大規模シミュレーションと大規模データ

石山 智明

筑波大学計算科学研究センター神戸分室

いただいたお題

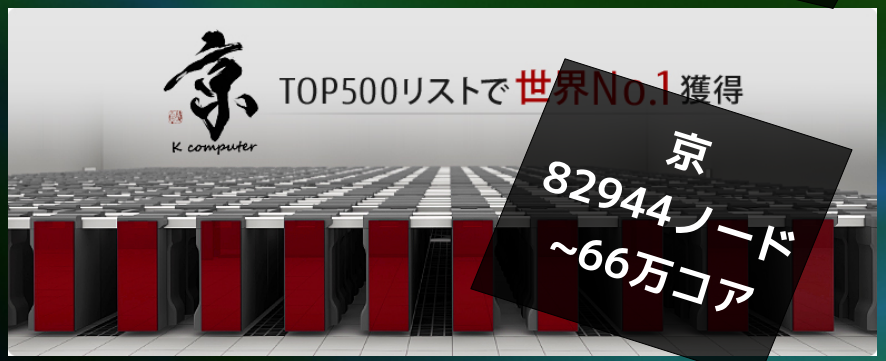
- **「計算機科学ーシミュレーション天文」に関する セッションで天文アプリ**

  というわけで

  サイエンスの話は少しにして、大規模シミュレーションを実現するために、どのようにアプリ開発 (並列化、チューニング)が裏で行われてきたかや、大規模データをどのように解析してきたかについて、実体験に基づきお話しいたします。

近年のスパコンの主流

- 大規模スカラー型
  - いわゆる汎用CPUを高速ネットワークで大量につなぐ



- さらにアクセラレータ (GPU, MIC)をのせる

©NVIDIA

©Intel

何が大変か

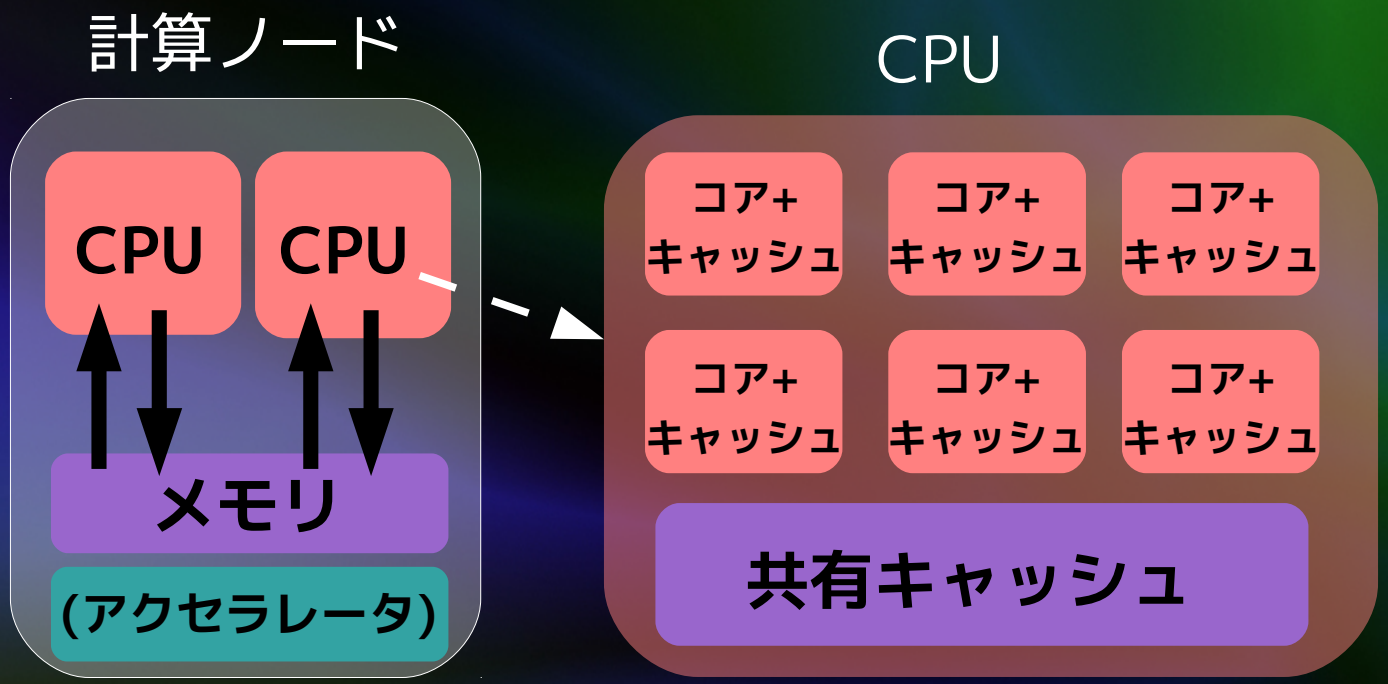
- コード開発の負担
  - 深いノード内並列処理の階層
    - ベクトル演算 (CPU内: SIMD)
    - マルチコア、共有メモリ (ノード内)
    - 分散メモリ (ノード間)
    - アクセラレータ、コプロセッサ (GPU, GRAPEs, Intel MIC, PEZY …)
  - 増大する並列数 (CPU、ノード数)
    - 京では最大82944ノード
  - アーキテクチャ依存の最適化

古くからあるコードそのままでは、これらの並列処理の恩恵を受けられることはない

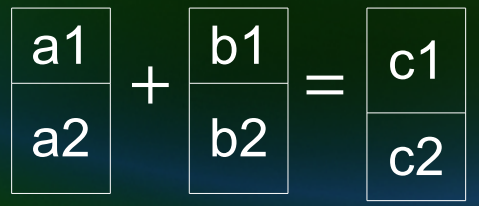
- 大規模シミュレーション実現 = 大規模データの解析

ノード内の並列階層

- それぞれ並列化のためのプログラミング手法が異なる



CPUコア内SIMD
(ベクトル演算)



- 性能を出すためには
  - 全コアを平等に使う (OpenMP)
  - SIMDが効くようにプログラミングする (handy-SIMD)
  - キャッシュの有効利用 (データのブロック化など)

