第9回
高速化チューニングとその関連技術2

渡辺宙志

東京大学物性研究所

Outline

1. 計算機の仕組み
2. プロファイラの使い方
3. メモリアクセス最適化
4. CPUチューニング
5. 並列化

計算機の仕組み

計算機の仕組みと高速化

高速化とは何か？

アルゴリズムを変えず、実装方法によって実行時間を短くする方法

→ アルゴリズムレベルの最適化は終了していることを想定

遅いアルゴリズムを実装で高速化しても無意味
十分にアルゴリズムレベルで最適化が済んでから高速化

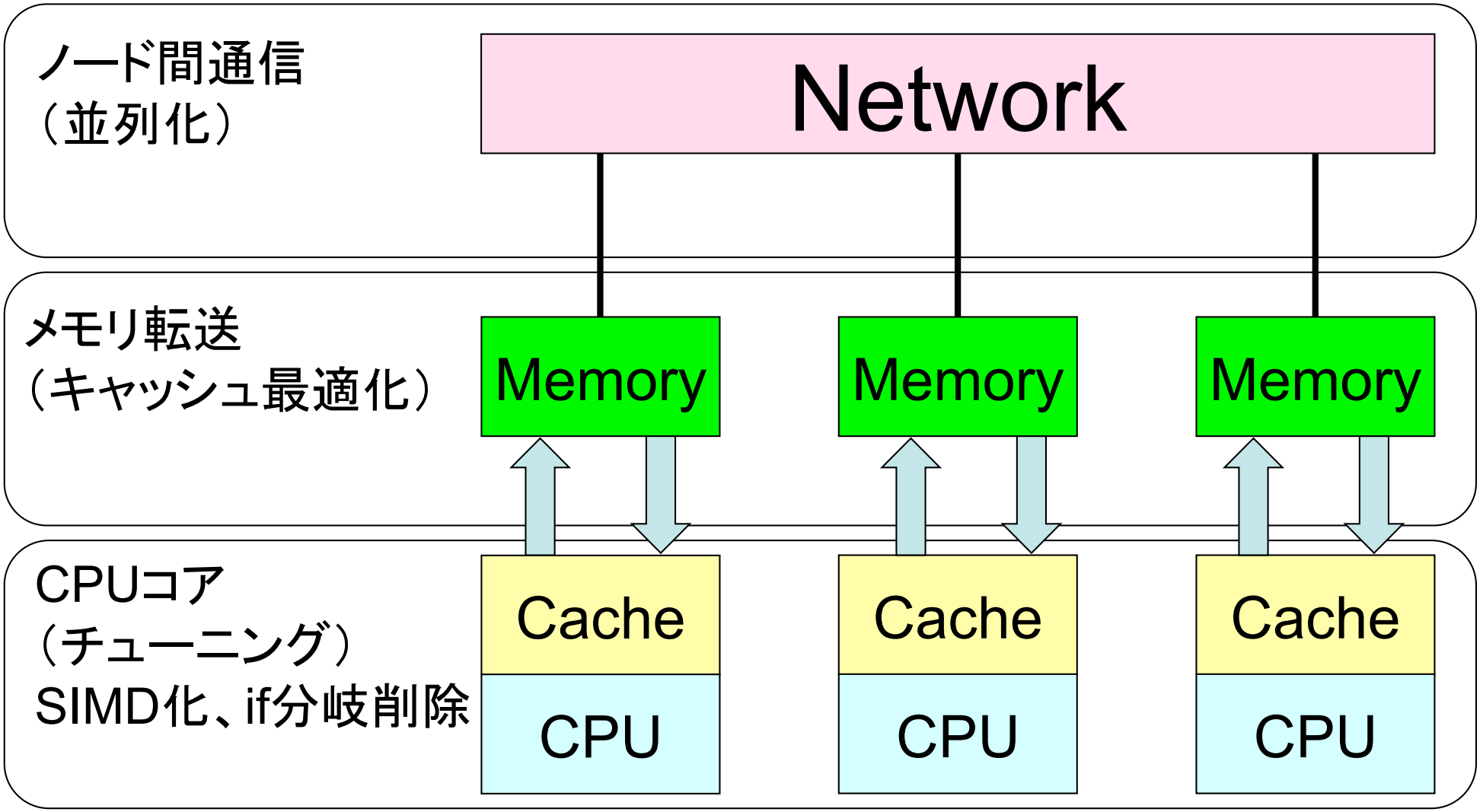
実装レベルの高速化とは何か？

「コンパイラや計算機にやさしいコードを書く事」

※やさしい＝理解しやすい、向いている

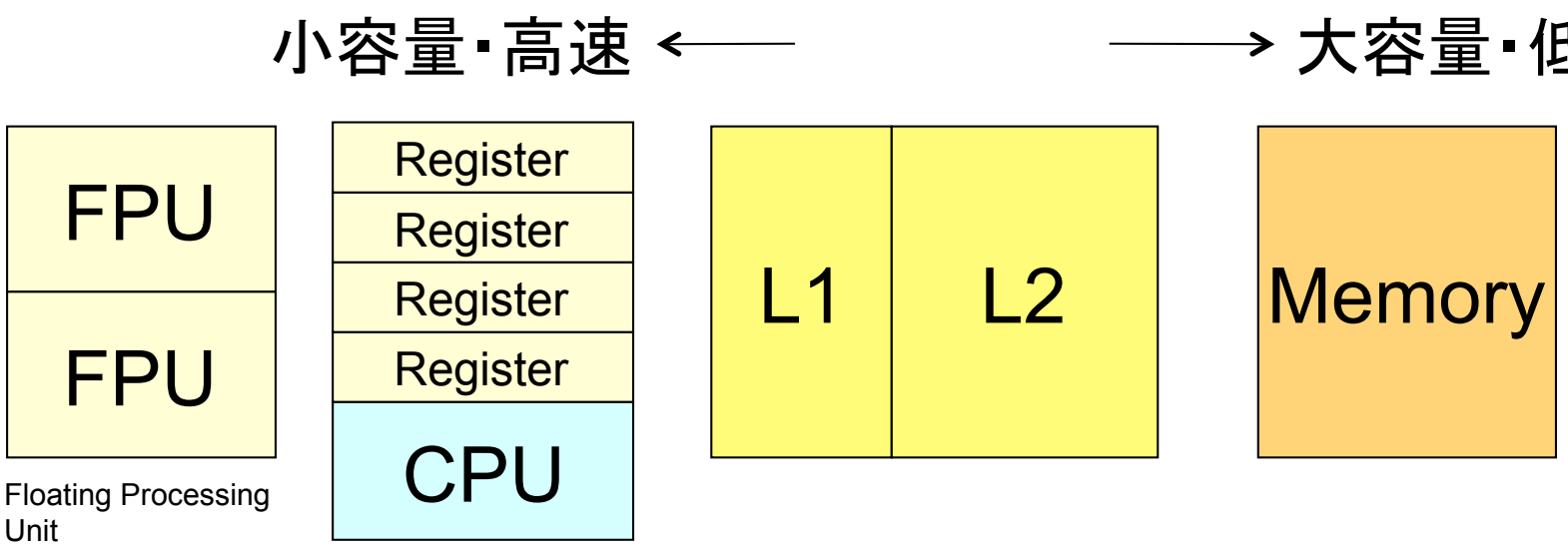
そのためには、計算機の仕組みを理解しておかなければならない

プログラム階層



この中で最も重要なのはメモリ転送

記憶階層



- 計算はレジスタ上で行う
- レジスタにのせられた情報はFPUに渡され、レジスタにかえってくる
- メモリアクセスで大事なのはバンド幅とレイテンシ

レイテンシ

CPUにデータを要求されてから、レジスタに届くまでのサイクル数
L1キャッシュで数サイクル、L2で10～20サイクル、主記憶は～数百サイクル

バンド幅

CPUにデータが届き始めてからのデータ転送速度
絶対値(GB/s)よりも、計算速度との比(B/F)の方が良く使われる

Byte / Flop

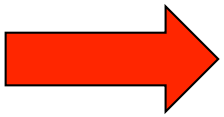
Byte/Flop

データ転送速度(Byte/s) と計算速度 (FLOPS)の比。略してB/F。
値が大きいほど、計算速度に比べたデータ転送能力が高い。

典型的には B/F = 0.1～0.5

B/Fの意味

一つの倍精度実数(64bit)の大きさ = 8 Byte
計算するには、最低二つの倍精度実数が必要 = 16 Byte
最低でも一つの倍精度実数を書き戻す必要がある = 8 Byte.
例えば一回掛け算をして、その結果をメモリに書き戻すと、24Byteの読み書きが発生
B/F = 0.5 のマシンでは、24バイトの読み書きの間に48回計算ができる



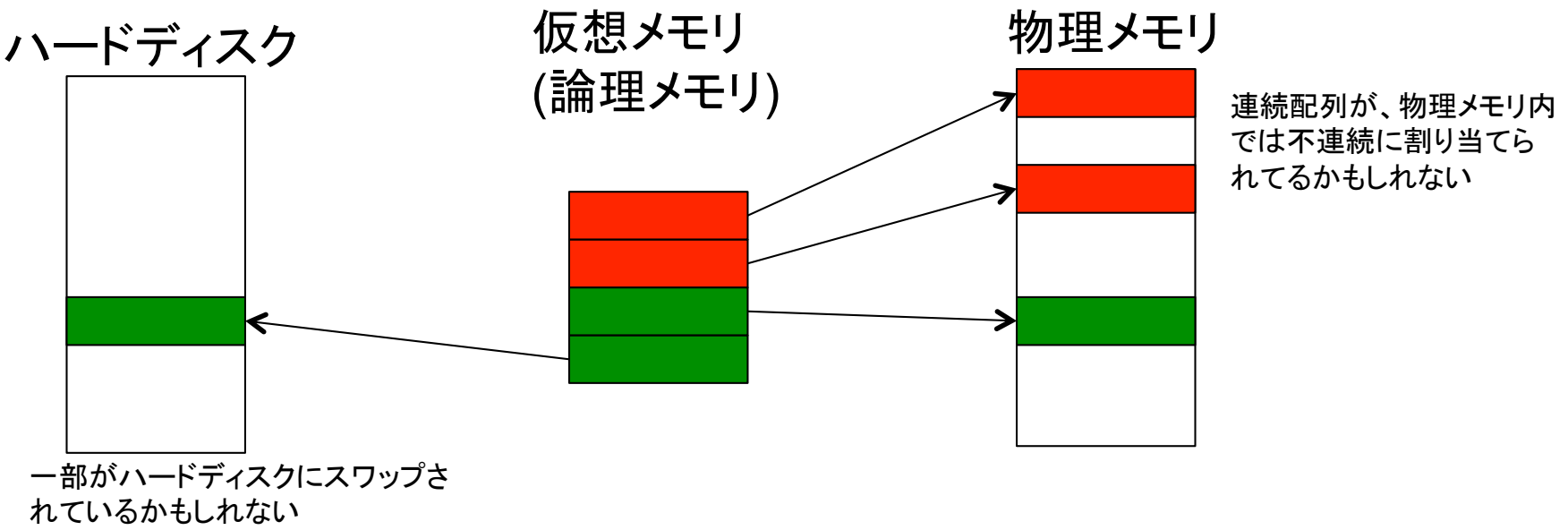
B/Fが0.5の計算機では、C = A*Bという単純な計算をくりかえすと
ピーク性能の2%ほどしか出せず、ほとんど時間CPUが遊ぶ

キャッシュ効率が性能に直結する

仮想メモリとページング (1/2)

仮想メモリとは

物理メモリと論理メモリをわけ、ページという単位で管理するメモリ管理方法



仮想メモリを使う理由

・不連続なメモリ空間を論理的には連続に見せることができる
　仮想メモリなしでは、メモリに余裕があっても連続領域が取れない場合にメモリ割り当てができない

・スワッピングが可能になる
　記憶容量としてハードディスクも使えるため、物理メモリより広い論理　メモリ空間が取れる。

論理メモリと実メモリの対応が書いてあるのがTLB (Translation Lookaside Buffer)
ページサイズを大きく取るのがラージページ

仮想メモリとページング (2/2)

数値計算で何が問題になるか？

F90のallocateや、Cでnew、mallocされた時点では物理メモリの
割り当てがされていない場合がある

```
real*8, allocatable :: work(:)
allocate (work(10000))
do i=1, 10000
  work(i) = i
end do
```

← この時点では予約だけされて、まだ物理アドレスが割り当てられない

← はじめて触った時点で、アドレスがページ単位で割り当てられる
(First touch の原則)

よくある問題:
- 最初に馬鹿でかい配列を動的に確保しているプログラムの初期化がすごく遅い
- 地球シミュレータから別の計算機への移植で問題になることが多い
- OSがLinuxである京でも発生する

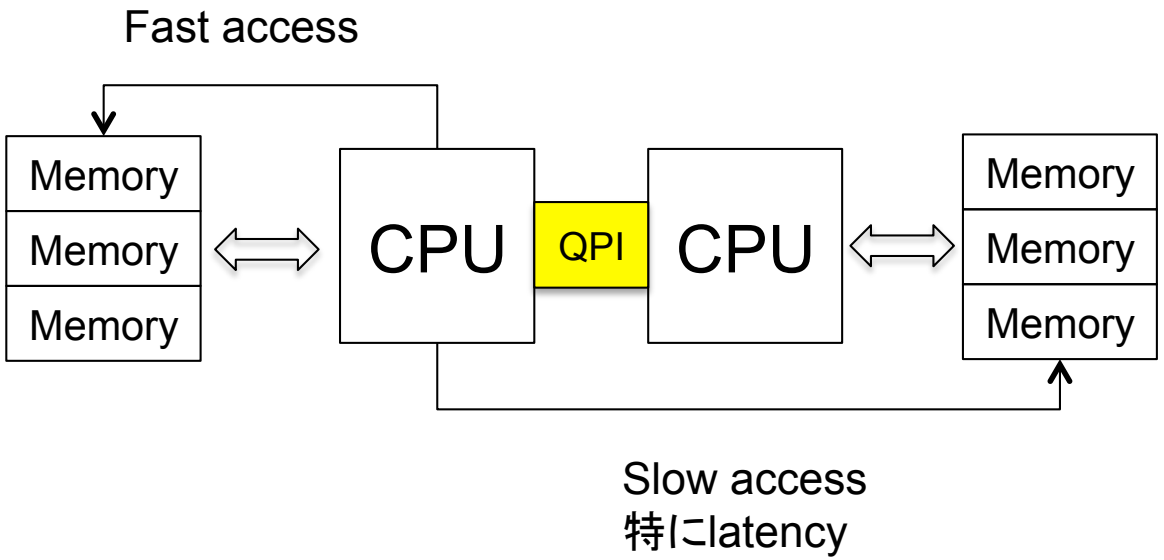
解決策:
- メモリを確保した直後、ページ単位で全体を触っておく
- メモリを静的に確保する(?)

