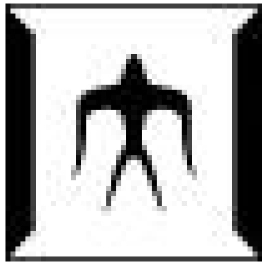
高性能計算のプログラミングの最前線

東京工業大学
学術国際情報センター/
数理・計算科学専攻
遠藤敏夫



私が使ってきたシステム(1)

Sun Enterprise 10000
Ultra SPARC x 64CPU
Share memory

SGI Origin 2000
R10000 x 128CPU
Share memory (NUMA)

IBM/Appro Blade cluster
Xeon x 2CPU x 200node
SMP cluster

米澤研においては、共有メモリマシンで
ガーベージコレクションの並列化
(64プロセッサ30倍のスケーラビリティ)

私が使ってきたシステム(2)

TSUBAME 1 (NEC/Sun)

Opteron x 16 CPU core x 655 node
+ClearSpeed x 360 board
+Tesla S1070 x680GPU

2006—07において日本最速

TSUBAME 2.0⇒2.5 (NEC/HP)

Xeon x 12core x 1408 node
+Tesla M2050 x4224GPU

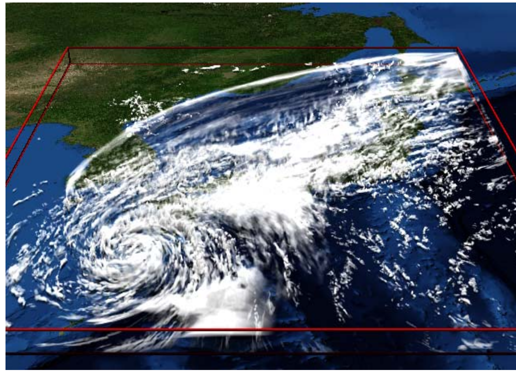
2010/11/1 稼働
日本初のペタコン
京に次ぐ国内二位
世界13位

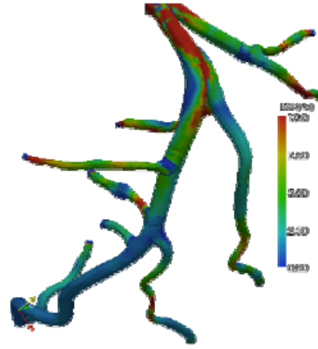
スーパーコンピューター

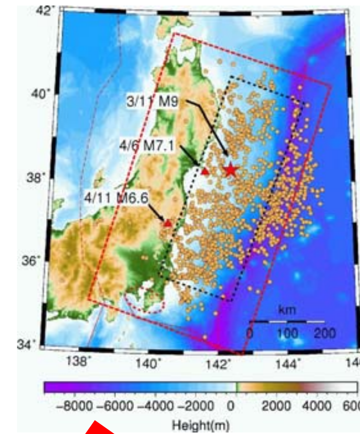
- 内部の演算処理速度がその時代の一般的なコンピュータより極めて高速な計算機
- 例: 京コンピュータ、東工大TSUBAME2、Tianhe-2, ・・・

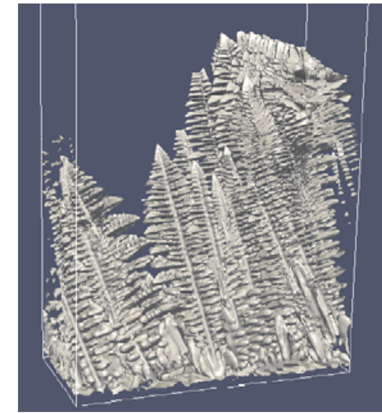
スパコンは何に使われる？

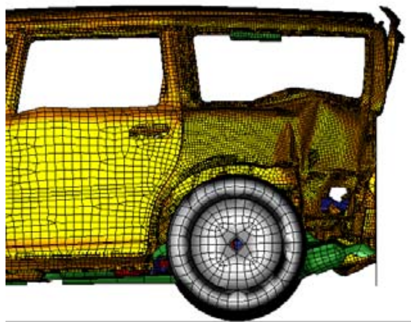
スパコンはあらゆる科学分野の仮想実験場

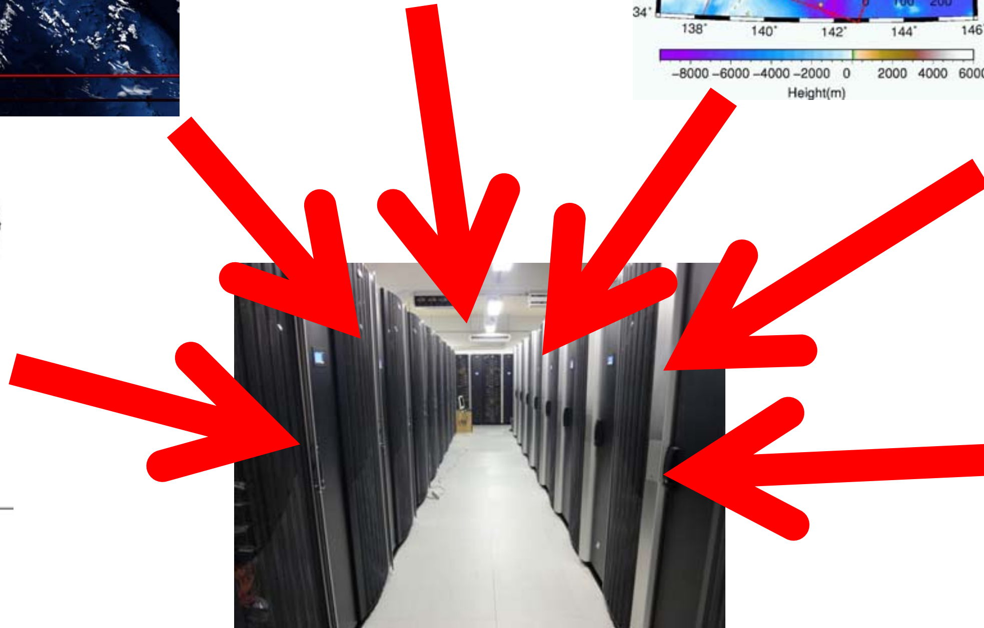












なぜスパコン・高性能計算を知ると良いか？

- 現代のコンピュータアーキテクチャの特徴が顕著に表れている
  - 今やケータイですら4コアCPU
  - Amazon・Google・SNSを裏側で支えているのはデータセンター(大規模クラスタが置いてある)

おことわり:

- 東工大講義「実践的並列コンピューティング」の、90分×14回分の内容＋αを、三時間でやるので色々飛ばします
- 今日の内容はあまり最前線ではないかも
  - 主に取り上げるのは古典的なOpenMP, MPI
  - GPU上のCUDA, OpenACCあたりはちょっと新しい
  - 「XXX大学による最新のYYY言語機能を取り入れたZZZ言語」・・・のようなものはカバーしていない
- いいわけ
  - たった今、京やTSUBAMEで走っているプログラムの多くはFortran+MPI
  - Ruby/Pythonでデータ解析をやっている人も
  - 多くの人は、美しい言語より、20年後まだ生きている道具に乗っかりたい

⇒その分、HPC+PROにより新たな研究チャンスも・・・？

スパコン開発競争の激化ー世界一スパコンの変遷

Linpack演算速度 (Gflops)

Jun-93	CM-5/1024	59.7	LANL	US
Nov-93	Numerical Wind Tunnel	1[illegible]	[illegible]	[illegible]apan
Jun-94	XP/S140	14[illegible]	[illegible]	[illegible]S
Nov-94	Numerical Wind Tunnel	170	NA[illegible]	[illegible]apan
Jun-95	Numerical Wind Tunnel	170	NAL	[illegible]pan
Nov-95	Numerical Wind Tunnel	170	NAL	[illegible]
Jun-96	SR2201/1024	[illegible]	[illegible]	[illegible]
Nov-96	CP-PACS/2048	[illegible]	[illegible]	[illegible]
Jun-97	ASCI Red	[illegible]	[illegible]	[illegible]
Nov-97	ASCI Red	[illegible]	[illegible]	[illegible]
Jun-98	ASCI Red	[illegible]	[illegible]	[illegible]
Nov-98	ASCI Red	[illegible]	[illegible]	[illegible]
Jun-99	ASCI Red	[illegible]	[illegible]	[illegible]
Nov-99	ASCI Red	2379	SNL	US
Jun-00	ASCI Red	2379	SNL	US
Nov-00	ASCI White	4938	LLNL	US
Jun-01	ASCI White	7226	LLNL	US
Nov-01	ASCI White	7226	LLNL	US
Jun-02	Earth-Simulator	35860	ES Center	Japan
Nov-02	Earth-Simulator	35860	ES Center	Japan

Jun-03	Earth-Simulator	35860	ES Center	Japan
Nov-03	Earth-Simulator	35860	ES Center	Japan
Jun-04	Earth-Simulator	35860	ES Center	Japan
Nov-04	BlueGene/L beta	70720	IBM/DOE	US
Jun-05	BlueGene/L	136800	DOE/NNSA/LLNL	US
Nov-05	BlueGene/L	280600	DOE/NNSA/LLNL	US
[illegible]	[illegible]	[illegible]	DOE/NNSA/LLNL	US
[illegible]	[illegible]	[illegible]	DOE/NNSA/LLNL	US
[illegible]	[illegible]	[illegible]	DOE/NNSA/LLNL	US
[illegible]	[illegible]	[illegible]	DOE/NNSA/LLNL	US
[illegible]	[illegible]	[illegible]	DOE/NNSA/LANL	US
[illegible]	[illegible]	[illegible]	DOE/NNSA/LANL	US
[illegible]	[illegible]	[illegible]	DOE/NNSA/LANL	US
No[illegible]	[illegible]r	1759000	ORNL	US
Jun-1[illegible]	[illegible]	1759000	ORNL	US
Nov-10	T[illegible]	2566000	NSC in Tianjin	China
Jun-11	K c[illegible]	8162000	RIKEN AICS	Japan
Nov-11	K [illegible]	[illegible]	[illegible]KEN AICS	Japan
Jun-12	Sequoia	[illegible]	[illegible]E/NNSA/LLNL	US
Nov-12	Titan	17590000	DOE/SC/ORNL	US
Jun-13	Tianhe-2	33862700	NUDT	China

60GFlops

33.9PFlops

20年間で性能560,000倍

アメリカと日本の一騎打ち

⇒中国の台頭

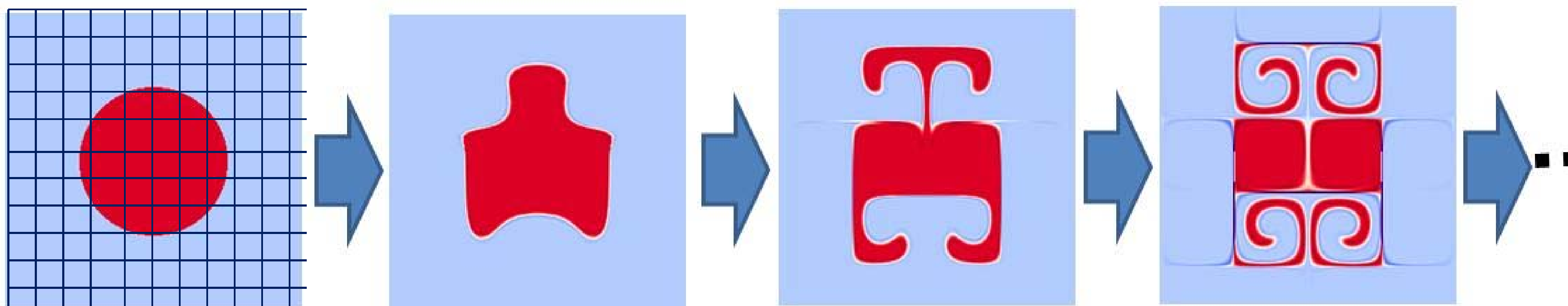
参考: www.top500.org

なぜ計算速度が重要?

- 仮想実験(シミュレーション)のためには，ばく大な量の計算を，タイムリーにこなさなければならないから

⇒明日の天気の計算に，一年かかっては意味がない！

コップにクリープを垂らしてかきまぜた時の例

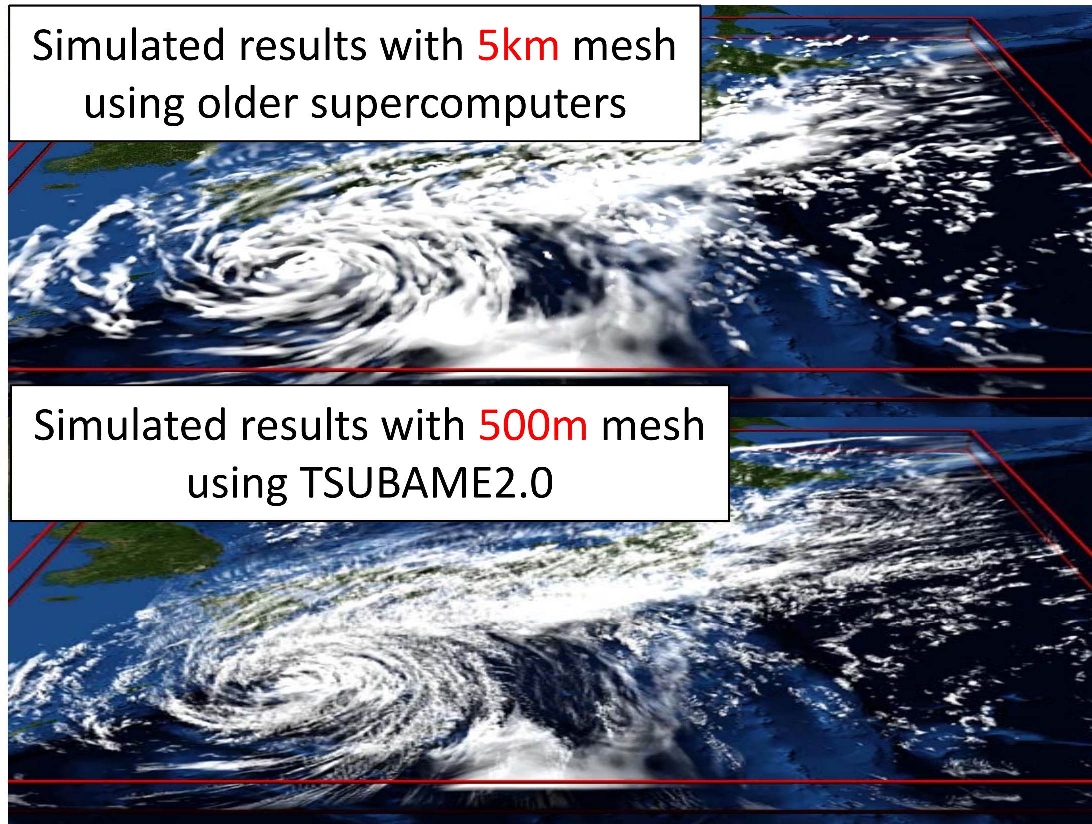


空間を，細かいマス目に分割　⇒１点ずつ計算が必要
ある瞬間の計算が終了したら，次の瞬間の計算へ

- 因果律：過去⇒現在⇒未来⇒もっと未来
- パラパラ漫画のように計算を続ける

なぜ計算速度が向上し続ける必要？

- より厳密なシミュレーションを行うには、細かい解像度が必要



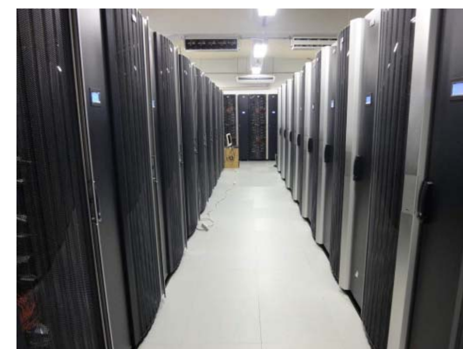
より細かい雲の
挙動が把握可能に！

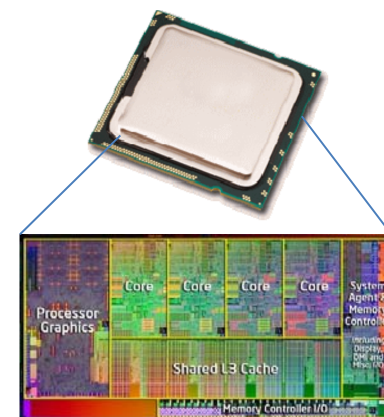
10倍の高解像度のためには
X方向10倍×Y方向10倍×Z方向10倍×時間方向10倍=10000倍
の計算をこなす必要！

一般的なスパコンの構造

階層構造が鍵

- システム = 多数の計算ノード + 外部ストレージ
  - パーツ間はネットワークで接続
- 計算ノード = 1以上のプロセッサ + メモリ + ローカルストレージ
  - パーツ間はPCI-e, QPIなどの通信路で接続
- プロセッサ = 1以上のコア + L3キャッシュ + その他
- コア = 複数の演算器 + レジスタ + L1/L2キャッシュ + その他





スパコンの計算性能は何で決まる？

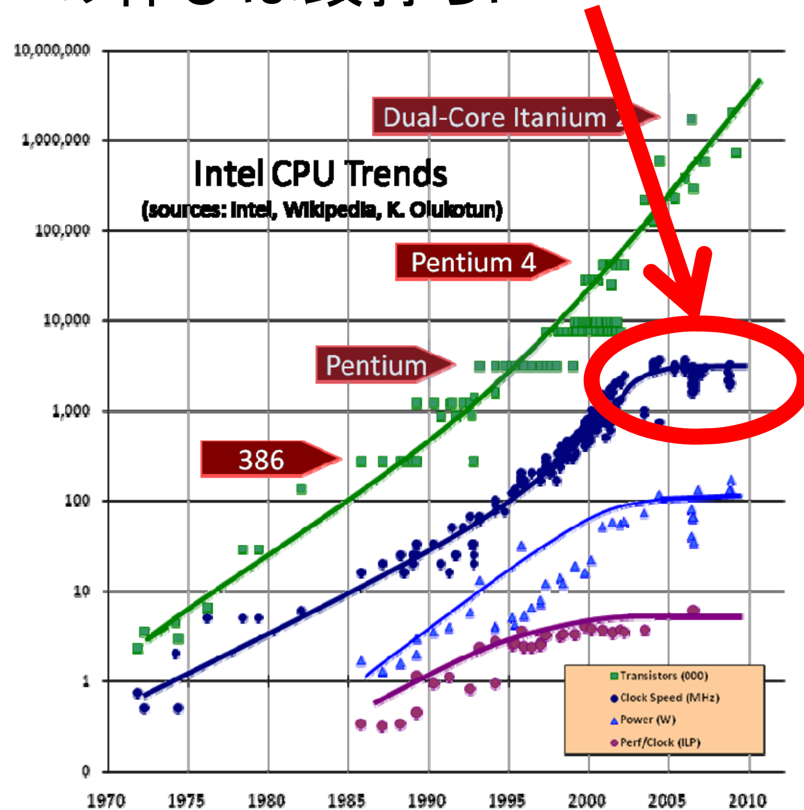
理論ピーク演算性能:

システムが仮に浮動小数演算のみを続けたときのFlops値。
実効演算性能とは区別が必要。以下の積となる:

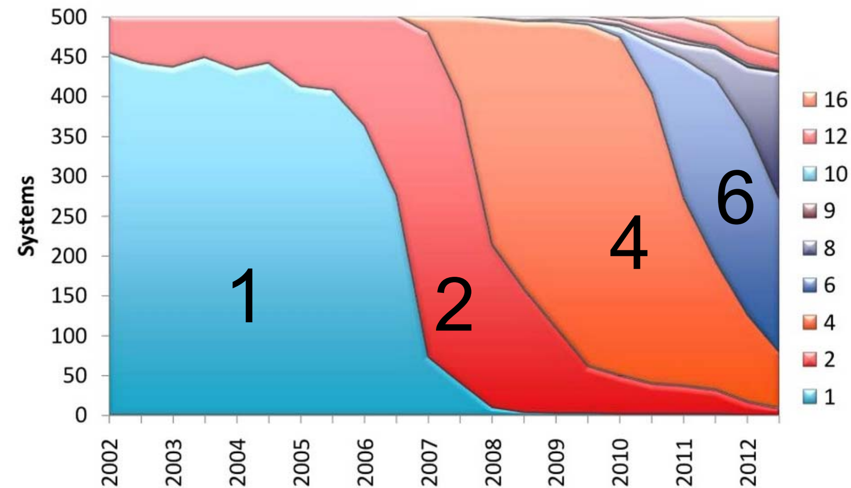
- クロック周波数(Hz=1/sec)
  - 1~3GHz程度。2003ごろより頭打ち
- 1クロックあたりの同時計算数(flop)
  - 倍精度:TSUBAME2のCPUでは4, Sandy Bridge世代(AVX)で8
  - 単精度や整数はその2倍のことが多い
- プロセッサあたりのコア数
  - 4〜16程度, これからも伸びる見込み
- 計算ノードあたりのプロセッサ数
  - 1〜4程度
- 計算ノード数
  - TSUBAME2では1400, 京コンピュータでは88000

近年のスパコンの動向

2003年ごろからクロック周波数の伸びは頭打ちに



プロセッサあたりのコア数を増やして性能を稼ぐ方向に



90年代までの常識:「5年待てば私のプログラムは速くなる」
今の常識:「並列プログラミング覚えないと!」

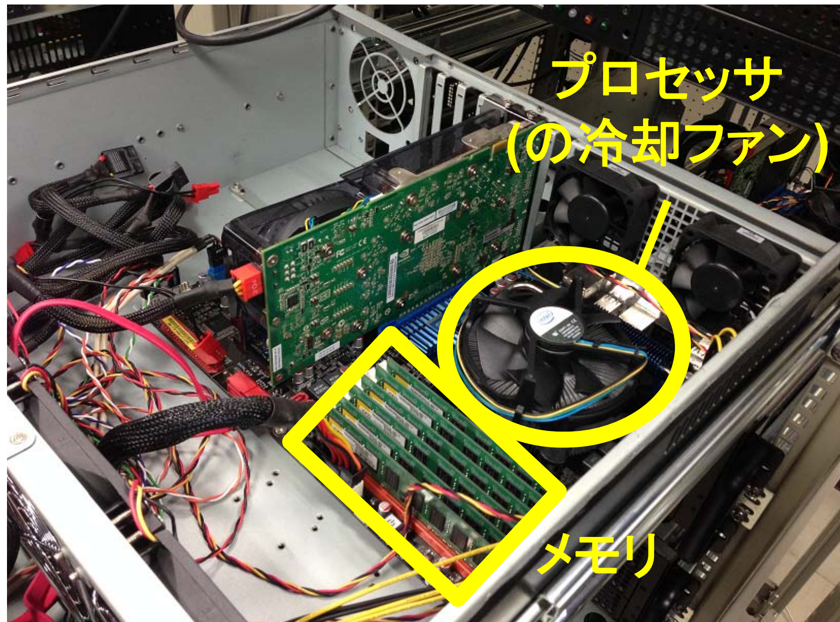
並列プログラミングの主な道具

アーキテクチャ階層に応じて使い分ける

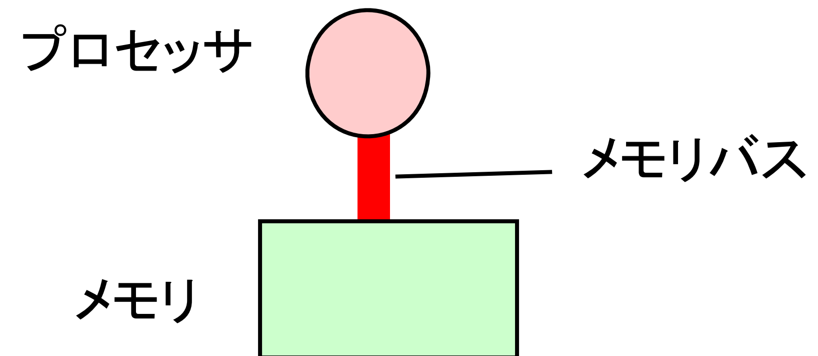
- ノード間並列
  - MPIなど
- ノード内・コア間並列
  - OpenMPなど
- コア内並列
  - AVX/SSEなど

複数の計算機要素をどうやって扱うか、メモリ構造の前提、などが異なる

単純化した
コンピュータアーキテクチャ



単純化したアーキテクチャ

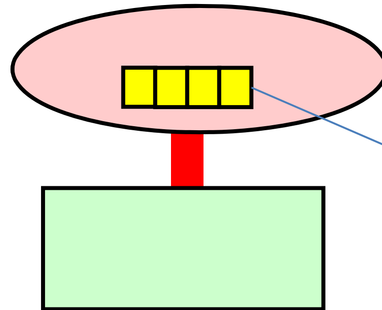


並列アーキテクチャの分類

SIMD (Single Instruction Multiple Data)

プロセッサ
(コア)

メモリ



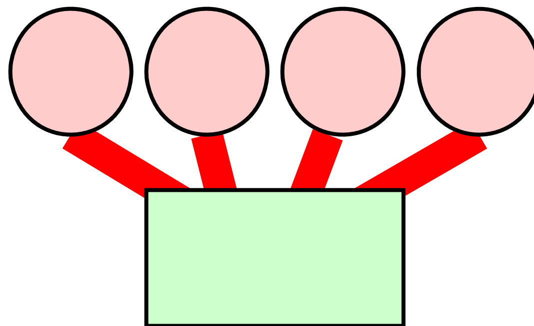
演算器

MIMD (Multiple Instruction Multiple Data)