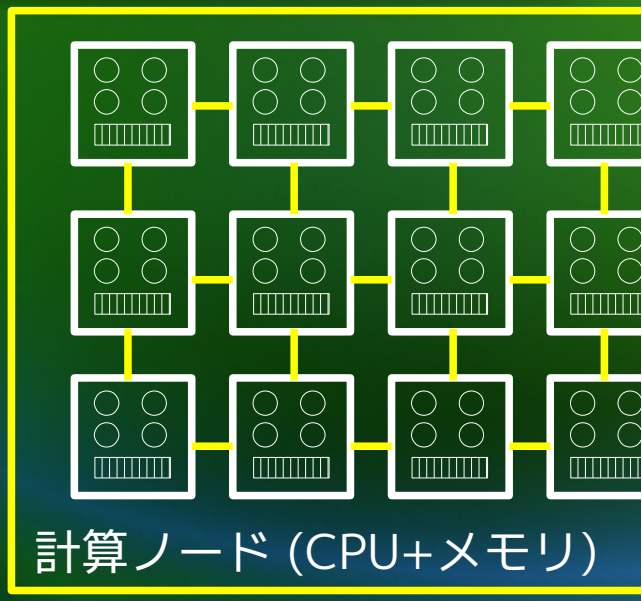
並列計算のスケーリング

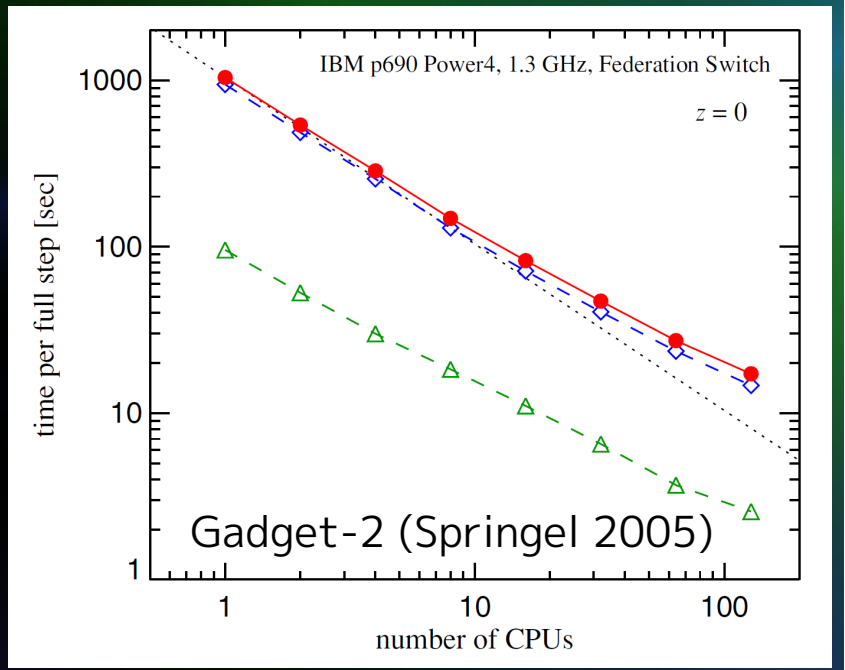
- 理想的には CPU 数を n 倍にしたら n 倍の規模の計算が、または同じ規模の計算が 1/n の時間でてきてほしい (スケーリング)

- 特に重力は遠距離力なので、並列数が増えるほど辛くなる
  - 例えばGadget-2 は 1000並列以上で良くスケールする気がしない
- でも粒子数を増やしたい or もっと速く計算したい

  →**新しい並列アルゴリズム、コード開発**

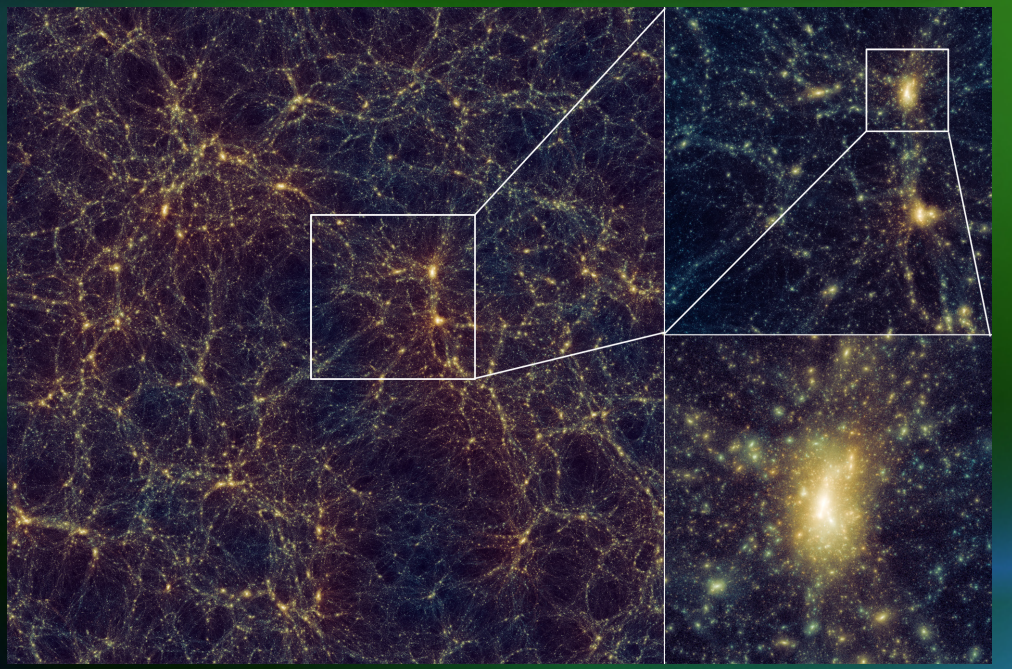
- 並列化プログラミングは基本的にはMPIというインターフェースを使って行う





アプリ例と使用しているスパコン

- 重力多体シミュレーションコード
  GreeM (Ishiyama et al. 2009, 2012)
- TreePM Poisson solver
- 再帰的多段領域分割
- 使用する計算機に合わせた、
  重力相互作用演算部の最適化
- Flat MPI または MPI+OpenMP の
  ハイブリッド並列
- Gadget-2 の2~6倍速い
- 京では1兆粒子規模のシミュレーションが可能に



京
82944ノード
~66万コア

CfCA アテルイ
1060ノード
25440コア

ノード内並列化

ハイブリッド並列

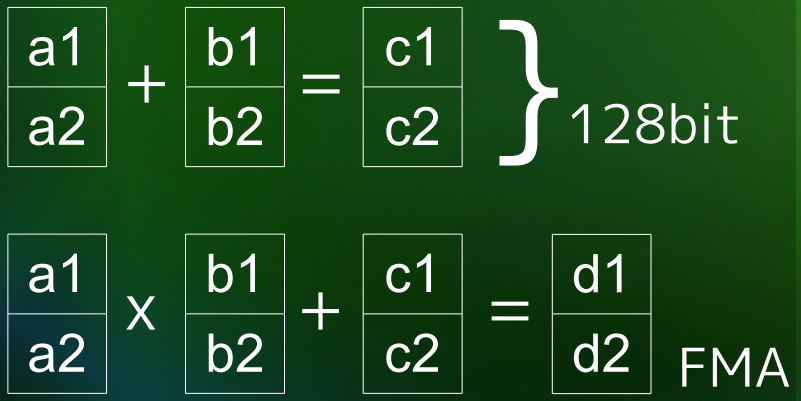
- 従来は個々のCPUコアを異なるスレッドで動作させるプログラミングが主流であった
  - いわゆる Flat MPI

- CPUコア数の増大 or アクセラレータの搭載
  →演算性能に対するネットワーク性能の低下
  - 京のノードあたり同時通信数 4 < コア数 8

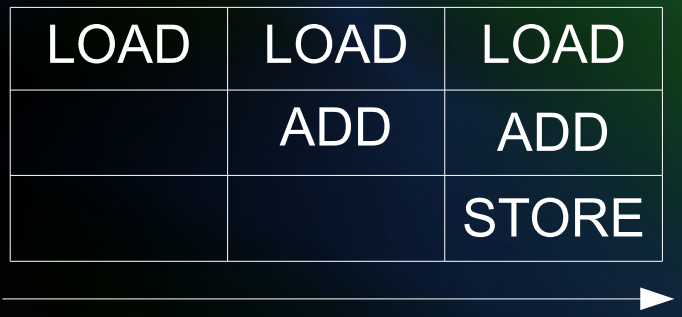
- Flat MPI ではなくハイブリッド並列にする
  - ノード内は OpenMP 並列。ノード間はMPI並列
  - プログラミングコストの増大
  - 単純なループでないと並列化性能が出にくい
    - Flat MPI の方が速いことも多い。だが未来は?

