スーパーコンピュータと アプリケーションの性能

2018年4月

国立研究開発法人理化学研究所
計算科学研究センター 運用技術部門
チューニング技術ユニット ユニットリーダー

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（CPU単体性能最適化）
- 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

K computer

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

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

2011年 iPhone4S
LINPACK 140Mflops

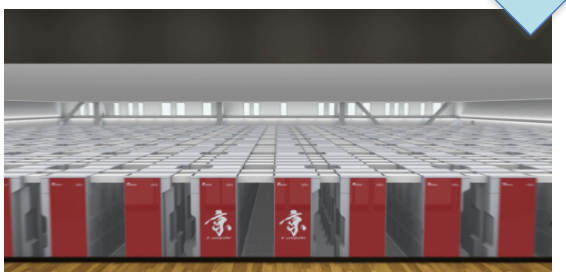
≒

2018年 iPhone8
LINPACK 3420Mflops

＞

6250万倍

2011年 「京」コンピュータ



10Pflop

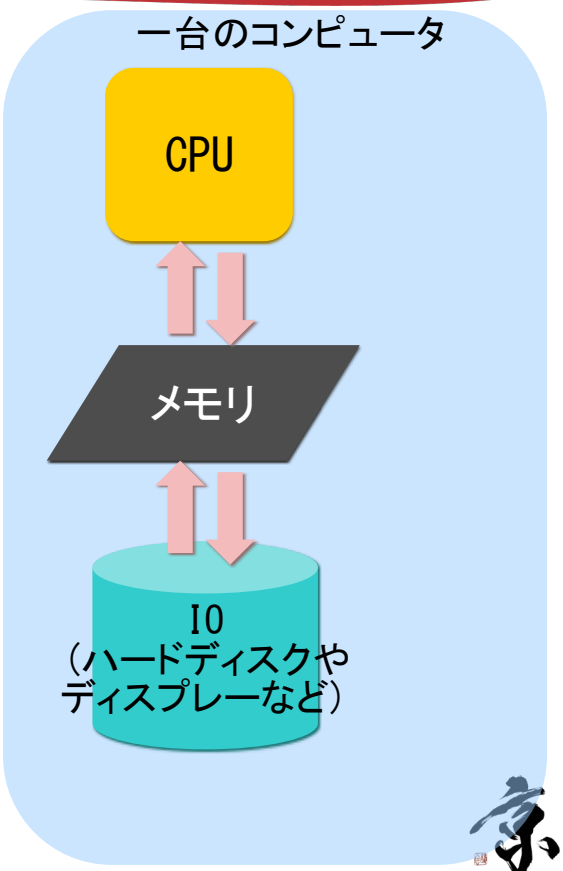
K computer

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に書出す（出力）



