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

Jun. 11, 2013@計算科学技術特論A

University of Tokyo

注意

今日話すことは、おそらく今後の人生にほとんど役にたちません

ただ、「こういうことをやる人々がいる」ということだけ知っておいてください

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」

→ 計算機の仕組みを理解する

高速化、その前に

やみくもに高速化をはじめるのはダメ、ゼッタイ
まず、どこが遅いか、どういう原因で遅いかを確認してから
→プロファイラによる確認

計算機の仕組み

計算機の仕組み (1/3)

計算機とは何か？

計算するところ (CPU)
高速小容量メモリ (キャッシュ)
低速大容量メモリ (メインメモリ)

} これらをまとめて「ノード」と呼ぶ

※ストレージ、通信関係等は省略

スパコンとは何か？

ノードをたくさん通信ケーブルでつないだもの

スパコンの構成要素は本質的には普通のPCと同じ
ただ量が違うだけ

量の違いは質の違いとなるか・・・？

計算機の仕組み (2/3)

ノード (国内)

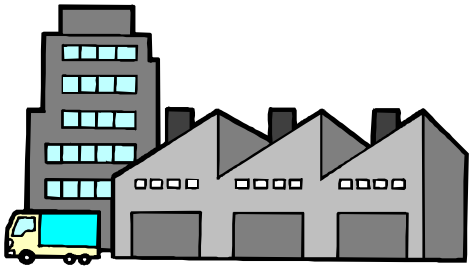
工場 (CPUコア)

物置
(キャッシュ)

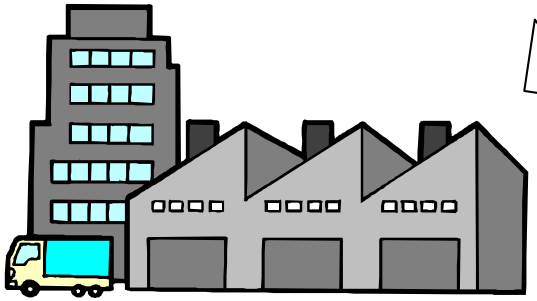
国内流通
(メモリ転送)

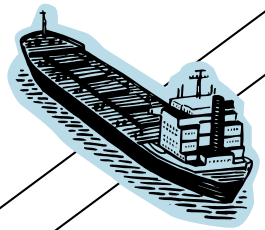
ボトルネック

倉庫(メモリ)



別の国の倉庫
(別のノードのメモリ)



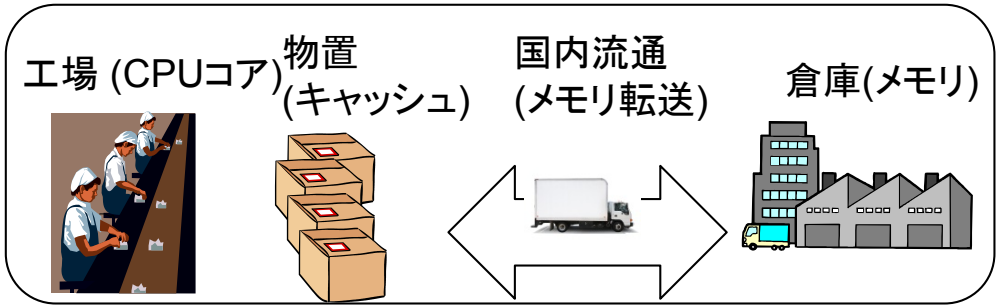


貿易
(ノード間通信)

計算機の仕組み (3/3)

CPUの仕事

- メモリからデータを取ってくる
- 計算する(ほとんど四則演算)
- 結果をメモリに書き戻す



レイテンシとバンド幅

レイテンシ：発注してからデータが届くまでの時間 (トラックの速度)

必要な材料が物置になかった→倉庫に発注
材料が届くまで工場は暇になる (およそ数百～千サイクル)

バンド幅　：定常的なデータの転送量 (道路の車線数)

現在の典型的なCPUでは、データを2個持ってきて1個書き戻す間に50回くらい計算できる能力がある
＝工場の能力に比べて道路の車線数が少なすぎる

補給あっての戦線
メモリまわりの最適化が終わってからCPUチューニングへ

Byte / Flop

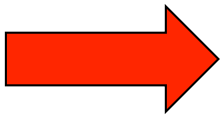
Byte/Flop

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
二つとってきて計算して書き戻すと24 Byteの読み書きが発生
B/F = 0.5 のマシンでは、24 Byteの読み書きの間に48回計算ができる



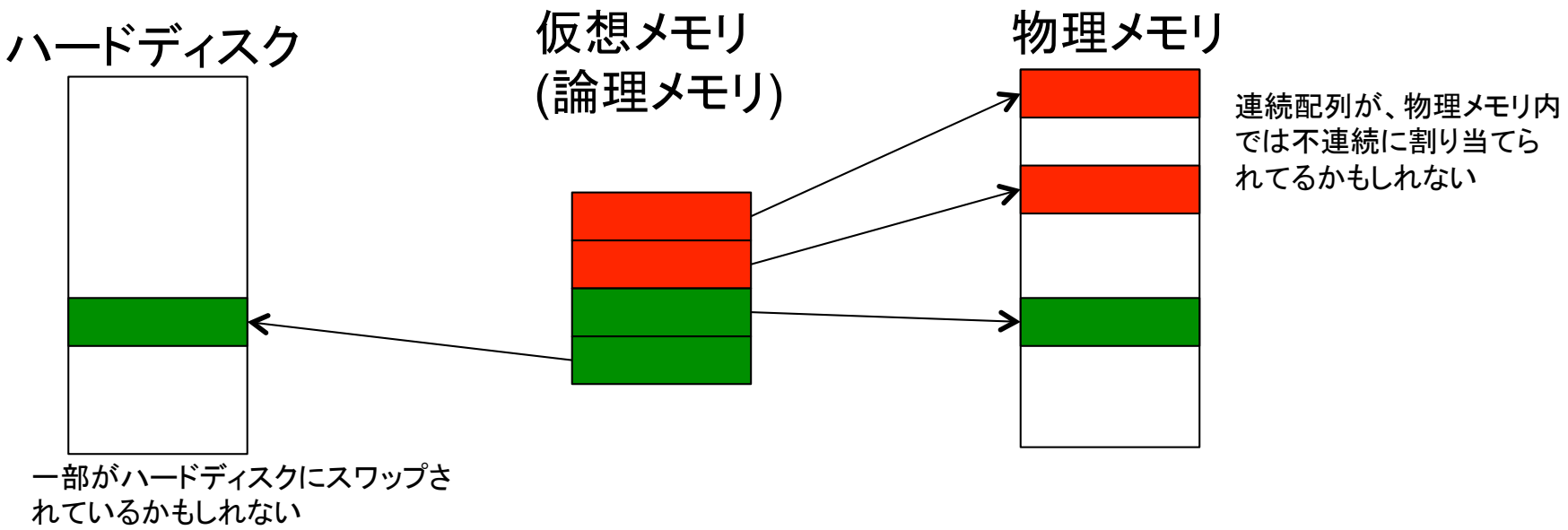
B/Fが0.5の計算機では、C = A*Bという単純な計算をくりかえすと
ピーク性能の2%ほどしか出せず、ほとんど時間CPUが遊ぶ

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



- 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
- スワッピングが可能となり、物理メモリより広い論理メモリ空間を取れる

