FDPS ユーザチュートリアル

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

理化学研究所 計算科学研究センター 粒子系シミュレータ研究チーム

0 目 次

1 変更記録 **6**

2 概要 **8**

3 入門：サンプルコードを動かしてみよう **9**
3.1 動作環境 . . . 9
3.2 必要なソフトウェア . . . 9
3.2.1 標準機能 . . . 9
3.2.1.1 逐次処理 . . . 9
3.2.1.2 並列処理 . . . 9
3.2.1.2.1 OpenMP . . . 9
3.2.1.2.2 MPI . . . 10
3.2.1.2.3 MPI+OpenMP . . . 10
3.2.2 拡張機能 . . . 10
3.2.2.1 Particle Mesh . . . 10
3.3 インストール . . . 10
3.3.1 取得方法 . . . 10
3.3.1.1 最新バージョン . . . 11
3.3.1.2 過去のバージョン . . . 11
3.3.2 インストール方法 . . . 11
3.4 サンプルコードの使用方法 . . . 11
3.4.1 重力 N 体シミュレーションコード . . . 12
3.4.1.1 概要 . . . 12
3.4.1.2 ディレクトリ移動 . . . 12
3.4.1.3 Makefile の編集 . . . 12
3.4.1.4 `make` の実行 . . . 13
3.4.1.5 実行 . . . 13

3.4.1.6 結果の解析 . . . 14
3.4.1.7 x86 版 Phantom-GRAPE を使う場合 . . . 15
3.4.1.8 NVIDIA の GPU を使う場合 . . . 15
3.4.2 SPH シミュレーションコード . . . 15
3.4.2.1 概要 . . . 16
3.4.2.2 ディレクトリ移動 . . . 16
3.4.2.3 Makefile の編集 . . . 16
3.4.2.4 make の実行 . . . 16
3.4.2.5 実行 . . . 16
3.4.2.6 結果の解析 . . . 17

4 サンプルコードの解説 18
4.1 *N* 体シミュレーションコード . . . 18
4.1.1 ソースファイルの場所と構成 . . . 18
4.1.2 ユーザー定義型・ユーザ定義関数 . . . 18
4.1.2.1 FullParticle 型 . . . 18
4.1.2.2 相互作用関数 calcForceEpEp . . . 19
4.1.3 プログラム本体 . . . 20
4.1.3.1 ヘッダーファイルのインクルード . . . 20
4.1.3.2 開始、終了 . . . 21
4.1.3.3 オブジェクトの生成・初期化 . . . 21
4.1.3.3.1 オブジェクトの生成 . . . 21
4.1.3.3.2 領域情報オブジェクトの初期化 . . . 21
4.1.3.3.3 粒子群オブジェクトの初期化 . . . 22
4.1.3.3.4 ツリーオブジェクトの初期化 . . . 22
4.1.3.4 ループ . . . 22
4.1.3.4.1 領域分割の実行 . . . 22
4.1.3.4.2 粒子交換の実行 . . . 22
4.1.3.4.3 相互作用計算の実行 . . . 23
4.1.3.4.4 時間積分 . . . 23
4.1.4 ログファイル . . . 23
4.2 固定長 SPH シミュレーションコード . . . 24
4.2.1 ソースファイルの場所と構成 . . . 24
4.2.2 ユーザー定義型・ユーザ定義関数 . . . 24
4.2.2.1 FullParticle 型 . . . 24
4.2.2.2 EssentialParticleI 型 . . . 25
4.2.2.3 Force 型 . . . 26
4.2.2.4 相互作用関数 calcForceEpEp . . . 27
4.2.3 プログラム本体 . . . 28
4.2.3.1 ヘッダーファイルのインクルード . . . 28
4.2.3.2 開始、終了 . . . 29

4.2.3.3 オブジェクトの生成・初期化 29
4.2.3.3.1 オブジェクトの生成 29
4.2.3.3.2 領域情報オブジェクトの初期化 29
4.2.3.3.3 粒子群オブジェクトの初期化 30
4.2.3.3.4 ツリーオブジェクトの初期化 30
4.2.3.4 ループ 30
4.2.3.4.1 領域分割の実行 30
4.2.3.4.2 粒子交換の実行 30
4.2.3.4.3 相互作用計算の実行 30
4.2.4 コンパイル 31
4.2.5 実行 31
4.2.6 ログファイル 31
4.2.7 可視化 31

5 サンプルコード **32**
5.1 N 体シミュレーション 32
5.2 固定長 SPH シミュレーション 41

6 拡張機能の解説 **51**
6.1 P^3M コード 51
6.1.1 サンプルコードの場所と作業ディレクトリ 51
6.1.2 ユーザー定義型 51
6.1.2.1 FullParticle 型 51
6.1.2.2 EssentialParticleI 型 52
6.1.2.3 Force 型 53
6.1.2.4 相互作用関数 calcForceEpEp 54
6.1.2.5 相互作用関数 calcForceEpSp 55
6.1.3 プログラム本体 56
6.1.4 ヘッダファイルのインクルード 56
6.1.4.1 開始、終了 57
6.1.4.2 オブジェクトの生成と初期化 57
6.1.4.2.1 オブジェクトの生成 57
6.1.4.2.2 オブジェクトの初期化 58
6.1.4.3 粒子分布の生成 59
6.1.4.3.1 領域分割の実行 60
6.1.4.3.2 粒子交換の実行 60
6.1.4.4 相互作用計算の実行 60
6.1.4.5 エネルギー相対誤差の計算 62
6.1.5 コンパイル 62
6.1.6 実行 62
6.1.7 結果の確認 62

6.2 TreePM コード . . . 64
6.2.1 サンプルコードの場所と作業ディレクトリ . . . 64
6.2.2 ヘッダファイルのインクルード . . . 64
6.2.3 ユーザー定義クラス . . . 65
6.2.3.1 FullParticle 型 . . . 65
6.2.3.2 EssentialParticleI 型 . . . 69
6.2.3.3 EssentialParticleJ 型 . . . 70
6.2.3.4 Force 型 . . . 71
6.2.3.5 calcForceEpEp 型 . . . 71
6.2.4 プログラム本体 . . . 72
6.2.4.1 開始、終了 . . . 73
6.2.4.2 オブジェクトの生成と初期化 . . . 73
6.2.4.2.1 オブジェクトの生成 . . . 73
6.2.4.2.2 オブジェクトの初期化 . . . 73
6.2.4.3 初期条件の設定 . . . 74
6.2.4.3.1 領域分割の実行 . . . 74
6.2.4.3.2 粒子交換の実行 . . . 75
6.2.4.4 相互作用計算の実行 . . . 75
6.2.4.5 時間積分ループ . . . 75
6.2.5 コンパイル . . . 76
6.2.6 実行 . . . 76
6.2.7 結果の確認 . . . 76

7 より実用的なアプリケーションの解説 78
7.1 *N* 体/SPH コード . . . 78
7.1.1 コードの使用方法 . . . 78
7.1.1.1 ディレクトリ移動 . . . 79
7.1.1.2 サンプルコードのファイル構成 . . . 79
7.1.1.3 Makefile の編集 . . . 79
7.1.1.4 MAGI を使った粒子データの生成 . . . 82
7.1.1.5 make の実行 . . . 83
7.1.1.6 実行 . . . 83
7.1.1.7 結果の解析 . . . 83
7.1.2 Springel の方法 . . . 84
7.1.3 ユーザー定義型 . . . 85
7.1.3.1 FullParticle 型 . . . 86
7.1.3.2 EssentialParticle 型 . . . 89
7.1.3.3 Force 型 . . . 91
7.1.4 相互作用関数 . . . 92
7.1.4.1 重力計算 . . . 92
7.1.4.2 密度計算 . . . 94

7.1.4.3 圧力勾配加速度計算 97
7.1.5 プログラム本体 98
7.1.5.1 ヘッダファイルのインクルード 99
7.1.5.2 開始、終了 99
7.1.5.3 オブジェクトの生成と初期化 99
7.1.5.3.1 粒子群オブジェクトの生成と初期化 99
7.1.5.3.2 領域情報オブジェクトの生成と初期化 100
7.1.5.3.3 ツリーオブジェクトの生成と初期化 100
7.1.5.4 初期条件の設定 101
7.1.5.5 領域分割の実行 101
7.1.5.6 粒子交換の実行 102
7.1.5.7 相互作用計算の実行 102
7.1.5.8 時間積分ループ 104

8 ユーザーサポート **105**
8.1 コンパイルできない場合 105
8.2 コードがうまく動かない場合 105
8.3 その他 105

9 ライセンス **106**

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)
- 2018/07/11
  - 第 3 節の以下の記述を修正・改善
    - 動作確認済みコンパイラを更新 (FDPS バージョン 4.1a で確認)
    - SPH サンプルコードの出力ファイルの拡張子の誤植
  - 第 4 節の以下の記述を修正・改善
    - 領域情報クラスの initialize メソッドの仕様書での節番号に誤植 (第 3.4.1 節)
    - チュートリアルの記述と実際のコードで calcForceAllAndWriteBack() の引数に不一致 (第 3.4.1 節)
  - 第 6 節の以下の記述を修正・改善
    - PP パート部分の記述とソースコードに不一致 (第 6.1 節)
    - ファイル構成の説明が、実際のサンプルコードと一致していない (第 6.2 節)
    - チュートリアルの記述と実際のコードで decomposeDomainAll の引数に不一致 (第 6.2 節)
- 2018/8/18
  - *N* 体/SPH サンプルコードの解説を追加 (第 7.1 節)

- 2019/7/19
  - N 体/SPH サンプルコードの記述を改訂 (第 7.1 節)

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++ 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 4.8.3 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 4.8.3 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 1.6.4 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 1.6.4 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 1.6.4 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、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. ********
```

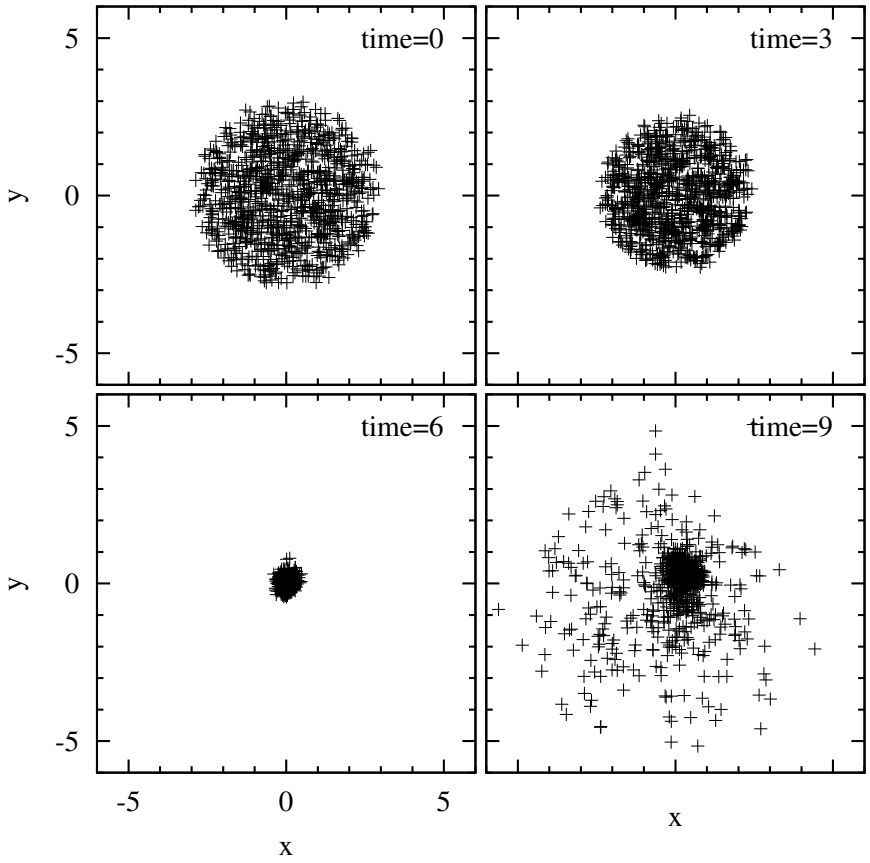

図 1:

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、Phantom-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.txt”(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.txt" using 3:9
```

正しい答が得られれば、図 2 のような図を描ける。

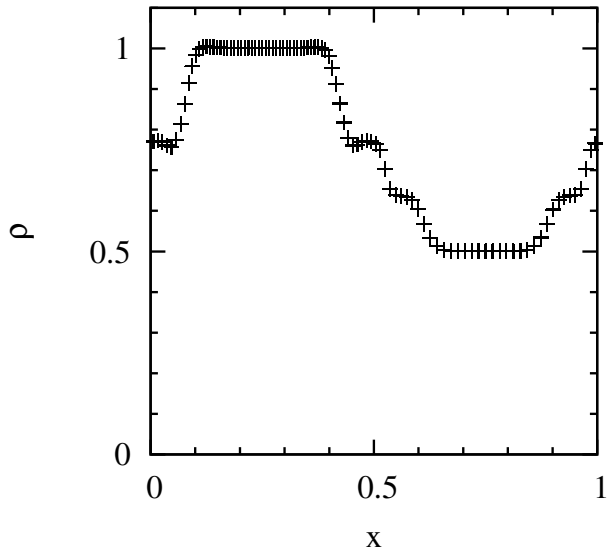

図 2: 衝撃波管問題の時刻 $t = 40$ における密度分布

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 (以後、**ユーザ定義型**と呼ぶ) や FDPS の各種 API の使い方について詳しく述べる。説明の重複を避けるため、いくつかの事項に関しては、その詳細な説明がN体シミュレーションコードの節でのみ行われている。そのため、SPH シミュレーションだけに興味があるユーザも、N 体シミュレーションコードの節に目を通して頂きたい。

4.1 N 体シミュレーションコード

4.1.1 ソースファイルの場所と構成

ソースファイルは`$(FDPS)/sample/c++/nbody` 以下にある。 サンプルコードは、次節で説明するユーザ定義型が記述されたソースコード `user-defined.hpp` と、N 体シミュレーションのメインループ等が記述されたソースコード `main.cpp` から構成される。この他に、GCC 用の `Makefile` がある。

4.1.2 ユーザー定義型・ユーザ定義関数

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と `void` 関数について記述する。

4.1.2.1 FullParticle 型

ユーザーはユーザ定義型の 1 つ 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、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 FullParticle 型の実装例を示す (user-defined.hpp を参照)。

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  };
```

本サンプルコードでは、FullParticle型がEssentialParticleI型、EssentialParticleJ型、そして、Force型を兼ねている。また、FullParticle 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。

4.1.2.2　相互作用関数 calcForceEpEp

ユーザーは粒子間相互作用の仕方を記述した相互作用関数 calcForceEpEp を記述しなければならない。`void` 関数 calcForceEpEp には、粒子-粒子相互作用計算の具体的な内容を書く必要がある。Listing 2 に、本サンプルコードでの実装を示す (`user-defined.hpp` を参照)。

Listing 2: 関数 calcForceEpEp

```
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;
    }
}
```

ここに示したのは、Phantom-GRAPE ライブラリを使用せずに CPU で実行する場合の実装である。

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。

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　ヘッダーファイルのインクルード

FDPS の標準機能を利用できるようにするため、`particle_simulator.hpp` をインクルードする。

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 3: ヘッダーファイル particle_simulator.hpp のインクルード

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

4.1.3.2　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 4: 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 5: FDPS の終了

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

4.1.3.3　オブジェクトの生成・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。

4.1.3.3.1　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、重力計算用の相互作用ツリー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 6: 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;
```

4.1.3.3.2　領域情報オブジェクトの初期化

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、領域情報オブジェクトの初期化は API initialize を実行するだけでよい:

Listing 7: 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、API initialize の第 1 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に詳しい解説があるので、そちらを参照されたい。

4.1.3.3.3　粒子群オブジェクトの初期化

次に、粒子群オブジェクトの初期化を行う必要がある。粒子群オブジェクトの初期化は、API `initialize` で行う:

