重力多体シミュレーションコードのチューニング

石山 智明

筑波大学計算科学研究センター神戸分室

前書き

- 私は宇宙磁気流体・プラズマシミュレーションを
  したことはありません

- 普段やっていること: 大規模重力多体シミュレーション

- 頂いたお題
  - 「京」向けのチューニング技法

- シミュレーション・アルゴリズムが異なるので、チューニングの
  ポイントもかなり異なってくると思いますが、一つの事例として
  - 共通項はあるはず
  - 話半分で

outline

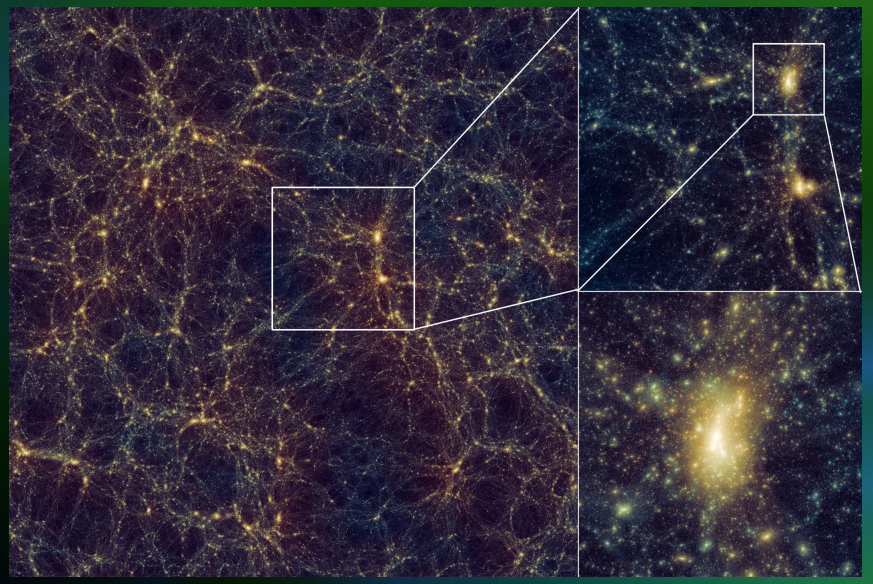
- 京でチューニングしたコードについて
- 京について
- 並列チューニング
  - トーラスマッピング
  - MPI_COMM_WORLD の分割
- 単体チューニング
  - SIMD、ソフトウェアパイプライン
  - メモリアクセス最適化
- その他
  - 大規模データの取扱い
  - 京を使っていて今苦戦しているところ

京でチューニング
したコードについて

重力多体 (N体) シミュレーション

- 重力多体系をN個の質点で表現。粒子間の相互作用重力を計算し、運動方程式を時間積分、系の時間発展を追う

$$\frac{d^2 r_i}{dt^2} = \sum_{j \neq i}^{N} G m_j \frac{r_j - r_i}{|r_j - r_i|^3}$$



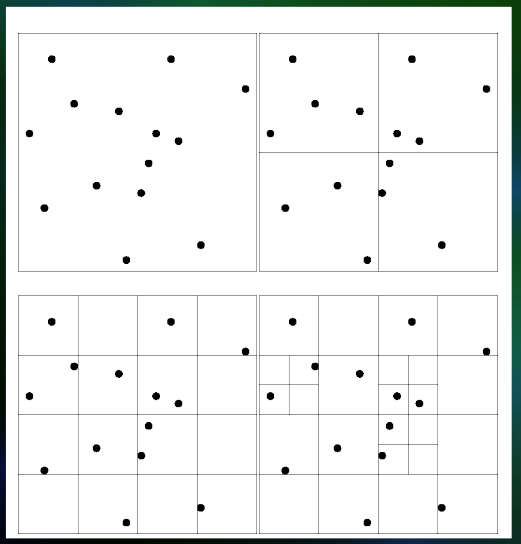
- 対象の例
  - **ダークマター構造形成**
  - 銀河・銀河団
  - 星団
  - 惑星系
  - ブラックホールの運動

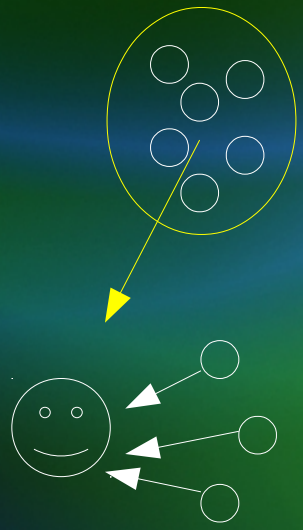
重力計算アルゴリズム

- **直接計算**：全粒子対全粒子の力を計算する
  - **O(N²)**

$$\frac{d^2 r_i}{dt^2} = \sum_{j \neq i}^{N} G m_j \frac{r_j - r_i}{|r_j - r_i|^3}$$

- **ツリー法**：近傍の粒子との相互作用は直接、
  遠方の粒子群との相互作用はまとめて
  多重極展開で計算する
  - **O(N²) から O(*N* log*N*)へ**
  - **Modified algorithm**:
    相互作用リストを粒子グループで共有する



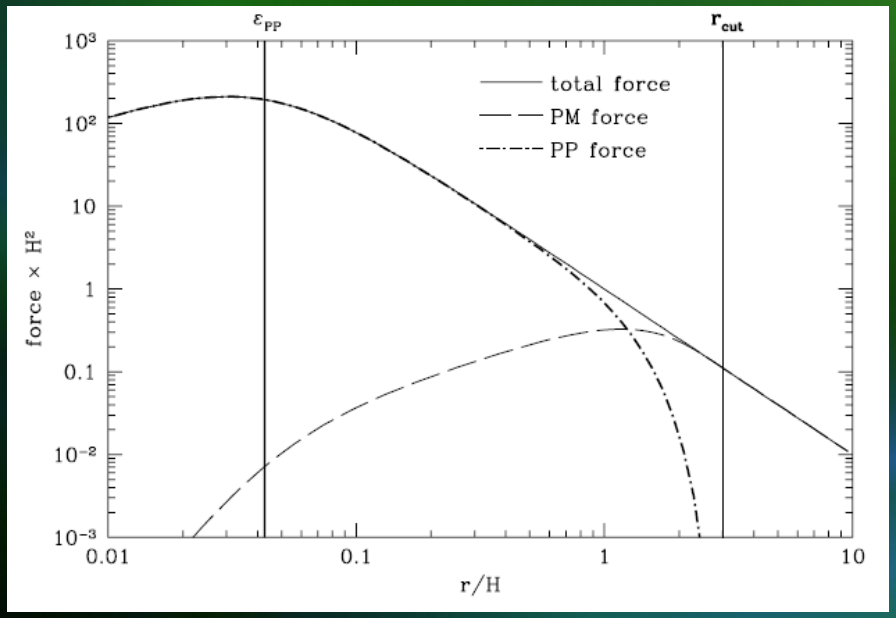


- **PM法**：一様格子上の密度場を計算し、FFTを用いてポアソン方程式を解く
  - 周期境界条件を自然に解ける

宇宙論N体での主流

TreePM法

- 近距離力(カットオフつき)をTree法、遠距離力をPM法で解く
  - $O(N^2) \rightarrow O(N \log N)$
  - 周期境界条件を実現
- 比較的新しいアルゴリズム
  - GOTPM (Dubinski+ 2004)
  - GADGET-2 (Springel 2005)
  - GreeM (Ishiyama+ 2009)
  - HACC (Habib+ 2012)