コンピュータとは？

ハードウェア

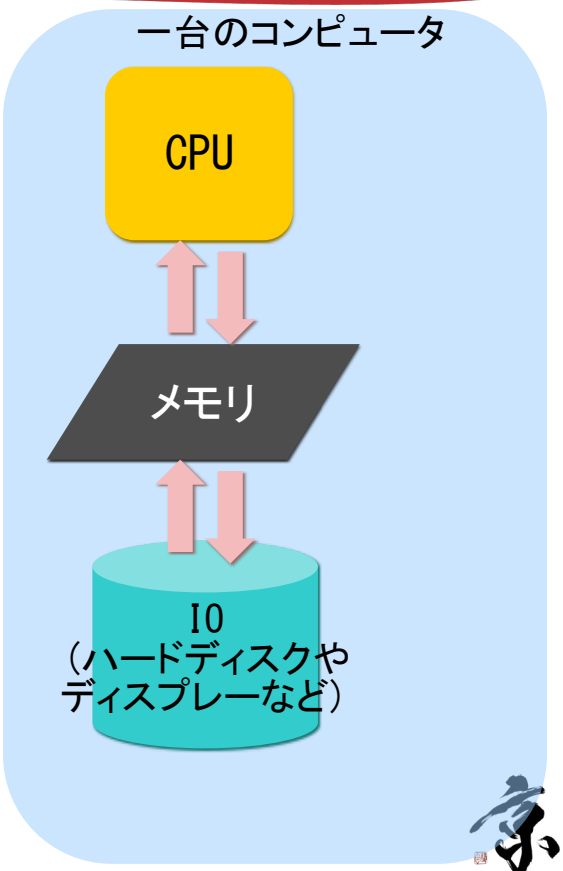
- CPU
- メモリ
- IO（入出力）

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

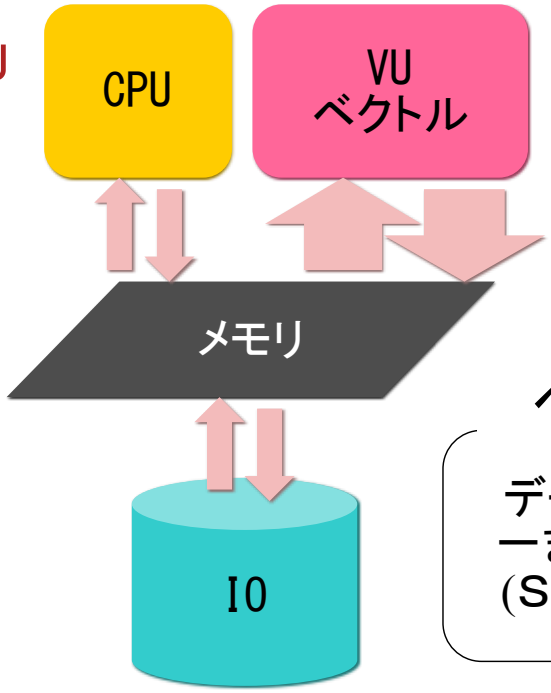
であった時代



昔のスーパーコンピュータ

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

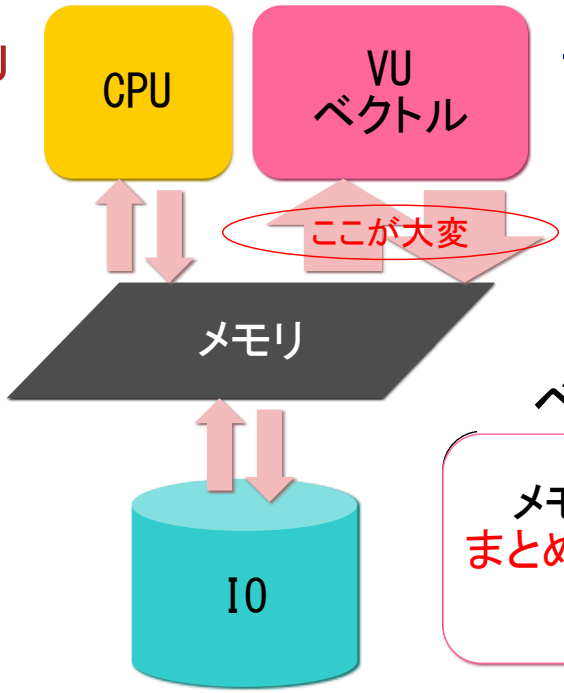
ベクトル機構

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)



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

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

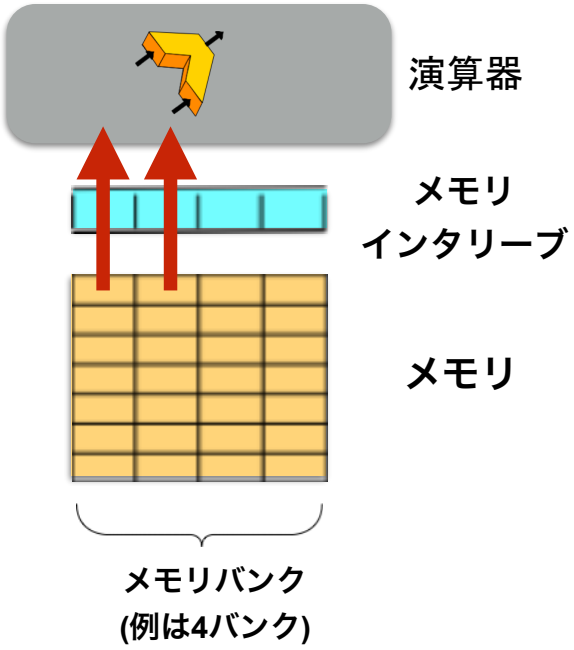
今もこの方式が無くなったわけではありませんが…

シングルプロセッサの問題

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)

