”FDPS: A Novel Framework for Developing High-Performance Particle Simulation Codes for Distributed-Memory Systems” のメモ

谷川

概 要

We have developed FDPS (Framework for Developing Particle Simulator), which enables researchers and programmers to develop high-performance particle simulation codes easily. The basic idea of FDPS is to separate the program code for complex parallelization including domain decomposition, redistribution of particles, and exchange of particle information for interaction calculation between nodes, from actual interaction calculation and orbital integration. FDPS provides the former part and the users write the latter. Thus, a user can implement a high-performance N-body code only in 120 lines. In this paper, we present the structure and implementation of FDPS, and describe its performance on two sample applications: gravitational N-body simulation and Smoothed Particle Hydrodynamics simulation. Both codes show very good parallel efficiency and scalability on the K computer. FDPS lets the researchers concentrate on the implementation of physics and mathematical schemes, without wasting their time on the development and performance tuning of their codes.

我々は高性能な粒子シミュレーションコードの開発を容易にするフレームワーク「FDPS (Framework for Developing Particle Simulator)」を開発した。FDPS の基本的な考えかたは、実際の相互作用計算や軌道積分と、並列化のための繁雑なプログラム (領域分割、粒子交換、相互作用計算のための粒子情報の交換など) を分離することである。FDPS が前者を提供するため、ユーザーは後者だけを書けばよい。従って、ユーザーは高性能な N 体コードをわずか 120 行で記述できる。この論文では、FDPS の構造とその実装、2 つのサンプルアプリケーションの性能を示す。どちらのコードも、京コンピュータ上で、良い効率とスケーラビリティを示した。FDPS のおかげで、研究者は自分のコードの開発やチューニングに時間をかける必要がなくなり、物理・数学スキームの実装に集中することができるようになる。

1 Introduction

この数十年間で、数値シミュレーションは実験、観測、理論による研究手法と同等の研究手法としての地位を確立した。数値シミュレーションの中でも、粒子シミュレーションは最も重要なシミュレーション手法の１つである。粒子シミュレーションは、系を粒子の集団とみなし、粒子の運動を追うことで系の進化を表現する。粒子シミュレーションは様々な理工学の分野で使われていて、たくさんのアプリケーションがある。例えば、重力 N 体シミュ

レーション、分子動力学シミュレーション、粉体シミュレーション、流体力学のためのメッシュフリーシミュレーションである。粒子シミュレーションは、格子シミュレーションに比べて、大きな変形を伴う系を扱うのに有利である。

時空方向にダイナミックレンジの大きい系の進化を追跡するために、高性能な粒子シミュレーションの必要性はますます高まっている。しかし、普通の研究者やプログラマーにとってこのような高性能な粒子シミュレーションコードを作成するのは難しい。このようなコードは大規模並列分散メモリ計算機で動作する必要がある。このような計算機で動作するためには、コードは以下のような処理を備えていなければならない。すなわち、ロードバランスのための動的な領域分割、領域分割に合わせた粒子情報の交換、相互作用を計算するための粒子情報の交換、である。さらに、研究者やプログラマーは、共有メモリ環境下でも高性能を達成するために、複数コア、キャッシュメモリ、SIMD ユニットの効率的な使用を考慮する必要がある。

これまで研究者、プロブラマー、それらのグループは個別にこのようなコードを開発してきた。しかし、これらのコードの設計、実装、テスト計算、デバッグには多大な時間がかかる。そのため、研究者やプログラマーはコード開発に時間をとられ、理工学の研究に専念できないでいる。

我々は、この状況を打破するために、高性能な粒子シミュレーションコードの開発を容易にするフレームワーク「FDPS (Framework for Developing Particle Simulator)」を開発した。FDPS を使えば、そのユーザーは、分散メモリ環境用の繁雑な処理や共有メモリ環境用のチューニングをすることなしに、高性能な粒子シミュレーションコードを作成できる。ユーザーがすることは、並列処理を行う FDPS ライブラリを呼出すことと、1 コア上での粒子間相互作用の計算と共有メモリ環境での粒子の軌道積分を実装することだけである。FDPS はすでに Github に公開されている。

FDPS の特長は、あらゆる粒子シミュレーションコードに対応していることである。これまでに、多くの高性能な粒子シミュレーションコードが開発されてきた。しかし、それらのコードはアプリケーションが限られていた。N 体シミュレーションなら GreeM、N 体+SPH なら Gadget や Gasoline、などのように。