Listing 8: 粒子群クラスの初期化

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

4.1.3.3.4　ツリーオブジェクトの初期化

次に、ツリーオブジェクトの初期化を行う必要がある。ツリーオブジェクトの初期化は API `initialize` で行う。この API 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 (`ntot`) をセットしておく事にする:

Listing 9: 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 API 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

4.1.3.4　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。

4.1.3.4.1　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、これを領域情報オブジェクトの API `decomposeDomainAll` を用いて行っている:

Listing 10: 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。

4.1.3.4.2　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、粒子群オブジェクトの API `exchangeParticle` を用いる:

Listing 11: 粒子交換の実行

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

4.1.3.4.3　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、ツリーオブジェクトの API `calcForceAllAndWriteBack` を用いる:

Listing 12: 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。

4.1.3.4.4　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、`void` 関数 `kick` と `void` 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 13: $D(\Delta t)K(\frac{\Delta t}{2})$ 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 14: $K(\frac{\Delta t}{2})$ 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、標準出力に、時間・エネルギー誤差の 2 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 15: 標準出力の例

```
1 time:    0.0000000000E+000, energy error:    -0.0000000000E+000
```

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　ソースファイルの場所と構成

ソースファイルは`$(FDPS)/sample/c++/sph` 以下にある。 サンプルコードは、次節で説明するユーザ定義型や SPH シミュレーションのメインループ等すべてが記述された `main.cpp` と GCC 用 `Makefile` から構成される。

4.2.2　ユーザー定義型・ユーザ定義関数

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と `void` 関数について記述する。

4.2.2.1　FullParticle 型

ユーザーはユーザ定義型の 1 つ 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。Listing 16 に本サンプルコード中で用いる FullParticle 型の実装例を示す (main.cpp を参照)。

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 };
```

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。

4.2.2.2　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。Listing 17 に、本サンプルコードの EssentialParticleI 型の実装例を示す (`main.cpp` 参照):

Listing 17: EssentialParticleI 型

```
struct EP{
   PS::F64vec pos;
   PS::F64vec vel;
   PS::F64    mass;
   PS::F64    smth;
   PS::F64    dens;
   PS::F64    pres;
   PS::F64    snds;
   void copyFromFP(const FP& rp){
      this->pos  = rp.pos;
      this->vel  = rp.vel;
      this->mass = rp.mass;
      this->smth = rp.smth;
      this->dens = rp.dens;
      this->pres = rp.pres;
      this->snds = rp.snds;
   }
   PS::F64vec getPos() const{
      return this->pos;
   }
   PS::F64 getRSearch() const{
      return kernelSupportRadius * this->smth;
   }
   void setPos(const PS::F64vec& pos){
      this->pos = pos;
   }
};
```

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。

4.2.2.3　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 の計算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。Listing 18 に、本サンプルコード中で用いる Force 型の実装例を示す。

Listing 18: Force 型

```
class Dens{
   public:
   PS::F64 dens;
   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 };
```

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。

4.2.2.4　相互作用関数 calcForceEpEp

ユーザーは粒子間相互作用の仕方を記述した相互作用関数 calcForceEpEp を記述しなければならない。相互作用関数 calcForceEpEp には、各 Force 型に対応する粒子-粒子相互作用計算の具体的な内容を書く必要がある。Listing 19 に、本サンプルコードでの実装を示す (main.cpp を参照)。

Listing 19: 関数 calcForceEpEp

```
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 };
```

本 SPH シミュレーションコードでは、2 種類の相互作用があるため、calcForceEpEp は 2 つ記述する必要がある。いずれの場合にも、void 関数の仮引数は、EssentialParticleI の配列、EssentialParticleI の個数、EssentialParticleJ の配列、EssentialParticleJ の個数、Force 型の配列である。

4.2.3　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 。

4.2.3.1　ヘッダーファイルのインクルード

FDPS の標準機能を利用できるようにするため、particle_simulator.hpp をインクルードする。

Listing 20: ヘッダーファイル particle_simulator.hpp のインクルード

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

4.2.3.2　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 21: 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 22: FDPS の終了

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

4.2.3.3　オブジェクトの生成・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。

4.2.3.3.1　オブジェクトの生成

SPH では、粒子群オブジェクト、領域情報オブジェクトに加え、密度計算用に Gather 型の短距離力用ツリーを 1 本、流体相互作用計算用に Symmetry 型の短距離力用ツリーを 1 本生成する必要がある。以下にそのコードを記す。

Listing 23: 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;
```

4.2.3.3.2　領域情報オブジェクトの初期化

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、まず領域情報オブジェクトの初期化について、解説する。領域情報オブジェクトの初期化が終わった後、領域情報オブジェクト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 24: 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));
```

4.2.3.3.3　粒子群オブジェクトの初期化

次に、粒子群オブジェクトの初期化を行う必要がある。粒子群オブジェクトの初期化は、次の一文だけでよい。

Listing 25: 粒子群クラスの初期化

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

4.2.3.3.4　ツリーオブジェクトの初期化

次に、ツリーオブジェクトの初期化を行う必要がある。ツリーオブジェクト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 26: 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());
```

4.2.3.4　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。

4.2.3.4.1　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。これには、領域情報オブジェクトの API `decomposeDomainAll` を用いる。

Listing 27: 領域分割の実行

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

4.2.3.4.2　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、粒子群オブジェクトの API `exchangeParticle` を用いる。

Listing 28: 粒子交換の実行

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

4.2.3.4.3　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、ツリーオブジェクトの API `calcForceAllAndWriteBack` を用いる。

Listing 29: 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);
```

4.2.4　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、サンプルコードに付属の Makefile をそのまま用いる事にする。

```
$ make
```

4.2.5　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。

4.2.6　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。

4.2.7　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 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)

```
#pragma once
class FileHeader{
public:
    PS::S64 n_body;
    PS::F64 time;
    PS::S32 readAscii(FILE * fp) {
        fscanf(fp, "%lf\n", &time);
        fscanf(fp, "%lld\n", &n_body);
        return n_body;
    }
    void writeAscii(FILE* fp) const {
        fprintf(fp, "%e\n", time);
        fprintf(fp, "%lld\n", n_body);
    }
};

class FPGrav{
public:
    PS::S64    id;
    PS::F64    mass;
    PS::F64vec pos;
    PS::F64vec vel;
    PS::F64vec acc;
    PS::F64    pot;

    static PS::F64 eps;

    PS::F64vec getPos() const {
        return pos;
    }

    PS::F64 getCharge() const {
        return mass;
    }

    void copyFromFP(const FPGrav & fp){
        mass = fp.mass;
        pos  = fp.pos;
    }

    void copyFromForce(const FPGrav & force) {
        acc = force.acc;
        pot = force.pot;
    }
```

```
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     etot = PS::Comm::getSum(etot_loc);
122     epot = PS::Comm::getSum(epot_loc);
123     ekin = PS::Comm::getSum(ekin_loc);
124 }
125
126 void printHelp() {
127     std::cerr<<"o: dir name of output (default: ./result)"<<std::endl;
128     std::cerr<<"t: theta (default: 0.5)"<<std::endl;
129     std::cerr<<"T: time_end (default: 10.0)"<<std::endl;
130     std::cerr<<"s: time_step (default: 1.0 / 128.0)"<<std::endl;
131     std::cerr<<"d: dt_diag (default: 1.0 / 8.0)"<<std::endl;
132     std::cerr<<"D: dt_snap (default: 1.0)"<<std::endl;
133     std::cerr<<"l: n_leaf_limit (default: 8)"<<std::endl;
134     std::cerr<<"n: n_group_limit (default: 64)"<<std::endl;
135     std::cerr<<"N: n_tot (default: 1024)"<<std::endl;
136     std::cerr<<"h: help"<<std::endl;
137 }
138
139 void makeOutputDirectory(char * dir_name) {
140     struct stat st;
141     PS::S32 ret;
142     if (PS::Comm::getRank() == 0) {
143         if (stat(dir_name, &st) != 0) {
144             ret = mkdir(dir_name, 0777);
145         } else {
146             ret = 0; // the directory named dir_name already exists.
147         }
148     }
149     PS::Comm::broadcast(&ret, 1);
150     if (ret == 0) {
151         if (PS::Comm::getRank() == 0)
152             fprintf(stderr, "Directory \"%s\" is successfully made.\n",
                    dir_name);
153     } else {
154         if (PS::Comm::getRank() == 0)
155             fprintf(stderr, "Directory %s fails to be made.\n", dir_name);
156         PS::Abort();
157     }
158 }
159
160 PS::F64 FPGrav::eps = 1.0/32.0;
161
162 int main(int argc, char *argv[]) {
163     std::cout<<std::setprecision(15);
164     std::cerr<<std::setprecision(15);
```

```
165
166     PS::Initialize(argc, argv);
167     PS::F32 theta = 0.5;
168     PS::S32 n_leaf_limit = 8;
169     PS::S32 n_group_limit = 64;
170     PS::F32 time_end = 10.0;
171     PS::F32 dt = 1.0 / 128.0;
172     PS::F32 dt_diag = 1.0 / 8.0;
173     PS::F32 dt_snap = 1.0;
174     char dir_name[1024];
175     PS::S64 n_tot = 1024;
176     PS::S32 c;
177     sprintf(dir_name,"./result");
178     opterr = 0;
179     while((c=getopt(argc,argv,"i:o:d:D:t:T:l:n:N:hs:")) != -1){
180         switch(c){
181         case 'o':
182             sprintf(dir_name,optarg);
183             break;
184         case 't':
185             theta = atof(optarg);
186             std::cerr << "theta␣=" << theta << std::endl;
187             break;
188         case 'T':
189             time_end = atof(optarg);
190             std::cerr << "time_end␣=␣" << time_end << std::endl;
191             break;
192         case 's':
193             dt = atof(optarg);
194             std::cerr << "time_step␣=␣" << dt << std::endl;
195             break;
196         case 'd':
197             dt_diag = atof(optarg);
198             std::cerr << "dt_diag␣=␣" << dt_diag << std::endl;
199             break;
200         case 'D':
201             dt_snap = atof(optarg);
202             std::cerr << "dt_snap␣=␣" << dt_snap << std::endl;
203             break;
204         case 'l':
205             n_leaf_limit = atoi(optarg);
206             std::cerr << "n_leaf_limit␣=␣" << n_leaf_limit << std::endl;
207             break;
208         case 'n':
209             n_group_limit = atoi(optarg);
210             std::cerr << "n_group_limit␣=␣" << n_group_limit << std::endl;
211             break;
212         case 'N':
213             n_tot = atoi(optarg);
214             std::cerr << "n_tot␣=␣" << n_tot << std::endl;
215             break;
216         case 'h':
217             if(PS::Comm::getRank() == 0) {
218                 printHelp();
219             }
```

```
220             PS::Finalize();
221             return 0;
222         default:
223             if(PS::Comm::getRank() == 0) {
224                 std::cerr<<"No␣such␣option!␣Available␣options␣are␣here."<<
                        std::endl;
225                 printHelp();
226             }
227             PS::Abort();
228         }
229     }
230
231     makeOutputDirectory(dir_name);
232
233     std::ofstream fout_eng;
234
235     if(PS::Comm::getRank() == 0) {
236         char sout_de[1024];
237         sprintf(sout_de, "%s/t-de.dat", dir_name);
238         fout_eng.open(sout_de);
239         fprintf(stdout, "This␣is␣a␣sample␣program␣of␣N-body␣simulation␣on␣
                FDPS!\n");
240         fprintf(stdout, "Number␣of␣processes:␣%d\n", PS::Comm::
                getNumberOfProc());
241         fprintf(stdout, "Number␣of␣threads␣per␣process:␣%d\n", PS::Comm::
                getNumberOfThread());
242     }
243
244     PS::ParticleSystem<FPGrav> system_grav;
245     system_grav.initialize();
246     PS::S32 n_loc    = 0;
247     PS::F32 time_sys = 0.0;
248     if(PS::Comm::getRank() == 0) {
249         setParticlesColdUniformSphere(system_grav, n_tot, n_loc);
250     } else {
251         system_grav.setNumberOfParticleLocal(n_loc);
252     }
253
254     const PS::F32 coef_ema = 0.3;
255     PS::DomainInfo dinfo;
256     dinfo.initialize(coef_ema);
257     dinfo.decomposeDomainAll(system_grav);
258     system_grav.exchangeParticle(dinfo);
259     n_loc = system_grav.getNumberOfParticleLocal();
260
261 #ifdef ENABLE_PHANTOM_GRAPE_X86
262     g5_open();
263     g5_set_eps_to_all(FPGrav::eps);
264 #endif
265
266     PS::TreeForForceLong<FPGrav, FPGrav, FPGrav>::Monopole tree_grav;
267     tree_grav.initialize(n_tot, theta, n_leaf_limit, n_group_limit);
268 #ifdef MULTI_WALK
269     const PS::S32 n_walk_limit = 200;
270     const PS::S32 tag_max = 1;
```

```
271     tree_grav.calcForceAllAndWriteBackMultiWalk(DispatchKernelWithSP,
272                                                 RetrieveKernel,
273                                                 tag_max,
274                                                 system_grav,
275                                                 dinfo,
276                                                 n_walk_limit);
277 #else
278     tree_grav.calcForceAllAndWriteBack(CalcGravity<FPGrav>,
279                                        CalcGravity<PS::SPJMonopole>,
280                                        system_grav,
281                                        dinfo);
282 #endif
283     PS::F64 Epot0, Ekin0, Etot0, Epot1, Ekin1, Etot1;
284     calcEnergy(system_grav, Etot0, Ekin0, Epot0);
285     PS::F64 time_diag = 0.0;
286     PS::F64 time_snap = 0.0;
287     PS::S64 n_loop = 0;
288     PS::S32 id_snap = 0;
289     while(time_sys < time_end){
290         if( (time_sys >= time_snap) || ( (time_sys + dt) - time_snap ) > (
                time_snap - time_sys) ){
291             char filename[256];
292             sprintf(filename, "%s/%04d.dat", dir_name, id_snap++);
293             FileHeader header;
294             header.time   = time_sys;
295             header.n_body = system_grav.getNumberOfParticleGlobal();
296             system_grav.writeParticleAscii(filename, header);
297             time_snap += dt_snap;
298         }
299
300         calcEnergy(system_grav, Etot1, Ekin1, Epot1);
301
302         if(PS::Comm::getRank() == 0){
303             if( (time_sys >= time_diag) || ( (time_sys + dt) - time_diag )
                      > (time_diag - time_sys) ){
304                 fout_eng << time_sys << "␣␣␣" << (Etot1 - Etot0) / Etot0
                          << std::endl;
305                 fprintf(stdout, "time:␣%10.7f␣energy␣error:␣%+e\n",
306                         time_sys, (Etot1 - Etot0) / Etot0);
307                 time_diag += dt_diag;
308             }
309         }
310
311
312         kick(system_grav, dt * 0.5);
313
314         time_sys += dt;
315         drift(system_grav, dt);
316
317         if(n_loop % 4 == 0){
318             dinfo.decomposeDomainAll(system_grav);
319         }
320
321         system_grav.exchangeParticle(dinfo);
322 #ifdef MULTI_WALK
```

