Vol.2015-HPC-150 No.9
2015/8/4

FDPS(Framework for Developing Particle Simulator): 大規模分散メモリー環境下での粒子系シミュレーション用フレームワークの開発

岩澤 全規[1,a)] 谷川 衝[1,b)] 細野 七月[1,c)] 似鳥 啓吾[1,d)] 村主 崇行[1,e)] 牧野 淳一郎[1,2,f)]

概要：
粒子法を用いたシミュレーションは科学や工学の幅広い分野で広く使われており，使用されるアルゴリズムも似ている．しかし，今までのソフトウェアは各分野で独立に開発されており，分野間で共用する事は難しかった．一方,「京」の様な大規模並列型スパコンで効率よく動作する粒子法プログラムの開発は容易ではなく，多くの研究者はソフトウェアの開発に多大な時間と労力を割いている．そこで，本研究では,「京」の様な大規模並列型スパコンで効率よく動作する粒子法シミュレーションプログラムをユーザーが容易に開発できるフレームワーク (FDPS: Framework for DevelopingParticle Simulator) を開発した．粒子法の大規模並列プログラムが複雑になるのは，計算領域分割やその領域に合わせた粒子の再配分，また効率的な相互作用のために必要な粒子のツリー構造での管理等が必要なためである．FDPS はこの部分を担当する．そのため，ユーザーは並列化を意識することなくプログラムの開発を行うことができる．我々は，実際に FDPS を使ったいくつかのプログラムの開発も行い,「京」などのスパコンで実行した．その結果，プログラムが数百行で書け，また非常に高い実行効率が出る事を確認した．

キーワード：粒子法，HPC，MPI+OpenMP 並列

1. はじめに

粒子法とは研究対象となる系を相互作用する多数の粒子によって表現し，個々の粒子の発展方程式を解くことで系の進化をシミュレートする方法である．粒子法では対象の系の運動に合わせて，自動的に粒子が移動し，系を自然な形で表現してくれる．そのため，物体の衝突や破壊等，形状が大きく変わる系のシミュレーションや密度コントラストが高い系のシミュレーション等に適しており，例えば天文学や生命科学，気象学，防災やものづくりに至るまで幅広い分野で使われている．

[1] 理化学研究所 計算科学研究機構
RIKEN Advanced Institute for Computational Science
[2] 東京工業大学
Tokyo Institute of Technology
a) masaki.iwasawa@riken.jp
b) ataru.tanikawa@riken.jp
c) natsuki.hosono@riken.jp
d) keigo@riken.jp
e) takayuki.muranushi@riken.jp
f) jmakino@riken.jp

近年，シミュレーションの解像度や精度を上げるために，大規模並列計算機を使われ始めているが，このような並列計算機上で効率よく動作する，粒子法シミュレーションプログラムを開発する事は容易ではない．効率の良いアプリケーション・プログラムを開発するためには，プログラマは各プロセスのロードバランスが取れるように計算領域の分割を行う必要があり，シミュレーション対象の系が動的に変化していく場合，計算領域を動的に決定する必要がある．

また，計算領域が分割されているので，各プロセスが粒子への相互作用を計算するためには他のプロセスから相互作用計算に必要となる粒子の情報を受け取る必要があり，プログラマはこの通信量が最小になる様にソフトウェア開発を行わなければならない [11], [14], [19], [21] ．更に，効率的な相互作用の計算のためにはキャッシュメモリや SIMD ユニットの効率的な利用も考慮する必要がある [16], [22], [23]．近年では，相互作用計算を加速するために GPGPU やその他の加速機も使われ始めており、対応が

ⓒ 2015 Information Processing Society of Japan

必要となる場合もある [3], [8], [9]．

効率的に動作するプログラムを開発するためには，上記の事柄を考慮する必要があり，コード開発だけで数年かかってしまうような大規模なプロジェクトになってしまう事がある．コード開発に膨大な時間がかかる事は計算科学全体の進歩を遅らせてしまう事になりかねない．

そこで，我々は上記の問題を解決するために FDPS(Framework for Developing Particle Simulator) の開発を行った．FDPS の目的はユーザーが大規模並列計算機で効率的に動作するプログラムを容易に開発できるようなソフトウェアフレームワークを提供する事である．FDPS の基本的な考え方は，粒子法シミュレーションプログラムを開発するときに困難となる部分 (例えば，領域分割や粒子交換，効率的な相互作用計算のための粒子の木構造管理等) を FDPS 側が担当することである．このため，ユーザーはこの困難な部分を意識することなく，プログラムできる。

