FDPS講習会の手引 (Fortran版)

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

2025年9月12日

目 次

1 準備

1.1 自分で用意した計算機で実行する場合

https://github.com/FDPS/FDPS から FDPS の最新版をダウンロードし、好きなディレクトリ下で解凍する。これによってディレクトリ FDPS-master が出来る。
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

  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 https://github.com/FDPS/FDPS.git
```

以下では、ウェブサイトからダウンロードした場合を想定し、ディレクトリ FDPS-master があるディレクトリの名前を fdps とする。

2 実習本番

実習で行うことは、FDPS を使って実装された重力 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を使用することである。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を使用する。

なお、実習の際に Makefile を更新し、コンパイルし直すという作業を何回か行うが、ここで注意しなくてはならないのは、Makefile **を編集しただけでは実行ファイルの再作成は行われない**ということである。この場合、きちんと前回作った実行ファイルを明示的に "$ rm ./nbody.out" などで消す必要がある。これを忘れた場合、「make: 'nbody.out' は更新済みです」と出る。

2.1 重力 N 体シミュレーションコード

ここでは、重力 N 体シミュレーションコードでの cold collapse を、並列環境無し、OpenMP を用いた並列計算環境、OpenMP + MPI を用いた並列計算環境の 3 つで行う。MPI は、環境があれば行う。

2.1.1 概要

ここでは、用意された重力 N 体シミュレーションコードを動かしてみよう。このコードは、重力多体系のコールドコラプスを計算する。この節でまず行うことは、シリアルコードのコンパイルと実行、出て来た結果の解析である。次にシリアルコードを Phantom-GRAPE を用いて高速化して、その速さを体験しよう。最後に OpenMP や MPI を利用して、さらにコードを高速化する。

2.1.2 シリアルコード

以下の手順で本コードを使用できる。

- ディレクトリ `fdps/FDPS-master/sample/fortran/nbody` に移動
- `make` を実行
- ジョブの投入
- 結果の解析
- OpenMP/MPI の利用 (オプション)

2.1.2.1 ディレクトリ移動

ディレクトリ `fdps/FDPS-master/sample/fortran/nbody` に移動する。

```
$ cd fdps/FDPS-master/sample/fortran/nbody
```

2.1.2.2 make の実行

`make` コマンドを実行する。

```
$ make
```

2.1.2.3 計算の実行

まずは、インタラクティブ実行で計算を実行する。これは、生成された実行ファイルの名前をそのまま実行すればよい。

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

正しくジョブが終了すると、標準入出力の最後には以下のようなログが出力されるはずである。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:    9.5000000000E+000, energy error:   -3.5093659762E-003
time:    9.6250000000E+000, energy error:   -3.5829042035E-003
time:    9.7500000000E+000, energy error:   -3.6160022980E-003
time:    9.8750000000E+000, energy error:   -3.5109567381E-003
******** FDPS has successfully finished. ********
```

ただし、後述する Phantom-GRAPE を用いた場合、energy error 数値は変わるので注意する。

2.1.2.4 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"snap000xx-prc000yy.dat"ができている。xx、yy はいずれも 0 埋めされた整数で、前者は時刻を、後者は MPI のランク番号 (逐次実行の場合には必ず 0 となる) 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/snap00009-proc00000.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)。

2.1.3 Phantom-GRAPE の利用

以下では、相互作用計算に Phantom-GRAPE を使う場合について、記述する。この場合、ユーザーはまずは Phantom-GRAPE 用の `Makefile` を利用環境 (使用するコンパイラ等) に合わせて適切に変更する必要がある。今回使う Phantom-GRAPE のソースコードは、`fdps/FDPS-master/src/phantom_grape_x86/G5/newton/libpg5/` 以下に存在するので、そこまで移動し、`Makefile` を編集する。以下は、現在 `sample/fortran/nbody/`に居る場合の例である。

```
$ cd ../../../src/phantom_grape_x86/G5/newton/libpg5/
$ vi Makefile
```

よほど古い CPU でなければ、`enable_avx2 = yes` のコメントアウトを外す。

編集が終わったら、元のディレクトリに戻る (サンプルコード側の `Makefile` で自動的に Phantom-GRAPE ライブラリをビルドする設定になっているので、ここでコンパイルする必要はない)。次に、`Makefile` の修正を行う。`Makefile` の 34 行目に Phantom-GRAPE を使用するか否かを決定しているスイッチが存在している。このスイッチはデフォルトではコメントアウトされているため、以下のようにしてコメントアウトを解除する。