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

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))   ← 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
do i=1, 10000
  work(i) = i            ← 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
end do
```

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かった？

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(?)

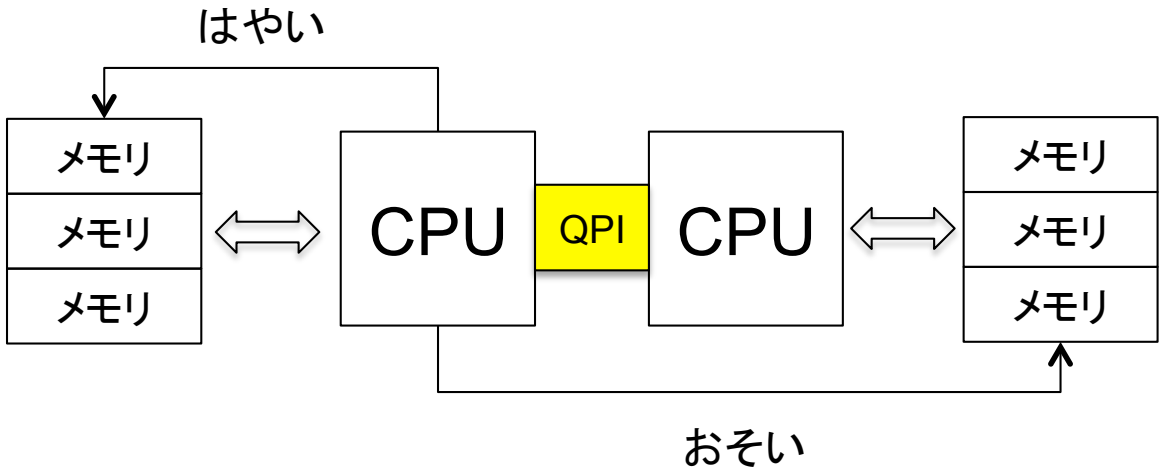
まぁ、そういうこともあると覚えておいてください

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Uniform Accessだが、XeonなどはNUMA

旧物性研システムB(SGI Altix ICE 8400EX) のノード構成



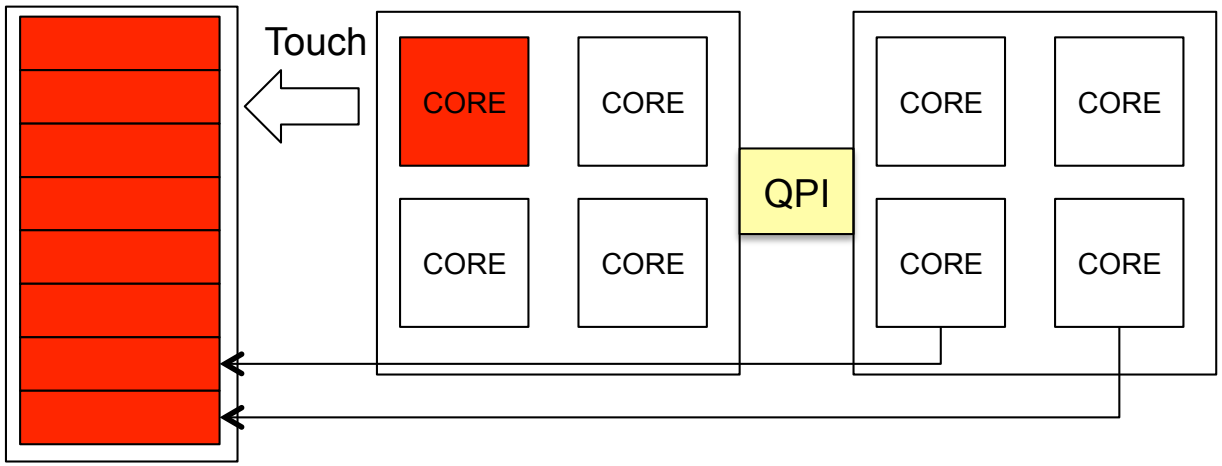
- ノード内にCPUが2つ (2ソケット)
- CPU同士はQPIで接続されている
- メモリは4GBが6枚
- CPUから見て3枚が「近いメモリ」、残り3枚が「遠いメモリ」

NUMA (2/2)

First Touchの原則

そのメモリに一番最初に触りにいったコアの一番近くに物理メモリを割り当てる

問題となる例:
OpenMPによるスレッド並列で、初期化を適当にやったらマスタースレッドの近くに割当。その後、スレッドが遠いメモリを読み書きするので遅くなる



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 (1/2)

CPUとは

CPU (Central Processing Unit、中央演算処理装置)は、コンピュータの「頭脳」
様々な種類があり、各々のCPUごとに得意/不得意や、使い方の「クセ」がある。

CPUコアとは

プロセッサコアのこと。これ一つでCPUの役割を果たせるもの
最近のCPUはほとんどマルチコア
複数のコアでキャッシュやメモリを共有(階層化されていることもある)
京(8コア) FX10 (16コア) IBM POWER6 (32コア) 最近のXeonは10～18コア?
60コアといった多数のコアを集積したものもある(メニーコア)

SIMD

Single Instruction Multiple Dataの略
一回の命令実行で、複数のデータを処理する仕組み
倍精度浮動小数点は64bit
128bit幅のSIMDなら二つ同時に計算できる
256bit幅のSIMDなら四つ同時に計算できる

CPU (2/2)

ゲーム機とスパコンでの採用例

第五世代	SS	SH-2
	PS	R3000A (MIPS)
	N64	VR4300 (MIPS)
第六世代	DC	SH-4
	PS2	MIPS (Emotion Engine)
	GC	IBM PowerPC カスタム (Gekko)
	Xbox	Intel Celeron (Pentium III 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	Wii U	IBM Espresso Power
	PS4	AMD Jaguar
	Xbox One	AMD Jaguar

物性研 SGI Origin 2800 (MIPS)

KEK Blue Gene/Q (PowerPC)
via http://scwww.kek.jp/

LANL Roadrunner (Cell)
via http://ja.wikipedia.org/wiki/Roadrunner

命令パイプライン (1/4)

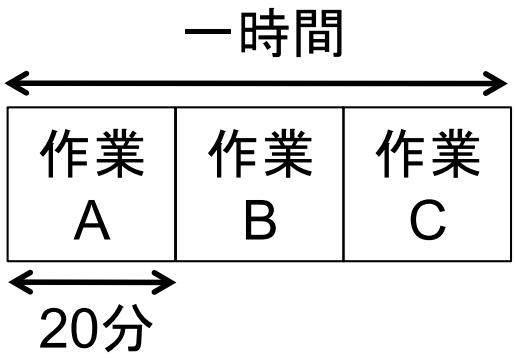
非パイプライン方式

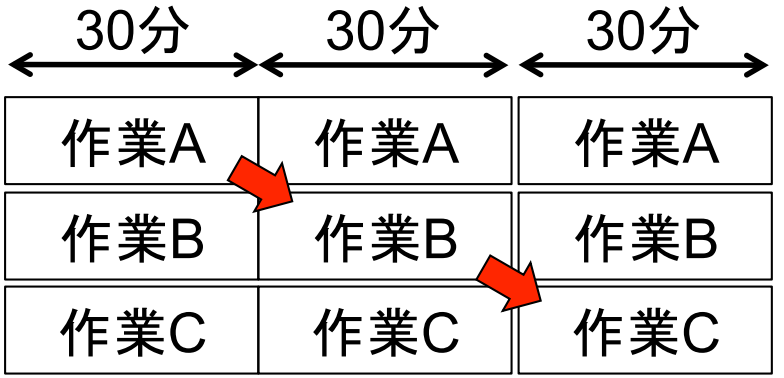
工場で熟練した職人が製品を作る
職人の手さばきが速くなるほど速く製品ができるが、限界がある

パイプライン方式

工程を簡単な作業にわけ、ベルトコンベア式に

職人一人で作ると1時間かかる作業を3つにわける
ベルトコンベアに部品を流し、各パートを作業員に任せる
しかし、三人が同時に作業できるので、一つの製品が30分で完成 (2倍の高速化)



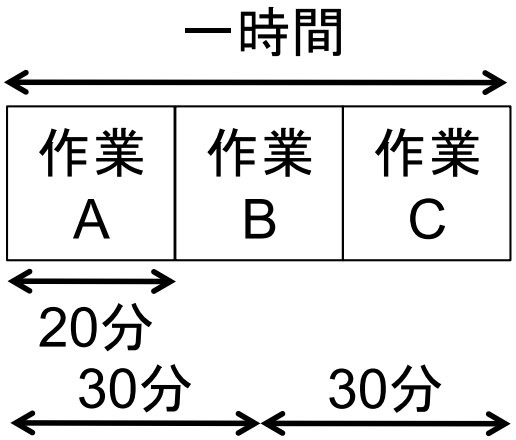


命令パイプライン (2/4)

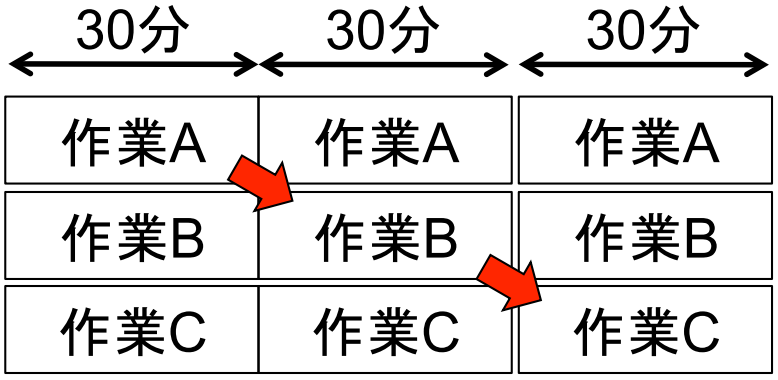
レイテンシ

一つの製品を作るのにかかるトータルの時間
(命令が発行されてから、値が返ってくるまでのクロック数)

職人がやると20分でできる仕事が、作業員だと30分に (一つ作るのに1時間半)



レイテンシ：一時間

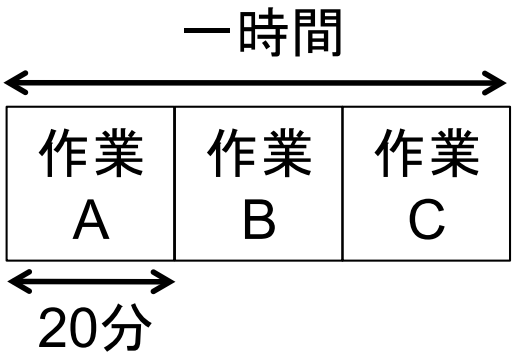


レイテンシ：一時間半

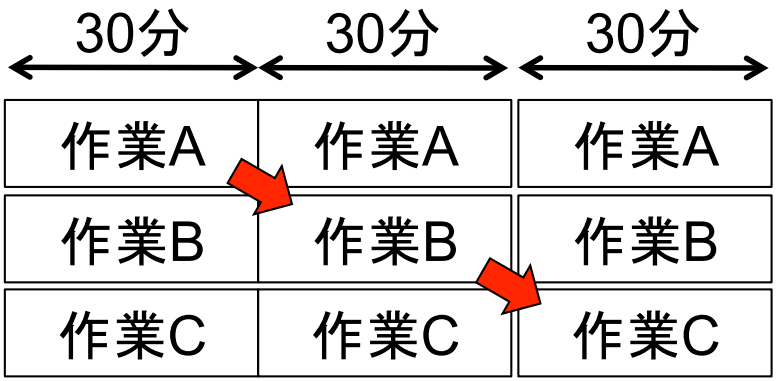
命令パイプライン (3/4)

スループット

単位時間あたり、何個の製品ができるか
(サイクルあたり何回計算できるか)



1時間に一つ製品が完成



一時間に二つ製品が完成
(2倍の高速化)

命令パイプライン (4/4)

パイプラインハザード： 暇な作業員が発生すること

データハザード

x ← x + y
w ← x * z

前の計算が終わるまで、次の計算を実行できない

作業A			作業A		
	作業B			作業B	
		作業C			作業C

データハザード

if x >0 then x ← y + z

条件の真偽が判明するまで、次にやるべき計算がわからない

作業A			作業A		
	作業B			作業B	
		作業C			作業C

積和命令とSIMD

積和、積差

積和	X = A*B+C	fmaddd X,A,B,C
積差	Y = A*B-C	fmsubd X,A,B,C

乗算一つ、加減算一つ、あわせて二つを一度に計算

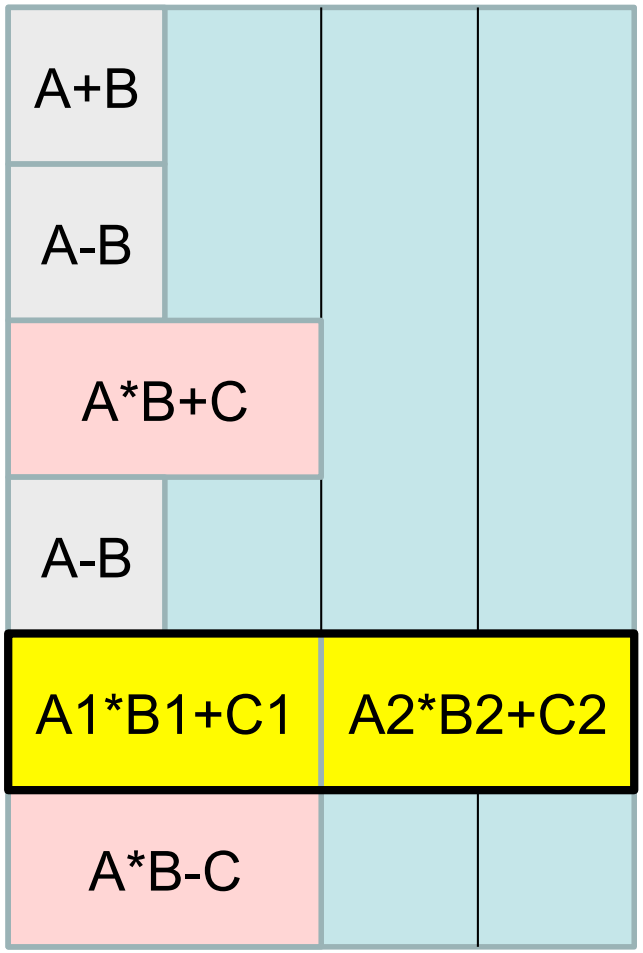
SIMD (Single Instruction Multiple Data)

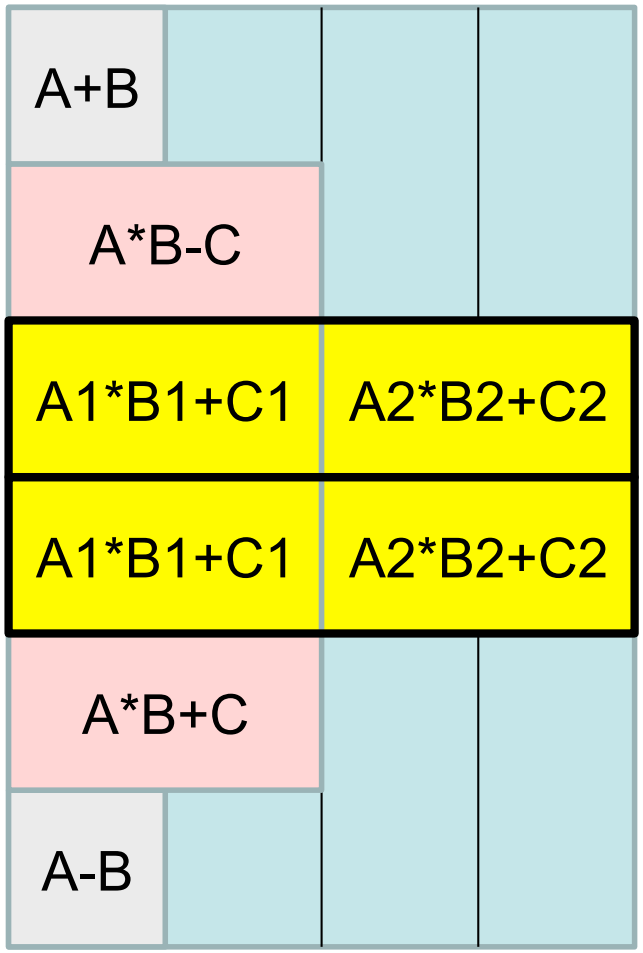
独立な同じ形の計算を同時に行う

和のSIMD計算 X1 = A1 + B1, X2 = A2 + B2 (faddd,s)
積のSIMD計算 X1 = A1 * B1, X2 = A2 * B2 (fmuld,s)
積和のSIMD計算 X1 = A1 * B1+C1, X2 = A2 * B2+C2 (fmaddd,s)

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

パイプラインのイメージ

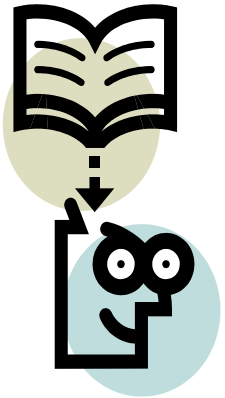




- 幅が4、長さが6のベルトコンベアが2本ある
- それぞれには、大きさ1、2、4の荷物を流せるが、同時に一つしか置けない
- ピーク性能比＝ベルトコンベアのカバレージ
- ピーク性能を出す＝なるべくおおきさ4の荷物を隙間無く流す
  (パイプラインの構造はアーキテクチャにより異なる)

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演算がたくさん必要
  → 「京」では、独立な積和のSIMD演算がたくさん必要



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、fmaddd,sなどでgrep)

閑話休題

研究成果のアウトリーチについて

プロファイラの使い方

チューニングの前にはプロファイリング

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

サンプリング

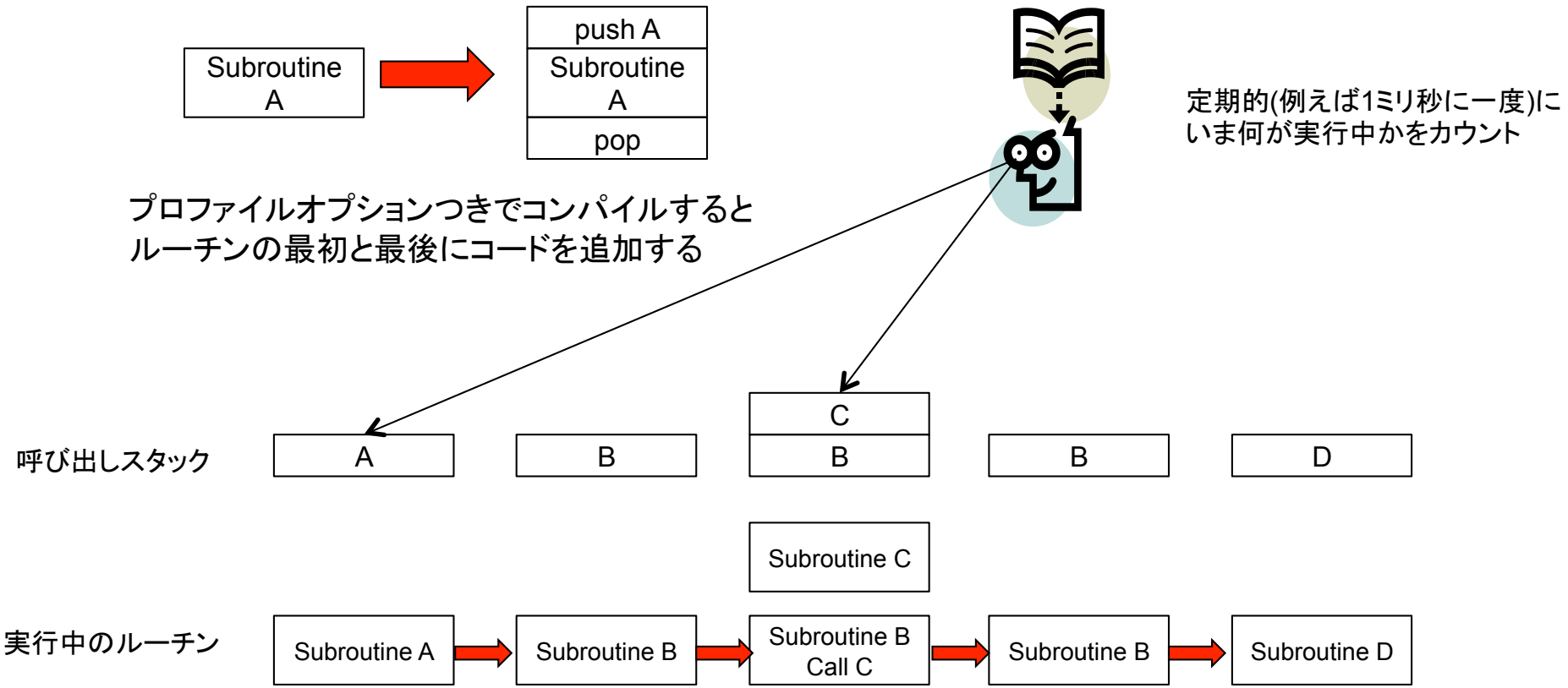
どのルーチンがどれだけの計算時間を使っているかを調べる
ルーチン単位で調査
ホットスポットを探すのに利用

イベント取得

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

サンプリング

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の使い方

gprofとは

広く使われるプロファイラ
(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

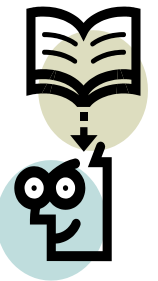
とりあえずEach % timeだけ見ればいいです

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

perfの使い方(サンプリング)

perfとは

Linuxのパフォーマンス解析ツール
プログラムの再コンパイル不要

使い方

