[] 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初心者にもできる！〜

神戸大学 惑星科学研究センター, 理化学研究所 計算科学研究センター
野村昴太郎, 行方大輔，岩澤全規，牧野淳一郎 makino@mail.jmlab.jp

概要
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 Developping Particle Simulators)」を紹介します。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す。連載第一回の今回は、FDPS の開発の背景、考え方と、実際にどのようなことができるか、という簡単な例を紹介します。

キーワード： FDPS, 粒子系シミュレーション

1 はじめに

我々が FDPS 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、ということです。FDPS はこの開発を容易にすることを目指すものです。

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ありませんでした。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す。

しかし、現在では、そう簡単ではありません。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ります。また、キャッシュメモリも有効に使わない逆に性能が桁で落ちることもあります。さらにスパコンを、と思うと MPI を使った並列化が必要です。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ります。

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ます。

- 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は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ます。
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)

構成を持っています。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っています。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、が違います。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す。なので、これはある程度ハードウェアがやってくれます。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きます。

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す。現代の計算機では、演算器のほうがメモリより速く、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っています。

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ありません。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ですが、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新しいデータにならないといけないといった問題があります。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です。

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きます。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ます。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ですし、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ません。

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ます。

実際、今この文章を読んでおられる読者の皆様の中でも、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、と思います。

では、どうすればいいのか？というのがここでの問題です。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ます。

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す。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すが、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がでるもの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います。

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ですが、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す。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す。

明確に分離、と書くのは簡単ですが、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ります。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 節で紹介します。4 節はまとめです。

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 (Framework for Developping Particle Simulators) によって実現しようとしていることです。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フレームワークが勝手にやる

ということを意味します。

私たち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わけで、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す。

そんなうまい話が本当にあるか、というわけですが、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、ということを考えてみると、基本的な構造は以下のようになります。

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ています。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います。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りますが、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す。

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

となるでしょう。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す。

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ですが、難しいものではありません。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、特に難しいことはありません。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ありません。

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す。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りますし、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貰う必要もあります。

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貰うことができればいいですが、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ります。

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貰うことで、ユーザー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することを可能にしています。具体的には、粒子が「オブジェクト」であることで、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貰うことで、長距離力のためのツリー構造を作ることもできます。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ー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ます。

FDPS はこのように C++ 言語の機能を使って実装されていますが、現在は Fortran 言語 (2003 以降)、C 言語にも対応しています。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ました。これらの機能を利用して、C 言語や Fortran で書かれたユーザープログラムからも FDPS を利用可能にしています。

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りますが、それによって、ユーザープログラムは 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ます。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、OpeMP 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ます。従って、ハイブリッド並列でないと大規模な実行ができない「京」のような計算機でも高い性能を出すことができますし、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ます。

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 が呼ばれています。ということは、ユーザーが書いたプログラムが複数の MPI プロセスを並列に実行されたり、1 コアで実行されたりするわけです。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します。

このため、ユーザー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 が変更することになります。もちろん、MPI プロセス全部で見ると、どこかに粒子がありますが、調べなければどこにあるかはわかりません。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す。

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．

$$U_{\rm LJ}(r_{ij}) = \begin{cases} 4\epsilon \left\{ \left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6} \right\} (\text{if } r_{ij} < r_c) \\ 0 \ (\text{else}) \end{cases} . \quad (1)$$

ϵ と σ 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_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://XXX.org 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．

```
#include <particle_simulator.hpp>
using namespace PS;
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な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，相互作用計算を行なう際の結果 (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な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なう (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，drifp) 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);
  const F64 kin = CalcKineticEnergy(ps);
  if(Comm::getRank()==0)
    cout << scientific << pot << "␣" << kin
         << "␣" << pot+kin << endl;
 }
```

最後に PS::Finalize を行い，終了処理を行う．

```
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なうことができる．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．

```
INC+= -I$(FDPS_PATH)/src
CXX=g++
CXXFLAGS= -O2 $(INC)
#FL+= -DPARTICLE_SIMULATOR_THREAD_PARALLEL
#FL+= -DPARTICLE_SIMULATOR_MPI_PARALLEL
SRC=main.cpp

all: argon.out
argon.out: $(SRC) Makefile
        $(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分かってもらえただろうか．

3.2 プラズマの例 1

3.3 プラズマの例 2

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 を紹介しました。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ますが、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ます。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す。

謝辞

参考文献