FDPSの開発

岩澤全規

構成

- FDPSのリリースまで
- FDPS開発の動機
- FDPSの概要
- FDPS Ver1以降の進化
  - アクセラレータ向け機能
    - 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
    - 性能、性能モデル
  - 多言語インターフェース
  - PIKG
- FDPSのこれから
- まとめ

FDPSのリリースまで

- 2012年4月，理研計算科学研究機構(現R-CCS)に粒子系シミュレータ研究チーム発足
- 2012年11月，似鳥，岩澤着任
  - 三宮で牧野さんと夕食食べながら，粒子系一般を扱えるようなDSLの話を聞く.
  - 「GreeMより2倍とか遅くなければよいから」と言われた.
  - その後しばらくは似鳥さんはPMMM, 岩澤はPPPTをやっていた.

FDPSのリリースまで

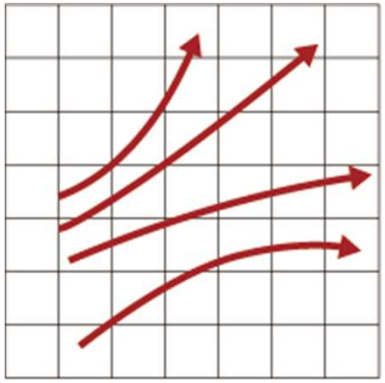
- 2013年4月，谷川着任
  - ここら辺からFDPS開発が動き始める.
  - 3人で議論して，牧野さんに持って行って(却下)を繰り返していた.
    - 仕様書を作る.
    - FDPSは仮名で，誰かがそのうち良い名前を付けるはずとほっといた.
  - 岩澤はリファレンスモデルとしての京の通信性能を調べつつ，並列ツリーコードを開発.
    - alltoallが死ぬほど遅かった.
    - 1M粒子を5ms.
- 2014年4月，細野着任
  - この頃から実装仕様書を作っていた.
  - 2014年の秋(確か)から実装を開始.
    - C++03までの機能しか使えなかったのが大変だった記憶

FDPSのリリースまで

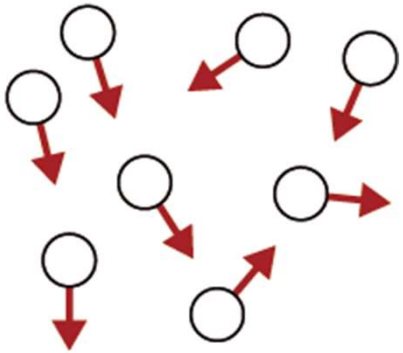
- 2015年3月 FDPS Ver. 1リリース
  - 天文学会で何人かにFDPSの事で話かけられた記憶がある.
  - 結構良さげな中華の店でお祝い.
    - よだれ鶏を知る.
    - 村主さんと結構お話した.
    - 牧野さんがおごってくれた．ごちそうさまでした.

粒子シミュレーション

- 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ションする。
  - 粒子が自動的に集まり系を表現するため、物体の衝突や破壊、形が大きく変わる系、密度コントラストの高い系等のシミュレーションに強い。
  - 科学、工学の幅広い分野で使われている。
  - 重力N体、MD、個別要素法、SPH、MPS、メッシュフリー法等。



メッシュ法



粒子法

FDPSの開発動機

- 大規模並列粒子シミュレーションコードの開発は容易ではない。
  - 多くの研究者がプログラムの開発に多くの時間を取られてしまい、本来行うべき研究の時間が少なくなってしまう。
  - プログラム開発を得意としない研究者がシミュレーションを諦めてしまう。

その分野の発展を鈍らせる

- 様々な粒子法を用いたアプリケーションが存在する。

  e.g. 天文学では GreeM, GAGET, pkdGrav, etc…

  工学分野では Lex-ADV, EMPS, etc…
  - これらは特定の問題に特化しており、他の問題へ応用する事は難しい。
  - 新しいアルゴリズム等の導入が困難。
- しかし、粒子法シミュレーションの並列化手法アルゴリズムは問題によらず似ている。
- 並列化アルゴリズムを一般化し、粒子法コード開発を支援するフレームワークの開発は可能か？

可能なら科学、工学、様々な分野の発展に寄与できる！

並列化粒子法コード開発を支援するフレームワーク？

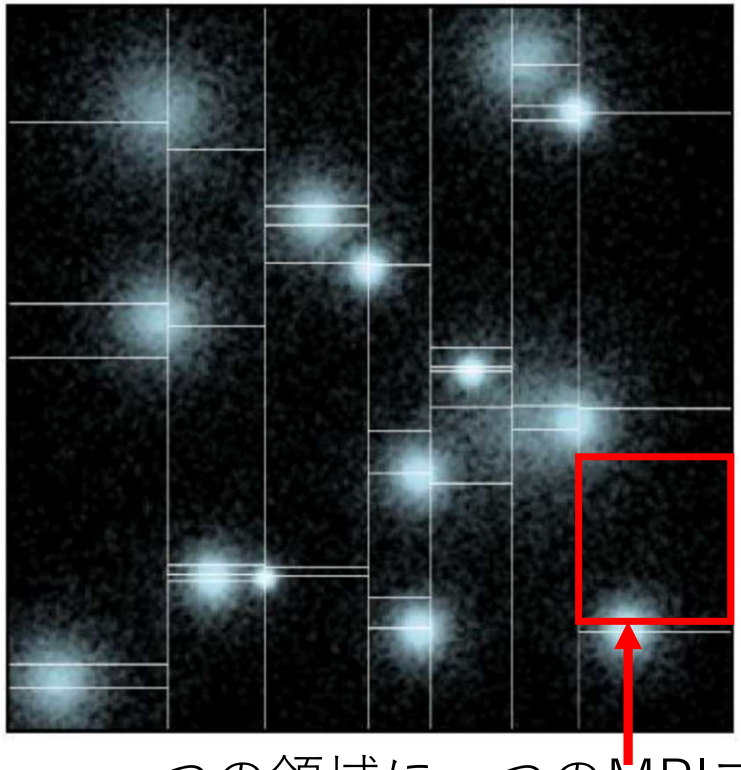
- 実現したい機能
  - ユーザは並列化やツリー構造による粒子管理など、プログラム開発を困難にしている部分を意識する必要がない
  - ユーザーが自ら相互作用関数を定義することで、任意の2粒子間相互作用を扱うことができる
- 実現する方法
  - 粒子法コードのアルゴリズムを抽象化して考える。

