FDPSの概要説明

行方大輔
理化学研究所 計算科学研究機構
粒子系シミュレータ開発チーム
エクサスケールコンピューティング開発プロジェクト コデザインチーム
2017/03/01 PCoMS東北大学　HPC技術講習会 FDPS 講習会

FDPSとは

- Framework for Developing Particle Simulator
- 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
- 重力N体、SPH、分子動力学、粉体、etc.
- 支配方程式

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

粒子データのベクトル

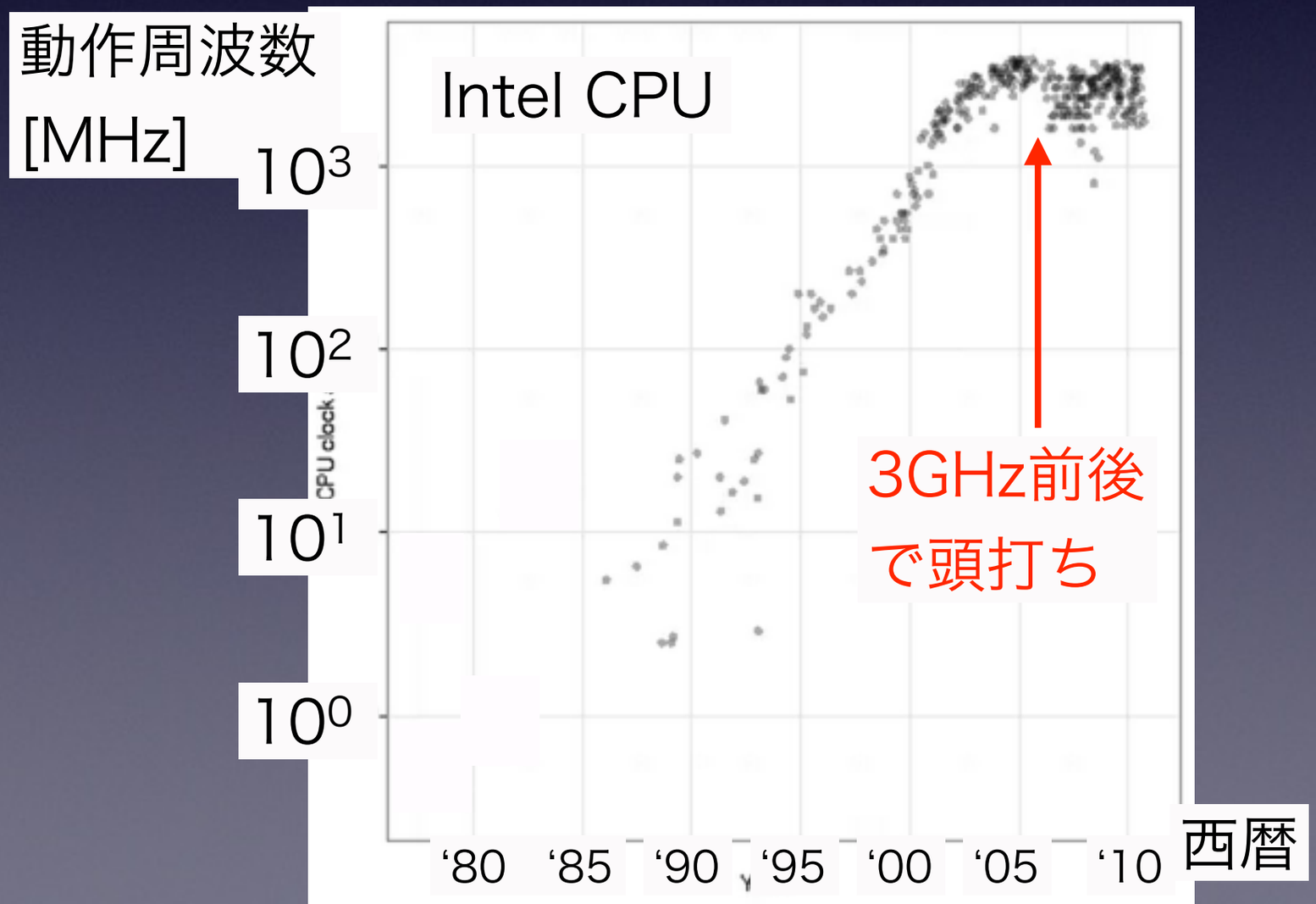
粒子の持つ物理量をその導関数に変換する関数

粒子間相互作用を表す関数

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

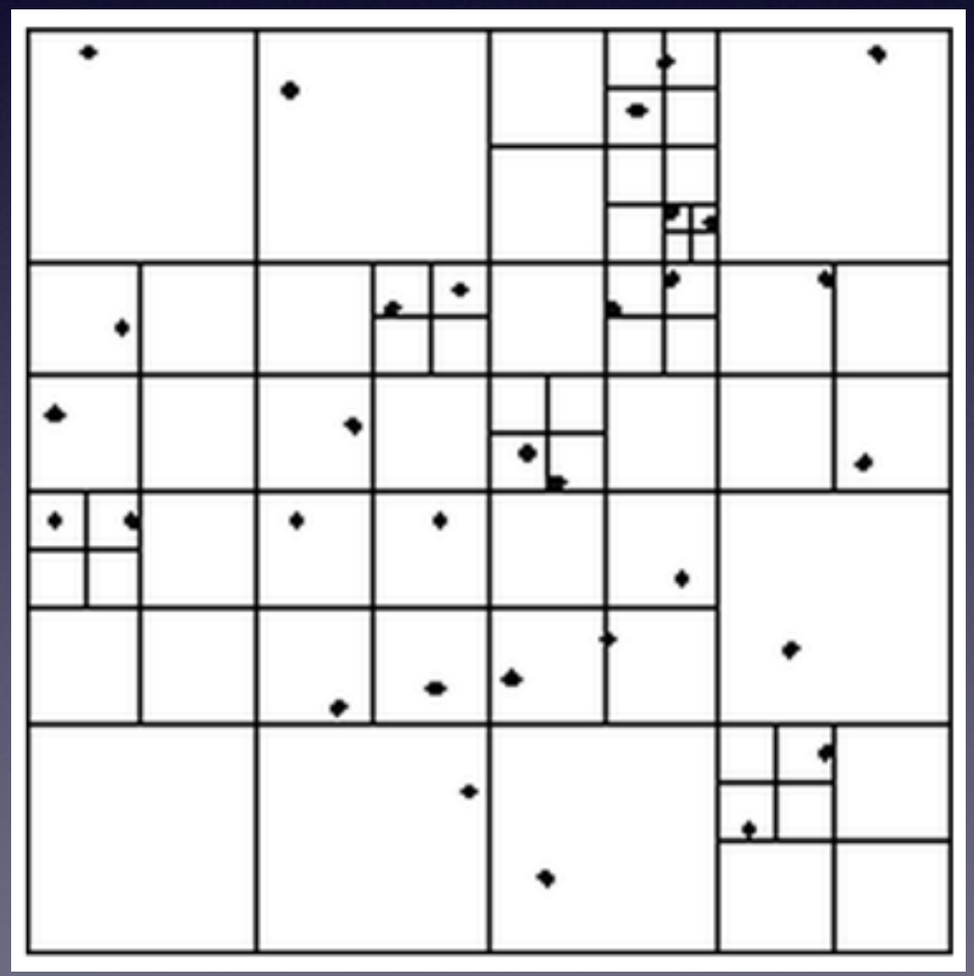
- 分散メモリ環境での並列化
  - 計算領域の分割と粒子データの交換
  - 相互作用計算のための粒子データの交換
- 共有メモリ環境での並列化
  - ツリー構造のマルチウォーク
  - 相互作用計算の負荷分散
- １コア内での並列化
  - SIMD演算器の有効利用

