

粒子シミュレーションと粒子シミュ レータ開発フレームワークFDPS

岩澤全規 @松江高専
2021/07/17 千葉工大 核物理×物性セミナー

発表構成

- 数値シミュレーション
- FDPS
 - FDPSの概要
 - FDPSの最近の機能
 - アクセラレータ対応
 - 多言語インターフェース

発表構成

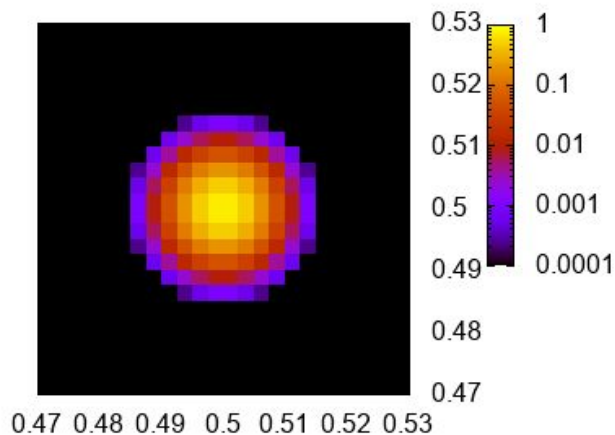
- 数値シミュレーション
- FDPS
 - FDPSの概要
 - FDPSの最近の機能
 - アクセラレータ対応
 - 多言語インターフェース

数値シミュレーション

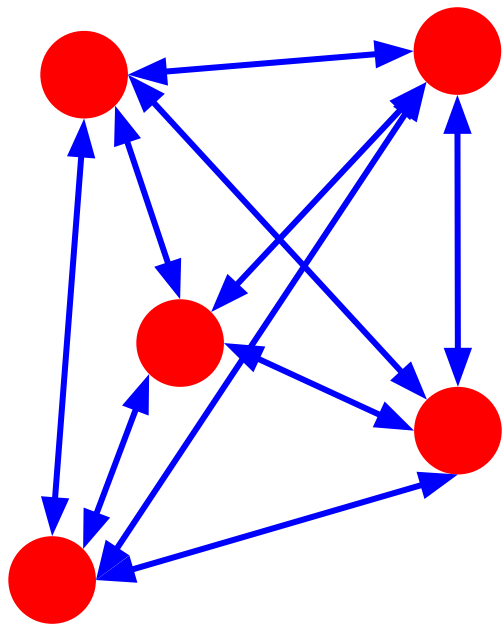
- 数値シミュレーション: 計算機中で対象とする系を数値的に表現し模擬実験
 - 理論、実験に続く第三の科学と呼ばれたりする。
- 実験の代わり
 - 時間短縮、コスト削減
 - 飛行機、自動車、創薬、防災など
 - 実験ができない現象の研究
 - e.g. 天体现象: 銀河形成、星形成、超新星爆発など
- 数値シミュレーション手法は大きく二つに分けられる
 - メッシュ法
 - 粒子法

メッシュ法

対象とする系を格子状に切り、物理量を各格子に持たせることで離散化し、各格子に発展方程式を適用する事でシミュレーションする。



粒子法



対象とする系を相互作用を及ぼしあう粒子の集団として表現する。

- 分布関数から粒子をサンプリングし離散化を行う。
 - 天文や分子動力学では星1つや分子1つを1粒子として扱う事もある
1. 各粒子が他の粒子から受ける力を計算
 - 遠距離力(e.g. 重力): 全粒子から力を受ける
 - 近距離力(e.g. 分子間力): 近傍の粒子のみから力を受ける。
 2. 各粒子の受ける力を数値積分し、 Δt 後の粒子の位置と速度を求める
 3. 1に戻る

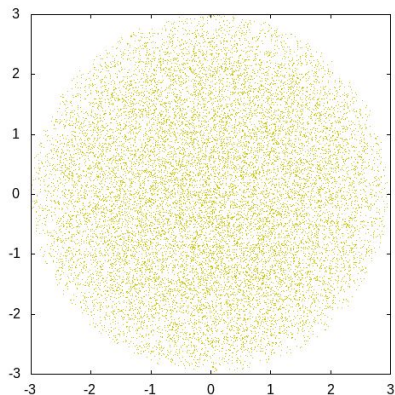
粒子法

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。

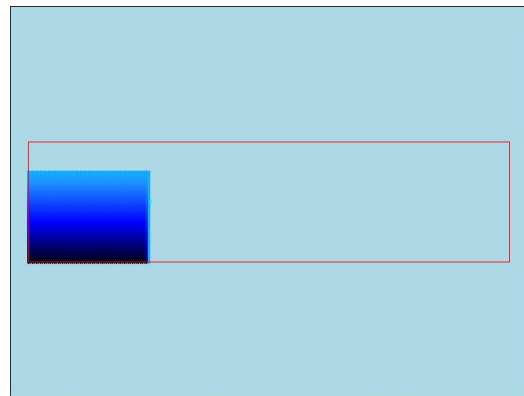
科学、工学の幅広い分野で使われている。

重力N体、MD、個別要素法、SPH、MPS、メッシュフリー法等。

コールドコロプス(重力N体シミュレーション)



ダムブレイク(メッシュフリー法)



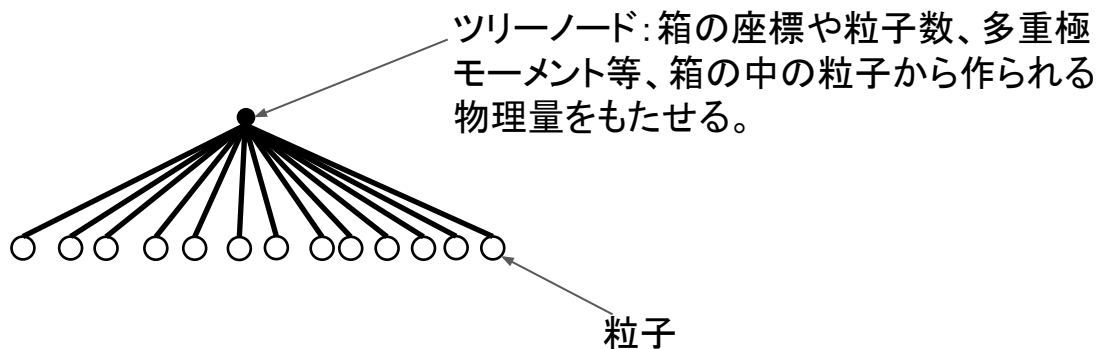
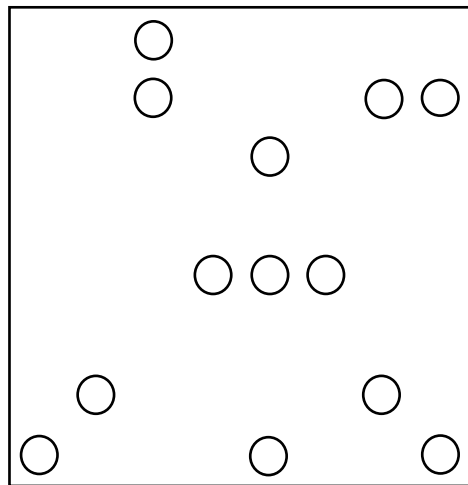
木構造を用いた粒子法

