GPUとGPAPEライブラリ

2022年度N体シミュレーション立春の学校

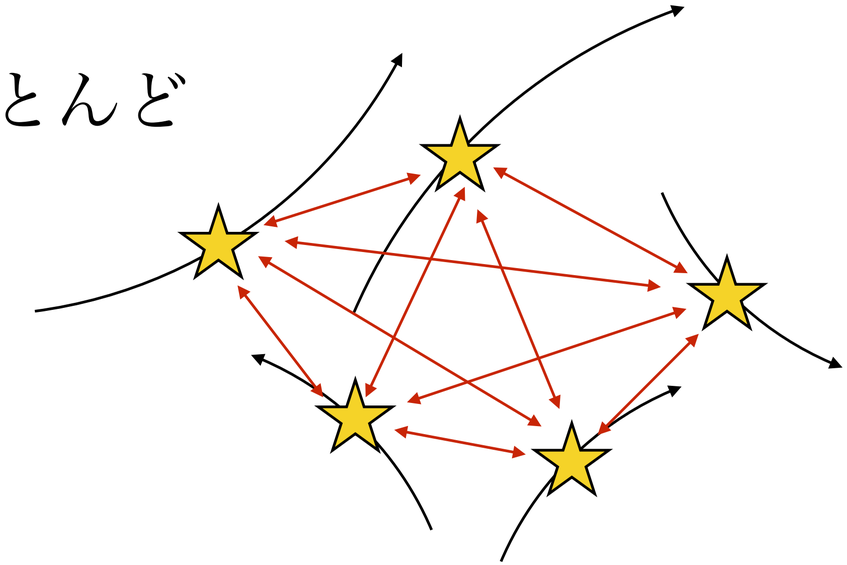
谷川衝（東京大学大学院総合文化研究科）

概要

- イントロダクション
- GPUクラスタでのN体シミュレーションGPUライブラリ
- Phantom-GRAPE
- FDPS: Framework for Developing Particle Simulator
- 衝突系のN体シミュレーションコード

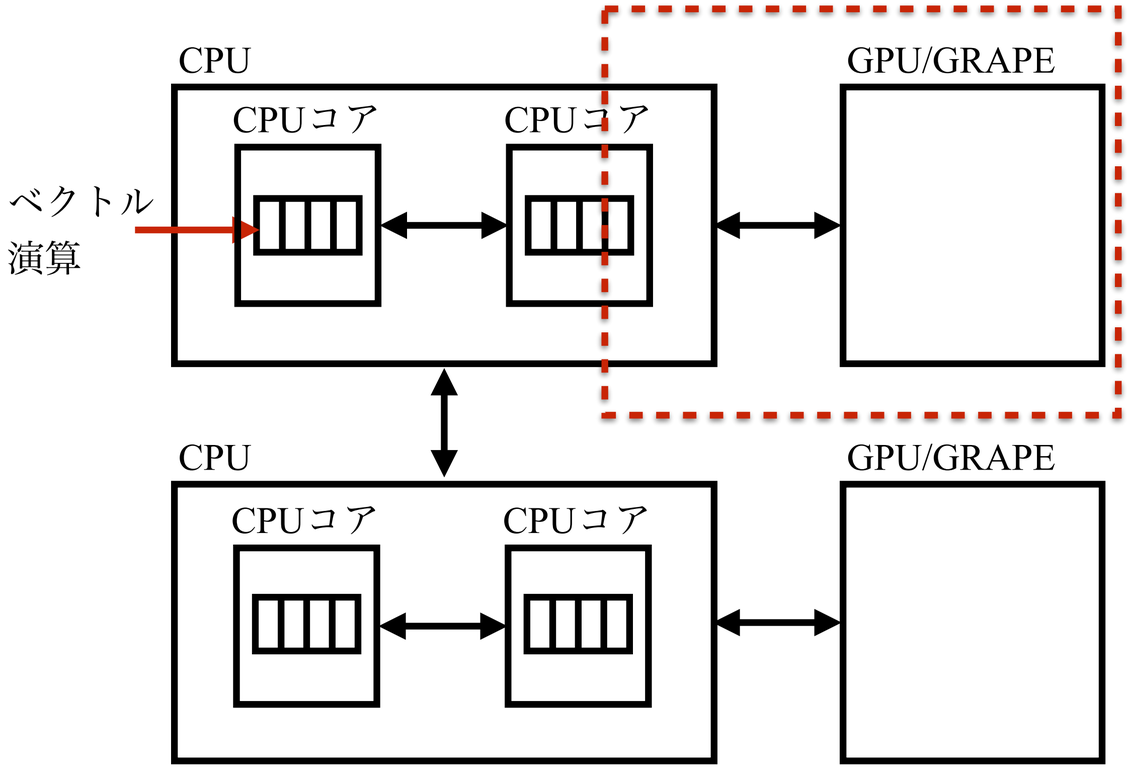
重力N体シミュレーション

- 運動方程式: $\frac{d^2\vec{r}_i}{dt^2} = \vec{f}_i \ldots \mathcal{O}(N)$
- 重力計算: $\vec{f}_i = \sum_{j \neq i}^{N} \frac{m_j}{|\vec{r}_j - \vec{r}_i|^2} \frac{\vec{r}_j - \vec{r}_i}{|\vec{r}_j - \vec{r}_i|} \ldots \mathcal{O}(N^2)$
- 重力計算が全計算量のほとんど
- 重力計算の高速化



高速化の手法

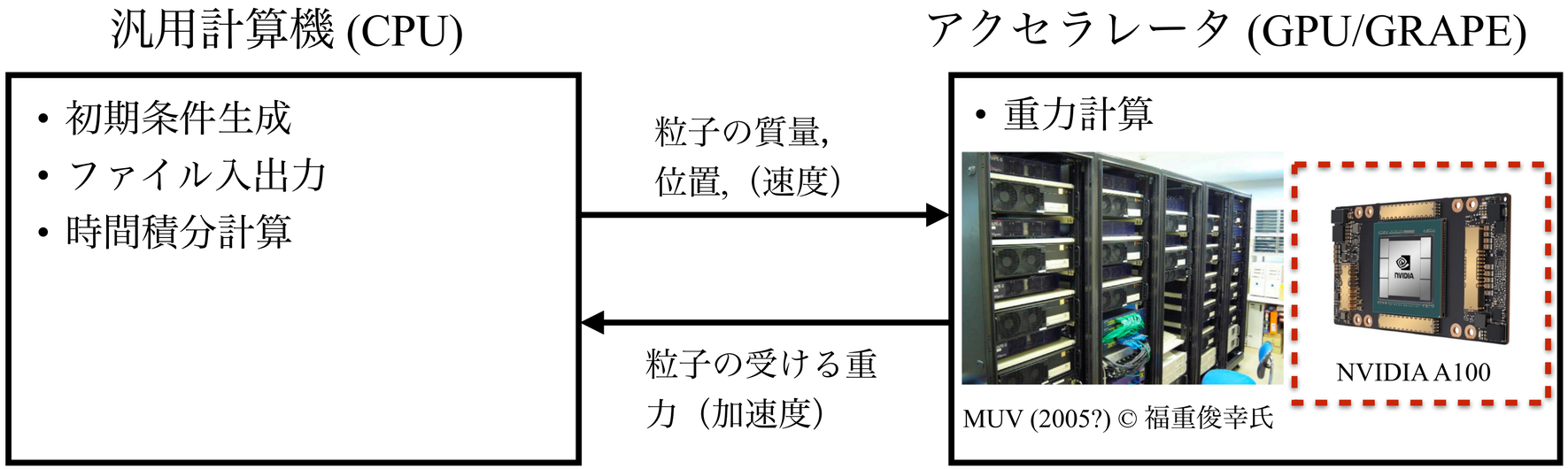
- アルゴリズム的な計算量の削減
  - ツリー法，P^{3}M法など
  - 独立時間刻み法など
- 重力計算の並列化
  - ベクトル演算（AVXなど）
  - CPUコア間（OpenMPなど）
  - CPU間（MPIなど）
- アクセラレータ（GPU，GRAPEなど）



概要

- イントロダクション
- GPUクラスタでのN体シミュレーションGPUライブラリ
- Phantom-GRAPE
- FDPS: Framework for Developing Particle Simulator
- 衝突系のN体シミュレーションコード

アクセラレータの使用モデル



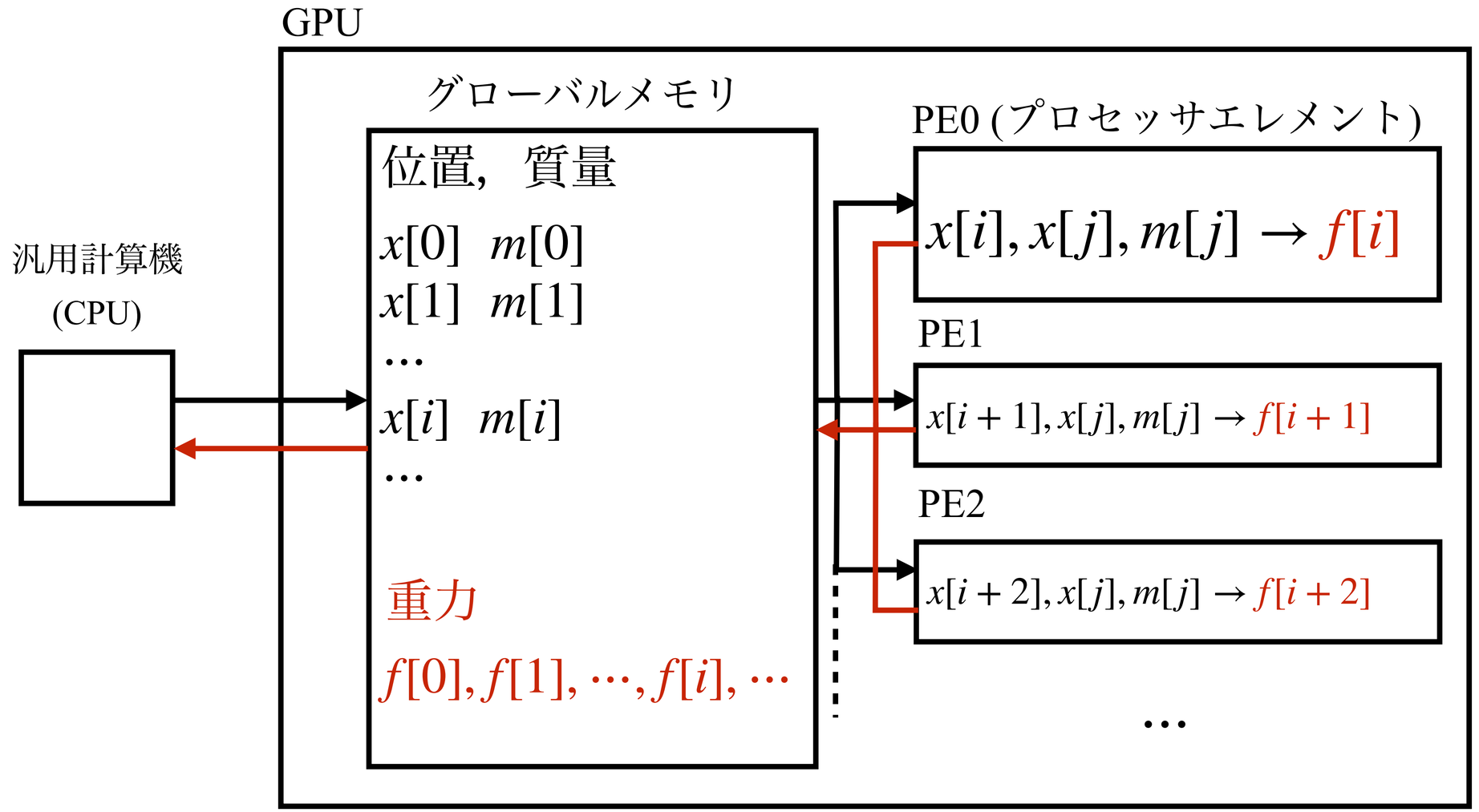
- 計算量の多い重力計算をアクセラータが担当
- （GPUなら時間積分計算なども実行可能）

