FDPSユーザーチュートリアル

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

目 次

1 TODO **2**

2 変更記録 **3**

3 概要 **4**

4 入門 **5**
4.1 動作環境 5
4.2 必要なソフトウェア 5
4.2.1 標準機能 5
4.2.1.1 逐次処理 5
4.2.1.2 並列処理 5
4.2.1.2.1 OpenMP 5
4.2.1.2.2 MPI 6
4.2.1.2.3 MPI+OpenMP 6
4.2.2 拡張機能 6
4.2.2.1 Particle Mesh 6
4.3 インストール 6
4.3.1 取得方法 6
4.3.1.1 最新バージョン 7
4.3.1.2 過去のバージョン 7
4.3.2 ビルド方法 7
4.4 サンプルコードの使用方法 7
4.4.1 重力 N 体シミュレーションコード 8
4.4.1.1 概要 8
4.4.1.2 ディレクトリ移動 8
4.4.1.3 Makefile の編集 8
4.4.1.4 make の実行 9
4.4.1.5 実行 9
4.4.1.6 結果の解析 9

4.4.1.7 x86 版 Phantom-GRAPE を使う場合 10
4.4.2 SPH シミュレーションコード 11
4.4.2.1 概要 11
4.4.2.2 ディレクトリ移動 11
4.4.2.3 Makefile の編集 11
4.4.2.4 make の実行 12
4.4.2.5 実行 12
4.4.2.6 結果の解析 13

5 使ってみよう **14**
5.1 サンプルコードのコンパイルと実行 14
5.2 前提知識 14
5.2.1 Vector 型 14
5.3 固定長 SPH シミュレーションコード 14
5.3.1 作業ディレクトリ 14
5.3.2 インクルード 14
5.3.3 ユーザー定義クラス 14
5.3.3.1 概要 14
5.3.3.2 FullParticle 型 14
5.3.3.3 EssentialParticleI 型 16
5.3.3.4 Force 型 17
5.3.3.5 calcForceEpEp 型 18
5.3.4 プログラム本体 20
5.3.4.1 概要 20
5.3.4.2 開始、終了 20
5.3.4.3 初期化 20
5.3.4.3.1 オブジェクトの生成 20
5.3.4.3.2 領域クラスの初期化 21
5.3.4.3.3 粒子群クラスの初期化 21
5.3.4.3.4 相互作用ツリークラスの初期化 21
5.3.4.4 ループ 21
5.3.4.4.1 領域分割の実行 22
5.3.4.4.2 粒子交換の実行 22
5.3.4.4.3 相互作用計算の実行 22
5.3.5 コンパイル 22
5.3.6 実行 22
5.3.7 ログファイル 23
5.3.8 可視化 23
5.4 (穴埋め式で)N 体シミュレーションコードを書く 23
5.4.1 作業ディレクトリ 23
5.4.2 ユーザー定義クラス 23

5.4.2.1 概要 . . . 23
5.4.2.2 FullParticle 型 . . . 24
5.4.2.3 calcForceEpEp 型 . . . 25
5.4.3 プログラム本体 . . . 26
5.4.3.1 概要 . . . 26
5.4.3.2 開始、終了 . . . 26
5.4.3.3 初期化 . . . 27
5.4.3.3.1 オブジェクトの生成 . . . 27
5.4.3.3.2 領域クラスの初期化 . . . 27
5.4.3.3.3 粒子群クラスの初期化 . . . 27
5.4.3.3.4 相互作用ツリークラスの初期化 . . . 27
5.4.3.4 ループ . . . 28
5.4.3.4.1 領域分割の実行 . . . 28
5.4.3.4.2 粒子交換の実行 . . . 28
5.4.3.4.3 相互作用計算の実行 . . . 28
5.4.3.4.4 時間積分 . . . 28
5.4.3.4.5 predict . . . 28
5.4.3.4.6 correct . . . 29
5.4.4 ログファイル . . . 29