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
  - Intel AVXは256bit、単精度8、倍精度4演算
  - SIMD化 できないと最大でもピークの 1/8 の
    性能しか出ない
  - 演算密度が高くないと効果は薄い
  - 京コンパイラではSIMD化されるような
    コードの(特殊な)書き方がある。



- **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
  - Intel系上ではコンパイラがうまいことやってくれるが……



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最適化することを考える
- 1コアで実行
- 京上で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%
- Xeon E5 3.2GHz (gcc)では
  0.0132 sec

```
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、キャッシュを念頭に置く

- 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、キャッシュ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.)),
                        r2,
                        rinv);
        const v2r8 poly2 =
                v2r8_nmsub(
                                v2r8_madd(
                                        immr8(3./35.),
                                        rinv,
                                        v2r8_madd(
                                                immr8(1./5.),
```

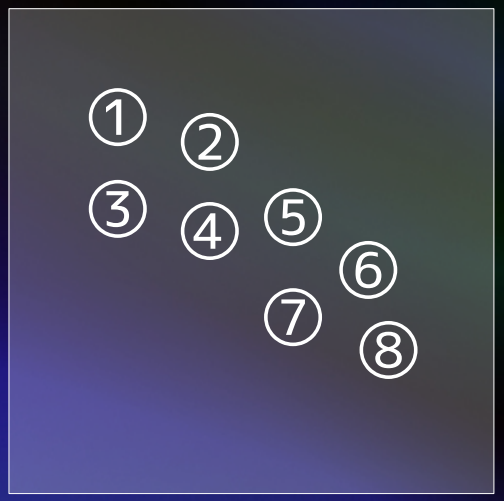
Using Phantom-GRAPE library
(Nitadori+ 2006,
Tanikawa+ 2012, 2013)

11.32Gflops、70.76%

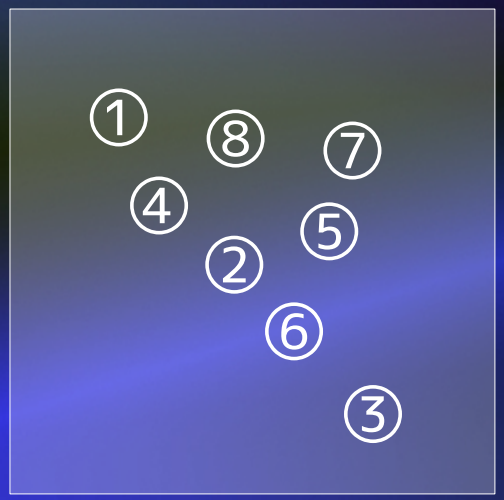
メモリアクセス最適化

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)

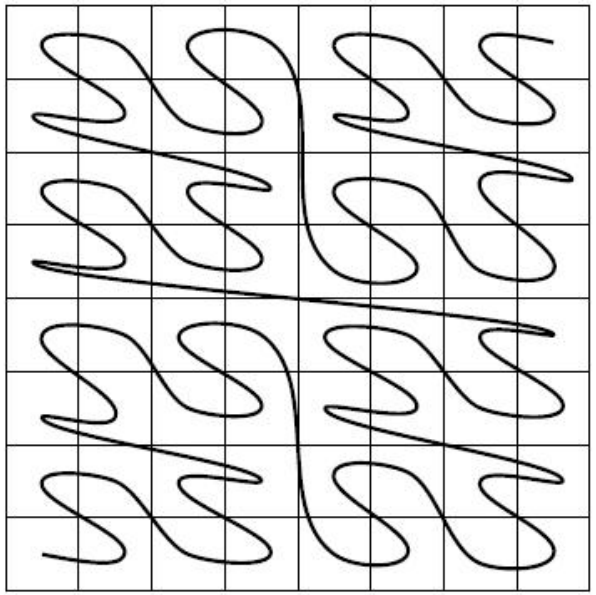
- 粒子の時間発展とともに、空間上の近接関係とメモリ上のものが大きくずれる

  → 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

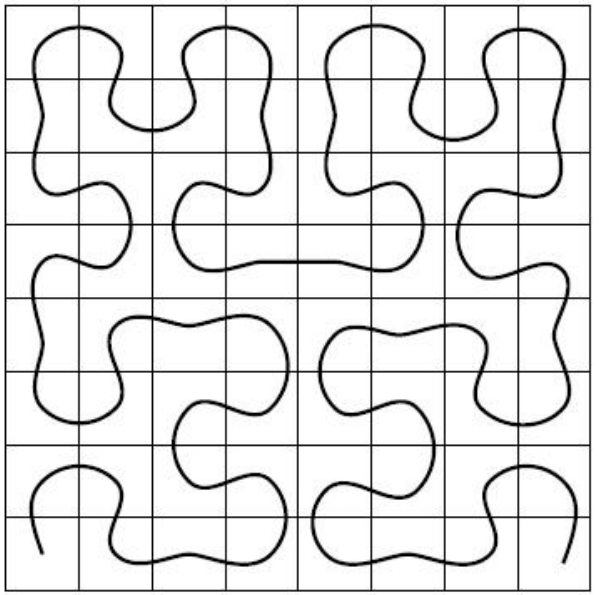
- 粒子を適度な頻度でソートする
  - 近接粒子が同じキャッシュラインにのり、メモリアクセスが最適化

