

# GPU と GPAPPE ライブラリ

2021年度N体シミュレーション雨水の学校  
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
- MUVにおける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

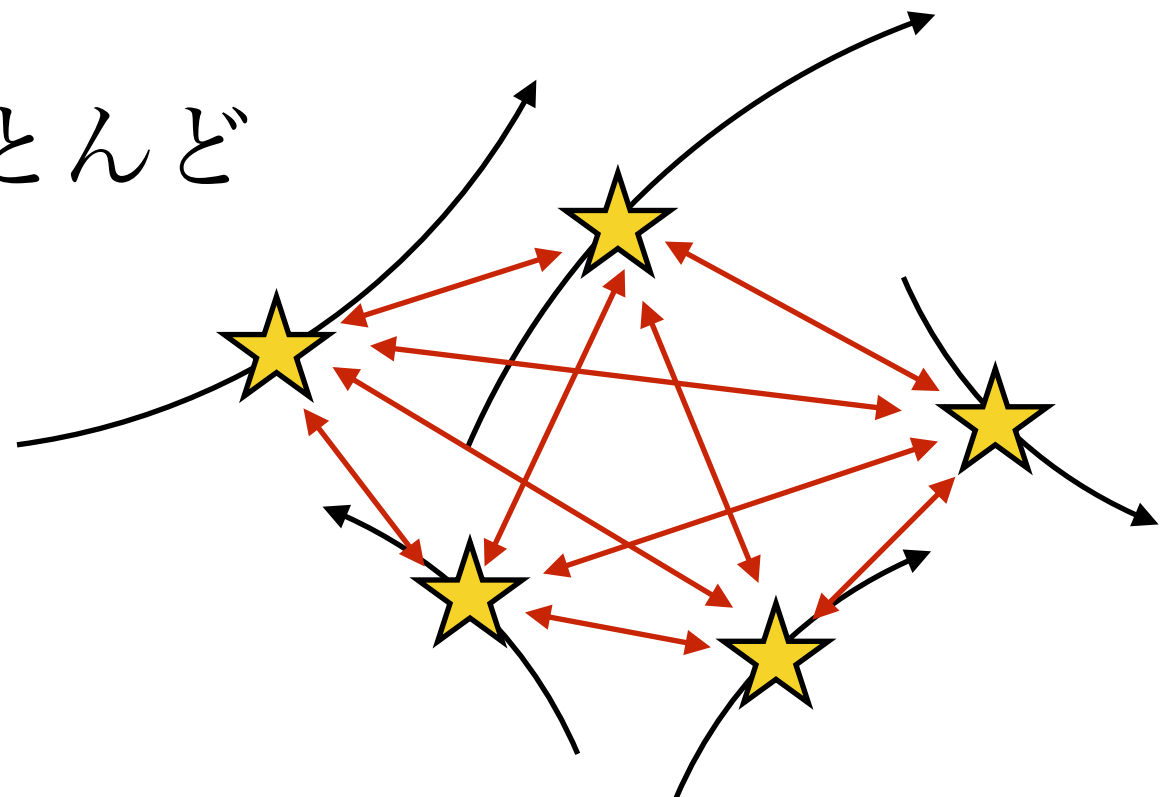
# 重力N体シミュレーション

- 運動方程式:  $\frac{d^2\vec{r}_i}{dt^2} = \vec{f}_i \dots \mathcal{O}(N)$

- 重力計算:  $\vec{f}_i = \sum_{j \neq i}^N \frac{m_j}{|\vec{r}_j - \vec{r}_i|^2} \frac{\vec{r}_j - \vec{r}_i}{|\vec{r}_j - \vec{r}_i|} \dots \mathcal{O}(N^2)$

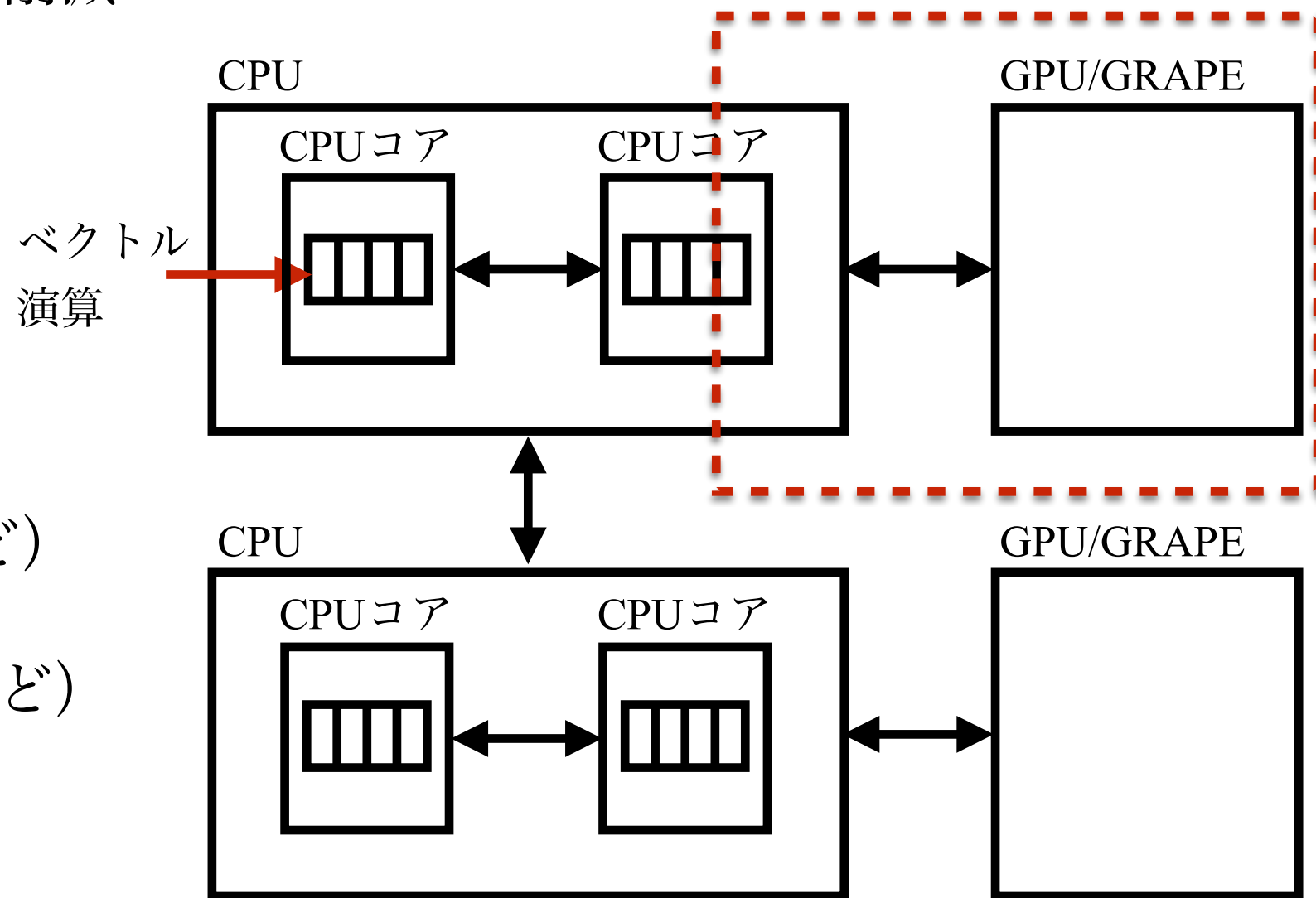
- 重力計算が全計算量のほとんど

- 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<sup>3</sup>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
- MUVにおける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

汎用計算機 (CPU)

アクセラレータ (GPU/GRAPE)

- 初期条件生成
- ファイル入出力
- 時間積分計算

粒子の質量,  
位置, (速度)

- 重力計算



NVIDIA A100

粒子の受ける重  
力 (加速度)

MUV (2005?) © 福重俊幸氏

- 計算量の多い重力計算をアクセラレータが担当
- (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

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

$$\vec{f}_i = \sum_{j \neq i}^N \frac{m_j}{|\vec{r}_j - \vec{r}_i|^2} \frac{\vec{r}_j - \vec{r}_i}{|\vec{r}_j - \vec{r}_i|}$$

$$\vec{f}_1 = \frac{m_2}{|\vec{r}_2 - \vec{r}_1|^2} \frac{\vec{r}_2 - \vec{r}_1}{|\vec{r}_2 - \vec{r}_1|} + \frac{m_3}{|\vec{r}_3 - \vec{r}_1|^2} \frac{\vec{r}_3 - \vec{r}_1}{|\vec{r}_3 - \vec{r}_1|} + \dots$$

$$\vec{f}_2 = \frac{m_1}{|\vec{r}_1 - \vec{r}_2|^2} \frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|} + \frac{m_3}{|\vec{r}_3 - \vec{r}_2|^2} \frac{\vec{r}_3 - \vec{r}_2}{|\vec{r}_3 - \vec{r}_2|} + \dots$$

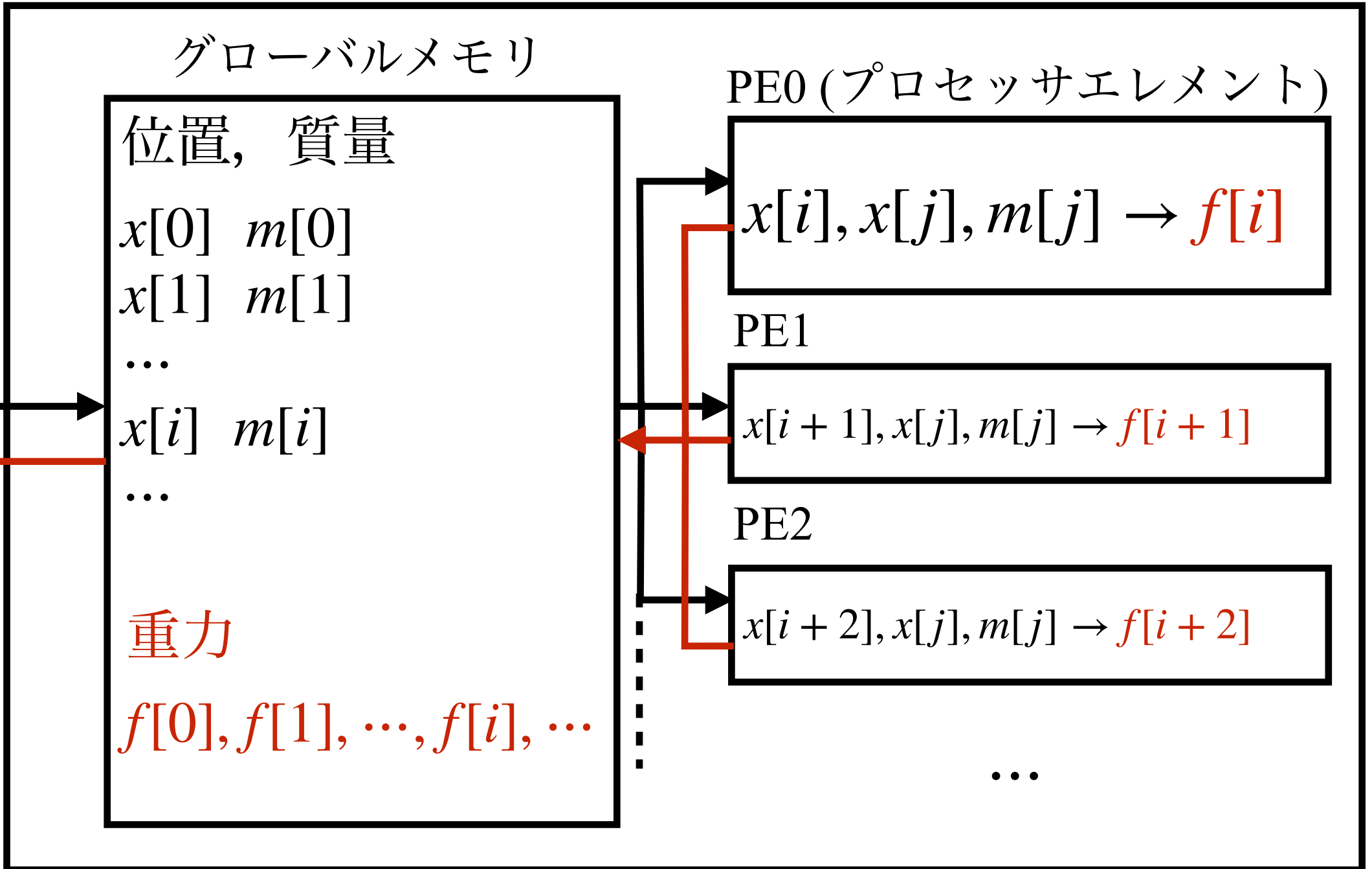
$$\vec{f}_3 = \dots$$

- $\mathcal{O}(N^2)$ 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

# GPUでの重力計算

GPU

汎用計算機  
(CPU)