```
323         tree_grav.calcForceAllAndWriteBackMultiWalk(DispatchKernelWithSP,
324                                                     RetrieveKernel,
325                                                     tag_max,
326                                                     system_grav,
327                                                     dinfo,
328                                                     n_walk_limit,
329                                                     true);
330 #else
331         tree_grav.calcForceAllAndWriteBack(CalcGravity<FPGrav>,
332                                            CalcGravity<PS::SPJMonopole>,
333                                            system_grav,
334                                            dinfo);
335 #endif
336
337         kick(system_grav, dt * 0.5);
338
339         n_loop++;
340     }
341
342 #ifdef ENABLE_PHANTOM_GRAPE_X86
343     g5_close();
344 #endif
345
346     PS::Finalize();
347     return 0;
348 }
```

5.2　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 32: 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  #include <sys/stat.h>
9
10 /* Parameters */
11 const short int Dim = 3;
12 const PS::F64 SMTH = 1.2;
13 const PS::U32 OUTPUT_INTERVAL = 10;
14 const PS::F64 C_CFL = 0.3;
15
16 /* Kernel Function */
17 const PS::F64 pi = atan(1.0) * 4.0;
```

```
18 const PS::F64 kernelSupportRadius = 2.5;
19
20 PS::F64 W(const PS::F64vec dr, const PS::F64 h){
21    const PS::F64 H = kernelSupportRadius * h;
22    const PS::F64 s = sqrt(dr * dr) / H;
23    const PS::F64 s1 = (1.0 - s < 0) ? 0 : 1.0 - s;
24    const PS::F64 s2 = (0.5 - s < 0) ? 0 : 0.5 - s;
25    PS::F64 r_value = pow(s1, 3) - 4.0 * pow(s2, 3);
26    //if # of dimension == 3
27    r_value *= 16.0 / pi / (H * H * H);
28    return r_value;
29 }
30
31 PS::F64vec gradW(const PS::F64vec dr, const PS::F64 h){
32    const PS::F64 H = kernelSupportRadius * h;
33    const PS::F64 s = sqrt(dr * dr) / H;
34    const PS::F64 s1 = (1.0 - s < 0) ? 0 : 1.0 - s;
35    const PS::F64 s2 = (0.5 - s < 0) ? 0 : 0.5 - s;
36    PS::F64 r_value = - 3.0 * pow(s1, 2) + 12.0 * pow(s2, 2);
37    //if # of dimension == 3
38    r_value *= 16.0 / pi / (H * H * H);
39    return dr * r_value / (sqrt(dr * dr) * H + 1.0e-6 * h);
40 }
41
42 /* Class Definitions */
43 //** Force Class (Result Class)
44 class Dens{
45    public:
46    PS::F64 dens;
47    PS::F64 smth;
48    void clear(){
49       dens = 0;
50    }
51 };
52 class Hydro{
53    public:
54    PS::F64vec acc;
55    PS::F64 eng_dot;
56    PS::F64 dt;
57    void clear(){
58       acc = 0;
59       eng_dot = 0;
60    }
61 };
62
63 //** Full Particle Class
64 struct FP{
65    PS::F64 mass;
66    PS::F64vec pos;
67    PS::F64vec vel;
68    PS::F64vec acc;
69    PS::F64 dens;
70    PS::F64 eng;
71    PS::F64 pres;
72    PS::F64 smth;
```

```
73     PS::F64 snds;
74     PS::F64 eng_dot;
75     PS::F64 dt;
76     PS::S64 id;
77     PS::F64vec vel_half;
78     PS::F64 eng_half;
79     void copyFromForce(const Dens& dens){
80        this->dens = dens.dens;
81     }
82     void copyFromForce(const Hydro& force){
83        this->acc     = force.acc;
84        this->eng_dot = force.eng_dot;
85        this->dt      = force.dt;
86     }
87     PS::F64 getCharge() const{
88        return this->mass;
89     }
90     PS::F64vec getPos() const{
91        return this->pos;
92     }
93     PS::F64 getRSearch() const{
94        return kernelSupportRadius * this->smth;
95     }
96     void setPos(const PS::F64vec& pos){
97        this->pos = pos;
98     }
99     void writeAscii(FILE* fp) const{
100       fprintf(fp,
101               "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t"
102               "%lf\t%lf\t%lf\t%lf\t%lf\n",
103               this->id, this->mass,
104               this->pos.x, this->pos.y, this->pos.z,
105               this->vel.x, this->vel.y, this->vel.z,
106               this->dens, this->eng, this->pres);
107    }
108    void readAscii(FILE* fp){
109       fscanf(fp,
110              "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t"
111              "%lf\t%lf\t%lf\t%lf\t%lf\n",
112              &this->id, &this->mass,
113              &this->pos.x, &this->pos.y, &this->pos.z,
114              &this->vel.x, &this->vel.y, &this->vel.z,
115              &this->dens, &this->eng, &this->pres);
116    }
117    void setPressure(){
118       const PS::F64 hcr = 1.4;
119       pres = (hcr - 1.0) * dens * eng;
120       snds = sqrt(hcr * pres / dens);
121    }
122 };
123
124 //** Essential Particle Class
125 struct EP{
126    PS::F64vec pos;
127    PS::F64vec vel;
```

```
128     PS::F64    mass;
129     PS::F64    smth;
130     PS::F64    dens;
131     PS::F64    pres;
132     PS::F64    snds;
133     void copyFromFP(const FP& rp){
134        this->pos  = rp.pos;
135        this->vel  = rp.vel;
136        this->mass = rp.mass;
137        this->smth = rp.smth;
138        this->dens = rp.dens;
139        this->pres = rp.pres;
140        this->snds = rp.snds;
141     }
142     PS::F64vec getPos() const{
143        return this->pos;
144     }
145     PS::F64 getRSearch() const{
146        return kernelSupportRadius * this->smth;
147     }
148     void setPos(const PS::F64vec& pos){
149        this->pos = pos;
150     }
151 };
152
153 class FileHeader{
154     public:
155     PS::S32 Nbody;
156     PS::F64 time;
157     int readAscii(FILE* fp){
158        fscanf(fp, "%lf\n", &time);
159        fscanf(fp, "%d\n", &Nbody);
160        return Nbody;
161     }
162     void writeAscii(FILE* fp) const{
163        fprintf(fp, "%e\n", time);
164        fprintf(fp, "%d\n", Nbody);
165     }
166 };
167
168 struct boundary{
169     PS::F64 x, y, z;
170 };
171
172
173 /* Force Functors */
174 class CalcDensity{
175     public:
176     void operator () (const EP* const ep_i, const PS::S32 Nip,
177                       const EP* const ep_j, const PS::S32 Njp,
178                       Dens* const dens){
179        for(PS::S32 i = 0 ; i < Nip ; ++i){
180           dens[i].clear();
181           for(PS::S32 j = 0 ; j < Njp ; ++j){
182              const PS::F64vec dr = ep_j[j].pos - ep_i[i].pos;
```

```
183             dens[i].dens += ep_j[j].mass * W(dr, ep_i[i].smth);
184          }
185       }
186    }
187 };
188
189 class CalcHydroForce{
190    public:
191    void operator () (const EP* const ep_i, const PS::S32 Nip,
192                      const EP* const ep_j, const PS::S32 Njp,
193                      Hydro* const hydro){
194       for(PS::S32 i = 0; i < Nip ; ++ i){
195          hydro[i].clear();
196          PS::F64 v_sig_max = 0.0;
197          for(PS::S32 j = 0; j < Njp ; ++j){
198             const PS::F64vec dr = ep_i[i].pos - ep_j[j].pos;
199             const PS::F64vec dv = ep_i[i].vel - ep_j[j].vel;
200             const PS::F64 w_ij = (dv * dr < 0) ? dv * dr / sqrt(dr * dr) :
                     0;
201             const PS::F64 v_sig = ep_i[i].snds + ep_j[j].snds - 3.0 * w_ij
                    ;
202             v_sig_max = std::max(v_sig_max, v_sig);
203             const PS::F64 AV = - 0.5 * v_sig * w_ij / (0.5 * (ep_i[i].dens
                     + ep_j[j].dens));
204             const PS::F64vec gradW_ij = 0.5 * (gradW(dr, ep_i[i].smth) +
                    gradW(dr, ep_j[j].smth));
205             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;
206             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;
207          }
208          hydro[i].dt = C_CFL * 2.0 * ep_i[i].smth / v_sig_max;
209       }
210    }
211 };
212
213 void makeOutputDirectory(char * dir_name) {
214     struct stat st;
215     PS::S32 ret;
216     if (PS::Comm::getRank() == 0) {
217         if (stat(dir_name, &st) != 0) {
218             ret = mkdir(dir_name, 0777);
219         } else {
220             ret = 0; // the directory named dir_name already exists.
221         }
222     }
223     PS::Comm::broadcast(&ret, 1);
224     if (ret == 0) {
225         if (PS::Comm::getRank() == 0)
226             fprintf(stderr, "Directory␣\"%s\"␣is␣successfully␣made.\n",
                    dir_name);
227     } else {
228         if (PS::Comm::getRank() == 0)
229             fprintf(stderr, "Directory␣%s␣fails␣to␣be␣made.\n", dir_name);
```

```
230         PS::Abort();
231     }
232 }
233
234 void SetupIC(PS::ParticleSystem<FP>& sph_system, PS::F64 *end_time,
        boundary *box){
235     // Place SPH particles
236     std::vector<FP> ptcl;
237     const PS::F64 dx = 1.0 / 128.0;
238     box->x = 1.0;
239     box->y = box->z = box->x / 8.0;
240     PS::S32 i = 0;
241     for(PS::F64 x = 0 ; x < box->x * 0.5 ; x += dx){
242        for(PS::F64 y = 0 ; y < box->y ; y += dx){
243           for(PS::F64 z = 0 ; z < box->z ; z += dx){
244              FP ith;
245              ith.pos.x = x;
246              ith.pos.y = y;
247              ith.pos.z = z;
248              ith.dens = 1.0;
249              ith.mass = 0.75;
250              ith.eng  = 2.5;
251              ith.id   = i++;
252              ith.smth = 0.012;
253              ptcl.push_back(ith);
254           }
255        }
256     }
257     for(PS::F64 x = box->x * 0.5 ; x < box->x * 1.0 ; x += dx * 2.0){
258        for(PS::F64 y = 0 ; y < box->y ; y += dx){
259           for(PS::F64 z = 0 ; z < box->z ; z += dx){
260              FP ith;
261              ith.pos.x = x;
262              ith.pos.y = y;
263              ith.pos.z = z;
264              ith.dens = 0.5;
265              ith.mass = 0.75;
266              ith.eng  = 2.5;
267              ith.id   = i++;
268              ith.smth = 0.012;
269              ptcl.push_back(ith);
270           }
271        }
272     }
273     for(PS::U32 i = 0 ; i < ptcl.size() ; ++ i){
274        ptcl[i].mass = ptcl[i].mass * box->x * box->y * box->z / (PS::F64)(
               ptcl.size());
275     }
276     std::cout << "#␣of␣ptcls␣is...␣" << ptcl.size() << std::endl;
277     // Scatter SPH particles
278     assert(ptcl.size() % PS::Comm::getNumberOfProc() == 0);
279     const PS::S32 numPtclLocal = ptcl.size() / PS::Comm::getNumberOfProc();
280     sph_system.setNumberOfParticleLocal(numPtclLocal);
281     const PS::U32 i_head = numPtclLocal * PS::Comm::getRank();
282     const PS::U32 i_tail = numPtclLocal * (PS::Comm::getRank() + 1);
```

```
283     for(PS::U32 i = 0 ; i < ptcl.size() ; ++ i){
284        if(i_head <= i && i < i_tail){
285           const PS::U32 ii = i - numPtclLocal * PS::Comm::getRank();
286           sph_system[ii] = ptcl[i];
287        }
288     }
289     // Set the end time
290     *end_time = 0.12;
291     // Fin.
292     std::cout << "setup..." << std::endl;
293 }
294
295 void Initialize(PS::ParticleSystem<FP>& sph_system){
296     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
297        sph_system[i].setPressure();
298     }
299 }
300
301 PS::F64 getTimeStepGlobal(const PS::ParticleSystem<FP>& sph_system){
302     PS::F64 dt = 1.0e+30; //set VERY LARGE VALUE
303     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
304        dt = std::min(dt, sph_system[i].dt);
305     }
306     return PS::Comm::getMinValue(dt);
307 }
308
309 void InitialKick(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
310     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
311        sph_system[i].vel_half = sph_system[i].vel + 0.5 * dt * sph_system[i
               ].acc;
312        sph_system[i].eng_half = sph_system[i].eng + 0.5 * dt * sph_system[i
               ].eng_dot;
313     }
314 }
315
316 void FullDrift(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
317     // time becomes t + dt;
318     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
319        sph_system[i].pos += dt * sph_system[i].vel_half;
320     }
321 }
322
323 void Predict(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
324     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
325        sph_system[i].vel += dt * sph_system[i].acc;
326        sph_system[i].eng += dt * sph_system[i].eng_dot;
327     }
328 }
329
330 void FinalKick(PS::ParticleSystem<FP>& sph_system, const PS::F64 dt){
331     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
332        sph_system[i].vel = sph_system[i].vel_half + 0.5 * dt * sph_system[i
               ].acc;
333        sph_system[i].eng = sph_system[i].eng_half + 0.5 * dt * sph_system[i
               ].eng_dot;
```

```
334     }
335 }
336
337 void setPressure(PS::ParticleSystem<FP>& sph_system){
338     for(PS::S32 i = 0 ; i < sph_system.getNumberOfParticleLocal() ; ++ i){
339         sph_system[i].setPressure();
340     }
341 }
342
343 void CheckConservativeVariables(const PS::ParticleSystem<FP>& sph_system){
344     PS::F64vec Mom=0.0; // total momentum
345     PS::F64    Eng=0.0; // total enegry
346     for(PS::S32 i = 0; i < sph_system.getNumberOfParticleLocal(); ++ i){
347         Mom += sph_system[i].vel * sph_system[i].mass;
348         Eng += (sph_system[i].eng + 0.5 * sph_system[i].vel * sph_system[i].
                vel)
349                * sph_system[i].mass;
350     }
351     Eng = PS::Comm::getSum(Eng);
352     Mom = PS::Comm::getSum(Mom);
353      if(PS::Comm::getRank() == 0){
354          printf("%.16e\n", Eng);
355          printf("%.16e\n", Mom.x);
356          printf("%.16e\n", Mom.y);
357          printf("%.16e\n", Mom.z);
358      }
359 }
360
361 int main(int argc, char* argv[]){
362     // Initialize FDPS
363     PS::Initialize(argc, argv);
364     // Make a directory
365     char dir_name[1024];
366     sprintf(dir_name,"./result");
367     makeOutputDirectory(dir_name);
368     // Display # of MPI processes and threads
369     PS::S32 nprocs = PS::Comm::getNumberOfProc();
370     PS::S32 nthrds = PS::Comm::getNumberOfThread();
371     std::cout << "========================================" << std::endl
372               << "␣This␣is␣a␣sample␣program␣of␣"             << std::endl
373               << "␣Smoothed␣Particle␣Hydrodynamics␣on␣FDPS!"  << std::endl
374               << "␣#␣of␣processes␣is␣" << nprocs              << std::endl
375               << "␣#␣of␣thread␣is␣␣␣␣" << nthrds              << std::endl
376               << "========================================" << std::endl
                  ;
377     // Make an instance of ParticleSystem and initialize it
378     PS::ParticleSystem<FP> sph_system;
379     sph_system.initialize();
380     // Define local variables
381     PS::F64 dt, end_time;
382     boundary box;
383     // Make an initial condition and initialize the particle system
384     SetupIC(sph_system, &end_time, &box);
385     Initialize(sph_system);
386     // Make an instance of DomainInfo and initialize it
```

```
387     PS::DomainInfo dinfo;
388     dinfo.initialize();
389     // Set the boundary condition
390     dinfo.setBoundaryCondition(PS::BOUNDARY_CONDITION_PERIODIC_XYZ);
391     dinfo.setPosRootDomain(PS::F64vec(0.0, 0.0, 0.0),
392                            PS::F64vec(box.x, box.y, box.z));
393     // Perform domain decomposition
394     dinfo.decomposeDomainAll(sph_system);
395     // Exchange the SPH particles between the (MPI) processes
396     sph_system.exchangeParticle(dinfo);
397     // Make two tree structures
398     // (one is for the density calculation and
399     //  another is for the force calculation.)
400     PS::TreeForForceShort<Dens, EP, EP>::Gather dens_tree;
401     dens_tree.initialize(3 * sph_system.getNumberOfParticleGlobal());
402
403     PS::TreeForForceShort<Hydro, EP, EP>::Symmetry hydr_tree;
404     hydr_tree.initialize(3 * sph_system.getNumberOfParticleGlobal());
405     // Compute density, pressure, acceleration due to pressure gradient
406     dens_tree.calcForceAllAndWriteBack(CalcDensity(), sph_system, dinfo);
407     setPressure(sph_system);
408     hydr_tree.calcForceAllAndWriteBack(CalcHydroForce(), sph_system, dinfo)
            ;
409     // Get timestep
410     dt = getTimeStepGlobal(sph_system);
411     // Main loop for time integration
412     PS::S32 step = 0;
413     for(PS::F64 time = 0 ; time < end_time ; time += dt, ++ step){
414        // Leap frog: Initial Kick & Full Drift
415        InitialKick(sph_system, dt);
416        FullDrift(sph_system, dt);
417        // Adjust the positions of the SPH particles that run over
418        // the computational boundaries.
419        sph_system.adjustPositionIntoRootDomain(dinfo);
420        // Leap frog: Predict
421        Predict(sph_system, dt);
422        // Perform domain decomposition again
423        dinfo.decomposeDomainAll(sph_system);
424        // Exchange the SPH particles between the (MPI) processes
425        sph_system.exchangeParticle(dinfo);
426        // Compute density, pressure, acceleration due to pressure gradient
427        dens_tree.calcForceAllAndWriteBack(CalcDensity(), sph_system, dinfo)
               ;
428        setPressure(sph_system);
429        hydr_tree.calcForceAllAndWriteBack(CalcHydroForce(), sph_system,
              dinfo);
430        // Get a new timestep
431        dt = getTimeStepGlobal(sph_system);
432        // Leap frog: Final Kick
433        FinalKick(sph_system, dt);
434        // Output result files
435        if(step % OUTPUT_INTERVAL == 0){
436           FileHeader header;
437           header.time = time;
438           header.Nbody = sph_system.getNumberOfParticleGlobal();
```

```
439             char filename[256];
440             sprintf(filename, "result/%04d.txt", step);
441             sph_system.writeParticleAscii(filename, header);
442             if (PS::Comm::getRank() == 0){
443                 std::cout << "================================" << std::endl;
444                 std::cout << "output␣" << filename << "." << std::endl;
445                 std::cout << "================================" << std::endl;
446             }
447         }
448         // Output information to STDOUT
449         if (PS::Comm::getRank() == 0){
450             std::cout << "================================" << std::endl;
451             std::cout << "time␣=␣" << time << std::endl;
452             std::cout << "step␣=␣" << step << std::endl;
453             std::cout << "================================" << std::endl;
454         }
455         CheckConservativeVariables(sph_system);
456     }
457     // Finalize FDPS
458     PS::Finalize();
459     return 0;
460 }
```