GPUで高速化が可能な理由

- 重力計算の高い並列性
  - $\vec{f}_i = \sum_{j \neq i}^{N} \frac{m_j}{|\vec{r}_j - \vec{r}_i|^2} \frac{\vec{r}_j - \vec{r}_i}{|\vec{r}_j - \vec{r}_i|}$
    - $\vec{f}_1 = \frac{m_2}{|\vec{r}_2 - \vec{r}_1|^2} \frac{\vec{r}_2 - \vec{r}_1}{|\vec{r}_2 - \vec{r}_1|} + \frac{m_3}{|\vec{r}_3 - \vec{r}_1|^2} \frac{\vec{r}_3 - \vec{r}_1}{|\vec{r}_3 - \vec{r}_1|} + \cdots$
    - $\vec{f}_2 = \frac{m_1}{|\vec{r}_1 - \vec{r}_2|^2} \frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|} + \frac{m_3}{|\vec{r}_3 - \vec{r}_2|^2} \frac{\vec{r}_3 - \vec{r}_2}{|\vec{r}_3 - \vec{r}_2|} + \cdots$
    - $\vec{f}_3 = \cdots$
  - $\mathcal{O}(N^2)$の数の全相互作用を独立に計算可能
- 浮動小数点数演算する単位の数（単精度の場合）の多さ
  - XC50 cpu: 20 core/cpu x 32 VE x 2FMA x 2.4GHz = 3.07TF
  - NVIDIA A100: 6912 core/gpu x 2FMA x 1.41GHz = 19.5TF

各相互作用に依存性がないので,
同時に（独立に）計算することが可能

GPUでの重力計算


GPU
グローバルメモリ
位置，質量
$x[0]$ $m[0]$
$x[1]$ $m[1]$
…
$x[i]$ $m[i]$
…
重力
$f[0], f[1], \cdots, f[i], \cdots$
汎用計算機
(CPU)
PE0 (プロセッサエレメント)
$x[i], x[j], m[j] \rightarrow f[i]$
PE1
$x[i+1], x[j], m[j] \rightarrow f[i+1]$
PE2
$x[i+2], x[j], m[j] \rightarrow f[i+2]$
…

実際の手順

1.重力を及ぼす側の全粒子(j粒子)の位置と質量をCPUから粒子メモリ（GPUのグローバルメモリ）に書き込む.

2.全粒子への重力が求まるまで以下を行う.

1)重力を受ける側の粒子(i粒子)の位置を各PE内のメモリ（レジスタ）に送る.

2)すべてのj粒子との重力を各PEでプログラムに基づいて計算し積算する.

3)求まったi粒子への重力を各PE内のメモリ（レジスタ）から粒子メモリ（GPUのグローバルメモリ）に送り返す.

3.粒子メモリ（GPUのグローバルメモリ）からCPUへ送り返す.

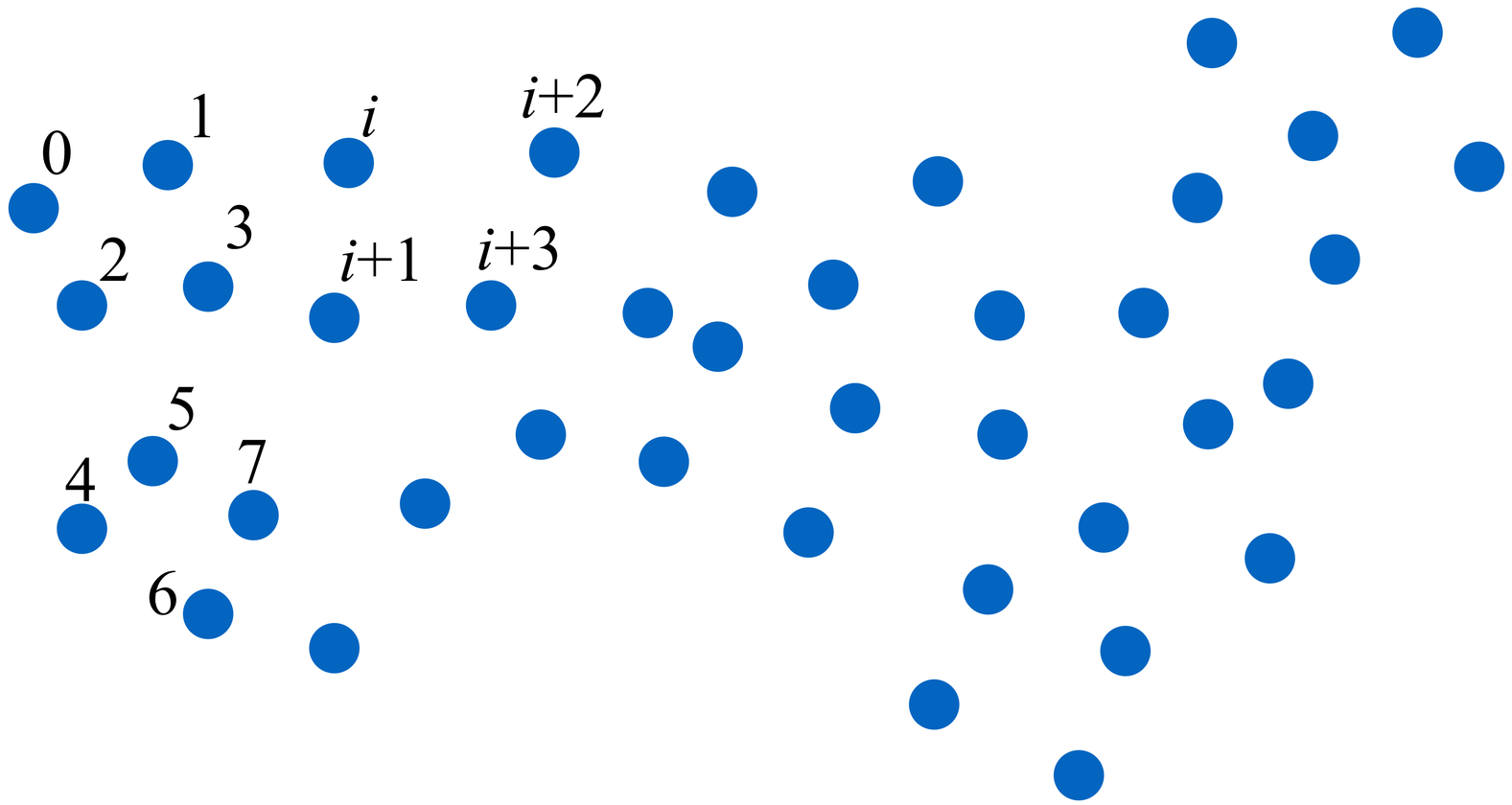
4.求まった重力を用いて運動方程式を積分し，次の時刻の位置を求める.

5.1へ戻る.

プログラム例

```
npipes = g5_get_number_of_pipelines();
for (int step = 0; step < final_step; step++) {
    g5_open();
    g5_set_range(xmin,xmax,mmin);
    g5_set_jp(0,n,m,x);
    g5_set_eps2_to_all(eps*eps);
    g5_set_n(n);
    for (int i = 0; i < n; i += npipes) {
        if (i + npipes > n) npipes = n - i;
        g5_calculate_force_on_x(x+i,acc+i,pot+i,npipes);
    }
    g5_close();
    // orbital integration
}
```

イメージ




```
for (i = 0; i < n; i += npipes){
    if (i + npipes > n) npipes = n - i;
    g5_calculate_force_on_x(x+i,acc+i,pot+i,npipes);
}
```