```
$ perf record ./a.out
$ perf report
```

出力

手元のMDコードを食わせてみた結果

```
Samples: 155K of event 'cycles', Event count (approx.): 139410304992
 85.93%  mdacp  mdacp                [.] ForceCalculator::CalculateForceNext(Var
  5.26%  mdacp  mdacp                [.] MeshList::SearchMesh(int, Variables*, S
  2.10%  mdacp  libgomp.so.1.0.0     [.] 0x000000000000f05c
  1.50%  mdacp  mdacp                [.] ForceCalculator::UpdatePositionHalf(Var
  1.08%  mdacp  mdacp                [.] MDUnit::MakeBufferForBorderParticles(in
  0.92%  mdacp  mdacp                [.] PotentialEnergyObserver::Observe(Variab
  0.72%  mdacp  mdacp                [.] VirialObserver::Observe(Variables*, Mes
  0.53%  mdacp  mdacp                [.] MeshList::MakeList(Variables*, Simulati
  0.45%  mdacp  mdacp                [.] PairList::CheckByDisplacement(Variables
```

86%が力の計算
5%がペアリストの作成　といったことがわかる

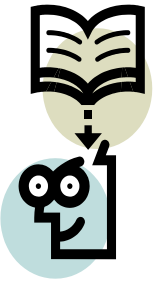
結果の解釈 (サンプリング)

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

取得可能な主なイベント:
- 実行命令 (MIPS)
- 浮動小数点演算 (FLOPS)　←こういうのに気を取られがちだが
- サイクルあたりの実行命令数 (IPC)
- キャッシュミス
- バリア待ち　←個人的にはこっちが重要だと思う
- 演算待ち

プロファイラの使い方

システム依存
Linux では perfを使うのが便利
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

perfの使い方(イベントカウント)

使い方

$ perf stat ./a.out

出力