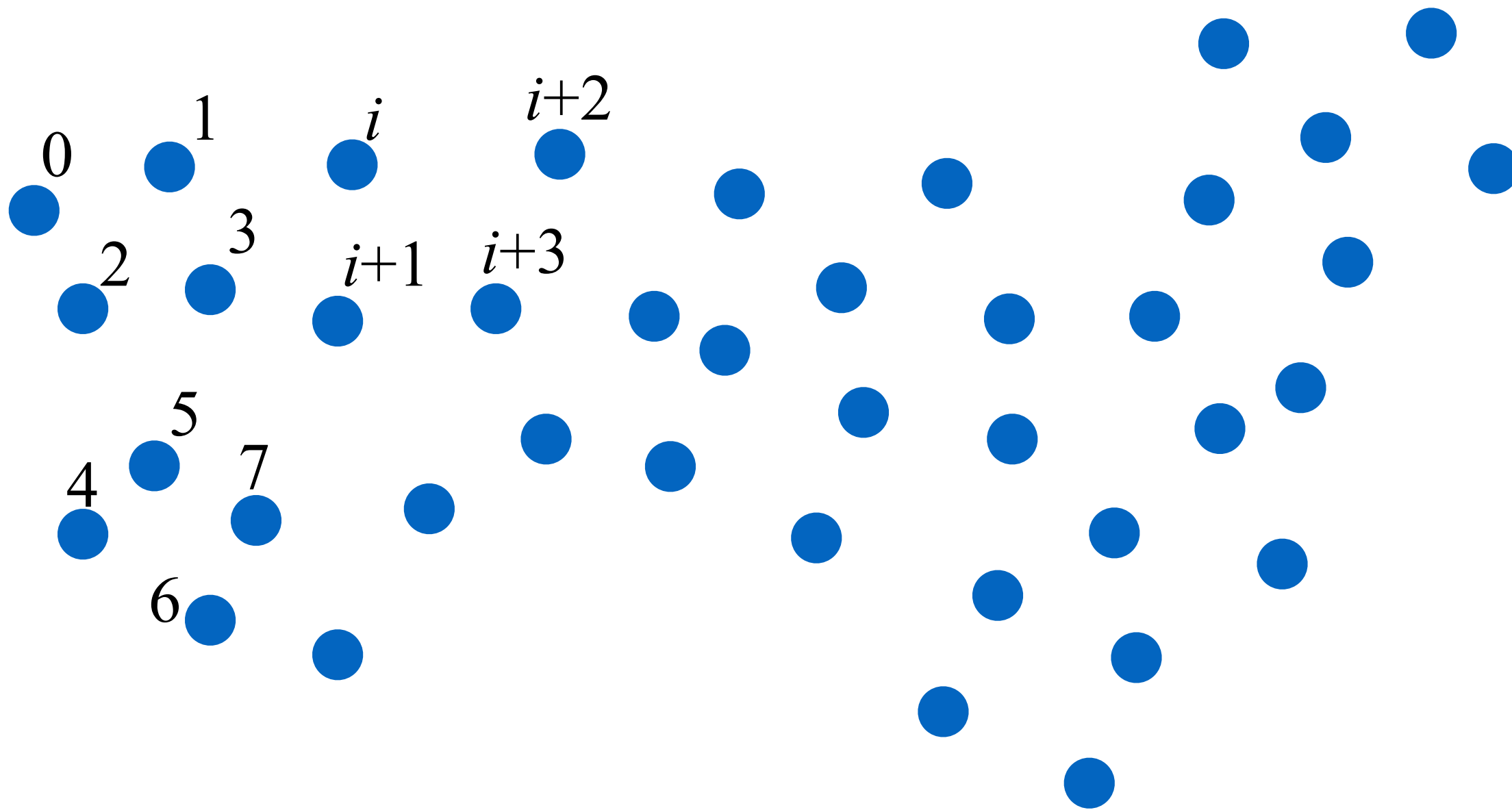
# 実際の手順

- 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 (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 (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) { ← 4  
    if (i + npipes > n) npipes = n - i;  
    g5_calculate_force_on_x(x+i, acc+i, pot+i, npipes);  
}
```

# 概要

- イントロダクション
- MUVにおける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, P<sup>3</sup>M法など

- 独立時間刻み法など

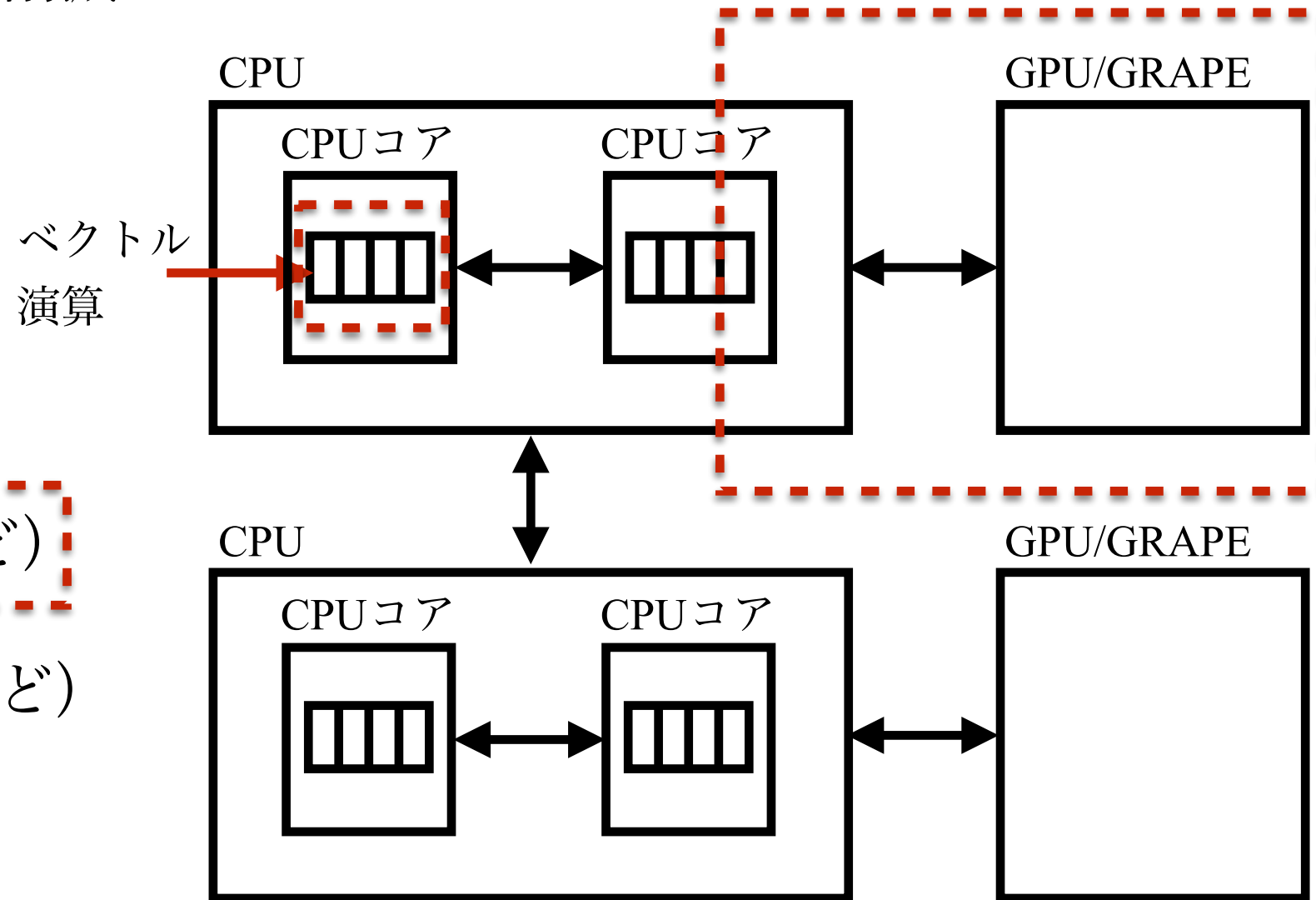
- 重力計算の並列化

- 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)



# Phantom-GRAPEのイメージ

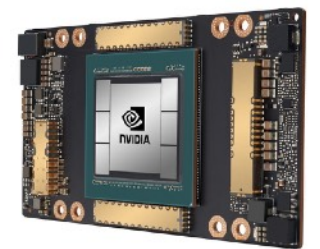
汎用計算機 (CPU)

アクセラレータ (GPU/GRAPE)

- 初期条件生成
- ファイル入出力
- 時間積分計算
- **重力計算**

粒子の質量,  
位置, (速度)

- **重力計算**



NVIDIA A100

粒子の受ける重  
力 (加速度)

MUV (2005?) © 福重俊幸氏

- **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)

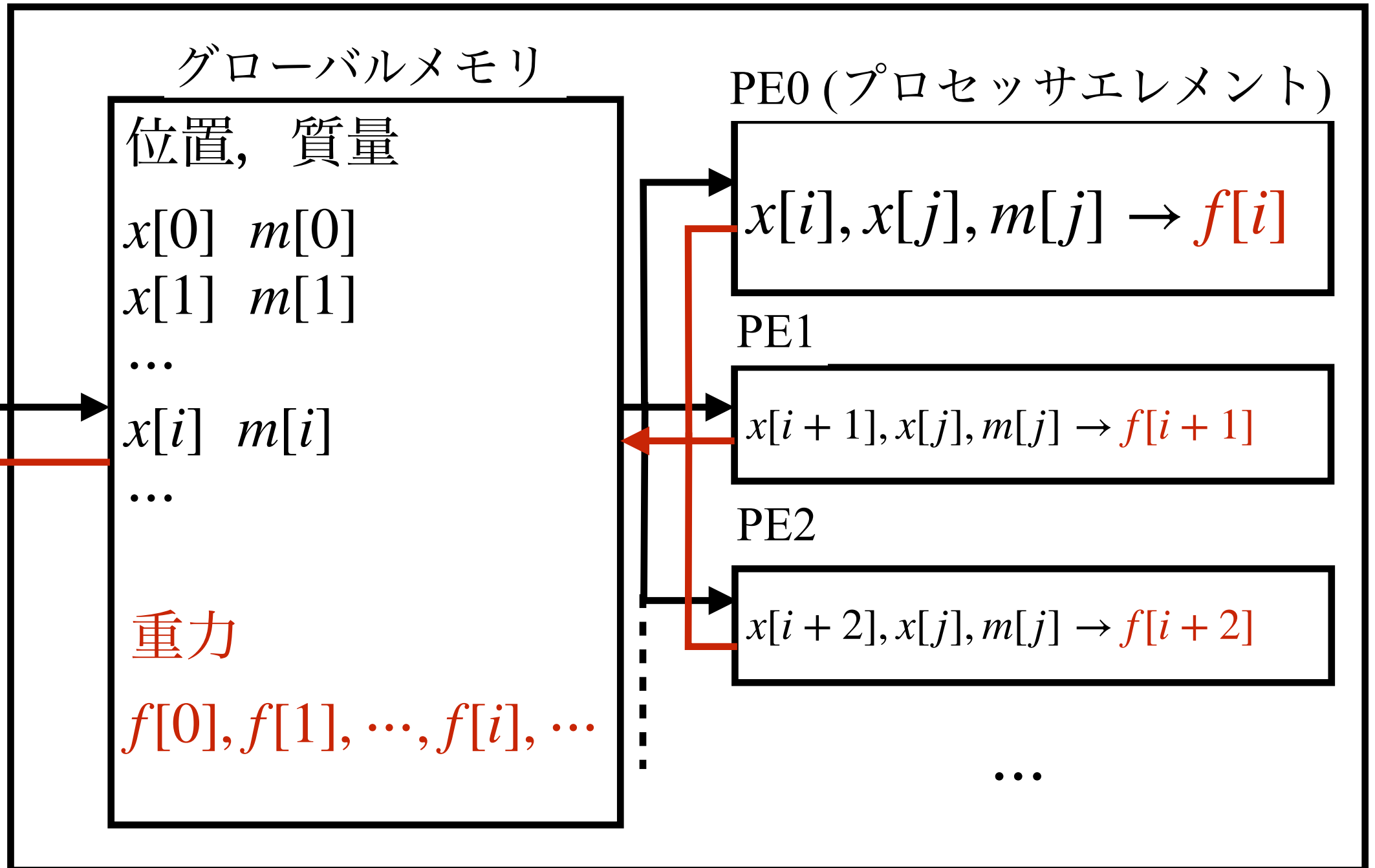
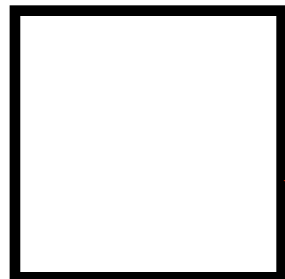
- AVX512
  - 単精度 16 並列, 倍精度 8 並列のベクトル演算
  - Intel Knights Landing (第 2 世代 Xeon Phi) やサーバー版 Intel Skylake から (2016-2017 年頃)