- 一般に粒子法の計算量は大きい。
- 重力やクーロン力等の遠距離力では、粒子がその他全ての粒子と相互作用するので相互作用の計算量は $O(N^2)$
- 分子間力等の実行的に相互作用の到達距離が短い場合も粒子同士の距離を計算する必要があるので $O(N^2)$
- 粒子の位置を木構造を用いて管理することで効率的に計算

ツリー構造の構築

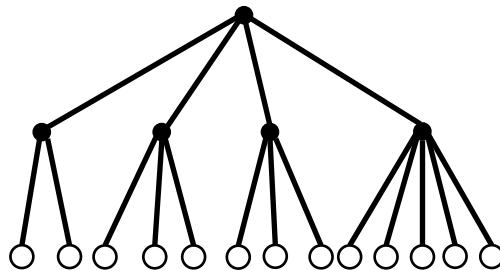
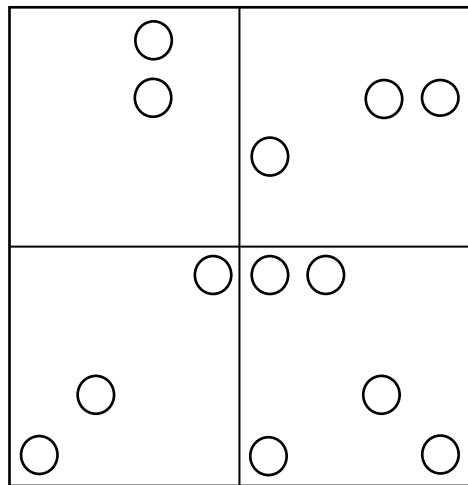
ある領域に粒子が存在し、この領域を囲む箱を考える。

箱の中の粒子数がある数以下(ここでは1個)になるまで箱を分割していく。

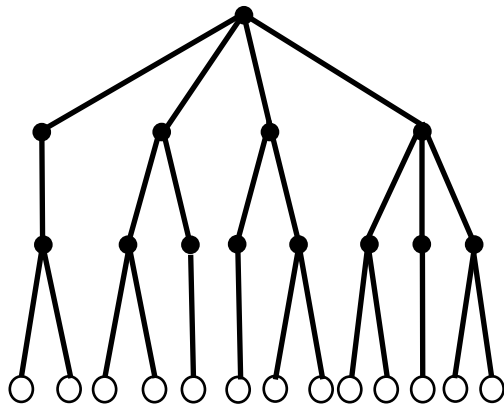
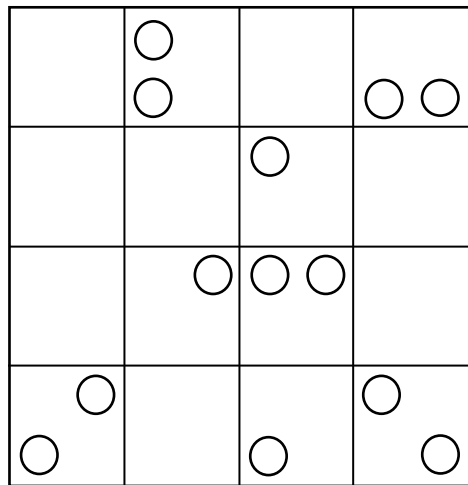


ツリー構造の構築

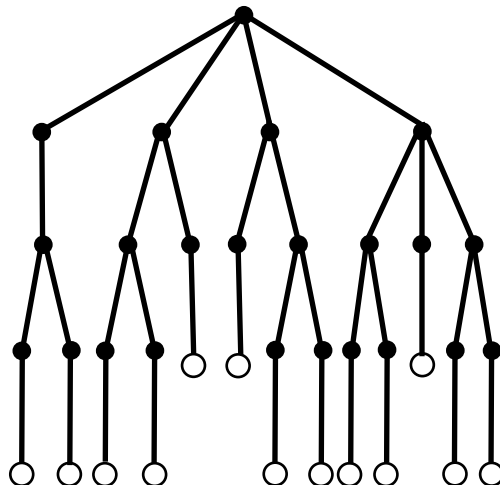
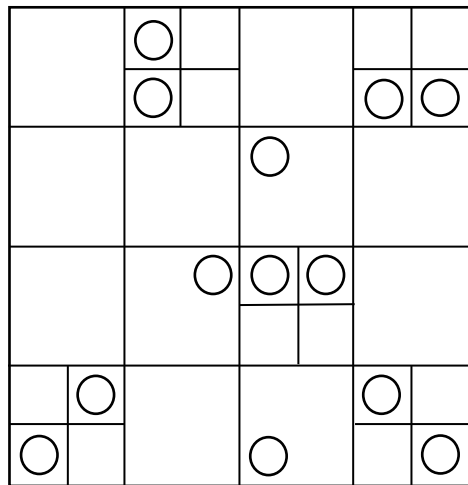
2次元なので4分木を使う。(3次元なら8分木)



ツリー構造の構築



ツリー構造の構築



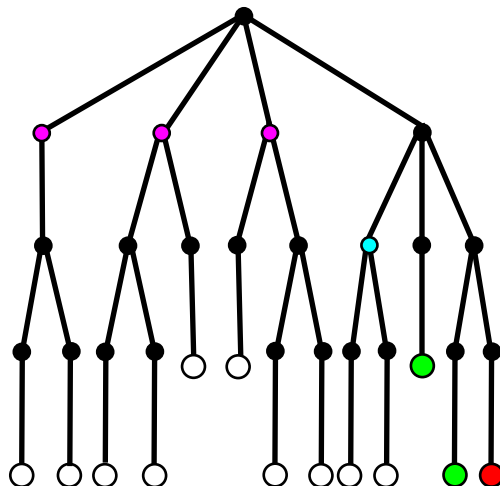
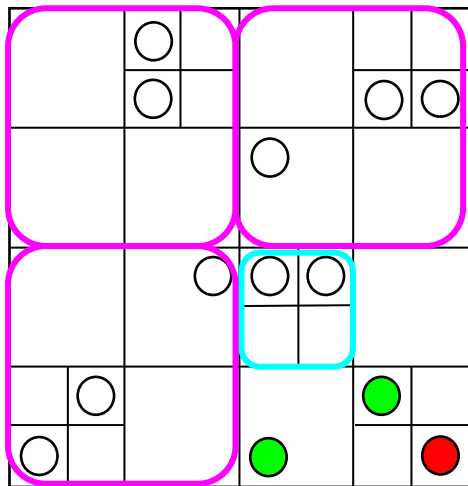
ツリー構造構築後に、各粒子の持つ物理量からボトムアップで各ノードが持つべき物理量や位置座標を計算する。