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。

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

サンプルコードは `main.cpp` と GCC 用の Makefile である `Makefile` からなる。

6.1.2 ユーザー定義型

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。

6.1.2.1 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

```
1  class Nbody_FP
2  {
3     public:
4        PS::S64 id;
5        PS::F64 m;
6        PS::F64 rc;
7        PS::F64vec x;
8        PS::F64vec v,v_half;
9        PS::F64vec agrv;
10       PS::F64 pot;
11       // Member functions required by FDPS
12       PS::F64 getCharge() const {
13          return m;
14       };
15       PS::F64 getChargeParticleMesh() const {
16          return m;
17       };
18       PS::F64vec getPos() const {
19          return x;
20       };
21       PS::F64 getRSearch() const {
22          return rc;
23       };
24       void setPos(const PS::F64vec& x) {
25          this->x = x;
26       };
27       void copyFromForce(const Nbody_PP_Results& result) {};
28       void copyFromForceParticleMesh(const PS::F64 apm) {};
29 };
```

6.1.2.2　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 };
```

6.1.2.3　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

```
1 class Nbody_PP_Results
2 {
3    public:
4       PS::F64 pot;
5       PS::F64vec agrv;
6       void clear() {
7          pot  = 0.0;
8          agrv = 0.0;
9       }
10 };
```

6.1.2.4　相互作用関数 calcForceEpEp

ユーザーは相互作用関数 calcForceEpEp を記述しなければならない。calcForceEpEp 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。calcForceEpEp は、`void` 関数或いは、ファンクタ (関数オブジェクト) として実装されなければならない。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。本サンプルコードの calcForceEpEp の実装を 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。

Listing 36: 相互作用関数 calcForceEpEp

```
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。

6.1.2.5　相互作用関数 calcForceEpSp

ユーザーは相互作用関数 calcForceEpSp を記述しなければならない[注 2)]。calcForceEpSp には、粒子-超粒子間相互作用計算の具体的な内容を書く必要がある。calcForceEpSp は、`void` 関数 或いは、ファンクタ (関数オブジェクト) として実装されなければならない。引数は、EssentialParticleI の配列、EssentialParticleI の個数、超粒子型の配列、超粒子型の個数、Force 型の配列である。本サンプルコードの calcForceEpSp の実装を Listing 37 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。

Listing 37: 相互作用関数 calcForceEpSp

```
1 class Calc_force_ep_sp{
2    public:
3       void operator () (const Nbody_EP* const ep_i,
4                         const PS::S32 Nip,
5                         const PS::SPJMonopoleCutoff* const ep_j,
6                         const PS::S32 Njp,
7                         Nbody_PP_Results* const result) {
8          for (PS::S32 i=0; i<Nip; i++) {
9             for (PS::S32 j=0; j<Njp; j++) {
```

注 2) 冒頭で述べたように、本サンプルコードでは相互作用計算に $\rm P^3M$ 法を用いる。FDPS の枠組み内で、これを実現するため、後述するように、見込み角の基準値 θ を 0 に指定して相互作用計算を行う。このため、粒子-超粒子相互作用は発生しないはずである。しかしながら、API `calcForceAllAndWriteBack` に、粒子-超粒子間相互作用を計算する関数のアドレスを渡す必要があるため、相互作用関数 calcForceEpSp を定義する必要がある。

```
10                  PS::F64vec dx = ep_i[i].x - ep_j[j].pos;
11                  PS::F64 rij = std::sqrt(dx * dx);
12                  PS::F64 rinv = 1.0/rij;
13                  PS::F64 rinv3 = rinv*rinv*rinv;
14                  PS::F64 xi = 2.0*rij/ep_i[i].rc;
15                  result[i].pot  += ep_j[j].mass * S2_pcut(xi) * rinv;
16                  result[i].agrv += ep_j[j].mass * S2_fcut(xi) * rinv3 * dx;
17              }
18          }
19
20      };
21  };
```

6.1.3　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 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^3M 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　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 38: Include FDPS

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

```
3 #include <param.h>
4 #include <param_fdps.h>
```

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 39: 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 40: 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^3M 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 41: `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 42: `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) 粒子群オブジェクトの初期化 粒子群オブジェクトの初期化は、以下のように行う:

Listing 43: 粒子群オブジェクトの初期化

```
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) 領域オブジェクトの初期化 領域オブジェクトの初期化は、以下のように行う:

Listing 44: 領域オブジェクトの初期化

```
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、領域オブジェクト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 void 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) ツリーオブジェクトの初期化 相互作用ツリーオブジェクトの初期化も、API initialize を使って、以下のように行う:

Listing 45: ツリーオブジェクトの初期化

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

ツリーオブジェクトの API initialize 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 API の第 1 引数として指定する。上記の例では、API 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、API 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 パートの計算で粒子-超粒子相互作用を発生させないようにするため、$\theta = 0$ を指定している。

本サンプルコードでは、ツリーオブジェクトの初期化は 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 オブジェクトの初期化 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 void 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 void 関数は、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クラス Crystal_Parameters 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、領域オブジェクトの API `decomposeDomainAll` を使用する:

Listing 46: 領域分割の実行

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

ここで、粒子分布の情報を領域オブジェクトに与えるため、引数に粒子群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、粒子群オブジェクトの API `exchangeParticle` を使用する:

Listing 47: 粒子交換の実行

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

ここで領域情報を粒子群オブジェクトに与えるため、引数に領域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 48: 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 49: 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 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、API setDomainInfoParticleMesh と setParticleParticleMesh 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 API calcMeshForceOnly で行う。Force 計算の結果を取得するため、API getPotential と API getForce 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 50: 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 51: PP パートの計算

```
1 pp_tree.calcForceAll(Calc_force_ep_ep(),
2                       Calc_force_ep_sp(),
3                       system, dinfo);
4 for (PS::S32 i=0; i<numPtclLocal; i++) {
5    Nbody_PP_Results result = pp_tree.getForce(i);
6    system[i].pot  += result.pot;
7    system[i].agrv += result.agrv;
8 }
```

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。

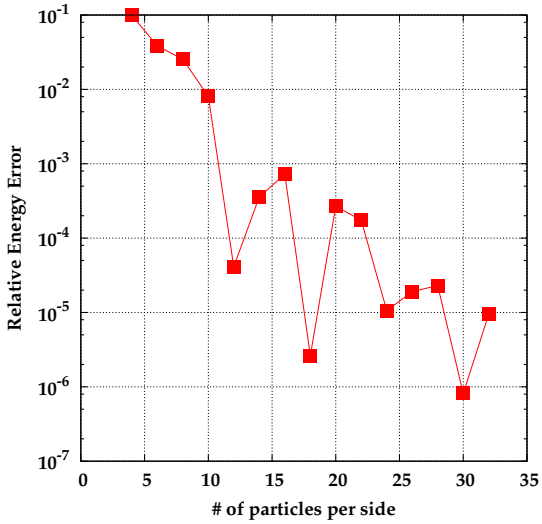

図 3: 1 辺あたりの粒子数とエネルギー相対誤差の関係 (メッシュ数は 16^3、カットオフ半径は 3/16)

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` 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
$ ls | awk '{print $0}'
IC/
Makefile
README_en.txt
README_ja.txt
constants.hpp
cosmology.hpp
fig/
make_directory.c
param_file_for_test.txt
prototype.h
result/
run_param.hpp
test.py*
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 52: 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 53 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 53: 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);
153         }
154     }
155
156     // for API of FDPS
157     void writeBinary(FILE *fp){
158         static const PS::F64 Mpc_m = 3.08567e22; // unit is m
159         static const PS::F64 Mpc_km = 3.08567e19; // unit is km
160         static const PS::F64 Msun_kg = 1.9884e30; // unit is kg
161         static const PS::F64 G = 6.67428e-11; // m^3*kg^-1*s^-2
162         static const PS::F64 Cl = FPtreepm::Lbnd;
163         static const PS::F64 Cv = (FPtreepm::Lbnd * FPtreepm::H0);
164         static const PS::F64 Cm = (pow(Mpc_m*FPtreepm::Lbnd, 3.0) / pow(
                Mpc_km/FPtreepm::H0, 2.0) / G / Msun_kg);
165         PS::F32vec x = pos * Cl;
166         PS::F32vec v = vel * Cv;
167         PS::F32 m = mass * Cm;
168         PS::S32 i = id;
169         fwrite(&x.x,   sizeof(PS::F32), 1, fp);
170         fwrite(&v.x,   sizeof(PS::F32), 1, fp);
171         fwrite(&x.y,   sizeof(PS::F32), 1, fp);
```

```
172         fwrite(&v.y,    sizeof(PS::F32), 1, fp);
173         fwrite(&x.z,    sizeof(PS::F32), 1, fp);
174         fwrite(&v.z,    sizeof(PS::F32), 1, fp);
175         fwrite(&m,    sizeof(PS::F32), 1, fp);
176         fwrite(&i,    sizeof(PS::S32), 1, fp);
177     }
178
179     PS::F64 calcDtime(run_param &this_run) {
180     PS::F64 dtime_v, dtime_a, dtime;
181     PS::F64 vnorm, anorm;
182     vnorm = sqrt(SQR(this->vel))+TINY;
183     anorm = sqrt(SQR(this->acc+this->acc_pm))+TINY;
184
185     dtime_v = this->eps/vnorm;
186     dtime_a = sqrt(this->eps/anorm)*CUBE(this_run.anow);
187
188     dtime = fmin(0.5*dtime_v, dtime_a);
189
190     return dtime;
191   }
192 };
```

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 54 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 54: EssentialParticleI 型

```
1  class EPItreepm {
2  public:
3    PS::S64    id;
4    PS::F32    eps;
5    PS::F64vec pos;
6
7    PS::F64vec getPos() const {
8      return this->pos;
9    }
10
11   void copyFromFP(const FPtreepm & fp) {
12     this->id = fp.id;
13     this->eps = fp.eps;
14     this->pos = fp.pos;
15   }
16
17 };
```

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 55 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 55: EssentialParticleJ 型

```
1  class EPJtreepm {
2  public:
3    PS::S64    id;
4    PS::F64vec pos;
5    PS::F64    mass;
6    //  PS::F64    rcut;
7
8    PS::F64vec getPos() const {
9      return this->pos;
10   }
11
12   PS::F64 getCharge() const {
13     return this->mass;
14   }
15
16   void copyFromFP(const FPtreepm & fp) {
17     this->id = fp.id;
18     this->mass = fp.mass;
19     this->pos = fp.pos;
20   }
21
22   PS::F64 getRSearch() const {
23     PS::F64 rcut = 3.0/SIZE_OF_MESH;
24     return rcut;
25   }
26
27   void setPos(const PS::F64vec pos_new) {
28     this->pos = pos_new;
29   }
30 };
```

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 56 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 56: 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 57 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 57: calcForceEpEp 型

```
1  template <class TPJ>
2  class calc_pp_force {
3  public:
4      void operator () (EPItreepm *iptcl,
5                        const PS::S32 ni,
6                        TPJ *jptcl,
7                        const PS::S32 nj,
8                        Result_treepm *ppforce) {
9          for (PS::S32 i=0;i < ni;i++) {
10             PS::F64 eps2 = SQR(iptcl[i].eps);
11             for (PS::S32 j=0; j < nj;j++) {
12                 PS::F64vec dr = iptcl[i].pos - jptcl[j].pos;
13                 PS::F64 rsq = dr*dr;
14                 PS::F64 rad = sqrt(rsq+eps2);
15                 PS::F64 gfact = gfactor_S2(rad, 3.0/SIZE_OF_MESH);
16                 PS::F64 rinv  = 1.0/rad;
17                 PS::F64 mrinv3 = jptcl[j].mass*CUBE(rinv);
18                 ppforce[i].acc -= dr*gfact*mrinv3;
```