抽象化された粒子法並列化アルゴリズム

1. 計算領域の分割
2. 計算領域に合わせた粒子交換
3. 相互作用計算の為のツリー構築とツリーウォーク（相互作用リスト作り）

抽象化された粒子法並列化アルゴリズム

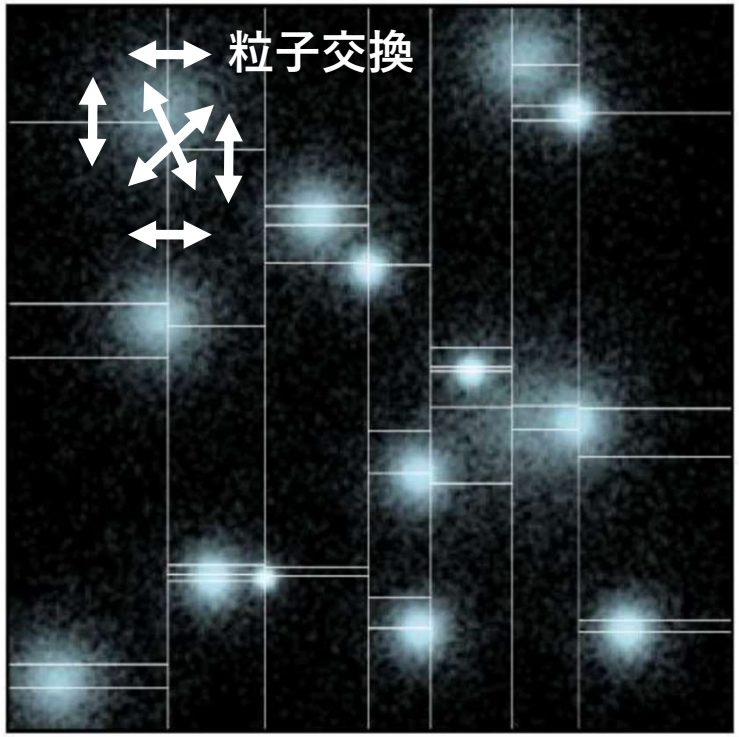
1. 計算領域の分割
   - 粒子の位置情報のみ必要。
2. 計算領域に合わせた粒子交換
   - 粒子の位置情報のみ必要。
3. 相互作用計算の為のツリー構築とツリーウォーク（相互作用リスト作り）
   - 粒子の位置情報が必要。
     - 遠距離力の場合は質量や電荷等も必要。
   - 力の種類(遠or近距離力)が必要。



一つの領域に一つのMPIプロセスを割り当てる。

抽象化された粒子法並列化アルゴリズム

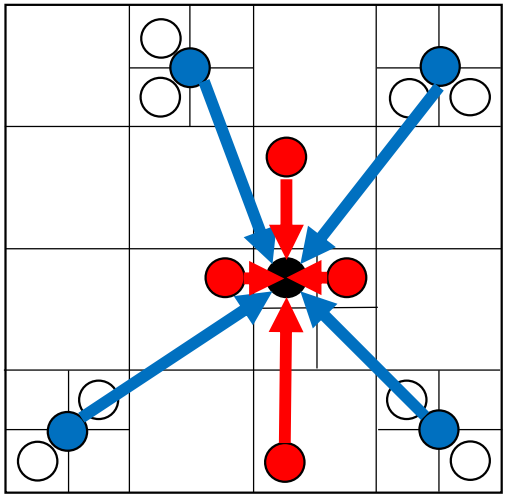
1. 計算領域の分割
   - 粒子の位置情報のみ必要。
2. 計算領域に合わせた粒子交換
   - 粒子の位置情報のみ必要。
3. 相互作用計算の為のツリー構築とツリーウォーク（相互作用リスト作り）
   - 粒子の位置情報が必要。
     - 遠距離力の場合は質量や電荷等も必要。
   - 力の種類(遠or近距離力)が必要。



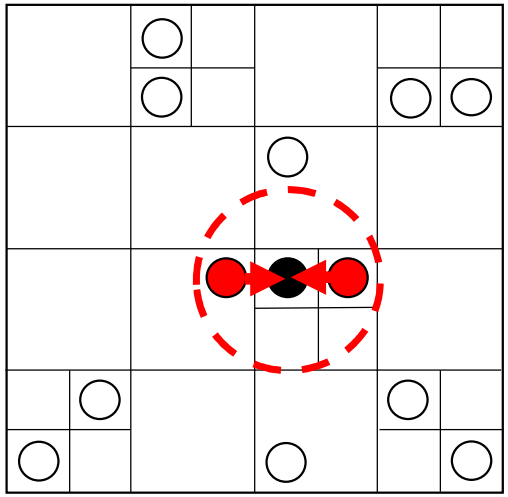
抽象化された粒子法並列化アルゴリズム

1. 計算領域の分割
   - 粒子の位置情報のみ必要。
2. 計算領域に合わせた粒子交換
   - 粒子の位置情報のみ必要。
3. 相互作用計算の為のツリー構築とツリーウォーク（相互作用リスト作り）
   - 粒子の位置情報が必要。
     - 遠距離力の場合は質量や電荷等も必要。
   - 力の種類(遠or近距離力)が必要。

遠距離力



短距離力



粒子の位置情報と相互作用の種類が分かれば粒子法の並列化の一般化は可能。
Framework for Developing Particle Simulators(FDPS)を開発。

構成

- FDPS開発の動機
- FDPSの概要
- アクセラレータ向け機能
  - 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
  - 性能、性能モデル
- 多言語インターフェース
- FDPSの次期バージョン
- まとめ

FDPSのデザインコンセプト

- FDPS開発の基本的なアイディア(最終目標)
  - アプリケーションプログラマが効率の良い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。

- 内部実装の言語はC++
  - 高い自由度と高い性能を両立させるためにFDPSはC++のテンプレートライブラリになっている。
  - ユーザーは相互作用関数と粒子データを定義し、FDPSのAPIを使ってプログラムの開発を行う。
- 並列化
  - 分散メモリー環境:MPI
  - 共有メモリー環境:OpenMP
  - APIの内部で使われている関数は並列化されており、ユーザーは並列化を意識してコードを書く必要がない。

FDPSを用いた粒子シミュレーションの流れ

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により計算領域分割を行う.
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

手順1-6をFDPSは担当。

FDPSは手順1,2,3-6に対応したC++のクラスをもつ。

- DomainInfoクラス: 領域のデータを持ち、領域分割を行う。
- ParticleSystemクラス: 粒子のデータを持ち、粒子交換を行う。
- TreeForForceクラス: 相互作用の計算を行う。

ユーザーはこれらのクラスを用いてプログラムを開発する

FDPSを使ったプログラム例

Listing 1 shows the complete code which can be actually compiled and run, not only on a single-core machine but also massively-parallel, distributed-memory machines such as the full-node configuration of the K computer. The total number of lines is only 117.

Listing 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",
               &mass, &eps,
               &pos.x, &pos.y, &pos.z,
               &vel.x, &vel.y, &vel.z);
    }
    void predict(F64 dt) {
        vel += (0.5 * dt) * acc;
        pos += dt * vel;
    }
    void correct(F64 dt) {
        vel += (0.5 * dt) * acc;
    }
};

template <class TPJ>
struct CalcGrav{
    void operator () (const Nbody * ip,
                      const S32 ni,
                      const TPJ * jp,
                      const S32 nj,
                      Nbody * force) {
        for(S32 i=0; i<ni; i++){
            F64vec xi  = ip[i].pos;
            F64    ep2 = ip[i].eps
                * ip[i].eps;
            F64vec ai = 0.0;
            for(S32 j=0; j<nj;j++){
                F64vec xj = jp[j].pos;
                F64vec dr = xi - xj;
                F64 mj  = jp[j].mass;
                F64 dr2 = dr * dr + ep2;
                F64 dri = 1.0 / sqrt(dr2);
                ai -= (dri * dri * dri
                       * mj) * dr;
            }
            force[i].acc += ai;
        }
    }
};

template<class Tpsys>
void predict(Tpsys &p,
             const F64 dt) {
    S32 n = p.getNumberOfParticleLocal();
    for(S32 i = 0; i < n; i++)
        p[i].predict(dt);
}

template<class Tpsys>
void correct(Tpsys &p,
             const F64 dt) {
    S32 n = p.getNumberOfParticleLocal();
    for(S32 i = 0; i < n; i++)
        p[i].correct(dt);
}