長距離力の計算

- ツリーをたどり、ツリーセルが力を計算したい粒子から十分遠ければ、そのセル内に入っている粒子の多重極モーメントを使って相互作用を計算。近ければ更に深く探索する。

[Barnes-Hut Tree code (1986)]

- 十分遠いかどうかの判定は、ツリーセルの見込み角を使う。
 $R > l/\theta$ (R :セルと粒子の距離、 l :セルの長さ、 θ :見込み角)



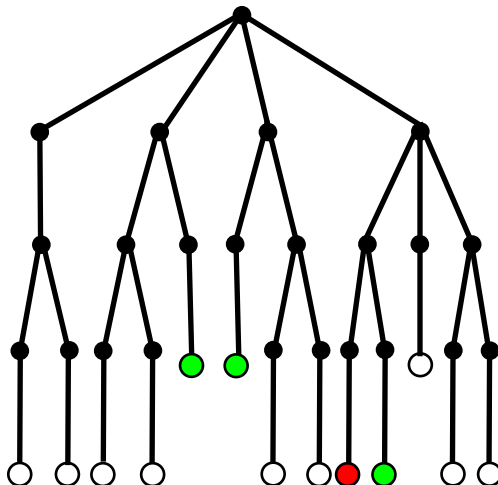
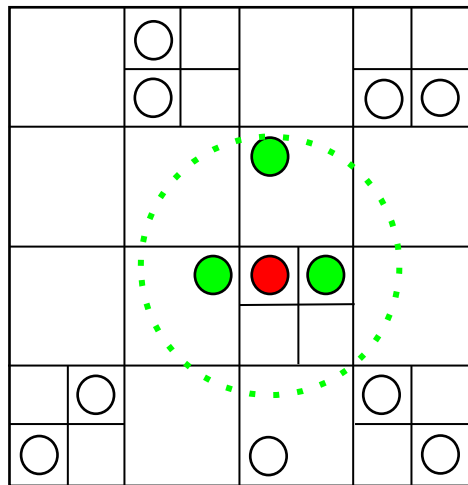
赤い粒子への力を計算

- 近傍の粒子からの力は直接計算
- 遠くの粒子からの力はセルの多重極モーメントを使う

相互作用の計算量は $O(N \log(N))$

短距離力の計算

ツリーをたどり、ツリーセルが力を計算したい粒子から十分遠ければ、ツリーをたどるのをやめる。近ければ更に深く探索する。



赤い粒子への力を計算

- 近傍の粒子のみから力を計算

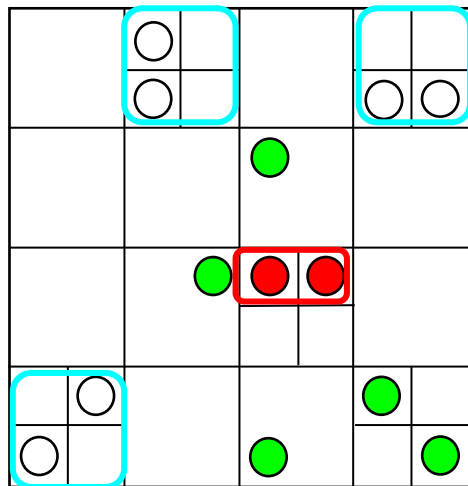
相互作用の計算量は $O(NN_{\text{ngh}})$

Barnesの方法(Barnes 1991)

複数の粒子に対してツリーを辿り相互作用を計算する。

ツリーを辿る回数が減る。

近傍の粒子が同じ粒子から相互作用を受けるので、並列化が可能



ツリーを辿る回数は減るが、相互作用の計算量自体は増えるので、計算機に依存した最適な(まとめる)粒子数が存在。

発表構成

- 数値シミュレーション
- FDPS
 - FDPSの概要
 - FDPSの最近の機能
 - アクセラレータ対応
 - 多言語インターフェース

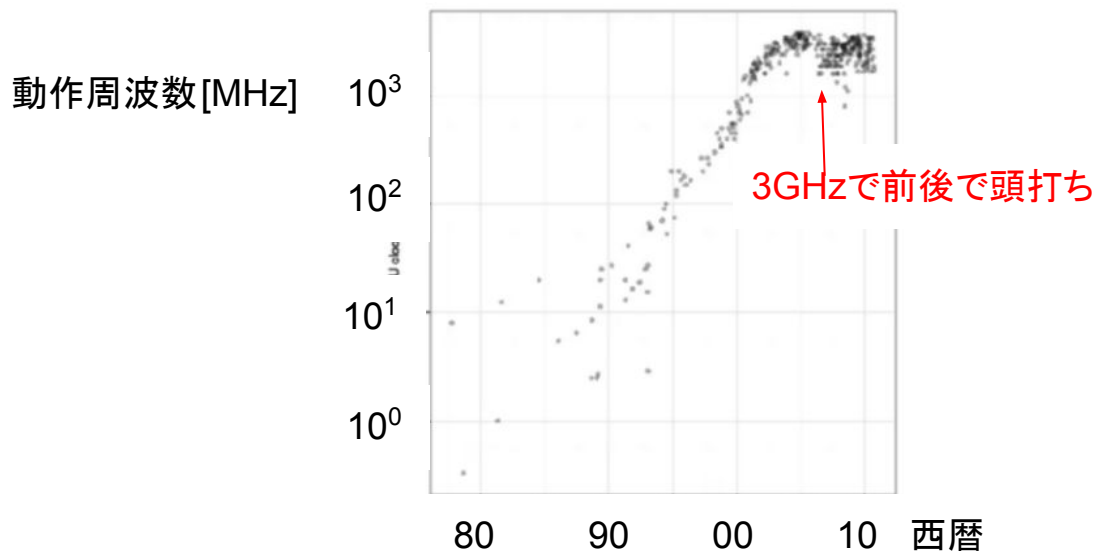
FDPSとは

Framework for Developing Particle Simulatorsの略

並列粒子シミュレーションコード開発のためのフレームワーク

並列粒子シミュレーションの必要性

- CPUの動作周波数は2000年前半に頭打ち。大規模な粒子シミュレーションを行うためには並列化が必須
 - プロセス並列、スレッド並列、SIMD化



大規模並列粒子シミュレーション

- 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。
- 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。
- 粒子法の並列化アルゴリズムはあまり問題に依存しない。

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？

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

The diagram shows the equation $\frac{d\vec{u}_i}{dt} = \vec{g} \left(\sum_j^N \vec{f}(\vec{u}_i, \vec{u}_j), \vec{u}_i \right)$. Red arrows point from three text boxes to parts of the equation: one to $\vec{g}$, one to $\vec{f}$, and one to $\vec{u}_i$ inside the summation.

