- **機能**

  LT を作り、モートンソートを行い、モーメント計算まで行う。LT づくりとモートンソートは cancel が true なら行わない。

```
void PS::TreeForForce::exchangeLocalEssentialTree()
```

- **引数**

  なし。

- **返り値**

  なし。

- **機能**

  LocalEssentialTree を作り、各プロセスに送る。イメージ粒子が存在する場合は、それらも LocalEssentialTree の中に組み込まれる。

```
void PS::TreeForForce::makeGlobalTree(const bool cancel=false);
```

- **引数**

  cancel: 入力。const bool 型。キャンセルフラグ。デフォルト false。

- **返り値**

  なし。

- **機能**

  GT を作り、モートンソートを行い、モーメント計算まで行う。GT づくりとモートンソートは cancel が true なら行わない。

5.7.2.4 力の計算

以下に述べる6つの関数はテンプレート関数となっており、関数の引数として Tfunc_ep_ep 型や Tfunc_ep_sp 型を取る。これらはユーザーが定義した相互作用関数のポインタもしくは相互作用関数オブジェクトを取る。

関数ポインタを使う場合

ソースコード 9: 関数ポインタを使う場合

```
1 void CalcForceEpEp(const EssentialPtclI * ep_i,
2                     const PS::S32 n_ip,
3                     const EssentialPtclJ * ep_j,
4                     const PS::S32 n_jp,
5                     ResultForce * force){
6 ......
7 }
```

関数オブジェクトを使う場合

ソースコード 10: 関数ポインタを使う場合

```
1 class CalcForceEpEp{
2 public:
3     void operator()(const EssentialPtclI * ep_i,
4                     const PS::S32 n_ip,
5                     const EssentialPtclJ * ep_j,
6                     const PS::S32 n_jp,
7                     ResultForce * force){
8 ......
9     }
10 }
```

関数ポインタ、関数オブジェクトどちらを使う場合も返り値は void 型とし、引数は第一引数から順に const EssentialParticleI *型、PS::S32 型、const EssentialParticleJ *型、PS::S32 型、Force *型。

```
template<class Tfunc_ep_ep>
void PS::TreeForForce::calcForce( Tfunc_ep_ep func_ep_ep);
```

- **引数**

  (func_ep_ep): 入力。返り値が void 型の EssentialParticleI と EssentialParticleJ 間の相互作用計算用関数ポインタ、もしくは関数オブジェクト。関数の引数は第一引数から順に const EssentialParticleI *型、PS::S32 型、const EssentialParticleJ *型、PS::S32 型、Force *型。

- **返り値**

  なし。

- **機能**

  全粒子に対して、GT 内のツリーウォークと相互作用計算を行い、結果を PS::TreeForForce::setParticleFromFullParticle() で読み込んだ時の粒子の並びと同じ順番で格納する。この関数を呼ぶ前に PS::TreeForForce::makeIPGroup() で、i 粒子をグルーピングしておく必要がある。

```
template<class Tfunc_ep_ep>
void PS::TreeForForce::calcForceAll( Tfunc_ep_ep func_ep_ep,  PS::ParticleSystem & psy
```

- **引数**

  (func_ep_ep): 入力。返り値が void 型の EssentialParticleI と EssentialParticleJ 間の相互作用計算用関数ポインタ、もしくは関数オブジェクト。関数の引数は第一引数から順に const EssentialParticleI *型、PS::S32 型、const EssentialParticleJ *型、PS::S32 型、Force *型。

  psys: 入力。PS::ParticleSystem &型。

- **返り値**

  なし。

- **機能**

  粒子移動、LT 作り、LET 交換、GT 作り、相互作用計算まで行い、結果を PS::TreeForForce::setParticleFromFullParticle() で読み込んだ時の粒子の並びと同じ順番で格納する。

```
template<class Tfunc_ep_ep>
void PS::TreeForForce::calcForceAll( Tfunc_ep_ep func_ep_ep,  PS::ParticleSystem & psy
```

- **引数**

  (func_ep_ep): 入力。返り値が void 型の EssentialParticleI と EssentialParticleJ 間の相互作用計算用関数ポインタ、もしくは関数オブジェクト。関数の引数は第一引数から順に const EssentialParticleI *型、PS::S32 型、const EssentialParticleJ *型、PS::S32 型、Force *型。

  psys: 入力。PS::ParticleSystem &型。

- **返り値**

  なし。

- **機能**

  粒子移動、LT 作り、LET 交換、GT 作り、相互作用計算まで行い、結果を PS::TreeForForce::setParticleFromFullParticle() で読み込んだ時の粒子の並びと同じ順番で格納し、さらに psys にも書き戻す。

```
template<class Tfunc_ep_ep, class Tfunc_ep_sp>
void PS::TreeForForce::calcForce( Tfunc_ep_ep func_ep_ep, Tfunc_ep_sp func_epsp);
```

- **引数**

  (func_ep_ep): 入力。返り値が void 型の EssentialParticleI と EssentialParticleJ 間の相互作用計算用関数ポインタ、もしくは関数オブジェクト。関数の引数は第一引数から順に const EssentialParticleI *型、PS::S32 型、const EssentialParticleJ *型、PS::S32 型、Force *型。

  (func_ep_sp): 入力。返り値が void 型の EssentialParticleI と SuperParticleJ 間の相互作用計算用関数ポインタ、もしくは関数オブジェクト。関数の引数は第一引数から順に const EssentialParticleI *型、PS::S32 型、const SuperParticleJ *型、PS::S32 型、Force *型。

- **返り値**

  なし。

- **機能**

  全粒子に対して、GT 内のツリーウォークと相互作用計算を行い、結果を PS::TreeForForce::setParticleFromFullParticle() で読み込んだ時の粒子の並びと同じ順番で格納する。この関数を呼ぶ前に PS::TreeForForce::makeIPGroup() で、i 粒子をグルーピングしておく必要がある。

```
template<class Tfunc_ep_ep, class Tfunc_ep_sp>
void PS::TreeForForce::calcForceAll( Tfunc_ep_ep func_ep_ep,   Tfunc_ep_sp func_ep_sp,
                                     PS::ParticleSystem & psys)
```

- **引数**

  (func_ep_ep): 入力。返り値が void 型の EssentialParticleI と EssentialParticleJ 間の相互作用計算用関数ポインタ、もしくは関数オブジェクト。関数の引数は第一引数から順

に const EssentialParticleI *型、PS::S32 型、const EssentialParticleJ *型、PS::S32 型、Force *型。

(func_ep_sp): 入力。返り値が void 型の EssentialParticleI と SuperParticleJ 間の相互作用計算用関数ポインタ、もしくは関数オブジェクト。関数の引数は第一引数から順に const EssentialParticleI *型、PS::S32 型、const SuperParticleJ *型、PS::S32 型、Force *型。

psys: 入力。PS::ParticleSystem &型。

- **返り値**

  なし。

- **機能**

  粒子移動、LT 作り、LET 交換、GT 作り、相互作用計算まで行い、結果を PS::TreeForForce::setParticleFromFullParticle() で読み込んだ時の粒子の並びと同じ順番で格納する。

```
template<class Tfunc_ep_ep, class Tfunc_ep_sp>
void PS::TreeForForce::calcForceAllAndWriteBack( Tfunc_ep_ep func_ep_ep,  Tfunc_ep_sp
                                                 PS::ParticleSystem & psys)
```

- **引数**

  (func_ep_ep): 入力。返り値が void 型の EssentialParticleI と EssentialParticleJ 間の相互作用計算用関数ポインタ、もしくは関数オブジェクト。関数の引数は第一引数から順に const EssentialParticleI *型、PS::S32 型、const EssentialParticleJ *型、PS::S32 型、Force *型。

  (func_ep_sp): 入力。返り値が void 型の EssentialParticleI と SuperParticleJ 間の相互作用計算用関数ポインタ、もしくは関数オブジェクト。関数の引数は第一引数から順に const EssentialParticleI *型、PS::S32 型、const SuperParticleJ *型、PS::S32 型、Force *型。

  psys: 入力。PS::ParticleSystem &型。

- **返り値**

  なし。

- **機能**

  粒子移動、LT 作り、LET 交換、GT 作り、相互作用計算まで行い、結果を PS::TreeForForce::setParticleFromFullParticle() で読み込んだ時の粒子の並びと同じ順番で格納し、さらに psys にも書き戻す。

```
void PS::TreeForForce::makeIPGroup()
```

- **引数**

  なし。

- **返り値**

  なし。

- **機能**

  i 粒子をツリー構造にしたがってグルーピングする。グループ内の最大粒子数は PS::TreeForForce::initialize() の第四引数でセットされた値である。

```
PS::S32 & PS::TreeForForce::getNumberOfIPG()
```