template <class TDI, class TPS, class TTFF
void calcGravAllAndWriteBack(TDI &dinfo,
                             TPS &ptcl,
                             TTFF &tree) {
    dinfo.decomposeDomainAll(ptcl);
    ptcl.exchangeParticle(dinfo);
    tree.calcForceAllAndWriteBack
        (CalcGrav<Nbody>(),
         CalcGrav<SPJMonopole>(),
         ptcl, dinfo);
}

int main(int argc, char *argv[]) {
    F32 time  = 0.0;
    const F32 tend  = 10.0;
    const F32 dtime = 1.0 / 128.0;
    PS::Initialize(argc, argv);
    PS::DomainInfo dinfo;
    dinfo.initialize();
    PS::ParticleSystem<Nbody> ptcl;
    ptcl.initialize();
    PS::TreeForForceLong<Nbody, Nbody,
        Nbody>::Monopole grav;
    grav.initialize(0);
    ptcl.readParticleAscii(argv[1]);
    calcGravAllAndWriteBack(dinfo,
                            ptcl,
                            grav);
    while(time < tend) {
        predict(ptcl, dtime);
        calcGravAllAndWriteBack(dinfo,
                                ptcl,
                                grav);
        correct(ptcl, dtime);
        time += dtime;
    }
    PS::Finalize();
    return 0;
}
```

FDPSのインストール(ヘッダーファイルのインクルード)

粒子クラスの定義

相互作用関数の定義

メインルーチン

大規模並列N体コードが117行で書ける

FDPSを使ったプログラム例(粒子クラス)

```
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(F64 dt) {
        vel += (0.5 * dt) * acc;
        pos += dt * vel;
    }
    void correct(F64 dt) {
        vel += (0.5 * dt) * acc;
    }
};
```

- いくつかのメンバ関数は必須、関数名は固定。
  - getPos(): 粒子座標を返す
  - getCharge(): 質量や電荷を返す
  - copyFromFP(): 粒子クラス間でのコピー
  - copyFromForce(): 相互作用の結果をコピー
  - clear(): 相互作用の結果を消去
- これらのメンバ関数を使ってFDPSは粒子クラスにアクセスする。

FDPSを使ったプログラム例(相互作用関数)

```
template <class TPJ>
struct CalcGrav{
    void operator () (const Nbody * ip,
                      const S32 ni,
                      const TPJ * jp,
                      const S32 nj,
                      Nbody * force) {
        for(S32 i=0; i<ni; i++){
            F64vec xi  = ip[i].pos;
            F64    ep2 = ip[i].eps
                * ip[i].eps;
            F64vec ai = 0.0;
            for(S32 j=0; j<nj;j++){
                F64vec xj = jp[j].pos;
                F64vec dr = xi - xj;
                F64 mj  = jp[j].mass;
                F64 dr2 = dr * dr + ep2;
                F64 dri = 1.0 / sqrt(dr2);
                ai -= (dri * dri * dri
                       * mj) * dr;
            }
            force[i].acc += ai;
        }
    }
};
```

相互作用を受ける粒子のループ
(i粒子ループ)

相互作用を及ぼす粒子のループ
(j粒子ループ)

- ニュートン重力の場合
- 関数内で2重ループを書く
  - Barnesの方法を使うため。

FDPSを使ったプログラム例(メイン関数)

```
int main(int argc, char *argv[]) {
    F32 time  = 0.0;
    const F32 tend  = 10.0;
    const F32 dtime = 1.0 / 128.0;
    PS::Initialize(argc, argv);
    PS::DomainInfo dinfo;
    dinfo.initialize();
    PS::ParticleSystem<Nbody> ptcl;
    ptcl.initialize();
    PS::TreeForForceLong<Nbody, Nbody,
        Nbody>::Monopole grav;
    grav.initialize(0);
    ptcl.readParticleAscii(argv[1]);
    calcGravAllAndWriteBack(dinfo,
                            ptcl,
                            grav);
    while(time < tend) {
        predict(ptcl, dtime);
        calcGravAllAndWriteBack(dinfo,
                                ptcl,
                                grav);
        correct(ptcl, dtime);
        time += dtime;
    }
    PS::Finalize();
    return 0;
}
```

FDPSの初期化

領域分割インスタン

粒子交換インスタンス

相互作用計算
インスタンス

FDPSの終了処理