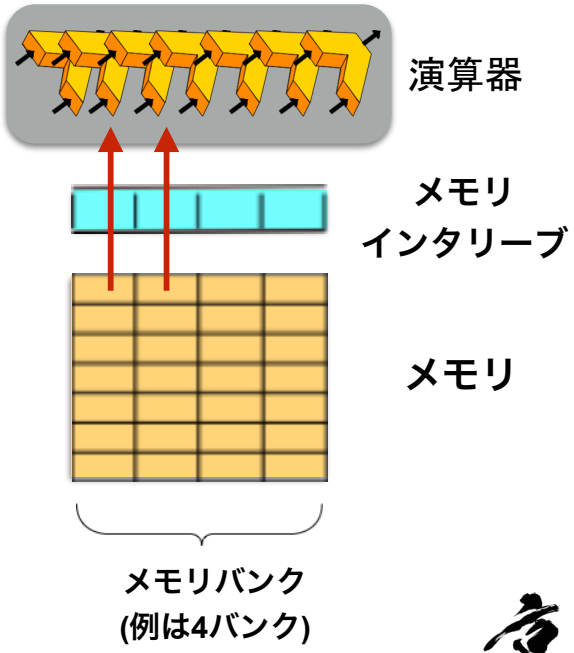


シングルプロセッサの問題

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

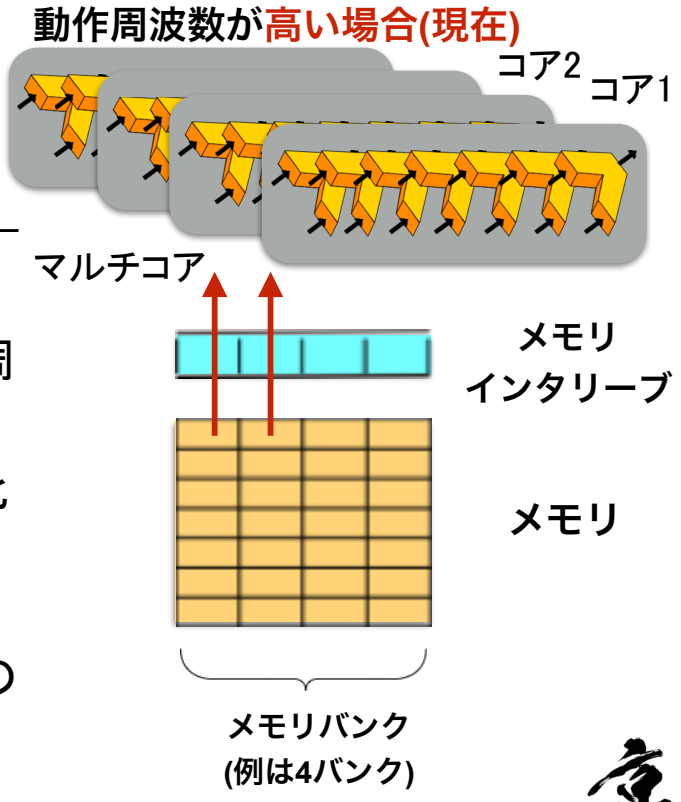
動作周波数が高い場合



シングルプロセッサの問題

メモリウォール問題

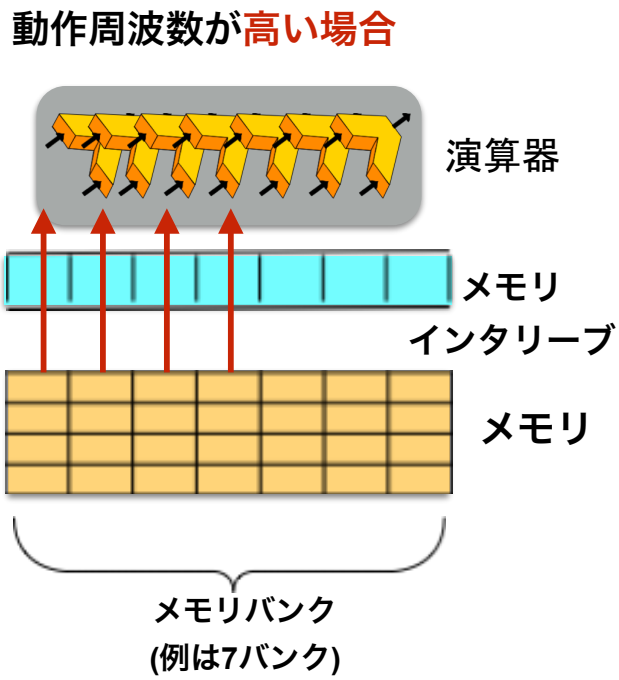
- メモリは動作周波数が高くなってくるともっと待つ事となる(100–200サイクル)
- メモリに比べて演算器は動作周波数が高くなると高速になった
- さらに半導体プロセスの微細化により演算器はCPUにたくさん搭載可能となった
- 結果的に演算器に比べメモリのデータ転送能力が低くなった



