FDPSユーザーチュートリアル

谷川衝、細野七月、岩澤全規、似鳥啓吾、村主崇行、牧野淳一郎
理化学研究所 計算科学研究機構 粒子系シミュレータ研究チーム

0 目 次

1 変更記録

- 2015/01/25
  - 作成
- 2015/03/17
  - バージョン 1.0 リリース
- 2015/03/18
  - Particle Mesh クラス関連のライセンス事項を修正。
- 2015/03/30
  - N 体コードのログを修正
- 2015/03/31
  - サンプルの N 体コードのエネルギー計算の位置を修正
- 2015/12/01
  - GPU を使用する場合についての説明を追加 (セクション 3.4.1.8)

2 概要

本節では、Framework for Developing Particle Simulator (FDPS) の概要について述べる。FDPS は粒子シミュレーションのコード開発を支援するフレームワークである。FDPS が行うのは、計算コストの最も大きな粒子間相互作用の計算と、粒子間相互作用の計算のコストを負荷分散するための処理である。これらはマルチプロセス、マルチスレッドで並列に処理することができる。比較的計算コストが小さく、並列処理を必要としない処理 (粒子の軌道計算など) はユーザーが行う。

FDPS が対応している座標系は、2 次元直交座標系と 3 次元直交座標系である。また、境界条件としては、開放境界条件と周期境界条件に対応している。周期境界条件の場合、x、y、z 軸方向の任意の組み合わせの周期境界条件を課すことができる。

ユーザーは粒子間相互作用の形を定義する必要がある。定義できる粒子間相互作用の形には様々なものがある。粒子間相互作用の形を大きく分けると 2 種類あり、1 つは長距離力、もう 1 つは短距離力である。この 2 つの力は、遠くの複数の粒子からの作用を 1 つの超粒子からの作用にまとめるか (長距離力)、まとめないか (短距離力) という基準でもって分類される。

長距離力には、小分類があり、無限遠に存在する粒子からの力も計算するカットオフなし長距離力と、ある距離以上離れた粒子からの力は計算しないカットオフあり長距離力がある。前者は開境界条件下における重力やクーロン力に対して、後者は周期境界条件下の重力やクーロン力に使うことができる。後者のためには Particle Mesh 法などが必要となるが、これは FDPS の拡張機能として用意されている。

短距離力には、小分類が 4 つ存在する。短距離力の場合、粒子はある距離より離れた粒子からの作用は受けない。すなわち必ずカットオフが存在する。このカットオフ長の決め方によって、小分類がなされる。すなわち、全粒子のカットオフ長が等しいコンスタントカーネル、 カットオフ長が作用を受ける粒子固有の性質で決まるギャザーカーネル、カットオフ長が作用を与える粒子固有の性質で決まるスキャッタカーネル、カットオフ長が作用を受ける粒子と作用を与える粒子の両方の性質で決まるシンメトリックカーネルである。コンスタントカーネルは分子動力学における LJ 力に適用でき、その他のカーネルは SPH などに適用できる。

ユーザーは、粒子間相互作用や粒子の軌道積分などを、C++言語を用いて記述する。

3 入門：サンプルコードを動かしてみよう

本節では、まずはじめに、FDPS の動作環境、必要なソフトウェア、インストール方法などを説明し、その後、サンプルコードの使用方法を説明する。サンプルコードの中身に関しては、次節 (第 4 節) で詳しく述べる。

3.1　動作環境

FDPS は Linux, Mac OS X, Windows などの OS 上で動作する。

3.2　必要なソフトウェア

本節では、FDPS を使用する際に必要となるソフトウェアを記述する。まず標準機能を用いるのに必要なソフトウェア、次に拡張機能を用いるのに必要なソフトウェアを記述する。

3.2.1　標準機能

本節では、FDPS の標準機能のみを使用する際に必要なソフトウェアを記述する。最初に逐次処理機能のみを用いる場合（並列処理機能を用いない場合）に必要なソフトウェアを記述する。次に並列処理機能を用いる場合に必要なソフトウェアを記述する。

3.2.1.1　逐次処理

逐次処理の場合に必要なソフトウェアは以下の通りである。

- make
- C++コンパイラ (gcc バージョン 4.4.5 以降なら確実, K コンパイラバージョン 1.2.0 で動作確認済)

3.2.1.2　並列処理

本節では、FDPS の並列処理機能を用いる際に必要なソフトウェアを記述する。まず、OpenMP を使用する際に必要なソフトウェア、次に MPI を使用する際に必要なソフトウェア、最後に OpenMP と MPI を同時に使用する際に必要なソフトウェアを記述する。

3.2.1.2.1　OpenMP

OpenMP を使用する際に必要なソフトウェアは以下の通り。

- make
- OpenMP 対応の C++コンパイラ (gcc version 4.4.5 以降なら確実, K コンパイラバージョン 1.2.0 で動作確認済)

3.2.1.2.2　MPI

MPI を使用する際に必要なソフトウェアは以下の通り。

- make
- MPI version 1.3 対応の C++コンパイラ (Open MPI 1.8.1 で動作確認済, K コンパイラバージョン 1.2.0 で動作確認済)

3.2.1.2.3　MPI+OpenMP

MPI と OpenMP を同時に使用する際に必要なソフトウェアは以下の通り。

- make
- MPI version 1.3 と OpenMP に対応の C++コンパイラ (Open MPI 1.8.1 で動作確認済, K コンパイラバージョン 1.2.0 で動作確認済)

3.2.2　拡張機能

本節では、FDPS の拡張機能を使用する際に必要なソフトウェアについて述べる。FDPS の拡張機能には Particle Mesh がある。以下では Particle Mesh を使用する際に必要なソフトウェアを述べる。

3.2.2.1　Particle Mesh

Particle Mesh を使用する際に必要なソフトウェアは以下の通りである。

- make
- MPI version 1.3 と OpenMP に対応の C++コンパイラ (Open MPI 1.8.1 で動作確認済)
- FFTW 3.3 以降

3.3　インストール

本節では、FDPS のインストールについて述べる。取得方法、ビルド方法について述べる。

3.3.1　取得方法

ここではFDPSの取得方法を述べる。最初に最新バージョンの取得方法、次に過去のバージョンの取得方法を述べる。

3.3.1.1　最新バージョン

以下の方法のいずれかでFDPSの最新バージョンを取得できる。

- ブラウザから
  1. ウェブサイト https://github.com/FDPS/FDPS で”Download ZIP”をクリックし、ファイル FDPS-master.zip をダウンロード
  2. FDPSを展開したいディレクトリに移動し、圧縮ファイルを展開
- コマンドラインから
  - Subversionを用いる場合：以下のコマンドを実行するとディレクトリ trunk の下をSubversionレポジトリとして使用できる

```
$ svn co --depth empty https://github.com/FDPS/FDPS
$ cd FDPS
$ svn up trunk
```

  - Gitを用いる場合：以下のコマンドを実行するとカレントディレクトリにディレクトリ FDPS ができ、その下をGitのレポジトリとして使用できる

```
$ git clone git://github.com/FDPS/FDPS.git
```

3.3.1.2　過去のバージョン

以下の方法でブラウザからFDPSの過去のバージョンを取得できる。

- ウェブサイト https://github.com/FDPS/FDPS/releases に過去のバージョンが並んでいるので、ほしいバージョンをクリックし、ダウンロード
- FDPSを展開したいディレクトリに移動し、圧縮ファイルを展開

3.3.2　インストール方法

configure などをする必要はない。

3.4　サンプルコードの使用方法

本節ではサンプルコードの使用方法について説明する。サンプルコードには重力 N 体シミュレーションコードと、SPH シミュレーションコードがある。最初に重力 N 体シミュレーションコード、次に SPH シミュレーションコードの使用について記述する。サンプルコードは拡張機能を使用していない。

3.4.1　重力 N 体シミュレーションコード

本サンプルコードは、FDPS を用いて書かれた無衝突系の N 体計算コードである。このコードでは一様球のコールドコラプス問題を計算し、粒子分布のスナップショットを出力する。

3.4.1.1　概要

以下の手順で本コードを使用できる。

- ディレクトリ`$(FDPS)/sample/c++/nbody` に移動。これ以後、ディレクトリ`$(FDPS)` は FDPS の最も上の階層のディレクトリを指す (`$(FDPS)` は環境変数にはなっていない)。`$(FDPS)` は FDPS の取得によって異なり、ブラウザからなら FDPS-master, Subversion からなら trunk, Git からなら FDPS である。
- カレントディレクトリにある `Makefile` を編集
- コマンドライン上で `make` を実行
- `nbody.out` ファイルの実行
- 結果の解析

最後に x86 版 Phantom-GRAPE を使う場合について述べる。

3.4.1.2　ディレクトリ移動

ディレクトリ`$(FDPS)/sample/c++/nbody` に移動する。

3.4.1.3　Makefile の編集

`Makefile` の編集項目は以下の通りである。OpenMP と MPI を使用するかどうかで編集方法が変ることに注意。

- OpenMP も MPI も使用しない場合
  - 変数 `CC` に C++コンパイラを代入する
- OpenMP のみ使用の場合

  - 変数 CC に OpenMP 対応の C++コンパイラを代入する
  - CFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp の行のコメントアウトを外す。Intel コンパイラの場合には、-fopenmp を-qopenmp か-openmp に変更する (どちらを指定すべきかはコンパイラのバージョンに依る)。

- MPI のみ使用の場合
  - 変数 CC に MPI 対応の C++コンパイラを代入する
  - CFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL の行のコメントアウトを外す

- OpenMP と MPI の同時使用の場合
  - 変数 CC に MPI 対応の C++コンパイラを代入する
  - CFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp の行のコメントアウトを外す。Intel コンパイラの場合には、-fopenmp を-qopenmp か-openmp に変更する (どちらを指定すべきかはコンパイラのバージョンに依る)。
  - CFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL の行のコメントアウトを外す

3.4.1.4　make の実行

make コマンドを実行する。

3.4.1.5　実行

実行方法は以下の通りである。

- MPI を使用しない場合、コマンドライン上で以下のコマンドを実行する

```
$ ./nbody.out
```

- MPI を使用する場合、コマンドライン上で以下のコマンドを実行する

```
$ MPIRUN -np NPROC ./nbody.out
```

ここで、MPIRUN には mpirun や mpiexec などが、NPROC には使用する MPI プロセスの数が入る。

正しく終了すると、標準エラー出力は以下のようなログを出力する。energy error は絶対値で 1×10^{-3} のオーダーに収まっていればよい。

```
time:  9.5000000 energy error: -3.804653e-03
time:  9.6250000 energy error: -3.971175e-03
time:  9.7500000 energy error: -3.822343e-03
time:  9.8750000 energy error: -3.884310e-03
******** FDPS has successfully finished. ********
```

3.4.1.6　結果の解析

ディレクトリ result に粒子分布を出力したファイル”000x.dat” ができている。x は 0 から 9 の値で、時刻を表す。 出力ファイルフォーマットは 1 列目から順に粒子の ID, 粒子の質量、位置の x, y, z 座標、粒子の x, y, z 軸方向の速度である。

ここで実行したのは、粒子数 1024 個からなる一様球 (半径 3) のコールドコラプスである。コマンドライン上で以下のコマンドを実行すれば、時刻 9 における xy 平面に射影した粒子分布を見ることができる。

```
$ gnuplot
$ plot "result/0009.dat" using 3:4
```

他の時刻の粒子分布をプロットすると、一様球が次第に収縮し、その後もう一度膨張する様子を見ることができる (図 1 参照)。

粒子数を 10000 個にして計算を行いたい場合には以下のように実行すればよい (MPI を使用しない場合)。

```
$ ./nbody.out -N 10000
```

3.4.1.7　x86 版 Phantom-GRAPE を使う場合

Phantom-GRAPE は SIMD 命令を効率的に活用することで重力相互作用の計算を高速に実行するライブラリである (詳細は Tanikawa et al.[2012, New Astronomy, 17, 82] と Tanikawa et al.[2012, New Astronomy, 19, 74] を参照のこと)。

まず、使用環境を確認する。SIMD 命令セット AVX をサポートする Intel CPU または AMD CPU を搭載したコンピュータを使用しているならば、x86 版 Phantom-GRAPE を使用可能である。

次にディレクトリ $(FDPS)/src/phantom_grape_x86/G5/newton/libpg5 に移動して、ファイル Makefile を編集し、コマンド make を実行して Phantom-GRAPE のライブラリ libpg5.a を作る。

最後に、ディレクトリ $(FDPS)/sample/c++/nbody に戻り、ファイル Makefile 内の ’’#use_phantom_grape_x86 = yes’’ の’’#’’ を消す。make を実行してコンパイルする (OpenMP, MPI の使用・不使用どちらにも対応) と、x86 版 Phantom-GRAPE を使用したコードができ

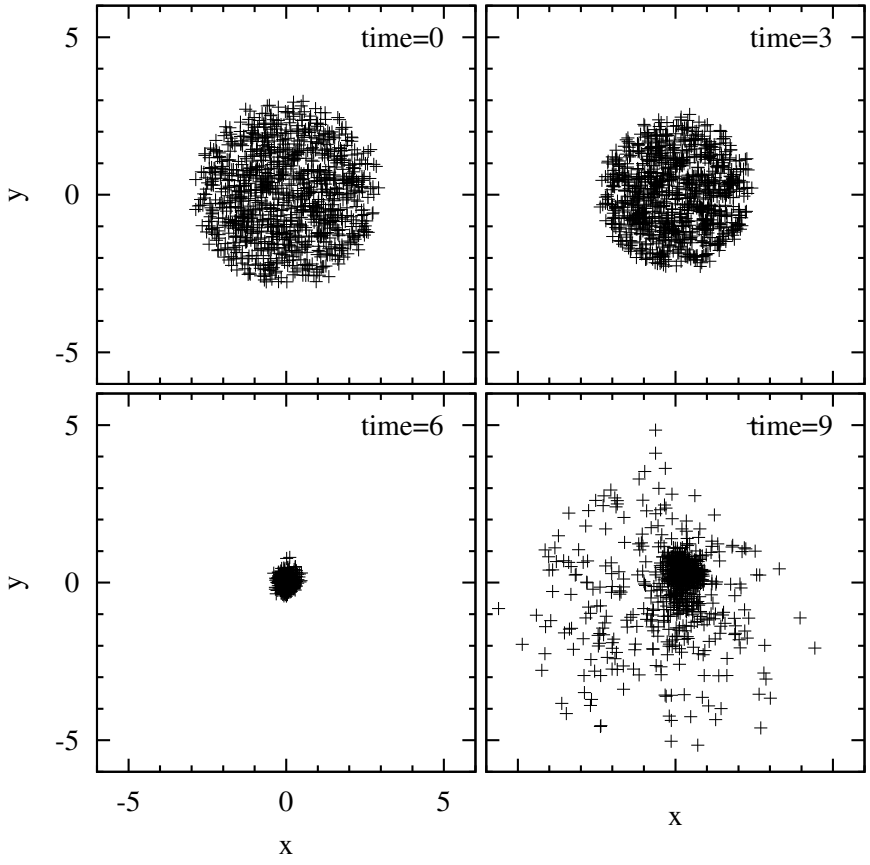

図 1:

ている。上と同様の方法で実行・結果の確認を行うとさきほどと同様の結果が得られる。

Intel Core i5-3210M CPU @ 2.50GHz の 2 コアで性能テスト (OpenMP 使用、MPI 不使用) をした結果、粒子数 8192 の場合に、Phatom-GRAPE を使うと、使わない場合に比べて、最大で 5 倍弱ほど高速なコードとなる。以下が最適化された実行例。

```
$ ./nbody.out -N 8192 -n 256
```

ここで、オプション「-n 数値」で相互作用リストを共有する粒子数の上限を指定している。

3.4.1.8　NVIDIA の GPU を使う場合

サンプルには CUDA によって書かれた NVIDIA の GPU 用のカーネルも付属している。ディレクトリ`$(FDPS)/sample/c++/nbody` の中のファイル `Makefile` 内の ’’`#use_cuda_gpu = yes`’’ の’’`#`’’ を消し、更に自分の環境に応じて、`CUDA_HOME` の場所を設定する。`make` を実行してコンパイルする (OpenMP, MPI の使用・不使用どちらにも対応) と、GPU を使用したコードができている。上と同様の方法で実行・結果の確認を行うとさきほどと同様の結果が得られる。

3.4.2　SPH シミュレーションコード

本サンプルコードには標準 SPH 法が FDPS を使って実装されている。簡単のため、smoothing length は一定値を取ると仮定している。コードでは、3 次元の衝撃波管問題の初期条件を生成し、衝撃波管問題を実際に計算する。

3.4.2.1　概要

以下の手順で本コードを使用できる。

- ディレクトリ`$(FDPS)/sample/c++/sph` に移動
- カレントディレクトリにある `Makefile` を編集 (後述)
- コマンドライン上で `make` を実行
- `sph.out` ファイルの実行 (後述)
- 結果の解析 (後述)

3.4.2.2　ディレクトリ移動

ディレクトリ`$(FDPS)/sample/c++/sph` に移動する。

3.4.2.3　Makefile の編集

Makefile の編集の仕方は N 体計算の場合と同一なので、 第 3.4.1.3 節を参照されたい。

3.4.2.4　make の実行

make コマンドを実行する。

3.4.2.5　実行

実行方法は以下の通りである。

- MPI を使用しない場合、コマンドライン上で以下のコマンドを実行する

```
$ ./sph.out
```

- MPI を使用する場合、コマンドライン上で以下のコマンドを実行する

```
$ MPIRUN -np NPROC ./sph.out
```

  ここで、MPIRUN には mpirun や mpiexec などが、NPROC には使用する MPI プロセスの数が入る。

正しく終了すると、標準エラー出力は以下のようなログを出力する。

```
******** FDPS has successfully finished. ********
```

