

シミュレーションが 未来をひらく



スーパーコンピュータと アプリケーションの性能

2016年4月

国立研究開発法人理化学研究所
計算科学研究機構 運用技術部門
ソフトウェア技術チーム チームヘッド

南 一生

minami_kaz@riken.jp



講義の概要

- **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)**
- **アプリケーションの性能最適化2 (高並列性能最適化)**
- **アプリケーションの性能最適化の実例1**
- **アプリケーションの性能最適化の実例2**

内容

- **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？**
- **高並列化のための重要点**
- **単体性能向上のための重要点**

スーパーコンピュータとは？

スーパーコンピュータとは・・・

- **スーパーコンピュータ=卓越した計算能力**

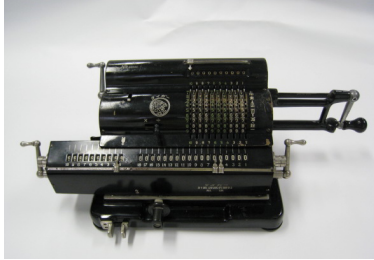
その時代の一般的なコンピュータよりも、極めて高速（浮動小数点演算性能で1000倍以上）な計算機

- **演算性能 50 TFlops 以上 (*)**

・・・現在の政府調達における「スーパーコンピュータ」の定義
(*) : 1秒間に50兆回以上の浮動小数点演算が可能

スーパーコンピュータの発展

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

2002年 当時のパソコン
PentiumIV
6.4Gflops

40倍

2011年 iPhone4S
LINPACK 140Mflops



≡

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



2015年 iPhone6S
LINPACK 870Mflops

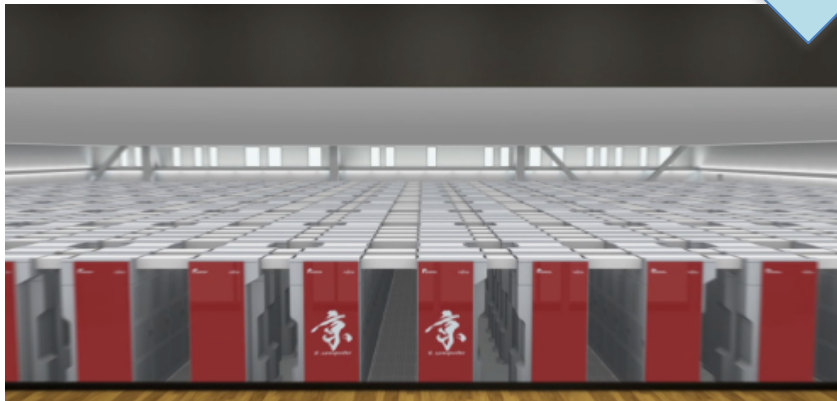
>

6250万倍



40Tflops

2011年 「京」コンピュータ



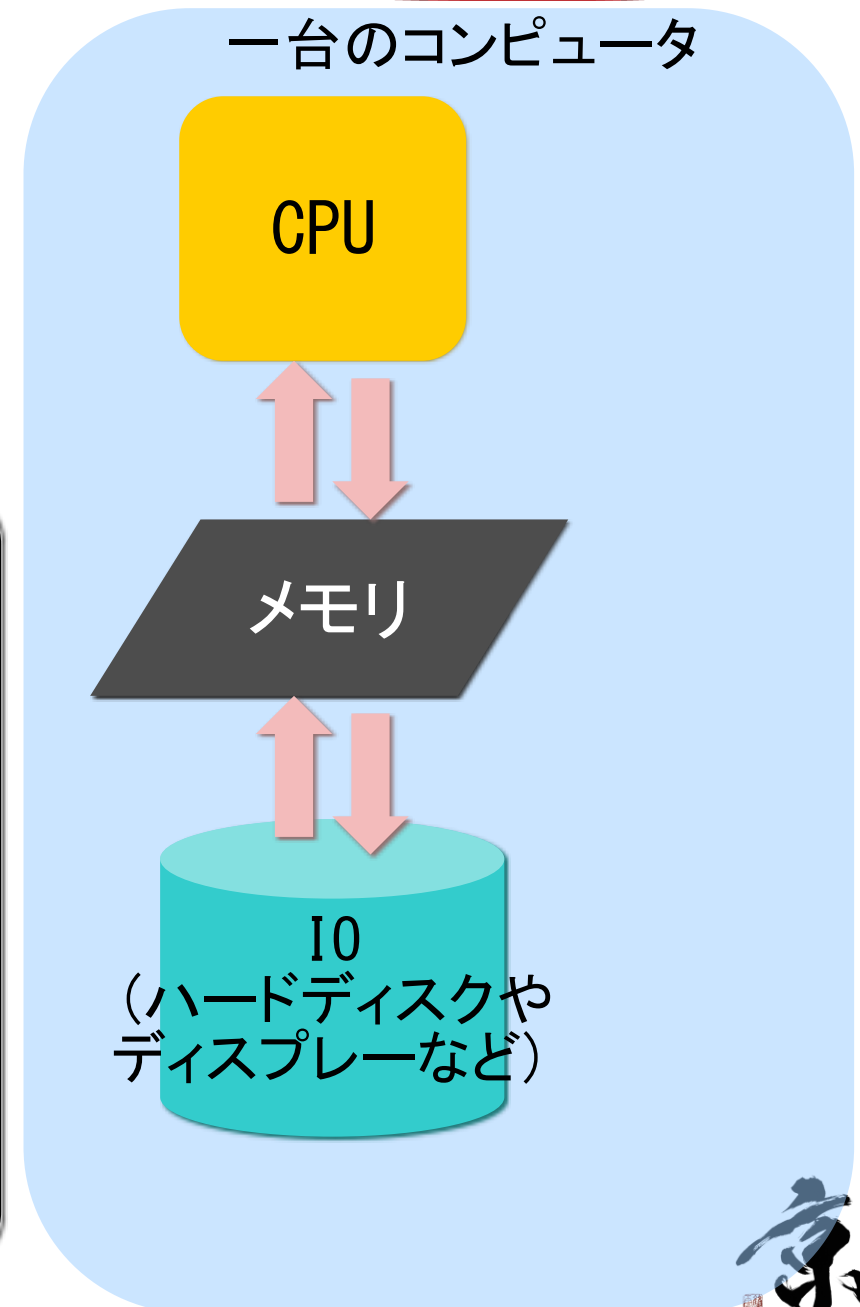
10Pflop

コンピュータとは？

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に出す、書出す（出力）



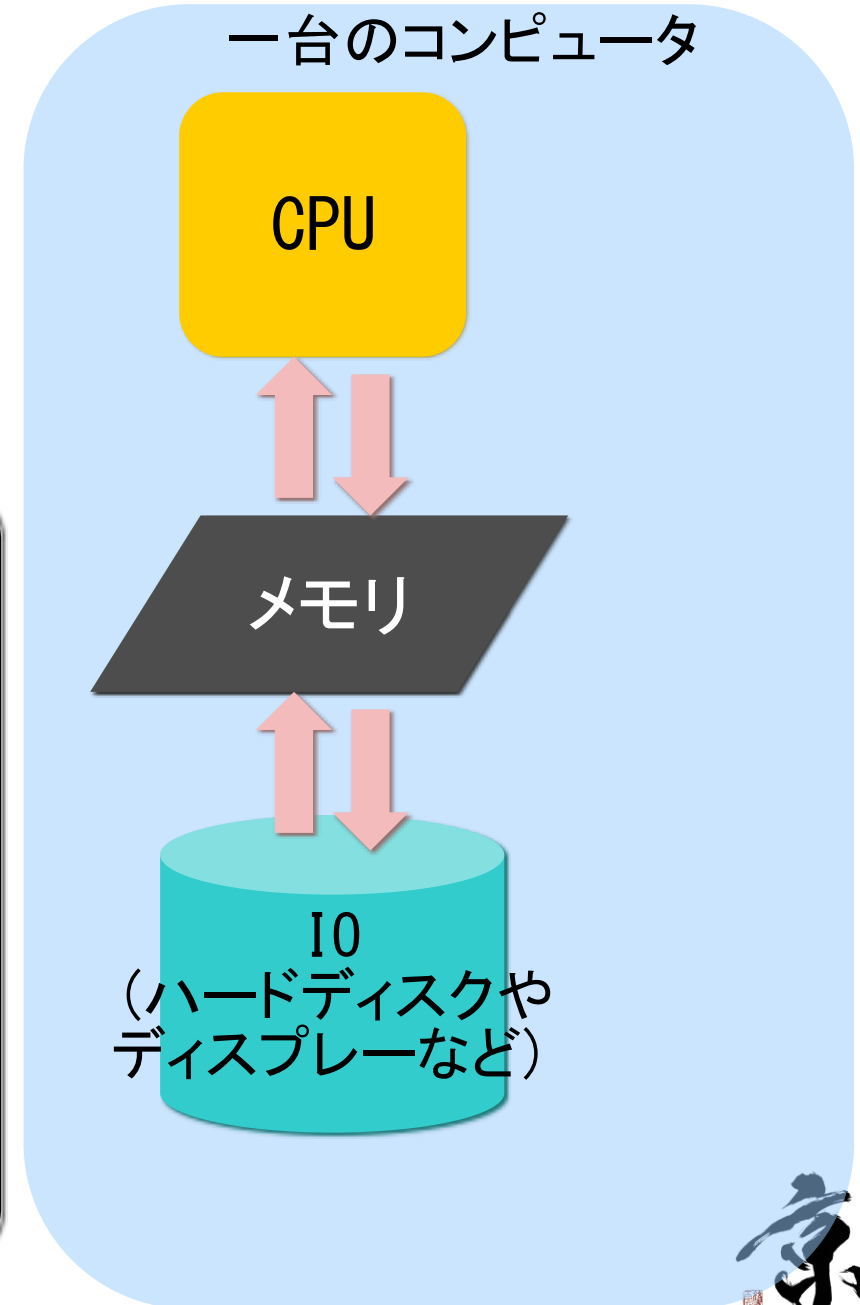
コンピュータとは？

ハードウェア ソフトウェア(進化とともに分化)

- CPU
- メモリ
- IO (入出力)
- 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

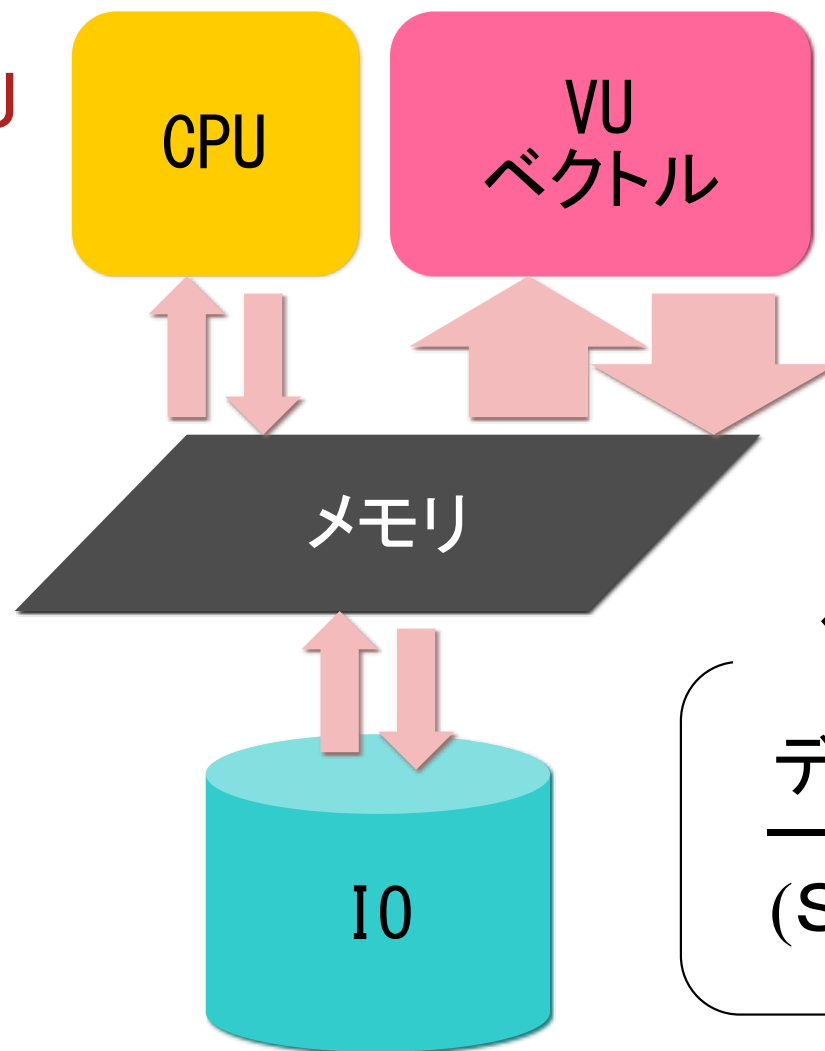
であった時代



昔のスーパーコンピュータ

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

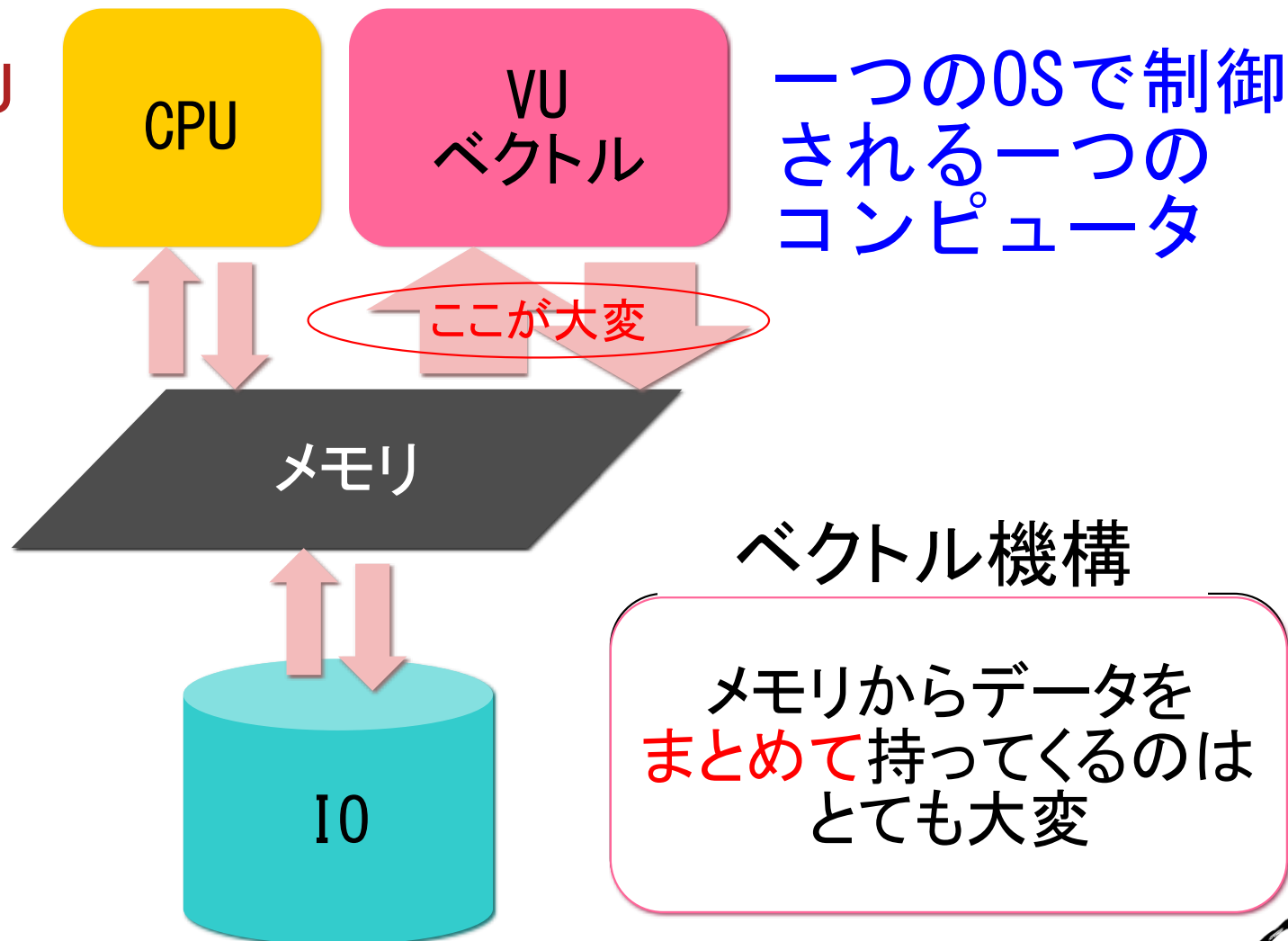
ベクトル機構

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

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)