```
19                }
20            }
21        }
22    };
```

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。6.1.2.4 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、`$(FDPS)/sample/c++/treepm/README_ja.txt` で説明されているので、そちらを参照されたい。また、`$(FDPS)/sample/c++/treepm/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 58: 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 59: 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 60: 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 61: 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 62: 領域分割の実行

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

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 63: 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 64: 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、4.1.3.4.4 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、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^3M 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。

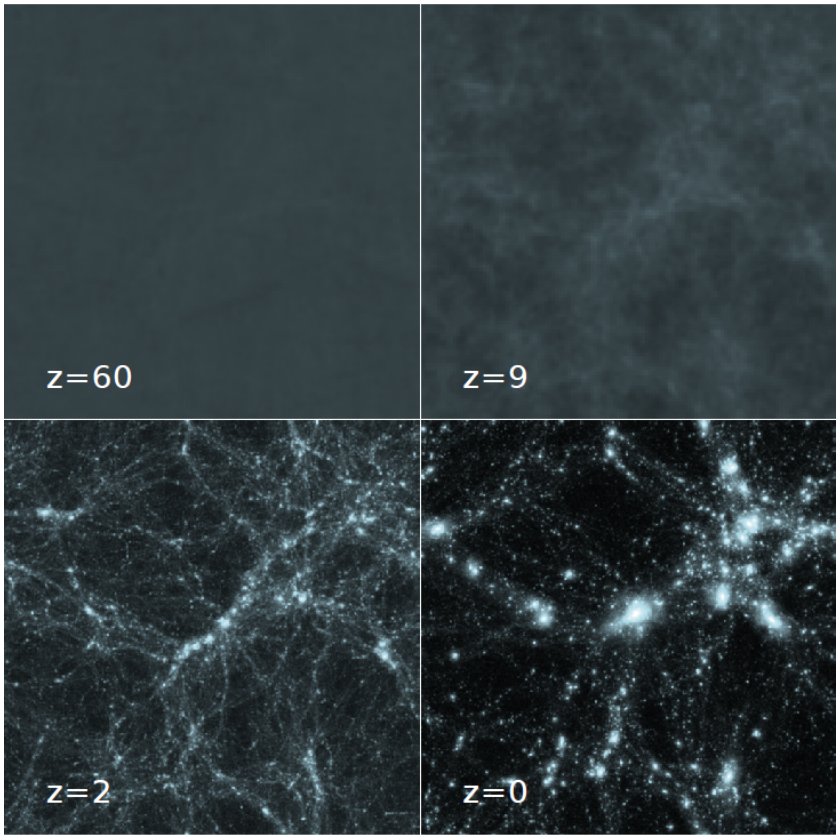

図 4: Santa Barbara Cluster Comparison テストの密度分布の時間発展 (粒子数 256^3)

7　より実用的なアプリケーションの解説

前節までは、比較的単純なサンプルコードを用いて、FDPS の基本的な機能について解説を行ってきた。しかしながら、実際の研究では、複数の粒子種の取り扱いが必要になる等、より複雑なアプリケーションを書く必要がある。そこで、本節では、より実用的なサンプルコードを使って、FDPS の他の機能について解説を行う。記述を簡潔にするため、読者は前節までの内容を理解しているものと仮定する。

7.1　N 体/SPH コード

本節では、より実用的なアプリケーションの一例として円盤銀河の時間発展を計算する N 体/SPH サンプルコードの解説を行う。このサンプルコードは、重力のみで相互作用するダークマターと星を N 体粒子、重力及び流体相互作用を行うガスを SPH 粒子として表現する。重力計算は Tree 法で、流体計算は SPH 法を用いて行う。SPH 法は Springel & Hernquist [2002, MNRAS, 333, 649] 及び Springel [2005, MNRAS, 364, 1105] で提案された方法 (以下、簡単のため、Springel の方法と呼ぶ) を使用している。本解説を読むことにより、ユーザは、FDPS で複数の粒子種を扱う方法を学ぶことができる。

以下では、まずはじめにコードの使用方法について説明を行う。次に Springel の方法について簡単な解説を行った後、サンプルコードの中身について具体的に解説する。

7.1.1　コードの使用方法

前節で述べた通り、本コードは円盤銀河の N 体/SPH シミュレーションを行うコードである。ダークマターと星の初期条件は、銀河の初期条件を作成するソフトウェア MAGI (Miki & Umemura [2018, MNRAS, 475, 2269]) で作成したファイルを読み込んで設定する。一方、ガスの初期条件は本コードの内部で作成する。したがって、以下の手順で本コードを使用する。

- ディレクトリ `$(FDPS)/sample/c++/nbody+sph` に移動
- カレントディレクトリにある `Makefile` を編集
- MAGI を使って初期条件に対応する粒子データを生成し、`./magi_data/dat` 以下に配置
- コマンドライン上で `make` を実行
- `nbodysph.out` ファイルの実行
- 結果の解析

以下、順に説明していく。

7.1.1.1　ディレクトリ移動

サンプルコードの場所は、$(FDPS)/sample/c++/nbody+sph である。まずは、そこに移動する。

7.1.1.2　サンプルコードのファイル構成

以下はサンプルコードのファイル構成である。

```
$ ls | awk '{print $0}'
Makefile
Makefile.K
Makefile.ofp
ic.hpp
job.K.sh
job.ofp.sh
leapfrog.hpp
macro_defs.hpp
magi_data/
main.cpp
mathematical_constants.cpp
mathematical_constants.h
physical_constants.cpp
physical_constants.h
test.py*
user_defined.hpp
```

各ソースファイルの内容について簡単に解説を行う。まず、ic.hpp には初期条件を作成する関数が実装されている。初期条件は円盤銀河の他、複数用意されている (後述)。leapfrog.hpp には粒子の軌道の時間積分を Leapfrog 法を用いて行う関数が実装されている。macro_defs.hpp には計算を制御するためのマクロが定義されている。main.cpp はアプリケーションのメインルーチンが実装されている。mathematical_constants.h 及び mathematical_constants.cpp には数学定数が、physical_constants.h 及び physical_constants.cpp には物理定数が定義されている。user_defined.hpp には、ユーザ定義クラスや相互作用関数が実装されている。

ディレクトリ magi_data には、銀河の初期条件を作成するソフトウェア MAGI に入力するパラメータファイル (magi_data/cfg/*) と MAGI を動作させるスクリプト (magi_data/sh/run.sh) が格納されている。

7.1.1.3　Makefile の編集

Makefile の編集項目は以下の通りである。

- 変数 CXX に使用する C++コンパイラを代入する。
- 変数 CXXFLAGS に C++コンパイラのコンパイルオプションを指定する。
- 本コードでは計算を制御するためにいくつかのマクロを用意している。表 1 にマクロ名とその定義の対応を示した。また、本コードには、INITIAL_CONDITION の値に応じて自動的に設定されて使用されるマクロも存在する。これらは一般に変更する必要はないが、詳細は macro_defs.h を参照して頂きたい。
- 本コードでは重力計算に x86 版 Phantom-GRAPE ライブラリを使用することができる。Phantom-GRAPE ライブラリを使用する場合、Makefile の変数 use_phantom_grape_x86 の値を yes にする。

OpenMP や MPI の使用/不使用の指定に関しては、第 3 節を参照して頂きたい。

マクロ名	定義
INITIAL_CONDITION	初期条件の種類の指定、或いは、コードの動作の指定に使用されるマクロ。0 から 3 までのいずれかの値を取る必要がある。値に応じて、次のように指定される。 0:円盤銀河の初期条件を選択、1:Cold collapse 問題の初期条件を選択、2:Evrard test 問題の初期条件を選択、3:ガラス状に分布した SPH 粒子データを生成するモードで実行ファイルを作成
ENABLE_VARIABLE_SMOOTHING_LENGTH	smoothing length が可変/固定を制御するマクロ。定義されている場合、可変となり Springel の方法で SPH 計算が行われる。未定義の場合、固定長カーネルの SPH コードとなる。
USE_ENTROPY	流体の熱力学状態を記述する独立変数としてエントロピーを使うか単位質量あたりの内部エネルギーを使用するかを指定するマクロ。定義されている場合エントロピーが用いられる。但し、後述するマクロ ISOTHERMAL_EOS が定義されている場合には、単位質量あたりの内部エネルギーが強制的に使用される (圧力の計算に内部エネルギーを使用する)。
USE_BALSARA_SWITCH	Balsara switch (Balsara [1995, JCP, 121, 357]) の使用/不使用を制御するマクロ。定義されている場合は使用する。
USE_PRESCR_OF_THOMAS_COUCHMAN_1992	Thomas & Couchman [1992, MNRAS,257, 11] で提案された SPH 計算の tensile 不安定を防ぐ簡便な方法を使用するかを制御するマクロ。定義されている場合は使用する。
ISOTHERMAL_EOS	流体を等温で取り扱うかどうかを指定するマクロ。定義されている場合は等温で扱い、未定義の場合にはエントロピー方程式、或いは、内部エネルギー方程式が解いて、熱力学的状態を時間発展させる。
READ_DATA_WITH_BYTESWAP	MAGI の粒子データを読み込む際に基本データ型単位で byteswap してデータを読み込むかを制御するマクロ。定義されている場合は byteswap する。

表 1: コンパイル時マクロの種類と定義

7.1.1.4　MAGI を使った粒子データの生成

前述した通り、ユーザは事前に銀河の初期条件を作成するソフトウェア MAGI を使い、以下に指定する手順でデータを作成する必要がある。MAGI を利用できないユーザは、指定するサイトからこちらが用意したデータをダウンロードすることも可能である。以下、各場合について詳しく述べる。

MAGI を使ってデータ作成を行う場合 以下の手順でデータ作成を行う。

1. https://bitbucket.org/ymiki/magi から MAGI をダウンロードし、Web の “How to compile MAGI” に記載された手順に従って、適当な場所にインストールする。但し、本サンプルコードは TIPSY ファイルの粒子データ読み込みしかサポートしていないため、MAGI は `USE_TIPSY_FORMAT=ON` の状態でビルドされている必要がある。
2. `./magi_data/sh/run.sh` を開き、変数 `MAGI_INSTALL_DIR` にコマンド `magi` がインストールされたディレクトリを、変数 `NTOT` に希望する N 体粒子の総数をセットする (ダークマターと星への振り分けは MAGI が自動的に行う)。
3. `./magi_data/cfg/*`を編集し、ダークマターと銀河のモデルを指定する。指定方法の詳細は上記サイトか、或いは、Miki & Umemura [2018, MNRAS, 475, 2269] の第 2.4 節を参照のこと。デフォルトの銀河モデル (以下、**デフォルトモデル**) は次の 4 成分から構成される:
   (i) ダークマターハロー (NFW profile, $M = 10^{12}$ M$_\odot$, $r_s = 21.5$ kpc, $r_c = 200$ kpc, $\Delta_c = 10$ kpc)
   (ii) バルジ (King モデル, $M = 5 \times 10^{10}$ M$_\odot$, $r_s = 0.7$ kpc, $W_0 = 5$)
   (iii) thick disk (Sérsic profile, $M = 2.5 \times 10^{10}$ M$_\odot$, $r_s = 3.5$ kpc, $n = 1.5$, $z_d = 1$ kpc, $Q_{T,\mathrm{min}} = 1.0$)
   (iv) thin disk (exponential disk, $M = 2.5 \times 10^{10}$ M$_\odot$, $r_s = 3.5$ kpc, $z_d = 0.5$ kpc, $Q_{T,\mathrm{min}} = 1.0$)

   デフォルトモデルでは、2 つの星円盤は bar モードに対してわずかに不安定であるため、弱い棒状構造を持つ渦巻き銀河になることが期待される初期条件となっている。MAGI の最新のリリース (version 1.1.1 [2019 年 7 月 19 日時点]) では、従来のリリースとデフォルトの動作モードが変更されたため、thick disk と thin disk のパラメータ指定の仕方が FDPS 5.0d 以前から変わっていることに注意されたい。具体的には、従来の MAGI では disk 成分の速度分散をパラメータ f を通して指定する形になっていたが (FDPS 5.0d 以前に付属するサンプルコードでは、$f = 0.125$)、最新のリリースでは Toomore Q value の最小値 $Q_{T,\mathrm{min}}$ を通して指定する方式になっている。
4. ディレクトリ `magi_data` に移動し、以下のコマンドを実行:

```
$ ./sh/run.sh
```

5. MAGI が正しく終了しているなら、magi_data/dat 以下に、拡張子が tipsy の粒子データが生成されているはずである。

データをダウンロードする場合 以下のサイトからダウンロードし、./magi_data/dat/以下に置く。各粒子データの銀河モデルはすべてデフォルトモデルで、粒子数だけ異なる。

- $N = 2^{21}$: http://particle.riken.jp/~fdps/magi_data/Galaxy/21/Galaxy.tipsy
- $N = 2^{22}$: http://particle.riken.jp/~fdps/magi_data/Galaxy/22/Galaxy.tipsy
- $N = 2^{23}$: http://particle.riken.jp/~fdps/magi_data/Galaxy/23/Galaxy.tipsy
- $N = 2^{24}$: http://particle.riken.jp/~fdps/magi_data/Galaxy/24/Galaxy.tipsy

7.1.1.5　make の実行

make コマンドを実行する。

7.1.1.6　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

```
$ ./nbodysph.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 ./nbodysph.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。

7.1.1.7　結果の解析

ディレクトリ result に N 体粒子と SPH 粒子の粒子データファイル “nbody0000x.dat” と “sph0000x.dat” が出力される。ここで x は時刻に対応する整数である。　N 体粒子データの出力フォーマットは、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 軸方向の速度となっている。一方、SPH 粒子データの出力フォーマットは、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、単位質量あたりの内部エネルギー、エントロピー、圧力となっている。

図 5 は、N 体粒子数 2^{21}、SPH 粒子数 2^{18} で円盤銀河のシミュレーションを行ったときの $T = 0.46$ における星分布とガス分布である。

以下では、まず Springel の方法について解説し、その後、サンプルコードの実装について説明していく。

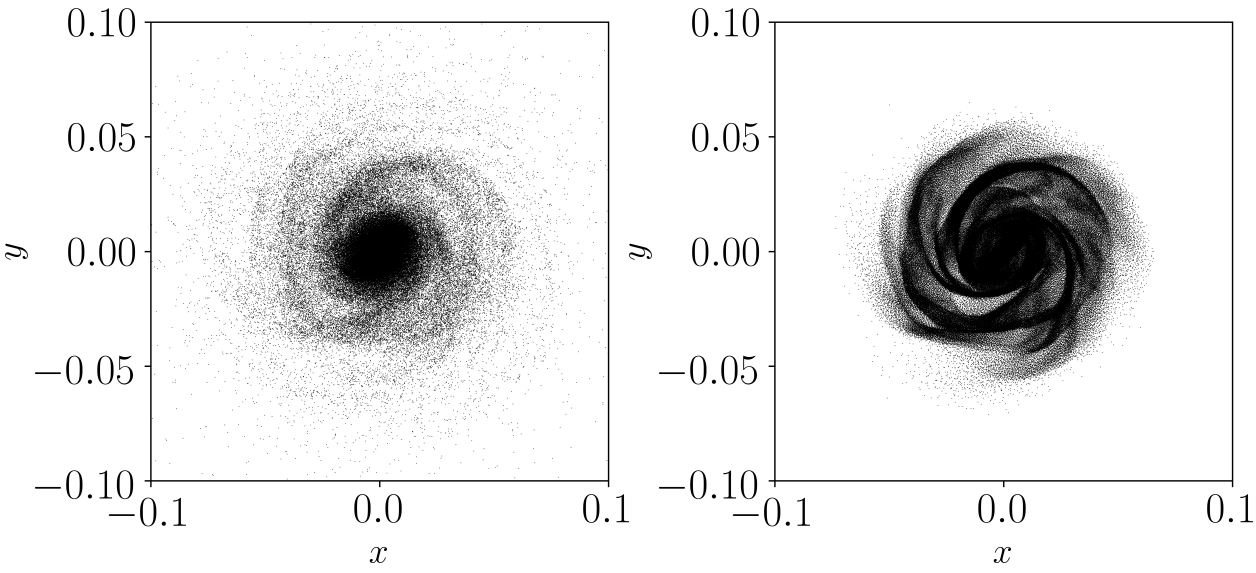

図 5: $T = 0.46$ における星分布 (左) とガス分布 (右) (計算は以下の条件で実施した: N 体粒子数 2^{21}、SPH 粒子数 2^{18}、等温、ガス温度 10^4 K、平均分子量 $\mu = 0.5$)

7.1.2　Springel の方法

Springel & Hernquist [2002, MNRAS, 333, 649] では、smoothing length が可変な場合でも、系のエネルギーとエントロピーが保存するようなスキーム (具体的には運動方程式) を定式化した。以下、彼らの定式化を手短に説明する。導出方針としては、smoothing length も独立変数とみて系の Lagrangian を立て、それを粒子数個の拘束条件の下、Euler-Lagrange 方程式を解く、というものである。

具体的には、彼らは系の Lagrangian として次のようなもの選んだ:

$$L(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \frac{1}{2}\sum_{i=1}^{N} m_i \dot{\boldsymbol{r}}_i^2 - \frac{1}{\gamma - 1}\sum_{i=1}^{N} m_i A_i \rho_i^{\gamma-1} \tag{7}$$