3.4.2.6　結果の解析

実行するとディレクトリ result にファイルが出力されている。ファイル名は”00xx.dat”(x には数字が入る) となっている。ファイル名は時刻を表す。 出力ファイルフォーマットは 1 列目から順に粒子の ID、粒子の質量、位置の x, y, z 座標、粒子の x, y, z 軸方向の速度、密度、内部エネルギー、圧力である。

コマンドライン上で以下のコマンドを実行すれば、横軸に粒子の x 座標、縦軸に粒子の密度をプロットできる (時刻は 40)。

```
$ gnuplot
$ plot "result/0040.dat" using 3:9
```

正しい答が得られれば、図 2 のような図を描ける。

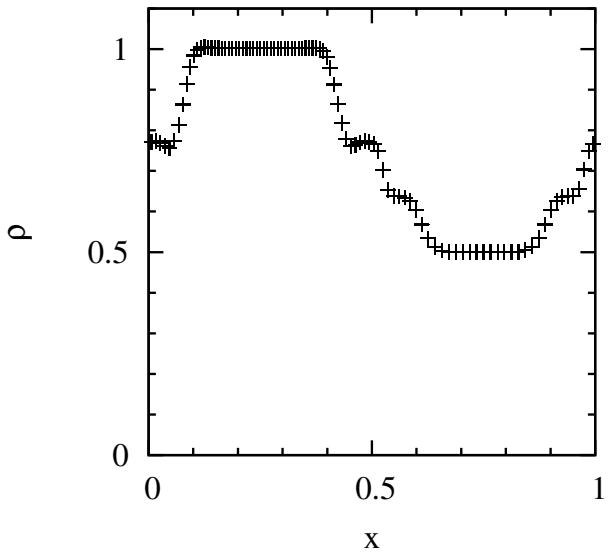

図 2: 衝撃波管問題の時刻 $t = 40$ における密度分布

4 サンプルコードの解説

本節では、前節 (第 3 節) で動かしたサンプルコードについての解説を行う。特に、ユーザが定義しなければならないクラスや FDPS の各種 API の使い方について詳しく述べる。

4.1 N 体シミュレーションコード

4.1.1 作業ディレクトリ

作業ディレクトリは$(FDPS)/sample/c++/nbody である。まずは、そこに移動する。

```
$ cd (FDPS)/sample/c++/nbody
```

4.1.2 ユーザー定義クラス

本節では、FDPS の機能を用いて N 体計算を行う際、ユーザーが記述しなければならないクラスについて記述する。

4.1.2.1 FullParticle 型

ユーザーは Full Particle 型を記述しなければならない。Full Particle 型には、シミュレーションを行うにあたって、N 体粒子が持っているべき全ての物理量が含まれている。Listing 1 に本サンプルコードの Full Particle 型の実装例を示す (`user-defined.hpp` を参照)。本サンプルコードでは、Full Particle型がEssential Particle I型、Essential Particle J型、そして、Force型を兼ねている。また、Full Particle 型には、データのコピーするのに必要なメンバ関数 `copyFromFP` と `copyFromForce` を持たせている。その他、粒子質量を返す関数である `getCharge`、粒子座標を返す関数である `getPos` が必要になる。また、積算対象のメンバ変数である加速度とポテンシャルを 0 クリアするための関数 `clear` が必要になる。本サンプルコードでは、FDPS に備わっているファイル入出力関数を使用するため、それに必要な関数である `writeAscii()` と `readAscii()` を書いてある。

Listing 1: FullParticle 型

```
1  class FPGrav{
2  public:
3      PS::S64    id;
4      PS::F64    mass;
5      PS::F64vec pos;
6      PS::F64vec vel;
7      PS::F64vec acc;
8      PS::F64    pot;
9
10     static PS::F64 eps;
11
```

```
12      PS::F64vec getPos() const {
13          return pos;
14      }
15
16      PS::F64 getCharge() const {
17          return mass;
18      }
19
20      void copyFromFP(const FPGrav & fp){
21          mass = fp.mass;
22          pos  = fp.pos;
23      }
24
25      void copyFromForce(const FPGrav & force) {
26          acc = force.acc;
27          pot = force.pot;
28      }
29
30      void clear() {
31          acc = 0.0;
32          pot = 0.0;
33      }
34
35      void writeAscii(FILE* fp) const {
36          fprintf(fp, "%lld\t%g\t%g\t%g\t%g\t%g\t%g\t%g\n",
37                  this->id, this->mass,
38                  this->pos.x, this->pos.y, this->pos.z,
39                  this->vel.x, this->vel.y, this->vel.z);
40      }
41
42      void readAscii(FILE* fp) {
43          fscanf(fp, "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\n",
44                 &this->id, &this->mass,
45                 &this->pos.x, &this->pos.y, &this->pos.z,
46                 &this->vel.x, &this->vel.y, &this->vel.z);
47          }
48
49  };
```

4.1.2.2　calcForceEpEp 型

ユーザーは calcForceEpEp 型を記述しなければならない。calcForceEpEp 型には、Force の計算の具体的な内容を書く必要がある。Listing 2 に、本サンプルコードの calcForceEpEp 型を示す (`user-defined.hpp` を参照)。これは、本サンプルコードの 3 つの動作モードの内、Phantom-GRAPE ライブラリを使用せずに CPU で実行する場合の calcForceEpEp 型である (動作モードについては、第 3 節を参照のこと)。

本サンプルコードではテンプレート関数[注 1)]を用いて実装している。また、テンプレート

[注 1)] テンプレート関数とは C++言語のテンプレート機能を関数に適用することで汎用的な関数としたものである。通常の関数では、関数の型および関数の引数の型をその場ですべて指定して記述/定義するが、テンプレート関数では、`templete <...>`内に指定された一般的なデータ型・クラスを関数定義に使用することがで

関数の引数は、EssentialParticleI の配列、EssentialParticleI の個数、EssentialParticleJ の配列、EssentialParticleJ の個数、Force 型の配列である。

Listing 2: calcForceEpEp 型

```
template <class TParticleJ>
void CalcGravity(const FPGrav * ep_i,
                 const PS::S32 n_ip,
                 const TParticleJ * ep_j,
                 const PS::S32 n_jp,
                 FPGrav * force) {
    PS::F64 eps2 = FPGrav::eps * FPGrav::eps;
    for(PS::S32 i = 0; i < n_ip; i++){
        PS::F64vec xi = ep_i[i].getPos();
        PS::F64vec ai = 0.0;
        PS::F64 poti = 0.0;
        for(PS::S32 j = 0; j < n_jp; j++){
            PS::F64vec rij    = xi - ep_j[j].getPos();
            PS::F64    r3_inv = rij * rij + eps2;
            PS::F64    r_inv  = 1.0/sqrt(r3_inv);
            r3_inv  = r_inv * r_inv;
            r_inv  *= ep_j[j].getCharge();
            r3_inv *= r_inv;
            ai     -= r3_inv * rij;
            poti   -= r_inv;
        }
        force[i].acc += ai;
        force[i].pot += poti;
    }
}
```

4.1.3　プログラム本体

本節では、FDPS を用いて N 体計算を行うにあたり、メイン関数に書かれるべき関数に関して解説する。メイン関数はサンプルコード `nbody.cpp` 内に記述されている。

4.1.3.1　開始、終了

まずは、FDPS の初期化/開始を行う必要がある。次のように、メイン関数に記述する。

Listing 3: FDPS の開始

```
PS::Initialize(argc, argv);
```

FDPS は、開始したら明示的に終了させる必要がある。今回は、プログラムの終了と同時に FDPS も終了させるため、メイン関数の最後に次のように記述する。

きる (“一般的なクラス”の情報は、関数呼び出しの際にテンプレート引数として渡す。このため、コンパイル時にテンプレート関数のすべての型は問題なく決定される)。これによって、一般的な関数を定義することが可能となる。

Listing 4: FDPS の終了

```
1 PS::Finalize();
```

4.1.3.2　初期化

FDPS の初期化に成功した場合、ユーザーはコード中で用いるオブジェクトを作成する必要がある。本節では、オブジェクトの生成/初期化の仕方について解説する。

4.1.3.2.1　オブジェクトの生成

今回の計算では、粒子群クラス、領域クラスに加え、重力計算用 tree を一本生成する必要がある。以下にそのコードを記す。これらはサンプルコード `nbody.cpp` の `main` 関数内に記述されている。

Listing 5: オブジェクトの生成

```
1 PS::DomainInfo dinfo;
2 PS::ParticleSystem<FPGrav> system_grav;
3 PS::TreeForForceLong<FPGrav, FPGrav, FPGrav>::Monopole tree_grav;
```

4.1.3.2.2　領域クラスの初期化

ユーザーはオブジェクトを作成したら、そのオブジェクトの初期化を行う必要がある。本サンプルコードでは周期境界等は用いていないため、領域クラスの初期化は `initialize` メソッドを実行するだけでよい:

Listing 6: 領域クラスの初期化

```
1 const PS::F32 coef_ema = 0.3;
2 dinfo.initialize(coef_ema);
```

ここで、`initialize` メソッドの引数は領域分割に使用される指数移動平均の平滑化係数を表す。この係数の意味については仕様書 § 9.2.1 に詳しい解説があるので、そちらを参照されたい。

4.1.3.2.3　粒子群クラスの初期化

次に、粒子群クラスの初期化を行う必要がある。粒子群クラスの初期化は、`initialize` メソッドで行う:

Listing 7: 粒子群クラスの初期化

```
1 system_grav.initialize();
```

4.1.3.2.4　相互作用ツリークラスの初期化

次に、相互作用ツリークラスの初期化を行う必要がある。ツリークラスの初期化は`initialize`メソッドで行う。このメソッドには、引数として大雑把な粒子数を渡す必要がある。今回は、全体の粒子数 (`n_tot`) をセットしておく事にする:

Listing 8: 相互作用ツリークラスの初期化

```
1 tree_grav.initialize(n_tot, theta, n_leaf_limit, n_group_limit);
```

このメソッドには3つの省略可能引数が存在し、サンプルコードではこれらを省略せずに指定している:

- `theta` — ツリー法で力の計算をする場合の見込み角についての基準
- `n_leaf_limit` — ツリーを切るのをやめる粒子数の上限
- `n_group_limit` — 相互作用リストを共有する粒子数の上限

4.1.3.3　ループ

本節では、時間積分ループの中で行わなければならないことについて、解説する。

4.1.3.3.1　領域分割の実行

まずは、粒子分布に基いて、領域分割を実行する。本サンプルコードでは、これを領域クラスの`decomposeDomainAll`メソッドを用いて行っている:

Listing 9: 領域分割の実行

```
1 if(n_loop % 4 == 0){
2    dinfo.decomposeDomainAll(system_grav);
3 }
```

ここで、計算時間の節約のため、領域分割は4ループ毎に1回だけ行うようにしている。

4.1.3.3.2　粒子交換の実行

次に、領域情報に基いて、プロセス間の粒子の情報を交換する。これには、粒子群クラスのメンバ関数である、以下の関数を用いる。

Listing 10: 粒子交換の実行

```
1 system_grav.exchangeParticle(dinfo);
```

4.1.3.3.3　相互作用計算の実行

領域分割・粒子交換が終了したら、相互作用の計算を行う。
これには、tree クラスの`calcForceAllAndWriteBack`メソッドを用いる。

Listing 11: 相互作用計算の実行

```
1 tree_grav.calcForceAllAndWriteBack(CalcGravity<FPGrav>(),
2                                    CalcGravity<PS::SPJMonopole>(),
3                                    system_grav,
4                                    dinfo);
```

ここで、メソッドの引数に `CalcGravity<....>()` のような記述があるが、この`<...>`内に書かれているものがテンプレート引数である。

4.1.3.3.4　時間積分

本サンプルコードでは、時間積分を Leapfrog 時間積分法によって行う。時間積分は形式的に、$K(\frac{\Delta t}{2})D(\Delta t)K(\frac{\Delta t}{2})$ と表される。ここで、Δt は時間刻み、$K(\cdot)$ は速度を指定された時間だけ時間推進するオペレータ、$D(\cdot)$ は位置を指定された時間だけ時間推進するオペレータである。本サンプルコードにおいて、これらのオペレータは、関数 `kick` と関数 `drift` として実装している。

時間積分ループの最初で、最初の $D(\Delta t)K(\frac{\Delta t}{2})$ の計算を行い、粒子の座標と速度の情報を更新している:

Listing 12: $D(\Delta t)K(\frac{\Delta t}{2})$ オペレータの計算

```
1 kick(system_grav, dt * 0.5);
2 drift(system_grav, dt);
```

時間積分ループの次の部分では、力の計算を行い、その後、最後の $K(\frac{\Delta t}{2})$ の計算を行っている:

Listing 13: $K(\frac{\Delta t}{2})$ オペレータの計算

```
1 kick(system_grav, dt * 0.5);
```

4.1.4　ログファイル

計算が正しく開始すると、標準エラー出力に、時間・エネルギー・エネルギー誤差の 3 つが出力される。以下はその出力の最も最初のステップでの例である。

Listing 14: 標準エラー出力

```
1 time:  0.0000000 energy: -1.974890e-01 energy error: +0.000000e+00
```

4.2　固定長 SPH シミュレーションコード

本節では、前節 (第 3 節) で使用した、固定 smoothing length での標準 SPH 法のサンプルコードの詳細について解説する。

4.2.1　作業ディレクトリ

作業ディレクトリは$(FDPS)/sample/c++/sph である。まずは、そこに移動する。

```
$ cd (FDPS)/sample/c++/sph
```

4.2.2　インクルード

FDPS はヘッダーファイルのみで構成されているため、ユーザーはソースコード中で particle_simulator.hpp を include するだけで、FDPS の機能が使用可能になる。

Listing 15: Include FDPS

```
1 #include <particle_simulator.hpp>
```

4.2.3　ユーザー定義クラス

本節では、FDPS の機能を用いて SPH の計算を行う際に、ユーザーが記述しなければならないクラスについて記述する。

4.2.3.1　FullParticle 型

ユーザーは FullParticle 型を記述しなければならない。FullParticle 型には、シミュレーションを行うにあたって、SPH 粒子が持っているべき全ての物理量が含まれている。また、FullParticle 型には後述する Force 型から、結果をコピーするのに必要なメンバ関数を持つ必要がある。その他、粒子質量を返す関数である `getCharge()`、粒子座標を返す関数である `getPos()`、近傍粒子の探索半径を返す関数である `getRSearch()`、粒子の座標を書き込む関数である `setPos()` が必要になる。本サンプルコードでは、FDPS に備わっているファイル入出力関数を用いる。そのため、これに必要な関数である `writeAscii()` と `readAscii()` を書いてある。また、これらに加え、状態方程式から圧力を計算するメンバ関数である、`setPressure()` が記述されているが、この関数は FDPS が用いるものではないため、必須のものではないことに注意する。以下は本サンプルコード中で用いる FullParticle 型の例である。

Listing 16: FullParticle 型

```
1 struct FP{
2     PS::F64 mass;
3     PS::F64vec pos;
4     PS::F64vec vel;
5     PS::F64vec acc;
6     PS::F64 dens;
7     PS::F64 eng;
8     PS::F64 pres;
9     PS::F64 smth;
```

```
10      PS::F64 snds;
11      PS::F64 eng_dot;
12      PS::F64 dt;
13      PS::S64 id;
14      PS::F64vec vel_half;
15      PS::F64 eng_half;
16      void copyFromForce(const Dens& dens){
17          this->dens = dens.dens;
18      }
19      void copyFromForce(const Hydro& force){
20          this->acc     = force.acc;
21          this->eng_dot = force.eng_dot;
22          this->dt      = force.dt;
23      }
24      PS::F64 getCharge() const{
25          return this->mass;
26      }
27      PS::F64vec getPos() const{
28          return this->pos;
29      }
30      PS::F64 getRSearch() const{
31          return kernelSupportRadius * this->smth;
32      }
33      void setPos(const PS::F64vec& pos){
34          this->pos = pos;
35      }
36      void writeAscii(FILE* fp) const{
37          fprintf(fp,
38                  "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t"
39                  "%lf\t%lf\t%lf\t%lf\t%lf\n",
40                  this->id, this->mass,
41                  this->pos.x, this->pos.y, this->pos.z,
42                  this->vel.x, this->vel.y, this->vel.z,
43                  this->dens, this->eng, this->pres);
44      }
45      void readAscii(FILE* fp){
46          fscanf(fp,
47                 "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t"
48                 "%lf\t%lf\t%lf\t%lf\t%lf\n",
49                 &this->id, &this->mass,
50                 &this->pos.x, &this->pos.y, &this->pos.z,
51                 &this->vel.x, &this->vel.y, &this->vel.z,
52                 &this->dens, &this->eng, &this->pres);
53      }
54      void setPressure(){
55          const PS::F64 hcr = 1.4;
56          pres = (hcr - 1.0) * dens * eng;
57          snds = sqrt(hcr * pres / dens);
58      }
59  };
```

4.2.3.2　EssentialParticleI 型

ユーザーは EssentialParticleI 型を記述しなければならない。EssentialParticleI 型には、Force の計算を行う際、i 粒子が持っているべき全ての物理量をメンバ変数として持っている必要がある。また、本サンプルコード中では、EssentialParticleJ 型も兼ねているため、j 粒子が持っているべき全ての物理量もメンバ変数として持っている必要がある。また、EssentialParticleI 型には前述した FullParticle 型から、値をコピーするのに必要なメンバ関数を持つ必要がある。その他、粒子座標を返す関数である `getPos()`、近傍粒子の探索半径を返す関数である `getRSearch()`、粒子の座標を書き込む関数である `setPos()` が必要になる。以下は本サンプルコード中で用いる EssentialParticleI 型の例である。

