J. Plasma Fusion Res. Vol.96, No.2 (2020) 57‐63

解説

# 超並列粒子法シミュレーションプログラム自動生成ツールの紹介 ～並列プログラミングが初心者にもできる！～

# Introduction to Auto-Generator Tool of Massive Parallel Simulation Code

野村昴太郎[1,2]，沼田龍介[3]，八柳祐一[4]，行方大輔[1,2]，岩澤全規[1,2]，牧野淳一郎[1,2]
NOMURA Kentaro[1,2], NUMATA Ryusuke[3], YATSUYANAGI Yuichi[4], NAMEKATA Daisuke[1,2],
IWASAWA Masaki[1,2] and MAKINO Junichiro[1,2]
[1]神戸大学惑星科学研究センター，[2]理化学研究所　計算科学研究センター，
[3]兵庫県立大学大学院シミュレーション学研究科，[4]静岡大学教育学部

（原稿受付：2019年10月18日）

本稿では，私達が開発・公開している多体シミュレーションプログラム開発フレームワーク「FDPS（Framework for Developing Particle Simulator）」を紹介する．FDPS は，粒子シミュレーションを研究に使っている多くの研究者が，並列化や計算機アーキテクチャ固有のチューニングに多大な時間を費やすことなく，自分の扱いたい問題向けのシミュレーションプログラムを容易に作成できるようになることを目標として開発したフレームワークである．

**Keywords:**
FDPS, particle simulation, plasma simulation

## 1．はじめに

我々が FDPS（Framework for Developing Particle Simulator）を開発した動機は，現在，大規模な粒子シミュレーションをする，特に，そのためのプログラムを開発することが非常に大変になっている，ということである．FDPS はこの開発を容易にすることをめざす．

30年前であれば，研究に使える粒子シミュレーションのプログラムを研究者，あるいは学生が自分で書くことはそれほど大変ではなかった．普通に書いた上で，ベクトル化等をすればよかったからである．

しかし，現在では，そう簡単ではない．まず，1CPU の計算機でも，CPU コアが複数あり，コアの中には SIMD ユニットがあるため，これらを有効に使えるかどうかで数十から数百倍性能が変わる．また，キャッシュメモリも有効に使わないと逆に性能が桁で落ちることもある．さらにスーパーコンピュータを，と思うと MPI を使った並列化が必要である．これらをあわせるとそもそも個人では困難，というレベルの大変な作業になる．

大変になっている要因を整理してみると，次のようにまとめられる．

- 並列化の階層の増加
- メモリボトルネックの発生とそれに対応したメモリ階層の増加・複雑化

以下，それぞれについて簡単に述べる．

現代の典型的な大規模並列計算機は，

- 複数の演算を並列に行う SIMD 演算ユニットを（コア内 SIMD）
- 複数もつ演算コアを（スーパースカラー）
- 複数もつプロセッサチップを（マルチコア）
- 複数ネットワークで接続した（分散メモリ並列）

構成を持つ．つまり，4つの違うレベルで並列動作する複数のユニットを持つ．しかも，それぞれのレベルで，どういうふうにすれば上手く並列化できるか，が違う．この中で，「スーパースカラー」というのは，CPU が，機械語プログラムの中から，並行して実行できそうなものを同時に実行する機能である．したがって，これはある程度ハードウェアがやる．しかし，他の3つのレベルのために固有の最適化が必要で，しかも分散メモリ並列では MPI を使ってプログラム全体を書き換え，例えば空間分割をして粒子を移動させるといった処理も書く必要がでてくる．

並列化の複数のレベルの扱いをさらに困難なものにしているのが，メモリ階層の存在である．現代の計算機では，演算器のほうがメモリより速いので，演算器の性能を生かすためには容量は小さいけれど高速なキャッシュメモリを使う．

しかし，この，キャッシュメモリは，並列計算機とは決して相性のいいものではない．マルチコアプロセッサでは，それぞれのコアが独立にキャッシュをもちたいわけだが，そうすると，あるコアがメモリのどこかに書いても，その同じアドレスのデータを別のコアが自分のキャッシュ

*corresponding author's e-mail: kentaro.nomura@riken.jp*



©2020 The Japan Society of Plasma Science and Nuclear Fusion Research

にもっているとか，あるいはまだ誰かのキャッシュの中でしか更新されていなくて主記憶には古いデータがあるものを主記憶から読んだ時にその誰かのキャッシュの中の新しいデータが来ないといけないといった問題が起きる．こういったややこしい状況でもちゃんと整合性がある結果を保証するのが「コヒーレントキャッシュ」というもので，そのためにコア間で複雑なやりとりをする大規模なハードウェアが必要になる．

このようなシステムで性能を出すためにはいかにしてキャッシュを制御するかが重要になり，キャッシュの特性を考慮したプログラムを書く必要がでてくる．最近のプロセッサでは 3 レベルから 4 レベルのキャッシュを持つようになっている．しかし，密行列乗算ならともかく，それ以外の計算アルゴリズムではこのような複数レベルのキャッシュを有効に使えるとは限らないし，できたとしてもプログラムは極めて複雑なものにならざるを得ない．