ユーザーは相互作用関数と粒子のデータ構造を定義し FDPS に与える事で任意の 2 粒子間相互作用を扱うことができる．このため，ユーザーは FDPS を用いて多くのアプリケーションプログラム (例えば，重力 N 体コード，SPH や MPS 等の流体コード，分子動力学コード，個別要素法等) を開発することができる．多粒子間相互作用に関してはユーザープログラムで対応することができる．

FDPS と同様のコンセプトを持つソフトウェアは幾つか存在するが，それらは相互作用が $1/r$ ポテンシャルのみや [25]，密度変化の大きな系では扱えないなど [13]，適用範囲が限定的であり，FDPS の様な汎用性は備えていない．また，アプリケーション・プログラムの性能についても非常に高い性能が出る事を確認した．この汎用性の高さと性能の高さの両立が FDPS の独自性である．

本論文では，FDPS の概要を 2 節で述べ、3 節ではユーザーコードの実装例を示し，4 節では FDPS 内部の実装について，5 節では我々が実際に開発した 2 つのアプリケーションプログラムとその性能について議論し，6 節で FDPS から利用できる有用なモジュール群に触れ，7 節で本論文のまとめを行う．

2. FDPS 概要

2.1 FDPS デザインコンセプト

この節では FDPS のデザインコンセプトについて述べる．

FDPS の目的はユーザーが大規模並列計算機で効率的に動作する粒子法プログラムを容易に開発できるフレームワークを提供することである．FDPS は汎用性と性能を両立させるために C++のテンプレートライブラリとして定義されている．FDPS 内では粒子のデータ構造や相互作用関数はテンプレート型として扱われているため，ユーザーは FDPS の提供するライブラリを使い任意の粒子データ構造や相互作用関数を持つ粒子シミュレーションを行う事が可能となっている．また，FDPS のライブラリは MPI 及び OpenMP による並列化に対応しており，ユーザーは並列化を意識する事なく大規模並列計算機で効率的に動くプログラムの開発を行うことができる．

FDPS は以下の用な形の常微分方程式を扱えるようにデザインされている．

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

ここで，N は系の粒子数，$\vec{u}_i$ は i 番目の粒子の物理量ベクトルであり，$\vec{f}$ は粒子 j からの寄与による粒子 i の物理量の時間微分を求めるための関数である．$\vec{g}$ は，$\vec{f}$ で計算された寄与の和を粒子 i の物理量の時間微分に変換するための関数である．以後，相互作用を受ける粒子を i 粒子，相互作用を与える粒子を j 粒子と呼ぶ．例えば重力 N 体シミュレーションの場合では，$\vec{f}$ はニュートン重力であり，$\vec{u}_j$ は粒子の質量，位置ベクトル，速度ベクトルからなる．$\vec{g}$ は例えば外場などが存在した場合にその寄与を与える関数となる．

式 1 から分かるように FDPS は二粒子間相互作用のみに対応している．分子動力学などでは多体間相互作用も考慮する必要があるが，そのような相互作用はユーザープログラム側で評価することが可能である．

2.2 FDPS を用いた粒子シミュレーションの流れ

FDPS を用いた粒子系シミュレーションは以下の様な流れとなる。

(1) 計算領域全体を分割し，各 MPI プロセスが分割された計算領域を担当する．
(2) 上で計算された領域に沿って，MPI プロセス間で粒子の交換を行う．
(3) 各プロセスが担当する粒子への相互作用の計算を行う．この際，粒子への相互作用を計算するために必要な情報を他の MPI プロセスから受信する．
(4) 粒子の情報を相互作用の結果を使って更新する．この際プロセス間の通信は必要ない．

上の手順 1，2，3 にはプロセス間の通信が必要な部分であり，FDPS はこの部分を計算するためのライブラリを提供する．FDPS では手順 1，2，3 を実行するためのクラスをそれぞれ用意しておりそれらは `DomainInfo` クラス，`ParticleSystem` クラス，`TreeForForce` クラスと呼ばれる．ユーザーはこれらのクラスのオブジェクトを作り，オブジェクトのメソッドを呼び出す事でそれぞれの機能を使うことができる．

手順 1 に関して，ユーザーは `DomainInfo` クラスを使うことで計算領域の分割を行う事ができる．このクラスは計算領域のデータ等を保持するクラスである．ユーザーはこのク

ⓒ 2015 Information Processing Society of Japan

ラスのメソッドである DomainInfo::decomposeDomain() を全てのプロセスで呼び出すことで FDPS に領域分割を行わせる．デフォルトでは各プロセスの保持する粒子数が同じ数になる様に分割するが，オプションを与えることで重みを付けて領域分割を行う事もできる．