この論文の構成は以下の通りである。2 節では、FDPS を使った粒子シミュレーションコードの実装方法を概観し、ユーザーが高性能化をする必要がないことを示す。3 節では、FDPS の中の実装を記述する。4 節では、FDPS のアプリケーションとして重力 N 体シミュレーションと SPH シミュレーションの性能を示す。これらのシミュレーションは科学的に重要な意義のあるものである。5 節ではこの論文をまとめる。

2　FDPS を使った粒子シミュレーションコードの実装

この節では、FDPS を使った粒子シミュレーションコードの実装方法を記述する。2.1 節では、実装方法を概観し、高性能化に伴う実装がユーザーから分離されていることを確認する。2.2 節では、サンプルコードを示し、実装方法を詳述する。

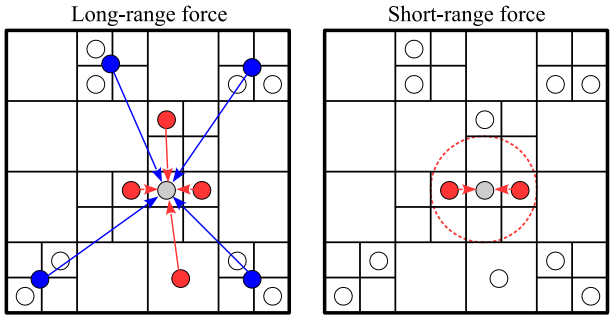

図 1: 長距離力と短距離力

2.1 概観

粒子シミュレーションコードは以下のような常微分方程式を解くために実装されるものである。

$$\frac{d\boldsymbol{u}_i}{dt} = \sum_j^N \boldsymbol{f}(\boldsymbol{u}_i, \boldsymbol{u}_j) + \boldsymbol{g}(\boldsymbol{u}_i) \tag{1}$$

ここで、N は全粒子の個数、$\boldsymbol{u}_i$ は i 粒子の物理量を表すベクトル、関数 $\boldsymbol{f}$ は j 粒子から i 粒子への作用を表す関数、関数 $\boldsymbol{g}$ は i 粒子自身の物理量だけで決まる作用 (外場など) である。

空間分割を必要に応じて動的に分割し、粒子を再配置する。

この 3 つに分けて上の 2 つは FDPS を呼出す。

メインルーチンで実行される処理は、1 タイムステップにつき、以下のようになっている。

1. FDPS の API を呼出してロードバランスをとる。式 (1) で言えば、どのプロセスがどの i 粒子の方程式を時間積分するかを決めることである。ロードバランスは以下のサブステップに分かれている。

   (a) 粒子の空間分布を反映させて、計算領域を分割する。各プロセスが担当する空間領域を決める。その空間領域にある粒子はそのプロセスが担当する。これは上で定義した粒子クラスを受け入れるテンプレートライブラリの API を呼出すことで実行される。

   (b) 粒子情報の交換。自分の担当する空間領域に存在する粒子の情報を得るために、他のプロセスと粒子情報の交換をする。これは上で定義した粒子クラスを受け入れるテンプレートライブラリの API を呼出すことで実行される。

2. FDPS の API を呼出して相互作用を計算する。上の方程式で言えば、$\sum$ 内を計算することである。この作業は、$\sum$ 内を計算するのに、必要な粒子情報を他のプロセスから受け取るという処理が必要となる。これは、上で定義した粒子クラスと相互作用関数を受け入れるテンプレートライブラリの API を呼出すことで実行される。

3. プロセス内の情報だけで閉じた計算をする。上の方程式で言えば、$\boldsymbol{g}(\boldsymbol{u}_i)$ の計算をすることや、$d\boldsymbol{u}_i/dt$ を使って時間積分をすることである。これは自分で記述する。

FDPS ユーザーは、粒子シミュレーションコードを実装するときに以下の 3 つを行うことになる。

- 粒子を定義する。これはある 1 つの粒子がどのような情報を持つかを決めることであり、式 (2) や (3) の物理量ベクトル $\boldsymbol{u}_i$ や $\hat{\boldsymbol{u}}_s$ の中身を決めることに対応する。$\boldsymbol{u}_i$ の中身には、例えば、粒子の質量、位置、速度、加速度などがある。これは C++のクラスという形で定義する。

- 相互作用を定義する。相互作用の定義とは、2 つの粒子の間に働く相互作用の形を決めることであり、式 (2) や (3) の関数 $\boldsymbol{f}$、$\hat{\boldsymbol{f}}$ を決めることに対応する。これは C++の関数オブジェクトという形で定義をする。