このため，単純なアルゴリズムでも，最新の高性能プロセッサで性能を出すのは容易なことではなくなっている．

実際，今この文章を読んでおられる読者の中でも，俺はMPIで並列化してキャッシュも有効利用しSIMD演算器も使ってプラズマシミュレーションプログラムを書いた，あるいは書ける，あるいは書く気がある，という人はあまり多くないのでは，と思う．

では，どうすればいいのか？というのがここでの問題である．多くの場合にとられているアプローチは，大規模なソフトウェアを開発チームを作ることでなんとか開発しよう，というもので，実際，様々な応用分野で，粒子法の並列化されたプログラムが公開され，利用可能になっている．

しかし，そういったプログラムは，あらかじめ開発グループが実装した機能を，開発グループがターゲットにしたマシン・OS で使うことしかできないのが普通である．もちろん，オープンソースで公開されているものでは，原理的にはソースコードを修正していろいろな機能を実装できるわけであるが，巨大で複雑なプログラムで，さらに特定のアーキテクチャ向けの最適化されたコードを修正して動くようにするのは容易なことではないのは，やってみようと思ったことがある人は良くご存じのことかと思う．

なので，自分でプログラムを開発できるようにしたいわけだが，それにはどうすればいいのか，というのが我々の問題意識である．我々が提案する方法は，特に MPI にかかわるような複雑な並列化とそのために必要なプログラムと，実際に扱う系の記述や時間積分の方法の記述とを明確に分離することである．

明確に分離，と書くのは簡単だが，実際にどのように実現するか，十分に色々なことを表現するにはどうするか，計算速度がでるようにするにはどうするのか，と，いろいろな問題がある．本解説では，まず基本的な考え方を 2 節で，またいくつかの FDPS をつかったサンプルを 3 節で紹介する． 4 節はまとめである．

1，2，4 節は牧野他 FDPS チームが，3 節はユーザ代表として野村，沼田，八柳が担当した．

## 2．FDPS の基本的な考え方

大規模並列化が可能で実行性能も高い MPI 並列化粒子法シミュレーションコードを「だれでも」開発できるようにする，というのが，私たちが FDPS によって実現しようとしていることである．ここで，「だれでも開発できる」というのは，具体的には，FDPS を使うと

- MPI を使った並列化は FDPS が勝手にやる
- OpenMP を使った並列化も FDPS が勝手にやる
- 長距離相互作用も短距離相互作用も，FDPS を使う人は粒子間の相互作用を計算する関数さえ用意すればよくてツリー法とかネイバーリストを使った高速化は FDPS が勝手にやる

ということを意味する．

我々が非常に単純な粒子系のプログラムを書く時には

1．MPI や OpenMP を使った並列化はしない
2．長距離力も短距離力もまずは全粒子からの力を単純に計算する
3．相互作用計算ループも単純に書く．SIMD 化とか特に意識しない

というふうにしたい．そういうふうに単純に作ったプログラムを，あまり手をかけないで並列化・高速化できることが目標である．

そんなうまい話が本当にあるか，というわけだが，粒子系のシミュレーションプログラムはどういう構造をしているか，を考えてみると，基本的な構造は以下のようになる．

1．粒子の初期分布を作る（ファイルから読む・内部で生成する）
2．粒子間相互作用を計算して，各粒子の加速度を求める
3．速度をアップデートする（時間刻みの中央まで）
4．位置をアップデートする
5．ステップ 2 に戻る

ここでは時間積分にはリープフロッグを使うとした．粒子法ではリープフロッグや，それと等価な方法を使うことが多いと思う．ルンゲクッタ等を使うなら相互作用の計算を中間結果を使って行う必要があるが，いずれにしても相互作用を計算する部分とそれ以外をする部分があるのは同じである．

これを，MPI で並列化された，領域分割し，領域毎に自分の担当の粒子を持つプログラムにするとすれば，

1．粒子の初期分布を作る（ファイルから読む・内部で生成する）
2．領域分割のしかたを決める
3．粒子を担当するプロセスが持つように転送する
4．粒子間相互作用を計算して，各粒子の加速度を求める
5．速度をアップデートする（時間刻みの中央まで）
6．位置をアップデートする
7．ステップ 2 に戻る

となる．主な違いは，ステップ 2，3 が入ること，ステップ

4では，各プロセスは自分の担当の粒子の加速度を計算するのに必要な情報を他のプロセスからもらってこなければいけないことである．

ステップ1は，大規模並列では入力ファイルを並列に読む必要があったりして若干面倒だが，難しいものではない．ステップ5，6も，単純に全粒子に対して，計算された加速度を使って時間積分の公式を適用するだけで，特に難しいことはない．積分公式がなにか違うものでも，全粒子に適用する，ということには変わりない．

FDPSがすることは，上のステップ2，3，4のそれぞれについて，そのためのライブラリ関数を提供することである．ステップ2，3のためには，ライブラリ関数は粒子を表すデータがどういうものかを知る必要があるし，また，MPIプロセス間で計算時間の差がでないようにするためになんらかの計算負荷に関する情報をもらう必要もある．