相互作用関数

物理量をその時間微分に変換する関数

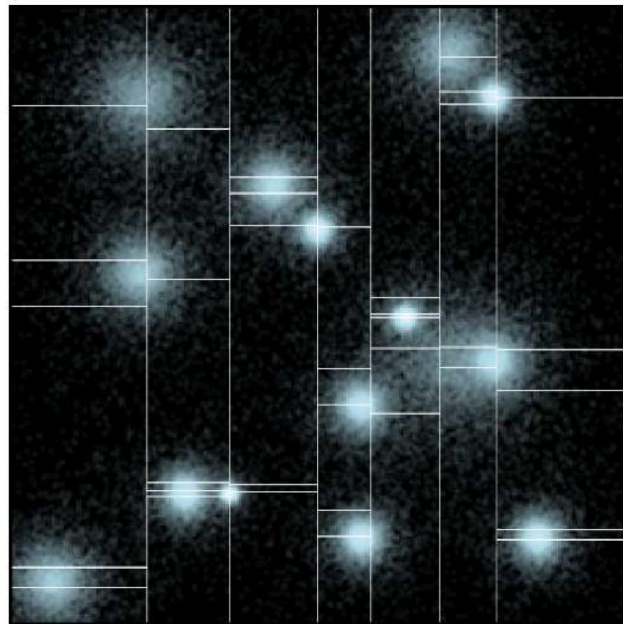
粒子データ

- 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
 - 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. ツリー構造を用いた粒子の管理とツリーを用いた相互作用リスト作成
 - 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

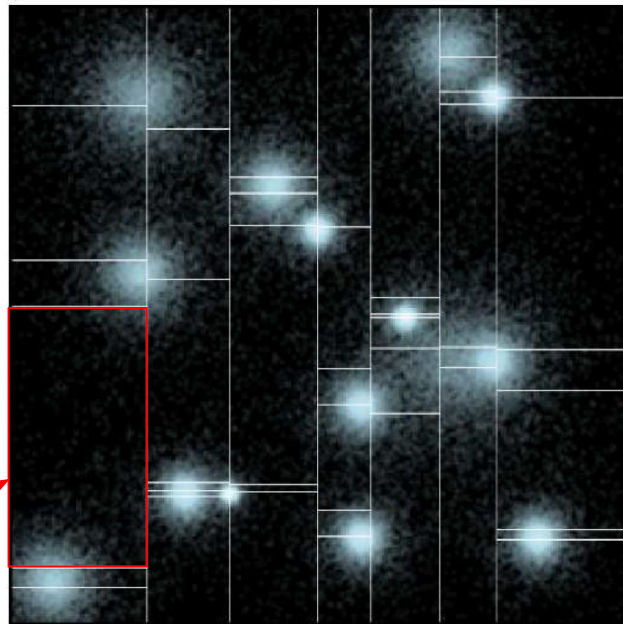
- 粒子の位置情報のみ必要。

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. ツリー構造を用いた粒子の管理とツリーを用いた相互作用リスト作成

- 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。



一つの領域を一つの計算ノードが担当

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

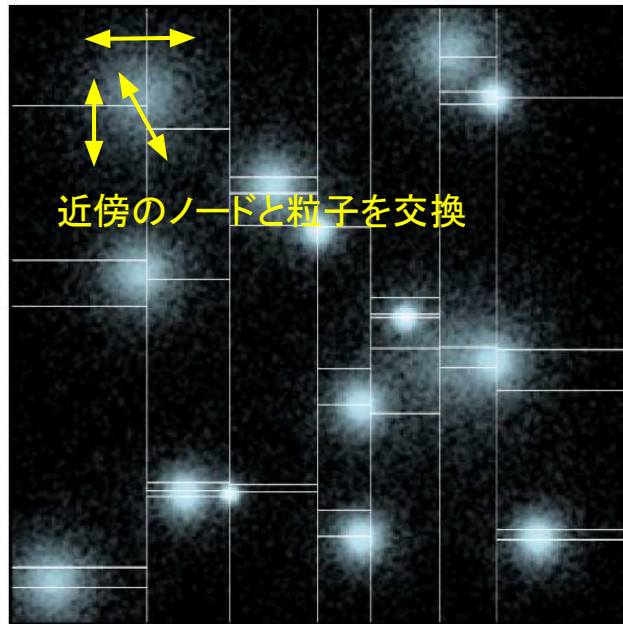
- 粒子の位置情報のみ必要。

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. ツリー構造を用いた粒子の管理とツリーを用いた相互作用リスト作成

- 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

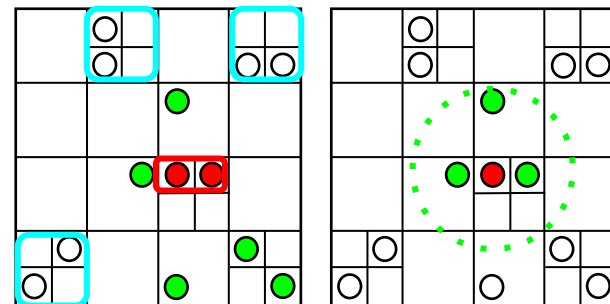
- 粒子の位置情報のみ必要。

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. ツリー構造を用いた粒子の管理とツリーを用いた相互作用リスト作成

- 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。



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。

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。

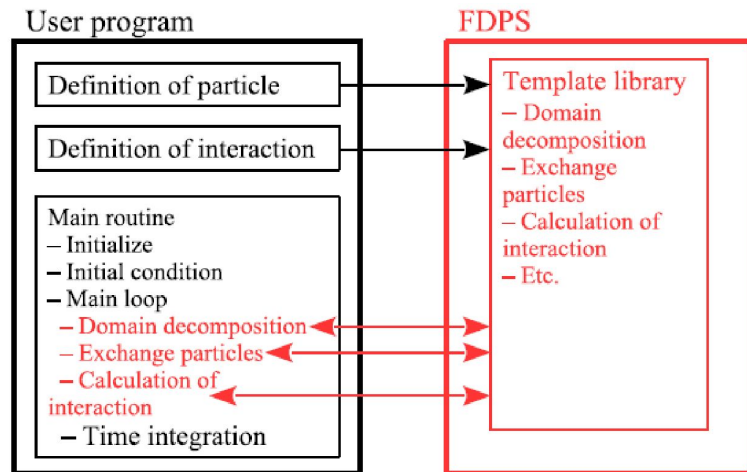
- 内部実装の言語は C++

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
- FDPS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のインスタンスを作り、メンバ関数を呼び出すことでそれぞれの処理を行う

