## FDPSの概要説明

岩澤全規

理化学研究所計算科学研究センター 粒子系シミュレータ研究チーム フラッグシップ2020プロジェクト コデザイン推進チーム 2018/08/06 FDPS講習会

#### FDPSとは

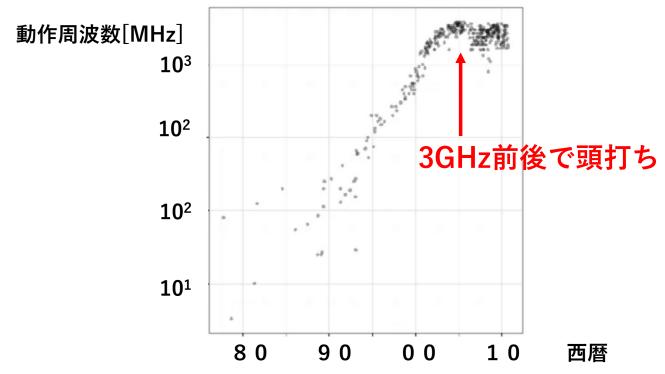
- Framework for Developing Particle Simulator
- 大規模並列シミュレーションコードの開発を支援するフレーム ワーク
- 重力N体、SPH、分子動力学、粉体、etc···
- 支配方程式

変換する関数

$$rac{dec{u}_i}{dt}=ec{g}igg(\sum_j^N ec{f}ec{u}_iec{u}_jec{v}_jec{u}_iec{v}_jec{v}_jec{v}_iec{v}_jec{v}_iec{v}_jec{v}_iec{v}_jec{v}_iec{v}_jec{v}_iec{v}_iec{v}_jec{v}_iev_iec{v}_iec{v}_iec{v}_ie$$

# 大規模並列粒子シミュレーションの必要性

- 大粒子数で積分時間の長いシミュレーション
- 逐次計算の速度はもう速くならない



#### 大規模並列粒子シミュレーションの困難

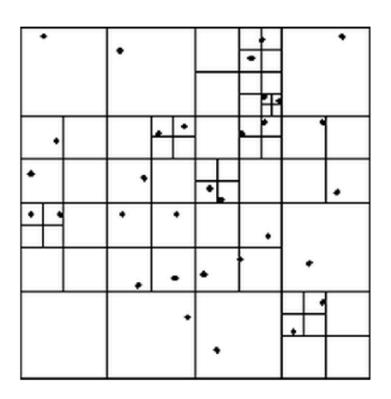
- 分散メモリ環境での並列化
  - 計算領域の分割と粒子データの交換
  - 相互作用計算のための粒子データの交換
- 共有メモリ環境での並列化
  - ツリー構造のマルチウォーク
  - 相互作用計算の負荷分散
- •1コア内での並列化
  - SIMD演算器の有効利用

#### 実は並列でなくても、、、

- キャッシュメモリーの有効利用
- ・ツリー構造の構築

$$rac{dec{u}_i}{dt} = ec{g}\left(\sum_j^N ec{f}(ec{u}_i, ec{u}_j), ec{u}_i
ight)$$

Nを小さい数に減らす方法



#### 粒子シミュレーションの手順

#### **FDPS**

- 計算領域の分割
- 粒子データの交換
- 相互作用計算のための粒子データの収集
- 実際の相互作用の計算
- 粒子の軌道積分

粒子の交換 領域分割 粒子収集

1つのプロセスが担当する領域

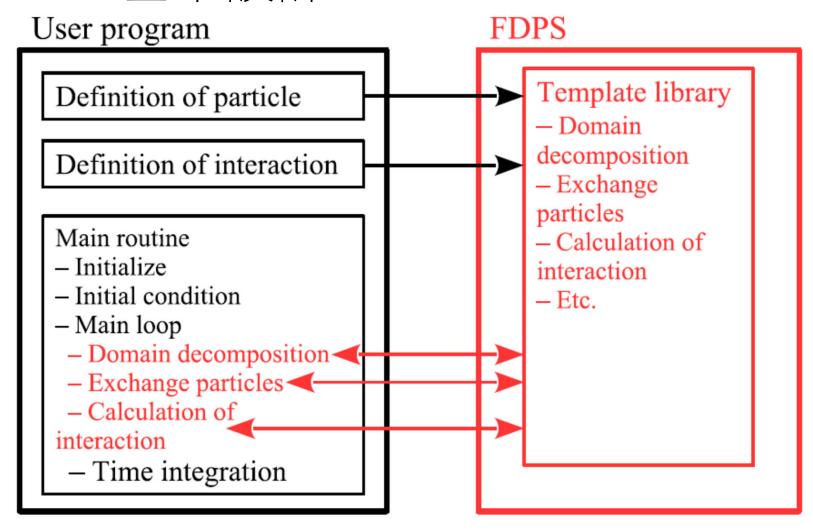
#### FDPSの実装方針(1)