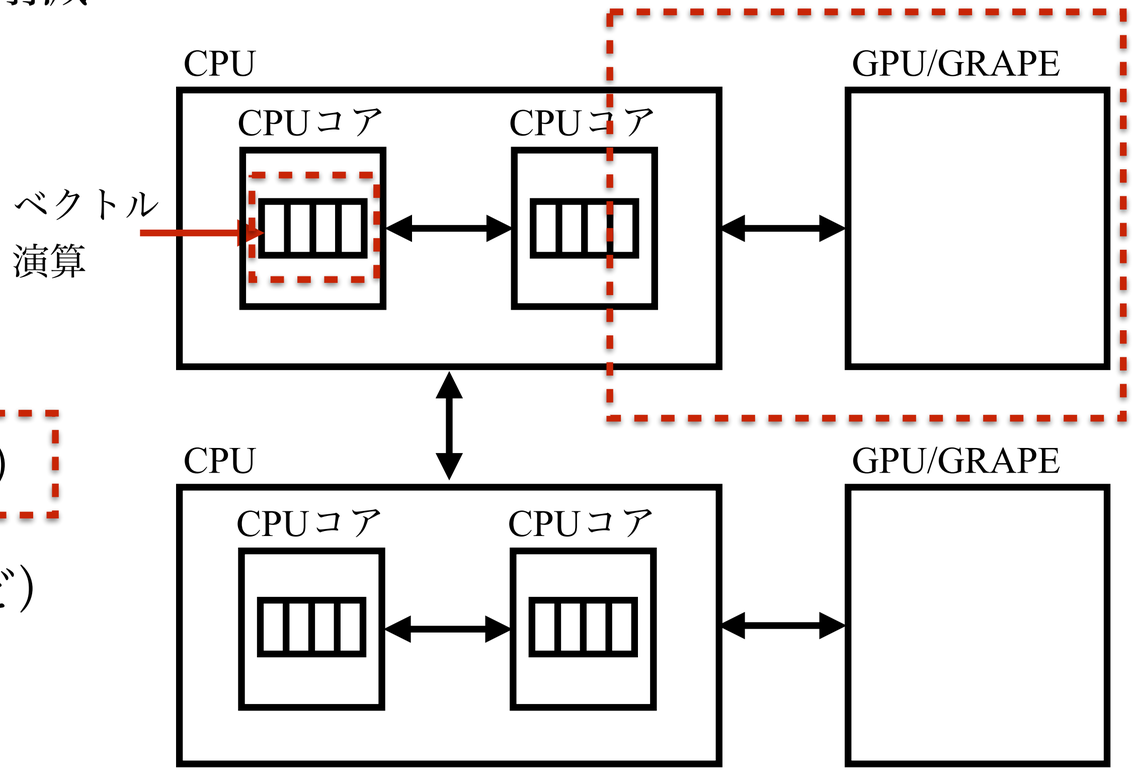
4

概要

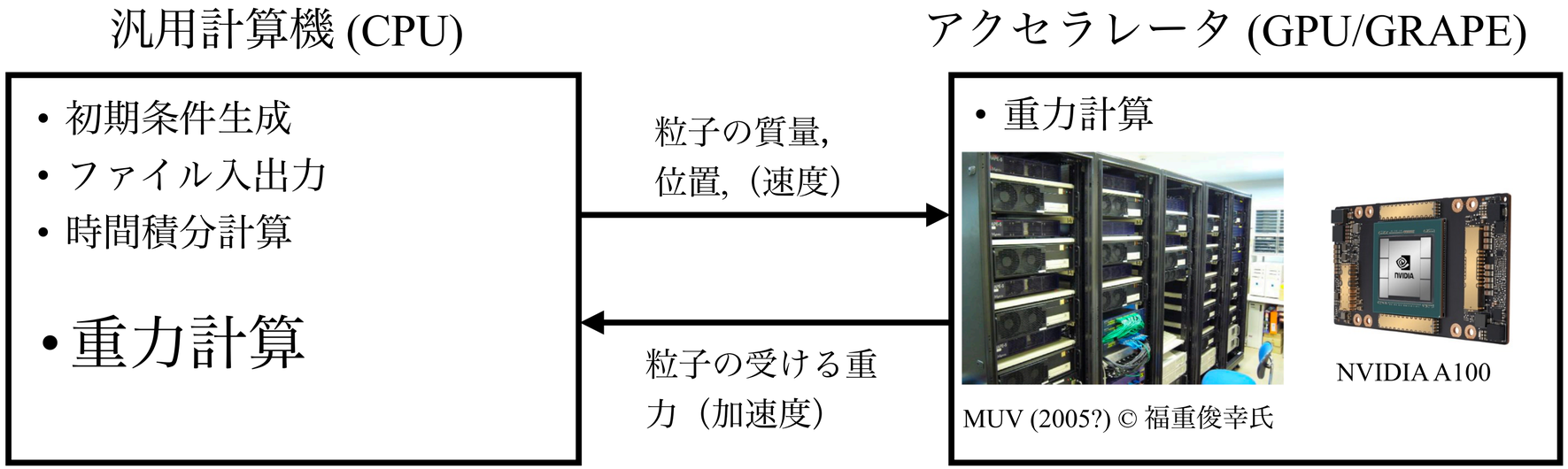
- イントロダクション
- GPUクラスタでのN体シミュレーションGPUライブラリ
- Phantom-GRAPE
- FDPS: Framework for Developing Particle Simulator
- 衝突系のN体シミュレーションコード

高速化の手法（再掲）

- アルゴリズム的な計算量の削減
  - ツリー法，P³M法など
  - 独立時間刻み法など
- 重力計算の並列化
  - ベクトル演算（AVXなど）
  - CPUコア間（OpenMPなど）
  - CPU間（MPIなど）
- アクセラレータ（GPU，GRAPEなど）



Phantom-GRAPEのイメージ



• すべて（時間積分計算や重力計算）CPUが担当

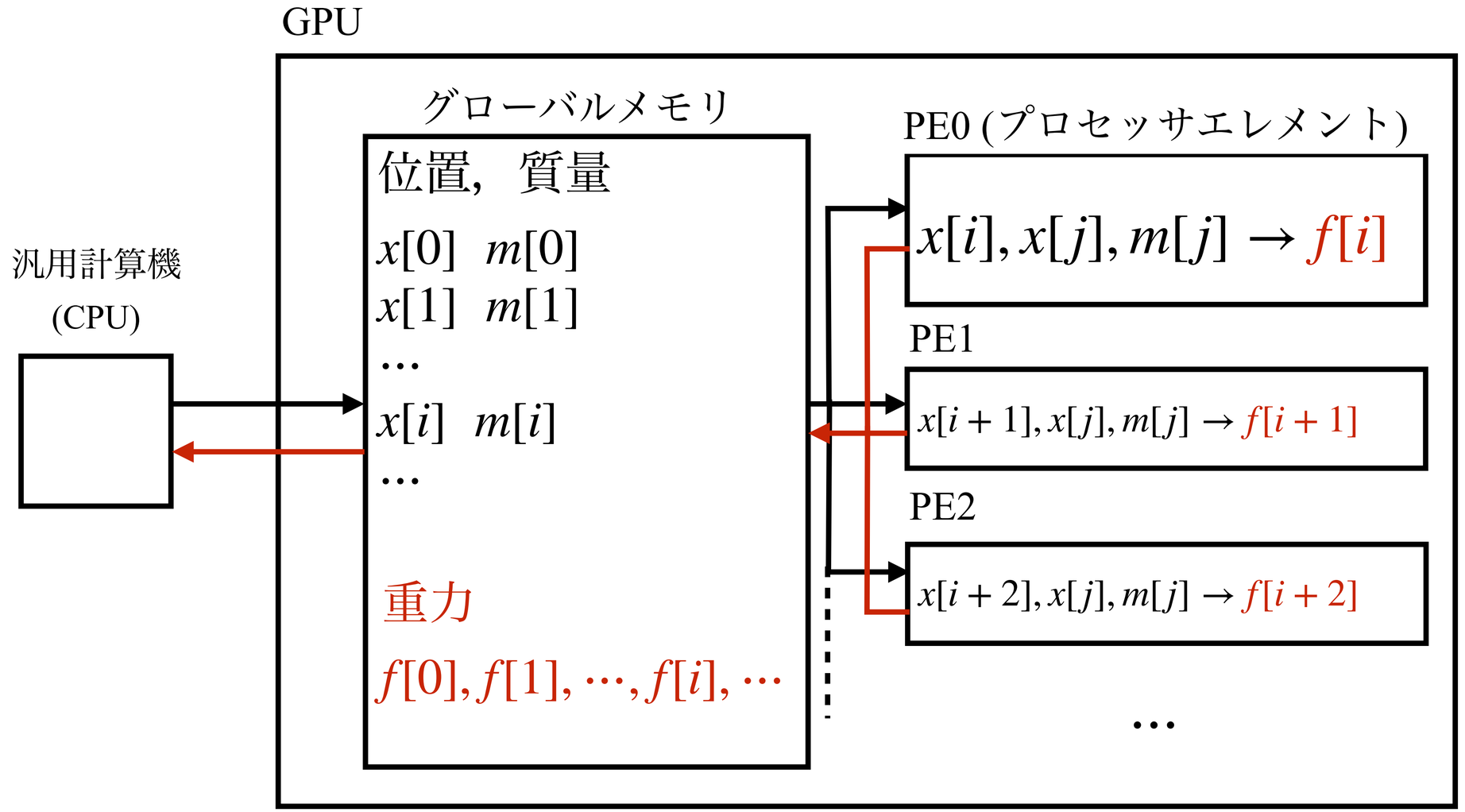
SSE/AVX (1)

- SSE (Streaming SIMD Extensions)
  - 単精度４並列，倍精度２並列のベクトル演算
  - Intel Pentium IIIから (1999年)
  - SSE対応Phantom-GRAPE (Nitadori et al. 2006)
- AVX (Advanced Vector eXtensions) / AVX2
  - 単精度８並列，倍精度４並列のベクトル演算
  - Intel Sandy Bridgeから (2011年), AVX2はIntel Haswellから (2013年)
  - AVX対応Phantom-GRAPE (Tanikawa et al. 2012; 2013)

SSE/AVX (2)

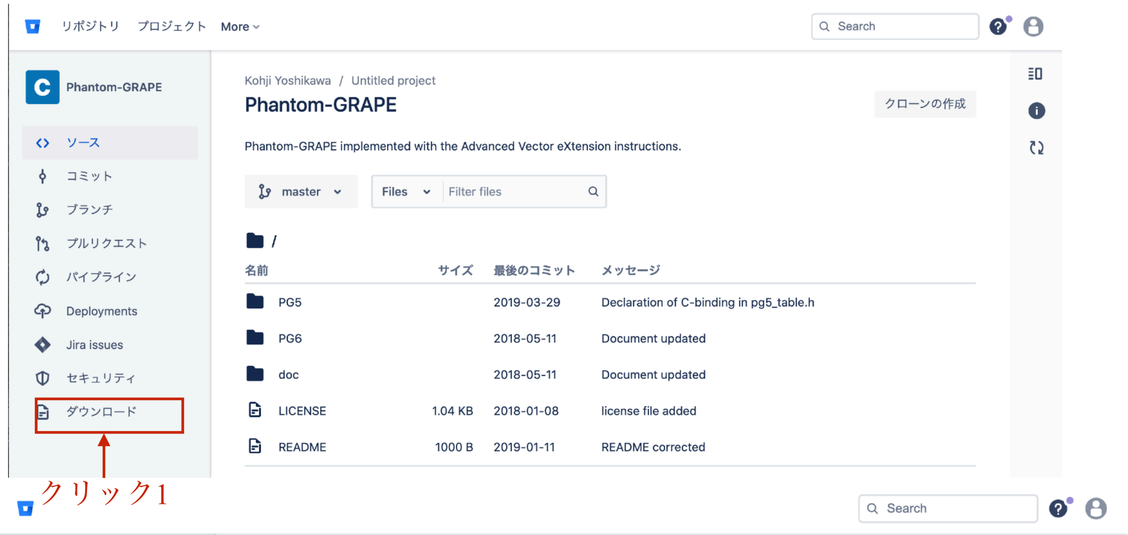
- AVX512
  - 単精度１６並列，倍精度８並列のベクトル演算
  - Intel Knights Landing (第２世代Xeon Phi)やサーバー版 Intel Skylakeから (2016-2017年頃)
  - AVX512対応Phantom-GRAPE (Yoshikawa, Tanikawa 2018)
- Phantom-GRAPE置き場
  - https://bitbucket.org/kohji/phantom-grape/src/master/