ライブラリが空間分割を決めるためには，例えば粒子の座標の配列をもらうことができればいいが，粒子を転送するためにはその粒子がどういうデータから構成されるか，メモリにどう配置されているかをライブラリが知る必要がある．

FDPSでは，C++言語のクラスを使って，オブジェクトとして粒子を表現してもらうことで，ユーザプログラム側で定義した粒子データの操作をライブラリ側で行うことを可能にしている．具体的には，粒子が「オブジェクト」で表現されることで，粒子の代入（コピー）が可能になり，また粒子の位置や電荷・質量を返す関数を提供してもらうことで，長距離力のためのツリー構造を作ることもできる．さらに，相互作用を計算する関数は，粒子データ自体を受け取って相互作用を計算する関数をユーザ側で定義することで，ライブラリ側は粒子データの中身を知らないままで相互作用を計算する関数を呼び出せる．

FDPSはこのようにC++言語の機能を使って実装されているが，現在はFortran言語（2003以降），C言語にも対応している．近代的なFortranでは，C++のクラスに相当する「構造型」があり，また，C言語で作った構造体や関数をFortran側からアクセスする方法も，言語仕様として定義された．これらの機能を利用して，C言語やFortranで書かれたユーザプログラムからもFDPSを利用可能にしている．

FDPSを使って粒子系シミュレーションプログラムを書くには

- 粒子を「構造体」で表現する
- 粒子間相互作用をその粒子構造を引数にとる関数で書く

というFDPS側の要請に従う必要があるが，それによって，ユーザプログラムはMPIによる並列化をほとんど意識することなく，MPI並列でないプログラムと同程度，ないしはより少ない手間で書くことができる．また，コンパイル時のフラグを変えるだけで，MPIを使わないこともできるし，OpenMPでの並列化も，またMPIとOpenMPを組み合わせたハイブリッド並列もできる．したがって，ハイブリッド並列でないと大規模な実行ができないスーパーコンピュータ「京」のような計算機でも高い性能を出すことができるし，その同じプログラムをノートパソコンで走らせることもできる．

もちろん，実際にはコンパイルされたプログラムではFDPS経由でMPIが呼ばれている．ということは，ユーザが書いた同一のプログラムがコンパイルしなおすだけで複数のMPIプロセスで並列に実行されたり，1コアで実行されたりする．MPI実行の場合には，FDPSが勝手に領域分割をして粒子をプロセス間で交換する．

このため，ユーザプログラムから見ると知らないうちに勝手に粒子が入れ替わって，粒子の数もFDPSが変更する．もちろん，MPIプロセス全部で見ると，どこかに粒子があるが，調べなければどこにあるかはわからない．これは領域分割する粒子系プログラムでは必ず起こることで，調べられるようにしておくためには，粒子データ構造体の中に粒子のインデックスをいれておくのが普通のやり方である．

## 3．実例

### 3.1　単純な分子動力学計算

本節では，希ガスの単原子分子を模したLennard-Jones（LJ）粒子のシミュレーションコード例について説明する．LJ粒子$i$と$j$の2体間相互作用は以下のポテンシャルで表される．

$$U_{\mathrm{LJ}}(r_{ij})=\begin{cases}4\epsilon\left\{\left(\dfrac{\sigma}{r_{ij}}\right)^{12}-\left(\dfrac{\sigma}{r_{ij}}\right)^{6}\right\} & (\text{if } r_{ij}<r_{\mathrm{c}})\\ 0 & (\text{else})\end{cases} \quad (1)$$

$\epsilon$と$\sigma$はそれぞれポテンシャルの深さと粒子の大きさを特徴づけるパラメータで，$r_{\mathrm{c}}$はカットオフ距離である．本解説では簡単のために$\epsilon=\sigma=1$としている．

紙面の都合上，最低限必要な部分のみを解説していく．ある程度，CもしくはC++の知識があることを前提としている．Fortranインターフェースを用いる場合でも，同様の情報を書く必要があるので参考にはなると思われる．コード全体についてはhttps://github.com/FDPS/JSPF_Sampleを参照してほしい．

まず，コードの本体であるmain.cppである．ここでは，

1．FDPSのヘッダーファイルのインクルード
2．粒子データクラス（クラス名FP，EP，Force）の定義
3．粒子間相互作用関数（関数名Kernel）の定義
4．main関数の定義

が行われている．FDPSのヘッダーファイルのインクルードはファイルの先頭で行われている．

```
1 #include <particle_simulator.hpp>
2 using namespace PS;
3 using namespace std;
```

また，ここでは記述を短くするために名前空間PS（ParticleSimulatorの略）とstd（C++標準ライブラリ）で定義さ

れている変数や関数，クラスを使うことを宣言している（PS::が省略されている）．

粒子データクラス（構造体）の定義では，粒子が持たなくてはならない情報とFDPSが必要とするデータを操作する関数を定義する．FPクラスについて見てみよう．FPはFullParticleクラスの略で，時間発展や相互作用を計算するにあたって必要な情報全てを持つ粒子データクラスである．

```
struct FP{
  F64vec  r,v,f; //  座標，速度，力
  F64     p;     //  ポテンシャル
  void copyFromForce(const Force& _f)
  {f = _f.f; p = _f.p;}
  F64vec getPos() const {return r;};
  void setPos(const F64vec& _r){r = _r;}
  void kick(const F64 dt){v += f*dt;}
  void drift(const F64 dt){r += v*dt;}
};
```