- メインルーチンを作成する。メインルーチンの中の並列処理は、FDPS が提供するテンプレートライブラリの API を呼出すことで実行される。このテンプレートライブラリは上で定義した粒子と相互作用を受け入れる形になっている。

これを概念図として書くと図 2 のようになる。

上で示した手順のように、ユーザーは並列処理を FDPS が提供するテンプレートライブラリの API を呼出すことで実行できる。そのため、ユーザーは並列処理の実装に頭を悩ます必要がない。ユーザーがメインルーチンの中で記述すべき部分は、時間積分などのプロセス内の情報だけで閉じた処理だけである。

粒子を集めるときには、ツリーアルゴリズムが使われて、高速になされるが、ユーザーはツリーを実装する必要はないし、意識する必要もない。

式 (1) の第 1 項は、全粒子からの i 粒子への作用を計算するということを示す。しかし、実際には、全粒子の数よりもはるかに少ない粒子からの相互作用しか計算しない。これは FDPS 内部で以下のような実装をしているからである。FDPS は相互作用を 2 つのタイプにわけ、1 つを無限遠まで到達する長距離力、もう 1 つを有限の距離しか到達しない短距離力とした。長距離力の場合には、ツリーアルゴリズムを用いて、遠くの複数の粒子を 1 つの超粒子としてまとめる (図 1 の左) ため、計算する相互作用の数は N よりはるかに少くなる。従って、FDPS では、長距離力の場合実質的に、以下の式を計算することになる。

$$\frac{d\boldsymbol{u}_i}{dt} = \sum_j^{N_{\rm J}} \boldsymbol{f}(\boldsymbol{u}_i, \boldsymbol{u}_j) + \sum_s^{N_{\rm S}} \hat{\boldsymbol{f}}(\boldsymbol{u}_i, \hat{\boldsymbol{u}}_s) + \boldsymbol{g}(\boldsymbol{u}_i) \tag{2}$$

ここで $N_{\rm J}$ と $N_{\rm S}$ はそれぞれ i 粒子に粒子として作用する粒子数と超粒子として作用する粒子数である。$\hat{\boldsymbol{u}}$ は超粒子の物理量、関数 $\hat{\boldsymbol{f}}$ は超粒子から i 粒子への作用を表す。短距離力の場合には、遠くの粒子との相互作用を計算する必要がない (図 1 の右)。従って、FDPS では、短距離力の場合実質的に、以下の式を計算することになる。

$$\frac{d\boldsymbol{u}_i}{dt} = \sum_j^{N_{\rm J}} \boldsymbol{f}(\boldsymbol{u}_i, \boldsymbol{u}_j) + \boldsymbol{g}(\boldsymbol{u}_i) \tag{3}$$

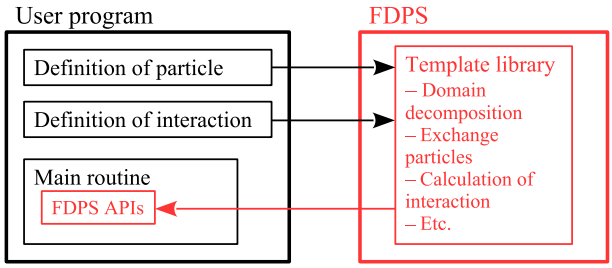

図 2: 概念図

2.2 Sample code

この節ではサンプルコードを示す。ここでは、アプリケーションとして重力 N 体シミュレーションを用いる。

重力 N 体シミュレーションを式 (1) に当てはめると以下のようになる。

$$\boldsymbol{u}_i = (m_i, \boldsymbol{r}_i, \boldsymbol{v}_i) \tag{4}$$

$$\boldsymbol{f}(\boldsymbol{u}_i, \boldsymbol{u}_j) = \left(0, 0, \frac{G m_j (\boldsymbol{r}_j - \boldsymbol{r}_i)}{(|\boldsymbol{r}_j - \boldsymbol{r}_i|^2 + \epsilon_i^2)^{3/2}}\right) \tag{5}$$

$$\boldsymbol{g}(\boldsymbol{u}_i) = (0, \boldsymbol{v}_i, 0) \tag{6}$$

ここで、G は重力定数、m_i、$\boldsymbol{r}_i$、$\boldsymbol{v}_i$ はそれぞれ粒子の質量、位置、速度、ϵ_i は粒子固有の重力ソフトニングである。