Listing 17: EssentialParticleI 型

```
1  struct EP{
2     PS::F64vec pos;
3     PS::F64vec vel;
4     PS::F64    mass;
5     PS::F64    smth;
6     PS::F64    dens;
7     PS::F64    pres;
8     PS::F64    snds;
9     void copyFromFP(const FP& rp){
10       this->pos  = rp.pos;
11       this->vel  = rp.vel;
12       this->mass = rp.mass;
13       this->smth = rp.smth;
14       this->dens = rp.dens;
15       this->pres = rp.pres;
16       this->snds = rp.snds;
17    }
18    PS::F64vec getPos() const{
19       return this->pos;
20    }
21    PS::F64 getRSearch() const{
22       return kernelSupportRadius * this->smth;
23    }
24    void setPos(const PS::F64vec& pos){
25       this->pos = pos;
26    }
27 };
```

4.2.3.3　Force 型

ユーザーは Force 型を記述しなければならない。Force 型には、Force の計算を行った際にその結果として得られる全ての物理量をメンバ変数として持っている必要がある。また、本サンプルコード中では、Force は密度の計算と、流体相互作用計算の 2 つが存在するため、Force 型は 2 つ書く必要がある。また、積算対象のメンバ変数を 0 ないし初期値に設定するための関数 clear が必要になる。以下は本サンプルコード中で用いる Force 型の例である。

この例の `Dens` クラスには、smoothing length を表すメンバ変数 `smth` が用意されている。

本来、固定長 SPH では、Force 型に smoothing length に対応するメンバを持たせる必要はない。しかし、ここでは、ユーザが将来的に可変長 SPH に移行することを想定して用意してある。可変長 SPH の formulation の 1 つである Springel [2005,MNRAS,364,1105] の方法では、密度計算と同時に smoothing length を計算する必要がある。そのような formulation を実装する場合には、この例のように、Force 型に smoothing length を持たせる必要が生じる。本サンプルコードでは固定長 SPH を使うため、メンバ関数 `clear` で `smth` を 0 クリアされないようにしている (0 クリアされては密度計算が破綻するため)。

また、`Hydro` クラスには各粒子の時間刻みを表すメンバ変数 `dt` が用意されている。本サンプルコードでは、`dt` を 0 クリアしていない。これは `dt` が積算対象でないため、0 クリアが不要であるからである。

Listing 18: Force 型

```
1  class Dens{
2     public:
3     PS::F64 dens;
4     PS::F64 smth;
5     void clear(){
6        dens = 0;
7     }
8  };
9  class Hydro{
10    public:
11    PS::F64vec acc;
12    PS::F64 eng_dot;
13    PS::F64 dt;
14    void clear(){
15       acc = 0;
16       eng_dot = 0;
17    }
18 };
```

4.2.3.4　calcForceEpEp 型

ユーザーは calcForceEpEp 型を記述しなければならない。calcForceEpEp 型には、Force の計算の具体的な内容を書く必要がある。今回のサンプルコード中では、ファンクタを用いて実装している。また、ファンクタの引数は、EssentialParticleI の配列、EssentialParticleI の個数、EssentialParticleJ の配列、EssentialParticleJ の個数、Force 型の配列である。また、本サンプルコード中では、Force は密度の計算と、流体相互作用計算の 2 つが存在するため、calcForceEpEp 型は 2 つ書く必要がある。以下は本サンプルコード中で用いる calcForceEpEp 型の例である。

Listing 19: calcForceEpEp 型

```
1  class CalcDensity{
2     public:
3     void operator () (const EP* const ep_i, const PS::S32 Nip,
4                       const EP* const ep_j, const PS::S32 Njp,
5                       Dens* const dens){
```

```
6         for(PS::S32 i = 0 ; i < Nip ; ++i){
7            dens[i].clear();
8            for(PS::S32 j = 0 ; j < Njp ; ++j){
9               const PS::F64vec dr = ep_j[j].pos - ep_i[i].pos;
10              dens[i].dens += ep_j[j].mass * W(dr, ep_i[i].smth);
11           }
12        }
13     }
14  };
15
16  class CalcHydroForce{
17     public:
18     void operator () (const EP* const ep_i, const PS::S32 Nip,
19                       const EP* const ep_j, const PS::S32 Njp,
20                       Hydro* const hydro){
21        for(PS::S32 i = 0; i < Nip ; ++ i){
22           hydro[i].clear();
23           PS::F64 v_sig_max = 0.0;
24           for(PS::S32 j = 0; j < Njp ; ++j){
25              const PS::F64vec dr = ep_i[i].pos - ep_j[j].pos;
26              const PS::F64vec dv = ep_i[i].vel - ep_j[j].vel;
27              const PS::F64 w_ij = (dv * dr < 0) ? dv * dr / sqrt(dr * dr) :
                     0;
28              const PS::F64 v_sig = ep_i[i].snds + ep_j[j].snds - 3.0 * w_ij
                     ;
29              v_sig_max = std::max(v_sig_max, v_sig);
30              const PS::F64 AV = - 0.5 * v_sig * w_ij / (0.5 * (ep_i[i].dens
                      + ep_j[j].dens));
31              const PS::F64vec gradW_ij = 0.5 * (gradW(dr, ep_i[i].smth) +
                     gradW(dr, ep_j[j].smth));
32              hydro[i].acc     -= ep_j[j].mass * (ep_i[i].pres / (ep_i[i].
                     dens * ep_i[i].dens) + ep_j[j].pres / (ep_j[j].dens *
                     ep_j[j].dens) + AV) * gradW_ij;
33              hydro[i].eng_dot += ep_j[j].mass * (ep_i[i].pres / (ep_i[i].
                     dens * ep_i[i].dens) + 0.5 * AV) * dv * gradW_ij;
34           }
35           hydro[i].dt = C_CFL * 2.0 * ep_i[i].smth / v_sig_max;
36        }
37     }
38  };
```

4.2.4　プログラム本体

本節では、FDPS を用いて SPH 計算を行う際に、メイン関数に書かれるべき関数に関して解説する。

4.2.4.1　開始、終了

まずは、FDPS の初期化/開始を行う必要がある。次のように、メイン関数に記述する。

Listing 20: FDPS の開始

```
1 PS::Initialize(argc, argv);
```

FDPS は、開始したら明示的に終了させる必要がある。今回は、プログラムの終了と同時に FDPS も終了させるため、メイン関数の最後に次のように記述する。

Listing 21: FDPS の終了

```
1 PS::Finalize();
```

4.2.4.2　初期化

FDPS の初期化に成功した場合、ユーザーはコード中で用いるオブジェクトを作成する必要がある。本説では、オブジェクトの生成/初期化の仕方について、解説する。

4.2.4.2.1　オブジェクトの生成

SPH では、粒子群クラス、領域クラスに加え、密度計算用の Gather tree を一本、相互作用計算用の Symmetry tree を一本生成する必要がある。以下にそのコードを記す。

Listing 22: オブジェクトの生成

```
1 PS::ParticleSystem<FP> sph_system;
2 PS::DomainInfo dinfo;
3 PS::TreeForForceShort<Dens, EP, EP>::Gather dens_tree;
4 PS::TreeForForceShort<Hydro, EP, EP>::Symmetry hydr_tree;
```

4.2.4.2.2　領域クラスの初期化

ユーザーはオブジェクトを作成したら、そのオブジェクトの初期化を行う必要がある。ここでは、まず領域クラスの初期化について、解説する。領域クラスの初期化が終わった後、領域クラスに周期境界の情報と、境界の大きさをセットする必要がある。今回のサンプルコードでは、x, y, z 方向に周期境界を用いる。

Listing 23: 領域クラスの初期化

```
1 dinfo.initialize();
2 dinfo.setBoundaryCondition(PS::BOUNDARY_CONDITION_PERIODIC_XYZ);
3 dinfo.setPosRootDomain(PS::F64vec(0.0, 0.0, 0.0),
4                        PS::F64vec(box.x, box.y, box.z));
```

4.2.4.2.3　粒子群クラスの初期化

次に、粒子群クラスの初期化を行う必要がある。粒子群クラスの初期化は、次の一文だけでよい。

Listing 24: 粒子群クラスの初期化

```
1 sph_system.initialize();
```

4.2.4.2.4　相互作用ツリークラスの初期化

次に、相互作用ツリークラスの初期化を行う必要がある。ツリークラスの初期化を行う関数には、引数として大雑把な粒子数を渡す必要がある。今回は、粒子数の 3 倍程度をセットしておく事にする。

Listing 25: 相互作用ツリークラスの初期化

```
1 dens_tree.initialize(3 * sph_system.getNumberOfParticleGlobal());
2 hydr_tree.initialize(3 * sph_system.getNumberOfParticleGlobal());
```

4.2.4.3　ループ

本節では、時間積分ループの中で行わなければならないことについて、解説する。

4.2.4.3.1　領域分割の実行

まずは、粒子分布に基いて、領域分割を実効する。これには、領域クラスのメンバ関数である、以下の関数を用いる。

Listing 26: 領域分割の実行

```
1 dinfo.decomposeDomainAll(sph_system);
```

4.2.4.3.2　粒子交換の実行

次に、領域情報に基いて、プロセス間の粒子の情報を交換する。これには、粒子群クラスのメンバ関数である、以下の関数を用いる。

Listing 27: 粒子交換の実行

```
1 sph_system.exchangeParticle(dinfo);
```

4.2.4.3.3　相互作用計算の実行

領域分割・粒子交換が終了したら、相互作用の計算を行う。これには、tree クラスのメンバ関数である、以下の関数を用いる。

Listing 28: 相互作用計算の実行

```
1 dens_tree.calcForceAllAndWriteBack(CalcDensity(), sph_system, dinfo);
2 hydr_tree.calcForceAllAndWriteBack(CalcHydroForce(), sph_system, dinfo);
```

4.2.5　コンパイル

作業ディレクトリで make コマンドを打てばよい。Makefile としては、sample に付属の Makefile をそのまま用いる事にする。

```
$ make
```

4.2.6　実行

MPI を使用しないで実行する場合、コマンドライン上で以下のコマンドを実行すればよい。

```
$ ./sph.out
```

もし、MPI を用いて実行する場合は、以下のコマンドを実行すればよい。

```
$ MPIRUN -np NPROC ./sph.out
```

ここで、“MPIRUN”には mpirun や mpiexec などの mpi 実行プログラムが、“NPROC”にはプロセス数が入る。

4.2.7　ログファイル

計算が終了すると、result フォルダ下にログが出力される。

4.2.8　可視化

ここでは、gnuplot を用いた可視化の方法について解説する。gnuplot で対話モードに入るために、コマンドラインから gnuplot を起動する。

```
$ gnuplot
```

対話モードに入ったら、gnuplot を用いて可視化を行う。今回は、50 番目のスナップショットファイルから、横軸を粒子の x 座標、縦軸を密度に取ったグラフを生成する。

```
gnuplot> plot "result/0040.txt" u 3:9
```

5　サンプルコード

5.1　固定長 SPH シミュレーション

固定長 SPH シミュレーションのサンプルコードを以下に示す。このサンプルは第 3, 4 節で用いた固定長 SPH シミュレーションのサンプルコードと同じものである。これをカット&ペーストしてコンパイルすれば、正常に動作する固定長 SPH シミュレーションコードを作ることができる。