```
Performance counter stats for './mdacp':

     38711.089599 task-clock                #    3.999 CPUs utilized
            4,139 context-switches          #    0.107 K/sec
                5 cpu-migrations            #    0.000 K/sec
            3,168 page-faults               #    0.082 K/sec
  138,970,653,568 cycles                    #    3.590 GHz
   56,608,378,698 stalled-cycles-frontend   #   40.73% frontend cycles idle
   16,444,667,475 stalled-cycles-backend    #   11.83% backend  cycles idle
  233,333,242,452 instructions              #    1.68  insns per cycle
                                            #    0.24  stalled cycles per insn
   11,279,884,524 branches                  #  291.386 M/sec
    1,111,038,464 branch-misses             #    9.85% of all branches

      9.681346735 seconds time elapsed
```

4CPUコアを利用

IPC = 1.68

分岐予測ミス

キャッシュミスなども取れる
取得できるイベントは perf listで確認

結果の解釈 (イベントカウンタ)

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

あきらめましょう
それでも対処したい人へ： 依存関係を減らしてsoftware pipeliningを促進したりとか

プロファイリングで遅いところと原因がわかった？
よろしい、ではチューニングだ

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

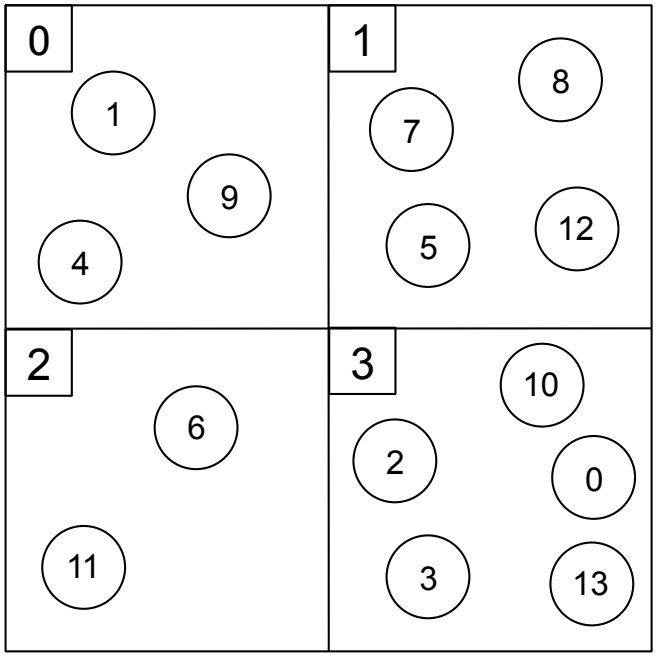
メモリ最適化1 セル情報の一次元実装 (1/2)

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
```

i番目のセルに入っているj番目の粒子番号 = GridParticleIndex[i][j];



GridParticleIndexの中身はほとんど空

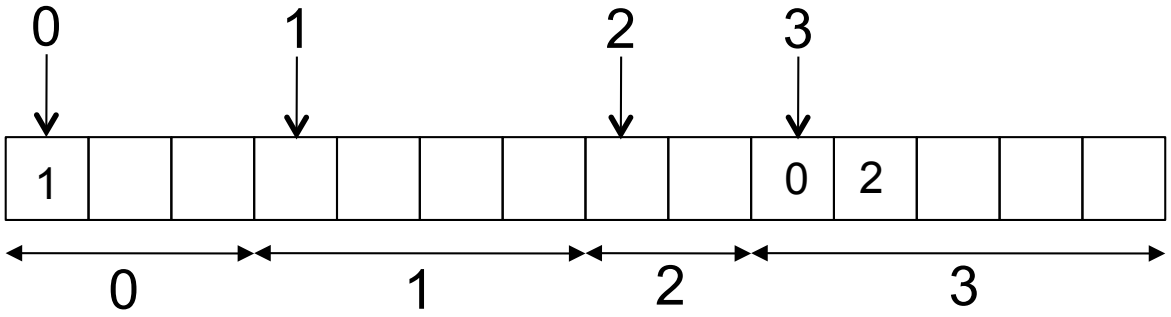
1	4	9							
7	5	8	12						
6	11								
0	2	3	10	13					

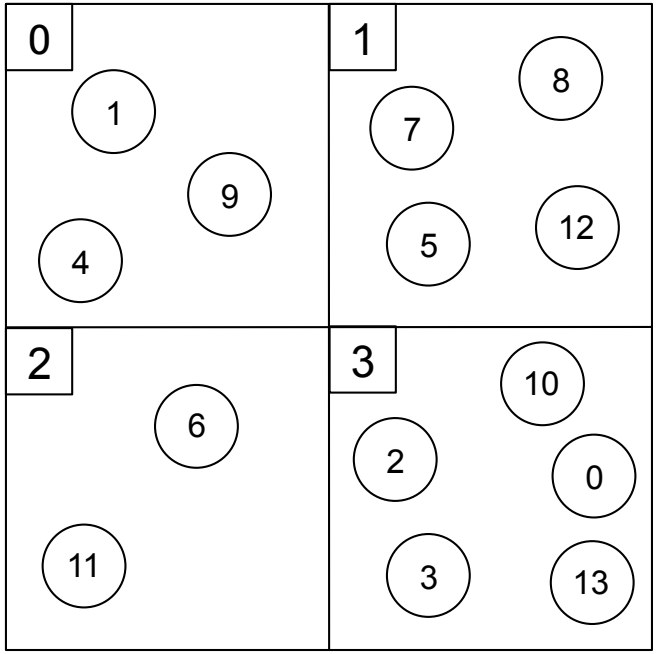
広いメモリ空間の一部しか使っていない
→ キャッシュミスの多発

メモリ最適化1 セル情報の一次元実装 (2/2)

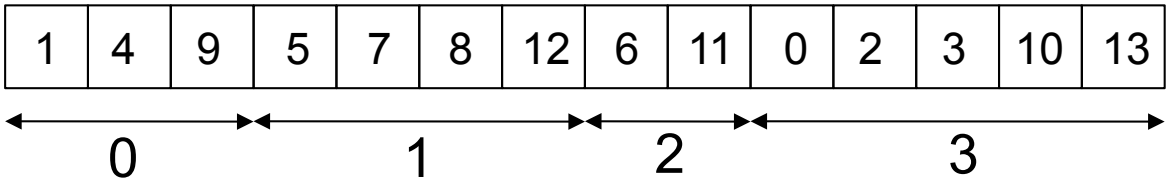
一次元実装

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)

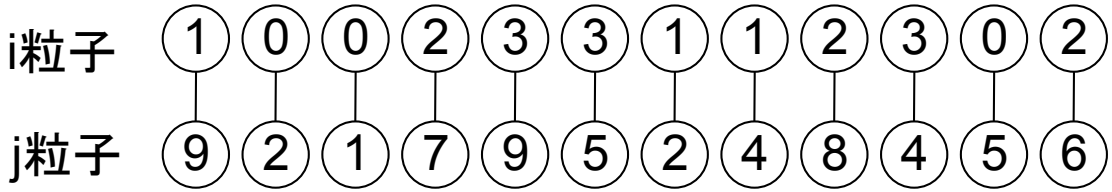
University of Tokyo

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i粒子、相手をj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

i粒子	1	0	0	2	3	3	1	1	2	3	0	2
j粒子	9	2	1	7	9	5	2	4	8	4	5	6

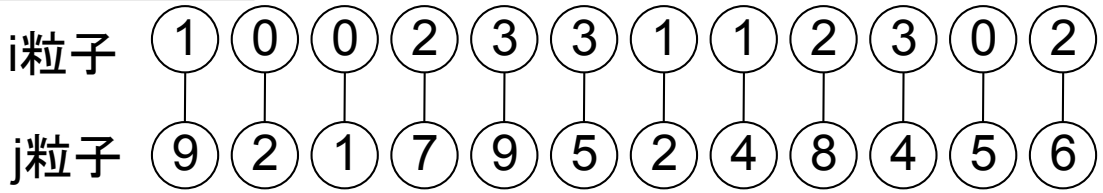
このまま計算すると

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)

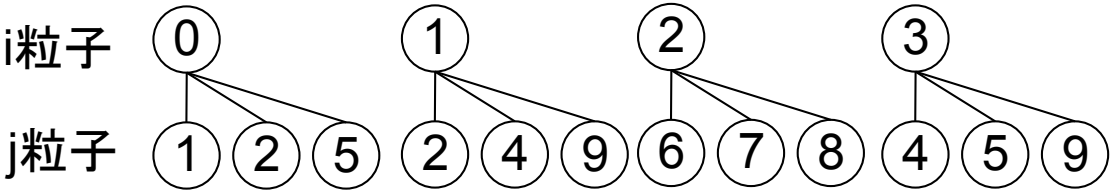
メモリ最適化2 相互作用ペアソート (2/2)