プログラムの例(粒子データ定義)

```
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で関数名は固定。
 - 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。

プログラムの例(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 (i粒子)のループ

相互作用を及ぼす粒子 (j粒子)のループ

ニュートン重力の場合

関数内を2重ループで記述

プログラムの例(メイン関数)

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

    while(time < tend) {
        predict(ptcl, dtype);
        calcGravAllAndWriteBack(dinfo,
                               ptcl,
                               grav);

        correct(ptcl, dtype);
        time += dtype;
    }
    PS::Finalize();
    return 0;
}
```

FDPSの初期化

領域分割クラス

粒子情報クラス

相互作用クラス

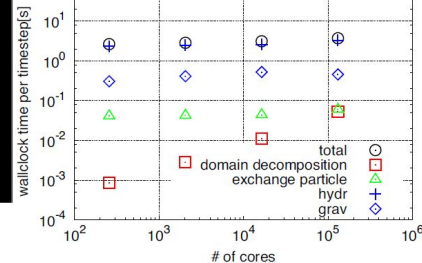
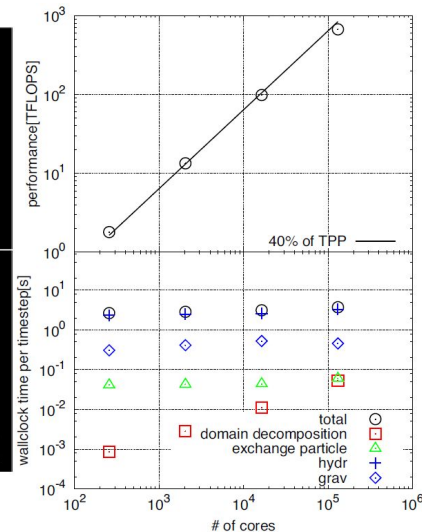
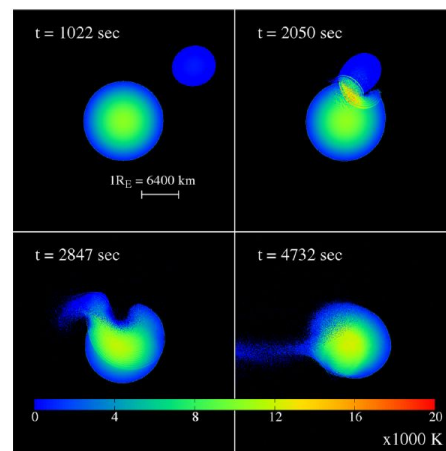
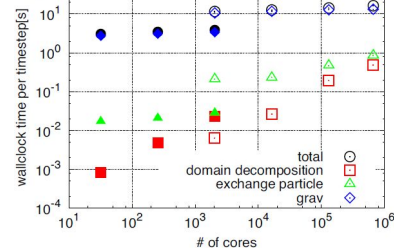
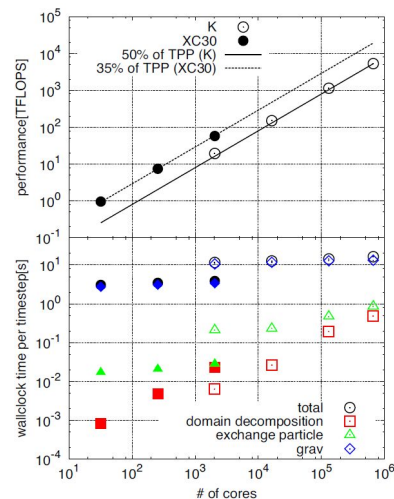
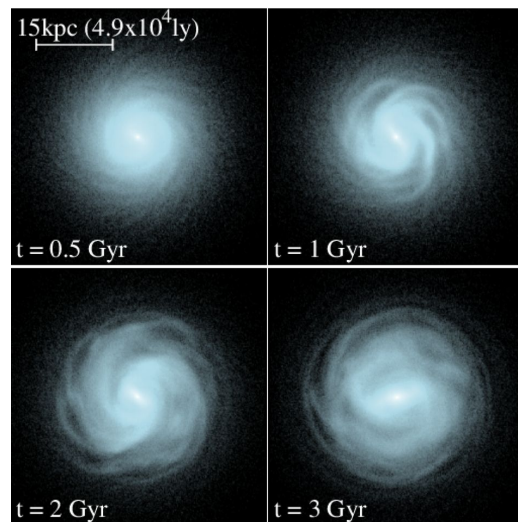
```
template <class TDI, class TPS, class TTFF>
void calcGravAllAndWriteBack(TDI &dinfo,
                             TPS &ptcl,
                             TTFF &tree) {
    dinfo.decomposeDomainAll(ptcl);
    ptcl.exchangeParticle(dinfo);
    tree.calcForceAllAndWriteBack
        (CalcGrav<Nbody>(),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

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

性能(京コンピュータ)

重力N体型さんでは効率50%

SPHでは効率40%



発表構成

- 数値シミュレーション
- FDPS
 - FDPSの概要
 - **FDPSの最近の機能**
 - アクセラレータ対応
 - 多言語インターフェース

FDPSの開発史

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実装。