- **引数**

  なし。

- **返り値**

  PS::S32 型。i 粒子グループの数を返す。

5.7.2.5 ネイバーリスト

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

- **テンプレート引数**

  Tsearchmode: 入力。SEARCH_MODE 型。

  Tptcl: 入力。Tptcl 型。入力しなくて OK。

- **引数**

  ptcl: 入力。const Tptcl &型。

  nnp: 出力。PS::S32 &型。

  epj: 出力。Tepj * (&) 型。

- **返り値**

  なし

- 機能

  ある１つの粒子のネイバーリストを返す。ネイバーリストの探し方をテンプレート第１引数 Tsearchmode で指定する。この相互作用ツリークラスの SEARCH_MODE 型が収集モードである場合に、Tsearchmode に散乱モードや対称モードを選んだら、コンパイルエラーとなる。テンプレート第２引数 Tptcl には入力の必要はないが、もし入力する場合は第１引数の型と同じでないと、コンパイルエラーとなる。粒子の指定を第１引数 ptcl で行う。この型はメンバ関数に getPos を含む必要がある。第２引数 nnp にネイバー粒子の数を返す。第３引数 epj にネイバーリストを返す。epj の型はこの相互作用ツリークラスの EssentialParticleJ 型と同じ型でなければならず、そうでない場合はコンパイルエラーとなる。epj のメモリの確保はこの関数内で行う。このリストの寿命は、次にネイバーリストを作る関数を呼ぶまでとする。

```
void PS::TreeForForce::getNeighborListOneIPGroup
               (const PS::S32 iipg,
                PS::S32 & nip,
                const Tepi * epi,
                PS::S32 & nnp,
                Tepj * (& epj));
```

- **引数**

  iipg: 入力。const PS::S32 型。

  nip: 出力。PS::S32 &型。

  epi: 出力。const epi *型。

  nnp: 出力。PS::S32 &型。

  epj: 出力。Tepj * (&) 型。

- **返り値**

  なし

- 機能

  この相互作用ツリークラスのある１つの i グループのネイバーリストの和集合 (以下ネイバーリストと省略) を返す。i グループの指定を第１引数 iipg で行う。ユーザーは i グループの数を PS::TreeForForce::getNumberOfIPG で知ることはできるので、for ループでまわせば、全 i グループのネイバーリストを得ることができる。第２引数 nip に指定した i グループの粒子の数を返す。第３引数 epi にこの i グループの粒子リスト

を返す。この型はこの相互作用ツリークラスの EssentialParticleI 型と同じである必要があり、そうでないとコンパイルエラーとなる。第４引数 nnp にこの i グループのネイバー粒子の数を返す。第５引数 epj にこの i グループのネイバーリストを返す。epj の型はこの相互作用ツリークラスの EssentialParticleJ 型と同じ型でなければならず、そうでない場合はコンパイルエラーとなる。epi と epj のメモリの確保は、この関数内で行う。これらの寿命は getNeighborListOneParticle のネイバーリストの寿命と同じ。

```
template <class Tsearchmode,
          class TTreeForForce,
          class Tepi2>
void PS::TreeForForce::getNeighborListOneIPGroup
               (const PS::S32 iipg,
                const TTreeForForce & ttf,
                PS::S32 & nip,
                const Tepi2 * epi,
                PS::S32 & nnp,
                Tepj * (& epj));
```

- **テンプレート引数**

  Tsearchmode: 入力。Tsearchmode 型。

  TTreeForForce: 入力。TTreeForForce 型。入力しなくても OK。

  Tepi2: 入力。Tepi2 型。入力しなくても OK。

- **引数**

  iipg: 入力。const PS::S32 型。

  ttf: 入力。const TTreeForForce &型。

  nip: 出力。PS::S32 &型。

  epi: 出力。const epi2 *型。

  nnp: 出力。PS::S32 &型。

  epj: 出力。Tepj * (&) 型。

- **返り値**

  なし

- 機能

  相互作用クラスのインスタンス ttf に属すある１つの i グループのネイバーリストの和集合 (以下ネイバーリストと省略) を返す。ネイバーリストの探し方をテンプレー

ト第１引数 Tsearchmode で指定する。ネイバーリストを返す側の相互作用クラスの SEARCH_MODE 型が収集モードである場合に、テンプレート第１引数に散乱モードや対称モードを選んだら、コンパイルエラーとなる。テンプレート第２ (TTreeForForce)、３引数 (Tepi2) に入力の必要はないが、入力する場合、それぞれ第２引数 ttf と第４引数 epi と同じでなければ、コンパイルエラーとなる。i グループの指定を第１引数 iipg で行う。ユーザーは i グループの数を ttf.getNumberOfIPG で知ることはできるので、for ループでまわせば、全 i グループのネイバーリストを得ることができる。第２引数 ttf でこの i グループの属す相互作用ツリークラスのインスタンスを指定する。第３引数 nip に i グループの粒子の数を返す。第４引数 epi にこの i グループの粒子リストを返す。epi の型は ttf の EssentialParticleI 型と同じである必要があり、そうでないとコンパイルエラーとなる。第５引数 nnp にこの i グループのネイバー粒子の数を返す。第６引数 epj にこの i グループのネイバーリストを返す。epj の型はネイバーリストを返す側の相互作用ツリークラスの EssentialParticleJ 型と同じ型でなければならず、そうでない場合はコンパイルエラーとなる。epi と epj のメモリの確保は、この関数内で行う。これらの寿命は getNeighborListOneParticle のネイバーリストの寿命と同じ。

```
void PS::TreeForForce::getNeighborListOneIPGroupEachParticle
              (const PS::S32 iipg,
               PS::S32 & nip,
               const Tepi * epi,
               PS::S32 * (& nnp),
               Tepj * (& epj));
```

- **引数**

  iipg: 入力。const PS::S32 型。

  nip: 出力。PS::S32 &型。

  epi: 出力。const epi *型。

  nnp: 出力。PS::S32 * (&) 型。

  epj: 出力。Tepj * (&) 型。

- **返り値**

  なし

- 機能

  この相互作用ツリークラスのある１つの i グループの粒子それぞれのネイバーリストを返す。i グループの指定を第１引数 iipg で行う。ユーザーは i グループの数を PS::TreeForForce::getNumberOfIPG で知ることはできるので、for ループでまわせば、全 i グループのネイバーリストを得ることができる。第２引数 nip に指定した i

グループの粒子の数を返す。第３引数 epi にこの i グループの粒子リストを返す。この型はこの相互作用ツリークラスの EssentialParticleI 型と同じである必要があり、そうでないとコンパイルエラーとなる。第４引数 nnp と第５引数 epj に返すものは以下の通りである。epj には、リスト epi の粒子順にネイバーリストを返す。ネイバーリスト内の変位を第４引数 nnp に返す。epj の型はこの相互作用ツリークラスの EssentialParticleJ 型と同じ型でなければならず、そうでない場合はコンパイルエラーとなる。epi、nnp、epj のメモリの確保は、この関数内で行う。これらのリストの寿命は getNeighborListOneParticle のネイバーリストの寿命と同じ。

```
template <class Tsearchmode,
          class TTreeForForce,
          class Tepi2>
void PS::TreeForForce::getNeighborListOneIPGroupEachParticle
               (const PS::S32 iipg,
                const TTreeForForce & ttf,
                PS::S32 & nip,
                const Tepi2 * epi,
                PS::S32 * (& nnp),
                Tepj * (& epj));
```

- **テンプレート引数**

  Tsearchmode: 入力。Tsearchmode 型。

  TTreeForForce: 入力。TTreeForForce 型。入力しなくても OK。

  Tepi2: 入力。Tepi2 型。入力しなくても OK。

- **引数**

  iipg: 入力。const PS::S32 型。

  ttf: 入力。const TTreeForForce &型。

  nip: 出力。PS::S32 &型。

  epi: 出力。const epi2 *型。

  nnp: 出力。PS::S32 * (&) 型。

  epj: 出力。Tepj * (&) 型。

- **返り値**

  なし

- 機能

相互作用クラスのインスタンス ttf に属すある 1 つの i グループの粒子それぞれのネイバーリストを返す。ネイバーリストの探し方をテンプレート第 1 引数 Tsearchmode で指定する。ネイバーリストを返す側の相互作用クラスの SEARCH_MODE 型が収集モードである場合に、Tsearchmode に散乱モードや対称モードを選んだら、コンパイルエラーとなる。テンプレート第 2 (TTreeForForce)、3 引数 (Tepi2) に入力の必要はないが、入力する場合、それぞれ第 2 引数 ttf と第 4 引数 epi と同じでなければ、コンパイルエラーとなる。i グループの指定を第 1 引数 iipg で行う。ユーザーは i グループの数を ttf.getNumberOfIPG で知ることはできるので、for ループでまわせば、全 i グループのネイバーリストを得ることができる。第 2 引数 ttf でこの i グループの属す相互作用ツリークラスのインスタンスを指定する。第 3 引数 nip に i グループの粒子の数を返す。第 4 引数 epi にこの i グループの粒子リストを返す。epi の型は ttf の EssentialParticleI 型と同じである必要があり、そうでないとコンパイルエラーとなる。第 5 引数 nnp と第 6 引数 epj に返すものは以下の通りである。epj には、リスト epi の粒子順にネイバーリストを返す。ネイバーリスト内の変位を nnp に返す。epj の型はこの相互作用ツリークラスの EssentialParticleJ 型と同じ型でなければならず、そうでない場合はコンパイルエラーとなる。epi、nnp、epj のメモリの確保は、この関数内で行う。これらの寿命は getNeighborListOneParticle のネイバーリストの寿命と同じ。

5.7.2.6 その他

```
FORCE & PS::TreeForForce::getForce(const PS::S32 id)
```

- **引数**

  id: 入力。const PS::S32。粒子インデクス。

- **返り値**

  Force &型。id 番目に挿入した粒子にかかる力を返す。

```
template<class TTreeForForce>
void PS::TreeForForce::copyLocalTreeStructure(const TTreeForForce & tff)
```

- **引数**

  tff: 入力。すでにローカルツリーを持っている相互作用ツリークラスのインスタンス。

- **返り値**

  なし。

- 機能

  tff の中にあるローカルツリーを自分にコピーする。

```
bool PS::TreeForForce::repeatLocalCalcForce()
```