重力は無限遠まで到達する長距離力である。そのため、遠くの複数の粒子は超粒子として 1 つにまとめる。このサンプルコードでは、超粒子を複数の粒子のモノポールとして近似する。このとき、式 (1) の第 1 項は以下のように近似できる。

$$\sum_j^N \boldsymbol{f}(\boldsymbol{u}_i, \boldsymbol{u}_j) \sim \sum_j^{N_\mathrm{J}} \boldsymbol{f}(\boldsymbol{u}_i, \boldsymbol{u}_j) + \sum_s^{N_\mathrm{S}} \boldsymbol{f}(\boldsymbol{u}_i, \hat{\boldsymbol{u}}_s) \tag{7}$$

ここで、N_J は i 粒子に作用する粒子の数、N_S は i 粒子に作用する超粒子の数、$\hat{\boldsymbol{u}}_s$ は超粒子の物理量ベクトルである。物理量ベクトルの質量を超粒子をなす粒子の全質量、位置を超粒子をなす粒子の重心とおけば、超粒子から i 粒子への作用は、粒子から i 粒子への作用と同様の関数で記述できる。一般に、ユーザーは、粒子から粒子への作用の関数に加え、超粒子から粒子への作用の関数を記述する必要があるが、ここでは必要ない。

この N 体シミュレーションを FDPS を使って実装すると、ソースコード 1 のようになる。このコードをコピー&ペーストして、C++コンパイラまたは MPI 対応の C++コンパイラでコンパイルすれば、ただちに動作する。さらに、このコードは世界最高レベルの性能を持つ。FDPS を使えば、このようなコードが 120 行足らずで書くことができるのである。

このサンプルコードは 4 つの部分からなる。すなわち、FDPS のインストール、粒子を定義する部分、相互作用を定義する部分、メインルーチンとそこで呼出される関数の定義である。以下、それぞれについて記述する。

1 行目から 2 行目で FDPS のインストールを行っている。これはヘッダファイル particle_simulator.hpp をインクルードするだけですむ。FDPS が提供する API やクラスは名前空間 PS で囲まれている。ここでは “using namespace PS” として名前空間の明示を省略できるようにした。

4 行目から 35 行目では粒子、すなわち $\boldsymbol{u}_i$ を定義している。この粒子は、質量、ソフトニング長、位置、速度、加速度の情報を持つ。メンバ関数 getPos、getCharge、copyFromFP、clear、readAscii は FDPS のライブラリとのデータの入出力を定義するものである。メンバ関数 predict、correct はメインルーチンの中で粒子の軌道の時間積分を行うときに用いる。

37 行目から 61 行目では相互作用の関数、すなわち $\boldsymbol{f}$ の定義をしている。この関数の引数を見ると、i 粒子の配列 ip、i 粒子数 ni、j 粒子または超粒子の配列 jp、j 粒子数 nj、結果を受け取る配列 force がある。前述したように、j 粒子から i 粒子への作用の形と超粒子から i 粒子への作用の形は同じである。そのため、この関数をテンプレート関数とし、引数に j 粒子だけでなく超粒子も取れるようにしている。この関数の中では、for 文の 2 重ループを使って、ni 個の i 粒子に対する nj 個の j 粒子または超粒子の作用を計算している。

並列処理以外のところでも、ユーザーが高性能化を意識しなくて済むように配慮がなされている。相互作用の関数を定義する際には、複数の i 粒子に対して複数の j 粒子が作用するという形で定義する。このとき i 粒子、j 粒子ともに配列の形で与えられている。そのため単純な for 文による二重ループを記述すればよい。この配列は連続したメモリ上にメモリ確保されているため、単純な二重ループさえ書けば、キャッシュヒット率が非常に高い。また、ユーザーがループ内を SIMD 化するのも容易である。この配列が与えられるのは、各スレッドでの上でのことなので、マルチスレッドについても意識する必要がない。

残りの部分では、メインルーチンとそこで呼び出される関数を定義している。main 関数は 6 つの部分に分かれている。すなわち以下の通りである。

- 時刻の初期化 (92 から 94 行目)
- FDPS の初期化と終了 (95 と 115 行目)
- FDPS が提供するテンプレートライブラリのオブジェクトの生成と初期化 (96 から 102 行目)
- 粒子情報のファイルからの入力 (103 行目)
- 初期の重力の計算 (104 から 106 行目)
- リープフロッグ法を用いた常微分方程式の時間発展 (107 から 114 行目)

