計算科学技術 特論B

第1回
スーパーコンピュータと アプリケーションの性能

2026年4月

株式会社 waveZ
テクノロジーディレクター
南 一生

minami@wave-z.com

講義全体の概要

waveZ

- スーパーコンピュータとアプリケーションの性能
- アプリケーションの性能最適化１（高並列性能最適化）
- スレッド並列＋コア単体性能最適化概要
- スレッド並列＋コア単体性能最適化詳細（1）
- スレッド並列＋コア単体性能最適化詳細（2）

自己紹介

名前：南　一生 <博士（工学）>

所属：株式会社 waveZ

経歴：

- 富士通株式会社（1981年−2000年）
- システムエンジニア．主に原子力分野のシミュレーションコードの性能最適化および業務システムの開発に従事．
- 高度情報科学技術研究機構（RIST）（1981年−2013年）
- 地球シミュレータ上で地球科学分野・ナノ分野等多方面のアプリケーション開発．
- 理化学研究所（2008年−2021年）
- 「京」開発プロジェクトのアプリケーション開発チームリーダー
  →「京」上のアプリケーション開発．
- 計算科学研究機構・計算科学研究センター／ソフトウェア技術チームヘッド・チューニング技術ユニットリーダー
  →「京」の運用および「富岳」のアプリケーション開発．
- ジャパンメディカルデバイス株式会社（2021年−2025年）
- 文部科学省スーパーコンピュータ「富岳」成果創出加速プログラム「マルチスケール心臓シミュレータと大規模臨床データの革新的統合による心不全パンデミックの克服」プロジェクト所属

本日の講義内容

waveZ

スーパーコンピュータとアプリケーションの性能

- スーパーコンピュータとは？
- アプリケーションの性能とは？
- アプリケーションの超並列性を引き出す
- CPU/GPUでの対処

スーパーコンピュータとは？

スーパーコンピュータの発展

waveZ

1923年　タイガー手回し計算機

1946年　ENIAC
世界初のコンピュータ

1976年　CRAY-1
世界初のスーパーコンピュータ

160Mflops

40倍

2002年　当時のパソコン
PentiumIV
6.4Gflops

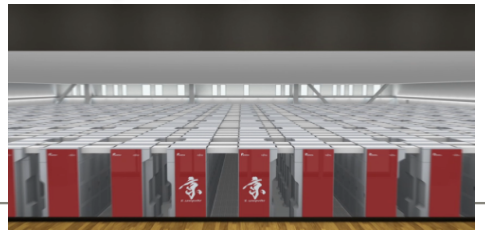
25万倍

2002年　地球シミュレータ
当時世界最速のスーパーコンピュータ

40Tflops

6250万倍

2011年
「京」コンピュータ



10Pflops

2011年　iPhone4S
LINPACK 140Mflops

≒

2020年　iPhone12
LINPACK 20550Mflops

≫

25億倍

2021年「富岳」コンピュータ
415.5Pflops

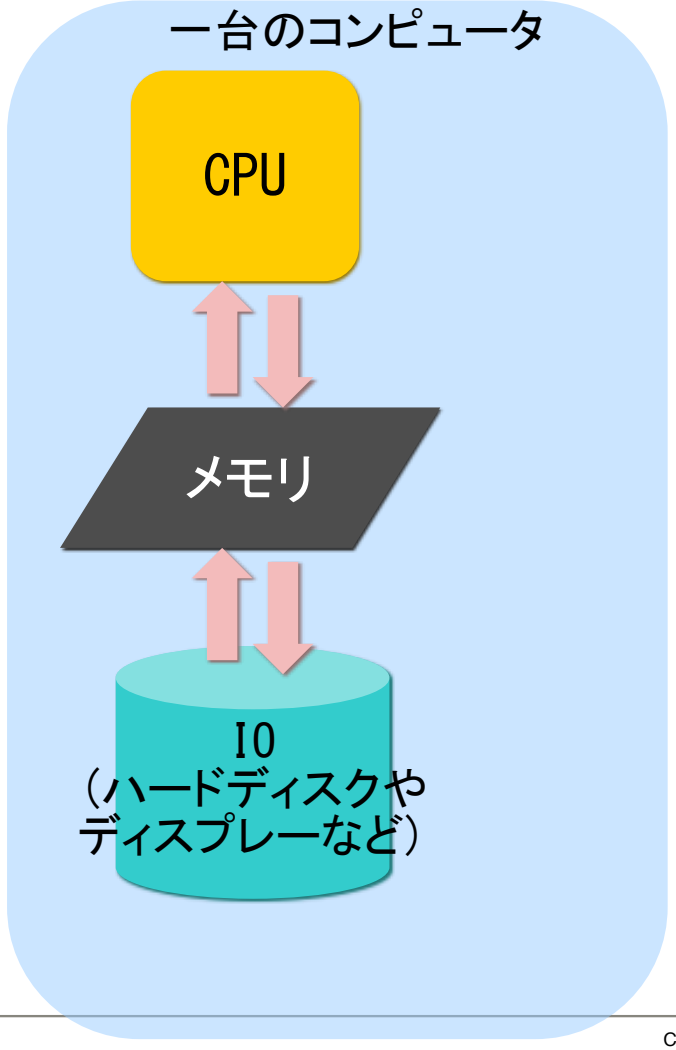
コンピュータとは？

waveZ

ハードウェア

- CPU
- メモリ
- IO（入出力）

IOから受け取った(入力)データとプログラムをメモリに置き、CPUでプログラムに従ってデータの処理を行なって、メモリに書き戻し、それをIOに書出す(出力)



コンピュータとは？

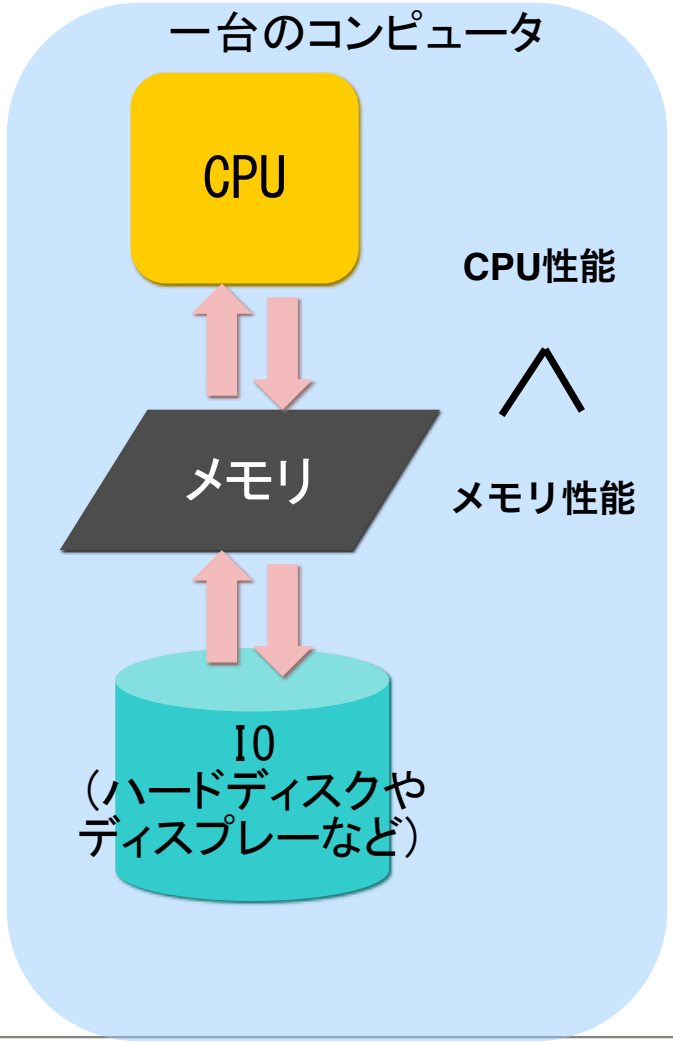
waveZ

ハードウェア
- CPU
- メモリ
- IO（入出力）

ソフトウェア(進化とともに分化)
- アプリケーション
- ミドルウェア
- OS

CPUの性能向上
≒コンピュータの性能向上

であった時代

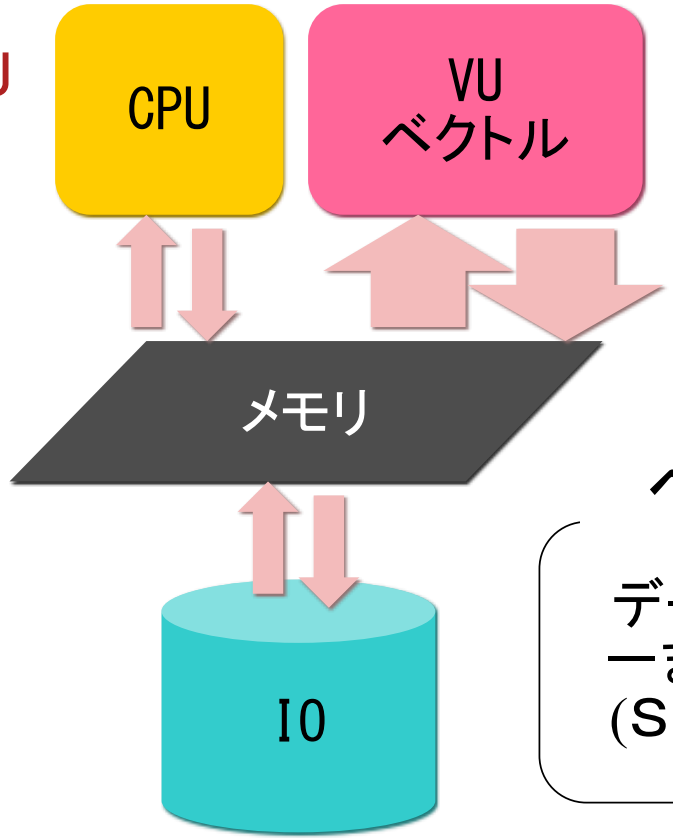


昔のスーパーコンピュータ

waveZ

昔は、ベクトル機構などによって、１台のコンピュータを高速化

- CPU(SU)＋VU
- メモリ
- IO(入出力)



一つのOSで制御される一つのコンピュータ

ベクトル機構

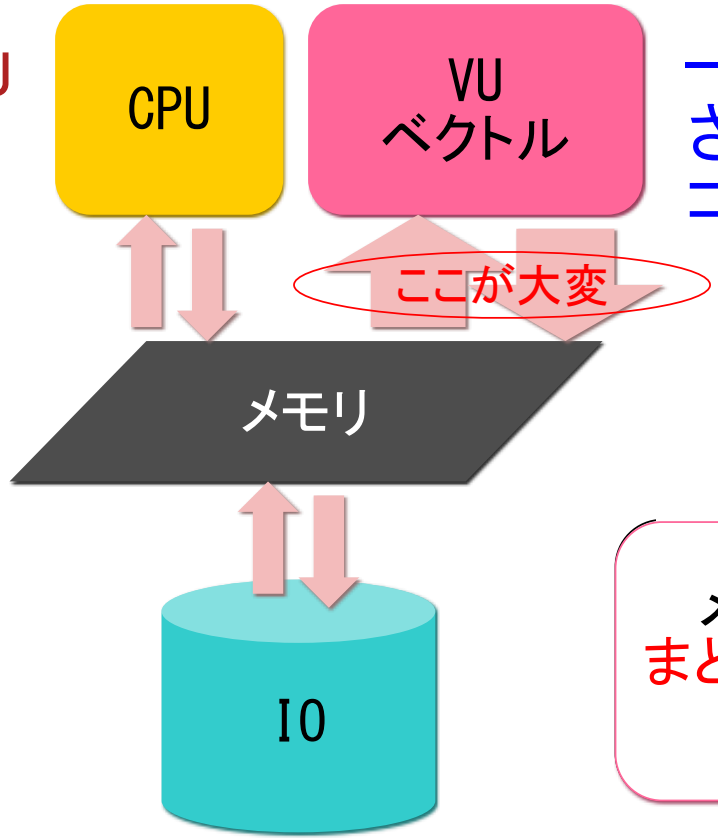
データのかたまりを一まとめで処理する
(SUとメモリを共有)

昔のスーパーコンピュータ

waveZ

昔は、ベクトル機構などによって、１台のコンピュータを高速化

- CPU(SU)＋VU
- メモリ
- IO(入出力)



一つのOSで制御される一つのコンピュータ

2002年　地球シミュレータ
当時世界最速のスーパーコンピュータ

40Tflops

ベクトル機構

メモリからデータを
まとめて持ってくるのは
とても大変

今もこの方式が無くなったわけではありませんが. . .