```
template <class TDI, class TPS, class TTFF>
void calcGravAllAndWriteBack(TDI &dinfo,
                             TPS &ptcl,
                             TTFF &tree) {
    dinfo.decomposeDomainAll(ptcl);
    ptcl.exchangeParticle(dinfo);
    tree.calcForceAllAndWriteBack
        (CalcGrav<Nbody>(),
         CalcGrav<SPJMonopole>(),
         ptcl, dinfo);
}
```

領域分割実行

粒子交換実行

相互作用計算実行

- 領域分割、粒子交換、相互作用計算に関するクラスのインスタンを作り、メンバ関数を呼び出す。
- 明示的にMPIを呼んでいない。

FDPSのリリースノート

- 2012年11月 FDPSの開発開始
- 2015年3月 FDPS Ver. 1.0
- 2016年1月 FDPS Ver. 2.0
  - アクセラレータ利用のために、Multiwalk法の実装
- 2016年12月 FDPS Ver. 3.0
  - Fortran Interfaceの実装
- 2017年11月 FDPS Ver. 4.0
  - SPH法やMD計算等で計算を高速化するために、相互作用リスト再利用のアルゴリズムの実装
- 2018年11月 FDPS Ver.5.0
  - C Interfaceの実装
- 2020年8月 FDPS Ver.6.0
  - PIKGの実装
- 2021年8月 FDPS Ver.7.0
  - 極座標でのツリー構築をサポートする機能の実装

構成

- FDPS開発の動機
- FDPSの概要
- アクセラレータ向け機能
  - 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
  - 性能、性能モデル
- 多言語インターフェース
- FDPSの次期バージョン
- まとめ

アクセラレータ対応機能開発の動機

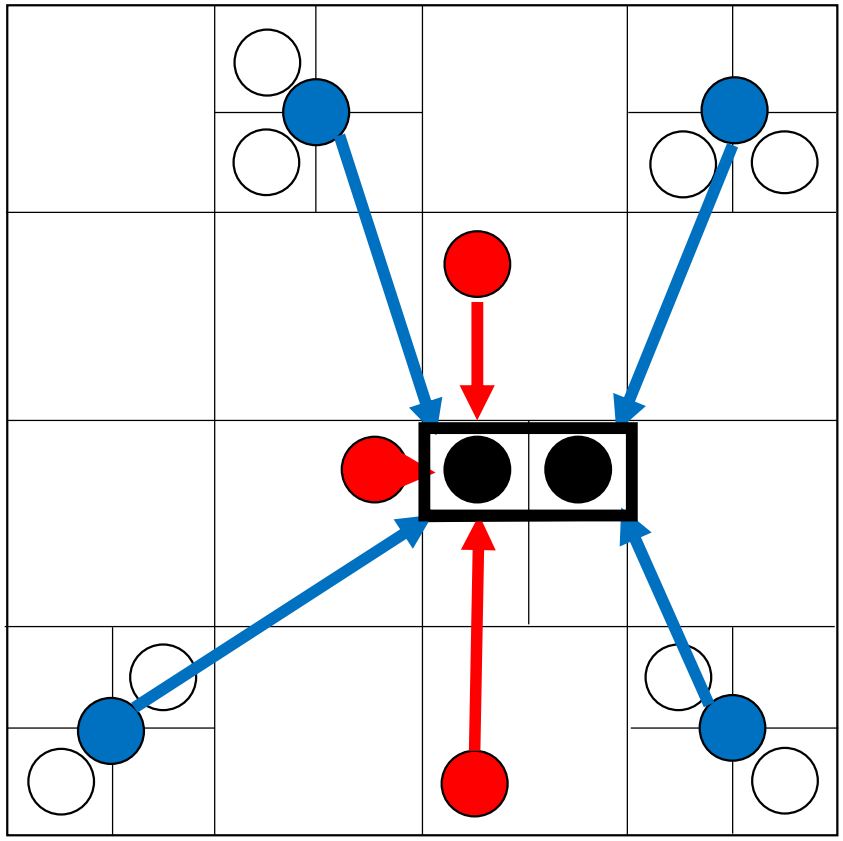
- FDPS Ver.1では主に「京」などの汎用計算機で性能が出るように設計されていた。
  - N体シミュレーションでは「京」のフルノードで実行効率50%.
- GPU等のアクセラレータを搭載した計算機が増えてきており、FDPSもこのような計算機で効率がでるようにしたい.
- 粒子シミュレーションをアクセラレータ搭載の計算機で行う場合主に二つの方法がある。
  - 全ての手順をアクセラレータに載せる。
  - 相互作用カーネルのみをアクセラレータで計算し、他の計算はホストコンピュータで行う。

  FDPSでは後者を選択。効率よくアクセラレータを使うために、以下の3つのアルゴリズムを導入。
    - マルチウォーク法
    - 粒子への間接アドレス参照
    - 相互作用リストの再利用

構成

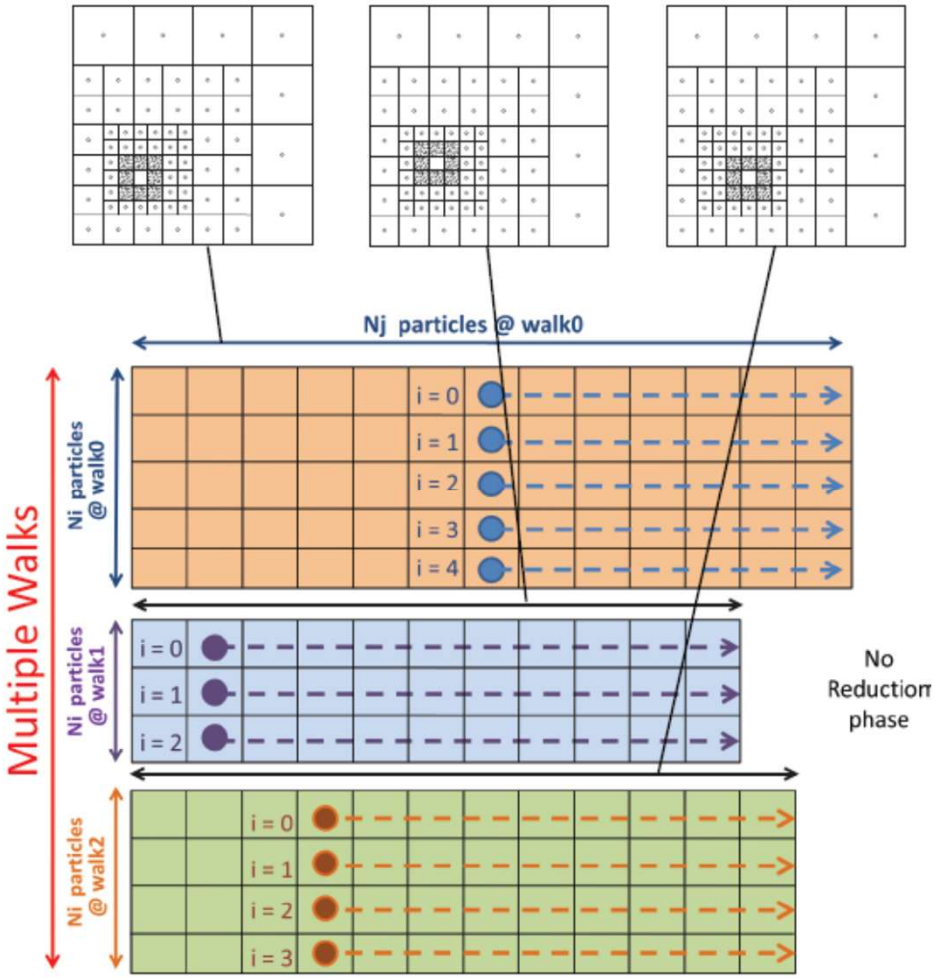
- FDPS開発の動機
- FDPSの概要
- アクセラレータ向け機能
  - マルチウォーク法
  - 粒子への間接アドレス指定
  - 相互作用リストの再利用
  - 性能、性能モデル
- 多言語インターフェース
- FDPSの次期バージョン
- まとめ

FDPS v1でアクセラレータを使った時の問題



- FDPS Ver.1ではアクセラレータの性能を十分に引き出せない。
  - 一つの粒子グループに対する相互作用リストを構築し相互作用を計算する。
    - 粒子グループの数が増えるとアクセラレータのカーネル起動のオーバーヘッドが無視できなくなる。
      - 相互作用リストを共有する粒子数は百程度。つまり、100万粒子の計算では、1万回カーネル起動する必要がある。
    - アクセラレータには大量の演算器が搭載されているため、使われない演算器がある。
      - 最新のGPUでは演算器は数千。

マルチウォーク法(Hamada+ 09)



- FDPS Ver.2 以降ではアクセラレータ向けにマルチウォーク法を実装
  - 複数の粒子グループに対して複数の相互作用リストを構築しそれらをまとめてアクセラレータで計算する。
    - カーネル起動のオーバーヘッドは無視できる。
    - 一度のカーネル発行で大量の演算を行うため、全ての演算器を使うことが可能。

構成

- FDPS開発の動機
- FDPSの概要
- アクセラレータ向け機能
  - 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
  - 性能、性能モデル
- 多言語インターフェース
- FDPSの次期バージョン
- まとめ

粒子への間接アドレス指定

- アクセラレータで相互作用を計算する簡単な方法は、相互作用計算の度に粒子の物理情報をアクセラレータに送り計算。