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。相互作用リスト再利用機能を実装。

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。

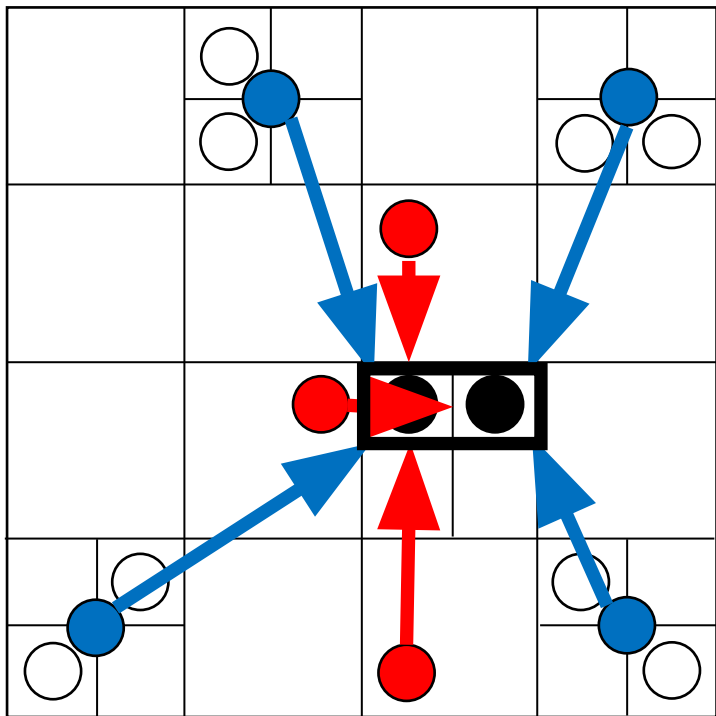
2020年8月 Ver.6をリリース。最適化された相互作用カーネルを生成するコードジェネレータPIKGを追加。

2021年7月末 Ver.7をリリース予定。

発表構成

- 数値シミュレーション
- FDPS
 - FDPSの概要
 - FDPSの最近の機能
 - アクセラレータ対応
 - 多言語インターフェース

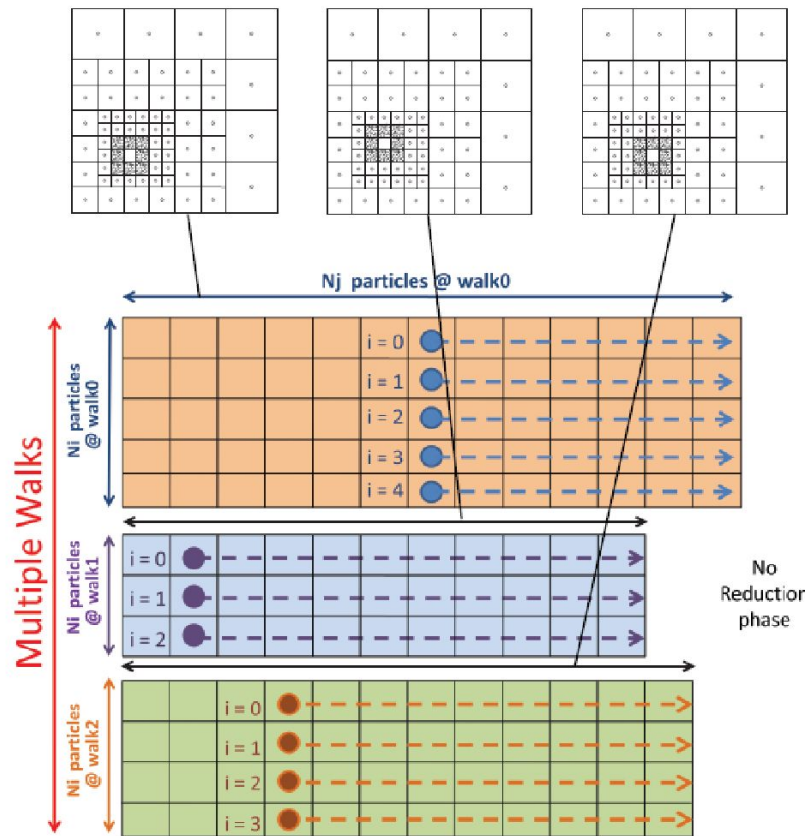
FDPS v1でアクセラレータを使った時の問題



- 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。

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

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($O(P \log N)$)、相互作用リスト($O(N/\langle n \rangle \log N)$)の構築を省くことができる。

性能(シングルノード)

N=4Mのcold collapse シミュレーション。

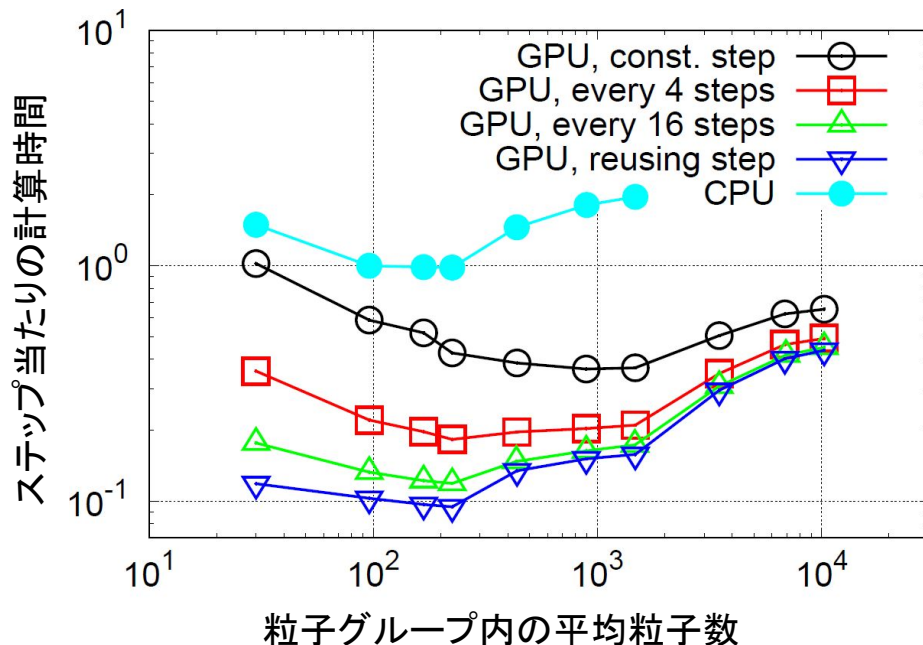
計算機

- 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

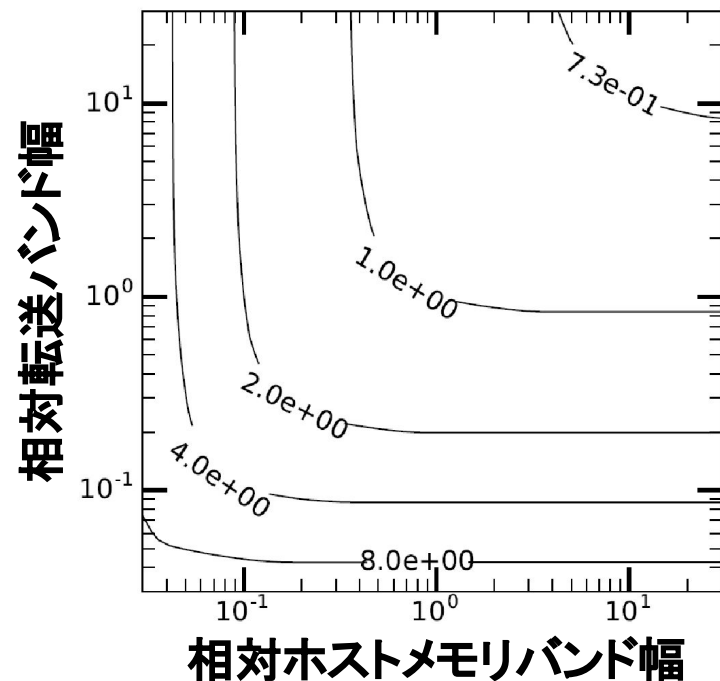
$$\begin{aligned}
 T_{\text{step,single}} &\sim \frac{1}{n_{\text{reuse}}} (T_{\text{step,const}} + (n_{\text{reuse}} - 1)T_{\text{step,reuse}}) \\
 &\sim 68(n + n_{\text{LET}})/B_{\text{host}} + (174n + 58n_{\text{LET}}) \max(42/B_{\text{host}}, 16/B_{\text{transfer}}) \\
 &\quad + n \max\left(n_{\text{list}}(39.1/F_{\text{GPU}} + 440/(\langle n_i \rangle B_{\text{GPU}})), 19.2/B_{\text{transfer}}, 282/B_{\text{host}}\right) \\
 &\quad + \frac{1}{n_{\text{reuse}}} \left[(1580n + 1157n_{\text{LET}} \right. \\
 &\quad + 54.4((n/4)\log_2(n/4) + ((2n + n_{\text{LET}})/4)\log_2((2n + n_{\text{LET}})/4)))/B_{\text{host}} \\