シングルプロセッサの問題

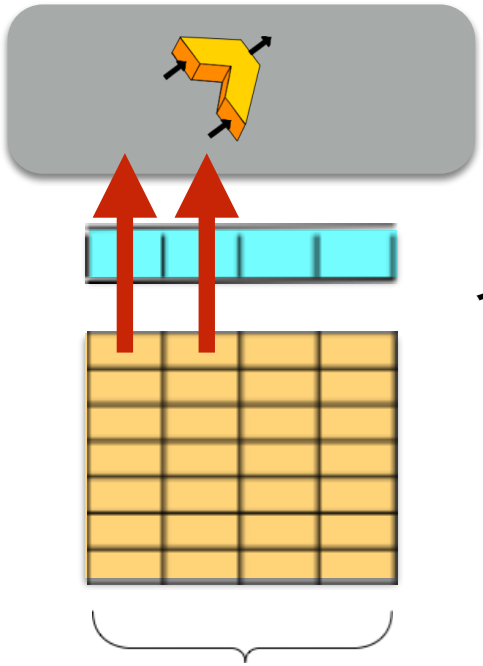
waveZ

メモリと演算器

- メモリは一定時間を経過しないと同じメモリバンクにアクセスできない
- 昔は20サイクルくらい待っていた
- しかし動作周波数が低かったため演算器も遅く演算速度とメモリ転送性能は釣り合っていた

動作周波数が低い場合(昔)

CPU性能 ≒ メモリ性能



シングルプロセッサの問題

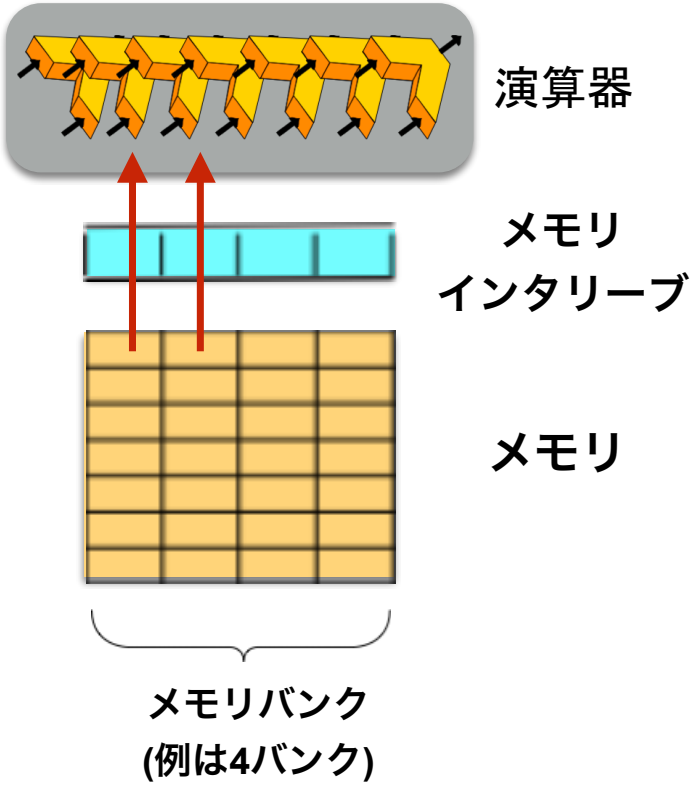
waveZ

メモリウォール問題

- メモリは動作周波数が高くなってくるともっと待つ事となる(100–200サイクル)
- メモリに比べて演算器は動作周波数が高くなると高速になった
- さらに半導体プロセスの微細化により演算器はCPUにたくさん搭載可能となった
- 結果的に演算器に比べメモリのデータ転送能力が低くなった

動作周波数が高い場合

CPU性能 ＞ メモリ性能

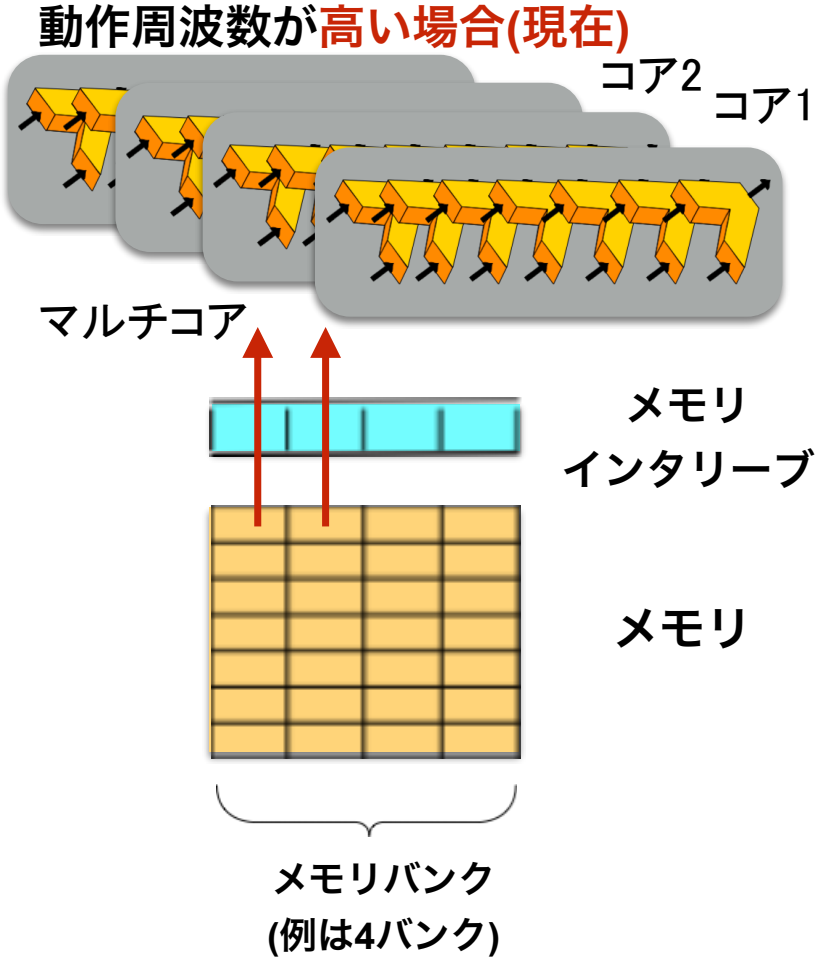


シングルプロセッサの問題

waveZ

メモリウォール問題

- メモリは動作周波数が高くなってくるともっと待つ事となる(100–200サイクル)
- メモリに比べて演算器は動作周波数が高くなると高速になった
- さらに半導体プロセスの微細化により演算器はCPUにたくさん搭載可能となった
- 結果的に演算器に比べメモリのデータ転送能力が低くなった

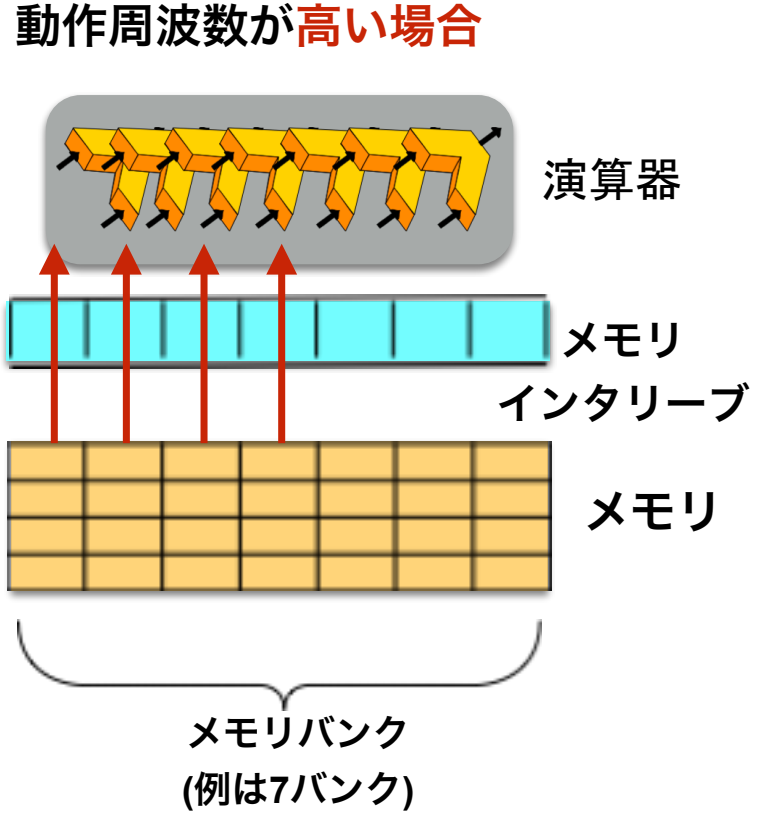


シングルプロセッサの問題

waveZ

対策1

- メモリバンクを増やし演算器に供給するデータ量を増大させた
- しかしこの方式は機構が複雑になり電力が増大する
- またコスト・価格も高くなる
- ベクトル機では数百のバンクを搭載した
  - ちなみに昔のベクトル機はこの方式により1要素(8バイト)単位のメモリアクセスで高いメモリアクセス性能を実現した
  - しかしここで述べたようにコスト・価格・電力面では高価である



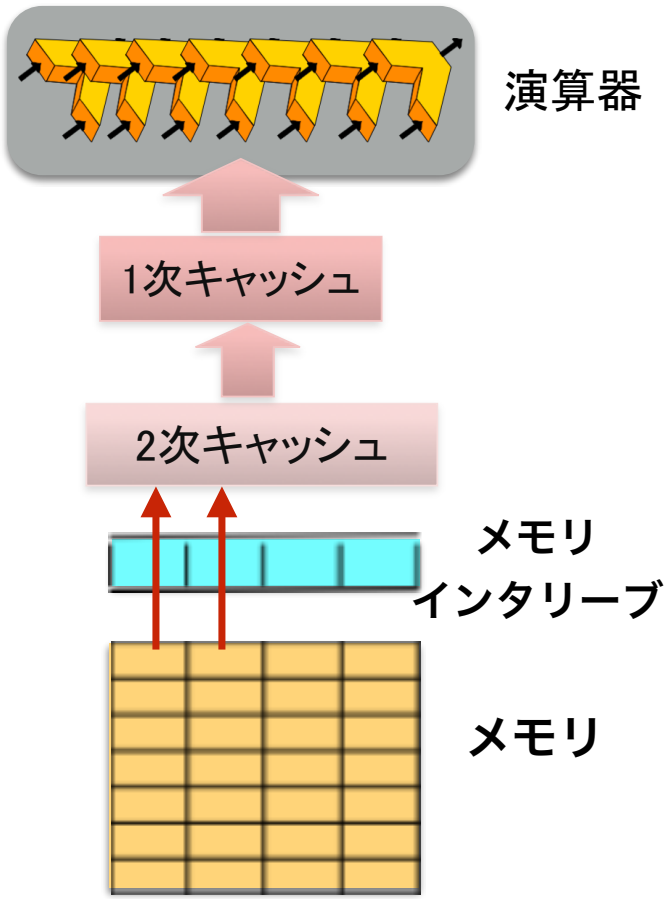
シングルプロセッサの問題

waveZ

対策2

- メモリのバンクは増やさない
- データ供給能力の高いキャッシュをメモリと演算器間に設ける
- データをなるべくキャッシュに置きデータを再利用する事でメモリのデータ供給能力の不足を補う
- こうすることで演算器の能力を使い切る
  - 多くの計算機はこの方式を取っている
  - キャッシュライン(数十から数百バイト)単位のメモリアクセスである

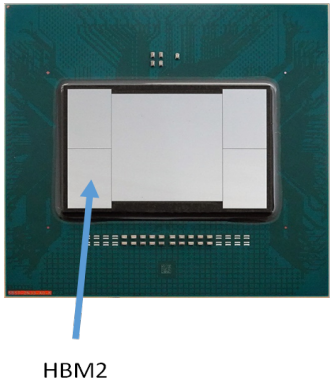
動作周波数が高い場合

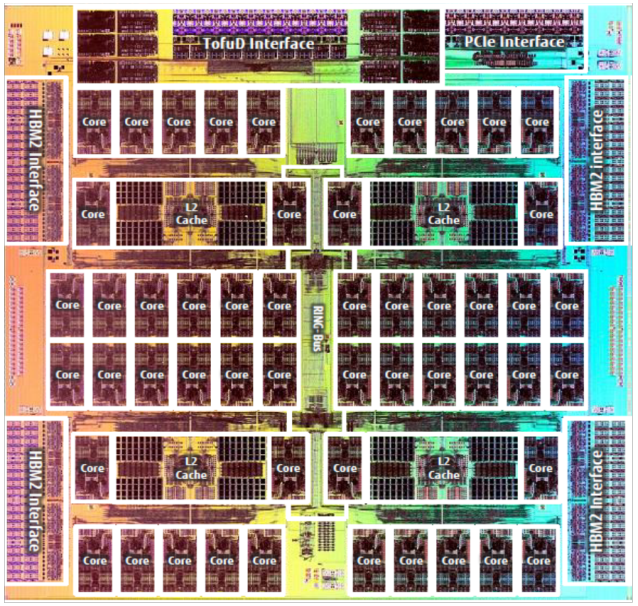


シングルプロセッサの問題

対策3：メモリとCPUの接続を強化する

A64FXプロセッサ





このチップの大きな特徴は、HBM による非常に高いメモリバンド幅である。HBM は、シリコン基盤上に、積層メモリとCPUチップをSi貫通電極 (TSV: through-silicon via) により接続するメモリ技術であり、HBMの最新規格のHBM2のメモリを使い、1つの積層メモリあたり256GB/sのメモリバンド幅を持つ。4つの積層メモリが接続されており、全体では32GiBの容量、1TB/sのメモリバンド幅を持つ。

(出典:富岳コデザインレポート)

シングルプロセッサの問題

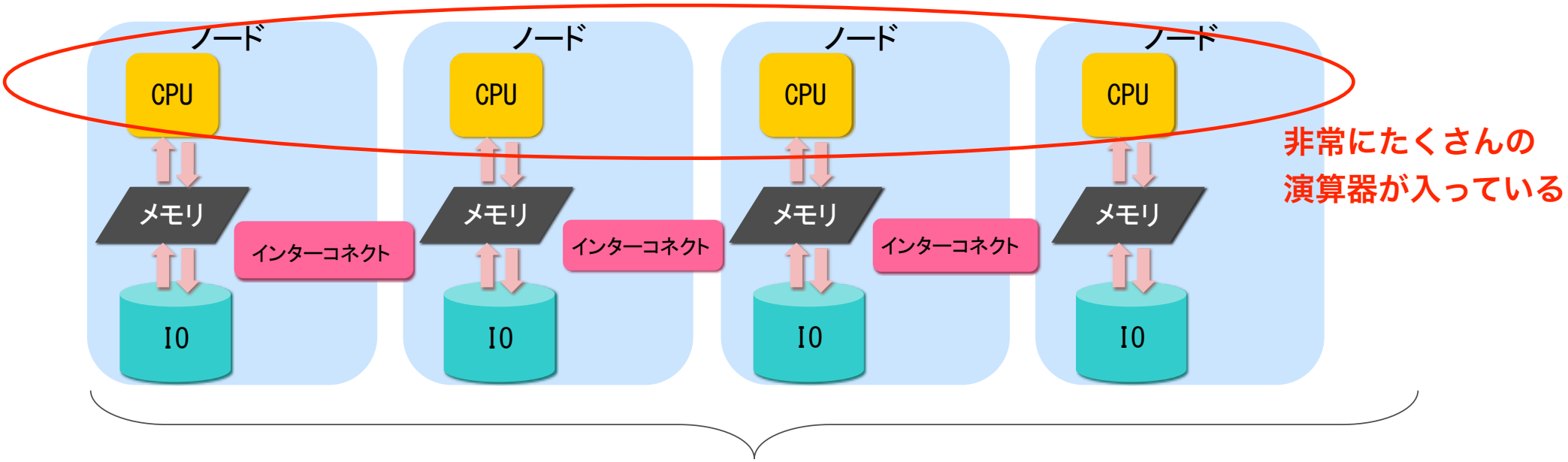
waveZ

一台のコンピュータの処理限界

- 動作周波数を上げる事で電力が吹き上がる
- 動作周波数の限界を迎えている
- メモリウォール問題もあり一台のコンピュータの演算能力を上げてもメモリの能力が追いつかない

