

FDPSの概要説明

岩澤全規

松江工業高等専門学校

2024/09/06 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(MD)、個別要素法(DEM)、etc...
- 支配方程式

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

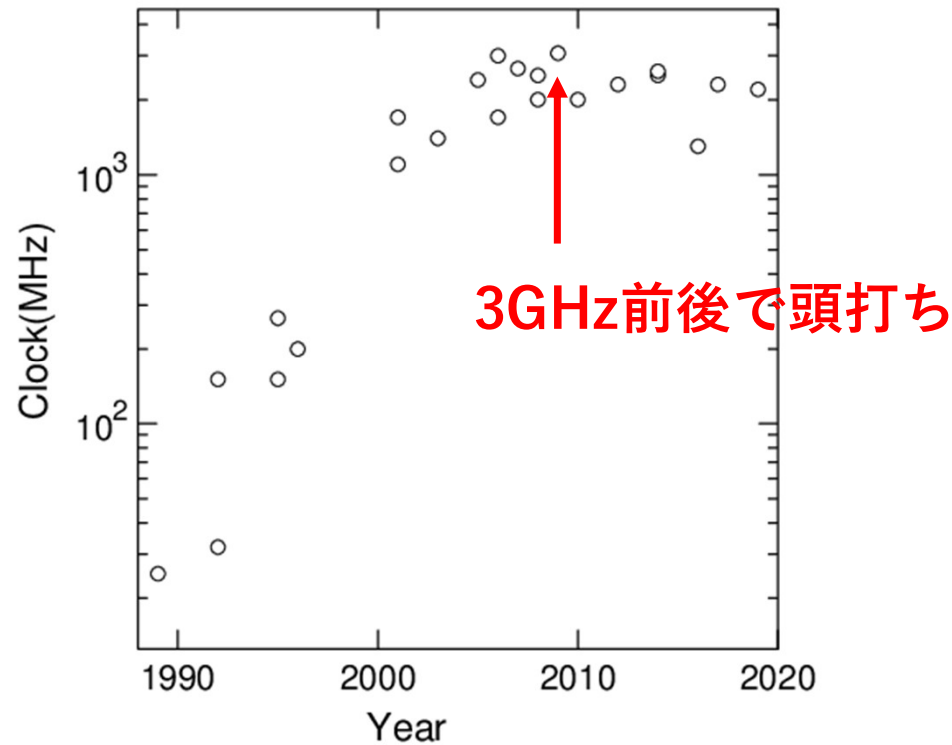
粒子データのベクトル

粒子間相互作用を表す関数

粒子の持つ物理量をその導関数に変換する関数

大規模並列粒子 シミュレーションの必要性

- 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

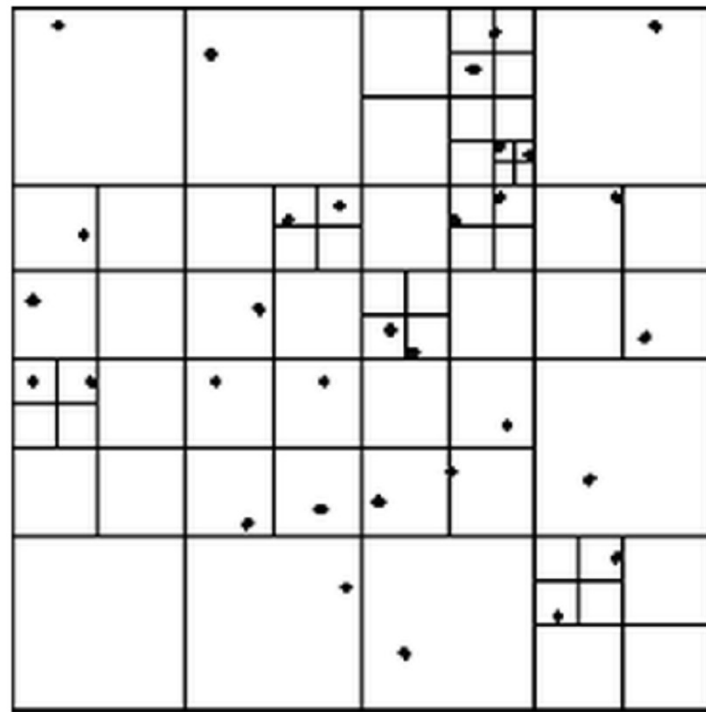
- 分散メモリ環境での並列化
 - 計算領域の分割と粒子データの交換
 - 相互作用計算のための粒子データの交換
- 共有メモリ環境での並列化
 - 相互作用計算の負荷分散
- 1コア内での並列化
 - SIMD演算器の有効利用