手順 2 に関して，ユーザーは ParticleSystem クラスを使うことで粒子の交換を行う事ができる．このクラスはユーザーが定義した粒子のデータ構造を保持するクラスでありプロセス間の粒子の交換等はこのクラスを介して行う．ユーザーはこのクラスのメソッドである ParticleSystem::exchangeParticle() を全てのプロセスで呼び出すことで FDPS に各プロセスの計算領域に合わせた粒子の交換を行わせる．ユーザーはこれらのクラスのオブジェクトを複数作ることで，様々な種類の粒子が存在するシミュレーションを行うことができる．

手順 3 に関して，ユーザーは TreeForForce クラスを使うことで相互作用の計算を行うことができる．このクラスは ParticleSystem クラスから粒子のデータ構造を読み取りこれらの粒子を八分木構造を使って管理する．これは相互作用計算を効率的に行うためである．FDPS では粒子間相互作用を長距離力型と短距離力型の 2 つの形に分類している．長距離力型とは重力やクーロン力等，遠くの粒子からの相互作用も考慮しなければならない形の力である．短距離力型とは，近傍の粒子からのみ力を受け，遠くの粒子からの力は無視出来る型の力であり，例えばレナードジョーンズポテンシャルなどである．また，粒子法に基づいた流体シミュレーションでは，ある場所での物理量を近傍の粒子の重ね合わせで表現するためこれも短距離力として扱う．短距離力の場合は、木構造を用いて近傍粒子の探査を行う．長距離力の場合は遠くの粒子からの相互作用への寄与は一般に小さいので，遠くの粒子からの寄与は近似的にそれらの多重極展開を使って計算する Barnes-Hut tree 法を採用してしている [2].

また，粒子間相互作用の計算において，ある粒子は自分のプロセス以外の粒子とも相互作用する可能性があるため，FDPS では相互作用に寄与する j 粒子を通信して相互作用を計算するのに必要な粒子を全て自分のプロセスに集める方法を採用している．この方法では短距離力の場合は近傍のプロセスと通信すれば良い．長距離力の場合には Barnes-Hut tree 法を採用しているため，遠くの計算領域のプロセスからは全ての粒子をもらってくる必要はなく粒子が作るポテンシャルの多重極展開の係数を通信すれば十分であり，FDPS もこの方法を採用している [14].

ユーザーがこのクラスのメソッド TreeForForce::calcForceAllAndWriteBack() を全てのプロセスで呼び出すと，FDPS は粒子座標から木構造を構築し，相互作用の計算に必要な j 粒子の通信を行い，更に相互作用の計算までを行う．この関数内での木構造構築や相互作用計算は OpenMP にも対応している．

複数種類の相互作用を計算する場合はこのクラスのオブジェクトを複数生成すれば良い．

3. FDPS を用いた重力 N 体シミュレーションコードの実装例

我々は FDPS のユーザーは以下の流れで粒子系シミュレーションコードの開発を行う事を想定している。

(1) 粒子のデータ構造を C++のクラスの形で定義する．

(2) 相互作用関数を C++の関数テンプレートもしくは関数ポインタの形で定義する．相互作用関数は i 粒子と j 粒子の配列を受け取り，i 粒子への相互作用を計算し，その結果を積算するように定義する．

(3) FDPS によって提供されているデータクラスと関数群を用いてユーザープログラムを作成する．

図 1 は重力 N 体シミュレーションの粒子クラスの例である．

```
 1 class Nbody{
 2 public:
 3   F64    mass, eps;
 4   F64vec pos, vel, acc;
 5   F64vec getPos() const {return pos;}
 6   F64 getCharge() const {return mass;}
 7   void copyFromFP(const Nbody &in){
 8     mass = in.mass;
 9     pos  = in.pos;
10     eps  = in.eps;
11   }
12   void copyFromForce(const Nbody &out) {
13     acc = out.acc;
14   }
15   void clear() {
16     acc = 0.0;
17   }
18 };
```

図 1 粒子クラス

粒子データクラスの名前やメンバ変数名等は自由であるが，いくつかのメンバ関数は決まった名前で用意する必要がある（上の例では getPos()，getCharge()，copyFromFP() 等）．これはその名前を使って FDPS がクラス内メンバにアクセスするからである．またメンバ変数の型にある F64 と F64vec は FDPS によって定義された型であり，それぞれ 64bit 浮動小数点と 64bit 浮動小数点を成分とする 3 次元ベクトル型である．ベクトル型には四則演算や内積外積等のメンバ関数を持っており，ユーザーはこれらを用いてユーザープログラムを記述することができる．