まぁ、そういうこともあると覚えておいてください

NUMA (1/2)

NUMAとは

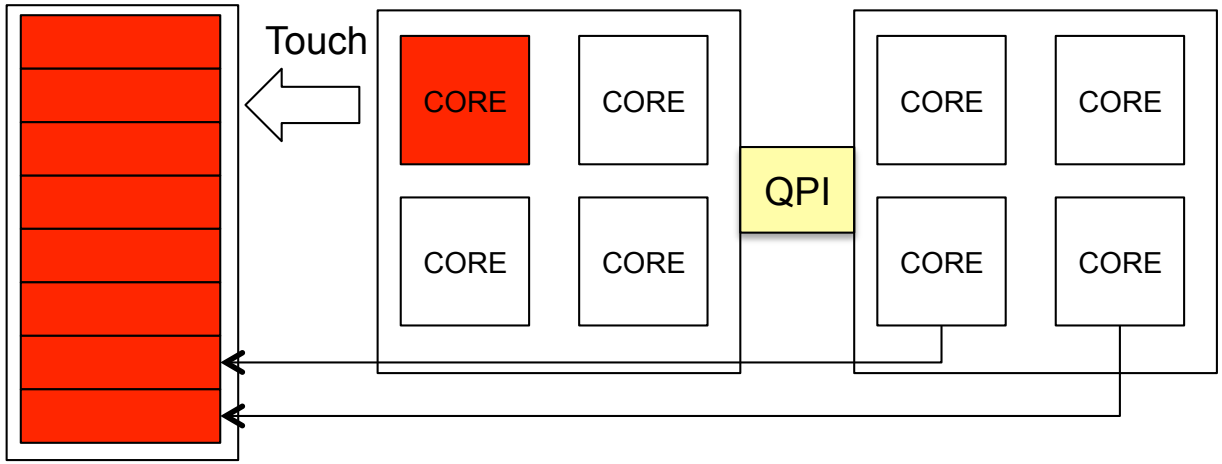
非対称メモリアクセス(Non-Uniform Memory Access)の略
CPUにつながっている物理メモリに近いメモリと遠いメモリの区別がある
京コンピュータはNUMAではないが、物性研System BはNUMA



NUMA (2/2)

数値計算で何が問題になるか？

OpenMPでCPUをまたぐ並列化をする際、初期化を適当にやると
作業領域がコアから遠い物理メモリに割り当てられてしまう



OpenMPのスレッドを作る前に配列を初期化
→root プロセスの近くにメモリ割り当て
その後OpenMPでスレッド並列
→遠いメモリをアクセス(遅い)

解決策:
スレッドに対応する物理コアに近いところにメモリが割り当てられるようにする
→ スレッドごとに配列を用意した上、OpenMPでスレッド並列化した後にtouch

詳しくはNUMA最適化でググってください

CPUアーキテクチャ (1/2)

RISCと CISC

世の中のCPUアーキテクチャは大別して RISC型とCISC型の二種類

CISC:
- 複雑な命令セット
- if分岐などに強い
- Intel Xeon, AMD Opteronなど。

RISC:
- 単純な命令セット
- 行列演算などに強い
- IBM POWER, SPARCなど (「京」はSPARC VIIIfx)

CISCが内部的にRISC的命令に変換していたりするので現在はあまり意味のある区別ではない
要するに「Intel系か、それ以外か」という区別

マルチコアとSIMD化

最近のCPUはほぼマルチコア化とSIMD化で性能をかせいでいる
- マルチコアで性能を出すには、キャッシュとメモリの構造を把握する必要あり
- 京(8コア) FX10 (16コア) IBM POWER6 (32コア)、SGI Altix ICE(4コア*2)
- SIMDについては後述
- そのうちメニーコア化？ (Xeon Phi)

CPUアーキテクチャ (2/2)

ゲーム機とCPU

第六世代
DC　SH-4
PS2　MIPS (Emotion Engine)
GC　IBM PowerPC カスタム (Gekko)
Xbox Intel Celeron (PenIII ベース)

第七世代
Wii　IBM PowerPC カスタム
Xbox 360 IBM PowerPC カスタム
PS3　IBM Cell 3.2

第八世代・・・？

レイテンシとパイプライン (1/3)

レイテンシとは

命令が発行されてから、値が帰ってくるまでのクロック数

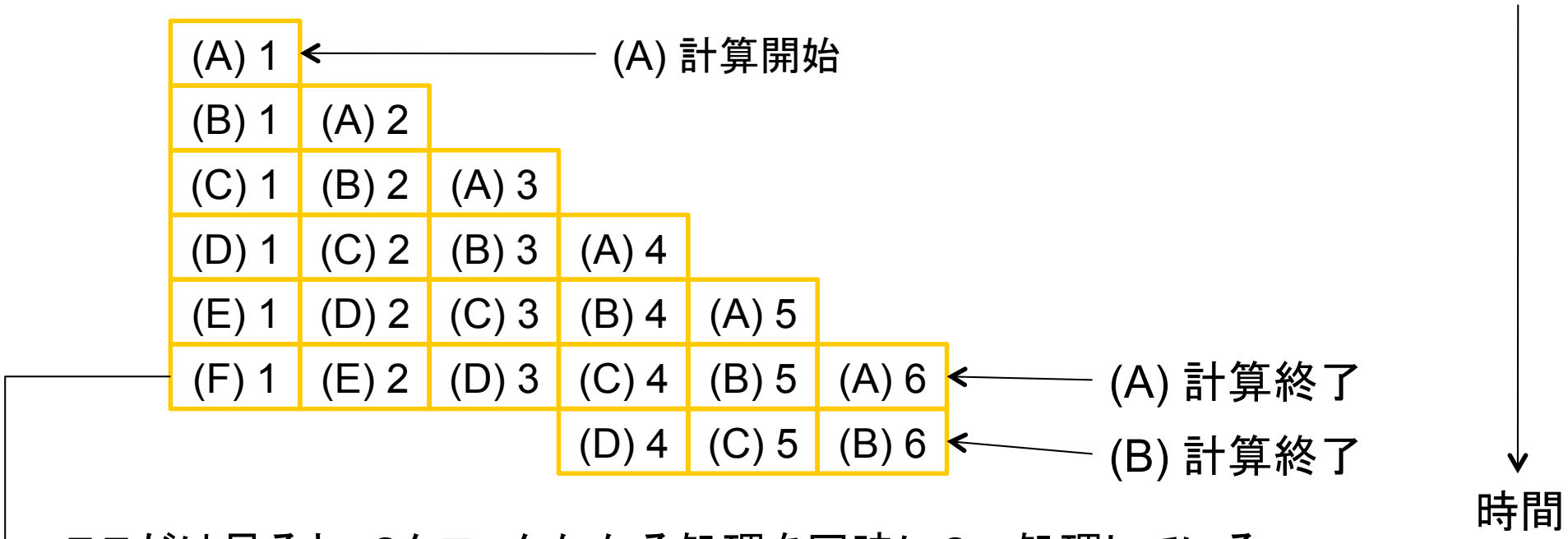
```
fadd    fp2,fp0,fp1     # fp2 ← fp0 + fp1
fadd    fp4,fp2,fp3     # fp4 ← fp2 + fp3
```

レイテンシとパイプライン (2/3)

パイプラインとは

浮動小数点の加算、減算、掛け算といった計算を流れ作業で行う仕組み

```
fadd    fp2,fp0,fp1  (A)
fadd    fp5,fp3,fp4  (B)
fadd    fp8,fp3,fp7  (C)
```

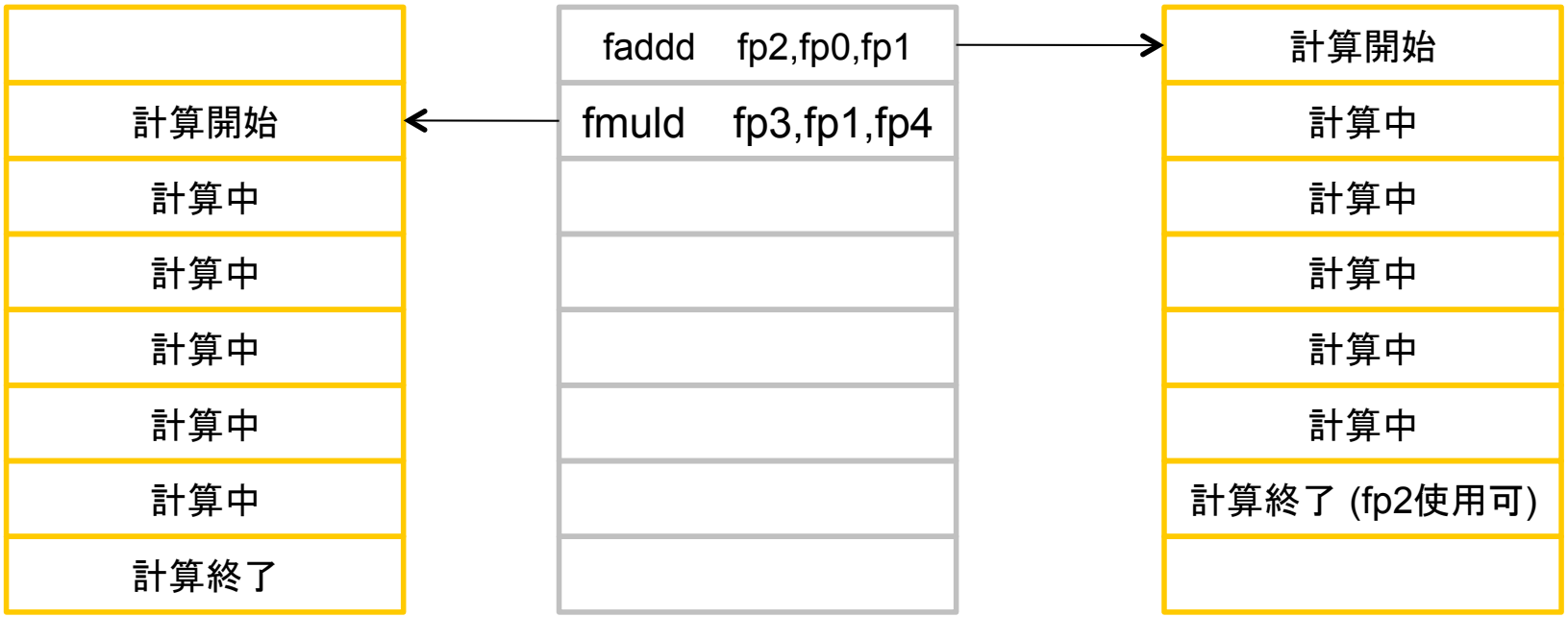


ここだけ見ると、6クロックかかる処理を同時に6つ処理している
→ 一クロックで一回の計算ができているように見える (**レイテンシの隠蔽**)
これを「スループット が1」と呼ぶ

レイテンシとパイプライン (3/3)

パイプライン数

一般的にはパイプラインは2本 ※ Load/Storeも出せる場合は、IPCは最大4



実質的に、1クロックで二つの計算が可能

積和(差)命令 fmaddd, fmsubd

積和、積差

積和　　X = A*B+C
積差　　Y = A*B-C

アセンブラでは

```
fmaddd    fp0,fp1,fp2,fp3        #   fp0 ← fp1*fp2+fp3
fmsubd    fp0,fp1,fp2,fp3        #   fp0 ← fp1*fp2+fp3
```