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

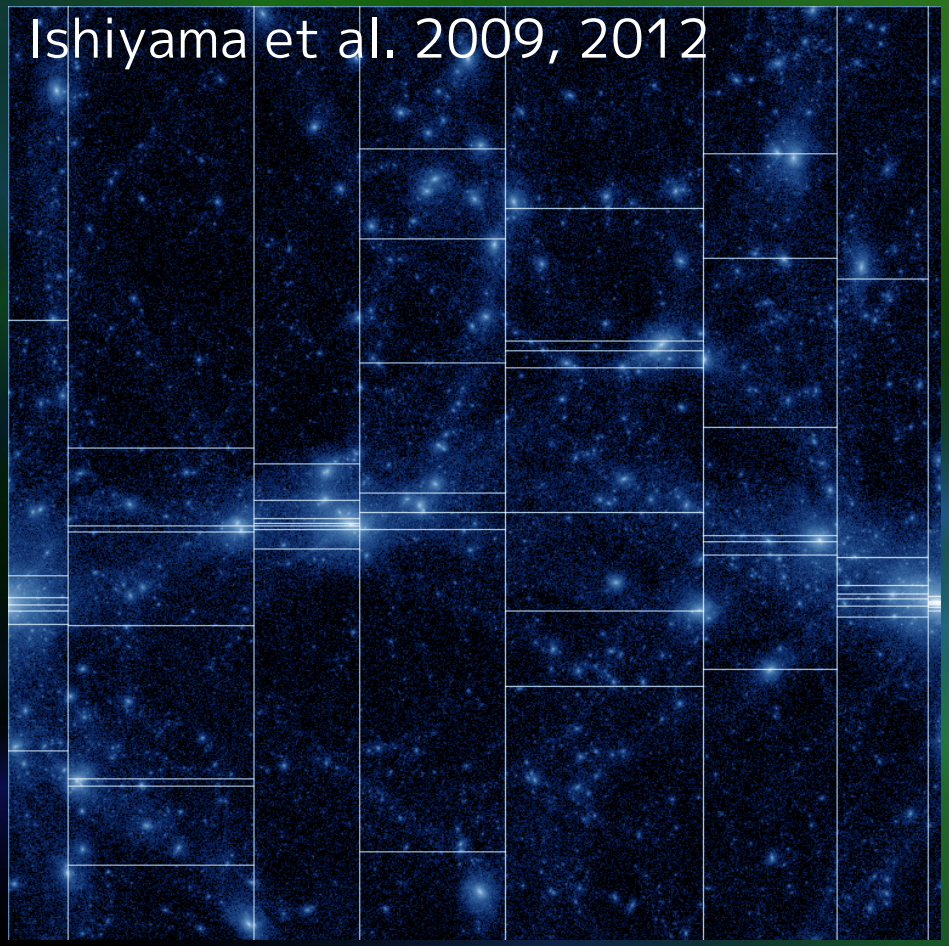
大雑把には

- 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)

ノード間並列

分散メモリ上での並列化

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



領域分割の方法はいろいろと流儀があるが、
構造の進化にあわせて動的に再設定して
いかないと、並列化効率が非常に悪化する

82944ノードまで使ってみた感想

- (コードによってはそんなことはないが)
  数千ノードまでは、数百ノードからの延長線上

- 数千ノード以上

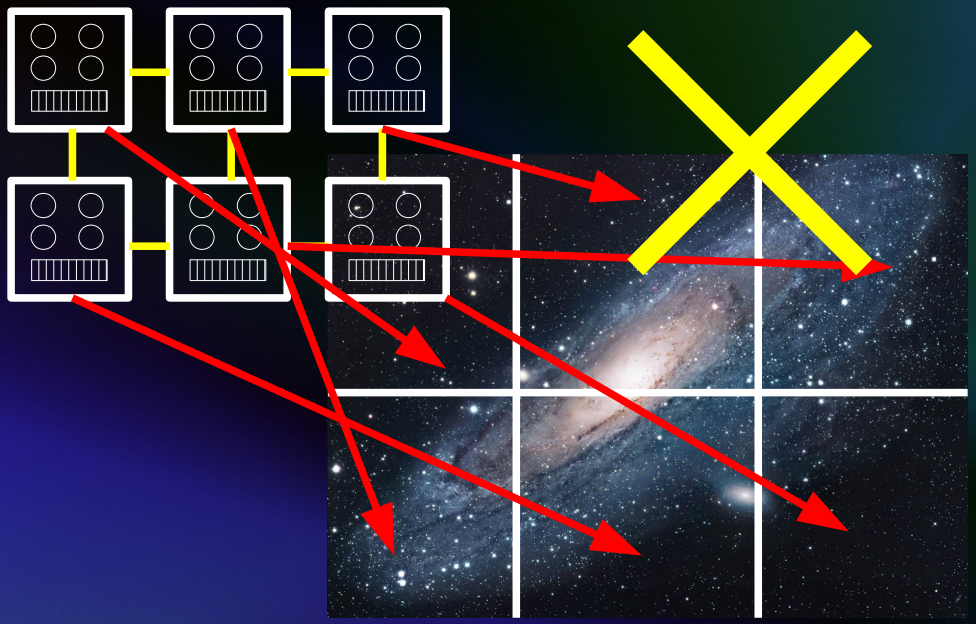
  長距離通信の性能低下が見え始めてくる

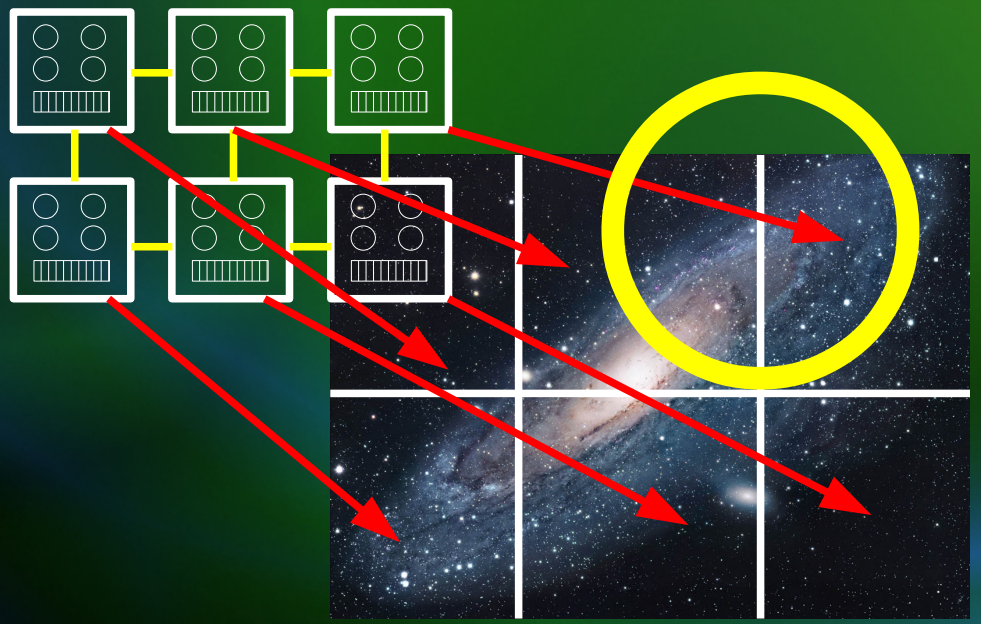
- 数万ノード以上

  ボトルネックになることも

ノードの物理配置とシミュレーション空間でのノード配置をマッピングすることと、全対通信の分割(階層化)で改善

マッピングによる通信最適化





京ではノードの物理座標を取得する
APIが公開されている

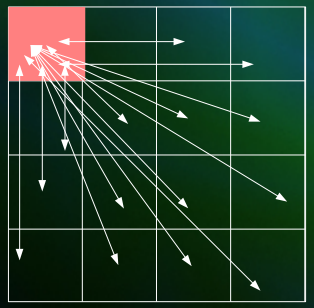
- 4096 ノード、16x16x16の空間分割で合計20MBの Allgather をした場合
  - ノード形状 10x15x28 0.06 sec
  - 16x16x16 0.04 sec