図 2 はニュートン重力の場合の相互作用関数の例である．この図から分かるように相互作用関数は複数の i 粒子

ⓒ 2015 Information Processing Society of Japan

```
1  struct CalcGrav{
2    void operator () (const Nbody * ip,
3                      const S32 ni,
4                      const Nbody * jp,
5                      const S32 nj,
6                      Nbody * force) {
7      for(S32 i=0; i<ni; i++){
8        F64vec xi  = ip[i].pos;
9        F64    ep2 = ip[i].eps * ip[i].eps;
10       F64vec ai = 0.0;
11       for(S32 j=0; j<nj;j++){
12         F64vec xj = jp[j].pos;
13         F64vec dr = xi - xj;
14         F64 mj  = jp[j].mass;
15         F64 dr2 = dr * dr + ep2;
16         F64 dri = 1.0 / sqrt(dr2);
17         ai -= (dri * dri * dri * mj) * dr;
18       }
19       force[i].acc += ai;
20     }
21   }
22 };
```

図 2　相互作用関数

への複数の j 粒子からの相互作用の計算をするものでなくてはならない．ここに出てくる S32 型は 32bit 符号付き整数型である．

図 3 はメイン関数の一部である．FDPS の機能を使うためにはまず，`PS::Initialize()` を呼ぶ必要がある．6 行目で `DomainInfo` クラス，8 行目で `ParticleSystem` クラス，10 行目で `TreeForForce` クラスのオブジェクトを生成している．ここでは `ParticleSystem` クラスと `TreeForForce` クラスのテンプレート引数に粒子クラスを与えている．18 行目で担当する計算領域の分割を行い，19 行目で粒子の交換を行っている．20 行目では，メソッド `calcForceAllAndWriteBack` の引数に相互作用関数を与えることで，相互作用の計算を行っている．このプログラムをみて分かる通り，どこにも明示的に MPI の API は呼ばれていない．しかし，このコードは MPI と OpenMP を用いたハイブリッド並列プログラムとして動く．付録に実際にハイブリッド並列で動作するシンプルな重力 N 体シミュレーションのプログラムを載せた．このプログラムは 120 行程で書かれている．

4. FDPS の内部実装

この節では FDPS の提供する関数がどの様に実装されているかを述べる．

4.1 領域分割と粒子交換

FDPS では領域分割に three-dimensional multi-section

```
1  int main(int argc, char *argv[]) {
2    F64 time  = 0.0;
3    const F64 tend  = 10.0;
4    const F64 dtime = 1.0 / 128.0;
5    PS::Initialize(argc, argv);
6    PS::DomainInfo dinfo;
7    dinfo.initialize();
8    PS::ParticleSystem<Nbody> ptcl;
9    ptcl.initialize();
10   PS::TreeForForceLong<Nbody, Nbody,
11       Nbody>::Monopole grav;
12   grav.initialize(0);
13   ptcl.readParticleAscii(argv[1]);
14   while(time < tend) {
15     S32 n = p.getNumberOfParticleLocal();
16     for(S32 i = 0; i < n; i++)
17       ptcl[i].predict(dttime);
18     dinfo.decomposeDomainAll(ptcl);
19     ptcl.exchangeParticle(dinfo);
20     tree.calcForceAllAndWriteBack
21         (CalcGrav<Nbody>(),
22          CalcGrav<SPJMonopole>(),
23          ptcl, dinfo);
24     n = p.getNumberOfParticleLocal();
25     for(S32 i = 0; i < n; i++)
26       ptcl[i].correct(dtime);
27     time += dtime;
28   }
29   PS::Finalize();
30   return 0;
31 }
```

図 3　メイン関数

法 (MS 法)[14] を採用している．この方法では，最初に計算領域を x 軸方向に沿って nx 個の領域に分割する．次に先ほど分割された各領域を y 軸方向に沿って ny 個の領域に分割する．z 軸方向についても同様に先ほど分割した領域を z 軸方向に沿って分割する．領域分割の方法は他に ORB[5] やモートン順序 [24] によるものが広く使われている．MS 法は ORB と違いプロセッサ数が 2 のべき乗である事を要求せず，任意のプロセッサ数で動作する．また，モートン順序による領域分割では領域に飛びが出来てしまう可能性があるが MS 法ではそのような事は起こらない．

FDPS では各プロセスが担当する計算領域の座標を求める方法としてサンプリング法 [14] を採用している．この方法では計算領域の位置座標を各プロセスからサンプリングした粒子のみによって決定する．プロセス数が大きくなるとサンプル数も増えてしまうため，FDPS ではこの部分も MPI を用いた並列化に対応している．また，サンプリングノイズを低減するために，計算領域の座標に指数移動平均を使うこともできる．

ⓒ 2015 Information Processing Society of Japan

この様にして計算された計算領域にしたがってプロセス間で粒子の交換を行う．これは MPI_Alltoall と MPI_Isend, MPI_Irecv を用いて実装されている．