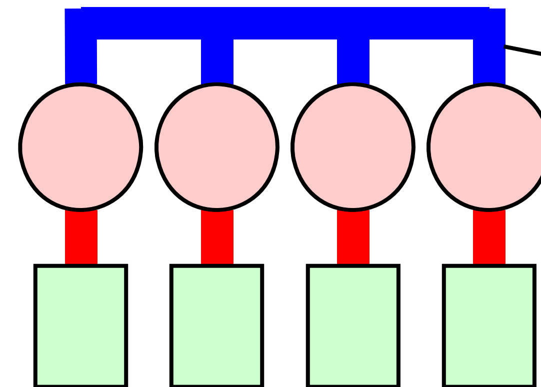
共有メモリ並列アーキテクチャ

プロセッサ
(コア)

メモリ



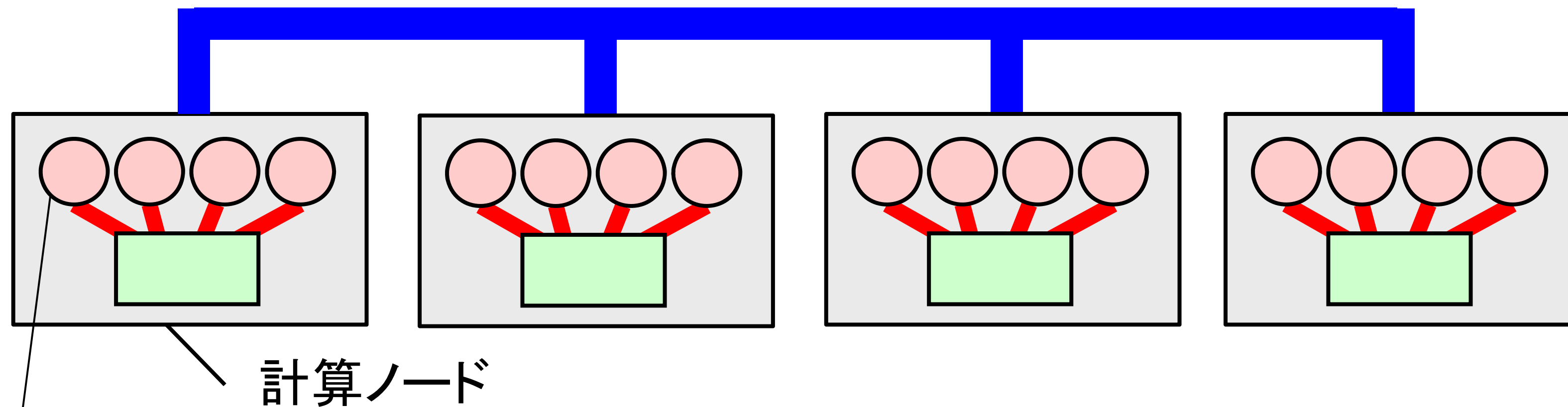
分散メモリ並列アーキテクチャ



ネット
ワーク

近年の並列アーキテクチャ

近年のスパコンのほとんど全てはSIMD・<u>共有メモリ・分散メモリの組み合わせ</u>
計算ノード内は共有メモリ，計算ノード間は分散メモリ

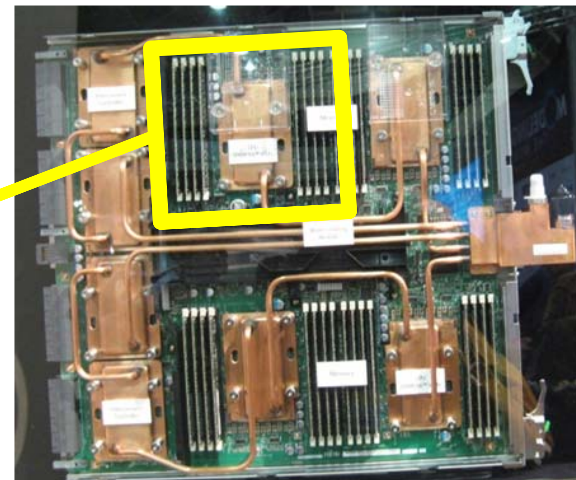


計算ノード

複数演算器を持つコア

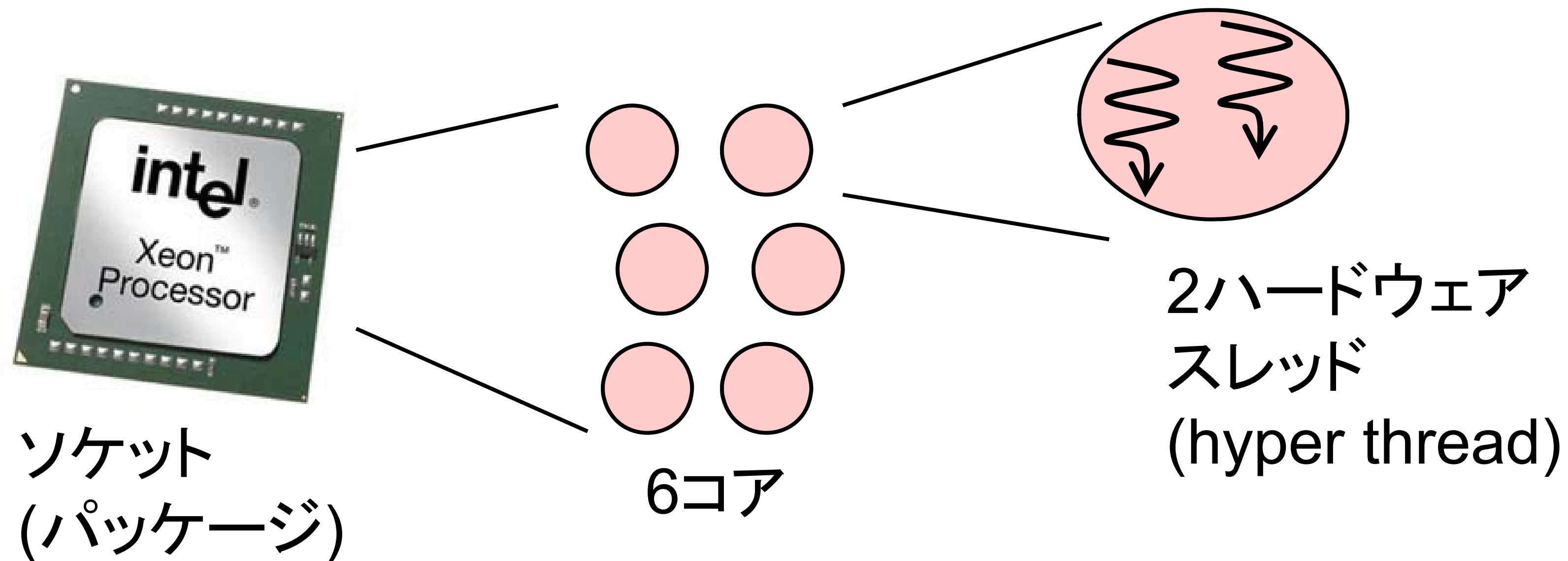
TSUBAME2
の計算ノード

京の
計算ノード



用語:ソケットやコア

- マルチコア時代になり，プロセッサの定義が複雑に
  - プロセッサの機能を持つ「コア/CPUコア」を複数，パッケージに詰め込むようになった
  - HyperThreadingによりさらに複雑に．1コアを2つのハードウェアスレッドが共有する．OSからはハードウェアスレッドがプロセッサに見える



京とTSUBAME2スパコンの理論性能

	京	TSUBAME2.5 (CPU部)
クロック周波数	2.0GHz	2.93GHz
コア性能	×8演算 = 16GFlops	×4演算 = 11.7GFlops
プロセッサ (ソケット)性能	×8コア = 128GFlops	×6コア = 70.3GFlops
ノード性能	×1ソケット = 128GFlops	×2ソケット = 140.6GFlops
システム性能	×88000ノード = 11.3PFlops	×1408ノード = 0.2PFlops

GPUをあわせると 5.7PFlops

なぜアーキテクチャはフラットな並列ではなく階層構造？

- 全てSIMD並列だったら？
  - 1つの命令列で全データを扱う→ 分岐がろくに書けず役に立たない
- 全て共有メモリ並列だったら？
  - 全プロセッサがアクセス可能なメモリ構造を作るのが、非効率的・高価に
  - NUMA(non-uniform memory access)でましにはなるが、それでもまだきつい
- 全て分散メモリ並列だったら？
  - 実現可能だが、まったく共有メモリが無いとプログラムしづらい傾向

アクセラレータへの注目

- プロセッサは一種類でよいのか？
- 汎用のプロセッサだけでなく、並列処理に強い(が汎用処理に弱め)なプロセッサも併用するアプローチ ⇒アクセラレータ

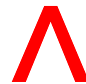
- 代表的アクセラレータ
  - NVIDIA社製GPU
  - Intel社製Xeon Phi
  - AMD社製GPU

アクセラレータ

- 「周波数はあまり高くせず、並列度で稼ぐ」という方針をさらに推し進めたのが、アクセラレータ
- GPU(graphic processing unit)はもともと画像出力用専用プロセッサだが、演算へ転用(GPGPU)

Intel Xeon X5670

2.93 GHz × 4 Flop × 6 core = 70.4GFlops



NVIDIA Tesla K20X

0.73 GHz × 128 Flop × 14 SM = 1310GFlops

- TSUBAME2.5, Titan(2012 No.1), Tianhe-1A(2010 No.1)などが採用
- NVIDIA GPU, AMD/ATI GPU, Intel Xeon Phiなど

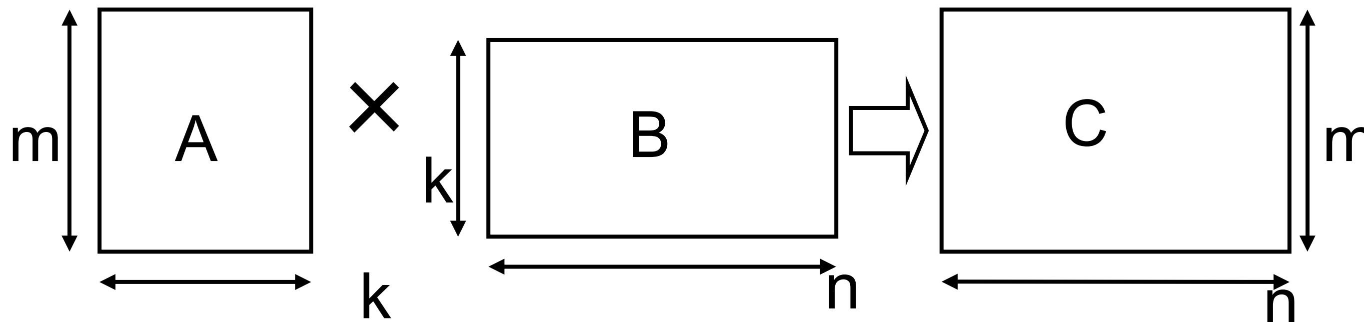
例題プログラム

例題:密行列×密行列の積 (Matrix multiply, Matmul)

(m×k)行列と(k×n)行列の積

- 三重のforループで記述
- 計算量:O(mnk)

```
for (j = 0; j < n; j++) {
  for (l = 0; l < k; l++) {
    double blj = B[l+j*ldb];
    for (i = 0; i < m; i++) {
      double ail = A[i+l*lda];
      C[i+j*ldc] += ail*blj;
    } }  }
```



注意：行列積を自前でプログラミングする機会はほぼない

- 既存のライブラリであるMKLやGotoBLASを用いたほうがはるかに速い

TSUBAME2 CPU上で様々な行列積実装を実行した結果

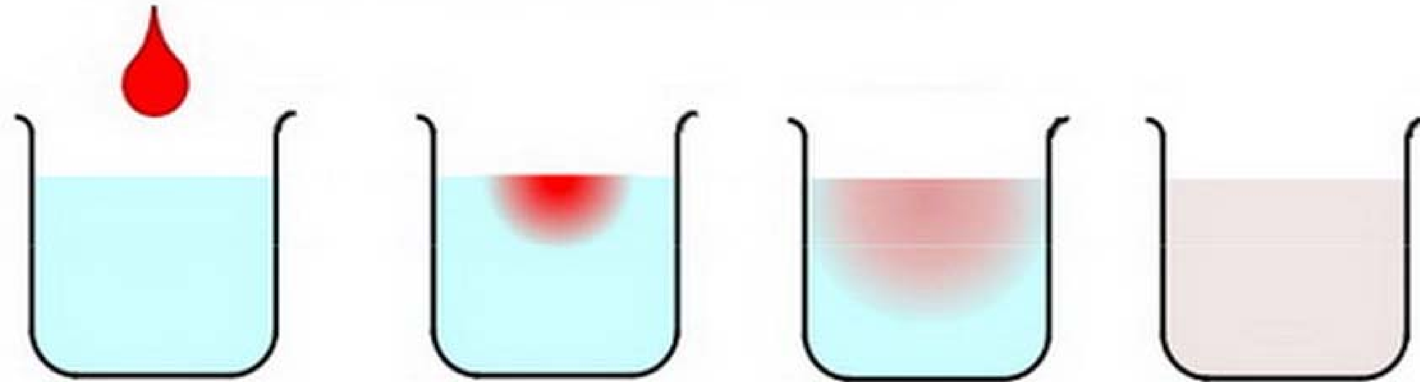
12コア

理想的に演算器を使えれば 4 x 2.93GHz x 12 = 140.6GFlops

実装	並列化なし	SIMD	OpenMP	SIMD+ OpenMP	GotoBLAS
Speed (Gflops)	1.92	3.71	20.3	26.7	119

例題：流体拡散シミュレーション (diffusion)

拡散現象

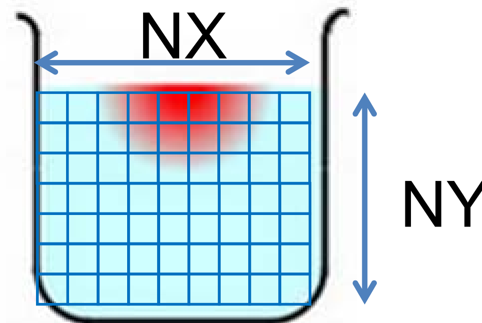


© 青木尊之

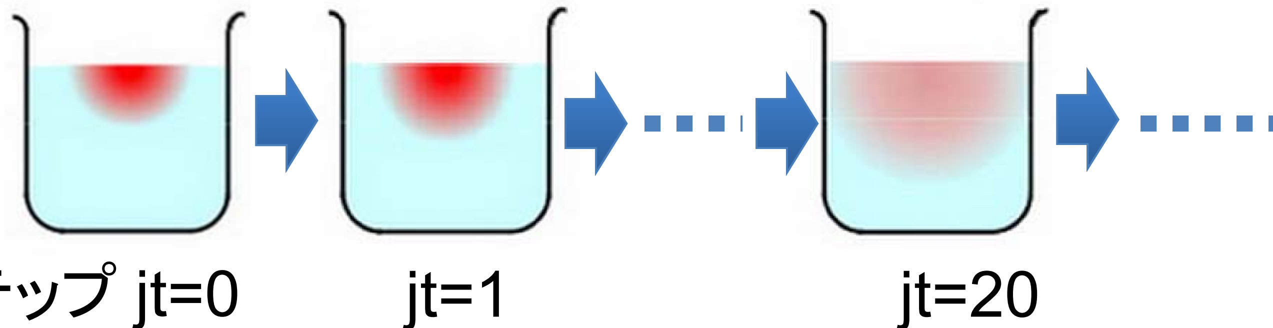
- 各点のインク濃度は、時間がたつと変わっていく→ その様子を計算機で計算
  - 天気予報などにも含まれる計算

diffusionのデータ構造

- シミュレーションしたい空間をマス目で区切り、配列で表す(本プログラムでは二次元配列)



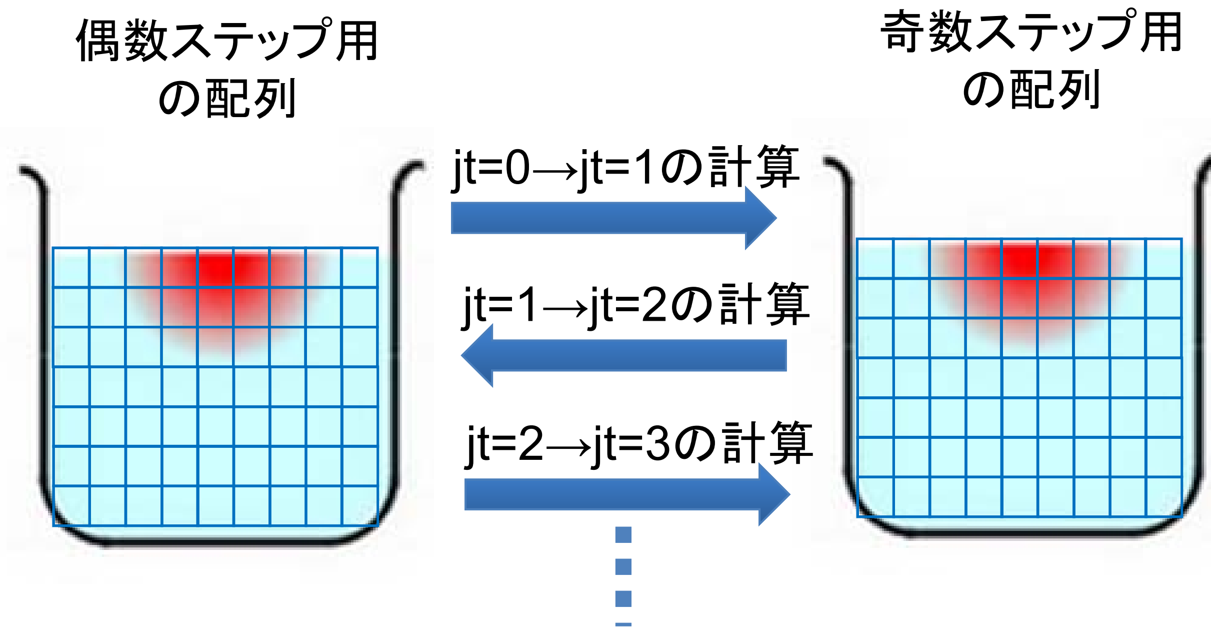
- 時間を少しずつ、パラパラ漫画のように進めながら計算する



ダブルバッファリング技術

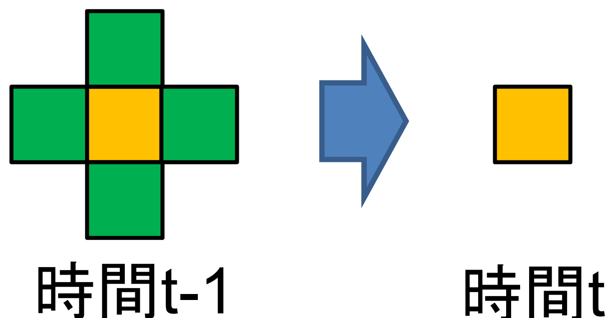
- 全時間ステップの配列を覚えておくとメモリ容量を食い過ぎる

→ 二ステップ分だけ覚えておき、二つの配列(ダブルバッファ)を使いまわす



diffusionの計算: ステンシル計算

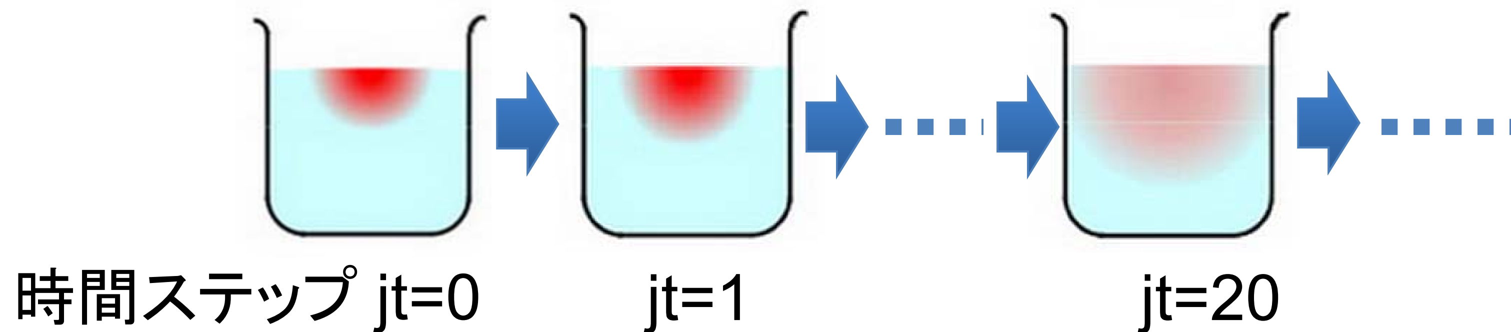
- 時間tにおける点(i,j)を計算するには？
- 時間t-1における下記を利用
  - 点(i,j)の値
  - 点(i,j)の近傍の値 (このサンプルでは上下左右)



本来の「ステンシル」

- このタイプの演算をステンシル計算と呼ぶ
- 以下が既知とする
  - 時間0の全点の温度(初期条件)
  - 各時間における、領域の「端」の点の温度(境界条件)

Diffusionの実装例




```
for (jt = 0; jt < NT; jt++) {              // 時間ループ
  for (jy = 1; jy < NY-1; jy++) {          // 空間ループ(y)、境界を除く
    for (jx = 1; jx < NX-1; jx++) {        // 空間ループ(x) 、境界を除く
      FN[jx][jy] = 0.2 * (F[jx][jy] +
             F[jx-1][jy]+F[jx+1][jy]+F[jx][jy-1]+F[jx][jy+1];
    } }
  swap(&F, &FN);              // ダブルバッファを交換
}
```

SSE/AVXによるSIMDプログラミング

コア内並列性の利用: SIMDプログラミング

- SIMD = Single Instruction Multiple Data
  - Multiple operations can be done simultaneously
- CPUアーキテクチャに大きく依存
  - Intel CPUでは、世代により, MMX → SSE → AVX
    - TSUBAME2 nodes support SSE
  - 富士通SPARCプロセッサは、違うSIMD命令体系
- コンパイラが勝手に利用してくれる場合もあるが限定的
- 基本的にアセンブリか、intrinsicsで書く
  - gcc and Intel compilers supports special methods called “intrinsics”
  - _mm_load_pd, _mm_mul_pd, _mm_add_pd…

Basics of SSE

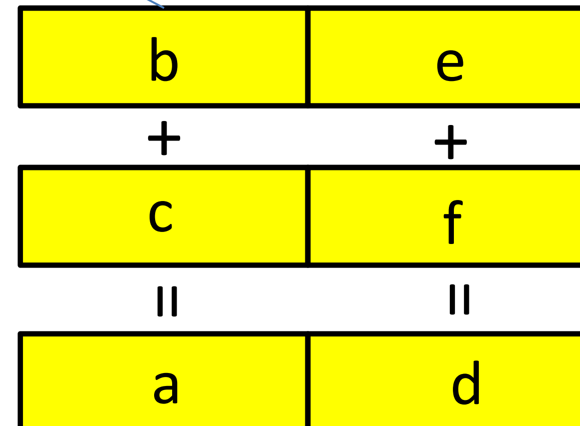
With normal operations

With SSE operations

```
ad = _mm_add_pd(be, cf);
```

- In SSE, 128 bit (16byte) packed type is used
  - __m128d value can contain 2 double values
  - __m128 value can contain 4 single values
- In AVX, 256 bit packed type is used

__m128d type



SSE Operations

Use gcc or Intel compiler

- __m128d a = _mm_load_pd(p);
  - Makes _m128d value that contains p[0], p[1]
  - Hereafter, a0, a1 mean contents of a
  - pd means “packed double”
- __m128d c = _mm_add_pd(a, b);
  - c0 = a0+b0; c1 = a1+b1;
- __m128d c = _mm_mul_pd(a, b);
  - c0 = a0*b0; c1 = a1*b1; (not dot product)
- __mm_store_pd(p, a);
  - p[0] = a0; p[1] = a1;
- Also there are “packed single” version
  - Such as __m128 a = _mm_load_ps(p);

SSEを使った行列積

With normal operations

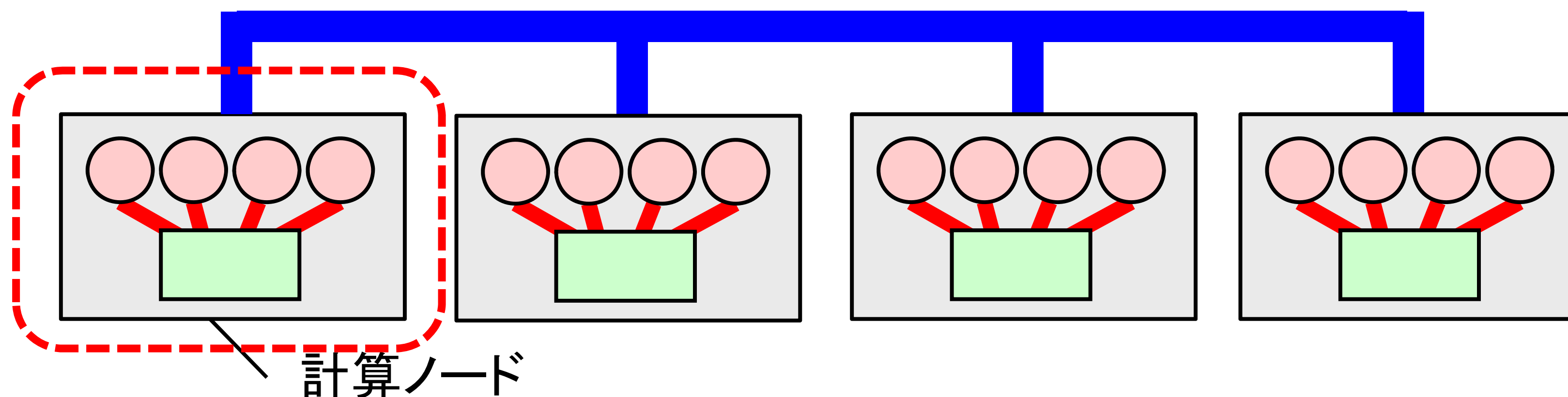
```
for (j = 0; j < n; j++) {
  for (l = 0; l < k; l++) {
    double blj = B[l+j*ldb];
    for (i = 0; i < m; i++) {
      double ail = A[i+l*lda];
      C[i+j*ldc] += ail*blj;
    } }  }
```