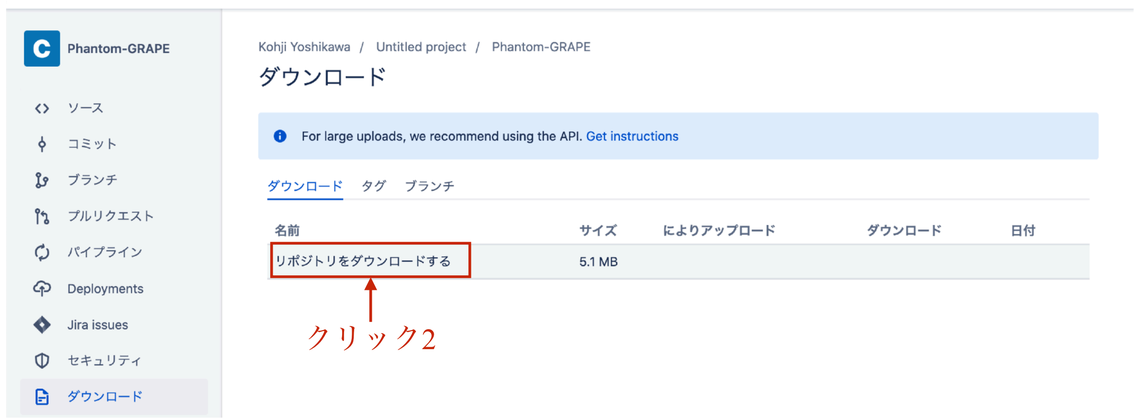
GPU での重力計算


GPU
グローバルメモリ
位置，質量
$x[0]$ $m[0]$
$x[1]$ $m[1]$
…
$x[i]$ $m[i]$
…
重力
$f[0], f[1], \cdots, f[i], \cdots$
汎用計算機
(CPU)
PE0 (プロセッサエレメント)
$x[i], x[j], m[j] \rightarrow f[i]$
PE1
$x[i+1], x[j], m[j] \rightarrow f[i+1]$
PE2
$x[i+2], x[j], m[j] \rightarrow f[i+2]$
…

使い方@GPUクラスタA100系

- 置き場: https://bitbucket.org/kohji/phantom-grape/src/master/
- kohji-phantom-grape-xxxx.zipを展開→kohji-phantom-grape-xxxxというディレクトリができる.
- ディレクトリkohji-phantom-grape-xxxx/PG5/newton/direct/に移動する.
- makeコマンド→"direct"という実行ファイルができる (デフォルトはAVX2).
- ジョブスクリプトjob.sh (右下) を作り，"sbatch job.sh"コマンドで実行する.
- N=1024のプラマーモデルの無衝突系N体シミュレーションができる．"pl1k.out"にスナップショット，stderr.logにログがでる.





job.sh →

```
#!/bin/bash
#SBATCH --partition=dgx-full
#SBATCH --gres=gpu:1
./direct init/pl1k pl1k.out >& stderr.log
```

GRAPE互換ライブラリ

- 低精度版 (PG5)
  - direct: 直接計算（銀河シミュレーション用）
  - table: テーブル計算（宇宙論シミュレーション用）
- 高精度版 (PG6)
  - 直接計算（４次精度エルミート法用）

GPUからPhantom-GRAPE

```
npipes = g5_get_number_of_pipelines();
for (step = 0; step < final_step; step++) {
    g5_open();
    g5_set_range(xmin,xmax,mmin); ←消す
    g5_set_jp(0,n,m,x);           ←g5_set_xmj(0,n,x,m);
    g5_set_eps2_to_all(eps*eps);  ←g5_set_eps_to_all(eps);
    g5_set_n(n);
    for (i = 0; i < n; i += npipes) {
        if (i + npipes > n) npipes = n - i;
        g5_calculate_force_on_x(x+i,acc+i,pot+i,npipes);
    }
    g5_close();
    // orbital integration
}
```

逆にサンプルコードを GPUクラスタのGPUで動かす場合

- ヘッダファイルの変更
  - #include “gp5util.h” → #include “g5util.h”
  - NJMAX (JMEMSIZE) → NJMAX 65536
- 廃止されたAPIの修正
  - g5_set_xmj(0,nj,xj,mj); → g5_set_jp(0, nj, mj, xj);
  - g5_set_eps_to_all(eps); → g5_set_eps2_to_all(eps*eps);
- 全エネルギーの計算の修正
  - e0 = ke + pe; → e0 = ke - pe;
  - e = ke + pe; → e = ke - pe;

Phantom-GRAPEの使い所

- Nが小さい場合（∵CPU-GPU間の通信がない）
  - 単純に必要とするNが少ない
  - ツリー法による重力相互作用数の節約時
  - 独立時間刻み法による時間積分計算の節約時
- GPUがない場合（ノートPC，デスクトップPC，CPUクラスタ，CfCA XC50など）
- 重力計算のほかに流体計算を必要とする場合 (N体+SPHなど)
- 計算能力だけでなくメモリ容量も必要な場合（宇宙論的N体など）

ただし開発・調整の余地はあると思われる

リファレンス

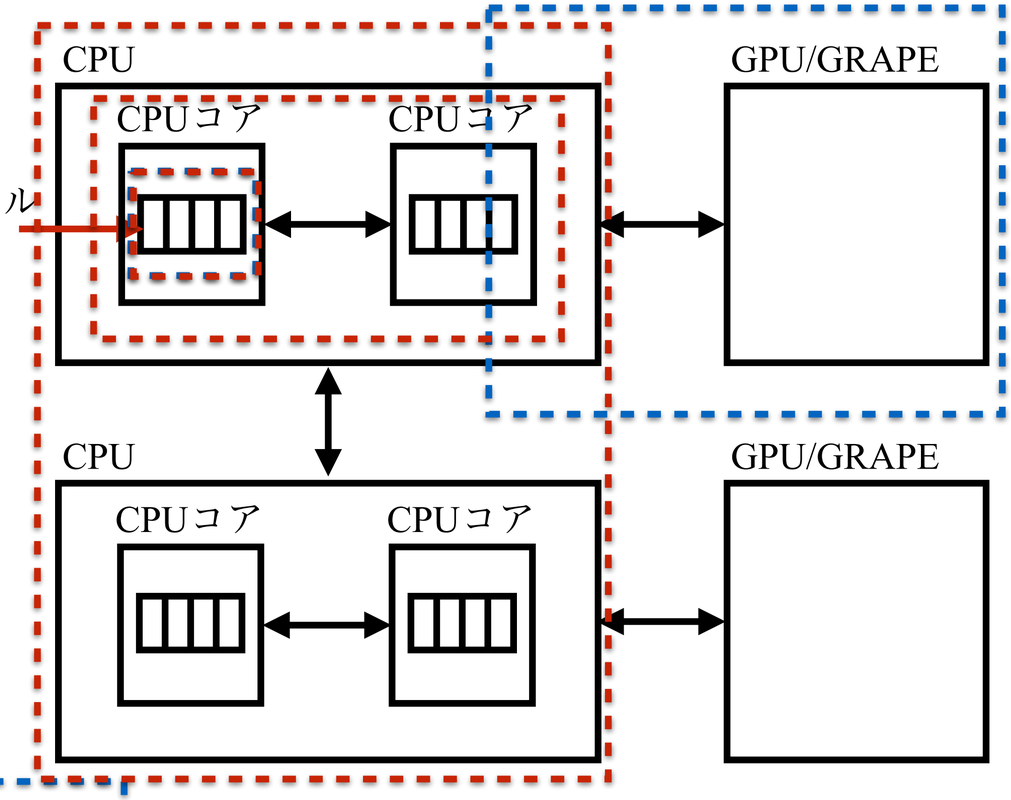
- SSE衝突系: Nitadori et al. (2006, NewA, 12, 169)
- AVX衝突系: Tanikawa et al. (2012, NewA, 17, 82)
- AVX無衝突系: Tanikawa et al. (2013, NewA, 19, 74)
- AVX-512衝突系・無衝突系: Yoshikawa, Tanikawa (2018, RNAAS, 2, 231)
- サイト: https://bitbucket.org/kohji/phantom-grape/src/master/

概要

- イントロダクション
- GPUクラスタでのN体シミュレーションGPUライブラリ
- Phantom-GRAPE
- FDPS: Framework for Developing Particle Simulator
- 衝突系のN体シミュレーションコード

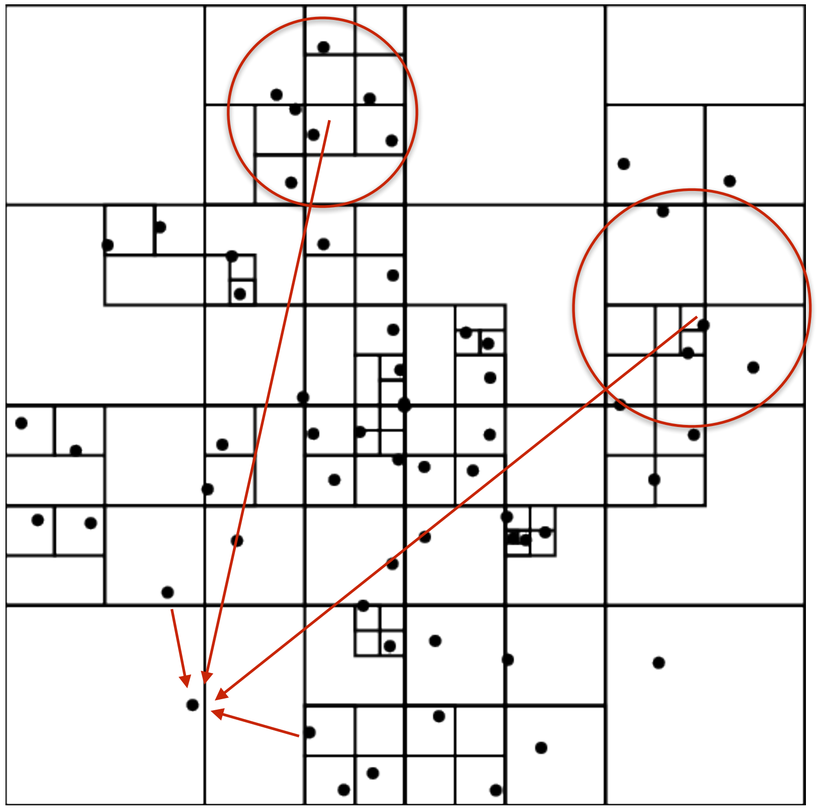
高速化の手法（再掲）

- アルゴリズム的な計算量の削減
  - ツリー法，P³M法など
  - 独立時間刻み法など
- 重力計算の並列化
  - ベクトル演算（AVXなど）
  - CPUコア間（OpenMPなど）
  - CPU間（MPIなど）
- アクセラレータ（GPU，GRAPEなど）