Listing 29: 固定長 SPH シミュレーションのサンプルコード

```
1  // Include FDPS header
2  #include <particle_simulator.hpp>
3  // Include the standard C++ headers
4  #include <cmath>
5  #include <cstdio>
6  #include <iostream>
7  #include <vector>
8
9  /* Parameters */
10 const short int Dim = 3;
11 const PS::F64 SMTH = 1.2;
12 const PS::U32 OUTPUT_INTERVAL = 10;
13 const PS::F64 C_CFL = 0.3;
14
15 /* Kernel Function */
16 const PS::F64 pi = atan(1.0) * 4.0;
17 const PS::F64 kernelSupportRadius = 2.5;
18
19 PS::F64 W(const PS::F64vec dr, const PS::F64 h){
20    const PS::F64 H = kernelSupportRadius * h;
21    const PS::F64 s = sqrt(dr * dr) / H;
22    const PS::F64 s1 = (1.0 - s < 0) ? 0 : 1.0 - s;
23    const PS::F64 s2 = (0.5 - s < 0) ? 0 : 0.5 - s;
24    PS::F64 r_value = pow(s1, 3) - 4.0 * pow(s2, 3);
25    //if # of dimension == 3
26    r_value *= 16.0 / pi / (H * H * H);
27    return r_value;
28 }
29
30 PS::F64vec gradW(const PS::F64vec dr, const PS::F64 h){
31    const PS::F64 H = kernelSupportRadius * h;
32    const PS::F64 s = sqrt(dr * dr) / H;
33    const PS::F64 s1 = (1.0 - s < 0) ? 0 : 1.0 - s;
34    const PS::F64 s2 = (0.5 - s < 0) ? 0 : 0.5 - s;
35    PS::F64 r_value = - 3.0 * pow(s1, 2) + 12.0 * pow(s2, 2);
36    //if # of dimension == 3
37    r_value *= 16.0 / pi / (H * H * H);
38    return dr * r_value / (sqrt(dr * dr) * H + 1.0e-6 * h);
39 }
40
41 /* Class Definitions */
42 //** Force Class (Result Class)
```

```
43 class Dens{
44    public:
45    PS::F64 dens;
46    PS::F64 smth;
47    void clear(){
48       dens = 0;
49    }
50 };
51 class Hydro{
52    public:
53    PS::F64vec acc;
54    PS::F64 eng_dot;
55    PS::F64 dt;
56    void clear(){
57       acc = 0;
58       eng_dot = 0;
59    }
60 };
61
62 //** Full Particle Class
63 struct FP{
64    PS::F64 mass;
65    PS::F64vec pos;
66    PS::F64vec vel;
67    PS::F64vec acc;
68    PS::F64 dens;
69    PS::F64 eng;
70    PS::F64 pres;
71    PS::F64 smth;
72    PS::F64 snds;
73    PS::F64 eng_dot;
74    PS::F64 dt;
75    PS::S64 id;
76    PS::F64vec vel_half;
77    PS::F64 eng_half;
78    void copyFromForce(const Dens& dens){
79       this->dens = dens.dens;
80    }
81    void copyFromForce(const Hydro& force){
82       this->acc     = force.acc;
83       this->eng_dot = force.eng_dot;
84       this->dt      = force.dt;
85    }
86    PS::F64 getCharge() const{
87       return this->mass;
88    }
89    PS::F64vec getPos() const{
90       return this->pos;
91    }
92    PS::F64 getRSearch() const{
93       return kernelSupportRadius * this->smth;
94    }
95    void setPos(const PS::F64vec& pos){
96       this->pos = pos;
97    }
```

```
 98     void writeAscii(FILE* fp) const{
 99        fprintf(fp,
100                "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t"
101                "%lf\t%lf\t%lf\t%lf\t%lf\n",
102                this->id, this->mass,
103                this->pos.x, this->pos.y, this->pos.z,
104                this->vel.x, this->vel.y, this->vel.z,
105                this->dens, this->eng, this->pres);
106     }
107     void readAscii(FILE* fp){
108        fscanf(fp,
109               "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t"
110               "%lf\t%lf\t%lf\t%lf\t%lf\n",
111               &this->id, &this->mass,
112               &this->pos.x, &this->pos.y, &this->pos.z,
113               &this->vel.x, &this->vel.y, &this->vel.z,
114               &this->dens, &this->eng, &this->pres);
115     }
116     void setPressure(){
117        const PS::F64 hcr = 1.4;
118        pres = (hcr - 1.0) * dens * eng;
119        snds = sqrt(hcr * pres / dens);
120     }
121 };
122
123 //** Essential Particle Class
124 struct EP{
125     PS::F64vec pos;
126     PS::F64vec vel;
127     PS::F64    mass;
128     PS::F64    smth;
129     PS::F64    dens;
130     PS::F64    pres;
131     PS::F64    snds;
132     void copyFromFP(const FP& rp){
133        this->pos  = rp.pos;
134        this->vel  = rp.vel;
135        this->mass = rp.mass;
136        this->smth = rp.smth;
137        this->dens = rp.dens;
138        this->pres = rp.pres;
139        this->snds = rp.snds;
140     }
141     PS::F64vec getPos() const{
142        return this->pos;
143     }
144     PS::F64 getRSearch() const{
145        return kernelSupportRadius * this->smth;
146     }
147     void setPos(const PS::F64vec& pos){
148        this->pos = pos;
149     }
150 };
151
152 class FileHeader{
```

```
153     public:
154     PS::S32 Nbody;
155     PS::F64 time;
156     int readAscii(FILE* fp){
157        fscanf(fp, "%lf\n", &time);
158        fscanf(fp, "%d\n", &Nbody);
159        return Nbody;
160     }
161     void writeAscii(FILE* fp) const{
162        fprintf(fp, "%e\n", time);
163        fprintf(fp, "%d\n", Nbody);
164     }
165 };
166
167 struct boundary{
168     PS::F64 x, y, z;
169 };
170
171
172 /* Force Functors */
173 class CalcDensity{
174     public:
175     void operator () (const EP* const ep_i, const PS::S32 Nip,
176                       const EP* const ep_j, const PS::S32 Njp,
177                       Dens* const dens){
178        for(PS::S32 i = 0 ; i < Nip ; ++i){
179           dens[i].clear();
180           for(PS::S32 j = 0 ; j < Njp ; ++j){
181              const PS::F64vec dr = ep_j[j].pos - ep_i[i].pos;
182              dens[i].dens += ep_j[j].mass * W(dr, ep_i[i].smth);
183           }
184        }
185     }
186 };
187
188 class CalcHydroForce{
189     public:
190     void operator () (const EP* const ep_i, const PS::S32 Nip,
191                       const EP* const ep_j, const PS::S32 Njp,
192                       Hydro* const hydro){
193        for(PS::S32 i = 0; i < Nip ; ++ i){
194           hydro[i].clear();
195           PS::F64 v_sig_max = 0.0;
196           for(PS::S32 j = 0; j < Njp ; ++j){
197              const PS::F64vec dr = ep_i[i].pos - ep_j[j].pos;
198              const PS::F64vec dv = ep_i[i].vel - ep_j[j].vel;
199              const PS::F64 w_ij = (dv * dr < 0) ? dv * dr / sqrt(dr * dr) :
                      0;
200              const PS::F64 v_sig = ep_i[i].snds + ep_j[j].snds - 3.0 * w_ij
                     ;
201              v_sig_max = std::max(v_sig_max, v_sig);
202              const PS::F64 AV = - 0.5 * v_sig * w_ij / (0.5 * (ep_i[i].dens
                      + ep_j[j].dens));
203              const PS::F64vec gradW_ij = 0.5 * (gradW(dr, ep_i[i].smth) +
                     gradW(dr, ep_j[j].smth));
```

```
204             hydro[i].acc     -= ep_j[j].mass * (ep_i[i].pres / (ep_i[i].
                      dens * ep_i[i].dens) + ep_j[j].pres / (ep_j[j].dens *
                      ep_j[j].dens) + AV) * gradW_ij;
205             hydro[i].eng_dot += ep_j[j].mass * (ep_i[i].pres / (ep_i[i].
                      dens * ep_i[i].dens) + 0.5 * AV) * dv * gradW_ij;
206          }
207          hydro[i].dt = C_CFL * 2.0 * ep_i[i].smth / v_sig_max;
208       }
209    }
210 };
211
212 void SetupIC(PS::ParticleSystem<FP>& sph_system, PS::F64 *end_time,
        boundary *box){
213    // Place SPH particles
214    std::vector<FP> ptcl;
215    const PS::F64 dx = 1.0 / 128.0;
216    box->x = 1.0;
217    box->y = box->z = box->x / 8.0;
218    PS::S32 i = 0;
219    for(PS::F64 x = 0 ; x < box->x * 0.5 ; x += dx){
220       for(PS::F64 y = 0 ; y < box->y ; y += dx){
221          for(PS::F64 z = 0 ; z < box->z ; z += dx){
222             FP ith;
223             ith.pos.x = x;
224             ith.pos.y = y;
225             ith.pos.z = z;
226             ith.dens = 1.0;
227             ith.mass = 0.75;
228             ith.eng  = 2.5;
229             ith.id   = i++;
230             ith.smth = 0.012;
231             ptcl.push_back(ith);
232          }
233       }
234    }
235    for(PS::F64 x = box->x * 0.5 ; x < box->x * 1.0 ; x += dx * 2.0){
236       for(PS::F64 y = 0 ; y < box->y ; y += dx){
237          for(PS::F64 z = 0 ; z < box->z ; z += dx){
238             FP ith;
239             ith.pos.x = x;
240             ith.pos.y = y;
241             ith.pos.z = z;
242             ith.dens = 0.5;
243             ith.mass = 0.75;
244             ith.eng  = 2.5;
245             ith.id   = i++;
246             ith.smth = 0.012;
247             ptcl.push_back(ith);
248          }
249       }
250    }
251    for(PS::U32 i = 0 ; i < ptcl.size() ; ++ i){
252       ptcl[i].mass = ptcl[i].mass * box->x * box->y * box->z / (PS::F64)(
              ptcl.size());
253    }
```

```
254     std::cout << "# of ptcls is ... " << ptcl.size() << std::endl;
255     // Scatter SPH particles
256     assert(ptcl.size() % PS::Comm::getNumberOfProc() == 0);
257     const PS::S32 numPtclLocal = ptcl.size() / PS::Comm::getNumberOfProc();
258     sph_system.setNumberOfParticleLocal(numPtclLocal);
259     const PS::U32 i_head = numPtclLocal * PS::Comm::getRank();
260     const PS::U32 i_tail = numPtclLocal * (PS::Comm::getRank() + 1);
261     for(PS::U32 i = 0 ; i < ptcl.size() ; ++ i){
262        if(i_head <= i && i < i_tail){
263           const PS::U32 ii = i - numPtclLocal * PS::Comm::getRank();
264           sph_system[ii] = ptcl[i];
265        }
266     }
267     // Set the end time
268     *end_time = 0.12;
269     // Fin.
270     std::cout << "setup..." << std::endl;
271 }
272
273 void Initialize(PS::ParticleSystem<FP>& sph_system){
274     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
275        sph_system[i].setPressure();
276     }
277 }
278
279 PS::F64 getTimeStepGlobal(const PS::ParticleSystem<FP>& sph_system){
280     PS::F64 dt = 1.0e+30; //set VERY LARGE VALUE
281     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
282        dt = std::min(dt, sph_system[i].dt);
283     }
284     return PS::Comm::getMinValue(dt);
285 }
286
287 void InitialKick(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
288     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
289        sph_system[i].vel_half = sph_system[i].vel + 0.5 * dt * sph_system[i
               ].acc;
290        sph_system[i].eng_half = sph_system[i].eng + 0.5 * dt * sph_system[i
               ].eng_dot;
291     }
292 }
293
294 void FullDrift(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
295     // time becomes t + dt;
296     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
297        sph_system[i].pos += dt * sph_system[i].vel_half;
298     }
299 }
300
301 void Predict(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
302     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
303        sph_system[i].vel += dt * sph_system[i].acc;
304        sph_system[i].eng += dt * sph_system[i].eng_dot;
305     }
306 }
```

```
307
308 void FinalKick(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
309    for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
310       sph_system[i].vel = sph_system[i].vel_half + 0.5 * dt * sph_system[i
              ].acc;
311       sph_system[i].eng = sph_system[i].eng_half + 0.5 * dt * sph_system[i
              ].eng_dot;
312    }
313 }
314
315 void setPressure(PS::ParticleSystem<FP>& sph_system){
316    for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
317       sph_system[i].setPressure();
318    }
319 }
320
321 void CheckConservativeVariables(const PS::ParticleSystem<FP>& sph_system){
322    PS::F64vec Mom=0.0; // total momentum
323    PS::F64    Eng=0.0; // total enegry
324    for(PS::S32 i = 0; i < sph_system.getNumberOfParticleLocal(); ++ i){
325       Mom += sph_system[i].vel * sph_system[i].mass;
326       Eng += (sph_system[i].eng + 0.5 * sph_system[i].vel * sph_system[i].
              vel)
327              * sph_system[i].mass;
328    }
329    Eng = PS::Comm::getSum(Eng);
330    Mom = PS::Comm::getSum(Mom);
331     if(PS::Comm::getRank() == 0){
332         printf("%.16e\n", Eng);
333         printf("%.16e\n", Mom.x);
334         printf("%.16e\n", Mom.y);
335         printf("%.16e\n", Mom.z);
336     }
337 }
338
339 int main(int argc, char* argv[]){
340    // Initialize FDPS
341    PS::Initialize(argc, argv);
342    // Display # of MPI processes and threads
343    PS::S32 nprocs = PS::Comm::getNumberOfProc();
344    PS::S32 nthrds = PS::Comm::getNumberOfThread();
345    std::cout << "========================================" << std::endl
346              << "␣This␣is␣a␣sample␣program␣of␣"           << std::endl
347              << "␣Smoothed␣Particle␣Hydrodynamics␣on␣FDPS!" << std::endl
348              << "␣#␣of␣processes␣is␣" << nprocs            << std::endl
349              << "␣#␣of␣thread␣is␣␣␣␣" << nthrds            << std::endl
350              << "========================================" << std::endl
                       ;
351    // Make an instance of ParticleSystem and initialize it
352    PS::ParticleSystem<FP> sph_system;
353    sph_system.initialize();
354    // Define local variables
355    PS::F64 dt, end_time;
356    boundary box;
357    // Make an initial condition and initialize the particle system
```

```
358     SetupIC(sph_system, &end_time, &box);
359     Initialize(sph_system);
360     // Make an instance of DomainInfo and initialize it
361     PS::DomainInfo dinfo;
362     dinfo.initialize();
363     // Set the boundary condition
364     dinfo.setBoundaryCondition(PS::BOUNDARY_CONDITION_PERIODIC_XYZ);
365     dinfo.setPosRootDomain(PS::F64vec(0.0, 0.0, 0.0),
366                            PS::F64vec(box.x, box.y, box.z));
367     // Perform domain decomposition
368     dinfo.decomposeDomainAll(sph_system);
369     // Exchange the SPH particles between the (MPI) processes
370     sph_system.exchangeParticle(dinfo);
371     // Make two tree structures
372     // (one is for the density calculation and
373     //  another is for the force calculation.)
374     PS::TreeForForceShort<Dens, EP, EP>::Gather dens_tree;
375     dens_tree.initialize(3 * sph_system.getNumberOfParticleGlobal());
376
377     PS::TreeForForceShort<Hydro, EP, EP>::Symmetry hydr_tree;
378     hydr_tree.initialize(3 * sph_system.getNumberOfParticleGlobal());
379     // Compute density, pressure, acceleration due to pressure gradient
380     dens_tree.calcForceAllAndWriteBack(CalcDensity(), sph_system, dinfo);
381     setPressure(sph_system);
382     hydr_tree.calcForceAllAndWriteBack(CalcHydroForce(), sph_system, dinfo)
            ;
383     // Get timestep
384     dt = getTimeStepGlobal(sph_system);
385     // Main loop for time integration
386     PS::S32 step = 0;
387     for(PS::F64 time = 0 ; time < end_time ; time += dt, ++ step){
388        // Leap frog: Initial Kick & Full Drift
389        InitialKick(sph_system, dt);
390        FullDrift(sph_system, dt);
391        // Adjust the positions of the SPH particles that run over
392        // the computational boundaries.
393        sph_system.adjustPositionIntoRootDomain(dinfo);
394        // Leap frog: Predict
395        Predict(sph_system, dt);
396        // Perform domain decomposition again
397        dinfo.decomposeDomainAll(sph_system);
398        // Exchange the SPH particles between the (MPI) processes
399        sph_system.exchangeParticle(dinfo);
400        // Compute density, pressure, acceleration due to pressure gradient
401        dens_tree.calcForceAllAndWriteBack(CalcDensity(), sph_system, dinfo)
               ;
402        setPressure(sph_system);
403        hydr_tree.calcForceAllAndWriteBack(CalcHydroForce(), sph_system,
               dinfo);
404        // Get a new timestep
405        dt = getTimeStepGlobal(sph_system);
406        // Leap frog: Final Kick
407        FinalKick(sph_system, dt);
408        // Output result files
409        if(step % OUTPUT_INTERVAL == 0){
```

```
410             FileHeader header;
411             header.time = time;
412             header.Nbody = sph_system.getNumberOfParticleGlobal();
413             char filename[256];
414             sprintf(filename, "result/%04d.txt", step);
415             sph_system.writeParticleAscii(filename, header);
416             if (PS::Comm::getRank() == 0){
417                 std::cout << "================================" << std::endl;
418                 std::cout << "output␣" << filename << "." << std::endl;
419                 std::cout << "================================" << std::endl;
420             }
421         }
422         // Output information to STDOUT
423         if (PS::Comm::getRank() == 0){
424             std::cout << "================================" << std::endl;
425             std::cout << "time␣=␣" << time << std::endl;
426             std::cout << "step␣=␣" << step << std::endl;
427             std::cout << "================================" << std::endl;
428         }
429         CheckConservativeVariables(sph_system);
430     }
431     // Finalize FDPS
432     PS::Finalize();
433     return 0;
434 }
```

5.2　N 体シミュレーション

N 体シミュレーションのサンプルコードを以下に示す。このサンプルは第 3, 4 節で用いた N 体シミュレーションのサンプルコードと同じものである。これをカット&ペーストしてコンパイルすれば、正常に動作する N 体シミュレーションコードを作ることができる。

Listing 30: N 体シミュレーションのサンプルコード (user-defined.hpp)

```
1  #pragma once
2  class FileHeader{
3  public:
4      PS::S64 n_body;
5      PS::F64 time;
6      PS::S32 readAscii(FILE * fp) {
7          fscanf(fp, "%lf\n", &time);
8          fscanf(fp, "%lld\n", &n_body);
9          return n_body;
10     }
11     void writeAscii(FILE* fp) const {
12         fprintf(fp, "%e\n", time);
13         fprintf(fp, "%lld\n", n_body);
14     }
15 };
16
17 class FPGrav{
18 public:
19     PS::S64    id;
```

```
20      PS::F64    mass;
21      PS::F64vec pos;
22      PS::F64vec vel;
23      PS::F64vec acc;
24      PS::F64    pot;
25
26      static PS::F64 eps;
27
28      PS::F64vec getPos() const {
29          return pos;
30      }
31
32      PS::F64 getCharge() const {
33          return mass;
34      }
35
36      void copyFromFP(const FPGrav & fp){
37          mass = fp.mass;
38          pos  = fp.pos;
39      }
40
41      void copyFromForce(const FPGrav & force) {
42          acc = force.acc;
43          pot = force.pot;
44      }
45
46      void clear() {
47          acc = 0.0;
48          pot = 0.0;
49      }
50
51      void writeAscii(FILE* fp) const {
52          fprintf(fp, "%lld\t%g\t%g\t%g\t%g\t%g\t%g\t%g\n",
53                  this->id, this->mass,
54                  this->pos.x, this->pos.y, this->pos.z,
55                  this->vel.x, this->vel.y, this->vel.z);
56      }
57
58      void readAscii(FILE* fp) {
59          fscanf(fp, "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\n",
60                 &this->id, &this->mass,
61                 &this->pos.x, &this->pos.y, &this->pos.z,
62                 &this->vel.x, &this->vel.y, &this->vel.z);
63          }
64
65  };
66
67
68  #ifdef ENABLE_PHANTOM_GRAPE_X86
69
70
71  template <class TParticleJ>
72  void CalcGravity(const FPGrav * iptcl,
73                   const PS::S32 ni,
74                   const TParticleJ * jptcl,
```

```
75                  const PS::S32 nj,
76                  FPGrav * force) {
77      const PS::S32 nipipe = ni;
78      const PS::S32 njpipe = nj;
79      PS::F64 (*xi)[3] = (PS::F64 (*)[3])malloc(sizeof(PS::F64) * nipipe *
              PS::DIMENSION);
80      PS::F64 (*ai)[3] = (PS::F64 (*)[3])malloc(sizeof(PS::F64) * nipipe *
              PS::DIMENSION);
81      PS::F64  *pi     = (PS::F64  *   )malloc(sizeof(PS::F64) * nipipe);
82      PS::F64 (*xj)[3] = (PS::F64 (*)[3])malloc(sizeof(PS::F64) * njpipe *
              PS::DIMENSION);
83      PS::F64  *mj     = (PS::F64  *   )malloc(sizeof(PS::F64) * njpipe);
84      for(PS::S32 i = 0; i < ni; i++) {
85          xi[i][0] = iptcl[i].getPos()[0];
86          xi[i][1] = iptcl[i].getPos()[1];
87          xi[i][2] = iptcl[i].getPos()[2];
88          ai[i][0] = 0.0;
89          ai[i][1] = 0.0;
90          ai[i][2] = 0.0;
91          pi[i]    = 0.0;
92      }
93      for(PS::S32 j = 0; j < nj; j++) {
94          xj[j][0] = jptcl[j].getPos()[0];
95          xj[j][1] = jptcl[j].getPos()[1];
96          xj[j][2] = jptcl[j].getPos()[2];
97          mj[j]    = jptcl[j].getCharge();
98          xj[j][0] = jptcl[j].pos[0];
99          xj[j][1] = jptcl[j].pos[1];
100         xj[j][2] = jptcl[j].pos[2];
101         mj[j]    = jptcl[j].mass;
102     }
103     PS::S32 devid = PS::Comm::getThreadNum();
104     g5_set_xmjMC(devid, 0, nj, xj, mj);
105     g5_set_nMC(devid, nj);
106     g5_calculate_force_on_xMC(devid, xi, ai, pi, ni);
107     for(PS::S32 i = 0; i < ni; i++) {
108         force[i].acc[0] += ai[i][0];
109         force[i].acc[1] += ai[i][1];
110         force[i].acc[2] += ai[i][2];
111         force[i].pot    -= pi[i];
112     }
113     free(xi);
114     free(ai);
115     free(pi);
116     free(xj);
117     free(mj);
118 }
119
120 #else
121
122 template <class TParticleJ>
123 void CalcGravity(const FPGrav * ep_i,
124                  const PS::S32 n_ip,
125                  const TParticleJ * ep_j,
126                  const PS::S32 n_jp,
```

```
127                     FPGrav * force) {
128     PS::F64 eps2 = FPGrav::eps * FPGrav::eps;
129     for(PS::S32 i = 0; i < n_ip; i++){
130         PS::F64vec xi = ep_i[i].getPos();
131         PS::F64vec ai = 0.0;
132         PS::F64 poti = 0.0;
133         for(PS::S32 j = 0; j < n_jp; j++){
134             PS::F64vec rij    = xi - ep_j[j].getPos();
135             PS::F64    r3_inv = rij * rij + eps2;
136             PS::F64    r_inv  = 1.0/sqrt(r3_inv);
137             r3_inv  = r_inv * r_inv;
138             r_inv  *= ep_j[j].getCharge();
139             r3_inv *= r_inv;
140             ai     -= r3_inv * rij;
141             poti   -= r_inv;
142         }
143         force[i].acc += ai;
144         force[i].pot += poti;
145     }
146 }
147
148 #endif
```

