Vol.2018-HPC-167 No.22
2018/12/18

粒子法シミュレーションコード開発のためのフレームワーク (FDPS) の開発

岩澤 全規[1,a)] 行方 大輔[1,b)] 坂本 亮[3,c)] 中村孝史[3,d)] 木村耕行[4,e)] 似鳥啓吾[1,f)]
野村昴太郎[1,g)] 坪内美幸[1,h)] 牧野 淳一郎[2,1,i)]

概要：
粒子法は粗密の大きい系や空隙のある系のシミュレーションに強く，科学や工学の幅広い分野で使われている．しかし、「京」の様な大規模並列計算機で効率よく動作する並列シミュレーションコードを開発することは容易ではなく，多くの研究者がコードの開発に膨大な時間を割いているのが現状である．しかし、粒子法シミュレーションコードの効率の良い並列化のアルゴリズムは，シミュレーション対象によらず似ている．そこで，我々は粒子法コードの開発を容易にするためのフレームワーク FDPS(Framework for Developing Particle Simulators) の開発を行った．我々は C++のテンプレート機能を用いて FDPS を開発した．これは，ユーザーが定義する粒子のデータ構造と相互作用を扱えるようにするためである．我々の経験では，FDPS を使うことで様々なアプリケーションが数百行で書ける．また相互作用カーネルを十分最適化すれば開発したアプリケーションは多くのスパコン上で理論ピーク性能の 30-50%程度の実行効率で動作する．

本稿では FDPS の概要および，Ver.1 リリース後に追加された新機能，特にアクセラレータを持つ計算機を効率よく使うためのマルチウォーク法や相互作用リストを再利用する機能，さらに FDPS を C++言語以外から使うための多言語インターフェースについて報告する．

キーワード：HPC，フレームワーク，粒子法

Masaki Iwasawa[1,a)] Daisuke Namekata[1,b)] Ryo Sakamoto[3,c)] Takashi Nakamura[3,d)]
Yasuyuki Kimura[4,e)] Keigo Nitarori[1,f)] Kentaro Nomura[1,g)] Miyuki Tsubouchi[1,h)]
Junichiro Makino[2,1,i)]

[1] 理化学研究所 計算科学研究センター
7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo, 650-0047, Japan
[2] 神戸大学
1-1, Rokkodai-cho, Nada-ku, Kobe, 657-8501, Japan
[3] 株式会社 PEZY Computing
1-11 Ogawa-machi, Chiyoda-ku, Tokyo, 101-0052, Japan
[4] 株式会社 ExaScaler
2-1 Ogawa-machi, Chiyoda-ku, 101-0052 Tokyo, Japan
[a)] masaki.iwasawa@riken.jp
[b)] daisuke.namekata@riken.jp
[c)] sakamoto@pezy.co.jp
[d)] nakamura@pezy.co.jp
[e)] yasuyuki@exascaler.co.jp
[f)] keigo@riken.jp
[g)] kentaro.nomura@riken.jp
[h)] miyuki.tsubouchi@riken.jp
[i)] jmakino@riken.jp

1. はじめに

粒子法とは研究対象となる系を多数の相互作用を及ぼしあう粒子の集団として表現し，その発展方程式を解くことで，系の進化を追跡する手法である．粒子は発展方程式に従い自動的に移動するため，粗密の大きな系や空隙のある系，また物体の衝突や破壊などのシミュレーションに強く，自然科学や工学の様々な分野で用いられている．近年，スーパーコンピュータは大規模並列化がすすみ，より大規模，高解像度のシミュレーションを行うためには粒子法シミュレーションコードの効率の良い並列化が必須となってきた．

効率の良い並列粒子法コードを開発するためにはアプリケーションプログラマーはロードバランスを考慮したシミュレーション領域の動的分割を考える必要がある．また，計算領域が分割されているので，各プロセスは相互作用計算を行うために他プロセスの持つ粒子の情報を必要とするが，この時の他プロセスとの通信量を小さくする必要もある．アプリケーションプログラマーは上記のような事柄を考慮してシミュレーションプログラムを開発する必要があるが，これらを考慮してアプリケーションプログラムを開発することは容易ではなく，多くの研究者がプログラム開発に膨大な時間を割いているのが現状である．

並列化された粒子法シミュレーションプログラムは数多く存在する [11][12][6][1][3] が，それらの多くはシミュレーションする対象が限定されており，他の分野へそれらのコードを適用する事は難しい．また，数値計算スキームも固定されているため，新しいアルゴリズムを開発したとしても，それらのプログラム上で新しいアルゴリズムを用いる事は困難である．

しかし，シミュレーションする対象が違っても効率の良い並列化アルゴリズムは大きくは変わらない．そこで，我々は並列粒子法コードの開発を容易にするためのフレームワーク FDPS(Framework for Developing Particle Simulators) の開発を行い，2015 年 3 月に FDPSver.1 のリリースを行った [7]．FDPS は並列粒子シミュレーションコードのために必要な関数を備えた C++のテンプレートライブラリであり，アプリケーションプログラマは行いたいシミュレーションに合わせて粒子データと相互作用関数を定義し，FDPS の API を用いて容易にプログラム開発する事ができる。実際，我々は FDPS を用いて，N 体シミュレーションや SPH シミュレーションコードを容易に開発できることを確認した．さらに，「京」等の大規模並列計算機上でシミュレーションを行い，アプリケーションの性能を測定した．FDPS を使って書かれたアプリケーションの実行効率は 30-50%と非常に高い事も確認した．