4.2 相互作用計算

この節では，相互作用の計算の内部実装について述べる．まず，TreeForForce クラスは ParticleSystem クラスから相互作用計算を行うために必要な粒子のデータを受け取り内部に保持する．この粒子データから八分木構造を作り粒子を管理する．八分木構造の作り方は以下の様になっている．

(1) 各粒子のモートン鍵を計算する．
(2) 基数ソートを使ってモートン鍵の順番に粒子を並べる．
(3) 木構造を階層ごとに上から作っていく．モートン鍵の上位 3bit をみて各粒子の対応するツリーセルを見つける．この際粒子はソートされているためバイナリーサーチを使うことができる．

FDPS ではこの全ての手順を OpenMP 化している．

次にこの木構造を利用して相互作用を計算するために必要な粒子の交換を行う．短距離力の場合は近傍の計算領域を担当するプロセスとだけ通信を行えば良い．これは MPI_Isend と MPI_Irecv を使って実装されている．長距離力の場合は各プロセッサから送られてくるメッセージサイズは比較的小さいが全体全の通信が必要になる．これは MPI_Alltoall(v) により実装されている．しかし,「京」コンピュータの場合は短いメッセージ長の MPI_Alltoall(v) はあまり最適化がなされていないため，メッセージの数を減らす様に同じ中継ポイントにいくメッセージを統合する方式をとっている．

送られてきた粒子情報から先ほどと同じ方法で再び木構造を構築する．この木構造を用いて j 粒子の探索を行う．ここでは，局所的にまとめた i 粒子の集団に対して j 粒子の探索を行い、ユーザーの定義した相互作用関数にしたがって相互作用を計算する．FDPS では OpenMP を用いて j 粒子探査と相互作用の計算をスレッド並列化している．

5. 性能

この節では FDPS を使って開発した 2 つのアプリケーションプログラムの性能について述べる．この節で行ったシミュレーションは全て「京」を使って行った．このシミュレーションでの並列化方法はノード内については OpenMP を使い，ノード間については MPI を用いたハイブリッド並列である．

5.1 アプリケーション・プログラムの性能

5.1.1 重力 N 体シミュレーション

本節では FDPS を用いた重力 N 体コードの性能を述べる．

ここで使ったプログラムは前の節で述べ，付録に掲載されているプログラムと本質的には同じであるが，1 つ大きな違いがある．ここでは，前の節の相互作用関数と違い高度に最適化された相互作用関数を使っている．

本シミュレーションでは，初期モデルとして，星団や銀河のシミュレーションで広く使われている力学平衡モデルの一つであるプラマーモデル [17] を用いた．重力の計算にはツリーの見込み角 $\theta = 0.4$ として，四重極子までの展開を使った．

図 4 は 1 プロセス当たりの平均の粒子数を約 210 万体とした場合の速度 (上図) と 1 ステップ当たりのウォールクロックタイム (下図) をプロセス数の関数としてプロットしたものである．非常に良くスケールしている事が分かる．実行効率も理論ピーク性能の 50%を達成している．

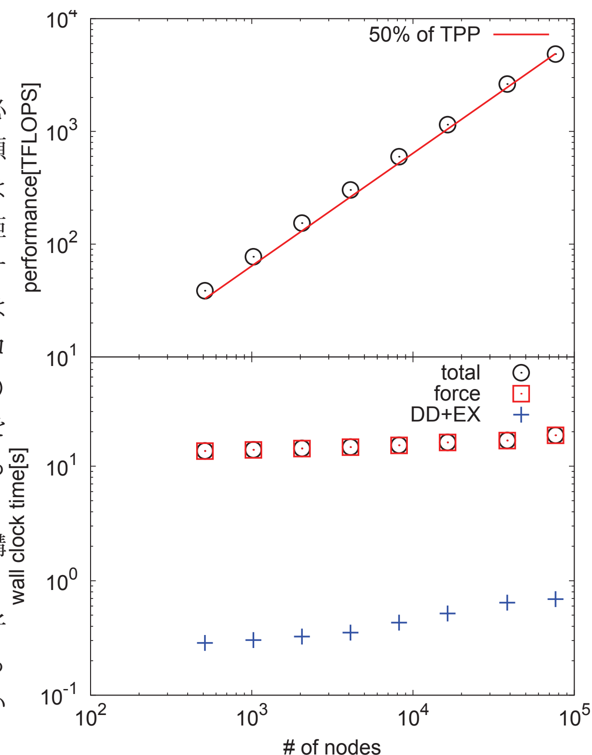

図 4 重力 N 体シミュレーションでの性能．上:黒丸は本アプリケーションの速度であり，赤線は理論ピーク性能の 50%を表している．下:黒丸は計算時間の合計，赤四角は重力の計算にかかった時間，青十字は領域分割と粒子交換にかかった時間の合計を表している．

5.1.2 SPH シミュレーション

本節では FDPS を用いた SPH 法での性能を述べる．

我々は SPH 法のシミュレーションとして，月形成の有力なシナリオである巨大衝突仮説 [6], [10] のシミュレーションを行った．巨大衝突仮説とは 50 億年程前に火星サイズの天体が原始地球に衝突しその際に撒き散らされた破片が集積し月になったとする説である．多くの研究者によって巨大衝突シミュレーションが行われている [1], [4], [7]．

重力の計算にはツリーの見込み角 $\theta = 0.5$ として，単極子までの展開を使った．流体部分の計算は標準 SPH 法を使った [15], [18], [20].

