[] 連載 FDPS 入門 (1)

神戸大学　理学研究科/理研 計算科学研究機構
牧野淳一郎　jmakino@people.kobe-u.ac.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： 粒子シミュレーション、分子動力学、HPC、並列化、MPI

1　はじめに

今号から 4 回にわたって、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)」[1] を紹介します。第一回となる本稿では、

- FDPS を開発している目的、背景はなにか
- FDPS は具体的には何をしてくれるか

といったあたりを紹介しようと思います。

2　FDPS 開発の背景と動機

2.1　背景

まず、背景はなにか、です。これは、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。30 年前であれば、粒子シミュレーションをしよう、と思った時に、自分でプログラムを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、チューニングといえばベクトル化程度だったからです。

ところが、現在では、「普通にプログラムを書く」のではどうにもならなくなっています。まず、1 台のデスクトップ PC や Intel CPU がのったサーバーでプログラムを走らせるにしても、CPU コアが複数あり、またそのコアの中には複数の演算を並列に行う SIMD ユニットがあるため、これらを有効に使うかどうかで性能が数十倍変わります。Intel の Xeon Phi CPU では、CPU コアが 60 個程度あり、またその中に、単精度だと 16 演算を並列に行う SIMD 演算ユニットが 2 つありますから、これらを有効に使うかどうかで最大 1000 倍におよぶ性能差が生じることになります。

昔だってスパコンを使うならチューニングとかベクトル化とかしたものだよ、と 30 年前に現役だった読者の方は思うかもしれませんが、現代における「並列化とか高度なチューニング」は当時の「ベクトル化」とは全く比較にならない、はるかに大変な作業になっています。

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。1970 年代から 80 年代前半にかけてのベクトル計算機は、半導体メモリが磁気コアメモリにとってかわった時代のものであり、演算器に対して十分な速度でデータを供給することができました。しかし、実はこんなことができたのは歴史的にはこの時期と、計算機の黎明期である 1950 年代だけで、それ以外では常に、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。

これは必要なものではありますが、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に、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 FMM や PME を実装して高速な並列 MD 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。

2.2 困難を減らす方法

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、というのが私たちの問題意識です。私たちが提案する方法は、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を、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。実際に詳細にどうするか、は連載に 2 回目以降で詳しく説明されますので、第一回である今回では、基本的な考え方を紹介しようと思います。

3 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、

- MPI を使った並列化はフレームワークが勝手にやる
- OpenMP を使った並列化もフレームワークが勝手にやる
- 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

というふうに FDPS を作る、ということを意味します。

これを言い換えると、要するに、私たちが非常に単純な粒子系のプログラムを書くと、

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。分子動力学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) 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、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 が適当にいじっていることになります。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。

分子動力学計算では、原子間の結合を表現したいことがあります。FDPS では今のところその辺は考えないで領域分割して粒子を移動するので、同じ分子に属する原子が別の MPI プロセスに分かれたりします。分子結合等の表現については連載 2 回以降で扱います。

4 FDPS の性能

並列化をなんのためにするかというと、計算速度を速くするためです。ですから、並列化はできますがあまり速くありません、となってしまっては FDPS の開発目標を実現したとはいいがたいと考えます。

私たちが FDPS を開発するにあたって設定した目標は、

- ある程度粒子数が大きい計算では
- 世界最大規模の計算機であっても、また粒子分布が著しく不均一でも
- 理論的に実現できる限界に近い性能をだせる
- さらに、アクセラレータをもうようなシステムでも高い実行効率を実現する

というものです。「ある程度」の意味は曖昧ですが、現在の通常の並列計算機では MPI プロセスあたり数万粒子、計算時間では 1 タイムステップあたり数十ミリ秒程度以上の、極端に少ない粒子数ではないところです。この程度以下になると、プロセス間通信の遅延時間等のために高い性能を出すのは現在の大規模な汎用並列計算機では困難です。

上の目標を実現するためには、

- MPI プロセス数が多い時に、理想に近いロードバランスを実現する。特に、計算時間が延びてしまうプロセスがないようにする
- 計算量のほとんどを占める、相互作用計算の計算カーネルを十分に最適化する。また、計算量が減るアルゴリズムを採用する
- 相互作用計算以外のところが足を引っ張らないよう、十分に効率的なアルゴリズムを採用し、実装も高速化する