 &\quad + n \left(\max\left(n_{\text{list}}(39.1/F_{\text{GPU}} + 440/(\langle n_i \rangle B_{\text{GPU}})), (12 + 3.4n_{\text{list}}/\langle n_i \rangle)/B_{\text{transfer}}, \right. \right. \\
 &\quad \left. \left. 19.2/B_{\text{transfer}}, (282 + 80n_{\text{list}}/\langle n_i \rangle)/B_{\text{host}}\right), \right. \\
 &\quad \left. \left. - \max\left(n_{\text{list}}(39.1/F_{\text{GPU}} + 440/(\langle n_i \rangle B_{\text{GPU}})), 19.2/B_{\text{transfer}}, 282/B_{\text{host}}\right) \right) \right] (33)
 \end{aligned}$$

● 実測した計算時間をもとにパフォーマンスモデルを構築。

- B_{host} : ホストのメモリバンド幅
- B_{transfer} : 転送バンド幅
- F_{GPU} : アクセラレータの演算速度
- B_{GPU} : アクセラレータのメモリバンド幅
- n : 粒子数
- $\langle n_i \rangle$: 平均 i 粒子数
- n_{list} : 相互作用リスト長

性能モデル

相対計算時間



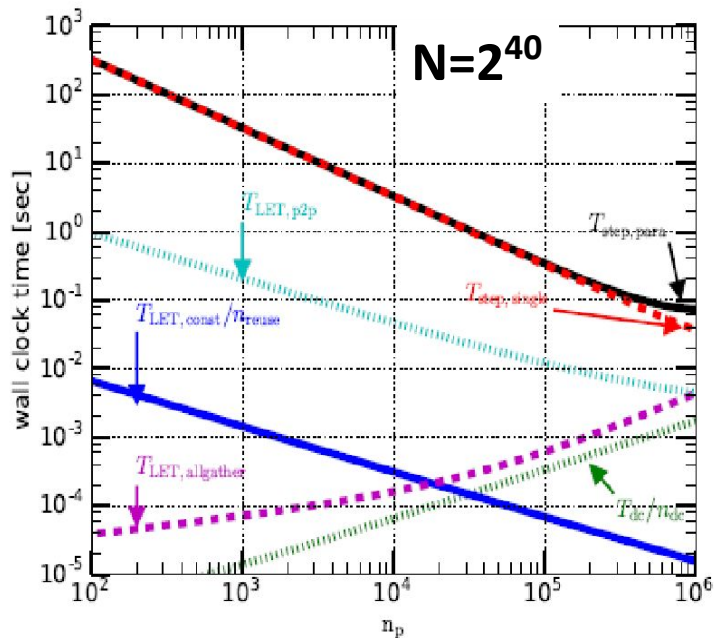
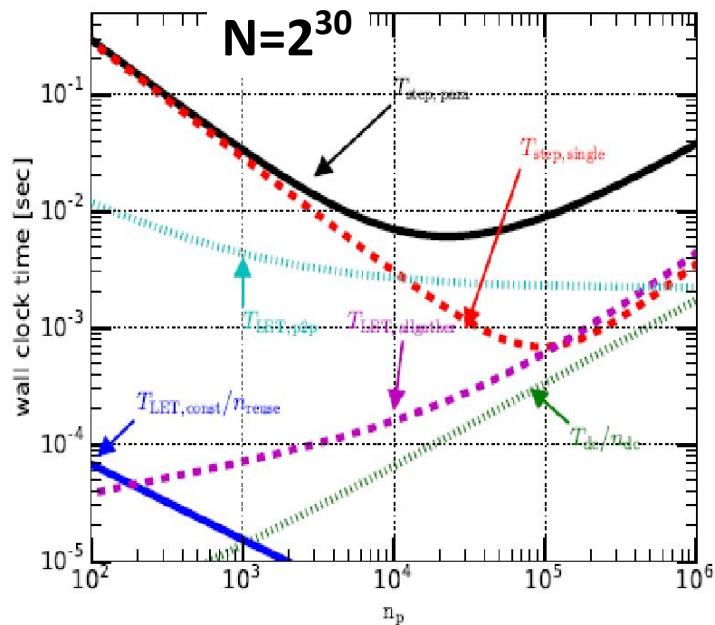
● リファレンスモデル

- アクセラレータ
 - 演算速度: 10Tflps
 - メモリバンド幅: 500GB/s
 - ホスト
 - メモリバンド幅: 100GB/s
 - 転送バンド幅: 10GB/s
- 提案したアルゴリズムを用いる事で、計算速度は (ホスト、転送) バンド幅によって律速されない。
- バンド幅据え置きで、アクセラレータの性能が 10倍上がると、計算時間は 1/4。

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



性能

● 惑星系リングシミュレーションでの弱スケールでの性能

○ $N=10^7/\text{proc}$

● 計算機

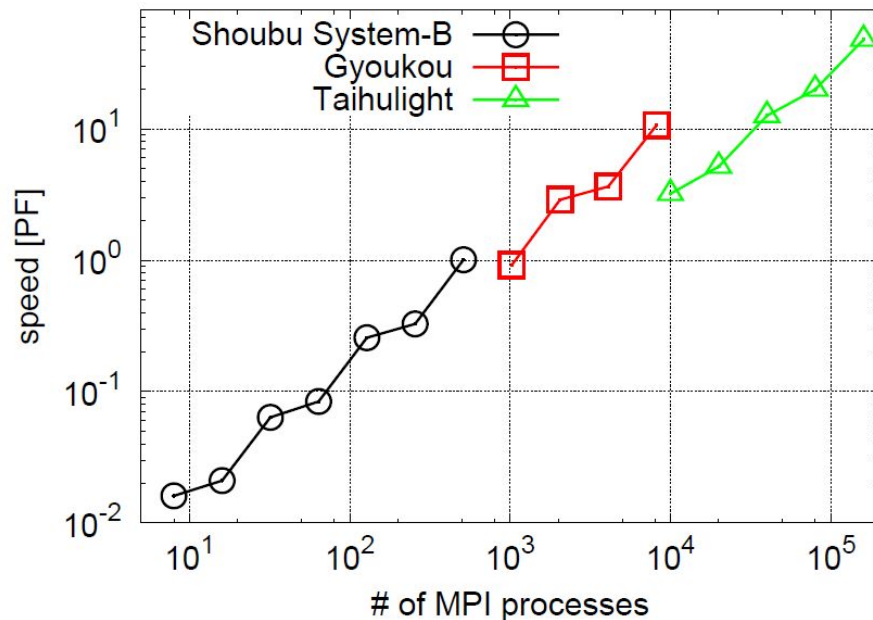
○ Shoubu System-B (PEZY-SC2)

○ 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

発表構成

- 数値シミュレーション
- FDPS
 - FDPSの概要
 - FDPSの最近の機能
 - アクセラレータ対応
 - 多言語インターフェース

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ースを生成。
- Fortranを使う場合は、Fortran用のインターフェースを自動生成。

① 粒子データ型 (Fortran/C言語)

粒子を Fortran 2003 の派生データ型 或いは C 言語の構造体を使って定義