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

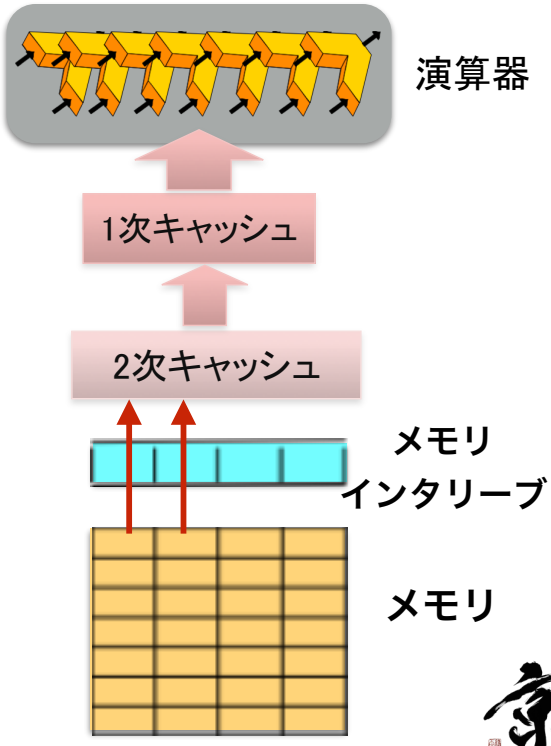


シングルプロセッサの問題

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

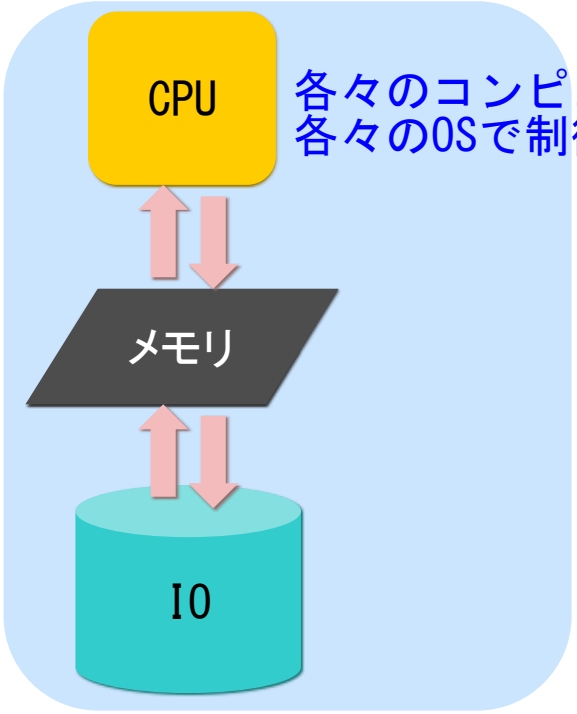
一台のコンピュータの処理限界

- 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

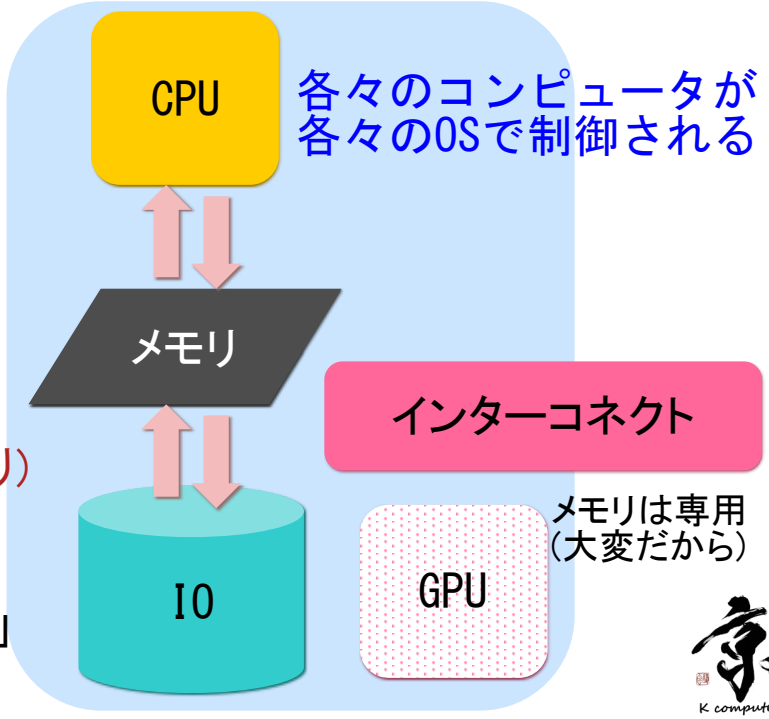
たくさんのコンピュータをつないで、システムとして一体で動かす

- CPU
- メモリ
- IO(入出力)

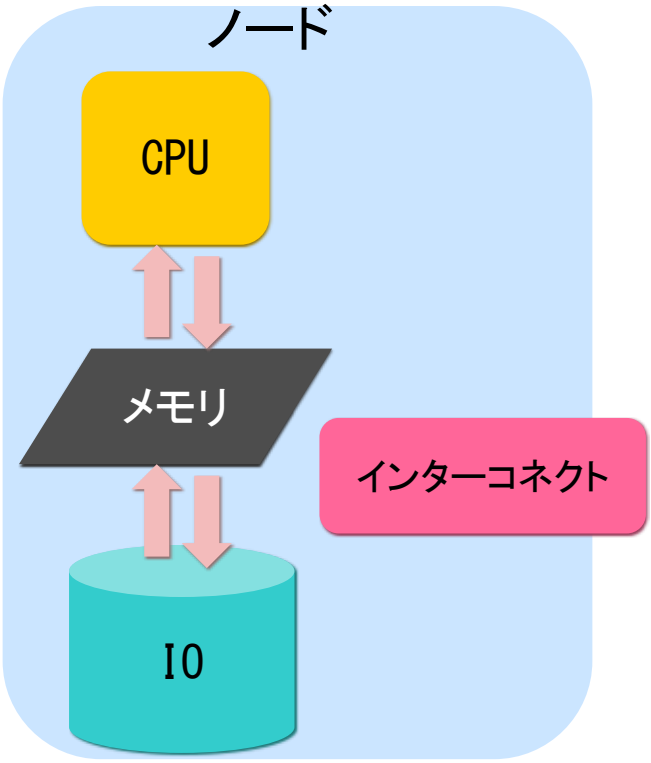
- インターコネクト(接続機構)

- GPU/アクセラレータ
(加速器:あったり、なかったり)