ツリー法

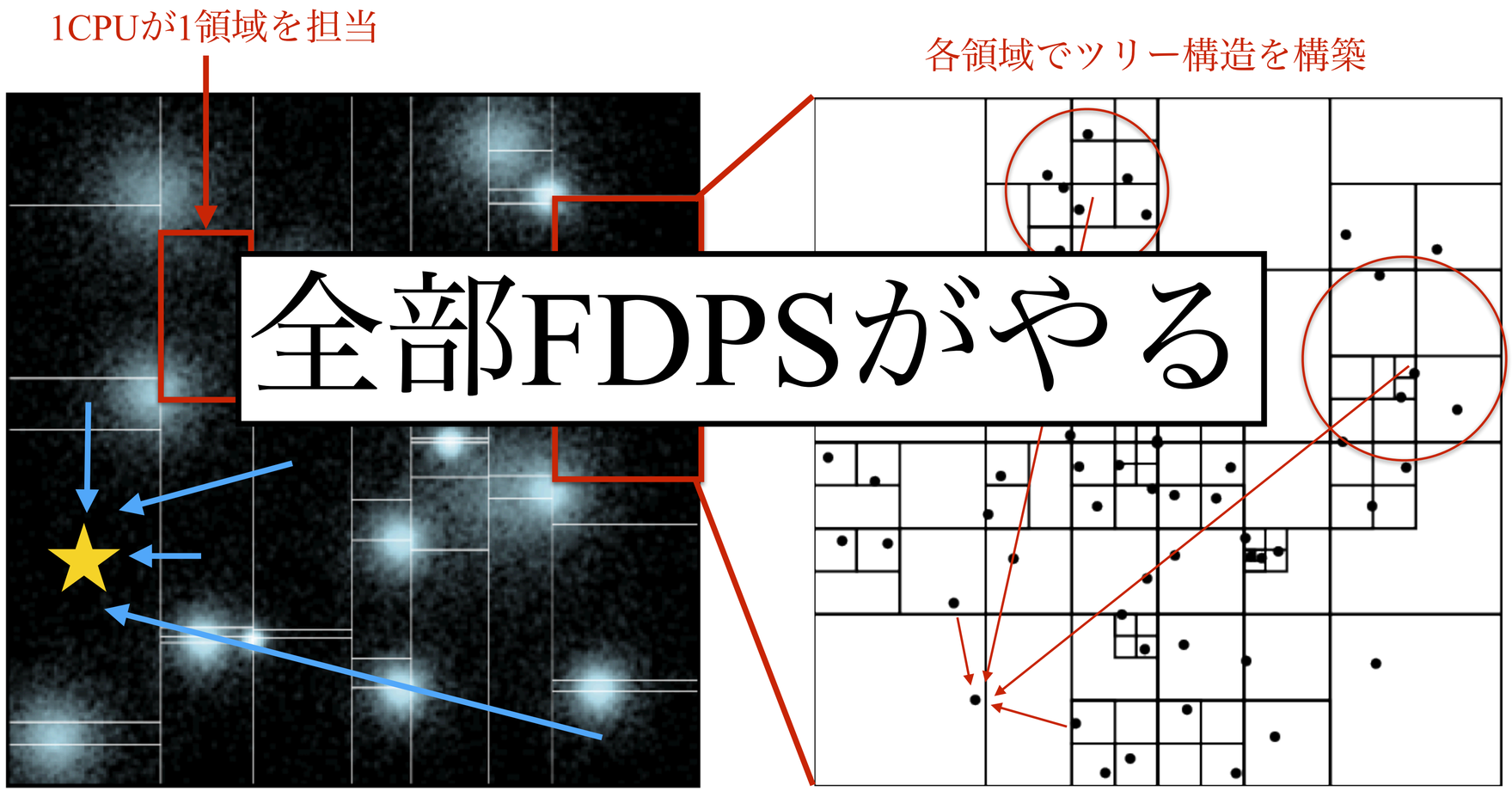
- 近くの粒子からの重力は１つ１つ計算
- 遠くの粒子からの重力はまとめて計算
- Barnes & Hut (1986)



FDPS

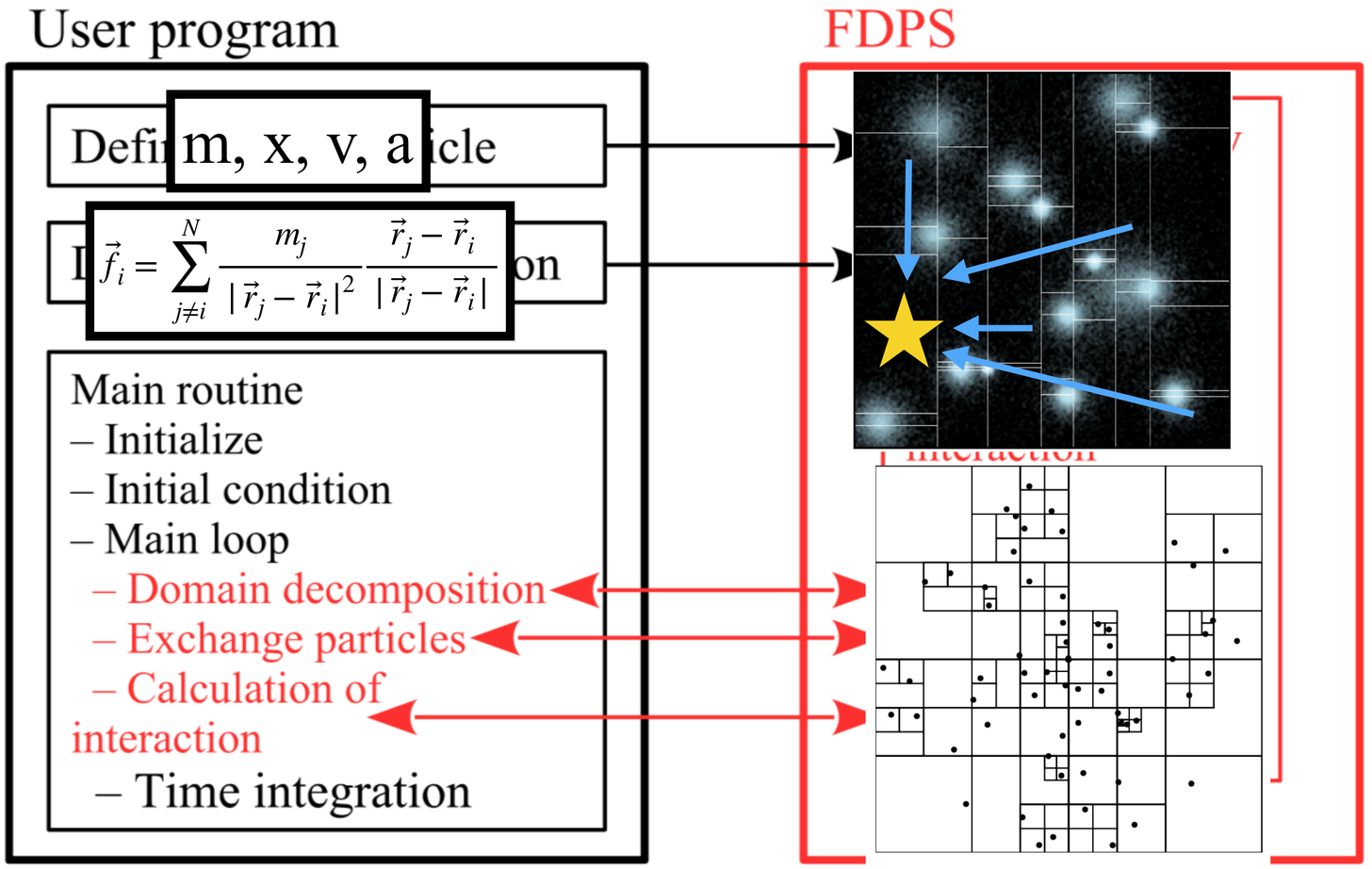
- 粒子シミュレーションコード開発を支援するフレームワーク（N体，SPH，分子動力学，粉体など）
- FDPSの担当：ロードバランス，ツリー構築，粒子の相互作用リスト構築
- ユーザーの担当：粒子の相互作用計算，粒子の時間積分
- ユーザーは並列化プログラムから解放され，よりサイエンスに集中できる

N体シミュレーションの並列化



領域分割？ツリー構築？粒子の遠近？CPU間の粒子データ通信？

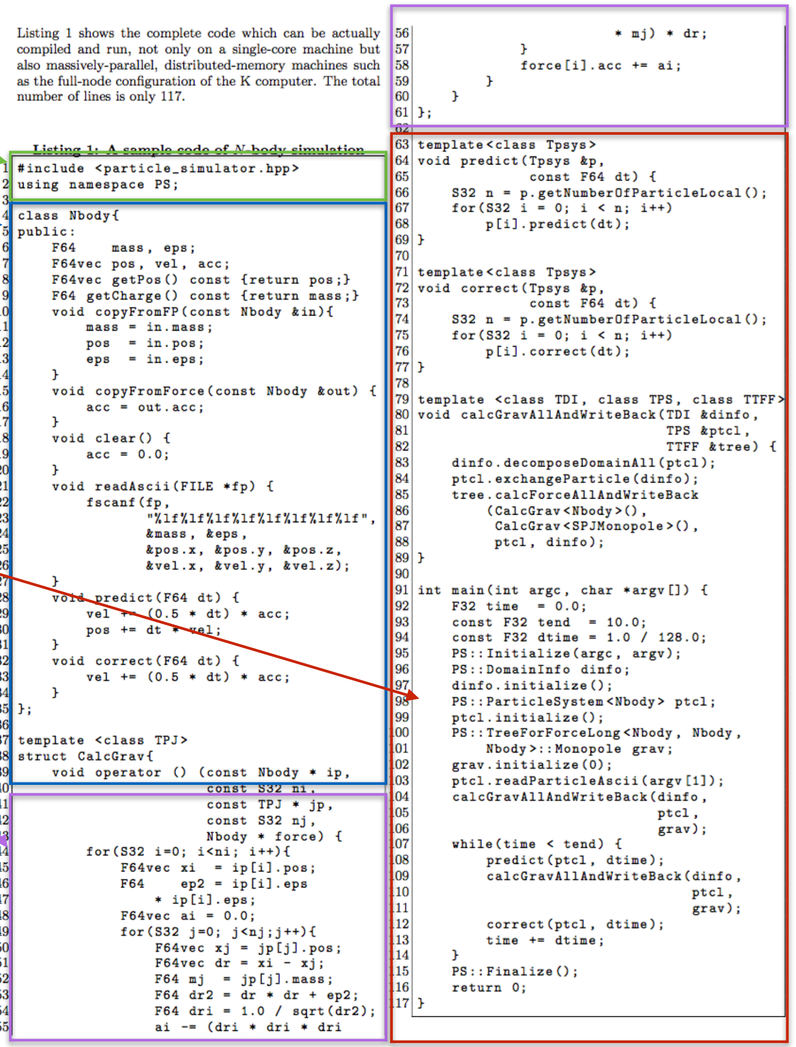
FDPSの使用モデル



Iwasawa, Tanikawa, et al. (2016, PASJ, 68, 54)

サンプルコード(N体)

- FDPSのインストール
- N体粒子の定義
- 重力間相互作用の定義
- メインルーチン
- ノートPC，デスクトップPC，PCクラスタ，CPUスパコン(CfCA XC50など)で動作するN体コード