With SSE operations

```
#include <emmintrin.h>
#include <xmmintrin.h>
        :
for (j = 0; j < n; j++) {
    for (l = 0; l < k; l++) {
        __m128d bv = _mm_load_pd1(&B[l+j*ldb]);
        double *ap = &A[l*lda];
        double *cp = &C[j*ldc];
        for (i = 0; i < m; i += 2) {
          __m128d av = _mm_load_pd(ap);
          __m128d cv = _mm_load_pd(cp);
          av = _mm_mul_pd(av, bv);
          cv = _mm_add_pd(cv, av);
          _mm_store_pd(cp, cv);
          ap += 2;
          cp += 2;
        }}}
```

OpenMPによる
共有メモリ並列プログラミング

OpenMPの利用可能な計算資源



- 一つのOpenMPプログラムが使えるのは一計算ノード中のCPUコアたち
- 複数計算ノードを(一プログラムから)用いたい場合は、MPIなどが必要
  - ただしMPIよりOpenMPのほうがとっつきやすい

OpenMPとは

- 共有メモリモデルによる並列プログラミングAPI
- C言語，C++, Fortranに対応
- 並列化のための指示文や，ライブラリ関数
  - 指示文：#pragma omp ~~
- 基本はFork-Joinモデル
- 変数は基本的にスレッド間で共有

  ⇒以下を明示的に記述
  - タスク分割
  - スレッド間同期
  - 変数の共有・プライベートの区別

OpenMPプログラムのコンパイル

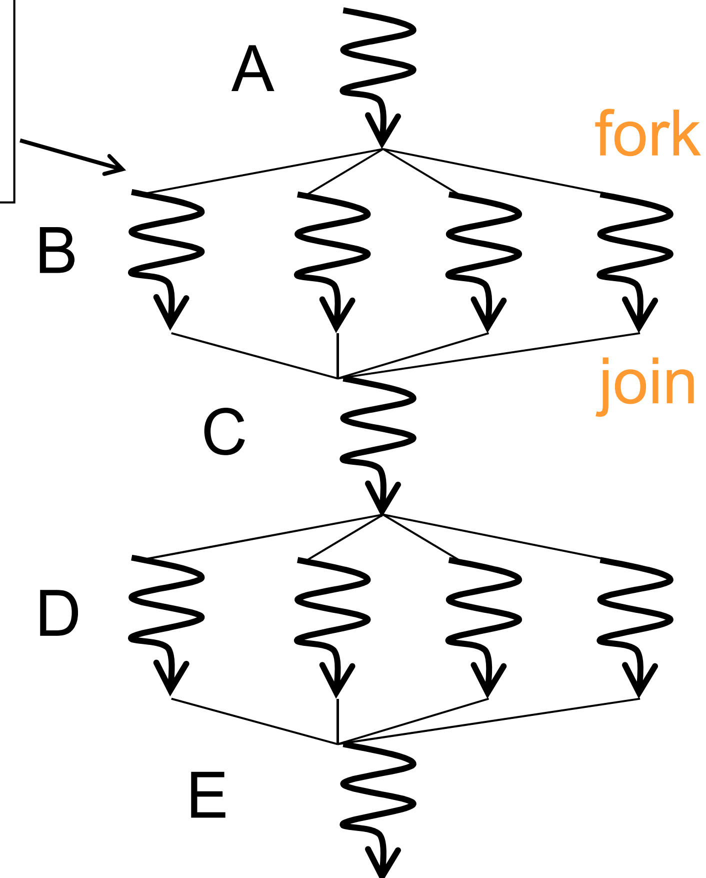
OpenMP対応コンパイラは近年増加

- PGIコンパイラ (pgcc)
  - コンパイル時・リンク時に-mpオプション
- Intelコンパイラ (icc)
  - コンパイル時・リンク時に-openmpオプション
- GCC 4.2以降
  - コンパイル時・リンク時に-fopenmpオプション

OpenMP並列実行の基本：
並列Region

```
#include <omp.h>

int main()
{
    A;
#pragma omp parallel
    {
          B;
    }
    C;
#pragma omp parallel
    D;
    E;
}
```

ここから
4threadsで
並列実行



#pragma omp parallelの直後の文・ブロックは並列Regionとなる
並列Regionから呼ばれる関数も並列実行

スレッド数の設定・取得

- スレッド数の設定
  - 実行時に、OMP_NUM_THREADS環境変数に指定しておく
- 全スレッド数の取得
  - omp_get_num_threads()関数

  「全体で何人いるか？」
- 自スレッドの番号の取得
  - omp_get_thread_num()関数
    - 0以上、「全スレッド数」未満

  ⇒番号によって違う処理をさせることができる

OpenMPの指示文

以下は並列region内で使われる

- #pragma omp critical
  - 次のブロック・文が「critical section」となる
  - 同時にcritical sectionを実行できるのみは1スレッドのみ、となる
- #pragma omp barrier
  - スレッド間でバリア同期をとる：全スレッドの進行がそろうまで待つ
  - ただし並列regionの終わりでは，自動的に全スレッドを待つ(暗黙のbarrier)
- #pragma omp single
  - 次のブロック・文を1スレッドのみで実行する
- #pragma omp for (後述)

OpenMPのワークシェアリング構文: for

単なる”omp parallel”よりも、気軽に並列化の記述可能！

```
{
   int s = 0;
#pragma omp parallel
   {
      int i;
#pragma omp for
      for (i = 0; i < 100; i++) {
         a[i] = b[i]+c[i];
      }
   }
}
```

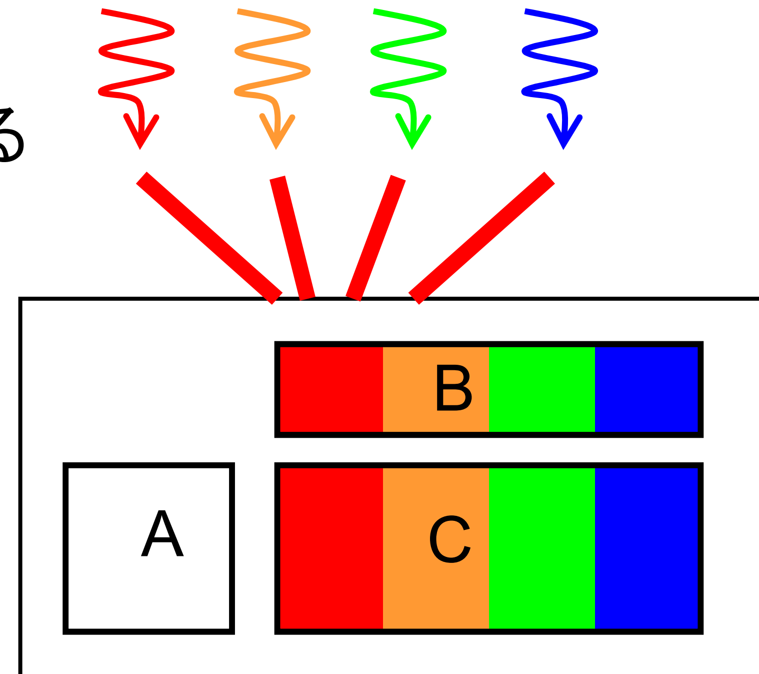
- “omp for”の直後のfor文は，複数スレッドにより並列実行される
- 左のプログラムが、もし4スレッドで実行されるならスレッドあたり25ずつ仕事
- ループ回数÷スレッド数が割り切れなくてもok

- omp parallelとomp forをまとめてomp parallel forとも書ける
- 残念ながら、どんなforでも対応できるわけではない。詳細は次回以降

行列積のOpenMPによる並列化

- 三重ループの最外ループを並列化
  - #pragma omp parallel for
  - nをスレッド間で分割することになる

```
#pragma omp parallel for
 for (int j = 0; j < n; j++) {
   for (int l = 0; l < k; l++) {
      double blj = B[l+j*ldb];
      for (int i = 0; i < m; i++) {
        double ail = A[i+l*lda];
        C[i+j*ldc] += ail*blj;
      } }  }
```

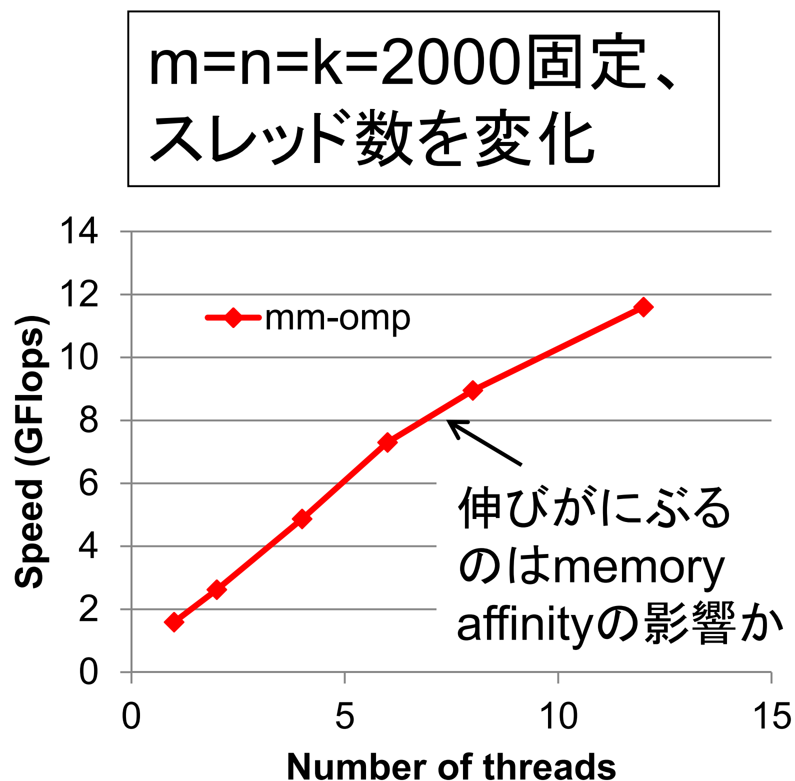


行列Aは全スレッドによってアクセスされる

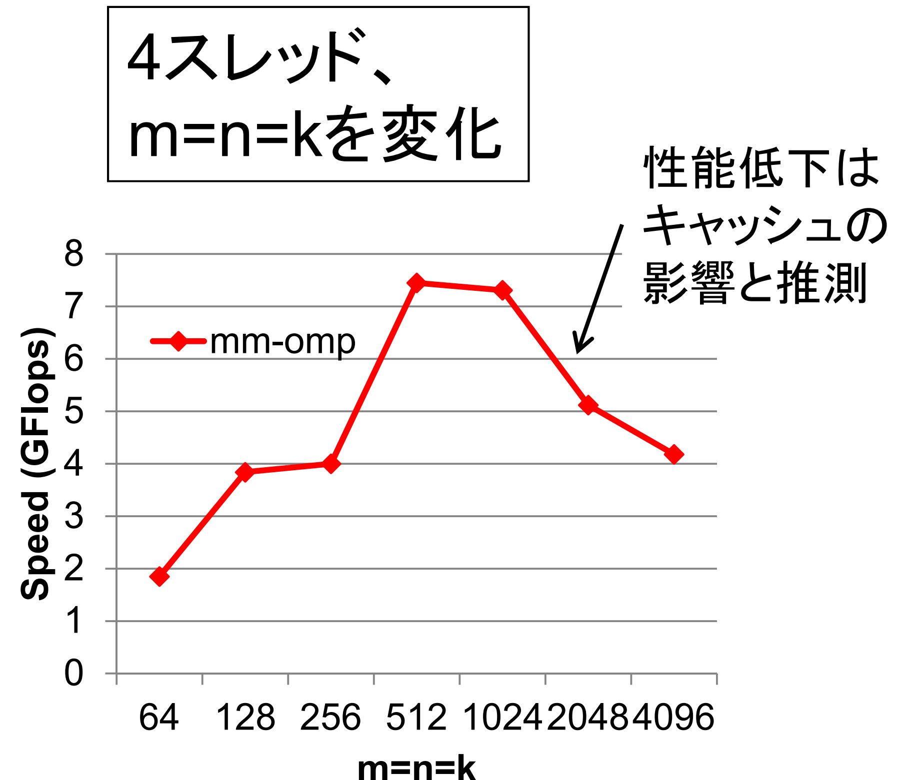
OpenMP版行列積の性能

- TSUBAME2ノード上(Xeon X5670 2.93GHz 12core)
- OMP_NUM_THREADS環境変数によりスレッド数指定
- (2mnk/経過時間)にてFlops単位の速度を取得

m=n=k=2000固定、スレッド数を変化

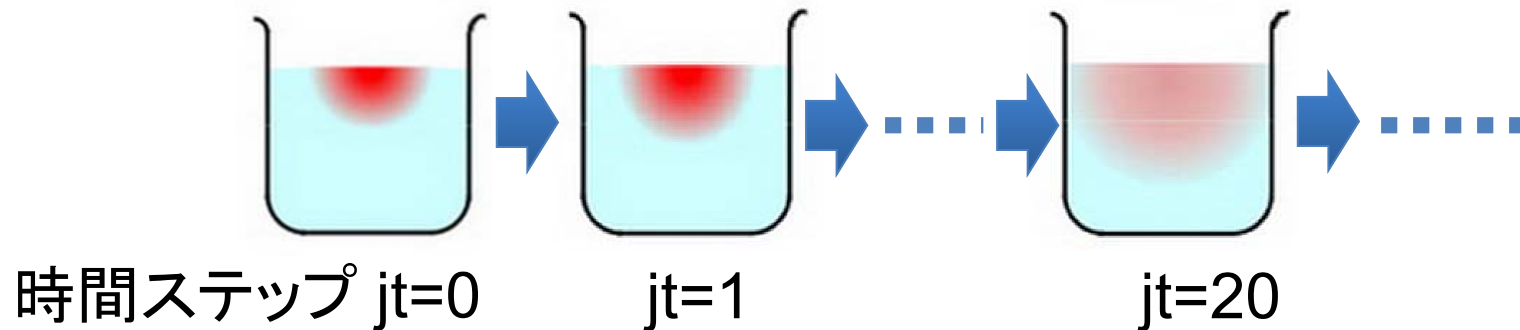


4スレッド、m=n=kを変化



45

Diffusionの並列化について



<u>これを並列化するには??</u>

- 空間ループをomp forで並列化が良い．結果的に空間を分割して，スレッドたちで分担することになる.
- 時間ループにomp forをつけてはいけない！なぜか？

OpenMP版Diffusion

```
for (jt = 0; jt < NT; jt++) {              // 時間ループ
#pragma omp parallel for
  for (jy = 1; jy < NY-1; jy++) {        // 空間ループ(y)、境界を除く
    for (jx = 1; jx < NX-1; jx++) {      // 空間ループ(x) 、境界を除く
        FN[jx][jy] = 0.2 * (F[jx][jy] +
              F[jx-1][jy]+F[jx+1][jy]+F[jx][jy-1]+F[jx][jy+1];
    } } // parallel forの効果はここまで
  swap(&F, &FN);            // ダブルバッファを交換
}
```

For指示文の補足情報: #pragma omp forが書ける条件

- 直後のfor文が「canonical form (正準形)」であること

```
#pragma omp for
  for (var = lb; var rel-op b; incr-expr)
    body
```

ここでincr-exprは ++var, --var, var++, var--, var+=c, var-=cなど

`for (i = 0; i < n; i++)` ⇒ For指示文可能！

`for (p = head; p != NULL; p = p->next)` ⇒ For指示文不可

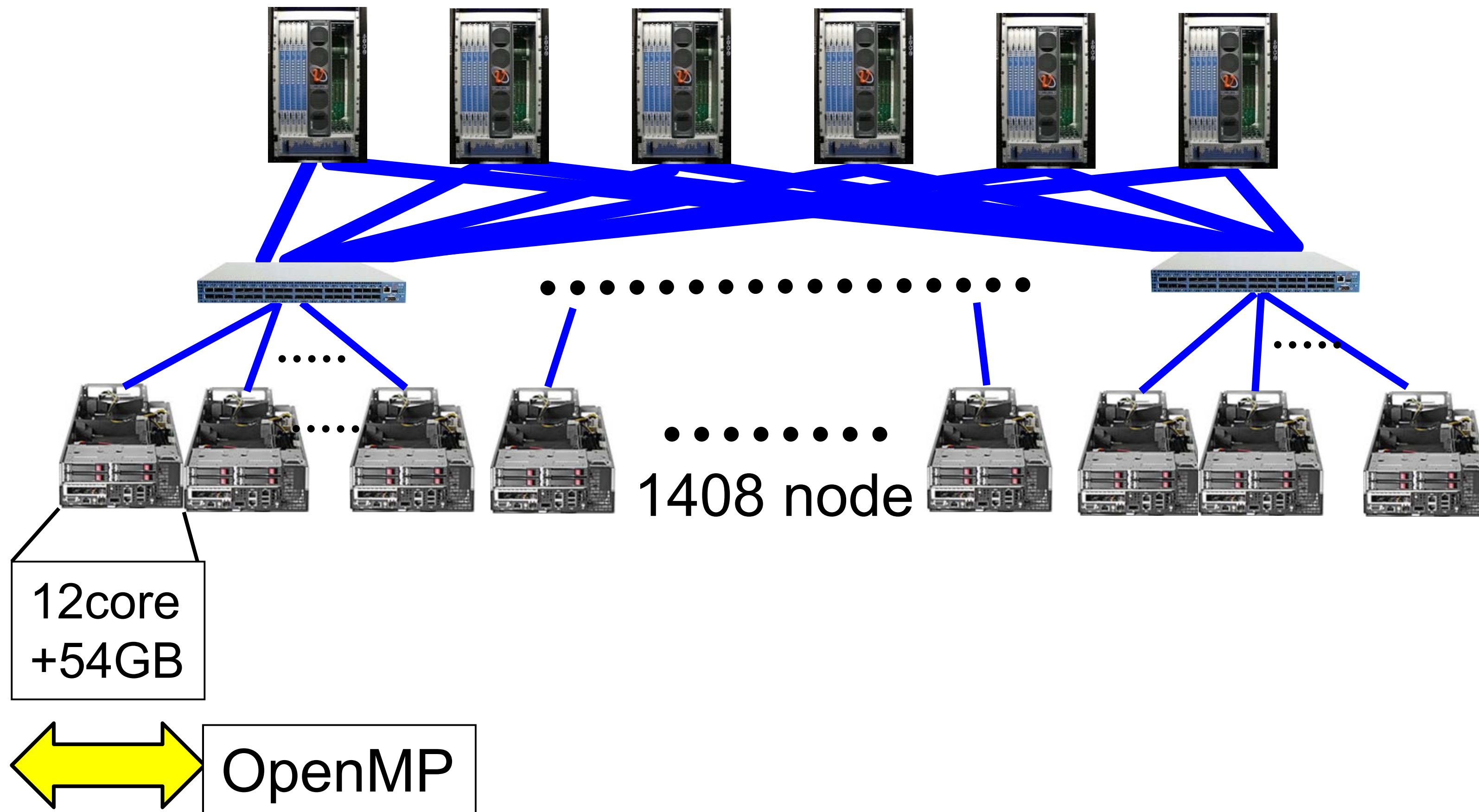
Canonical formであっても、プログラムの挙動の正しさは
やはりプログラマの責任

OpenMPのまとめ

- 逐次プログラムに「＋α」することで複数コアを用いることができる
- #pragma部分を読み飛ばせば、逐次プログラムに戻る
- 特にparallel forが強力。うまくはまれば一行追加で性能が数倍に
- 依存関係を壊さないか、Race conditionをおこさないかはユーザの責任

MPIによる
分散メモリ並列プログラミング

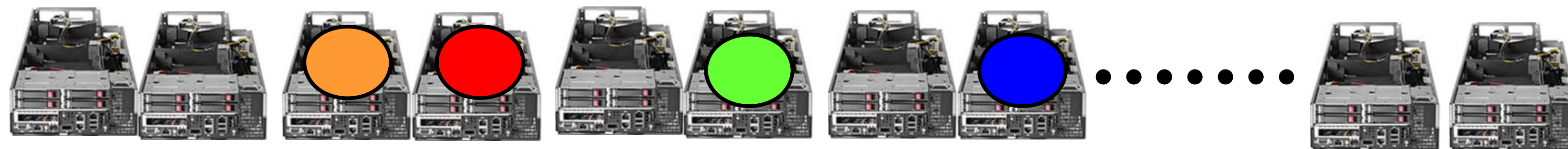
スパコンシステムの多数の計算ノードを活用するには？



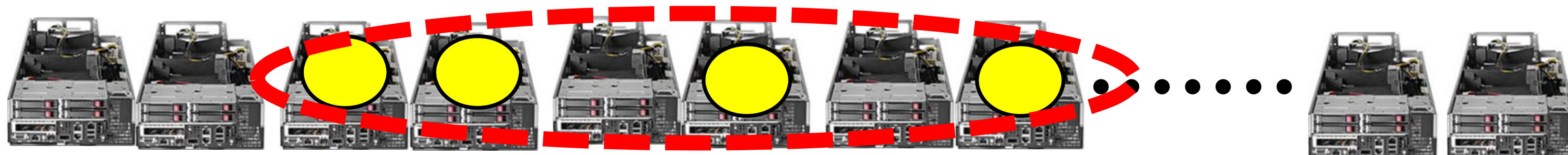
OpenMPは1ノードの中のCPUコアだけを使う

多数の計算ノードを活用するには？

1. (役割のそれぞれ違う)複数ジョブをバッチキューシステムに投入
   – パラメータをそれぞれ変えて投入することをパラメータスイープと呼ぶ
   – ジョブは独立に動き、原則的に協調しない



2. 一つのジョブが複数ノードを使いたい時には、分散メモリプログラミングを用いる
   - MPIやHadoop
   - Hadoopは、プロセス間の協調パターンが、Map-Reduceというパターンに限られる



MPI(message-passing interface)とは

- 分散メモリ並列プログラミングの規格
- C, C++, Fortranに対応
- メッセージパッシングのためのライブラリ
- SPMDモデル．プロセス間の相互作用はメッセージで
  - MPI-2規格では，さらにRMA(remote memory access)が追加

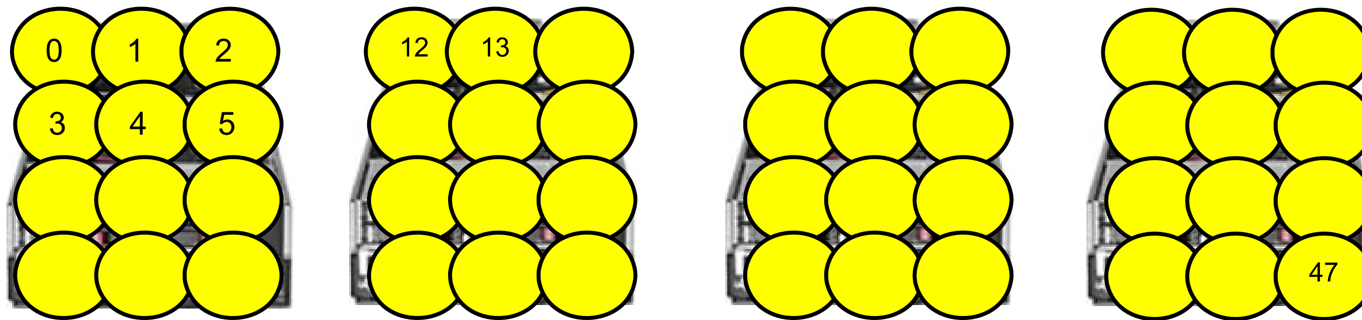
科学技術演算でメジャーなMPI

京スパコン上で稼働中のソフトウェア(一部)

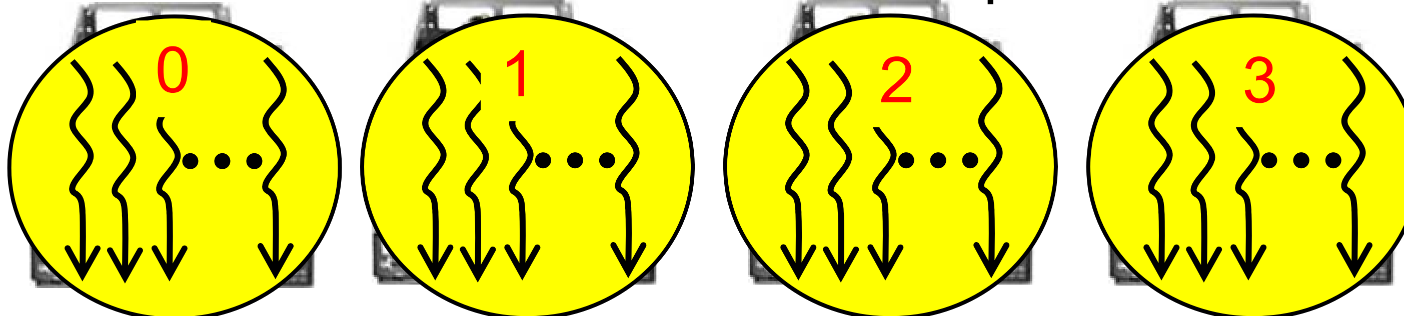
ソフトウェア名	説明	並列化方法
feram	強誘電体MD	OpenMP
STATE	第一原理MD	MPI+OpenMP
FFVC	差分非圧縮熱流体	MPI+OpenMP
GT5D	5次元プラズマ乱流	MPI+OpenMP
FrontFlow/blue	有限要素法非圧縮熱流体	MPI+京並列コンパイラ
OpenFMO	FMO第一原理計算	MPI+OpenMP
pSpatiocyte	細胞内シグナル伝播計算	MPI+OpenMP
NEURON_K+	神経回路シミュレーション	MPI+OpenMP
SiGN-L1	遺伝子ネットワーク推定	MPI+OpenMP
NTChem/RI-MP2	電子相関計算	MPI+OpenMP+並列BLAS
HPL	Linpackベンチマーク	MPI+並列BLAS