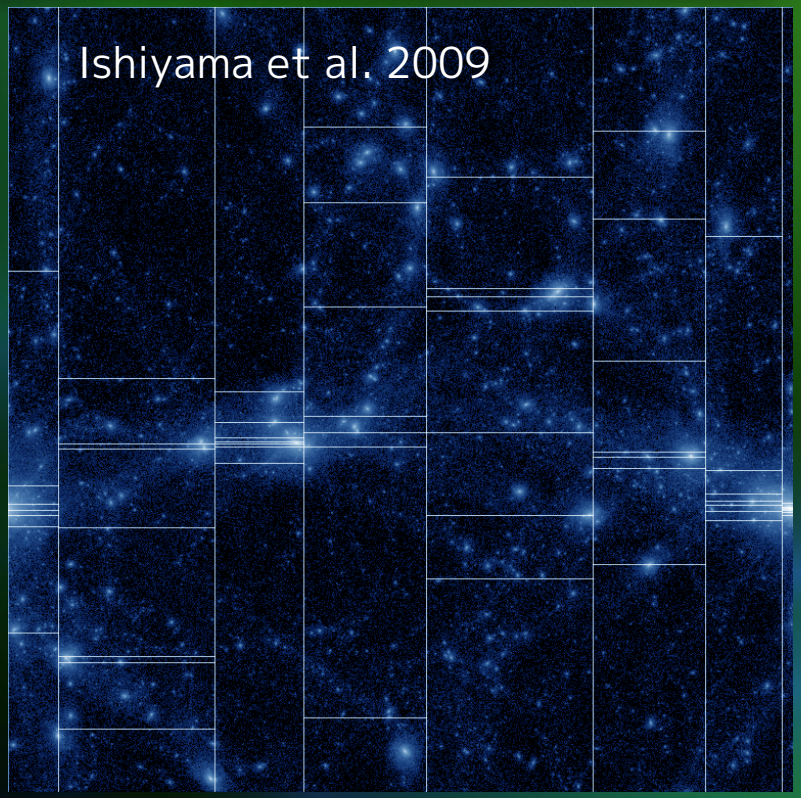


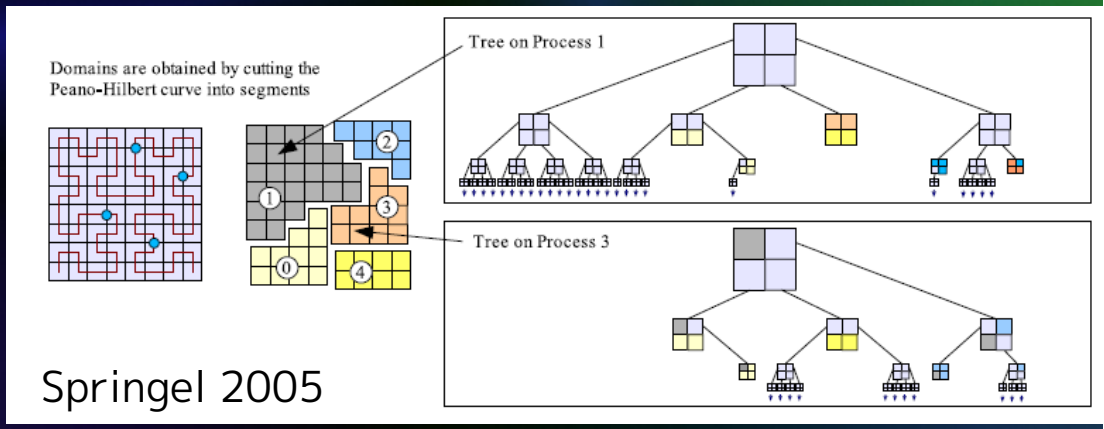
$$\boldsymbol{a}_i = \sum_{i \neq i} \frac{m_j(\boldsymbol{r}_j - \boldsymbol{r}_i)}{|\boldsymbol{r}_j - \boldsymbol{r}_i|^3} g_{\mathrm{P3M}}(|\boldsymbol{r}_j - \boldsymbol{r}_i|)/\eta),$$

$$g_{\mathrm{P3M}}(R) = \begin{cases} 1 - \frac{1}{140}\left(224R^3 - 224R^5 + 70R^6 + 48R^7 - 21R^8\right) & (0 \leq R \leq 1) \\ 1 - \frac{1}{140}\left(12 - 224R^2 + 869R^3 - 840R^4 + 224R^5 + 70R^6 - 48R^7 + 7R^8\right) & (1 \leq R \leq 2) \\ 0 & (2 \leq R) \end{cases},$$

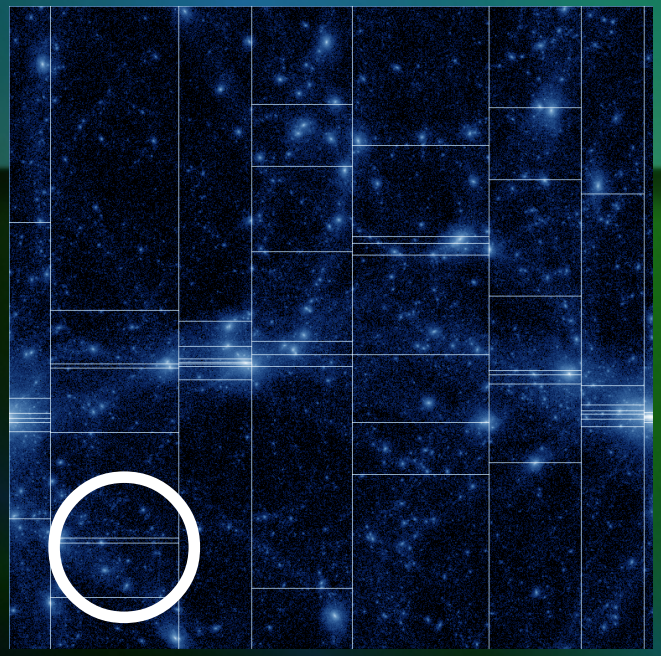
並列化

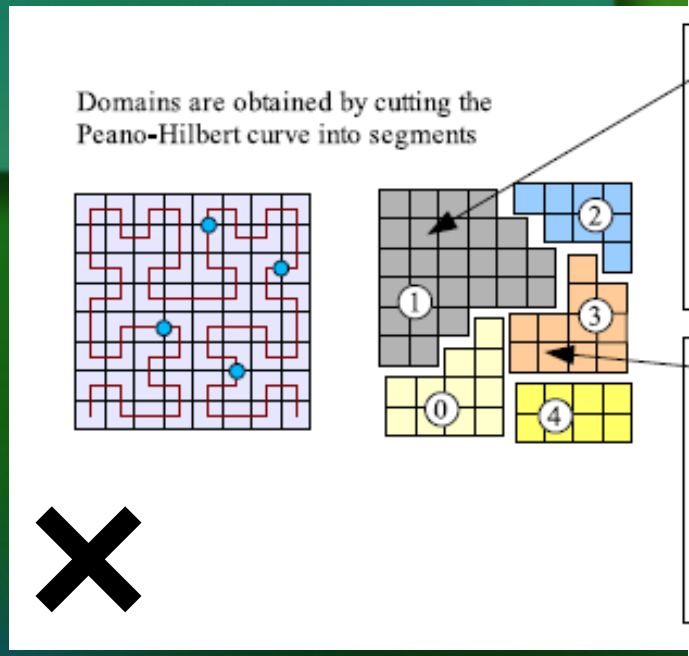
1. 全空間を分割し計算ノードに割り当て、
   粒子を再配分する
   - サンプリング粒子の集約 (全対通信)
   - 一部粒子情報を通信 (隣接通信)
1. 長距離重力の計算 (PM 法)
   - 全メッシュ情報を通信
     (全対通信、粒子よりは通信量小)
2. 短・中距離重力の計算 (Tree法)
   - 一部ツリー情報を通信 (隣接通信)
3. 粒子の時間積分
4. 1に戻る