実は並列でなくても、、、

- キャッシュメモリの有効利用
- ツリー構造の構築

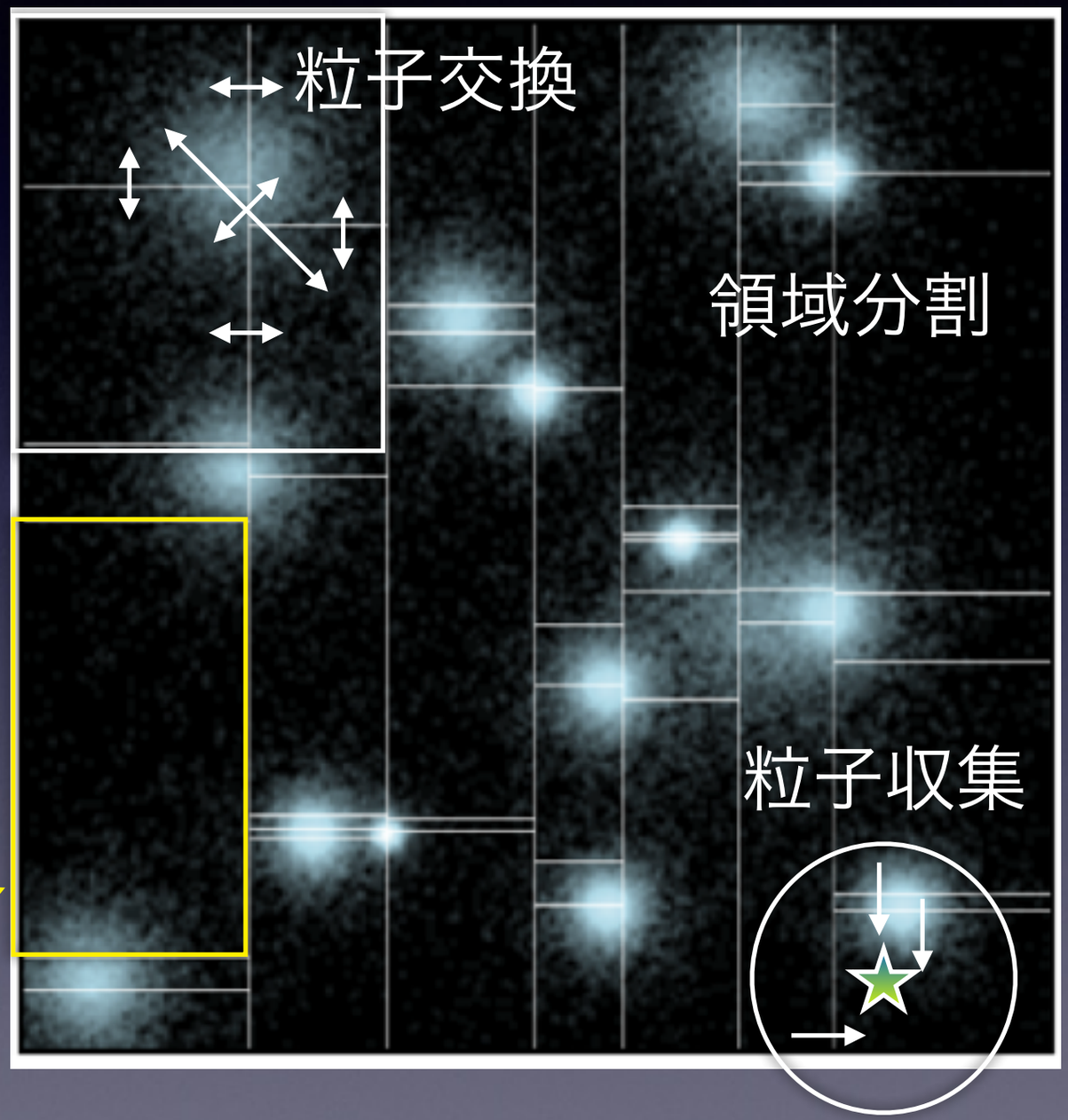
$$\frac{d\vec{u}_i}{dt} = \vec{g}\left(\sum_{j}^{N} \vec{f}(\vec{u}_i, \vec{u}_j), \vec{u}_i\right)$$