MPI+OpenMPとは？

- TSUBAMEでは1ノード12コア、京では1ノード8コアある。それを有効利用するには？
  - 1. MPIのみ使う。図では48プロセス起動



  - 2. MPI+OpenMP(ハイブリッド並列)。図では4プロセス起動し、それぞれが12スレッドのOpenMP並列



※ 8プロセス×3スレッドなどもあり
※ 1.より性能高い傾向にあるがプログラミング大変

OpenMPとMPI

- OpenMP
  - 共有メモリモデル
  - スレッド間のデータ移動は共有変数で
  - 排他制御によりrace conditionを防ぐ
  - 利用可能な並列度はノード内(TSUBAME2では12CPUコア)
  - #pragmaを無視すると逐次プログラムとして動作する場合が多い
- MPI
  - 分散メモリモデル
  - プロセス間のデータ移動はメッセージで
  - Critical sectionの代わりにメッセージで同期
  - 利用可能な並列度はノードを超える(TSUBAME2では10000CPUコア以上)
  - 逐次プログラムを基にする場合、全体の構造への大幅な変更が必要になりがち

MPIプロセスとメモリ

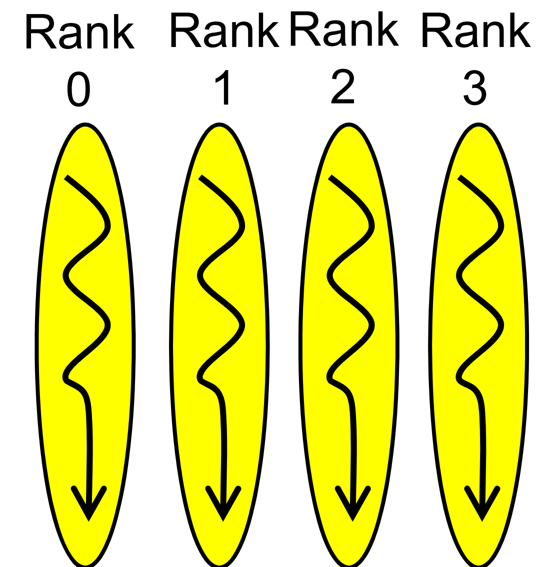
- 複数のプロセスが同一プログラムを実行(SPMDモデル)
- プロセスごとに別のメモリ空間 → 全ての変数(大域変数・局所変数)は各プロセスで別々
- プロセスには，0, 1, 2・・・という番号(rank)がつく
  - `MPI_Comm_rank(MPI_COMM_WORLD, &rank);` ランク取得
  - `MPI_Comm_size(MPI_COMM_WORLD, &size);` 全プロセス数取得
  - 0 ≦ rank < size
  - MPI_COMM_WORLDは，「全プロセスを含むプロセス集団(=コミュニケータ) 」
  - メッセージの送信先，受信元としてrankを利用

MPIプログラムの概要

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);

        (計算・通信)

    MPI_Finalize();
}
```

← MPIの初期化

← MPIの終了



MPIの基本中の基本：メッセージの送信・受信

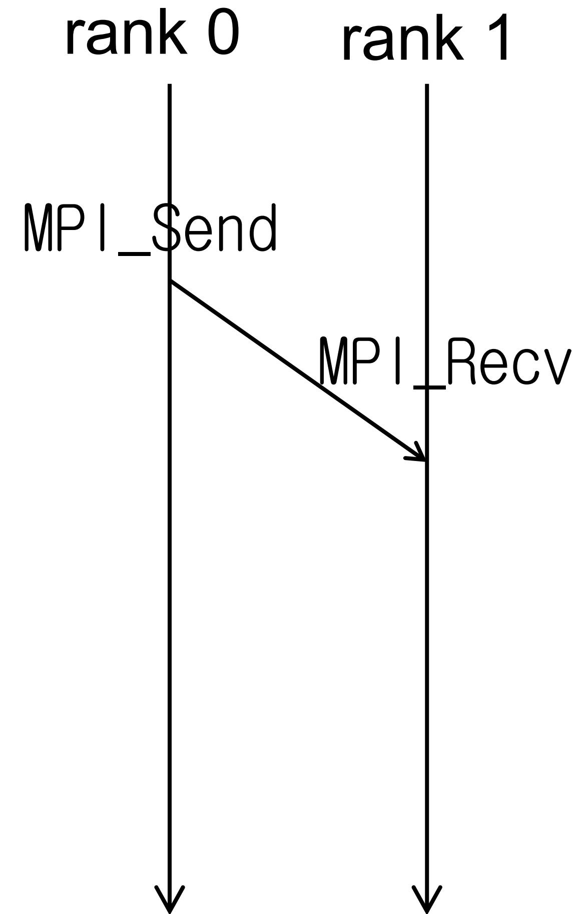
rank 0からrank1へ，int a[16]の中身を送りたい場合

- rank0側で

```
MPI_Send(a, 16, MPI_INT, 1,
  100, MPI_COMM_WORLD);
```

- rank1側で

```
MPI_Recv(b, 16, MPI_INT, 0,
  100, MPI_COMM_WORLD, &stat);
```



MPI_Send

```
MPI_Send(a, 16, MPI_INT, 1, 100, MPI_COMM_WORLD);
```

- a: メッセージとして送りたいメモリ領域の先頭アドレス
- 16: 送りたいデータ個数
- MPI_INT: 送りたいデータ型
  - 他にはMPI_CHAR, MPI_LONG. MPI_DOUBLE, MPI_BYTE・・・
- 1: メッセージの宛先プロセスのrank
- 100: メッセージにつけるタグ(整数)
- MPI_COMM_WORLD: コミュニケータ

MPI_Recv

```
MPI_Status stat;
MPI_Recv(b, 16, MPI_INT, 0, 100, MPI_COMM_WORLD, &stat);
```

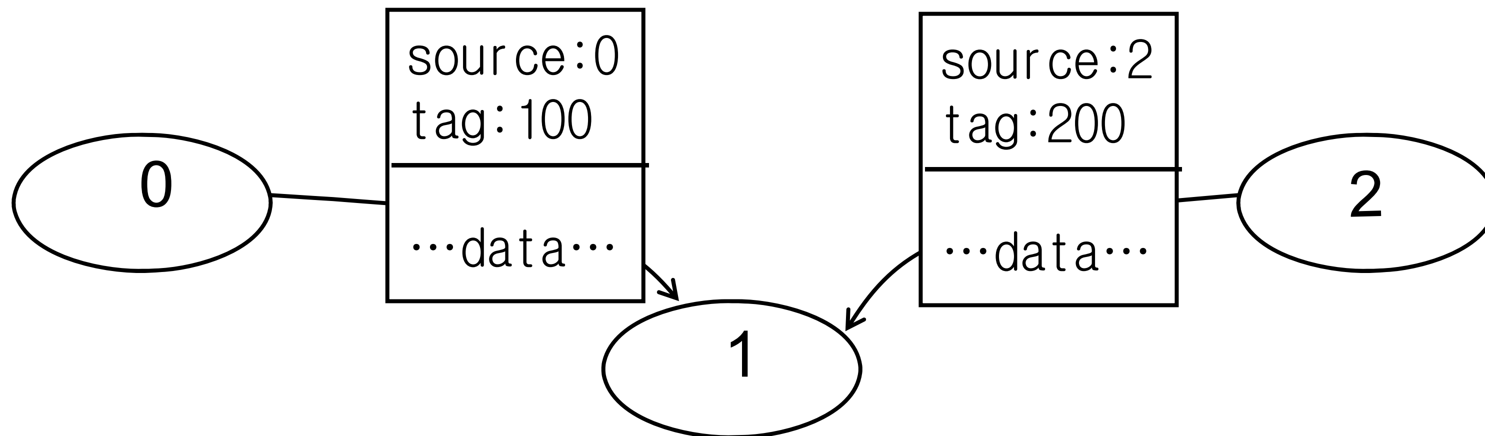
- b: メッセージを受け取るメモリ領域の先頭アドレス
  - 十分な領域を確保しておくこと
- 16: 受け取るデータ個数
- MPI_INT: 受け取るデータ型
- 0: 受け取りたいメッセージの送信元プロセスのrank
- 100: 受け取りたいメッセージのタグ．ユーザが決める整数
  - MPI_Sendで指定したものと同じなら受け取れる
- MPI_COMM_WORLD: コミュニケータ
- &stat: メッセージに関する補足情報が受け取れる

MPI_Recvを呼ぶと，メッセージが到着するまで待たされる (ブロッキング)

MPI_Recvのマッチング処理

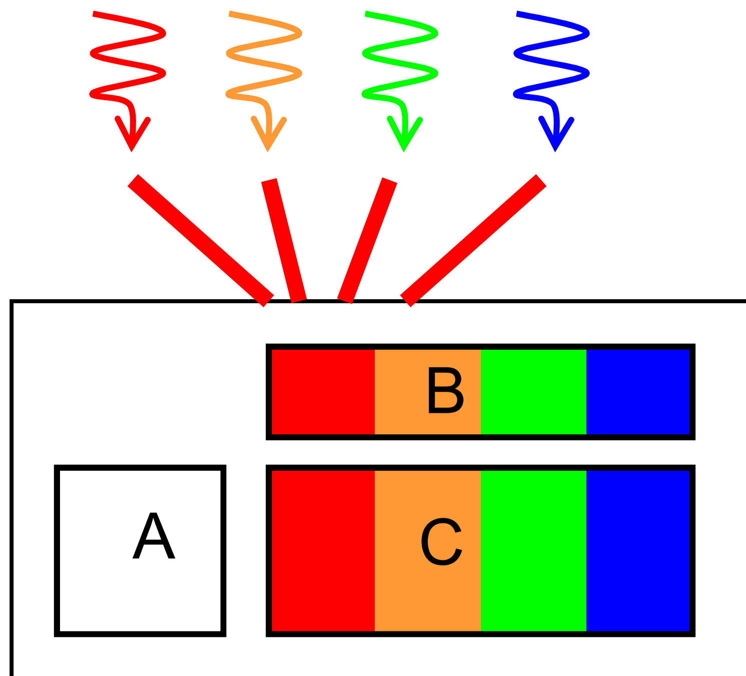
受信側には複数メッセージがやってくるかも → 受け取りたい条件を指定する

- 受け取りたい送信元を指定するか，MPI_ANY_SOURCE (誰からでもよい)
- 受け取りたいタグを指定するか，MPI_ANY_TAG(どのタグでもよい)

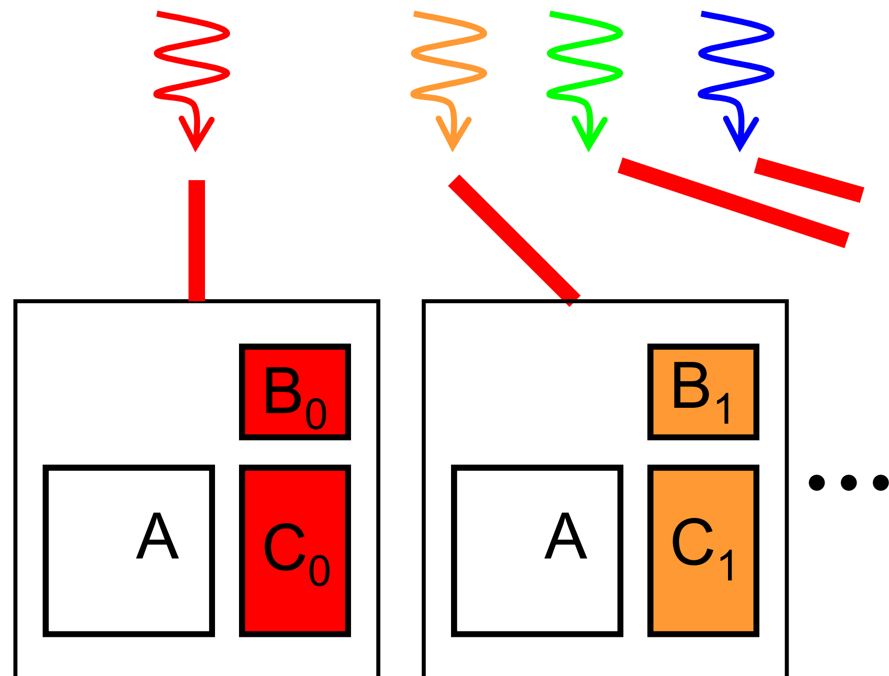


分散メモリと共有メモリの違い 行列積(C=AxB)の例

- 共有メモリ：計算をどうスレッドに分割するか
- 分散メモリ：計算とデータをどうプロセスに分割するか



行列Aは全スレッドによってアクセスされる

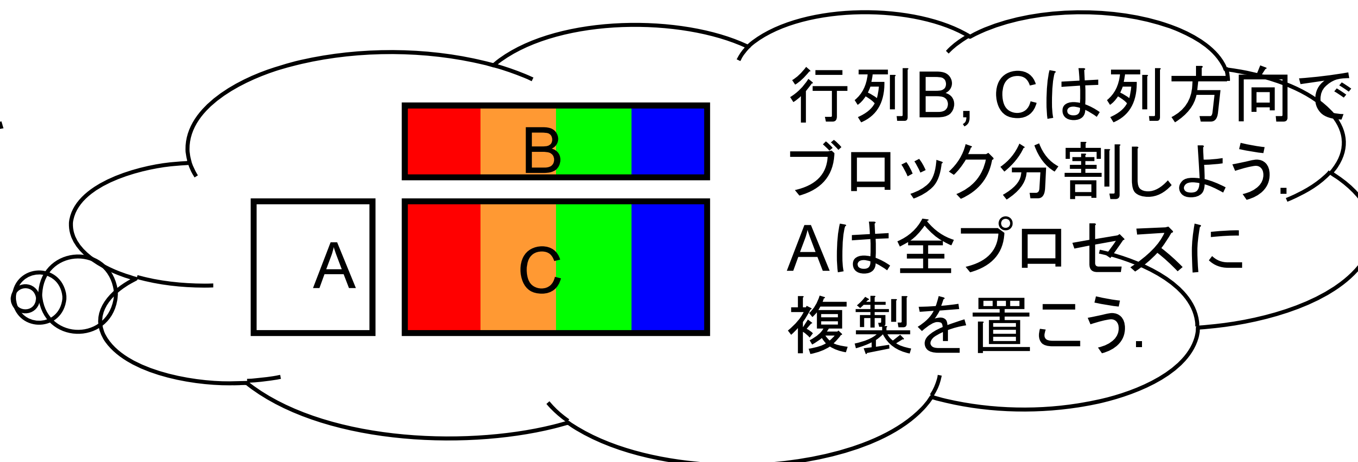


行列Aは全プロセスに置かれる

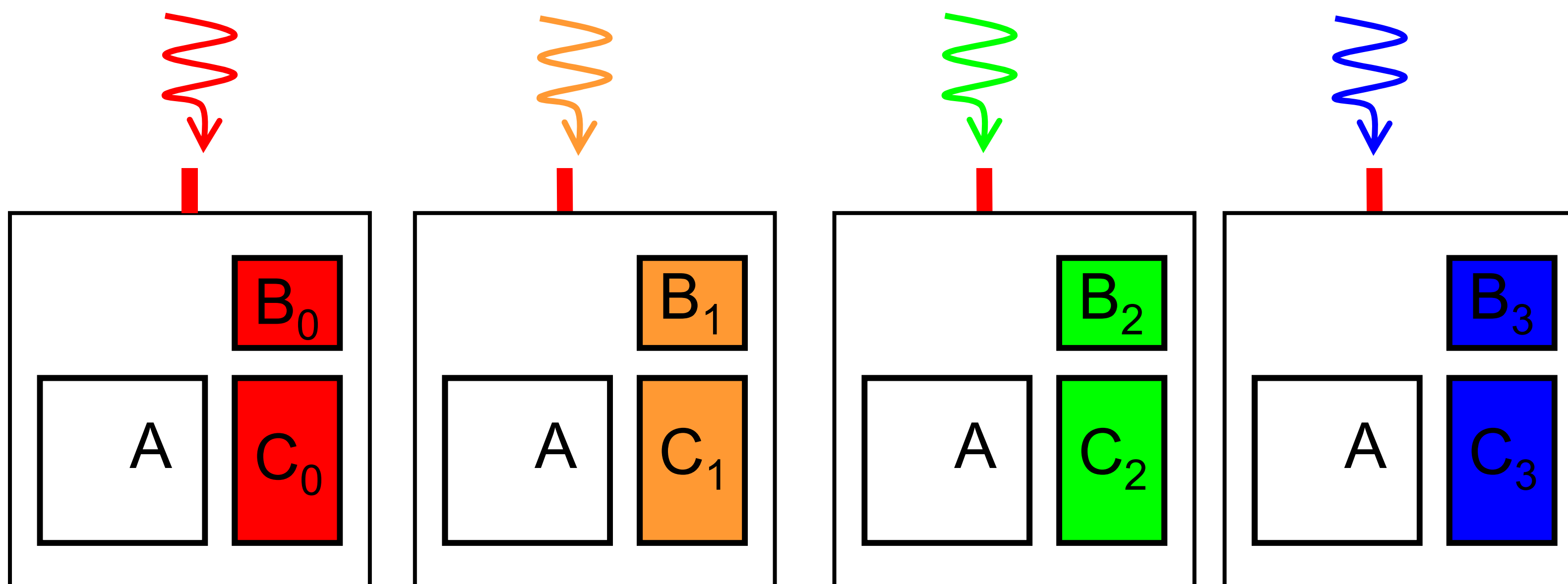
分散メモリプログラミングとデータ配置

(mm-mpiを題材に)

配置方法を決める:



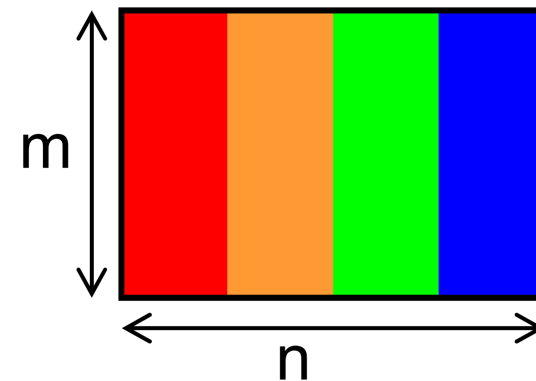
実際の配置をプログラミング:



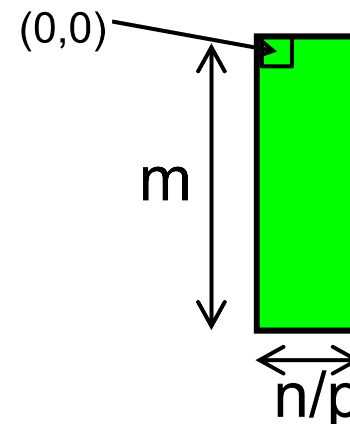
データ分散とプログラミング

- m×n行列をpプロセスで分割するとはどういうことか？
  – データ並びはcolumn-majorとする
  – ここでは割り切れる場合を仮定
- 各プロセスが持つのは，m×(n/p)の部分行列
  – m*(n/p)*sizeof(データ型)のサイズの領域をmallocすることに
  – 部分行列と全体行列の対応を意識する必要
    - プロセスrの部分行列の(i,j)要素⇔全体行列の(i, n/p*r + j)要素に対応

全体のイメージ



プロセスが持つ領域

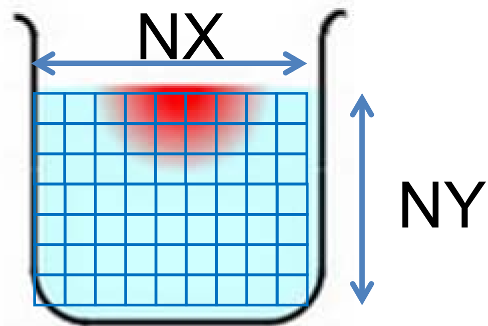


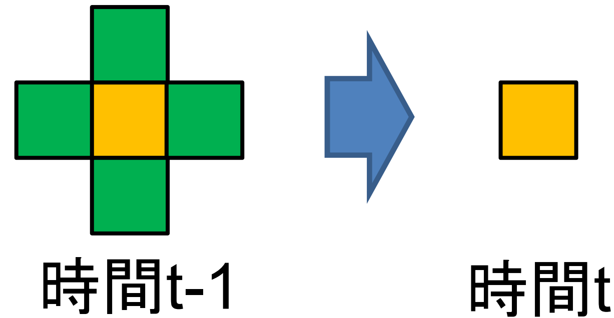
MPI版行列積

- LB・LC配列が、B・Cの部分行列だとする

```
MPI_Init(&argc, &argv);
    :
ln = n/nprocs;
for (int j = 0; j < ln; j++) {
   for (int l = 0; l < k; l++) {
      double blj = B[l+j*ldb];
      for (int i = 0; i < m; i++) {
        double ail = A[i+l*lda];
        C[i+j*ldc] += ail*blj;
      } }  }
```

DiffusionのMPI化に向けて





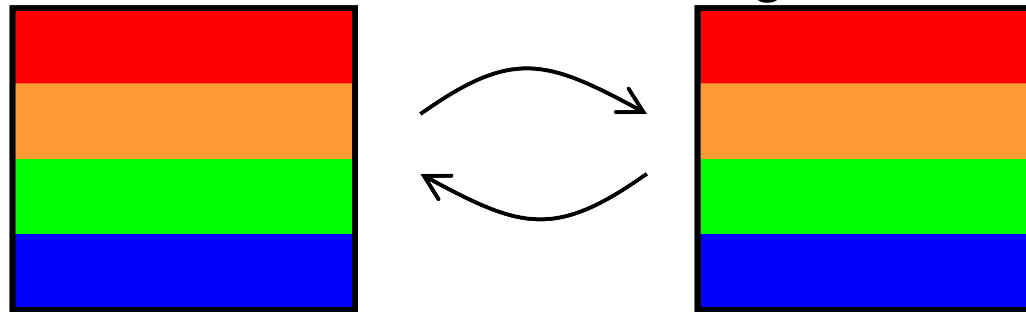
隣の点の情報を用いて値更新

並列化は、基本的に空間分割
→ プロセス間の境界が問題になる

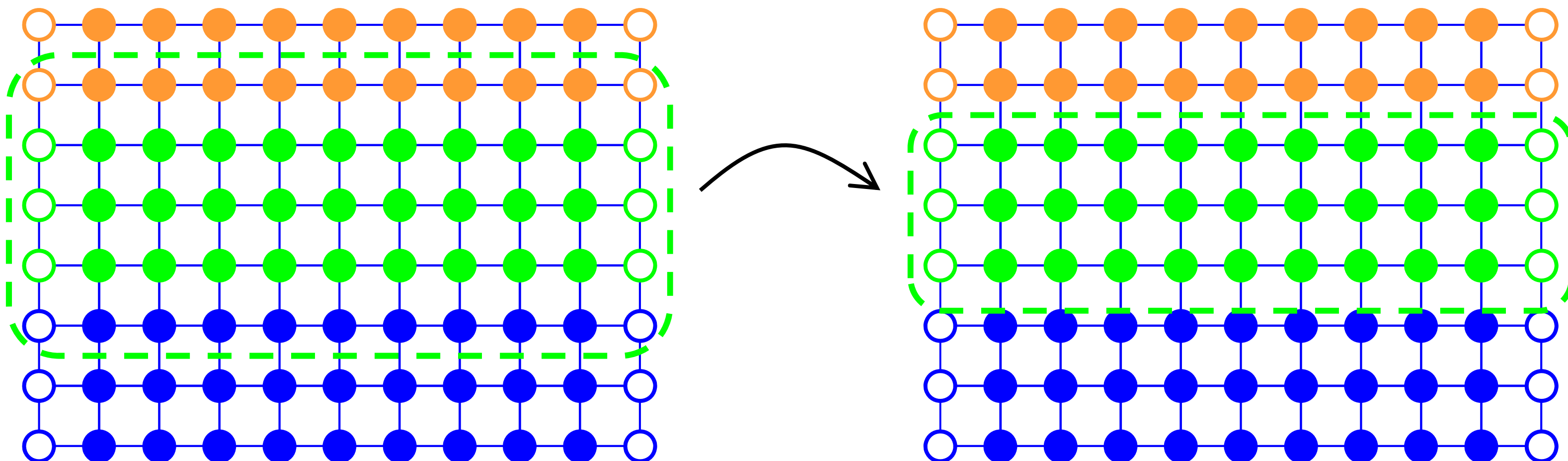
Diffusionの並列化方針

- 二次元配列をそれぞれ行方向分割

Double buffering



- 各プロセスがwriteする領域よりもreadする領域が大きい → プロセス間に依存関係



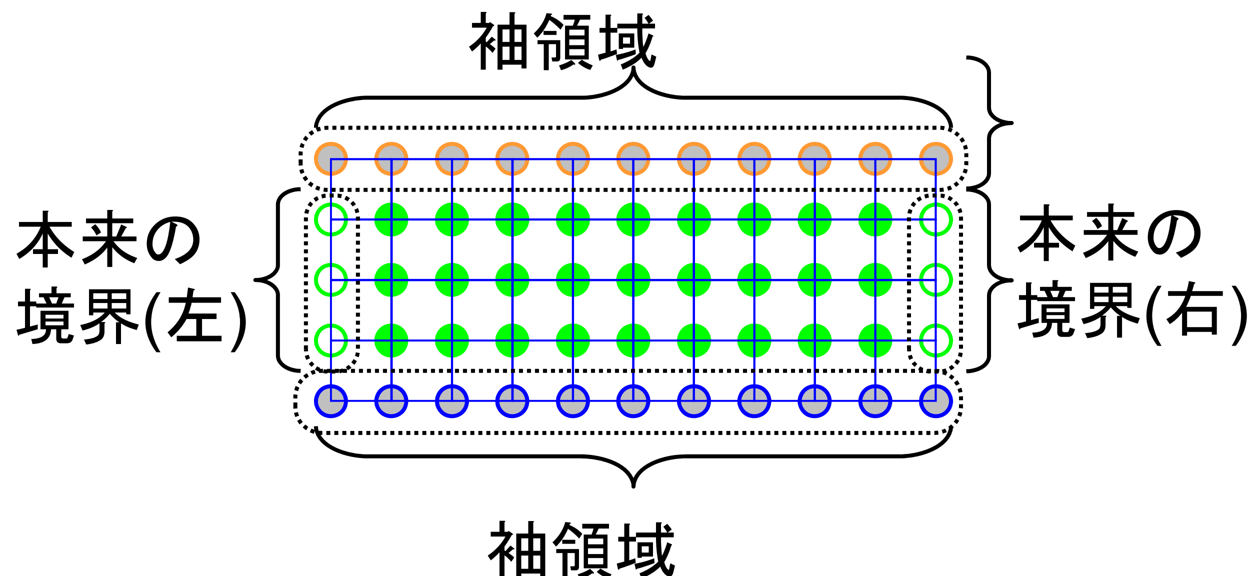
OpenMPではどうだったか

- データ構造は逐次と同じまま，forループを並列化すればよい
- 隣のスレッドのデータもそのまま読める，が・・・
- スレッド間で足並みをそろえる必要

→ parallel regionの終了時に自動でそろえられていた(バリア同期)

MPIによる並列化 (1)

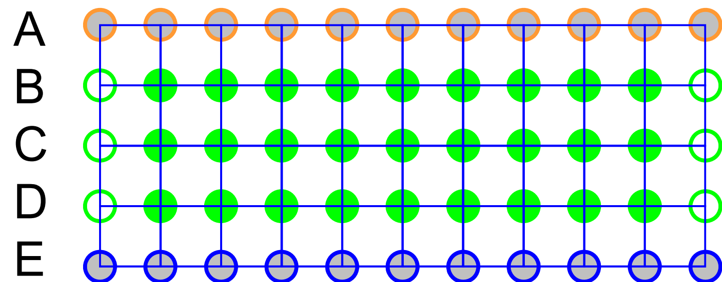
- 各プロセスは，自分の担当領域配列を持つ
  - 最初と最後のプロセスは，上下境界部分に注意
  - 端数処理
- 隣プロセスのデータを読むためには，send/recvが必要
- 「袖領域(のりしろ領域)」つきの配列を持つのがよい



MPIによる並列化 (2)

簡単化のため，以下ではのりしろ領域一行として説明