  - AVX512 対応 Phantom-GRAPe (Yoshikawa, Tanikawa 2018)
- Phantom-GRAPe 置き場
  - <https://bitbucket.org/kohji/phantom-grape/src/master/>



# GPU での重力計算

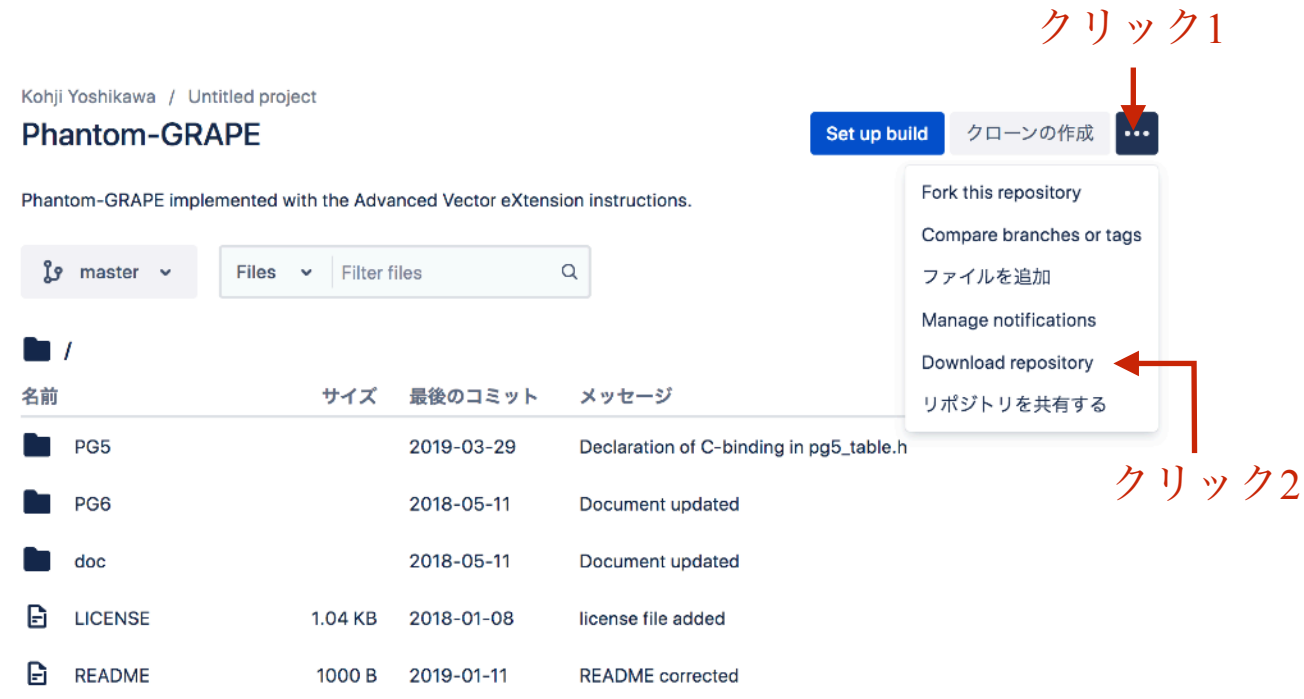
GPU

汎用計算機  
(CPU)



# 使い方@MUV

- 置き場: <https://bitbucket.org/kohji/phantom-grape/src/master/>
- muv02で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)
- 右のようなジョブスクリプト(job.sh)を作り, ”qsub 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。



Kohji Yoshikawa / Untitled project  
**Phantom-GRAPE**

Phantom-GRAPE implemented with the Advanced Vector eXtension instructions.

master Files Filter files

名前	サイズ	最後のコミット	メッセージ
PG5		2019-03-29	Declaration of C-binding in pg5_table.h
PG6		2018-05-11	Document updated
doc		2018-05-11	Document updated
LICENSE	1.04 KB	2018-01-08	license file added
README	1000 B	2019-01-11	README corrected

Set up build クローンの作成

- Fork this repository
- Compare branches or tags
- ファイルを追加
- Manage notifications
- Download repository ← クリック2
- リポジトリを共有する

クリック1