乗算一つ、加減算一つ、あわせて二つが(実質)1クロックで実行
そのパイプラインが二つあるので、あわせて1クロックで4つの演算

SIMD

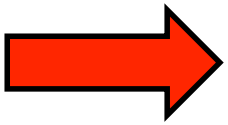
SIMDとは

Single Instruction Multiple Data の略。通称シムド、シムディー
ベクトル命令のようなもの
独立な同じ計算を同時に行う

積和、積差のSIMD

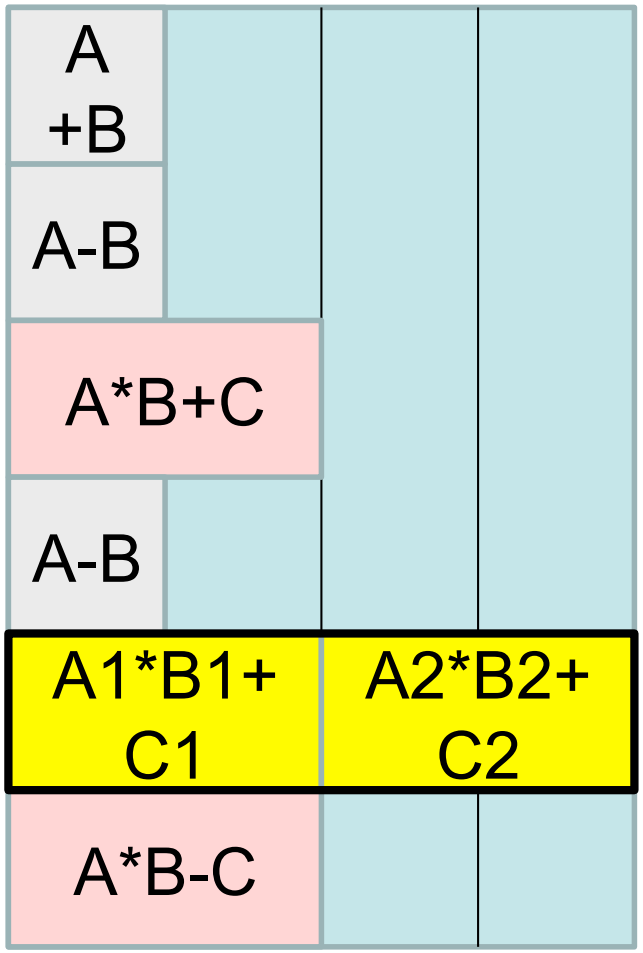
fmadd,s　X1 = A1*B1+C1 , X2 = A2*B2+C2 を同時に実行
fmsub,s　X1 = A1*B1-C1 , X2 = A2*B2-C2 を同時に実行

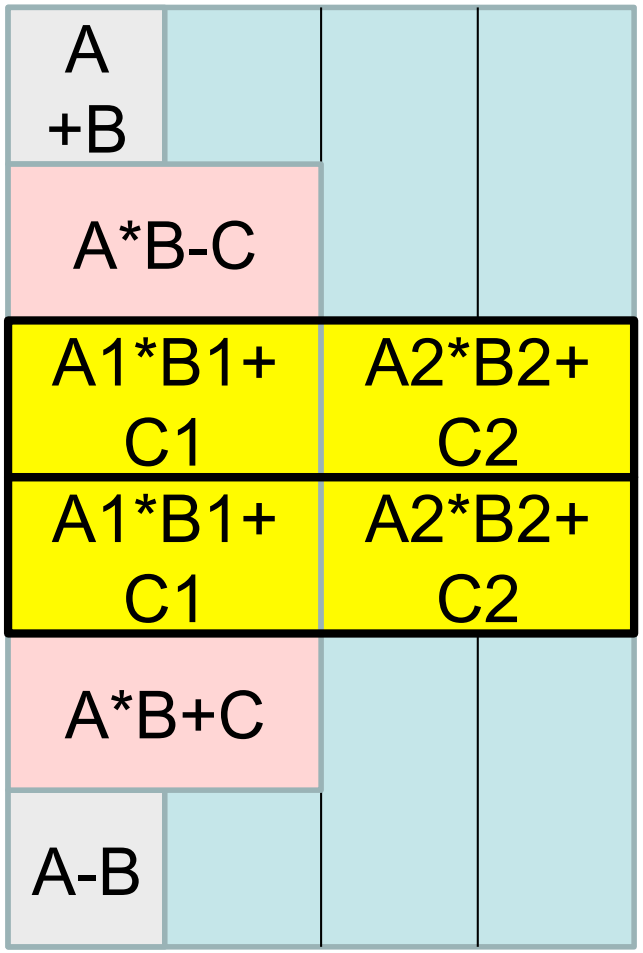
fmadd,sは、一つで4個の浮動小数点演算を行う。
「京」はこれが同時に二つ走る
4 * 2 * 2 GHz = 16GFlops (ピーク性能)



アセンブリにfmaddd,s fmsubd,sが
ずらずら並んでいなければダメ

パイプラインのイメージ

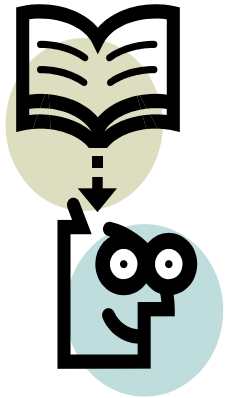




- 幅が4、長さが6のベルトコンベアがある
- それぞれには、大きさ1、2、4の荷物を流せるが、同時に一つしか置けない
- ピーク性能比＝ベルトコンベアのカバレージ
- ピーク性能を出す＝なるべくおおきさ4の荷物を隙間無く流す
  (Intel系の命令パイプラインはやや異なる)

計算機の仕組みのまとめ

- 期待する性能がでない時、どこが問題かを調べるのに
  計算機の知識が必要 (特にメモリアクセス)
- 積和のパイプラインを持つCPUで性能を出すためには、
  独立な積和演算がたくさん必要
- SIMDを持つCPUで性能を出すためには、
  独立なSIMD命令がたくさん必要
  → 「京」では、独立な積和のSIMD命令がたくさん必要



アセンブリを読みましょう
変にプロファイラと格闘するより直接的です

(-S 付きでコンパイルし、fmadd,sなどでgrep)

プロファイラの使い方

プログラムのホットスポット

ホットスポットとは

プログラムの実行において、もっとも時間がかかっている場所
(分子動力学計算では力の計算)

多くの場合、一部のルーチンが計算時間の大半を占める
(80:20の法則)

チューニングの方針

まずホットスポットを探す
チューニング前に、どの程度高速化できるはずかを見積もる
チューニングは、ホットスポットのみに注力する
ホットスポットでないルーチンの最適化は行わない
→ 高速化、最適化はバグの温床となるため
　ホットスポットでないルーチンは、速度より可読性を重視

プロファイラ

プロファイラとは

プログラムの実行プロファイルを調べるツール
Sampler型とイベント取得型がある

Sampler型

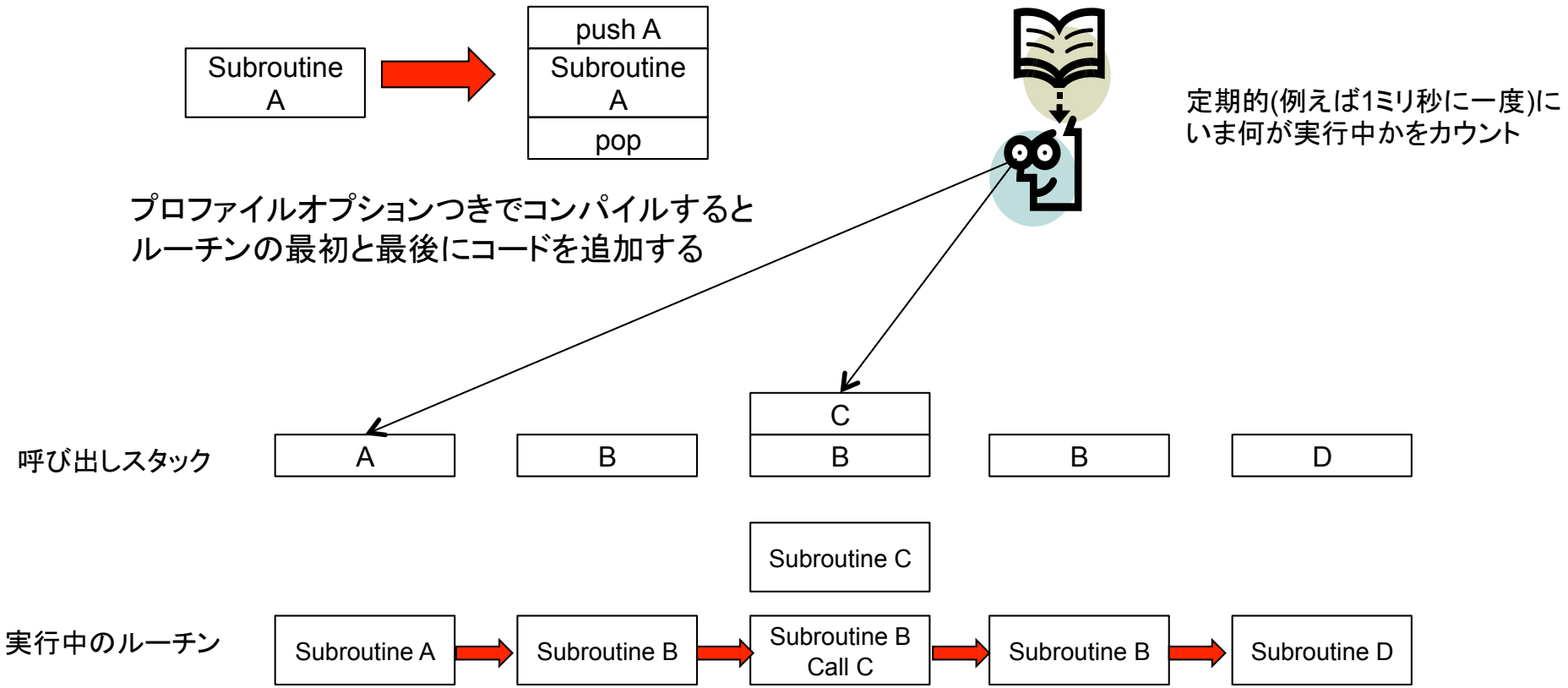
どのルーチンがどれだけの計算時間を使っているかを調べる
ルーチン単位で調査
ホットスポットを探すのに利用

イベント取得型

どんな命令が発効され、どこでCPUが待っているかを調べる
範囲単位で調査 (プログラム全体、ユーザ指定)
高速化の指針を探るのに利用

プロファイラ(Sampler型)

実行中、一定間隔で「いまどこを実行中か」を調べ、実行時間は
その出現回数に比例すると仮定する



gprof

gprofとは

広く使われるプロファイラ(sampler)
(Macでは使えないようです)

使い方

```
$ gcc -pg test.cc
$ ./a.out
$ ls
a.out gmon.out test.cc
$ gprof a.out gmon.out
```

出力

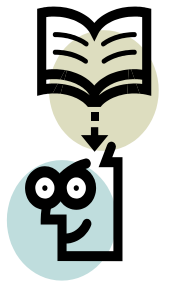
とりあえずEach % timeだけ見ればいいです

Flat profile:

Each sample counts as 0.01 seconds.