本稿では Ver.1 リリース後，現在 (2018 年 11 月) の最新バージョンである Ver.5 までに追加した機能について，特にアクセラレータを持つ計算機を効率よく使うためのマルチウォーク法や相互作用リストを再利用する機能，さらに FDPS を C++言語以外から使うための多言語インターフェースについて報告する [10]．本稿の構成は以下のとおりである．2 節では FDPS の概要について述べる．3 節では Ver.2 および Ver.4 で追加されたアクセラレータ向けの新機能について述べる．4 節では Ver.3 および Ver.5 で追加された C++以外の言語から FDPS を使用するための多言語インターフェースについて述べる．5 節で本稿をまとめる．

2. FDPS の概要

2.1 FDPS のデザインコンセプト

FDPS 開発の基本的なアイディア（そして、最終的なゴール）はアプリケーションプログラマーが効率の良い並列粒子シミュレーションコードを短時間で開発できるようにする事である．そのために FDPS は効率の良いプログラムを開発するために必要な API を持っている．これらの API から呼び出される FDPS 内部の関数は MPI および OpenMP により並列化がされているためにユーザーは特に並列化を意識することなく並列化粒子シミュレーションコードを開発することができる．また，FDPS は任意の粒子間相互作用を扱えるように C++のテンプレート機能を用いて開発されている。アプリケーションプログラマは粒子のデータ構造と相互作用関数を定義し，さらに FDPS が提供している API を用いてプログラム開発を行う事が要求される．

2.2 FDPS を用いた粒子シミュレーションの流れ

FDPS を用いた粒子シミュレーションの流れは以下のとおりである．

(1) 各プロセスから粒子をサンプルし，サンプルした粒子からマルチセクション法 [9] により計算領域の分割を行う．
(2) 各プロセスが担当する粒子が担当する領域内に収まるように，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のみで木構造（ローカルツリー）を構築する．
(4) 各プロセスが他のプロセスが相互作用計算を行うために必要な情報 (LET:Local Essentail Tree) を他の全てのプロセスへ送る．
(5) 担当している粒子と LET を用いて木構造（グローバルツリー）を再び構築．
(6) 木構造を用いて相互作用リストを作成し相互作用を計算．
(7) 相互作用計算の結果を用いて粒子の物理量を更新する．
(8) 1 に戻る．

上記の手順1から6までをFDPSは担当する．手順1,2,3–6に対応して，FDPSは`DomainInfo`，`ParticleSystem`，`TreeForForce`と呼ばれるC++のクラスを持ち，ユーザーはこれらのクラスのメンバ関数を用いてプログラム開発を行う．そのため，初期のFDPSではアプリケーションプログラムをC++で書く必要があった．しかし，最新バージョンであるVer.5ではFORTRANやC言語からもFDPSを使用するためのインターフェースが導入されている．これについては4節で詳しく述べる．

2.3 FDPSの開発史

FDPSは2015年3月のリリース以降，2018年11月までに4回のメジャーアップデートを行ってきた．FDPSのメジャーアップデートの歴史とその時の主な機能追加は以下のようになっている．

- 2015年3月 Ver.1をリリース．
- 2016年1月 Ver.2をリリース．マルチウォーク法を導入．
- 2016年12月 Ver.3をリリース．Fortranインターフェースを追加．
- 2017年11月 Ver.4をリリース．相互作用リストを再利用する機能を導入．
- 2018年11月 Ver.5をリリース．C言語インターフェースを追加．

Ver.1では主に「京」のような汎用CPUをベースとした大規模並列計算機で性能が出るように開発された．しかし，ここ10年ほどでGPGPU(General Purpose Graphics Processing Units)等のアクセラレータを搭載したスパコンが増えてきており，FDPSもそれらの計算機に対応させる必要があった．そこで、Ver.2からは浜田等[5]によって開発されたマルチウォーク法をFDPSに導入しアクセラレータ対応を図った．さらにVer.4からは，同じツリー構造や相互作用リストを複数ステップの間使い続ける機能を導入した．さらに，ホストとアクセラレータ間での相互作用リストの転送量を減らすために，相互作用計算を始める前に全粒子とLETをアクセラレータに送り，相互作用リストとしては粒子のアクセラレータ上でのアドレスのみを送る方法も導入した．Ver.2およびVer.4で追加されたこれらの機能については3節で詳しく解説する．

FDPSはC++により開発されており，Ver.1ではアプリケーションプログラマはC++言語によりアプリケーションを書くことが求められていた．しかし，HPCではFORTRANやC言語もよく用いられており，C++になじみの薄い開発者も多い．そこでVer.3ではFDPSをFortranから呼び出すためのインターフェースの開発を行った．またVer.5では同様にC言語からもFDPSを呼び出すことのできるインターフェースの開発を行った．Ver.3およびVer.5で追加されたこれらのインターフェースについては4節で詳しく解説する．