動的領域分割

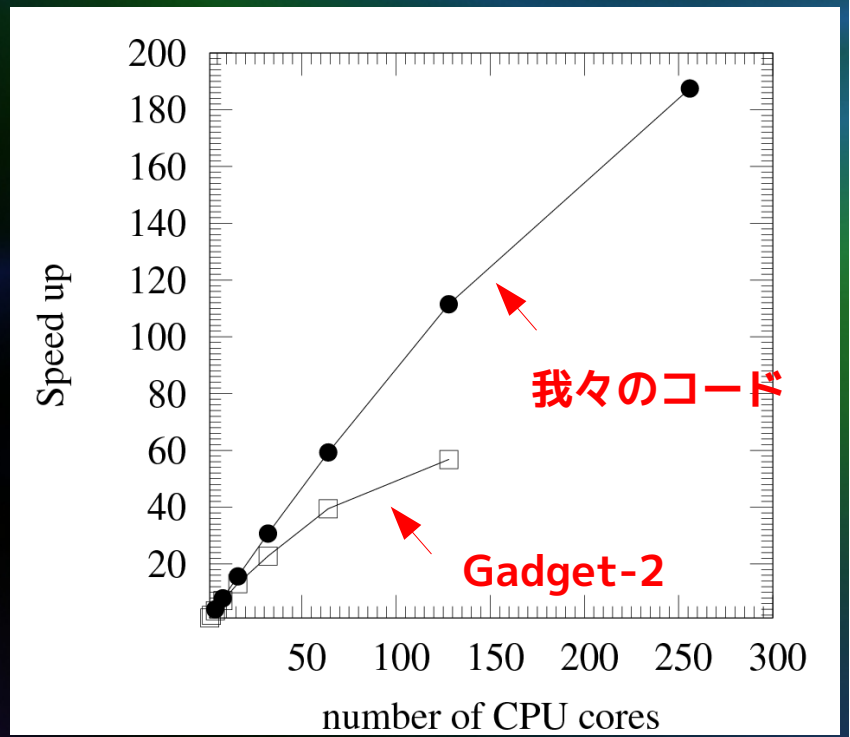




- 領域分割の仕方が重要

- × 粒子数均等
- ○ 相互作用数均等
- ◎ 相互作用均等 + 補正
  - 計算時間が一定になるようにする

- △ 空間充填曲線
- ○ 再帰的多段分割
  - 隣接ノードが自明



Load balancer

- Sampling method to determine the geometry
  - Each node samples particles and shares with all nodes
  - Sample frequency depends on the local calculation cost -> realizes near-ideal load balance

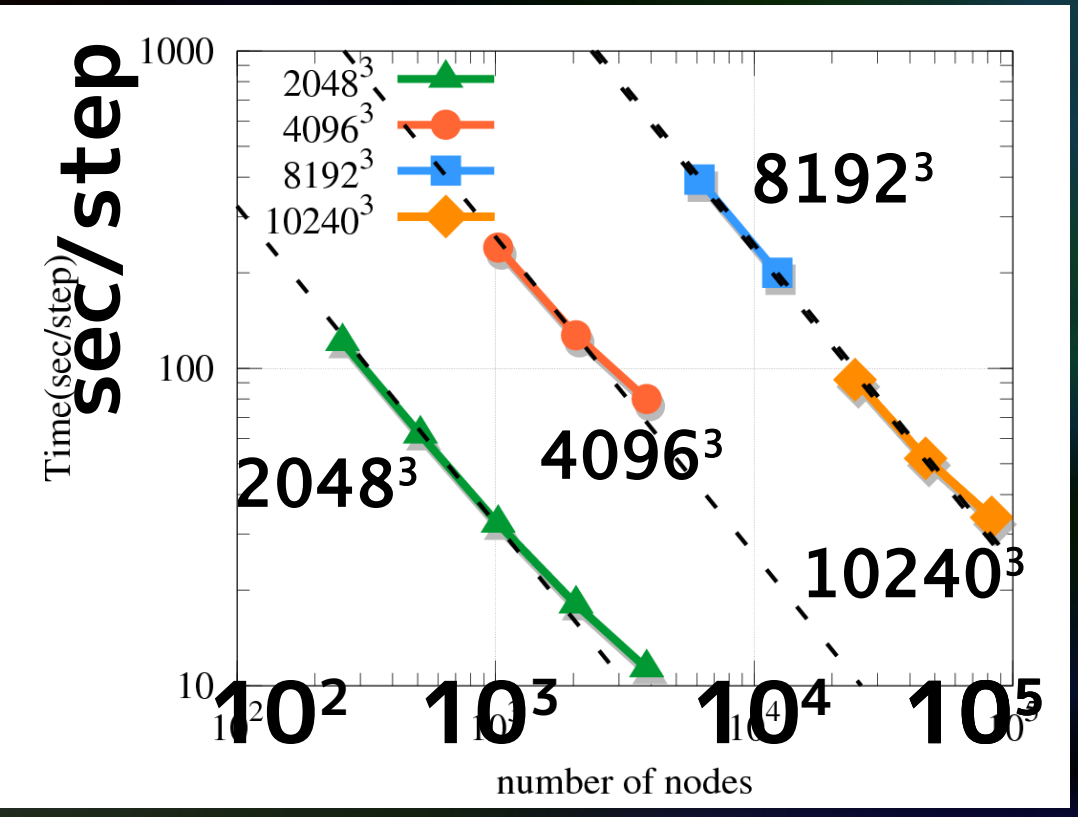
$$n_{\mathrm{samp},i} = N R_{\mathrm{samp}} f_{\mathrm{samp},i}$$

R~10^{-3} ~ 10^{-5}

$$f_{\mathrm{samp},i} = \frac{t_{\mathrm{PP},i} + t_{\mathrm{PM},i}}{\sum_j \left(t_{\mathrm{PP},j} + t_{\mathrm{PM},j} \right)}$$

  - New geometry is adjusted so that all domains have the same number of sampled particles
  - Linear weighted moving average for last 5 steps

Performance results on K computer



Ishiyama, Nitadori, and Makino, 2012 (arXiv: 1211:4406), SC12 Gordon Bell Prize Winner

- Scalability (2048^3 – 10240^3)
  - Excellent strong scaling
  - 10240^3 simulation is well scaled from 24576 to 82944 (full) nodes of K computer
- Performance (12600^3)
  - The average performance on full system (82944=48x54x32) is ~**5.8Pflops**,

~**55**% of the peak speed

京について

京コンピュータ

- SPARC64™ VIIIfx 2.0GHz octcore (128Gflops / CPU)
- 16 GB memory / CPU
- 6D torus network

- Total 82944 nodes (663552 CPU core)
- 1.3PB memory
- 10.6 Pflops peak speed

1	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560640	17590	27113	8209
2	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1572864	16325	20133	7890
3	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705024	10510	11280	12660
4	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom	786432	8162	10066	3945

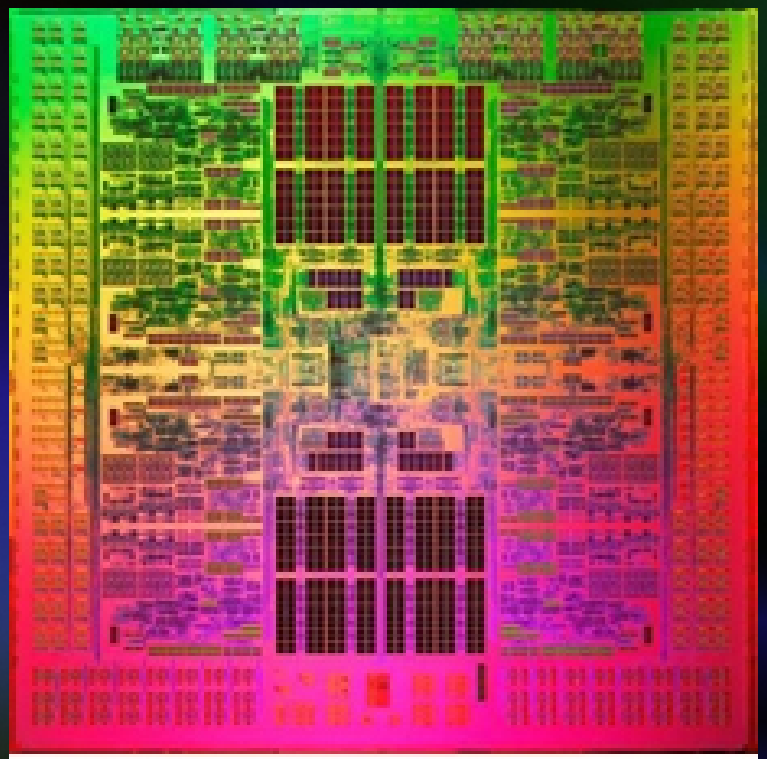
2011年5、11月のTop 500 で世界 **No.1**

2012年5月に No.2 に転落

2012年11月に No.3 に転落

だが消費電力 No.1 は継続中

SPARC64™ VIIIfx


SPARC64™ VIIIfx

(c) Fujitsu Co.