高速に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

- 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



１つのプロセスが
担当する領域

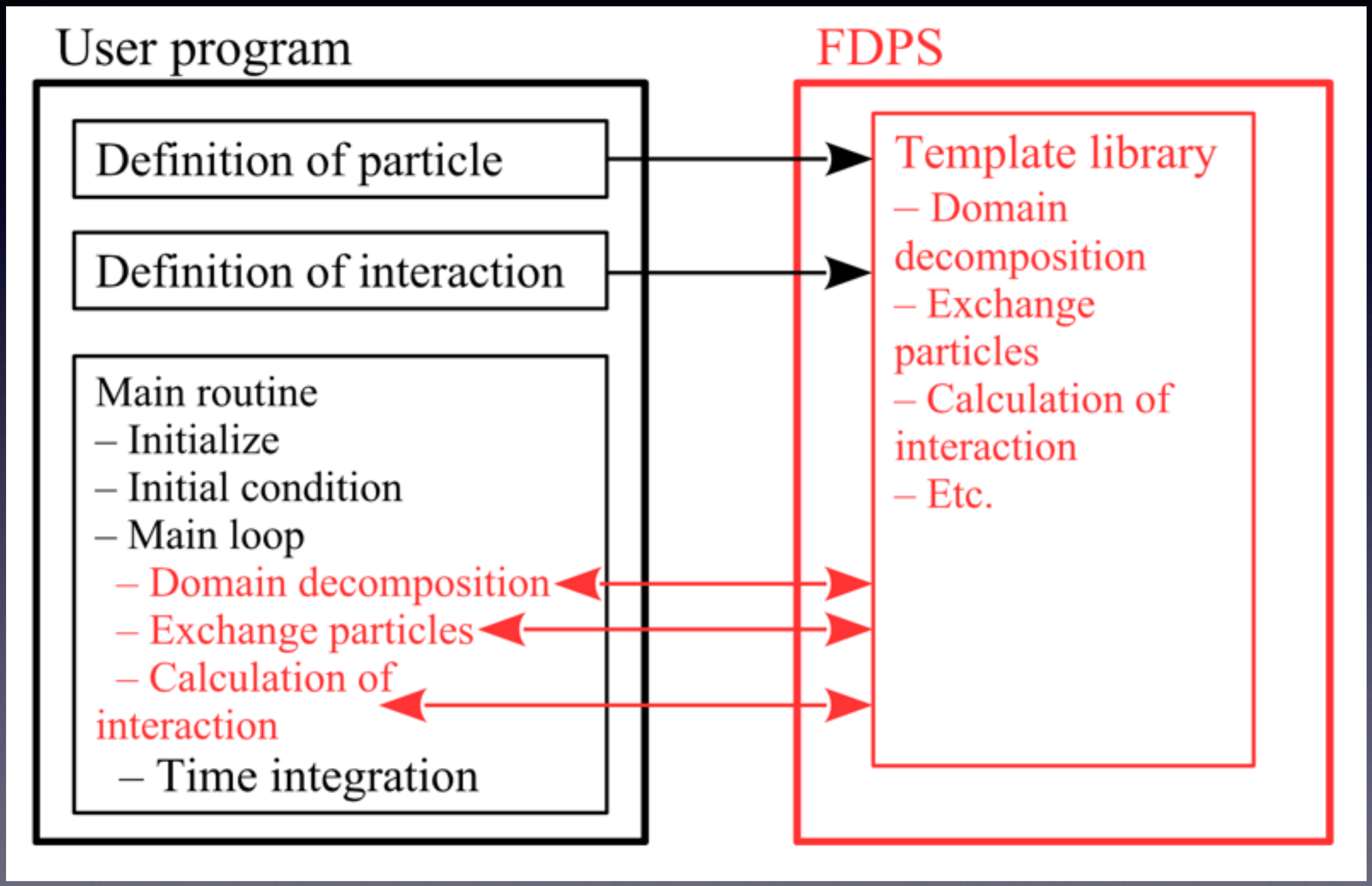
FDPSの実装方針(1)