粒子データクラスが持つメンバ変数として座標，速度，力，ポテンシャルエネルギーと，FDPSがこのデータを操作するのに必要なメンバ関数（copyFromForce，getPos，setPos）が定義されている．また，kickとdriftという粒子の時間発展を行うメンバ関数が定義されている．このように，FDPSが必要としていないメンバ関数もユーザの必要に応じて追加することができる．粒子データクラスの定義では，更にForceとEP（EssentialParticleの略）クラスを定義している．それぞれ，相互作用計算を行う際の結果（力とポテンシャル）と相互作用計算に必要（Essential）なデータ（ここでは座標のみ）を持ったクラスである．それぞれについて，FDPSが必要とする関数が少しずつ異なる．このように相互作用計算のみに使われる粒子データクラスを定義して相互作用計算のときに利用することにより，メモリ使用量の削減や最適化を行いやすくしている．EPクラスは，$i$ 粒子と $j$ 粒子（相互作用を受ける粒子とおよぼす粒子）それぞれについて定義できるが，本コードでは両方に同じEPクラスを用いている．最も簡単にコードを書く場合，Forceクラス，EPクラスを定義せずにFPクラスを全ての用途で使い回すこともできる．

相互作用関数の定義では，上記のポテンシャルから力の計算を行っている．この関数は，$i$ と $j$ 粒子のリストとその長さ（epi，epjとni，nj）を受け取って，結果のリスト（force）に計算結果を書き込んでいる．FDPSはFPのリストから $i$ と $j$ 粒子のリストを作り，相互作用関数を呼び出し，結果をFPに書き戻す．この際，分散メモリ上の並列計算については，複数のプロセス間での通信などが必要に応じて行われる．

```
void Kernel(const EP *epi,const S32 ni,
            const EP *epj,const S32 nj,
            Force *force){
  const F64 rc2 = RCUT*RCUT;
  for(S32 i=0; i<ni; i++){
    F64vec ri = epi[i].r, fi = force[i].f;
    F64 pi = force[i].p;
    for(S32 j=0; j<nj; j++){
      F64vec rij = ri - epj[j].r;
      const F64 r2 = rij * rij;
      if(r2==0.0 || r2>rc2) continue;
      const F64 r2i = 1.0/r2;
      const F64 r6i = r2i * r2i * r2i;
      fi += r6i*(48.0*r6i-24.0)*r2i * rij;
      pi += 4.0*r6i*(r6i-1.0);
    }
    force[i].f = fi;
    force[i].p = 0.5*pi;
  }
}
```

main関数の定義では，シミュレーションを行うのに必要なインスタンスを作成し，実際にシミュレーションを行う．main関数では，まず最初にPS::Initializeを呼び出す．この関数はFDPSに必要な初期化を行う（主にMPIプロセスの初期化など）．

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

次にParticleSystemやDomainInfo，TreeForForceを定義した粒子データクラスを使用してインスタンスにして，各インスタンスにおいてInitialize関数を呼び出す．本コードでは，初期条件として粒子を格子上に配置しているが，ここでは解説を省略する．

```
ParticleSystem<FP> ps;
ps.initialize();
```

```
DomainInfo di;
di.initialize(0.3);
di.decomposeDomainAll(ps);
ps.exchangeParticle(di);
TreeForForceShort<Force,EP,EP>::Scatter t;
t.initialize(3*n*n*n,0.0,64,256);
t.calcForceAllAndWriteBack(Kernel,ps,di);
```

以下のループでは実際にシミュレーションを進めている．時間発展（kick, drift）を行い，空間の分割（DomainInfo::decomposeDomainAll），粒子の分配（ParticleSystem::exchangeParticle），力の計算（TreeForForce::calcForceAllAndWriteBack）を行っている．

```
S64 nl = ps.getNumberOfParticleLocal();
for(int s=0;s<10000;s++){
  for(int i=0;i<nl;i++){
    ps[i].kick(dth);
    ps[i].drift(dt);
  }
  ps.adjustPositionIntoRootDomain(di);
  di.decomposeDomainAll(ps);
  ps.exchangeParticle(di);
  nl = ps.getNumberOfParticleLocal();
  t.calcForceAllAndWriteBack(Kernel,ps,di);
  for(int i=0;i<nl;i++) ps[i].kick(dth);
  const  F64 pot = AccumulatePotential(ps);
```

```
14   const F64 kin = CalcKineticEnergy(ps);
15   if(Comm::getRank()==0)
16     cout << scientific << pot << " " << kin
17                << " " << pot+kin << endl;
18 }
```