そこで！

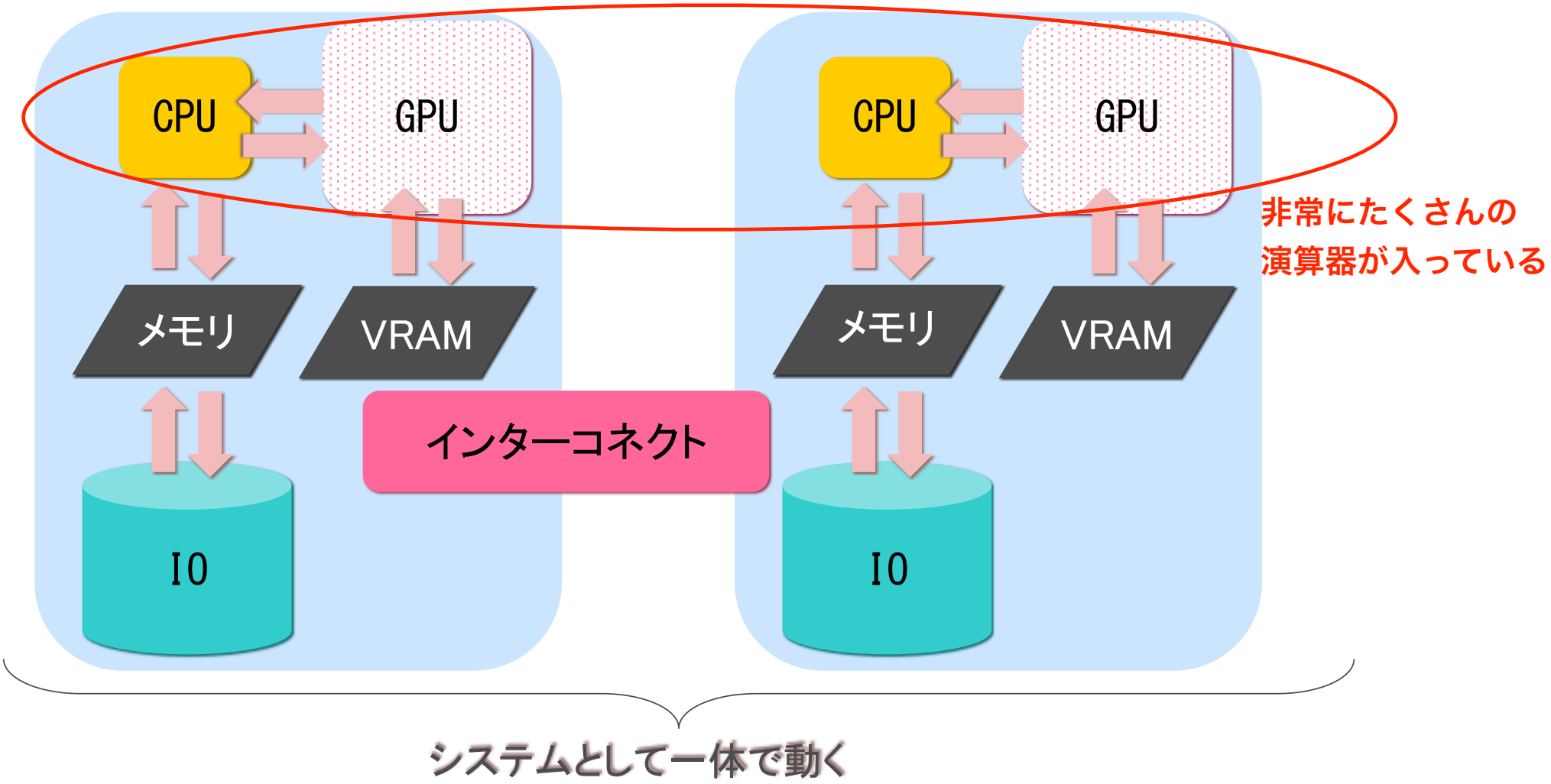
CPUの場合- ノード同士を繋ぐ



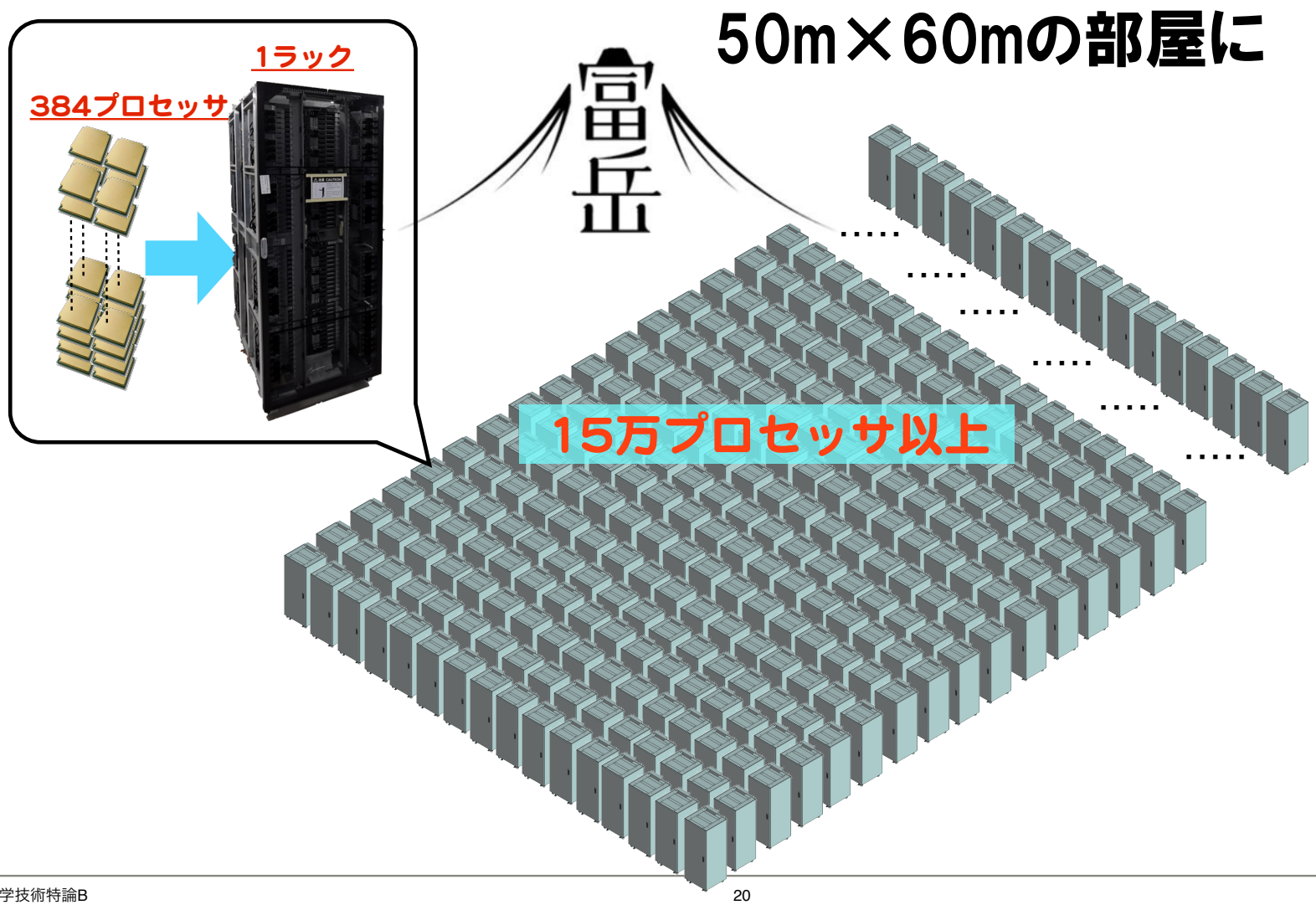
システムとして一体で動く

（一つの仕事をやる、たくさんの仕事は互いに連携して分担しあって片づける）

GPUの場合- GPUを追加&ノード同士を繋ぐ



CPUシステム「富岳」の場合



スーパーコンピュタとは？まとめ

waveZ

- 1940年代半ばのENIACの登場
- 最初はシングルプロセッサの時代
- ベクトル/CISC/RISC/スーパースカラ等色々なアーキテクチャが登場
- その時代は演算器を増やすことにより高速処理を実現.
- メモリウォール問題及び電力，動作周波数を上げられない問題によりシングルプロセッサの限界となった.
- それらによりスーパーコンピュータが並列プロセッサアーキテクチャへと変化