  - しかし、同じ粒子が異なる相互作用リスト内にも存在する。大雑把に同じ粒子を10回以上送る必要がある。
  - ホスト-アクセラレータ間の通信が問題になる。
- そこで、最初にすべてのローカルにある粒子(+LET+ツリーの多重極モーメント)をアクセラレータに送ってしまい、相互作用計算時には相互作用リストとしてアクセラレータ上の粒子のインデックスだけを送る。
  - e.g 重力の場合　16Byte->4Byte
  - SPHの場合 50Byte->4Byte

構成

- FDPS開発の動機
- FDPSの概要
- アクセラレータ向け機能
  - 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
  - 性能、性能モデル
- 多言語インターフェース
- FDPSの次期バージョン
- まとめ

相互作用リストの再利用

- SPH等の流体計算やMDシミュレーション等では粒子が時間刻みあたりに移動する距離が短い。
- 同じ相互作用リストを複数時間刻みの間使い続けることができる。
- N体シミュレーションでも惑星形成や惑星系リングのシミュレーションでは粒子の相対位置はほとんど変化しないので、リストの再利用が可能。

相互作用計算の手順

リスト構築ステップ

1. ローカルツリーを構築する。
2. LETを構築し交換する。この時LETとその送信先を記憶する。
3. グローバルツリーを構築する。
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. 相互作用の結果を用いて粒子の物理量を更新。

リスト再利用ステップ

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. 相互作用の結果を用いて粒子の物理量を更新。

ローカルツリー構築(O(N))、グローバルツリー構築(O(N))、LET構築(O(PlogN))、相互作用リスト(O(N/<ni>logN))の構築を省くことができる。

構成

- FDPS開発の動機
- FDPSの概要
- アクセラレータ向け機能
  - 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
  - 性能、性能モデル
- 多言語インターフェース
- FDPSの次期バージョン
- まとめ

性能（シングルノード）

N=4M

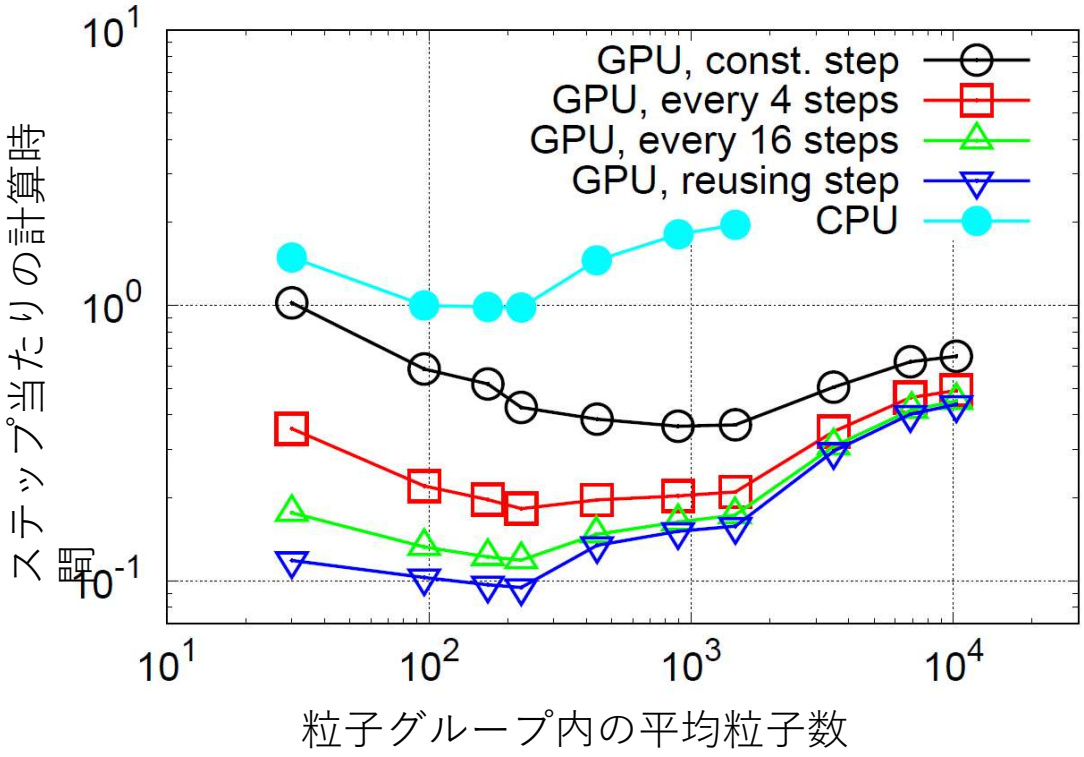
計算機

- Xeon E5-2670 v3
  - 883Gflops (単精度)
  - 68GB/s
- Titan V
  - 13.8Tflops (単精度)
  - 883GB/s
- 転送
  - PCIe3 15.75GB/s

16ステップリストを使いまわすことで、CPUのみの計算より8.5倍程度高速に動作

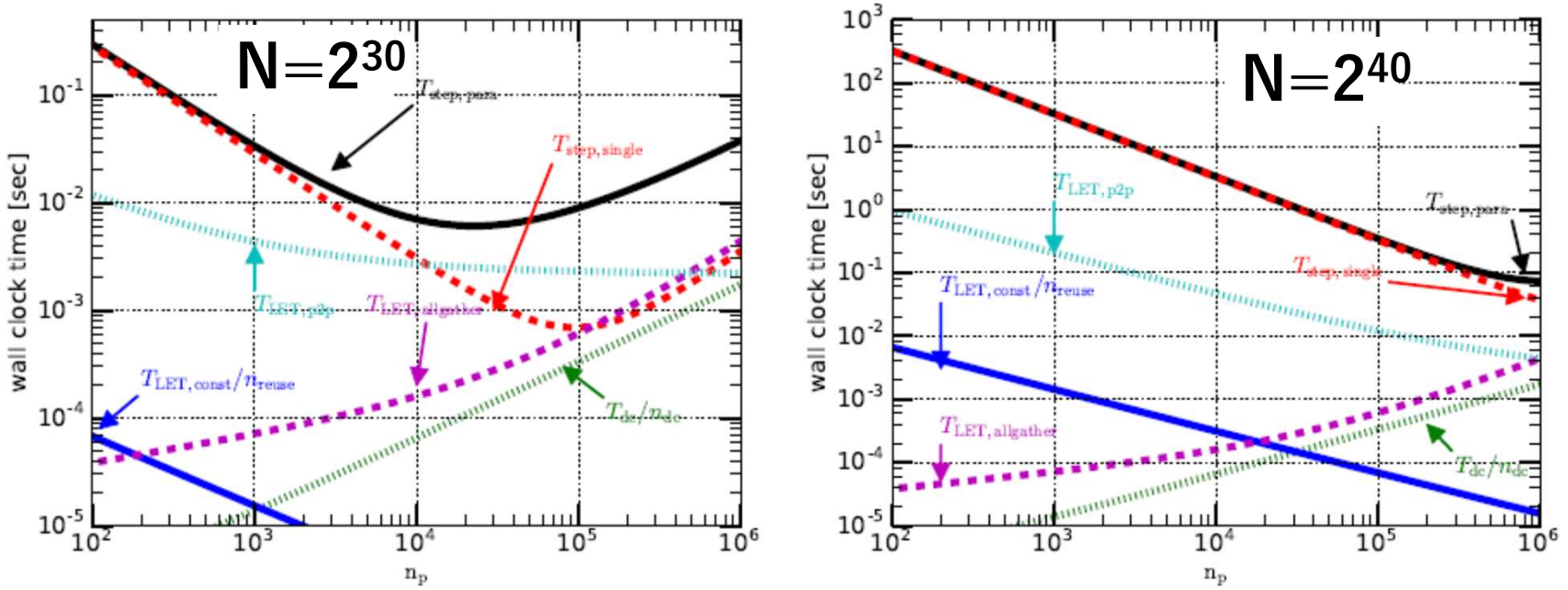
CPUのみ： 432Gflops(49%)

CPU+GPU: 3.7Tflops(27%)



性能モデル

- 並列の場合のパフォーマンスモデルも構築。
  - ネットワークは「京」と同じものを想定。ストロングスケーリングの性能



- N=2^{30}で数千プロセス、N=2^{40}で10^6プロセスまでスケール。

構成

- FDPS開発の動機
- FDPSの概要
- アクセラレータ向け機能
  - 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
  - 性能、性能モデル
- 多言語インターフェース
- FDPSの次期バージョン
- まとめ

多言語インターフェースの開発動機

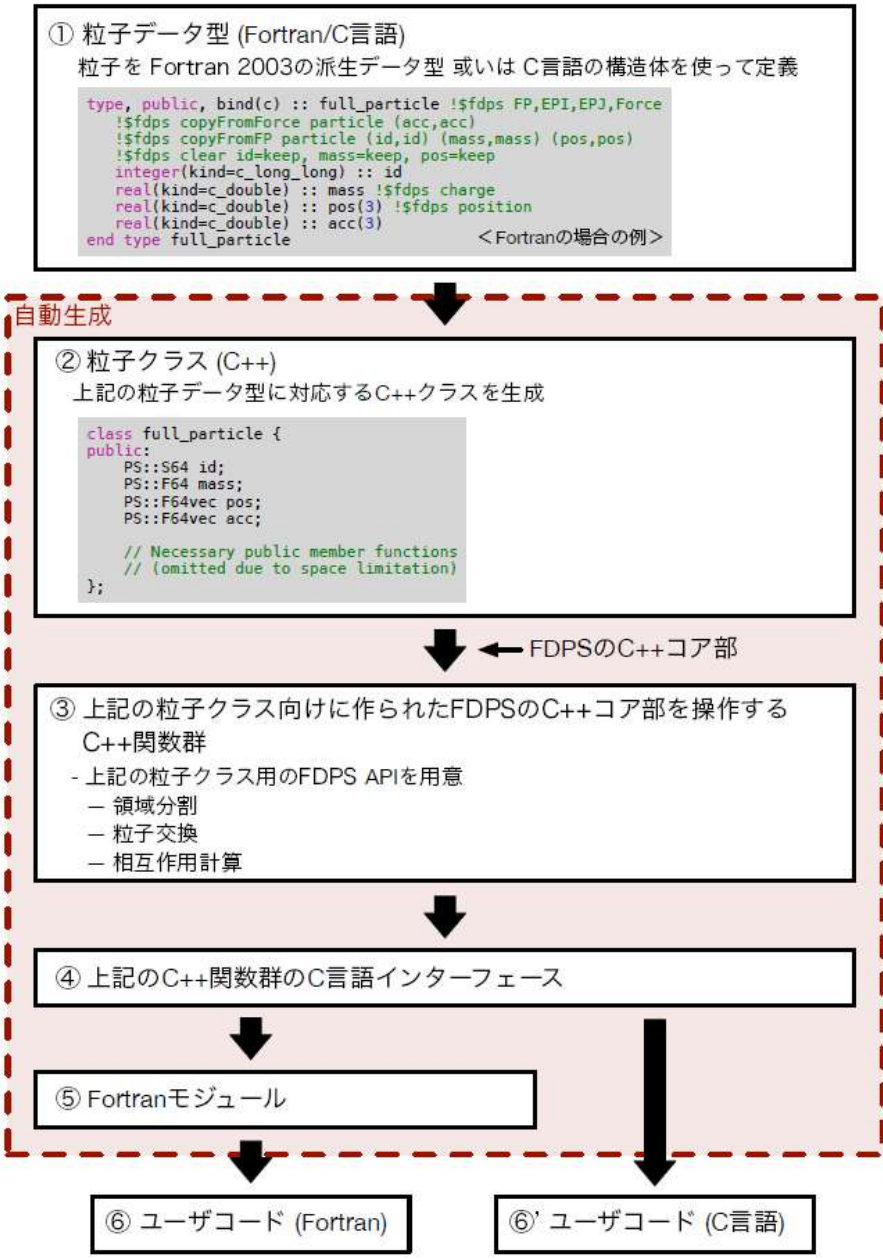
- スパコン利用者にはC言語やFortranユーザーも多い。
  - これらのユーザーがFDPSを利用するためには、C++を習得する必要がある。
  - 今までの、CやFortranの資産を生かせない。

- C/FortranからFDPSを使うには
  - FDPSのテンプレート関数をあらかじめ型を指定したC++の“普通”の関数に変換する必要がある。
  - FDPSでは粒子クラスに幾つかのメンバ関数を必要とするため、C/Fortranの粒子構造体/派生データ型からC++のクラスを生成する必要がある。

これらに対応するC++コードをC/Fortranコードから自動生成するインターフェースを開発。

インターフェースの仕組み

- ユーザーは構造体もしくは派生データ型により粒子データを定義.
  - FDPSが必要とする情報をFDPS指示文を用いて明示する.
- ユーザー定義の粒子データ構造から、同等のC++のクラスを自動生成.
- FDPSのテンプレート引数を上記の粒子データ型に置き換えたC++の関数群を自動生成.
- C++の関数群をCから使うためのインターフェスを生成.
- Fortranを使う場合は、Fortran用のインターフェースを自動生成.



インターフェースの利用例（粒子構造体）