- **引数**

  なし。

- **返り値**

  bool 型。

- 機能

  ユーザーが相互作用計算カーネル内で EssentialParticleI 型のサーチ半径を変更した後、もう一度ローカルだけで相互作用の計算が可能かどうかを判定する。可能な場合は true を返し、不可能な場合は false を返す。

5.8 通信クラス (PS::Comm)

5.8.1 概要

並列化手法 (MPI, OpenMP, SIMD) には様々な最適化パラメータが存在する。MPI 関連のパラメータやコミュニケータはシングルトンパターンを使って管理する。

以下のように定義する。

ソースコード 11: Comm の定義

```
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(const bool
               local);
9          static bool synchronizeConditionalBranchOR(const bool
               local);
10         template<class T>
11         static T getMinValue(const T val);
12         template<class T>
13         static T getMaxValue(const T val);
14         template<class Tfloat, class Tint>
15         static void getMinValue(const Tfloat f_in, const Tint
               i_in, Tfloat & f_out, Tint & i_out);
```

```
16          template<class Tfloat, class Tint>
17          static void getMaxValue(const Tfloat f_in, const Tint
                i_in, Tfloat & f_out, Tint & i_out);
18          template<class T>
19          static T getSum(const T val);
20  }
```

5.8.2 API

```
static PS::S32 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。

```
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。

```
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。

```
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。

```
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 AND を取り、結果を返す。

```
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 OR を取り、結果を返す。

```
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 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::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。

```
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。

```
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。

6 使用例

6.1 N 体シミュレーション

本節では、PSL の使用例を記述する。サンプルコードには省略したものだけを示した。

N 体シミュレーションの時間積分のループは以下の手順で実行される。ここで時間積分法はリープフロッグ法を使用する。

1. m_i, $\boldsymbol{x}_i^{(0)}$, $\boldsymbol{v}_i^{(0)}$, $\boldsymbol{a}_i^{(0)}$ は既知。上付き添字 “(0)” は時刻を表す。
2. $\boldsymbol{x}_i^{(1)}$ を以下の式で導出し、

$$\boldsymbol{x}_i^{(1)} = \boldsymbol{x}_i^{(0)} + \Delta t \boldsymbol{v}_i^{(0)} + 0.5\Delta t^2 \boldsymbol{a}_i^{(0)} \tag{2}$$

   $\boldsymbol{v}_i^{(1/2)}$ を以下の式で導出 ($X = \boldsymbol{v}$)。

$$X_i^{(1/2)} = X_i^{(0)} + 0.5\Delta t \dot{X}_i^{(0)} \tag{3}$$

3. m_i と $\boldsymbol{x}_i^{(1)}$ を用いて $\boldsymbol{a}_i^{(1)}$ を導出。
4. 以下の式を用いて $\boldsymbol{v}_i^{(1)}$ を導出 ($X = \boldsymbol{v}$)。

$$X_i^{(1)} = X_i^{(1/2)} + 0.5\Delta t \dot{X}_i^{(1)} \tag{4}$$

5. 計算領域の分割と粒子の分配を実行。
6. ステップ 1 に戻る。

以上の手順を PSL を用いて実装すると以下のようなコードとなる。

ソースコード 12: N 体シミュレーションのサンプルコード

```
1  #include<particle_simulator.h> // 必須
2
3  class ForceGrav{
4  public:
5      PS::F64vec acc;
6      PS::F64 pot;
7      void clear(){
8          acc = 0.0;
9          pot = 0.0;
10     }
11 };
12
13 class FPGrav{
14 public:
```

```
15      PS::S64 id;
16      PS::F64 mass;
17      PS::F64vec pos;
18      PS::F64vec vel;
19      PS::F64vec acc;
20      PS::F64 pot;
21      PS::F64vec getPos() const { return pos; }
22      void copyFromForce(const ForceGrav & force){
23          acc = force.acc;
24          pot = force.pot;
25      }
26  };
27
28  class EPIGrav{
29  public:
30      PS::S64 id;
31      PS::F64vec pos;
32      static PS::F64 eps;
33      PS::F64vec getPos() const {pos;}
34      void copyFromFP(const FPGrav & fp){
35          pos = fp.pos;
36          id = fp.id;
37      }
38  };
39
40  PS::F64 EPIGrav::eps = 1.0/32.0;
41
42  class EPJGrav{
43  public:
44      PS::S64 id;
45      PS::F64 mass;
46      PS::F64vec pos;
47      void copyFromFP(const FPGrav & fp){
48          mass = fp.mass;
49          pos = fp.pos;
50          id = fp.id;
51      }
52      PS::F64vec getPos() const { return pos; }
53      PS::F64 getCharge() const { return mass; }
54  };
```

```
55
56 struct CalcForceEpEp{
57     void operator () (const EPIGrav * ep_i,
58                       const PS::S32 n_ip,
59                       const EPJGrav * ep_j,
60                       const PS::S32 n_jp,
61                       ForceGrav * force){
62         PS::F64 eps2 = EPIGrav::eps * EPIGrav::eps;
63         for(PS::S32 i=0; i<n_ip; i++){
64             PS::F64vec xi = ep_i[i].pos;
65             PS::F64vec ai = 0.0;
66             PS::F64 poti = 0.0;
67             PS::S64 idi = ep_i[i].id;
68             for(PS::S32 j=0; j<n_jp; j++){
69                 if( idi == ep_j[j].id ) continue;
70                 PS::F64vec rij = xi - ep_j[j].pos;
71                 PS::F64 r3_inv = rij * rij + eps2;
72                 PS::F64 r_inv = 1.0/sqrt(r3_inv);
73                 r3_inv = r_inv * r_inv;
74                 r_inv *= ep_j[j].mass;
75                 r3_inv *= r_inv;
76                 ai -= r3_inv * rij;
77                 poti -= r_inv;
78             }
79             force[i].acc += ai;
80             force[i].pot += poti;
81         }
82     }
83 };
84
85 struct CalcForceSpEp{
86     void operator () (const EPIGrav * ep_i,
87                       const PS::S32 n_ip,
88                       const PS::SPJMonoPole * sp_j,
89                       const PS::S32 n_jp,
90                       ForceGrav * force){
91         PS::F64 eps2 = EPIGrav::eps * EPIGrav::eps;
92         for(PS::S32 i=0; i<n_ip; i++){
93             PS::F64vec xi = ep_i[i].pos;
94             PS::F64vec ai = 0.0;
```

```
95              PS::F64 poti = 0.0;
96              for(PS::S32 j=0; j<n_jp; j++){
97                  PS::F64vec rij = xi - sp_j[j].pos;
98                  PS::F64 r3_inv = rij * rij + eps2;
99                  PS::F64 r_inv = 1.0/sqrt(r3_inv);
100                 r3_inv = r_inv * r_inv;
101                 r_inv *= sp_j[j].mass;
102                 r3_inv *= r_inv;
103                 ai -= r3_inv * rij;
104                 poti -= r_inv;
105             }
106             force[i].acc += ai;
107             force[i].pot += poti;
108         }
109     }
110 };
111
112 template<class Tpsys>
113 void Kick(Tpsys & system,
114           const PS::F64 dt){
115     PS::S32 n = system.getNumberOfParticleLocal();
116     for(int i=0; i<n; i++){
117         system[i].vel  += system[i].acc * dt;
118     }
119 }
120
121 template<class Tpsys>
122 void Drift(Tpsys & system,
123            const PS::F64 dt){
124     PS::S32 n = system.getNumberOfParticleLocal();
125     for(int i=0; i<n; i++){
126         system[i].pos  += system[i].vel * dt;
127     }
128 }
129
130 int main(int argc, char *argv[]){
131     std::cout<<std::setprecision(15);
132     std::cerr<<std::setprecision(15);
133     PS::Initialize(argc, argv);
134
```

```
135     char sinput[1024];
136     int c;
137     while((c=getopt(argc,argv,"i:h")) != -1){
138         switch(c){
139         case 'i':
140             sprintf(sinput,optarg);
141             break;
142         case 'h':
143             std::cerr<<"i: input file name (nemo ascii)"<<std::
                    endl;
144             return 0;
145         }
146     }
147
148     const PS::F32 dt = 1.0/128.0;
149     const PS::F32 time_end = 10.0;
150     PS::ParticleSystem<FPGrav> system_grav;
151     system_grav.initialize();
152     PS::S32 n_grav_glb, n_grav_loc;
153     PS::F32 time_sys;
154     ReadNemoAscii(system_grav, n_grav_glb, n_grav_loc, time_sys
          , sinput);
155
156     PS::F32 coef_ema = 0.7;
157     PS::DomainInfo dinfo;
158     dinfo.initialize(coef_ema);
159     dinfo.decomposeDomainAll();
160
161     system_grav.exchangeParticle(dinfo);
162     n_grav_loc = system_grav.getNumberOfParticleLocal();
163
164     PS::TreeType<ForceGrav, EPIGrav, EPJGrav>::TreeForMonoPole
          tree_grav;
165
166     PS::F32 theta = 0.5;
167     tree_grav.initialize(n_grav_glb, theta);
168
169     tree_grav.calcForceAllAndWriteBack(CalcForceEpEp(),
          CalcForceSpEp(), system_grav, dinfo);
170
```