- CPU/GPUシステムともに非常に多くの演算器を持っておりメモリウォール問題を抱えているのが現代のスーパーコンピュータである

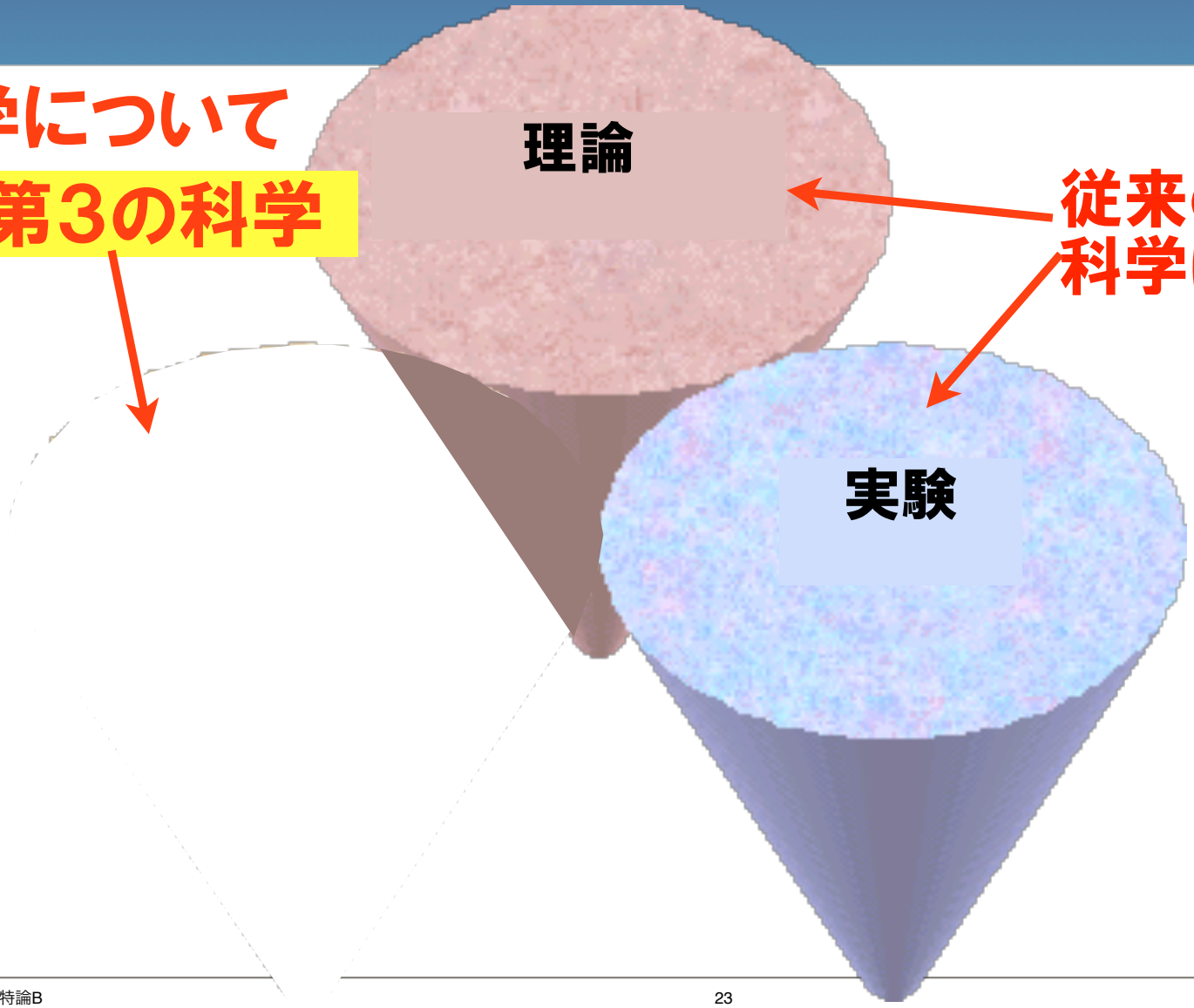
アプリケーションの性能とは？

スーパーコンピュータは　　シミュレーションに使う

waveZ

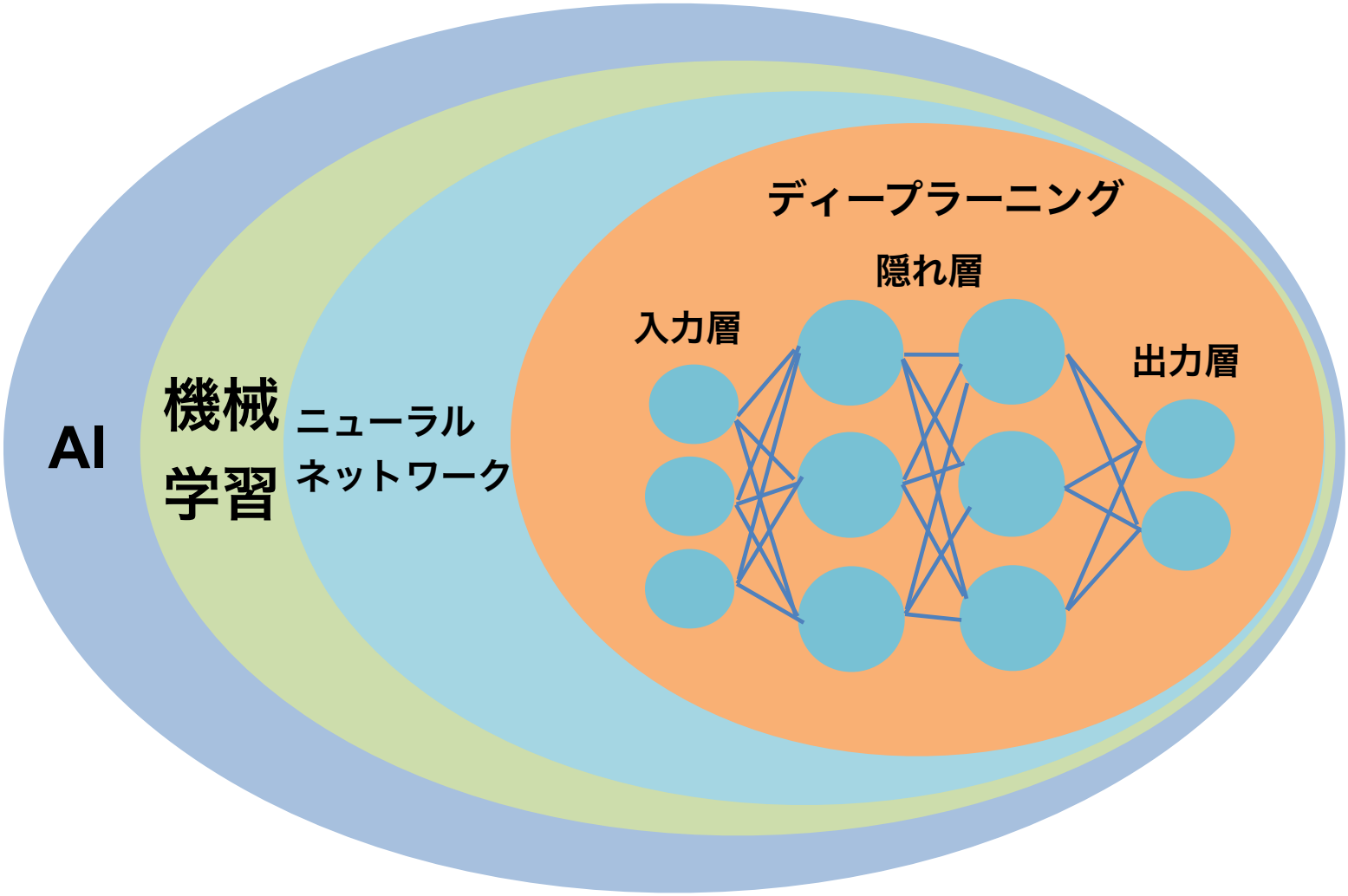
科学について

第3の科学



従来の
科学は

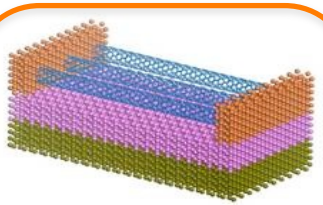
近年はAIにも使われる



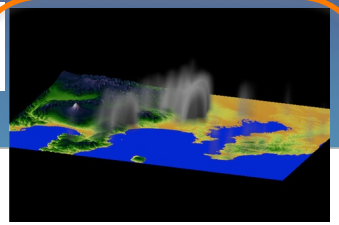
具体的なアプリケーション

waveZ

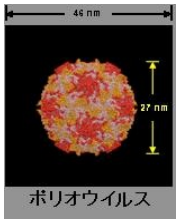
シミュレーション分野



次世代のデバイス全体のシミュレーションによるエレクトロニクスへの貢献



難しい大気現象の解明、また正確な台風の進路・強度の予測による気象への貢献



水中のウィルスの丸ごとシミュレーションによる医療への貢献

10^{21}m

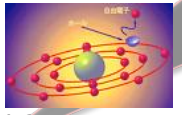
10^{7}m

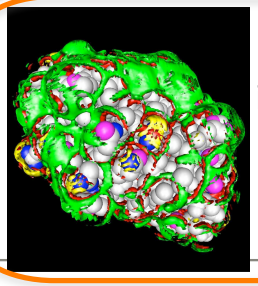
10^{2}m

10^{0}m

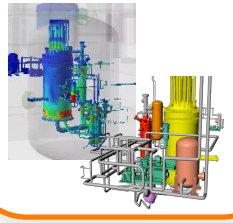
10^{-8}m

10^{-10}m



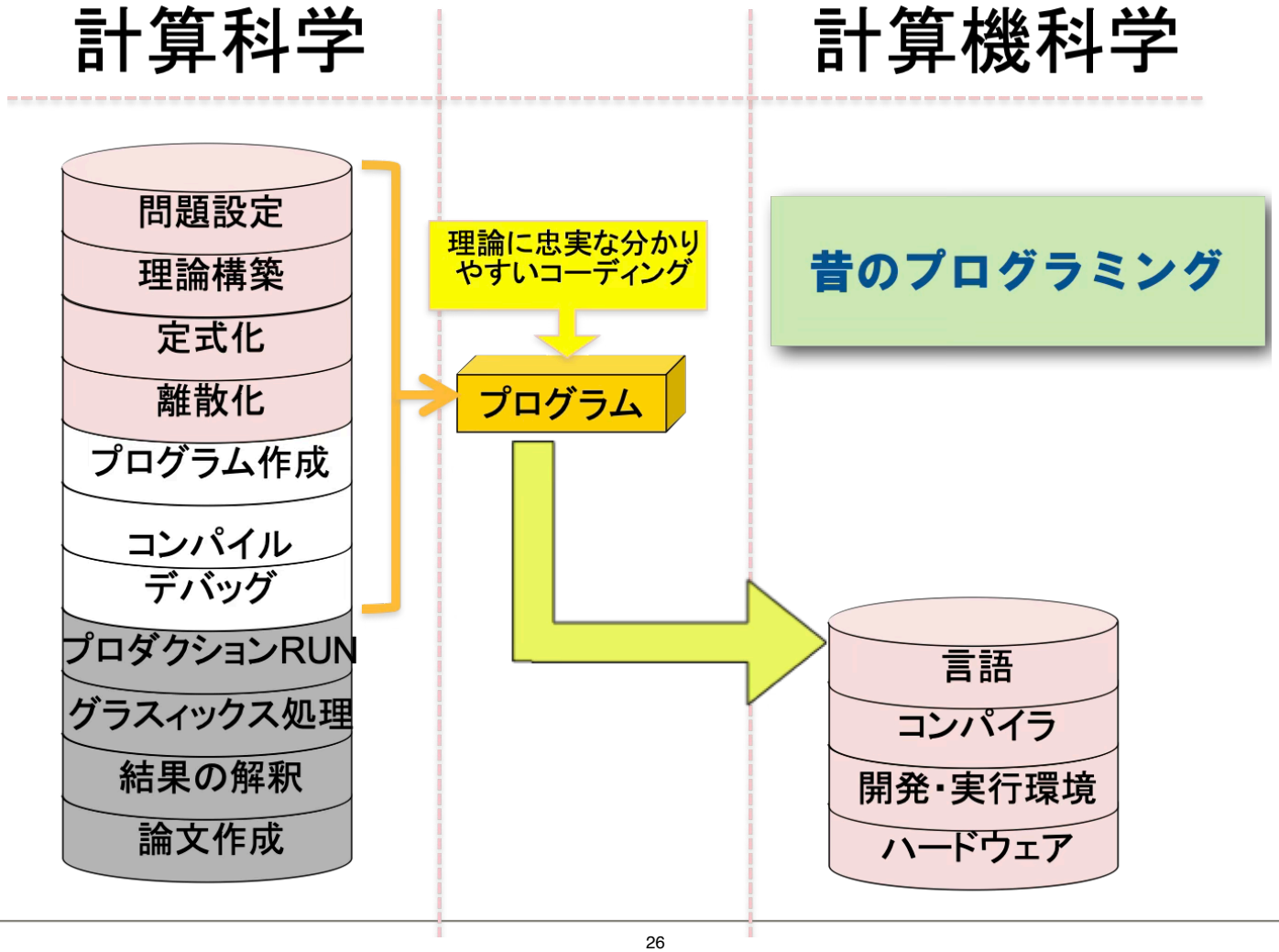


セルロース分解酵素のシミュレーションによる安価なバイオ燃料の提供等エネルギーへの貢献

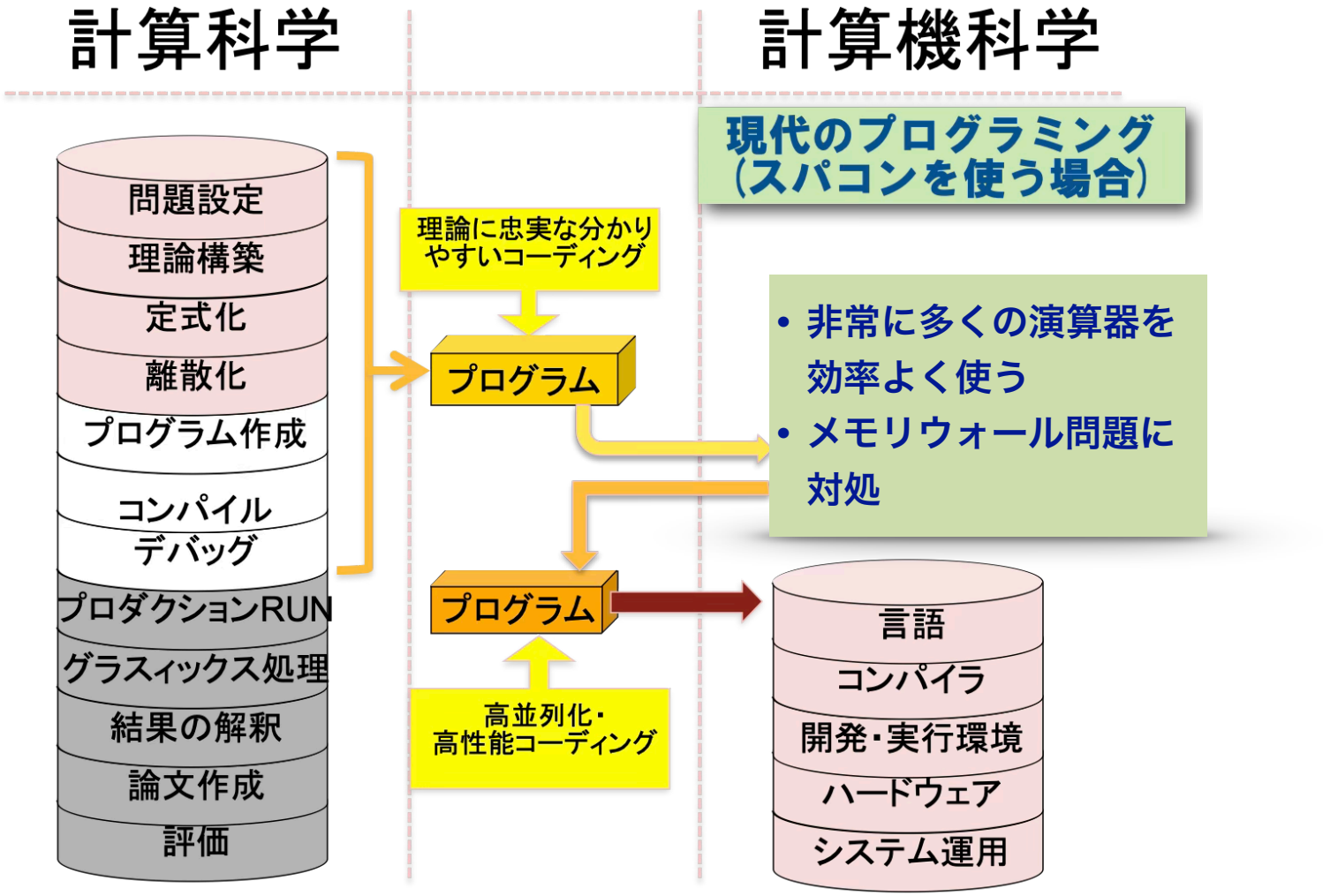


短周期地震波動の地震波シミュレーション、構造物の耐震シミュレーションを組み合わせた防災への貢献

現代のスパコン利用の難しさ

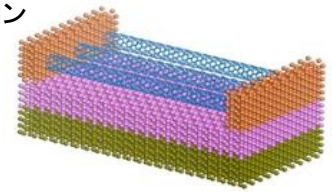


現代のスパコン利用の難しさ



スーパーコンピュータの威力

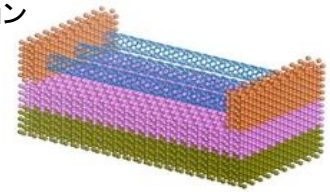
例えば次世代のデバイス
全体のシミュレーション

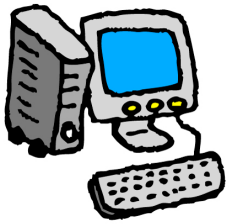


たくさんの原子を用いたシミュレーション

スーパーコンピュータの威力

例えば次世代のデバイス
全体のシミュレーション

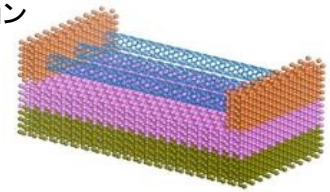


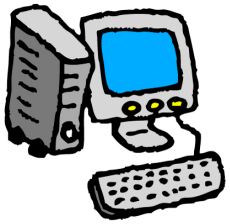


パソコンで100年かかる計算

スーパーコンピュータの威力

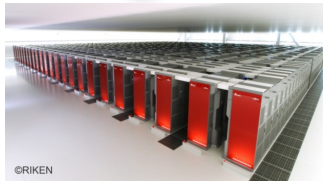
例えば次世代のデバイス
全体のシミュレーション





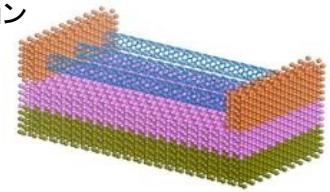
パソコンで100年かかる計算

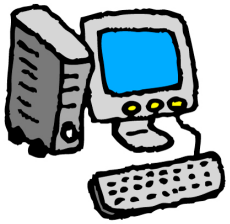
京を使って
73000倍



スーパーコンピュータの威力

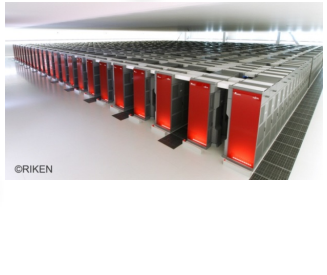
例えば次世代のデバイス
全体のシミュレーション





パソコンで100年かかる計算

京を使って
73000倍



2011年
ゴードンベル賞受賞

半日で終わる

アプリケーション性能についてまとめ

waveZ

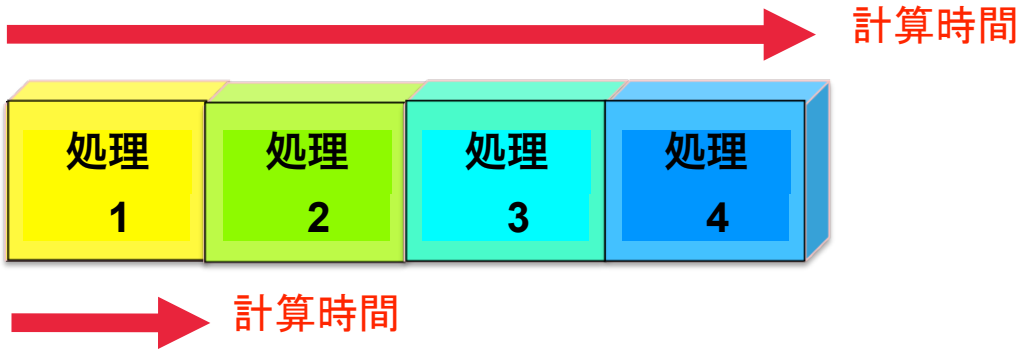
- プログラマーは、非常に多くの演算器を使えるように**アプリケーションの並列性**を最大限利用したプログラミングを行う必要がある
- また遅いメモリ・多数の演算器を特徴とした**メモリウォール問題に対処したプログラムのチューニング**も必要となる