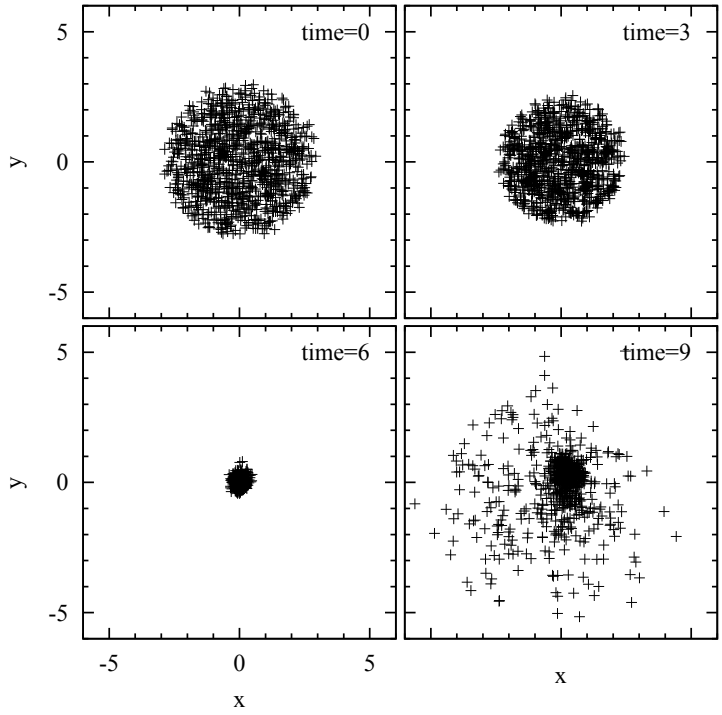

図 1:

```
use_phantom_grape_x86 = yes
```

make コマンドを実行する。

```
$ make -B
```

ここで、 -B オプションをつけているのは、ソースファイルは変更していないが再コンパイルさせるためである。単に make では、変更がないためコンパイルされないことに注意する。このあとも同様で、Makefile だけを変更した場合にはその直後では make -B とする必要がある。

無事にコンパイルが通れば、以降の実行・解析の手順は同様である。実行直後に次のような表示がされれば、正しく実行ができている。

```
******** FDPS has successfully begun. ********
====================================
 Paralleization infomation:
   # of processes is 1
   # of thread is    1
====================================
rsqrt: MSE = 1.158186e-04,  Bias = 8.375360e-08
(以下省略)
```

2.1.4 PIKG の利用

以下では、PIKG で生成された相互作用計算カーネルを使う場合について記述する。まず Makefile の修正を行う。Makefile の 35 行目に PIKG を使用するか否かを決定しているスイッチが存在している。このスイッチはデフォルトではコメントアウトされているため、以下のようにしてコメントアウトを解除する。

```
use_pikg_x86 = yes
```

make を実行し無事にコンパイルが通れば、以降の実行・解析の手順は同様である。

2.1.5 OpenMP/MPI の利用

OpenMP や MPI を利用する場合について以下に記述する。

- OpenMP のみ使用の場合
  - Makefile の編集
    - マクロ FC, CXX に、それぞれ OpenMP 対応の Fortran, C++コンパイラを代入する。
    - “FCFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp” の行のコメントアウトを外す
    - “CXXFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp” の行のコメントアウトを外す
  - 環境変数 OMP_NUM_THREADS の値を使用したいスレッド数にする。利用環境が Linux あるいは UNIX で、シェルが bash の場合、スレッド数を 4 に設定するには以下を実行する。

    ```
    $ export OMP_NUM_THREADS = 4
    ```

  - make コマンドを実行する。

– 実行方法はシリアルコードの場合と同じである。

```
(省略)
******** FDPS has successfully begun. ********
====================================
 Paralleization infomation:
   # of processes is 1
   # of thread is    4
====================================
(以下省略)
```

`# of thread is 4` と表示されている。これで、4 スレッドでの並列計算が行われている事が確認できた。

- OpenMP と MPI の同時使用の場合
  - `Makefile` の編集
    - マクロ `FC`, `CXX` に、それぞれ MPI 対応の Fortran, C++コンパイラを代入する。
    - “`FCFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`” の行のコメントアウトを外す
    - “`CXXFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`” の行のコメントアウトを外す
    - “`FCFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`” の行のコメントアウトを外す
    - “`CXXFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`” の行のコメントアウトを外す
  - 環境変数 `OMP_NUM_THREADS` の値を使用したいスレッド数に設定する。
  - `make` コマンドを実行する。
  - システムの MPI 環境の方法で実行する。(例えば、 mpirun -np 2 ./nbody.out) 正しく実行された場合、以下のように表示されるはずである。

```
(省略)
******** FDPS has successfully begun. ********
====================================
 Paralleization infomation:
   # of processes is 2
   # of thread is    2
====================================
(以下省略)
```