Listing 31: N 体シミュレーションのサンプルコード (nbody.cpp)

```
 1 #include<iostream>
 2 #include<fstream>
 3 #include<unistd.h>
 4 #include<sys/stat.h>
 5 #include<particle_simulator.hpp>
 6 #ifdef ENABLE_PHANTOM_GRAPE_X86
 7 #include <gp5util.h>
 8 #endif
 9 #ifdef ENABLE_GPU_CUDA
10 #define MULTI_WALK
11 #include"force_gpu_cuda.hpp"
12 #endif
13 #include "user-defined.hpp"
14
15 void makeColdUniformSphere(const PS::F64 mass_glb,
16                            const PS::S64 n_glb,
17                            const PS::S64 n_loc,
18                            PS::F64 *& mass,
19                            PS::F64vec *& pos,
20                            PS::F64vec *& vel,
21                            const PS::F64 eng = -0.25,
22                            const PS::S32 seed = 0) {
23
24     assert(eng < 0.0);
25     {
26         PS::MTTS mt;
27         mt.init_genrand(0);
28         for(PS::S32 i = 0; i < n_loc; i++){
29             mass[i] = mass_glb / n_glb;
30             const PS::F64 radius = 3.0;
31             do {
```

```
32                  pos[i][0] = (2. * mt.genrand_res53() - 1.) * radius;
33                  pos[i][1] = (2. * mt.genrand_res53() - 1.) * radius;
34                  pos[i][2] = (2. * mt.genrand_res53() - 1.) * radius;
35              }while(pos[i] * pos[i] >= radius * radius);
36              vel[i][0] = 0.0;
37              vel[i][1] = 0.0;
38              vel[i][2] = 0.0;
39          }
40      }
41
42      PS::F64vec cm_pos  = 0.0;
43      PS::F64vec cm_vel  = 0.0;
44      PS::F64    cm_mass = 0.0;
45      for(PS::S32 i = 0; i < n_loc; i++){
46          cm_pos  += mass[i] * pos[i];
47          cm_vel  += mass[i] * vel[i];
48          cm_mass += mass[i];
49      }
50      cm_pos /= cm_mass;
51      cm_vel /= cm_mass;
52      for(PS::S32 i = 0; i < n_loc; i++){
53          pos[i] -= cm_pos;
54          vel[i] -= cm_vel;
55      }
56  }
57
58  template<class Tpsys>
59  void setParticlesColdUniformSphere(Tpsys & psys,
60                                     const PS::S32 n_glb,
61                                     PS::S32 & n_loc) {
62
63      n_loc = n_glb;
64      psys.setNumberOfParticleLocal(n_loc);
65
66      PS::F64    * mass = new PS::F64[n_loc];
67      PS::F64vec * pos  = new PS::F64vec[n_loc];
68      PS::F64vec * vel  = new PS::F64vec[n_loc];
69      const PS::F64 m_tot = 1.0;
70      const PS::F64 eng   = -0.25;
71      makeColdUniformSphere(m_tot, n_glb, n_loc, mass, pos, vel, eng);
72      for(PS::S32 i = 0; i < n_loc; i++){
73          psys[i].mass = mass[i];
74          psys[i].pos  = pos[i];
75          psys[i].vel  = vel[i];
76          psys[i].id   = i;
77      }
78      delete [] mass;
79      delete [] pos;
80      delete [] vel;
81  }
82
83  template<class Tpsys>
84  void kick(Tpsys & system,
85            const PS::F64 dt) {
86      PS::S32 n = system.getNumberOfParticleLocal();
```

```
87      for(PS::S32 i = 0; i < n; i++) {
88          system[i].vel  += system[i].acc * dt;
89      }
90  }
91
92  template<class Tpsys>
93  void drift(Tpsys & system,
94             const PS::F64 dt) {
95      PS::S32 n = system.getNumberOfParticleLocal();
96      for(PS::S32 i = 0; i < n; i++) {
97          system[i].pos  += system[i].vel * dt;
98      }
99  }
100
101 template<class Tpsys>
102 void calcEnergy(const Tpsys & system,
103                 PS::F64 & etot,
104                 PS::F64 & ekin,
105                 PS::F64 & epot,
106                 const bool clear=true){
107     if(clear){
108         etot = ekin = epot = 0.0;
109     }
110     PS::F64 etot_loc = 0.0;
111     PS::F64 ekin_loc = 0.0;
112     PS::F64 epot_loc = 0.0;
113     const PS::S32 nbody = system.getNumberOfParticleLocal();
114     for(PS::S32 i = 0; i < nbody; i++){
115         ekin_loc += system[i].mass * system[i].vel * system[i].vel;
116         epot_loc += system[i].mass * (system[i].pot + system[i].mass /
                FPGrav::eps);
117     }
118     ekin_loc *= 0.5;
119     epot_loc *= 0.5;
120     etot_loc  = ekin_loc + epot_loc;
121 #ifdef PARTICLE_SIMULATOR_MPI_PARALLEL
122     etot = PS::Comm::getSum(etot_loc);
123     epot = PS::Comm::getSum(epot_loc);
124     ekin = PS::Comm::getSum(ekin_loc);
125 #else
126     etot = etot_loc;
127     epot = epot_loc;
128     ekin = ekin_loc;
129 #endif
130 }
131
132 void printHelp() {
133     std::cerr<<"o: dir name of output (default: ./result)"<<std::endl;
134     std::cerr<<"t: theta (default: 0.5)"<<std::endl;
135     std::cerr<<"T: time_end (default: 10.0)"<<std::endl;
136     std::cerr<<"s: time_step (default: 1.0 / 128.0)"<<std::endl;
137     std::cerr<<"d: dt_diag (default: 1.0 / 8.0)"<<std::endl;
138     std::cerr<<"D: dt_snap (default: 1.0)"<<std::endl;
139     std::cerr<<"l: n_leaf_limit (default: 8)"<<std::endl;
140     std::cerr<<"n: n_group_limit (default: 64)"<<std::endl;
```

```
141     std::cerr<<"N: n_tot (default: 1024)"<<std::endl;
142     std::cerr<<"h: help"<<std::endl;
143 }
144
145 void makeOutputDirectory(char * dir_name) {
146     struct stat st;
147     if(stat(dir_name, &st) != 0) {
148         PS::S32 ret_loc = 0;
149         PS::S32 ret     = 0;
150         if(PS::Comm::getRank() == 0)
151             ret_loc = mkdir(dir_name, 0777);
152         PS::Comm::broadcast(&ret_loc, ret);
153         if(ret == 0) {
154             if(PS::Comm::getRank() == 0)
155                 fprintf(stderr, "Directory \"%s\" is successfully made.\n"
                            , dir_name);
156         } else {
157             fprintf(stderr, "Directory %s fails to be made.\n", dir_name);
158             PS::Abort();
159         }
160     }
161 }
162
163 PS::F64 FPGrav::eps = 1.0/32.0;
164
165 int main(int argc, char *argv[]) {
166     std::cout<<std::setprecision(15);
167     std::cerr<<std::setprecision(15);
168
169     PS::Initialize(argc, argv);
170     PS::F32 theta = 0.5;
171     PS::S32 n_leaf_limit = 8;
172     PS::S32 n_group_limit = 64;
173     PS::F32 time_end = 10.0;
174     PS::F32 dt = 1.0 / 128.0;
175     PS::F32 dt_diag = 1.0 / 8.0;
176     PS::F32 dt_snap = 1.0;
177     char dir_name[1024];
178     PS::S64 n_tot = 1024;
179     PS::S32 c;
180     sprintf(dir_name,"./result");
181     opterr = 0;
182     while((c=getopt(argc,argv,"i:o:d:D:t:T:l:n:N:hs:")) != -1){
183         switch(c){
184         case 'o':
185             sprintf(dir_name,optarg);
186             break;
187         case 't':
188             theta = atof(optarg);
189             std::cerr << "theta =" << theta << std::endl;
190             break;
191         case 'T':
192             time_end = atof(optarg);
193             std::cerr << "time_end = " << time_end << std::endl;
194             break;
```

```
195         case 's':
196             dt = atof(optarg);
197             std::cerr << "time_step = " << dt << std::endl;
198             break;
199         case 'd':
200             dt_diag = atof(optarg);
201             std::cerr << "dt_diag = " << dt_diag << std::endl;
202             break;
203         case 'D':
204             dt_snap = atof(optarg);
205             std::cerr << "dt_snap = " << dt_snap << std::endl;
206             break;
207         case 'l':
208             n_leaf_limit = atoi(optarg);
209             std::cerr << "n_leaf_limit = " << n_leaf_limit << std::endl;
210             break;
211         case 'n':
212             n_group_limit = atoi(optarg);
213             std::cerr << "n_group_limit = " << n_group_limit << std::endl;
214             break;
215         case 'N':
216             n_tot = atoi(optarg);
217             std::cerr << "n_tot = " << n_tot << std::endl;
218             break;
219         case 'h':
220             if(PS::Comm::getRank() == 0) {
221                 printHelp();
222             }
223             PS::Finalize();
224             return 0;
225         default:
226             if(PS::Comm::getRank() == 0) {
227                 std::cerr<<"No such option! Available options are here."<<
                        std::endl;
228                 printHelp();
229             }
230             PS::Abort();
231         }
232     }
233
234     makeOutputDirectory(dir_name);
235
236     std::ofstream fout_eng;
237
238     if(PS::Comm::getRank() == 0) {
239         char sout_de[1024];
240         sprintf(sout_de, "%s/t-de.dat", dir_name);
241         fout_eng.open(sout_de);
242         fprintf(stdout, "This is a sample program of N-body simulation on
                FDPS!\n");
243         fprintf(stdout, "Number of processes: %d\n", PS::Comm::
                getNumberOfProc());
244         fprintf(stdout, "Number of threads per process: %d\n", PS::Comm::
                getNumberOfThread());
245     }
```

```
246
247     PS::ParticleSystem<FPGrav> system_grav;
248     system_grav.initialize();
249     PS::S32 n_loc    = 0;
250     PS::F32 time_sys = 0.0;
251     if(PS::Comm::getRank() == 0) {
252         setParticlesColdUniformSphere(system_grav, n_tot, n_loc);
253     } else {
254         system_grav.setNumberOfParticleLocal(n_loc);
255     }
256
257     const PS::F32 coef_ema = 0.3;
258     PS::DomainInfo dinfo;
259     dinfo.initialize(coef_ema);
260     dinfo.decomposeDomainAll(system_grav);
261     system_grav.exchangeParticle(dinfo);
262     n_loc = system_grav.getNumberOfParticleLocal();
263
264 #ifdef ENABLE_PHANTOM_GRAPE_X86
265     g5_open();
266     g5_set_eps_to_all(FPGrav::eps);
267 #endif
268
269     PS::TreeForForceLong<FPGrav, FPGrav, FPGrav>::Monopole tree_grav;
270     tree_grav.initialize(n_tot, theta, n_leaf_limit, n_group_limit);
271 #ifdef MULTI_WALK
272     const PS::S32 n_walk_limit = 200;
273     const PS::S32 tag_max = 1;
274     tree_grav.calcForceAllAndWriteBackMultiWalk(DispatchKernelWithSP,
275                                                 RetrieveKernel,
276                                                 tag_max,
277                                                 system_grav,
278                                                 dinfo,
279                                                 n_walk_limit);
280 #else
281     tree_grav.calcForceAllAndWriteBack(CalcGravity<FPGrav>,
282                                        CalcGravity<PS::SPJMonopole>,
283                                        system_grav,
284                                        dinfo);
285 #endif
286     PS::F64 Epot0, Ekin0, Etot0, Epot1, Ekin1, Etot1;
287     calcEnergy(system_grav, Etot0, Ekin0, Epot0);
288     PS::F64 time_diag = 0.0;
289     PS::F64 time_snap = 0.0;
290     PS::S64 n_loop = 0;
291     PS::S32 id_snap = 0;
292     while(time_sys < time_end){
293         if( (time_sys >= time_snap) || ( (time_sys + dt) - time_snap ) > (
                time_snap - time_sys) ){
294             char filename[256];
295             sprintf(filename, "%s/%04d.dat", dir_name, id_snap++);
296             FileHeader header;
297             header.time   = time_sys;
298             header.n_body = system_grav.getNumberOfParticleGlobal();
299             system_grav.writeParticleAscii(filename, header);
```

```
300             time_snap += dt_snap;
301         }
302
303         calcEnergy(system_grav, Etot1, Ekin1, Epot1);
304
305         if(PS::Comm::getRank() == 0){
306             if( (time_sys >= time_diag) || ( (time_sys + dt) - time_diag )
                        > (time_diag - time_sys) ){
307                 fout_eng << time_sys << "␣␣␣" << (Etot1 - Etot0) / Etot0
                            << std::endl;
308                 fprintf(stdout, "time:␣%10.7f␣energy␣error:␣%+e\n",
309                         time_sys, (Etot1 - Etot0) / Etot0);
310                 time_diag += dt_diag;
311             }
312         }
313
314
315         kick(system_grav, dt * 0.5);
316
317         time_sys += dt;
318         drift(system_grav, dt);
319
320         if(n_loop % 4 == 0){
321             dinfo.decomposeDomainAll(system_grav);
322         }
323
324         system_grav.exchangeParticle(dinfo);
325 #ifdef MULTI_WALK
326         tree_grav.calcForceAllAndWriteBackMultiWalk(DispatchKernelWithSP,
327                                                     RetrieveKernel,
328                                                     tag_max,
329                                                     system_grav,
330                                                     dinfo,
331                                                     n_walk_limit,
332                                                     true);
333 #else
334         tree_grav.calcForceAllAndWriteBack(CalcGravity<FPGrav>,
335                                            CalcGravity<PS::SPJMonopole>,
336                                            system_grav,
337                                            dinfo);
338 #endif
339
340         kick(system_grav, dt * 0.5);
341
342         n_loop++;
343     }
344
345 #ifdef ENABLE_PHANTOM_GRAPE_X86
346     g5_close();
347 #endif
348
349     PS::Finalize();
350     return 0;
351 }
```

6 拡張機能の解説

6.1 P^3M コード

本節では、FDPS の拡張機能 Particle Mesh (以下、PM と省略する) の使用方法について、P^3M(Particle-Particle-Particle-Mesh) 法のサンプルコードを用いて解説を行う。このサンプルコードでは、塩化ナトリウム (NaCl) 結晶の系全体の結晶エネルギーを P^3M 法で計算し、結果を解析解と比較する。P^3M 法では、力、及び、ポテンシャルエネルギーの計算を、Particle-Particle(PP) パートと Particle-Mesh(PM) パートに split して行われる。このサンプルコードでは PP パートを FDPS 標準機能を用いて計算し、PM パートを FDPS 拡張機能を用いて計算する。なお、拡張機能 PM の仕様の詳細は、仕様書 9.2 節で説明されているので、そちらも参照されたい。

6.1.1 サンプルコードの場所と作業ディレクトリ

サンプルコードの場所は、`$(FDPS)/sample/c++/p3m` である。まずは、そこに移動する。このディレクトリにある `main.cpp` がサンプルコードである。

```
$ cd (FDPS)/sample/c++/p3m
```

6.1.2 ヘッダファイルのインクルード

拡張機能 PM を FDPS 標準機能とともに使用するため、`particle_simulator.hpp` に加え、`particle_mesh.hpp` と `param_fdps.h` をインクルードする。これに加え、このサンプルコードでは拡張機能の非公開定数 `CUTOFF_RADIUS` を参照するため、`param.h` もインクルードしている。

Listing 32: Include FDPS

```
1 #include <particle_simulator.hpp>
2 #include <particle_mesh.hpp>
3 #include <param.h>
4 #include <param_fdps.h>
```

6.1.3 ユーザー定義クラス

本節では、FDPS の機能を用いて P^3M 法の計算を行うにあたって、ユーザーが記述しなければならないクラスについて記述する。

6.1.3.1　FullParticle 型

ユーザーは FullParticle 型を記述しなければならない。Listing 33 に、サンプルコードの FullParticle 型を示す。FullParticle 型には、計算を行うにあたって、粒子が持っているべき全ての物理量が含まれている必要がある。また、以下のメンバ関数を持たせる必要がある:

- `getCharge()` — FDPS が粒子の電荷量を取得するのに必要
- `getChargeParticleMesh()` — FDPS の PM モジュールが粒子の電荷量を取得するために必要
- `getPos()` — FDPS が粒子座標を取得するのに必要
- `getRSearch()` — FDPS がカットオフ半径を取得するのに必要
- `setPos()` — FDPS が粒子の座標を書き込むのに必要
- `copyFromForce()` — Force 型から結果をコピーするのに必要なメンバ関数
- `copyFromForceParticleMesh()` — PM モジュールが力の計算結果を書き込むために必要

なお、このサンプルコードでは `copyFromForce()` と `copyFromForceParticleMesh()` が空関数になっている。これはサンプルコードでは後述する API `getForce()` 等を用いて、明示的に結果を Force 型から FullParticle 型にコピーする実装になっているためである。

Listing 33: FullParticle 型

```
class Nbody_FP
{
   public:
      PS::S64 id;
      PS::F64 m;
      PS::F64 rc;
      PS::F64vec x;
      PS::F64vec v,v_half;
      PS::F64vec agrv;
      PS::F64 pot;
      // Member functions required by FDPS
      PS::F64 getCharge() const {
         return m;
      };
      PS::F64 getChargeParticleMesh() const {
         return m;
      };
      PS::F64vec getPos() const {
         return x;
      };
      PS::F64 getRSearch() const {
         return rc;
      };
      void setPos(const PS::F64vec& x) {
         this->x = x;
      };
      void copyFromForce(const Nbody_PP_Results& result) {};
      void copyFromForceParticleMesh(const PS::F64 apm) {};
};
```