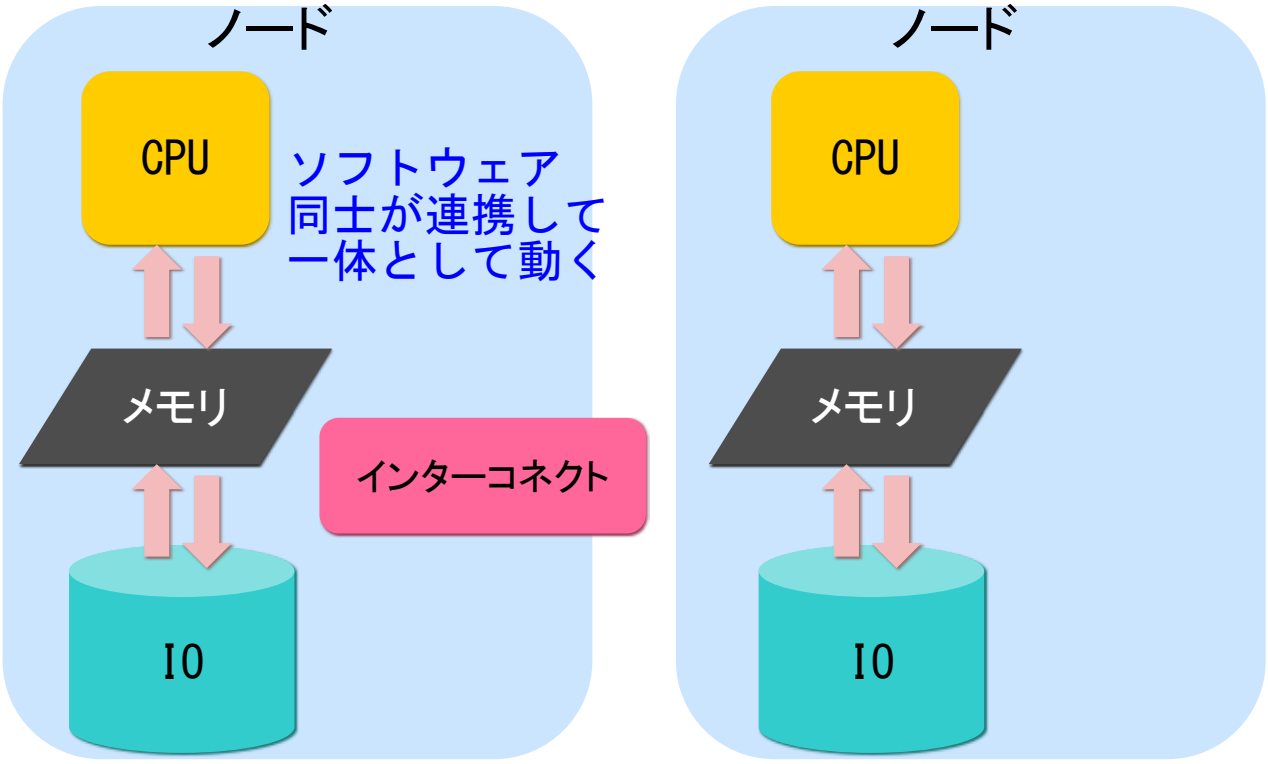
各々のコンピュータ=「ノード」



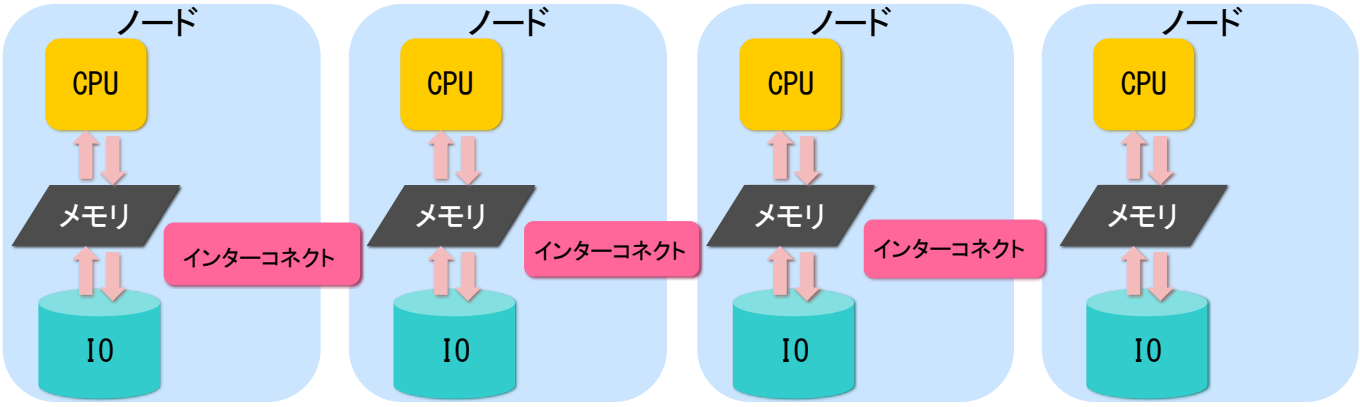
コンピュータ（ノード）どうしをつなぐ



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



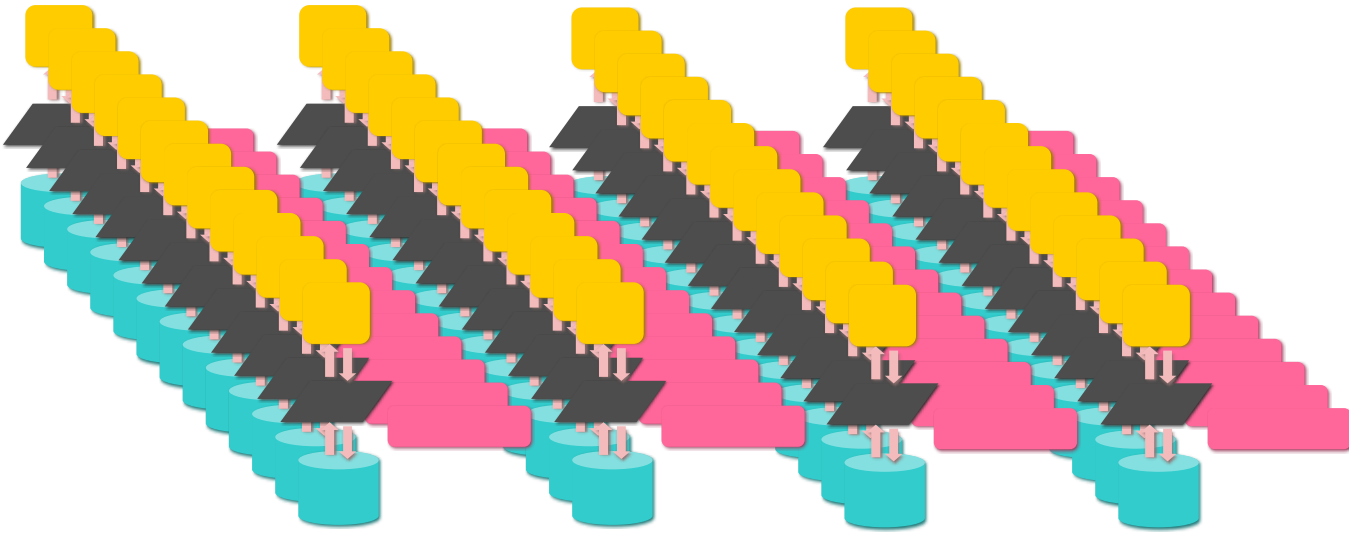
システムとして一体で動く

（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