ただしネットワークトポロジーが
3Dのようなものとは限らない

大規模な全対通信

- 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
  - 京では1ノードの同時通信数は8

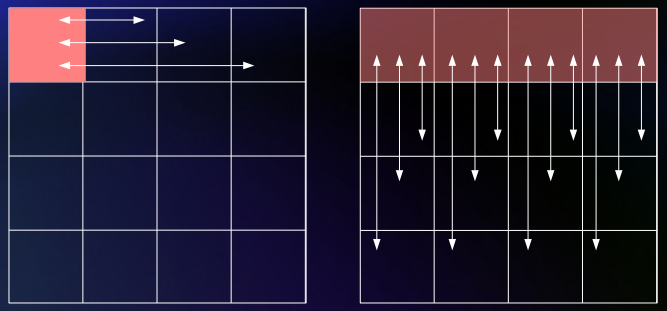


MPI_Alltoallv
(…, MPI_COMM_WORLD)

$O(p)$

- 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

- 1024ノードに分散した、合計384GByte のデータを4096ノードに分配する場合
  - 1回で通信 → **17.7 sec**
  - 階層的通信 → **4.2 sec**



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

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

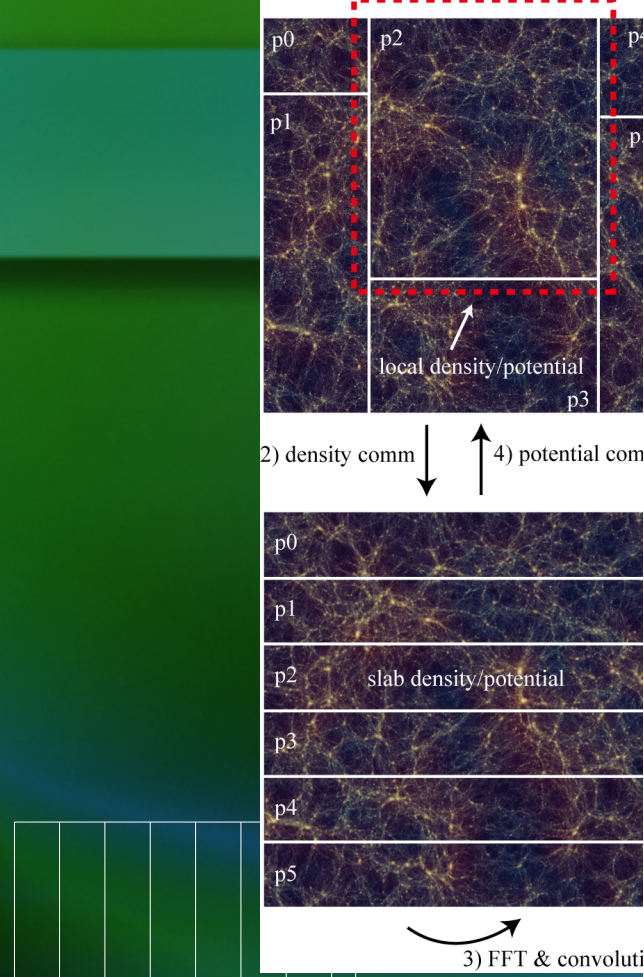
適用例

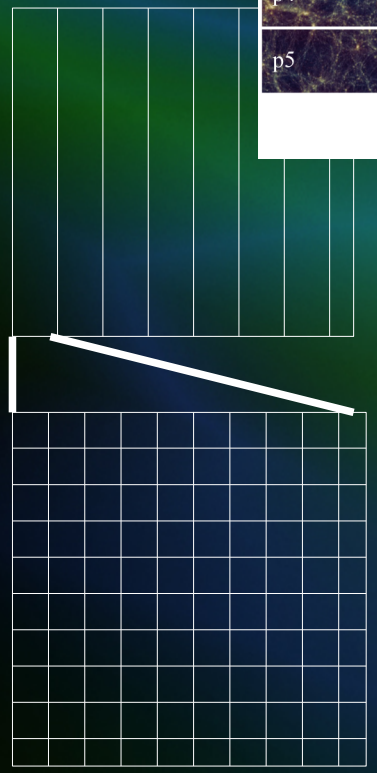
並列PM

- 空間分割は不規則三次元
- FFTライブラリでは、データ構造は各ノードで一様に
  1or2or3 次元分割されている必要がある

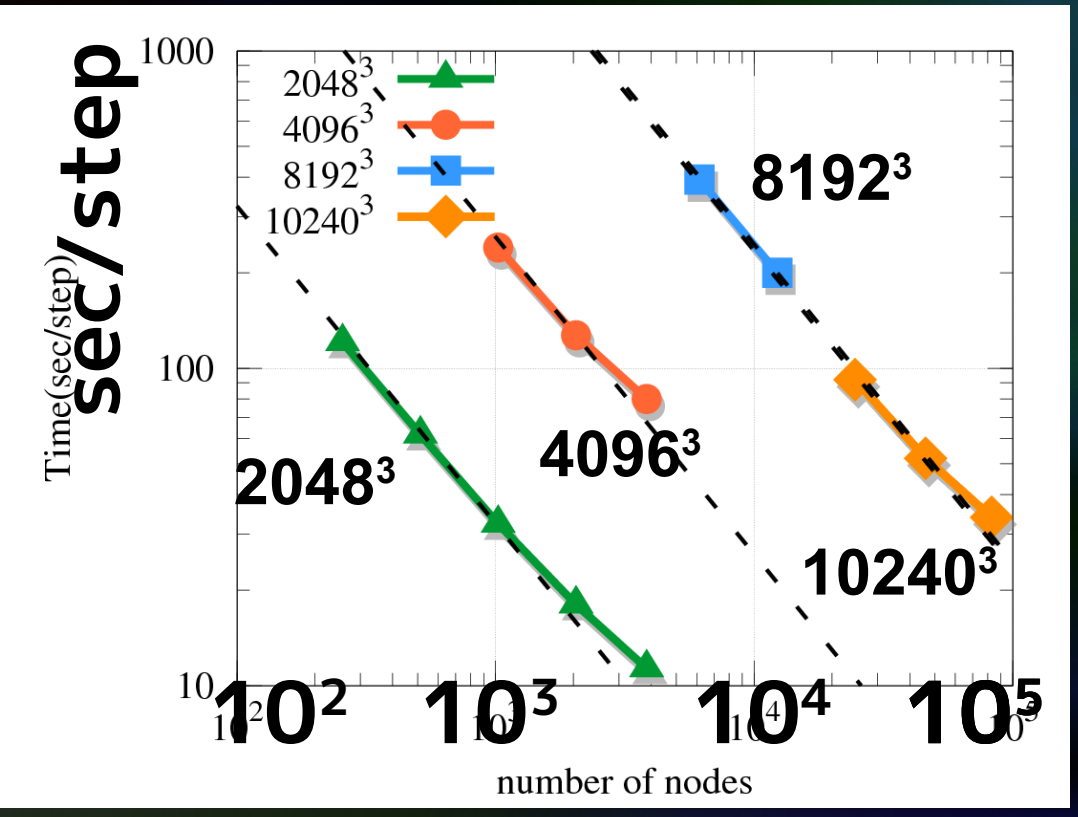
- 特にデータ構造を 3D->1Dにした場合、
  たくさんのノード → 1ノードへの通信が発生
  - 場合によっては数千。通信競合が発生
  - 2D、3D FFT なら幾分ましではある

- 通信を二段階に分けることで3〜4倍の高速化に成功
- MPI_Alltoallv (..., MPI_COMM_WORLD)
  - MPI_Alltoallv (..., COMM_SMALLA2A)
  - MPI_Reduce(..., COMM_REDUCE)





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

大規模データの取扱い

～宇宙論N体を例に～

よくある質問

- データ解析
  - 解析コードも並列化しているのか?
  - 並列化してなかったら、計算時間やIOがボトルネックでは?