```
for (jt = 0; jt < NT; jt++) {
  行Bを前のプロセスへ送信， Dを次のプロセスへ送信
  行Aを前のプロセスから受信，Eを次のプロセスから受信  (注)

  B-Dの全点を計算
  二つの配列の切り替え
}
```



(注)実はこれはデッドロックするダメなプログラム.

デッドロック(deadlock)とは、互いに「待ちあって」プログラムが進まなくなること

MPIのまとめ

- 分散メモリモデル
- OpenMPでは、処理の並列化のみ考えればよかった(parallel forを使えばそれすら考えていないかも)
- MPIでは、さらにデータの分割を考える必要あり
- 純粋にMPIを使う方法と、MPI+OpenMPのハイブリッドにする方法あり

アクセラレータのプログラミング

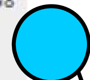
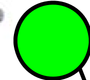
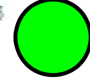
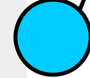
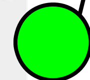
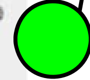
アクセラレータのプログラミング環境

- 乱立状況から、最近二年ほどで収束の方向へ

	NVIDIA GPU	Intel Xeon Phi	AMD GPU
CUDA	○	×	×
OpenMP	×	○	×
Intel Directive	×	○	×
OpenCL	○	○	○
OpenACC	○	○	○

- 「Openなんとか」により、収束の方向へあるが、依然専用環境のほうが性能が出やすい傾向に
- 上記はすべて、アクセラレータ1基向け。複数アクセラレータ、複数ノードのためには"MPI+なんとか"など

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3120000	33862.7	54902.4	17808
2	DOE/SC/Oak Ridge National Laboratory United States	Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560640	17590.0	27112.5	8209
3	DOE/NNSA/LLNL United States	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1572864	17173.2	20132.7	7890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705024	10510.0	11280.4	12660
5	DOE/SC/Argonne National Laboratory United States	Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786432	8586.6	10066.3	3945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115984	6271.0	7788.9	2325
7	Texas Advanced Computing Center/Univ. of Texas United States	Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462462	5168.1	8520.1	4510
8	Forschungszentrum Juelich (FZJ) Germany	JUQUEEN - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	458752	5008.9	5872.0	2301
9	DOE/NNSA/LLNL United States	Vulcan - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	393216	4293.3	5033.2	1972
10	Government United States	Cray XC30, Intel Xeon E5-2697v2 12C 2.7GHz, Aries interconnect Cray Inc.	225984	3143.5	4881.3	
11	Exploration & Production - Eni S.p.A. Italy	HPC2 - iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.8GHz, Infiniband FDR, NVIDIA K20x IBM	62640	3003.0	4006.3	1067
12	Leibniz Rechenzentrum Germany	SuperMUC - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR IBM	147456	2897.0	3185.1	3423
13	GSIC Center, Tokyo Institute of Technology Japan	TSUBAME 2.5 - Cluster Platform SL390s G7, Xeon X5670 6C 2.93GHz, Infiniband QDR, NVIDIA K20x	76032	2785.0	5735.7	1399

トップスパコンでのアクセラレータ利用

Intel Xeon Phi

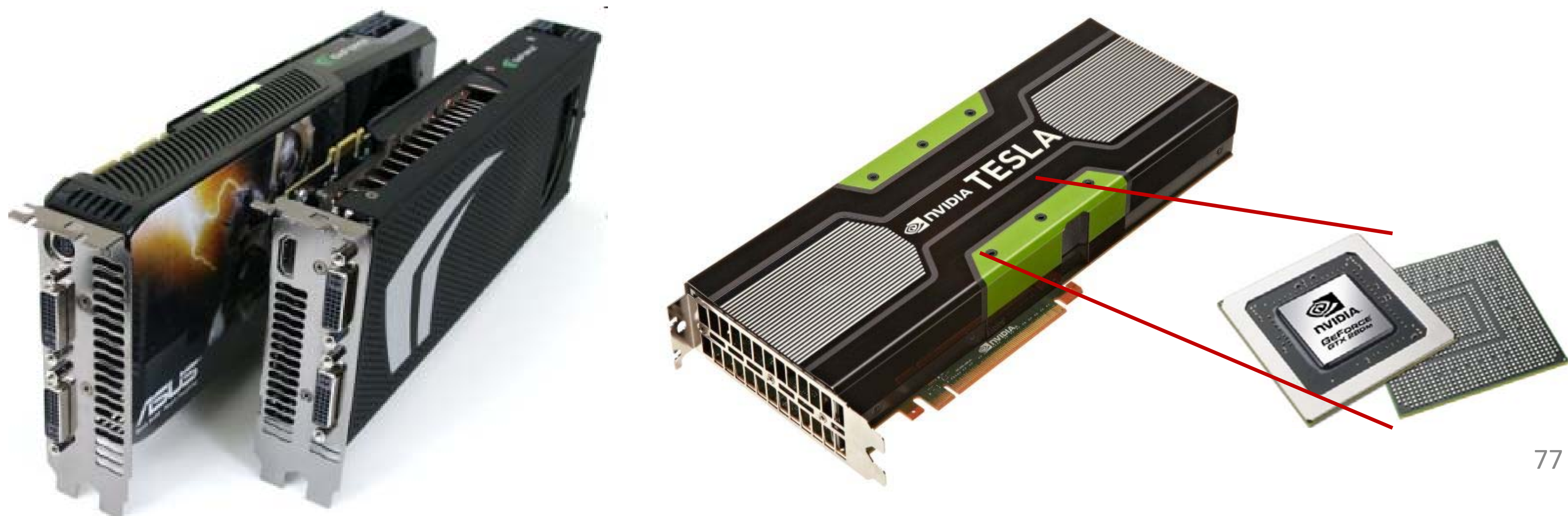
NVIDIA GPU

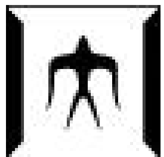
www.top500.org
2014/6ランキング

CUDAによるGPUプログラミング (TSUBAMEスパコンの紹介も含む)

GPUコンピューティングとは

- グラフィックプロセッサ (GPU)は、グラフィック・ゲームの画像計算のために、進化を続けてきた
  - 現在、CPUのコア数は2～12個に対し、GPU中には数百コア
- そのGPUを一般アプリケーションの高速化に利用！
  - GPGPU (General-Purpose computing on GPU) とも言われる
- 2000年代前半から研究としては存在。2007年にNVIDIA社のCUDA言語がリリースされてから大きな注目





TSUBAME2スーパーコンピュータ

Tokyo-Tech
Supercomputer and
UBiquitously
Accessible
Mass-storage
Environment

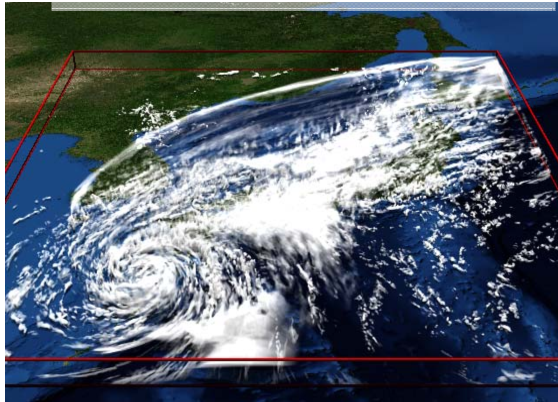
「ツバメ」は東京工業大学のシンボルマークでもある

- TSUBAME1: 2006年～2010年に稼働したスパコン
- TSUBAME2.0: 2010年に稼働開始したスパコン
  - 2010年当初には、世界4位、日本1位の計算速度性能
- TSUBAME2.5: 2013年にGPUを最新へ入れ替え
  - 現在、世界13位、日本2位

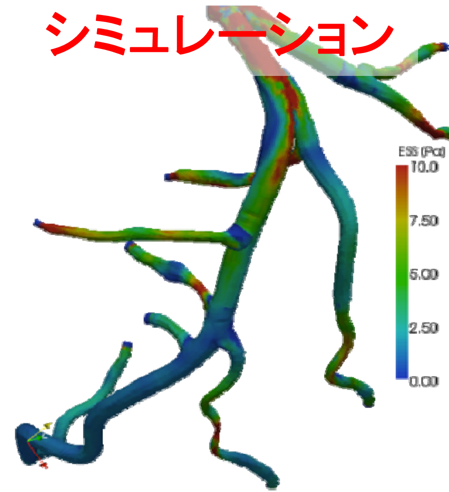
高性能の秘訣が
GPUコンピューティング

TSUBAME2スパコン・GPUは様々な研究分野で利用されている

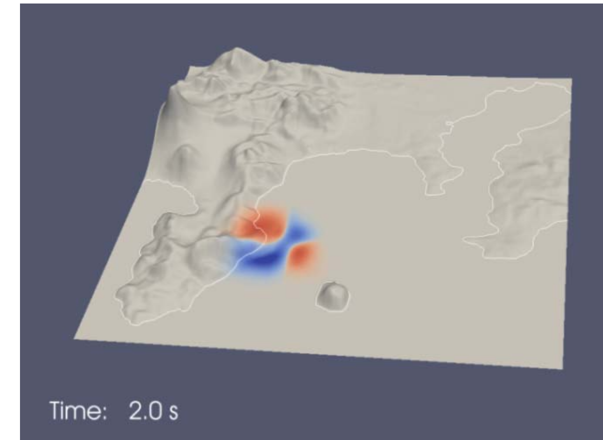
気象シミュレーション



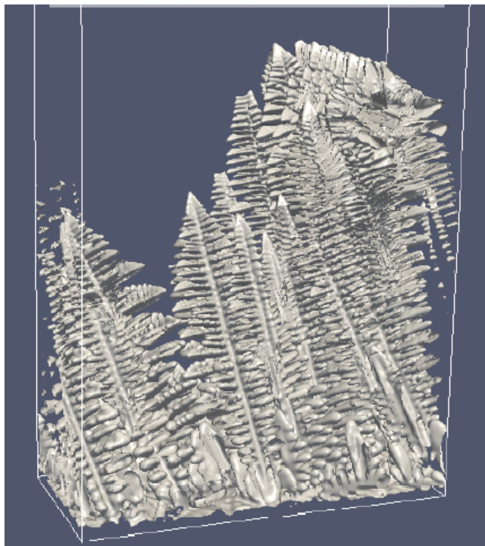
動脈血流シミュレーション



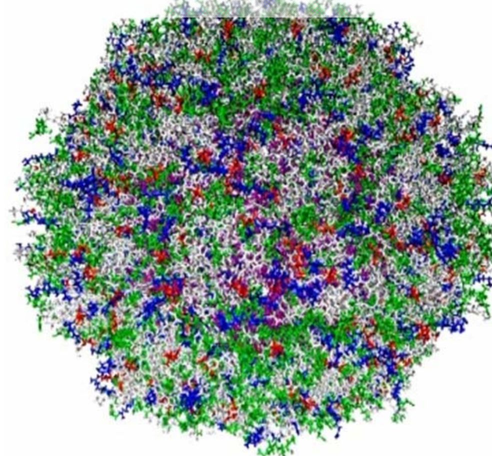
津波・防災シミュレーション



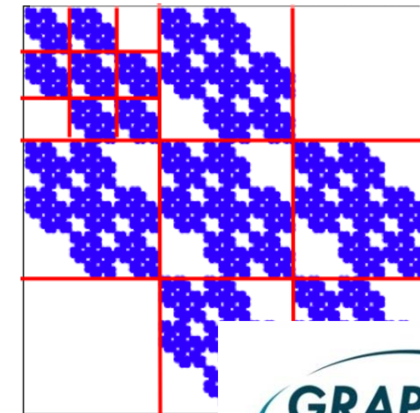
金属結晶凝固シミュレーション



ウィルス分子シミュレーション



グラフ構造解析



TSUBAME2.5の計算ノード

- TSUBAME2.0は、約1400台の計算ノード(コンピュータ)を持つ
- 各計算ノードは、CPUとGPUの両方を持つ
  - CPU: Intel Xeon 2.93GHz 6コア x 2CPU=12 コア
  - GPU: NVIDIA Tesla K20X x 3GPU

    CPU 0.07TFlops x 2 + GPU 1.31TFlops x 3 = 4.08TFlops

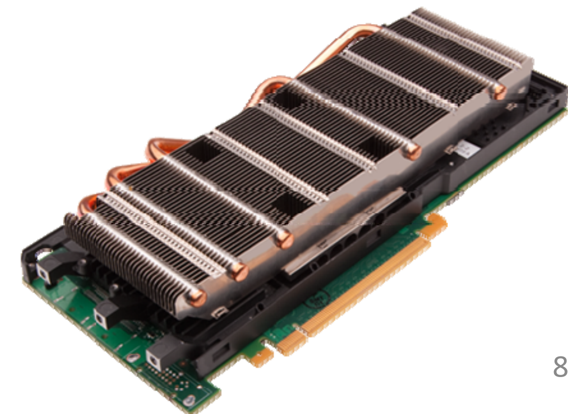
96%の性能がGPUのおかげ

  - メインメモリ(CPU側メモリ): 54GB
  - SSD: 120GB
  - ネットワーク: QDR InfiniBand x 2 = 80Gbps
  - OS: SUSE Linux 11 (Linuxの一種)

GPUの特徴 (1)

- コンピュータにとりつける増設ボード
  ⇒単体では動作できず、CPUから指示を出してもらう
- 多数コアを用いて計算
  ⇒多数のコアを活用するために、多数のスレッドが協力して計算
- メモリサイズは1～12GB
  ⇒CPU側のメモリと別なので、「データの移動」もプログラミングする必要

コア数・メモリサイズは、製品によって違う



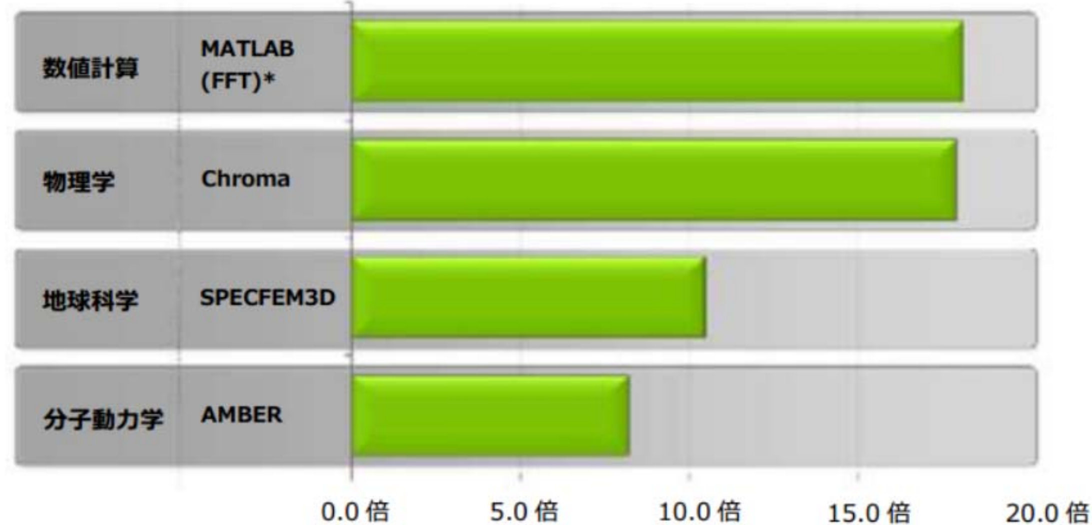
GPUの特徴 (2)

K20X GPU 1つあたりの性能

- 計算速度: 1.31 TFlops (倍精度)、3.95 TFlops (単精度)
  - CPUは20〜100GFlops程度
- コア数:
  - 14SMX x 192CUDAコア = 2688CUDAコア
- メモリ容量: 6GB
  - 2688コアが、6GBのメモリを共有している。ホストメモリとは別
- メモリバンド幅: 約250 GB/s
  - CPUは10〜50GB/s程度
- その他の特徴
  - キャッシュメモリ (L1, L2)
  - ECC
  - CUDA, OpenAcc, OpenCLなどでプログラミング

以前のGPUにはキャッシュメモリが無かったので、高速なプログラム作成がより大変だった

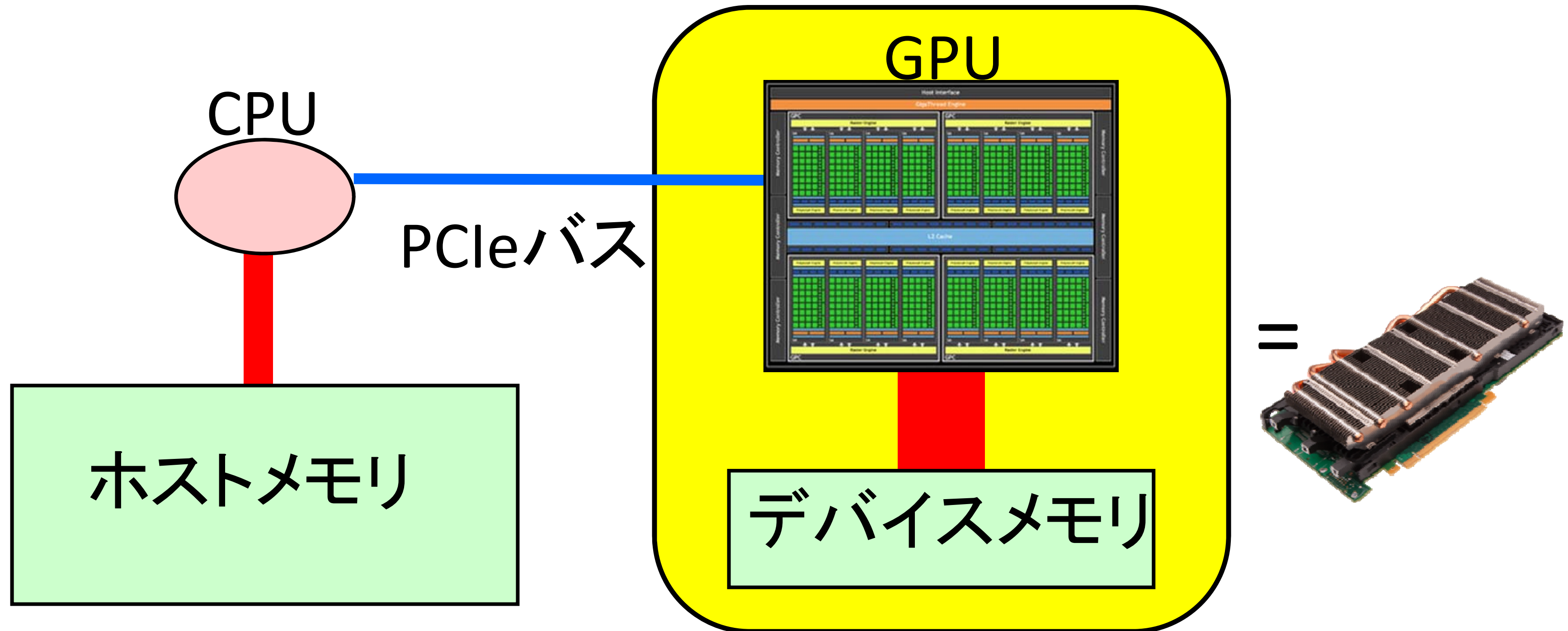
GPUの性能



NVIDIAの公開資料より

- CPU版の、同じ計算をするプログラムより数倍高速
  - CPU版もすでに並列化されている(はず)
- 宣伝通りにいくかどうかは、計算の性質とプログラミングの最適化しだい
  - どうしてもGPUに向かない計算はある

GPUを持つ計算機アーキテクチャ

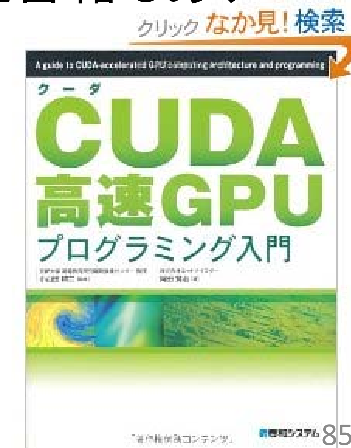


- ホストメモリとデバイスメモリは別の(分散)メモリ
- GPU中の全SMXは「デバイスメモリ」を共有
- SMX中のCUDA coreはSIMD的に動作

プログラミング言語CUDA

- NVIDIA GPU向けのプログラミング言語
  - 2007年2月に最初のリリース
  - TSUBAME2で使えるのはV5.5
  - 基本的に1GPU向け → 多数GPUはCUDA+MPIなどで
- 標準C言語サブセット＋GPGPU用拡張機能
  - C言語の基本的な知識(特にポインタ)は必要となります
  - Fortran版もあり
- nvccコマンドを用いてコンパイル
  - ソースコードの拡張子は.cu

CUDA関連書籍もあり



サンプルプログラム: inc_seq.cu

int型配列の全要素を1加算

GPUであまり意味がない
(速くない)例ですが

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>

#define N  (32)
__global__ void inc(int *array, int len)
{
   int i;
   for (i = 0; i < len; i++)
       array[i]++;
   return;
}
int main(int argc, char *argv[])
{
   int i;
   int arrayH[N];
   int *arrayD;
   size_t array_size;
```