以下、2、3、4、5 番目について詳述する。1 番目は自明であるし、6 番目は時間積分の部分を除けば 5 番目の繰り返しである。

FDPS の初期化は MPI の初期化と OpenMP の初期化を行っており、FDPS の終了は MPI の終了を行っている。FDPS のテンプレートライブラリを使用するのは、この初期化と終了の間である必要がある。

次に行っているのは FDPS が提供するテンプレートライブラリのオブジェクトの生成と初期化である。このライブラリは DomainInfo クラス、ParticleSystem クラス、TreeForForce

クラスとして実装されている。それぞれのクラスのオブジェクトが持つ情報と API について以下に記す。

- DomainInfo クラス: 全プロセスに割当てられた計算領域の情報を持ち、計算領域の分割を行うメソッドの API を持つ。
- ParticleSystem クラス: 各プロセスの担当粒子の情報を持ち、プロセス間で粒子情報を交換するメソッドの API を持つ。このクラスはユーザーが定義する粒子クラスをテンプレート引数とするテンプレートクラスになっている。
- TreeForForce クラス: 相互作用計算を高速にするためのツリー構造をデータとして持ち、相互作用計算を行うメソッドの API を持つ。オブジェクトを生成する際に、計算する力のタイプ (TreeForForce“Long”)、計算すべき相互作用の情報と相互作用を計算するのに必要な i 粒子の情報と相互作用を計算するのに必要な j 粒子の情報 (テンプレート第１、２、３引数)、相互作用を計算するのに必要な超粒子の情報 (“Monopole”) を指定する。ここでは、テンプレート第１、２、３引数がすべて “Nbody” であるが、通信量の削減のために、それぞれのためにクラスを定義してテンプレート引数にすることが可能である。

オブジェクト ptcl への粒子情報のファイルからの入力がメソッド readParticleAscii によってなされる。この引数 argv[1] はコマンドライン上で実行する際の第一引数のことである。この API の中で粒子クラスで定義した readAscii が呼出され、実際に粒子にデータが入力される。

各粒子への重力が計算される。これは関数 calcGravAllAndWriteBack で実行される。この関数は main 関数の前に定義されている。以下、この関数の手順を説明する。各手順は、2.1 節の手順に対応する。

1. ロードバランスを以下の２ステップで取る。
   (a) 計算領域が分割される。これはオブジェクト dinfo のメソッド decomposeDomainAll が呼び出されて実行される。粒子分布が反映されるため、オブジェクト ptcl が引数となる。
   (b) 分割した計算領域に合せて、プロセス間で粒子情報が交換される。これはオブジェクト ptcl のメソッド exchangeParticle が呼出されて実行される。全プロセスが担当する領域の情報が必要なので、オブジェクト dinfo が引数となる。
2. 相互作用を計算する。これはオブジェクト tree のメソッド calcForceAllAndWriteBack が呼出されて実行される。この中で相互作用を計算する関数を呼出すため、第１、２引数に j 粒子から i 粒子への作用と超粒子から i 粒子への作用を計算する関数 CalcGrav がある。j 粒子と超粒子の指定はテンプレート引数によってなされ、それぞれ Nbody と SPJMonopole となっている。SPJMonopole は FDPS が提供する超粒子のクラスである。また、相互作用の計算には、粒子情報と領域情報も必要なため、第３引数にオブジェクト ptcl、第４引数にオブジェクト dinfo がある。