```
#!/bin/bash
#PBS -q gk20-1-debug
#PBS -l walltime=00:05:00
cd $PBS_O_WORKDIR
./direct init/pl1k pl1k.out >& stderr.log
```

# GRAPE互換ライブラリ

- 低精度版 (PG5)
  - 直接計算 (銀河シミュレーション向け)
  - テーブル計算 (宇宙論的シミュレーション向け)
- 高精度版 (PG6)
  - 直接計算 (4次精度エルミート法向け)

# 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

## MUV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);`

# 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
- MUVにおける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, P<sup>3</sup>M法など

- 独立時間刻み法など

- 重力計算の並列化

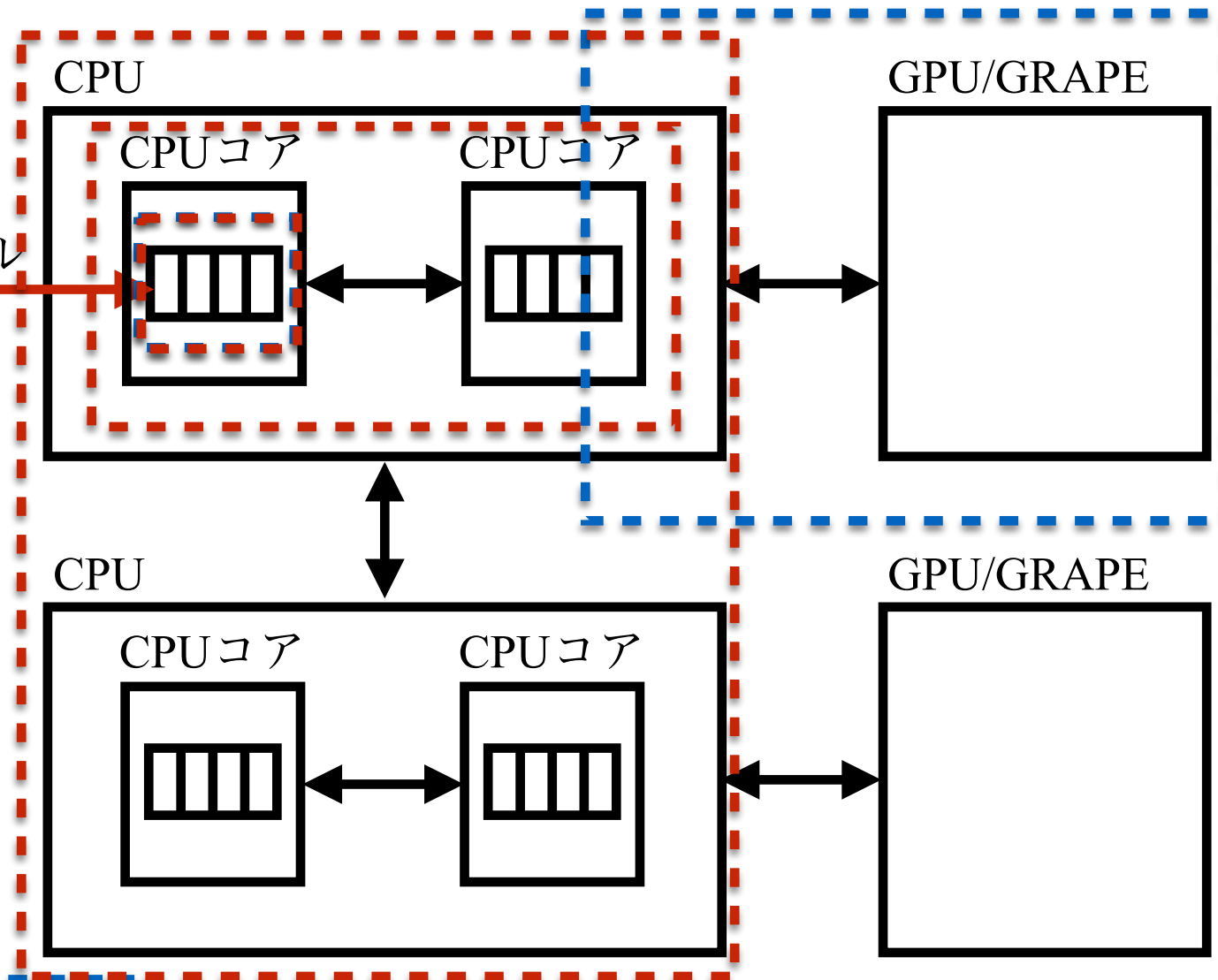
- ベクトル演算 (AVXなど)

- CPUコア間 (OpenMPなど)

- CPU間 (MPIなど)

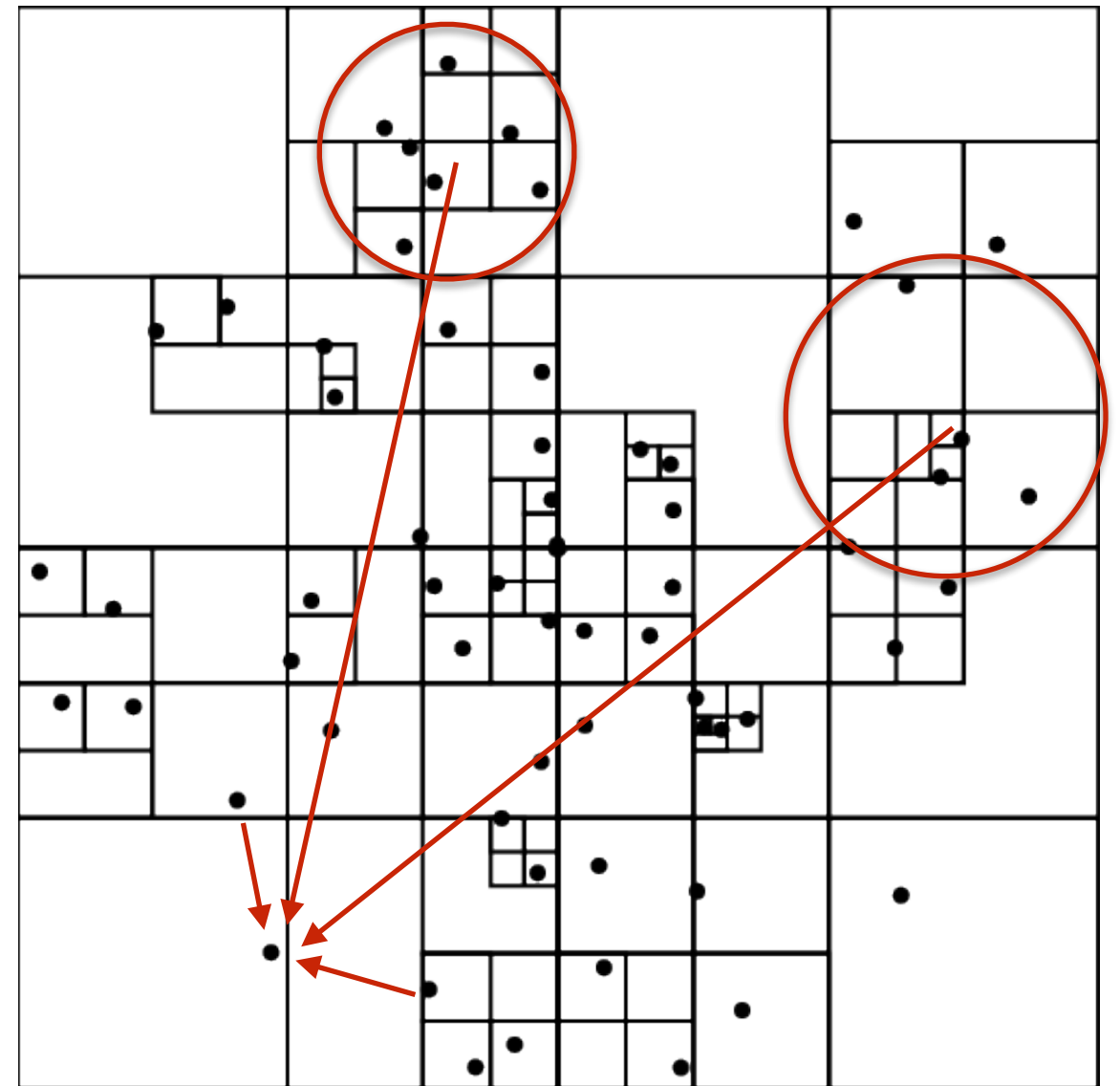
- アクセラレータ (GPU, GRAPEなど)

ベクトル  
演算



# 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)



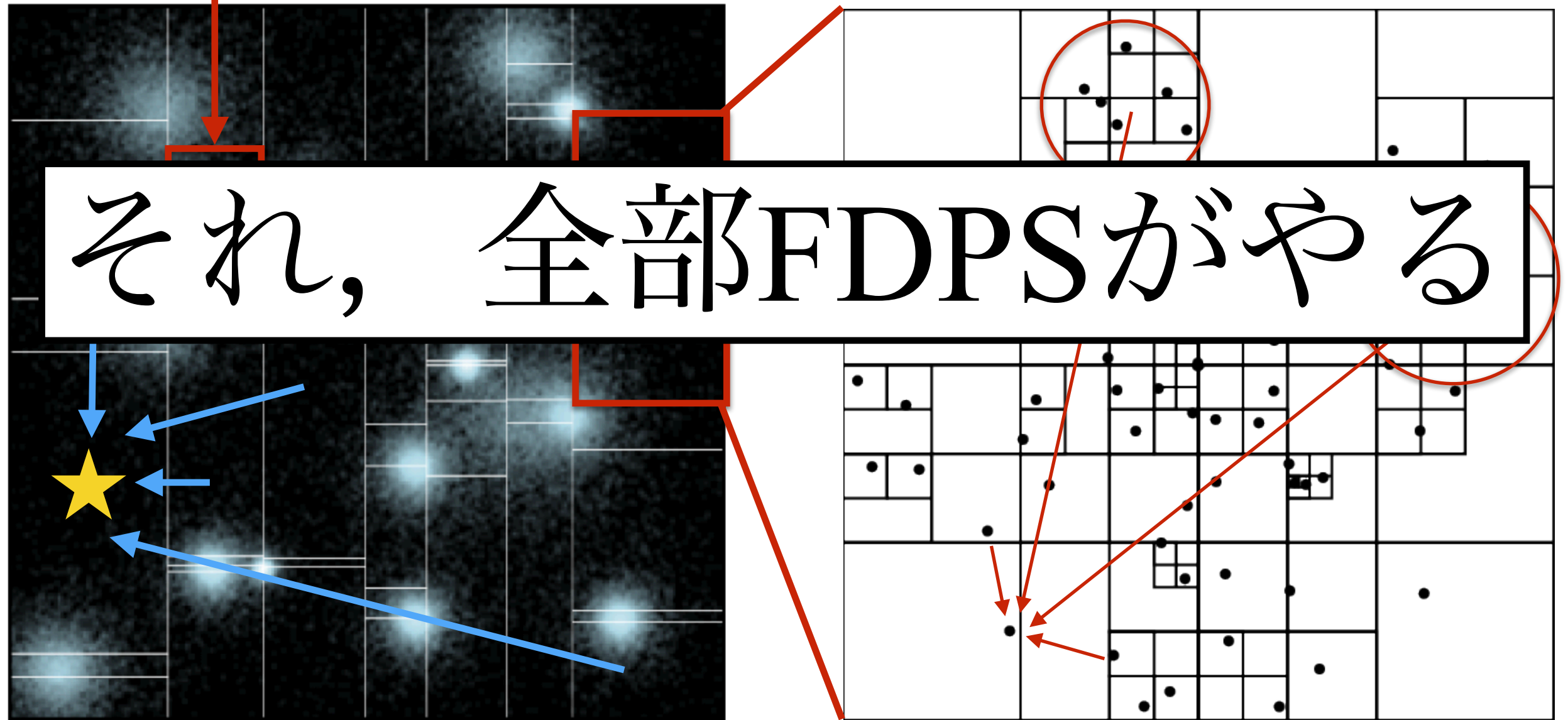
# FDPS

- 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

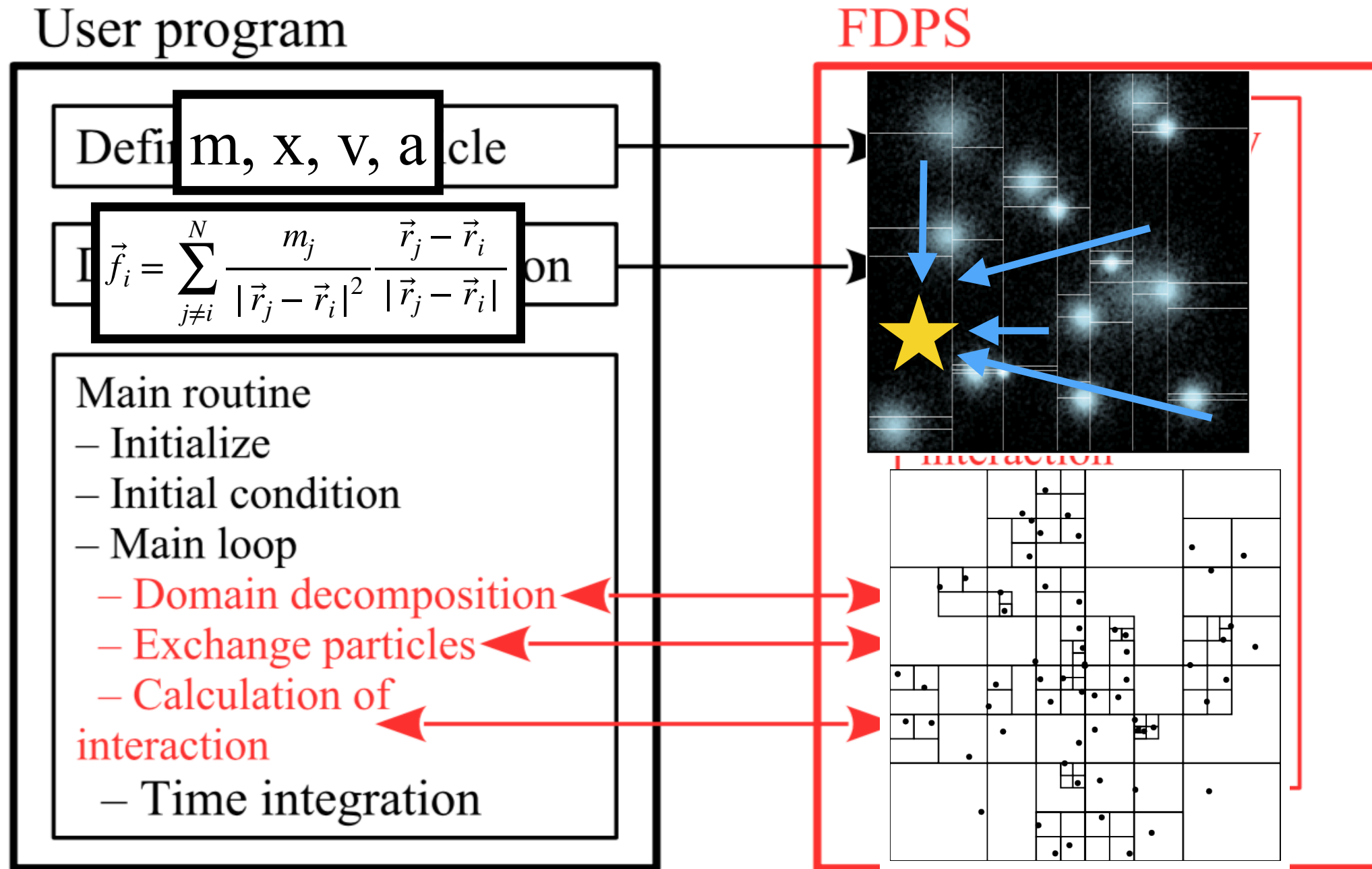
1CPUが1領域を担当

各領域でツリー構造を構築



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





# 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