```
171     Kick(system_grav, dt*0.5);
172
173     while(time_sys < time_end){
174         time_sys += dt;
175         Drift(system_grav, dt);
176         if( fmod(time_sys, 1.0/32.0) == 0.0){
177             dinfo.decomposeDomainAll();
178         }
179         system_grav.exchangeParticle(dinfo);
180         tree_grav.calcForceAllAndWriteBack(CalcForceEpEp(),
                CalcForceSpEp(), system_grav, dinfo);
181         Kick(system_grav, dt);
182     }
183
184     PS::Finalize();
185     return 0;
186 }
```

6.2 stdSPH シミュレーション

stdSPH シミュレーションは以下の手順で実行される。時間積分法は先の N 体シミュレーションと同様にリープフロッグ法である。

1. m_i, $\boldsymbol{x}_i^{(0)}$, $\boldsymbol{v}_i^{(0)}$, $\boldsymbol{a}_i^{(0)}$, $u_i^{(0)}$, $\dot{u}_i^{(0)}$, $\alpha_i^{(0)}$, $\dot{\alpha}_i^{(0)}$, $h_i^{(0)}$ は既知。u は内部エネルギー、α は人工粘性に用いる変数である。

2. $\boldsymbol{x}_i^{(1)}$ を式 (**??**) で導出、$\boldsymbol{v}_i^{(1/2)}$, $u_i^{(1/2)}$ $\alpha_i^{(1/2)}$ を式 (**??**) で導出、$h_i^{(\mathrm{p})} = h_i^{(0)}$ とし、$\boldsymbol{v}_i^{(\mathrm{p})}$, $u_i^{(\mathrm{p})}$, $\alpha_i^{(\mathrm{p})}$ は以下のように導出 ($X = \boldsymbol{v}$, u, α)。ここで上付き添字 (p) は予測値であることを示す。

$$X_i^{(\mathrm{p})} = X_i^{(0)} + \Delta t \dot{X}_i^{(0)} \tag{5}$$

3. 以下のループにより $\rho_i^{(1)}$, $h_i^{(1)}$, $(\partial\rho_i/\partial h_i)^{(1)}$, $(\nabla\cdot\boldsymbol{v}_i)^{(1)}$, $(\nabla\times\boldsymbol{v}_i)^{(1)}$ を導出。

   ??.1. 以下の式を用いて $\rho_i^{(1)}$, $(\partial\rho_i/\partial h_i)^{(1)}$, $(\nabla\cdot\boldsymbol{v}_i)^{(1)}$, $(\nabla\times\boldsymbol{v}_i)^{(1)}$ を導出 (上付き添字 "(1)" は省略。以下、式中は同様)。このとき近傍粒子数 $N_{\mathrm{n},i}$(j 粒子数と同義) もカ

ウント。

$$\rho_i = \sum_j m_j W(\boldsymbol{x}_{ij}, h_i^{(\mathrm{p})}) \tag{6}$$

$$\left(\frac{\partial \rho_i}{\partial h_i}\right) = \sum_j m_j \frac{\partial W}{\partial h_i}(\boldsymbol{x}_{ij}, h_i^{(\mathrm{p})}) \tag{7}$$

$$\rho_i \left(\nabla \cdot \boldsymbol{v}_i\right) = -\sum_j m_j \boldsymbol{v}_{ij}^{(\mathrm{p})} \cdot \nabla W(\boldsymbol{x}_{ij}, h_i^{(\mathrm{p})}) \tag{8}$$

$$\rho_i \left(\nabla \times \boldsymbol{v}_i\right) = -\sum_j m_j \boldsymbol{v}_{ij}^{(\mathrm{p})} \times \nabla W(\boldsymbol{x}_{ij}, h_i^{(\mathrm{p})}) \tag{9}$$

??.2. $\rho_i^{(1)}$, m_i, $h_i^{(\mathrm{p})}$, $N_{\mathrm{n},i}$ が拘束式を満たしたら、$h_i^{(1)} = h_i^{(\mathrm{p})}$ として次ステップへ。そうでなければ、$h_i^{(\mathrm{p})}$ を $\rho_i^{(1)}$ と m_i から導出してステップ 3.1 へ戻る。

4. 圧力 $p_i^{(1)}$ と音速 $c_{\mathrm{s},i}^{(1)}$ を $\rho_i^{(1)}$ と $u_i^{(\mathrm{p})}$ から導出。

5. $\boldsymbol{a}_i^{(1)}$ と $\dot{u}_i^{(1)}$ を以下の式を用いて導出。

$$\boldsymbol{a}_i = -\sum_j m_j \left\{ f_i^{\mathrm{grad}} \frac{p_i}{\rho_i^2} \nabla W(\boldsymbol{x}_{ij}, h_i) + f_j^{\mathrm{grad}} \frac{p_j}{\rho_j^2} \nabla W(\boldsymbol{x}_{ij}, h_j) + \Pi_{ij} \nabla W(\boldsymbol{x}_{ij}, h_{ij}) \right\} \tag{10}$$

$$\dot{u}_i = f_i^{\mathrm{grad}} \frac{p_i}{\rho_i^2} \sum_j m_j \boldsymbol{v}_{ij}^{(\mathrm{p})} \cdot \nabla W(\boldsymbol{x}_{ij}, h_i) + \frac{1}{2} \sum_j m_j \Pi_{ij} \boldsymbol{v}_{ij}^{(\mathrm{p})} \cdot \nabla W(\boldsymbol{x}_{ij}, h_{ij}) \tag{11}$$

ここで、

$$f_i^{\mathrm{grad}} = \left(1 + \frac{1}{3} \frac{h_i}{\rho_i} \frac{\partial \rho_i}{\partial h_i}\right)^{-1} \tag{12}$$

$$\Pi_{ij} = \begin{cases} F_i \dfrac{-\alpha_i c_{\mathrm{s},ij} \mu_{ij} + \beta_i \mu_{ij}^2}{\rho_{ij}} & (\boldsymbol{x}_{ij} \cdot \boldsymbol{v}_{ij} < 0) \\ 0 & (\text{otherwise}) \end{cases} \tag{13}$$

$$\tag{14}$$

さらに

$$\mu_{ij} = \frac{h_{ij} \boldsymbol{x}_{ij} \cdot \boldsymbol{v}_{ij}}{|\boldsymbol{r}_{ij}|^2 + \varepsilon h_{ij}^2} \tag{15}$$

$$F_i = \frac{\nabla \cdot \boldsymbol{v}_i}{|\nabla \cdot \boldsymbol{v}_i| + |\nabla \times \boldsymbol{v}_i| + \varepsilon_{\mathrm{b}} c_{\mathrm{s},i} / h_i} \tag{16}$$

$$\beta_i \propto \alpha_i \tag{17}$$

6. $\dot{\alpha}_i^{(1)}$ を以下の式から導出。

$$\dot{\alpha}_i = -\frac{\alpha_i - \alpha_{\min}}{\tau_i} + \max\left[-(\nabla \cdot \boldsymbol{v}_i)(\alpha_{\max} - \alpha_i), 0\right] \tag{18}$$

ここで

$$\tau_i = \frac{h_i}{\xi c_{\mathrm{s},i}} \tag{19}$$

7. $\boldsymbol{v}_i^{(1)}$, $u_i^{(1)}$, $\alpha_i^{(1)}$ を式 (??) から導出。

8. 計算領域の分割と粒子の分配を実行。

9. ステップ 1 に戻る。

以下のサンプルはステップ 3 の密度とカーネル半径を求めるコードである。拘束条件はカーネル半径内の粒子数を一定とした。

ソースコード 13: stdSPH シミュレーションのサンプルコード

```
1  #include<particle_simulator.h> // 必須
2
3  class ResultDens{
4  public:
5      void clear(){ //絶対必要。名前固定。
6          density = 0.0;
7          r_search_next = 0.0;
8          n_neighbour = 0;
9          kernel_length = 0.0;
10     }
11     PS::F64 density;
12     PS::F64 r_search_next;
13     PS::S32 n_neighbour;
14     PS::F64 kernel_length;
15 };
16
17 class FullPtcl{
18 public:
19     PS::F64 getCharge() const { //絶対必要。名前固定。
20         return this->mass;
21     }
22     PS::F64vec getPos() const { //絶対必要。名前固定。
23         return this->pos;
24     }
25     PS::F64 getRSearch() const { //絶対必要。名前固定。
26         return this->r_search;
27     }
28     void copyFromForce(const ResultDens & _dens){//絶対必要。名前
           固定。
```

```
29          this->density = _dens.density;
30          this->kernel_length = _dens.kernel_length;
31      }
32
33      //以下名前含めユーザー適宜。
34      PS::S64 id;
35      PS::F64 mass;
36      PS::F64 kernel_length;
37      PS::F64 r_search;
38      PS::F64 density;
39      PS::F64vec pos;
40      PS::F64vec vel;
41      PS::S32 loadOneParticle(FILE * fp) {
42          PS::S32 ret = 0;
43          ret = fscanf(fp, "%ld%lf%lf%lf%lf%lf%lf%lf%lf",
44                       &this->id, &this->mass, &this->
                                kernel_length,
45                       &this->pos.x, &this->pos.y, &this->pos.z,
46                       &this->vel.x, &this->vel.y, &this->vel.z);
47          return ret;
48      }
49      void dumpOneParticle(FILE * fp){
50          fprintf(fp, "%lf␣␣␣%lf␣␣␣%lf␣␣␣%lf␣␣␣%lf␣␣␣%lf␣␣␣%lf␣␣␣
                \n",
51                     this->mass,
52                     this->pos.x, this->pos.y, this->pos.z,
53                     this->vel.x, this->vel.y, this->vel.z);
54      }
55  };
56
57  class EPIDens{
58  public:
59      enum{
60          n_neighbour_crit = 50,
61      };
62      PS::F64vec pos;
63      PS::F64 r_search;
64
65      void copyFromFP(const FullPtcl & rp){
66          this->r_search = rp.getRSearch();
```