相互作用相手でソート

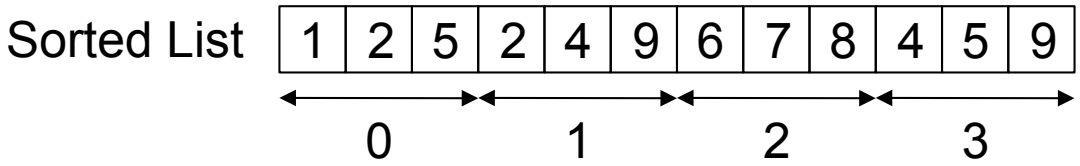
相互作用ペア



i粒子でソート



配列表現

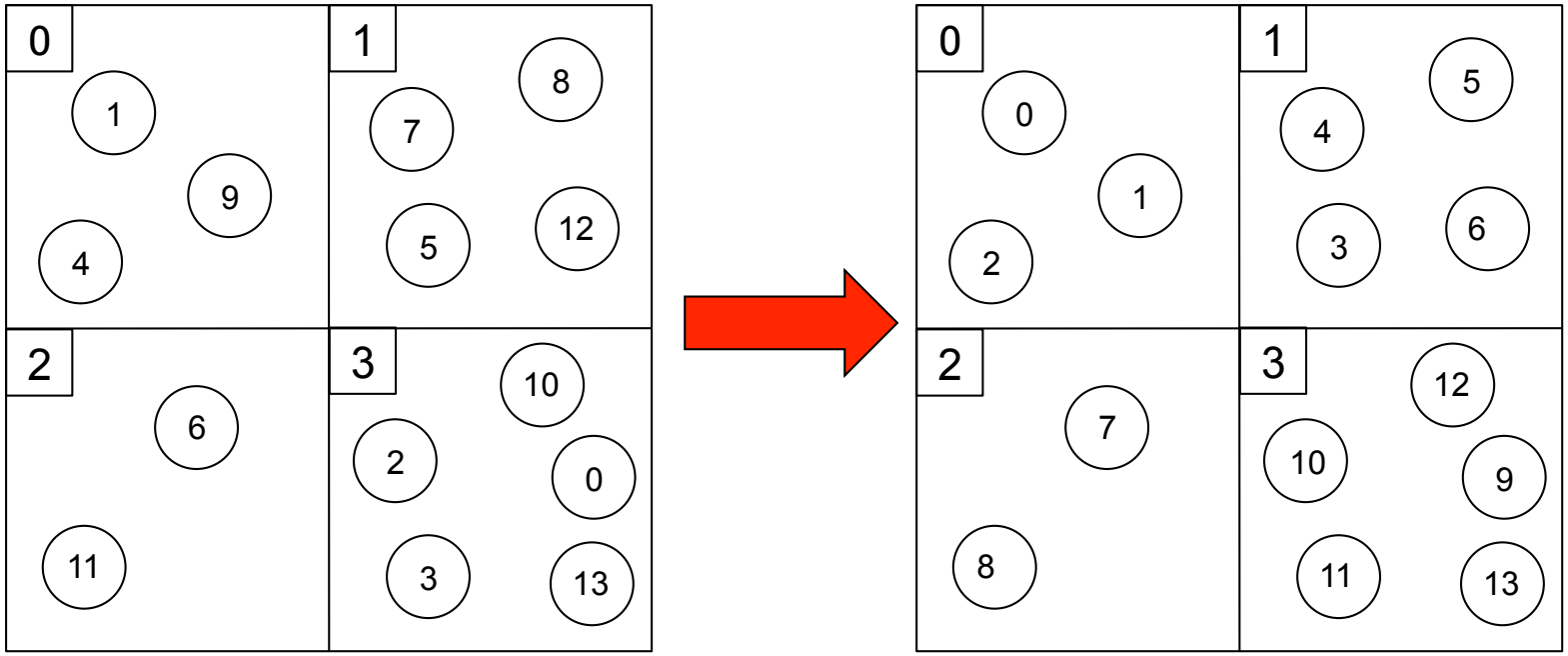


i粒子の情報がレジスタにのる
→ 読み込み、書き込みがj粒子のみ
→ メモリアクセスが半分に

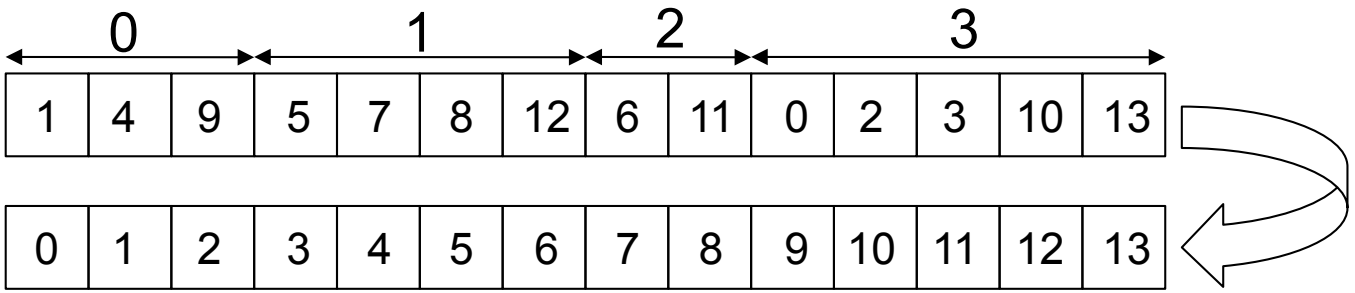
メモリ最適化3 空間ソート (1/2)

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

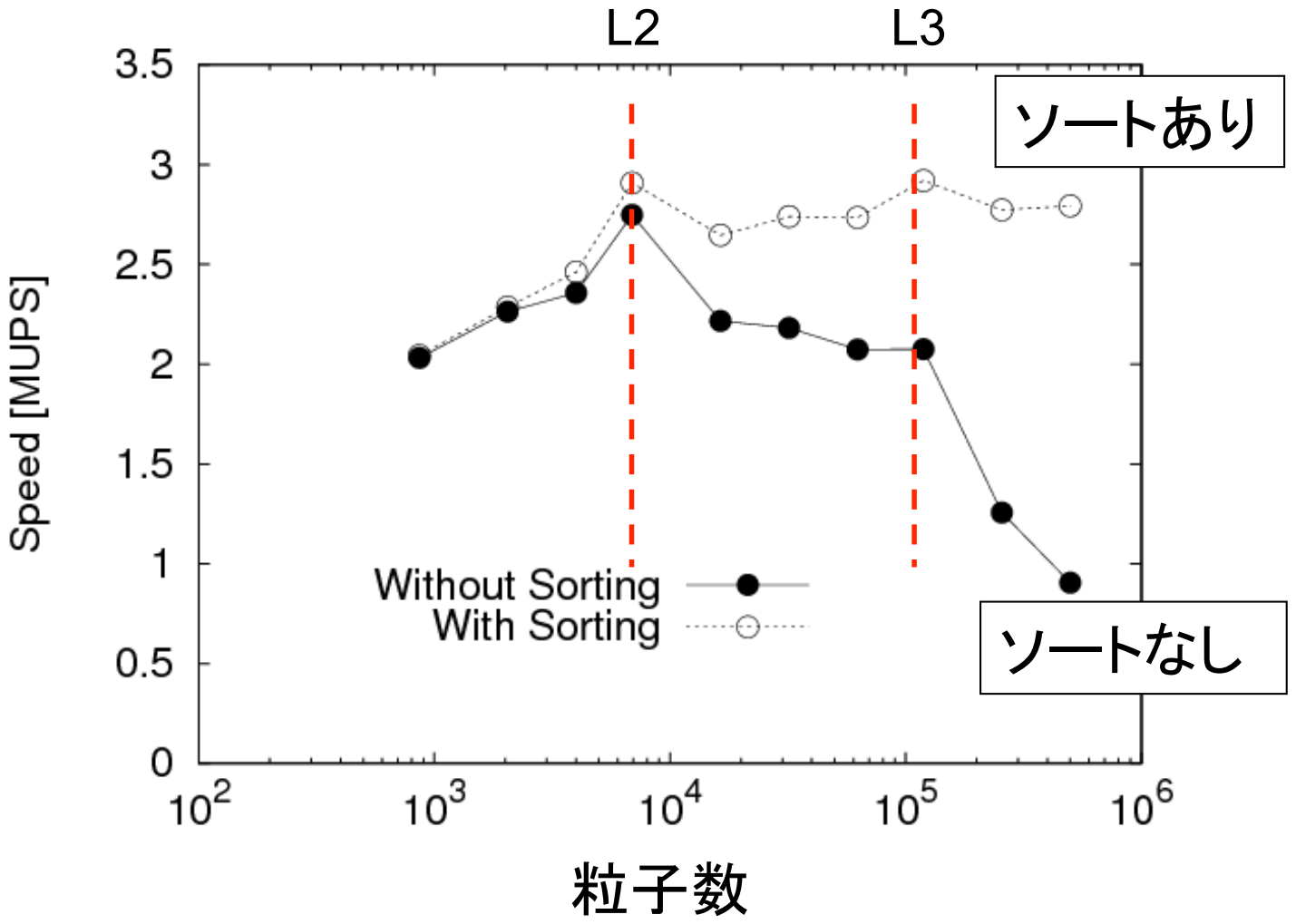


ソートのやりかたはセル情報の一次元実装と同じ



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

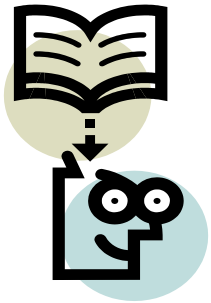
必要なデータがほぼキャッシュに載っており、CPUの
計算待ちがほとんどになって初めてCPUチューニングへ