実は並列でなくとも、、、

- キャッシュメモリの有効利用
- ツリー構造の構築

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

Nを小さい数に減らす方法



FDPSで困難を解決

- 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
 - 相互作用計算の負荷分散
- キャッシュメモリの有効利用
- Tree構造による粒子分布の管理
- 1コア内での並列化
 - SIMD演算器の有効利用



FDPS



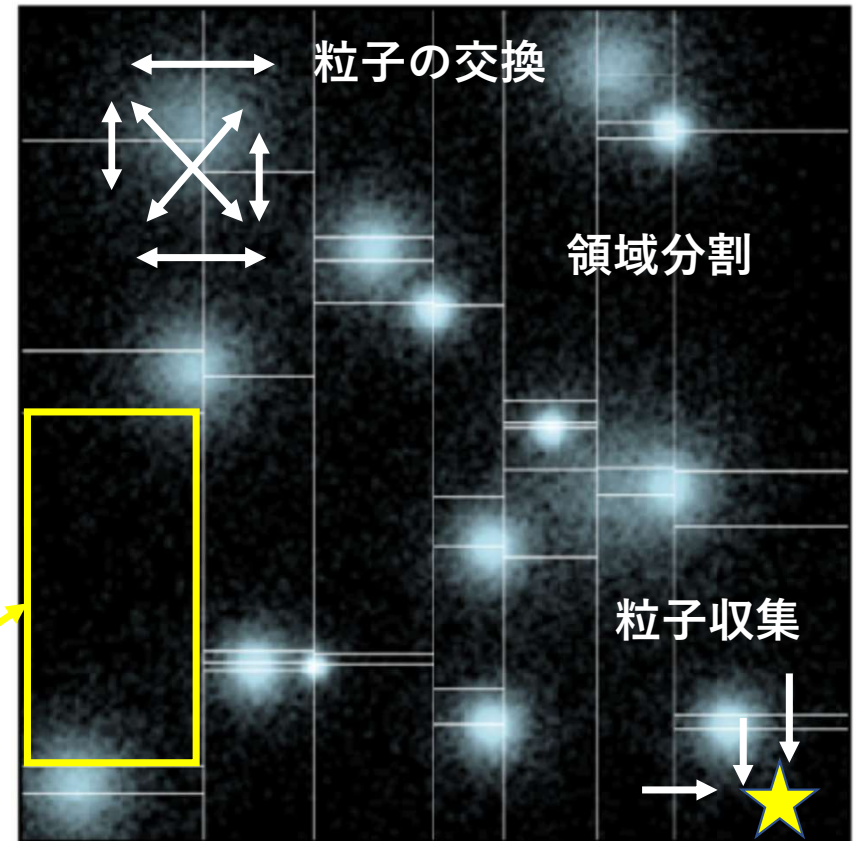
PIKG (牧野さん講義)