3. アクセラレータ対応

本節ではGPGPU等のアクセラレータを持つ計算機上で高い実行効率を達成するために追加された機能について述べるo．追加された機能は大きく以下の3つである．

- マルチウォーク法
- 粒子への間接アドレス指定
- 相互作用リストの再利用

以下これらの機能について解説する．

3.1 マルチウォーク法

FDPSでは相互作用関数をユーザーが任意に定義できるため，粒子法で最もコストがかかる相互作用計算のみをアクセラレータで行わせることができる。しかし，Ver.1ではアクセラレータを持つ計算機では高い実行効率をあげる事は困難であった．これは以下の理由による．FDPSでは相互作用の計算にはBarnesによって開発された手法[2]を用いている．これは一つの粒子に対して相互作用リストを作るのではなく，近傍にいる粒子をグループ化し，その粒子グループについて相互作用リストを作り，相互作用を計算する手法である．Ver.1では一つの粒子グループに対して相互作用リストをつくり計算するということを繰り返し行っていた．しかし，GPGPU等のアクセラレータではカーネルの起動オーバーヘッドが大きく相互作用リストの数が大きくなるとこのオーバーヘッドが無視できなくなる．また，非常に大量の演算コアを搭載しているため，1つの相互作用リストについて計算するだけでは全ての演算コアを使いきることはできない．そこで，Ver.2ではマルチウォーク法と呼ばれる手法を導入した．

マルチウォーク法では、FDPSは一度に複数の相互作用リストを作り，それらをまとめてアクセラレータに転送し相互作用計算を実行する．これにより、カーネル起動オーバーヘッドを削減でき，また全ての演算コアを動作させることができる．さらに，アクセラレータでの相互作用計算を行っている間にCPU側で次の相互作用リストを作ることができるため，相互作用リスト作成にかかる時間を隠ぺいすることができる．

3.2 粒子への間接アドレス指定

アクセラレータを搭載している計算機上で相互作用リストを用いて計算を行う簡単な方法の一つは，それぞれの粒子グループとそれに対応する相互作用リストに対して，計算に必要な物理量をアクセラレータに送り計算する事である．しかし，相互作用リスト内の粒子数は非常に大きく，大雑把にはローカルに存在する粒子数の10倍以上になる．一般に，LETの数はローカルの粒子数より非常に少ないので，これは同じ粒子をアクセラレータに10回以上送って

いる事を意味する．

同じ粒子を複数回送らないようにするためには，相互作用計算を始める前に全てのローカル粒子と LET をアクセラレータに送ってしまい，相互作用リストには粒子もしくは LET のアクセラレータ上でのインデックスのみを記憶させればよい．こうすることで相互作用の時に粒子の物理量を送るのではなく，インデックスだけを送ればよいことになる．これにより，例えば重力計算の場合は粒子のデータサイズが 16 バイト程度であるのに対して，インデックスは 4 バイトあればよいので転送データサイズを四分の一にできる．また，SPH 等の流体シミュレーションの場合は粒子データが 50 バイト程度になることもあり，転送量を十分の一にすることができる．

3.3 相互作用リストの再利用

SPH シミュレーション等の流体計算や分子動力学シミュレーション等では安定性条件から時間刻みが制限されるため，粒子が時間刻みあたりに移動する距離は短い．そのため，一度作った相互作用リストを複数時間刻みの間，再利用することができる．また，重力 N 体シミュレーションの場合でも，例えば惑星形成シミュレーションや惑星系リングのシミュレーションなどでは，時間刻み当たりの粒子間距離の変化は小さく，相互作用リストの再利用が可能である．銀河形成シミュレーション等の N 体+SPH シミュレーションの場合等でも，時間刻みは流体粒子の時間刻みできまり，この刻みの間では N 体粒子の移動距離は短いと考えられるためやはり，相互作用リストの再利用は可能である。このため，FDPS Ver.4 では相互作用リストの再利用を行う機能を搭載した．

相互作用リストを再利用した場合には相互作用リストを構築するステップ (construction step) とリストを再利用するステップ (reuse step) の手順は異なる．construction step 時相互作用計算の手順は以下のとおりである．

(1) ローカルツリーを構築する．
(2) LET を交換する．
(3) グローバルツリーを構築.
(4) 木構造を用いて相互作用リストを作り、相互作用を計算。この時相互作用リストを記憶する．
(5) 相互作用計算の結果を用いて粒子の物理量を更新する．

construction step 時の手順に対して，reuse step 時の手順は以下のようになる．

(1) ローカルツリーの物理量を更新する．
(2) LET を交換する．
(3) グローバルツリーの物理量を更新する．
(4) 記憶されている相互作用リストを用いて相互作用を計算.
(5) 相互作用計算の結果を用いて粒子の物理量を更新する．

つまり，reuse step では以下の手順を省くことができる．a) 木構造の構築，b)LET の構築，c) 相互作用リストの構築．a) の計算コストは O(n) であり，b) および c) の計算コストは O($n$log$n$) となる．それゆえ，これらの計算時間は無視できるものではなく，これらの手順を省ける事による効果は大きい.