最後に PS::Finalize を行い，終了処理を行う．

```
1 Finalize();
```

次に Makefile を見てみよう．FDPS_PATH は github からダウンロードしてきたファイルのパスである．FDPS を用いたコードをコンパイルする際には$(FDPS_PATH)/src/particle_simulator.hpp を include する必要があるためヘッダーファイルの検索パスを追加している．MPI を用いて並列化する場合は，MPI 用コンパイラを用いて，PARTICLE_SIMULATOR_MPI_PARALLEL マクロを有効にする必要がある．OpenMP を利用する際は，OpenMP 用のオプションを有効化して PARTICLE_SIMULATOR_THREAD_PARALLEL マクロを有効化することでスレッド並列化を行うことができる．このとき，時間発展などのユーザー記述部分はスレッド並列化されないので注意されたい．また，両方を用いるハイブリッド並列化も可能である．

```
1  INC+= -I$(FDPS_PATH)/src
2  CXX=g++
3  CXXFLAGS= -O2 $(INC)
4  #FL+= -DPARTICLE_SIMULATOR_THREAD_PARALLEL
5  #FL+= -DPARTICLE_SIMULATOR_MPI_PARALLEL
6  SRC=main.cpp
7
8  all:   argon.out
9  argon.out:      $(SRC) Makefile
10        $(CXX) $(FL) $(SRC) -o argon.out
```

LJ 粒子のサンプルコードの解説は以上である．シリアルなコードをコンパイル時にマクロを変更することで簡単に並列化が行えることがわかってもらえただろうか．

### 3.2 プラズマの例 1：Coulomb 相互作用するプラズマのシミュレーション

分子動力学モデルを用いたプラズマの研究として，強結合プラズマ（下記で定義するプラズマパラメータ $\Lambda$ が 1 より小さい）の物性を取り扱ったものがある[1]．ここでは，結合の強さを限定せず，主に弱結合の核融合・宇宙プラズマのダイナミクスを議論する目的で粒子モデルに基づくシミュレーションを行う．同様のアプローチの研究が[2,3]にある．また，Escande らは粒子モデルを用いて理論的に Debye 遮蔽や Landau 減衰を議論している[4]．本節では，周期境界領域内（領域は一辺 $L$ の立方体とする）で Coulomb 相互作用するプラズマの集団現象として Debye 遮蔽のシミュレーション例を示す．Coulomb 相互作用する $N$ 個の粒子の運動方程式は以下のように書ける．

$$\frac{\mathrm{d}\hat{\boldsymbol{r}}_i}{\mathrm{d}\hat{t}}=\hat{\boldsymbol{v}}_i,\qquad \frac{\mathrm{d}\hat{\boldsymbol{v}}_i}{\mathrm{d}\hat{t}}=\frac{1}{\Lambda_0}\sum_{j=1,j\neq i}^{N}\frac{\hat{q}_i\hat{q}_j}{\hat{m}_i}\frac{\hat{\boldsymbol{r}}_i}{|\hat{\boldsymbol{r}}_i-\hat{\boldsymbol{r}}_j|^3}. \qquad (2)$$

下付き添え字 $i=1,\cdots,N$ は粒子番号，$\boldsymbol{r}_i$，$\boldsymbol{v}_i$ は粒子の位置，速度，$q_i$，$m_i$ は粒子の電荷，質量である．ハット記号は無次元量をあらわし，各変数は以下のように無次元化されている．

$$\hat{\boldsymbol{r}}_i=\frac{\boldsymbol{r}_i}{\lambda_{\mathrm{D0}}},\quad \hat{\boldsymbol{v}}_i=\frac{\boldsymbol{v}_i}{v_{\mathrm{t0}}/\sqrt{2}},\quad \hat{t}=t\omega_{\mathrm{p0}}$$

$$\hat{m}_i=\frac{m_i}{m_0},\quad q_i=\frac{q_i}{q_0}$$

下付き添え字 0 は，参照粒子をあらわす．

$$\omega_{\mathrm{p0}}\equiv\sqrt{\frac{n_0q_0^2}{\epsilon_0m_0}},\qquad \lambda_{\mathrm{D0}}\equiv\sqrt{\frac{\epsilon_0T_0}{n_0q_0^2}} \qquad (3)$$

は，　それぞれ参照プラズマのプラズマ周波数，Debye 長であり，これらを用いると熱速度は $v_{\mathrm{t0}}=\sqrt{2}\lambda_{\mathrm{D0}}\omega_{\mathrm{p0}}$ で与えられる．$\Lambda_0\equiv4\pi n_0\lambda_{\mathrm{D0}}^3$ は参照プラズマのプラズマパラメータであるが，Coulomb 相互作用の強さを規定するパラメータである．なお，密度と長さの単位は独立ではないため系全体の（平均）密度 $n_{\mathrm{avg}}$ は重み $w$ を用いて以下のように無次元化される：$n_{\mathrm{avg}}=wN/L^3=(w/\lambda_{\mathrm{D0}}^3)(N/\hat{L}^3)$．ここで，$n_0=w/\lambda_{\mathrm{D0}}^3$ となるので，$\Lambda_0=4\pi w$ に他ならない．