6 サンプルコード **30**

7 ユーザーサポート **31**
7.1 コンパイルできない場合 . . . 31
7.2 コードがうまく動かない場合 . . . 31
7.3 その他 . . . 31

8 ライセンス **32**

1 TODO

2　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

3 概要

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

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

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

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

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

ユーザーは、粒子間相互作用や粒子の軌道積分などを、C++言語を用いて記述する。将来的には Fortran 言語でも記述できるように検討する。

4 入門

本節では、FDPS の入門について記述する。FDPS を使用する環境、FDPS に必要なソフトウェア、FDPS のインストール方法、サンプルコードの使用方法、の順で記述する。

4.1 動作環境

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

4.2 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。

4.2.1 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。

4.2.1.1 逐次処理

逐次処理の場合に必要なソフトウェアは以下の通りである。

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

4.2.1.2 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。

4.2.1.2.1 OpenMP

OpenMP を使用する際に必要なソフトウェアは以下の通り。

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

4.2.1.2.2　MPI

MPI を使用する際に必要なソフトウェアは以下の通り。

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

4.2.1.2.3　MPI+OpenMP

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

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

4.2.2　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。

4.2.2.1　Particle Mesh

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

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

4.3　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。

4.3.1　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。

4.3.1.1 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 のしたを 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
```

4.3.1.2 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

4.3.2 ビルド方法

configure などをする必要はない。

4.4 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記述する。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。

4.4.1 重力 N 体シミュレーションコード

4.4.1.1 概要

以下の手順で本コードを使用できる。

- ディレクトリ$(FDPS)/sample/nbody に移動。これ以後、ディレクトリ$(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。

4.4.1.2 ディレクトリ移動

ディレクトリ$(FDPS)/sample/nbody に移動する。

4.4.1.3 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
  - マクロ CC に C++コンパイラを代入する
- OpenMP のみ使用の場合
  - マクロ 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 (インテルコンパイラの場合は-fopenmp を外す)
- MPI のみ使用の場合
  - マクロ CC に MPIC++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

- マクロ 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 (インテルコンパイラの場合は-fopenmp を外す)
- ”CFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL”の行のコメントアウトを外す

4.4.1.4 make の実行

make コマンドを実行する。

4.4.1.5 実行

実行方法は以下の通りである。

- MPI を使用しない場合、コマンドライン上で以下のコマンドを実行する

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

- MPI を使用する場合、コマンドライン上で以下のコマンドを実行する

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

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

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

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

4.4.1.6 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。

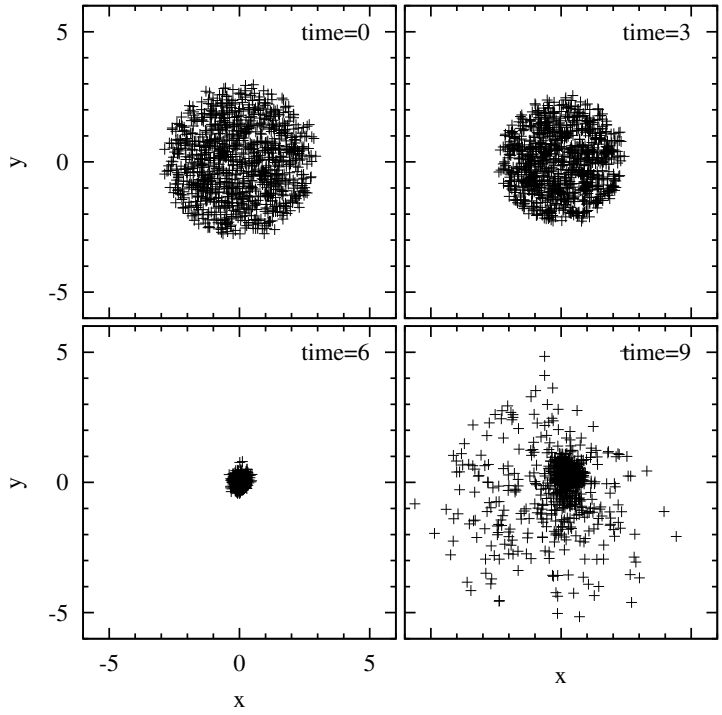

図 1:

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
```