3.4 API

この節では 3.1 から 3.3 節で述べた機能を使用するための API について述べる．

FDPS では一回の API 呼び出しで，相互作用計算に必要な計算を全て行う高レベル API が存在する．マルチウォーク法を用いる場合は，ユーザーは `CalcForceAllAndWriteBackMultiWalk` もしくは `CalcForceAllAndWriteBackMultiWalkIndex` を使用すればよい．これら二つの関数の違いは 3.2 節で述べた粒子への間接アドレス指定を用いるかどうかである．どちらの関数を使用する場合でもリストを再利用することは可能である．リストを再利用するかどうかは上記の高レベル関数の引数として指定する．

図 1 はメイン関数の例である．6,7,8 行目で `ParticleSystem` クラス，`DomainInfo` クラス，`TreeForForce` クラスのオブジェクトを生成している．14 行目からが時間積分のループである．この例では 8 ステップごとに領域分割 (17 行目)，粒子交換 (18 行目)，ツリーと相互作用リスト作成を行っている．`CalcForceAllAndWriteBackMultiWalkIndex` の最後の引数に `MAKE_LIST_FOR_REUSE` を用いることで，FDPS は木構造や相互作用リストを作成し，それらを内部に保存する (construction step)．reuse step では引数に `REUSE_LIST` を与えることで，FDPS は内部に保存されているツリーや相互作用リストを使用し，相互作用計算を行う．

マルチウォーク法を使用する場合はユーザー定義の相互作用関数は 2 つの部分に分ける必要がある．一つは dispatch 関数 (22 行目) と呼ばれるもので，ホストからアクセラレータへの転送および，アクセラレータ上で実行される相互作用計算のためのカーネルを起動させるためのものある．もう一つは retrieve 関数 (23 行目) と呼ばれアクセラレータ内での計算結果を回収するためのものである．

関数が二つに分かれている理由はアクセラレータで相互作用を計算している間に次の粒子グループの相互作用リストを作り，また既に相互作用を計算した粒子の積分を行うことで，これらにかかる時間を隠ぺいするためである．

dispatch 関数および retrieve 関数の実装例については FDPS 付属のサンプルコードを参照されたい．

3.5 性能

この節では，上記の三つのアルゴリズムを用いた場合の N 体シミュレーションコードの性能について述べる．3.5.1

```
using namespace PS;
int main(int argc, char *argv[]) {

  // ＦＤＰＳの初期化（省略）

  ParticleSystem<FP> system;
  DomainInfo dinfo;
  TreeForForceLong<FORCE,EPI,EPJ>::Monopole
    tree;

  // 初期条件設定（省略）

  int n_loop =0;
  while(time_sys < time_end){
    INTERACTION_LIST_MODE int_mode=REUSE_LIST;
    if(n_loop % 8 == 0){
      dinfo.decomposeDomainAll(system);
      system.exchangeParticle(dinfo);
      int_mode = MAKE_LIST_FOR_REUSE;
    }
    tree.calcForceAllAndWriteBackMultiWalkIndex
      (DispatchKernelIndex,
       RetrieveKernel,
       tag_max,
       system,
       dinfo,
       n_walk_limit,
       true,
       int_mode);

    // 軌道積分（省略）

    n_loop++;
  }
  return 0;
}
```

図 1 メイン関数の例．この例では，8 ステップ毎にツリーと相互作用リストを作り直している．

節では CPU のみを用いた場合とアクセラレータとして GPGPU を用いた場合のシングルノードでの性能を，3.5.2 節ではヘテロジニアスメニーコアである Sunway 26010 と PEZY-SC2 を用いた大規模並列システムでの性能について述べる．

3.5.1 GPGPU を用いた場合のシングルノードでの性能

アクセラレータとして Nvidia TITAN V を用い，ホスト CPU は Intel Xeon E5-2670 v3 を用いた．これらの演算性能は単精度の場合 13.8Tflops,883Gflops である．ホストのメインメモリは DDR4 であり，理論ピーク転送速度は 68 GB/s．CPU と GPGPU は PCI Express 3.0 で接続されており，理論ピーク転送速度は 15.75 GB/s である．

シミュレーションの初期条件としては速度分散がない一様球を考える．この場合，系は自己相似的に崩壊するため，相互作用リストの再利用が可能である．粒子数は 4M(=2^{22}) とし、ツリーのオープニングクライテリオン θ は 0.5 とした．

図 2 は CPU を使用した場合と GPGPU を使用し上記の三つのアルゴリズムを用いた場合の時間ステップ当たりの計算時間を表している．横軸は相互作用リストを共有する粒子の数の平均である (平均 i 粒子数)。計算時間は平均 i 粒子数に依存するが、最適な平均 i 粒子数では、CPU のみを用いた場合に対して，GPU を用いた計算は construction step で 3 倍，reusing step で 10 倍ほど速くなっていることが分かる．つまり，GPU を用い Ver.5 までに追加された機能を用いることで，CPU を用い Ver.1 の機能のみを使った場合に比べて最大で 10 倍高速に計算する事が可能になった．