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

1つのプロセスが担当する領域



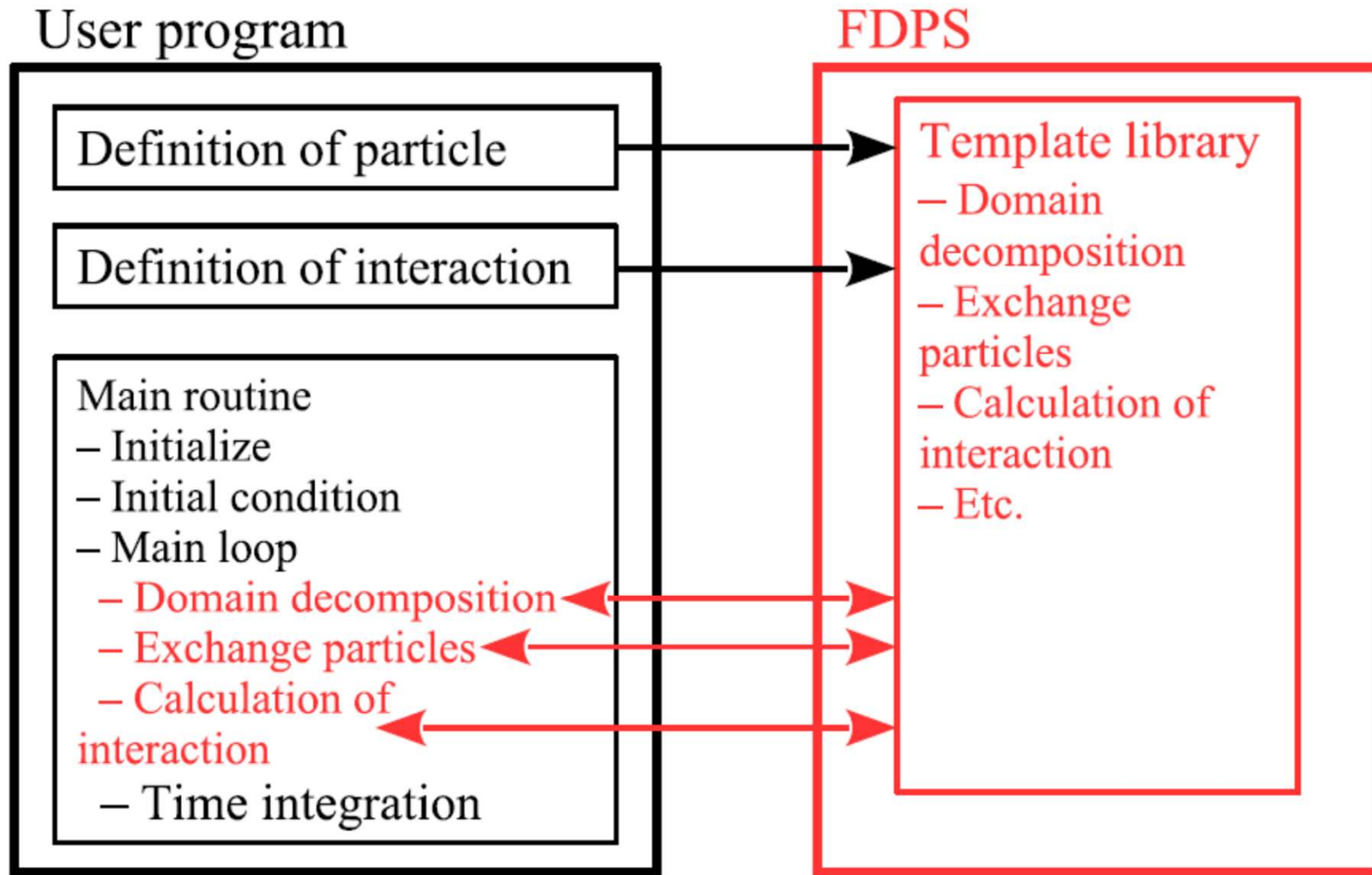
FDPSの実装方針(1)