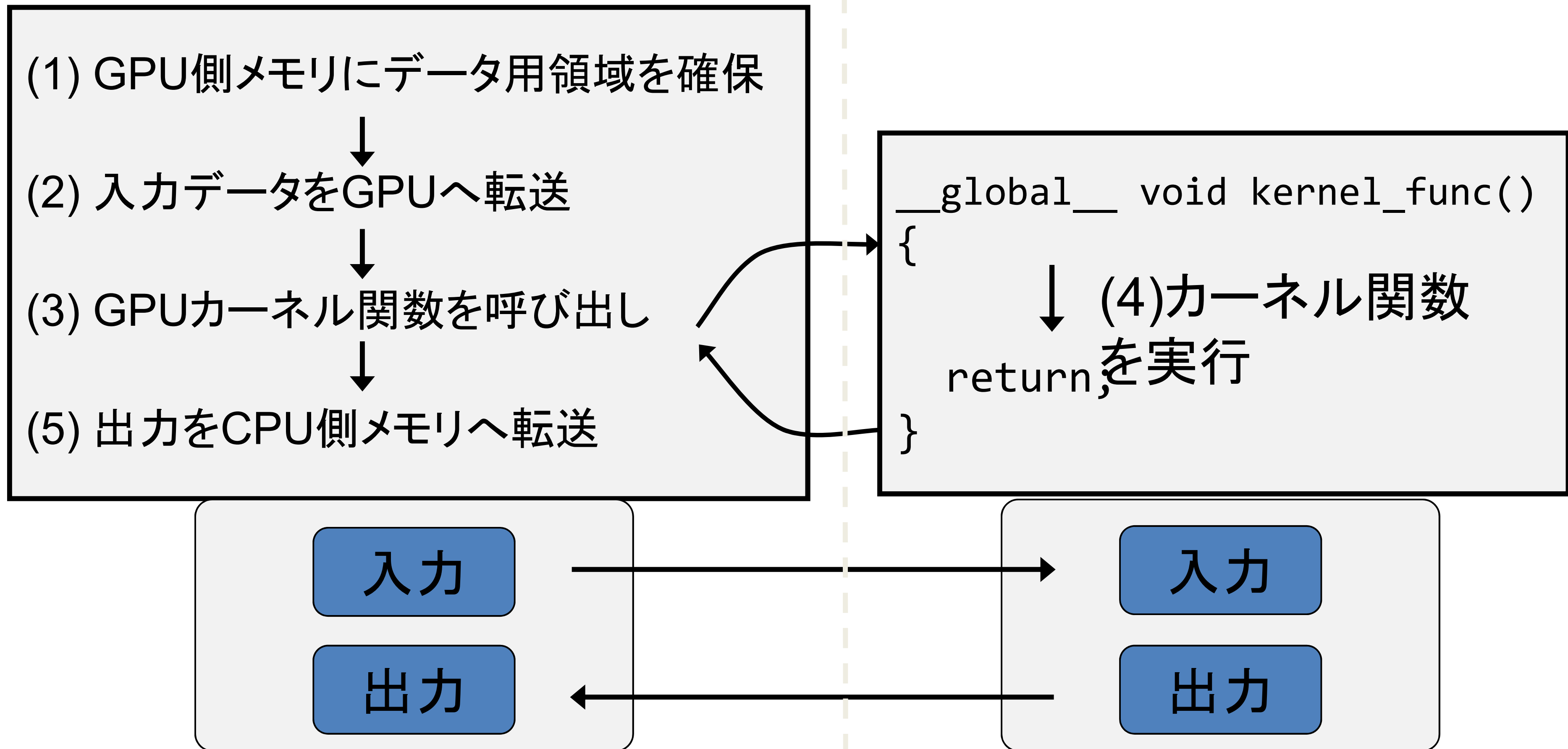
```
   for (i=0; i<N; i++) arrayH[i] = i;
   printf(“input: “);
   for (i=0; i<N; i++)
       printf(“%d “, arrayH[i]);
   printf(“¥n”);

   array_size = sizeof(int) * N;
   cudaMalloc((void **)&arrayD, array_size);
   cudaMemcpy(arrayD, arrayH, array_size,
           cudaMemcpyHostToDevice);
   inc<<<1, 1>>>(arrayD, N);
   cudaMemcpy(arrayH, arrayD, array_size,
              cudaMemcpyDeviceToHost);
   printf(“output: “);
   for (i=0; i<N; i++)
       printf(“%d “, arrayH[i]);
   printf(“¥n”);
   return 0;
}
```

CUDAプログラム構成

ホスト関数 + GPUカーネル関数

- 二種類の関数がcuファイル内に混ざっている
- ホスト関数
  - CPU上で実行される関数
  - ほぼ通常のC言語。main関数から処理がはじまる
  - GPUに対してデータ転送、GPUカーネル関数呼び出しを実行
- GPUカーネル関数
  - GPU上で実行される関数 (サンプルではinc関数)
  - ホストプログラムから呼び出されて実行
  - (単にカーネル関数と呼ぶ場合も)

典型的な制御とデータの流れ



(1) CPU上: GPU側メモリ領域確保

- cudaMalloc(void **devpp, size_t count)
  - GPU側メモリ(*デバイスメモリ*、*グローバルメモリ*と呼ばれる)に領域を確保
  - devpp: デバイスメモリアドレスへのポインタ。確保したメモリのアドレスが書き込まれる
  - count: 領域のサイズ
- cudaFree(void *devp)
  - 指定領域を開放

例: 長さ1024のintの配列を確保

```
#define N (1024)
int *arrayD;
cudaMalloc((void **)&arrayD, sizeof(int) * N);
// arrayD has the address of allocated device memory
```

(2) CPU上：入力データ転送

- cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
  - 先にcudaMallocで確保した領域に指定したCPU側メモリのデータをコピー
  - dst: 転送先デバイスメモリ
  - src: 転送元CPUメモリ
  - count: 転送サイズ(バイト単位)
  - kind: 転送タイプを指定する定数。ここではcudaMemcpyHostToDeviceを与える

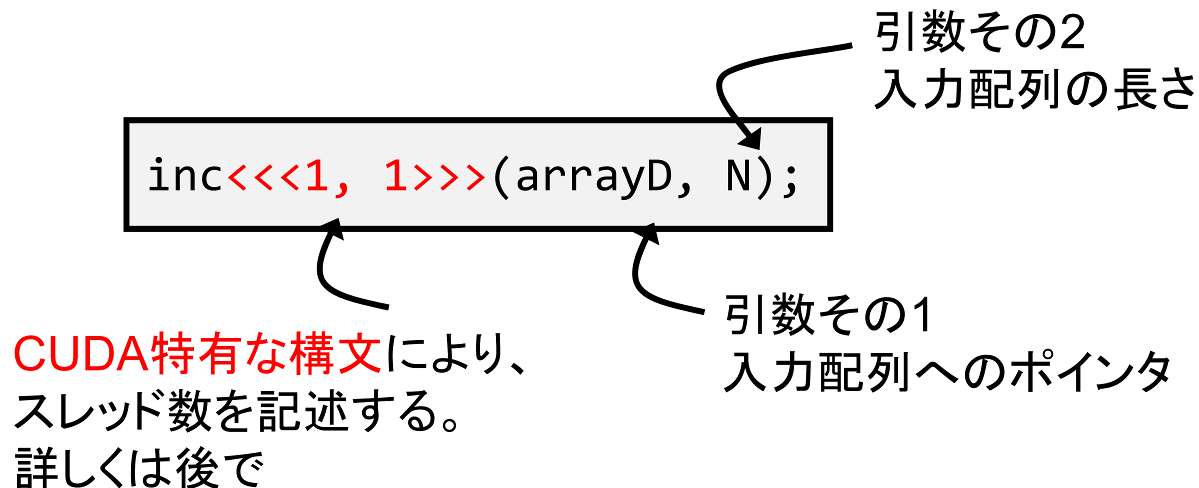
例： 先に確保した領域へCPU上のデータarrayHを転送

```
int arrayH[N];
cudaMemcpy(arrayD, arrayH, sizeof(int)*N,
           cudaMemcpyHostToDevice);
```

(3) CPU上: GPUカーネルの呼び出し

- `kernel_func<<<grid_dim, block_dim>>>(kernel_param1, …);`
  - `kernel_func`: カーネル関数名
  - `kernel_param`: カーネル関数の引数

例: カーネル関数 “inc” を呼び出し



(4) GPU上: カーネル関数

- GPU上で実行される関数
  - __global__というキーワードをつける

  注:「global」の前後にはアンダーバー2つずつ
- GPU側メモリのみアクセス可、CPU側メモリはアクセス不可
- 引数利用可能
- 値の返却は不可 (voidのみ)

例: int型配列をインクリメントするカーネル関数

```
__global__ void inc(int *array, int len)
{
    int i;
    for (i = 0; i < len; i++) array[i]++;
    return;
}
```

(5) CPU上: 結果の返却

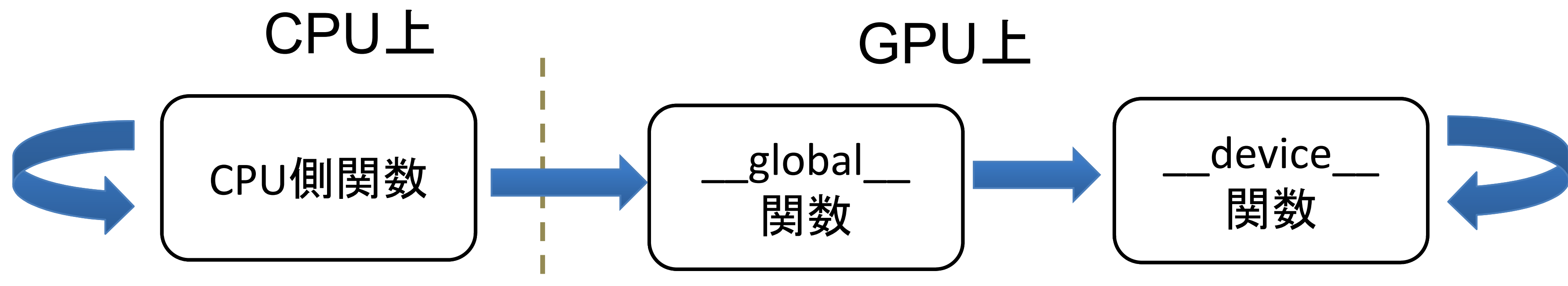
- 入力転送と同様にcudaMemcpyを用いる
- ただし、転送タイプは cudaMemcpyDeviceToHost を指定

例： 結果の配列をCPU側メモリへ転送

```
cudaMemcpy(arrayH, arrayD, sizeof(int)*N,
           cudaMemcpyDeviceToHost);
```

カーネル関数内でできること・できないこと

- if, for, whileなどの制御構文はok
- GPU側メモリのアクセスはok、CPU側メモリのアクセスは不可
  - inc_seqサンプルで、arrayDと間違ってarrayHをカーネル関数に渡してしまうとバグ!! (何が起こるか分からない)
- ファイルアクセスなどは不可
  - printfは例外的にokなので、デバグに役立つ
- 関数呼び出しは、「__device__つき関数」に対してならok



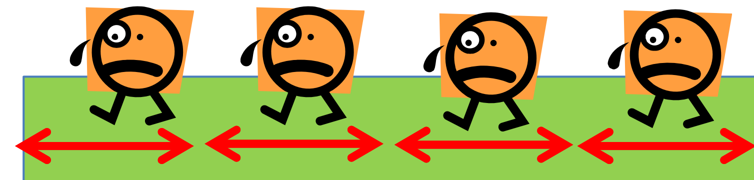
- 上図の矢印の方向にのみ呼び出しできる
  - GPU内からCPU関数は呼べない
- __device__つき関数は、返り値を返せるので便利

CUDAにおける並列化

- たくさんのスレッドがGPU上で並列に動作することにより、初めてGPUを有効活用できる
  - inc_seqプログラムは1スレッドしか使っていない
- データ並列性を基にした並列化が一般的
  - 例：巨大な配列があるとき、各スレッドが一部づつを分担して処理 → 高速化が期待できる

一人の小人が大きな畑を耕す場合

複数の小人が分担して耕すと速く終わる



CUDAにおけるスレッド

- CUDAでのスレッドは階層構造になっている
  - グリッドは、複数のスレッドブロックから成る
  - スレッドブロックは、複数のスレッドから成る
- カーネル関数呼び出し時にスレッド数を二段階で指定

```
kernel_func<<<100, 30>>>(a, b, c);
```

スレッドブロックの数

(スレッドブロックあたりの)スレッドの数

- この例では、100x30=3000個のスレッドがkernel_funcを　　並列に実行する

サンプルプログラムの改良

inc_parは、inc_seqと同じ計算を行うが、
N要素の計算のためにNスレッドを利用する点が違う

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>

#define N  (32)
#define BS (8)
__global__ void inc(int *array, int len)
{
    int i = blockIdx.x * blockDim.x +
        threadIdx.x;
    array[i]++;
    return;
}

int main(int argc, char *argv[])
{
   int i;
   int arrayH[N];
   int *arrayD;
   size_t array_size;
```

```
   for (i=0; i<N; i++) arrayH[i] = i;
   printf(“input: “);
   for (i=0; i<N; i++)
       printf(“%d “, arrayH[i]);
   printf(“¥n”);

   array_size = sizeof(int) * N;
   cudaMalloc((void **)&arrayD, array_size);
   cudaMemcpy(arrayD, arrayH, array_size,
           cudaMemcpyHostToDevice);
   inc<<<N/BS, BS>>>(arrayD, N);
   cudaMemcpy(arrayH, arrayD, array_size,
              cudaMemcpyDeviceToHost);
   printf(“output: “);
   for (i=0; i<N; i++)
       printf(“%d “, arrayH[i]);
   printf(“¥n”);
   return 0;
}
```

inc_parプログラムのポイント (1)

- N要素の計算のためにNスレッドを利用

```
inc<<<N/BS, BS>>>(.....);
```

グリッドサイズ

スレッドブロックサイズ
この例では、前もってBS=8とした

ちなみに、<<<N, 1>>>や<<<1, N>>>でも動くのだが非効率的である。

ちなみに、このままでは、NがBSで割り切れないときに正しく動かない。どう改造すればよいか？

inc_parプログラムのポイント (2)

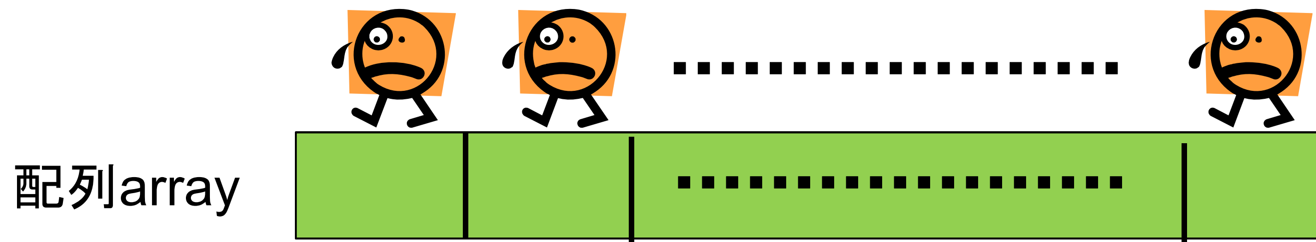
inc_parの並列化の方針

•(通算で)0番目のスレッドにarray[0]の計算をさせる

•1番目のスレッドにarray[1]の計算

:

•N-1番目のスレッドにarray[N-1]の計算



- 各スレッドは「自分は通算で何番目のスレッドか?」を知るために、下記を計算

i = blockIdx.x * blockDim.x + threadIdx.x;

使いまわせる
便利な式

- 1スレッドは"array[i]"の1要素だけ計算 → forループは無し

なぜCUDAではスレッドが二段階か

- ハードウェアの構造に合わせてある

ハードウェア (数値はK20Xの場合):

1 GPU = 14 SM

1 SM = 192 CUDA core

CUDAのモデル:

1 Grid = 複数thread block

1 thread block = 複数thread

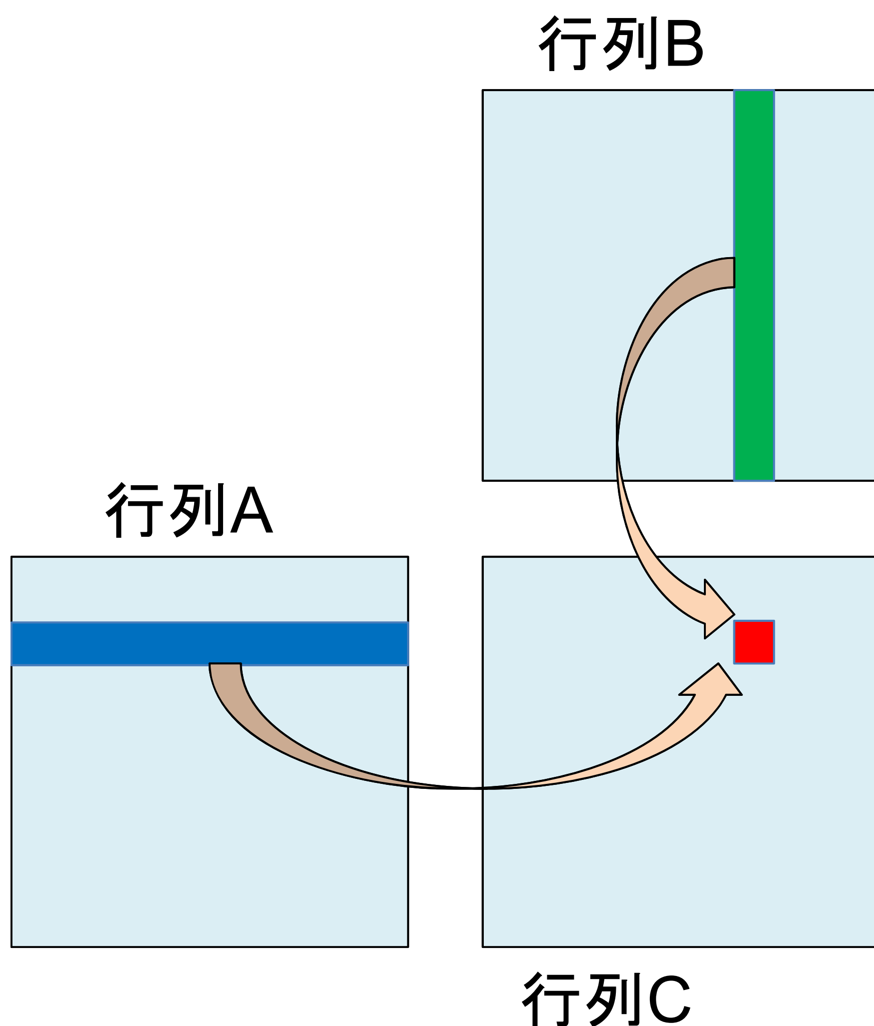
GPUの構造

1スレッドブロックは、必ず1SM上で動作
(複数スレッドブロックがSMを共有するのはあり)
1スレッドは、必ず1 CUDA coreで動作
(複数スレッドがCUDA coreを共有するのはあり)

スレッド数はどう決めればよい？

- CPUではスレッド数>コア数にしても、効率は上がらないか、むしろ下がる
- グリッドサイズが14以上、かつスレッドブロックサイズが192以上の場合に効率的
  - K20X GPUでは
    - GPU中のSM数=14
    - SM中のCUDA core数=192 なので
  - ぎりぎりよりも、数倍以上にしたほうが効率的な場合が多い(ベストな点はプログラム依存)
  - 理由は、メモリアクセスのオーバーラップができるから
    - メモリ待ちでプロセッサが待つ代わりに、他のスレッド達を実行できる
    - CPUでもhyperthreadingで同様の効果あるが、せいぜいコアあたり2ハードウェアスレッド

CUDA版行列積の考え方(例)



CPU版:

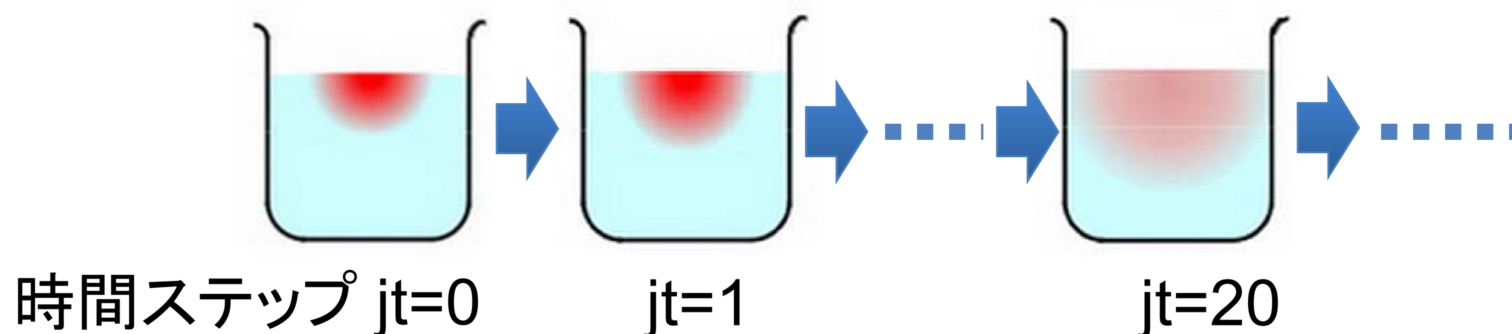
C全体を計算するためには、三重のforループ

CUDA版:

m*n個のスレッドを立ち上げ、各自がCの一点のみを計算すればよい

⇒カーネルの中身は一重ループ

CUDA版Diffusionの考え方



- 時間ループは順序そのまま
- 空間ループをスレッドたちで分割
  - NX*NY個のスレッドを起動することにすれば、カーネル内にはループなし！

CUDA版diffusionの流れ

CPU上で配列確保・初期条件作成

cudaMallocでGPUメモリ上の領域確保(配列二枚分)

初期条件の二次元格子データをCPUからGPUへ(cudaMemcpy)

For (jt = 0; jt < NT; jt++) //時間ループ

  全格子点をGPUで計算 // <<< >>> 構文を使う

  二つのバッファを交換

結果の二次元格子データをGPUからCPUへ(cudaMemcpy)

※ 時間ループの中に(格子全体の)cudaMemcpyを置くと非常に遅い

CUDAではコア間並列とSIMD並列は統一的

- あたかも、各スレッドは独立に動いているように「見える」
- スレッドブロック内のブロック達は、(プログラマからは見えないが)32スレッドごとの塊(warp)単位で動作している
- Warpの中の32スレッドは、「常に」足並みをそろえて動いている

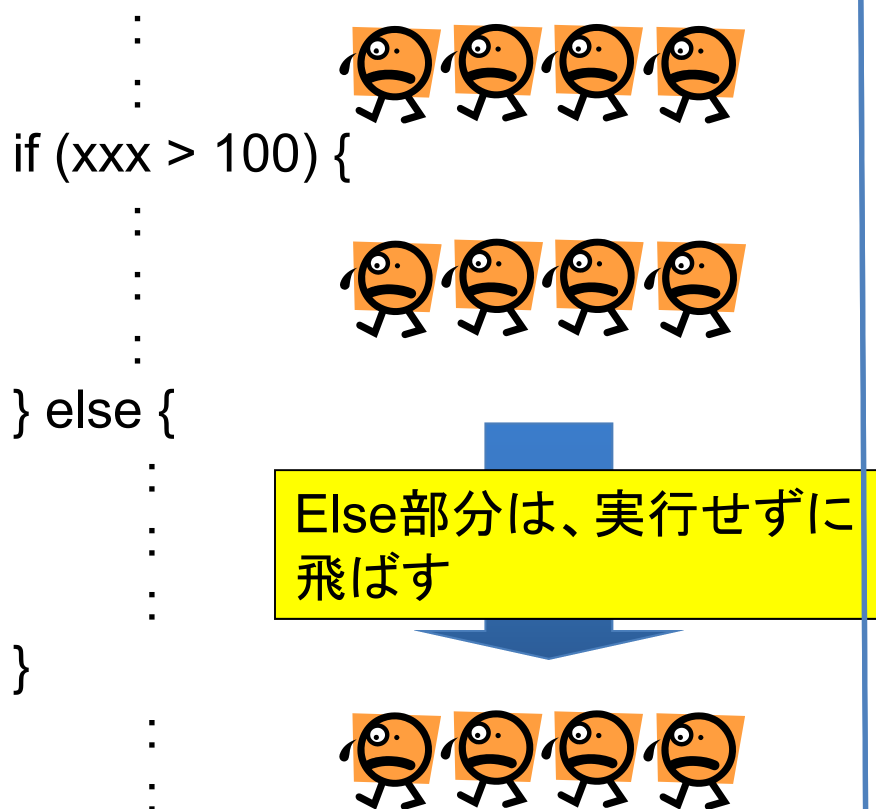
→If文などの分岐があるとどうなる？

  – Warp内のスレッド達の「意見」がそろうか、そろわないかで、動作が異なる

GPU上のif文の実行のされ方

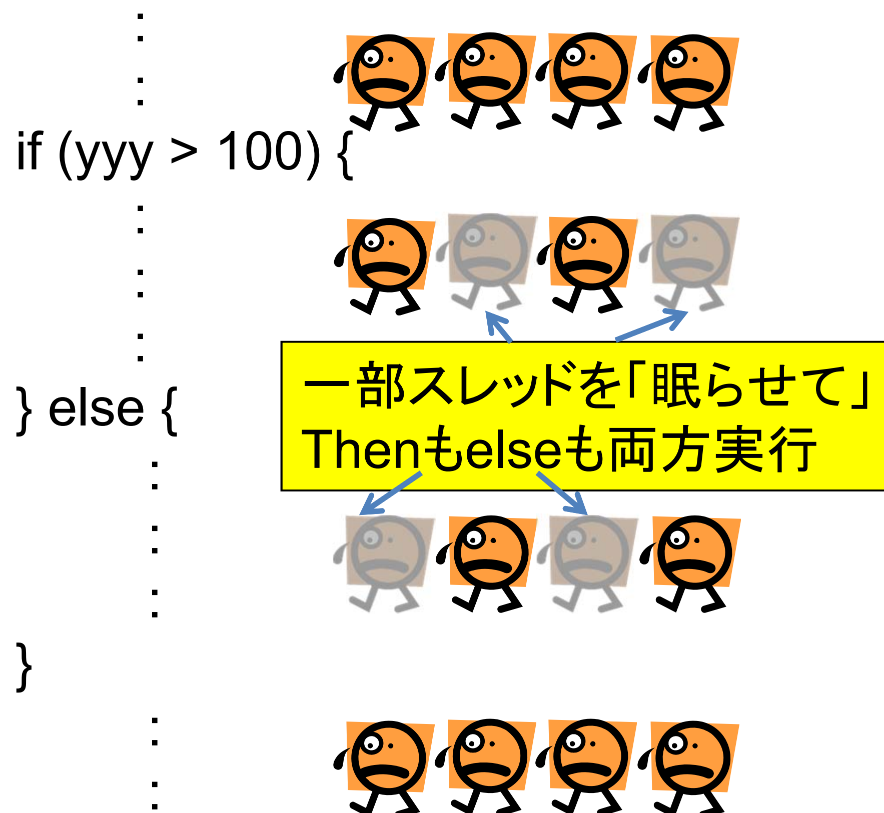
(a) スレッド達の意見がそろう場合

- 全員、xxx>100だとする



(b) スレッド達の意見が違う場合

- あるスレッドではyyy>100だが、別スレッドは違う場合



これを**divergent 分岐**と呼ぶ

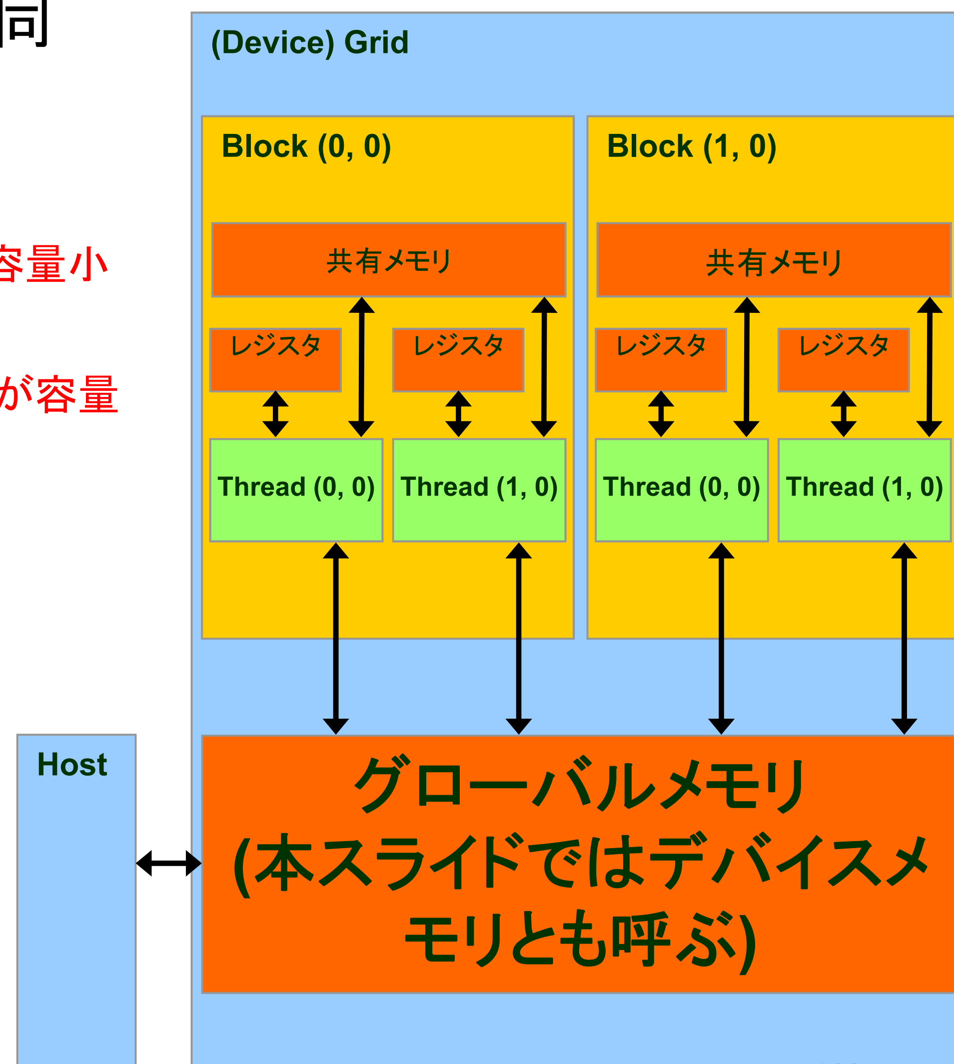
Divergent分岐はなぜ非効率?

- CPUの常識では、if文はthen部分とelse部分の片方しか実行しないので、片方だけの実行時間がかかる
- Divergent分岐があると、then部分とelse部分の両方の時間がかかってしまう

GPUのメモリは実はもっと複雑

スレッドが階層化されているのと同様、**メモリも階層化されている**

- スレッド固有
  - レジスタ → 局所変数を格納。高速だが容量小
- ブロック内共有
  - 共有メモリ → 本スライドで登場。高速だが容量小
  - （L1キャッシュ）
- グリッド内（全スレッド）共有
  - グローバルメモリ→ __global__変数やcudaMallocで利用。容量大きいが低速
  - （L2キャッシュ）



Source: Kirk and Hwu, ECE 498AL, UIUC

それぞれ速度と容量にトレードオフ有
（高速＆小容量 vs. 低速＆大容量）
→ メモリアクセスの局所性が重要

CUDAのまとめ

- NVIDIA GPU用の最も普及したプログラミング環境
- ホストメモリとデバイスメモリは別の(分散)メモリ
- GPU上の仕事(カーネル関数)を呼び出すときは <<< , >>>構文 ← ここがC言語から逸脱
- 多数(数百万)のスレッドを起動可能なため、元プログラムのループそのものが消える場合も

OpenACCによる
アクセラレータプログラミング

OpenACCの特徴

- 2012年ごろに発足、まだいろいろ流動的
- アクセラレータの種類を問わずに動作可能
- ディレクティブ(#pragma acc XXX)によりプログラミング
  - OpenMP的
  - ディレクティブを読み飛ばせば、CPU用の逐次プログラムとして動く
- 今後要注目！！

OpenACC対応コンパイラ

- PGIコンパイラ ← 2013年、PGI社をNVIDIAが買収。NVIDIAは本気らしい
- CAPS社HMPPコンパイラ
- Crayコンパイラ

上記は残念ながら有料。無料OpenACCコンパイラプロジェクトも動いている

行列積

OpenMP版