また，最適な平均 i 粒子数での計算速度は CPU のみを用いた場合は 432Gflops でありこれは理論ピーク性能の 49%にあたる．GPU を用いた場合の計算速度は construction step で 3.0Tflops，reuse step で 4.5Tflops であり、これはそれぞれ、理論ピーク性能の 22%と 33%にあたり、非常に高い効率で動作していることが分かる．

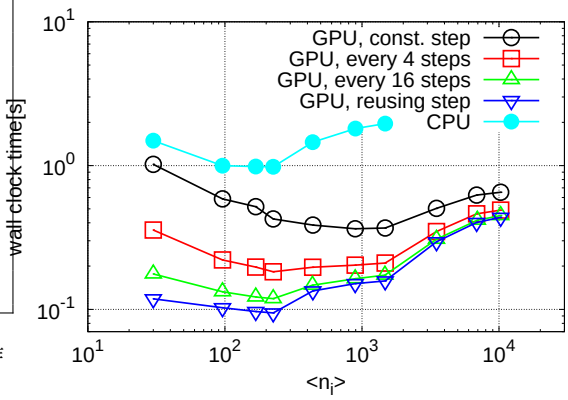

図 2 ステップ当たりの計算時間を平均 i 粒子数に対してプロットした．

3.5.2 大規模並列ヘテロジニアスメニコアシステムでの性能

ここでは大規模並列システムでの性能について述べる．ここでは計算機として以下の 3 システムを使用した．

- Taihulight
- Gyoukou
- Shoubu System-B

Taihulight は 40960 個の Sunway 26010 プロセッサからなる [4]．1 つの Sunway 26010 プロセッサは 4 つの Core Group(CG) からなり，1 つの CG には 1 個の MPE(Management Processing Element) と 64 個の

CPE(Computing Processing Element) からなる．各 CG は DDR3 のメインメモリをもち，理論ピーク転送速度は 34 GB/s である．各 PE のクロックは 1.45 GHz であり，1 サイクルに 4 つの倍精度 FMA を行う事ができるため，CG 当たりの理論ピーク性能は 754Gflops である．我々は TaihuLight をもちいるときは 1 つの CG に 1 つの MPI プロセスを割り当てた．

Gyoukou および Shoubu System-B はそれぞれ 13312 個と 512 個の PEZY-SC2 プロセッサチップからなる．各チップは DDR4 のメインメモリをもち，理論ピーク転送速度は 76.8GB/s である．PEZY-SC2 プロセッサは 2048 個のコア (有効なコアは 1984 個) からなる．各コアは 1 サイクルに倍精度の積和演算を一回実行することができる．単精度や半精度の場合は 2 もしくは 4 ウェイの SIMD として動作し，コアの動作周波数は 700MHz である．そのため，1 チップあたりの理論ピーク性能は倍精度で 2.8Tflops，単精度で 5.6Tflops，半精度で 11.1Tflops となる．我々は PEZY-SC2 ベースのシステムを使用するときは一つのチップに一つの MPI プロセスを割り当てた．また，相互作用の計算には単精度を使用した．

これらのシステムを用いて我々は土星の環のシミュレーションを行った．粒子数は MPI プロセスあたりに 10^7 個とした．図 3 は弱スケーリングでの性能である．全てのシステムで非常によくスケールしていることが分かる．性能は Shoubu System-B の 512 プロセスで 1.01Pflops，Gyoukou の 8192 プロセスで 10.6Pflops，TaihuLight の 16 万プロセスで 47.9Pflops である．これはそれぞれ理論ピーク性能の 35.5%，23.5%，40.0%にあたる。同じ PEZY-SC2 のシステムであるが，Gyoukou の性能が Shoubu System-B の性能よりも低いのは Gyoukou が 2018 年 3 月に停止したため，コードのチューニングが十分でなかったためである．なお、これらの計算に使用したコードでは大規模リング計算向けの最適化も行われている．詳細は [8] にある．

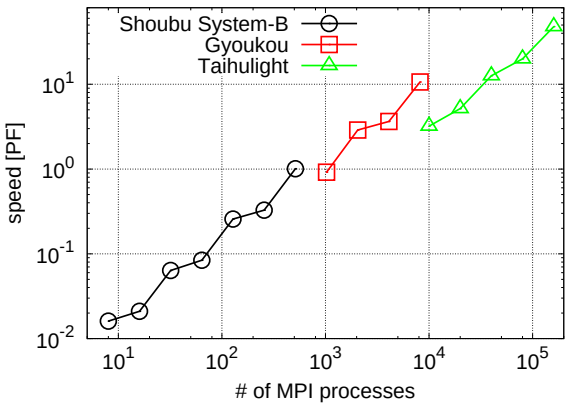

図 3 弱スケーリング時の性能．縦軸は PetaFlops 単位での計算速度、横軸は MPI のプロセス数．

4. 多言語インターフェース

4.1 インターフェースの設計

Fortran/C 言語インターフェースを開発する上での一つの困難は，Fortran や C 言語から C++のテンプレート機能を直接利用する手段がないことである．二節で述べた通り，C++のテンプレート機能は FDPS に任意の粒子データ型や粒子間相互作用をサポートさせる上で不可欠なものであった．したがって，Fortran/C 言語インターフェースでも任意の粒子データ型や粒子間相互作用をサポートするためには，何らかの方法でこの問題を解決する必要がある．