6.1.3.2　EssentialParticleI 型

ユーザーは EssentialParticleI 型を記述しなければならない。EssentialParticleI 型には、PP パートの Force 計算を行う際、i 粒子が持っているべき全ての物理量をメンバ変数として持っている必要がある。また、本チュートリアル中では、EssentialParticleJ 型も兼ねているため、j 粒子が持っているべき全ての物理量もメンバ変数として持っている必要がある。Listing 34 に、サンプルコードの EssentialParticleI 型を示す。この EssentialParticleI 型には前述した FullParticle 型から、値をコピーするのに必要なメンバ関数 `copyFromFP()` を持つ必要がある。その他、粒子の電荷量を返す関数である `getCharge()`、粒子座標を返す関数である `getPos()`、粒子のカットオフ半径を返す関数である `getRSearch()`、粒子の座標を書き込む関数である `setPos()` が必要になる。

Listing 34: EssentialParticleI 型

```
1  class Nbody_EP
2  {
3     public:
4        PS::S64 id;
5        PS::F64 m;
6        PS::F64 rc;
7        PS::F64vec x;
8        // Member functions required by FDPS
9        PS::F64 getCharge() const {
10          return m;
11       };
12       PS::F64vec getPos() const {
13          return x;
14       };
15       PS::F64 getRSearch() const {
16          return rc;
17       };
18       void setPos(const PS::F64vec& x) {
19          this->x = x;
20       };
21       void copyFromFP(const Nbody_FP& FP) {
22          id  = FP.id;
23          m   = FP.m;
24          rc  = FP.rc;
25          x   = FP.x;
26       };
27 };
```

6.1.3.3　Force 型

ユーザーは Force 型を記述しなければならない。Force 型は、PP パートの Force の計算を行った際にその結果として得られる全ての物理量をメンバ変数として持っている必要がある。本サンプルコードの Force 型を Listing 35 に示す。このサンプルコードでは、Force は Coulomb 相互作用計算のみであるため、Force 型が 1 つ用意されている。また、積算対象のメンバ変数を 0 ないし初期値に設定するための関数 `clear()` が必要になる。

Listing 35: Force 型

```
class Nbody_PP_Results
{
   public:
      PS::F64 pot;
      PS::F64vec agrv;
      void clear() {
         pot  = 0.0;
         agrv = 0.0;
      }
};
```

6.1.3.4　calcForceEpEp 型

ユーザーは calcForceEpEp 型を記述しなければならない。calcForceEpEp 型には、PP パートの Force の計算の具体的な内容を書く必要がある。サンプルコードの calcForceEpEp 型を Listing 36 に示す。このサンプルコードでは、ファンクタ (関数オブジェクト) を用いて実装している。ファンクタの引数は、EssentialParticleI の配列、EssentialParticleI の個数、EssentialParticleJ の配列、EssentialParticleJ の個数、Force 型の配列である。

Listing 36: calcForceEpEp 型

```
class Calc_force_ep_ep{
   public:
      void operator () (const Nbody_EP* const ep_i,
                        const PS::S32 Nip,
                        const Nbody_EP* const ep_j,
                        const PS::S32 Njp,
                        Nbody_PP_Results* const result) {
         for (PS::S32 i=0; i<Nip; i++) {
            for (PS::S32 j=0; j<Njp; j++) {
               PS::F64vec dx = ep_i[i].x - ep_j[j].x;
               PS::F64 rij = std::sqrt(dx * dx);
               if ((ep_i[i].id == ep_j[j].id) && (rij == 0.0)) continue;
               PS::F64 rinv = 1.0/rij;
               PS::F64 rinv3 = rinv*rinv*rinv;
               PS::F64 xi = 2.0*rij/ep_i[i].rc;
               result[i].pot  += ep_j[j].m * S2_pcut(xi) * rinv;
               result[i].agrv += ep_j[j].m * S2_fcut(xi) * rinv3 * dx;
            }
            //* Self-interaction term
            result[i].pot -= ep_i[i].m * (208.0/(70.0*ep_i[i].rc));
         }

      };
};
```

P^{3}M 法の PP パートは、(距離に関する) カットオフ付きの 2 体相互作用である。そのため、ポテンシャルと加速度の計算にカットオフ関数 (`S2_pcut()`, `S2_fcut()`) が含まれていることに注意されたい。ここで、各カットオフ関数は、粒子の形状関数 $S(r)$ が $S2(r)$ のときのカットオフ関数である必要がある。ここで、$S2(r)$ は Hockney & Eastwood (1988) の式 (8.3) で

定義される形状関数で、以下の形を持つ:

$$S2(r) = \begin{cases} \dfrac{48}{\pi a^4}\left(\dfrac{a}{2} - r\right) & r < a/2, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

ここで、r は粒子からの距離、a は形状関数のスケール長である。粒子の電荷量を q とすれば、この粒子が作る電荷密度分布 $\rho(r)$ は $\rho(r) = q\,S2(r)$ と表現される。これは r に関して線形な密度分布を仮定していることを意味する。PP パートのカットオフ関数が $S2(r)$ を仮定したものでなければならない理由は、PM パートのカットオフ関数が S2 型形状関数を仮定して実装されているためである (PM パートと PP パートのカットオフ関数は同じ形状関数に基づく必要がある)。

カットオフ関数はユーザが定義する必要がある。サンプルコードの冒頭に `S2_pcut()` と `S2_fcut()` の実装例がある。これらの関数では、Hockney & Eastwood (1988) の式 (8-72),(8-75) が使用されている。カットオフ関数は、PP 相互作用が以下の形となるように定義されている:

$$\Phi_{\rm PP}(\boldsymbol{r}) = \frac{m}{|\boldsymbol{r} - \boldsymbol{r}'|}\texttt{S2\_pcut}(\xi) \tag{2}$$

$$\boldsymbol{f}_{\rm PP}(\boldsymbol{r}) = \frac{m(\boldsymbol{r} - \boldsymbol{r}')}{|\boldsymbol{r} - \boldsymbol{r}'|^3}\texttt{S2\_fcut}(\xi) \tag{3}$$

ここで、$\xi = 2|\boldsymbol{r} - \boldsymbol{r}'|/a$ である。本サンプルコードでは a を変数 `rc` で表現している。

Hockney & Eastwood (1988) の式 (8-75) を見ると、$r = 0$ のとき、メッシュポテンシャル ϕ^m が次の有限値を持つことがわかる (ここで、$1/4\pi\varepsilon_0$ の因子は省略した):

$$\phi^m(0) = \frac{208}{70a} \tag{4}$$

この項はサンプルコードの i 粒子のループの最後で考慮されている:

```
1 result[i].pot -= ep_i[i].m * (208.0/(70.0*ep_i[i].rc));
```

この項を考慮しないと解析解と一致しないことに注意する必要がある。

6.1.4　プログラム本体

本節では、サンプルコード本体について解説を行う。詳細な説明に入る前に、サンプルコードの内容と全体構造について説明を与える。6.1 節で述べたように、このサンプルコードでは NaCl 結晶の結晶エネルギーを $\mathrm{P^3M}$ 法によって計算し解析解と比較する。NaCl 結晶は一様格子状に並んだ粒子として表現される。Na と Cl は互い違いに並んでおり、Na に対応する粒子は正の電荷を、Cl に対応する粒子は負の電荷を持っている。この粒子で表現された結晶を、大きさが $[0, 1]^3$ の周期境界ボックスの中に配置し、結晶エネルギーを計算する。結晶エネルギーの計算精度は周期境界ボックスの中の粒子数や粒子の配置に依存するはずなので、サンプルコードでは、これらを変えてエネルギーの相対誤差を調べ、結果をファイルに出力する内容となっている。

コードの全体構造は以下のようになっている:

(1) FDPS で使用するオブジェクトの生成と初期化
(2) 指定された粒子数と配置の結晶を生成 (メイン関数の NaCl_IC())
(3) 各粒子のポテンシャルエネルギーを P^{3}M 法で計算 (メイン関数の Nbody_objs.calc_gravity())
(4) 系全体のエネルギーを計算し、解析解と比較 (メイン関数の calc_energy_error())
(5) (2)〜(4) を繰り返す

以下で、個々について詳しく説明を行う。

6.1.4.1　開始、終了

まずは、FDPS の初期化/開始を行う必要がある。次のように、メイン関数に記述する。

Listing 37: FDPS の開始

```
1 PS::Initialize(argc, argv);
```

FDPS は、開始したら明示的に終了させる必要がある。今回は、プログラムの終了と同時に FDPS も終了させるため、メイン関数の最後に次のように記述する。

Listing 38: FDPS の終了

```
1 PS::Finalize();
```

6.1.4.2　オブジェクトの生成と初期化

FDPS の初期化に成功した場合、ユーザーはコード中で用いるオブジェクトを作成する必要がある。本節では、オブジェクトの生成/初期化の仕方について、解説する。

6.1.4.2.1　オブジェクトの生成

P^{3}M 法の計算では、粒子群クラス、領域クラスに加え、PP パートの計算用の tree を 1 本、さらに PM パートの計算に必要な ParticleMesh オブジェクトの生成が必要である。サンプルコードでは、これらのオブジェクトを Nbody_Objects クラスにまとめている。以下にそのコードを記す。

Listing 39: Nbody_Objects クラス

```
1 class Nbody_Objects {
2    public:
3       PS::ParticleSystem<Nbody_FP> system;
4       PS::DomainInfo dinfo;
5       PS::TreeForForceLong<Nbody_PP_Results, Nbody_EP, Nbody_EP>::
              MonopoleWithCutoff pp_tree;
6       PS::PM::ParticleMesh pm;
7 }
```

サンプルコードでは、メイン関数のローカル変数として Nbody_Objects オブジェクトを 1 個生成している:

Listing 40: Nbody_Objects クラスのオブジェクト生成

```
1 Nbody_Objects Nbody_objs;
```

6.1.4.2.2　オブジェクトの初期化

ユーザーはオブジェクトを生成したら、そのオブジェクトを使用する前に、初期化を行う必要がある。以下で、各オブジェクトの初期化の仕方について解説を行う。

(i) 粒子群クラスの初期化 粒子群クラスの初期化は、以下のように行う:

Listing 41: 粒子群クラスの初期化

```
1 Nbody_objs.system.initialize();
```

サンプルコードではメイン関数の冒頭で呼び出されている。

(ii) 領域クラスの初期化 領域クラスの初期化は、以下のように行う:

Listing 42: 領域クラスの初期化

```
1 Nbody_objs.dinfo.initialize();
```

サンプルコードではメイン関数の冒頭で呼び出されている。

初期化が完了した後、領域クラスには境界条件と境界の大きさをセットする必要がある。サンプルコードでは、この作業は粒子分布を決定する関数 NaCl_IC() の中で行われている:

```
1 dinfo.setBoundaryCondition(PS::BOUNDARY_CONDITION_PERIODIC_XYZ);
2 dinfo.setPosRootDomain(PS::F64vec(0.0,0.0,0.0),
3                        PS::F64vec(1.0,1.0,1.0));
```

(iii) ツリークラスの初期化 相互作用ツリークラスの初期化も、initialize メソッドを使って、以下のように行う:

Listing 43: ツリークラスの初期化

```
1 void init_tree() {
2    PS::S32 numPtclLobal = system.getNumberOfParticleLobal();
3    PS::U64 ntot = 3 * numPtclLobal;
4    pp_tree.initialize(ntot,0.0);
5 };
```

ツリークラスの initialize メソッドには引数として、大雑把な粒子数を渡す必要がある。これは initialize メソッドの第一引数として指定する。上記の例では、initialize が呼ばれた時点でのローカル粒子数の 3 倍の値がセットされるようになっている。一方、initialize の第 2 引数は省略可能引数で、tree 法で力を計算するときの opening angle criterion θ を指定する。本サンプルコードでは PP パートの計算で tree 構造を使用しないため、$\theta = 0$ を指定している。

本サンプルコードでは、ツリークラスの初期化は Nbody_objs.init_tree() を通して行っている (メイン関数を参照のこと)。

```
if (is_tree_initialized == false) {
   Nbody_objs.init_tree();
   is_tree_initialized = true;
}
```

ここで、初期化はプログラム中で 1 回だけ行う必要があるため、上記のような if 文の処理が必要となる。

(iv) ParticleMesh クラスの初期化 特に明示的に初期化を行う必要はない。

6.1.4.3　粒子分布の生成

本節では、粒子分布を生成する関数 NaCl_IC の動作とその中で呼ばれている FDPS の API について解説を行う。この関数では、周期境界ボックスの 1 次元あたりの粒子数と、原点 $(0, 0, 0)$ に最も近い粒子の座標を引数として、3 次元粒子分布を生成する。サンプルコードでは、これらのパラメータは Crystal_Parameters クラスのオブジェクト NaCl_params を使って渡されている:

```
class Crystal_Parameters
{
   public:
      PS::S32 numPtcl_per_side;
      PS::F64vec pos_vertex;
};
/* In main function */
Crystal_Parameters NaCl_params;
NaCl_IC(Nbody_objs.system,
        Nbody_objs.dinfo,
        NaCl_params);
```

NaCl_IC の前半部分において、渡されたパラメータを使って粒子分布を生成している。この結晶の系全体のエネルギーは

$$E = -\frac{N\alpha m^2}{R_0} \tag{5}$$

と解析的に書ける。ここで、N は分子の総数 (原子の数は $2N$ 個)、m は粒子の電荷量、R_0 は最隣接原子間距離、α はマーデリング (Madelung) 定数である。NaCl 結晶の場合、$\alpha \approx 1.747565$ である (例えば、キッテル著 「固体物理学入門 (第 8 版)」を参照せよ)。計算結果をこの解析解と比較するとき、系全体のエネルギーが粒子数に依存しては不便である。そこで、サンプルコードでは、系全体のエネルギーが粒子数に依存しないように、粒子の電荷量 m を

$$\frac{2Nm^2}{R_0} = 1 \tag{6}$$

となるようにスケールしていることに注意されたい。

粒子分布の生成後、FDPS の API を使って、領域分割と粒子交換を行っている。以下で、これらの API について解説する。

6.1.4.3.1　領域分割の実行

粒子分布に基いて領域分割を実行するには、領域クラスの `decomposeDomainAll` メソッドを使用する:

Listing 44: 領域分割の実行

```
1 dinfo.decomposeDomainAll(system);
```

ここで、粒子分布の情報を領域クラスに与えるため、引数に粒子群クラスのオブジェクトが渡されていることに注意されたい。

6.1.4.3.2　粒子交換の実行

領域情報に基いてプロセス間の粒子の情報を交換するには、粒子群クラスの `exchangeParticle` メソッドを使用する:

Listing 45: 粒子交換の実行

```
1 system.exchangeParticle(dinfo);
```

ここで領域情報を粒子群クラスに与えるため、引数に領域クラスのオブジェクトが渡されていることに注意する。

6.1.4.4　相互作用計算の実行

粒子分布を決定し、領域分割・粒子交換が終了したら、相互作用の計算を行う。サンプルコードでは、この作業をメイン関数で行っている:

Listing 46: 相互作用計算の実行

```
1 Nbody_objs.calc_gravity();
```

`Nbody_objs.calc_gravity()` の中身は以下のようになっており、(i) ポテンシャルエネルギーと加速度の 0 クリア、(ii) PM パートの計算、(iii) PP パートの計算から構成される:

Listing 47: 相互作用計算の中身

```
1  void calc_gravity() {
2     //* Local variables
3     PS::S32 numPtclLocal = system.getNumberOfParticleLocal();
4
5     //* Reset potential and accelerations
6     for (PS::S32 i=0; i<numPtclLocal; i++) {
7        system[i].pot  = 0.0;
8        system[i].agrv = 0.0;
9     }
10
11    //=================
12    //* [1] PM part
13    //=================
14    pm.setDomainInfoParticleMesh(dinfo);
```

```
15     pm.setParticleParticleMesh(system);
16     pm.calcMeshForceOnly();
17     for (PS::S32 i=0; i<numPtclLocal; i++) {
18        PS::F32vec x32 = system[i].x;
19        system[i].pot  -= pm.getPotential(x32);
20        system[i].agrv -= pm.getForce(x32);
21     }
22
23     //=================
24     //* [2] PP part
25     //=================
26     pp_tree.calcForceAll(Calc_force_ep_ep(),
27                          Calc_force_ep_sp(),
28                          system, dinfo);
29     for (PS::S32 i=0; i<numPtclLocal; i++) {
30        Nbody_PP_Results result = pp_tree.getForce(i);
31        system[i].pot  += result.pot;
32        system[i].agrv += result.agrv;
33     }
34 };
```

PM パートの計算部分を以下に示す。PM の Force 計算を行うためには、ParticleMesh クラスのオブジェクト pm が領域情報と粒子情報を事前に知っている必要がある。そのため、サンプルコードでは、まず、setDomainInfoParticleMesh メソッドと setParticleParticleMesh メソッドで、領域情報と粒子情報を pm オブジェクトに渡している。これで Force 計算を行う準備ができたことになる。Force 計算は calcMeshForceOnly メソッドで行う。Force 計算の結果を取得するため、getPotential メソッドと getForce メソッドで粒子の位置でのポテンシャルと加速度を取得し、それを FullParticle 型のオブジェクト system に足し込んでいる。この足し込み演算を-=で行っていることに注意して頂きたい。+=ではなく、-=を使用する理由は、FDPS の拡張機能 PM は重力を想定してポテンシャルを計算するからである。すなわち、拡張機能 PM では、電荷 $m(>0)$ は正のポテンシャルを作るべきところを、質量 $m>0$ の重力ポテンシャルとして計算する。このポテンシャルは負値である。したがって、拡張機能 PM を Coulomb 相互作用で使用するには符号反転が必要となる。