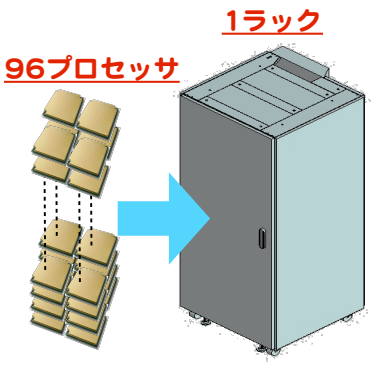
実際はハードディスクは、まとめる

ちなみに「京」の場合は？

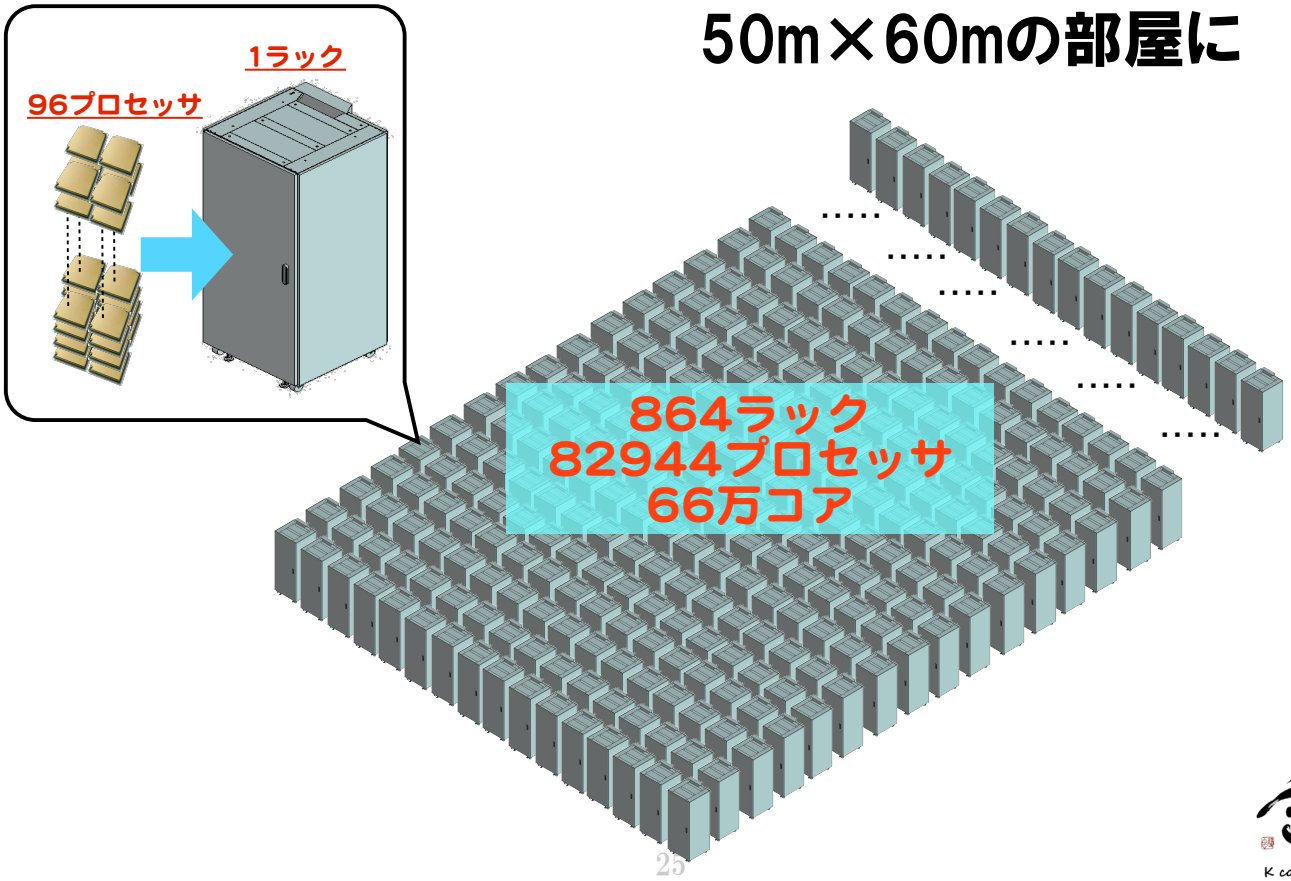
50m×60mの部屋に

ちなみに「京」の場合は？

50m×60mの部屋に

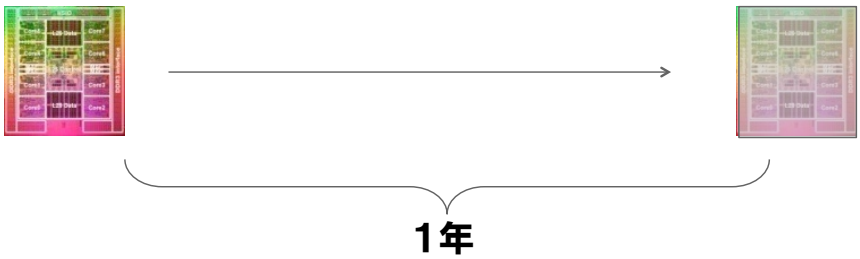


ちなみに「京」の場合は？

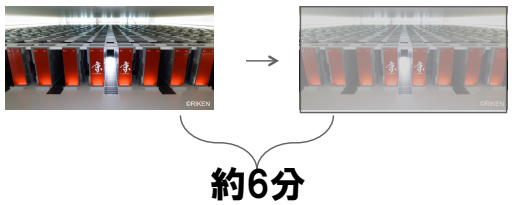