図 5 は 1 プロセス当たりの平均の粒子数を約 31 万体とした場合の速度 (上図) と 1 ステップ当たりのウォールクロックタイム (下図) をプロセス数の関数としてプロットしたものである．本シミュレーションも良くスケールしている事が分かる．実行効率も理論ピーク性能の 40%程度を達成している．図から重力の計算に比べて，流体の計算にかかる時間が大きい事が分かる．これは流体の 1 相互作用当たりの計算量は重力相互作用のそれに比べて非常に大きいためである．

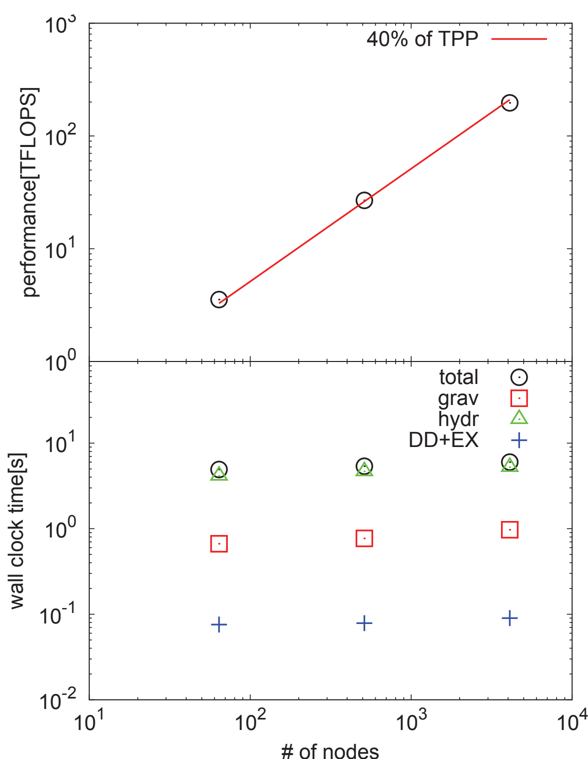

図 5 巨大衝突シミュレーションの性能．上:黒丸は本アプリケーションの速度であり，赤線は理論ピーク性能の 40%を表している．下:黒丸は計算時間の合計，赤四角は重力の計算にかかった時間，緑三角は流体の密度と面積力を求めるのにかかった時間，青十字は領域分割と粒子交換にかかった時間の合計を表している．

6. 外部モジュール

FDPS では有用ないくつかの外部モジュールも提供している．その一つは Particle-Mesh 法により力を計算するモジュールである．これは GreeM[11], [12] で使われている Particle-Mesh 法のモジュールを FDPS 用にラップしたものである．これにより，ユーザーは PPPM 法や PMTree 法 [26] のプログラムを容易に開発できる．また，他の外部モジュールとしては高度に最適化されたニュートン重力の計算を実行するライブラリが用意されている [16], [22], [23].

外部モジュールは今後も開発していく予定である．

7. まとめ

我々は大規模並列粒子法シミュレーションプログラム開発を容易にするフレームワーク FDPS の開発を行った．FDPS を使う事でユーザーは並列化を意識することなく，容易にソフトウェア開発をできるようになった．また，実際に FDPS を用いた 2 つのアプリケーション・プログラム（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）の開発を行い，性能，スケーラビリティーを測定した．結果，両アプリケーション共に，非常に高い実行性能で動作することが分かった．

謝辞 外部モジュールとして PMTree コードを提供して下さった石山智明氏，ユーザーの立場から多大な助言をして下さった丸山豊氏，本プロジェクト全体の管理をして下さった坪内美幸氏に感謝します．

参考文献

[1] Asphaug, E. and Reufer, A.: Mercury and other iron-rich planetary bodies as relics of inefficient accretion, *Nature Geoscience*, Vol. 7, pp. 564–568 (online), DOI: 10.1038/ngeo2189 (2014).

[2] Barnes, J. and Hut, P.: A hierarchical O(N log N) force-calculation algorithm, *Nature*, Vol. 324, pp. 446–449 (1986).