```
#pragma omp parallel for
 for (int j = 0; j < n; j++) {

   for (int l = 0; l < k; l++) {
     double blj = B[l+j*ldb];
     for (int i = 0; i < m; i++) {
       double ail = A[i+l*lda];
       C[i+j*ldc] += ail*blj;
     } }  }
```

OpenACC版

```
#pragma acc data
copyin(A[0:m*k], B[0:k*n])
copyout(C[0:m*n])
#pragma acc kernels loop
independent
 for (int j = 0; j < n; j++) {
#pragma acc loop independent
   for (int l = 0; l < k; l++) {
     double blj = B[l+j*ldb];
     for (int i = 0; i < m; i++) {
       double ail = A[i+l*lda];
       C[i+j*ldc] += ail*blj;
     } }  }
```

Intel Xeon Phiについて少々

Intel Xeon Phi

- Intel社のアクセラレータ
  - Larrabeeプロジェクトの後継
- 現状は、GPUに似た拡張ボード型
- ホストメモリとデバイスメモリは別

GPUとの大きな違い

- ボード上でLinuxが動いており、ボードにログインできる
- 約60個のコアを持っており、各コアはIntel x86互換
  - 一コアあたり4 hyper threadで、Linuxから見ると240コア共有メモリマシン！

Xeon Phi上の主に2種類のプログラミング手法

- Nativeモード
  - ボードにログインして、ふつうの共有メモリマシンとして利用
  - OpenMPプログラムが動く
- Offloadモード
  - Intelコンパイラのディレクティブ付プログラムを開発し、ホスト側から実行
  - ディレクティブで指定された部分がXeon Phi上で実行される ← OpenACCとほぼ同じ実行モデル

ここまでのまとめ

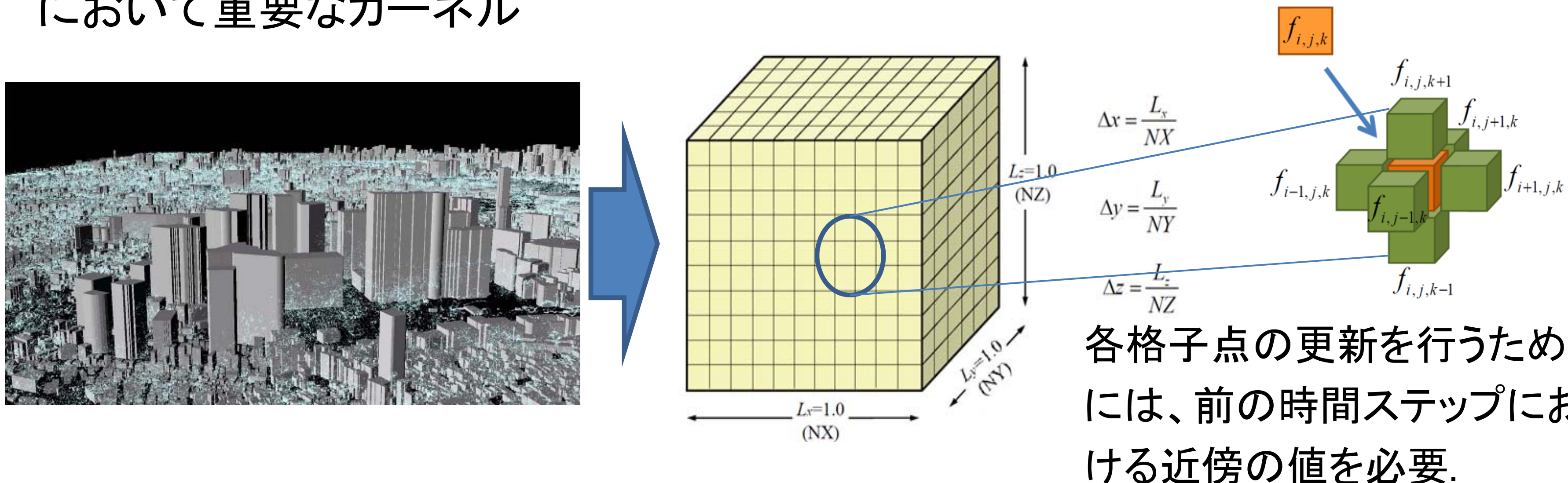
- 様々な並列プログラミング環境を駆け足で解説した
  - OpenMP, MPI, AVX/SSE, CUDA, OpenACC…
- 種類が多いのは、現代の計算機アーキテクチャの複雑さ・階層性に対応している
- アクセラレータのプログラミング環境は乱立 ⇒収束しつつあるか？

遠藤らの最近の研究：ポストペタスケール時代のメモリ階層の深化に対応するソフトウェア技術

JST-CREST「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」(2012-2017)

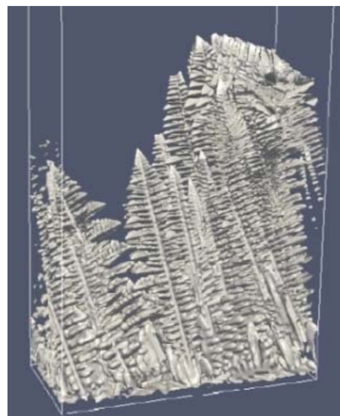
広範な応用を持つステンシル計算

ステンシル計算：連続体シミュレーションを中心とする様々な科学分野計算において重要なカーネル

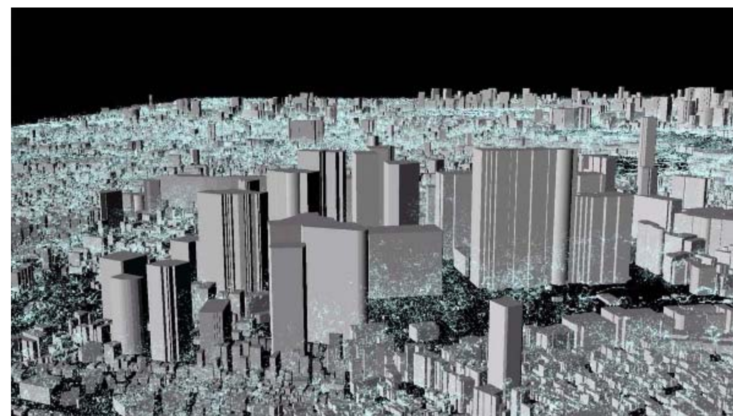


各格子点の更新を行うためには、前の時間ステップにおける近傍の値を必要.

気象コードASUCA



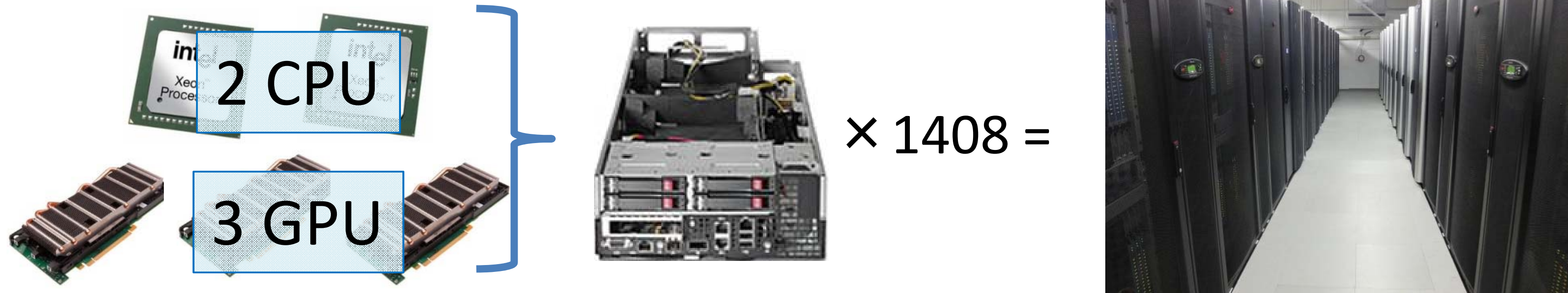
Phase-Field計算
(2011 Gordon Bell賞)



都市気流シミュレーション
(HPCS 2013 最優秀論文)

アクセラレータを持つスパコン

東工大TSUBAME2スパコン

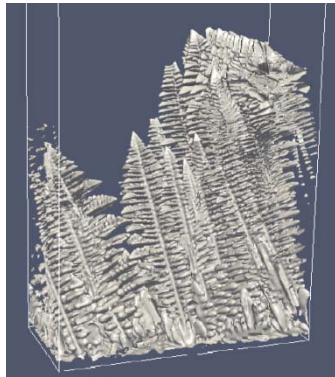


- 世界一のTianhe-2, 二位Titan, 筑波HA-PACS…
- Green500のTop10全部がアクセラレータ型スパコン

<u>ステンシル計算に向く理由</u>

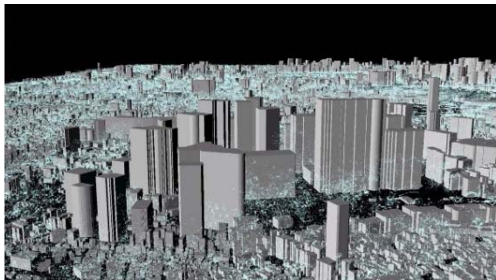
- GPUアクセラレータの特徴利用
  - 高いメモリバンド幅: 250GB/s × 4200
  - 高い演算速度: 3930GFlops(SP) × 4200
- マルチGPU時でも良好な計算・通信比
  - 領域分割⇒隣接のみの通信なので、通信量のオーダーは一つ低い

TSUBAME2上の ステンシルアプリケーション例



<u>デンドライトシミュレーション</u>

- 2011Gordon Bell賞アプリ
- 4000GPUで3.4PFlop/s(SP)



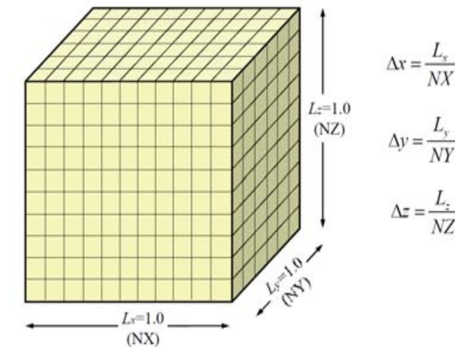
<u>都市気流シミュレーション</u>

- ステンシル (LBM)
- 要求B/F=1.83
- 1.14PFlops/s

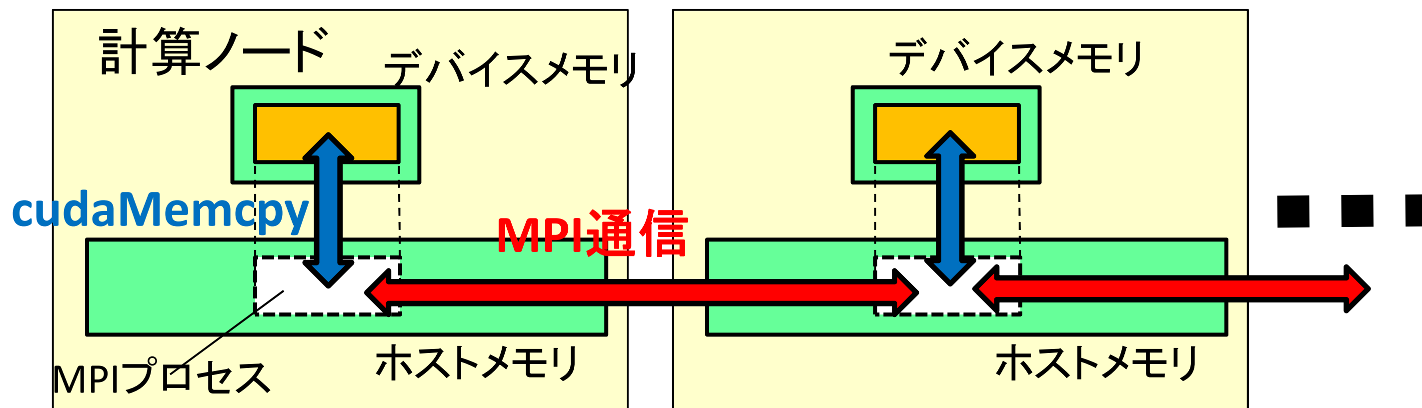
速度性能はok
しかし、精細度は6GB×4000に縛られる
GPUメモリ容量が10倍欲しい！

デバイスメモリ容量による限界

- 格子の細かさはGPUデバイスメモリ容量により限定される
  - TSUBAME2.5: 6GB × 4000 = 24TB
  - シミュレーションの精度を上げるため、もっと格子を細かくしたい！

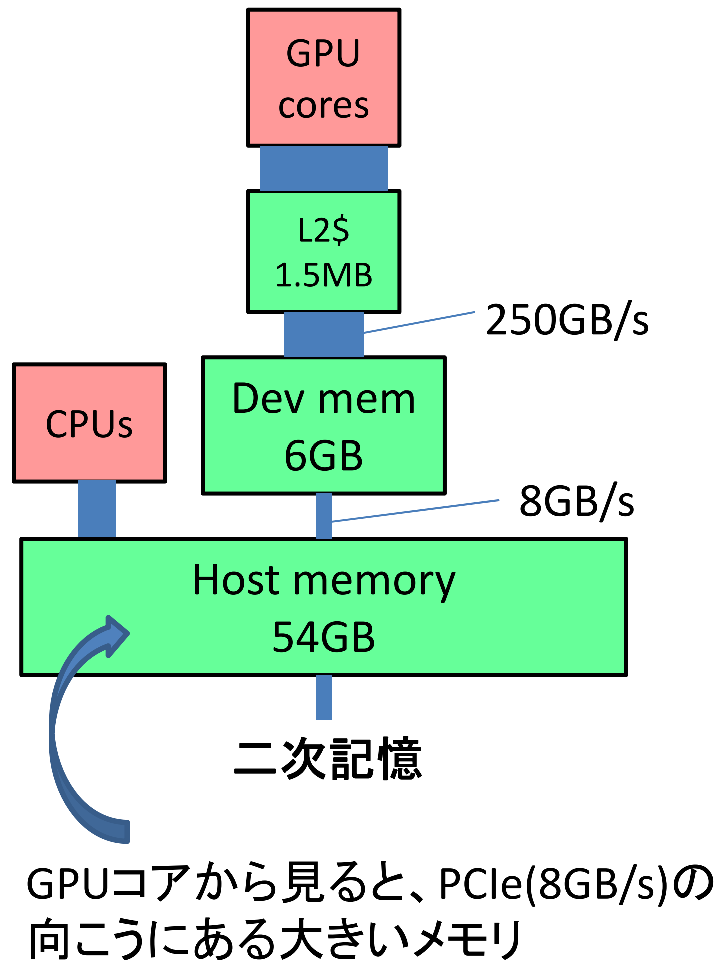


MPI+CUDAでかかれた典型的ステンシルプログラム実行中の様子



⇒ **ホストメモリの容量(TSUBAME2では計100TB)を活用できれば、さらに高精細なシミュレーション可能！**

メモリ階層としての デバイスメモリとホストメモリ



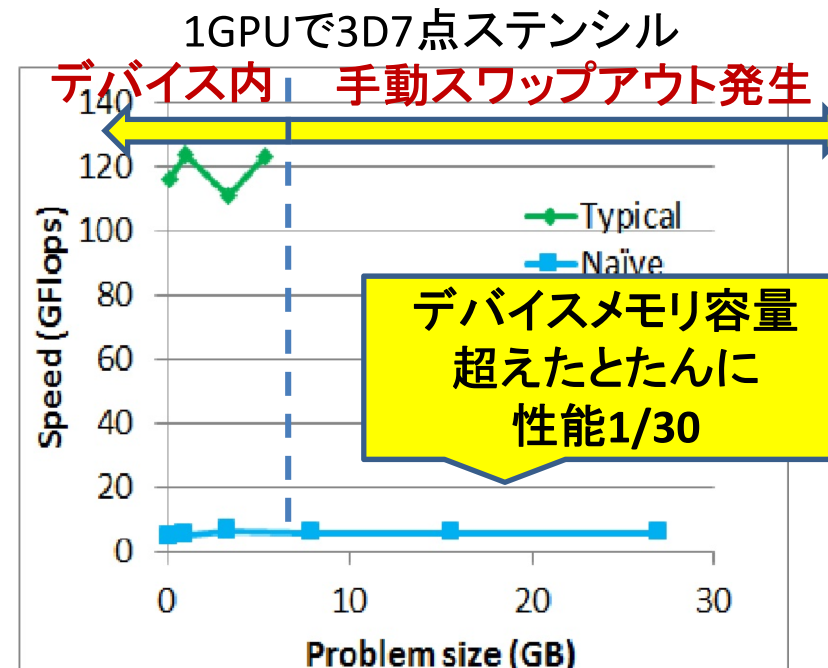
GPUコアから見ると、PCIe(8GB/s)の向こうにある大きいメモリ

デバイスメモリ⇔ホストメモリ間でスワップ機能があれば大容量使えそう

しかし問題！！

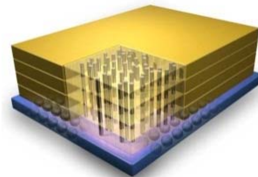
- [機能面] 現状GPUにスワップ機能なし
- [性能面] ステンシルで単純にスワップすると性能がひどいことに!

1GPUで3D7点ステンシル



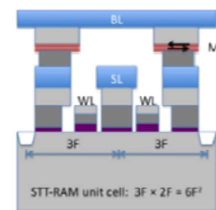
エクサ時代へ向けたメモリ階層の深化
アクセラレータ型スパコンだけの問題ではない

Hybrid Memory Cube (HMC)
- DRAMチップの3D積層化による高帯域化