```
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",
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 xi = ip[i].pos;
46             F64    ep2 = ip[i].eps
47                 * ip[i].eps;
48             F64vec ai = 0.0;
49             for(S32 j=0; j<nj; j++){
50                 F64vec xj = jp[j].pos;
51                 F64vec dr = xi - xj;
52                 F64 mj = jp[j].mass;
53                 F64 dr2 = dr * dr + ep2;
54                 F64 dri = 1.0 / sqrt(dr2);
55                 ai -= (dri * dri * dri
56
57                     * mj) * dr;
58                 force[i].acc += ai;
59             }
60         }
61     };
62 };
63
64 template<class Tpsys>
65 void predict(Tpsys &p,
66             const F64 dt) {
67     S32 n = p.getNumberofParticleLocal();
68     for(S32 i = 0; i < n; i++)
69         p[i].predict(dt);
70 }
71
72 template<class Tpsys>
73 void correct(Tpsys &p,
74             const F64 dt) {
75     S32 n = p.getNumberofParticleLocal();
76     for(S32 i = 0; i < n; i++)
77         p[i].correct(dt);
78 }
79
80 template <class TDI, class TPS, class TTF>
81 void calcGravAllAndWriteBack(TDI &dinfo,
82                             TPS &ptcl,
83                             TTF &tree) {
84     dinfo.decomposeDomainAll(ptcl);
85     ptcl.exchangeParticle(dinfo);
86     tree.calcForceAllAndWriteBack
87         (CalcGrav<Nbody>(),
88         CalcGrav<SPJMonopole>(),
89         ptcl, dinfo);
90 }
91
92 int main(int argc, char *argv[]) {
93     F32 time = 0.0;
94     const F32 tend = 10.0;
95     const F32 dt = 1.0 / 128.0;
96     PS::Initialize(argc, argv);
97     PS::DomainInfo dinfo;
98     dinfo.initialize();
99     PS::ParticleSystem<Nbody> ptcl;
100    ptcl.initialize();
101    PS::TreeForForceLong<Nbody, Nbody,
102        Nbody>::Monopole grav;
103    grav.initialize(0);
104    ptcl.readParticleAscii(argv[1]);
105    calcGravAllAndWriteBack(dinfo,
106                            ptcl,
107                            grav);
108
109    while(time < tend) {
110        predict(ptcl, dt);
111        calcGravAllAndWriteBack(dinfo,
112                                ptcl,
113                                grav);
114
115        correct(ptcl, dt);
116        time += dt;
117    }
118
119    PS::Finalize();
120    return 0;
121 }
```

# 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@MUV

- 置き場: <https://github.com/FDPS/FDPS/>

ただしFDPS最新版はC++17規格が使われているので、  
gcc-4.8.5のMUVでは1つ前のバージョンを使用  
muv02でFDPS-7.0b.zipを展開→FDPS-7.0bというディレクトリ  
ができる。

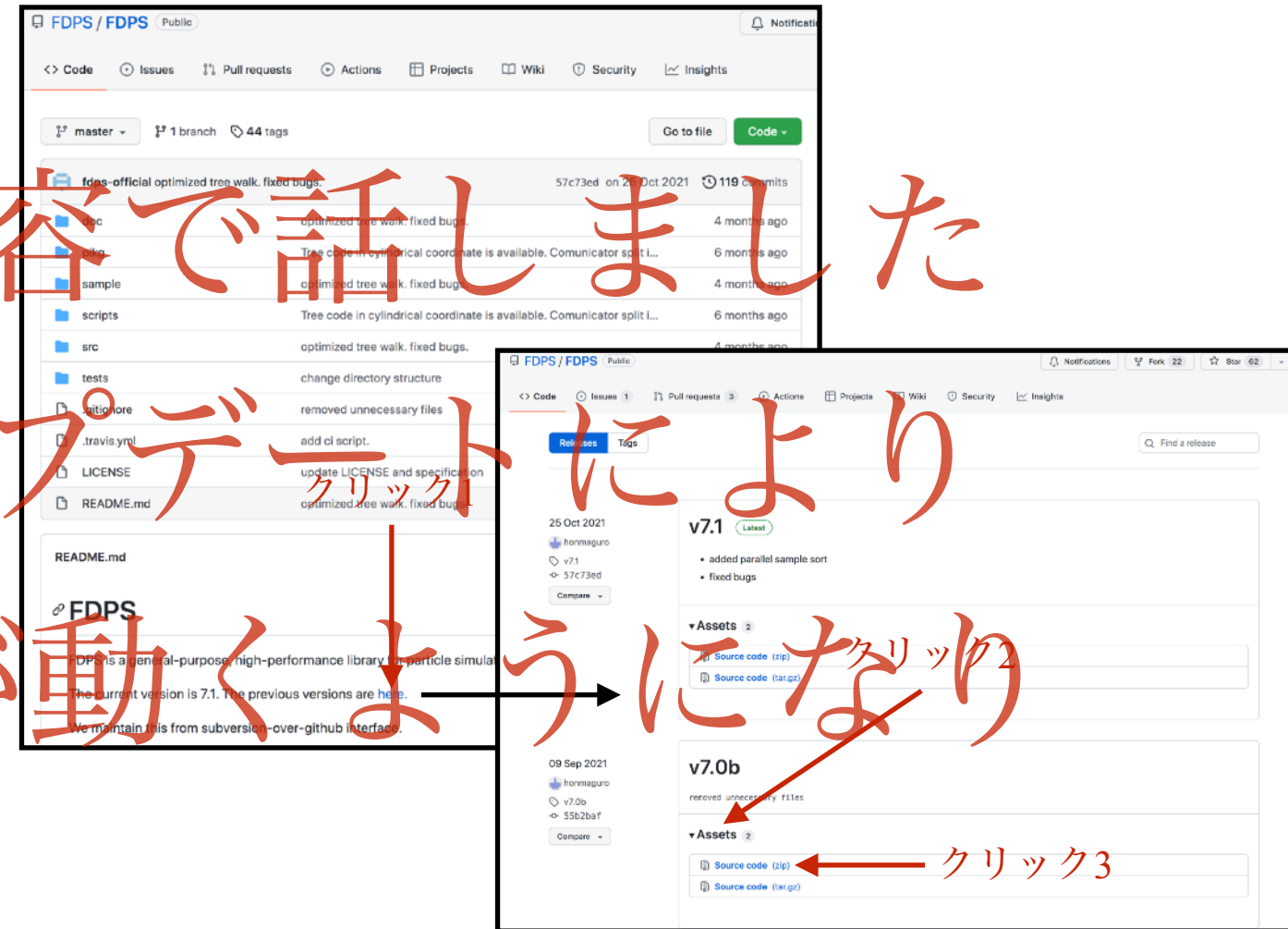
ディレクトリFDPS-7.0b/sample/c++nbodyに移動する。  
Makefileを編集する

#use\_gpu\_cuda = yesの”#”を消す (GPUの使用をオン)  
NVCC = time \$(CUDA\_HOME)/bin/nvcc...の”time”を消す  
(MUVにtimeコマンドがなかったの)。

makeコマンド→”nbody.out”という実行ファイルができる。  
右のようなジョブスクリプト(job.sh)を作り, ”qsub 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

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。

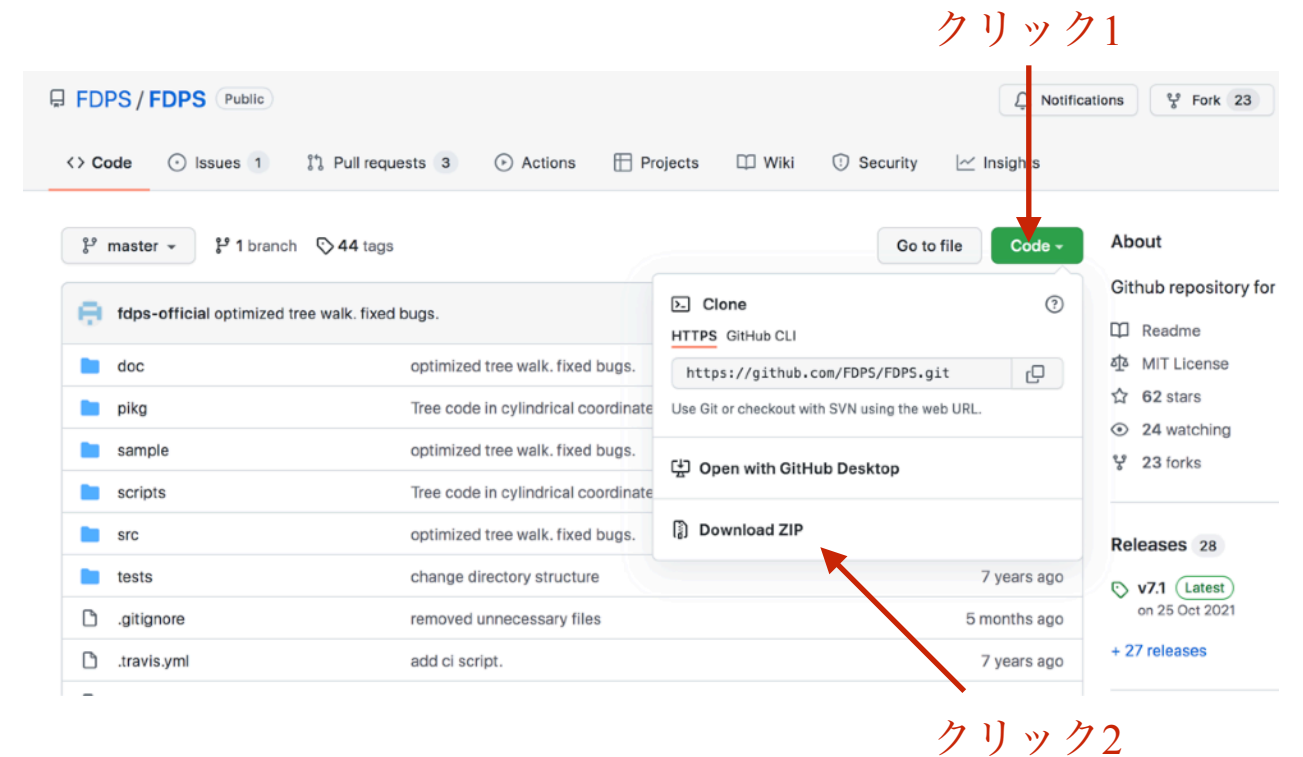