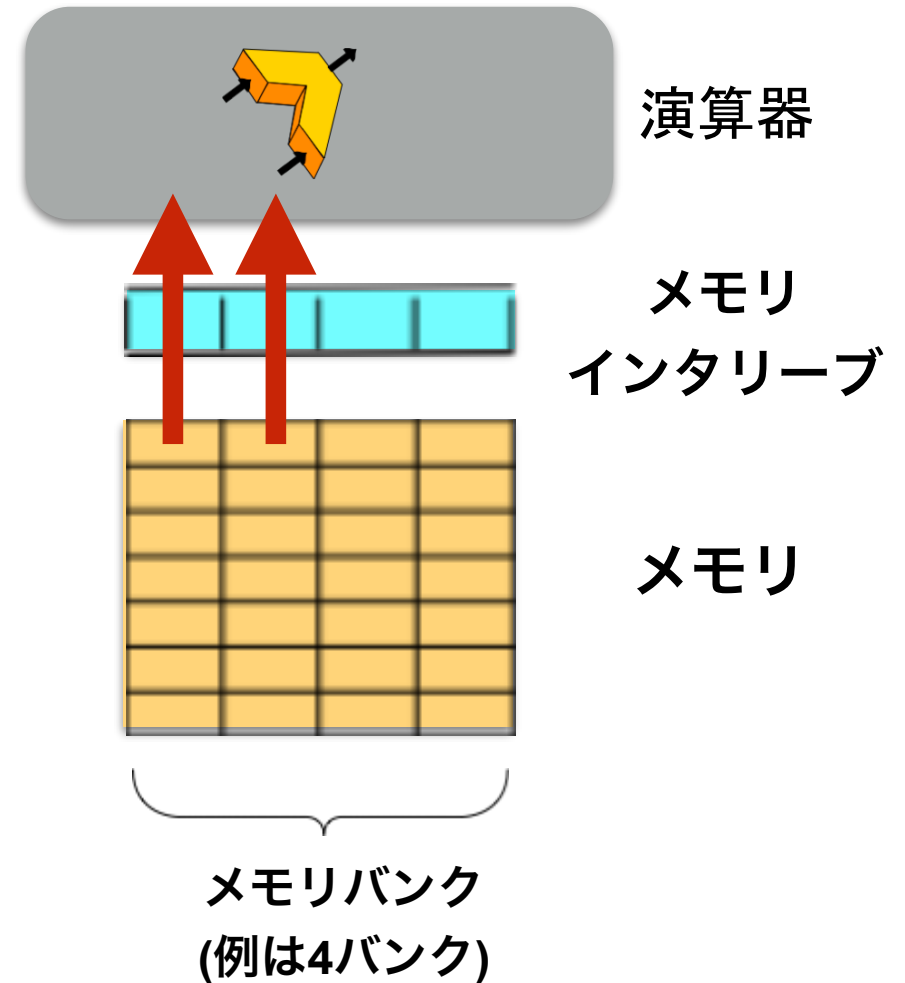
今もこの方式が無くなったわけではありませんが...

シングルプロセッサの問題

動作周波数が低い場合(昔)

メモリウォール問題

- メモリは一定時間を経過しないと同一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

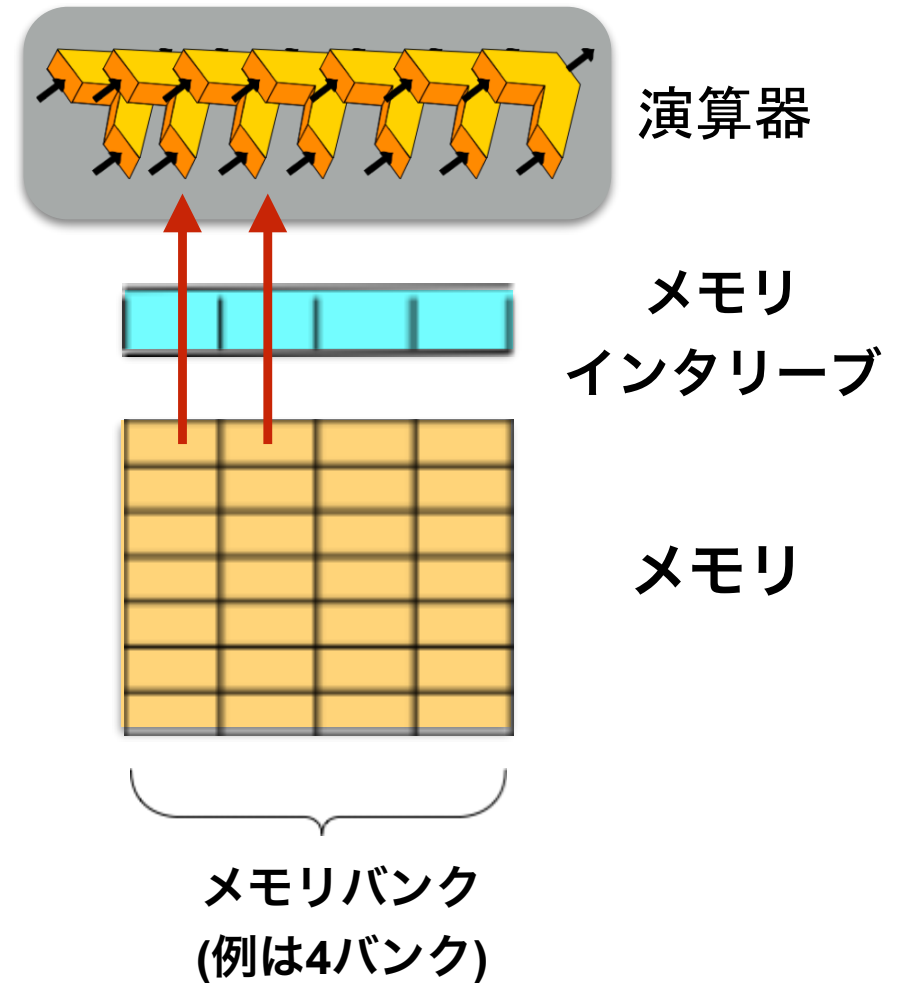


シングルプロセッサの問題

動作周波数が高い場合(現在)

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

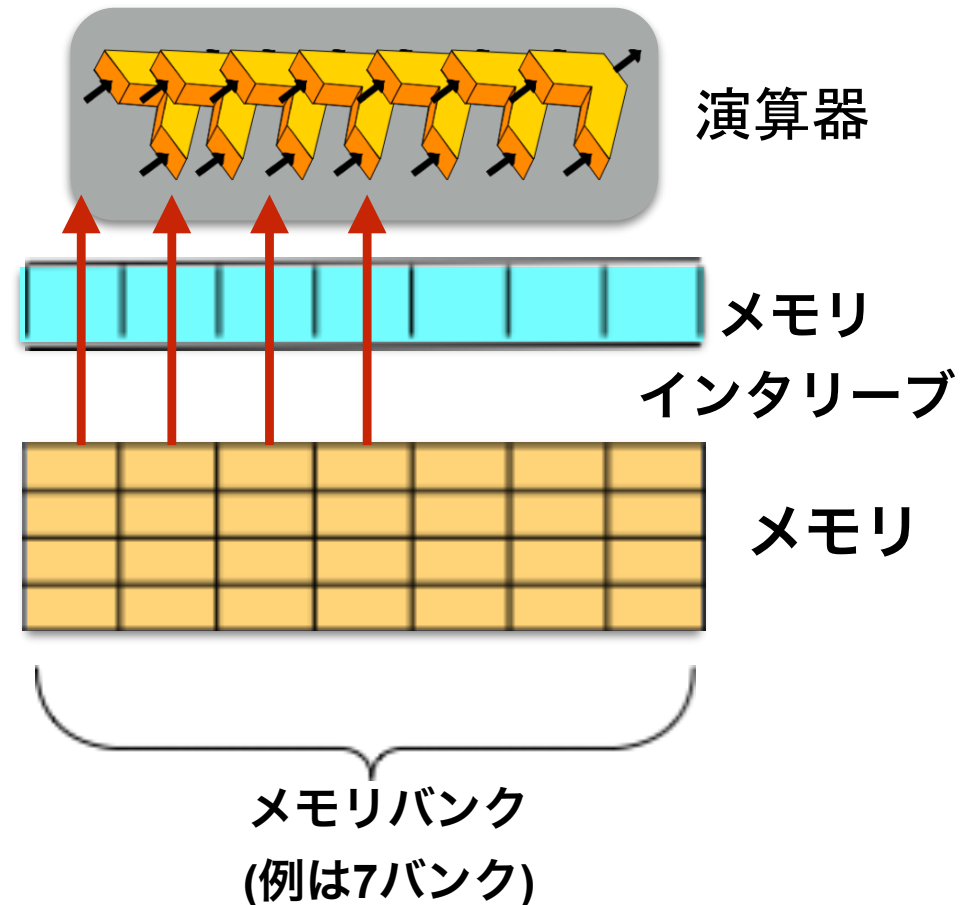


シングルプロセッサの問題

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

動作周波数が高い場合

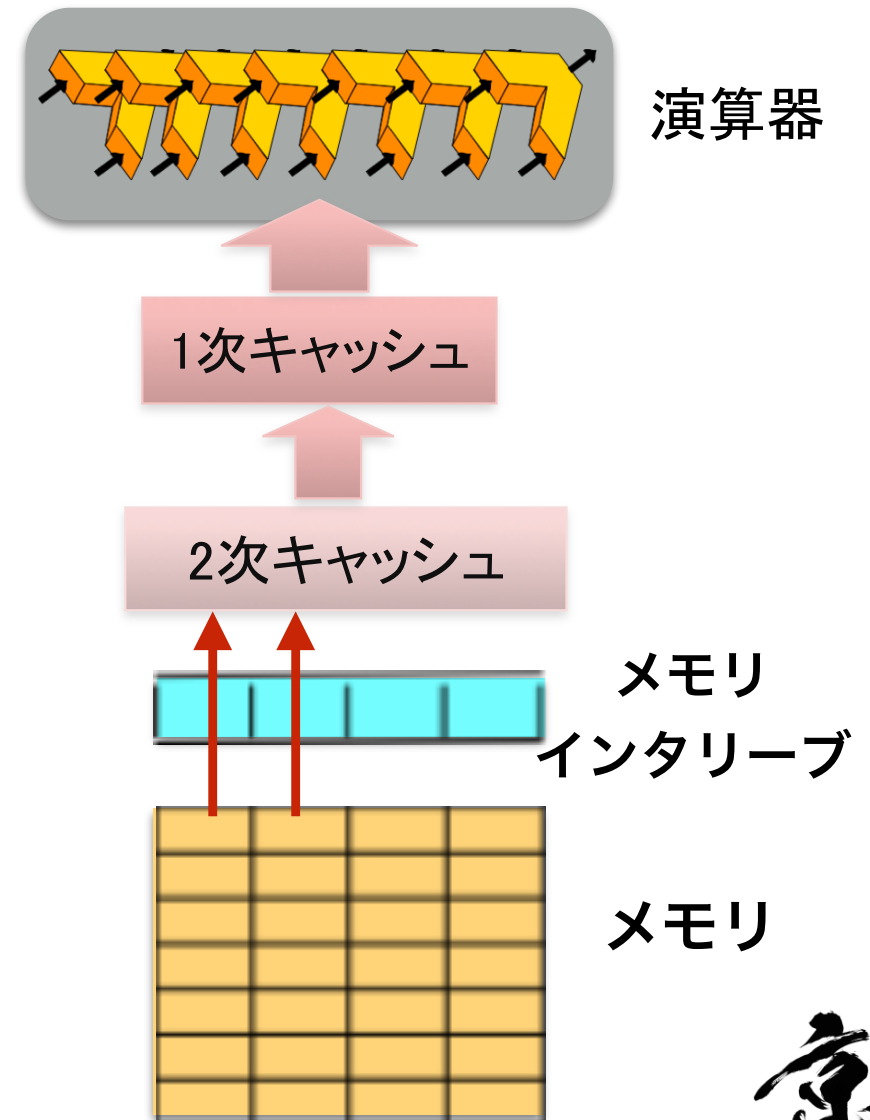


シングルプロセッサの問題

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
- 多くのスカラー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

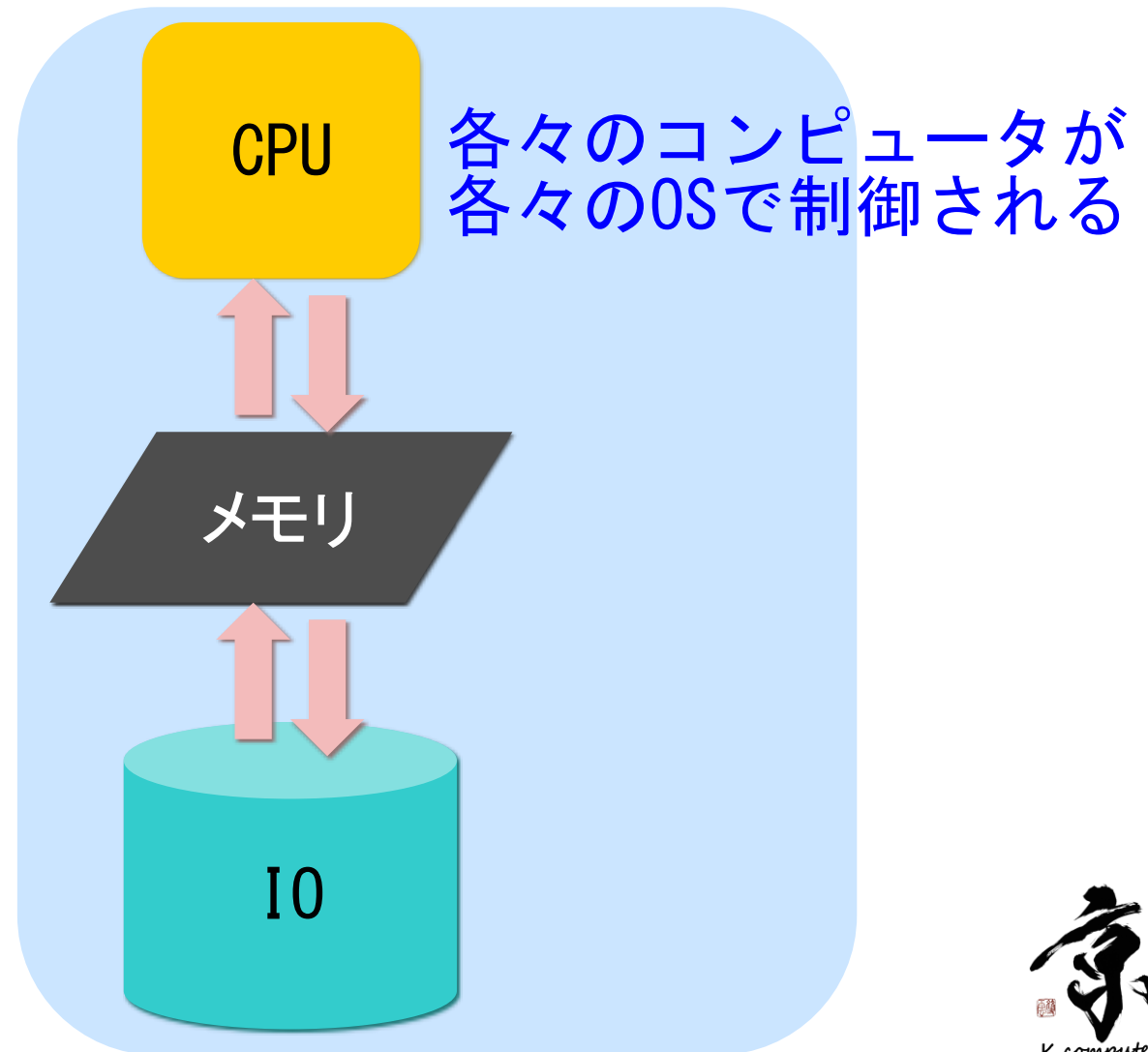
一台のコンピュータの処理限界

- 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てもメモリの能力が追いつかない

そこで！

最近のスーパーコンピュータ

- CPU
- メモリ
- IO(入出力)