```
type, public, bind(c) :: full_particle !$fdps FP,EPI,EPJ,Force
!$fdps copyFromForce particle (acc,acc)
!$fdps copyFromFP particle (id,id) (mass,mass) (pos,pos)
!$fdps clear id=keep, mass=keep, pos=keep
integer(kind=c_long_long) :: id
real(kind=c_double) :: mass !$fdps charge
real(kind=c_double) :: pos(3) !$fdps position
real(kind=c_double) :: acc(3)
end type full_particle
```

< Fortran の場合の例 >

自動生成

② 粒子クラス (C++)

上記の粒子データ型に対応するC++クラスを生成

```
class full_particle {
public:
    PS::S64 id;
    PS::F64 mass;
    PS::F64vec pos;
    PS::F64vec acc;

    // Necessary public member functions
    // (omitted due to space limitation)
};
```

← FDPSのC++コア部

③ 上記の粒子クラス向けに作られたFDPSのC++コア部を操作する

C++関数群

- 上記の粒子クラス用のFDPS APIを用意
 - 領域分割
 - 粒子交換
 - 相互作用計算

④ 上記のC++関数群のC言語インターフェース

⑤ Fortranモジュール

⑥ ユーザコード (Fortran)

⑥ ユーザコード (C言語)

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      //$fdps copyFromFP full_particle (id,id) (mass,mass) (eps,eps)
6      ↪ (pos,pos)
7      //$fdps clear id=keep, mass=keep, eps=keep, pos=keep, vel=keep
8      long long id; // $fdps id
9      double mass; //$fdps charge
10     double eps;
11     fdps_f64vec pos; //$fdps position
12     fdps_f64vec vel;
13     double pot;
14     fdps_f64vec acc;
15 } 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。

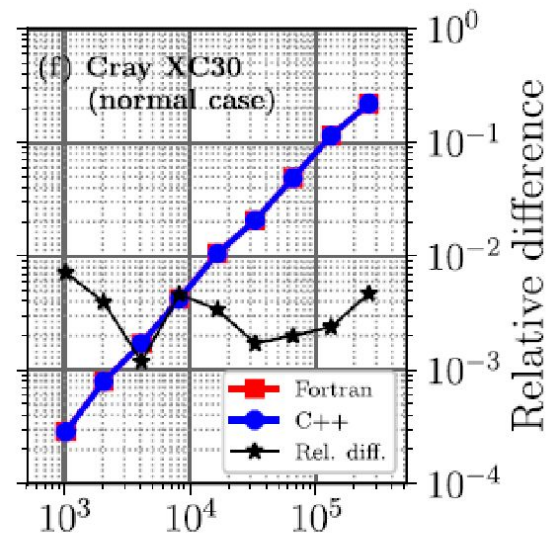
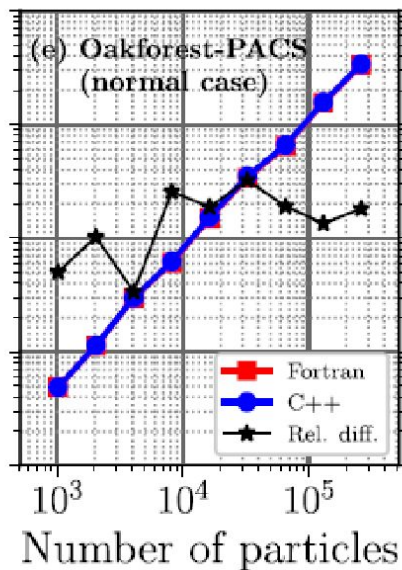
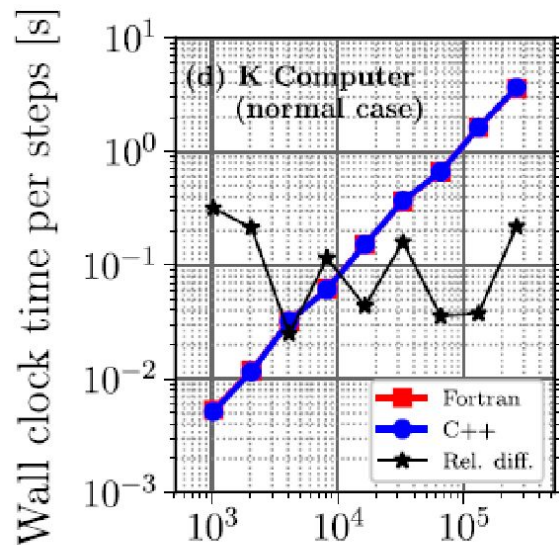
インターフェースの利用例(メイン関数)

```
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(cinfo_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 }
```

- 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。

まとめ

- 粒子シミュレーション
 - 効率のよい計算のために木構造が用いられる。
- FDPS
 - 数百行で並列粒子シミュレーションコードの開発が可能
 - 並列化効率は非常に高い。
 - アクセラレータにも対応
 - C言語、Fortranのインターフェースもある。
 - Cの関数を呼び出す事ができる言語なら、FDPSを使うことができる
- FDPSはMITライセンスで公開中
 - <https://github.com/FDPS/FDPS>