- 内部実装の言語としてC++を選択
  - 高い自由度
    - 粒子データの定義にクラスを利用
    - 相互作用の定義に関数ポインタ関数オブジェクトを利用
  - 高い性能
    - 上のクラスや相互作用関数を受け取るためにテンプレートを利用
    - コンパイル時に静的にコード生成するため

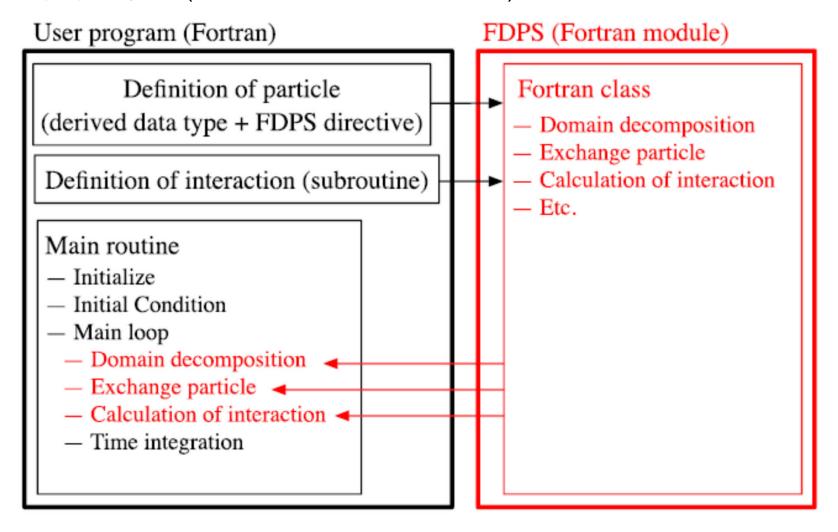
#### FDPSの実装方針(2)

- 並列化
  - 分散メモリ環境(ノード間): MPI
  - 共有メモリ環境(ノード内): OpenMP

#### FDPSの基本設計



### 基本設計(Fortranの場合)



#### 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(Hamada et al 2009) を実装
- 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

#### 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.

```
include <particle_simulator.hpp
ising namespace PS;
 iblic:
   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:
   wold readAscii(FILE *fp) {
       fscanf(fp.
               "%11%11%11%11%11%11%11%11%11"
              &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:
cemplate <class TPJ>
truct CalcGrav{
   void operator () (const Nbody * ip,
                     const S32 n1,
                     const TPJ . jp.
                     const S32 nj.
                     Nbody . force) {
       for($32 1-0; 1<n1; 1++){
           F64vec x1 - 1p[1].pos;
           F64 ep2 = 1p[1].eps
               ip[1].eps;
           F64vec a1 - 0.0:
           for(S32 j=0; j<nj;j++){
```

F64vec xj = jp[j].pos; F64vec dr = x1 - xj;

F64 mj - jp[j].mass; F64 dr2 - dr + dr + ep2;