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

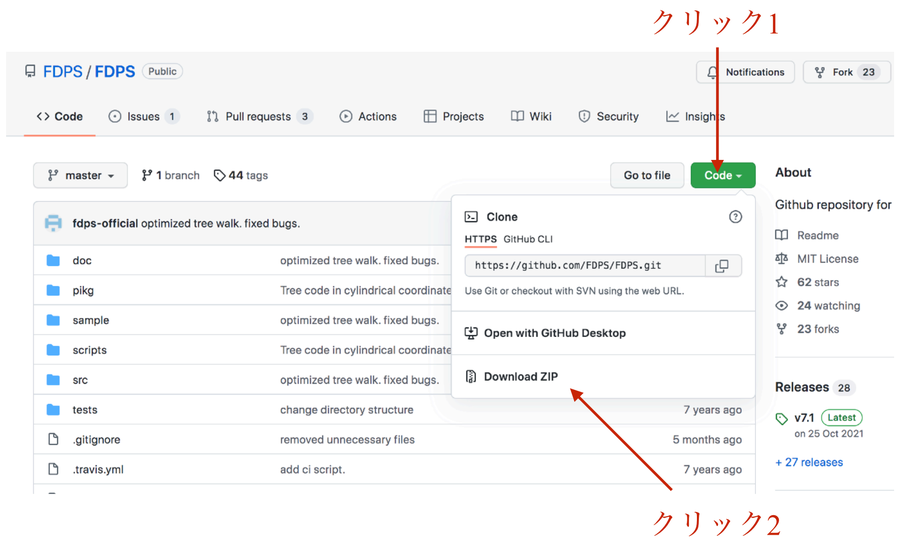
```
#include <particle_simulator.hpp>
using namespace PS;

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%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置き場

- https://github.com/FDPS/FDPS/
- ディレクトリ FDPS/sample/c++/nbodyにC++で書かれたN体コードのサンプルあり（GPUにも対応）
- c++をcやfortranに変えれば，CやFortranで書かれたN体コードのサンプルにたどり着ける（GPUには対応していないかも）

使い方@GPUクラスタA100系

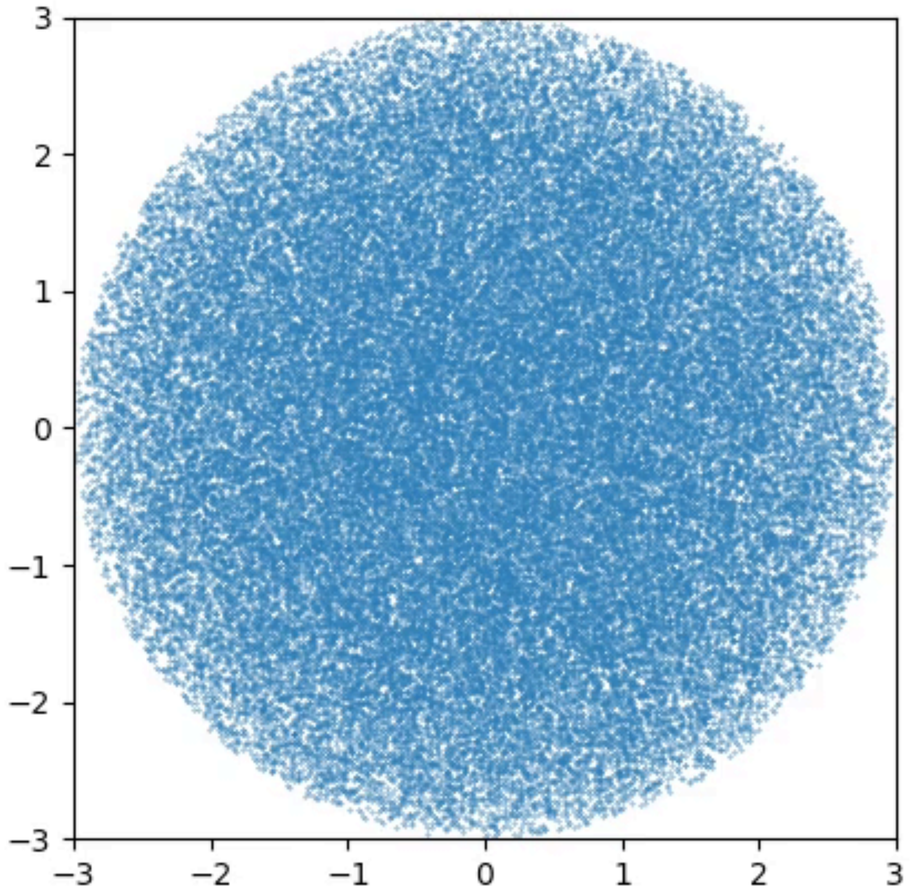
- 置き場: https://github.com/FDPS/FDPS/
- FDPS-master.zipを展開→FDPS-masterというディレクトリができる．
- ディレクトリFDPS-master/sample/c++/nbodyに移動する．
- “module load cuda-toolkit/11.4”とコマンドを実行する
- Makefileを編集する
  - #use_gpu_cuda = yesの”#”を消す（GPUの使用をオン).
  - NVCC = time $(CUDA_HOME)/bin/nvcc -std=c++11 -Xcompiler="-std=c++11 -O3”のc++11をc++17に変更（２箇所あるので注意）
- makeコマンド→”nbody.out”という実行ファイルができる（色々警告は出るがコンパイルはできる）．
- 右のようなジョブスクリプト(job.sh)を作り，”sbatch job.sh”コマンドで実行する．
- N=1024の無衝突系N体シミュレーションができる．resultディレクトリにスナップショットなどの結果が出力されている．
  - “./nbody.out” → “./nbody.out -N 65536”とするとN=65536に