といったことが必要になります。実際にこれが実現できているか、ということですが、まあまあのところまではきているのではないかと考えてています。

私たちのところで、分子動力学のシミュレーションの大規模なテストはまだ行っていませんが、重力多体系や、SPH 法での流体シミュレーションでは、相互作用計算が計算時間のほとんどになり、さらに「京」全ノードやその近くまでの大規模計算でも、粒子数をノード数に比例させてふやせば性能もノード数にほぼ比例して上がる、という、十分に良い結果がえられています。また、実行効率についても、理論ピークの 50% 程度が実現できています。

なお、このような高い効率を出すためには、相互作用計算の関数は十分最適化する必要があります。相互作用関数は現在のところユーザー側が提供するものであり、FDPS 側で勝手に最適化はしません。但し、OpenMP による並列化までは FDPS 側でやりますので、ユーザー側では 1 コアでの性能の最適化、具体的には SIMD 化や、多くの場合に性能ボトルネックになっている平方根や除算等の高速化が主にするべきことになります。

この、相互作用関数についても、抽象度の高い記述から自動的に対象とするプロセッサの命令セットやマイクロアーキテクチャに対して最適化したコードを出すといったことはできなくはなさそうですが、対応しないといけないアーキテクチャが多く、またコンパイラの状況他色々なものに左右されそうなので、まだ公開できるようなものはできていません。

5 簡単な例

と、いろいろ能書きを並べてきたわけですが、実際のところどんなコードを書くとどんなことができるのか、を最後に紹介します。サンプルにするのは、私が (かなり適当に) 書いた Lennard-Jones もどきの力で相互作用する分子動力学計算コードです。

これは実際にダウンロード・コンパイル・実行することができます。チュートリアル[1] の手順に従って FDPS をダウンロード・展開したら、ディレクトリ sample/c++/vdw_test に移動して、make; ./vdw_test.out を実行すれば

```
******** FDPS has successfully begun. *****
./result/t-de.dat
./result/t-tcal.dat
MakeUniformCube: n_loc=512 and n_1d=8
np_ave=512
time:  0.0625000 energy error: +4.895409e-05
time:  0.1250000 energy error: +8.980742e-05
time:  0.1875000 energy error: +8.766743e-05
...
time: 10.0000000 energy error: +1.590795e-04
******** FDPS has successfully finished. ***
```

というような出力がでるはずです。チュートリアルの、重力多体向けの記述を参考に、Makefile を修正すれば、OpenMP や MPI での並列実行もできます。

このコードで、FDPS とのやりとりに必須な

- 粒子クラス
- 相互作用関数

を定義するコードを以下に示します。

```
class FPLJ{
public:
    PS::S64 id;
    PS::F64 mass, pot;
    PS::F64vec pos, vel, acc;
    PS::F64 search_radius;
    void clear(){
        acc = 0.0; pot = 0.0;
    }
    PS::F64 getRSearch() const{
        return this->search_radius;
    }
    PS::F64vec getPos() const {return pos;}
    void copyFromForce(const FPLJ & force){
        acc = force.acc;
        pot = force.pot;
    }
// (I/O 用関数を省略)
    void copyFromFP(const FPLJ & fp){
        mass = fp.mass;
        pos = fp.pos;
        id = fp.id;
        search_radius = fp.search_radius;
    }
};

void CalcForceFpFp(const FPLJ * ep_i,
                   const PS::S32 n_ip,
                   const FPLJ * ep_j,
                   const PS::S32 n_jp,
                   FPLJ * force){
    const PS::F64 r0 = 3;
    const PS::F64 r0sq = r0*r0;
    const PS::F64 r0inv = 1/r0;
    const PS::F64 r0invp6 = 1/(r0sq*r0sq*r0sq);
    const PS::F64 r0invp7 = r0invp6*r0inv;
    const PS::F64 foffset = -12.0*r0invp6
                          *r0invp7+6*r0invp7;
    const PS::F64 poffset = -13.0*r0invp6
                          *r0invp6+7*r0invp6;
    for(PS::S32 i=0; i<n_ip; i++){
        PS::F64vec xi = ep_i[i].pos;
        PS::F64vec ai = 0.0;
        PS::F64 poti = 0.0;
        PS::S64 idi = ep_i[i].id;
        for(PS::S32 j=0; j<n_jp; j++){
            if( idi == ep_j[j].id ) continue;
            PS::F64vec rij = xi - ep_j[j].pos;
            PS::F64 r2 = rij * rij;
            if (r2 < r0sq){
                PS::F64 r_inv = 1.0/sqrt(r2);
                PS::F64 r = r2*r_inv;
                PS::F64 r2_inv = r_inv * r_inv;
                PS::F64 r6_inv = r2_inv * r2_inv * r2_inv;
                PS::F64 r12_inv = r6_inv * r6_inv;
                poti += r12_inv - r6_inv-foffset*r+poffset;
                ai += (12*r12_inv*r2_inv - 6*r6_inv*r2_inv
                       + foffset*r_inv) * rij;
            }
        }
        force[i].acc += ai;
        force[i].pot += poti;
    }
}
```