- 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・関数ポインタ・関数オブジェクトを受け取るために**テンプレートクラス**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
    或いは
    GPU

FDPSの基本設計



ユーザーはC++の一部の知識を必要とする

サンプルコード(N体)

FDPSのインストール(ヘッダファイルをインクルードするだけ)

粒子データの定義
(C++のクラス)

粒子間の相互作用の定義
(C++の関数オブジェクト
または関数ポインタ)

メインルーチン(メイン
関数)の実装

大規模並列N体コードが
117行で書ける！

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

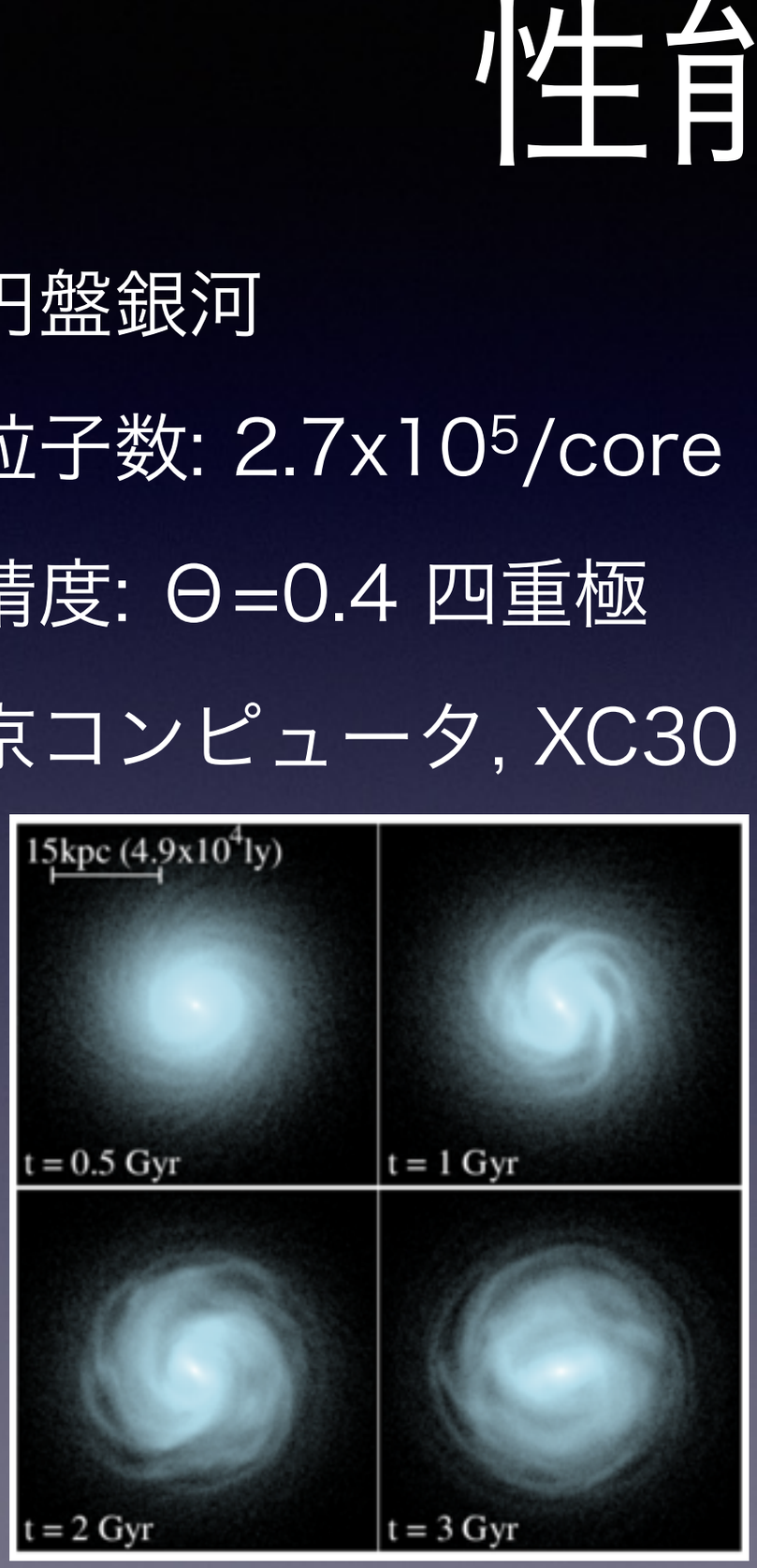
```
#include <particle_simulator.hpp>
using namespace PS;

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;
}
```