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・・・

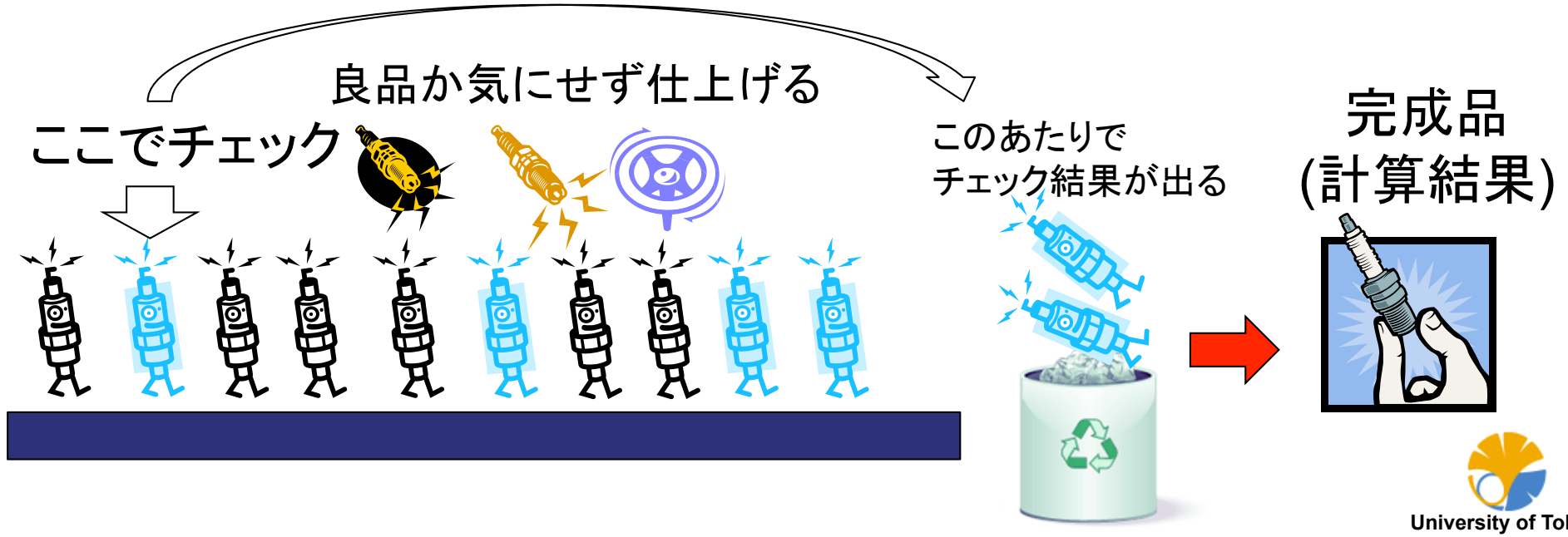


それ、条件分岐が原因かもしれません。

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

条件分岐削除 (1/3)

製品のチェックをして、良品だけ最後の仕上げをしたい



条件分岐削除 (2/3)

1. 粒子の距離を計算
2. ある程度以上遠ければ次のペアへ ← ここが重い原因(パイプラインハザード)
3. 粒子間の力を計算
4. 速度を更新
5. 次のペアへ

次のループが回るかわからない
性能は分岐予測の精度に依存

1. 粒子の距離を計算
2. 粒子間の力を計算
3. もし距離が遠ければ力をゼロに上書き ← マスク処理
4. 速度を更新
5. 次のペアへ

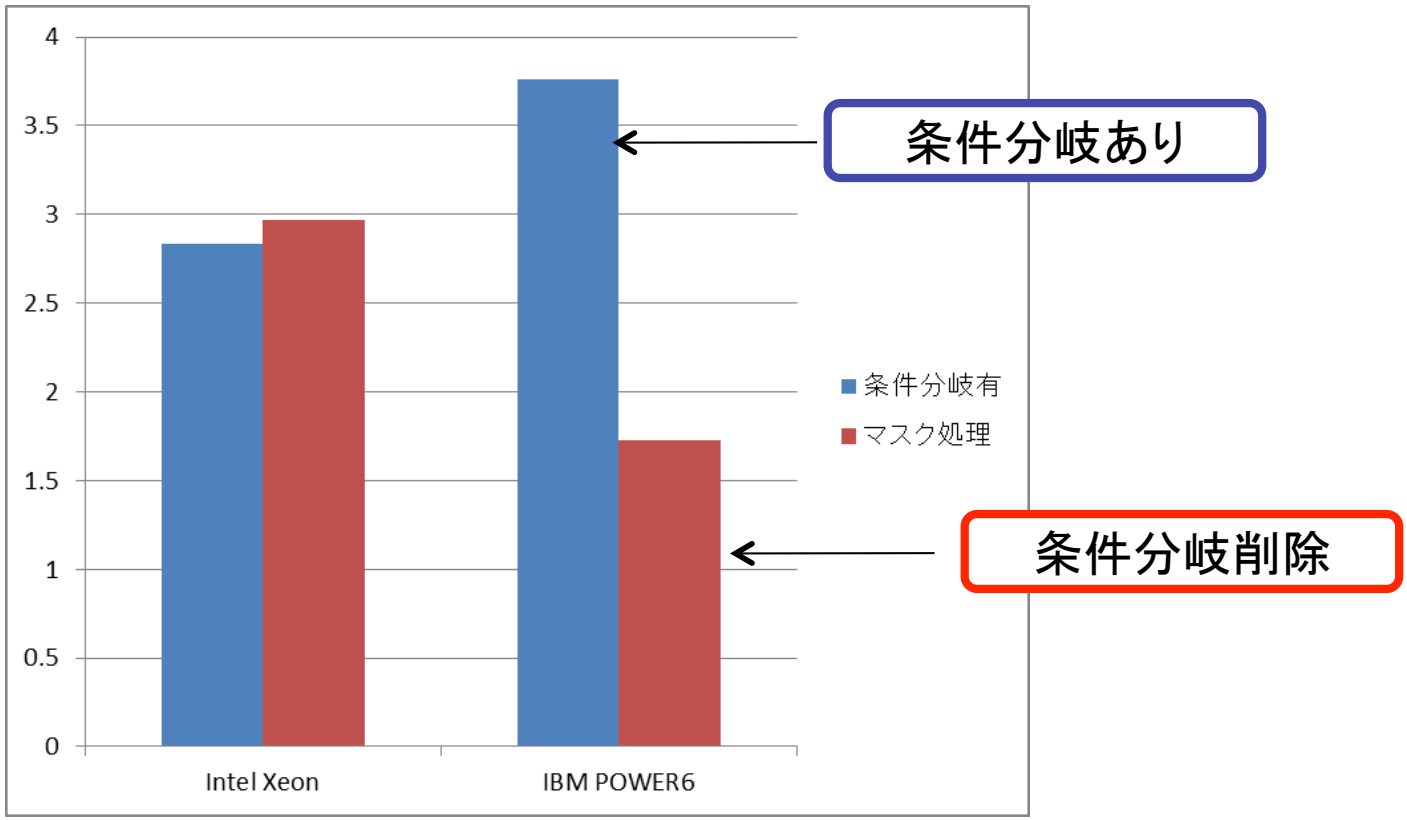
(全てのループが確実にまわる)

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早くなる
※遅くなることもある

条件分岐削除 (3/3)

条件分岐削除の結果 (2万粒子 N^2計算)

経過時間[s]



→ Intel Xeon (2.93GHz)では実行時間が微増 (2.8秒→3.0秒)
IBM POWER6 (3.5GHz) では大幅に高速化(3.8秒→1.7秒)

マスク処理による条件分岐削除はSIMD化にも有効

SIMD化 (1/2)

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

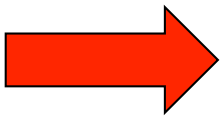
スカラ命令: 一度にひとつの演算をする
SIMD命令(ベクトル命令):複数の対象にたいして、同種の演算を一度に行う

実際にやること

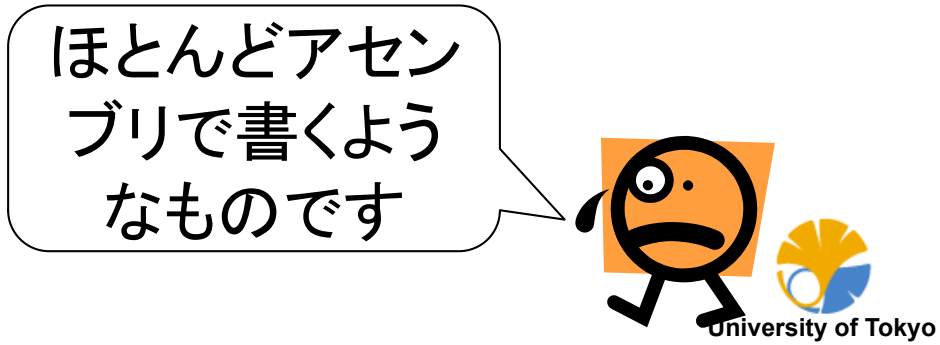
コンパイラがSIMD命令を出しやすいようにループを変形する

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 (2/2)

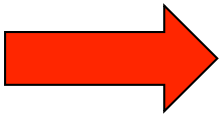
我々が実際にやったこと

作用反作用を使わない
→ 京、FX10ではメモリへの書き戻しがボトルネックだったため
(計算量が2倍になるが、速度が3倍になったので、性能が1.5倍に)

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

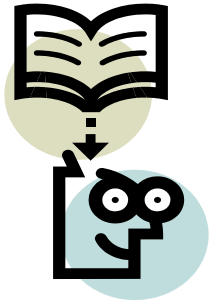


力の計算の速度は2倍に、全体でも30%程度の速度向上

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。

趣味の範囲を超えた**人外魔道チューニング**に踏み出すかどうかはあなた次第・・・

並列化

大規模並列で経験した困難をいくつか紹介

並列化への心構

「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

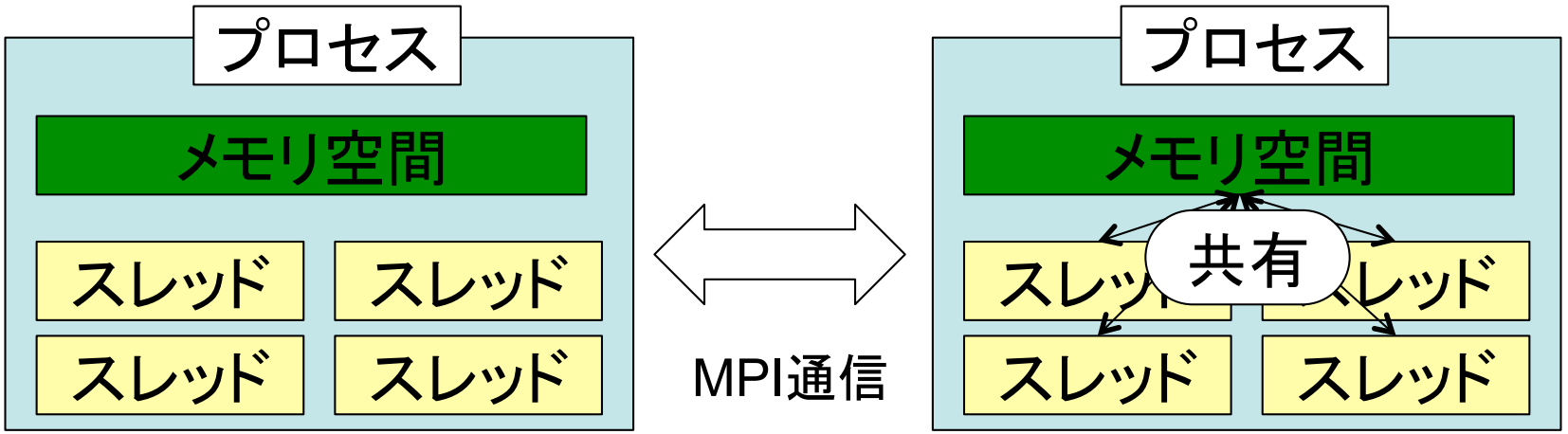
計算科学においては、「科学論文を書くこと」が一応のゴール
大規模なベンチマークに成功しても、それだけでは論文が書けない
「ベンチマークコード」を書くのと、それを「実用コード」にするのは
同じくらい時間がかかる