ここで、$\boldsymbol{q} = (\boldsymbol{r}_1, ..., \boldsymbol{r}_N, h_1, ...h_N)$ であり、下付きの整数はすべて粒子番号を表す。$\boldsymbol{r}_i$ は位置、h_i は smoothing length、m_i は質量、γ は比熱比、ρ_i は密度、A_i はエントロピー関数と呼ばれ、単位質量あたりの内部エネルギー u_i と次の関係がある:

$$u_i = \frac{A_i}{\gamma - 1}\rho_i^{\gamma-1} \tag{8}$$

式 (7) の第 1 項目は運動エネルギー、第 2 項目は内部エネルギーを表す。この Lagrangian をそのまま Euler-Lagrangian 方程式を使って解くと、$4N$ 個の方程式になってしまうので、彼らは次の N 個の拘束条件を導入した。

$$\phi_i = \frac{4\pi}{3} h_i^3 \rho_i - \overline{m} N_{\mathrm{neigh}} = 0 \tag{9}$$

ここで、$\overline{m}$ は SPH 粒子の平均質量[注 3)]、N_{neigh} は近傍粒子数 (定数) である。この拘束条件の下、Lagrange の未定乗数法を使って、Euler-Lagrange 方程式をとけば、以下の運動方程式が

[注 3)] 拘束条件に使用していることから、定数扱いであることに注意。

得られる:

$$\frac{\mathrm{d}\boldsymbol{v}_i}{\mathrm{d}t} = -\sum_{j=1}^{N} m_j \left[f_i \frac{P_i}{\rho_i^2} \nabla_i W(r_{ij}, h_i) + f_j \frac{P_j}{\rho_j^2} \nabla_i W(r_{ij}, h_j) \right] \tag{10}$$

ここで、P_i は圧力、$r_{ij} = |\boldsymbol{r}_i - \boldsymbol{r}_j|$、$W$ はカーネル関数、f_i は ∇h term と呼ばれる量で、

$$f_i = \left(1 + \frac{h_i}{3\rho_i} \frac{\partial \rho_i}{\partial h_i} \right)^{-1} \tag{11}$$

と定義される。

系の熱力学的状態はエントロピー A_i を独立変数として記述される。断熱過程の場合、エントロピーは衝撃波以外のところでは流れに沿って一定である。Springel [2005, MNRAS, 364, 1105] では、衝撃波でのエントロピー増加と速度の変化を人工粘性を使って次のようにモデル化している:

$$\frac{\mathrm{d}A_i}{\mathrm{d}t} = \frac{1}{2} \frac{\gamma - 1}{\rho_i^{\gamma - 1}} \sum_{j=1}^{N} m_j \Pi_{ij} \boldsymbol{v}_{ij} \cdot \nabla_i \overline{W}_{ij} \tag{12}$$

$$\left. \frac{\mathrm{d}\boldsymbol{v}_i}{\mathrm{d}t} \right|_{\mathrm{visc}} = -\sum_{j=1}^{N} m_j \Pi_{ij} \nabla_i \overline{W}_{ij} \tag{13}$$

ここで、$\boldsymbol{v}_{ij} = \boldsymbol{v}_i - \boldsymbol{v}_j$、$\boldsymbol{v}_i$ は速度、$\overline{W}_{ij} = \frac{1}{2}(W(r_{ij}, h_i) + W(r_{ij}, h_j))$ である。Π_{ij} に関しては、原論文を参照して頂きたい。

したがって、SPH 計算の手順は次のようになる:

(1) 式 (9) と以下の式を無矛盾に解き、密度 ρ_i と h_i を決定する。

$$\rho_i = \sum_{j=1}^{N} m_j W(r_{ij}, h_i) \tag{14}$$

(2) 式 (11) で定義される ∇h term を計算する。
(3) 式 (10)、(12)、(13) の右辺を計算する。
(4) SPH 粒子の位置、速度、エントロピーを時間積分する。

以下、まずユーザ定義クラスと相互作用関数の実装について解説を行い、次にメインルーチンの実装について解説を行う。複数粒子種の取扱は後者で解説する。

7.1.3　ユーザー定義型

本サンプルコードのユーザ定義型はすべて `user_defined.hpp` に定義されている。はじめに用意されているユーザ定義型の種類について簡単に説明しておく。冒頭で述べたように、本サンプルコードは 2 種類の粒子 (N 体粒子, SPH 粒子) を扱う。そのため、FullParticle 型も 2 種類用意している (N 体粒子用に `FP_nbody` クラス を, SPH 粒子用に `FP_sph` クラス)。相互作用は重力相互作用と流体相互作用の 2 種類ある。そのため、Force 型を 3 種類用意している

(重力計算用に Force_grav クラス を、密度計算用に Force_dens クラス を、そして、圧力勾配による加速度 (以下、単に圧力勾配加速度) の計算用に Force_hydro クラス を; 第 4 節も参照のこと)。本サンプルコードでは、簡単のため、EssentialParticleI 型と EssentialParticleJ 型を 1 つのクラスで兼ねることにし (以下、単に EssentialParticle 型)、密度計算と圧力勾配加速度計算に同じ EssentialParticle 型を使用する。したがって、EssentialParticle 型の種類は 2 種類となっている (重力計算用に EP_grav クラス を、SPH 計算用に EP_hydro クラス)。

以下、各ユーザ定義型の実装について説明する。

7.1.3.1　FullParticle 型

まず、N 体粒子用の FullParticle 型である FP_nbody クラス について解説する。このクラスには、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 65 に、このクラスの実装を示す。メンバ変数 およびメンバ関数 の構成は、第 3-4 節で紹介した N 体計算サンプルコードとほぼ同じであり、詳細はそちらを参照されたい。

Listing 65: FullParticle 型 (FP_nbody クラス)

```
1  class FP_nbody{
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     PS::F64vec getPos() const {
11         return pos;
12     }
13     PS::F64 getCharge() const {
14         return mass;
15     }
16     void setPos(const PS::F64vec& pos){
17        this->pos = pos;
18     }
19     void copyFromForce(const Force_grav & f) {
20         this->acc = f.acc;
21         this->pot = f.pot;
22     }
23     void writeAscii(FILE* fp) const {
24         fprintf(fp, "%lld\t%g\t%g\t%g\t%g\t%g\t%g\t%g\n",
25                 this->id, this->mass,
26                 this->pos.x, this->pos.y, this->pos.z,
27                 this->vel.x, this->vel.y, this->vel.z);
28     }
29     void readAscii(FILE* fp) {
30         fscanf(fp, "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\n",
31                &this->id, &this->mass,
32                &this->pos.x, &this->pos.y, &this->pos.z,
33                &this->vel.x, &this->vel.y, &this->vel.z);
34     }
```

```
35 };
```

次に、SPH 粒子用の FullParticle 型である FP_sph クラス について解説する。このクラスには、SPH 粒子が持っているべき全ての物理量が含まれている。Listing 66 に、このクラスの実装を示す。メンバ変数の内、主要な変数の意味は次の通りである: id (識別番号)、mass (質量)、pos (位置 $[\boldsymbol{r}_i]$)、vel (速度 $[\boldsymbol{v}_i]$)、acc_grav (重力加速度)、pot_grav (重力ポテンシャル)、acc_hydro (圧力勾配加速度)、dens (密度 $[\rho_i]$)、eng (単位質量あたりの内部エネルギー $[u_i]$)、ent (エントロピー関数 [以下、単にエントロピー]$[A_i]$)、pres (圧力 $[P_i]$)、smth (smoothing length[注 4)] $[h_i]$)、gradh (∇h term$[f_i]$)、divv ($(\nabla \cdot \boldsymbol{v})_i$、ここで下付きの i は粒子 i の位置での微分を示している)、rotv ($(\nabla \times \boldsymbol{v})_i$)、 BalSW (Balsara switch のための係数で、定義式は Balsara [1995, JCP, 121, 357] の $f(a)$)、snds (音速)、eng_dot (eng の時間変化率)、ent_dot (ent の時間変化率)、dt (この粒子の軌道を時間積分するときに許される最大の時間刻み幅)。

メンバ関数の構成は第 3-4 節で紹介した SPH サンプルコードと似ているが、以下の相違点がある:

- SPH 粒子が関わる相互作用計算は、重力計算、密度計算、圧力勾配加速度計算の 3 種類あるので、それに応じて copyFromForce も 3 つ用意されている。
- メンバ関数 writeBinaryPos の存在。この関数は、本サンプルコードのガラス状分布の SPH データを取得するためのモードで使用される (詳細は後述)。
- メンバ関数 setEntropy の存在。この関数はエントロピーの初期値を設定するのに使用される。

他のメンバ関数については、第 3-4 節を参照されたい。マクロについては、後ほど、まとめて解説する。

Listing 66: FullParticle 型 (FP_sph クラス)

```
 1 class FP_sph {
 2 public:
 3     PS::S64    id;
 4     PS::F64    mass;
 5     PS::F64vec pos;
 6     PS::F64vec vel;
 7     PS::F64vec acc_grav; // gravitational acceleration
 8     PS::F64    pot_grav; // gravitational potential
 9     PS::F64vec acc_hydro; // acceleration due to pressure-gradient
10     PS::S32    flag;
11     PS::F64    dens; // mass density
12     PS::F64    eng; // specific internal energy
13     PS::F64    ent; // entropy
14     PS::F64    pres; // pressure
15     PS::F64    smth; // smoothing length
16     PS::F64    gradh; // grad-h term
17     PS::F64    divv; // divergence of velocity
18     PS::F64vec rotv; // rotation of velocity
19     PS::F64    BalSW; // Balsara switch
```

注 4) カーネル関数が 0 になる距離と定義。

```
20      PS::F64    snds; // sound speed
21      PS::F64    eng_dot; // time rate of change of ‘eng‘
22      PS::F64    ent_dot; // time rate of change of ‘ent‘
23      PS::F64    dt; // hydrodynamic time step for this particle
24      PS::F64vec vel_half;
25      PS::F64    eng_half;
26      PS::F64    ent_half;
27
28      void copyFromForce(const Force_grav& f) {
29          this->acc_grav = f.acc;
30          this->pot_grav = f.pot;
31      }
32      void copyFromForce(const Force_dens& f){
33          this->flag  = f.flag;
34          this->dens  = f.dens;
35          this->smth  = f.smth;
36          this->gradh = f.gradh;
37          this->divv  = f.divv;
38          this->rotv  = f.rotv;
39
40      }
41      void copyFromForce(const Force_hydro& f){
42          this->acc_hydro = f.acc;
43          this->eng_dot   = f.eng_dot;
44          this->ent_dot   = f.ent_dot;
45          this->dt        = f.dt;
46      }
47      PS::F64 getCharge() const{
48          return this->mass;
49      }
50      PS::F64vec getPos() const{
51          return this->pos;
52      }
53      PS::F64 getRSearch() const{
54          return this->smth;
55      }
56      void setPos(const PS::F64vec& pos){
57         this->pos = pos;
58      }
59      void writeAscii(FILE* fp) const{
60          fprintf(fp,
61                  "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t"
62                  "%lf\t%lf\t%lf\t%lf\t%lf\t%lf\n",
63                  this->id, this->mass,
64                  this->pos.x, this->pos.y, this->pos.z,
65                  this->vel.x, this->vel.y, this->vel.z,
66                  this->dens, this->eng, this->ent, this->pres);
67      }
68      void readAscii(FILE* fp){
69          fscanf(fp,
70                 "%lld\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t"
71                 "%lf\t%lf\t%lf\t%lf\t%lf\t%lf\n",
72                 &this->id, &this->mass,
73                 &this->pos.x, &this->pos.y, &this->pos.z,
74                 &this->vel.x, &this->vel.y, &this->vel.z,
```

```
75                  &this->dens, &this->eng, &this->ent, &this->pres);
76      }
77      void writeBinaryPos(FILE* fp) const {
78          fwrite(&this->pos, sizeof(this->pos), 1, fp);
79      }
80      void setEntropy(){
81          ent = (specific_heat_ratio - 1.0) * eng / std::pow(dens,
                    specific_heat_ratio - 1.0);
82      }
83      void setPressure(){
84  #if defined(ISOTHERMAL_EOS)
85          // In this case, eng = const.
86          pres = (specific_heat_ratio - 1.0) * dens * eng;
87          ent  = pres / std::pow(dens, specific_heat_ratio);
88  #else
89  #if defined(USE_ENTROPY)
90          pres = ent * std::pow(dens, specific_heat_ratio);
91          eng  = pres / ((specific_heat_ratio - 1.0) * dens);
92  #else
93          pres = (specific_heat_ratio - 1.0) * dens * eng;
94          ent  = pres / std::pow(dens, specific_heat_ratio);
95  #endif
96  #endif
97          snds = std::sqrt(specific_heat_ratio * pres / dens);
98  #if defined(USE_BALSARA_SWITCH)
99          BalSW = std::fabs(divv) / (std::fabs(divv) + std::sqrt(rotv * rotv
                    ) + 1.0e-4 * snds / smth);
100 #else
101         BalSW = 1.0;
102 #endif
103     }
104 };
```

7.1.3.2　EssentialParticle 型

まず、重力計算用の EssentialParticle 型である EP_grav クラス について解説する。このクラスには、重力計算を行う際、i 粒子と 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たせている。Listing 67 に、このクラスの実装を示す。 EssentialParticle 型は FullParticle 型から値をコピーするのに必要なメンバ関数 copyFromFP() を持つ必要があるが、本サンプルコードでは粒子種が 2 種類のため、2 つの copyFromFP を実装する必要があることに注意されたい。

Listing 67: EssentialParticle 型 (EP_grav クラス)

```
1  class EP_grav {
2  public:
3      PS::S64 id;
4      PS::F64 mass;
5      PS::F64vec pos;
6
7      void copyFromFP(const FP_nbody& fp) {
8          this->id   = fp.id;
```

```
 9          this->mass = fp.mass;
10          this->pos  = fp.pos;
11      }
12      void copyFromFP(const FP_sph& fp) {
13          this->id   = fp.id;
14          this->mass = fp.mass;
15          this->pos  = fp.pos;
16      }
17      PS::F64 getCharge() const {
18          return this->mass;
19      }
20      PS::F64vec getPos() const {
21          return this->pos;
22      }
23  };
```

次に、密度計算と圧力勾配加速度計算用の EssentialParticle 型である EP_hydro クラス について解説する。このクラスには、密度計算と圧力勾配加速度計算を行う際、i 粒子と j 粒子が持つべき全ての物理量をメンバ変数として持たせている。Listing 68 に、このクラスの実装を示す。 ここで、メンバ関数 getRSearch の定義が、単に smth を返すのではなく、係数 SCF_smth(1 より大きいが 1 に近い値) を掛けて返している。これは、効率的に密度計算を行うための仕掛けで、後ほど解説を行う。

Listing 68: EssentialParticle 型 (EP_hydro クラス)

```
 1  class EP_hydro {
 2  public:
 3     PS::S64    id;
 4     PS::F64vec pos;
 5     PS::F64vec vel;
 6     PS::F64    mass;
 7     PS::F64    smth;
 8     PS::F64    dens;
 9     PS::F64    pres;
10     PS::F64    gradh;
11     PS::F64    snds;
12     PS::F64    BalSW;
13
14     void copyFromFP(const FP_sph& fp){
15        this->id    = fp.id;
16        this->pos   = fp.pos;
17        this->vel   = fp.vel;
18        this->mass  = fp.mass;
19        this->smth  = fp.smth;
20        this->dens  = fp.dens;
21        this->pres  = fp.pres;
22        this->gradh = fp.gradh;
23        this->snds  = fp.snds;
24        this->BalSW = fp.BalSW;
25     }
26     PS::F64vec getPos() const{
27        return this->pos;
28     }
29     PS::F64 getRSearch() const{
```

```
30        return SCF_smth * this->smth;
31     }
32     void setPos(const PS::F64vec& pos){
33        this->pos = pos;
34     }
35 };
```

7.1.3.3　Force 型

まず、重力計算用の Force 型である Force_grav クラス について解説する。このクラスは、重力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。Listing 69 にこのクラスの実装を示す。

Listing 69: Force 型 (Force_grav クラス)

```
1 class Force_grav {
2 public:
3     PS::F64vec acc;
4     PS::F64 pot;
5     void clear() {
6         acc = 0.0;
7         pot = 0.0;
8     }
9 };
```