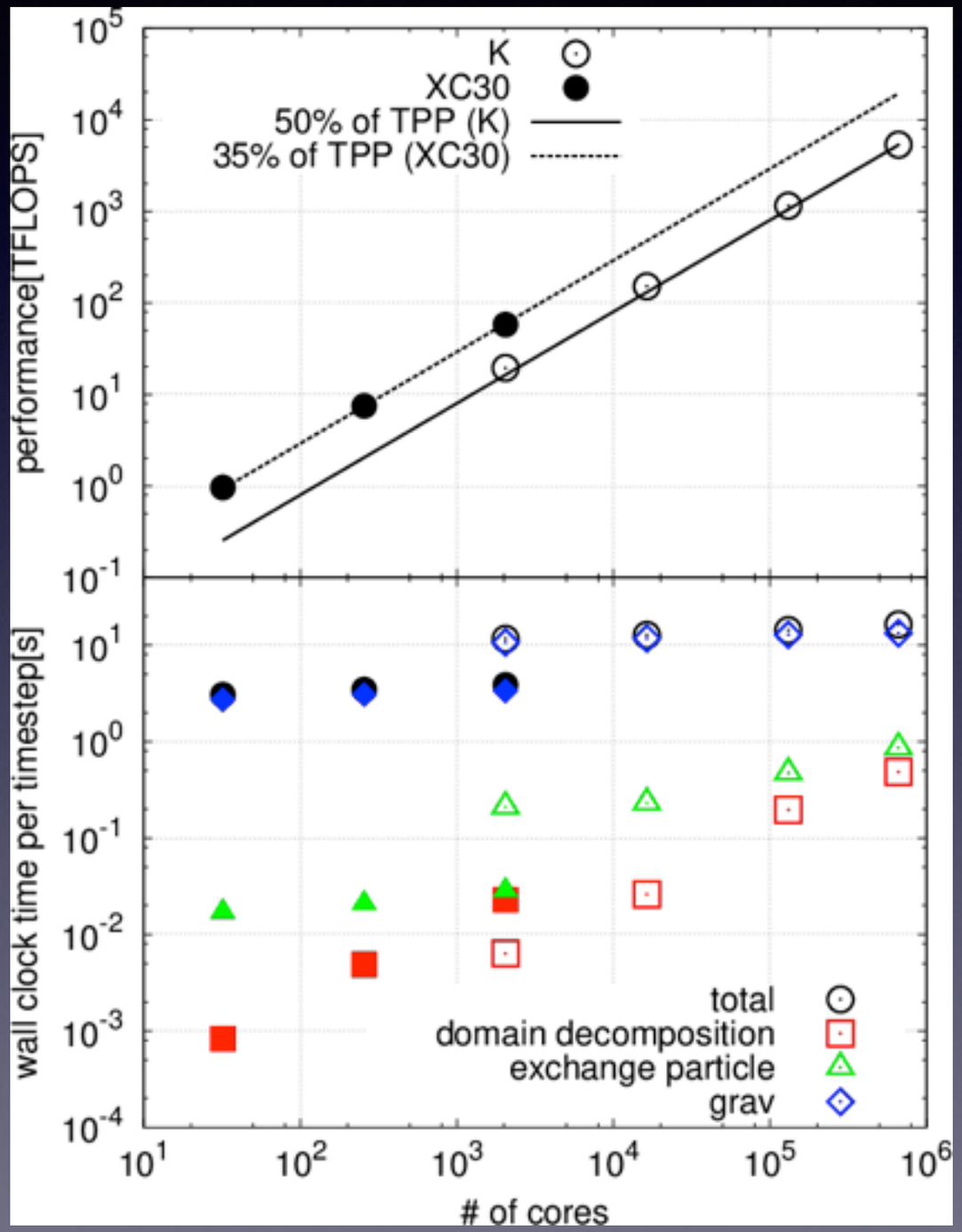
重要なポイント

- 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の計算
  - チューニングが必要(FDPSチームに相談可)
    - 除算回数の最小化
    - SIMD演算器の有効利用

性能(N体)