```
#!/bin/bash
#SBATCH --partition=dgx-full
#SBATCH --gres=gpu:1
./nbody.out >& stderr.log
```

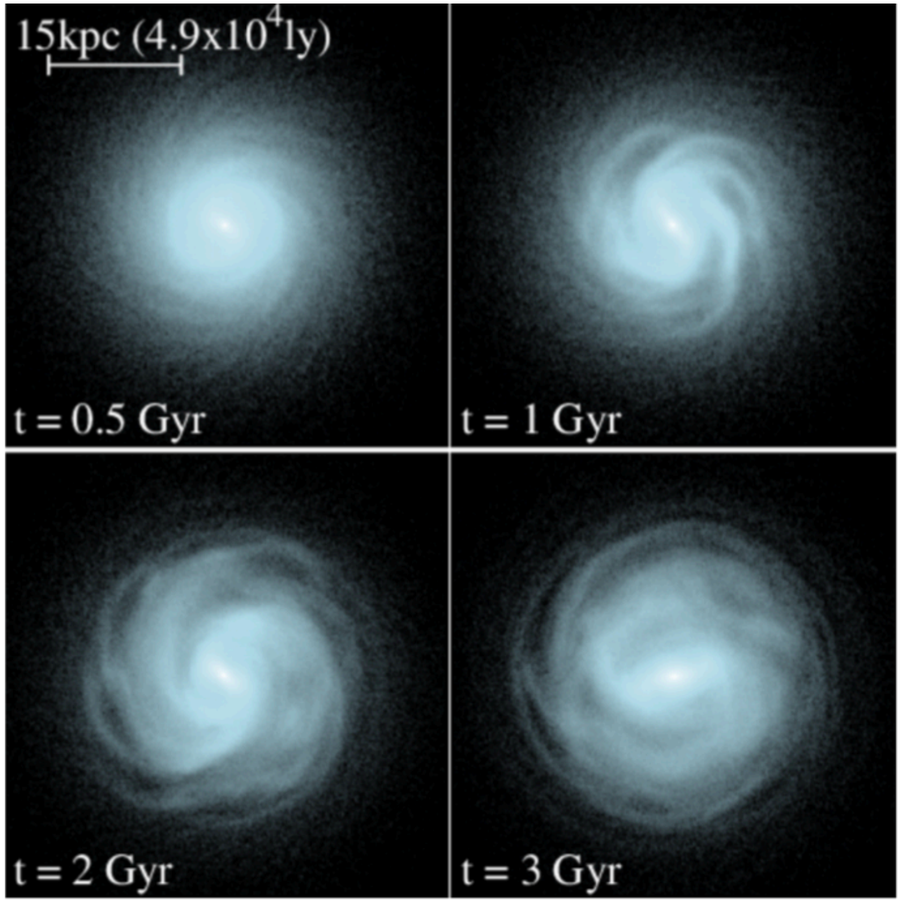
注）ここはGPUの使い方だが，OpenMPの有無，MPIの有無など
ほとんどの組み合わせで実験できる．

N=65536アニメーション

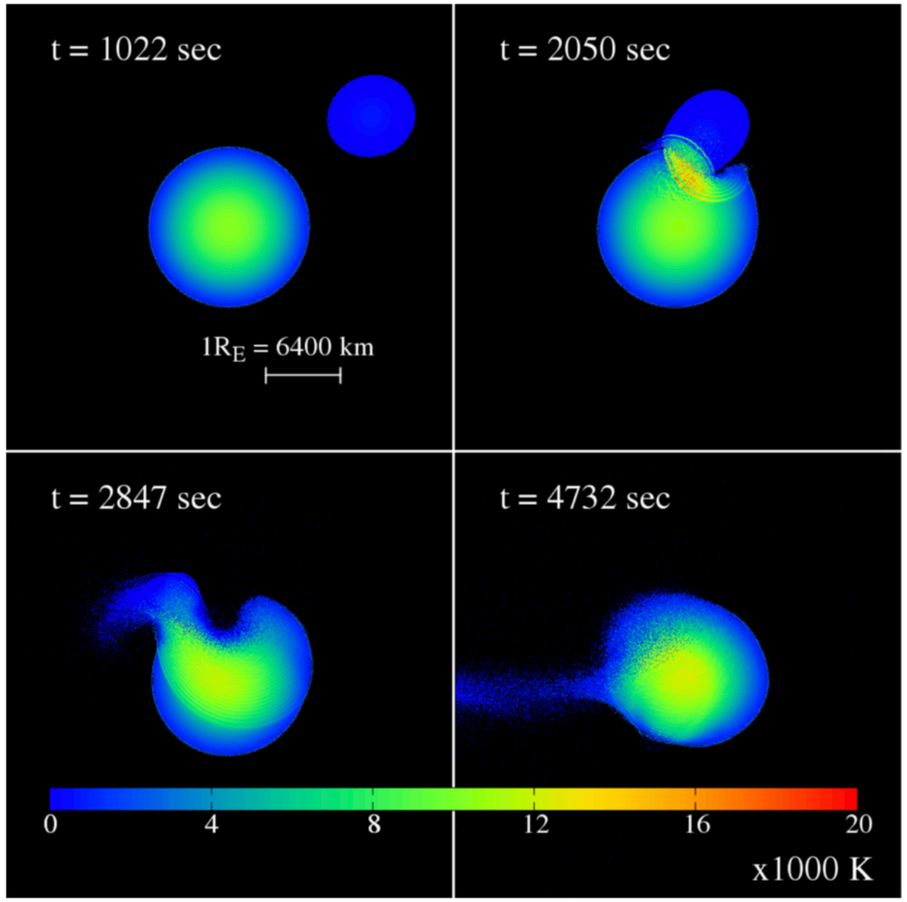


FDPS使用例

N体（銀河円盤）

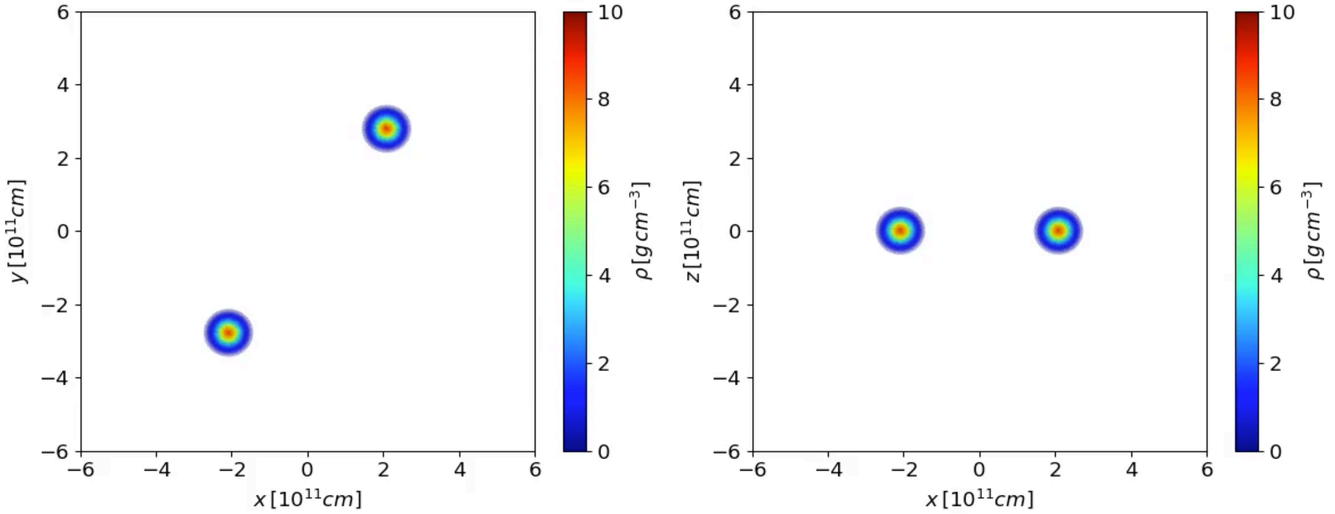


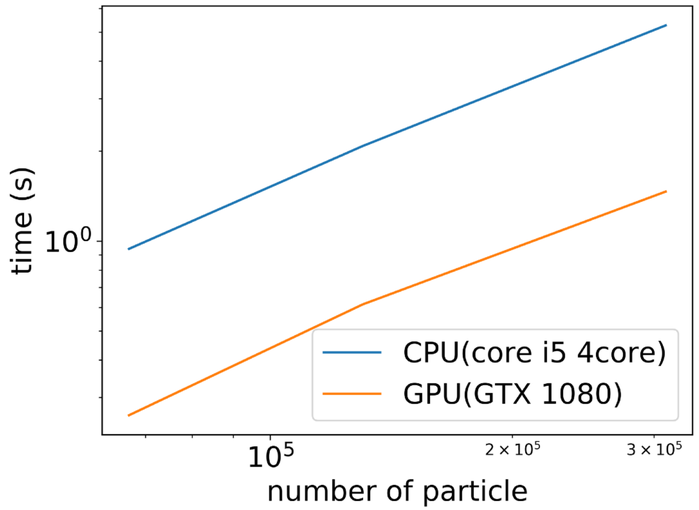
SPH（Giant impact）



Iwasawa, Tanikawa, et al. (2016, PASJ, 68, 54)

恒星衝突 (SPH GPU)





山内俊典（東大駒場修士論文）

Intel Skylake世代
NVIDIA Pascal世代

リファレンス

- オリジナル：Iwasawa, Tanikawa, et al. (2016, PASJ, 68, 54)
- Fortran対応版：Namekata et al. (2018, PASJ, 70, 70)
- GPU対応版：Iwasawa et al. (2020, PASJ, 72, 13)
- くわしくはhttps://github.com/FDPS/FDPS/
  - 日本語マニュアルがあるので読みやすい

概要

- イントロダクション
- GPUクラスタでのN体シミュレーションGPUライブラリ
- Phantom-GRAPE
- FDPS: Framework for Developing Particle Simulator
- 衝突系のN体シミュレーションコード

衝突系N体シミュレーション

- ４次精度エルミート法 (Makino, Aarseth 1992)
- 星団粒子を４次精度エルミート法，銀河粒子をリープフロッグ法＋ツリー法 (BRIDGE: Fujii et al. 2007)
- 近傍粒子との相互作用を４次精度エルミート法，遠方粒子との相互作用をリープフロッグ法＋ツリー法 (P^{3}T法: Oshino et al. 2011)
- P^{3}T法の原始惑星系円盤への応用：PENTACLE (Iwasawa et al. 2017)，GPLUM (Ishigaki et al. 2021)
- P^{3}T法の散開星団，球状星団への応用：PeTaR (Wang et al. 2020)

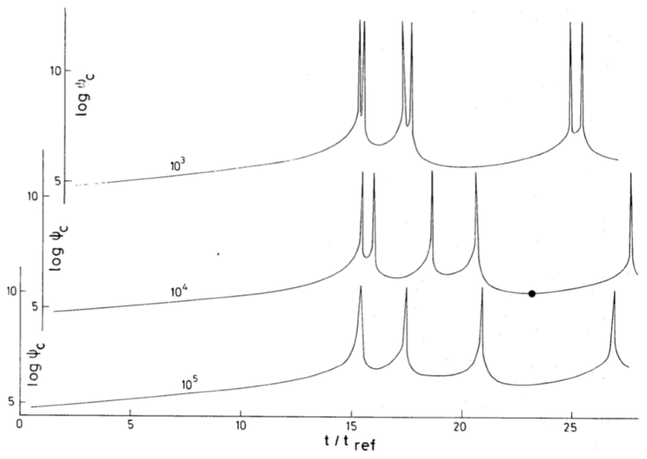
高密度星団

- ～ 10^2 – 10^6の恒星が密集した天体
- 衝突系：二体緩和時間≪寿命

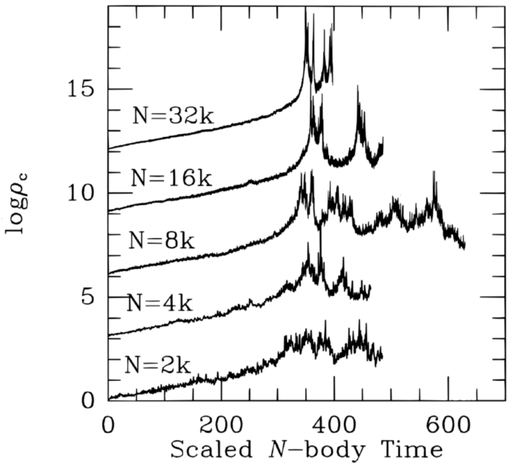
→ 高精度計算が必要

連星

- 連星は高密度星団の構造（特にコア構造）に影響大
- 星団コアのエネルギー：$E_{\rm c} \sim \dfrac{GM_{\rm c}^2}{r_{\rm c}} \sim 8.6\times10^{47}\ {\rm erg}\left(\dfrac{M_{\rm c}}{10^3 M_\odot}\right)^2\left(\dfrac{r_{\rm c}}{0.1{\rm pc}}\right)^{-1}$
- 連星のエネルギー：$E_{\rm b} \sim \dfrac{GM_{\rm b}^2}{r_{\rm b}} \sim 3.8\times10^{47}\ {\rm erg}\left(\dfrac{M_{\rm b}}{1 M_\odot}\right)^2\left(\dfrac{r_{\rm b}}{10 R_\odot}\right)^{-1}$



Bettwieser, Sugimoto (1984)

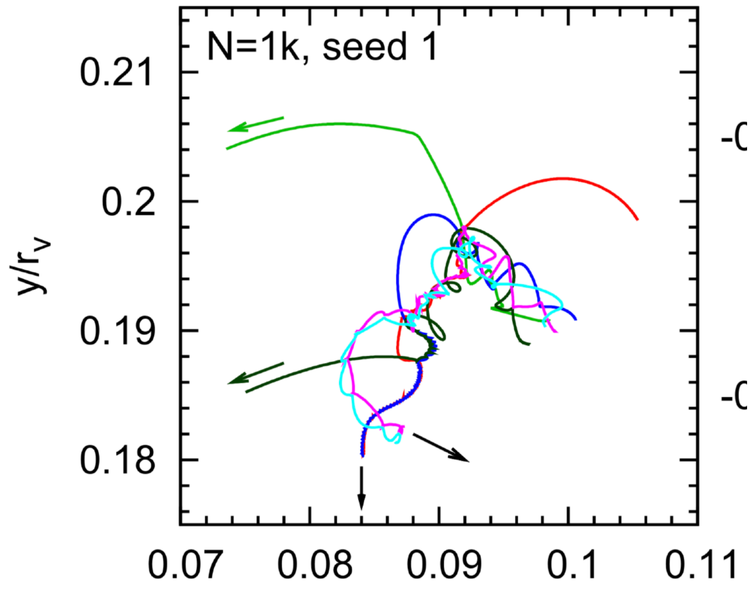


Makino (1996)

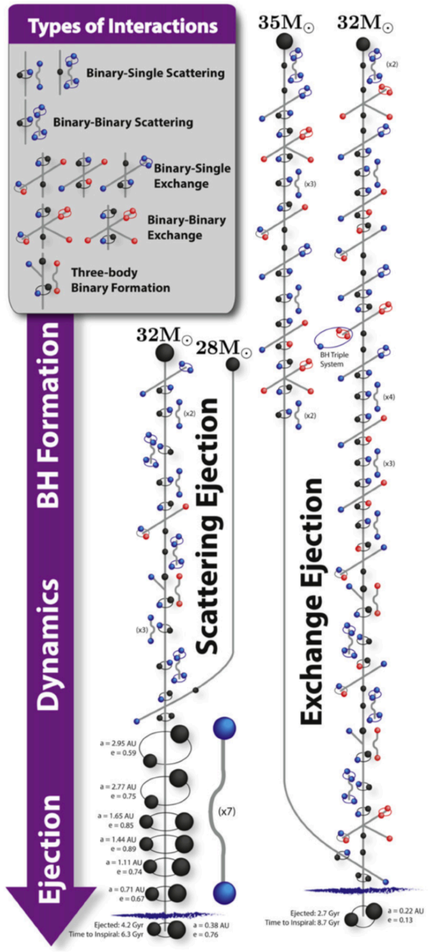
Gravothermal oscillation
(重力熱力学的コア振動)

力学的連星形成

- 自発的な連星形成
- 連星BHの形成



Tanikawa et al. (2012, NewA, 17, 272)



Rodriguez et al. (2016)

連星の（数値計算的）厄介さ

- 連星の軌道周期：

$$P = 2\pi \left(\frac{r_{\rm b}^3}{GM_{\rm b}} \right)^{1/2} \sim 3.7\ {\rm day} \left(\frac{r_{\rm b}}{10 R_\odot} \right)^{3/2} \left(\frac{M_{\rm b}}{1 M_\odot} \right)^{-1/2}$$

- 星団の横断時間：

$$t_{\rm cr} \sim \left(\frac{r_{\rm cl}^3}{GM_{\rm cl}} \right)^{1/2} \sim 1.5 \times 10^4\ {\rm yr} \left(\frac{r_{\rm cl}}{1 {\rm pc}} \right)^{3/2} \left(\frac{M_{\rm cl}}{10^6 M_\odot} \right)^{-1/2}$$

- 球状星団の寿命：$t_{\rm cl} \gtrsim t_{\rm Hubble}$

- $t_{\rm cr}/t_{\rm b} \sim 1.5 \times 10^6$, $t_{\rm cl}/t_{\rm b} \sim 9.9 \times 10^{11}$ ⇨ 大きな誤差に

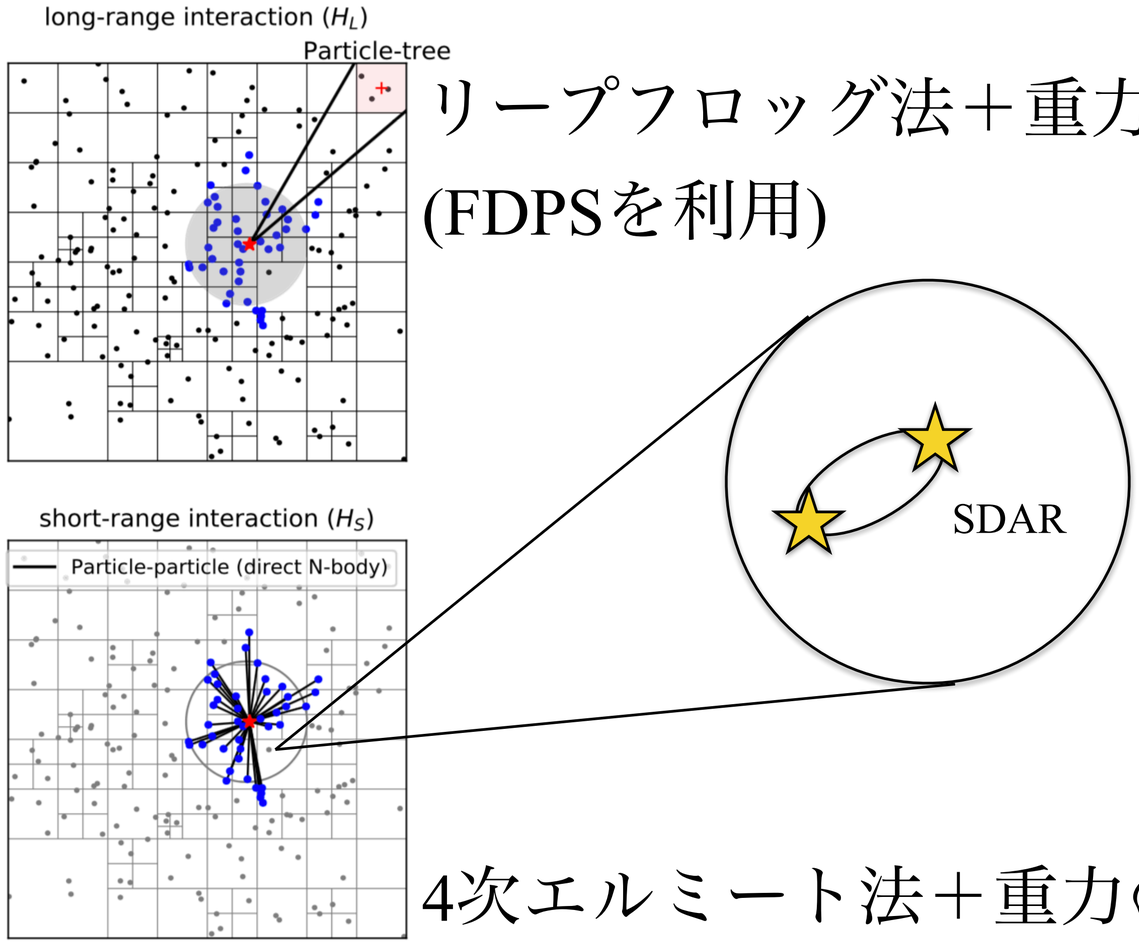
連星軌道の正則化

- 運動方程式
  - KS regularization (Kustaanheimo, Stiefel 1965)
  - KS regularization + slow-down method (Mikkola, Aarseth 1996)
- 時間積分公式