- 8コア、6MB 共有L2キャッシュ
- クロック 2GHz

- 2pipe x 2SIMD x 2FMA = 8演算 / 1 clock
  - 倍精度。単精度でも使えるが8演算なのは一緒
- コアあたりピーク 16GFlops
- CPUあたり 128GFlops

- メモリスループット　64GB/s

  → B/F = 0.5

FMA(積和演算) : d = a*b + c のような演算を1クロックで処理可能

Tofu network

- 1CPU (8コア) / ノード
- 16GB memory / ノード

- Tofu (Torus Fusion) インターコネクト
  - 3次元の仮想トーラスとして利用可
  - 各軸方向に 5GB/s peak
  - 同時通信数 4

チューニングのポイント

- トーラスネットワークなので、近接通信が速い
  - 長距離通信は不利、必要な場合はアルゴリズムを工夫する
- 同時通信数 4 > コア数 8
  - Flat MPI ではなくハイブリッド並列にする

    (京では通信バッファ用に確保されるメモリが大きく、8 プロセス / ノードはメモリ不足で実行できない)

- 2pipe x 2SIMD x 2FMA
  - SIMD化 できないと最大 1/8 の性能しか出ない
  - コンパイラがSIMD化してくれるように、コードを書く

- B/F = 0.5
  - 演算密度が小さいとどうしようもない?
  - キャッシュ再利用性を高める

例 (**要求 B/F 24**)

```
for(int i=0; i<n; i++){
  c[i] = a[i] + b[i]
}
```

並列チューニング

ネットワーク性能について使ってみた印象

- (コードによってはそんなことはないだろうが) 数千ノードまでは、
  他のスパコンと大差はない印象

- 数千ノード以上

  長距離通信の性能低下が見え始めてくる

- 数万ノード以上

  ボトルネックに

トーラスマッピングと、
MPI_COMM_WORLD の分割で改善

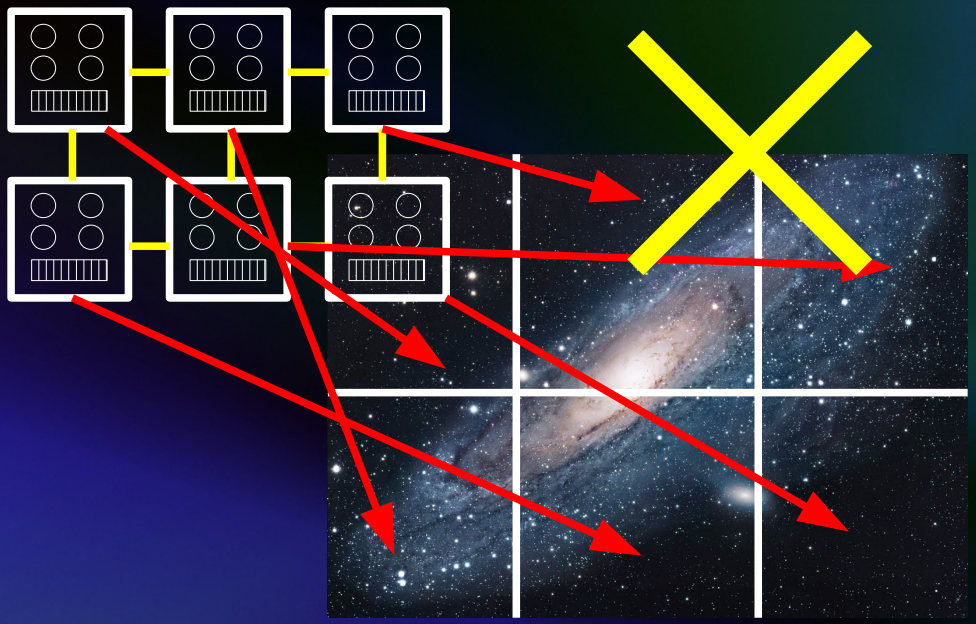
トーラスマッピングによる通信最適化

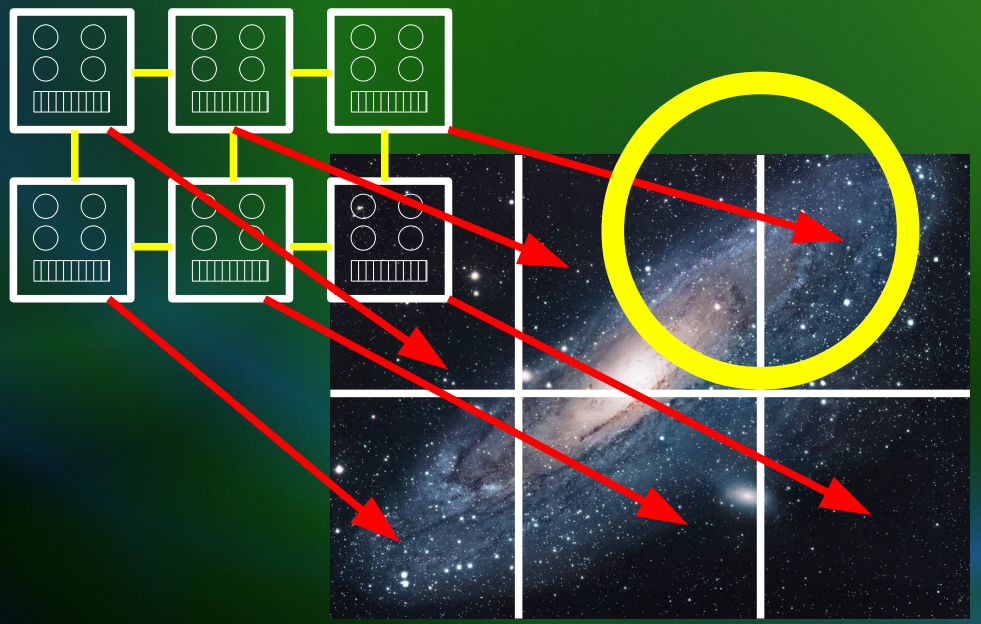
- 特に何も指定しないと、どういった形状でノードが確保されるかわからない

- 32ノードジョブで 32x1x1 のような形状になる可能性もある
  - 長距離通信のホップ数が大きい
  - シミュレーション空間を 4x4x2 で切ってしまうと、シミュレーション空間上の隣接通信が、ノード上は隣接していない通信になることも

- ノード形状を指定
  - ただしジョブは流れにくくなる
- 各領域を担当するノードの物理配置を、実際の領域分割にあわせる
  - 仮想3次元トーラス上の座標を取得するAPIが存在する