サンプリングレートも少しだけ気にすること

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
100.57	0.93	0.93	1	925.26	925.26	matmat()
0.00	0.93	0.00	1	0.00	0.00	global constructors keyed to A
0.00	0.93	0.00	1	0.00	0.00	__static_initialization_and_destruction_0(int, int)
0.00	0.93	0.00	1	0.00	0.00	init()
0.00	0.93	0.00	1	0.00	0.00	matvec()
0.00	0.93	0.00	1	0.00	0.00	vecvec()



University of Tokyo

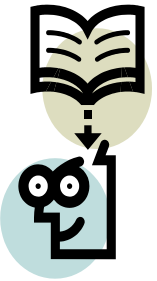
結果の解釈 (Sampler型)

一部のルーチンが80%以上の計算時間を占めている
→そのルーチンがホットスポットなので、高速化を試みる

多数のルーチンが計算時間をほぼ均等に使っている
→最適化をあきらめる

あきらめたらそこで試合終了じゃないんですか？

世の中あきらめも肝心です



※最適化は、常に費用対効果を考えること

プロファイラ(イベント取得型)

Hardware Counter

CPUがイベントをカウントする機能を持っている時に使える(京など)
イベントをサンプリングするプロファイラもある (Intel VTuneなど)

取得可能な主なイベント:

- 整数演算
- 浮動小数点演算
- キャッシュミス
- バリア待ち
- 演算待ち

←こっちに気を取られがちだが

←個人的にはこっちが重要だと思う

プロファイラの使い方

システム依存
京では、カウントするイベントの種類を指定
プログラムの再コンパイルは不必要
カウンタにより取れるイベントが決まっている
→同じカウンタに割り当てられたイベントが知りたい場合、複数回実行する必要がある

Profile結果の解釈 (HW Counter)

バリア同期待ち

OpenMPのスレッド間のロードインバランスが原因
自動並列化を使ったりするとよく発生
対処：自分でOpenMPを使ってちゃんと考えて書く(それができれば苦労はしないが)

キャッシュ待ち

浮動小数点キャッシュ待ち
対処：メモリ配置の最適化 (ブロック化、連続アクセス、パディング・・・)
ただし、本質的に演算が軽い時には対処が難しい

演算待ち

浮動小数点演算待ち
A=B+C
D=A*E ←この演算は、前の演算が終わらないと実行できない
対処：アルゴリズムの見直し (それができれば略)

SIMD化率が低い

対処できる場合もあるが、あきらめた方がはやいと思う
それでも対処したい人へ：
ループカウンタ間の依存関係を減らしてsoftware pipeliningを促進

メモリアクセス最適化

メモリアクセス最適化のコツ

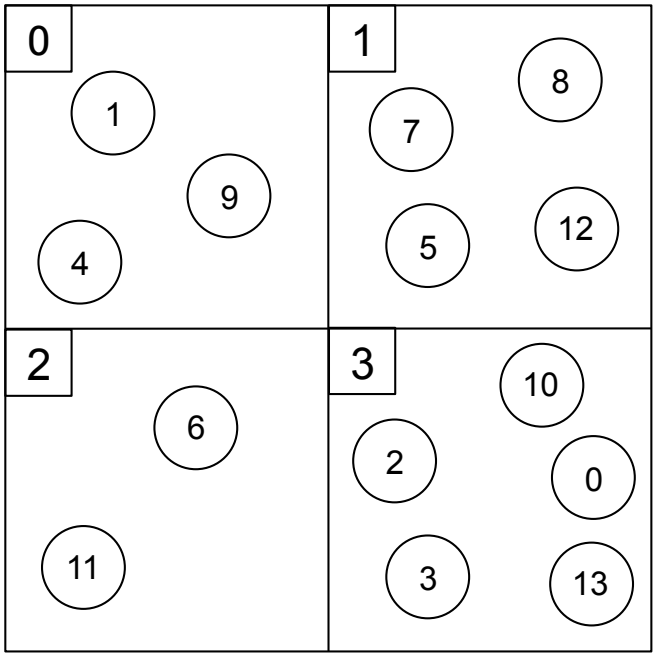
計算量を増やしてでもメモリアクセスを減らす

メモリ最適化1 セル情報の一次元実装 (1/2)

セル情報

相互作用粒子の探索で空間をセルに分割し、それぞれに登録する
ナイーブな実装→ 多次元配列

```
int GridParticleNumber[4];  どのセルに何個粒子が存在するか
int GridParticleIndex[4][10]; セルに入っている粒子番号
```



GridParticleIndexの中身はほとんど空

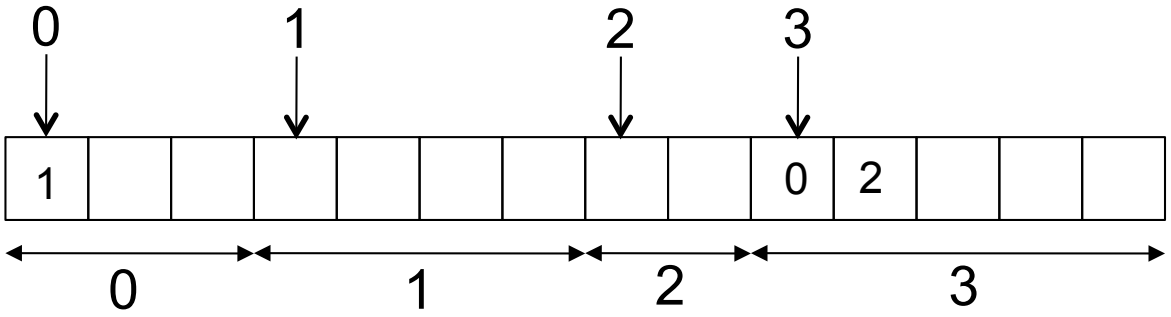
1	4	9							
7	5	8	12						
6	11								
0	2	3	10	13					

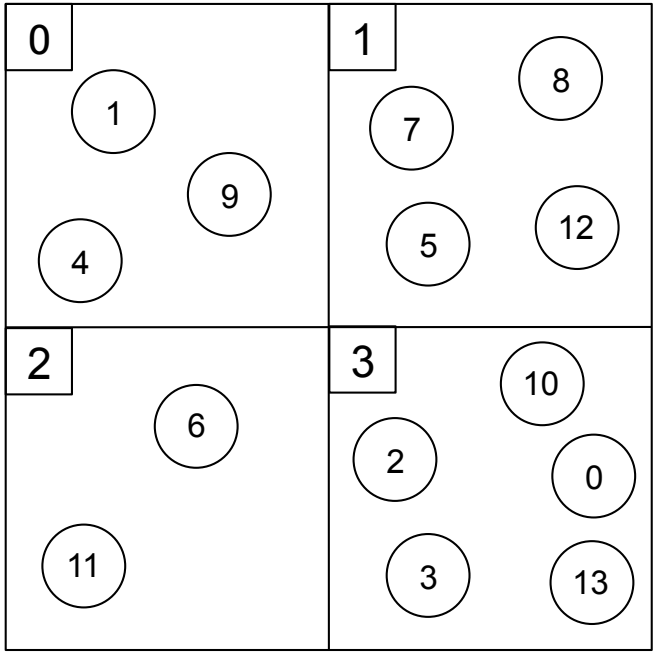
広いメモリ空間の一部しか使っていない
→ キャッシュミスの多発

メモリ最適化1 セル情報の一次元実装 (2/2)

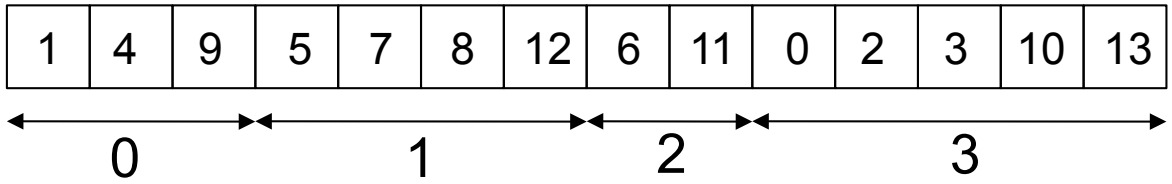
一次元実装

1. 粒子数と同じサイズの配列を用意する
2. どのセルに何個粒子が入る予定かを調べる
3. セルの先頭位置にポインタを置く
4. 粒子を配列にいれるたびにポインタをずらす
5. 全ての粒子について上記の操作をしたら完成





完成した配列 (所属セル番号が主キー、粒子番号が副キーのソート)



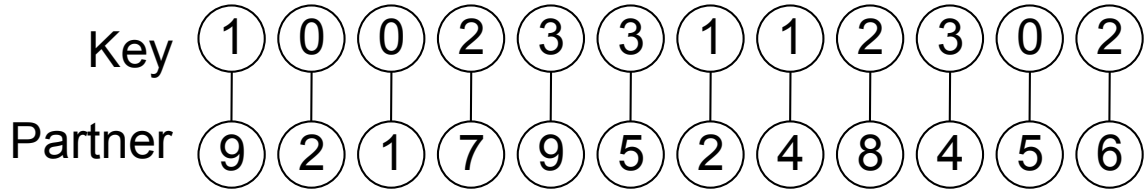
メモリを密に使っている (キャッシュ効率の向上)

メモリ最適化2 相互作用ペアソート (1/2)

力の計算とペアリスト

力の計算はペアごとに行う
相互作用範囲内にあるペアは配列に格納
ペアの番号の若い方をkey粒子、残りをpartner粒子と呼ぶ

得られた相互作用ペア



相互作用ペアの配列表現

Key	1	0	0	2	3	3	1	1	2	3	0	2
Partner	9	2	1	7	9	5	2	4	8	4	5	6

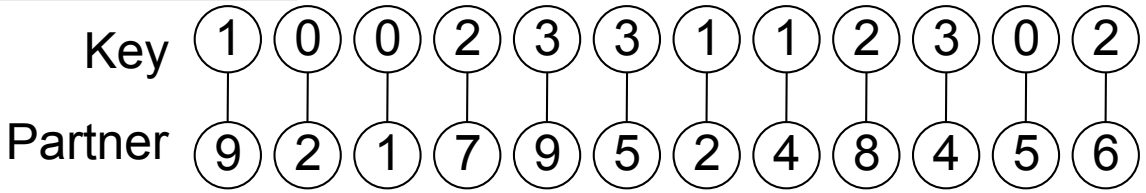
このまま計算すると

2個の粒子の座標を取得 (48Byte) 計算した運動量の書き戻し (48Byte)
力の計算が50演算とすると、B/F～2.0を要求 (※キャッシュを無視している)

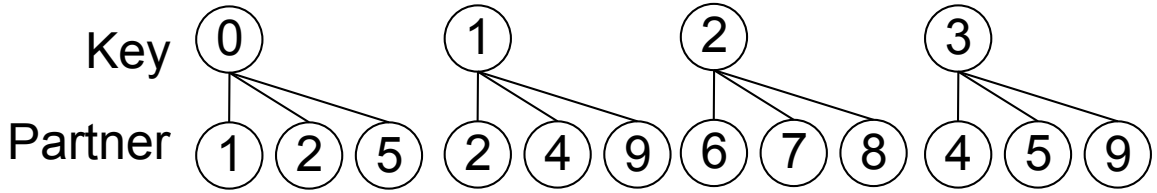
メモリ最適化2 相互作用ペアソート (2/2)

相互作用相手でソート

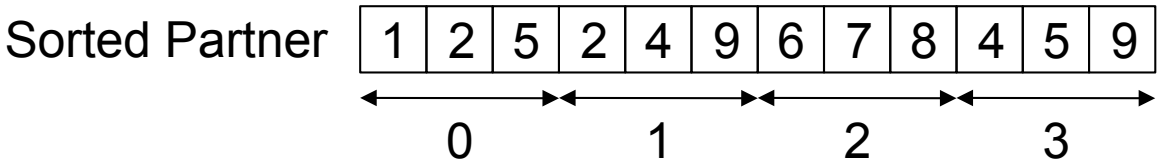
相互作用ペア



Key粒子でソート



配列表現

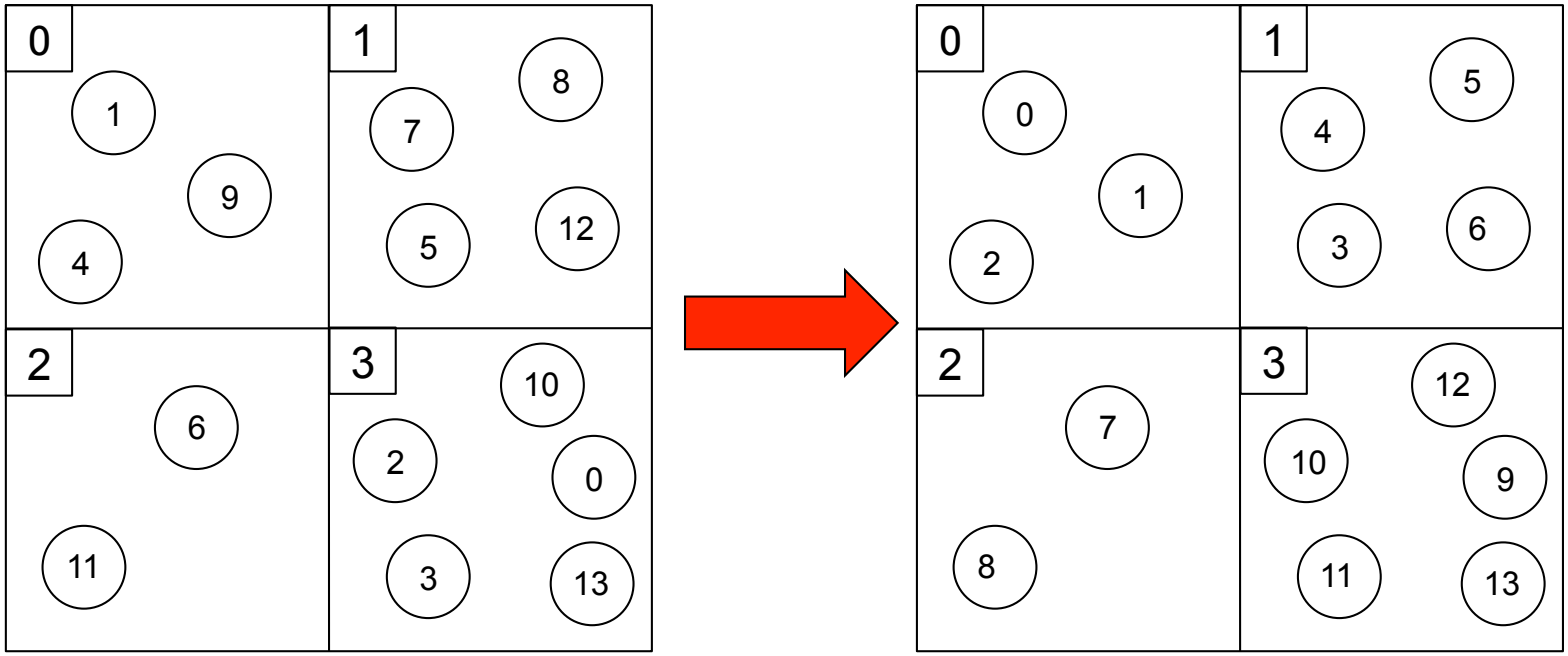


Key粒子の情報がレジスタにのる
→ 読み込み、書き込みがPartner粒子のみ
→ メモリアクセスが半分に

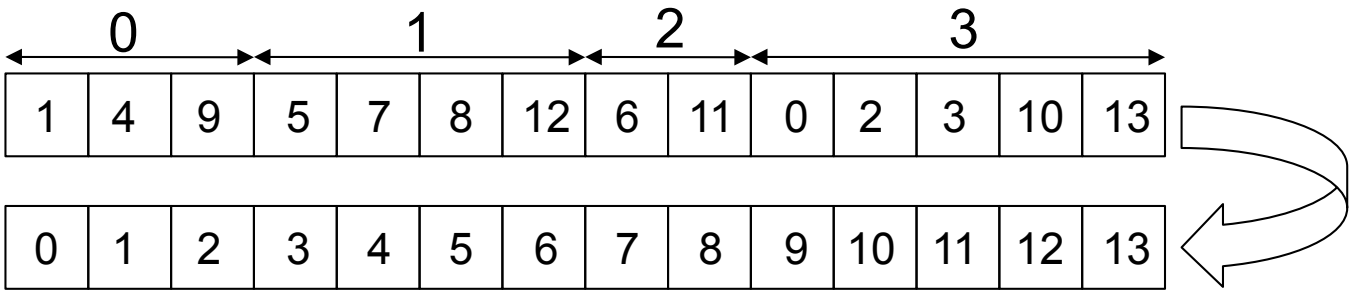
メモリ最適化3 空間ソート (1/2)