```
1   #pragma once
2   #include "FDPS_c_if.h"
3   typedef struct full_particle { //$fdps FP,EPI,EPJ,Force
4       //$fdps copyFromForce full_particle (pot,pot) (acc,acc)
5       //$fdps copyFromFP full_particle (id,id) (mass,mass) (eps,eps)
        ↪   (pos,pos)
6       //$fdps clear id=keep, mass=keep, eps=keep, pos=keep, vel=keep
7       long long id; // $fdps id
8       double  mass; //$fdps charge
9       double  eps;
10      fdps_f64vec pos; //$fdps position
11      fdps_f64vec vel;
12      double  pot;
13      fdps_f64vec acc;
14  } Full_particle;
```

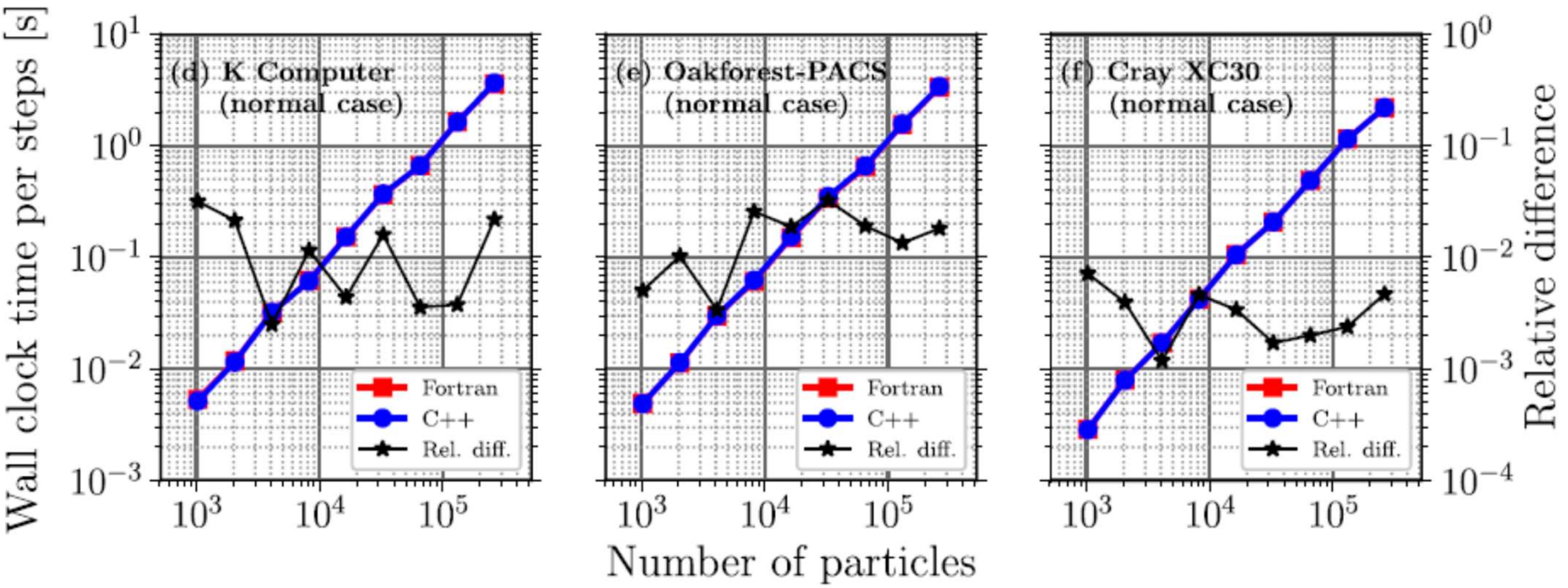
- FDPS指示文(//$fdpsで始まる文)を用いて、FDPSが必要とするメンバ関数(位置座標を返す、データをコピーする等)を自動生成するための情報を与える。

インターフェースの利用例(メイン関数)