並列化の難しさは規模依存

経験的に、並列規模が10倍になると本質的に異なる困難に直面する

100並列で動いたから1000並列で動くとは限らない
1000並列で動いたから10000並列で動くとは限らない
それが並列計算

プロセスとスレッド



プロセス

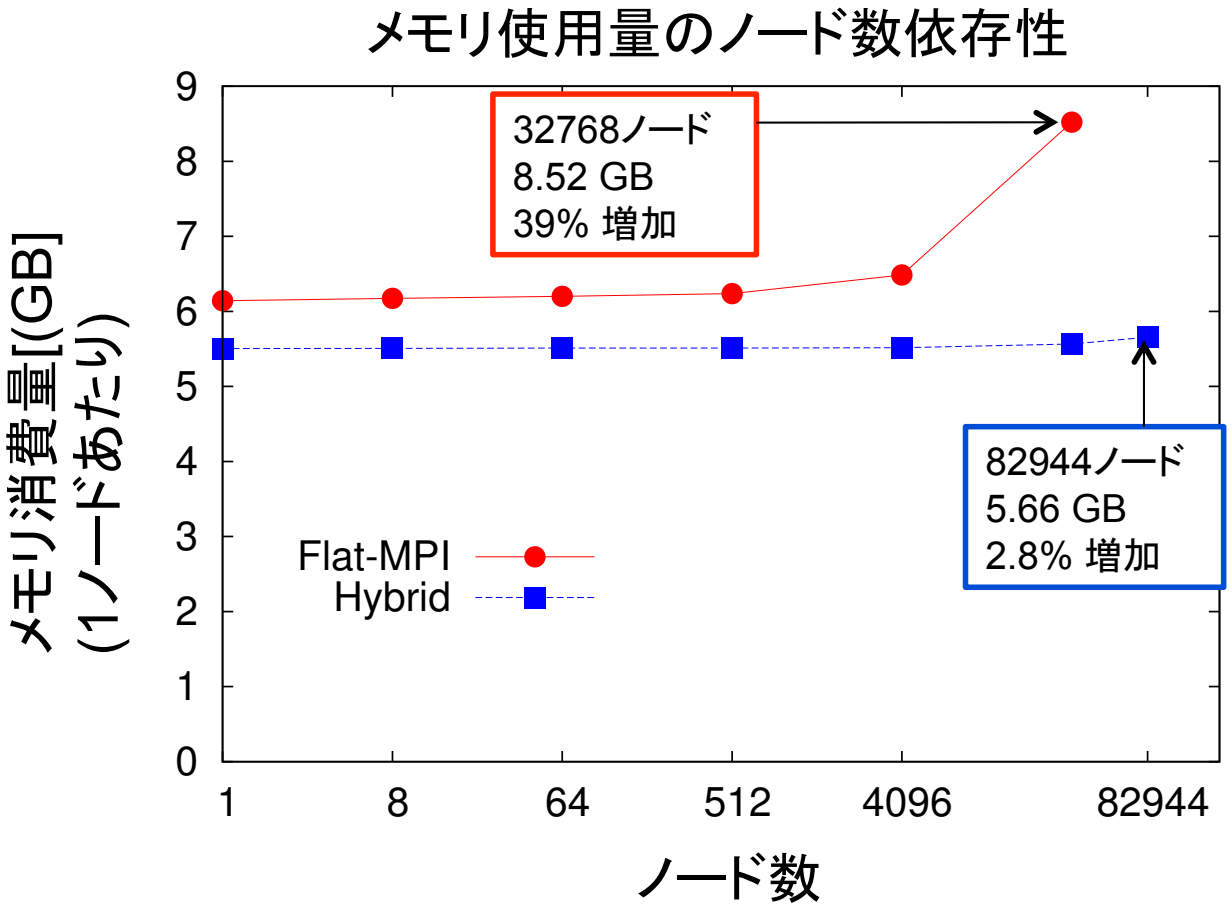
比較的重い (メモリ消費が激しい)
OSがリソース管理する単位
各プロセスは個別にメモリを持っている＝**通信が必要**
各プロセスは最低一本のスレッドを持っている

スレッド

比較的軽い
CPUに割り当てる仕事の単位
同じプロセスに所属するスレッドはメモリを共有＝**排他制御が必要**

Flat-MPI: 各プロセスが1本だけスレッドを持っている
ハイブリッド: 複数プロセスが、それぞれ複数本のスレッドを管理

Flat MPIとハイブリッド



計算条件

京コンピュータ
flat-MPI: 8プロセス/ノード
ハイブリッド: 8スレッド/ノード
400万粒子/ノード
最大 3318億粒子

Flat-MPIとハイブリッド、どちらが速いかは計算に依存
しかし、Flat-MPIはメモリ消費が激しく、超並列計算では事実上ハイブリッド一択

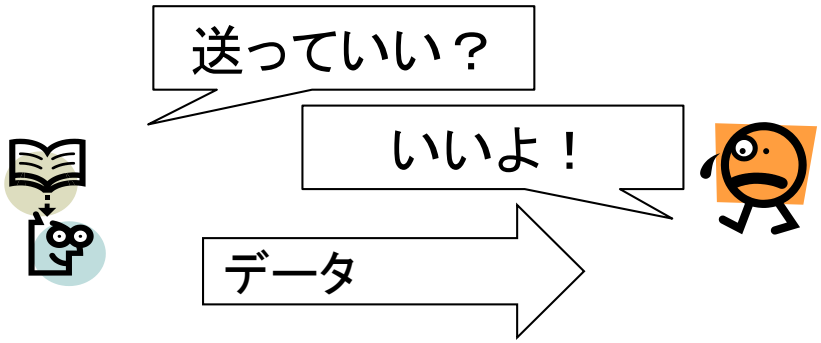
MPIのリソース枯渇

小さいジョブが成功したのに大きなジョブが失敗

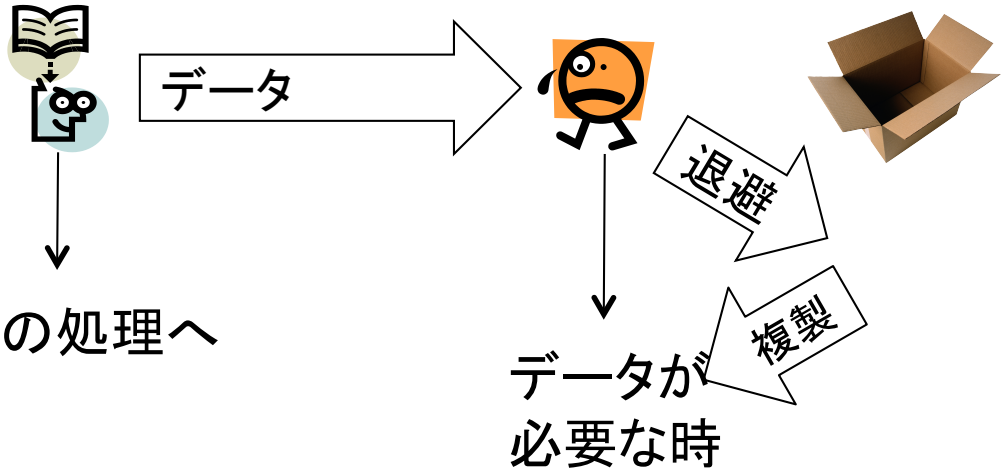
「MPI_なんとか_MAXが足りない」というエラーメッセージが多い

→ MPIの使うリソースの枯渇

Rendezvous(ランデブー)通信:
相手の準備が整うのを待ってから通信



Eager通信:
送信側：いきなり送りつける
受信側：とりあえずバッファに退避
必要になったらコピー



MPIの実装はベンダーに強く依存
例えばメッセージ長などで通信方法を切り替えている
ノンブロッキング通信を多用するとリソースが枯渇することが多い

→ 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・・・

OSジッタとハイパースレッディング (1/3)