コードは，FDPS に含まれるサンプルコード nbody の重力相互作用を Coulomb 相互作用に置き換えればほぼ完成である．周期境界条件を実現するためには，Coulomb 力を粒子—粒子と粒子—メッシュの相互作用の和で表現する Particle-Mesh 法（FDPS の拡張機能として提供されている）を用いるが，この部分もサンプルコード p3m を参考に作成した．

サイズ $L=\lambda_{\mathrm{D0}}$ の領域の中心に固定された電荷 $q_{\mathrm{i}}$ のイオンを置き，$N_{\mathrm{e}}=8{,}192$ 個の電子によるポテンシャルの遮蔽のシミュレーションを行う．$\Lambda_0=10$ とした．イオンの電荷と質量を $q_{\mathrm{i}}=N_{\mathrm{e}}q_{\mathrm{e}}$，$m_{\mathrm{i}}=10^{10}m_{\mathrm{e}}$ とする（立体の添え字 i, e はそれぞれイオン，電子をあらわす）．電子の初期位置はランダムに配置し，初速はゼロとする．系の全エネルギーが保存するミクロカノニカルアンサンブルを考えているため，電子はある程度の時間が経過した後で温度一定の熱平衡状態に達する（**図 1**）．平衡状態における系全体の平均電子温度は $T_{\mathrm{e}}^{\mathrm{sat}}/T_0\approx440$ となった．このとき電子のプラズマパラメータは $\Lambda_{\mathrm{e}}\approx10^3$，Debye 長は $\lambda_{\mathrm{De}}/L\approx0.23$ となる．**図 2** に $y$ 方向の静電ポテンシャル分布を示す（$x/L=0.5$，$z/L=0.5$ とした）．電子によるポテンシャルの遮蔽がよく再現できることが確認された（境界付近でのずれは，サンプル粒子数が十分でないためであると思われる）．ここでは，系の全エネルギーが一定のアンサンブルを考えているが，温度一定のカノニカルアンサンブルを考えるほうが実際のプラズマとの比較がしやすい．温度を一定に保つ熱浴の実装を検討している．また，ここでは粒子数が$10^4$程度に留まり，FDPS の威力は十分発揮されているとは言えないが，今後大規模計算を実施する際にその効果を体感できることを期待している．

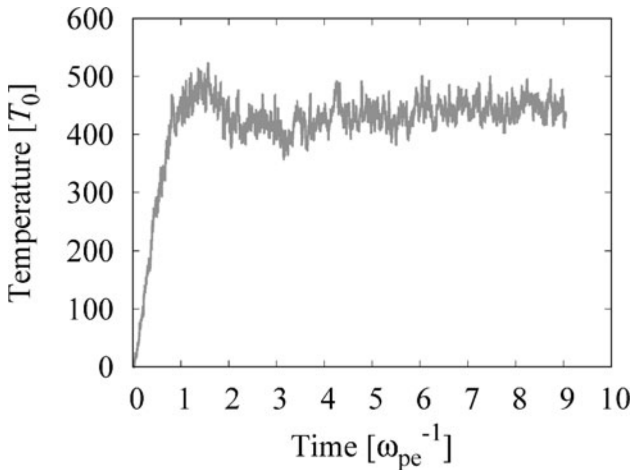


図1 電子温度の時間変化．温度は $x$，$y$，$z$ 各方向でほぼ等しいため，3方向の平均温度を取った．

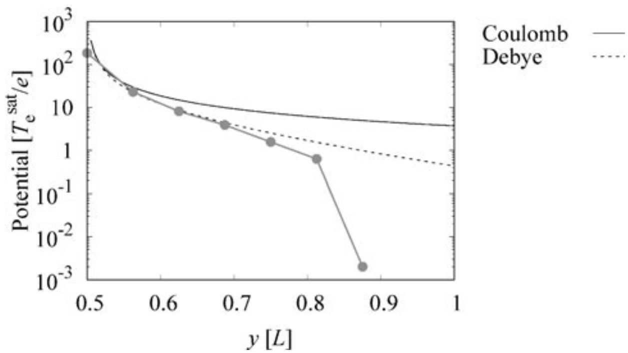


図2 静電ポテンシャルの空間分布．

### 3.3 プラズマの例2：純電子プラズマ実験に対応した2次元点渦シミュレーション

本節では，FDPSの開発チームより，2次元系の計算が正しく行えるか，チェックを行ってほしい旨の依頼を受けたので，点渦系シミュレーションをFDPSを用いて実装し検証を行った．なお，点渦系は，モデルに内在する運動論的効果により[5]，現象を再現できる下限の粒子数で計算を行った結果が，一番理にかなったものになるケースが多いため，FDPSが得意とする多粒子，多並列でのシミュレーションを行う意味が薄い．よって，本解説は，FDPSで2次元シミュレーションも作成できることを示すことを第一目的とし，ベンチマーク結果は掲載しない．