- 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



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

```

1 #include <particle_simulator.hpp>
2 using namespace PS;
3
4 class Nbody{
5 public:
6     F64 mass, eps;
7     F64vec pos, vel, acc;
8     F64vec getPos() const {return pos;}
9     F64 getCharge() const {return mass;}
10    void copyFromFP(const Nbody &in){
11        mass = in.mass;
12        pos = in.pos;
13        eps = in.eps;
14    }
15    void copyFromForce(const Nbody &out) {
16        acc = out.acc;
17    }
18    void clear() {
19        acc = 0.0;
20    }
21    void readAscii(FILE *fp) {
22        fscanf(fp,
23            "%lf%lf%lf%lf%lf%lf%lf%lf%lf%lf",
24            &mass, &eps,
25            &pos.x, &pos.y, &pos.z,
26            &vel.x, &vel.y, &vel.z);
27    }
28    void predict(F64 dt) {
29        vel += (0.5 * dt) * acc;
30        pos += dt * vel;
31    }
32    void correct(F64 dt) {
33        vel += (0.5 * dt) * acc;
34    }
35 };
36
37 template <class TPJ>
38 struct CalcGrav{
39     void operator () (const Nbody * ip,
40                     const S32 ni,
41                     const TPJ * jp,
42                     const S32 nj,
43                     Nbody * force) {
44         for(S32 i=0; i<ni; i++){
45             F64vec x1 = ip[i].pos;
46             F64 ep2 = ip[i].eps
47                 * ip[i].eps;
48             F64vec a1 = 0.0;
49             for(S32 j=0; j<nj; j++){
50                 F64vec xj = jp[j].pos;
51                 F64vec dr = x1 - xj;
52                 F64 mj = jp[j].mass;
53                 F64 dr2 = dr * dr + ep2;
54                 F64 dri = 1.0 / sqrt(dr2);
55                 a1 += (dri * dri * dri

```

```

56                 * mj) * dr;
57             force[i].acc += a1;
58         }
59     }
60 };
61
62 template <class Tpsys>
63 void predict(Tpsys &p,
64             const F64 dt) {
65     S32 n = p.getNumberOfParticleLocal();
66     for(S32 i = 0; i < n; i++)
67         p[i].predict(dt);
68 }
69
70 template <class Tpsys>
71 void correct(Tpsys &p,
72             const F64 dt) {
73     S32 n = p.getNumberOfParticleLocal();
74     for(S32 i = 0; i < n; i++)
75         p[i].correct(dt);
76 }
77
78 template <class TDI, class TPS, class ITFF>
79 void calcGravAllAndWriteBack(TDI &dinfo,
80                             TPS &ptcl,
81                             ITFF &tree) {
82     dinfo.decomposeDomainAll(ptcl);
83     ptcl.exchangeParticle(dinfo);
84     tree.calcForceAllAndWriteBack
85         (CalcGrav<Nbody>(),
86          CalcGrav<SPJMonopole>(),
87          ptcl, dinfo);
88 }
89
90 int main(int argc, char *argv[]) {
91     F32 time = 0.0;
92     const F32 tend = 1000;
93     const F32 dtime = 1.0 / 128.0;
94     PS::Initialize(argc, argv);
95     PS::DomainInfo dinfo;
96     dinfo.initialize();
97     PS::ParticleSystem<Nbody> ptcl;
98     ptcl.initialize();
99     PS::TreeForForceLong<Nbody, Nbody,
100         Nbody>::Monopole grav;
101     grav.initialize(0);
102     ptcl.readParticleAscii(argv[1]);
103     calcGravAllAndWriteBack(dinfo,
104                             ptcl,
105                             grav);
106
107     while(time < tend) {
108         predict(ptcl, dtime);
109         calcGravAllAndWriteBack(dinfo,
110                                 ptcl,
111                                 grav);
112         correct(ptcl, dtime);
113         time += dtime;
114     }
115     PS::Finalize();
116     return 0;
117 }