空間ソート

時間発展を続けると、空間的には近い粒子が、メモリ上では
遠くに保存されている状態になる → ソート

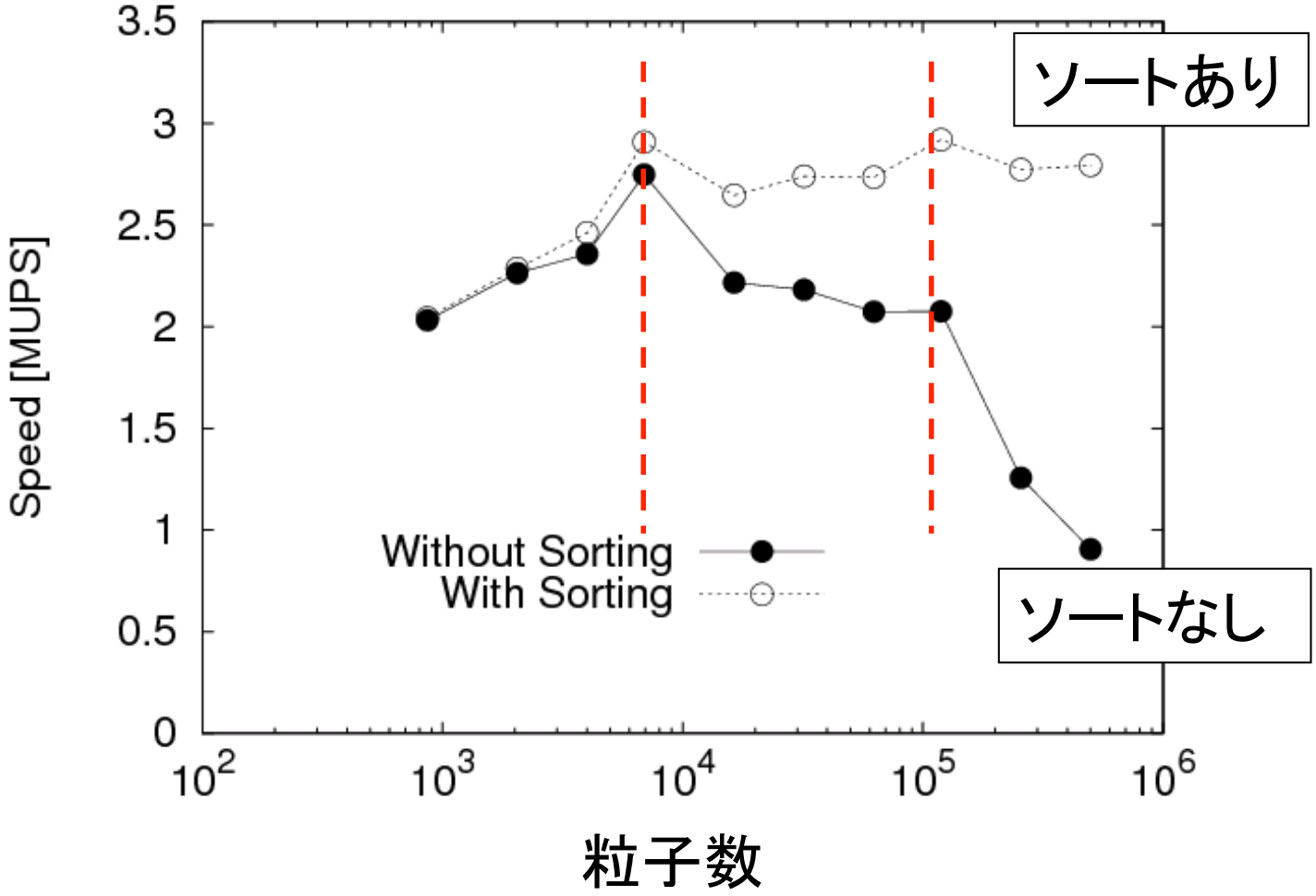


ソートのやりかたはセル情報の一次元実装と同じ



メモリ最適化3 空間ソート (2/2)

空間ソートの効果



ソートなし：粒子数がキャッシュサイズをあふれる度に性能が劣化する
ソートあり：性能が粒子数に依存しない

メモリ最適化3 作用反作用(1/2)

作用反作用

ペアごとに力積を計算、作用反作用により、ペアの両方の力積を更新
Partner粒子 (j粒子)の力積の和をまとめて書き戻し
→ Key粒子(i粒子)の運動量の書き戻しは無視できる
Partner粒子の力積は毎回書き戻さなければならない

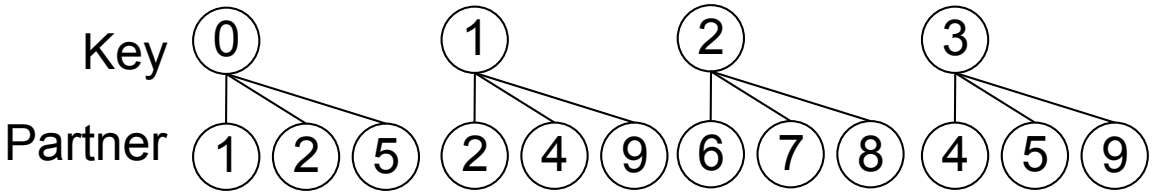
```
DO i=1,N
  ptemp = 0
  DO j in Pair(I)
    f = CalcForce(I,J)   <-------- 力の計算
    ptemp = ptemp + f*dt <-------- i粒子の力積 (一時変数に積算)
    p[j] = p[j] - f *dt  <-------- j粒子の力積 (書き戻し)
  ENDDO
  p[i] = p[i] +ptemp     <-------- i粒子の力積をまとめて書き戻し
ENDDO
```

何が問題か？

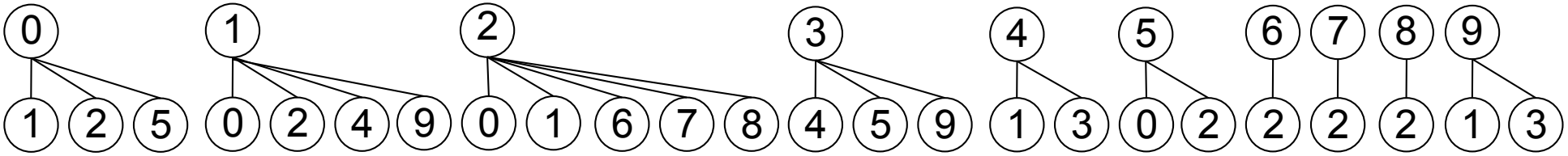
j粒子の書き戻しは間接参照
j粒子の依存関係をコンパイラは判断できない

メモリ最適化3 作用反作用(2/2)

作用反作用をあきらめる



↑このペアリストから、↓このペアリストを作る




```
DO i=1,N
  ptemp = 0
  DO j in Pair(I)
    f = CalcForce(I,J)
    ptemp = ptemp + f*dt
    //p[j] = p[j] - f *dt
  ENDDO
  p[i] = p[i] +ptemp
ENDDO
```

←―― j粒子の力積の書き戻しをしない

計算量が二倍になるかわりにメモリへの書き込みが消える
ループ間の依存関係が消える

メモリアクセス最適化のまとめ

メモリアクセス最適化とは

計算量を犠牲にメモリアクセスを減らす事
使うデータをなるべくキャッシュ、レジスタにのせる
→ソートが有効であることが多い
計算サイズの増加で性能が劣化しない
→キャッシュを効率的に使えている

メモリアクセス最適化の効果

一般に大きい
不適切なメモリ管理をしていると、100倍以上遅くなる場合がある
→ 100倍以上の高速化が可能である場合がある
アーキテクチャにあまり依存しない
→ PCでの最適化がスパコンでも効果を発揮

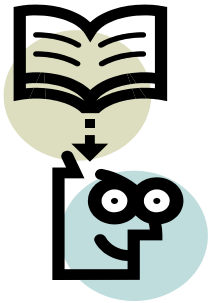
必要なデータがほぼキャッシュに載っており、CPUの
計算待ちがほとんどになって初めてCPUチューニングへ

CPUチューニング

やれることは少ない

CPUチューニング

PCで開発したコード、スパコンでの実行性能が
すごく悪いんですが・・・



それ、条件分岐が原因かもしれません。

開発ではIntel系のCPUを使うことが多く、スパコンは
別のRISCタイプアーキテクチャを使うことが多い

条件分岐コスト (1/2)

条件分岐なし

```
void
calcforce(void){
  for(int i=0;i<N-1;i++){
    for(int j=i+1;j<N;j++){
      const double dx = q[j][X] - q[i][X];
      const double dy = q[j][Y] - q[i][Y];
      const double dz = q[j][Z] - q[i][Z];
      const double r2 = (dx*dx + dy*dy + dz*dz);
      const double r6 = r2*r2*r2;
      double df = (24.0*r6-48.0)/(r6*r6*r2)*dt;
      p[i][X] += df*dx;
      p[i][Y] += df*dy;
      p[i][Z] += df*dz;
      p[j][X] -= df*dx;
      p[j][Y] -= df*dy;
      p[j][Z] -= df*dz;
    }
  }
}
```

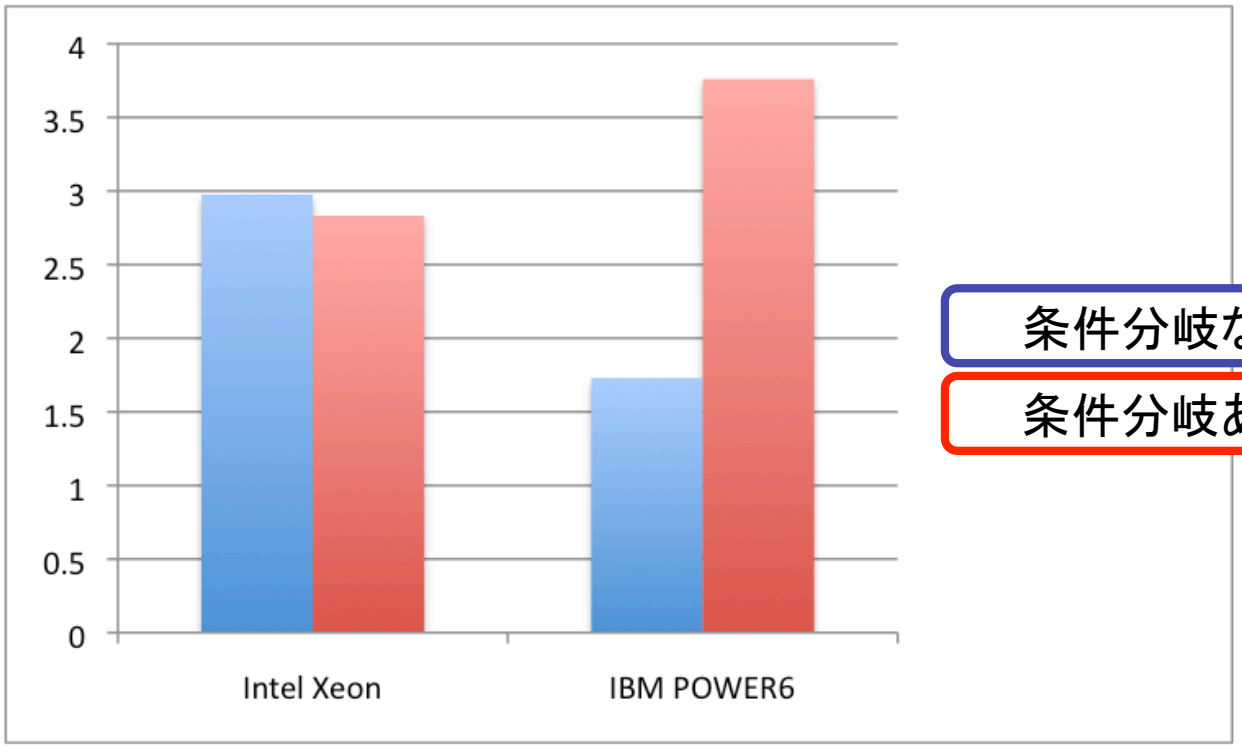
条件分岐あり

```
void
calcforce(void){
  for(int i=0;i<N-1;i++){
    for(int j=i+1;j<N;j++){
      const double dx = q[j][X] - q[i][X];
      const double dy = q[j][Y] - q[i][Y];
      const double dz = q[j][Z] - q[i][Z];
      const double r2 = (dx*dx + dy*dy + dz*dz);
      if (r2 > CUTOFF) continue;
      const double r6 = r2*r2*r2;
      double df = (24.0*r6-48.0)/(r6*r6*r2)*dt;
      p[i][X] += df*dx;
      p[i][Y] += df*dy;
      p[i][Z] += df*dz;
      p[j][X] -= df*dx;
      p[j][Y] -= df*dy;
      p[j][Z] -= df*dz;
    }
  }
}
```