```
#!/bin/bash
#PBS -q gk20-1-debug
#PBS -l walltime=00:05:00
cd $PBS_O_WORKDIR
./nbody.out >& stderr.log
```



# 使い方@MUV

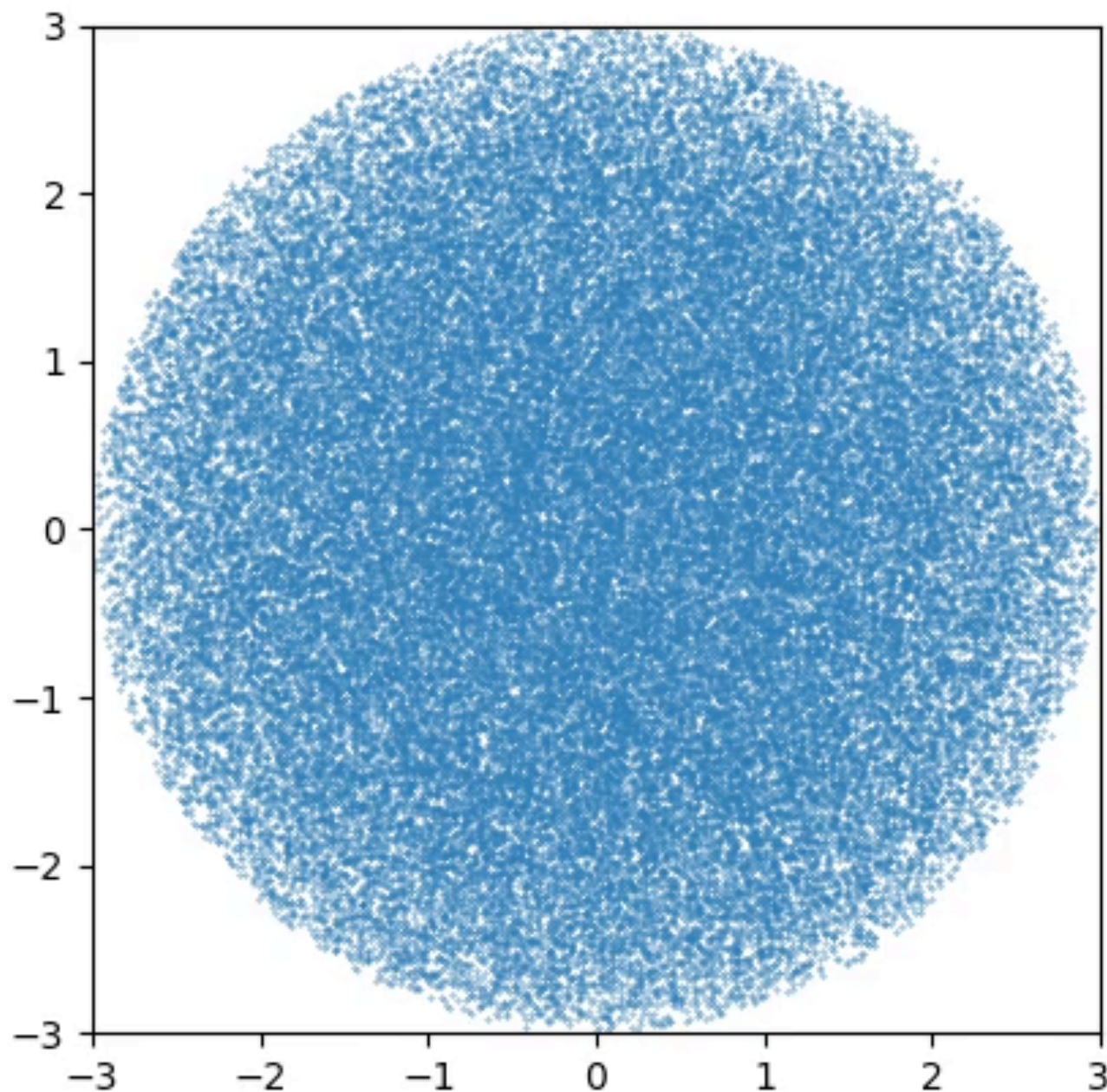
- 置き場: <https://github.com/FDPS/FDPS/>
- muv02でFDPS-master.zipを展開→FDPS-masterというディレクトリができる。
- ディレクトリFDPS-master/sample/c++/nbodyに移動する。
- “module load gcc/8.2.0 cuda-toolkit/11.0.3”とコマンドを実行する
- Makefileを編集する
  - #use\_gpu\_cuda = yesの”#”を消す (GPUの使用をオン)。
  - “CUDA\_HOME=/usr/local/cuda”を”CUDA\_HOME=/opt/muv/cuda-11.0”に書き換える
  - NVCC = time \$(CUDA\_HOME)/bin/nvcc ...の”time”を消す (MUVにtimeコマンドがなかったの)。
- 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, ”qsub 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
#PBS -q gk40-1-middle
#PBS -l walltime=00:05:00
cd $PBS_O_WORKDIR
module load gcc/8.2.0 cuda-toolkit/11.0.3
./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。

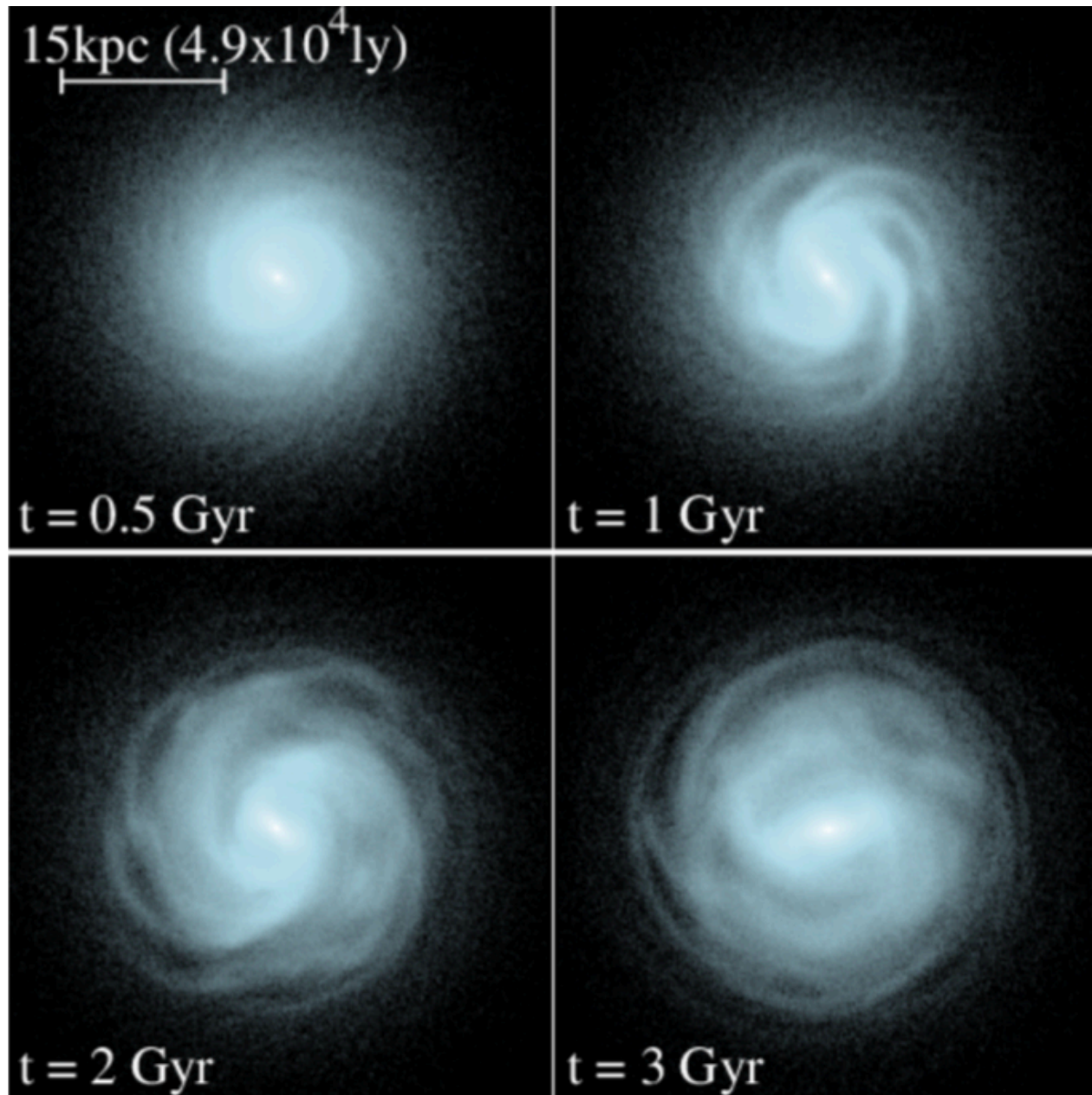
# N=65536 アニメーション



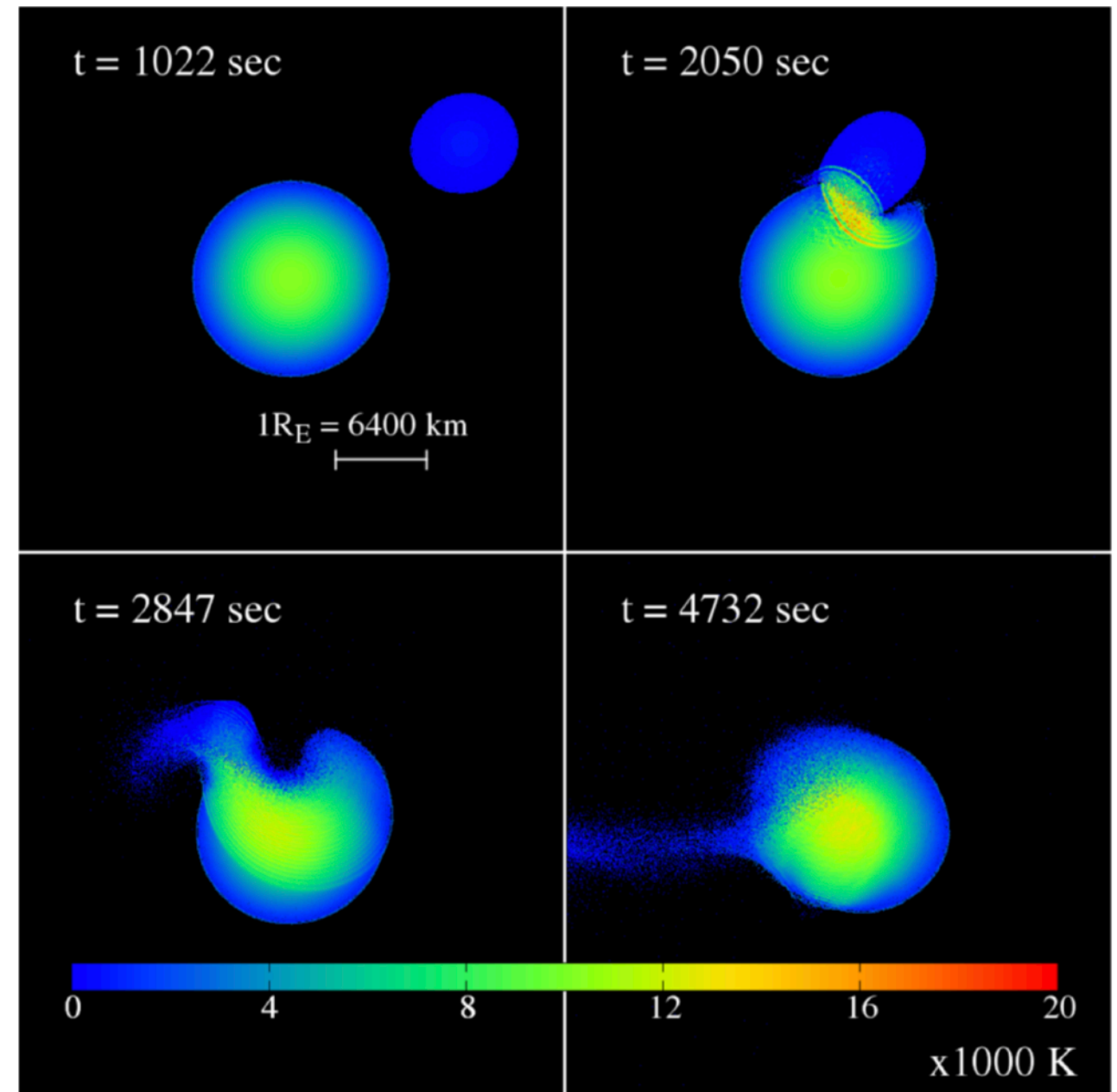


# FDPS使用例

N体 (銀河円盤)

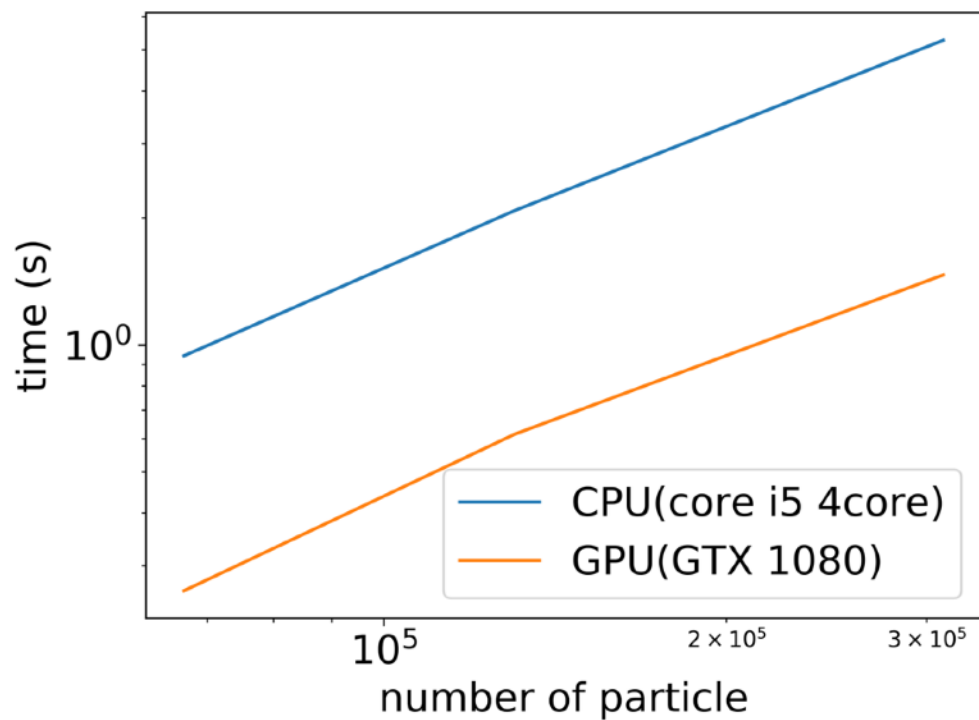
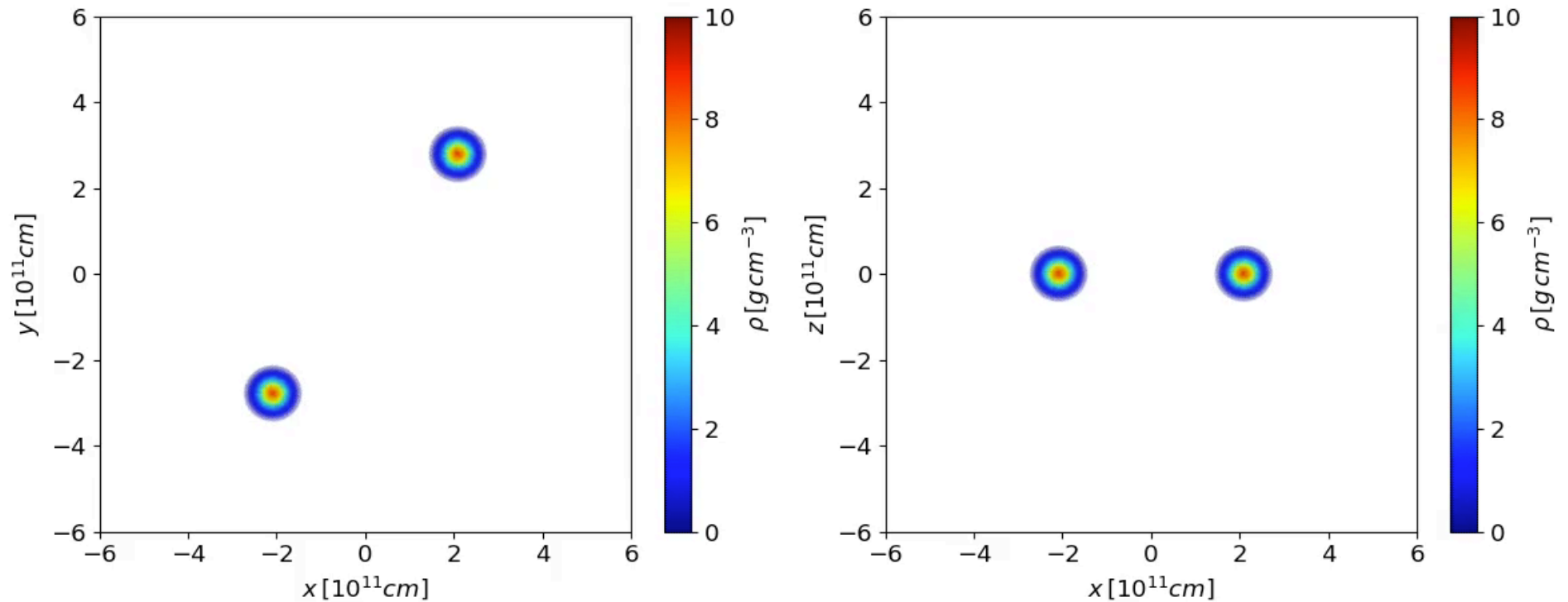


SPH (Giant impact)



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
- MUVにおける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<sup>3</sup>T法: Oshino et al. 2011)