- 円盤銀河
- 粒子数: 2.7×10^5/core
- 精度: Θ=0.4 四重極
- 京コンピュータ, XC30

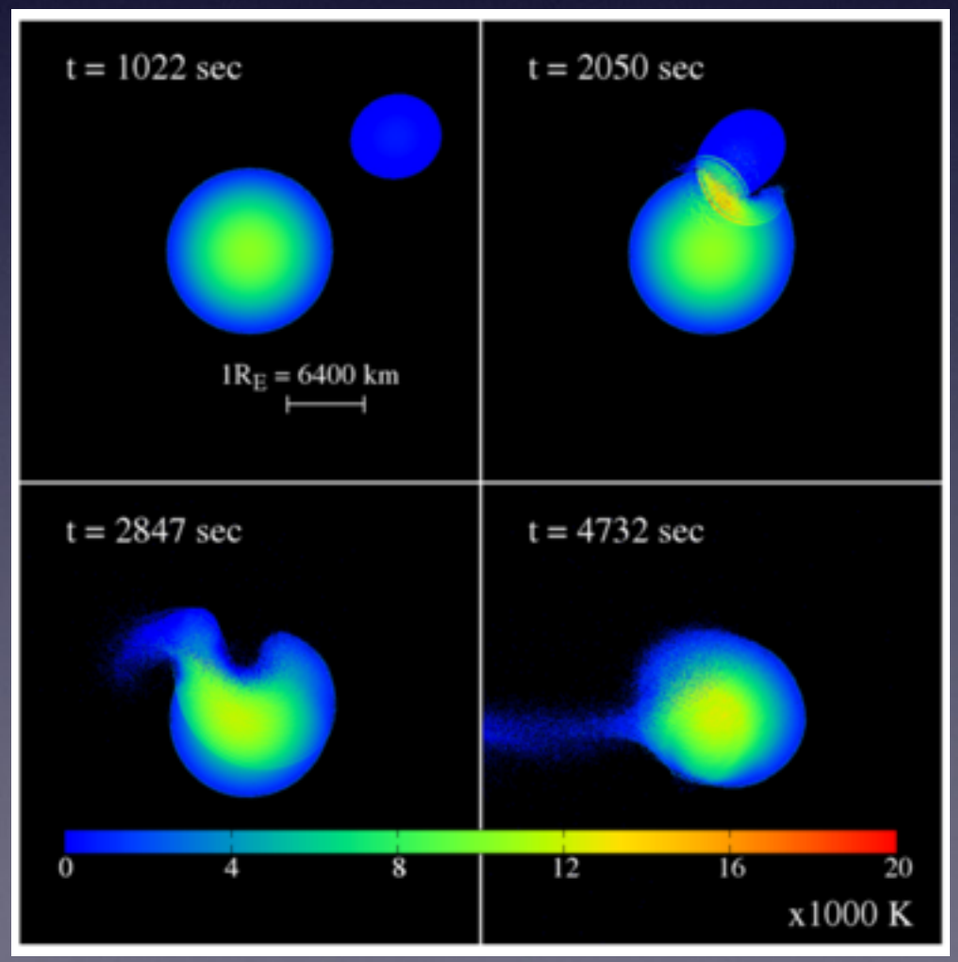


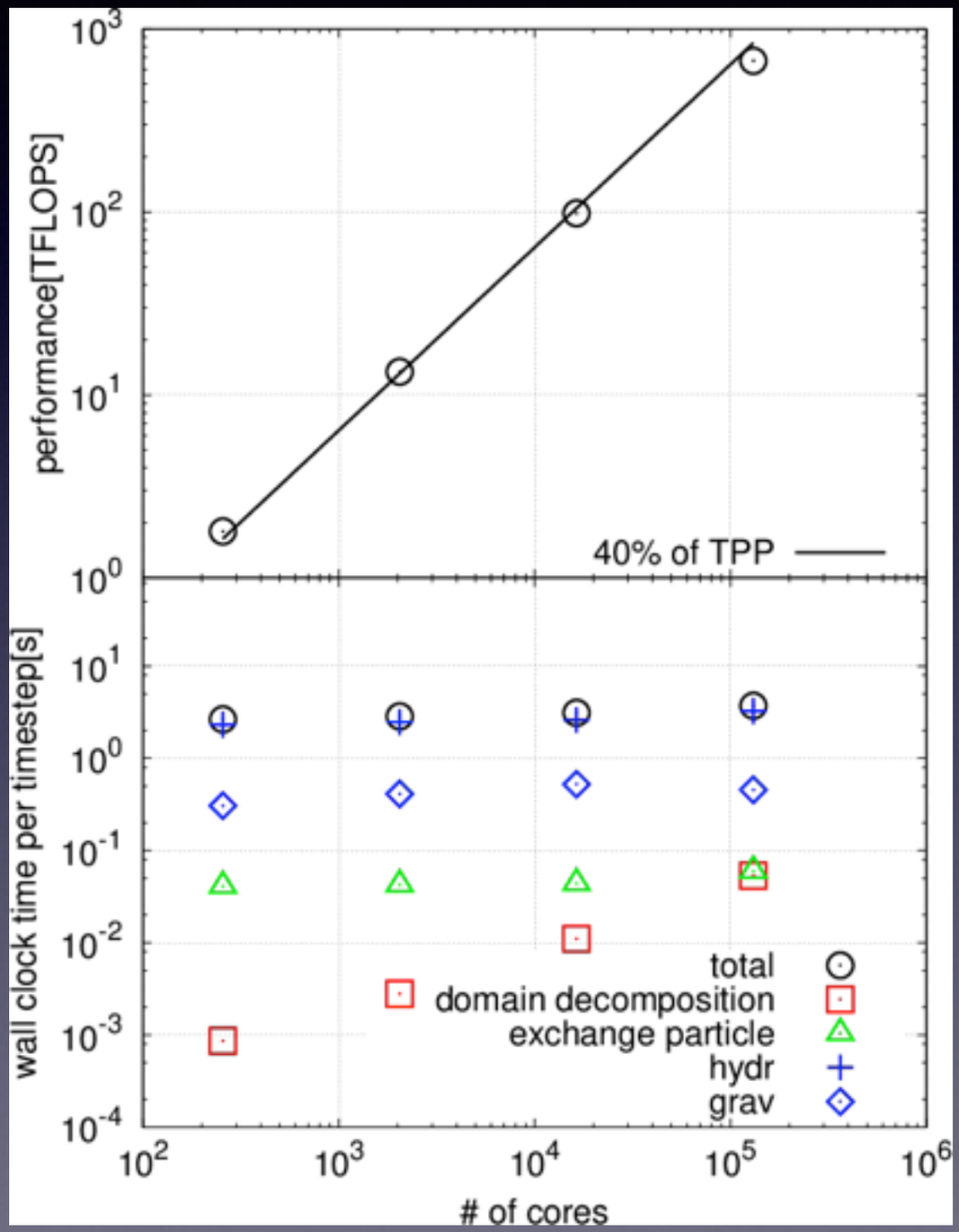


Iwasawa et al. (2016)

性能(SPH)

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





Iwasawa et al. (2016)

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117行で記述
- 京コンピュータで理論ピーク性能の40、50%の性能を達成