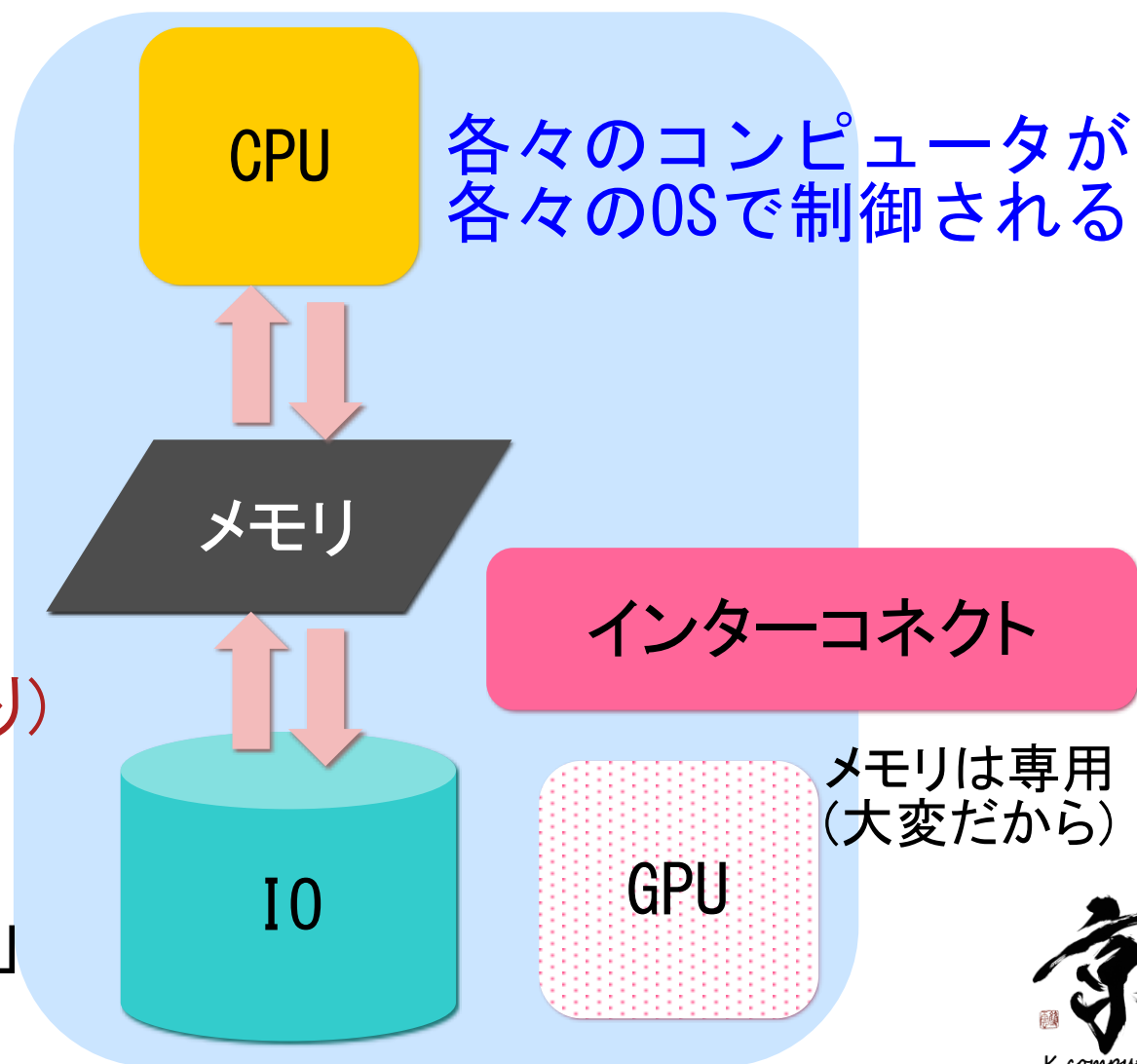


最近のスーパーコンピュータ

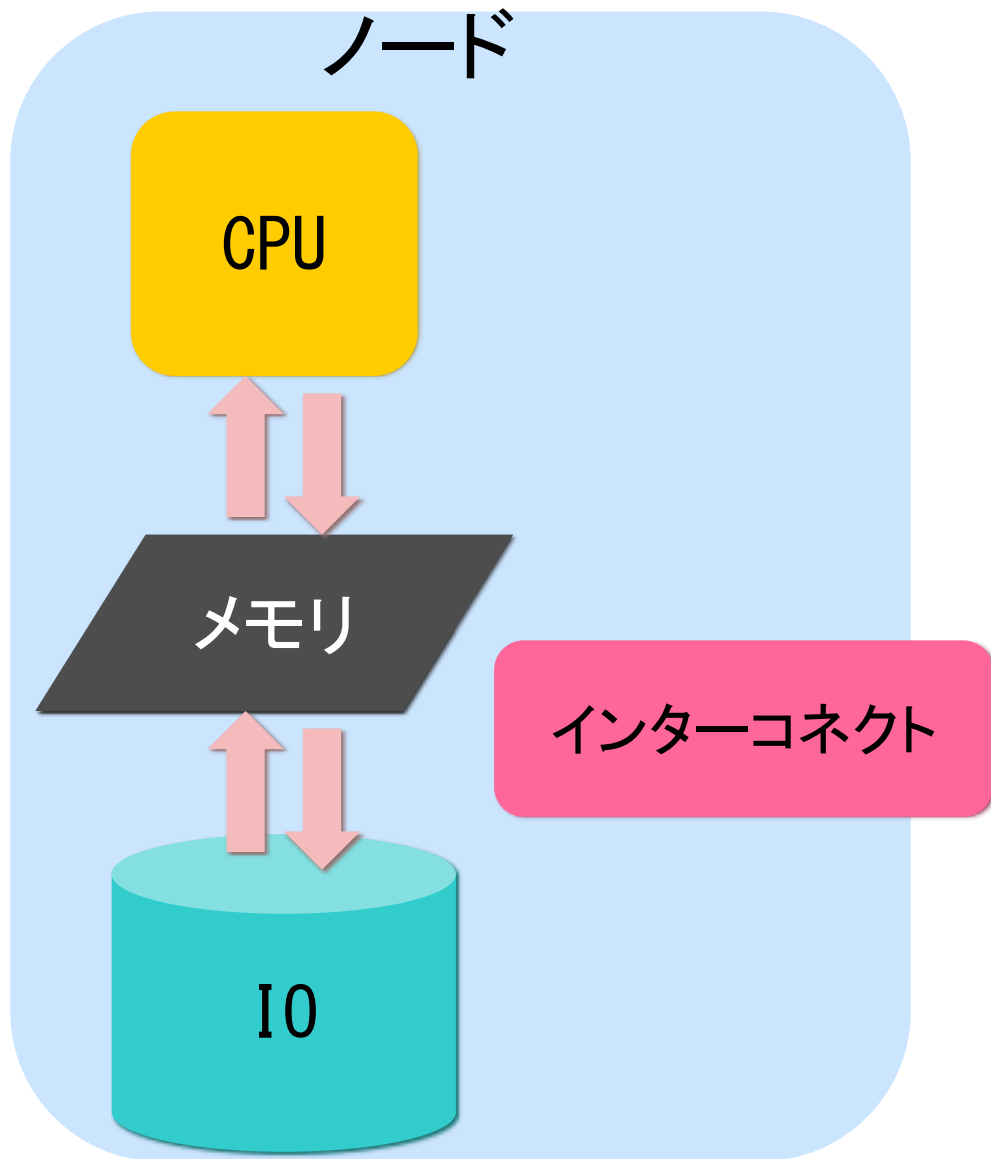
たくさんのコンピュータをつないで、システムとして一体で動かす

- CPU
- メモリ
- IO(入出力)
- インターコネク(接続機構)
- GPU/アクセラレータ
(加速器: あったり、なかったり)

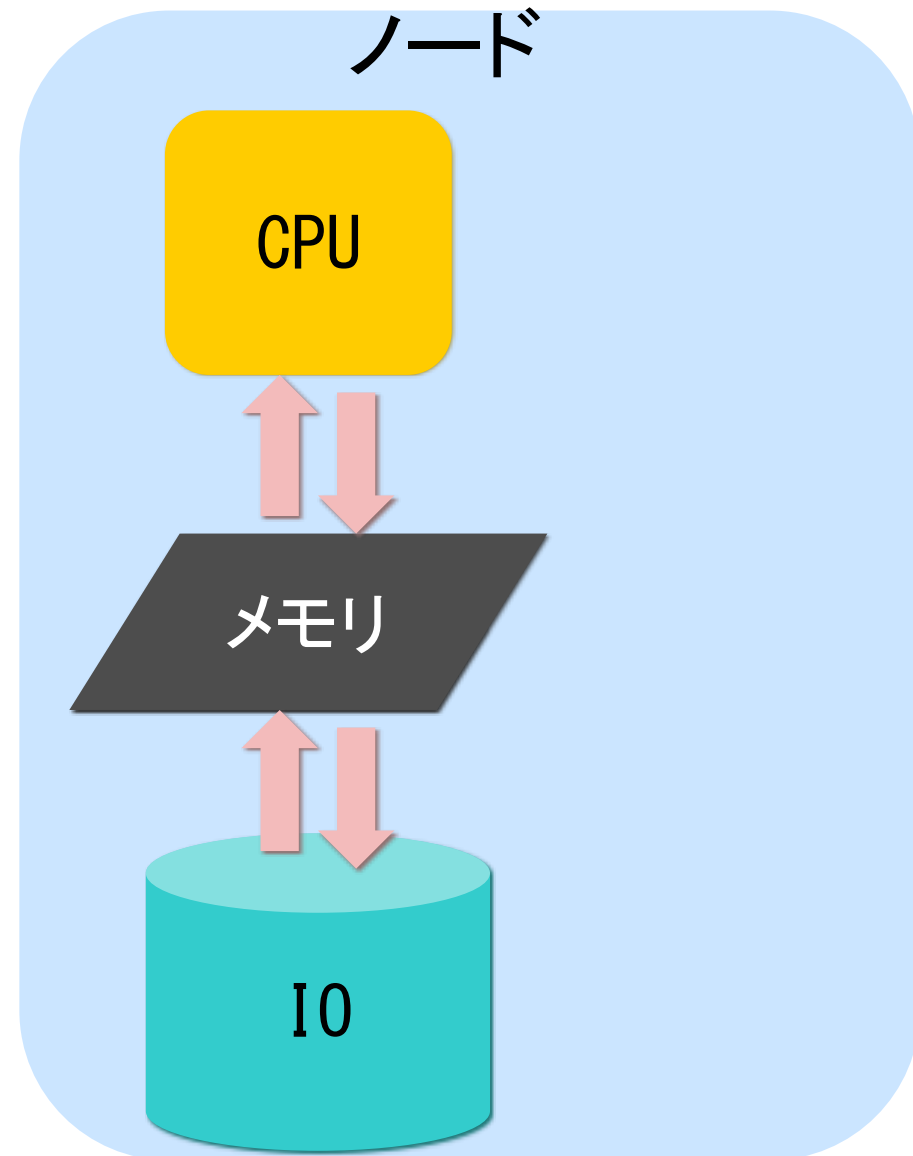
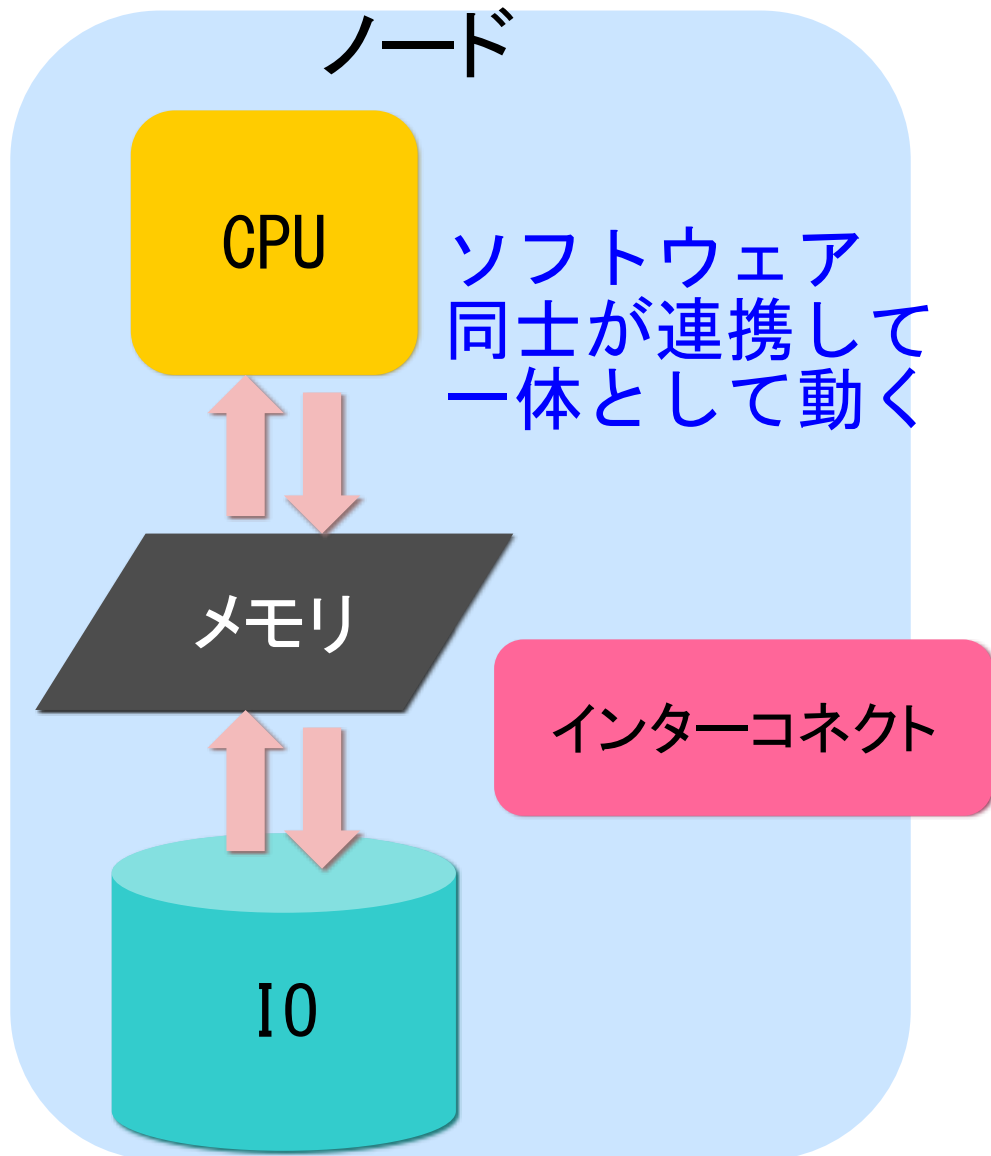
各々のコンピュータ=「ノード」