次に、密度計算用の Force 型である Force_dens クラス について解説する。このクラスは、密度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。Listing 70 に、このクラスの実装を示す。本サンプルコードの SPH 法では、smoothing length は固定ではなく、密度に応じて変化する。そのため、メンバ変数として smth を持つ。また、密度計算と同時に、∇h term および $(\nabla \cdot \boldsymbol{v})_i$、$(\nabla \times \boldsymbol{v})_i$ の計算を同時行うため、メンバ変数に gradh, divv, rotv を持つ。メンバ変数 flag は ρ_i と h_i を決定するイテレーション計算が収束したかどうかの結果を格納する変数である (詳細は、相互作用関数の節を参照のこと)。

Listing 70: Force 型 (Force_dens クラス)

```
1 class Force_dens{
2 public:
3    PS::S32 flag;
4    PS::F64 dens;
5    PS::F64 smth;
6    PS::F64 gradh;
7    PS::F64 divv;
8    PS::F64vec rotv;
9    void clear(){
10       flag  = 0;
11       dens  = 0.0;
12       gradh = 0.0;
13       divv  = 0.0;
14       rotv  = 0.0;
15    }
```

```
16 };
```

最後に、圧力勾配加速度計算用の Force 型である Force_hydro クラス について解説する。このクラスは、圧力勾配加速度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。Listing 71 に、このクラスの実装を示す。

Listing 71: Force 型 (Force_hydro クラス)

```
1  class Force_hydro{
2  public:
3     PS::F64vec acc;
4     PS::F64 eng_dot;
5     PS::F64 ent_dot;
6     PS::F64 dt;
7     void clear(){
8        acc = 0.0;
9        eng_dot = 0.0;
10       ent_dot = 0.0;
11    }
12 };
```

7.1.4　相互作用関数

本サンプルコードで使用する相互作用関数はすべて user_defined.hpp に実装されている。全部で 4 種類あり、重力計算 (粒子間相互作用及び粒子-超粒子間相互作用)、密度計算、圧力勾配加速度計算に使用される。以下、順に説明していく。

7.1.4.1　重力計算

重力計算用の相互作用関数は 関数テンプレート CalcGravity として実装されている。Listing 72 にこれらの実装を示す。実装は第 3-4 節で紹介した N 体計算サンプルコードのものとほぼ同じであり、詳細はそちらを参照されたい。

Listing 72: 相互作用関数 (重力計算用)

```
1  #if defined(ENABLE_PHANTOM_GRAPE_X86)
2  template <class TParticleJ>
3  void CalcGravity(const EP_grav * iptcl,
4                   const PS::S32 ni,
5                   const TParticleJ * jptcl,
6                   const PS::S32 nj,
7                   Force_grav * force) {
8      const PS::S32 nipipe = ni;
9      const PS::S32 njpipe = nj;
10     PS::F64 (*xi)[3] = (PS::F64 (*)[3])malloc(sizeof(PS::F64) * nipipe *
           PS::DIMENSION);
11     PS::F64 (*ai)[3] = (PS::F64 (*)[3])malloc(sizeof(PS::F64) * nipipe *
           PS::DIMENSION);
12     PS::F64  *pi     = (PS::F64  *    )malloc(sizeof(PS::F64) * nipipe);
13     PS::F64 (*xj)[3] = (PS::F64 (*)[3])malloc(sizeof(PS::F64) * njpipe *
           PS::DIMENSION);
```

```
14      PS::F64  *mj     = (PS::F64  *    )malloc(sizeof(PS::F64) * njpipe);
15      for(PS::S32 i = 0; i < ni; i++) {
16          xi[i][0] = iptcl[i].getPos()[0];
17          xi[i][1] = iptcl[i].getPos()[1];
18          xi[i][2] = iptcl[i].getPos()[2];
19          ai[i][0] = 0.0;
20          ai[i][1] = 0.0;
21          ai[i][2] = 0.0;
22          pi[i]    = 0.0;
23      }
24      for(PS::S32 j = 0; j < nj; j++) {
25          xj[j][0] = jptcl[j].getPos()[0];
26          xj[j][1] = jptcl[j].getPos()[1];
27          xj[j][2] = jptcl[j].getPos()[2];
28          mj[j]    = jptcl[j].getCharge();
29      }
30      PS::S32 devid = PS::Comm::getThreadNum();
31      g5_set_xmjMC(devid, 0, nj, xj, mj);
32      g5_set_nMC(devid, nj);
33      g5_calculate_force_on_xMC(devid, xi, ai, pi, ni);
34      for(PS::S32 i = 0; i < ni; i++) {
35          force[i].acc[0] += ai[i][0];
36          force[i].acc[1] += ai[i][1];
37          force[i].acc[2] += ai[i][2];
38          force[i].pot    -= pi[i];
39      }
40      free(xi);
41      free(ai);
42      free(pi);
43      free(xj);
44      free(mj);
45  }
46  #else
47  template <class TParticleJ>
48  void CalcGravity (const EP_grav * ep_i,
49                    const PS::S32 n_ip,
50                    const TParticleJ * ep_j,
51                    const PS::S32 n_jp,
52                    Force_grav * force) {
53      const PS::F64 eps2 = eps_grav * eps_grav;
54      for(PS::S32 i = 0; i < n_ip; i++){
55          PS::F64vec xi = ep_i[i].getPos();
56          PS::F64vec ai = 0.0;
57          PS::F64 poti = 0.0;
58          for(PS::S32 j = 0; j < n_jp; j++){
59              PS::F64vec rij    = xi - ep_j[j].getPos();
60              PS::F64    r3_inv = rij * rij + eps2;
61              PS::F64    r_inv  = 1.0/sqrt(r3_inv);
62              r3_inv  = r_inv * r_inv;
63              r_inv  *= ep_j[j].getCharge();
64              r3_inv *= r_inv;
65              ai     -= r3_inv * rij;
66              poti   -= r_inv;
67          }
68          force[i].acc += ai;
```

```
69          force[i].pot += poti;
70      }
71  }
72  #endif
```

7.1.4.2　密度計算

密度計算用の相互作用関数は 関数オブジェクト CalcDensity として実装されている。Listing 73 に、実装を示す。実装はマクロ ENABLE_VARIABLE_SMOOTHING_LENGTH が定義されているか否かで分かれる。このマクロが未定義の場合には、固定長カーネルコードとなり、実装は第 3-4 節で紹介した SPH サンプルコードとほぼ同じであるので、そちらを参照されたい。以下、このマクロが定義されている場合の実装について解説する。

第 7.1.2 節で説明したように、密度 ρ_i と smoothing length h_i は式 (14) と式 (9) を無矛盾に解いて決定する必要がある。これには 2 つの方程式を反復的に解く必要がある。このイテレーションを無限 for ループの中で行っている。本サンプルコードでは ρ_i と h_i の計算を効率的に行うため、smoothing length の値を定数 SCF_smth 倍してから密度計算を実行している (EP_hydro クラスのメンバ関数 getRSearch の実装を参照のこと) 。このため、定数倍する前の smoothing length の値を $h_{i,0}$ とすると、このイテレーションの間に h_i を 0 から $h_{\mathrm{max,alw}} \equiv$ scf_smth $\times h_{i,0}$ までの間なら変化させてもよいことになる。なぜなら、j 粒子リストの取りこぼしは発生しないからである。逆にこの範囲でイテレーションが収束しなければ、求めたい h_i は $h_{\mathrm{max,alw}}$ よりも大きいということになり、既存の j 粒子リストでは ρ_i と h_i を決定できないということになる。この場合、$h_{i,0}$ を大きくした上で、密度計算をやり直す必要がある。この外側のイテレーションは main.cpp の void 関数 calcDensity(関数名の先頭が小文字であることに注意) で行われている。この void 関数の詳細は第 7.1.5 節で行う。

無限 for ループの後には、∇h term の計算、$(\nabla \cdot \boldsymbol{v})_i$ 及び $(\nabla \times \boldsymbol{v})_i$ の計算を行っている。

Listing 73: 相互作用関数 (密度計算用)

```
1  class CalcDensity{
2  public:
3      void operator () (const EP_hydro * ep_i,
4                        const PS::S32 n_ip,
5                        const EP_hydro * ep_j,
6                        const PS::S32 n_jp,
7                        Force_dens * force){
8  #if defined(ENABLE_VARIABLE_SMOOTHING_LENGTH)
9          const PS::F64 eps = 1.0e-6;
10         const PS::F64 M_trgt = mass_avg * N_neighbor;
11         for (PS::S32 i = 0; i < n_ip; i++) {
12             PS::F64 dens = 0.0;
13             PS::F64 h = ep_i[i].smth;
14             const PS::F64 h_max_alw = SCF_smth * h; // maximum allowance
15             PS::F64 h_L = 0.0;
16             PS::F64 h_U = h_max_alw;
17             PS::F64 dh_prev = 0.0;
18             PS::S32 n_unchanged = 0;
19             // Software caches
20             PS::F64 * mj  = (PS::F64 *)malloc(sizeof(PS::F64) * n_jp);
```

```
21              PS::F64 * rij = (PS::F64 *)malloc(sizeof(PS::F64) * n_jp);
22              for (PS::S32 j = 0; j < n_jp; j++) {
23                  mj[j] = ep_j[j].mass;
24                  const PS::F64vec dr = ep_j[j].pos - ep_i[i].pos;
25                  rij[j] = std::sqrt(dr * dr);
26              }
27              for (;;) {
28                  // Calculate density
29                  dens = 0.0;
30                  for (PS::S32 j = 0; j < n_jp; j++) {
31                     dens += mj[j] * W(rij[j], h);
32                  }
33                  // Check if the current value of the smoohting length
                        satisfies
34                  // Eq.(5) in Springel (2005).
35                  const PS::F64 M = 4.0 * math_const::pi * h * h * h * dens
                        / 3.0;
36                  if ((h < h_max_alw) && (std::abs(M/M_trgt - 1.0) < eps)) {
37                      // In this case, Eq.(5) holds within a specified
                            accuracy.
38                      force[i].flag = 1;
39                      force[i].dens = dens;
40                      force[i].smth = h;
41                      break;
42                  }
43                  if (((h == h_max_alw) && (M < M_trgt)) || (n_unchanged ==
                        4)) {
44                      // In this case, we skip this particle forcibly.
45                      // In order to determine consistently the density
46                      // and the smoohting length for this particle,
47                      // we must re-perform calcForceAllAndWriteBack().
48                      force[i].flag = 0;
49                      force[i].dens = dens;
50                      force[i].smth = h_max_alw;
51                      break;
52                  }
53                  // Update h_L & h_U
54                  if (M < M_trgt) {
55                     if (h_L < h) h_L = h;
56                  }
57                  else if (M_trgt < M) {
58                     if (h < h_U) h_U = h;
59                  }
60                  const PS::F64 dh = h_U - h_L;
61                  if (dh == dh_prev) {
62                     n_unchanged++;
63                  }
64                  else {
65                     dh_prev = dh;
66                     n_unchanged = 0;
67                  }
68                  // Update smoothing length
69                  h = std::pow((3.0 * M_trgt)/(4.0 * math_const::pi * dens),
                        1.0/3.0);
70                  if ((h <= h_L) || (h == h_U)) {
```

```
71                      // In this case, we switch to the bisection search.
72                      // The inclusion of '=' in the if statement is very
73                      // important to escape a limit cycle.
74                      h = 0.5 * (h_L + h_U);
75                  }
76                  else if (h_U < h) {
77                      h = h_U;
78                  }
79              }
80              // Calculate grad-h term
81              if (force[i].flag == 1) {
82                  PS::F64 drho_dh = 0.0;
83                  for (PS::S32 j = 0; j < n_jp; j++) {
84                      drho_dh += mj[j] * dWdh(rij[j], h);
85                  }
86                  force[i].gradh = 1.0 / (1.0 + (h * drho_dh) / (3.0 * dens)
                        );
87              }
88              else {
89                  force[i].gradh = 1.0; // dummy value
90              }
91  #if defined(USE_BALSARA_SWITCH)
92              // Compute \div v & \rot v for Balsara switch
93              for (PS::S32 j = 0; j < n_jp; j++) {
94                  const PS::F64vec dr = ep_i[i].pos - ep_j[j].pos;
95                  const PS::F64vec dv = ep_i[i].vel - ep_j[j].vel;
96                  force[i].divv -= mj[j] * dv * gradW(dr, force[i].smth);
97                  force[i].rotv -= mj[j] * dv ^ gradW(dr, force[i].smth);
98              }
99              force[i].divv /= force[i].dens;
100             force[i].rotv /= force[i].dens;
101 #endif
102             // Release memory
103             free(mj);
104             free(rij);
105         }
106 #else
107         for (PS::S32 i = 0; i < n_ip ; i++){
108             // Compute density
109             for (PS::S32 j = 0; j < n_jp; j++){
110                 const PS::F64vec dr = ep_j[j].pos - ep_i[i].pos;
111                 const PS::F64 rij = std::sqrt(dr * dr);
112                 force[i].dens += ep_j[j].mass * W(rij, ep_i[i].smth);
113             }
114             force[i].smth = ep_i[i].smth;
115             force[i].gradh = 1.0;
116 #if defined(USE_BALSARA_SWITCH)
117             // Compute \div v & \rot v for Balsara switch
118             for (PS::S32 j = 0; j < n_jp; j++) {
119                 const PS::F64vec dr = ep_i[i].pos - ep_j[j].pos;
120                 const PS::F64vec dv = ep_i[i].vel - ep_j[j].vel;
121                 force[i].divv -= ep_j[j].mass * dv * gradW(dr, force[i].
                        smth);
122                 force[i].rotv -= ep_j[j].mass * dv ^ gradW(dr, force[i].
                        smth);
```

```
123             }
124             force[i].divv /= force[i].dens;
125             force[i].rotv /= force[i].dens;
126 #endif
127         }
128 #endif
129     }
130 };
```

7.1.4.3　圧力勾配加速度計算

圧力勾配加速度用の相互作用関数は 関数オブジェクト CalcHydroForce として実装されている。Listing 74 に、実装を示す。この void 関数では、式 (10)、(12)、(13) の右辺の計算、及び、Springel [2005, MNRAS, 364, 1105] の式 (16) に従って dt の計算を行っている (dt については FP_sph クラス の説明を参照のこと)。

Listing 74: 相互作用関数 (圧力勾配加速度計算用)

```
1  class CalcHydroForce{
2  public:
3      void operator () (const EP_hydro * ep_i,
4                        const PS::S32 n_ip,
5                        const EP_hydro * ep_j,
6                        const PS::S32 n_jp,
7                        Force_hydro * force){
8          for (PS::S32 i = 0; i < n_ip; i++){
9              const PS::F64vec pos_i = ep_i[i].pos;
10             const PS::F64vec vel_i = ep_i[i].vel;
11             const PS::F64 smth_i   = ep_i[i].smth;
12             const PS::F64 dens_i   = ep_i[i].dens;
13             const PS::F64 pres_i   = ep_i[i].pres;
14             const PS::F64 f_i      = ep_i[i].gradh;
15             const PS::F64 snds_i   = ep_i[i].snds;
16             const PS::F64 povrho2_i = pres_i / (dens_i * dens_i);
17             PS::F64 v_sig_max = 0.0;
18             for (PS::S32 j = 0; j < n_jp; j++){
19                 const PS::F64vec dr = pos_i - ep_j[j].pos;
20                 const PS::F64vec dv = vel_i - ep_j[j].vel;
21                 const PS::F64 w_ij = (dv * dr < 0) ? dv * dr / std::sqrt(dr
                       * dr) : 0;
22                 const PS::F64 v_sig = snds_i + ep_j[j].snds - 3.0 * w_ij;
23                 v_sig_max = std::max(v_sig_max, v_sig);
24                 const PS::F64 AV = - 0.5 * alpha_AV * v_sig * w_ij / (0.5 *
                       (dens_i + ep_j[j].dens))
25                                  * 0.5 * (ep_i[i].BalSW + ep_j[j].BalSW);
26                 const PS::F64vec gradW_i  = gradW(dr, smth_i);
27                 const PS::F64vec gradW_j  = gradW(dr, ep_j[j].smth);
28                 const PS::F64vec gradW_ij = 0.5 * (gradW_i + gradW_j);
29                 const PS::F64 povrho2_j = ep_j[j].pres / (ep_j[j].dens *
                       ep_j[j].dens);
30                 const PS::F64 f_j = ep_j[j].gradh;
31                 force[i].acc     -= ep_j[j].mass * (f_i * povrho2_i *
                       gradW_i
```

```
32                                                  +f_j * povrho2_j *
                                                          gradW_j
33                                                  +AV * gradW_ij);
34                 force[i].eng_dot += ep_j[j].mass * (f_i * povrho2_i *
                           gradW_i
35                                                  +0.5 * AV * gradW_ij) *
                                                          dv;
36                 force[i].ent_dot += 0.5 * ep_j[j].mass * AV * gradW_ij * dv;
37              }
38              const PS::F64 p = specific_heat_ratio - 1.0;
39              force[i].ent_dot *= p/std::pow(dens_i, p);
40              force[i].dt = CFL_hydro * 2.0 * ep_i[i].smth / v_sig_max;
41          }
42      }
43  };
```