アプリケーションの超並列性を引き出す（CPU/GPU共通）

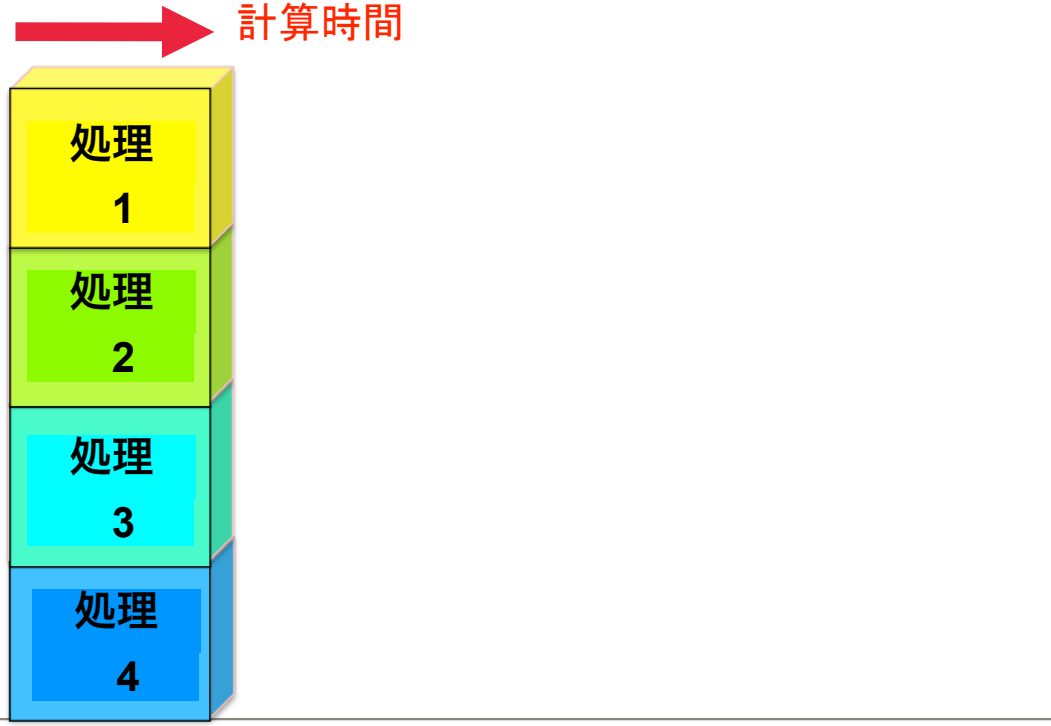
そもそも並列化とは

waveZ

逐次計算



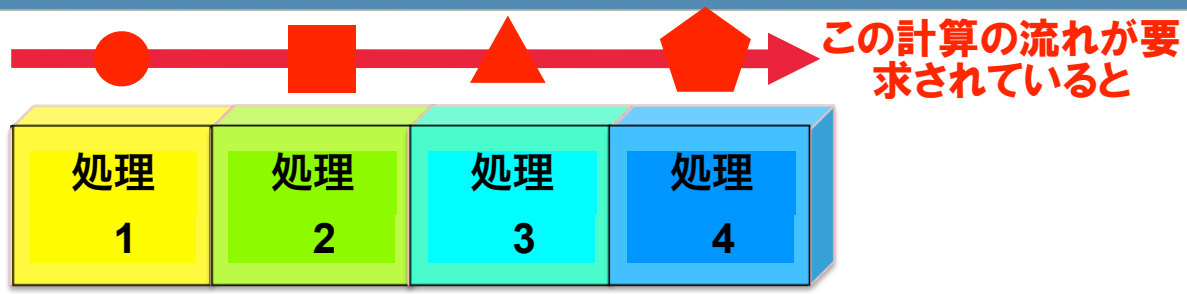
並列計算



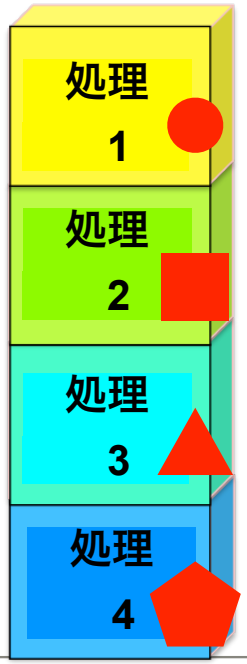
並列化できる条件

waveZ

逐次計算



並列計算



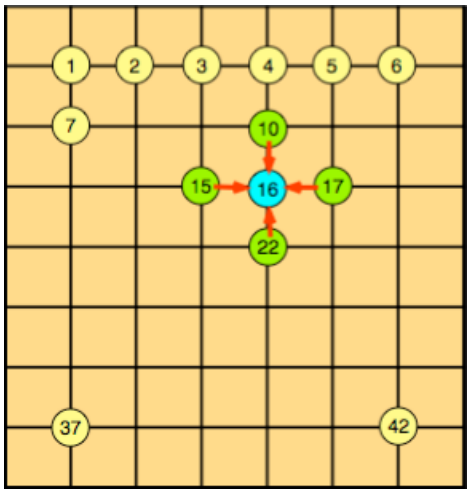
同時計算不可！

上記のような計算の流れを要求しない並列計算可能なアルゴリズムを採用する必要がある！

並列化できるアルゴリズムを採用する

waveZ

2次元偏微分方程式-離散化



- 左図のような2次元の領域で微分方程式を解くとする.
- 5点差分で差分化すると自分自身の差分点と周りの4つの差分点の式ができる.

$$C_{1,3}u_1 + C_{1,4}u_2 + C_{1,5}u_7 = \alpha_1 \quad \text{①の差分点について}$$

$$C_{2,2}u_1 + C_{2,3}u_2 + C_{2,4}u_3 + C_{2,5}u_8 = \alpha_2 \quad \text{②の差分点について}$$

. . . . . .

$$C_{16,1}u_{10} + C_{16,2}u_{15} + C_{16,3}u_{16} + C_{16,4}u_{17} + C_{16,5}u_{22} = \alpha_{16} \quad \text{⑯の差分点について}$$

. . . . . .

- 全部で42点の差分点が存在するので42元の連立方程式となる.
- したがって元の微分方程式は，離散化により係数を作り、以下のような連立一次方程式を解く事に帰着させる事ができる.

$$\mathbf{Ax = b}$$

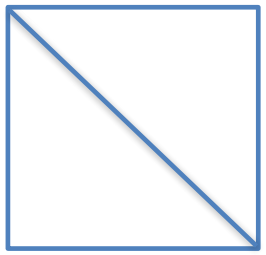
並列化できるアルゴリズムを採用する

waveZ

連立一次方程式の解法

陽解法

A=



対角行列

同じ時間の別の点を参照しない.

並列化できる

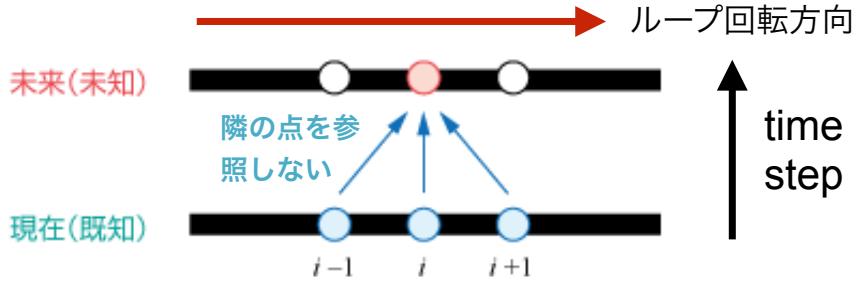
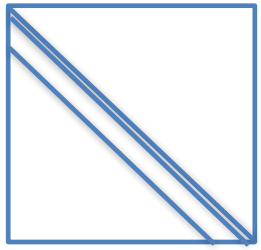


図6.42　陽解法の計算イメージ

陰解法

A=

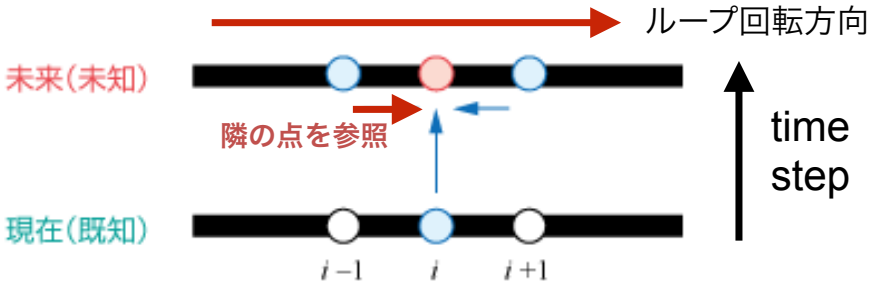


帯行列

同じ時間の別の点を参照する.

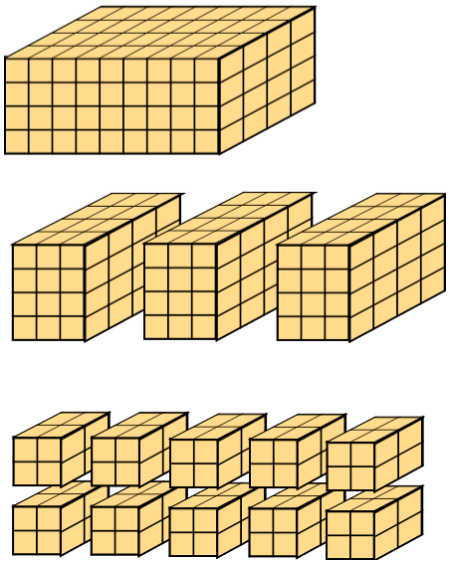
並列化できない！

陰解法の並列化は次回！



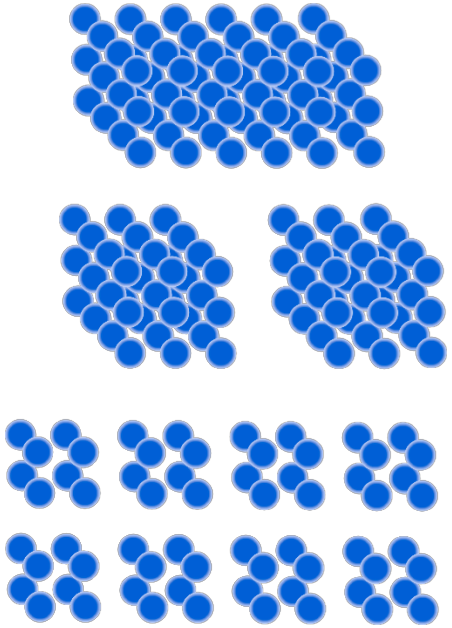
並列化の物理的イメージ

領域分割



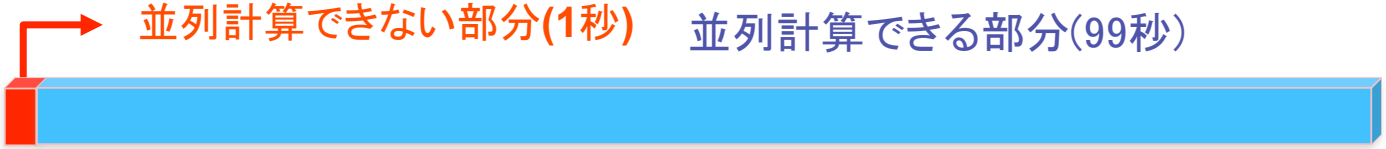
・原理的にメッシュ数以上には分割できない
・実際的にはそんなに分割すると非効率となる
・以下の手順で分割数を見積もる事が重要
(1) 解きたいメッシュ数を設定し (2) 実行時間を見積もる
(3) 解きたい時間を設定し (4) 分割数を設定する
(5) 分割数が多すぎる場合，並列数の拡大を検討
(5) については全てのケースでできる訳ではないが後の講義でテクニックを例示する．