[3] Bédorf, J., Gaburov, E., Fujii, M. S., Nitadori, K., Ishiyama, T. and Zwart, S. P.: 24.77 Pflops on a Gravitational Tree-code to Simulate the Milky Way Galaxy with 18600 GPUs, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, Piscataway, NJ, USA, IEEE Press, pp. 54–65 (online), DOI: 10.1109/SC.2014.10 (2014).

[4] Benz, W., Slattery, W. L. and Cameron, A. G. W.: The origin of the moon and the single-impact hypothesis. I, *Icarus*, Vol. 66, pp. 515–535 (online), DOI: 10.1016/0019-1035(86)90088-6 (1986).

[5] Blackston, D. and Suel, T.: Highly Portable and Efficient Implementations of Parallel Adaptive N-body Methods, *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, SC '97, New York, NY, USA, ACM, pp. 1–20 (online), DOI: 10.1145/509593.509597 (1997).

[6] Cameron, A. G. W. and Ward, W. R.: The Origin of the Moon, *Lunar and Planetary Science Conference*, Lunar and Planetary Science Conference, Vol. 7, p. 120 (1976).

[7] Canup, R. M., Barr, A. C. and Crawford, D. A.: Lunar-forming impacts: High-resolution SPH and AMR-CTH simulations, *Icarus*, Vol. 222, pp. 200–219 (online), DOI: 10.1016/j.icarus.2012.10.011 (2013).

[8] Hamada, T., Nitadori, K., Benkrid, K., Ohno, Y., Morimoto, G., Masada, T., Shibata, Y., Oguri, K. and Taiji, M.: A novel multiple-walk parallel algorithm for the Barnes-Hut treecode on GPUs-towards cost effective, high performance N-body simulation, *Computer Science-Research and Development*, Vol. 24, No. 1, pp. 21–31 (2009).

[9] Hamada, T., Narumi, T., Yokota, R., Yasuoka, K., Ni-

© 2015 Information Processing Society of Japan

tadori, K. and Taiji, M.: 42 TFlops Hierarchical N-body Simulations on GPUs with Applications in Both Astrophysics and Turbulence, *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, ACM, pp. 62:1–62:12 (online), DOI: 10.1145/1654059.1654123 (2009).
[10] Hartmann, W. K. and Davis, D. R.: Satellite-sized planetesimals and lunar origin, *Icarus*, Vol. 24, pp. 504–514 (online), DOI: 10.1016/0019-1035(75)90070-6 (1975).
[11] Ishiyama, T., Fukushige, T. and makino, J.: GreeM: Massively Parallel TreePM Code for Large Cosmological N -body Simulations, *Publications of the Astronomical Society of Japan*, Vol. 61, No. 6, pp. 1319–1330 (2009).
[12] Ishiyama, T., Nitadori, K. and makino, J.: 4.45 Pflops Astrophysical N-Body Simulation on K computer – The Gravitational Trillion-Body Problem, *SC '12 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Salt Lake City, UT, USA, Nov 11 - 15, 2012* (2012).
[13] Leisheng, L., Chaowei, W., Zhitao, M., Zhigang, H. and Rong, T.: petaPar: A Scalable and Fault Tolerant Petascale Free Mesh Simulation System, *Journal of Computer Research and Development*, Vol. 52, No. 4, p. 823 (online), DOI: 10.7544/issn1000-1239.2015.20131332 (2015).
[14] Makino, J.: A Fast Parallel Treecode with GRAPE, *Publications of Astronomical Society of Japan*, Vol. 56, pp. 521–531 (2004).
[15] Monaghan, J. J.: SPH and Riemann Solvers, *Journal of Computational Physics*, Vol. 136, pp. 298–307 (online), DOI: 10.1006/jcph.1997.5732 (1997).
[16] Nitadori, K., Makino, J. and Hut, P.: Performance tuning of N-body codes on modern microprocessors: I. Direct integration with a hermite scheme on x86_64 architecture, *New Astronomy*, Vol. 12, pp. 169–181 (2006).
[17] Plummer, H. C.: On the problem of distribution in globular star clusters, *Monthly Notices of the Royal Astronomical Society*, Vol. 71, pp. 460–470 (1911).
[18] Rosswog, S.: Astrophysical smooth particle hydrodynamics, *New Astronomy Reviews*, Vol. 53, pp. 78–104 (online), DOI: 10.1016/j.newar.2009.08.007 (2009).
[19] Salmon, J. K. and Warren, M. S.: Skeletons from the treecode closet, *Journal of Computational Physics*, Vol. 111, pp. 136–155 (online), DOI: 10.1006/jcph.1994.1050 (1994).
[20] Springel, V.: Smoothed Particle Hydrodynamics in Astrophysics, *Annual Review of Astronomy and Astrophysics*, Vol. 48, pp. 391–430 (online), DOI: 10.1146/annurev-astro-081309-130914 (2010).
[21] Springel, V.: The cosmological simulation code GADGET-2, Vol. 364, No. 4, pp. 1105–1134 (2005).
[22] Tanikawa, A., Yoshikawa, K., Nitadori, K. and Okamoto, T.: Phantom-GRAPE: Numerical software library to accelerate collisionless N-body simulation with SIMD instruction set on x86 architecture, *New Astronomy*, Vol. 19, pp. 74–88 (2013).
[23] Tanikawa, A., Yoshikawa, K., Okamoto, T. and Nitadori, K.: N-body simulation for self-gravitating collisional systems with a new SIMD instruction set extension to the x86 architecture, Advanced Vector eXtensions, *New Astronomy*, Vol. 17, pp. 82–92 (2012).
[24] Warren, M. S. and Salmon, J. K.: A Parallel Hashed Oct-Tree N-body Algorithm, *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, New York, NY, USA, ACM, pp. 12–21 (online), DOI: 10.1145/169627.169640 (1993).
[25] Warren, M. S. and Salmon, J. K.: A portable parallel particle program, *Computer Physics Communications*, Vol. 87, pp. 266–290 (1995).
[26] Yoshikawa, K. and Fukushige, T.: PPPM and TreePM Methods on GRAPE Systems for Cosmological N-Body Simulations, *Publications of Astronomical Society of Japan*, Vol. 57, pp. 849–860 (online), DOI: 10.1093/pasj/57.6.849 (2005).