我々は以下の事柄を組み合わせてこの問題を解決した．(i) C++関数は `extern "C"`修飾子によって C 言語からも呼び出せる事，(ii) FDPS の C++コア部 (C++で実装された FDPS の部分) を操作する C++プログラムの自動生成，(iii) Fortran 2003 で導入された Fortran と C 言語の相互運用を保証する仕組み．(i) と (iii) によって，特定の粒子データ型向けに作られた FDPS ライブラリ (C++コア部を操作する一連の C++関数群のことと定義) を Fortran や C 言語から利用可能になる．しかしながら，実現したいのは任意の粒子データ型を扱える FDPS ライブラリを Fortran や C 言語から利用できるようにすることである．これを実現する唯一の方法は，与えられた粒子データ型に応じた FDPS ライブラリを生成することである．我々は FDPS の利用者が Fortan や C 言語のみで，アプリケーションを開発できるようにしたいので，それぞれの言語で粒子を表すデータ構造から FDPS ライブラリを生成する必要がある．我々は粒子を表すためのデータ構造として，Fortran では派生データ型，C 言語では構造体を採用した．したがって，我々の解決案は次のようなものである．まず，FDPS の利用者は Fortran 2003 の派生データ型 或いは C 言語の構造体で粒子を定義する．次に，この粒子データ型に対応する C++クラスを生成する．最後に，この C++クラス向けの FDPS ライブラリを生成する．これが (ii) が必要な理由である．これらの生成は我々が用意する Python スクリプトが自動的に行う．

残る問題点は，どのようにして Fortran 或いは C 言語で書かれた粒子データ型から対応する C++クラスを生成するのかということである．適切な生成には，以下の情報が必要となる．

- データ型が粒子を表すものであること．
- データ型のどのメンバ変数が FDPS が必要とする物理量に対応するか．FDPS は粒子の位置等，特定の物理量を必要とする．C++から FDPS を利用する場合，利用者に位置を返すメンバ関数等，いくつかのメンバ関数を定義してもらうことでこの問題を解決する設計となっていた．しかしながら，Fortran の派生データ型や C 言語の構造体はメンバ関数は持てない．

我々は独自の指示文を導入してこの問題を解決した．指示文は特定のフォーマットのコメント文で，自動生成を行う Python スクリプトに必要な情報を渡すために使用される．具体的には，例えば Fortran の場合，派生データ型名の隣にコメント文`!$fdps FP` を記述すると，スクリプトは，この派生データ型が FullParticle 型であると解釈する．ここで，FullParticle 型は粒子のすべての情報を持つデータ型のことである．同様に，メンバ変数名の隣にコメント文`!$fdps position` を記述すると，このメンバ変数は粒子の位置を表すとスクリプトは解釈する．このような指示文を用いることで，スクリプトは必要な C++クラスを生成することが可能となる．

以上が Fortran/C 言語インターフェースの設計である．図 4 は，インターフェースの仕組みを模式的に表したものである．まずユーザは粒子を Fortran の派生データ型 或いは C 言語の構造体で定義する．このとき，指示文を使って必要な情報も記述する (ステップ①)．自動生成スクリプトは，ユーザが書いたソースコードの中から指示文を手がかりに粒子を表す派生データ型 ないしは 構造体を自動的にすべて見つけ，対応する C++クラスを (複数) 生成する (ステップ②)．同時に，これら C++クラス向けの FDPS ライブラリを生成する (ステップ③)．これは C++から利用可能な FDPS の API とほぼ同じものを提供する C++関数群から構成される．スクリプトはこの C++関数群の C 言語インターフェースも生成する (ステップ④)．これが FDPS の C 言語インターフェースと呼んでいるものである．Fortran から使用可能にするための Fortran モジュールも生成される (ステップ⑤)．このモジュールは Fortran 用のインターフェースを提供する．ユーザはこれらのインターフェースで提供される API を使用して，アプリケーションを開発する (ステップ⑥, ⑥')．