- P<sup>3</sup>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<sup>3</sup>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

- $\sim 10^2 - 10^6$ の恒星が密集した天体
- 衝突系：二体緩和時間 $\ll$ 寿命

→ 高精度計算が必要

散開星団 (プレアデス)



球状星団 (M15)



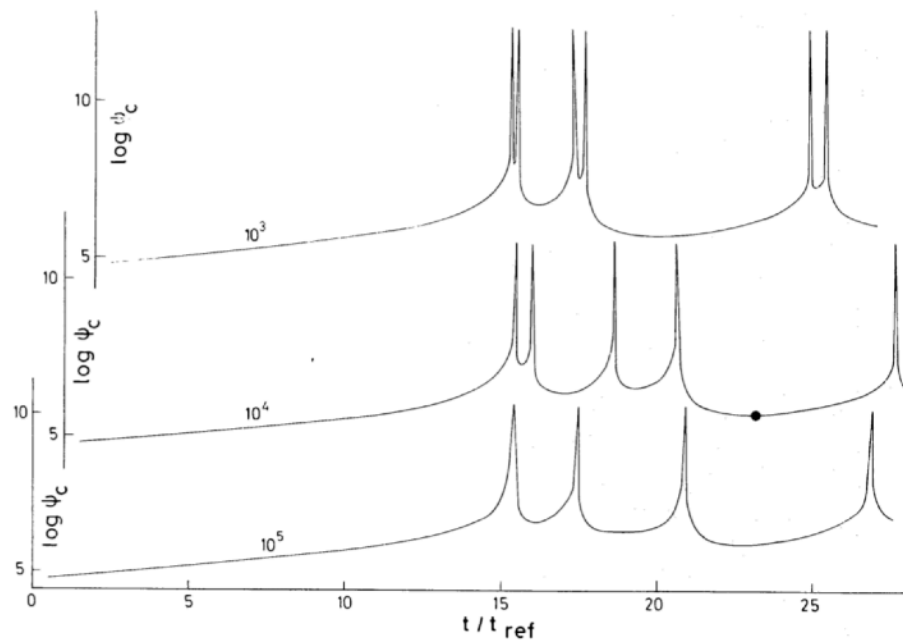


# 連星

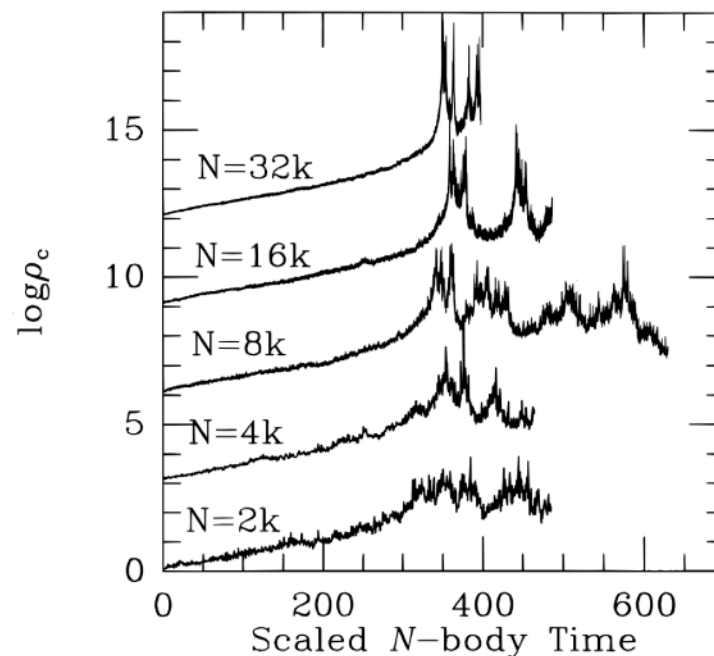
- 連星は高密度星団の構造（特にコア構造）に影響大

- 星団コアのエネルギー： $E_c \sim \frac{GM_c^2}{r_c} \sim 8.6 \times 10^{47} \text{ erg} \left( \frac{M_c}{10^3 M_\odot} \right)^2 \left( \frac{r_c}{0.1 \text{ pc}} \right)^{-1}$

- 連星のエネルギー： $E_b \sim \frac{GM_b^2}{r_b} \sim 3.8 \times 10^{47} \text{ erg} \left( \frac{M_b}{1 M_\odot} \right)^2 \left( \frac{r_b}{10 R_\odot} \right)^{-1}$



Bettwieser, Sugimoto (1984)

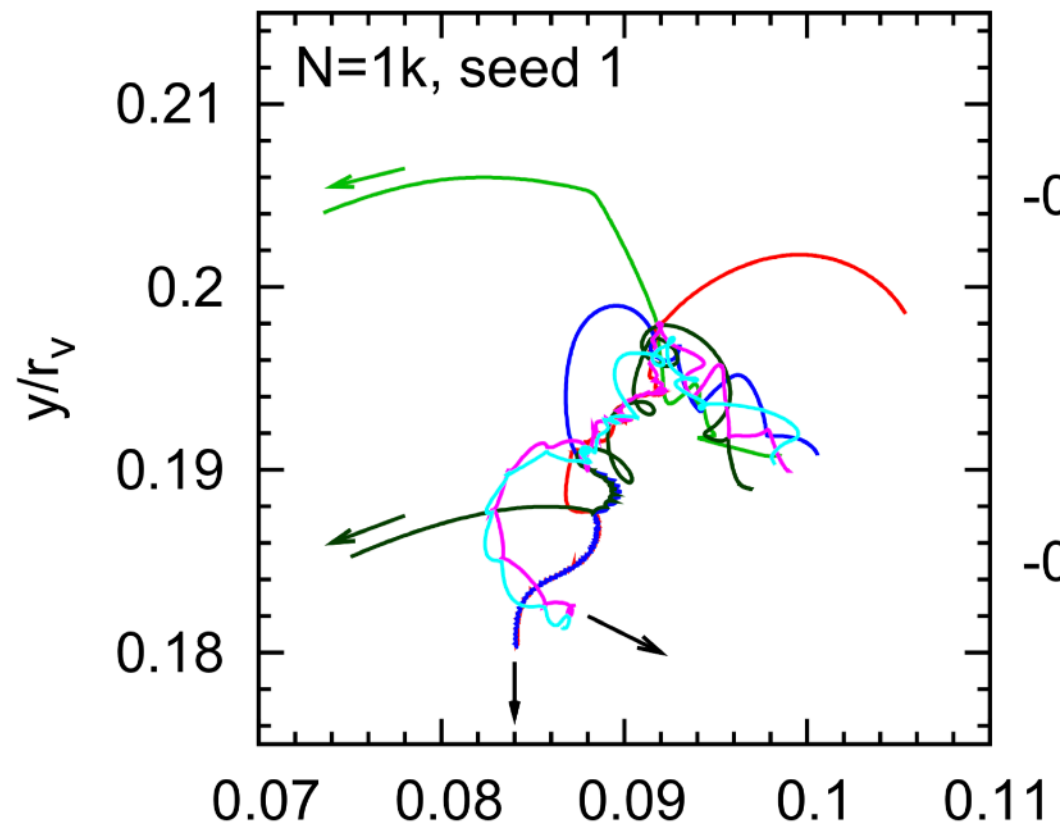


Makino (1996)

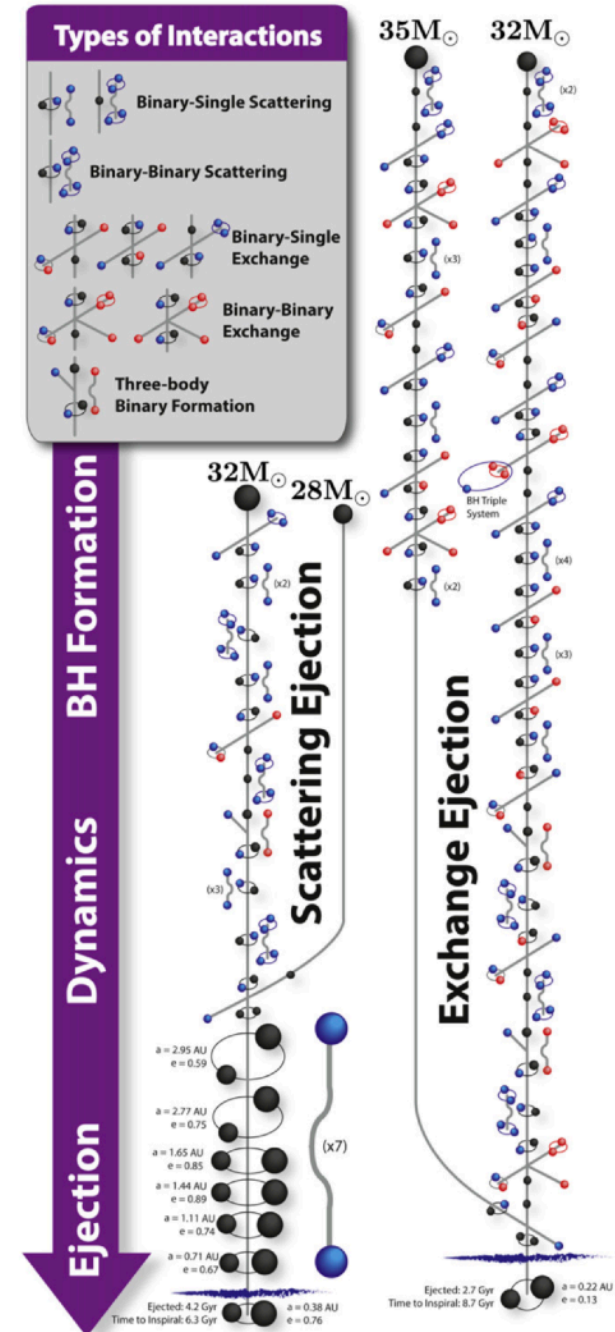
Gravothermal oscillation  
(重力熱力学的コア振動)

# 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：

$$P = 2\pi \left( \frac{r_b^3}{GM_b} \right)^{1/2} \sim 3.7 \text{ day} \left( \frac{r_b}{10R_\odot} \right)^{3/2} \left( \frac{M_b}{1M_\odot} \right)^{-1/2}$$

- 星団の横断時間：

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

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

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

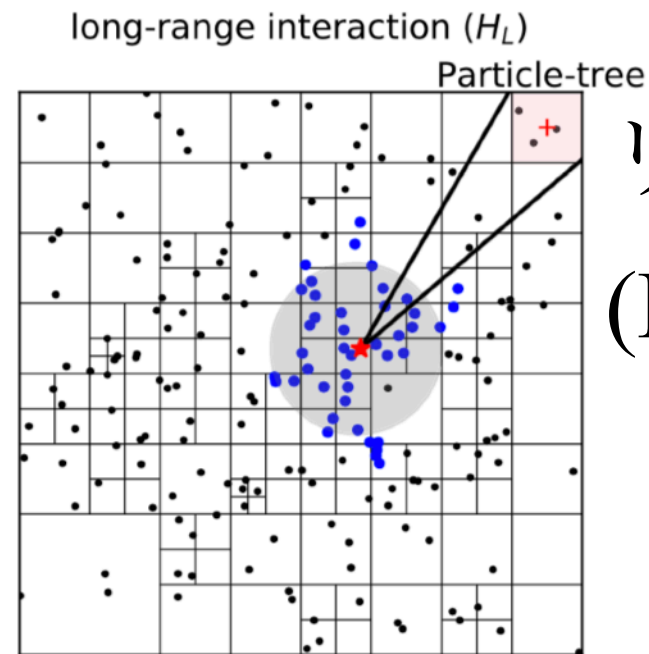
# 連星軌道の正則化

- 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)