```
int c_main()
{
    // FDPS の初期化
    fdps_initialize();
    // 領域情報オブジェクトの生成と初期化
    int dinfo_num;
    float coef_ema=0.3;
    fdps_create_dinfo(&dinfo_num);
    fdps_init_dinfo(dinfo_num,coef_ema);
    // 粒子群オブジェクトの生成と初期化
    int psys_num;
    fdps_create_psys(&psys_num,"full_particle");
    fdps_init_psys(psys_num);
    // ツリーオブジェクトの生成と初期化
    int tree_num;
    fdps_create_tree(&tree_num,
                     "Long,full_particle,full_particle,full_particle,Monopole");
    int ntot=1024;
    double theta = 0.5;
    int n_leaf_limit = 8;
    int n_group_limit = 64;
    fdps_init_tree(tree_num, ntot, theta, n_leaf_limit, n_group_limit);

    // 初期条件作成 (省略)

    // 領域分割と粒子交換
    fdps_decompose_domain_all(dinfo_num,psys_num,-1.0);
    fdps_exchange_particle(psys_num,dinfo_num);
    // 相互作用計算
    fdps_calc_force_all_and_write_back(tree_num,
                                       calc_gravity_ep_ep,
                                       calc_gravity_ep_sp,
                                       psys_num,
                                       dinfo_num,
                                       true,
                                       FDPS_MAKE_LIST);
    // 粒子の軌道の時間積分
    double time_sys  = 0;
    double time_end = 10.0;
    double dt = 1.0/128.0;
    while (time_sys <= time_end){
        kick(psys_num,0.5*dt);
        time_sys +=  dt;
        drift(psys_num,dt);
        // 領域分割と粒子交換
        fdps_decompose_domain_all(dinfo_num,psys_num,-1.0);
        fdps_exchange_particle(psys_num,dinfo_num);
        // 相互作用計算
        fdps_calc_force_all_and_write_back(tree_num,
                                           calc_gravity_ep_ep,
                                           calc_gravity_ep_sp,
                                           psys_num,
                                           dinfo_num,
                                           true,
                                           FDPS_MAKE_LIST);
        kick(psys_num,0.5*dt);
    }
    fdps_finalize();
    return 0;
}
```

- メイン関数の名前はc_main
  - 本当のメイン関数はC++で生成されており、その中でc_mainを呼ぶ。
- ほぼ、C++のメイン関数と同じ。

多言語インターフェースを用いた時の性能



- 多言語インターフェースを用いたプログラムとC++を用いたプログラムでの性能差はほとんどない。

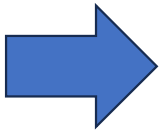
PIKG(Ver.6)

- カーネルの最適化はアプリケーションユーザー側の責任
  - サンプルには最適化したカーネルをつけておいた.
- PIKGを開発
  - Partile-particle Interaction Kernel Generator
  - 最適化されたカーネルを生成するDSL(コンバータ)
  - PIKGで書かれたコードはSIMD化したコードに変換してくれる.
    - AVX2, AVX512, SVE, CUDA
  - AoS, SoA変換
  - 演算は対応した組み込み関数に変換
  - 分岐などはマスク処理

重力N体

```
EPI F32vec xi:pos

EPJ F32vec xj:pos
EPJ F32    mj:mass

FORCE F32vec acc:acc
FORCE F32    pot:pot

F32 eps2

rij    = xi - xj
r2 = rij * rij + eps2
r_inv  = rsqrt(r2)
r2_inv = r_inv * r_inv
mr_inv  = mj * r_inv
mr3_inv = r2_inv * mr_inv
acc -= mr3_inv * rij
pot -= mr_inv
```

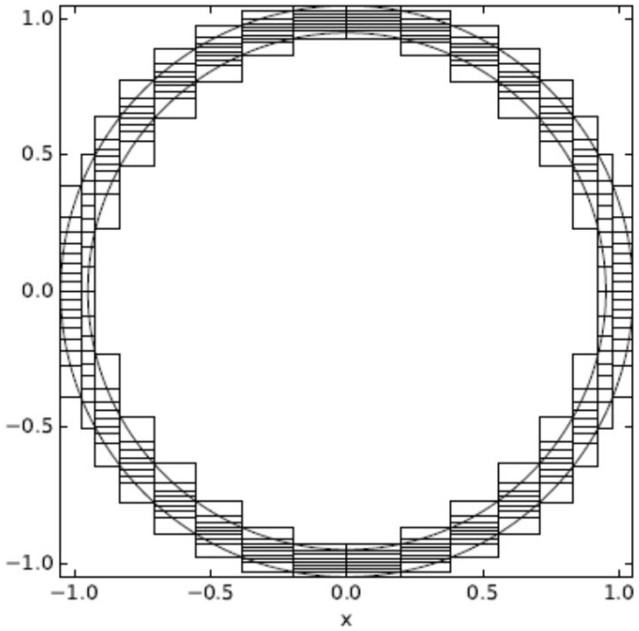
変換

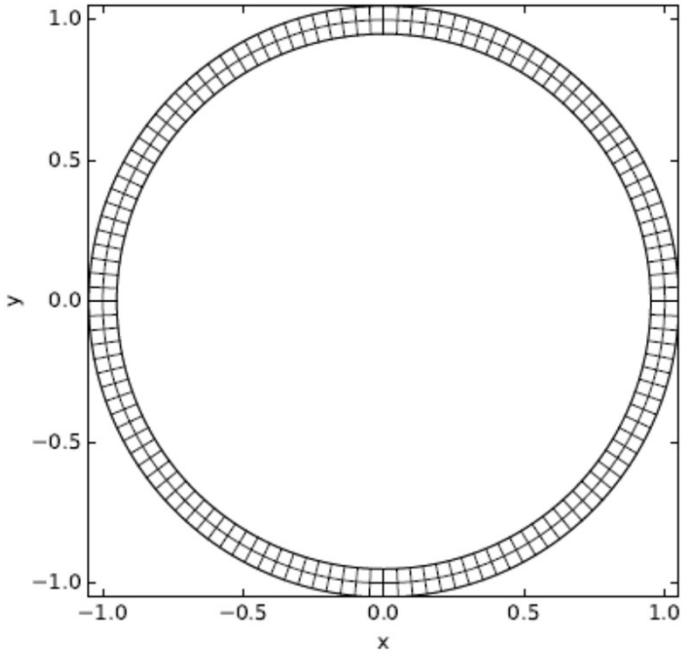