付　　録

A.1 重力 N 体シミュレーションサンプルコード

```
1  #include <particle_simulator.hpp>
2  using namespace PS;
3  class Nbody{
4  public:
5    F64    mass, eps;
6    F64vec pos, vel, acc;
7    F64vec getPos() const {return pos;}
8    F64 getCharge() const {return mass;}
9    void copyFromFP(const Nbody &in){
10     mass = in.mass;
11     pos  = in.pos;
12     eps  = in.eps;
13   }
14   void copyFromForce(const Nbody &out) {
15     acc = out.acc;
16   }
17   void clear() {
18     acc = 0.0;
19   }
20   void readAscii(FILE *fp) {
21     fscanf(fp,
22            "%lf%lf%lf%lf%lf%lf%lf%lf",
23            &mass, &eps,
24            &pos.x, &pos.y, &pos.z,
25            &vel.x, &vel.y, &vel.z);
26   }
27   void predict(F64 dt) {
28     vel += (0.5 * dt) * acc;
29     pos += dt * vel;
30   }
31   void correct(F64 dt) {
32     vel += (0.5 * dt) * acc;
33   }
34 };
35
36 template <class TPJ>
```

ⓒ 2015 Information Processing Society of Japan

```
37 struct CalcGrav{
38   void operator () (const Nbody * ip,
39                     const S32 ni,
40                     const TPJ * jp,
41                     const S32 nj,
42                     Nbody * force) {
43     for(S32 i=0; i<ni; i++){
44       F64vec xi  = ip[i].pos;
45       F64    ep2 = ip[i].eps
46           * ip[i].eps;
47       F64vec ai = 0.0;
48       for(S32 j=0; j<nj;j++){
49         F64vec xj = jp[j].pos;
50         F64vec dr = xi - xj;
51         F64 mj  = jp[j].mass;
52         F64 dr2 = dr * dr + ep2;
53         F64 dri = 1.0 / sqrt(dr2);
54         ai -= (dri * dri * dri
55               * mj) * dr;
56       }
57     force[i].acc += ai;
58     }
59   }
60 };
61
62 template<class Tpsys>
63 void predict(Tpsys &p,
64              const F64 dt) {
65   S32 n = p.getNumberOfParticleLocal();
66   for(S32 i = 0; i < n; i++)
67     p[i].predict(dt);
68 }
69
70 template<class Tpsys>
71 void correct(Tpsys &p,
72              const F64 dt) {
73   S32 n = p.getNumberOfParticleLocal();
74   for(S32 i = 0; i < n; i++)
75     p[i].correct(dt);
76 }
77
78 template<class TDI, class TPS,
79          class TTF>
80 void calcGravAllAndWriteBack(TDI &dinfo,
81                              TPS &ptcl,
82                              TTF &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   PS::Initialize(argc, argv);
96   PS::DomainInfo dinfo;
97   dinfo.initialize();
98   PS::ParticleSystem<Nbody> ptcl;
99   ptcl.initialize();
100  PS::TreeForForceLong<Nbody, Nbody,
101      Nbody>::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  PS::Finalize();
116  return 0;
117 }
```

ⓒ 2015 Information Processing Society of Japan