7.1.5　プログラム本体

本節では、主に main.cpp に実装された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。7.1 節冒頭で述べたように、このサンプルコードでは円盤銀河の N 体/SPH シミュレーションを行うものであるが、初期条件としては円盤銀河の他、簡単なテスト計算用の初期条件も用意されている。具体的に以下の 4 つの場合に対応している:

(a) 円盤銀河用の初期条件。この初期条件はコンパイルオプション時に-DINITIAL_CONDITION=0 が指定された場合に選択される。初期条件作成は ic.hpp の void 関数 galaxy_IC で行われる。ダークマターと星の分布は事前に MAGI で作成されたファイルを読み込んで設定される。一方、ガスの初期分布はこの void 関数内部で生成される。デフォルトでは粒子数 2^{18} で exponential disk ($M = 10^{10}$ M$_\odot$, $R_s = 7$ kpc [scale radius], $R_t = 12.5$ kpc [truncation radius], $z_d = 0.4$ kpc [scale height], $z_t = 1$ kpc [truncation height]) が生成される。

(b) Cold collapse 問題用の初期条件。この初期条件はコンパイルオプション時に-DINITIAL_CONDITION=1 が指定された場合に選択される。初期条件作成は ic.hpp の void 関数 cold_collapse_test_IC で行われる。

(c) Evrard test (Evrard [1988,MNRAS,235,911] の第 3.3 節) 用の初期条件。この初期条件はコンパイルオプション時に-DINITIAL_CONDITION=2 が指定された場合に選択される。初期条件作成は ic.hpp の void 関数 Evrard_test_IC で行われる。作成方法は 2 つあり、void 関数の最後の引数の値を手動で 0 か 1 にして指定する。0 の場合、格子状に並んだ SPH 粒子から Evrard 球の密度分布を作成する。1 の場合、ガラス状に分布した SPH 粒子から Evrard 球の密度分布を作成する。1 を選択するためには、事前に次項で説明するモードで SPH 粒子のデータを作成しておく必要がある。

(d) $[-1, 1)^3$ の立方体中に一様密度のガラス状の SPH 粒子分布を作成するための初期条件/動作モード。この初期条件はコンパイルオプション時に-DINITIAL_CONDITION=3 が指定された場合に選択される。初期条件作成は ic.hpp の void 関数 make_glass_IC で行われる。

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、或いは、初期条件の作成
(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。

7.1.5.1　ヘッダファイルのインクルード

FDPS の機能を使用するため main.cpp のファイル冒頭部分で、particle_simulator.hpp をインクルードしている。

Listing 75: FDPS のヘッダーファイルのインクルード

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

7.1.5.2　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 76: 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 77: FDPS の終了

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

7.1.5.3　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。

7.1.5.3.1　粒子群オブジェクトの生成と初期化

本サンプルコードでは、N 体粒子と SPH 粒子のデータを異なる粒子群オブジェクトを用いて管理する。 具体的には N 体粒子のデータには psys_nbody、SPH 粒子のデータには psys_sph という粒子群オブジェクトを使う。コードではこれらの粒子群オブジェクトを生成し、initialize メソッドで初期化を行っている。

Listing 78: 粒子群オブジェクトの生成・初期化

```
1 PS::ParticleSystem<FP_nbody> psys_nbody;
2 PS::ParticleSystem<FP_sph> psys_sph;
3 psys_nbody.initialize();
4 psys_sph.initialize();
```

7.1.5.3.2　領域情報オブジェクトの生成と初期化

本サンプルコードでは、計算領域の分割を、N 体粒子と SPH 粒子を合わせた粒子全体が等分割されるように行うこととする。この場合、必要な領域情報オブジェクトは 1 つである。したがって、本コードでは dinfo という領域情報オブジェクトを 1 つ生成し、initialize メソッドで初期化している。

Listing 79: 領域情報オブジェクトの生成・初期化

```
1 PS::DomainInfo dinfo;
2 dinfo.initialize();
```

7.1.5.3.3　ツリーオブジェクトの生成と初期化

本サンプルコードでは、重力計算用、密度計算、圧力勾配加速度計算のそれぞれに 1 つずつツリーを用意している。ツリーオブジェクトの初期化の際には、API initialize の第 1 引数に計算で使用する大雑把な粒子数を渡す必要がある。重力計算用のツリーオブジェクト tree_grav では、ローカル粒子数の 3 倍の値を渡している。一方、密度計算と圧力勾配加速度計算に使用されるツリーオブジェクト tree_dens では、ローカルの SPH 粒子数の 3 倍の値を渡している。

Listing 80: ツリーオブジェクトの生成・初期化

```
1  const PS::S64 numPtclSPH = std::max(psys_sph.getNumberOfParticleGlobal()
       ,1);
2  const PS::S64 numPtclAll = psys_nbody.getNumberOfParticleGlobal()
3                           + numPtclSPH;
4
5  const PS::F32 theta_grav = 0.5;
6  PS::TreeForForceLong<Force_grav, EP_grav, EP_grav>::Monopole tree_grav;
7  tree_grav.initialize(3 * numPtclAll, theta_grav);
8
9  PS::TreeForForceShort<Force_dens, EP_hydro, EP_hydro>::Gather tree_dens;
10 tree_dens.initialize(3 * numPtclSPH);
11
12 PS::TreeForForceShort<Force_hydro, EP_hydro, EP_hydro>::Symmetry
       tree_hydro;
13 tree_hydro.initialize(3 * numPtclSPH);
```

7.1.5.4　初期条件の設定

初期条件の設定は 関数 setupIC で行われる。この void 関数はマクロ INITIAL_CONDITION の値に応じて、内部でさらに別の void 関数を呼び出しており、呼び出される void 関数とマクロの値の対応は、既に述べた通りである。引数の time_dump, dt_dump, time_end は、データ出力の最初の時刻、出力時間間隔、シミュレーション終了時間を表す変数であり、個々の初期条件作成関数の中で設定すべきものである。また、境界条件、重力ソフトニングの値 (eps_grav)、系に許される最大の時間刻み (dt_max) も設定する必要がある (dt_max に関しては必ずしも設定する必要はない)。

Listing 81: 初期条件の設定

```
1 setupIC(psys_nbody, psys_sph, dinfo, time_dump, dt_dump, time_end);
```

以下、円盤銀河の初期条件を設定する 関数 GalaxyIC について、留意事項を述べておく。

- MAGI が作成する粒子データは MAGI のコード内単位系で出力される。単位系の情報は MAGI を実行したときに出力されるファイル doc/unit.txt に記述されている。このファイルに記載された単位質量、単位長さ、単位時間の値と、定数 magi_unit_mass, magi_unit_leng, magi_unit_time は一致させなければならない。

- 関数が読み込むファイルは ./magi_data/dat/Galaxy.tipsy である。別なファイルを読み込ませたい場合、手動でソースコードを変更する必要がある。

- 関数が生成するガス分布は R ($\equiv \sqrt{x^2+y^2}$) 方向と z 方向に exponential な密度分布を持つガス円盤である。それぞれの方向のスケール長が変数 Rs, zd で、分布を打ち切る距離は変数 Rt, zt である。

- 初期のガスの熱力学的状態はガス温度 temp と水素原子に対する平均分子量 mu を与えて指定する。コンパイル時マクロ USE_ENTROPY が定義済み/未定義に関わらず、粒子の熱力学的状態は単位質量あたりの内部エネルギーとして与える必要がある (fp_sph のメンバ変数 eng)。USE_ENTROPY が定義済みの場合、メイン関数 main() で呼び出されている 関数 setEntropy によって、計算された密度と内部エネルギーの初期値から初期エントロピーが自動的に決定される。未定義の場合、ここで設定した eng の値がそのまま内部エネルギーの初期値となる。

7.1.5.5　領域分割の実行

複数の粒子種がある場合に、これらを合わせた粒子分布に基づいて領域分割を実行するには、領域情報オブジェクト用の 2 つの API collectSampleParticle と decomposeDomain を併用する必要がある。まず、API collectSampleParticle でそれぞれの粒子群オブジェクトからサンプル粒子を集める。**このとき、2種類目以降の粒子種に対する呼び出しでは、第 2 引数に false を指定する必要がある。この指定がないと、1種類目の粒子群オブジェクトの情報がクリアされてしまうからである。**すべての粒子群オ

ブジェクトに対して、この API の呼び出しが終わったら、API decomposeDomain で領域分割を実行する。

Listing 82: 領域分割の実行

```
dinfo.collectSampleParticle(psys_nbody);
dinfo.collectSampleParticle(psys_sph,false);
dinfo.decomposeDomain();
```

7.1.5.6　粒子交換の実行

先程計算した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、粒子群オブジェクト用 API exchangeParticle を使用する:

Listing 83: 粒子交換の実行

```
psys_nbody.exchangeParticle(dinfo);
psys_sph.exchangeParticle(dinfo);
```

7.1.5.7　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、本サンプルコードにおける初期条件作成後最初の相互作用計算の実装を示す。最初に重力計算をし、その後、密度計算・圧力勾配加速度計算を行っている。

Listing 84: 相互作用計算の実行

```
    //- Gravity calculations
#if defined(ENABLE_GRAVITY_INTERACT)
    tree_grav.setParticleLocalTree(psys_nbody);
    tree_grav.setParticleLocalTree(psys_sph,false);
    tree_grav.calcForceMakingTree(CalcGravity<EP_grav>,
                                  CalcGravity<PS::SPJMonopole>,
                                  dinfo);
    for (PS::S32 i = 0; i < psys_nbody.getNumberOfParticleLocal(); i++) {
        psys_nbody[i].copyFromForce(tree_grav.getForce(i));
    }
    const PS::S32 offset = psys_nbody.getNumberOfParticleLocal();
    for (PS::S32 i = 0; i < psys_sph.getNumberOfParticleLocal(); i++) {
        psys_sph[i].copyFromForce(tree_grav.getForce(i+offset));
    }
#endif

    //- SPH calculations
#if defined(ENABLE_HYDRO_INTERACT)
    calcDensity(psys_sph, dinfo, tree_dens);
#if defined(USE_ENTROPY)
    setEntropy(psys_sph);
#endif
    setPressure(psys_sph);
    tree_hydro.calcForceAllAndWriteBack(CalcHydroForce(), psys_sph, dinfo)
          ;
```

```
25 #endif
```

まず重力計算の方法について説明する。重力計算は、N 体粒子と SPH 粒子の両方が関わる。このような複数の粒子種の間で 1 つの相互作用計算を行うには、ツリーオブジェクト用の API `setParticleLocalTree` と `calcForceMakingTree` を合わせて使用する必要がある。まず、各粒子群オブジェクトに対して、API `setParticleLocalTree` を使って、粒子情報をツリーオブジェクトに渡す。**このとき、2種類目以降の粒子群オブジェクトに対する呼び出しでは、第 2 引数に `false` を指定する必要がある。この指定が無いと、これまでツリーオブジェクトに渡した粒子情報がクリアされてしまうからである。**重力計算に関係するすべての粒子群オブジェクトに対して、この API の呼び出しが完了したら、API `calcForceMakingTree` で相互作用計算を行う。相互作用計算の結果を取得するためには、API `getForce` を使う。この API は引数に整数 i を取り、API `setParticleLocalTree` で i 番目に読み込んだ粒子が受ける相互作用を返す。したがって、2 種類目以降の粒子種の相互作用の結果を取得する場合、適切にオフセット値を指定する必要があることに注意されたい。

次に密度計算と圧力勾配加速度計算について説明する。これらの計算は 1 粒子種しか関わらないため、本チュートリアルでこれまで使ってきた API `calcForceAllAndWriteBack` が使用できる。圧力勾配加速度に関しては、メイン関数 `main` 内でこの API を直接呼び出している。一方、密度計算は、第 7.1.4 節でも述べた通り、ρ_i と h_i のイテレーション計算が収束しなかったときのための対処が必要であり、これを `void` 関数 calcDensity の中で行っている。実装は次のようになっている。実装はマクロ `ENABLE_VARIABLE_SMOOTHING_LENGTH` が定義済みか未定義かで分岐しており、未定義の場合には固定長カーネルの SPH コードとなるので、単に、API `calcForceAllAndWriteBack` を 1 回だけ実行している。一方、上記マクロが定義済みの場合、すべての粒子の ρ_i と h_i が無矛盾に決定されるまで、API `calcForceAllAndWriteBack` を繰り返し実行する。各粒子が収束したかの情報はクラス `fp_sph` のメンバ変数 `flag` に格納されており、値が 1 のときに収束していることを示す。`flag` が 1 を取る粒子数が全 SPH 粒子数に一致したときに計算を終わらせている。

Listing 85: 関数 `calcDensity` の実装

```
1  void calcDensity(PS::ParticleSystem<FP_sph> & psys,
2                   PS::DomainInfo & dinfo,
3                   PS::TreeForForceShort<Force_dens, EP_hydro, EP_hydro>::
                          Gather & tree) {
4  #if defined(ENABLE_VARIABLE_SMOOTHING_LENGTH)
5      const PS::S32 n_loc = psys.getNumberOfParticleLocal();
6      const PS::S64 n_glb = psys.getNumberOfParticleGlobal();
7      // Determine the density and the smoothing length so that Eq.(6) in
              Springel (2005)
8      // holds within a specified accuracy.
9      SCF_smth = 1.25;
10     PS::S32 iter = 0;
11     for (;;) {
12         iter++;
13         if (PS::Comm::getRank() == 0) std::cout << "iter␣=␣" << iter <<
                   std::endl;
14         // Compute density, etc.
```

```
15          tree.calcForceAllAndWriteBack(CalcDensity(), psys, dinfo);
16          // Check convergence
17          PS::S32 n_compl_loc = 0;
18          for (PS::S32 i = 0; i < n_loc; i++) {
19              if (psys[i].flag == 1) n_compl_loc++;
20          }
21          const PS::S64 n_compl = PS::Comm::getSum(n_compl_loc);
22          if (n_compl == n_glb) break;
23      }
24      // Reset SCF_smth
25      SCF_smth = 1.0;
26  #else
27      SCF_smth = 1.0;
28      tree.calcForceAllAndWriteBack(CalcDensity(), psys, dinfo);
29  #endif
30  }
```

void 関数 setEntropy は、初期条件作成後 1 回だけ呼び出される void 関数で、エントロピーの初期値をセットする。式 (8) から、エントロピーを計算するには初期密度が必要である。そのため、void 関数 calcDensity の後に配置されている。void 関数 setEntropy では、計算された密度と u_i の初期値を使って、エントロピーをセットする。これ以降は、エントロピーが独立変数となる。

7.1.5.8　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、第 4.1.3.4.4 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 void 関数 full_drift、粒子速度を時間推進する $K(\cdot)$ オペレータは void 関数 initial_kick, final_kick として実装されている。

8 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。

8.1 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)

8.2 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)

8.3 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。

9 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. (PASJ, 70, 70) 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。