2次元点渦シミュレーションでターゲットとする物理系は，純電子プラズマ実験である．一般的なプラズマと異なり，電気的中性条件が破れたプラズマを非中性プラズマとよび[6]，特に電子のみから構成される場合，純電子プラズマとよぶ．軸方向に強磁場を掛けた真空容器内における磁場に垂直な断面内の電子の運動は，Larmor半径＝0の極限（案内中心近似）で，$\boldsymbol{E}\times\boldsymbol{B}$ ドリフト運動となり，その時間発展方程式は2次元非圧縮流体方程式であるEuler方程式（正確には，回転微分をとった渦度方程式）で記述される[7]．このとき，流れ関数が静電ポテンシャル，渦度が電子の数密度に比例する*1．この関係により，純電子プラズマ実験では電場と磁場により流体内部が制御可能な流体実験を行うことができ，渦結晶に代表される自己組織化現象を観測してきた[8,9]．非粘性渦度方程式は，点渦法で数値的に解くことが可能なため，筆者（八柳）は，純電子プラズマ実験で見られる自己組織化現象とリンクした理論／シミュレーション研究を展開してきた．

2次元点渦系では，渦度方程式

$$\frac{\partial\omega_z(\boldsymbol{r},t)}{\partial t}+\nabla\cdot(\omega_z(\boldsymbol{r},t)\boldsymbol{u}(\boldsymbol{r},t))=0 \quad (4)$$

の「形式的」解をDiracのデルタ関数 $\delta(\boldsymbol{r})$ で表現された点の集合体とする．点の総数は $N$ で表す．

$$\omega_z(\boldsymbol{r},t)=\sum_{i=1}^{N}\Omega_i\delta(\boldsymbol{r}-\boldsymbol{r}_i) \quad (5)$$

ここで $\omega_z(\boldsymbol{r},t)$ は渦度，$\Omega_i$，$\boldsymbol{r}_i$ は $i$ 番目の点渦の強さを表す循環，2次元位置ベクトルである．点渦の時間発展は，離散化されたBiot-Savart積分となる．

$$\frac{\mathrm{d}\boldsymbol{r}_i}{\mathrm{d}t}=-\frac{1}{2\pi}\sum_{j=1(\neq i)}^{N}\Omega_i\frac{(\boldsymbol{r}_i-\boldsymbol{r}_j)\times\hat{\boldsymbol{z}}}{|\boldsymbol{r}_i-\boldsymbol{r}_j|^2}+\frac{1}{2\pi}\sum_{j=1}^{N}\Omega_i\frac{(\boldsymbol{r}_i-\overline{\boldsymbol{r}}_j)\times\hat{\boldsymbol{z}}}{|\boldsymbol{r}_i-\overline{\boldsymbol{r}}_j|^2} \quad (6)$$

右辺第2項は半径 $R$ の円形境界を表現する $\overline{\boldsymbol{r}}_i=R^2\boldsymbol{r}_i/|\boldsymbol{r}_i|^2$ に置かれた鏡像渦からの影響を表す．

数値シミュレーションでは，適当な初期分布で $\boldsymbol{r}_i$ を初期化したあと，相互作用(6)を計算する．今回は，わかりやすい例として，Kelvin-Helmholtz（Diocotron）不安定性の計算を行った．結果を**図3**に示す．初期設定から求めた線形不安定モードは，不安定な順に2と3である[10]．その中から，数値揺らぎにより，多くのエネルギーが蓄積されたモード3が立ち上がっていると考えられる．また，$T=120, 140$ のプロットから，境界をまたいで越境してしまう粒子がいないことから，円形境界も鏡像渦によって十分な精度で計算されていることがわかる．

紙面の残りを使って，今回，著者がつまずいた点，特にMPIに慣れていないことに由来する困難について述べてみたい．FDPSでは領域分割技法であるTree法や，疎結合並列計算のためのAPIであるMPIに関する知識はまったく必要ない．しかし，FDPS側でどのような内部処理を行っているのか，その概要を理解していないと，どうにも解決できないバグを埋め込む結果となる*2．粒子系におけるクーロン力の計算は，力を計算する場所の粒子（$i$ 粒子）それぞれについて，力の源泉となる粒子（$j$ 粒子）からの相互作用を計算する．このとき，非MPI環境下であれば，粒子に関連付けられる添字の値が変化することはないので，同一粒子間で発散する相互作用計算を回避する目的で，ループ内において $i\neq j$ という条件が使用可能である．しかし，FDPSの場合，各粒子は時々刻々変化する位置に応じて領域ごとに配分しなおされ，各MPIノードへ渡される $i$ 粒子

＊1 この関係の導出過程はベクトル解析のよい練習問題なので，学生の皆さんはぜひチャレンジしてみてください．
＊2 開発チームのみなさま，問題解決に協力していただき，大変感謝しております．