`# of processes is 2`、`# of thread is 2` と表示されており、2 プロセス 2 ス

レッドでの並列計算が行われている事が確認できた。

2.2 SPH シミュレーションコード

2.2.1 概要

ここでは、SPH シミュレーションコードを動かす。用意されているコードは、衝撃波管問題の計算を行う。この節でまず行うことは、シリアルコードのコンパイルと実行、出て来た結果の解析である。最後に OpenMP や MPI を利用して、さらにコードを高速化する。

2.2.2 シリアルコード

以下の手順で本コードを使用できる。

- ディレクトリ `fdps/FDPS-master/sample/fortran/sph` に移動
- `make` を実行
- ジョブの投入
- 結果の解析
- OpenMP/MPI の利用 (オプション)

2.2.2.1 ディレクトリ移動

ディレクトリ `fdps/FDPS-master/sample/fortran/sph` に移動する。

2.2.2.2 make の実行

`make` コマンドを実行する。

2.2.2.3 計算の実行

まずは、インタラクティブ実行で計算を実行する。これは、生成された実行ファイルの名前をそのまま実行すればよい。

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

正しくジョブが終了すると、標準入出力の最後には以下のようなログが出力されるはずである。

```
(省略)
================================
time  =   1.1809149264408111E-001
nstep =     54
================================
================================
time  =   1.2037329341359981E-001
nstep =     55
================================
******** FDPS has successfully finished. ********
STOP 0
```

2.2.2.4 結果の解析

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”snap000xx-proc00yy.dat”となっている。xx, yy は 0 埋めされた整数で、前者が時刻を、後者が MPI のランク番号を表す。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。

以下のコマンドを実行すれば、横軸に x、縦軸に密度の図が作成される。

```
$ gnuplot
> plot "result/snap00040-proc00000.dat" using 3:9 w p
```

正しい答が得られれば、図 2 のような図を描ける。

2.2.3 OpenMP/MPI の利用

OpenMP や MPI を利用する場合を以下に示す。

- OpenMP のみ使用の場合
  - Makefile の編集
    - マクロ FC, CXX に、それぞれ OpenMP 対応の Fortran, C++コンパイラを代入する
    - “FCFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp” の行のコメントアウトを外す
    - “CXXFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp” の行のコメントアウトを外す
  - 環境変数 OMP_NUM_THREADS の値を使用したいスレッド数にする。
  - make コマンドを実行する。

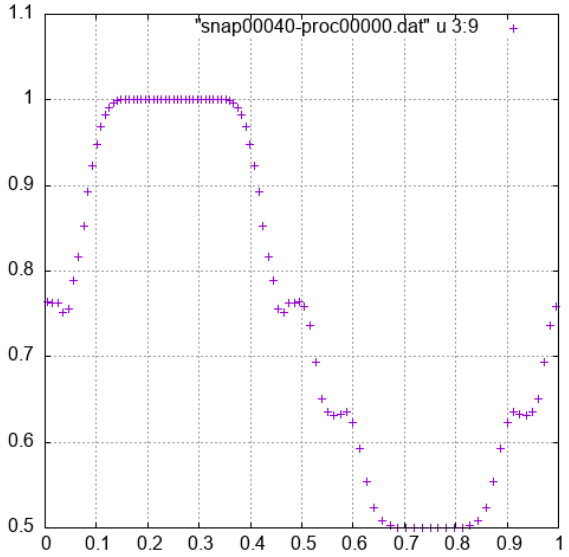

図 2:

– シリアルコードと同じように実行する。正しく実行された場合、以下のように表示されるはずである。

```
******** FDPS has successfully begun. ********
====================================
 Paralleization infomation:
   # of processes is 1
   # of thread is    2
====================================
```

- OpenMP と MPI の同時使用の場合
  - `Makefile` の編集
    - マクロ `FC`, `CXX` に、それぞれ MPI 対応の Fortran, C++コンパイラを代入する
    - “`FCFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`” の行のコメントアウトを外す
    - “`CXXFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`” の行のコメントアウトを外す
    - “`FCFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`” の行のコメントアウトを外す
    - “`CXXFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`” の行のコメントアウトを外す
  - 環境変数 `OMP_NUM_THREADS` の値を使用したいスレッド数にする。

– make コマンドを実行する。
– システムの MPI 環境の使い方に従って実行する。正しく実行された場合、以下のように表示されるはずである。

```
******** FDPS has successfully begun. ********
====================================
 Paralleization infomation:
   # of processes is 2
   # of thread is    2
====================================
```