4.4.1.7 x86 版 Phantom-GRAPE を使う場合

まず、使用環境を確認する。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/nbody に戻り、ファイル Makefile 内の "#use_phantom_grape_x86 = yes" の"#" を消す。make を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、Phatom-GRAPE を使うと、使わない場合に比べて、最大で 5 倍弱ほど高速なコードとなる。以下が最適化された実行例。

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

4.4.2 SPH シミュレーションコード

4.4.2.1 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/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)

4.4.2.2 ディレクトリ移動

ディレクトリ$(FDPS)/sample/nbody に移動に移動する。

4.4.2.3 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
  - マクロ CC に C++コンパイラを代入する
- OpenMP のみ使用の場合
  - マクロ 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 (インテルコンパイラの場合は-fopenmp を外す)

- MPI のみ使用の場合
  - マクロ CC に MPIC++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
  - マクロ 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 (インテルコンパイラの場合は-fopenmp を外す)
  - ”CFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL”の行のコメントアウトを外す

4.4.2.4 make の実行

make コマンドを実行する。

4.4.2.5 実行

実行方法は以下の通りである。

- MPI を使用しない場合、コマンドライン上で以下のコマンドを実行する

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

- MPI を使用する場合、コマンドライン上で以下のコマンドを実行する

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

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

正しく終了すると、標準エラー出力は以下のようなログを出力する。

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

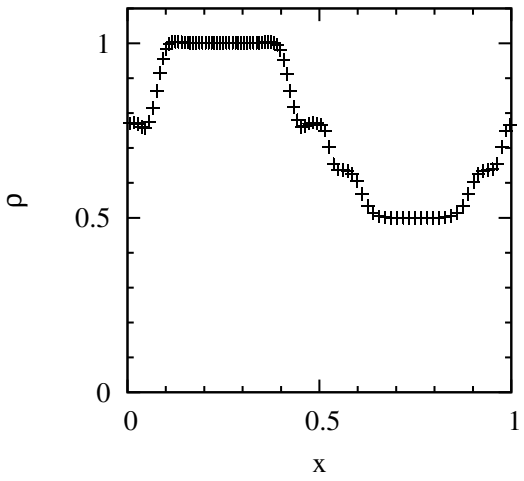

図 2:

4.4.2.6 結果の解析

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

これは 3 次元の衝撃波管問題である。コマンドライン上で以下のコマンドを実行すれば、横軸に粒子の x 座標、縦軸に粒子の密度をプロットできる (時刻は 40)。

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

正しい答が得られれば、図 2 のような図を描ける。

5 使ってみよう

5.1 サンプルコードのコンパイルと実行

5.2 前提知識

5.2.1 Vector 型

5.3 固定長 SPH シミュレーションコード

本節では、固定 smoothing length での標準 SPH 法を FDPS 上で実装する方法について、解説する。今回のチュートリアルコードでは、コード中で 3 次元の衝撃波管問題の初期条件を生成し、それを計算している。

5.3.1 作業ディレクトリ

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

```
$ cd (FDPS)/tutorial/sph
```

5.3.2 インクルード

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

ソースコード 1: Include FDPS

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

5.3.3 ユーザー定義クラス

5.3.3.1 概要

本節では、FDPS の機能を用いて SPH の計算を行うにおいて、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。

5.3.3.2 FullParticle 型

ユーザーは FullParticle 型を記述しなければならない。FullParticle 型には、シミュレーションを行うにあたって、SPH 粒子が持っているべき全ての物理量が含まれている。また、FullParticle 型には後述する Force 型から、結果をコピーするのに必要なメンバ関数を持つ必要がある。その他、粒子質量を返す関数である getCharge()、粒子座標を返す関数である getPos()、近傍粒子の探索半径を返す関数である getRSearch()、粒子の座標を書き込む関数