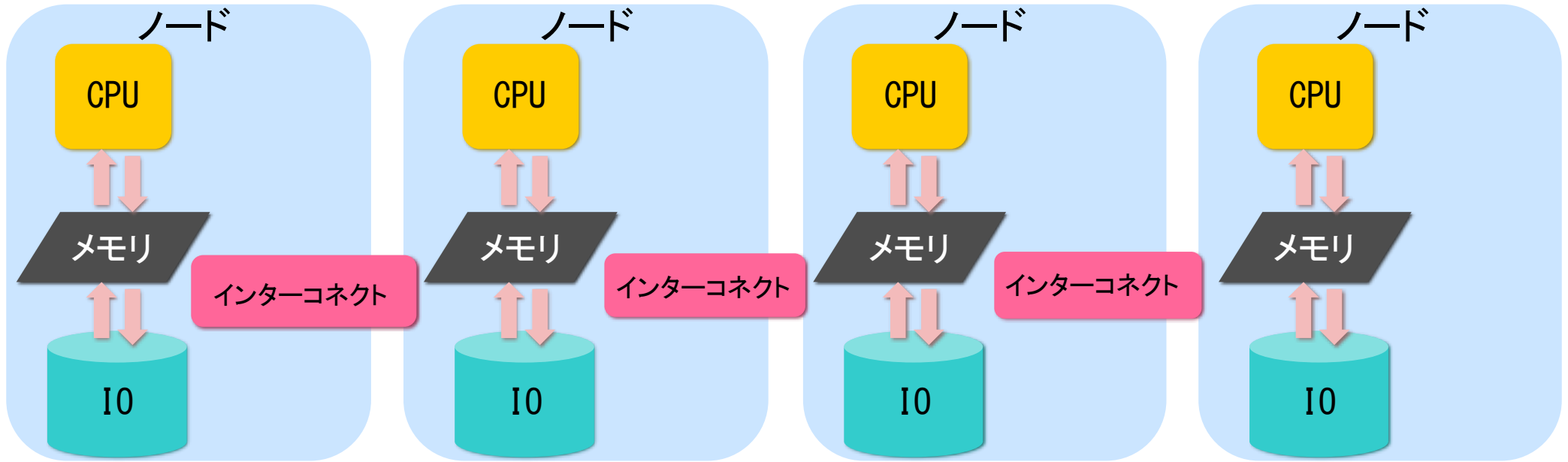
コンピュータ (ノード) どうしをつなぐ



コンピュータ (ノード) どうしをつなぐ



コンピュータ (ノード) どうしをつなぐ

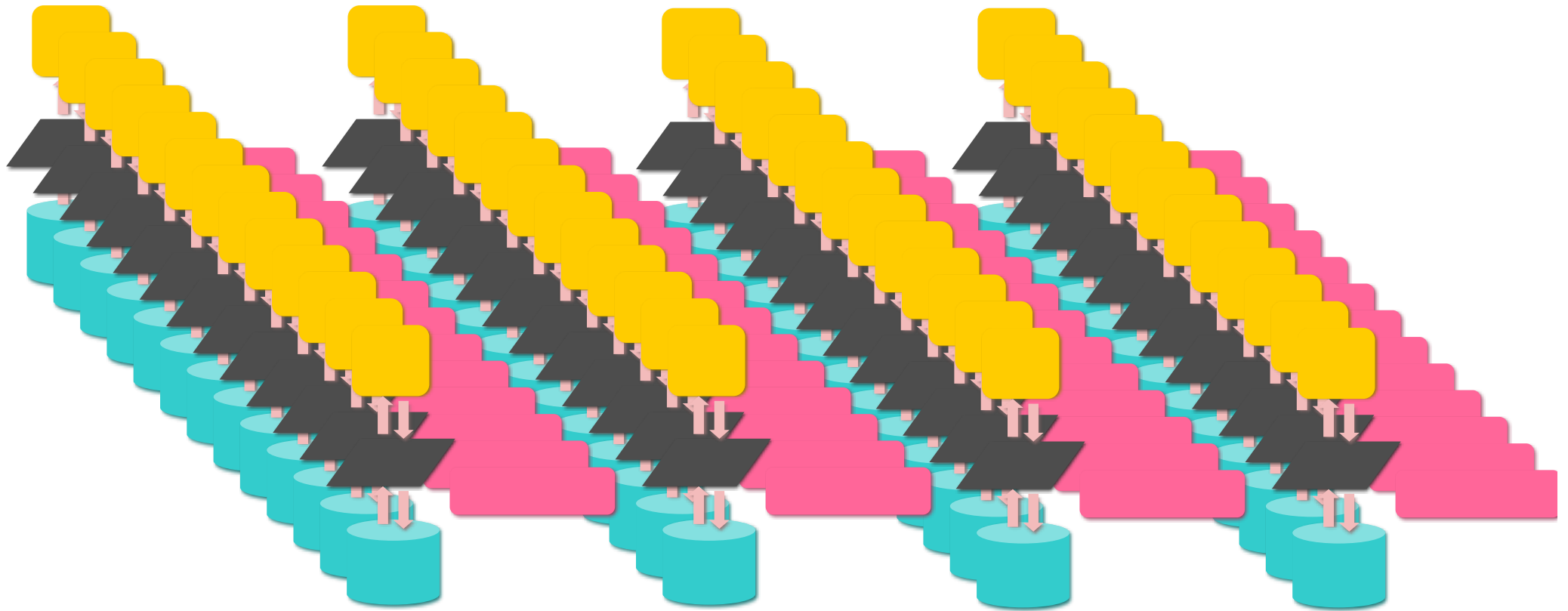


システムとして一体で動く

(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)

もっともっと、たくさんのコンピュータ（ノード）どうしをつなぐ

どれだけたくさんつなげるか？⇒システムとして高性能
一つのシステムとして動く



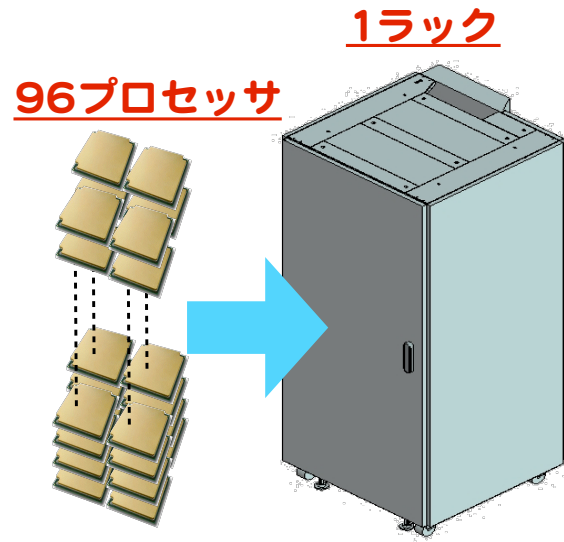
実際はハードディスクは、まとめる

ちなみに「京」の場合は？

50m×60mの部屋に

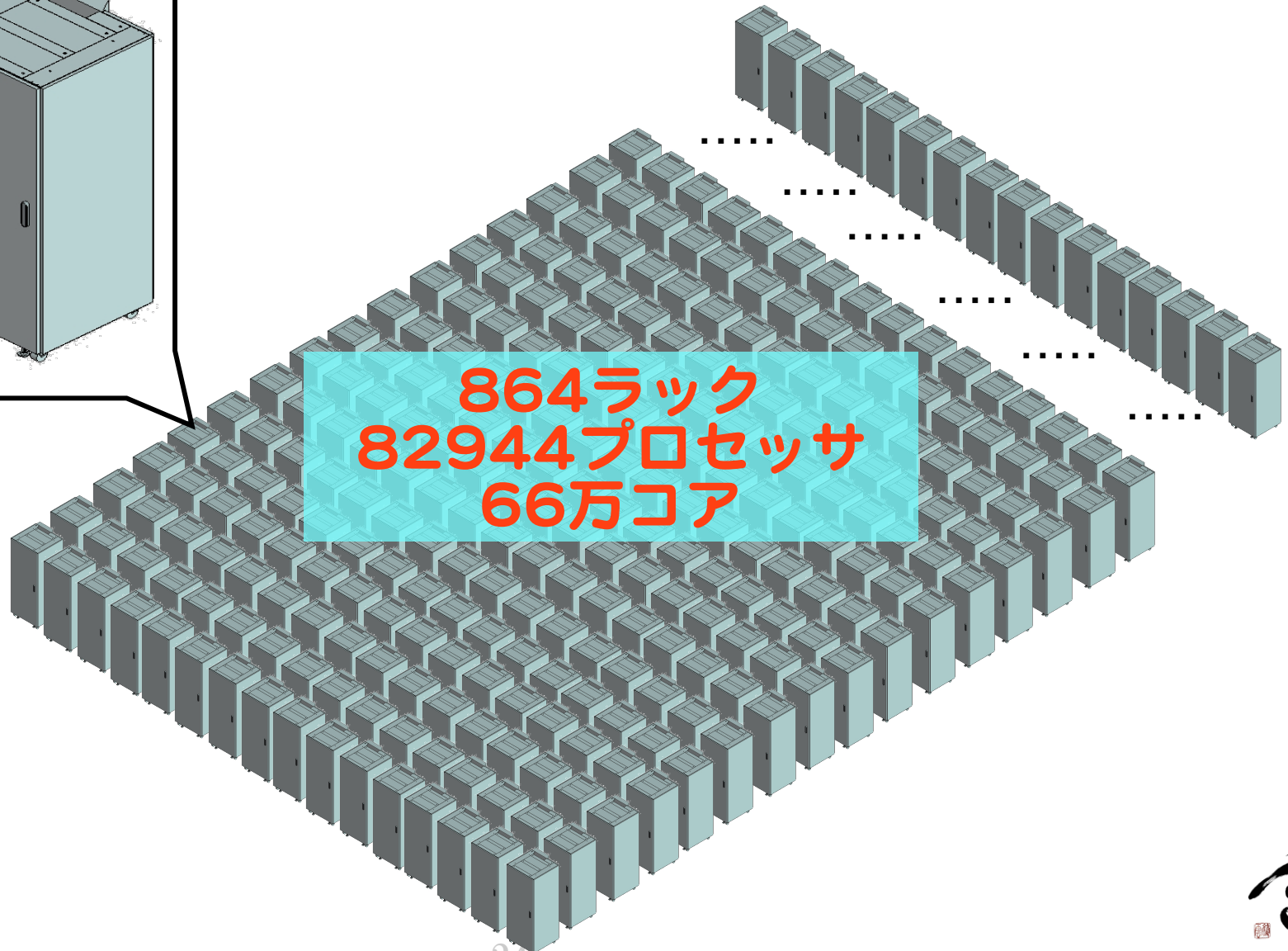
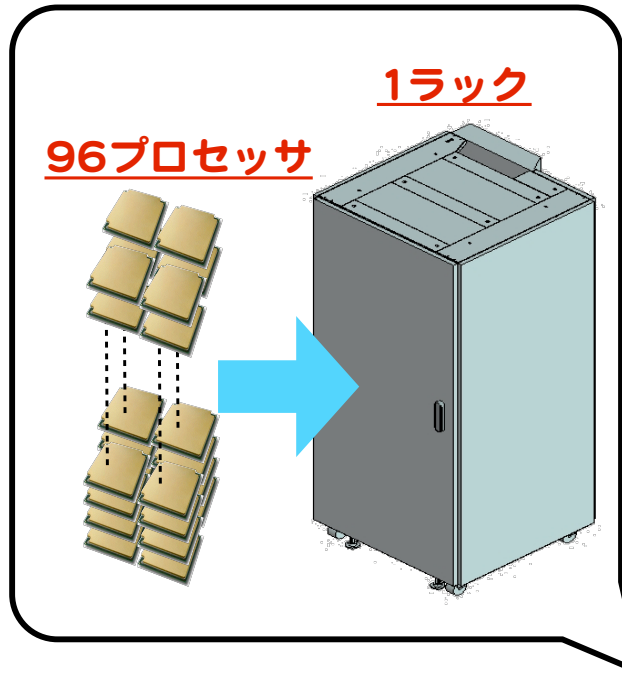
ちなみに「京」の場合は？

50m×60mの部屋に



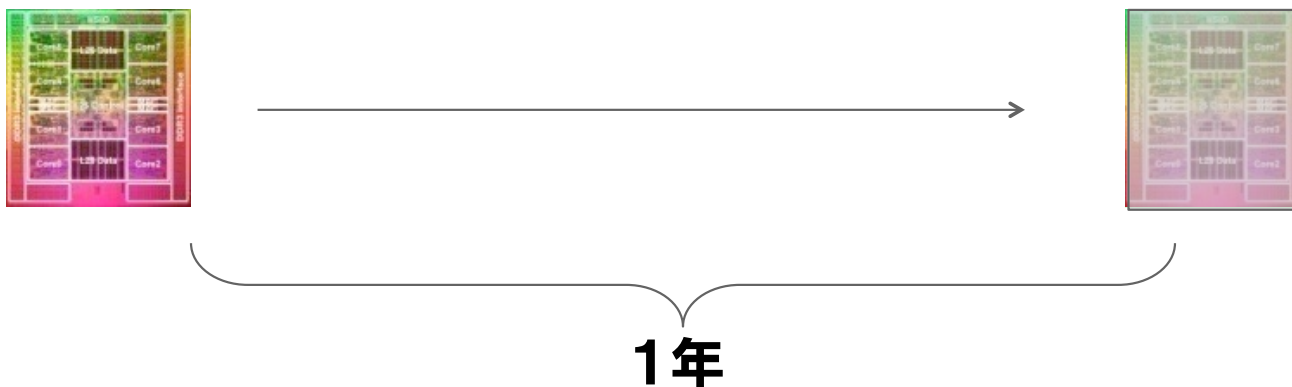
ちなみに「京」の場合は？

50m×60mの部屋に

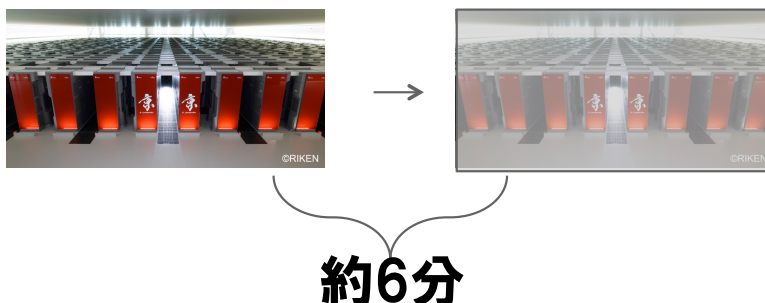


これだけ巨大システムでCPUが頻繁に故障すると・・・

- ✓ 仮にひとつのCPUが、1年に1回故障したとします



- ✓ システム全体で見ると、**約6分に1回**の故障・・・



CPUの故障率を抑えることは極めて重要

スーパーコンピュータについてまとめると

- 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

- スーパーコンピュータは一つのノードの構成としては普通のコンピュータと基本的には変わらないが・・・
- トータルとしての計算能力・演算性能がきわめて高い
- そのためには高速なインターコネクトが要求される
- またシステムトータルとしての高い省電力性能・高信頼性が高いレベルで要求される