通信がほとんど無いはずなのに、大規模並列時に性能が劣化して困る

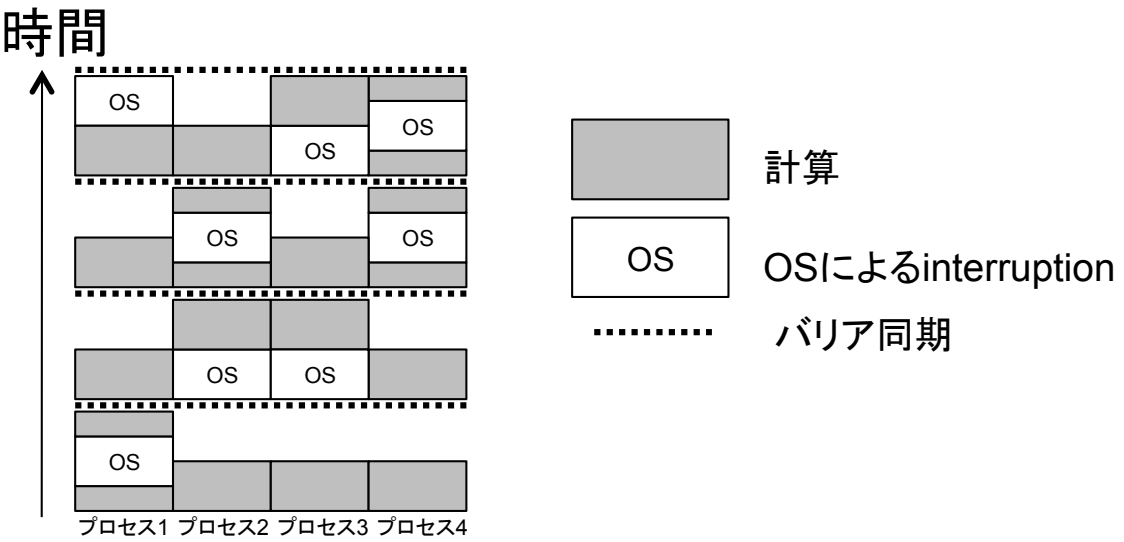
調べてわかったこと

(1) 力の計算時間を測定してみると、通信を含まないはずなのにプロセスごとに時間がばらついている
(2) 時間のばらつきはプロセス数を増やすと大きくなり、全体同期により性能劣化を招いている
(3) まったく同じ計算をしても、遅いプロセスは毎回異なる

OSジッタ

OSは計算以外にも仕事がある
その仕事が割り込んでくる
実効的なロードインバランスに
→性能が落ちる

計算が軽い時に顕著

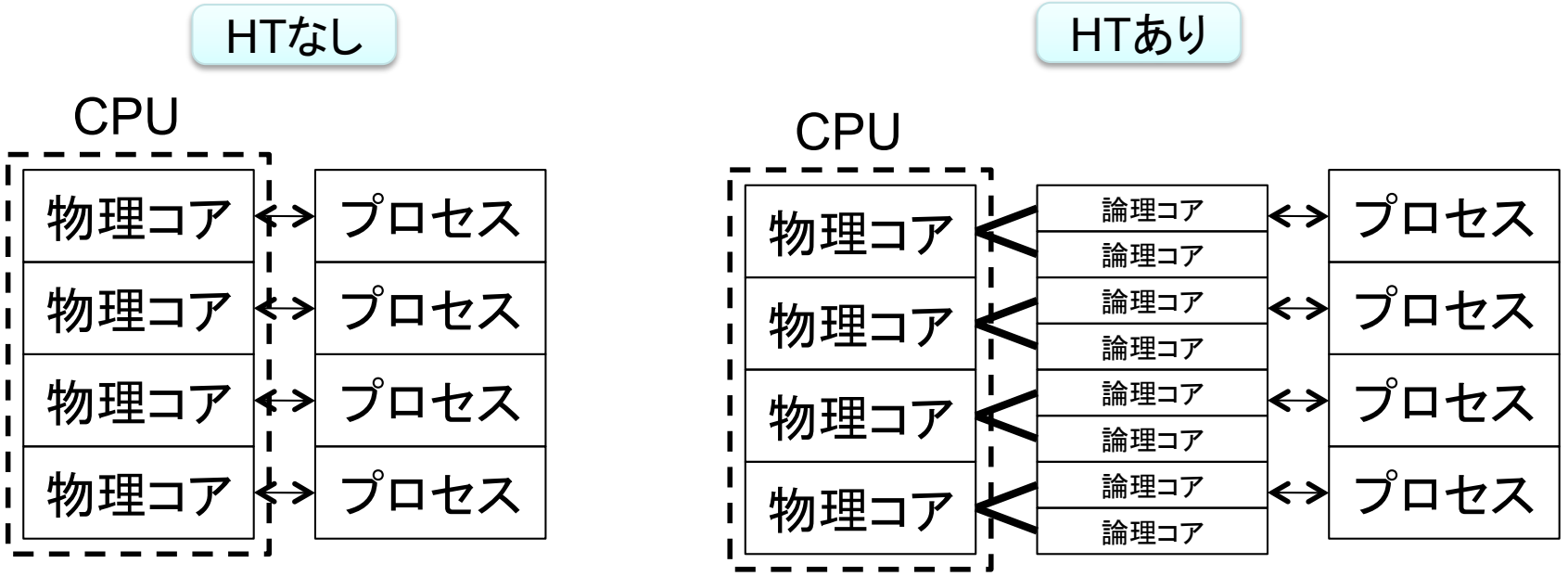


システムノイズ(OSジッタ)だろうか？
しかしOSジッタにしては影響が大きすぎる

OSジッタとハイパースレッディング (2/3)

ハイパースレッディング(HT)

OSから物理コアを論理的に二つ(以上)に見せる技術
→ 厨房の数は増やさず、窓口を増やす
OS由来ならHTの有無で性能が変わるはず？

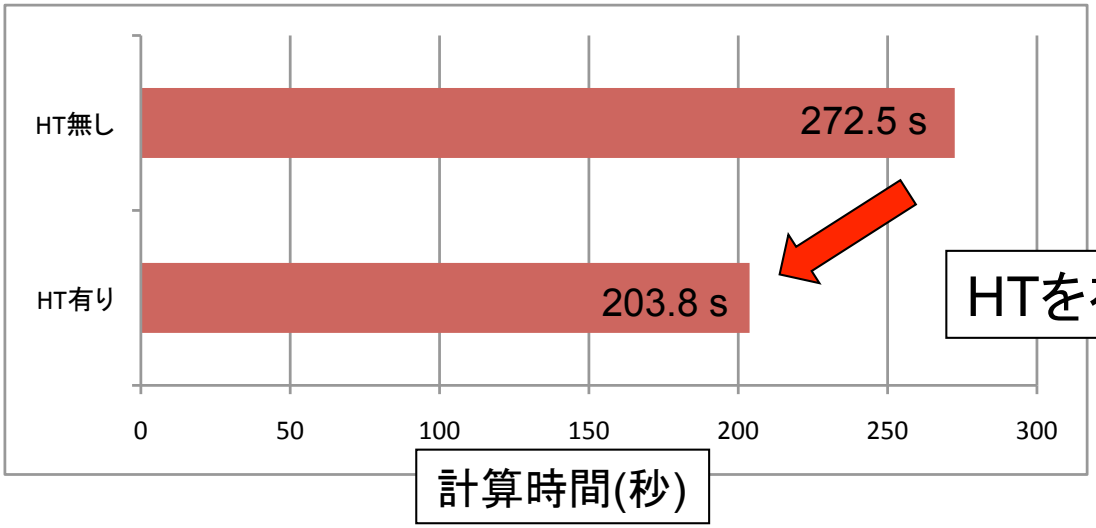


物理コアひとつにMPIプロセス一つをバインドする。

計算資源：東京大学物性研究所 システムB (SGI Altix ICE 8400EX) 1024ノード (8192プロセス)
詳細な条件などは以下を参照：
http://www.slideshare.net/kaityo256/130523-ht

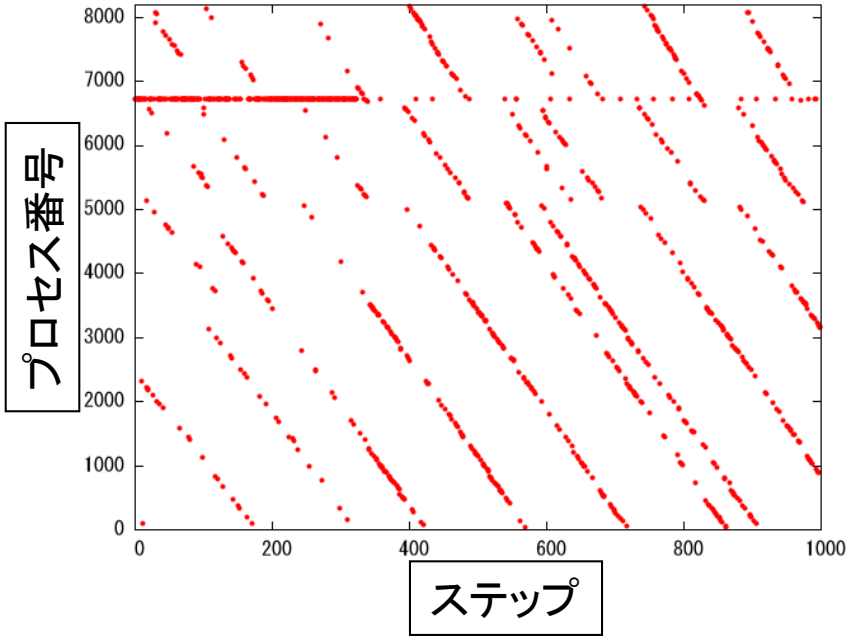
OSジッタとハイパースレッディング (3/3)

計算時間



HTを有効にするだけで性能が33%向上

各ステップで最も遅かったプロセス番号



ラウンドロビンで何かやってるらしい

通信の後処理が割り込んでいる？
なぜ大規模並列時のみ問題となるかは不明

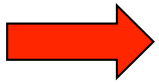
他に経験した事例

通信がほとんど無いはずなのに、大規模並列時に性能が劣化して困る Part 2

調べてわかったこと

(1) ハイブリッド実行時、特定のプロセスのみ実行が遅くなる(ことがある)
(2) Flat-MPIでは発生しない
(3) 1ノードでは発生しない、256ノード以上で高確率で発生
(4) 遅くなるプロセスは毎回異なるが、実行中は固定
(5) 利用していないオブジェクトファイルをリンクしたら発生

これ以上調べてもさっぱりわからなかったので、ベンダーに調査を依頼



原因はTLBミス

(1) ハイブリッド実行で、
(2) 256ノード以上で
(3) そのオブジェクトファイルをリンクした時の

使用メモリ量がちょうどTLBミスが頻発する条件に

→ ラージページの指定で部分的に解決

並列化のまとめ

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・・・あるんだろうか？
→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