これだけ巨大システムでCPUが頻繁に故障すると・・・

✓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

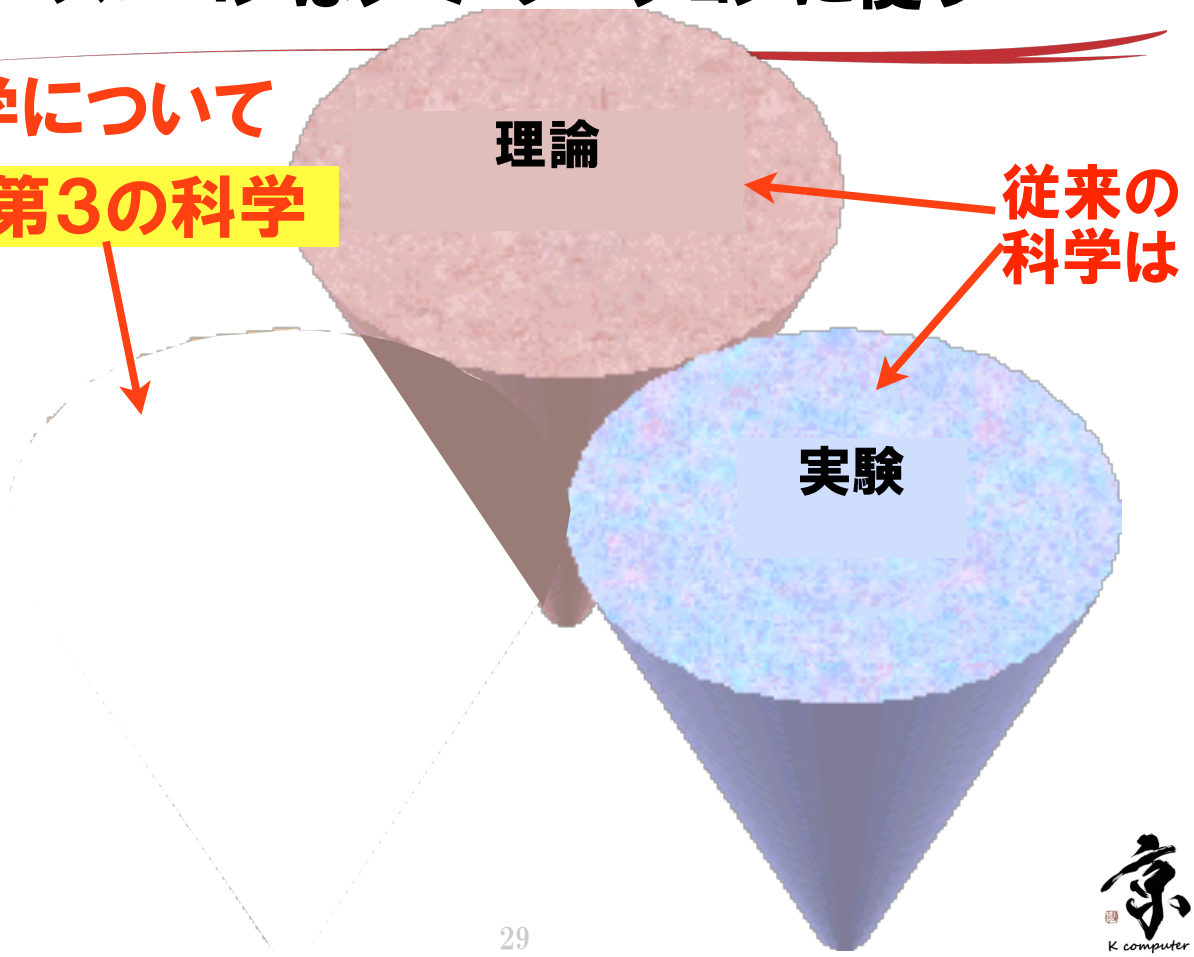
- スーパーコンピュータは一つのノードの構成としては普通のコンピュータと基本的には変わらないが・・
- トータルとしての計算能力・演算性能がきわめて高い
- そのためには高速なインターコネクトが要求される
- またシステムトータルとしての高い省電力性能・高信頼性が高いレベルで要求される