原子分割

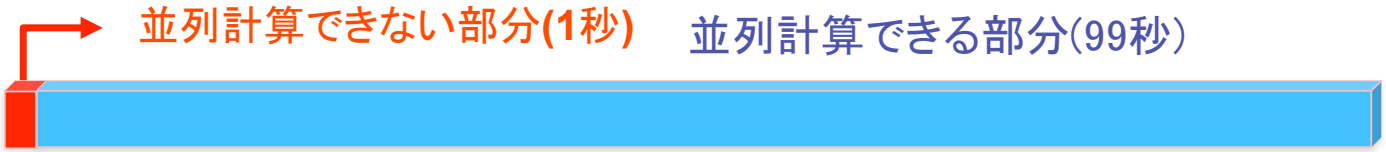


・原理的に原子数以上には分割できない
・実際的にはそんなに分割すると非効率になる
・後は領域分割と同様

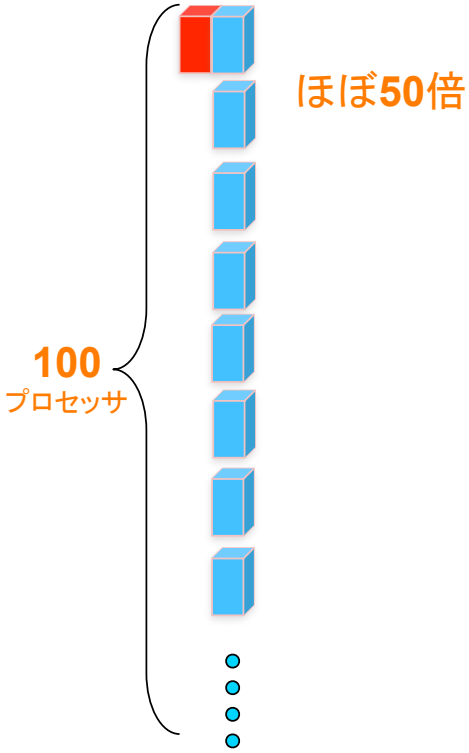
非並列部分を極力なくす



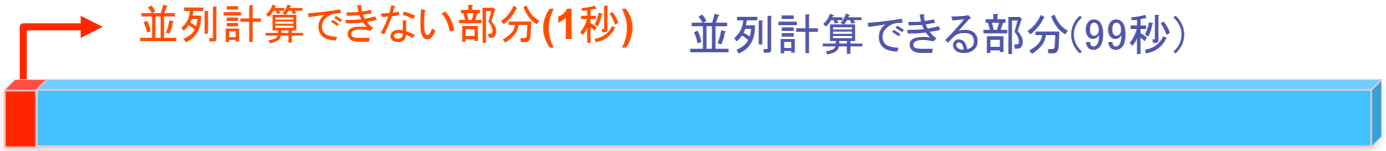
非並列部分を極力なくす

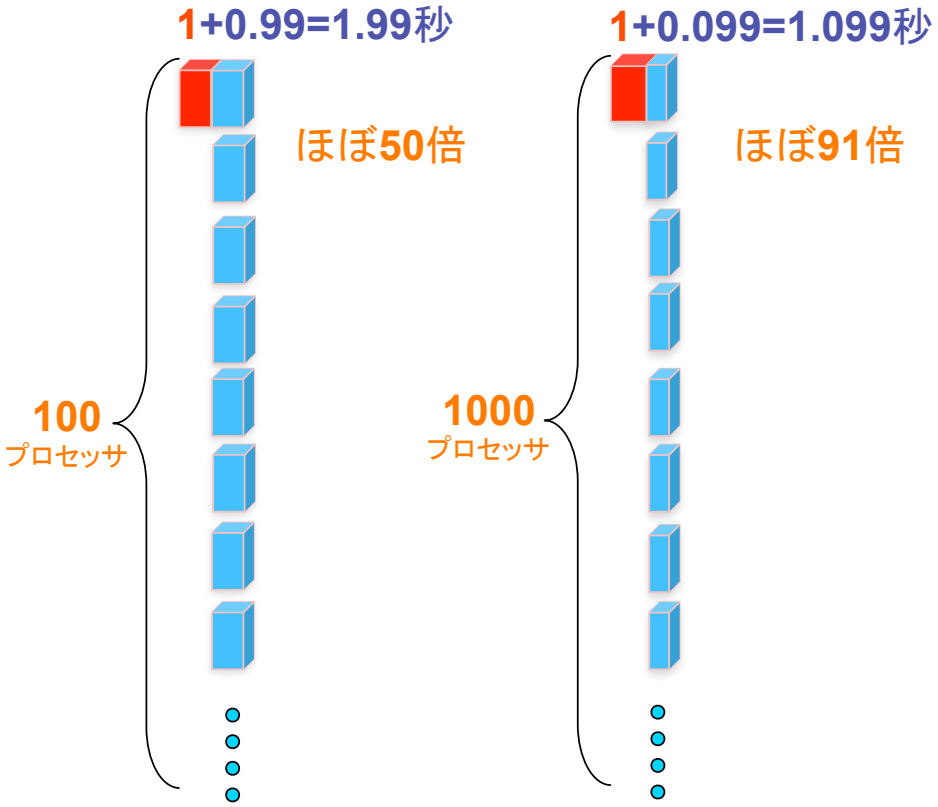


1+0.99=1.99秒



非並列部分を極力なくす



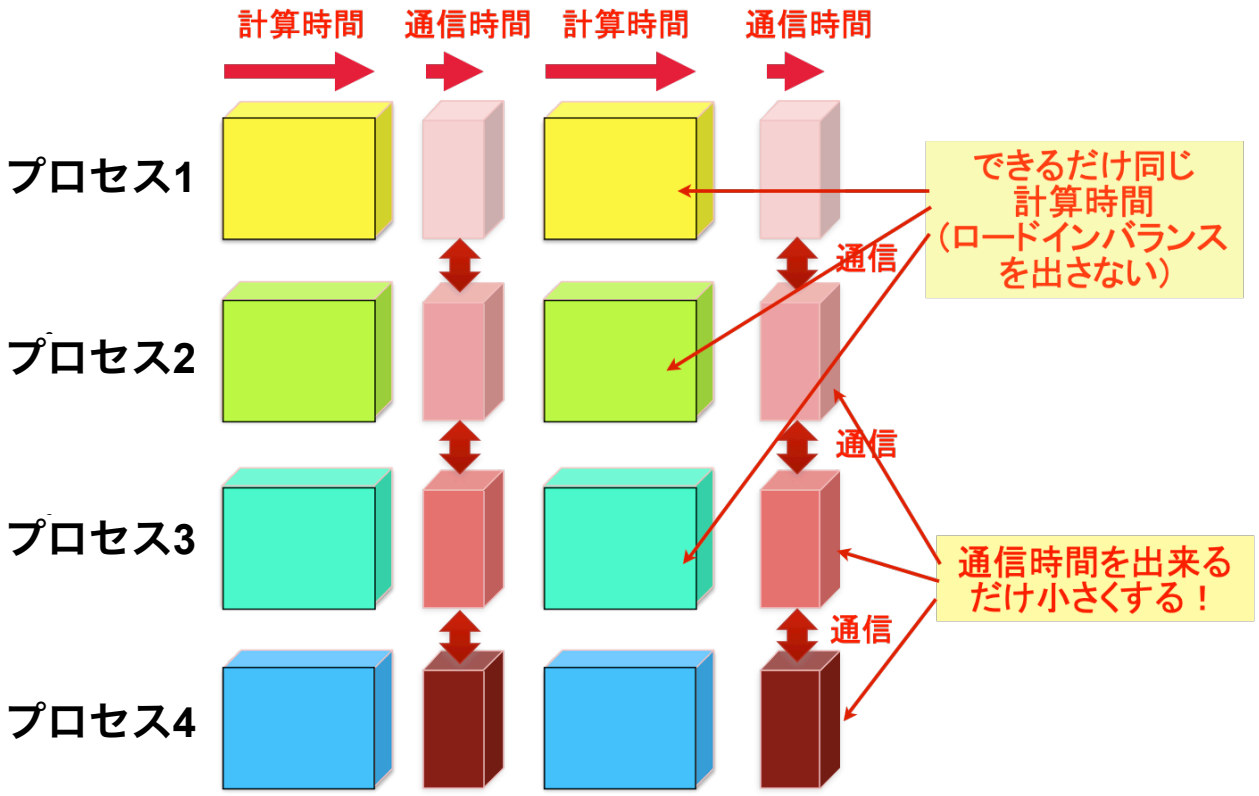


非並列部分が問題となる場合

- 領域分割の場合，完全に領域分割されていれば非並列部が発生する場合は少ない．
- 領域分割されていない配列や処理が主要な計算部に残る場合あり．
- また初期処理に分割されていない処理が残っている場合もある．具体的には通信テーブルの作成等．
- 量子計算のアプリ等で領域分割でなくエネルギーバンド並列等を使用している場合は非並列部が残る場合がある．

```
subroutine m_es_vnonlocal_w(ik,iksnl,ispin,switch_of_eko_part)
   +-call tstatc0_begin
   | loop_ntyp: do it = 1, ntyp
   |  loop_natm : do ia = 1, natm ---------原子数のループ
   |   +-call calc_phase
   |   | T-do lmt2 = 1, ilmt(it)
   |   | +-call vnonlocal_w_part_sum_over_lmt1
   |   |   +-call add_vnlph_l_without_eko_part
   |   |     subroutine add_vnlph_l_without_eko_part()
   |   |     | T-if(kimg == 1) then
   |   |     |   T-do ib = 1, np_e  ---------エネルギーバンド並列部
   |   |     |   | T-do i = 1, iba(ik)
   |   |     |   | V-end do
   |   |     |   V-end do
   |   |     | +-else
   |   |     |   T-do ib = 1, np_e  ---------エネルギーバンド並列部
   |   |     |   | T-do i = 1, iba(ik)
   |   |     |   | V-end do
   |   |     |   V-enddo
   |   |     | V-end if
   |   |     end subroutine add_vnlph_l_without_eko_part
   |   | V-end do
   |  V-end do loop_natm
   | V-end do loop_ntyp
end subroutine m_es_vnonlocal_w
```

プロセス並列



プログラミングイメージ

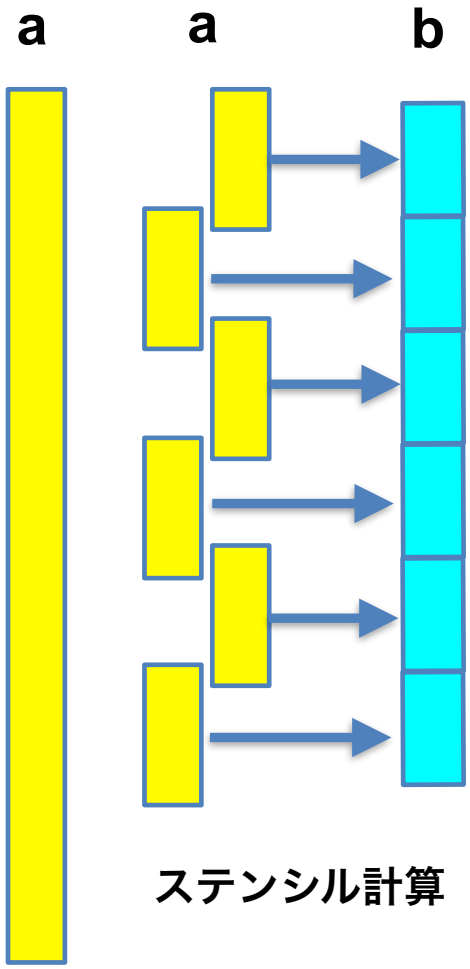
```
    call MPI_COMM_RANK(..,ip,…)
```
→ipに自分のプロセッサ番号取得

```
    do i=1,10
      array(i,ip)=・・・
    end do
```
→自分のプロセッサ分を計算し配列に格納

```
    ipp=ip+1
    ipm=ip-1
```
→ipから通信相手のプロセッサ番号を計算

```
    call MPI_Send(..,ipp,..array(10,ip),..)
    call MPI_Recv(..,ipm,..array(0,ip),..)
```
→通信相手とデータを通信

スレッド並列



ステンシル計算

プログラミングイメージ(openACC)

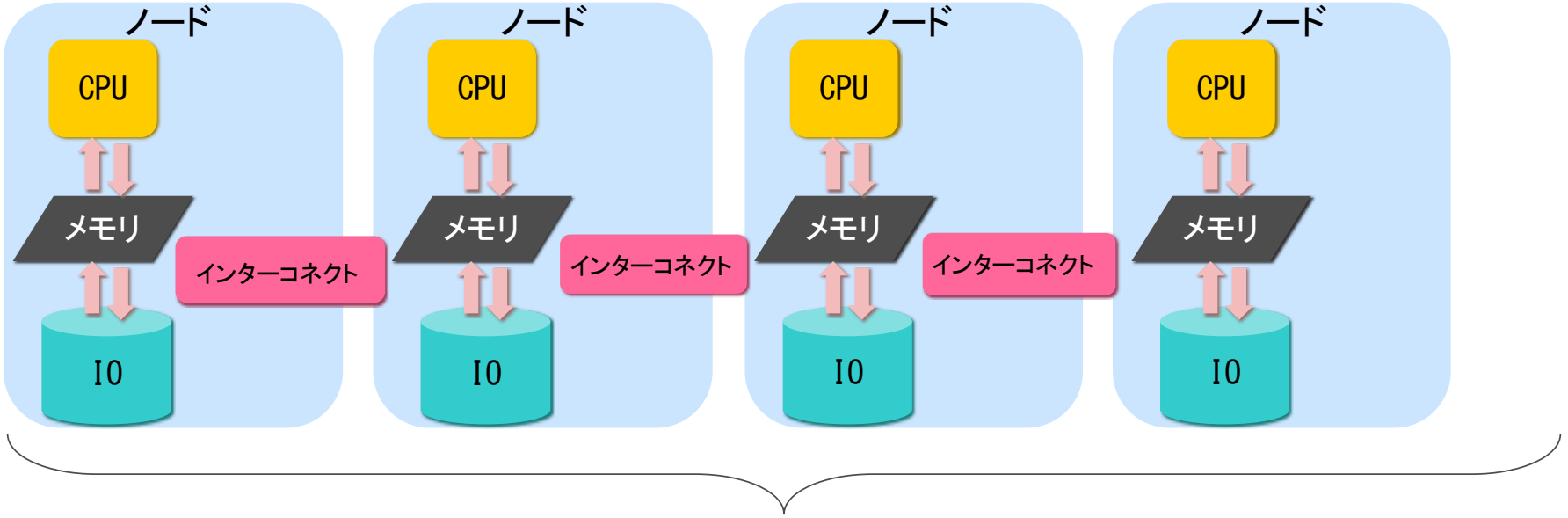
```
 ! 初期化
  do i = 1, N
    a(i) = dble(i)
  end do
!$acc data copy(a) create(b)
! 時間発展
  do iter = 1, 100
!$acc parallel loop
    do i = 2, N-1
      b(i) = (a(i-1) + a(i) + a(i+1)) / 3.0d0
    end do
! 境界条件
!$acc parallel loop
    do i = 1, 1
      b(1) = a(1)
      b(N) = a(N)
    end do
! 配列コピー（GPU上で実行）
!$acc parallel loop
    do i = 1, N
      a(i) = b(i)
    end do
  end do
```

プログラミングイメージ(openMP)

```
! 初期化
  do i = 1, N
    a(i) = dble(i)
  end do
! 時間発展（複数ステップ）
  do iter = 1, 100
!$omp parallel do private(i)
    do i = 2, N-1
      b(i) = (a(i-1) + a(i) + a(i+1)) / 3.0d0
    end do
!$omp end parallel do
! 境界条件（そのままコピー）
    b(1) = a(1)
    b(N) = a(N)
! 配列の入れ替え
    a = b
  end do
```