```

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

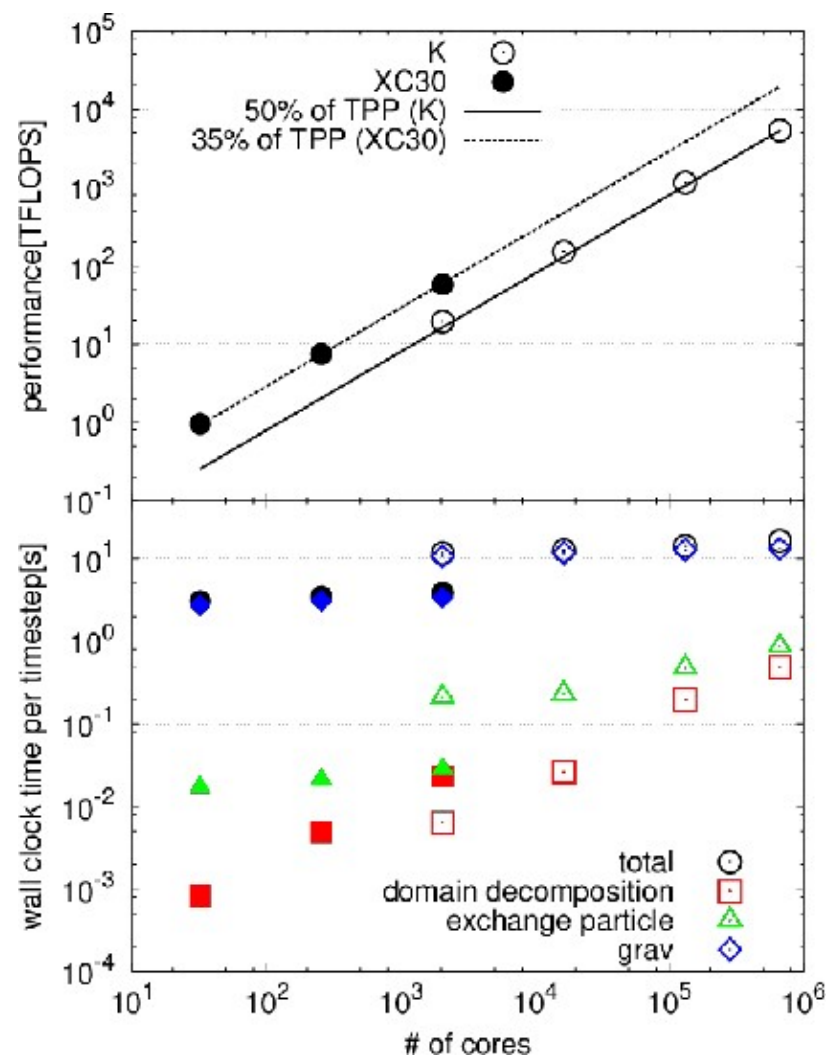
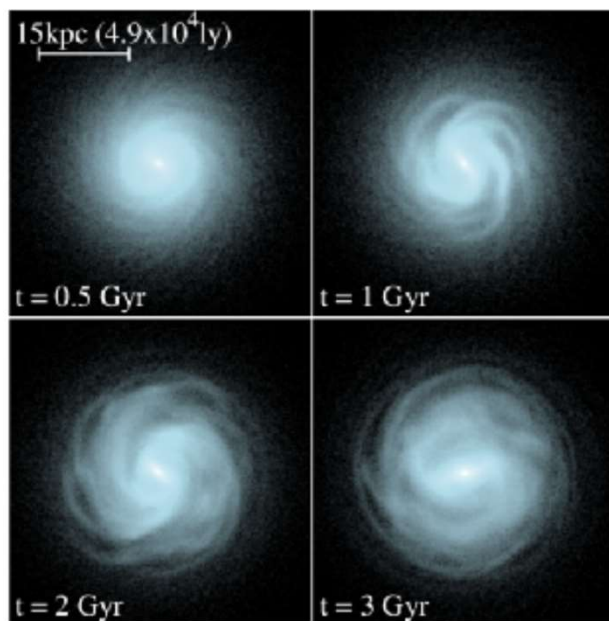
大規模並列N体コードが117行で書ける
 • MPIやOpenMPのAPIを明示的に使っていない

重要なポイント

- ユーザーはMPIやOpenMPによる並列化を考えなくてよい.
- 相互作用関数はチューニングが必要
 - ※PIKGを使えば最適化された相互作用関数を生成してくれる.