```
67          this->pos = rp.getPos();
68      }
69  };
70
71  class EPJDens{
72  public:
73      PS::F64 mass;
74      PS::F64vec pos;
75      void copyFromFP(const FullPtcl & rp){
76          this->mass = rp.getCharge();
77          this->pos = rp.getPos();
78      }
79  };
80
81  PS::F64 CubicSpline(const PS::F64 r_sq,
82                      const PS::F64 h_inv){
83      PS::F64 xi = sqrt(r_sq)*h_inv;
84      PS::F64 xi10 = (1.0-xi > 0.0) ? 1.0-xi : 0.0;
85      PS::F64 xi05 = (0.5-xi > 0.0) ? 0.5-xi : 0.0;
86      return xi10*xi10*xi10 - 4.0*xi05*xi05*xi05;
87  }
88
89  void CalcDensityEpEp(const EPIDens * ep_i,
90                       const PS::S32   n_ip,
91                       const EPJDens * ep_j,
92                       const PS::S32   n_jp,
93                       ResultDens * dens)
94  {
95      std::vector < std::pair < PS::F64, PS::F64 > >
            r_neighbour_sq_with_mass;
96      r_neighbour_sq_with_mass.reserve(EPIDens::n_neighbour_crit
            *3+100);
97      for(PS::S32 i=0; i<n_ip; i++){
98          if (ep_i[i].r_search == 0.0) continue;
99          r_neighbour_sq_with_mass.clear();
100         const PS::F64 rsearch2 = ep_i[i].r_search*ep_i[i].
                r_search;
101         for(PS::S32 j=0; j<n_ip; j++){
102             const PS::F64vec dr = ep_i[i].pos - ep_j[j].pos;
103             const PS::F64 r_sq = dr * dr;
```

```
104             const PS::F64 mj = ep_j[j].mass;
105             if( r_sq < rsearch2 ){
106                 r_neighbour_sq_with_mass.push_back( std::
                        make_pair(r_sq, mj) );
107             }
108         }
109         const size_t ncrit = EPIDens::n_neighbour_crit;
110         if( r_neighbour_sq_with_mass.size() <=  ncrit ){
111             // search radius was too small !!
112             dens[i].r_search_next = ep_i[i].r_search * cbrt
                    (2.0);
113             continue;
114         }
115         else{
116             dens[i].r_search_next = 0.0; // DON'T need to
                    search next
117 
118             std::sort(r_neighbour_sq_with_mass.begin(),
                    r_neighbour_sq_with_mass.end());
119             dens[i].kernel_length = sqrt(
                    r_neighbour_sq_with_mass[ncrit-1].first );
120             PS::F64 tmp_dens = 0.0;
121             PS::F64 h_inv = 1.0/dens[i].kernel_length;
122             PS::F64 Cnorm = (16.0/M_PI) * (h_inv * h_inv *
                    h_inv);
123             for(PS::S32 k=0; k<PS::S32(ncrit); k++){
124                 tmp_dens += r_neighbour_sq_with_mass[k].second
                        * CubicSpline(r_neighbour_sq_with_mass[k
                        ].first, h_inv);
125             }
126             dens[i].density = Cnorm * tmp_dens;
127         }
128     }
129 }
130 
131 int main(int argc, char *argv[]){
132 
133     PS::Initialize(argc, argv); //初期化
134 
```

```
135     PS::TreeType<ResultDens, EPIDens, EPJDens>::
            TreeForGatherSearch tree_dens; // for density
136
137     PS::ParticleSystem<FullPtcl> sph_system; //粒子種の数だけ生成
138
139     sph_system.loadParticleSingle(argv[1], "r", &FullPtcl::
            loadOneParticle); //単一ファイ
            ル用
140
141     PS::DomainInfo dinfo;
142     dinfo.initialize("domain_info.para");
143     tree_dens.initialize(sph_system.getNumberOfParticleTotal
            ()); //引数は配列の領域確保に
            使う。
144
145     dinfo.decomposeDomainAll(sph_system);
146
147     PS::S32 nloc = sph_system.getNumberOfParticleLocal();
148     //探査する半径を膨らます。
149     for(PS::S32 i=0; i<nloc; i++){
150         sph_system[i].r_search = sph_system[i].kernel_length;
151     }
152     for(bool repeat = true; repeat; ){
153         tree_dens.calcForceAll(CalcDensityEpEp, dinfo,
                sph_system);
154         PS::S32 nploc = tree_dens.getNumberOfParticleLocal();
155         repeat = false;
156
157         for(PS::S32 i=0; i<nploc; i++){
158             ResultDens dens_tmp = tree_dens.getForce(i);
159             sph_system[i].r_search    = dens_tmp.r_search_next;
160             sph_system[i].density     = dens_tmp.density;
161             sph_system[i].kernel_length = dens_tmp.
                    kernel_length;
162             if(dens_tmp.r_search_next > 0.0){
163                 repeat = true;
164             }
165         }
166         repeat = PS::Comm::synchronizeConditionalBranch(repeat
                );
167     }
```

```
168     PS::Finalize();
169     return 0;
170 }
```

相互作用クラスは、tree_dens であり、ツリークラスのタイプは TreeForSearch を使っている。for 文の内側はカーネル半径 (kernel_length) を求めるためのループである。LET 交換時に、推定されるカーネル半径よりやや大きめの値 (r_search) を使って LET をとってくることで、通信が頻繁に起こるのを防いでいる。もし、(r_search) より内側にいる粒子数が拘束条件を下回ったら、再び通信からやり直す。この際拘束条件を満たした粒子は r_search に 0 を代入することで、無駄な通信を減らす。

6.3 複数種類のシミュレーション

N 体＋ SPH シミュレーションを行う場合 (sph は密度のみ求めてる)、以下のように実装される。

ソースコード 14: N 体＋ SPH シミュレーションのサンプルコード

```
 1 #include<particle_simulator.h> // 必須
 2
 3 // Gravity //
 4 class ResultForce{ //名前自由。
 5 public:
 6     void clear(){ //絶対必要。名前固定。
 7         acc = 0.0;
 8         pot = 0.0;
 9     }
10
11     //以下名前含めユーザー定義。
12     PS::F64vec acc;
13     PS::F64 pot;
14 };
15
16 class GravFP{ //絶対必要。名前自由。
17 public:
18     PS::F64vec getPos() const { //絶対必要。名前固定。
19         return this->pos;
20     }
21     PS::F64 getCharge() const { //絶対必要。名前固定。
22         return this->mass;
23     }
```

```
24      void copyFromForce(const ResultForce & _force){//絶対必要。名
            前固定。
25          this->acc = _force.acc;
26          this->pot = _force.pot;
27      }
28
29      //以下名前含めユーザー定義。
30      PS::S64 id;
31      PS::F64 mass;
32      PS::F64vec pos;
33      PS::F64vec vel;
34      PS::F64vec acc;
35      PS::F64 pot;
36      PS::S32 loadOneParticle(FILE * fp) {
37          PS::S32 ret = 0;
38          ret = fscanf(fp, "%lf%lf%lf%lf%lf%lf%lf",
39                       &this->mass,
40                       &this->pos[0], &this->pos[1], &this->pos
                             [2],
41                       &this->vel[0], &this->vel[1], &this->vel
                             [2]);
42          std::cout<<"this->mass"<<this->mass<<std::endl;
43          return ret;
44      }
45      void dumpOneParticle(FILE * fp){
46          fprintf(fp, "%lf␣␣␣%lf␣␣␣%lf␣␣␣%lf␣␣␣%lf␣␣␣%lf␣␣␣%lf␣␣␣
                 \n",
47                     this->mass,
48                     this->pos[0], this->pos[1], this->pos[2],
49                     this->vel[0], this->vel[1], this->vel[2]);
50      }
51  };
52
53  class GravEPI{
54  public:
55      void copyFromFP(const GravFP & rp){ //絶対必要。名前固定。
56          pos = rp.pos;
57          id = rp.id;
58      }
59
60      //以下名前含めユーザー定義。
```

```
61      PS::F64vec pos;
62      PS::S64 id;
63  };
64
65  class GravEPJ{
66  public:
67      void copyFromFP(const GravFP & rp){ //絶対必要。名前固定。
68          mass = rp.mass;
69          pos = rp.pos;
70          id = rp.id;
71      }
72
73      //以下名前含めユーザー定義。
74      PS::S64 id;
75      PS::F64 mass;
76      PS::F64vec pos;
77  };
78
79  //相互作用関数
80  void CalcGravEpEp(const GravEPI * ep_i,
81                    const PS::S32 n_ip,
82                    const GravEPJ * ep_j,
83                    const PS::S32 n_jp,
84                    ResultForce * force){
85      for(PS::S32 i=0; i<n_ip; i++){
86          PS::F64vec xi = ep_i[i].pos;
87          PS::F64vec ai = 0.0;
88          PS::F64 poti = 0.0;
89          PS::S64 idi = ep_i[i].id;
90          for(PS::S32 j=0; j<n_jp; j++){
91              if( idi == ep_j[j].id ) continue;
92              PS::F64vec rij = xi - ep_j[j].pos;
93              PS::F64 r3_inv = rij * rij;
94              PS::F64 r_inv = 1.0/sqrt(r3_inv);
95              r3_inv = r_inv * r_inv;
96              r_inv *= ep_j[j].mass;
97              r3_inv *= r_inv;
98              ai -= r3_inv * rij;
99              poti -= r_inv;
100         }
```