トーラスマッピングによる通信最適化





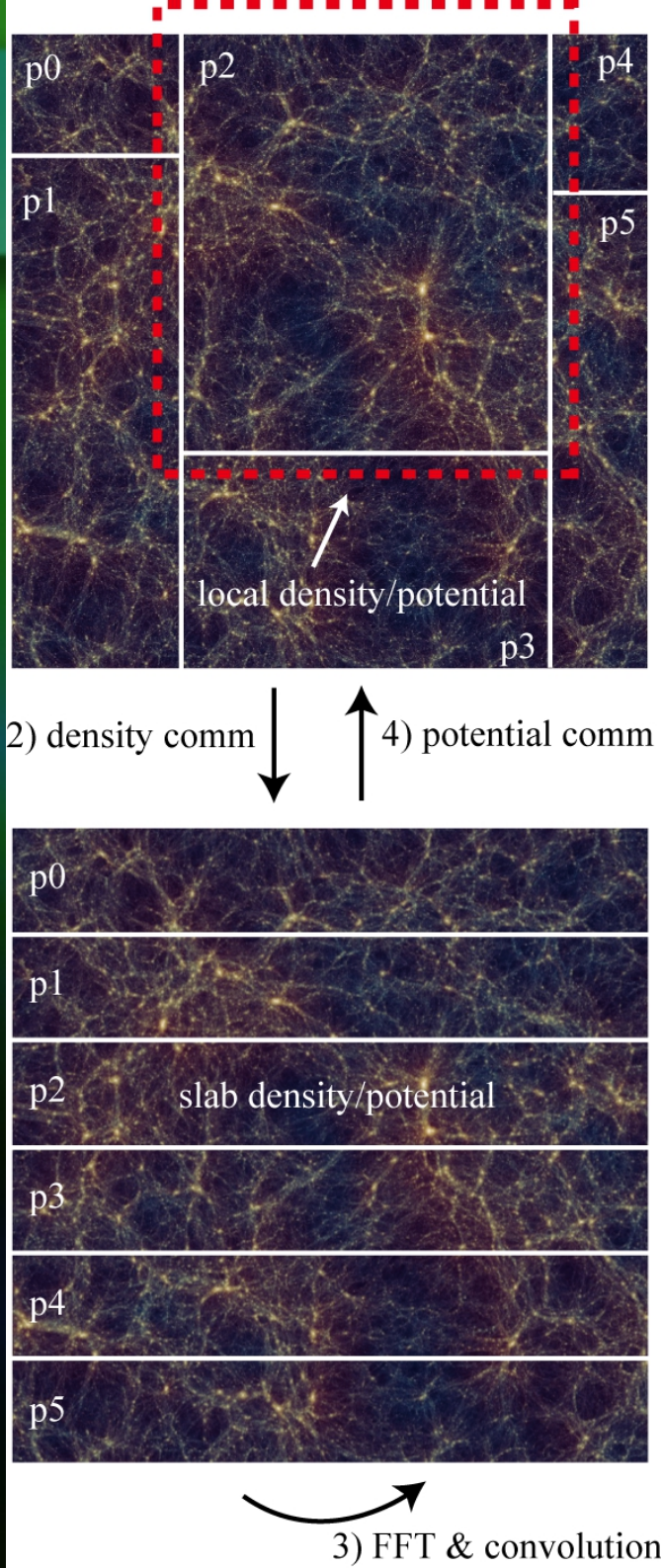
- 4096 ノード、16x16x16の空間分割で合計20MBの Allgather をした場合
  - ノード形状 10x15x28 0.06 sec
  - 16x16x16 0.04 sec

MPI_COMM_WORLDの分割による改善例1

並列PM

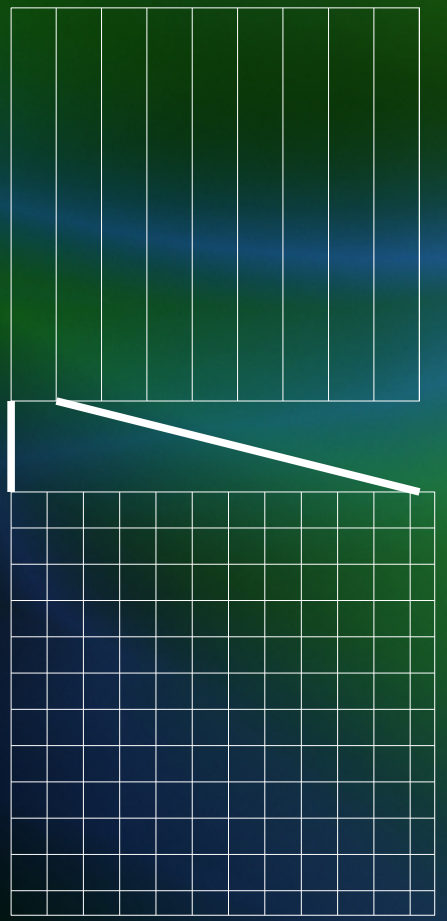
1. density assignment
2. local density → slab density
   - **長距離通信**
3. FFT (density → potential)
   - 長距離通信 (library 任せ)
4. slab potential → local potential
   - **長距離通信**
5. force interpolation

空間分割は不規則三次元分割だが、FFTは規則的な 1or2or3 次元分割のデータ構造になっている必要があるため



もう少し具体的に

- ~10000^3粒子、4096^3 PMメッシュのシミュレーションを京のフルシステム(48x54x32=82944 ノード)で行う場合
  - 最適な領域分割は 48x54x32 (network topology 依存)
  - 1D並列FFTW を使うと FFT は 4096 並列で可能



1FFTプロセスは **数千ノード** から
メッシュデータを受け取る必要有!!!

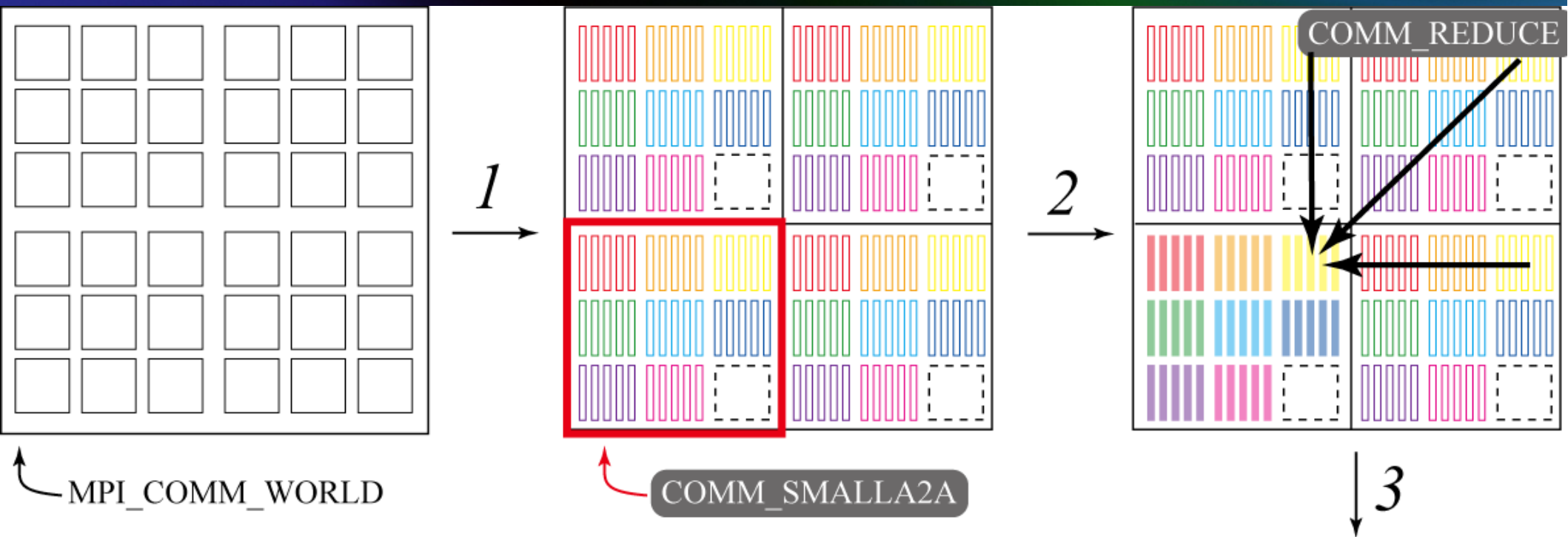
- MPI_Isend、MPI_Irecv は無理
- MPI_Alltoallv を使うと簡単に実装できるが、、、
- 2D、3D FFT なら幾分ましだが、通信競合は発生する

MPI_COMM_WORLDの分割による改善例1：並列PM

通信の階層化 (MPI_Comm_split を使う)

- MPI_Alltoallv (..., MPI_COMM_WORLD)
  1. MPI_Alltoallv (..., COMM_SMALLA2A)
  2. MPI_Reduce(..., COMM_REDUCE)

3~4 倍の通信高速化に成功!