※条件分岐により、必要な計算量は80%程度に減る

条件分岐コスト (2/2)

経過時間[s]



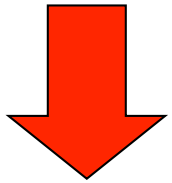
条件分岐がなければIBM POWERの方がIntel Xeonより早いが
条件分岐があると、大幅に遅くなる

IBM POWER、SPARC (京やFX10)は、条件分岐に弱い

条件分岐削除 (1/2)

1. foreach interacting particles
2. r ← particle distance
3. if distance > cutoff length then **continue** ← ここが重い
4. f ← calculate force
5. p ← update momenta
6. next

(次のループが回るかわからないから)



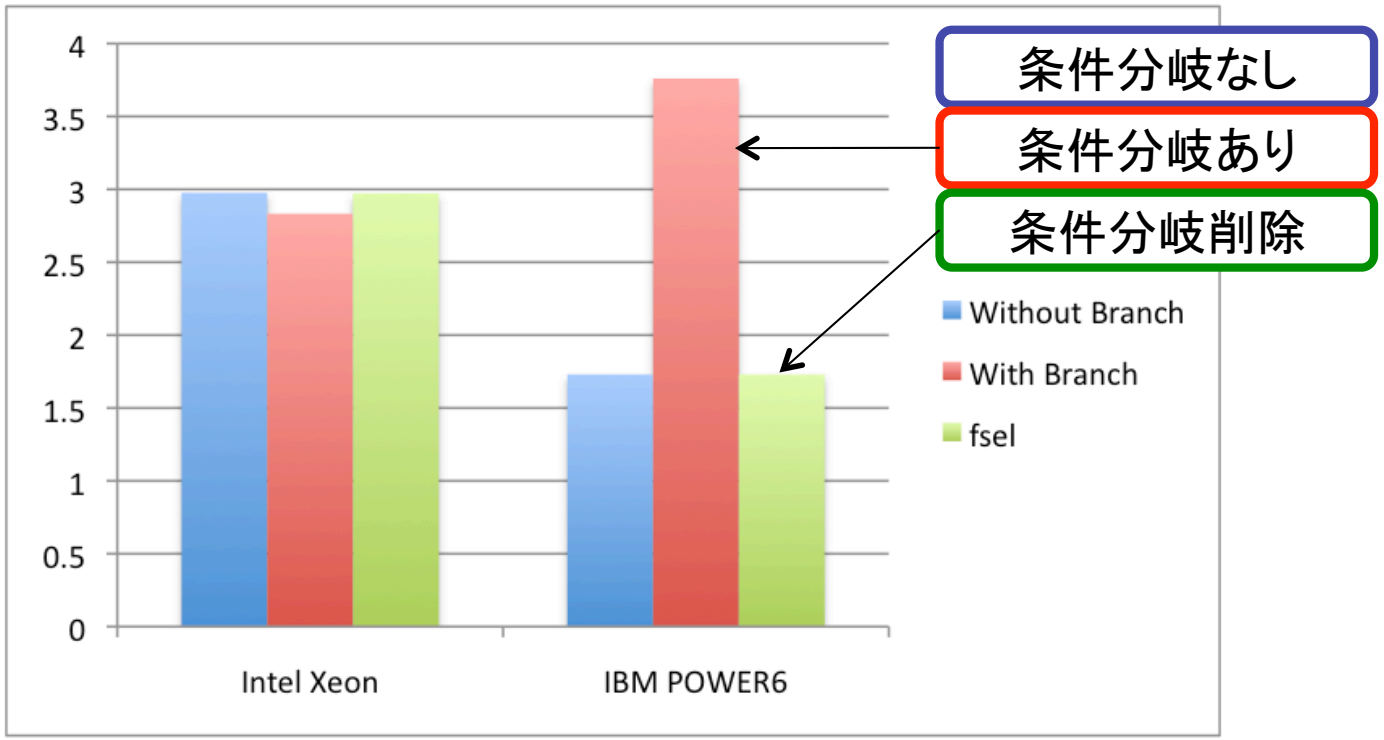
1. foreach interacting particles
2. r ← particle distance
3. f ← calculate force
4. if distance > cutoff length then f ←0 ← fsel 命令が使われる
5. p ← update momenta
6. next

(全てのループが確実にまわる)

余計な計算量が増えるが、パイプラインがスムーズに流れる
ために計算が早くなる(こともある)

条件分岐削除 (2/2)

経過時間[s]



IBM POWER の実行が大幅に高速化された
IBM POWERや、京には条件代入命令(fsel)がある

```
fsel   fp0, fp1, fp2, fp3
fp0 = (fp1 > 0)? fp2: fp3;
```

これにより、マスク処理が可能

SIMD化 (1/4)

SIMD化とは何か

コンパイラが出力する命令のうち、SIMD命令の割合を増やす事

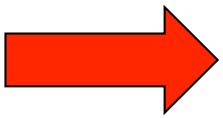
スカラ命令: 一度にひとつの演算をする
SIMD命令(ベクトル命令): 複数の対象にたいして、同種の演算を一度に行う

実際にやること

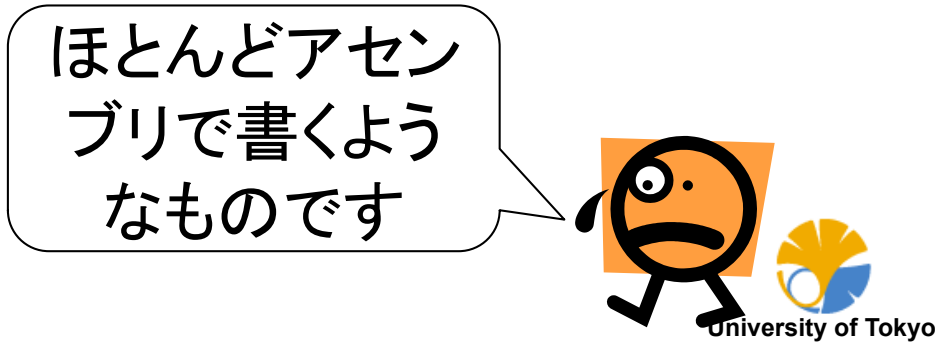
コンパイラがSIMD命令を出しやすいようにループを変形する

SIMD命令を出しやすいループ
＝演算に依存関係が少ないループ

コンパイラがどうしてもSIMD命令を出せなかったら？



手作業によるSIMD化
(Hand-SIMDize)



SIMD化 (2/4)

ループアンローリング

```
DO I = 1, N
  C[i] = A[i] + B[i]
  E[i] = C[i] + D[i]
  G[i] = E[i] + F[i]
  I[i] = G[i] + H[i]
END DO
```

依存関係があるのでSIMD命令が発行されない。

ループを二倍展開し、独立な計算を作る (馬鹿SIMD化)

A[1]+B[1]	A[2]+B[2]
C[1]+D[1]	C[2]+D[2]
E[1]+F[1]	E[2]+F[2]
I[1]+G[1]	I[2]+G[2]

京やFX10で
Loop Unrolled x times
とメッセージが出たらこれ

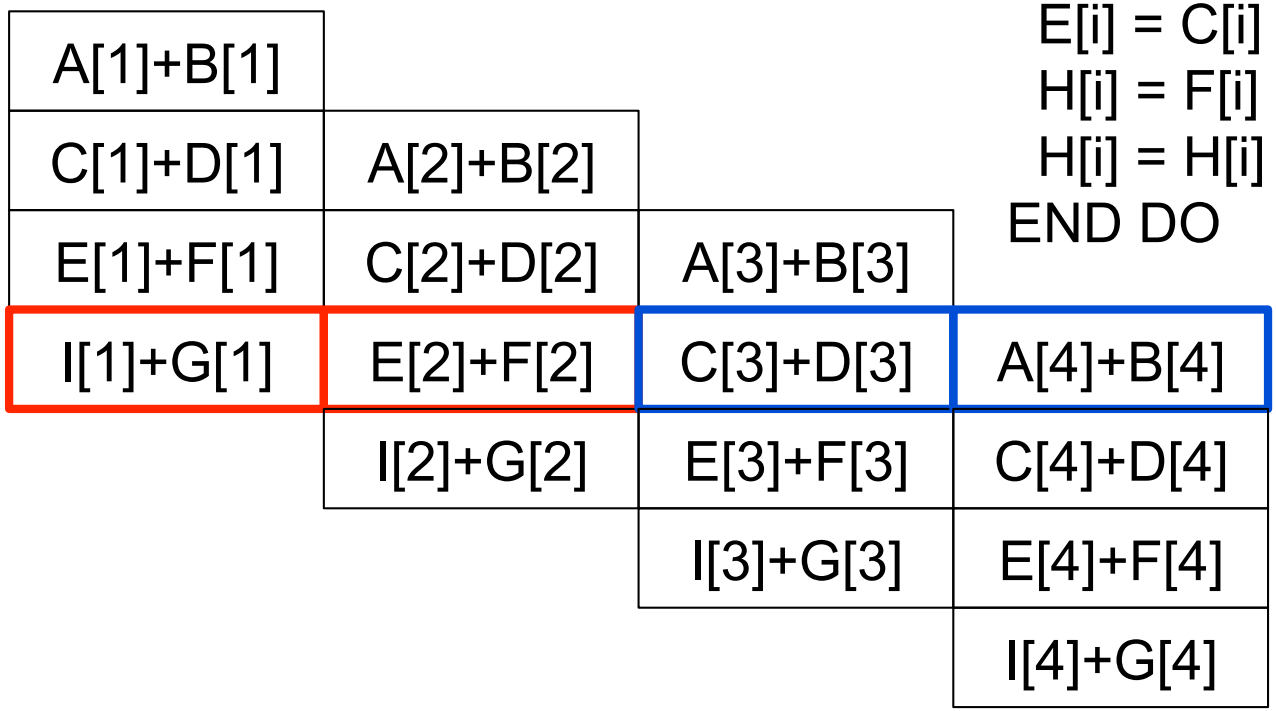
問題点:
・レイテンシを隠しづらい
・積和が作りにくい
→ 遅い

SIMD化 (3/4)

ソフトウェアパイプライニング

```
DO I = 1, N
  C[i] = A[i] + B[i]
  E[i] = C[i] + D[i]
  H[i] = F[i] + G[i]
  H[i] = H[i] + I[i]
END DO
```

ループインデックス



独立な計算を増やしやすい
レイテンシを隠し易い
人間の手でやるのはかなり厳しい
だが、京、FX10ではこれをやらないと速度が出ない
→いかにコンパイラに「Loop software pipelined」を出させるか

SIMD化 (4/4)

我々が実際にやったこと

作用反作用を使わない (二倍計算する)
→ 京、FX10ではメモリへの書き戻しがボトルネック

ループを2倍展開した馬鹿SIMDループをさらに二倍アンロール (4倍展開)
→ レイテンシ隠蔽のため、手でソフトウェアパイプライニング

局所一次元配列に座標データをコピー
→ 運動量の書き戻しは行わないので、局所座標の読み出しがネックに
→ 「グローバル、static配列でないとsoftware pipeliningできない」
というコンパイラの仕様に対応するため

除算のSIMD化
→ 京、FX10には除算のSIMD命令がない
→ 逆数近似命令(低精度)＋精度補正コードを手で書いた

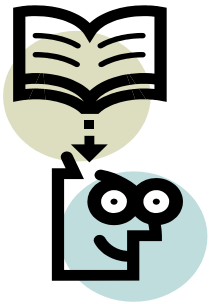
チューニング成果

力の計算の速度は2倍に、全体でも30%程度の速度向上

CPUチューニングのまとめ

CPU依存のチューニングは(当然のことながら)CPUに依存する
→ CPUを変えるたびにチューニングしなおし
→ プログラムの管理も面倒になる

そこまでやる必要あるんですか？



基本的には趣味の世界です。

世の中には全部intrinsic 命令でホットスポットを書く廃人もいるにはいる

並列化

並列化、その前に (1/3)