ソースコード 1: A sample code of N-body simulation

```
#include <particle_simulator.hpp>
using namespace PS;

class Nbody{
public:
    F64    mass, eps;
    F64vec pos, vel, acc;
    F64vec getPos() const {return pos;}
    F64 getCharge() const {return mass;}
    void copyFromFP(const Nbody &in){
        mass = in.mass;
        pos  = in.pos;
        eps  = in.eps;
    }
    void copyFromForce(const Nbody &out) {
        acc = out.acc;
    }
    void clear() {
        acc = 0.0;
    }
    void readAscii(FILE *fp) {
        fscanf(fp,
               "%lf%lf%lf%lf%lf%lf%lf%lf",
               &mass, &eps,
               &pos.x, &pos.y, &pos.z,
               &vel.x, &vel.y, &vel.z);
    }
    void predict(F32 dt) {
        vel += (0.5 * dt) * acc;
        pos += dt * vel;
    }
    void correct(F32 dt) {
        vel += (0.5 * dt) * acc;
    }
};

template <class TPJ>
struct CalcGrav{
    void operator () (const Nbody * ip,
```

```
40                          const S32 ni,
41                          const TPJ * jp,
42                          const S32 nj,
43                          Nbody * force) {
44          for(S32 i=0; i<ni; i++){
45              F64vec xi  = ip[i].pos;
46              F64    ep2 = ip[i].eps
47                  * ip[i].eps;
48              F64vec ai = 0.0;
49              for(S32 j=0; j<nj;j++){
50                  F64vec xj = jp[j].pos;
51                  F64vec dr = xi - xj;
52                  F64 mj  = jp[j].mass;
53                  F64 dr2 = dr * dr + ep2;
54                  F64 dri = 1.0 / sqrt(dr2);
55                  ai -= (dri * dri * dri
56                          * mj) * dr;
57              }
58              force[i].acc += ai;
59          }
60      }
61  };
62
63  template<class Tpsys>
64  void predict(Tpsys &p,
65               const F64 dt) {
66      S32 n = p.getNumberOfParticleLocal();
67      for(S32 i = 0; i < n; i++)
68          p[i].predict(dt);
69  }
70
71  template<class Tpsys>
72  void correct(Tpsys &p,
73               const F64 dt) {
74      S32 n = p.getNumberOfParticleLocal();
75      for(S32 i = 0; i < n; i++)
76          p[i].correct(dt);
77  }
78
79  template <class TDI, class TPS, class TTFF>
```

```
80 void calcGravAllAndWriteBack(TDI &dinfo,
81                              TPS &ptcl,
82                              TTFF &tree) {
83     dinfo.decomposeDomainAll(ptcl);
84     ptcl.exchangeParticle(dinfo);
85     tree.calcForceAllAndWriteBack
86         (CalcGrav<Nbody>(),
87          CalcGrav<SPJMonopole>(),
88          ptcl, dinfo);
89 }
90
91 int main(int argc, char *argv[]) {
92     F32 time  = 0.0;
93     const F32 tend  = 10.0;
94     const F32 dtime = 1.0 / 128.0;
95     Initialize(argc, argv);
96     DomainInfo dinfo;
97     dinfo.initialize();
98     ParticleSystem<Nbody> ptcl;
99     ptcl.initialize();
100    TreeForForceLong<Nbody, Nbody, Nbody>
101        ::Monopole grav;
102    grav.initialize(0);
103    ptcl.readParticleAscii(argv[1]);
104    calcGravAllAndWriteBack(dinfo,
105                            ptcl,
106                            grav);
107    while(time < tend) {
108        predict(ptcl, dtime);
109        calcGravAllAndWriteBack(dinfo,
110                                ptcl,
111                                grav);
112        correct(ptcl, dtime);
113        time += dtime;
114    }
115    Finalize();
116    return 0;
117 }
```

3 Implementation

この節ではFDPSの中の実装について述べる。FDPSの中では分散メモリ環境での並列処理にはMPIを用い、共有メモリ環境での並列処理にはOpenMPを用いている。これらを用いるのはこれらのポータビリティが高いためである。この節で具体的に扱うのは、FDPS内で行われているロードバランスの実装と相互作用計算の実装である。以下では、この順に記述する。

3.1 ロードバランス

FDPSでは、計算時間がプロセスの間で均等になるように、ロードバランスを取る。これは、計算領域を分割して各プロセスに割り当て、各プロセスは担当領域内にある粒子の時間発展を追うのに責任を持つ、という形で実現される。

計算領域の分割の仕方は、3-dimensional multi-section decompositionと呼ばれる方法である。この方法では、まず計算領域をx軸に垂直な面でnx個に分割、今度は各領域をy軸に垂直な面でny個に分割、最後に各領域をz軸に垂直な面でnz個に分割するというものである。従って、各プロセスが担当する領域は直方体となる。領域分割した結果は図を参照。nx, ny, nzの決め方には自由度があるが、なるべくnx, ny, nzの値が等しくなるようにする。このようにするのは相互作用計算などをする際に交換する粒子の数を少なくできるからである。

他にも良い分割方法はある。しかし、FDPSの方法が他の方法に比べて実装がシンプルである。また、シンプルであるのにも関わらず、ロードバランスも悪くない。

以下、ロードバランスの実装方法について述べる。ロードバランスは領域分割と粒子交換の２ステップに分かれる。これらはFDPSが提供するDomainInfoクラスのメソッドdecomposeDomainAllとParticleSystemクラスのメソッドexchangeParticleに対応する。