である setPos() が必要になる。本チュートリアルでは、FDPS に備わっているファイル入出力関数を用いるのに必要な関数である writeAscii() と readAscii() を書いてある。また、これらに加え、状態方程式から圧力を計算するメンバ関数である、setPressure() が記述されているが、この関数は FDPS が用いるものではないため、必須のものではないことに注意する。以下は本チュートリアル中で用いる FullParticle 型の例である。

ソースコード 2: 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, "%ld\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\
                        t%lf\t%lf\t%lf\t%lf\n", this->id,  this->
                        mass,  this->pos.x,  this->pos.y,  this->
                        pos.z,  this->vel.x,  this->vel.y,  this
                        ->vel.z,  this->dens,  this->eng,  this->
                        pres);
38          }
39          void readAscii(FILE* fp){
40                  fscanf(fp, "%ld\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t
                        %lf\t%lf\t%lf\t%lf\n", &this->id, &this->
                        mass, &this->pos.x, &this->pos.y, &this->
                        pos.z, &this->vel.x, &this->vel.y, &this
                        ->vel.z, &this->dens, &this->eng, &this->
                        pres);
41          }
42          void setPressure(){
43                  const PS::F64 hcr = 1.4;
44                  pres = (hcr - 1.0) * dens * eng;
45                  snds = sqrt(hcr * pres / dens);
46          }
47  };
```

5.3.3.3 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。また、EssentialParticleI 型には前述した FullParticle 型から、値をコピーするのに必要なメンバ関数を持つ必要がある。その他、粒子座標を返す関数である getPos()、近傍粒子の探索半径を返す関数である getRSearch()、粒子の座標を書き込む関数である setPos() が必要になる。以下は本チュートリアル中で用いる EssentialParticleI 型の例である。

ソースコード 3: EssentialParticleI 型

```
1  struct EP{
2          PS::F64vec pos;
3          PS::F64vec vel;
4          PS::F64    mass;
```

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

5.3.3.4 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、本チュートリアル中では、Force は密度の計算と、流体相互作用計算の 2 つが存在するため、Force 型は 2 つ書く必要がある。また、メンバ変数を 0 クリアするための関数 clear が必要になる。以下は本チュートリアル中で用いる Force 型の例である。

ソースコード 4: Force 型

```
1  class Dens{
2           public:
3           PS::F64 dens;
4           PS::F64 smth;
5           void clear(){
6                   dens = 0;
7           }
```

```
8 };
9
10 class Hydro{
11         public:
12         PS::F64vec acc;
13         PS::F64 eng_dot;
14         PS::F64 dt;
15         void clear(){
16                 acc = 0;
17                 eng_dot = 0;
18         }
19 };
```

5.3.3.5 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、Force の計算の具体的な内容を書く必要がある。今回のチュートリアル中では、ファンクタを用いて実装している。また、ファンクタの引数は、EssentialParticleI の配列、EssentialParticleI の個数、EssentialParticleJ の配列、EssentialParticleJ の個数、Force 型の配列である。また、本チュートリアル中では、Force は密度の計算と、流体相互作用計算の 2 つが存在するため、calcForceEpEp 型は 2 つ書く必要がある。以下は本チュートリアル中で用いる calcForceEpEp 型の例である。

ソースコード 5: calcForceEpEp 型

```
1  class CalcDensity{
2          public:
3          void operator () (const EP* const ep_i, const PS::S32
                 Nip, const EP* const ep_j, const PS::S32 Njp,
                 Dens* const dens){
4                  for(PS::S32 i = 0 ; i < Nip ; ++ i){
5                          dens[i].clear();
6                          for(PS::S32 j = 0 ; j < Njp ; ++ j){
7                                  const PS::F64vec dr = ep_j[j].
                                         pos - ep_i[i].pos;
8                                  dens[i].dens += ep_j[j].mass *
                                         W(dr, ep_i[i].smth);
9                          }
10                 }
11         }
12 };
```

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

5.3.4 プログラム本体

5.3.4.1 概要

本説では、FDPS を用いて SPH 計算を行うにおいて、メインルーチンに書かれるべき関数に関して、解説する。

5.3.4.2 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、メインルーチンに記述する。

ソースコード 6: 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、メインルーチンの最後に次のように記述する。

ソースコード 7: FDPS の終了

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

5.3.4.3 初期化

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

5.3.4.3.1 オブジェクトの生成

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

ソースコード 8: 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;
```