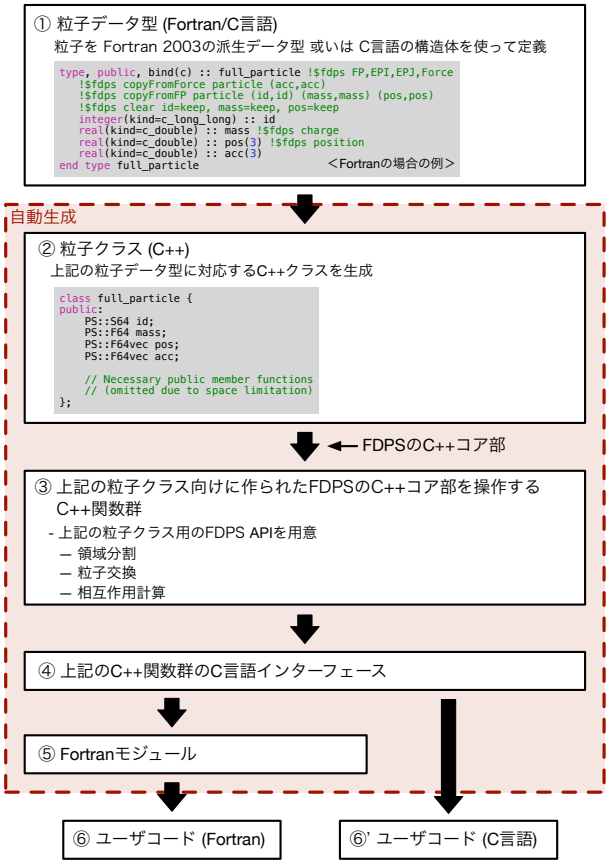

図 4　Fortran/C 言語インターフェースの仕組み

4.2　インターフェースの利用例

本節では C 言語インターフェースの利用例を簡単に説明する．前節で述べた通り，ユーザはまず C 言語の構造体と指示文を用いて粒子を定義する必要がある．そこで，粒子を定義した C 言語ソースファイルの例を図 5 に示す．上から順に説明していく．まず，ファイル冒頭でヘッダーファイル `FDPS_c_if.h` がインクルードされている．このヘッダーファイルには FDPS の C 言語用 API のプロトタイプ宣言や FDPS から提供されるデータ型が記述されており，これらを使用する場合には必ずインクルードする必要がある．

次に構造体タグ名の右隣に以下のコメント文がある．

```
//$fdps FP,EPI,EPJ,Force
```

これはこの構造体が粒子であることを示す指示文である．文字列`//$fdps` で始まるコメント文はすべて自動生成ス

```c
#pragma once
#include "FDPS_c_if.h"
typedef struct full_particle { //$fdps FP,EPI,EPJ,Force
    //$fdps copyFromForce full_particle (pot,pot) (acc,acc)
    //$fdps copyFromFP full_particle (id,id) (mass,mass) (eps,eps)
    ↪  (pos,pos)
    //$fdps clear id=keep, mass=keep, eps=keep, pos=keep, vel=keep
    long long id; // $fdps id
    double  mass; //$fdps charge
    double  eps;
    fdps_f64vec pos; //$fdps position
    fdps_f64vec vel;
    double  pot;
    fdps_f64vec acc;
} Full_particle;
```

図 5　C 言語での粒子の実装例

クリプトに指示文として解釈される．文字列`//$fdps`の右側には粒子の種別を表すキーワードが続いる．`FP`は FullParticle 型，`EPI`は EssentialParticleI 型，`EPJ`は EssentialParticleJ 型，`Force`は Force 型であることを示すキーワードである．複数のキーワードが指定された場合，それらを同時に兼ねることを示す．FullParticle 型以外のデータ型の定義については FDPS の仕様書を参照されたい．

メンバ変数宣言部の前の部分 (3-6 行目) に 3 つの指示文がある．これらは FDPS 内部でのデータ処理の仕方を指示するための指示文である．こちらも詳細については，仕様書を参照されたい．

メンバ変数宣言部に注目すると，いくつかのメンバ変数の右隣に指示文がある．文字列`//$fdps`に続く文字列はどの物理量に対応しているかを示すキーワードであり，`id`は粒子 ID，`charge`は電荷 (質量)，`position`は位置を表す．

次にメイン関数の実装例を図 6 に示す．紙面の都合上，ヘッダーファイルのインクルード等の部分は省いた．ここで使用されている関数の内，関数名が`fdps_`で始まるものはすべて C 言語用の API である．図 1 と比較してみると，使用している API は異なるが，全体的な構造は C++で実装する場合と似た構造となる．すなわち，はじめに FDPS を初期化するための API を呼び出す (API `fdps_initialize`)．次に，`DomainInfo`, `ParticleSystem`, `TreeForForce` オブジェクトを生成・初期化する (API `fdps_create_*`, `fdps_init_*` [*は正規表現])．そして，これらを使って，領域分割 (API `fdps_decompose_domain_all`)，粒子交換 (API `fdps_exchange_particle`)，そして相互作用計算 (API `fdps_calc_force_all_and_write_back`) を行う，という構造となっている．