F64 dri = 1.0 / sqrt(dr2) a1 -= (dri \* dri \* dri

```
force[1].acc +- a1:
63 template < class Tpsys
                const F64 dt) 4
       S32 n = p.getNumberOfParticleLocal();
       for($32 1 - 0: 1 < n: 1++)
           p[1].predict(dt);
71 template < class Tpsys>
   void correct (Tpsys &p.
               const F64 dt) {
       S32 n - p.getNumberOfParticleLocal();
       for(S32 1 - 0; 1 < n; 1++)
           p[1].correct(dt);
   template <class TDI, class TPS, class TTFF
   void calcGravAllAndWriteBack(TDI &dinfo.
                                 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
       const F32 dtime
                           .0 / 128.0;
       PS::Initial
                     e(argc, argv);
                 iInfo dinfo:
             initialize():
           :ParticleSystem < Nbody > ptcl;
       ptcl initialize():
       PS::TreeForForceLong < Nbody . Nbody .
           Nbody > :: Monopole gray:
       gray.initialize(0):
       ptcl.readParticleAscii(argv[i]);
       calcGravAllAndWriteBack(dinfo.
                                ptcl.
       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

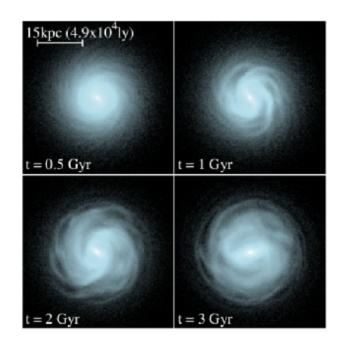
大規模並列N体コードが117行で書ける

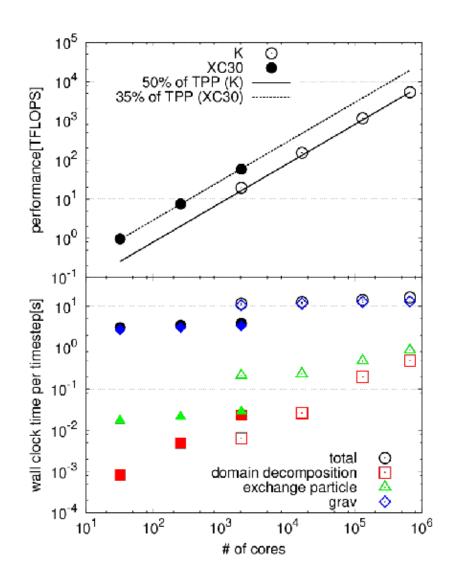
#### 重要なポイント

- ユーザーはMPIやOpenMPを考えなくてよい。
- •相互作用関数の実装について
  - 2重ループ:複数の粒子に対する複数の粒子からの作用を計算
  - チューニングが必要(FDPSチームに相談可)
    - 徐算回数の最小化
    - SIMD演算器の有効利用

#### 性能(N体)

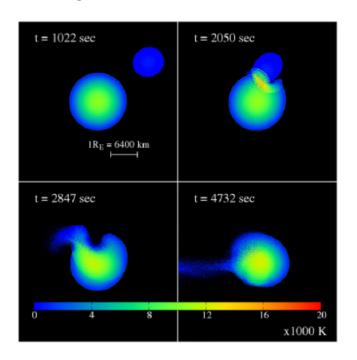
- 円盤銀河
- 粒子数: 2.7x10<sup>5</sup>/core
- 精度: Θ=0.4 四重極
- 京コンピュータ, XC30

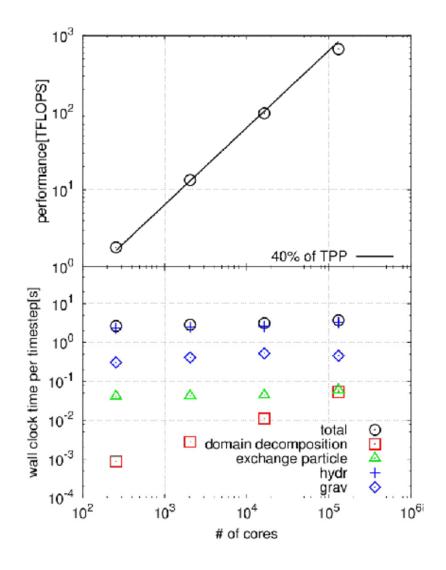




#### 性能(SPH)

- 巨大衝突シミュレーション
- 粒子数: 2.0x10<sup>4</sup>/core
- 京コンピュータ





#### まとめ

- FDPSは大規模並列粒子シミュレーションコードの開発を支援 するフレームワーク
- FDPSのAPIを呼び出すだけで粒子シミュレーションを並列化
- N体コードを100行程度で記述
- ・京コンピュータで理論ピーク性能の40、50%の性能を達成