  - Algorithmic regularization (Mikkola, Tanikawa 1999; Preto, Tremaine 1999)
  - Algorithmic regularization + slow-down method (SDAR: Wang et al. 2020)

SDAR

- Algorithmic Regularization (Mikkola, Tanikawa 1999; Preto, Tremaine 1999)
  - シンプレクティック時間積分法
  - タイムステップΔtは可変で，仮想的タイムステップΔsが不変
- The slow-down method (Mikkola, Aarseth 1996)
  - 他の星からの摂動による軌道要素の変化を維持しつつ，連星の軌道周期を人工的に長くする
  - 連星軌道の追跡に時間を割かれないようにする

PeTaR コード



リープフロッグ法＋重力のツリー計算

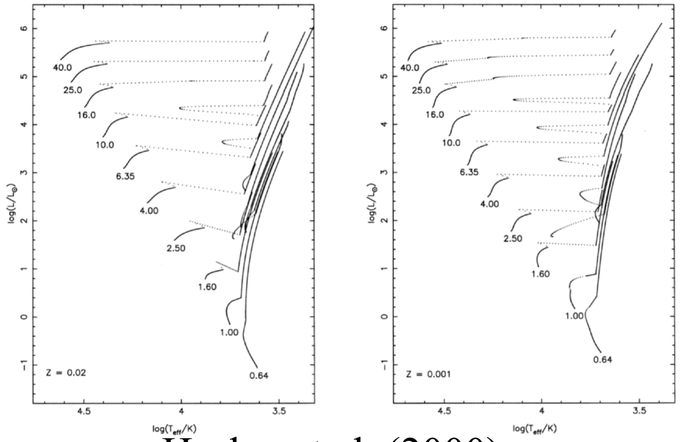
(FDPSを利用)

4次エルミート法＋重力の直接計算

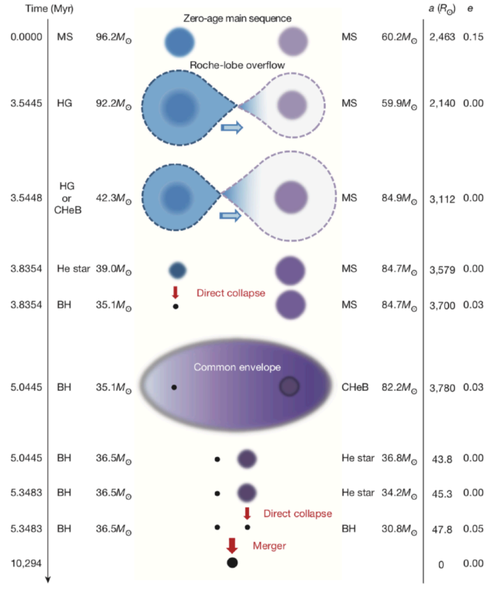
Wang et al. (2020)

恒星進化・連星進化モデル

- SSE/BSE (Hurley et al. 2000; 2002)
  - $Z = 0.0001 - 0.03$
  - 様々な連星進化モデル
- MOBSE (Giacobbo et al. 2018)
  - より現代的な恒星風モデル, 超新星モデルなど
- BSEEMP (Tanikawa et al. 2020)
  - $Z = 2 \times 10^{-10} - 2 \times 10^{-6}$



Hurley et al. (2000)



Belczynski et al. (2016)

PeTaRがサポートする計算機

- SSE/AVX/AVX512，GPUに対応
  - CfCA XC50，GPUシステム両方で使用可能
- A64FXに対応
  - スーパーコンピュータ「富岳」で使用可能

使い方@GPUクラスタA100系(1)

- PeTaR: https://github.com/lwang-astro/PeTar
- SDAR: https://github.com/lwang-astro/SDAR
- PeTar-master.zipとSDAR-master.zipを解凍→ディレクトリPeTar-masterとSDAR-masterができる．
- ディレクトリPeTar-masterに移動
- “./configure --prefix=INSTALLDIR --with-fdps-prefix=FDPSDIR --with-sdar-prefix=SDARDIR CXX=g++ --enable-cuda --with-mpi=no --with-simd=avx2”を実行
  - INSTALLDIR, FDPSDIR, SDARDIRにはPeTaRをインストールするディレクトリ，FDPSDIRにはFDPSのあるディレクトリ，SDARDIRにはSDARのあるディレクトリを指定
  - “—with-mpi=no”でMPIをオフ，”—with-simd=avx2”を使用
  - 恒星進化はオフにしてある（オンにしたければ”--with-interrupt=bse”を加える）
    - bse → mobseでMOBSEを，bse→bseEmpでBSEEMPを使用可能に
- makeコマンドを実行すると，ディレクトリbuild内に”petar.omp.avx2.gpu”という実行ファイルができる．
- make installを実行すると，ディレクトリINSTALLDIRにPeTaRの実行ファイル一式がインストールされる．

使い方@GPUクラスタA100系(2)

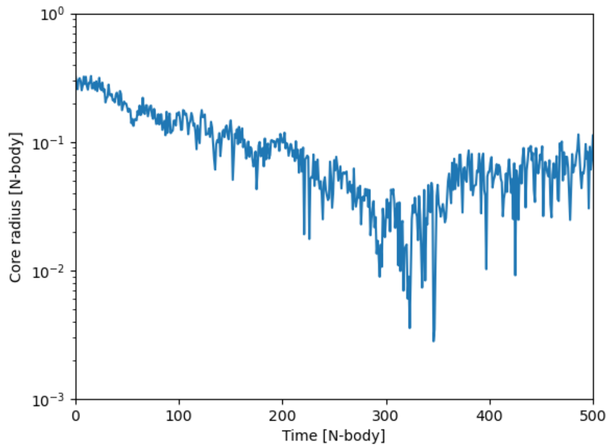
- パスの設定
  - PATHにINSTALLDIR/bin追加
  - PYTHONPATHにINSTALLDIR/include追加
- “sbatch job.sh”を実行する．
- data.0, data.1, …, data.9などのスナップショットが出力される．
- “sbatch analysis.sh”を実行すると様々なファイルができる．
- data.coreにはコア半径などの情報があり，1列目をx軸，8列目をy軸としてプロットするとコア半径の時間進化を描ける．
  - “job.sh”はT=10までしか計算しないようになっているが，見本のためT=500まで計算した．
  - job.shの実行時のオプションで”-t 500”とするとT=500まで計算できる．
  - N=1024なのでコア崩壊はあまり深くない．

job.sh

```
#!/bin/bash
#SBATCH --partition=dgx-full
#SBATCH --gres=gpu:1
petar.omp.avx2.gpu -n 1024 __Plummer >& stderr.log
```

analysis.sh

```
#!/bin/bash
#SBATCH --partition=dgx-full
#SBATCH --gres=gpu:1
module load anaconda/3
ls | egrep '^data.[0-9]+$' | sort -n -k 1.6 > file.lst
petar.data.process file.lst
```



リファレンス

- PETAR: Wang et al. (2020, MNRAS, 497, 536)
  - https://github.com/lwang-astro/PeTar
- SDAR: Wang et al. (2020, MNRAS, 493, 3398)
  - https://github.com/lwang-astro/SDAR
- マニュアルが充実しているので使いやすい

まとめ

- N体シミュレーション用GPUライブラリ@GPUクラスタ
- 汎用計算機用N体シミュレーションライブラリPhantom-GRAPE
- 粒子シミュレーションコード開発支援ツールFDPS
- 星団用N体シミュレーションコードPeTaR