以上，C 言語インターフェースの利用例についての簡単な説明を行った．Fortran インターフェースを用いる場合でも，メイン関数の構造はほぼ同じ構造となる．Fortran インターフェースを使った例に関しては，FDPS に付属するドキュメントや [10] を参照されたい．

```
1  int c_main()
2  {
3      // FDPS の初期化
4      fdps_initialize();
5      // 領域情報オブジェクトの生成と初期化
6      int dinfo_num;
7      float coef_ema=0.3;
8      fdps_create_dinfo(&dinfo_num);
9      fdps_init_dinfo(dinfo_num,coef_ema);
10     // 粒子群オブジェクトの生成と初期化
11     int psys_num;
12     fdps_create_psys(&psys_num,"full_particle");
13     fdps_init_psys(psys_num);
14     // ツリーオブジェクトの生成と初期化
15     int tree_num;
16     fdps_create_tree(&tree_num,
17                      "Long,full_particle,full_particle,full_particle,Monopole");
18     int ntot=1024;
19     double theta = 0.5;
20     int n_leaf_limit = 8;
21     int n_group_limit = 64;
22     fdps_init_tree(tree_num, ntot, theta, n_leaf_limit, n_group_limit);
23
24     // 初期条件作成 (省略)
25
26     // 領域分割と粒子交換
27     fdps_decompose_domain_all(dinfo_num,psys_num,-1.0);
28     fdps_exchange_particle(psys_num,dinfo_num);
29     // 相互作用計算
30     fdps_calc_force_all_and_write_back(tree_num,
31                                        calc_gravity_ep_ep,
32                                        calc_gravity_ep_sp,
33                                        psys_num,
34                                        dinfo_num,
35                                        true,
36                                        FDPS_MAKE_LIST);
37     // 粒子の軌道の時間積分
38     double time_sys  = 0;
39     double time_end  = 10.0;
40     double dt = 1.0/128.0;
41     while (time_sys <= time_end){
42         kick(psys_num,0.5*dt);
43         time_sys +=  dt;
44         drift(psys_num,dt);
45         // 領域分割と粒子交換
46         fdps_decompose_domain_all(dinfo_num,psys_num,-1.0);
47         fdps_exchange_particle(psys_num,dinfo_num);
48         // 相互作用計算
49         fdps_calc_force_all_and_write_back(tree_num,
50                                            calc_gravity_ep_ep,
51                                            calc_gravity_ep_sp,
52                                            psys_num,
53                                            dinfo_num,
54                                            true,
55                                            FDPS_MAKE_LIST);
56         kick(psys_num,0.5*dt);
57     }
58     fdps_finalize();
59     return 0;
60 }
```

図 6 C 言語でのメイン関数の実装例．関数`drift`, `kick`は，それぞれ粒子の位置と速度を時間積分する関数である．また，関数`calc_gravity_ep_ep`, `calc_gravity_ep_sp`はそれぞれ粒子間相互作用，粒子-超粒子間相互作用を計算する関数である．

5. まとめ

我々は FDPS Ver.1 をリリース以降今までに 4 回のメジャーアップデートを行ってきた．Ver.2 および Ver.4 では主にアクセラレータを搭載した計算機向けの機能強化を行ってきた．結果，これらの追加機能を用いることで GPU を用いた計算は CPU のみを用いた計算に比べて計算時間が 10 倍程度短くできることが分かった．また，Gyoukou や TaihuLight 等の大規模ヘテロジニアスメニコアシステムでも性能測定を行い非常に高い実行効率を達成することができた．

また，Ver.3 および Ver.5 では FDPS を Fortran や C 言語から呼び出すためのインターフェースの開発を行った．これらのインターフェースは Fortran や C 言語から FDPS

の機能を呼び出すための API や，これらの言語で書かれた粒子データから C++のクラスを生成するコードジェネレータを備えている．これらのインターフェースを用いることで C++になじみの薄い Fortran や C 言語のユーザーも容易に FDPS を使うことができる．

謝辞

この研究は「次世代領域研究開発」（高性能汎用計算機高度利用補助金）「ヘテロジニアス・メニーコア計算機による大規模計算科学」，計算科学振興財団 研究教育拠点（COE）形成推進事業「 ポスト「京」，ポスト・ポスト「京」をみすえたハードウェア・アルゴリズム・ソフトウェアの総合的研究 」，および JSPS 科研費（JP18K11334）の補助を受けています．

参考文献

[1] Abrahama, M. J., Murtolad, T., Schulzb, R., Palla, S., Smithb, J., Hessa, B. & Lindahl., E. 2015, SoftwareX, 1, 19
[2] Barnes, J. E. 1990, Journal of Computational Physics, 87, 161
[3] Bédorf, J., Gaburov, E., Fujii, M. S., Nitadori, K., Ishiyama, T. and Zwart, S. P., Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, Piscataway, NJ, USA, IEEE Press, pp. 54–65
[4] Fu H, Liao J, Yang J, Wang L, Song Z, Huang X, Yang C, Xue W, Liu F, Qiao F et al. 2016, Science China Information Sciences, 59(7): 072001.
[5] Hamada, T., Narumi, T., Yokota, R., Yasuoka, K., Nitadori, K., & Taiji, M. 2009, Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 62, 1
[6] Ishiyama, T., Fukushige, T., & Makino, J. 2009, Publications of the Astronomical Society of Japan, 61, 1319
[7] Iwasawa, M., Tanikawa, A., Hosono, N., et al. 2016, Publications of the Astronomical Society of Japan, 68, 54
[8] Iwasawa, M., Long, W., Keigo, N., et al. 2018, Lecture Notes in Computer Science, vol 10860. Springer, Cham
[9] Makino, J. 2004, Publications of the Astronomical Society of Japan, 56, 521
[10] Namekata, D., Iwasawa, M., Nitadori, K., et al. 2018, Publications of the Astronomical Society of Japan, 70, 70
[11] Plimpton, S. 1995, J. Comp. Phys., 117, 1
[12] Springel, V. 2005, Monthly Notices of the Royal Astronomical Society, 364, 1105