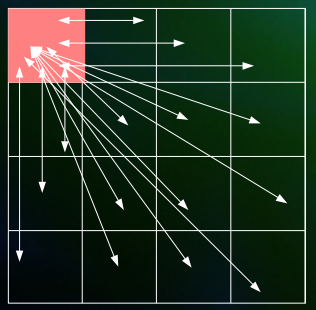


MPI_COMM_WORLDの分割による改善例2

全データのシャッフル

- 全ノードのデータをシャッフルしたくなることもあるかもしれない
  - Restart 時にノード数が変わる時など

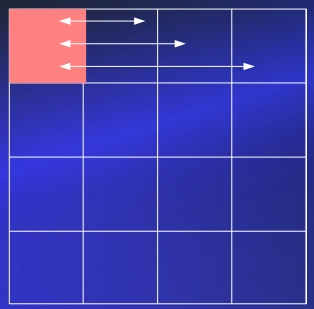
- 実装は Alltoallv が楽。だが数千〜数万並列になると色々問題が……
  - 通信回数は O(p)、p は並列数

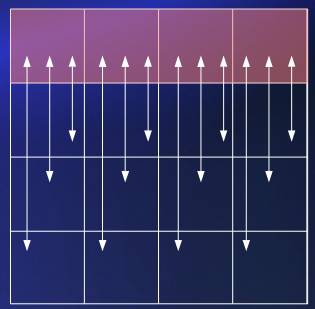


MPI_Alltoallv
(…, MPI_COMM_WORLD)

O (p)

- 通信を階層的にすることで解決
  - 斜め通信を避け、通信競合を防ぐ
  - 近接通信に使えることも





MPI_Alltoallv (…, COMM_X) + MPI_Alltoallv (…, COMM_Y)
+ MPI_Alltoallv (…, COMM_Z)

O ($3p^{1/3}$)

MPI_COMM_WORLDの分割による改善例2

全データのシャッフル

- 粒子数2048^3、1024分割、合計384GByte
  - 4096ノード MPI_COMM_WORLD → **17.7 sec**
  - 4096ノード 階層的通信 → **4.2 sec**

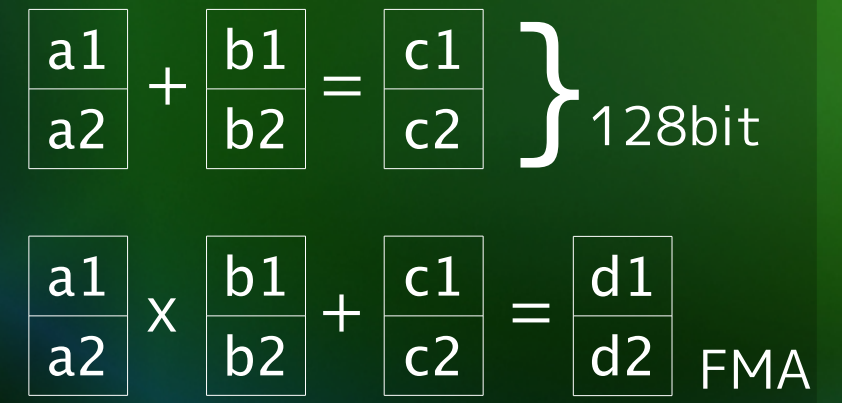
- たいした違いはないと思われるかもしれないが、全ノード (82944) では、前者は1時間以上かかりそうな勢いだった記憶が
  (データがなくてすいません)
- 後者はそのために急遽実装するはめになった

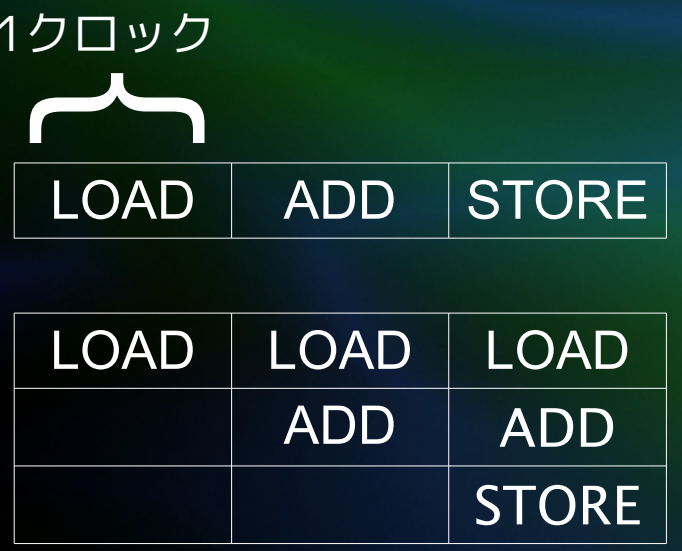
単体チューニング

SIMD と ソフトウェアパイプライニング

- **SIMD**
  1クロックで複数の演算を行う
  - 京は 128bit (FMA) x 2pipe、8演算
    倍精度のみ対応
  - SSEは128bit、単精度4、倍精度2演算
  - AVXは256bit、単精度8、倍精度4演算

- **ソフトウェアパイプライニング**
  一連の処理を分割して、並列に行う
  CPUは通常いくつかの命令を
  同時に処理できる

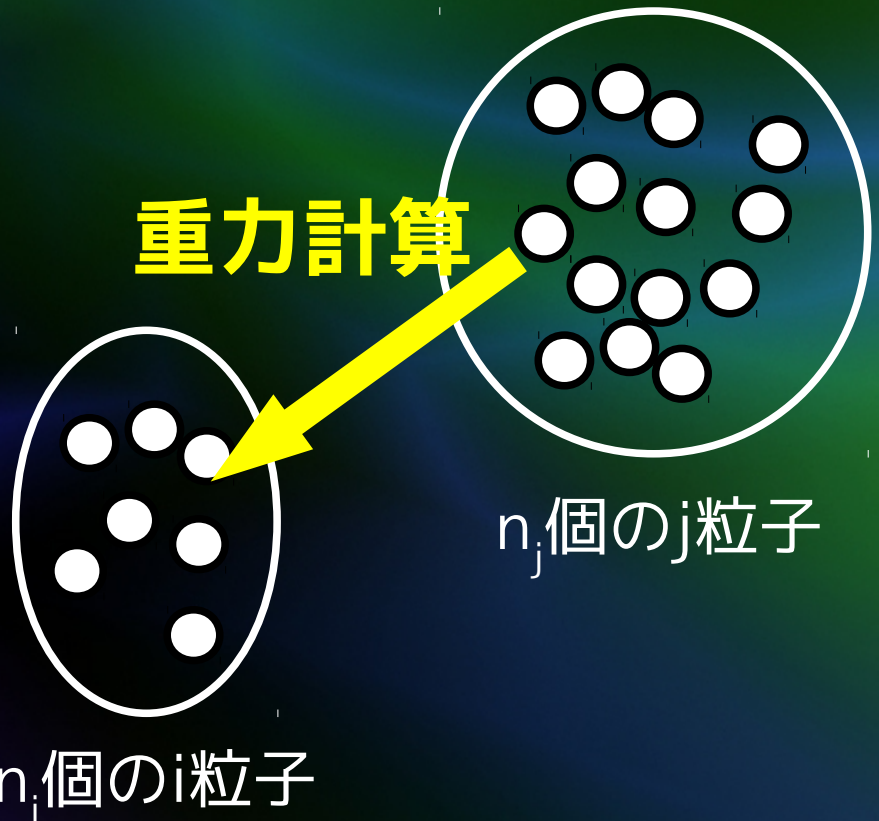




重力多体で一番最適化したいところ

- 全計算時間の5~7割を占める相互作用計算部
  → j粒子グループからi粒子グループへの重力計算
- 両グループの構成粒子は多少重複があるが、ツリーでは基本的に別
  → 作用反作用は利用しない

- 粒子毎に重いツリーウォークをしない
  (modified algorithm)
- 相互作用数は増える
- キャッシュ利用効率は向上
  - だいたい要求 B/F が $1/\langle n_i \rangle$
- i方向に並列性が加わる



京で有用かもしれないオプション・ツール

- コンパイルオプション *-Nsrc*

  コンパイル時にコードとともに
  最適化情報を出力

```
      for( int j=0; j<nj; j++){
2v      double dx = xi[i][0] – xj[j][0];
2v      double dy = xi[i][1] – xj[j][1];
2v      double dz = xi[i][2] – xj[j][2];
                    (省略)
```

アンロール展開数

SIMD化されているかどうか

- 基本プロファイラ *fipp*

  サンプリングに基づいたプロファイラ (gprof のようなもの)

- 詳細プロファイラ *fapp*

  指定区間の浮動小数点演算数、MFLOPS値、SIMD化率、キャッシュ、
  メモリアクセス情報などの細かい情報が得られる

  – 同じジョブを8回動かしてデータを取得する必要あり

チューニングの流れ

- (なんらかのプロファイラを使って、ホットスポットを見つける)
- -Nsrc オプションをつけてコンパイルして最適化情報を見る

  C、C++ではパイプライン情報はコード横には表示されないが、メッセージとしては出力される
  - (例) line 74: Loop software pipelined
  - Fortran だとコードとともに表示されるらしい?