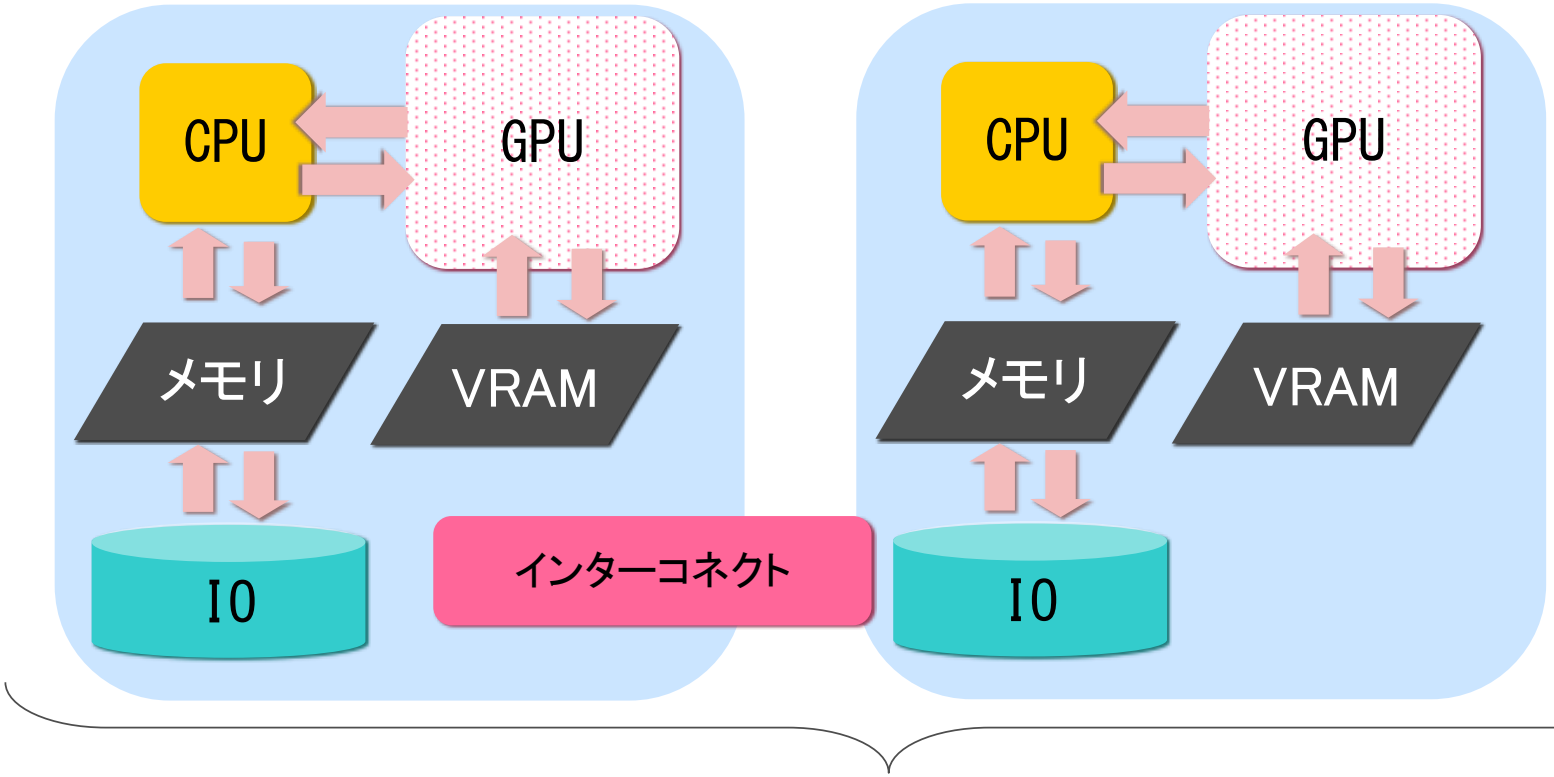
プロセス並列/スレッド並列の使い方

CPUの場合



- 非常に多くのプロセス：「富岳」の場合→60万プロセス(典型例)
- プロセス内に少ないスレッド数：「富岳」の場合→12スレッド(典型例)
-

GPUの場合



「Perlmutter」：
A100ベースのシステム
以下の例は1GPUで
1プロセスの例

- **比較的少ないプロセス**：「Perlmutter」の場合→6千プロセス程度
- **プロセス内に非常に多くのスレッド数**：「Perlmutter」の場合→10万程度以上

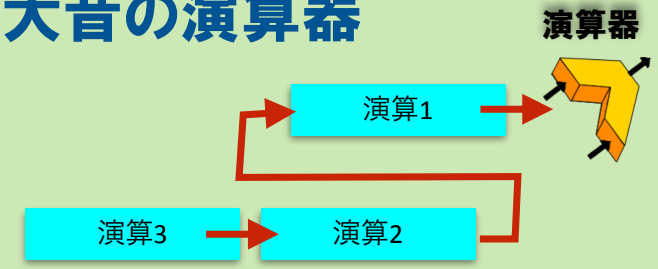
CPU/GPUの以下への対処

- 非常に多くの演算器を使えるようにするための対処
- メモリウォール問題への対処

CPU-SIMD演算器

waveZ

大昔の演算器

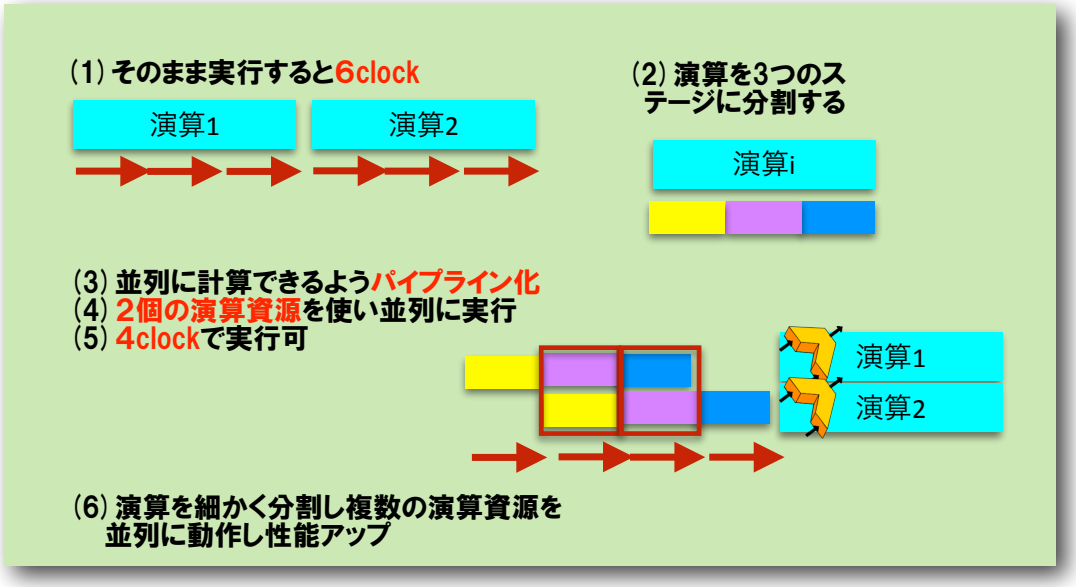


1個の演算器に逐次に入力し逐次実行

CPUでよく使われる方式

SIMD演算器
(Single Instruction Multi Data)

単体性能も並列性を利用しないと性能向上しない！



CPU-ソフトウェアパイプラインニング

waveZ

<前提>
・ロード2つorストアと演算とは同時実行可能
・ロードとストアは2クロックで実行可能
・演算は4クロックで実行可能
・ロードと演算とストアはパイプライン化されている

スケジューリング機能

< 何もしないと>

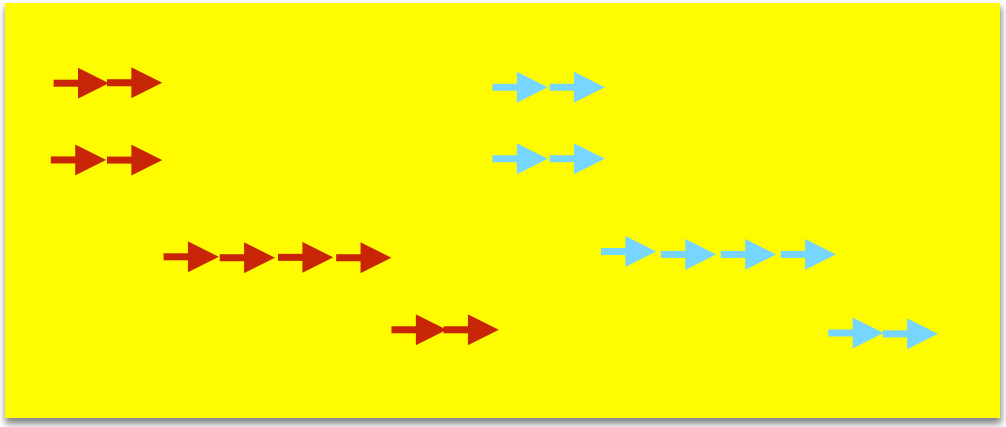
do i=1,100
　　a(i)のロード

　　b(i)のロード

　　a(i)とb(i)の演算

　　i番目の結果のストア

end do



< 1要素で8クロック，4要素で8×4=32クロックかかる>

CPU-ソフトウェアパイプラインニング

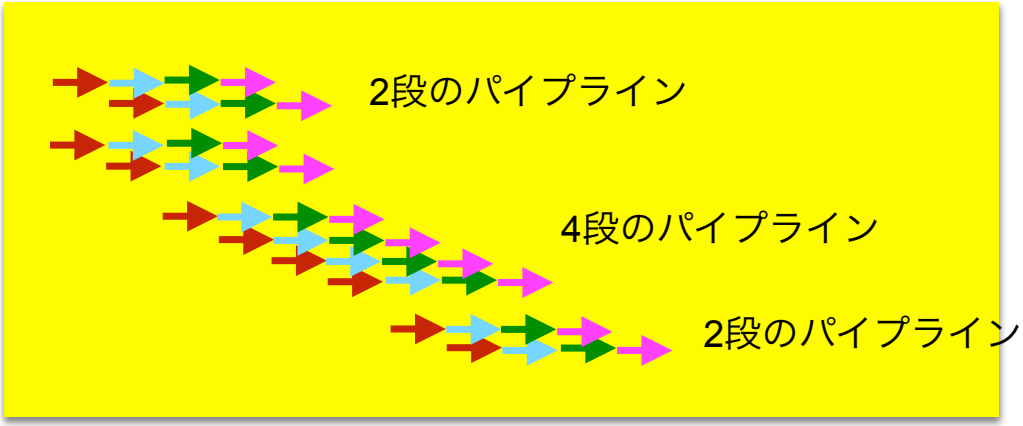
waveZ

<前提>
・ロード2つorストアと演算とは同時実行可能
・ロードとストアは2クロックで実行可能
・演算は4クロックで実行可能
・ロードと演算とストアはパイプライン化されている

スケジューリング機能

< ソフトウェアパイプラインニング>

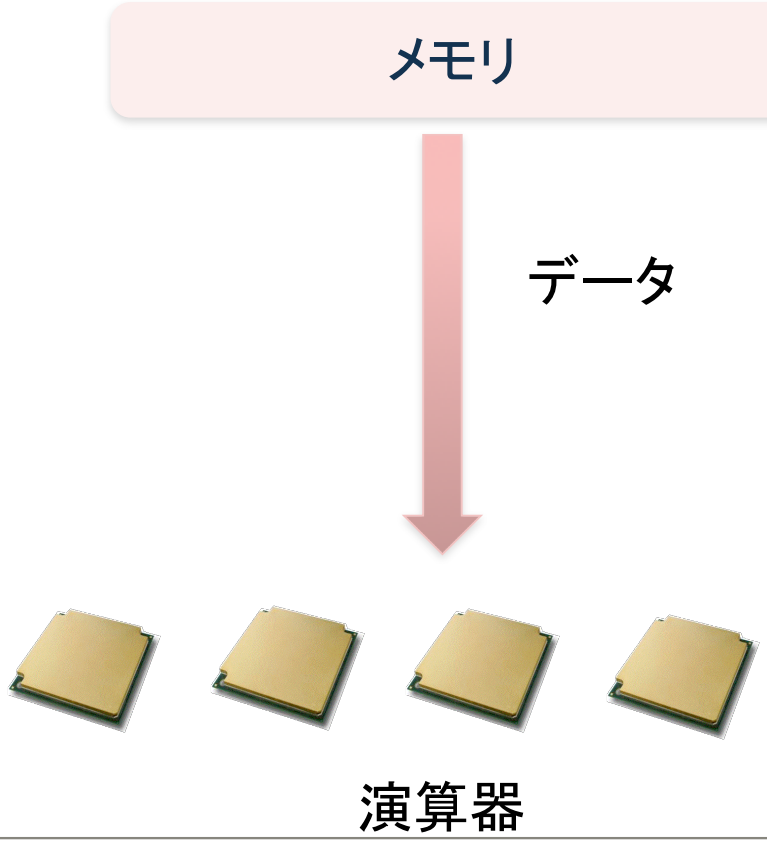
```
do i=1,100
      a(i)のロード

      b(i)のロード

      a(i)とb(i)の演算

      i番目の結果のストア

end do
```



< 4要素を11クロックで計算できる>

CPU&GPU−キャッシュの活用

- かつては研究者やプログラマーは物理モデル式に忠実に素直にプログラミングすることが一般的であった



メモリウォール問題

✓ 昔の計算機はメモリのデータ供給能力と演算器の能力がバランスしていた

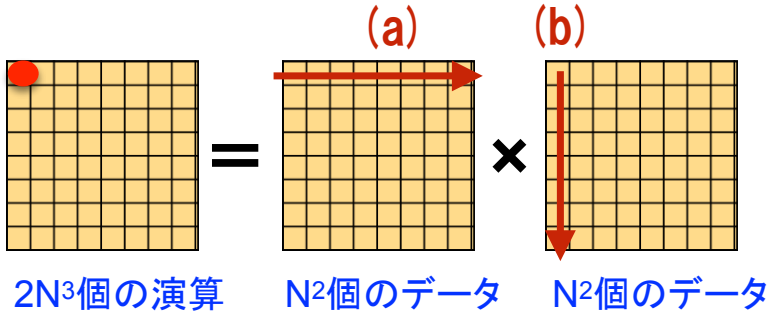
- 現代の計算機は演算器の能力が高くなりメモリのデータ供給能力が相対的に不足している