粒子クラスは FPLJ という名前で、id, mass, pos, vel, acc, pot, search_radius という変数を持ちます。粒子番号、質量、速度、加速度、ポテンシャル、ネイバーサーチの半径です。LJ ポテンシャル用には、短距離力計算する FDPS のライブラリ関数を使います。これにはサーチ半径を渡すので、それを与えるのが getRSearch です。また、 FDPS は getPos という名前の関数が位置を返すことを期待します。さらに、 copyFrom なんとか、という関数が 2 つありますが、これは、FDPS がキャッシュを考慮して、粒子に対していくつかのデータ型を定義できるようになっているために、その間のコピーのために定義が必要になっています。このサンプルではデータ型 1 つですますのですが、コピーする関数は必要です。 PS は FDPS で定義する名前空間、S64, F64 はそれぞれ 64 ビットの整数、浮動小数点数、F64vec は 3 次元ベクトル型です。C++ のクラスを利用して 3 次元ベクトル型を定義しています。

相互作用を計算する関数は、CalcForceFpFp という名前です。この関数の引数は、力を及ぼす粒子の配列、力を受ける粒子の配列、それぞれの粒子数、の 4 つでいいはずです。実際には、力を受ける粒子のデータを入

[1] https://github.com/FDPS/FDPS/blob/master/doc/doc_tutorial_cpp_ja.pdf

力と出力の 2 つにわけているために引数が 5 個になっています。なお、現在のところ、FDPS では力の対称性は使っていません。このため、計算量・通信量は短距離力については若干大きくなっています。今後、中点法のような、通信量・計算量を減らすアルゴリズムの実装も検討します。

以下に、メインプログラムから FDPS を呼んで、領域分割し、粒子交換し、近傍探索のためのツリー構造を作って粒子間相互作用を計算するところまでを示します。

```
PS::ParticleSystem<FPLJ> system_grav;
system_grav.initialize();
(ここで粒子データ設定)
PS::DomainInfo dinfo;
dinfo.initialize(coef_ema);
dinfo.setBoundaryCondition(
      PS::BOUNDARY_CONDITION_PERIODIC_XYZ);
dinfo.setPosRootDomain(
      PS::F64vec(-boxdh,-boxdh,-boxdh),
      PS::F64vec(boxdh,boxdh,boxdh));
dinfo.collectSampleParticle(system_grav);
dinfo.decomposeDomain();
system_grav.exchangeParticle(dinfo);
n_grav_loc = system_grav.getNumberOfParticleLocal();
PS::TreeForForceShort<FPLJ, FPLJ, FPLJ>::Scatter
       tree_grav;
tree_grav.initialize(n_grav_glb, theta,
          n_leaf_limit, n_group_limit);
tree_grav.calcForceAllAndWriteBack(CalcForceFpFp,
          system_grav, dinfo);
```

今回詳しい説明は省略しますが、基本的にはこれだけで、MPI を使った領域分割から相互作用計算までが終わります。CalcForceFpFp が 1 コアで最適化されていれば、OpenMP や MPI で並列化しても高い性能がでます。

なお、GPGPU 等アクセラレータにも対応していますが、現在のところ相互作用関数の書き方等が少し変わります。

次回以降では、より詳しい解説にはいっていきます。

参考文献

[1] M. Iwasawa, A. Tanikawa, N. Hosono, K. Nitadori, T. Muranushi, and J. Makino, *Publ. Astron. Soc. J.*, **68**, 54 (2016).

著者紹介

牧野淳一郎 (学術博)：〔経歴〕1989 年東京大学大学院総合文化研究科博士課程修了，同年東京大学大学教養学部助手．2016 年から現所属．〔専門〕天体物理、計算科学．