SIMD化、パイプライン化

されなかった

- されるようにコードを書き換える (以下実例)

された

- fapp を使って詳細解析
  - SIMD化率、キャッシュ利用効率等をチェック
  - 満足するまで最適化

サンプル

セッティング
ni=1024, nj=1024
座標は [0:1)で乱数
x[j][3]はj粒子の質量で1

- とりあえず右の教科書的コードを高速化することを考える
- B/F < 0.01
- 1コアで実行

- このままコンパイル、実行すると悲惨
  0.041 sec
  1.03 GFlops 、
  実行効率 6.41%

```
void gravity (...){
  for( int i=0; i<ni; i++){
    for( int j=0; j<nj; j++){
      double dx = xi[i][0] - xj[j][0];
      double dy = xi[i][1] - xj[j][1];
      double dz = xi[i][2] - xj[j][2];
      double r2 = dx*dx + dy*dy + dz*dz + eps2;
      double rinv = 1.0 / sqrt(r2);
      double mjr3inv = xj[j][3] * rinv * rinv * rinv;
      ai[i][0] -= mjr3inv * dx;
      ai[i][1] -= mjr3inv * dy;
      ai[i][2] -= mjr3inv * dz;
      pi[i] -= mjr3inv * r2;
    }
  }
}
```

レジスタを念頭に置く

- 毎 j での ai、pi への
  メモリ書き込みを防ぐようにする
- 一時変数を使って、常にレジスタ
  に置かれるようにする
  → パイプライン化、
  SIMD化を促進
- コンパイラに自動でやってほしい
  が、配列・ポインタの使い方に
  かなり依存するようである

- 0.0088sec
  4.66Gflops、実行効率29.12%
- Xeon E5 3.2GHz (gcc)では
  0.0132 → 0.0129 sec

```
for( int i=0; i<ni; i++){
  double ax = 0.0;  double ay = 0.0;
  double az = 0.0;  double p = 0.0;
  for( int j=0; j<nj; j++){
    double dx = xi[i][0] – xj[j][0];
    double dy = xi[i][1] – xj[j][1];
    double dz = xi[i][2] – xj[j][2];
    double r2 = dx*dx + dy*dy + dz*dz + eps2;
    double rinv = 1.0 / sqrt(r2);
    double mjr3inv = xj[j][3] * rinv * rinv * rinv;
    ax –= mjr3inv * dx;
    ay –= mjr3inv * dy;
    az –= mjr3inv * dz;
    p –= mjr3inv * r2;
  }
  ai[i][0] = ax;  ai[i][1] = ay;
  ai[i][2] = az;  pi[i] = p;
 }
```

手動アンロール

- j方向に手動で2アンロール
- 9.0Gflops、56.37%

- さらに性能を向上させたい場合は
  → 手動SIMD + アンロール

```
for( int j=0; j<nj; j+=2){
   double dx1 = xi[i][0] - xj[j][0];
   double dy1 = xi[i][1] - xj[j][1];
   double dz1 = xi[i][2] - xj[j][2];
   double r2_1 = dx1*dx1+dy1*dy1+dz1*dz1+eps2;
   double rinv1 = 1.0 / sqrt(r2_1);
   double mjr3inv1 = xj[j][3] * rinv1 * rinv1 * rinv1;
    ax1 -= mjr3inv1 * dx1;
    ay1 -= mjr3inv1 * dy1;
    az1 -= mjr3inv1 * dz1;
    p1  -= mjr3inv1 * r2_1;
    double dx2 = xi[i][0] - xj[j+1][0];
    double dy2 = xi[i][0] - xj[j+1][1];
    double dz2 = xi[i][0] - xj[j+1][2];
    double r2_2 = dx2*dx2+dy2*dy2+dz2*dz2+eps2;
    double rinv2 = 1.0 / sqrt(r2_2);
    double mjr3inv2 = xj[j+1][3] * rinv2 * rinv2 * rinv2;
    ax2 -= mjr3inv2 * dx2;
    ay2 -= mjr3inv2 * dy2;
    az2 -= mjr3inv2 * dz2;
    p2  -= mjr3inv2 * r2_2;
  }
```

```
#define v2r8_rcp    __builtin_fj_rcpa_v2r8
#define v2r8_rsqrt  __builtin_fj_rsqrta_v2r8
#define v2r8_madd   __builtin_fj_madd_v2r8
#define v2r8_msub   __builtin_fj_msub_v2r8
#define v2r8_nmadd  __builtin_fj_nmadd_v2r8
#define v2r8_nmsub  __builtin_fj_nmsub_v2r8
#define v2r8_abs    __builtin_fj_abs_v2r8
#define v2r8_and    __builtin_fj_and_v2r8
#define v2r8_cmplt  __builtin_fj_cmplt_v2r8
#define immr8(x)    __builtin_fj_set_v2r8(x, x)

static v2r8 gp3m_force(const v2r8 r2, const v2r8 r, const v2r8 rinv){
        const v2r8 mask = v2r8_cmplt(r2, immr8(4.0));
        const v2r8 s    = v2r8_sub(v2r8_max(r, immr8(1.0)), immr8(1.0));
        const v2r8 s2   = v2r8_mul(s, s);
        const v2r8 s6   = v2r8_mul(s2, v2r8_mul(s2, s2));
        const v2r8 poly1 =
                v2r8_madd(
                        v2r8_msub(
                                v2r8_madd(
                                        v2r8_msub(
                                                v2r8_msub(
                                                        immr8(3./20.),
                                                        r,
                                                        immr8(12./35.)),
                                                r,
                                                immr8(1./2.)),
                                        r,
                                        immr8(8./5.)),
                                r2,
                                immr8(8./5.)),
```

11.32Gflops、70.76%

手動SIMD、アンロール

This implementation is done by **Keigo Nitadori** as an extension of Phantom-GRAPE library (Nitadori+ 2006, Tanikawa+ 2012, 2013)

$$g_{\mathrm{P3M}}(R) = \begin{cases} 1 - \frac{1}{140}\left(224R^3 - 224R^5 + 70R^6 + 48R^7 - 21R^8\right) & (0 \le R \le 1) \\ 1 - \frac{1}{140}\left(12 - 224R^2 + 869R^3 - 840R^4 + 224R^5 + 70R^6 - 48R^7 + 7R^8\right) & (1 \le R \le 2) \\ 0 & (2 \le R) \end{cases}$$

1. One if-branch is reduced
2. optimized for a SIMD hardware with FMA

$$g_{\mathrm{P3M}}(R) = 1 + R^3\left(-\frac{8}{5} + R^2\left(\frac{8}{5} + R\left(-\frac{1}{2} + R\left(-\frac{12}{35} + R\frac{3}{20}\right)\right)\right)\right) - S^6\left(\frac{3}{35} + R\left(\frac{18}{35} + R\frac{1}{5}\right)\right) \qquad (0 \le R \le 2)$$

$$S \equiv \max(0, R-1),$$

- Theoretical limit is 12Gflops/core (16Gflops for DGEMM)
- **11.65** Gflops on a simple kernel benchmark
  - **97**% of the theoretical limit (73% of the peak)

条件分岐がある場合

- Continue は パイプライン化、SIMD化を著しく阻害
- ループの j 依存性が解析困難に
- **1.2 Gflops、7.53%**

```
for( int j=0; j<nj; j++){
    double dx = xi[i][0] - xj[j][0];
    double dy = xi[i][1] - xj[j][1];
    double dz = xi[i][2] - xj[j][2];
    double r2 = dx*dx + dy*dy + dz*dz + eps2;
    if( r2 > 0.9)  continue;
    double rinv = 1.0 / sqrt(r2);
    double mjr3inv = xj[j][3] * rinv * rinv * rinv;
    ax -= mjr3inv * dx;
    ay -= mjr3inv * dy;
    az -= mjr3inv * dz;
    p  -= mjr3inv * r2;
  }
```