- ムービー作成
  - ファイルサイズが巨大過ぎて、
    - 可視化ツールが使えない
    - 時間がかかりすぎる
    - ムービー作るのに必要なだけのスナップショットを保存できないのでは?

- 設備
  - どんな計算機で解析しているのか?
    スパコンか、計算サーバーか?

データ解析

- 全粒子の情報が必要な解析はスパコン上で並列にやるか on the fly
  - Halo finding、 power spectrum など

- ただし解析中はいろいろとパラメータをいじったり、
  結果を見てリトライしたくなるものである
  - スパコン上での解析はやってられないこともある
    - ジョブが動くまで時間がかかる (特に大規模であればあるほど)
  - 数日くらいの計算時間で済むなら分散並列化せず計算サーバでやる
    - ジョブ並列やOpenMPを適当に活用する

IOがネックの場合

- 4096^3粒子なら 2TB/snapshot、8192^3粒子なら 16TB/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
  - 100MB/s (1GBネットワークの限界くらい)で読み込んでも、
    全データを読み込むのに 20,000 (160,000) 秒かかる

- 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
  - 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ヘッダか別ファイルに記録

0 (IO format)
n (Number of particles)
0 0.5 0 0.5 0 0.5 (領域情報)
. (header)
.
particle 0 (position, velocity, ID)
particle 1
.
.
particle n-1

ムービー作成

- ファイルサイズが巨大過ぎて、
  - 可視化ツールが使えない
  - 時間がかかりすぎる
  - ムービー作るのに必要なだけのスナップショットを保存できないので

  フルデータを使ってプロにカッコいいムービーを作ってもらうのは諦める

- 広報用のムービーは全粒子を使わなくて良いことにする
  - 適当にまびいて(ex: 4096^3->1024^3)プロにおまかせ
  - ムービー専用のデータを作成して細かい時間間隔で保存
    - 16bit固定小数点や半精度(16bit浮動小数点)でよいことにする

- 解析のためのムービー、可視化は自前のツールでなんとかする

データをどこで解析するか?

CfCAの場合

- スパコン上にある高速ファイルシステムは、ユーザあたりの使用可能量が厳しく制限されている
  - シミュレーション規模、ラン数をリミット
- スパコン本体とは別にファイルサーバや、計算サーバが用意されていて非常に便利
  - 京にもある
  - 転送しながらシミュレーションを継続する
  - IOやネットワーク性能はスパコン本体より劣る
  - 全ユーザで共有
    - やっぱりデータ使用可能量に上限があり、シミュレーション規模、ラン数をリミット

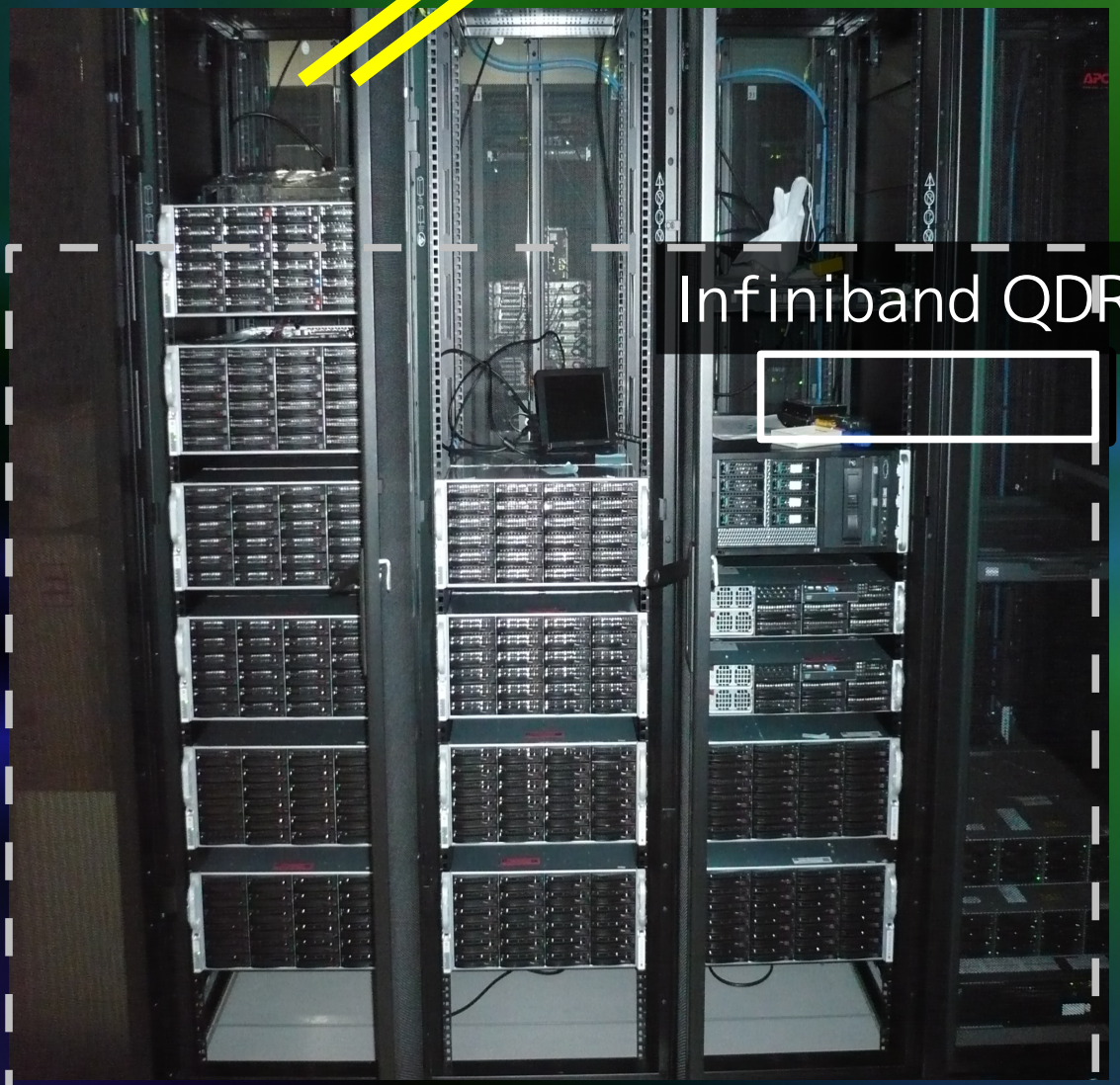
結局は自前で

解析サーバーシステムの構築

- 財源はHPCI戦略プログラム5

- 128GBメモリの計算サーバ x2
- ファイルサーバ 9ノード
  - 総容量およそ1PB
  - 1ノードは256GBメモリの計算サーバを兼ねる (IOが一番速く使いやすい)