並列化とは何か？

複数のノードを使う事で、時間か空間を稼ぐ方法

時間を稼ぐ並列化

一つのコアあたりの計算量を減らすことで、計算規模をそのままに
計算時間を短くする (ストロングスケーリング)

空間を稼ぐ並列化

ひとつのノードでは扱えないような大きな問題を扱いたい
ひとつのコアあたりの計算量はそのままに、コア数を増やすことで
サイズを大きくする(ウィークスケーリング)

パラメタ数や統計精度を稼ぐ並列化

自明並列
別名馬鹿パラ

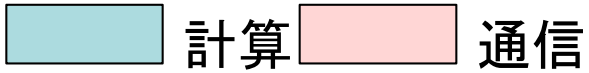
並列化、その前に (2/3)

通信時間

通信時間には、レイテンシとバンド幅が関係
レイテンシ：通信がはじまるまでの時間
バンド幅：単位時間あたりどれだけのデータを通信できるか

並列化の粒度

計算時間と通信時間の比



Granularity が疎 (計算ドミナント)
大規模計算向き

Granularity が密(通信ドミナント)
大規模計算が難しい

ストロングスケーリング(計算規模をそのままにコア数を増やす)
では、粒度が密になり、レイテンシが問題となる

並列化、その前に (3/3)

現象と計算コスト

非平衡非定常系

$$L^3$$

タイムスケールはサイズ依存性が弱い
(核生成、音速、爆発)

平衡系、非平衡定常系

$$L^3 \times L^2 = L^5$$

緩和にかかる時間が系のサイズに依存
(遅い緩和を持つ現象はさらに厳しい)

粒子数と扱える現象

我々がこれまでおこなったもの

ベンチマーク	非平衡現象 (多重核生成)	非平衡現象 (単一気泡生成)	平衡状態 (気液相転移の臨界指数)
384億粒子	14億粒子	1200万粒子	160万粒子

同じ計算資源を使った場合、定常状態は非定常状態に比べてサイズが桁で落ちる
大規模計算では非平衡非定常系の計算が向いている

ベンチマーク　> 1～3桁 > 非平衡 > 1～2桁 > 平衡状態

University of Tokyo

並列化への心構 (1/2)

「並列化」の種類

自明並列: パラメタや乱数の種を変えて並列計算 (通信ゼロ)
弱い並列 : 拡張アンサンブル、交換MC等 (通信ほぼゼロ)
非自明並列 : 領域分割、ループ分割、タスク分割等 (頻繁に通信)

「並列化」が必要か？

自分のやりたい計算は並列化しないとできない計算か？
並列化のための並列化になっていないか？

「並列化」が可能か？

ソートやイベントドリブン処理などは本質的に並列化が困難
並列化が可能であっても、緩和時間が問題に
→ 実計算時間による、系のサイズへの制限

並列化への心構 (2/2)

「並列化」の限界

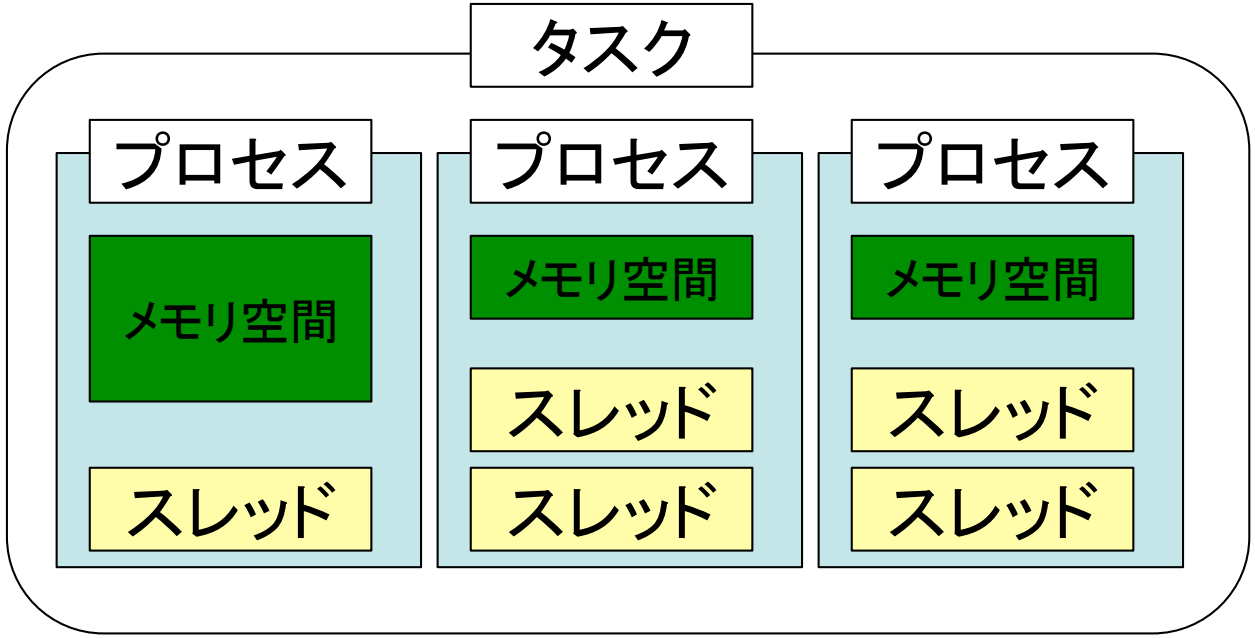
既存のソースコードを「並列化」して得られる性能には限りがある
本当に大規模計算を志すなら、最初から「並列コード」を書くべき
最終的には何度も書き直すことになる

ベンチマークからが勝負

計算科学においては、「科学論文を書くこと」が一応のゴール
大規模なベンチマークに成功しても、それだけでは論文が書けない
特に、観測ルーチンの並列化や、計算条件の調査に時間がかかる

MPIとOpenMP (1/2)

スレッドとプロセス



プロセス：OSから見た利用単位
　　ひとつ以上のスレッドと、プロセスごとに固有メモリ空間を持つ
スレッド：CPUから見た利用単位
　　同じプロセス中のスレッドはメモリを共有する
タスク：人間から見た利用単位
　　一つ以上のプロセスが協調して一つの仕事を実行する単位

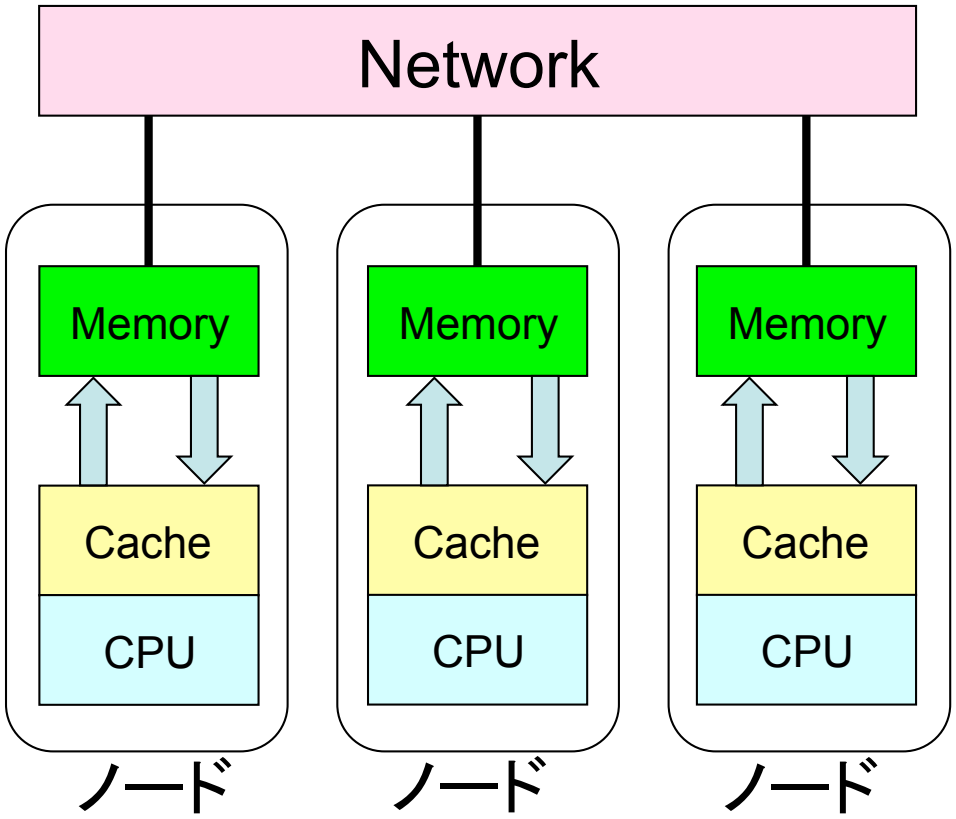
MPIとOpenMP (2/2)

OpenMP

スレッド並列の規格
各並列単位でメモリは共有

MPI

プロセス並列の規格
メモリは独立 (分散)



ノード内は共有メモリ、ノード間は分散メモリ

➡ ノードをまたぐ並列化はMPI必須

Flat MPIか、ハイブリッドか

コア・CPU・ノード

複数のコアの集合→CPU
複数のCPUの集合→ノード
複数のノードの集合→スパコン

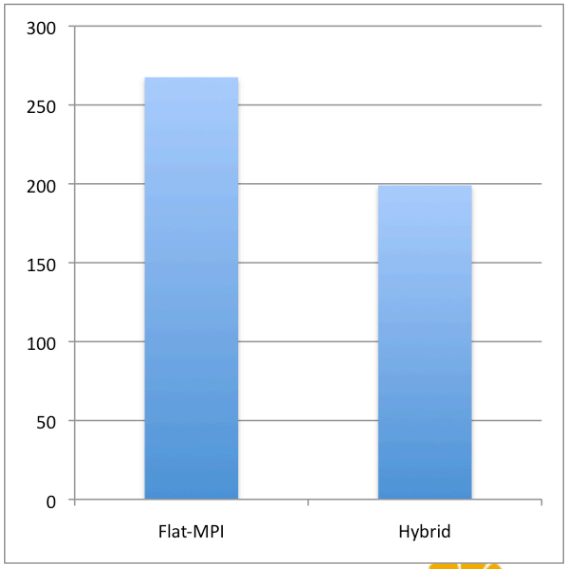
ノード内はメモリ共有
ノード間はメモリ分散

OpenMPとMPI

OpenMP　共有メモリ型並列スキーム　(ノード内) ← スレッド
MPI　　分散メモリ型並列スキーム　(ノード間) ← プロセス

一般に、Flat-MPIの方が早いが、メモリ消費も大きい

物性研 1024コア、コアあたり50万粒子
Flat-MPI: 1024 プロセス　　276.5GB
Hybrid: 128プロセス*8スレッド　199.1GB



全く同じ計算をしても、Flat-MPIではノードあたり
600MB程度利用メモリが増加する

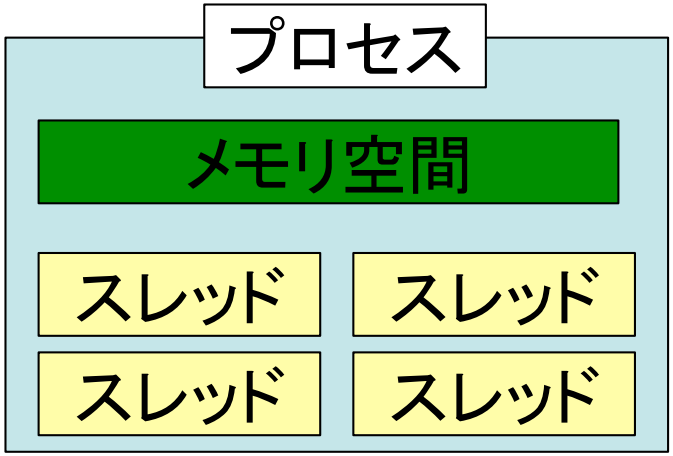
→　大規模計算ではMPI+OpenMPのハイブリッド並列必須

ハイブリッド並列化(1/5)