まず領域分割の実装方法について述べる。FDPSではサンプリング法という方法を採用している。この方法では、各プロセスから粒子をサンプルし、それらの粒子の空間分布から領域の分割の仕方を決定するものである。全粒子の空間分布から決定するよりもランダムノイズは入りやすいが、扱う粒子数が小さいために計算コストを小さくできる。FDPSでは、ランダムノイズを抑えるために、現ステップのサンプル粒子から決めた領域情報から領域分割をするのではなく、１ステップ前の領域情報も使って、現ステップの領域情報を決定する。

以下、領域分割の手順を記述する。

1. 各プロセスが自分の担当粒子からランダムに粒子をサンプルする。全プロセスでサンプルする粒子数はプロセス数に比例した数である。各プロセスでサンプルする粒子数は、全プロセスでサンプルする粒子数にある割合をかけたもので決まる。この割合を決定するのはユーザーであるが、適切なロードバランスを取るには、各プロセスで前のステップにかかった計算時間を前のステップでかかった全プロセスの計算時間の和で割ったものなどがよい。

2. 各プロセスがそれぞれのサンプル粒子の数をMPI_Allgatherを用いてランク0のプロセスに集め、その後、各プロセスがそれぞれのサンプル粒子の位置情報をMPI_Allgathervを用いてランク0のプロセスに集める。

3. 以下３つのサブステップで暫定的な領域分割を行う。ここでは 1 プロセスだけがソートを行うが、あまり計算時間はかからない。ソートするのはサンプル粒子なので、数が少ないからである。

   (a) ランク 0 のプロセスで、サンプル粒子をそれらの x 座標でソートし、サンプル粒子が x 軸方向に nx 等分されるように x 軸に垂直な面で空間を分割する。

   (b) ランク 0 のプロセスで、上で分割した各領域のサンプル粒子を y 座標でソートし、サンプル粒子が y 軸方向に ny 等分されるように y 軸に垂直な面で空間を分割する。

   (c) ランク 0 のプロセスで、上で分割した各領域のサンプル粒子を z 座標でソートし、サンプル粒子が z 軸方向に nz 等分されるように z 軸に垂直な面で空間を分割する。

4. 上で求めた暫定的な領域情報と現在の領域情報を用いて、指数移動平均を行い、領域情報を決定する。これは領域分割に全粒子の空間情報ではなく、サンプルした粒子の空間情報を決めることによるランダムノイズを小さくするために行う。

5. MPI_Bcast を用いて、ランク 0 のプロセスが、分割した領域の情報を他のすべてのプロセスに放送する。

次に粒子交換の実装方法について述べる。これは以下のような手順で実行される。

1. 各プロセスが、自分の担当粒子のうち自分の担当領域からはみでている粒子を探す。ここでは、いきなり担当粒子すべての行先を探すということはしない。前のタイムステップでロードバランスが取られていれば、自分の担当領域からはみでている粒子はあまり多くないし、行先を探すコストは大きいからである。

2. 各プロセスが、自分の担当領域からはみでている粒子すべての送り先のプロセスを探す。ここでは、各粒子の位置と他のすべてのプロセスの担当領域を比較する、というようなことはしない。それよりも、まず x 軸方向で一致するプロセス群を探し、その後 y 軸方向で一致するプロセス群を探し、最後に z 軸方向で一致するプロセスを探す。前者の計算コストは $\mathcal{O}(p)$ であるが、後者の計算コストは $\mathcal{O}(3p^{1/3})$ である。プロセス数が 10 を越えれば、確実に後者の計算コストの方が小さい。

3. MPI_Alltoall を呼び、送りつける粒子の数を交換し、その後 MPI_Alltoallv を呼び、粒子情報を交換する。

3.2 相互作用計算

相互作用の計算は、TreeForForce クラスのメソッド calcForceAllAndWriteBack によって実行される。この中では、各プロセスが担当する粒子の相互作用の計算に必要な粒子の情報を他のプロセスから集め、ユーザーが定義した相互作用を計算する関数を呼び出し、相互作用を計算する。以下では、この実装について記述する。