CPU&GPU－キャッシュの活用

waveZ

B/F値が小さい計算

行列行列積の計算



B/F値
=移動量(Byte)/演算量(Flop)
=$2N^2/2N^3$
=1/N

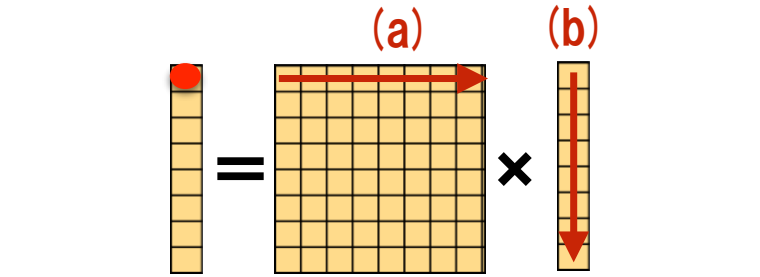
原理的にはNが大きい程小さな値

CPU&GPU－キャッシュの活用

waveZ

B/F値が大きい計算

行列ベクトル積の計算



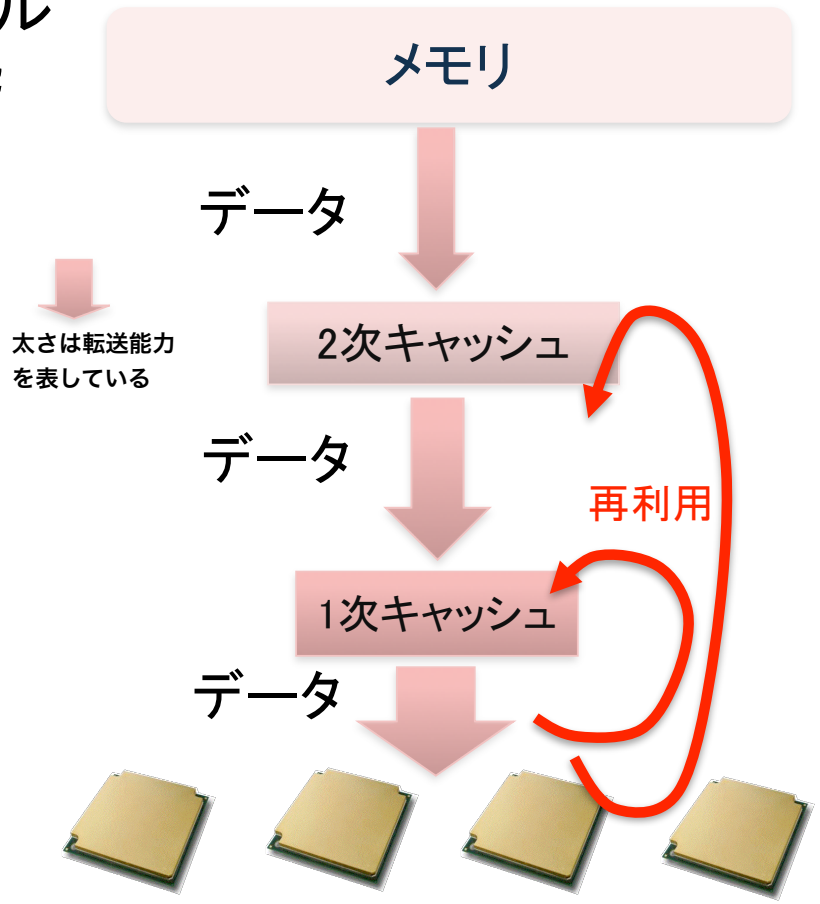
B/F値
=移動量(Byte)/演算量(Flop)
$=(N^2+N)/2N^2$
$\fallingdotseq 1/2$

原理的には1/Nより大きな値

CPU&GPU−キャッシュの活用

waveZ

メモリウォール
問題への対処



・データ供給能力の高いキャッシュを設ける
・キャッシュに置いたデータを何回も再利用し
演算を行なう
・こうする事で演算器の能力を十分使い切る

「アプリケーションが要求
するByte/Flop値が低い」タイプ
の計算(*1)

例えば行列行列積
(2N²個のデータで2N³個の演算可)

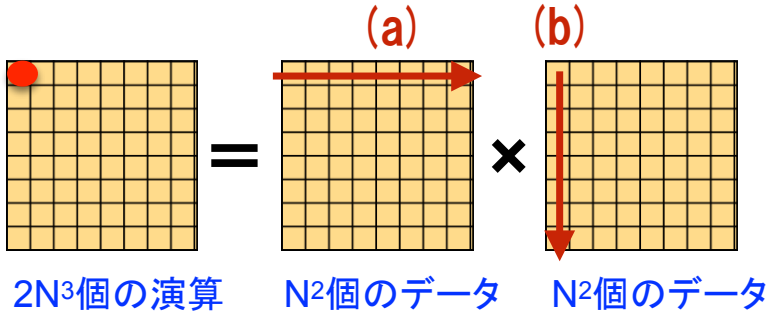
(*1)演算量(Flop) に比べ
データの移動量(Byte)が小さい計算

キャッシュの有効利用

CPU&GPU－キャッシュの活用

waveZ

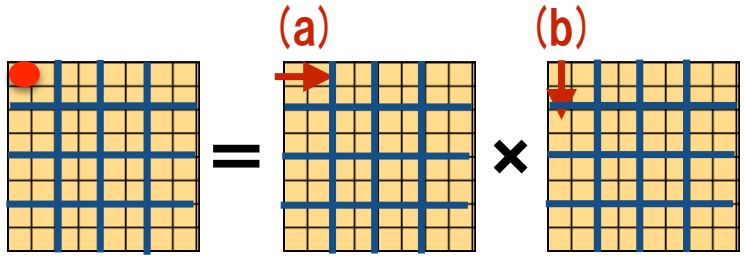
行列行列積の計算



B/F値
=移動量(Byte)/演算量(Flop)
$=2N^2/2N^3$
$=1/N$

原理的にはNが大きい程小さな値

- **現実のアプリケーションではNがある程度の大きさになるとメモリ配置的には (a) はキャッシュに乗っても (b) は乗らない事となる**

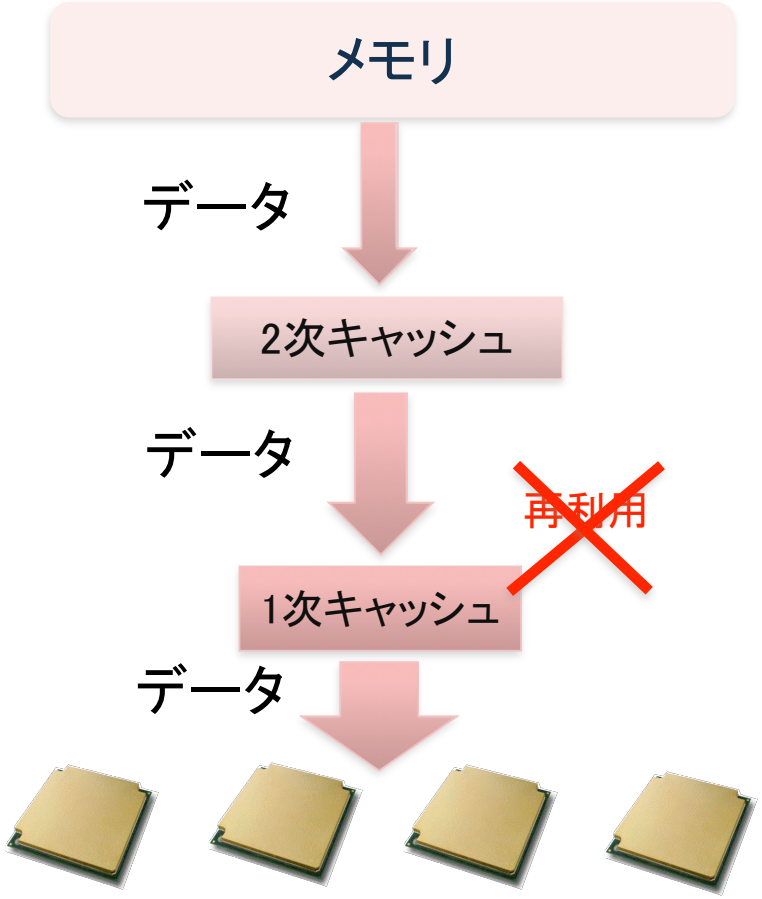


- **そこで行列を小行列にブロック分割し (a) も (b) もキャッシュに乗るようにしてキャッシュ上のデータだけで計算するようにする事で性能向上を実現する.**
- **一度メモリから持ってくれば，キャッシュ上にデータがあるために，次回からはデータ転送時間が短いキャシュのアクセス時間だけで済むために演算時間が短くて済む.**

CPU&GPU-キャッシュの活用

waveZ

とは言っても・・・・



・再利用出来ない問題もある
・こうなる演算器の能力を十分使い切る事が出来ない

「アプリケーションが要求するByte/Flop値が高い」タイプの計算(*2)

(*2)演算量(Flop) に比べデータの移動量(Byte)が大きい計算

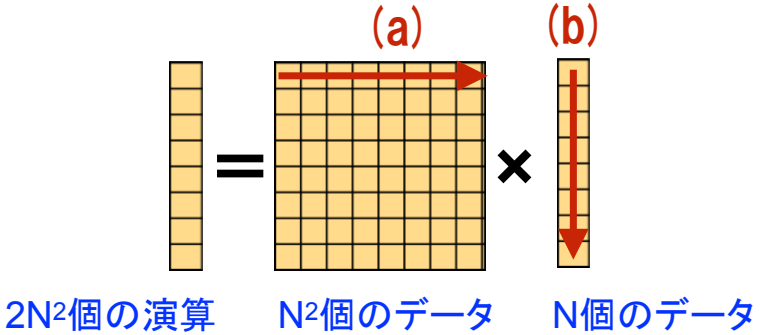
キャッシュの有効利用が難しい

演算器

CPU&GPU－キャッシュの活用

waveZ

行列ベクトル積の計算



B/F値
=移動量(Byte)/演算量(Flop)
$=(N^2+N)/2N^2$
$\fallingdotseq 1/2$

原理的には1/Nより大きな値

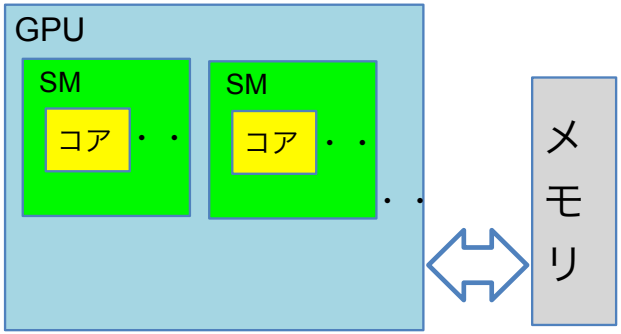
- **行列を小行列にブロック分割して(a)も(b)もキャッシュに乗るようにしても再利用の度合いが小さいために性能向上はできない．**
- **メモリからデータを持ってくるために，その転送時間がネックになる．**

演算器の能力は最大限引き出せないが
メモリの性能を上限まで使うのが重要！
（第3回に講義します）

GPU-SIMTアーキテクチャ

waveZ

- 1GPUは複数のSM (Streaming Multiprocessor) を持つ (A100の場合108個)
- GPUはSIMTアーキテクチャを採用しており1つの命令で複数のスレッドを動作させる
- 1つの命令で動作するスレッドの単位をワープといい通常32個のスレッドを含む
- 1GPU内に多量のスレッドを発生させワープスケジューラーで多量のスレッドを捌く
- 演算コアはCPUのようなSIMDでなく比較的単純なコアとなっている



SIMTアーキテクチャ
(Single Instruction Multi Threads)

CPUとGPUのハードウェア量比較

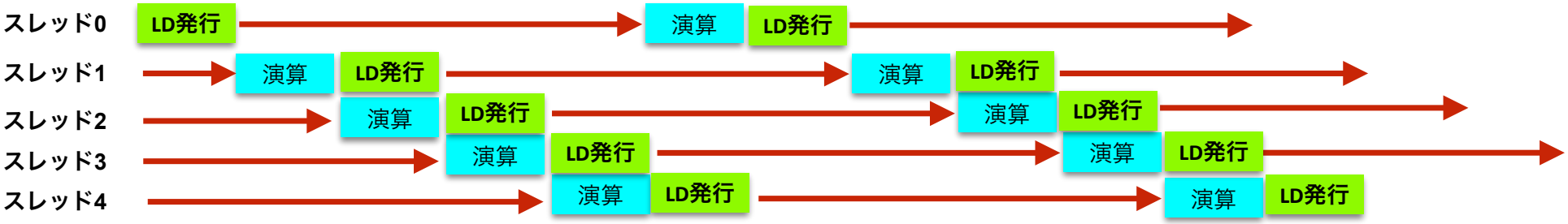
waveZ

	NVIDIA A100		富岳		A100/富岳
	1SM	108SM	1CMG	1CPU	
レジスタ	レジスタ幅：4B レジスタ数：65536個 総量：4B×65536 =256 KB	総レジスタ量： 256KB×108=**27MB** (FP64変数の消費量： 2個／変数)	1コア：64B×32個=2KB CMG：2KB×12=24KB (ベクトルレジスタ)	CPU：24KB×4=**96KB** (ベクトルレジスタ)	**281倍**
演算器	INT32：64個 FP32：64個 FP64：32個 Tensor：4個	INT32： 64×108=6912個 FP32： 64×108=6912個 FP64： **32×108=3456個** Tensor： 4×108=432個	12コア/CMG 1 × 8 × 2 ×12 (m&a)(simd)(pipe)(core) =192個 (A100のFP64コアは積和も可能.対比のためにm&aは演算1個とする)	4CMG/CPU 12×4=48コア/CPU 192×4 =**768個**	**4.5倍** **(FP64比較)**
Cache	L1D：192KB/SM	L1D：192KB×108 =**20.7MB** L2D：**40MB**	L1D：64KB/コア L2D：8MB/CMG	L1D： 64KB×48=**3.07MB** L2D：8MB×4=**32MB**	**L1D：6.7倍** **L2D：1.3倍**
メモリ		**40GB**	8GB/CMG	8GB×4=**32GB**	**1.3倍**

GPU-メモリレイテンシーの隠蔽

waveZ

- CPU-GPUのハードウェア比較で見たようにレジスタの物量がCPUに比較すると非常に多い
- 多くのスレッドはそれぞれ個別のレジスタが割り当てられるためレジスタの物量が必要となっている
- CPUと比較するとキャッシュを簡略化しスレッドごとのローカルデータをレジスタに保持する仕組みとなっている
- 複数のワープ（スレッド）を同時に動作可能な状態しあるスレッドがメモリ待ちする間に他のスレッドが単純な演算コアを使用することでメモリレイテンシーを隠蔽する仕組みをとっている



- メモリアクセスは時間がかかる（→はメモリレイテンシ）
- 複数スレッドを同時に動作させる（例は5スレッド）
- スレッド0のメモリアクセス中に1-4スレッドの演算実行
- スレッド0のメモリアクセスは隠蔽できている

- この図では各スレッドが別の動作をするように見えているが実際はワープ単位で命令は実行される
- 説明の簡素化のための図である

まとめ

waveZ

- スーパーコンピュータとは？
  - シングルプロセッサの時代
  - メモリウォール等の問題
  - 並列プロセッサアーキテクチャへ
- アプリケーションの性能とは？
  - 多くの演算器を使うための並列化プログラミングが必要
  - メモリウォール問題に対処のためのプログラムチューニングも必要
- アプリケーションの超並列性を引き出す
  - 並列化できるアルゴリズムを採用する
  - 並列化プログラミングのイメージ
  - 非並列部を最小化する
- CPU/GPUでの以下への対処
  - 非常に多くの演算器を使えるようにするための対処
  - メモリウォール問題への対処

CPUもGPUも並列化可能なアルゴリズムを使用することが重要