SIMDを利用するようにする

- 演算量は増えるが、パイプライン化、SIMD化が有効に
- **9.6Gflops、59.74%**
- Xeon E5 3.2GHz (gcc)では
  0.0116 → 0.0129 sec
  逆に演算量が増えた分遅くなっている
- 以下の形ならSIMD化可能
  (fsel 命令)

```
If(a>0)  b=c
else     b=d
または
b = (a>0) ? c : d
```

min、max等にも使える

```
return (a-b>0) ? a : b
```

```
for( int j=0; j<nj; j++){
    double dx = xi[i][0] - xj[j][0];
    double dy = xi[i][1] - xj[j][1];
    double dz = xi[i][2] - xj[j][2];
    double r2 = dx*dx + dy*dy + dz*dz + eps2;
    double f = 1.0;
    if( r2 > 0.9){
      f=0.0;
    }
    double mjr3inv = f * xj[j][3] * rinv * rinv * rinv;
    double rinv = 1.0 / sqrt(r2);
    double mjr3inv = xj[j][3] * rinv * rinv * rinv;
    ax -= mjr3inv * dx;
    ay -= mjr3inv * dy;
    az -= mjr3inv * dz;
    p  -= mjr3inv * r2;
  }
```

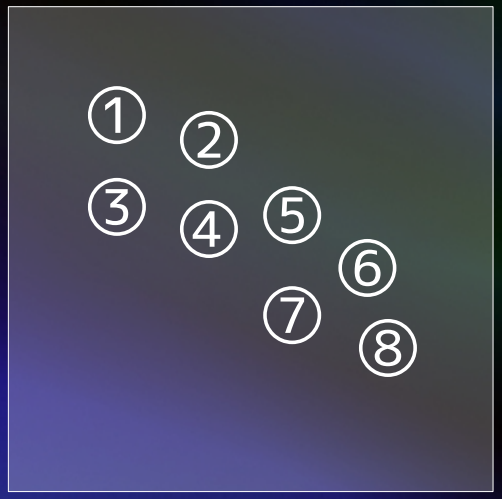
性能のまとめ

- 京では、パイプライン化、SIMD化してくれるコードの書き方がある
  - 適当に書けば intel 系が速いことが多い。特に条件分岐
- 京は gcc ではやってくれない SIMD化をしてくれることもある
  - コードがコンパイラに理解できるように書いてあれば
  - 手動SIMDよりはやや劣る
  - 演算密度が高くないとどうしようもない
- 手動SIMDするとピークに近い性能を出せることもある
  - ただし SSE、AVX (gcc) でも出る (Tanikawa et al. 2012, 2013)

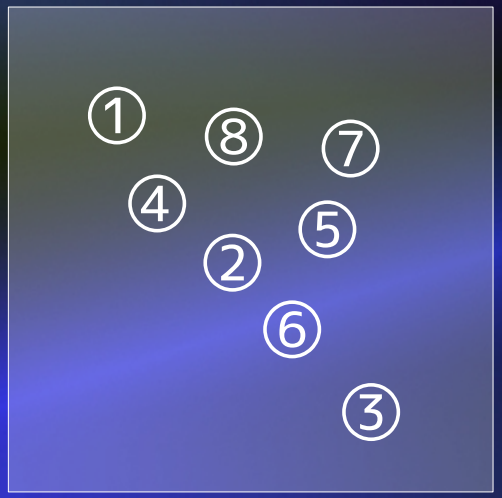
メモリアクセス最適化

粒子のソート1

シミュレーション空間



時間発展



メモリ空間
(ソートなし)

メモリ空間
(ソートあり)

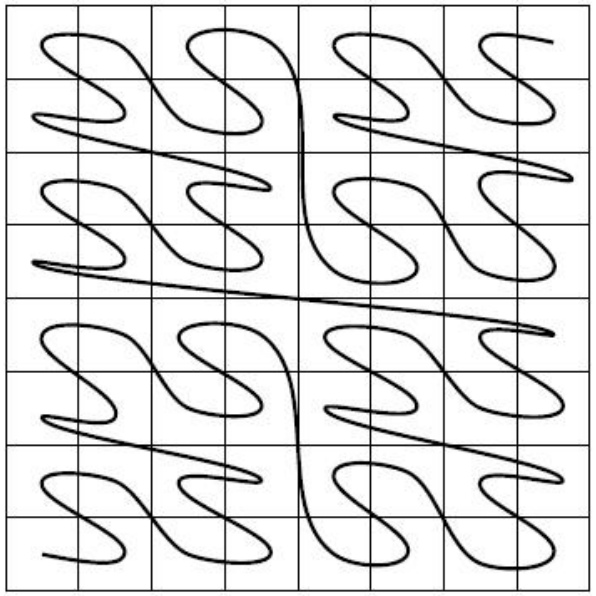
- 粒子の時間発展とともに、空間上の近接関係とメモリ上のものが大きくずれる

  → ツリー生成、ウォークの際にランダムアクセス、キャッシュミスが頻発

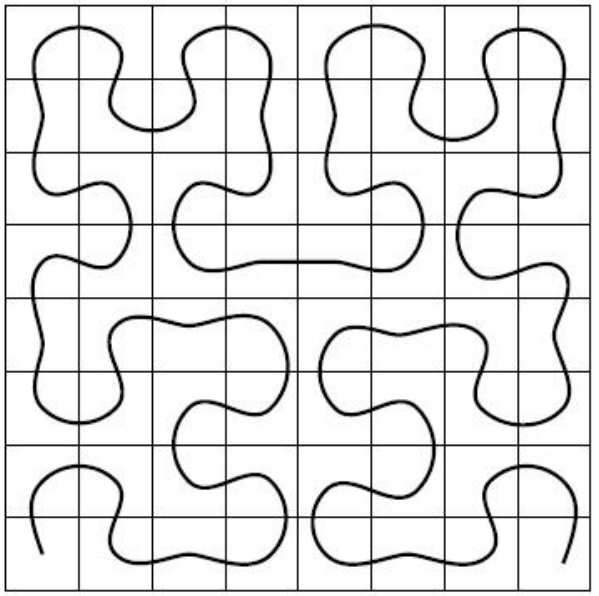
- 粒子を適度な頻度でソートする
  - 近接粒子が同じキャッシュラインにのり、メモリアクセスが最適化

メモリアクセス最適化

粒子のソート2



Morton (z) ordering



Peano hilberto ordering

Warren et al. 1992

- 空間充填曲線を用いて粒子をソート (本コードでは morton)
  - ツリー構築は4割程度
  - ツリーウォークは1割程度高速に

その他

大規模データをどう解析するか?

- Output は 1粒子あたり 32bytes
  - 4096^3粒子なら 2TB/snapshot、10240^3粒子なら 32TB/snapshot
  - 計算ノード数で分割、または数ノードで一つにまとめる
  - 個々の分割ファイルがどの領域の粒子を持っているかを記憶

    → 一部の領域の粒子が欲しいときに一部ファイルだけ読むようにする

- 粒子からの可視化は諦める (スナップショット数をそれほど多くできない)
  - 可視化に必要な情報だけを粗くして、細かい時間刻みで output

- 全粒子の情報が必要な解析は並列にやるか on the fly
  - Halo finding、 power spectrum など

- 解析は (自前の)ファイルサーバーのIO性能ネックになることも
  - SSD RAID0 中間ファイルサーバーを導入、実験中

京を使っていて今苦戦しているところ

- ステージアウトの時間は2時間まで
- 4096^3 シミュレーションだと 1 snapshot 2TByte
- 1 snapshot を 8192 分割すると 200~300MByte/file

- 5 snapshot 出力すると合計 10TByte、40000 file

  → ステージアウトが終わらない
  (このサイズだと2時間で30000file程度が限界)

- 4ノードで1ファイルにまとめた場合 → 10000 file

  → ステージアウトが終わることもあれば終わらないこともある

- とりあえず実行時間を短く (1ジョブあたりのスナップショット数を少なくする)して、その場しのぎ

まとめ

- 8万並列66万CPUコア (京の全システム) まで良くスケールする重力多体(N体)シミュレーション用コードを開発した

- 並列化のポイント
  - トーラスマッピング
  - MPI_COMM_WORLD の分割

- 単体チューニングのポイント
  - コンパイラが理解できるようなコードを書く
    SIMD化、ソフトウェアパイプライニングの促進
  - キャッシュ利用効率を高める