5.3.4.3.2 領域クラスの初期化

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

ソースコード 9: 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), PS::F64vec(
      box.x, box.y, box.z));
```

5.3.4.3.3 粒子群クラスの初期化

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

ソースコード 10: 粒子群クラスの初期化

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

5.3.4.3.4 相互作用ツリークラスの初期化

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

ソースコード 11: 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
      ());
```

5.3.4.4 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。

5.3.4.4.1 領域分割の実行

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

ソースコード 12: 領域分割の実行

```
1 dinfo.decomposeDomain();
```

5.3.4.4.2 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、粒子群クラスのメンバ関数である、以下の関数を用いる。

ソースコード 13: 粒子交換の実行

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

5.3.4.4.3 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、tree クラスのメンバ関数である、以下の関数を用いる。

ソースコード 14: 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);
```

5.3.5 コンパイル

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

```
$ make
```

5.3.6 実行

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

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

もし、MPI を用いて実行する場合は、以下のコマンドを実行すればよい。

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

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

5.3.7 ログファイル

計算が終了すると、result フォルダ下にログが出力される。

5.3.8 可視化

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

```
$ gnuplot
```

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

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

5.4 (穴埋め式で)N 体シミュレーションコードを書く

5.4.1 作業ディレクトリ

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

```
$ cd (FDPS)/tutorial/nbody
```

5.4.2 ユーザー定義クラス

5.4.2.1 概要

本節では、FDPS の機能を用いて SPH の計算を行うにおいて、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。

5.4.2.2 FullParticle 型

ユーザーは Full Particle 型を記述しなければならない。Full Particle 型には、シミュレーションを行うにあたって、N 体粒子が持っているべき全ての物理量が含まれている。また、本チュートリアル中の N 体コードは、Full Particle 型が Essential Particle I 型ないし Essential Particle J 型を兼ねている。また、Full Particle 型には、データのコピーするのに必要なメンバ関数 copyFromFP と copyFromForce を持たせている。その他、粒子質量を返す関数である getCharge、粒子座標を返す関数である getPos が必要になる。また、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。また、これらに加え、LeapFrog 法を用いて時間積分をする際に用いるメンバ関数である、predict と correct が記述してある。

ソースコード 15: FullParticle 型

```
1  class FPGrav{
2  public:
3      PS::F64    mass;
4      PS::F64vec pos;
5      PS::F64vec vel;
6      PS::F64vec acc;
7      PS::F64    pot;
8      PS::F64vec vel2;
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 predict(PS::F32 dt) {
36          pos  = pos  +       vel * dt + 0.5 * acc * dt * dt;
37          vel2 = vel  + 0.5 * acc * dt;
38      }
39
40      void correct(PS::F32 dt) {
41          vel  = vel2 + 0.5 * acc * dt;
42      }
43  };
44
45  PS::F64 FPGrav::eps = 1.0 / 32.0;
```

5.4.2.3 calcForceEpEp 型

ユーザーは calcForceEpEp 型を記述しなければならない。calcForceEpEp 型には、Force の計算の具体的な内容を書く必要がある。今回のチュートリアル中では、ファンクタを用いて実装している。また、ファンクタの引数は、EssentialParticleI の配列、EssentialParticleI の個数、EssentialParticleJ の配列、EssentialParticleJ の個数、Force 型の配列である。以下は本チュートリアル中で用いる calcForceEpEp 型の例である。

ソースコード 16: calcForceEpEp 型