このメソッドは以下の３ステップを実行する。

1. 各プロセスが全担当粒子の相互作用の計算に必要な粒子の情報を他のプロセスから集める。

2. 各プロセスが担当粒子それぞれの相互作用の計算に必要な粒子のリストを作る。

3. 相互作用の計算をする。

このうちの最後のステップはユーザーの定義する関数の中で行われるため、実装はユーザーに任される。従って、以下では最初と 2 番目のステップの実装を記述する。

まず、各プロセスが担当粒子の相互作用計算に必要な粒子を他のプロセスから集める方法を述べる。これは、基本的には、octree structure を用いた BH ツリーアルゴリズムや近傍粒子探査法の並列版である。BH ツリーアルゴリズムは力が長距離力の場合に、近傍粒子探査法は力が短距離力の場合に、用いられる。厳密にいえば、それぞれの方法には若干の違いがあるが、ここではその違いには深く立ち入らずに記述する。

以下、その手順を記述する。

1. i プロセスで担当粒子すべてからなる octree 構造を構築する。この octree 構造をローカルツリーと呼ぶ。

2. i プロセスがローカルツリーの各ツリーセルのモーメントを計算する。このモーメントは長距離力ならばこのツリーセルに含まれる粒子すべての多重極モーメントであり、短距離力ならばこのツリーセルに含まれる粒子の探査半径の最大値である。

3. i プロセスが自分の担当粒子にとって必要な j プロセスの担当粒子を集める。これは次のようなステップで行われる。

   (a) i プロセスが、自分のローカルツリーを使い、自分の担当粒子のうち j プロセスの担当粒子を必要とする粒子を探し、j プロセスに自分の担当粒子を送る。また j プロセスから、j プロセスの担当粒子のうち i プロセスの担当粒子を必要とする粒子を受け取る。このステップは力の影響範囲が、i 粒子固有の探査半径で決まっていない場合にはスキップできる。この場合、i プロセスと j プロセスの位置関係だけで、i プロセスの担当粒子にとって必要な j プロセスの担当粒子が決まるからである。

   (b) i プロセスが、自分のローカルツリーを使って、j プロセスにとって必要な自分の担当粒子を探す。

   (c) i プロセスが j プロセスにとって必要な自分の担当粒子を送り、j プロセスからは i プロセスにとって必要な j プロセスの担当粒子を受け取る。

4. i プロセスは自分の担当粒子すべてと他のプロセスから受け取ったすべての粒子からなる octree 構造を構築する。この octree 構造をグローバルツリーと呼ぶ。

5. i プロセスがグローバルツリーの各ツリーセルのモーメントを計算する。

次に、各プロセスが担当粒子の相互作用の計算に必要な粒子のリストを作成すについて述べる。このリストは複数の i 粒子に対する共通のリストである。このようなリストを作るのは、ユーザーが相互作用関数を作る際、キャッシュヒット率の向上を意識する必要がなく、また SIMD 化もしやすいからである。

以下、リスト作成の手順を述べる。

1. 各プロセスの各スレッドが、作用する粒子のリストを共有する i 粒子のグループを作る。この i 粒子のグループは、粒子数 $n_{\rm group}$ を上限とするツリーセルに属するすべての粒子から構成される。

2. 各プロセスの各スレッドが、グローバルツリーを使って、i 粒子のグループに作用する粒子のリストを作る。この粒子リストは連続するメモリに格納される。相互作用計算の際のキャッシュヒット率を高くするためである。

各リストが各スレッドで作成されるために、相互作用関数は各スレッドで動作することになっている。従って、ユーザーが相互作用関数を作成する際に OpenMP を意識する必要がない。

4 Performance

組込むべき言葉

- 使用した計算機のピーク性能、アプリのピーク性能、アプリの実行性能
- N-body (スパイラル): 岩澤担当
  - ベンチマーク (京、x86)
  - エネルギーエラーがあったほうが後々役立つかも?
  - きれいな絵
- std SPH (Giant Impact): 細野担当
  - ベンチマーク (京、x86)
  - きれいな絵

4.1 Gravitational N-body simulation

4.2 SPH simulation

$$\frac{dm}{dt} = 0 \tag{8}$$

$$\frac{d\boldsymbol{r}}{dt} = \boldsymbol{v} \tag{9}$$

$$\frac{d\boldsymbol{v}}{dt} = -\frac{\nabla p}{\rho} + \boldsymbol{g} \tag{10}$$

5 Discussion and Conclusion

6 気になる文

- C++プログラムから呼び出せる関数群からなるライブラリ
- メッセージパッシングを用いたプログラムを設計、開発するのが非常に難しい
- collective procedure
- force calculation: the same way as in the case of the uniprocessor code
- Time integration: sec. 3.4 Makino (2004)