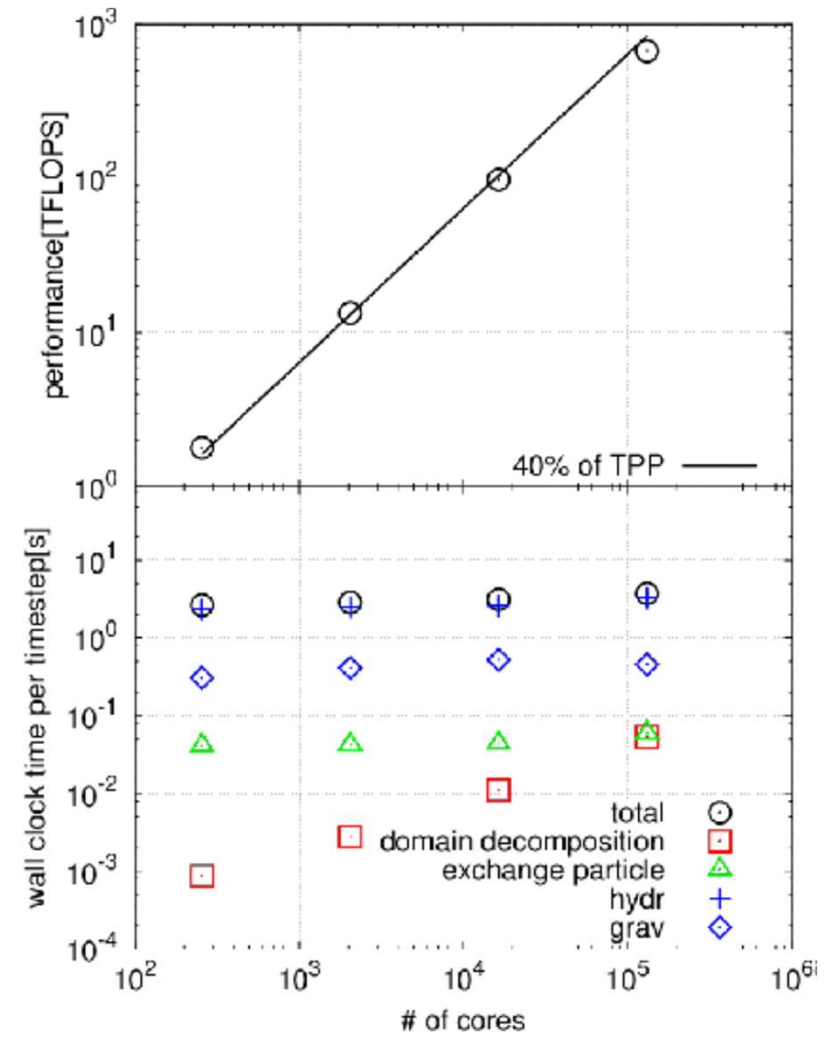
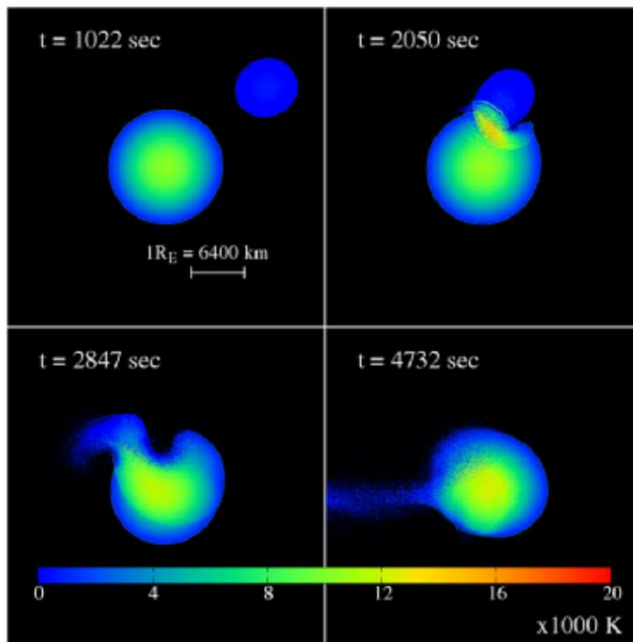
性能(N体)

- 円盤銀河
- 粒子数: $2.7 \times 10^5 / \text{core}$
- 精度: $\Theta = 0.4$ 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: 2.0×10^4 /core
- 京コンピュータ



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
- 2024年x月 FDPS Ver.8.0
 - PMMM法の導入
- 2025年y月 FDPS Ver.9.0
 - GPU対応

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体コードを数百行程度で記述
- 京コンピュータで理論ピーク性能の40、50%の性能を達成