```
101         force[i].acc = ai;
102         force[i].pot = poti;
103     }
104 }
105
106 // SPH //
107 class ResultDens{
108 public:
109     void clear(){ //絶対必要。名前固定。
110         density = 0.0;
111         r_search_next = 0.0;
112         n_neighbour = 0;
113         kernel_length = 0.0;
114     }
115     PS::F64 density;
116     PS::F64 r_search_next;
117     PS::S32 n_neighbour;
118     PS::F64 kernel_length;
119 };
120
121 class DensFP{
122 public:
123     PS::F64 getCharge() const { //絶対必要。名前固定。
124         return this->mass;
125     }
126     PS::F64vec getPos() const { //絶対必要。名前固定。
127         return this->pos;
128     }
129     PS::F64 getRSearch() const { //絶対必要。名前固定。
130         return this->r_search;
131     }
132     void copyFromForce(const ResultDens & _dens){//絶対必要。名前
          固定。
133         this->density = _dens.density;
134         this->kernel_length = _dens.kernel_length;
135     }
136
137     //以下名前含めユーザー定義。
138     PS::S64 id;
139     PS::F64 mass;
140     PS::F64 kernel_length;
```

```
141     PS::F64 r_search;
142     PS::F64 density;
143     PS::F64vec pos;
144     PS::F64vec vel;
145     PS::S32 loadOneParticle(FILE * fp) {
146         PS::S32 ret = 0;
147         ret = fscanf(fp, "%ld%lf%lf%lf%lf%lf%lf%lf%lf",
148                      &this->id, &this->mass, &this->
                                kernel_length,
149                      &this->pos.x, &this->pos.y, &this->pos.z,
150                      &this->vel.x, &this->vel.y, &this->vel.z);
151         return ret;
152     }
153     void dumpOneParticle(FILE * fp){
154         fprintf(fp, "%lf␣␣␣%lf␣␣␣%lf␣␣␣%lf␣␣␣%lf␣␣␣%lf␣␣␣%lf␣␣␣
                \n",
155                    this->mass,
156                    this->pos.x, this->pos.y, this->pos.z,
157                    this->vel.x, this->vel.y, this->vel.z);
158     }
159 };
160
161 class EPIDens{
162 public:
163     void copyFromFP(const DensFP & rp){//絶対必要。名前固定。
164         this->r_search = rp.getRSearch();
165         this->pos = rp.getPos();
166     }
167     //以下名前含めユーザー定義。
168     enum{
169         n_neighbour_crit = 50,
170     };
171     PS::F64vec pos;
172     PS::F64 r_search;
173
174 };
175
176 class EPJDens{
177 public:
178     void copyFromFP(const GravFP & rp){//絶対必要。名前固定。
```

```
179         this->mass = rp.getCharge();
180         this->pos = rp.getPos();
181     }
182     //以下名前含めユーザー定義。
183     PS::F64 mass;
184     PS::F64vec pos;
185
186 };
187
188 // 相互作用関数（密度）
189 PS::F64 CubicSpline(const PS::F64 r_sq,
190                     const PS::F64 h_inv){
191     PS::F64 xi = sqrt(r_sq)*h_inv;
192     PS::F64 xi10 = (1.0-xi > 0.0) ? 1.0-xi : 0.0;
193     PS::F64 xi05 = (0.5-xi > 0.0) ? 0.5-xi : 0.0;
194     return xi10*xi10*xi10 - 4.0*xi05*xi05*xi05;
195 }
196
197 void CalcDensityEpEp(const EPIDens * ep_i,
198                      const PS::S32   n_ip,
199                      const EPJDens * ep_j,
200                      const PS::S32   n_jp,
201                      ResultDens * dens)
202 {
203     std::vector < std::pair < PS::F64, PS::F64 > >
          r_neighbour_sq_with_mass;
204     r_neighbour_sq_with_mass.reserve(EPIDens::n_neighbour_crit
          *3+100);
205     for(PS::S32 i=0; i<n_ip; i++){
206         if (ep_i[i].r_search == 0.0) continue;
207         r_neighbour_sq_with_mass.clear();
208         const PS::F64 rsearch2 = ep_i[i].r_search*ep_i[i].
              r_search;
209         for(PS::S32 j=0; j<n_ip; j++){
210             const PS::F64vec dr = ep_i[i].pos - ep_j[j].pos;
211             const PS::F64 r_sq = dr * dr;
212             const PS::F64 mj = ep_j[j].mass;
213             if( r_sq < rsearch2 ){
214                 r_neighbour_sq_with_mass.push_back( std::
                        make_pair(r_sq, mj) );
```

```
215             }
216         }
217         const size_t ncrit = EPIDens::n_neighbour_crit;
218         if( r_neighbour_sq_with_mass.size() <=  ncrit ){
219             // search radius was too small !!
220             dens[i].r_search_next = ep_i[i].r_search * cbrt
                    (2.0);
221             continue;
222         }
223         else{
224             dens[i].r_search_next = 0.0; // DON'T need to
                    search next
225
226             std::sort(r_neighbour_sq_with_mass.begin(),
                    r_neighbour_sq_with_mass.end());
227             dens[i].kernel_length = sqrt(
                    r_neighbour_sq_with_mass[ncrit-1].first );
228             PS::F64 tmp_dens = 0.0;
229             PS::F64 h_inv = 1.0/dens[i].kernel_length;
230             PS::F64 Cnorm = (16.0/M_PI) * (h_inv * h_inv *
                    h_inv);
231             for(PS::S32 k=0; k<PS::S32(ncrit); k++){
232                 tmp_dens += r_neighbour_sq_with_mass[k].second
                        * CubicSpline(r_neighbour_sq_with_mass[k
                        ].first, h_inv);
233             }
234             dens[i].density = Cnorm * tmp_dens;
235         }
236     }
237 }
238
239 int main(int argc, char *argv[]){
240
241     PS::Initialize(argc, argv); //初期化
242
243     //領域分割クラスは1つ。
244     PS::DomainInfo dinfo;
245     dinfo.initialize("domain_info.para");
246
247     // 粒子群クラス2種類生成
```

```
248     PS::ParticleSystem<GravFP> nbody_system;
249     PS::ParticleSystem<DensFP> sph_system;
250     //ファイル読み込み。
251     nbody_system.loadParticleSingle(argv[1], "r", &GravFP::
            loadOneParticle);
252     sph_system.loadParticleSingle(argv[2], "r", &DensFP::
            loadOneParticle);
253
254     // 相互作用クラス2種類生成
255     PS::TreeType<ResultDens, EPIDens, EPJDens>::
            TreeForGahterSearch tree_dens; // 流体（密度計算）用
256     PS::TreeType<ResultForce, GravEPI, GravEPJ>::
            TreeForMonoBaryCenter grav_tree; // 重力用
257     // 領域確保
258     tree_dens.initialize(sph_system.getNumberOfParticleTotal
            ());
259     grav_tree.initialize( nbody_system.getNumberOfParticleTotal
            () + sph_system.getNumberOfParticleTotal() );
260
261     //複数種あるので、decomposeDomainAllは呼べない。
262     dinfo.sampleParticle(nbody_system, true);
263     dinfo.sampleParticle(sph_system, false);
264     dinfo.decomposeDomain();
265
266     //重力計算。
267     grav_tree.calcForce(CalcGravEpEp, dinfo, nbody_system);
268
269     //以下、密度計算。
270     //探査する半径を膨らます。
271     PS::S32 nloc = sph_system.getNumberOfParticleLocal();
272     for(PS::S32 i=0; i<nloc; i++){
273         sph_system[i].r_search = sph_system[i].kernel_length
                *1.2;
274     }
275     for(bool repeat = true; repeat; ){
276         tree_dens.calcForceAll(CalcDensityEpEp, dinfo,
                sph_system);
277         PS::S32 nploc = tree_dens.getNumberOfParticleLocal();
278         repeat = false;
279         for(PS::S32 i=0; i<nploc; i++){
```