スレッド並列

プロセス内で並列処理 (～ノード内並列)
メモリ共有型
→排他制御が必要
OpenMP、コンパイラの自動並列

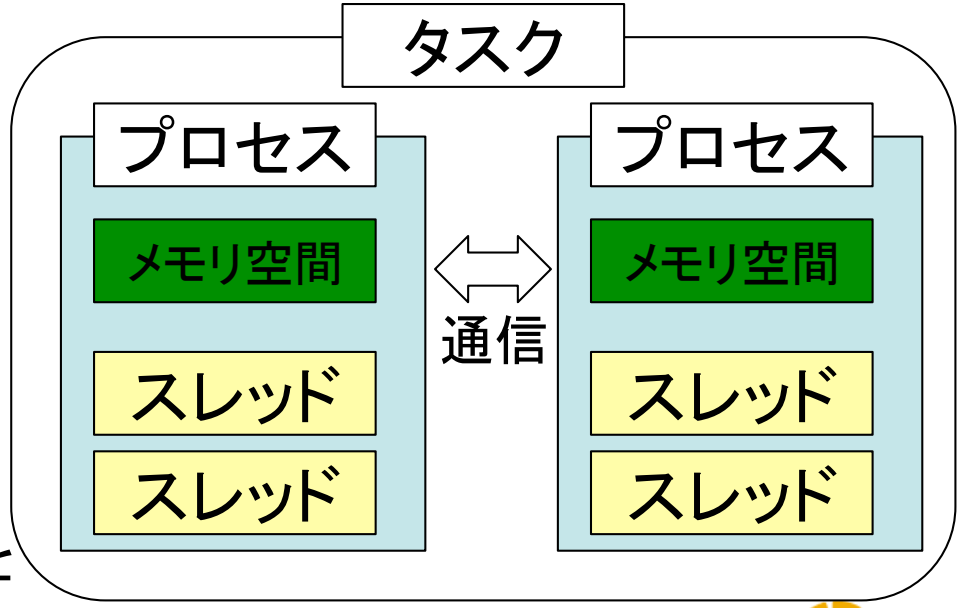
始めるのは簡単、性能を出すのは困難



プロセス並列

プロセス間で並列処理 (～ノード間並列)
メモリ分散型
→明示的通信が必要
MPIライブラリ

敷居は高いが、性能予測はし易い



ハイブリッド並列

プロセス並列とスレッド並列を両方やること

面倒くさい

ハイブリッド並列化(2/5)

一般的方法

MPI: 空間分割
OpenMP: ループ分割

```
DO I=1,N  <-------------------  ここか、
  DO J in Pair(I)  <----------  ここにスレッド並列をかける
    CalcForce(I,J)
  ENDDO
ENDDO
```

問題点

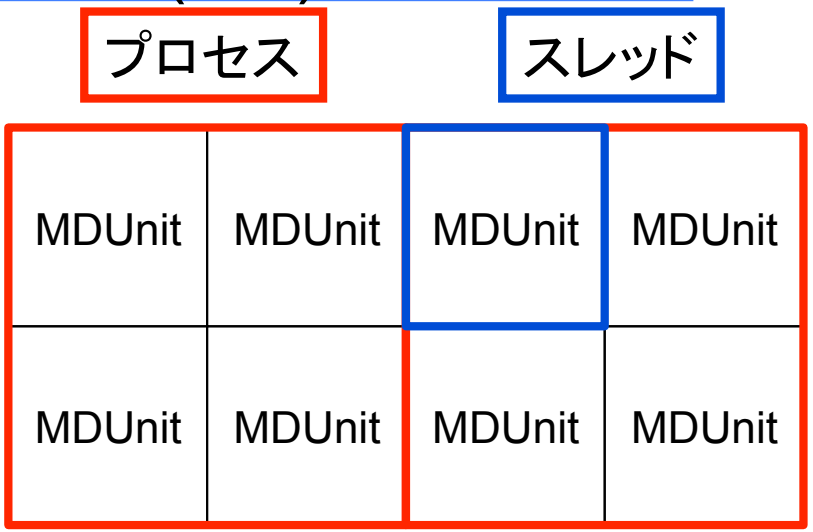
運動量の書き戻しで同じ粒子にアクセスする可能性がある
→ テンポラリバッファを用意して衝突をさける
→ 作用反作用を使わない
ペアリスト作成もスレッド並列化しなければならない
力の計算というボトルネックで、SIMD化とOpenMP化を同時に扱う必要がある

ハイブリッド並列化(3/5)

完全空間分割

MPI: 空間分割
OpenMP: 空間分割

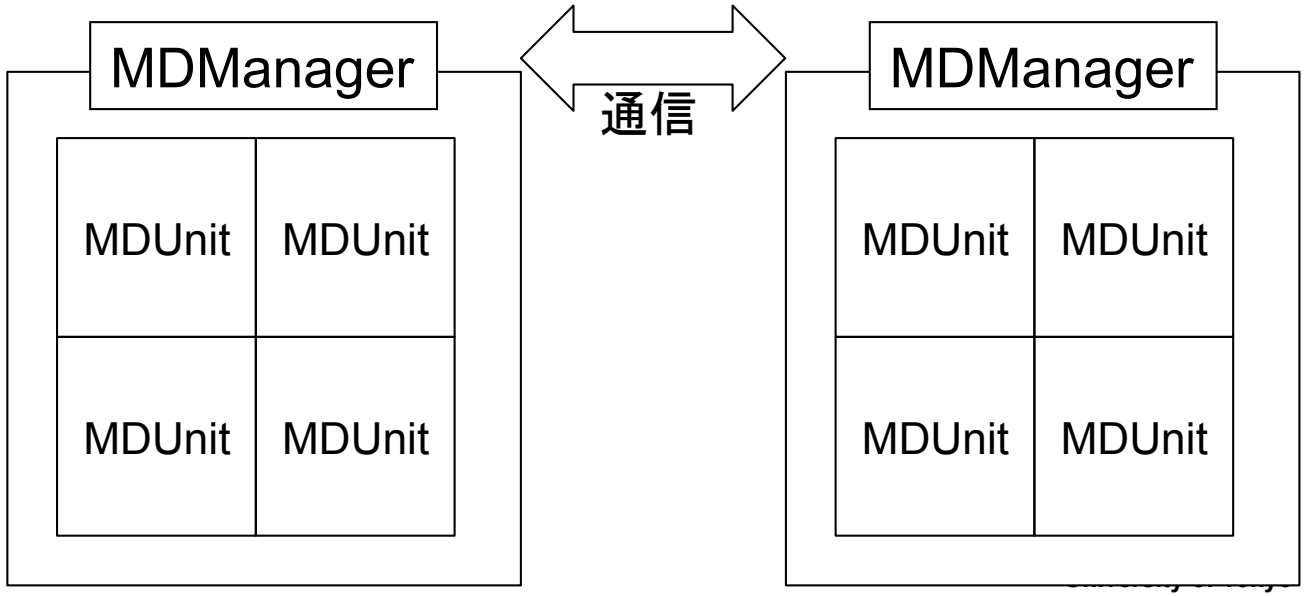
MDUnitクラス:
分割された領域を担当するクラス
MDManagerクラス
MDUnitを束ねて、通信をするクラス




```
DO i=1,THREAD_NUM
  CALL MDUnit[i]->Calculate()
ENDDO
```

← ここにスレッド並列をかける

MPIプロセス = MDManager
MDManagerがMDUnitを1つ担当 = Flat MPI
MDManagerがMDUnitを複数担当=Hybrid



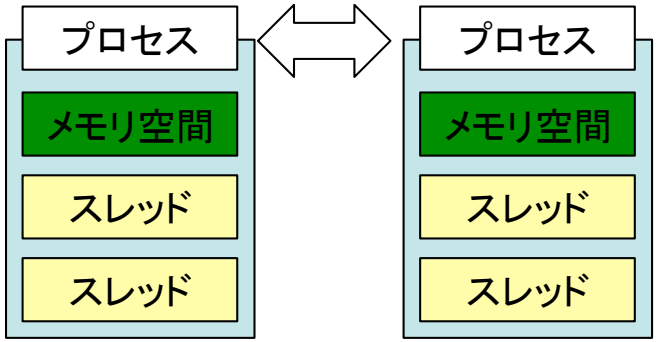
ハイブリッド並列化(4/5)

スレッドからのMPI通信

スレッドはプロセスより「軽い」かわりに、できることも限られる
例えば一般にスレッドからのMPI通信は許されて(実装されて)いない
以下の4種類の実装レベルが規定されている

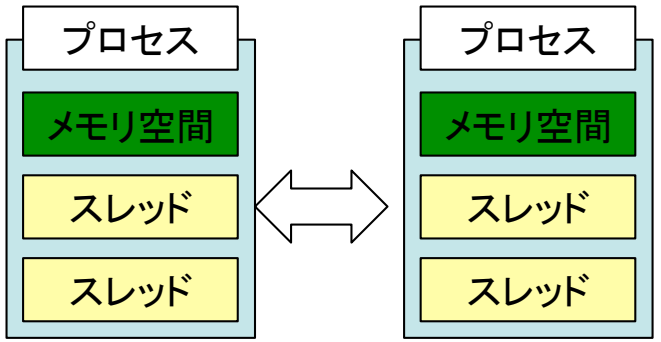
MPI_THREAD_SINGLE

スレッドの通信不可



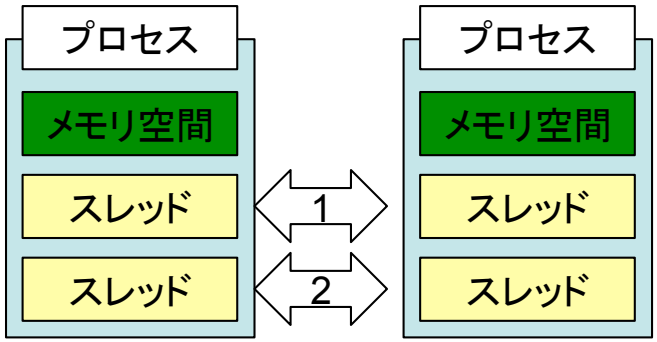
MPI_THREAD_FUNNELD

メインスレッドからなら通信可



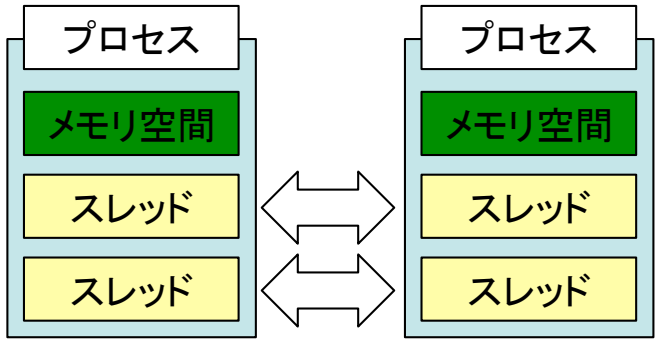
MPI_THREAD_SERIALIZED

同時に一つまでなら通信可能 (京、物性研)



MPI_THREAD_MULTIPLE

どのスレッドからでも自由に通信可



ハイブリッド並列化(5/5)

完全空間分割のメリット

MDUnitクラスは、自分がスレッドであるかプロセスであるかを意識しなくてよい
二種類のロードバランスを考えなくてよい
→ Flat-MPIがスケールする領域であれば、ハイブリッドでもスケールする

NUMA最適化を考えなくてよい(京、FX10では関係ない)

力の計算は、SIMD化のみ考えれば良い

完全空間分割のデメリット

通信が面倒くさい
→処理系によっては、スレッドからのMPI通信が実装されていない
→マスタースレッドにまとめて通信する必要がある

観測ルーチンを組むのが面倒くさい
→通信がすべて二段になっているため

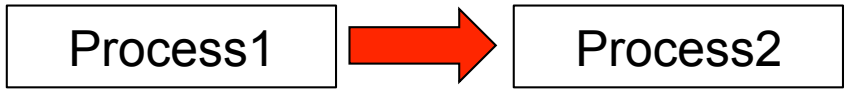
MPIを使う際の注意点

小さいジョブが成功したのに大きなジョブが失敗

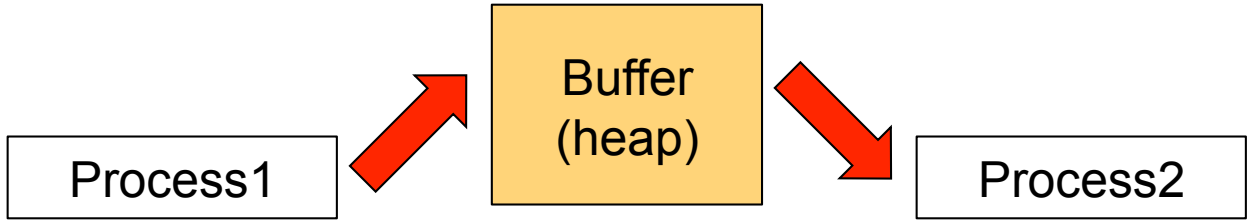
→ MPIの使うリソースの枯渇を疑う

ノンブロッキング通信

直接通信



バッファ通信



ノンブロッキング通信はバッファを多く使うため、大きいジョブでメモリ不足に陥り易い
(実はブロッキング通信を指定しても、ノンブロッキング通信が呼ばれる可能性がある)

「MPI_なんとか_MAXが足りない」というエラーメッセージがでたらこれ
対応策
→通信を小分けにする、こまめにバッファをクリアするなど・・・

MPIの実装はベンダーに強く依存
あるシステムで問題なく動作したプログラムが他のところで動かない、ということはよくある

並列化のまとめ

非自明並列はとても大変
→手をつける前に、やりたい計算に必要な資源を見積もっておく

既存のコードの「並列化」には限界がある
→ 並列化をにらんで最初から何度も組み直す覚悟

ベンチマークが取れてからが勝負
→ベンチマークとプロダクトランの間には高い高い壁がある

並列計算の障害
→並列計算環境は、環境依存が大きい
→並列計算特有のノウハウ

それら全てを乗り越えると、そこには桃源郷が・・・？
→他の人にはできない計算が気軽にできるようになる
→セレンディピティ

全体のまとめ

高速化、並列化は好きな人がやればよい

なぜなら科学とは好きなことをするものだから