Listing 48: PM パートの計算

```
1 pm.setDomainInfoParticleMesh(dinfo);
2 pm.setParticleParticleMesh(system);
3 pm.calcMeshForceOnly();
4 for (PS::S32 i=0; i<numPtclLocal; i++) {
5    PS::F32vec x32 = system[i].x;
6    system[i].pot  -= pm.getPotential(x32);
7    system[i].agrv -= pm.getForce(x32);
8 }
```

次に、PP パートの計算部分を以下に示す。PP パートの Force 計算はツリークラスの calcForceAll メソッドによって行う (ここで、calcForceAllAndWriteBack メソッドを使用しないのは、これを使うと PM パートの結果が 0 クリアされてしまうためである)。次に、getForce メソッドを使用して、Force 計算で求めた粒子の位置でのポテンシャルと加速度を取得し、FullParticle 型のオブジェクト system に書き込んでいる。

Listing 49: PP パートの計算

```
1 pp_tree.calcForceAll(Calc_gravity(), system, dinfo);
2 for (PS::S32 i=0; i<numPtclLocal; i++) {
3    Nbody_PP_Results result = pp_tree.getForce(i);
4    system[i].pot  += result.pot;
5    system[i].agrv += result.agrv;
6 }
```

6.1.4.5　エネルギー相対誤差の計算

エネルギーの相対誤差の計算は関数 calc_energy_error で行っている。ここでは、解析解の値としては $E_0 \equiv 2E = -1.7475645946332$ を採用した。これは、PM^3(Particle-Mesh Multipole Method) で数値的に求めた値である。

6.1.5　コンパイル

本サンプルコードでは FFTW ライブラリ (http://www.fftw.org) を使用するため、ユーザ環境に FFTW3 をインストールする必要がある。コンパイルは、付属の Makefile 内の変数 FFTW_LOC と FDPS_LOC に、FFTW と FDPS のインストール先の PATH をそれぞれ指定し、make コマンドを実行すればよい。

```
$ make
```

コンパイルがうまく行けば、work ディレクトリに実行ファイル p3m.x が作成されているはずである。

6.1.6　実行

FDPS 拡張機能の仕様から、本サンプルコードはプロセス数が 2 以上の MPI 実行でなければ、正常に動作しない。そこで、以下のコマンドでプログラムを実行する:

```
$ MPIRUN -np NPROC ./p3m.x
```

ここで、“MPIRUN”には mpirun や mpiexec などの mpi 実行プログラムが、“NPROC”にはプロセス数が入る。

6.1.7　結果の確認

計算が終了すると、work フォルダ下にエネルギーの相対誤差を記録したファイルが出力される。結果をプロットすると、図 3 のようになる。

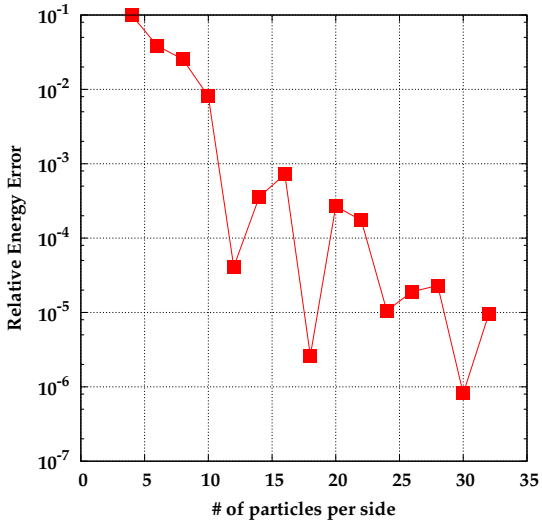

図 3: 1 辺あたりの粒子数とエネルギー相対誤差の関係 (メッシュ数は 16^3、カットオフ半径は 3/16)

6.2　TreePM コード

本節では、FDPS の拡張機能 Particle Mesh (以下、PM と省略する) の使用方法について、TreePM(Tree-Particle-Mesh) 法のサンプルコードを用いて解説を行う。このサンプルコードでは、宇宙論的 N 体シミュレーションを TreePM 法を用いて実行する。TreePM 法は第 6.1 節で説明した P^3M 法と同様、重力計算を PP パートと PM パートに split して行う。したがって、使用する FDPS の機能は P^3M 法のサンプルコードとほぼ同じである。2 つの方法の違いは、TreePM 法では PP パートの計算を直接法 (ダイレクトサム法) ではなく Tree 法を使って計算する点にある。

6.2.1　サンプルコードの場所と作業ディレクトリ

サンプルコードの場所は、`$(FDPS)/sample/c++/treepm/src` である。まずは、そこに移動する。下に示すように、サンプルコードは複数のファイルから構成される。この内、FDPS に関係した部分は主に `treepm.hpp` と `treepm.cpp` に実装されている。以下の説明では、適宜、ファイル名を参照していく。

```
$ cd (FDPS)/sample/c++/treepm/src
$ ls | awk '{print $0}'
IC/
Makefile
constants.hpp
cosmology.hpp
fig/
make_directory.c
prototype.h
result/
run_param.hpp
timing.c
treepm.cpp
treepm.hpp
utils/
```

6.2.2　ヘッダファイルのインクルード

FDPS の標準機能と拡張機能の両方を使うため、メイン関数が定義されたファイル `treepm.cpp` で、`particle_simulator.hpp` と `particle_mesh.hpp` をインクルードしている:

Listing 50: Include FDPS

```
1 #include <particle_simulator.hpp>
2 #include <particle_mesh.hpp>
```

6.2.3　ユーザー定義クラス

FDPS を使用するため、ユーザはユーザ定義クラスを実装しなければならない。本節では、ユーザ定義クラスをサンプルコードでどのように実装しているかを説明する。

6.2.3.1　FullParticle 型

ユーザーは FullParticle 型を記述しなければならない。FullParticle 型には、計算を行うにあたり、粒子が持っているべき全ての物理量が含まれている必要がある。Listing 51 に、サンプルコードの FullParticle 型を示す。このサンプルコードでは通常の N 体計算に必要なメンバ変数 (id, mass, eps, pos, vel, acc) に加え、PM パートの加速度を格納する acc_pm、ハッブル定数を格納する H0、計算領域の大きさを Mpc h^{-1} 単位で格納する Lbnd が用意されている。また、FDPS の標準機能・拡張機能を使うため、以下のメンバ関数を持たせる必要がある:

- getCharge() — FDPS が粒子の質量を取得するのに必要
- getChargeParticleMesh() — FDPS の PM モジュールが粒子の電荷量を取得するために必要
- getPos() — FDPS が粒子座標を取得するのに必要
- getRSearch() — FDPS がカットオフ半径を取得するのに必要
- setPos() — FDPS が粒子の座標を書き込むのに必要
- copyFromForce() — Force 型から結果をコピーするのに必要なメンバ関数
- copyFromForceParticleMesh() — PM モジュールが力の計算結果を書き込むために必要

さらに、FDPS の入出力関数を使用するため、以下のメンバ関数を定義してある:

- readBinary()
- writeBinary()

但し、この 2 つは必須ではなく、ユーザ独自の入出力関数を定義してもよい。

Listing 51: FullParticle 型

```
1  class FPtreepm {
2  private:
3      template<class T>
4      T reverseEndian(T value){
5          char * first = reinterpret_cast<char*>(&value);
6          char * last = first + sizeof(T);
7          std::reverse(first, last);
8          return value;
9      }
10 public:
11     PS::S64    id;
12     PS::F32    mass;
13     PS::F32    eps;
14     PS::F64vec pos;
15     PS::F64vec vel;
```

```
16      PS::F64vec acc;
17      PS::F64vec acc_pm;
18
19      //static PS::F64vec low_boundary;
20      //static PS::F64vec high_boundary;
21      //static PS::F64 unit_l;
22      //static PS::F64 unit_m;
23      static PS::F64 H0;
24      static PS::F64 Lbnd;
25
26      PS::F64vec getPos() const {
27          return pos;
28      }
29
30      PS::F64 getCharge() const {
31          return mass;
32      }
33
34      void copyFromForce(const Result_treepm & force) {
35          this->acc = force.acc;
36      }
37
38      PS::F64 getRSearch() const {
39          PS::F64 rcut = 3.0/SIZE_OF_MESH;
40          return rcut;
41      }
42
43      void setPos(const PS::F64vec pos_new) {
44          this->pos = pos_new;
45      }
46
47      PS::F64 getChargeParticleMesh() const {
48          return this->mass;
49      }
50
51      void copyFromForceParticleMesh(const PS::F64vec & acc_pm) {
52          this->acc_pm = acc_pm;
53      }
54
55      /*
56      void writeParticleBinary(FILE *fp) {
57          int count;
58          count = 0;
59
60          count += fwrite(&mass,   sizeof(PS::F32),1,fp);
61          count += fwrite(&eps,    sizeof(PS::F32),1,fp);
62          count += fwrite(&pos[0], sizeof(PS::F64),1,fp);
63          count += fwrite(&pos[1], sizeof(PS::F64),1,fp);
64          count += fwrite(&pos[2], sizeof(PS::F64),1,fp);
65          count += fwrite(&vel[0], sizeof(PS::F64),1,fp);
66          count += fwrite(&vel[1], sizeof(PS::F64),1,fp);
67          count += fwrite(&vel[2], sizeof(PS::F64),1,fp);
68      }
69      */
70      /*
```

```
71      int readParticleBinary(FILE *fp) {
72          int count;
73          count = 0;
74
75          count += fread(&mass,   sizeof(PS::F32),1,fp);
76          count += fread(&eps,    sizeof(PS::F32),1,fp);
77          count += fread(&pos[0], sizeof(PS::F64),1,fp);
78          count += fread(&pos[1], sizeof(PS::F64),1,fp);
79          count += fread(&pos[2], sizeof(PS::F64),1,fp);
80          count += fread(&vel[0], sizeof(PS::F64),1,fp);
81          count += fread(&vel[1], sizeof(PS::F64),1,fp);
82          count += fread(&vel[2], sizeof(PS::F64),1,fp);
83
84          return count;
85      }
86      */
87
88      void writeParticleBinary(FILE *fp) {
89          PS::F32 x = pos[0];
90          PS::F32 y = pos[1];
91          PS::F32 z = pos[2];
92          PS::F32 vx = vel[0];
93          PS::F32 vy = vel[1];
94          PS::F32 vz = vel[2];
95          PS::S32 i = id;
96          PS::S32 m = mass;
97          fwrite(&x, sizeof(PS::F32),1,fp);
98          fwrite(&vx, sizeof(PS::F32),1,fp);
99          fwrite(&y, sizeof(PS::F32),1,fp);
100         fwrite(&vy, sizeof(PS::F32),1,fp);
101         fwrite(&z, sizeof(PS::F32),1,fp);
102         fwrite(&vz, sizeof(PS::F32),1,fp);
103         //fwrite(&mass,   sizeof(PS::F32),1,fp);
104         fwrite(&m,    sizeof(PS::F32),1,fp);
105         fwrite(&i,    sizeof(PS::F32),1,fp);
106         //fwrite(&id,   sizeof(PS::F32),1,fp);
107     }
108
109
110     // for API of FDPS
111     // in snapshot, L unit is Mpc/h, M unit is Msun, v unit is km/s
112     void readBinary(FILE *fp){
113         static PS::S32 ONE = 1;
114         static bool is_little_endian = *reinterpret_cast<char*>(&ONE) ==
                ONE;
115         static const PS::F64 Mpc_m = 3.08567e22; // unit is m
116         static const PS::F64 Mpc_km = 3.08567e19; // unit is km
117         static const PS::F64 Msun_kg = 1.9884e30; // unit is kg
118         static const PS::F64 G = 6.67428e-11; // m^3*kg^-1*s^-2
119         static const PS::F64 Cl = 1.0 / FPtreepm::Lbnd;
120         static const PS::F64 Cv = 1.0 / (FPtreepm::Lbnd * FPtreepm::H0);
121         static const PS::F64 Cm = 1.0 / (pow(Mpc_m*FPtreepm::Lbnd, 3.0) /
                pow(Mpc_km/FPtreepm::H0, 2.0) / G / Msun_kg);
122         PS::F32 x, y, z, vx, vy, vz, m;
123         PS::S32 i;
```

```
124         fread(&x, 4, 1, fp);
125         fread(&vx, 4, 1, fp);
126         fread(&y, 4, 1, fp);
127         fread(&vy, 4, 1, fp);
128         fread(&z, 4, 1, fp);
129         fread(&vz, 4, 1, fp);
130         fread(&m,  4, 1, fp);
131         fread(&i,  4, 1, fp);
132         if( is_little_endian){
133             pos.x = x * Cl;
134             pos.y = y * Cl;
135             pos.z = z * Cl;
136             vel.x = vx * Cv;
137             vel.y = vy * Cv;
138             vel.z = vz * Cv;
139             mass = m * Cm;
140             //mass = m / 1.524e17;
141             id = i;
142         }
143         else{
144             pos.x = reverseEndian(x) * Cl;
145             pos.y = reverseEndian(y) * Cl;
146             pos.z = reverseEndian(z) * Cl;
147             vel.x = reverseEndian(vx) * Cv;
148             vel.y = reverseEndian(vy) * Cv;
149             vel.z = reverseEndian(vz) * Cv;
150             mass = reverseEndian(m) * Cm;
151             //mass = reverseEndian(m) / 1.524e17;
152             id = reverseEndian(i);
```

6.2.3.2　EssentialParticleI 型

ユーザーは EssentialParticleI 型を記述しなければならない。EssentialParticleI 型には、PP パートの Force 計算を行う際、i 粒子が持っているべき全ての物理量をメンバ変数として持っている必要がある。Listing 52 に、サンプルコードの EssentialParticleI 型を示す。この EssentialParticleI 型には、前述した FullParticle 型から値をコピーするのに必要なメンバ関数 `copyFromFP()` と、EssentialParticleI 型の粒子座標を返すメンバ関数 `getPos()` を持たせる必要がある。

Listing 52: EssentialParticleI 型

```
1     void writeBinary(FILE *fp){
2         static const PS::F64 Mpc_m = 3.08567e22; // unit is m
3         static const PS::F64 Mpc_km = 3.08567e19; // unit is km
4         static const PS::F64 Msun_kg = 1.9884e30; // unit is kg
5         static const PS::F64 G = 6.67428e-11; // m^3*kg^-1*s^-2
6         static const PS::F64 Cl = FPtreepm::Lbnd;
7         static const PS::F64 Cv = (FPtreepm::Lbnd * FPtreepm::H0);
8         static const PS::F64 Cm = (pow(Mpc_m*FPtreepm::Lbnd, 3.0) / pow(
              Mpc_km/FPtreepm::H0, 2.0) / G / Msun_kg);
9         PS::F32vec x = pos * Cl;
10        PS::F32vec v = vel * Cv;
```

```
11          PS::F32 m = mass * Cm;
12          PS::S32 i = id;
13          fwrite(&x.x,    sizeof(PS::F32), 1, fp);
14          fwrite(&v.x,    sizeof(PS::F32), 1, fp);
15          fwrite(&x.y,    sizeof(PS::F32), 1, fp);
16          fwrite(&v.y,    sizeof(PS::F32), 1, fp);
```

6.2.3.3　EssentialParticleJ 型

ユーザーは EssentialParticleJ 型を記述しなければならない。6.1 節の P^3M コードの例では、EssentialParticleJ 型は EssentialParticleI 型で兼ねていたが、このサンプルコードでは別のクラスとして定義してある。EssentialParticleJ 型には、PP パートの Force 計算を行う際、j 粒子が持っているべき全ての物理量をメンバ変数として持っている必要がある。Listing 53 に、サンプルコードの EssentialParticleJ 型を示す。この EssentialParticleJ 型には、以下のメンバ関数を持たせる必要がある:

- `getPos()` — FDPS が粒子位置を取得するのに必要
- `getCharge()` — FDPS が粒子質量を取得するのに必要
- `copyFromFP()` — FDPS が FullParticle 型から EssentialParticleJ 型に必要な情報を渡すのに必要
- `getRSearch()` — FDPS がカットオフ半径を取得するのに必要
- `setPos()` — FDPS が粒子座標を書き込むのに必要

Listing 53: EssentialParticleJ 型

```
 1          fwrite(&v.z,    sizeof(PS::F32), 1, fp);
 2          fwrite(&m,    sizeof(PS::F32), 1, fp);
 3          fwrite(&i,    sizeof(PS::S32), 1, fp);
 4      }
 5
 6      PS::F64 calcDtime(run_param &this_run) {
 7      PS::F64 dtime_v, dtime_a, dtime;
 8      PS::F64 vnorm, anorm;
 9      vnorm = sqrt(SQR(this->vel))+TINY;
10      anorm = sqrt(SQR(this->acc+this->acc_pm))+TINY;
11
12      dtime_v = this->eps/vnorm;
13      dtime_a = sqrt(this->eps/anorm)*CUBE(this_run.anow);
14
15      dtime = fmin(0.5*dtime_v, dtime_a);
16
17      return dtime;
18    }
19 };
20
21 //PS::F64vec FPtreepm::low_boundary;
22 //PS::F64vec FPtreepm::high_boundary;
23 //PS::F64 FPtreepm::unit_l;
24 //PS::F64 FPtreepm::unit_m;
```

```
25 PS::F64 FPtreepm::H0;
26 PS::F64 FPtreepm::Lbnd;
27
28 class EPItreepm {
29 public:
```

6.2.3.4　Force 型

ユーザーは Force 型を記述しなければならない。Force 型は、PP パートの Force の計算を行った際にその結果として得られる全ての物理量をメンバ変数として持っている必要がある。本サンプルコードの Force 型を Listing 54 に示す。Force 型には積算対象のメンバ変数を 0 ないし初期値に設定するための関数 `clear()` が必要になる。

Listing 54: Force 型

```
1  class Result_treepm {
2  public:
3    PS::F32vec acc;
4    PS::F32    pot;
5
6    void clear() {
7      acc = 0.0;
8      pot = 0.0;
9    }
10 };
```