```
280             ResultDens dens_tmp = tree_dens.getForce(i);
281             sph_system[i].r_search    = dens_tmp.r_search_next;
282             sph_system[i].density     = dens_tmp.density;
283             sph_system[i].kernel_length = dens_tmp.
                    kernel_length;
284             if(dens_tmp.r_search_next > 0.0){
285                 repeat = true;
286             }
287         }
288         repeat = PS::Comm::synchronizeConditionalBranch(repeat
                );
289     }
290
291     PS::Finalize();
292     return 0;
293 }
```

7 ロードマップ

6 月末に ver0 シリアルバージョン完成予定。12 月末に mpi バージョン完成予定。

A ユーザー定義粒子クラス、Force クラス

以下では、相互作用計算や通信を効率的に行う為にユーザーが定義出来る粒子クラスおよび、結果を格納するための Force クラスについて説明する。これらのクラスは名前空間 PS の下にある必要はない。

A.1 FullParticle 型

シミュレーションを行う上で必要な粒子が持つべき情報が入っているクラスで絶対に必要。ParticleSystem がこのクラスの配列を持つ。ノード間での粒子交換はこのクラスが交換される。メンバ変数は private でも public でもよい。TreeParticle はこのクラスから生成される。TreeParticle は FullParticle から必要な情報をもらう為、FullParticle は決まった名前のアクセサを持つ必要がある（メンバ変数名は自由）。

- getPos()　相互作用の形によらず絶対必要。
- getCharge()　長距離力の計算をする場合。
- getRSearch()　cutoff 付長距離力、短距離力の計算をする場合。

- getEps()　対称化ソフトニングを用いる場合。

また、TreeForForceが直接Forceの値をFullParticleに書き込めるようにcopyFromForce()というメソッドも必要。

A.2 EssentialParticleI型

粒子-粒子間相互作用を計算するのに必要なI粒子が持っている情報を持つクラス。TreeForForceがこのクラスの配列を持つ。相互作用に必要なFullParticleのメンバをコピーするので、copyFromFPという名前のメンバ関数が必要。

A.3 EssentialParticleJ型

粒子-粒子間相互作用を計算するのに必要なJ粒子が持っている情報を持つクラス。TreeForForceがこのクラスの配列を持つ。LET交換で通信される。相互作用に必要なFullParticleのメンバをコピーするので、copyFromFP という名前のメンバ関数が必要。

A.4 SuperParticleJ型

セル-粒子間相互作用を計算するのに必要なセルの情報を持ったクラス。TreeForForce型がこのクラスの配列を持つ。短距離力の場合はsizeこのクラスは必要ない。長距離力の場合は、chargeとposが必要である。さらに双極子、四重極子を計算する場合はそれぞれdiとquadを加えることができる。対称化されたソフトニングを用いる場合はepsを加えることもできるが、その場合はFullParticle型のメンバ変数にepsが必要となる。あらかじめ、以下の場合については最初から用意されている。

- 重心周りで展開した単極子からの力の計算。
- 重心周りで展開した四重極子までの力の計算。
- ソフトニングが対称化された重心周りで展開した単極子からの力の計算
- 幾何中心周りで展開した単極子からの力の計算。
- 幾何中心周りで展開した双極子までの力の計算。
- 幾何中心周りで展開した四重極子までの力の計算。

A.5 Force型

相互作用の結果を格納するクラス。TreeForForceがこのクラスの配列を持つ。TreeForForceが値の初期化等をしなければならないので、初期化をするメンバ関数clear()が必要。

B Treeを構築しているサブクラス。

B.1 TreeParticle

TreeForForceが持つクラスで、ユーザーからは見えない。FullParticleから粒子情報をもらいツリーの構築、モーメントの計算等に用いる。これらのクラスはFullParticleにアクセスする必要があり、FullParticleのユーザ定義メソッド(FullParticle::getPos()等)を用いてアクセスする。

ソースコード 15: TreeParticle

```
1  namespace ParticleSimulator{
2      template<class Tprop>
3      class TreeParticle{
4      public:
5          U64 key_;
6          S32 next_adr_tp_;
7          Tprop prop_;
8          template<class Trp>
9          void setFromFP(const Trp & rp){
10             key_ = MortonKey<DIMENSION>::getKey( rp.getPos() );
11             prop_.setFromFP(rp);
12         }
13     };
14 }
```

メンバprop_はツリーセルのモーメントや情報等を作る為に使われる。これはユーザーがツリークラスのオブジェクトを作るときに、自動的に適したクラスが採用される。

以下は重力用のクラスである。

ソースコード 16: PropertyLong

```
1  namespace ParticleSimulator{
2      class PropertyLong{
3      public:
4          F32 charge_;
5          F32vec pos_;
6      };
7  }
```

B.2 TreeCell

TreeForForceが持つセルクラスで、ユーザーからは見えない。Tree Particleから粒子情報をもらいセルのモーメントの計算を行い値を格納する。

以下に短距離力用の TreeCell を記述する。

ソースコード 17: TreeCell

```
1  namespace ParticleSimulator{
2      template<class Tmom>
3      class TreeCell{
4      public:
5          S32 n_ptcl_;
6          S32 adr_tc_;
7          S32 adr_tp_;
8          S32 lev_ni_;
9          Tmom mom_;
10     };
11 }
```

メンバ mom_はツリーセルのモーメント等の情報を保持するモーメントクラス。ユーザーがツリークラスのオブジェクトを作るときに、自動的に適したモーメントクラスが採用される。以下は短距離力用のモーメントクラスである。

ソースコード 18: MomentSearch

```
1  namespace ParticleSimulator{
2      class MomentSearch{
3      public:
4          F32ort vertex_in_;
5          F32ort vertex_out_;
6          template<class Tprop> void accumulateAtLeaf(const Tprop
               & _prop);
7          void accumulate(const MomentSearch & _mom);
8      };
9  }
```

ここで、PS::F32ort が出てくるが、これは PS::Orthotope クラスといわれるもので、内部に二つの PS::F32vec 型を持ち、それぞれが位置座標の小さい頂点と大きい頂点の位置座標を持つ。

C　相互作用関数の定義方法

ユーザーは任意の相互作用関数を書くことができる。関数名は任意であるが、引数は第一引数から順に、I 粒子へのポインタ、I 粒子の個数、J 粒子へのポインタ、J 粒子の個数、force へのポインタとなる。粒子の型と力の型は任意であるが、相互作用関数内で定義されている型と対応していなければならない。以下は重力の場合の例である。

```
void CalcForceEpEp(const EssentialPtclI * ep_i,
                   const PS::S32 n_ip,
                   const EssentialPtclJ * ep_j,
                   const PS::S32 n_jp,
                   Force * force){
    for(PS::S32 i=0; i<n_ip; i++){
        PS::F64vec xi = ep_i[i].pos;
        PS::F64vec ai = 0.0;
        PS::F64 poti = 0.0;
        PS::S32 idi = ep_i[i].id;
        for(PS::S32 j=0; j<n_jp; j++){
            if(idi == ep_j[j].id) continue;
            PS::F64vec rij = xi - ep_j[j].pos;
            PS::F64 r3_inv = rij * rij;
            PS::F64 r_inv = 1.0/sqrt(r3_inv);
            r3_inv = r_inv * r_inv;
            r_inv *= ep_j[j].mass;
            r3_inv *= r_inv;
            ai -= r3_inv * rij;
            poti -= r_inv;
        }
        force[i].acc += ai;
        force[i].pot += poti;
    }
}
```

D 実装

D.1 粒子サンプリング

以下は重み付けを使った粒子のサンプル方法のコードである。重みにはプロセスの計算時間や相互作用数等が適切と考えられる。各プロセスで同じ粒子数にしたい場合は、第三引数が各プロセスで同じ値を入れる（デフォルト引数を使えばよい）。

```
    template<class Tpsys>
    void DomainInfo::sampleParticle(const Tpsys & psys, const
         bool clear=true, const PS::F32 wgh=1.0){
        static S32 n_sample_ = 0;
        if(clear==true){ n_sample_ = 0; }
        PS::F32 wgh_tot;
```

```
6           MPI::COMM_WORLD.Allreduce(&wgh, &wgh_tot, 1, MPI::FLOAT
                , MPI_SUM);
7           S32 n_sample_tmp = n_sample_max_ * (wgh / wgh_tot);
8           S32 interval = ( (this->getNumberOfParticleLocal()) -
                1) / n_sample_tmp;
9           interval = (interval <= 0) ? 1 : interval;
10          for(S32 i=0; i<psys.getNumberOfParticleLocal(); i +=
                interval){
11              pos_sample_[n_sample_] = psys[i].getPos();
12              n_sample_++;
13          }
14      }
```

D.2 ルートドメインの分割

D.2.1 ルートドメインの分割

D.2.2 ドメイン、ツリーセルの番号付け

各プロセスの持つドメインやツリー構造は2、もしくは3次元構造をしているが1次元の番号を持っている。この番号付けのルールはz->y->x(2次元の場合はy->x)の順で付ける。例えば、直方体をx,y,zにnx,ny,nz個に分割し、それぞれの箱に軸方向のID番号(idx,idy,idz)を小さい方から順につけていくとした場合。1次元のidは

$$id = idx \times ny \times nz + idy \times nz + idz. \tag{20}$$

となる。ツリーセルの番号付けもこれと同じである。

D.2.3 ルートドメインの分割の順番

ルートドメインの分割の方法はMakino(2004)の方法に従う。切り方はx方向から先に切り、y、zと切っていく。

D.2.4 実装例

以下にルートドメインの分割のサンプルコードを記述する。ここでの方法はサンプルした粒子をルートノードに集めて全てのドメインの座標をルートノードで計算するようになっているが、下にそれを回避する方法も記しておく。

サンプル粒子の少なさが原因となる境界のポアソンノイズを抑えるために、過去の境界を用いて移動平均を使う。ここでは記述の簡単さから指数移動平均を用いることとする(平滑化係数はalphaとする)。

ソースコード 19: ルートドメインの分割