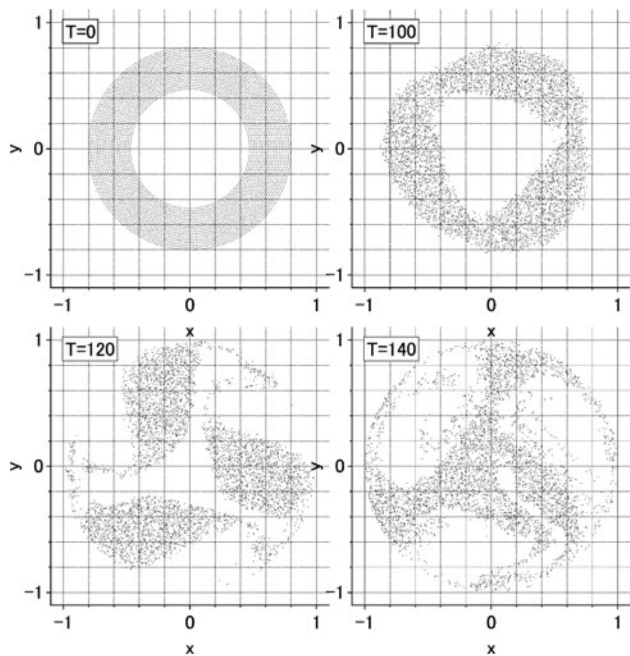


図 3　円形境界半径を 1 として，ドーナツの外側半径 = 0.8，内側半径 = 0.46，総粒子数 = 4177，ドーナツの内部も点渦で満たされた円盤の剛体回転タイムスケール = 10.8．

の個数をほぼ同一にするような仕組みをもっている．このため，粒子の添字が時々刻々変化するため，自己相互作用計算を排除する目的で $i \neq j$ という条件は使用できず，$i$ 粒子と $j$ 粒子の距離が0.0超か否かにより判定することになる．指摘されてみれば当たり前の結論であるが，MPI でのプログラミング経験がないと，障壁になりやすいと感じたので，敢えてここに tips として掲載した．

## 4．おわりに

本稿では，粒子系大規模並列化のためのフレームワーク FDPS を紹介した．現代の大規模計算では MPI 等での並列化が必須になっているが，FDPS を使うことでその部分は自分で開発しなくても，高性能な並列プログラムを実現できる．本稿が読者の皆様の研究のためのプログラム開発の一助になれば幸いである．

### 参 考 文 献

［1］S. Hamaguchi, J. Plasma Fusion Res. **78**, 313 (2002).
［2］A. Panarese *et al.,* J. Plasma Phys. **84**, 905840308 (2018).
［3］C.-P. Wang and Y. Nishimura, IEEE Trans. Plasma Sci. **47**,1196 (2018).
［4］D.F. Escande *et al.,* Rev. Mod. Plasma Phys. **2**, 9 (2018).
［5］Y. Yatsuyanagi and T. Hatori, Fluid Dyn. Res. **47**, 065506 (2015).
［6］毛利明博：プラズマ・核融合学会誌 **77**, 213 (2001).
［7］際本泰士：プラズマ・核融合学会誌 **77**, 329 (2001).
［8］D.Z. Jin and D.H.E. Dubin, Phys. Rev. Lett. **84**, 1443 (2000).
［9］A. Sanpei, *et al.,* Phys. Rev. E **68**, 016404 (2003).
［10］R.C. Davidson, *Physics of nonneutral plasmas* (Addison-Wesley, Reading, 1990) §6.

のむらけんたろう
野村昴太郎

神戸大学大学院理学研究科惑星科学研究センター．分子動力学（MD）シミュレーションを用いたナノチューブ閉じ込め系の水に関する相挙動研究を行ってきた．あわせて，Xeon Phi や GPU，PEZY-SC など様々なプロセッサを用いた MD シミュレーションコード開発を行った．現在は，N 体，MD，SPH 向けに各種プロセッサに最適化された相互作用計算カーネル開発および，プレファードネットワークスと開発している深層学習向けプロセッサのシミュレーション向け応用を進めている．

やつやなぎゆういち
八柳祐一

静岡大学教育学部 准教授．長距離相関に支配されたN体系の自己組織化に関する理論／シミュレーション研究．最近，講談社青い鳥文庫版のシャーロック・ホームズ・シリーズを再読中．既読の新潮文庫版と比べて，19世紀末のロンドンの時代背景描写が小中学生向けに豊富なので，再発見が多数（冷や汗）．アイリーンの印象が変わりました．

ぬまたりゅうすけ
沼田龍介

兵庫県立大学大学院シミュレーション学研究科　准教授．2004年東京大学大学院新領域創成科学研究科修了・博士（科学）．磁気リコネクションや乱流などの非線形複雑現象のシミュレーション研究を行っています．粒子モデルでプラズマの乱流は再現できるでしょうか．昨年，KOBE 六甲全山縦走大会に参加，公称 55 km を約15時間かけて完走しました．人生で最長距離行でしたがなんとかなるものです．

なめかただいすけ
行方大輔

いわさわまさき
岩澤全規

神戸大学理学研究科 特命准教授 2009年東京大学総合文化研究科広域科学専攻修了 博士（学術）．専門は数値天文学．またそのためのソフトウェアやアルゴリズムの研究開発も行っている．

まきのじゅんいちろう
牧野淳一郎

神戸大学理学研究科惑星学専攻／理研計算科学研究センター．球状星団の多体シミュレーションから研究をスタートし，多体問題向け専用計算機　GRAPE　の開発から，「京」，富岳の開発にもかかわってきました．今はプレファードネットワークスと進めている深層学習向けプロセッサの様々なシミュレーションへの応用を進めています．