6.2.3.5　calcForceEpEp 型

ユーザーは calcForceEpEp 型を記述しなければならない。calcForceEpEp 型には、PP パートの Force の計算の具体的な内容を書く必要がある。サンプルコードの calcForceEpEp 型を Listing 55 に示す。このサンプルコードでは、calcForceEpEp 型はファンクタ (関数オブジェクト) として実装されている (テンプレート関数として実装されていることに注意されたい)。また、Phantom-GRAPE ライブラリを使用するかどうかに応じて (マクロ定義 `ENABLE_PHANTOM_GRAPE_X86` で判定している)、場合分けして実装されている。いずれの場合でも、ファンクタの引数は、EssentialParticleI の配列、EssentialParticleI の個数、EssentialParticleJ の配列、EssentialParticleJ の個数、Force 型の配列である。

Listing 55: calcForceEpEp 型

```
1    PS::S64    id;
2    PS::F64vec pos;
3    PS::F64    mass;
4    //  PS::F64    rcut;
5
6    PS::F64vec getPos() const {
7      return this->pos;
8    }
9
10   PS::F64 getCharge() const {
```

```
11      return this->mass;
12    }
13
14    void copyFromFP(const FPtreepm & fp) {
15      this->id = fp.id;
16      this->mass = fp.mass;
17      this->pos = fp.pos;
18    }
19
20    PS::F64 getRSearch() const {
21      PS::F64 rcut = 3.0/SIZE_OF_MESH;
22      return rcut;
23    }
24
25    void setPos(const PS::F64vec pos_new) {
26      this->pos = pos_new;
27    }
28  };
29
30  inline PS::F64 gfactor_S2(const PS::F64 rad, const PS::F64 eps_pm)
31  {
32    PS::F64 R;
33    PS::F64 g;
34    PS::F64 S;
35
36    R = 2.0*rad/eps_pm;
37    R = (R > 2.0) ? 2.0 : R;
38    S = R-1.0;
39    S = (S > 0.0) ? S : 0.0;
40
41    g = 1.0 + CUBE(R)*(-1.6+SQR(R)*(1.6+R*(-0.5+R*(0.15*R-12.0/35.0))))
42      -CUBE(S)*CUBE(S)*(3.0/35.0+R*(18.0/35.0+0.2*R));
43
44    return g;
45  }
46
47  #ifdef ENABLE_PHANTOM_GRAPE_X86
48  template <class TPJ>
49  class calc_pp_force {
50  public:
51      void operator()(EPItreepm *iptcl,
52                      const PS::S32 ni,
53                      TPJ *jptcl,
54                      const PS::S32 nj,
55                      Result_treepm *ppforce){
56          static const PS::S32 IPTCL_MAX = 4096;
57          static const PS::S32 JPTCL_MAX = 16384;
58          static __thread PS::F64vec xi[IPTCL_MAX];
59          static __thread PS::F64vec ai[IPTCL_MAX];
60          static __thread PS::F64 pi[IPTCL_MAX];
61          static __thread PS::F64vec xj[JPTCL_MAX];
62          static __thread PS::F64 mj[JPTCL_MAX];
63          const int j_loop_max = ((nj-1) / JPTCL_MAX) + 1;
64          assert(ni <= IPTCL_MAX);
65          for (int i=0; i<ni; i++){
```

```
66              xi[i] = iptcl[i].getPos();
67              ai[i] = 0.0;
68              pi[i] = 0.0;
69          }
70          for (int j_loop=0; j_loop<j_loop_max; j_loop++){
```

TreePM 法の PP パートは、P^{3}M 法と同様、距離に関するカットオフ付きの 2 体相互作用である。そのため、ここでも加速度計算にカットオフ関数が掛かる。6.1.3.4 節で解説したように、カットオフ関数は Hockney & Eastwood (1988) の $S2$ 型の粒子形状関数に対応したカットオフ関数である必要がある。Phantom-GRAPE ライブラリを使用しない場合の実装では、カットオフ関数が関数 `gfactor_S2()` として定義されている。一方、Phantom-GRAPE ライブラリを使用する場合には、カットオフが考慮されたバージョンの Phantom-GRAPE ライブラリが使用されるようになっている。Phantom-GRAPE ライブラリに指定したカットオフ半径で計算させるため、API `pg5_gen_s2_force_table()` を事前に呼び出しておく必要がある。サンプルコードでは、メイン関数でこれを行っている:

```
1 #ifdef ENABLE_PHANTOM_GRAPE_X86
2     //g5_open();
3     pg5_gen_s2_force_table(EPS_FOR_PP, 3.0/SIZE_OF_MESH);
4 #endif
```

6.2.4　プログラム本体

本節では、サンプルコード本体について解説を行う。詳細な説明に入る前に、サンプルコードの内容と全体構造について説明を与える。6.2 節冒頭で述べたように、このサンプルコードでは宇宙論的 N 体シミュレーションを TreePM 法を用いて実行する。初期条件としては、以下の 3 つの場合に対応している:

(a) Santa Barbara Cluster Comparison Test (Frenk et al.[1999, ApJ, 525, 554]) で用いられた初期条件 (以下から初期条件を入手可能。$N = 128^3$: http://particle.riken.jp/~fdps/data/sb/ic_sb128.tar 、$N = 256^3$: http://particle.riken.jp/~fdps/data/sb/ic_sb256.tar)
(b) 上記テストで用いられた初期条件ファイルと同じフォーマットで記述された初期条件
(c) ランダムにおいた粒子分布

実行時のコマンドライン引数として初期条件を指定した後、初期条件ファイル内で指定された終了時刻 (赤方偏移 z) まで、TreePM 法で粒子の運動を計算する。各初期条件に対応したパラメータファイルのファイルフォーマットについては、`$(FDPS)/sample/c++/treepm/README.txt` で説明されているので、そちらを参照されたい。また、`$(FDPS)/sample/c++/treepm/src/result/input.para` に、(a) の場合のパラメータファイルの記述例があるので、そちらも参照されたい。

コード全体の構造は以下のようになっている:

(1) FDPS で使用するオブジェクトの生成と初期化
(2) (必要であれば)Phantom-GRAPE ライブラリの初期化
(3) 初期条件ファイルの読み込み

(4) 終了時刻まで粒子の運動を計算

以下で、個々について詳しく説明を行う。

6.2.4.1　開始、終了

まずは、FDPS の初期化/開始を行う必要がある。次のように、メイン関数に記述する。

Listing 56: FDPS の開始

```
1 PS::Initialize(argc, argv);
```

FDPS は、開始したら明示的に終了させる必要がある。今回は、プログラムの終了と同時に FDPS も終了させるため、メイン関数の最後に次のように記述する。

Listing 57: FDPS の終了

```
1 PS::Finalize();
```

6.2.4.2　オブジェクトの生成と初期化

FDPS の初期化に成功した場合、ユーザーはコード中で用いるオブジェクトを作成する必要がある。本節では、オブジェクトの生成/初期化の仕方について、解説する。

6.2.4.2.1　オブジェクトの生成

TreePM 法の計算では、P^3M 法のときと同様、粒子群クラス、領域クラス、PP パートの計算で使用する tree を 1 本、そして、PM パートの計算に必要な ParticleMesh オブジェクトの生成が必要である。本サンプルコードでは、`treepm.cpp` のメイン関数内でオブジェクトの生成が行われている:

Listing 58: オブジェクトの生成

```
1 int main(int argc, char **argv)
2 {
3    PS::PM::ParticleMesh pm;
4    PS::ParticleSystem<FPtreepm> ptcl;
5    PS::DomainInfo domain_info;
6    PS::TreeForForceLong<Result_treepm, EPItreepm, EPJtreepm>::
           MonopoleWithCutoff treepm_tree;
7 }
```

上記はサンプルコードからオブジェクト生成部分だけを抜き出してきたものであることに注意されたい。

6.2.4.2.2　オブジェクトの初期化

ほとんどの FDPS のオブジェクトは、生成後、初期化してから使用する必要がある。前節で説明した 4 つのオブジェクトの内、明示的な初期化が不要なのは ParticleMesh クラスであ

る。それ以外のオブジェクトに関しては、initialize メソッドで初期化を行う。以下に、サンプルコードでのオブジェクトの初期化を示す:

Listing 59: オブジェクトの初期化

```
1  int main(int argc, char **argv)
2  {
3     // Initialize ParticleSystem
4     ptcl.initialize();
5
6     // Initialize DomainInfo
7     domain_info.initialize();
8     domain_info.setBoundaryCondition(PS::BOUNDARY_CONDITION_PERIODIC_XYZ);
9     domain_info.setPosRootDomain(PS::F64vec(0.0, 0.0, 0.0),
10                                 PS::F64vec(1.0, 1.0, 1.0));
11
12    // Initialize Tree
13    treepm_tree.initialize(3*ptcl.getNumberOfParticleGlobal(),
14                           this_run.theta);
15 }
```

粒子群クラスのオブジェクトの初期化は、単に initialize メソッドを引数無しで呼び出すだけである。

領域クラスに関しては、initialize メソッド呼び出し後に、境界条件と境界の大きさを指定する必要がある。これらはそれぞれ setBoundaryCondition メソッドと setPosRootDomain メソッドで行う。

ツリーオブジェクトの初期化の際には、initialize メソッドに計算で使用する大雑把な粒子数を第 1 引数として渡す必要がある。本サンプルコードでは、全粒子数の 3 倍の値を渡している。第 2 引数には、tree 法で力の計算を行う際の opening angle criterion θ を指定する。本サンプルコードでは、θ をはじめとした、計算を制御するパラメータ群を構造体 this_run のメンバ変数としてまとめている。

6.2.4.3　初期条件の設定

初期条件を指定するパラメータファイルの読み込みは、メイン関数で呼ばれる関数 read_param_file() 内で行われる:

```
1  read_param_file(ptcl, this_run, argv[1]);
```

この関数ではプログラム実行時に指定されたパラメータファイルを読み込み、粒子群クラスのオブジェクト ptcl に粒子データをセットする。サンプルコードでは、この後、FDPS の API を使って、領域分割と粒子交換を行っている。以下でこれらの API について解説する。

6.2.4.3.1　領域分割の実行

粒子分布に基いて領域分割を実行するには、領域クラスの decomposeDomainAll メソッドを使用する:

Listing 60: 領域分割の実行

```
1 domain_info.decomposeDomainAll(system);
```

ここで、粒子分布の情報を領域クラスに与えるため、引数に粒子群クラスのオブジェクトが渡されていることに注意されたい。この領域分割はメイン関数内で行われている。

6.2.4.3.2　粒子交換の実行

領域情報に基いてプロセス間の粒子の情報を交換するには、粒子群クラスの `exchangeParticle` メソッドを使用する:

Listing 61: 粒子交換の実行

```
1 ptcl.exchangeParticle(domain_info);
```

ここで領域情報を粒子群クラスに与えるため、引数に領域クラスのオブジェクトが渡されていることに注意する。

6.2.4.4　相互作用計算の実行

領域分割・粒子交換が完了したら、計算開始時の加速度を決定するため、相互作用計算を行う必要がある。以下に、本サンプルコードでの相互作用計算の実装を示す。本サンプルコードでは、PP パートの計算にはツリーオブジェクトの `calcForceAllAndWriteBack` メソッドを使用している。このメソッドを実行することで、粒子群オブジェクトのメンバ変数 `acc` に PP パートの加速度が格納される。PM パートの計算には ParticleMesh オブジェクトの `calcForceAllAndWriteBack` メソッドを使用している。これによって、粒子群クラスのメンバ変数 `acc_pm` に PM パートの加速度が格納される。

Listing 62: 相互作業計算の実行

```
1 //* PP part
2 treepm_tree.calcForceAllAndWriteBack
3     (calc_pp_force<EPJtreepm>(),
4      calc_pp_force<PS::SPJMonopoleCutoff>(),
5      ptcl,
6      domain_info);
7
8 //* PM part
9 pm.calcForceAllAndWriteBack(ptcl, domain_info);
```

6.2.4.5　時間積分ループ

本サンプルコードでは、時間積分を Leapfrog 時間積分法によって行っている (この方法に関しては、4.1.3.3.4 節を参照されたい)。粒子位置を時間推進する $D(\cdot)$ オペレータは関数 `drift_ptcl`、粒子速度を時間推進する $K(\cdot)$ オペレータは関数 `kick_ptcl` として実装されている。宇宙膨張の効果は関数 `kick_ptcl` 内で考慮されている。また、スケールファクターや

ハッブルパラメータの時間発展は構造体 this_run のメンバ関数 update_expansion で計算されている。

6.2.5　コンパイル

README.txt で説明されているように、src ディレクトリの Makefile を適宜編集し、make コマンドを実行することでコンパイルすることができる。P^{3}M コード同様、本サンプルコードでも FFTW ライブラリを使用するため、ユーザ自身でインストールする必要がある。コンパイルが成功すれば、実行ファイル treepm が作成されているはずである。

6.2.6　実行

FDPS の拡張機能 ParticleMesh の仕様上、プロセス数が 2 以上の MPI 実行でなければ正常に動作しない。そこで、以下のように実行する必要がある:

```
$ MPIRUN -np NPROC ./treepm
```

ここで、”MPIRUN”には mpirun や mpiexec などの MPI 実行プログラムが、”NPROC”にはプロセス数が入る。

6.2.7　結果の確認

計算が終了するとパラメータファイルで指定されたディレクトリに計算結果が出力されるはずである。粒子数 256^3 で、Santa Barbara Cluster Comparison Test を実行した場合の、ダークマター密度分布の時間発展の様子を図 4 に示す。

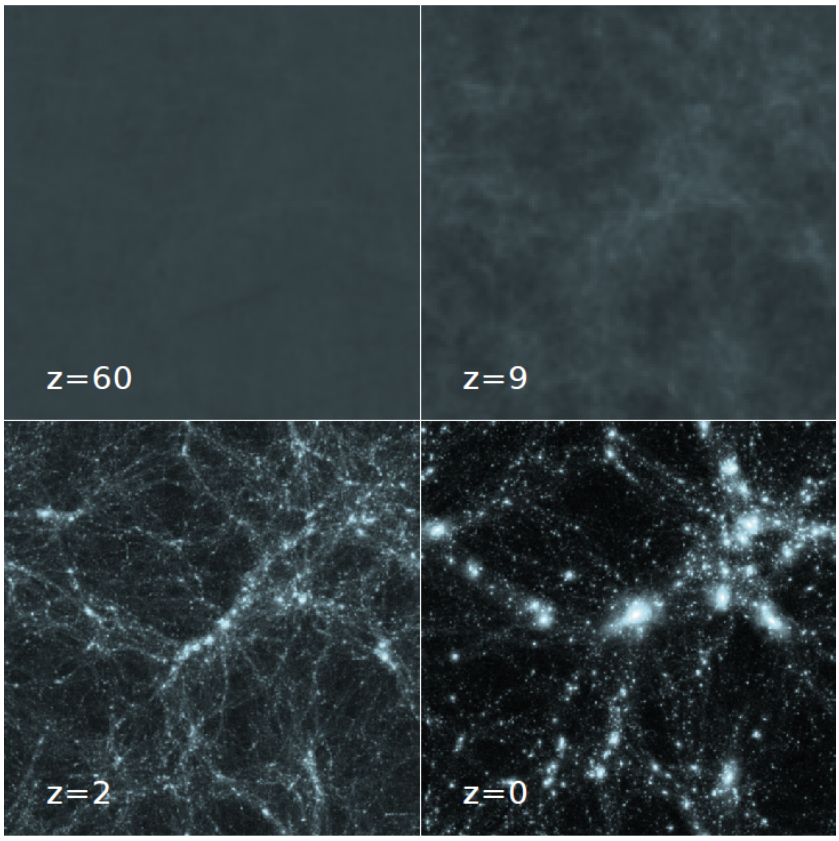

図 4: Santa Barbara Cluster Comparison テストの密度分布の時間発展 (粒子数 256^3)

7　ユーザーサポート

FDPS を使用したコード開発に関する相談は fdps-support<at>mail.jmlab.jp で受け付けています (<at>は@に変更お願い致します)。以下のような場合は各項目毎の対応をお願いします。

7.1　コンパイルできない場合

ユーザーには以下の情報提供をお願いします。

- コンパイル環境
- コンパイル時に出力されるエラーメッセージ
- ソースコード (可能ならば)

7.2　コードがうまく動かない場合

ユーザーには以下の情報提供をお願いします。

- 実行環境
- 実行時に出力されるエラーメッセージ
- ソースコード (可能ならば)

7.3　その他

思い通りの性能がでない場合やその他の相談なども、上のメールアドレスにお知らせください。

8 ライセンス

MIT ライセンスに準ずる。標準機能のみ使用する場合は、Iwasawa et al. (PASJ, 68, 54)、Namekata et al. (in prep) の引用をお願いします。

拡張機能の Particle Mesh クラスは GreeM コード (開発者 : 石山智明、似鳥啓吾) (Ishiyama, Fukushige & Makino 2009, Publications of the Astronomical Society of Japan, 61, 1319; Ishiyama, Nitadori & Makino, 2012 SC'12 Proceedings of the International Conference on High Performance Computing, Networking Stroage and Analysis, No. 5) のモジュールを使用している。GreeM コードは Yoshikawa & Fukushige (2005, Publications of the Astronomical Society of Japan, 57, 849) で書かれたコードをベースとしている。Particle Mesh クラスを使用している場合は、上記 3 つの文献の引用をお願いします。

拡張機能のうち x86 版 Phantom-GRAPE を使用する場合は Tanikawa et al.(2012, New Astronomy, 17, 82) と Tanikawa et al.(2012, New Astronomy, 19, 74) の引用をお願いします。