```
1  void DomainInfo::decomposeDomain(){
2      PS::S32 n_proc = MPI::COMM_WORLD.Get_size();
3      std::vector<PS::S32> n_sample_array;
4      std::vector<PS::S32> n_sample_array_disp;
5      n_sample_array.reserve(n_proc);
6      n_sample_array_disp.reserve(n_proc+1);
7      MPI::COMM_WORLD.Gather(&n_sample_, 1, MPI::INT,
8                             &n_sample_array[0], 1, MPI::INT, 0);
9      n_sample_array[0] = 0;
10     for(PS::S32 i=0; i<n_proc; i++){
11         n_sample_array_disp[i+1] = n_sample_array_disp[i] +
               n_sample_array[i];
12     }
13     MPI::COMM_WORLD.Gatherv(pos_sample_, n_sample_, MPI_F32VEC,
14                             pos_sample_tot_, &n_sample_array
                                   [0], &n_sample_array_disp[0],
15                             MPI_F32VEC, 0);
16     PS::S32 rank = Comm::getRank();
17     PS::S32 np[PARTICLE_SIMULATOR_DIMENSION];
18     for(int k=0; k<PARTICLE_SIMULATOR_DIMENSION; k++){ np[k] =
           Comm::getNProcXD(k); }
19     if(rank == 0){
20         std::vector<PS::F32rec> domain_old;
21         domain_old.reserve(n_proc);
22         for(PS::S32 i=0; i<n_proc; i++) domain_old[i] =
               domain_rem_[i];
23         std::vector<PS::S32> istart, iend;
24         istart.reserve(n_proc);
25         iend.reserve(n_proc);
26         quickSort(pos_sample_tot_, 0, n_sample_-1, 0);
27         for(PS::S32 i=0; i<n_proc; i++){
28            istart[i] = (i*n_sample_)/n_proc;
29             if(i>0)
30                 iend[i-1] = istart[i] - 1;
31         }
32         iend[n_proc-1] = n_sample_-1;
33
34         for(PS::S32 ix=0; ix<np[0]; ix++){
35             PS::F32 xlow = 0.0;
```

```
36              PS::F32 xhigh = 0.0;
37              PS::S32 ix0 = ix*np[1]*np[2];
38              PS::S32 ix1 = (ix+1)*np[1]*np[2];
39              getBoxCoordinate(n_sample_, pos_sample_tot_, 0,
                     istart[ix0], iend[ix1-1], half_length_root_,
                     xlow, xhigh);
40              for(PS::S32 i=ix0; i<ix1; i++){
41                  domain_rem_[i].low_[0] = alpha_*xlow + (1.0-
                         alpha_)*domain_old[i].low_[0];
42                  domain_rem_[i].high_[0] = alpha_*xhigh + (1.0-
                         alpha_)*domain_old[i].high_[0];
43              }
44          }
45
46          for(PS::S32 ix=0; ix<np[0]; ix++){
47              PS::S32 ix0 = ix*np[1]*np[2];
48              PS::S32 ix1 = (ix+1)*np[1]*np[2];
49              PS::S32 nsample_y = iend[ix1-1] - istart[ix0] + 1;
50              quickSort(pos_sample_tot_, istart[ix0], iend[ix1
                     -1], 1);
51              for(PS::S32 iy=0; iy<np[1]; iy++){
52                  PS::F32 ylow = 0.0;
53                  PS::F32 yhigh = 0.0;
54                  PS::S32 iy0 = ix0+iy*np[2];
55                  PS::S32 iy1 = ix0+(iy+1)*np[2];
56                  getBoxCoordinate(nsample_y, pos_sample_tot_+
                         istart[ix0], 1, istart[iy0]-istart[ix0],
57                                      iend[iy1-1]-istart[ix0],
                                            half_length_root_, ylow,
                                             yhigh);
58                  for(int i=iy0; i<iy1; i++){
59                      domain_rem_[i].low_[1] = alpha_*ylow +
                             (1.0-alpha_)*domain_old[i].low_[1];
60                      domain_rem_[i].high_[1] = alpha_*yhigh +
                             (1.0-alpha_)*domain_old[i].high_[1];
61                  }
62              }
63          }
64  #if (PARTICLE_SIMULATOR_DIMENSION == 3)
65          for(PS::S32 ix=0; ix<np[0]; ix++){
```

```
66              PS::S32  ix0 = ix*np[1]*np[2];
67              for(PS::S32 iy=0; iy<np[1]; iy++){
68                  PS::S32 iy0 = ix0 + iy*np[2];
69                  PS::S32 iy1 = ix0 + (iy+1)*np[2];
70                  PS::S32 nsample_z = iend[iy1-1] - istart[iy0] +
                            1;
71                  quickSort(pos_sample_tot_, istart[iy0], iend[
                          iy1-1], 2);
72                  for(PS::S32 iz=0; iz<np[2]; iz++){
73                      PS::F32 zlow = 0.0;
74                      PS::F32 zhigh = 0.0;
75                      int iz0 = iy0 + iz;
76                      getBoxCoordinate
77                      (nsample_z, pos_sample_tot_+istart[iy0], 2,
                                istart[iz0]-istart[iy0],
78                      iend[iz0]-istart[iy0], half_length_root_,
                             zlow, zhigh);
79                      domain_rem_[iz0].low_[2] = alpha_*zlow
80                                               + (1.0-alpha_)*
                                                     domain_old[
                                                     iz0].low_
                                                     [2];
81                      domain_rem_[iz0].high_[2] = alpha_*zhigh
82                                                + (1.0-alpha_)*
                                                      domain_old[
                                                      iz0].high_
                                                      [2];
83                  }
84              }
85          }
86  #endif
87      }
88      MPI::COMM_WORLD.Bcast(domain_rem_, n_proc, MPI_F32VEC, 0);
89  }
```

D.2.4.1　ルートノード以外も使う方法　上の場合だと、全ての計算をルートノードにやらせているため、サンプリングや一番最初のソートがやや重いと考えられる。なので、以下にルートノードに全ての粒子を集めない方法を記述する。

1. 同じ x ランクを持つ (y-z) スラブ内で粒子のサンプリングを行う。

2．前回求めたドメインの x 座標のみを考えて、収まるべきスラブに粒子を移動させる。

3．プロセスの粒子数の prefix sum を求めておいて、スラブの分割点になりそうな粒子をクイックセレクトで求める。この時平滑化係数を使っていれば、スラブの境界と粒子が重なる事はあまりないと思われる。

4．新しく決まったスラブに収まるように粒子を移動させる。

5．各スラブ内で、y,z のドメインを決める。

D.3 粒子交換

粒子交換の実装について述べる。やり方は大きく分けて 2 通り考えられる。一つはプロセス数でループを回し、さらに粒子方向にもループを回して、粒子の行き先のプロセスを探して、送信バッファに入れていく方法。もう一つは粒子方向でループを回し、各粒子がどのプロセスに行くかを探す方法である。後者では粒子の行き先を探すのにルートドメインの分割のツリー構造が使えるため、高速であるが、各送信バッファに入る粒子数は最後までわからない。しかし、送信バッファのサイズは前回のサイズから推定する事も出来るし、そうでなくてもリンクトリストを使ったり、ループを二回回せば良い。

以下にループを 2 回回した場合のサンプルコードを示す。

ソースコード 20: 粒子交換

```
1  PS::PS::S32 PS::ParticleSystem::whereToGo(const PS::F64vec &
       pos, const PS::DomainInfo & dinfo){
2      PS::S32 id_node = 0;
3  #if (PARTICLE_SIMULATOR_DIMENSION == 3)
4      static const PS::S32 nz = Comm::getNProcXD[2];
5      static const PS::S32 ny = Comm::getNProcXD[1];
6      static const PS::S32 nynz = ny * nz;
7      PS::F32rec * dm_tmp = dinfo.domain_rem_;
8      while( dm_tmp[id_node].high_.x <= pos.x )
9          id_node += nynz;
10     while( dm_tmp[id_node].high_.y <= pos.y )
11         id_node += nz;
12     while( dm_tmp[id_node].high_.z <= pos.z )
13         id_node++;
14 #elif (PARTICLE_SIMULATOR_DIMENSION == 2)
15     static const PS::S32 ny = Comm::getNProcXD[1];
16     while( dm_tmp[id_node].high_.x <= pos.x)
17         id_node += ny;
18     while( dm_tmp[id_node].high_.y <= pos.y)
19         id_ ++;
```

```
20 #endif
21     return id_node;
22 }
23
24 void PS::ParticleSystem::exchangeParticle(const PS::DomainInfo
       & dinfo){
25     static const PS::S32 n_proc = Comm::getNProc();
26     static const PS::S32 rank = Comm::getRank();
27     std::vector<PS::S32> n_send;
28     n_send.reserve(n_proc);
29     std::vector<PS::S32> n_send_disp;
30     n_send_disp.reserve(n_proc+1);
31     for(PS::S32 i=0; i<n_proc; i++) n_send[i] = 0;
32     PS::S32 nsend_net = 0;
33     for(PS::S32 i=0; i<n_ptcl_loc_; i++){
34         PS::S32 id_send = whereToGo(ptcl_[i].pos, dinfo);
35         n_send[id_send]++;
36     }
37     n_send_disp[0] = 0;
38     for(PS::S32 i=0; i<n_proc; i++){
39         n_send_disp[i+1] = n_send_disp[i] + n_send[i];
40     }
41     std::vector<Tptcl> ptcl_send;
42     ptcl_send.reserve(n_send_disp[n_proc]);
43     for(PS::S32 i=0; i<n_ptcl_loc_; i++){
44         PS::S32 id_send = whereToGo(ptcl_[i].pos, dinfo);
45         ptcl_send[n_send_disp[id_send] + n_send[id_send]] =
               ptcl_[i];
46     }
47     std::vector<PS::S32> n_recv;
48     n_recv.reserve(n_proc);
49     MPI::COMM_WORLD.Alltoall(&n_send[0], 1, MPI::INT, &n_recv
           [0], 1, MPI::INT );
50     // do something to exchange particle
51 }
```