アプリケーションの 性能とは？

スパコンはシミュレーションに使う

科学について

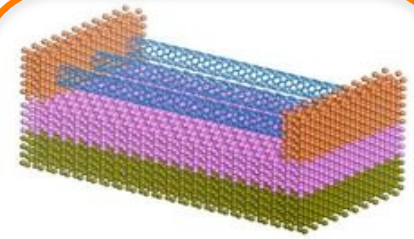
第3の科学

理論

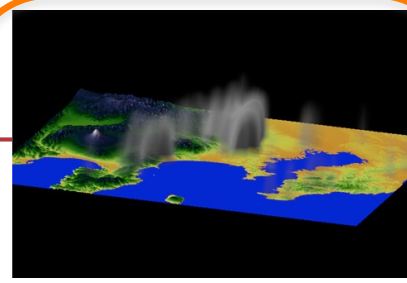
従来の
科学は

実験

具体的なアプリケーション



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

$10^{21}m$



10^7m



10^2m

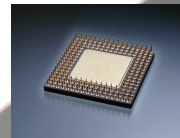


10^0m

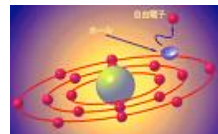


ポリオウイルス

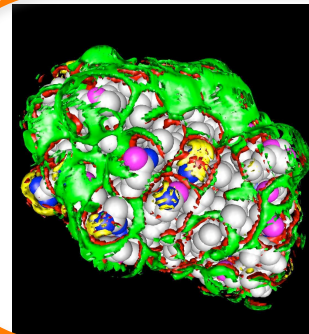
水中のウイルスの丸ごとシミュレーションによる医療への貢献



$10^{-8}m$

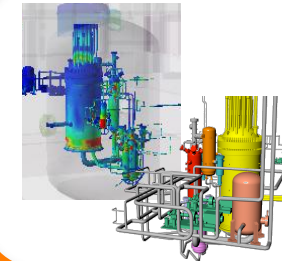


$10^{-10}m$

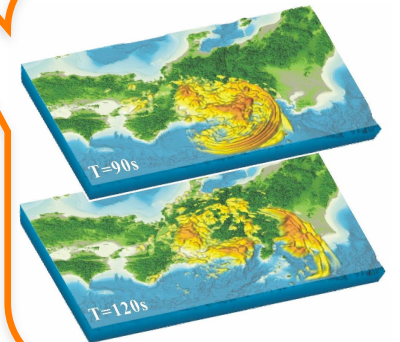


セルロース分解酵素のシミュレーションによる安価なバイオ燃料の提供等エネルギーへの貢献

29

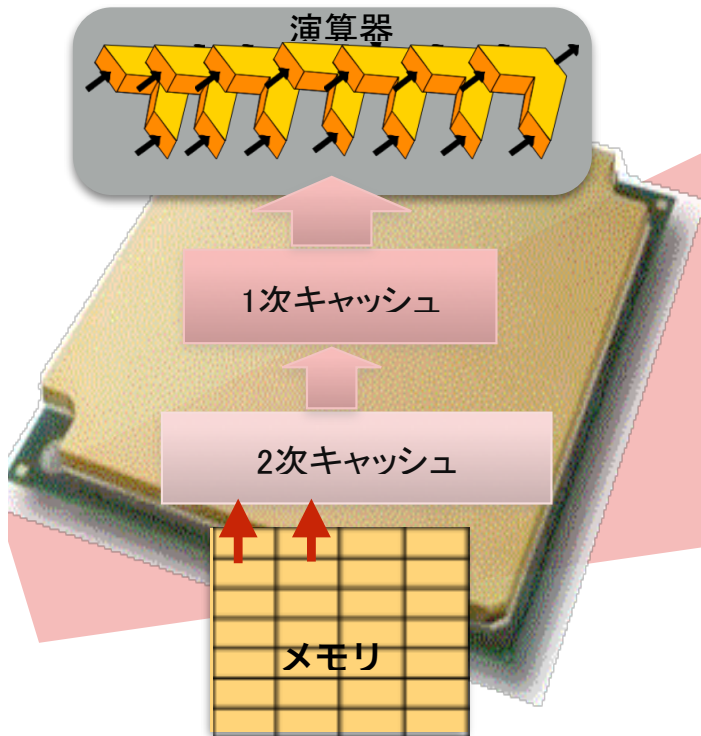


短周期地震波動の地震波シミュレーション、構造物の耐震シミュレーションを組み合わせた防災への貢献

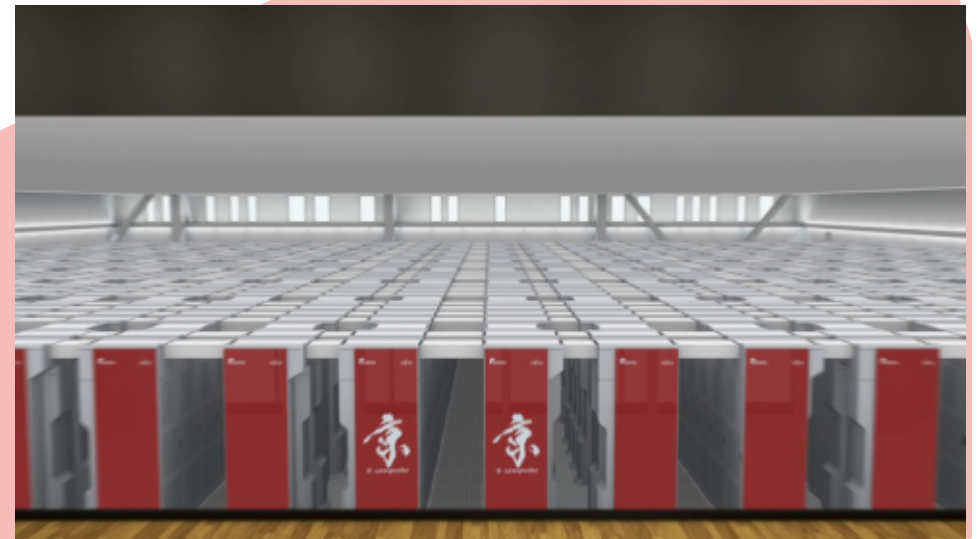


現代のスパコン利用の難しさ

現代のプロセッサ



現代のスーパーコンピュータシステム

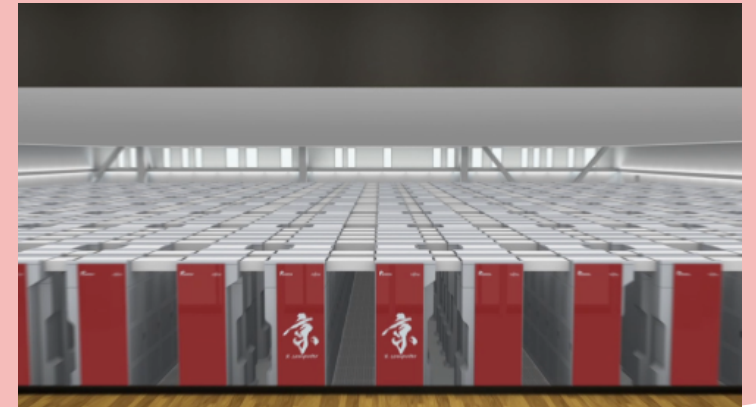


現代のスパコン利用の難しさ

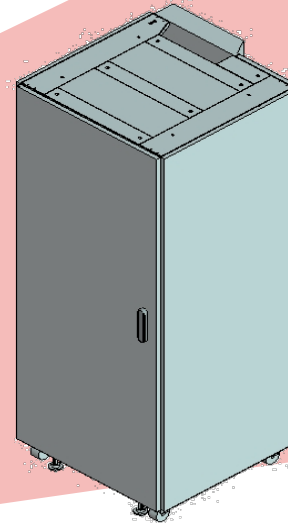
アプリケーションの
超並列性を引き出す

プロセッサの単体性能
を引き出す

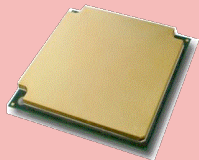
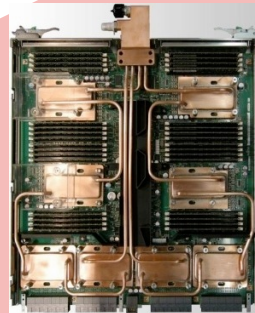
System



Rack

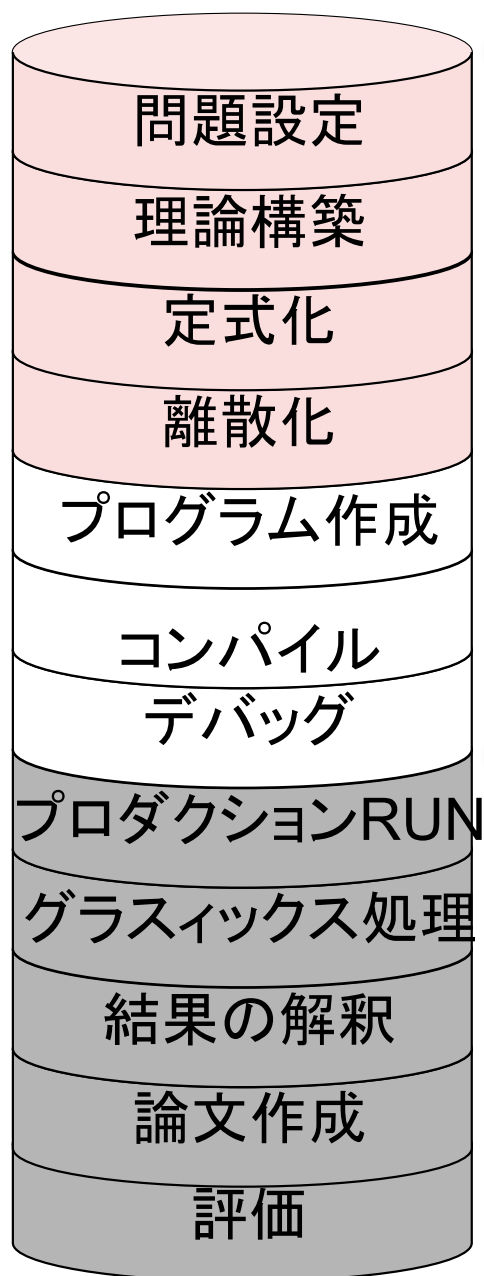


System Board



計算科学

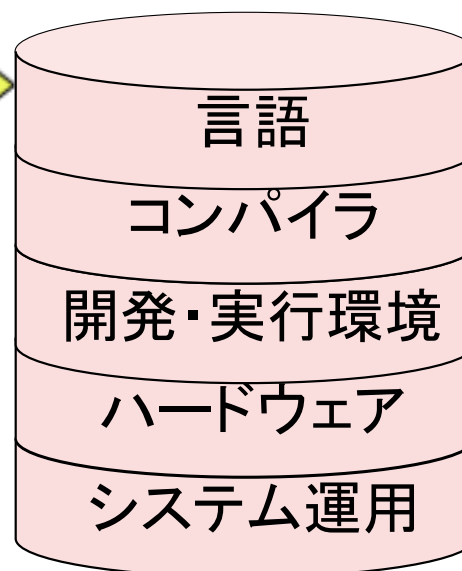
計算機科学



理論に忠実な分かりやすいコーディング

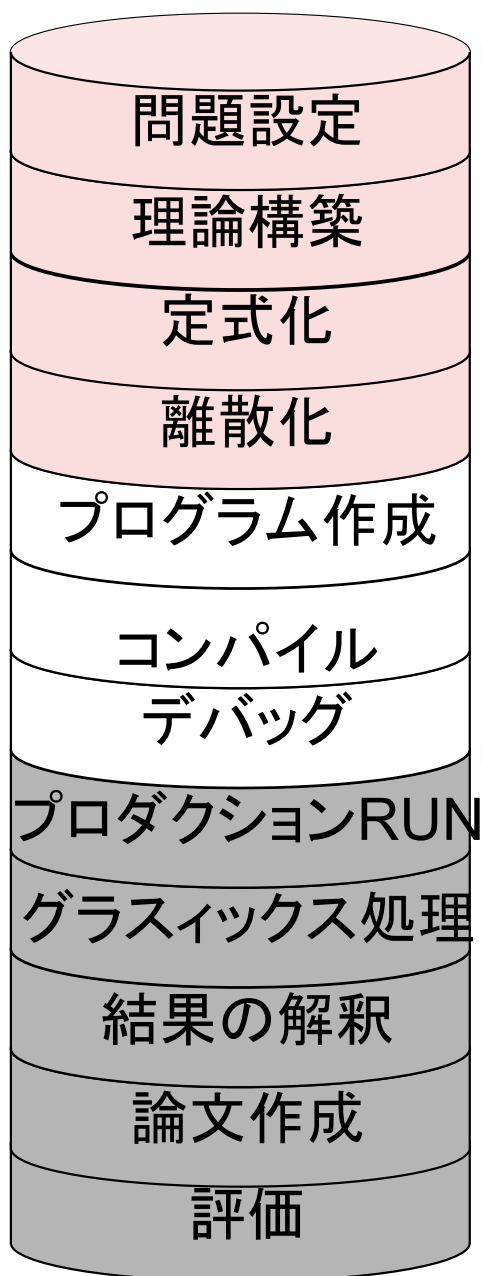


昔のプログラミング



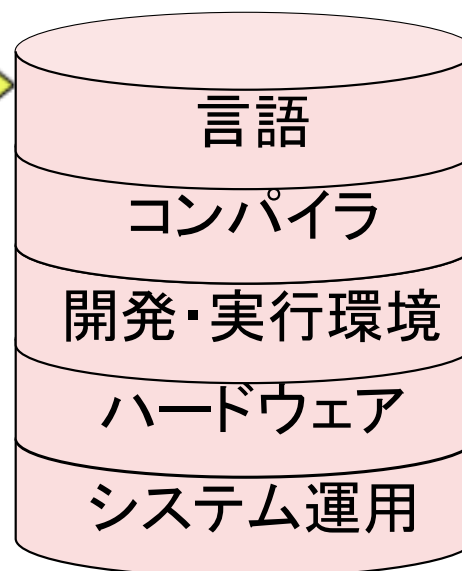
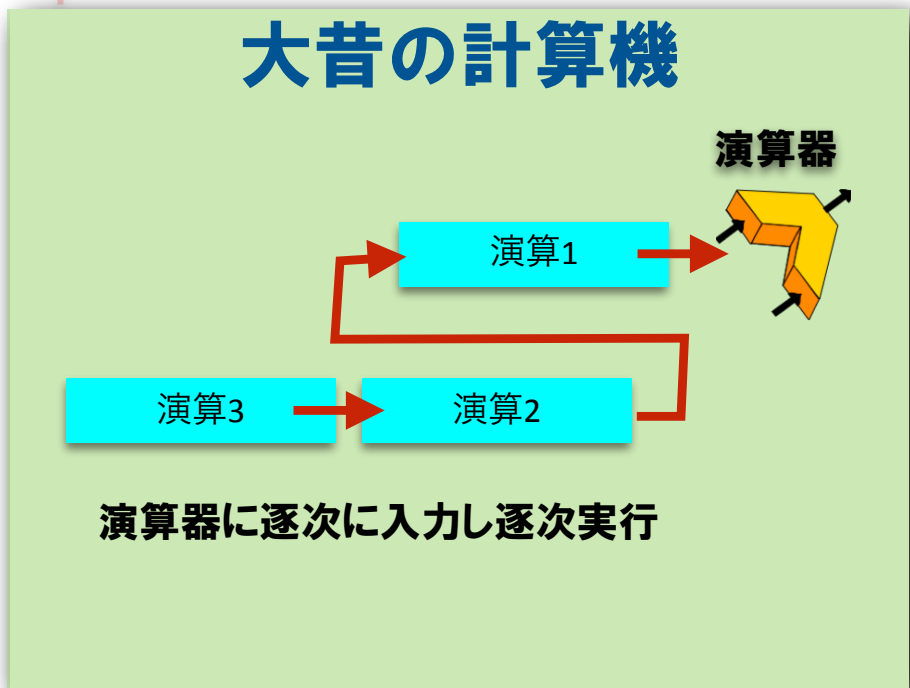
計算科学