```
void Kernel_I1_J8(const Epi* __restrict__ epi,const PIKG::S32 ni,const Epj* __restrict__ epj,const
    PIKG::S32 i;
    PIKG::S32 j;
    for(i = 0;i < (ni/1)*1;++i){
        __m256x3 xi;
        xi.v0 = _mm256_set1_ps(epi[i+0].pos.x);
        xi.v1 = _mm256_set1_ps(epi[i+0].pos.y);
        xi.v2 = _mm256_set1_ps(epi[i+0].pos.z);
        __m256x3 acc;
        acc.v0 = _mm256_set1_ps(0.0f);
        acc.v1 = _mm256_set1_ps(0.0f);
        acc.v2 = _mm256_set1_ps(0.0f);
        __m256 pot;
        pot = _mm256_set1_ps(0.0f);
        for(j = 0;j < (nj/8)*8;j += 8){
            __m256 mj;
            alignas(32) int index_gather_load7[8] = {0,4,8,12,16,20,24,28};
            __m256i vindex_gather_load7 = _mm256_load_si256((const __m256i*)index_gather_load7);
            mj = _mm256_i32gather_ps(((float*)&epj[j].mass),vindex_gather_load7,4);
            __m256x3 xj;
            alignas(32) int index_gather_load8[8] = {0,4,8,12,16,20,24,28};
            __m256i vindex_gather_load8 = _mm256_load_si256((const __m256i*)index_gather_load8);
            xj.v0 = _mm256_i32gather_ps(((float*)&epj[j].pos.x),vindex_gather_load8,4);
            alignas(32) int index_gather_load9[8] = {0,4,8,12,16,20,24,28};
            __m256i vindex_gather_load9 = _mm256_load_si256((const __m256i*)index_gather_load9);
            xj.v1 = _mm256_i32gather_ps(((float*)&epj[j].pos.y),vindex_gather_load9,4);
            alignas(32) int index_gather_load10[8] = {0,4,8,12,16,20,24,28};
            __m256i vindex_gather_load10 = _mm256_load_si256((const __m256i*)index_gather_load10);
            xj.v2 = _mm256_i32gather_ps(((float*)&epj[j].pos.z),vindex_gather_load10,4);

            rij.v0 = _mm256_sub_ps(xi.v0,xj.v0);
            rij.v1 = _mm256_sub_ps(xi.v1,xj.v1);
            rij.v2 = _mm256_sub_ps(xi.v2,xj.v2);
            __fkg_tmp1 = _mm256_fmadd_ps(rij.v0,rij.v0,_mm256_set1_ps(eps2));
            __fkg_tmp0 = _mm256_fmadd_ps(rij.v1,rij.v1,__fkg_tmp1);
            r2 = _mm256_fmadd_ps(rij.v2,rij.v2,__fkg_tmp0);
            r_inv = rsqrt(r2);
            r2_inv = _mm256_mul_ps(r_inv,r_inv);
            mr_inv = _mm256_mul_ps(mj,r_inv);
            mr3_inv = _mm256_mul_ps(r2_inv,mr_inv);
            acc.v0 = _mm256_fnmadd_ps(mr3_inv,rij.v0,acc.v0);
            acc.v1 = _mm256_fnmadd_ps(mr3_inv,rij.v1,acc.v1);
            acc.v2 = _mm256_fnmadd_ps(mr3_inv,rij.v2,acc.v2);
            pot = _mm256_sub_ps(pot,mr_inv);
        } // loop of j
```

曲座標の導入(Ver. 8)

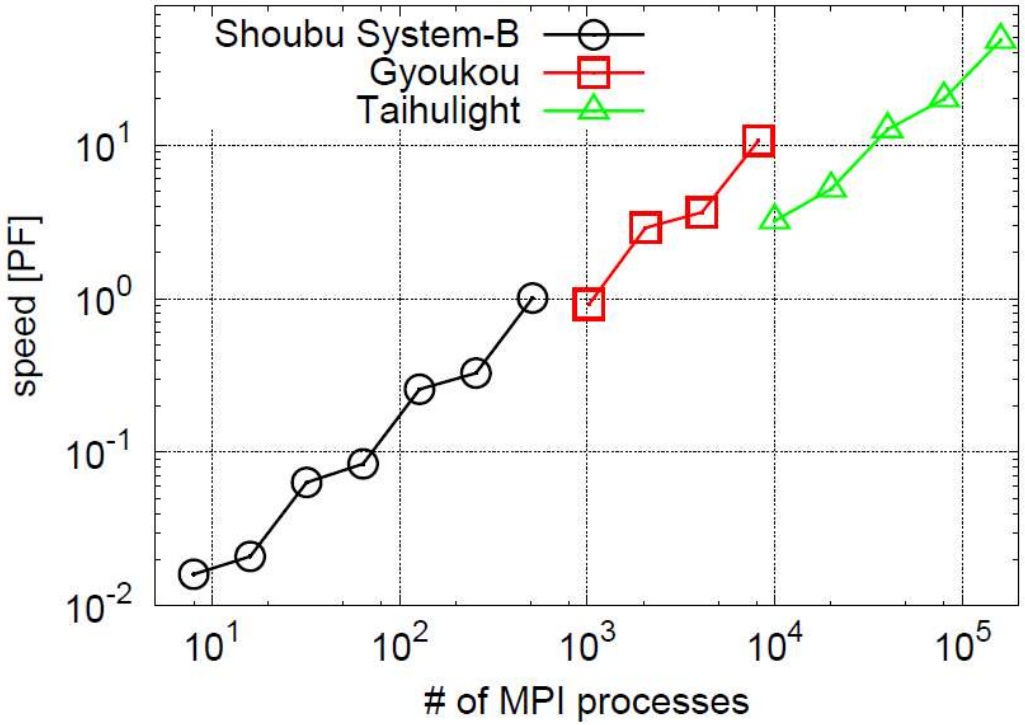
- リングシミュレーション用に極座標を導入





惑星系リングシミュレーションの性能

- 相互作用リストの再利用，極座標などのあらゆる機能を使っての性能測定
- N=10^7/proc
- 計算機
  - Shoubu System-B (PEZY-SC2)
  - Gyoukou (PEZY-SC2)
  - TaihuLight (Snuway 26010)
    - PEXY-SC2: 5.6Tflops(単精度)、76.8GB/s
    - Sunway 26010 754Glops(倍精度)、34GB/s



Shoubu System-B 1.01Pflops(35.5%) @ 512proc
Gyoukou 10.6Pflops(23.5%) @ 8192 proc
TaihuLight 47.9Pflops (40.0%) @ 160000 proc

構成

- FDPS開発の動機
- FDPSの概要
- アクセラレータ向け機能
  - 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
  - 性能、性能モデル
- 多言語インターフェース
- FDPSの次期バージョン
- まとめ

FDPSの今後

- Ver.8ではParticle-Mesh Multipole Methodを導入予定
  - FMMとPMの中間的スキーム
  - PMほど大規模なFFTを必要としない
  - FMMほど計算量が大きくない
  - MDや宇宙論シミュレーション等で有効

FDPSの今後

- Ver.9ではGPU対応を強化する.
  - 銀河や星団等の自己重力多体系は粒子の速度分散が大きく、相互作用リストの再利用は困難.
  - 木構築から相互作用カーネル計算までをGPUで計算させる.
    - 最初にGPUに相互作用計算に必要な粒子の情報(質量、位置等)を送り、後はGPU内で計算

まとめ

- FDPSは現在も新しい機能を追加しながら開発中
- Ver.2 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
  - GPU、PEZY-SC2、Sunway 26010で高い実行効率を達成。
- Ver.3 Ver.5では多言語インターフェースを追加
  - FDPSをC/Fortranから使うことができる。
  - C/Fortranで書かれた粒子データ構造をc++のクラスにするコードジェネレータ

  原理的にはCの関数を呼び出すことができる言語なら、FDPSが使える。
- 最適化されたカーネルを生成するDSL, PIKGをVer6で追加
- 極座標をVer7で追加
- 今後は，PMMMの追加，GPU対応の強化を行っていく．