```
1  template <class TParticleJ>
2  struct CalcGravity{
3      void operator () (const FPGrav * iptcl,
4                        const PS::S32 ni,
5                        const TParticleJ * jptcl,
6                        const PS::S32 nj,
7                        FPGrav * force) {
8
9          PS::F64 eps2 = FPGrav::eps * FPGrav::eps;
10
11         for(PS::S32 i = 0; i < ni; i++){
12
13             PS::F64vec posi = iptcl[i].pos;
14             PS::F64vec acci = 0.0;
```

```
15              PS::F64    poti = 0.0;
16
17              for(PS::S32 j = 0; j < nj; j++){
18                  PS::F64vec posj   = jptcl[j].pos;
19                  PS::F64    massj  = jptcl[j].mass;
20
21                  PS::F64vec drvec  = posi - posj;
22                  PS::F64    dr2    = drvec * drvec + eps2;
23                  PS::F64    drinv  = 1.0 / sqrt(dr2);
24                  PS::F64    mdrinv = drinv * massj;
25
26                  poti -= mdrinv;
27                  acci -= mdrinv * drinv * drinv * drvec;
28              }
29
30              force[i].acc += acci;
31              force[i].pot += poti;
32          }
33      }
34  };
```

5.4.3 プログラム本体

5.4.3.1 概要

本説では、FDPS を用いて N 体計算を行うにおいて、メインルーチンに書かれるべき関数に関して、解説する。

5.4.3.2 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、メインルーチンに記述する。

ソースコード 17: 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、メインルーチンの最後に次のように記述する。

ソースコード 18: FDPS の終了

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

5.4.3.3 初期化

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

5.4.3.3.1 オブジェクトの生成

今回のチュートリアルでは、粒子群クラス、領域クラスに加え、重力計算用 tree を一本生成する必要がある。以下にそのコードを記す。

ソースコード 19: 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;
```

5.4.3.3.2 領域クラスの初期化

ユーザーはオブジェクトを作成したら、そのオブジェクトの初期化を行う必要がある。本チュートリアルの N 体計算中では、周期境界などは用いていないため、領域クラスの初期化は以下の一文だけでよい。

ソースコード 20: 領域クラスの初期化

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

5.4.3.3.3 粒子群クラスの初期化

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

ソースコード 21: 粒子群クラスの初期化

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

5.4.3.3.4 相互作用ツリークラスの初期化

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

ソースコード 22: 相互作用ツリークラスの初期化

```
1 tree_grav.initialize(ntot);
```

5.4.3.4 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。

5.4.3.4.1 領域分割の実行

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

ソースコード 23: 領域分割の実行

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

5.4.3.4.2 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、粒子群クラスのメンバ関数である、以下の関数を用いる。

ソースコード 24: 粒子交換の実行

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

5.4.3.4.3 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、tree クラスのメンバ関数である、以下の関数を用いる。

ソースコード 25: 相互作用計算の実行

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

5.4.3.4.4 時間積分

5.4.3.4.5 *predict*

タイムステップの最初で、predict を行い、粒子の座標と速度の情報を更新する。

ソースコード 26: predict

```
1 predict(system_grav, dtime);
```

5.4.3.4.6　correct

力の計算が終わったら、粒子の速度を correct する。

ソースコード 27: correct

```
1 correct(system_grav, dtime);
```

5.4.4　ログファイル

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

ソースコード 28: 標準エラー出力

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

6　サンプルコード

7　ユーザーサポート

FDPSを使用したコード開発に関する相談は以下のメールアドレスfdps-support@mail.jmlab.jpで受け付けています。以下のような場合は各項目毎の対応をお願いします。

7.1　コンパイルできない場合

ユーザーには以下の情報提供をお願いします。

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

7.2　コードがうまく動かない場合

ユーザーには以下の情報提供をお願いします。

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

7.3　その他

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

8 ライセンス

MIT ライセンスに準ずる。標準機能のみ使用する場合は、Iwasawa et al. (2015 in prep), Tanikawa et al. (2016 in prep) の引用を義務とする。

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つの文献を引用することを義務とする。

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) の引用を義務とする。