計算機科学



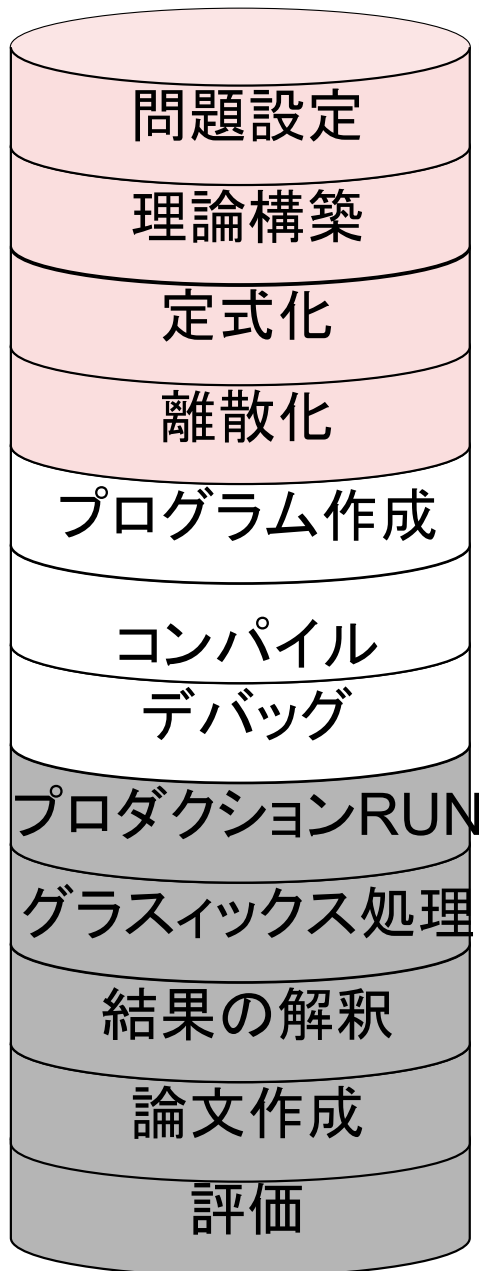
理論に忠実な分かりやすいコーディング

プログラム



計算科学

計算機科学



理論に忠実な分かりやすいコーディング

プログラム

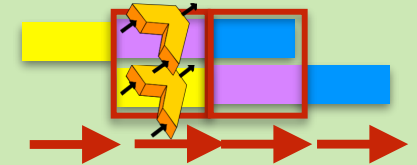
性能向上

(1) そのまま実行すると6clock

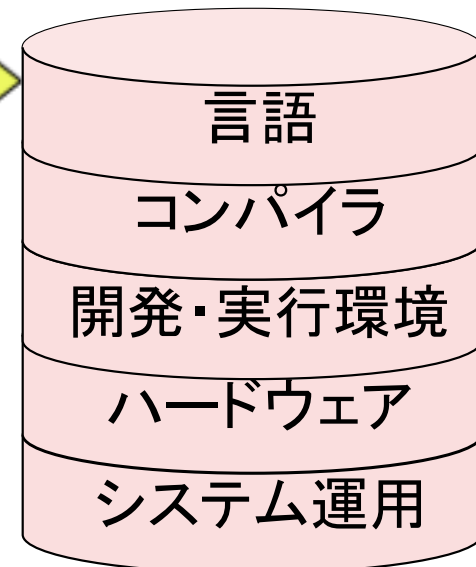
(2) 演算を3つのステージに分割する



(3) 並列に計算できるようにスケジューリング
(4) 2個の演算資源を使い並列に実行
(5) 4clockで実行可



(6) 演算を細かく分割し複数の演算資源を並列に動作し性能アップ



計算科学

計算機科学



理論に忠実な分かりやすいコーディング

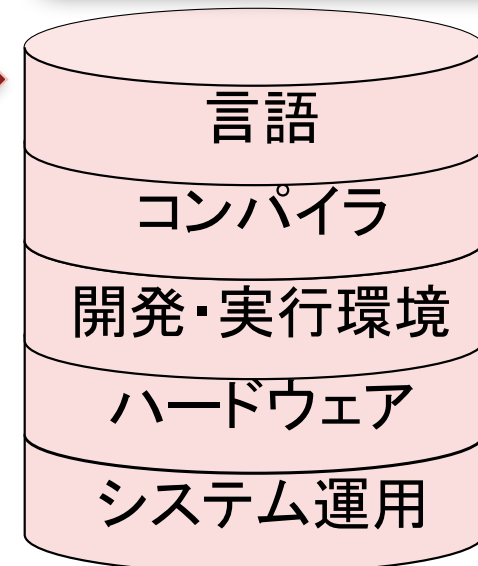
プログラム

プログラム

高並列化・高性能コーディング

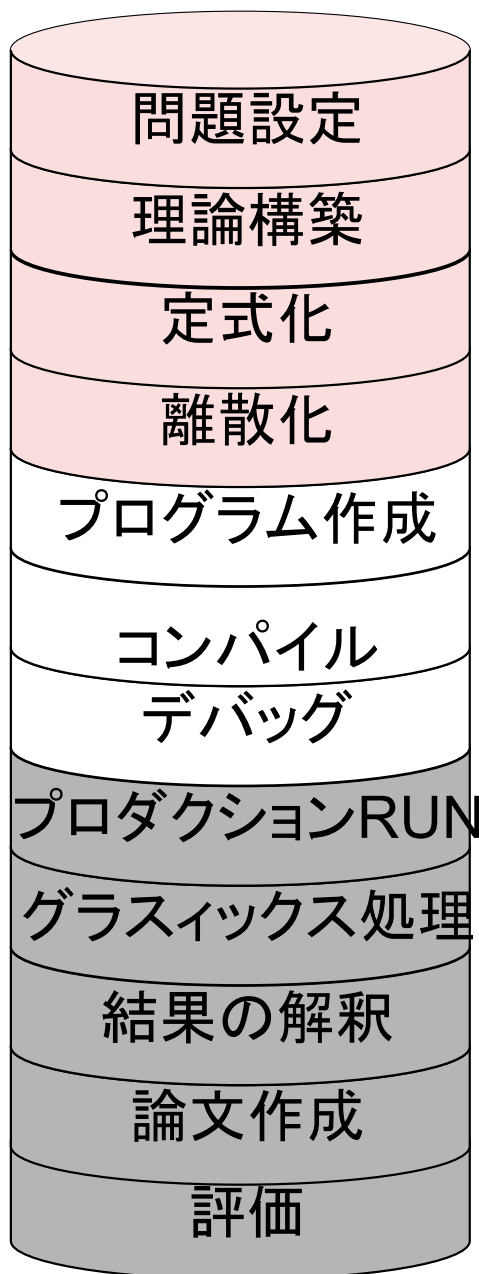
現代のプログラミング
(スパコンを使う場合)

- アプリケーションの**超並列性**を引き出す
- プロセッサの**単体性能**を引き出す



計算科学

計算機科学

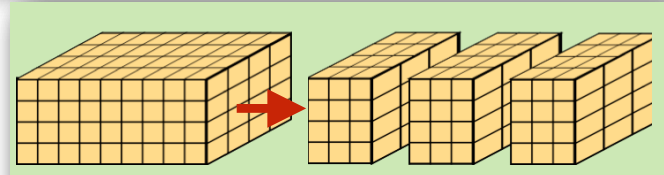


理論に忠実な分かりやすいコーディング

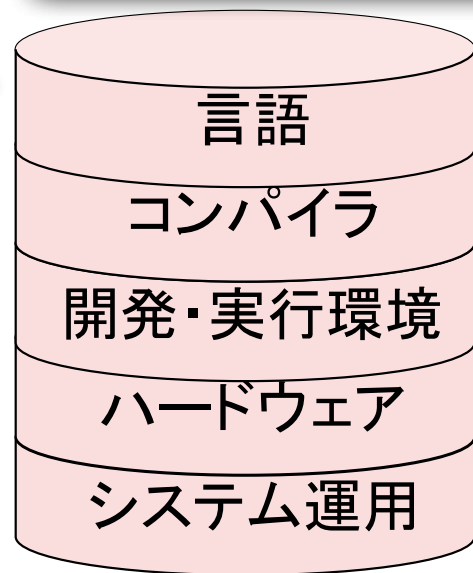


高並列化・高性能コーディング

現代のプログラミング (スパコンを使う場合)



- データを各プロセッサへ分割
- 処理を各プロセッサへ割当
- プロセッサ間での通信を記述
- キャッシュの有効利用プログラミング
- 演算器有効利用プログラミング
- メモリ性能有効利用プログラミング
- コンパイラ有効利用プログラミング

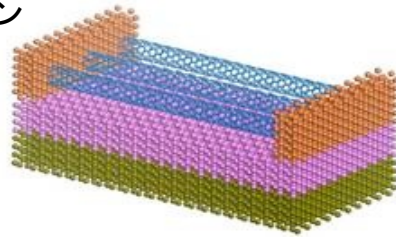


アプリケーションの性能についてまとめると

- プログラマーは、プロセス間の並列性を意識して並列化し、またプロセス毎のデータの分散を意識してプログラミングすることが必要となった。
- そうすることにより数千から数万に及ぶ**プロセス間の並列性**を最大限利用した超高速計算が実現可能となる
- また**シングルプロセッサの性能を最大限使いきる**プログラムのチューニングも必要となる
- もしシングルプロセッサの性能の1%しか出せなければ、並列化されていてもシステムトータルでも1%の性能

スーパーコンピュータの威力

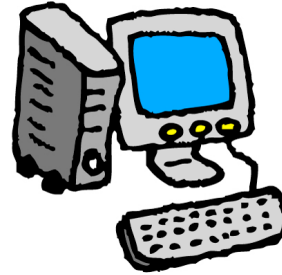
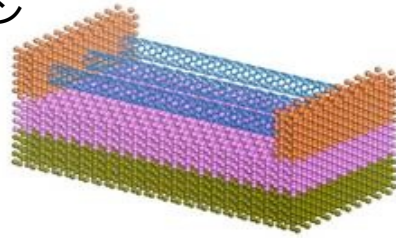
例えば次世代のデバイス
全体のシミュレーション



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

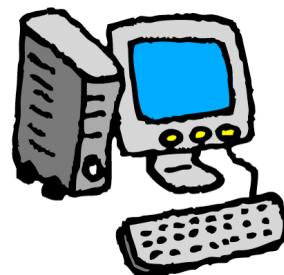
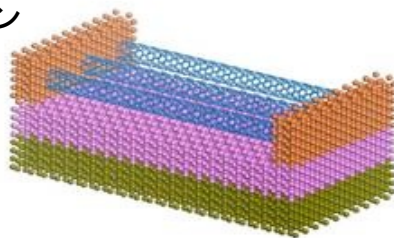


パソコンで100年かかる計算



スーパーコンピュータの威力

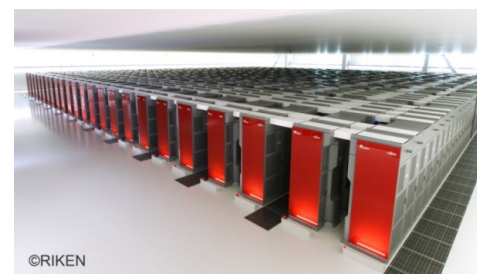
例えば次世代のデバイス
全体のシミュレーション



パソコンで100年かかる計算

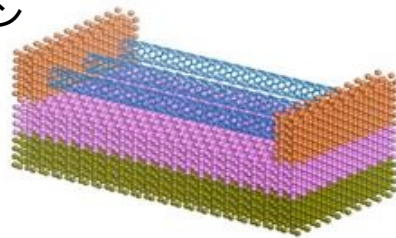


京を使って
73000倍
の速度で実行



スーパーコンピュータの威力

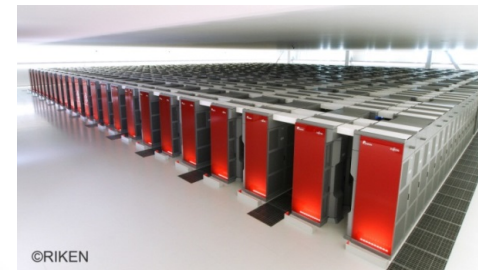
例えば次世代のデバイス
全体のシミュレーション



パソコンで100年かかる計算



京を使って
73000倍
の速度で実行



半日で終わる

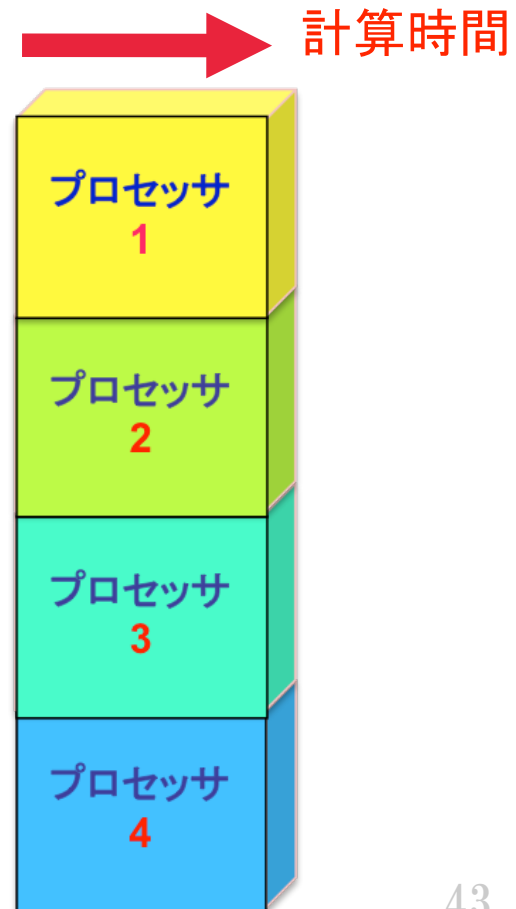
高並列化のための重要点

そもそも並列化とは？ (1)