- DDRより電力あたり容量は不利
- Micron/Intelなど

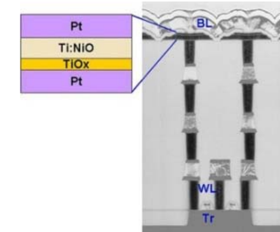


次世代 不揮発メモリ(NVRAM)
- DRAMと異なる記憶方式
- アクセス速度・密度・write耐性まちまち

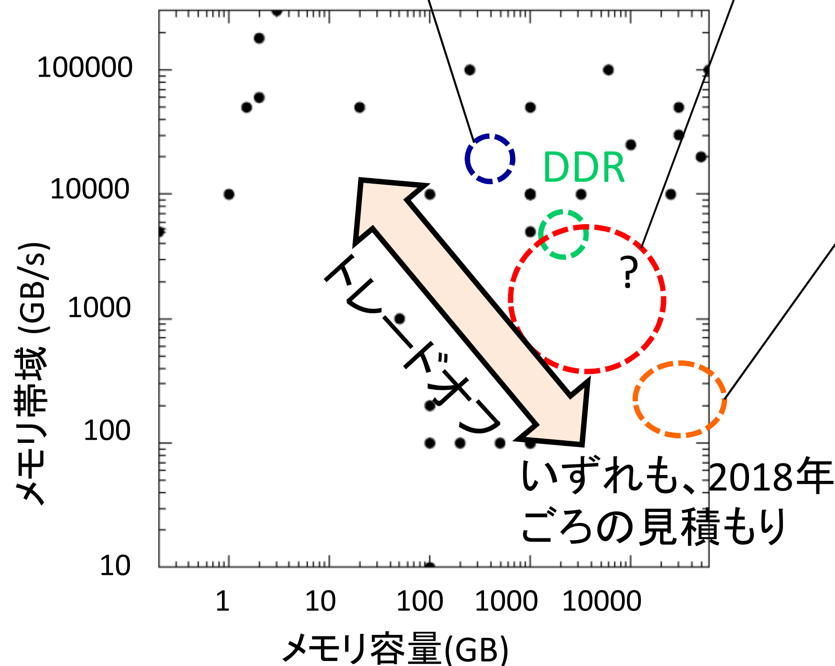
STT-MRAM



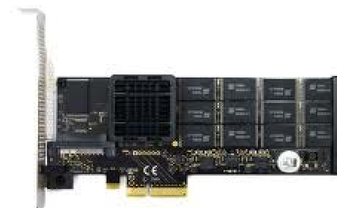
ReRAM



他、PCM, FeRAM・・・



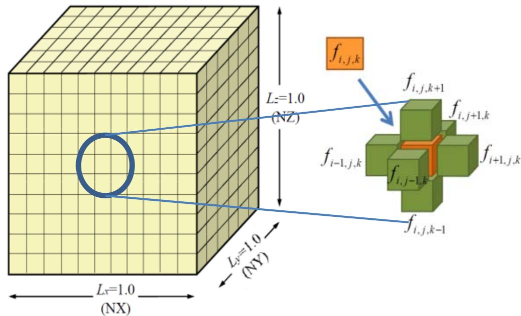
高速Flashメモリ
- PCI-Express直接接続・デバイス並列化によりO(GB/s)の帯域
- Solid State Accelerator(SSA)とも



Fusion-io社ioDrive

目標：ステンシル計算において、**問題大規模性と高性能性**を両立するには？

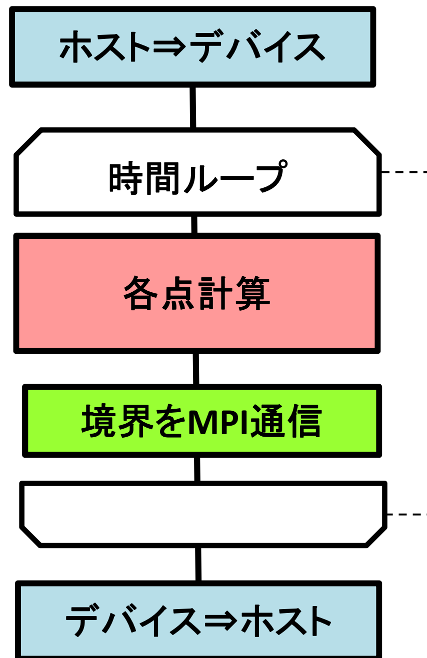
1GPU上の単純なステンシル計算と大容量対応

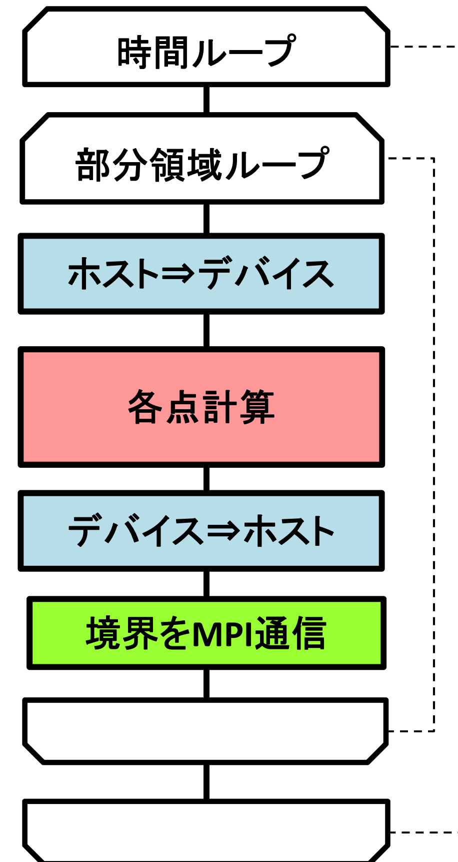


ハンドコーディングで大容量対応するには？

- 各ランクの担当領域をさらに細切れにし、ちょっとずつGPUへ送って計算

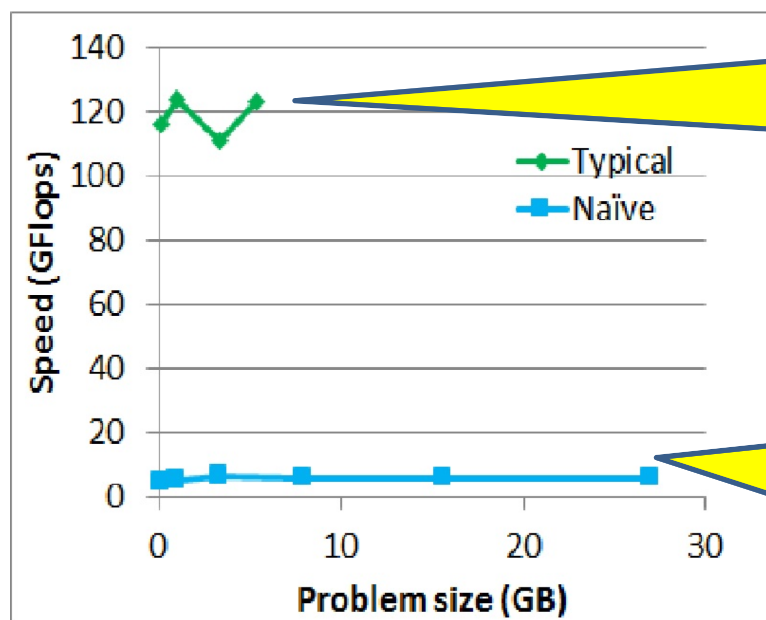
典型的な処理の流れ





単純なステンシル計算の性能

- TSUBAME2.5上のTesla K20X GPUを1GPU利用
- 3D立方体領域で7点ステンシルを実行



ふつうのステンシル実装はデバイスメモリ容量超えられない

デバイスメモリ容量超えられる実装は激しく低性能!!

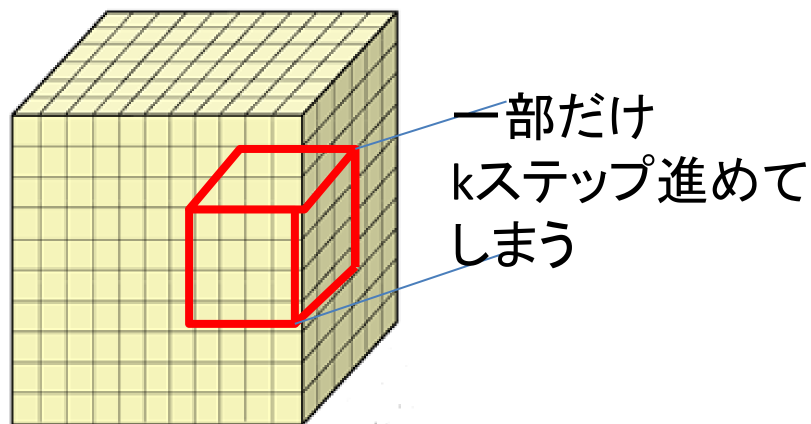
- 単純なステンシルの局所性の低さが問題
- 通信削減技術との組み合わせが必要
  - この文脈では、「メモリ階層間のデータ移動」削減 (not MPI)

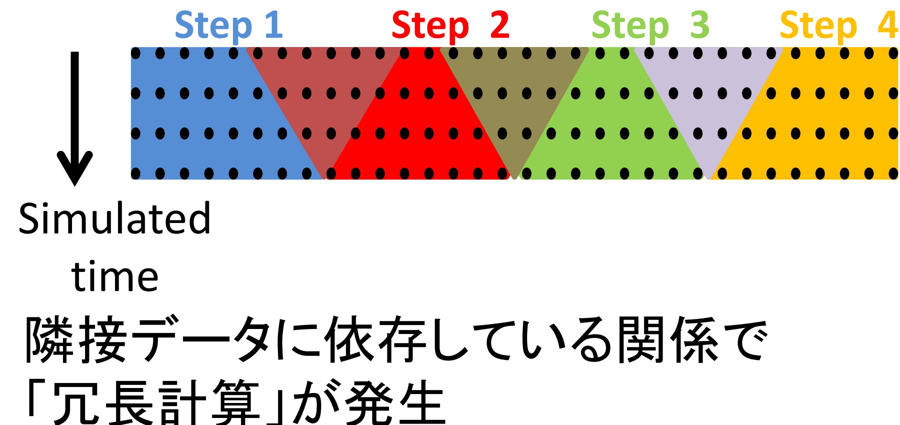
テンポラルブロッキングによる局所性向上

- データがO(1)回だけアクセスされて、デバイスメモリから追い出されるのが問題

  ⇒ 通信削減の必要！(Demmelsグループ等)

- テンポラルブロッキング: 部分領域に対し、複数回時間ステップを進めてしまう [Wolf 91] [Wonnacott 00] など
  - 以下、ブロッキング段数をkとする

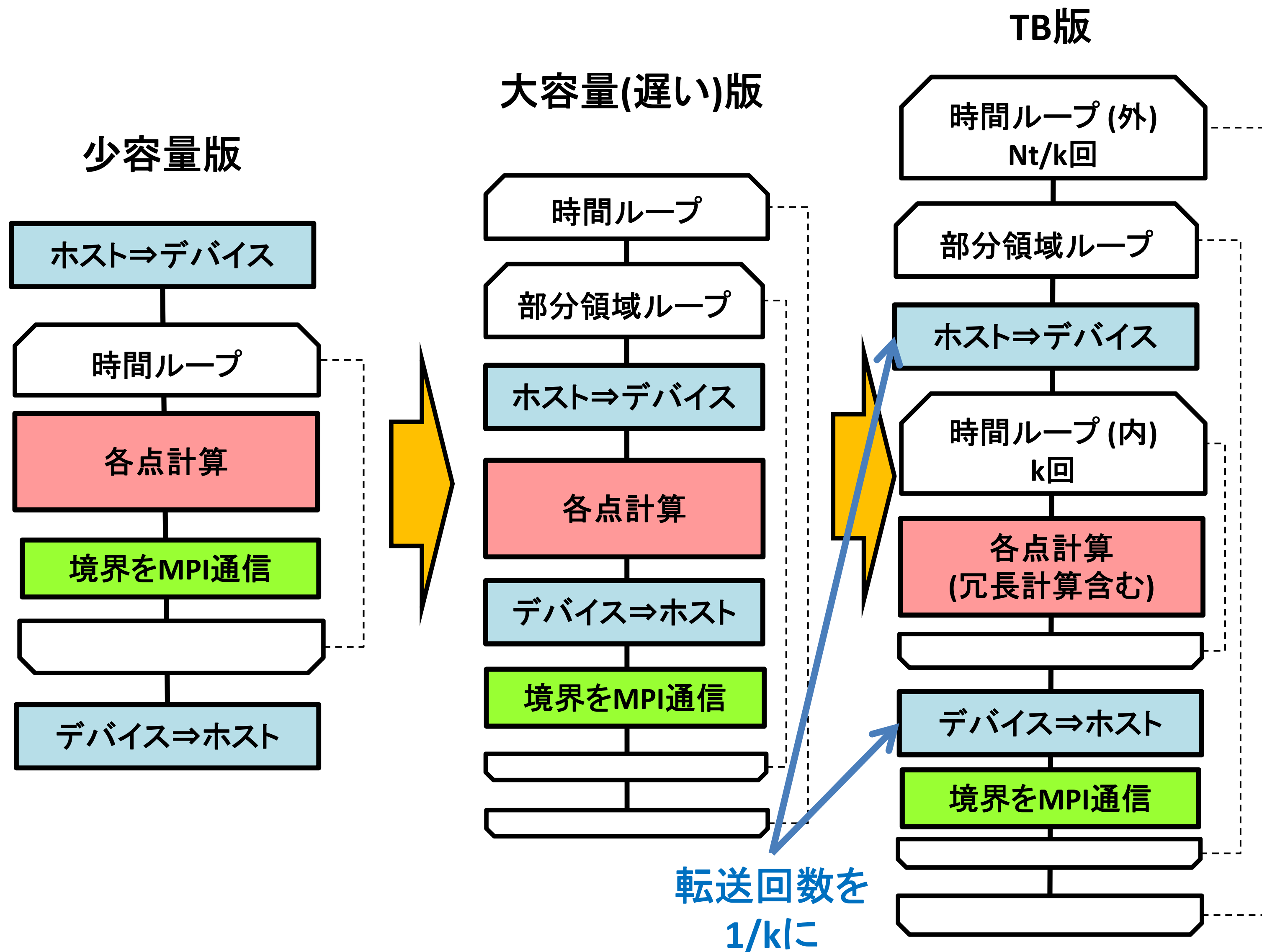




隣接データに依存している関係で「冗長計算」が発生

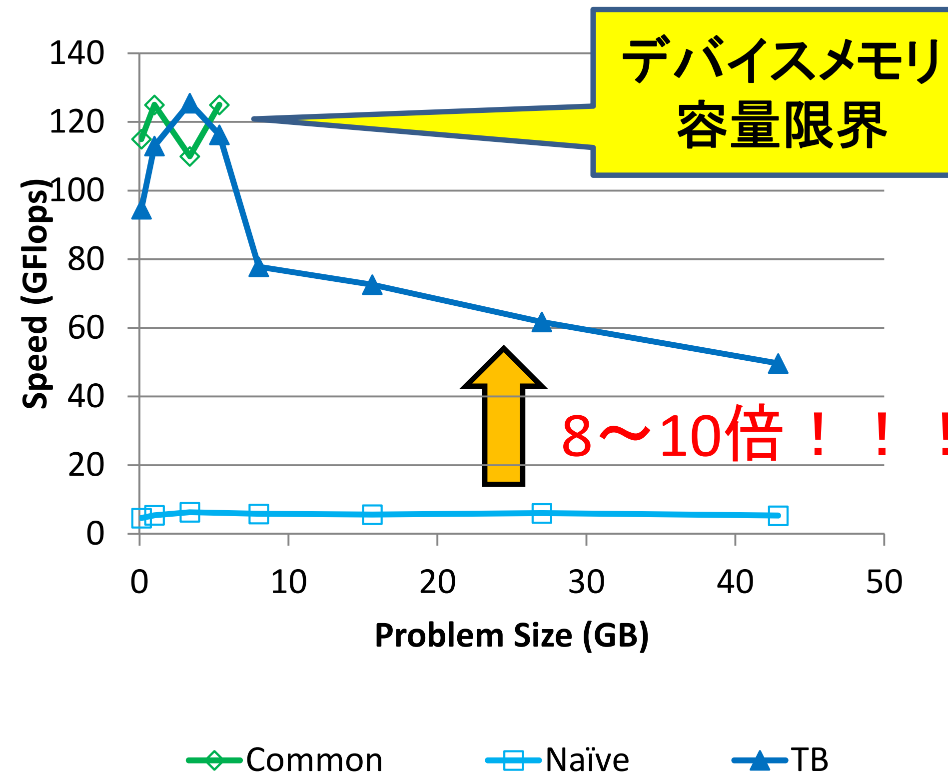
注) s-step Krylov部分空間法と性質は似ているが、こちらは元の方法と完全に同じ結果

テンポラルブロッキング導入に伴うユーザプログラム変更



テンポラルブロッキング導入時の性能

- Tesla M2050を1GPU用い、3D立方体領域で7点ステンシル
- ブロッキング段数kについては、各条件のパラメータスイープより決定



- テンポラルブロッキングにより、大規模時の性能大幅アップ!!

チューニングすべきパラメータ

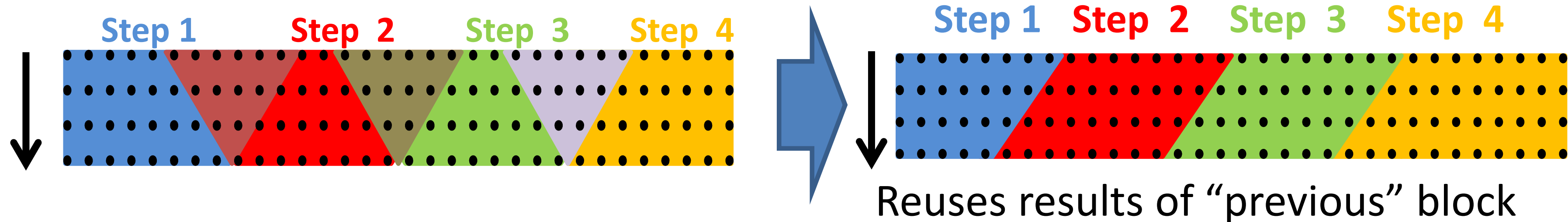
- 時間ブロックサイズk
  - kが小さいと通信量減らせず、kが大きいと冗長計算が増えてしまう⇒トレードオフ
  - ひたすらパラメータスイープ
  - 条件によって最適点は異なり、k~100の場合も
    - キャッシュ効率向上をメインにした既存研究では、k=2~8

一辺サイズ	720	840	1200	1440	1680	1920	2160
最適なk	2	3	10	16	28	40	60

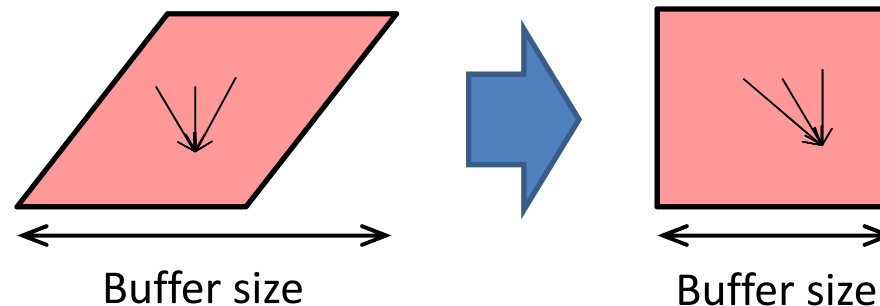
- 空間ブロックサイズ
  - 三次元空間を、今回はz方向のみで分割
  - ブロック体積×2≒GPUデバイスメモリ容量
  - x, y, z分割するともっとパラメータ膨大

テンポラルブロッキングの最適化

- 冗長計算の除去 (Wolfら、Demmelsら、北大岩下ら)
  – 前提：各ブロックが逐次的に処理されること



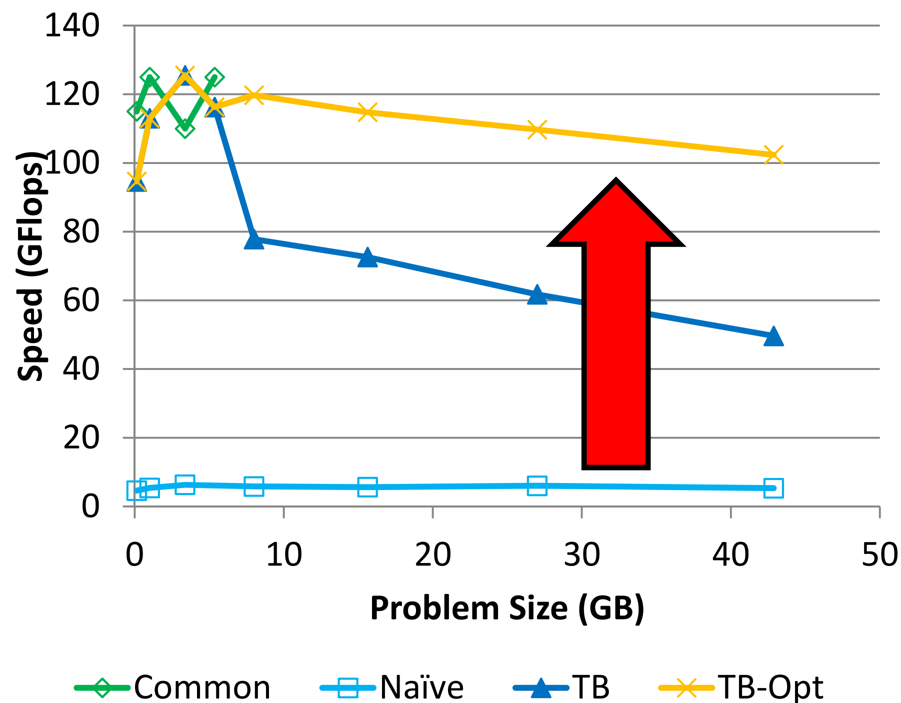
- ブロック用配列の「シフト利用」により、配列サイズ削減
  [Jin,Endo,Matsuoka 13]



- GPUカーネル内でレジスタを利用し2step分計算
  – つまり、二重のテンポラルブロッキング

最適化TB版の性能

3D 7point stencil on a K20X GPU



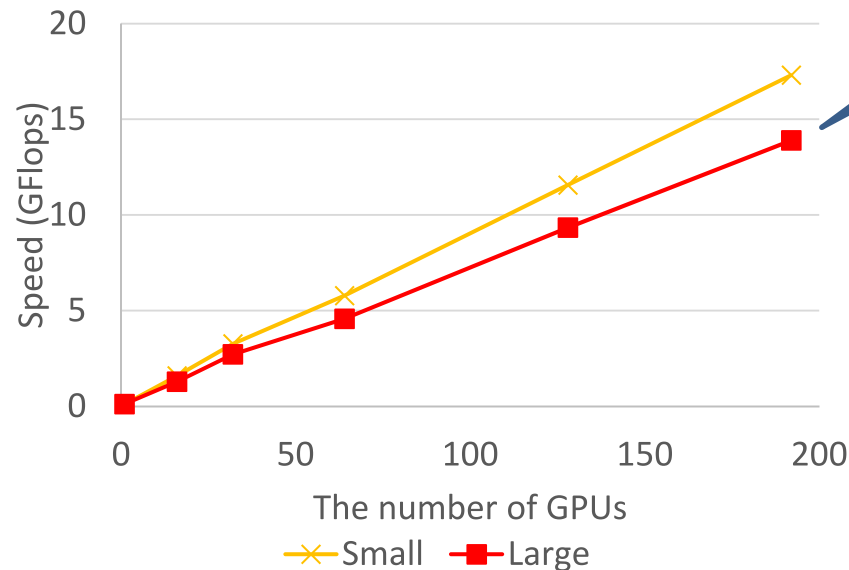
- 最適化TB版では、デバイスメモリ容量の7倍 の問題サイズ(体積)を、たった~20%のオーバヘッドで計算 !!

マルチGPUでの性能 [Cluster13]

- GPU 100個使うときの容量限界は 6GB x 100 = 600GB

→ テンポラルブロッキングを使えばそれを超えられる

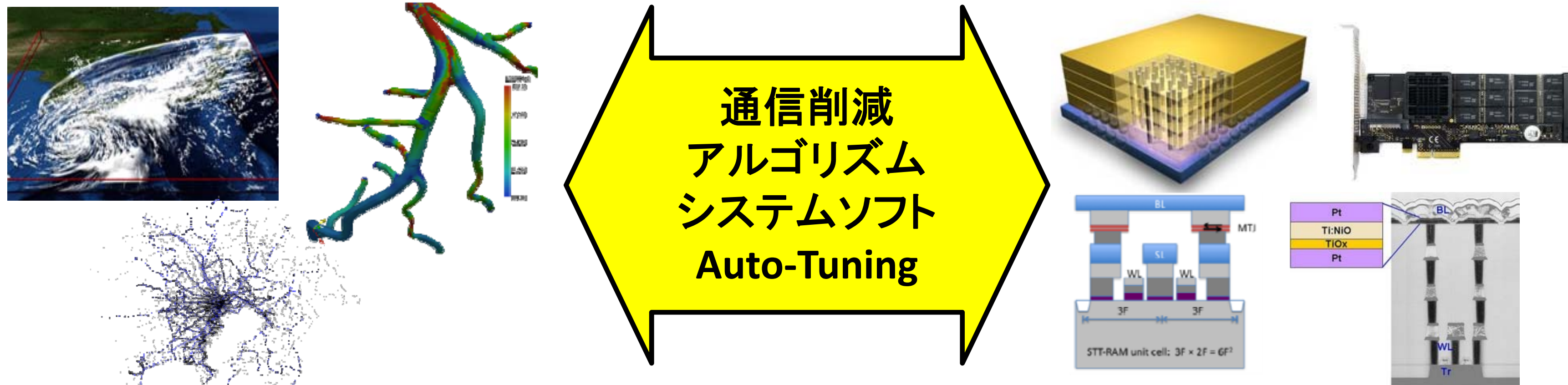
TSUBAME2.5上のWeak scalability
Small: 3.4GB per GPU
Large: 16GB per GPU (>6GB!)
※ ノードあたり1GPU利用



問題サイズ 3TBで14TFlops

192GPUでも良好なスケーリング！！
→ TSUBAME2の4000GPUで実験予定

- メモリウォール問題により、高性能と大問題サイズは今後ますますトレードオフに
- アルゴリズム・システムソフトウェア・アーキテクチャにまたがったco-designが必要　⇒ 複雑化するアーキ・アルゴリズム上の自動チューニングへの期待



  - HMC・NVRAMなどアーキテクチャ分野への要件のフィードバック
  - 局所性向上の自動化・パッケージ化によりアプリ・シミュレーション分野へのフィードバック
  - TSUBAME3.0などポストペタスパコンのデザインへのフィードバック

全体の終わりに

- 高性能計算の概観をかけ足でおこなった
- 取り上げられなかった点
  - キャッシュメモリとそれに応じた最適化
  - ビッグデータへの対応
    - 並列分散ファイルシステム、MapReduceプログラミングモデル・・・
  - PGAS (partitioned global address space)

- プログラミング言語と高性能計算の分野間で「話が通じる」一助となれば