- 京とのデータ転送は10Gなので高速。1日に5〜10TBは転送できる
- ただしCfCAとは、、、一日に1TB転送できない
- 物理的に配達が最速



大規模宇宙論的N体の役割

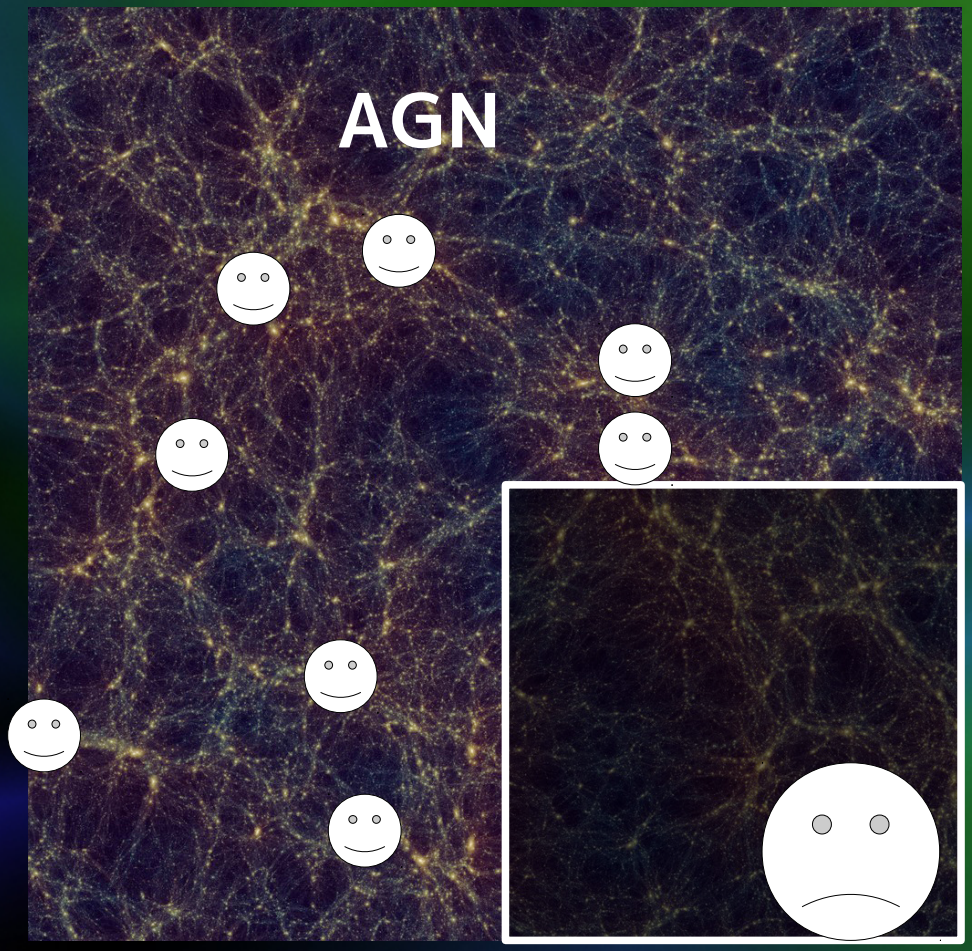
- ダークハローの分布、構造の研究
- 準解析的銀河形成モデルと組み合わせる
  - モデル上ではバリオンの進化を、ダークマターハローの合体による階層的構造形成史 (merger tree)の枠組みのなかで現象論的に解いていく
  - 大きいボックスサイズをとれる (rare object の形成)
  - Press Shecheter やその派生でも merger tree を得られるが、空間情報が得られない

- 銀河形成シミュレーションは、個々の銀河や100Mpc立方くらいの領域がまだ限界
- 星形成や超新星などはサブグリッドで、結果がモデルに依存する

HSCによるAGNの観測と比較するためには

- High-z AGNの個数密度
  ~10^{-6} Mpc^{-3}
- **数百Mpc立方以上の領域**を
  シミュレーションする必要有
- 銀河の階層的構造形成は追いたい
  (ある程度高い質量分解能)