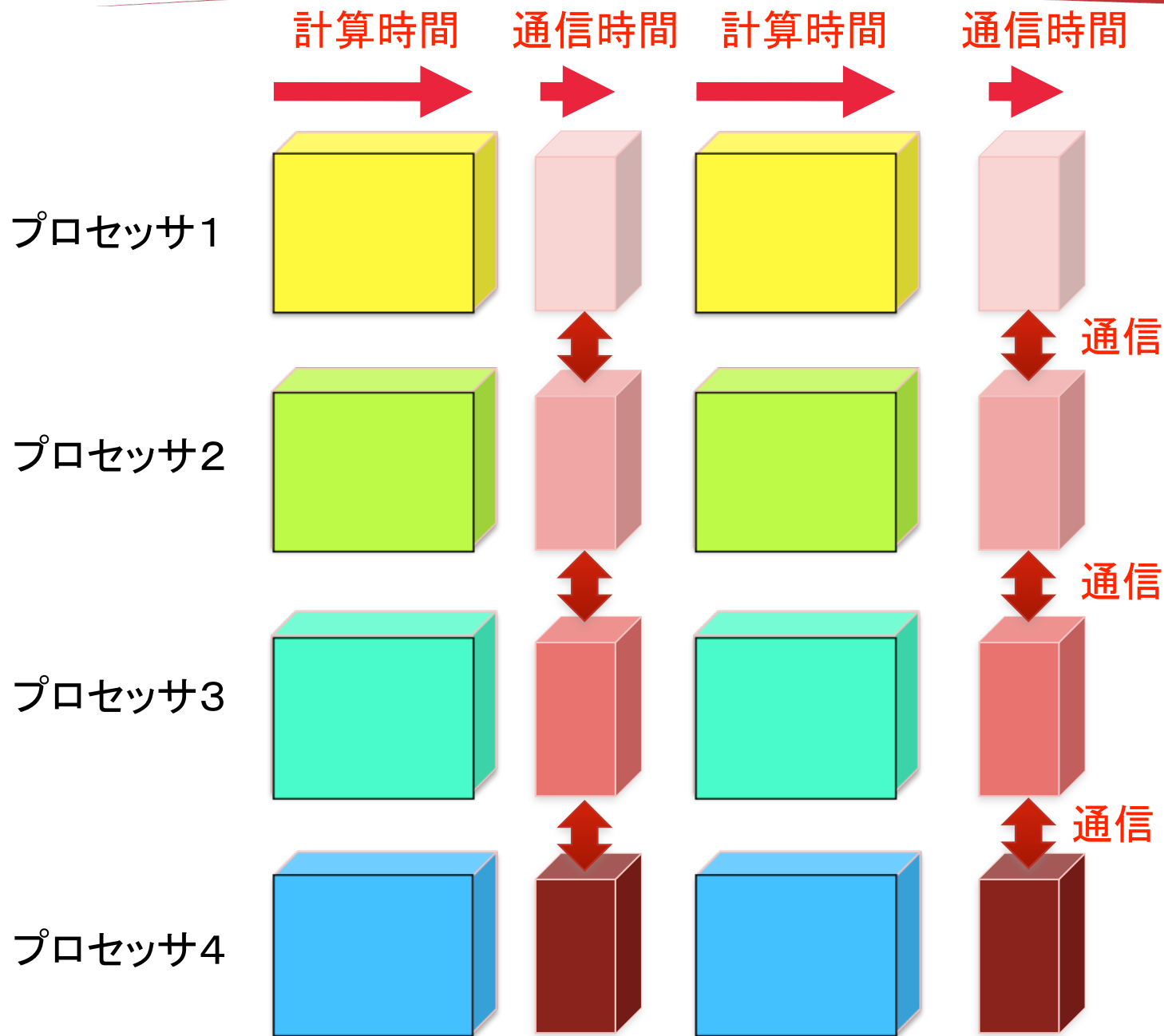
逐次計算



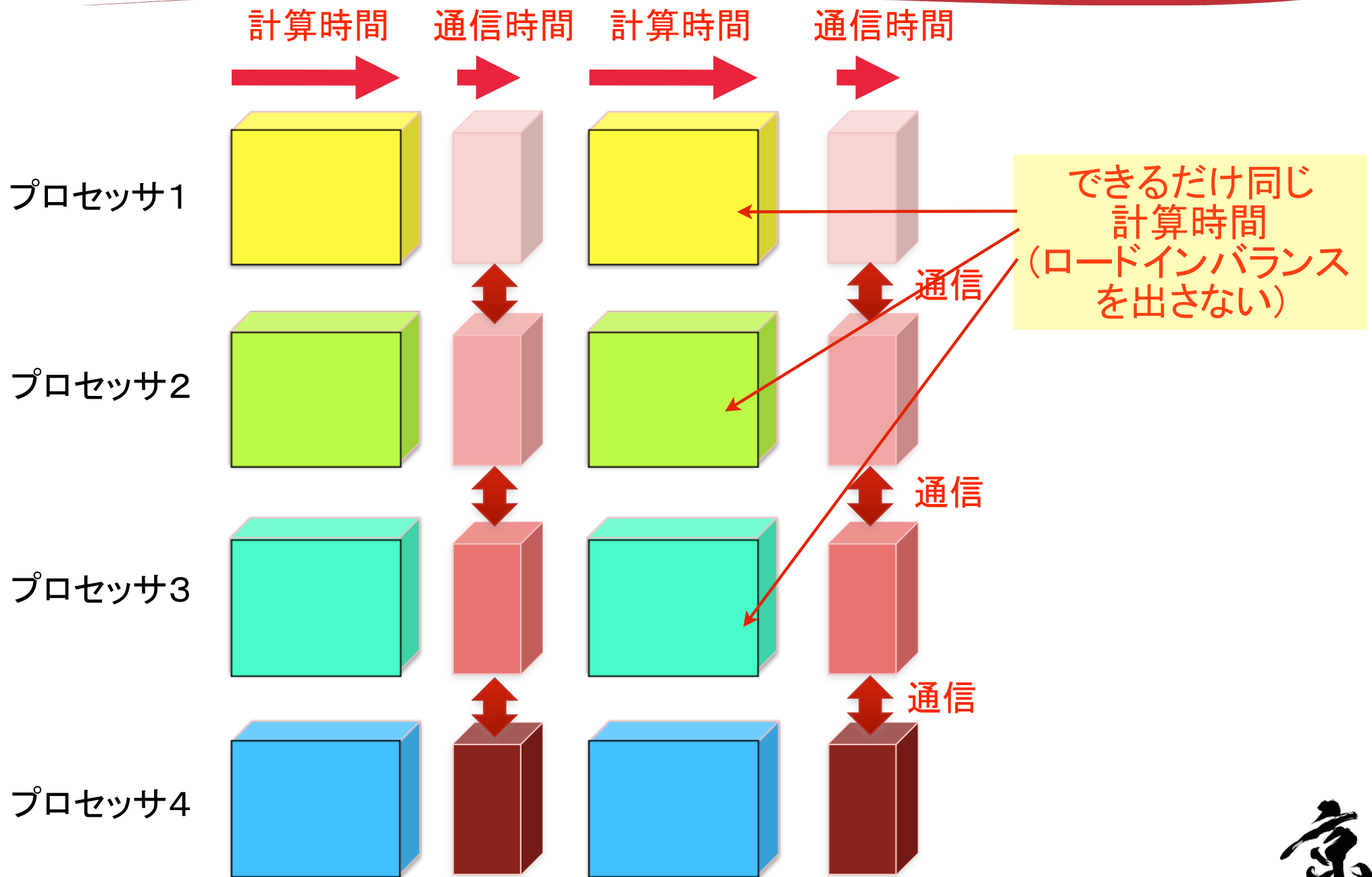
並列計算



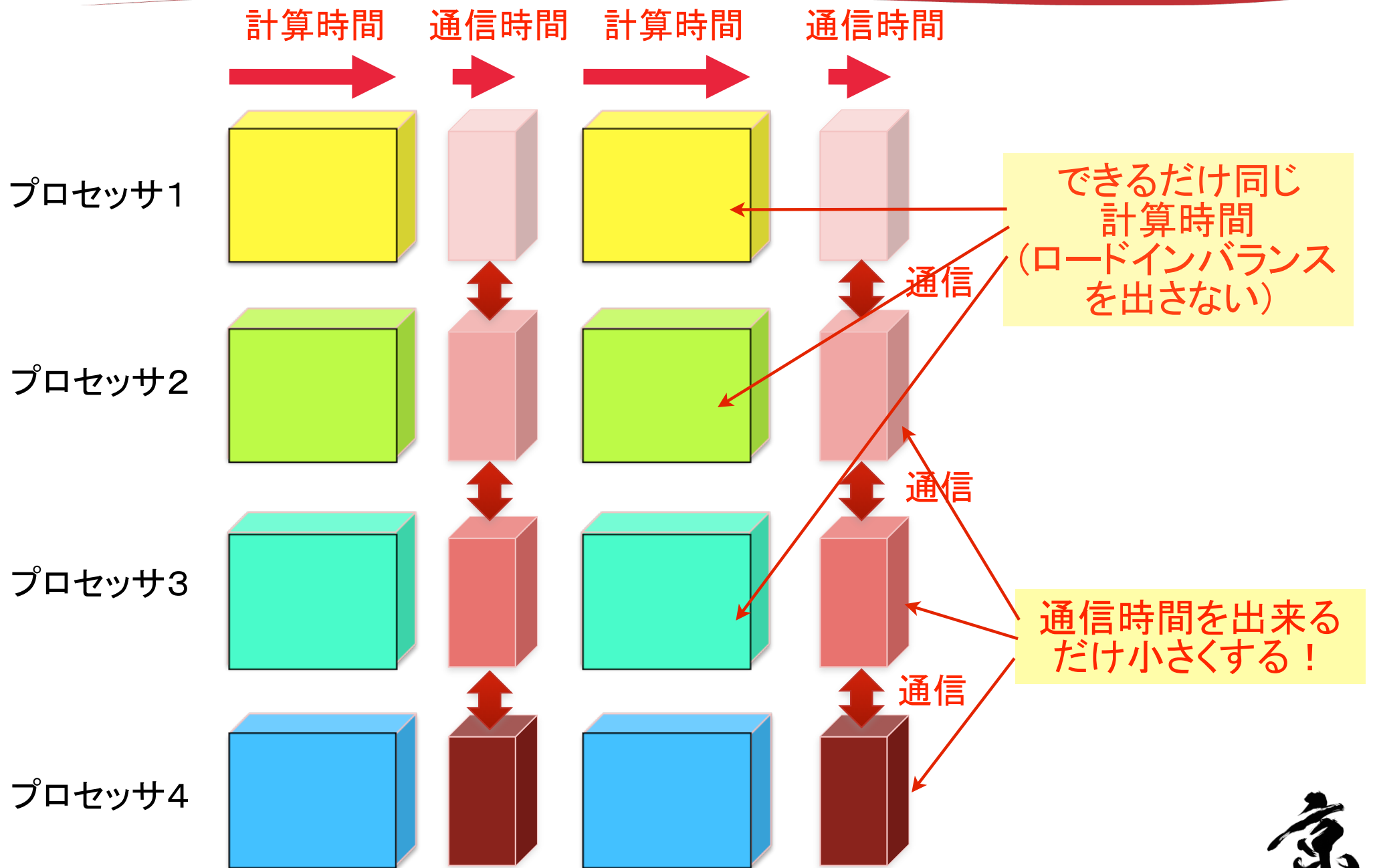
そもそも並列化とは？ (2)



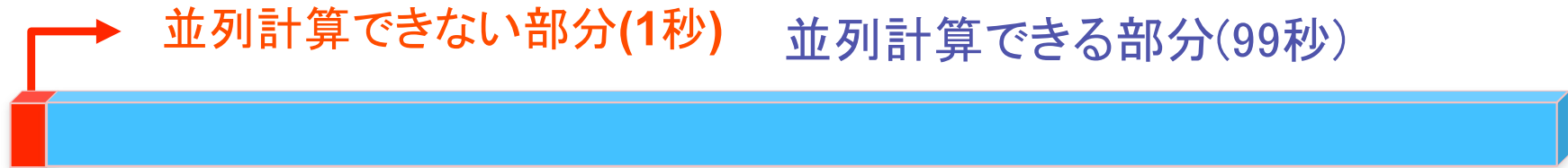
そもそも並列化とは？ (2)



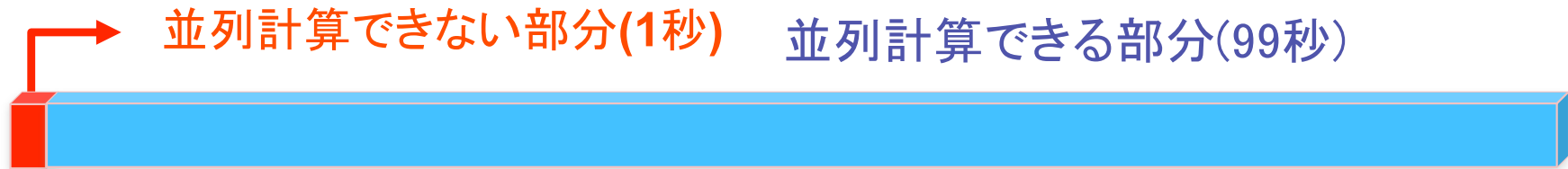
そもそも並列化とは？ (2)



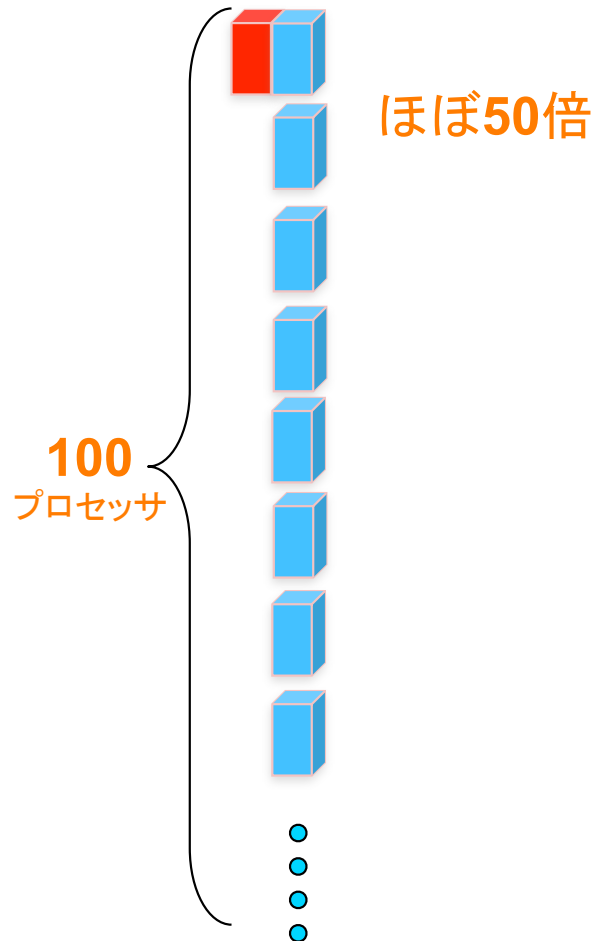
そもそも並列化とは？ (3)



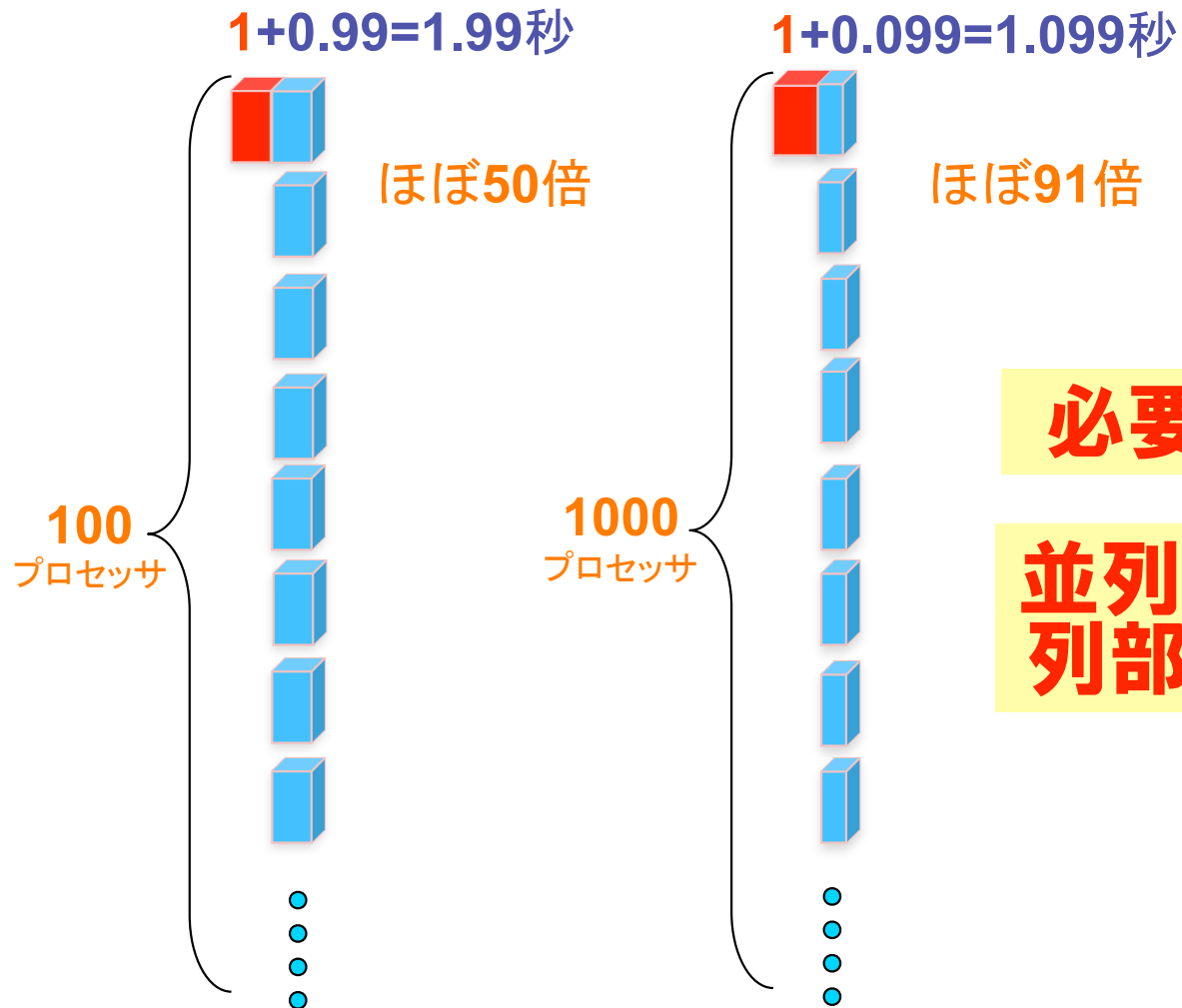
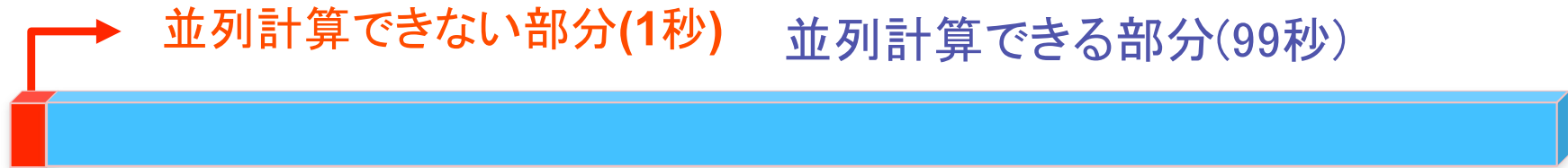
そもそも並列化とは？ (3)



$$1+0.99=1.99\text{秒}$$



そもそも並列化とは？ (3)