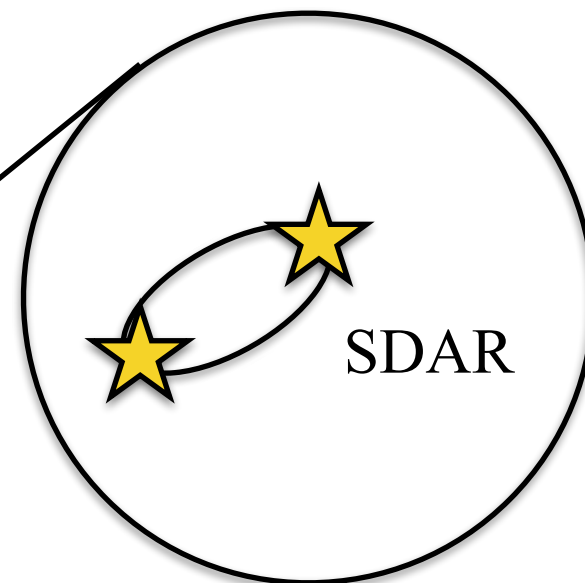
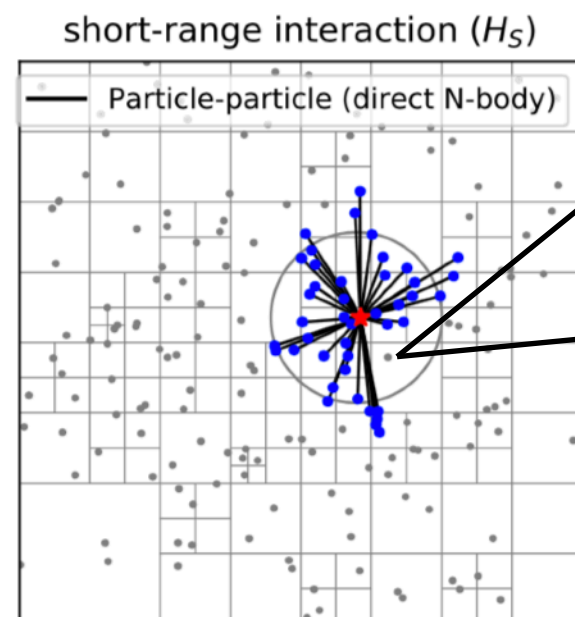
# SDAR

- 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 $\Delta t$ は可変で，仮想的タイムステップ $\Delta s$ 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



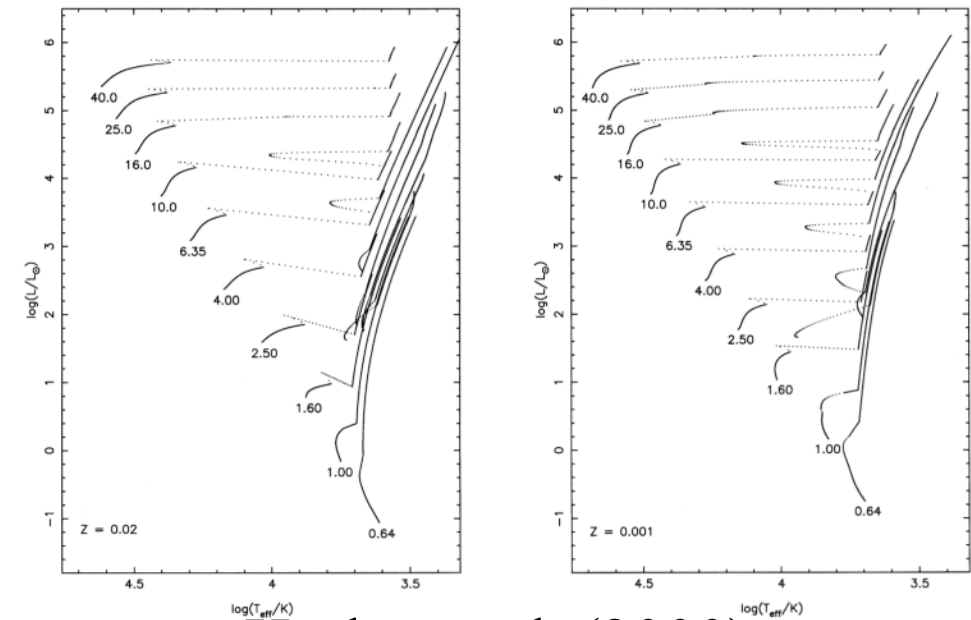
リープフロッグ法 + 重力のツリー計算  
(FDPSを利用)



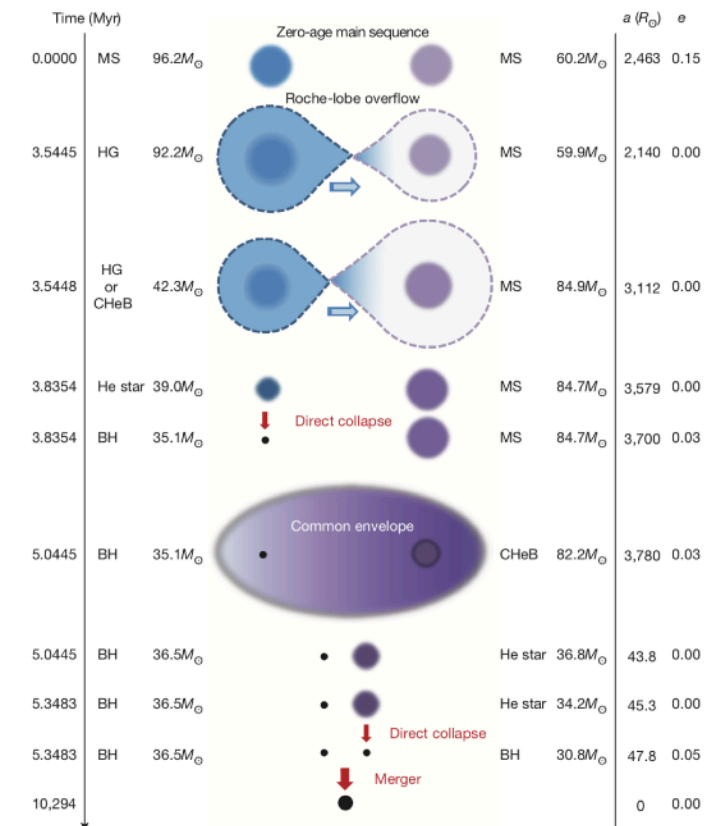
4次エルミート法 + 重力の直接計算

# 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@MUV (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@MUV (2)

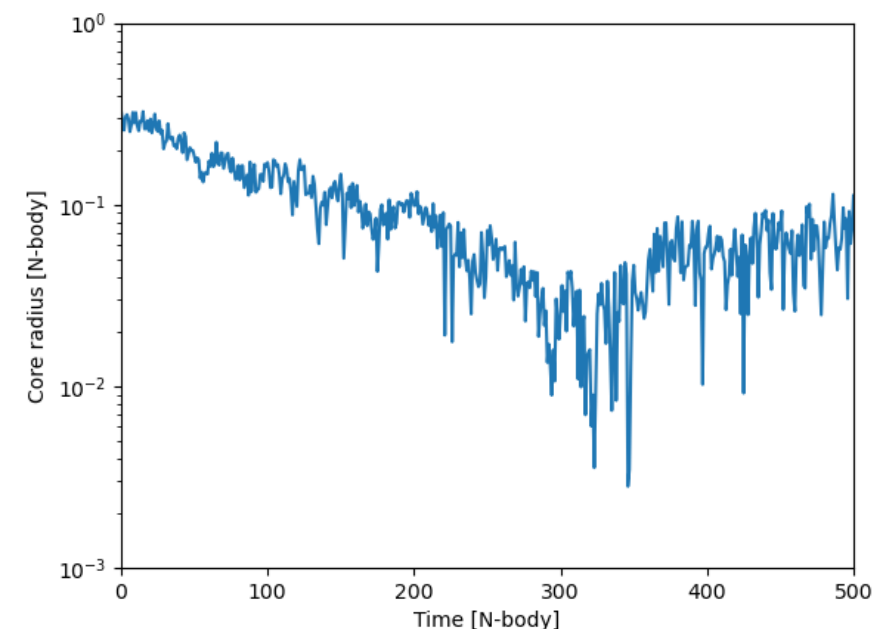
- パスの設定
  - PATHにINSTALLDIR/bin追加
  - PYTHONPATHにINSTALLDIR/include追加
    - ただしnumpyが必要
- “qsub job.sh”を実行する.
- data.0, data.1, ..., data.9などのスナップショットが出力される.
- “qsub analysis.sh”を実行すると様々なファイルができる.
- data.coreにはコア半径などの情報があり, 1列目をx軸, 8列目をy軸としてプロットするとコア半径の時間進化を描ける.
  - “job.sh”はT=10までしか計算しないようになっているが, 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
#PBS -q gk20-1-middle
#PBS -l walltime=02:00:00
cd $PBS_O_WORKDIR
petar.omp.avx2.gpu -n 1024 __Plummer >& stderr.log
```

analysis.sh

```
#!/bin/bash
#PBS -q gk20-1-middle
#PBS -l walltime=02:00:00
cd $PBS_O_WORKDIR
ls | egrep '^data.[0-9]+$' | 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 @MUV
- 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