  → **必要な粒子数 4096^3 以上**

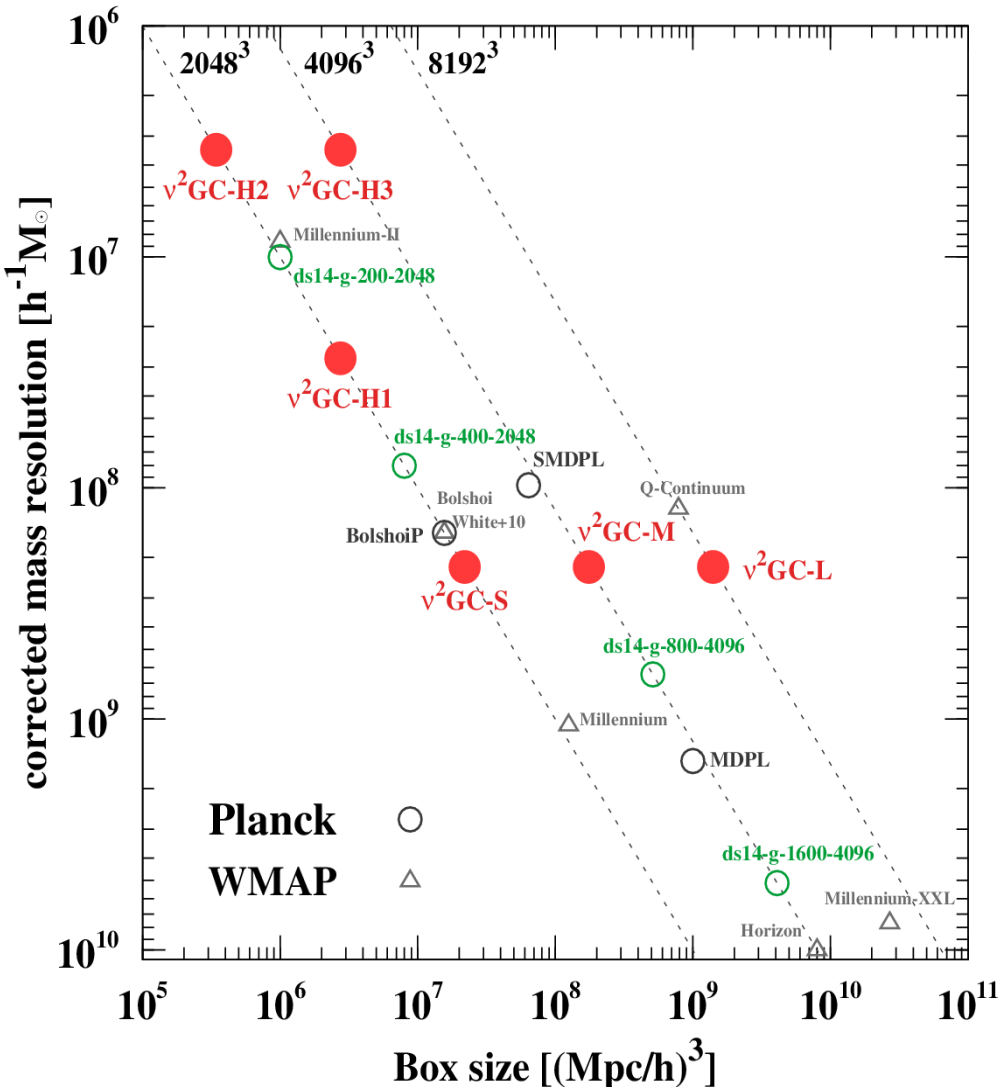


ν²GC simulation suite

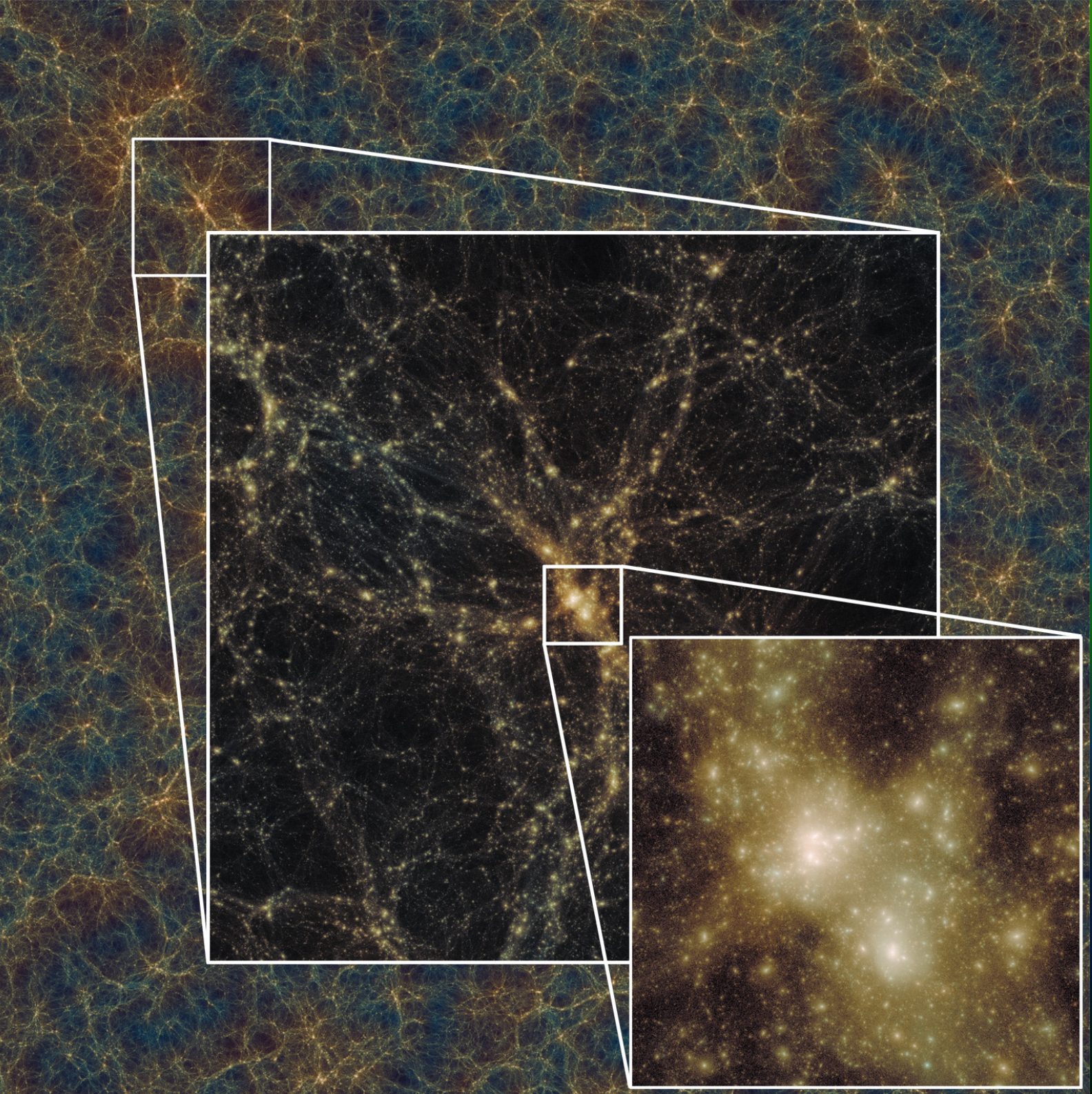
Ishiyama, Enoki, Kobayashi, Makiya, Nagashima, Oogi, arXiv:1412.2860

- Millennium simulation (Springel+05)に比べ
  - **11倍**大きい空間体積
  - **4倍**良い質量分解能

- 宇宙論も最新(Planck)
- 様々な空間体積、質量分解能のラン
  → **Low- and high-z 銀河・AGN を幅広くカバー**



Name	N	$L(h^{-1}\mathrm{Mpc})$	$m(h^{-1}M_\odot)$	$\varepsilon(h^{-1}\mathrm{kpc})$
ν^2GC-L	$8192^3 = 549,755,813,888$	1120.0	2.20×10^8	4.27
ν^2GC-M	$4096^3 = 68,719,476,736$	560.0	2.20×10^8	4.27
ν^2GC-S	$2048^3 = 8,589,934,592$	280.0	2.20×10^8	4.27
ν^2GC-H1	$2048^3 = 8,589,934,592$	140.0	2.75×10^7	2.14
ν^2GC-H2	$2048^3 = 8,589,934,592$	70.0	3.44×10^6	1.07



$N = 8192^3 = 549,755,813,888$

$L = 1.12$ Gpc/h
$m = 2.2 \times 10^8$ Msun/h

Planck Cosmology

16,384 nodes
(131,072 CPU cores)
on K computer

スナップショットは
800TB!
→
Merger tree
にすると**850**GB

- 新しい準解析的銀河形成モデル, ν²GC (**N**ew **N**umerical **G**alaxy **C**atalog: Makiya+ in prep, ポスター72)と組み合わせ、疑似カタログを作成中
  - 銀河、AGNのカタログはそのうち公開
  - 関連するポスター: 10 (榎さん), 11(大木さん), 26(小林さん), 33(白方さん), 72(真喜屋さん)
  - lightcone も生成可能

- 一部ハロー、サブハローカタログは公開済
  - 2048^3、280Mpc/h、140Mpc/h、70Mpc/h のシミュレーション
  - z=0, 1, 3, 7
  - 要望があれば他のデータも……

    http://www2.ccs.tsukuba.ac.jp/Astro/Members/ishiyama/nngc/

- 個人で扱えるデータ量に限界を感じてきた……

まとめ

- 各階層それぞれでの並列化が重要であるが、階層ごとにプログラミングモデルが異なり、非常にコスト高
  - 汎用CPUでも三段階 (SIMD, CPUコア、ノード間)
  - さらにアクセラレータを使うとなると……
  - 最適化しないと大規模シミュレーションができない

- 大規模シミュレーションからうまれる巨大なデータがシミュレーション規模やラン数をリミット
- 巨大データを一個人で管理するにも限界がある