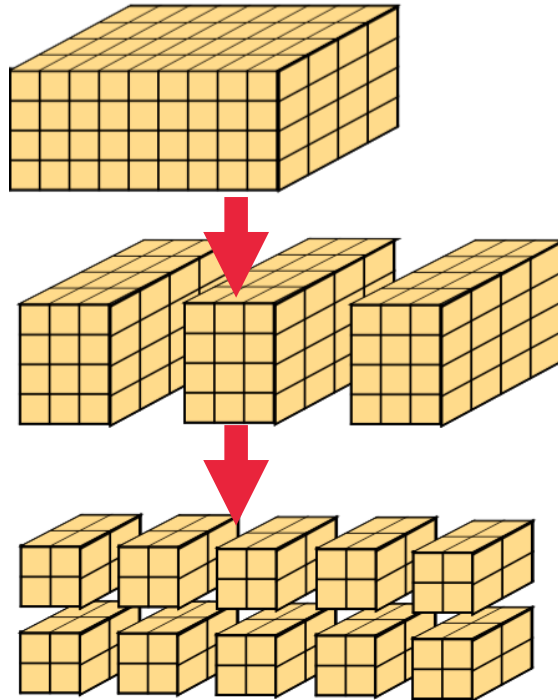


必要な並列度を確保する！

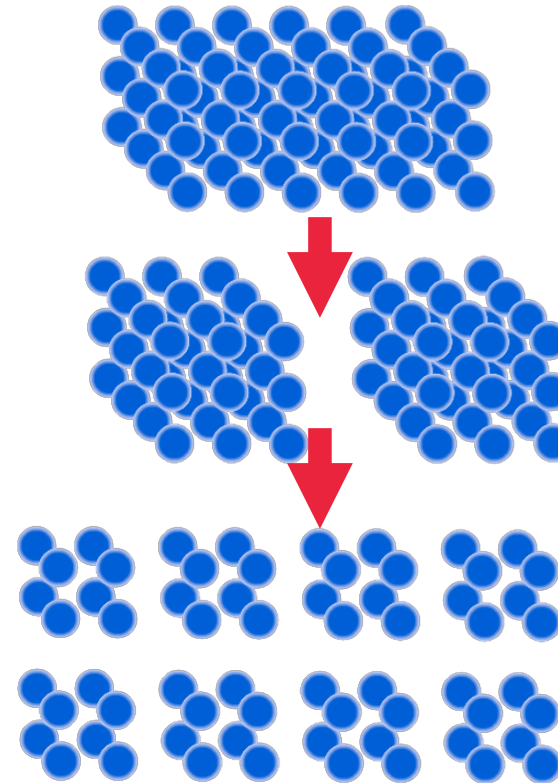
並列計算できない部分(非並列部)を限りなく小さくする！

必要な並列度を確保するとは？

領域分割



原子分割



- 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通信ばかりになる
- 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通信ばかりになる
- 後は領域分割と同様

非並列部が問題になる場合

- 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.
- 領域分割されていぬ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,iksntl,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-end do
          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
```

単体性能向上のための重要点

プロセッサの単体性能を引き出す(1)

- 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

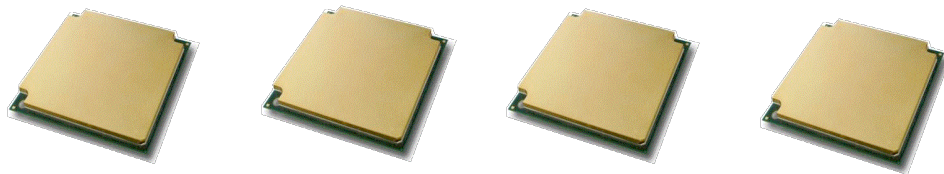
メモリ

メモリウォール問題

データ

- ✓ 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

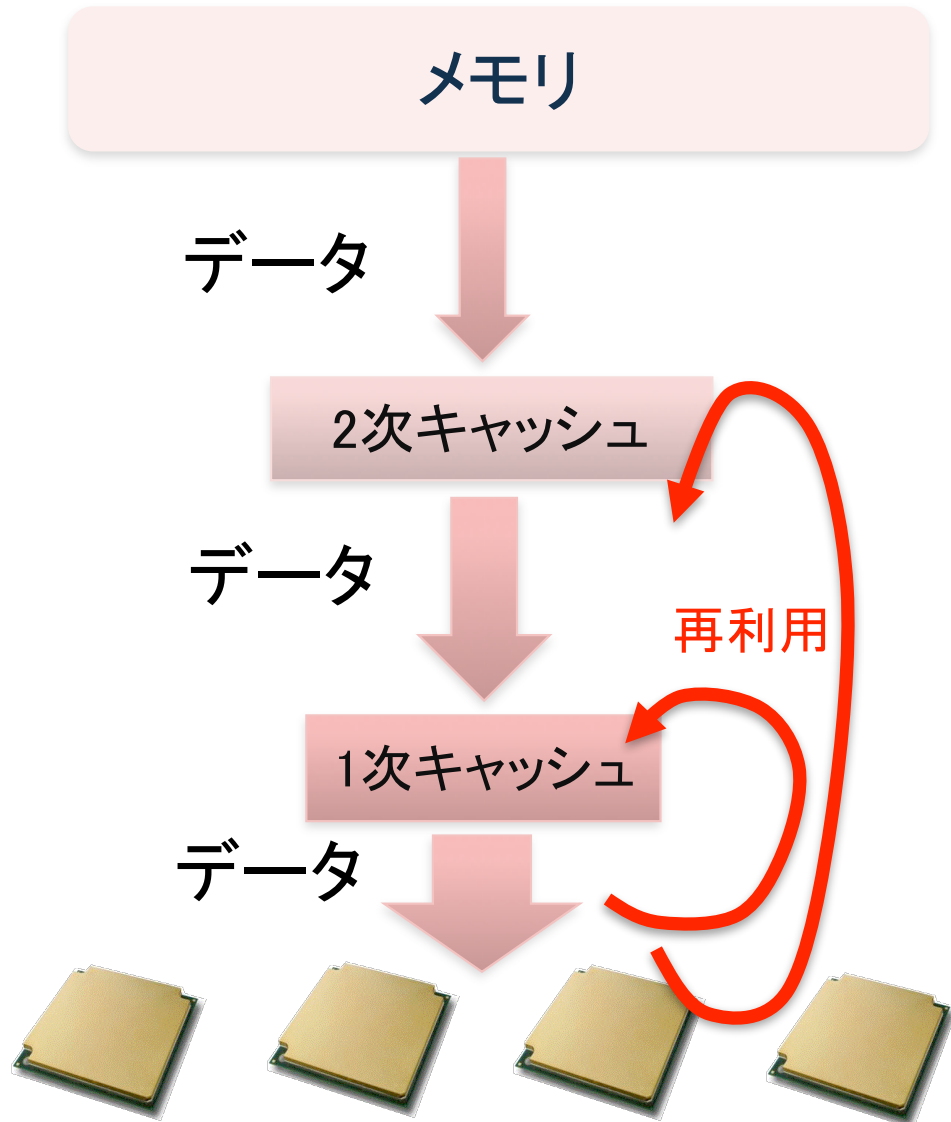
- 現代の計算機は演算器の能力が高くなりメモリのデータ供給能力が相対的に不足している



演算器

プロセッサの単体性能を引き出す (2)

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
- ・こうすること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)

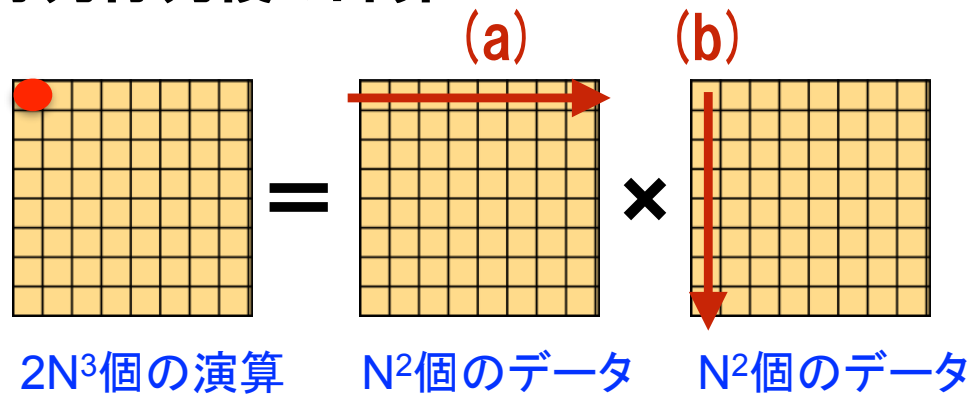
例えば行列行列積
($2N^2$ 個のデータで $2N^3$ 個の演算可)

(*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

もう少し詳しく説明すると

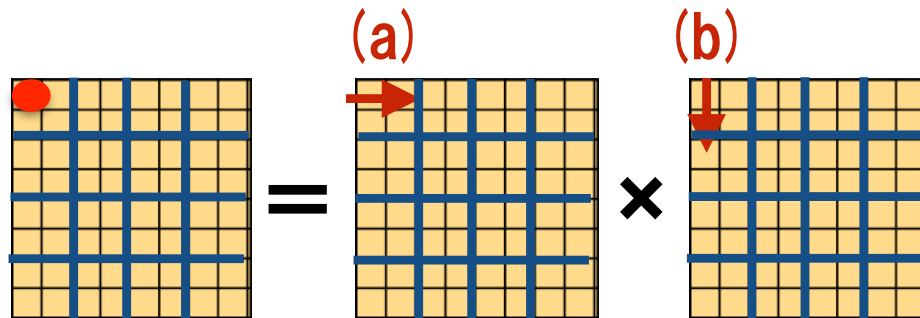
行列行列積の計算



$$\begin{aligned} \text{B/F値} &= \text{移動量(Byte)}/\text{演算量(Flop)} \\ &= 2N^2/2N^3 \\ &= 1/N \end{aligned}$$

原理的にはNが大きい程小さな値

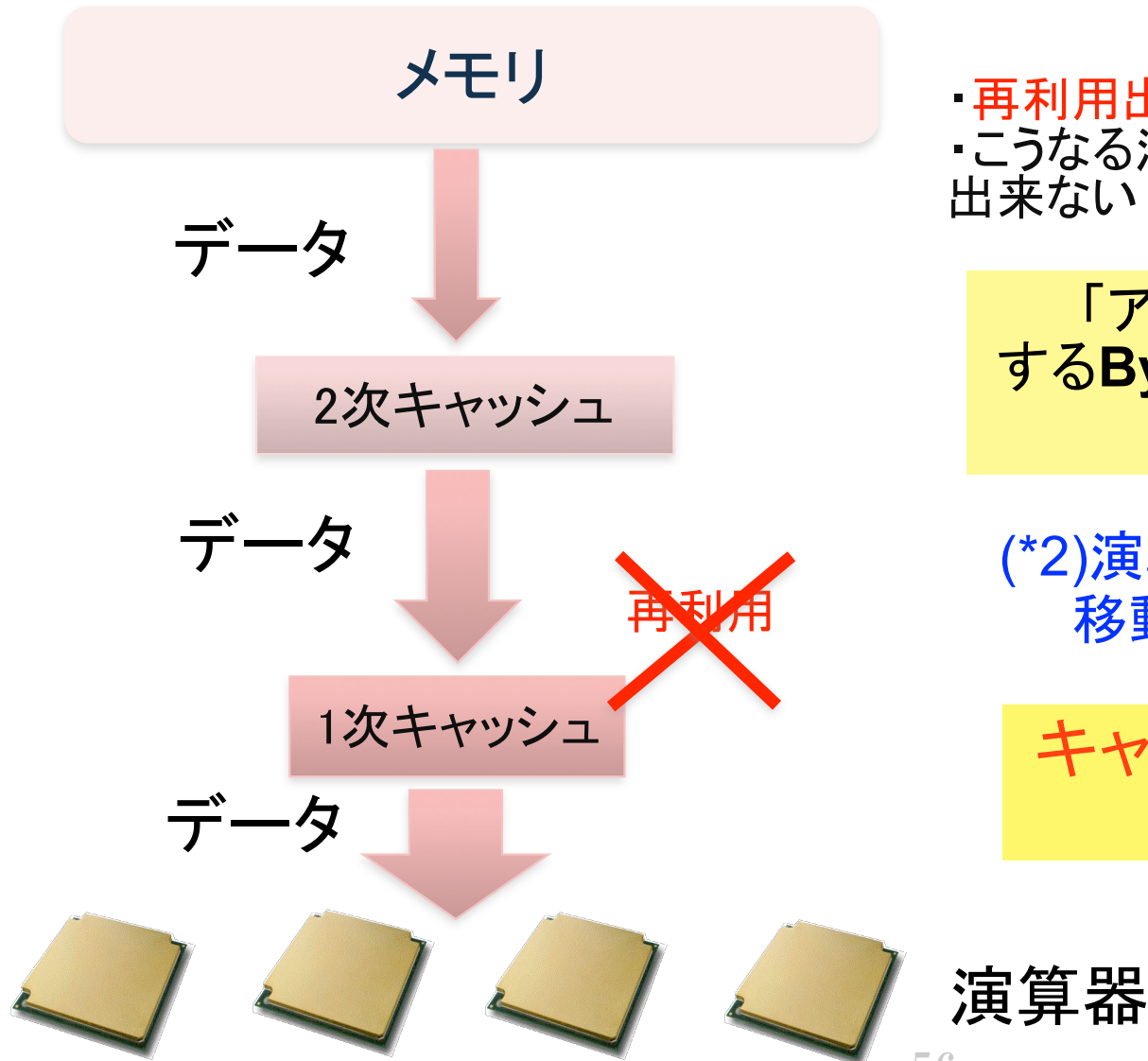
- 現実のアプリケーションではNがある程度の大きさになるとメモリ配置的には (a) はキャッシュに乗っても (b) は乗らない事となる



- 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。

プロセッサの単体性能を引き出す (3)

とは言っても……



- ・再活用出来ない問題もある
- ・こなる演算器の能力を十分使い切る事が出来ない

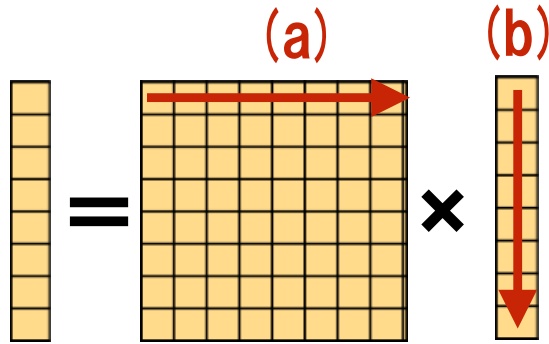
「アプリケーションが要求するByte/Flop値が高い」タイプの計算(*2)

(*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

例えば

行列ベクトル積の計算



$2N^2$ 個の演算

N^2 個のデータ

N 個のデータ

$$\begin{aligned} & \text{B/F値} \\ & = \text{移動量(Byte)}/\text{演算量(Flop)} \\ & = (N^2+N)/2N^2 \\ & \doteq 1/2 \end{aligned}$$

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B/F値は大きいので性能向上はできない。

まとめ

- **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？**
 - アプリケーションの超並列性を引き出す
 - プロセッサの単体性能を引き出す
- **高並列化のための重要点**
 - 並列度の確保
 - 非並列部を最小化する
 - 通信時間を再消化する
 - ロードインバランスを解消する
- **単体性能向上のための重要点**
 - B/F値とは？
 - キャッシュの有効利用ができる例
 - キャッシュの有効利用がしにくい例