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

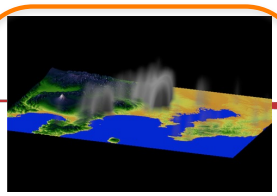
第3の科学



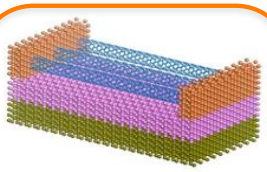
従来の科学は

K computer

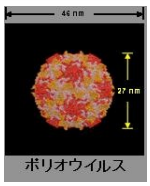
具体的なアプリケーション



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



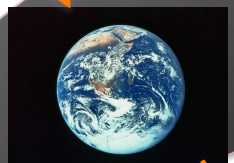
次世代のデバイス全体のシミュレーションによるエレクトロニクスへの貢献



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

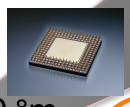
10^{7}m



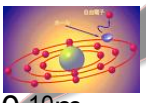
10^{2}m

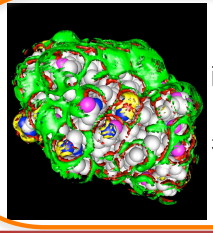
10^{0}m

10^{-8}m

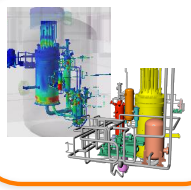


10^{-10}m





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

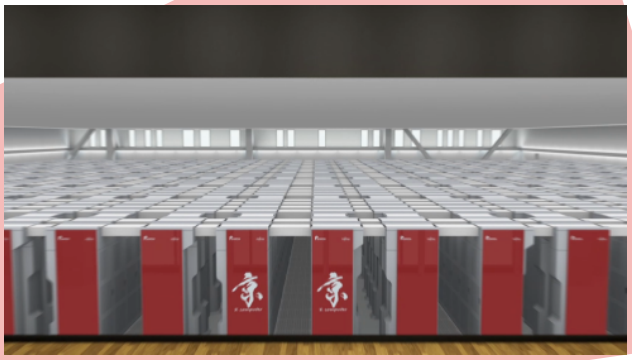


短周期地震波動の地震波シミュレーション、構造物の耐震シミュレーションを組み合わせた防災への貢献

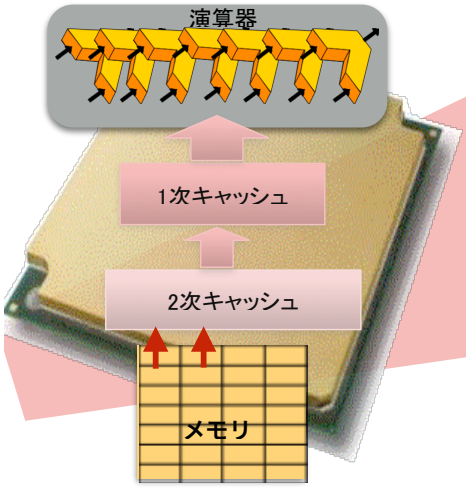
K computer

現代のスパコン利用の難しさ

現代のスーパーコンピュータシステム



現代のプロセッサ

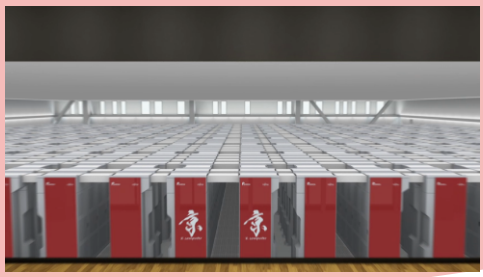


現代のスパコン利用の難しさ

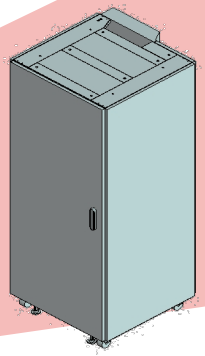
アプリケーションの
超並列性を引き出す

プロセッサの単体性能
を引き出す

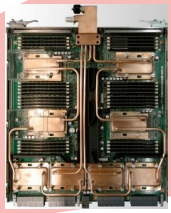
System

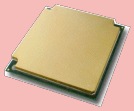


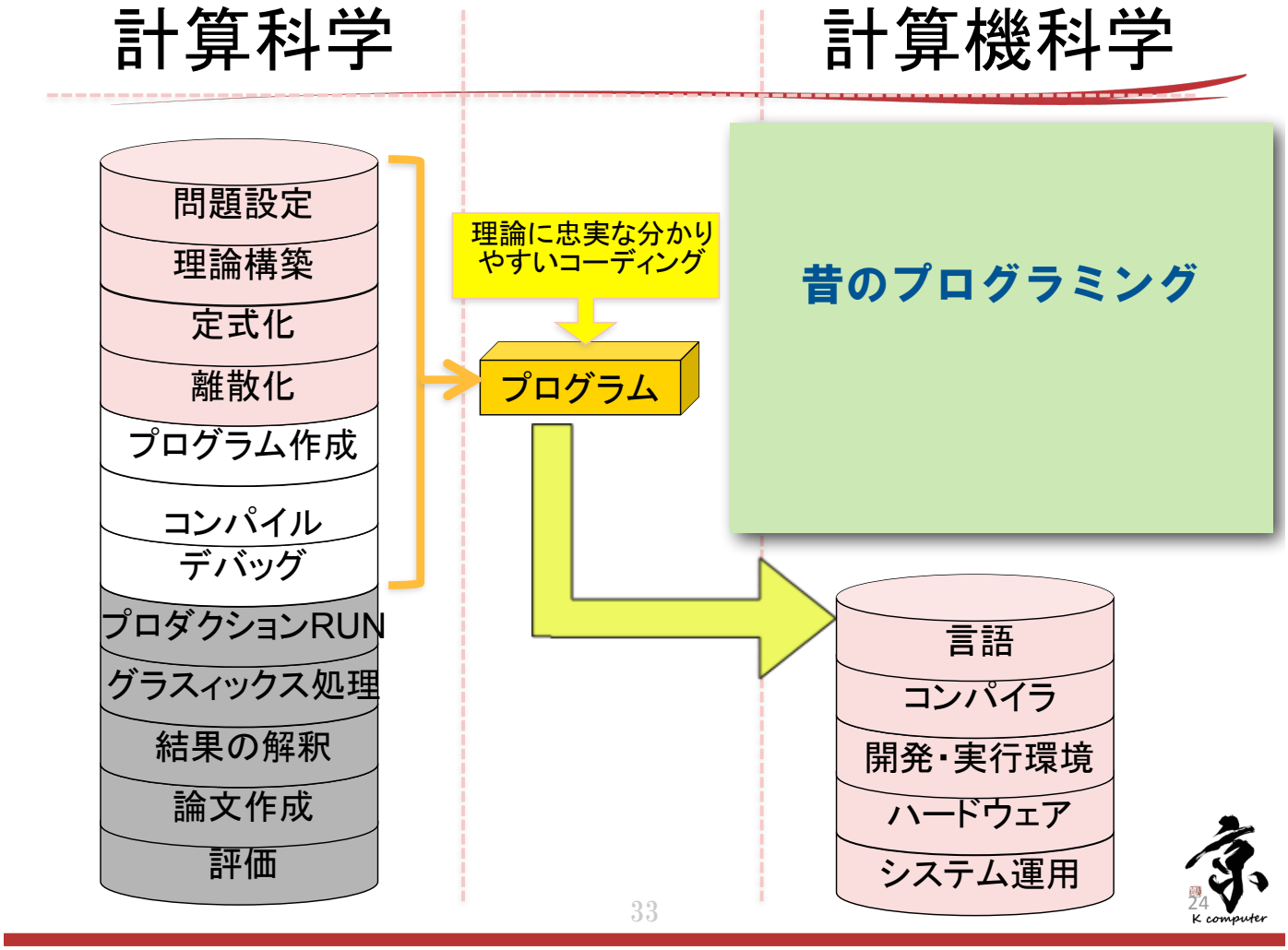
Rack

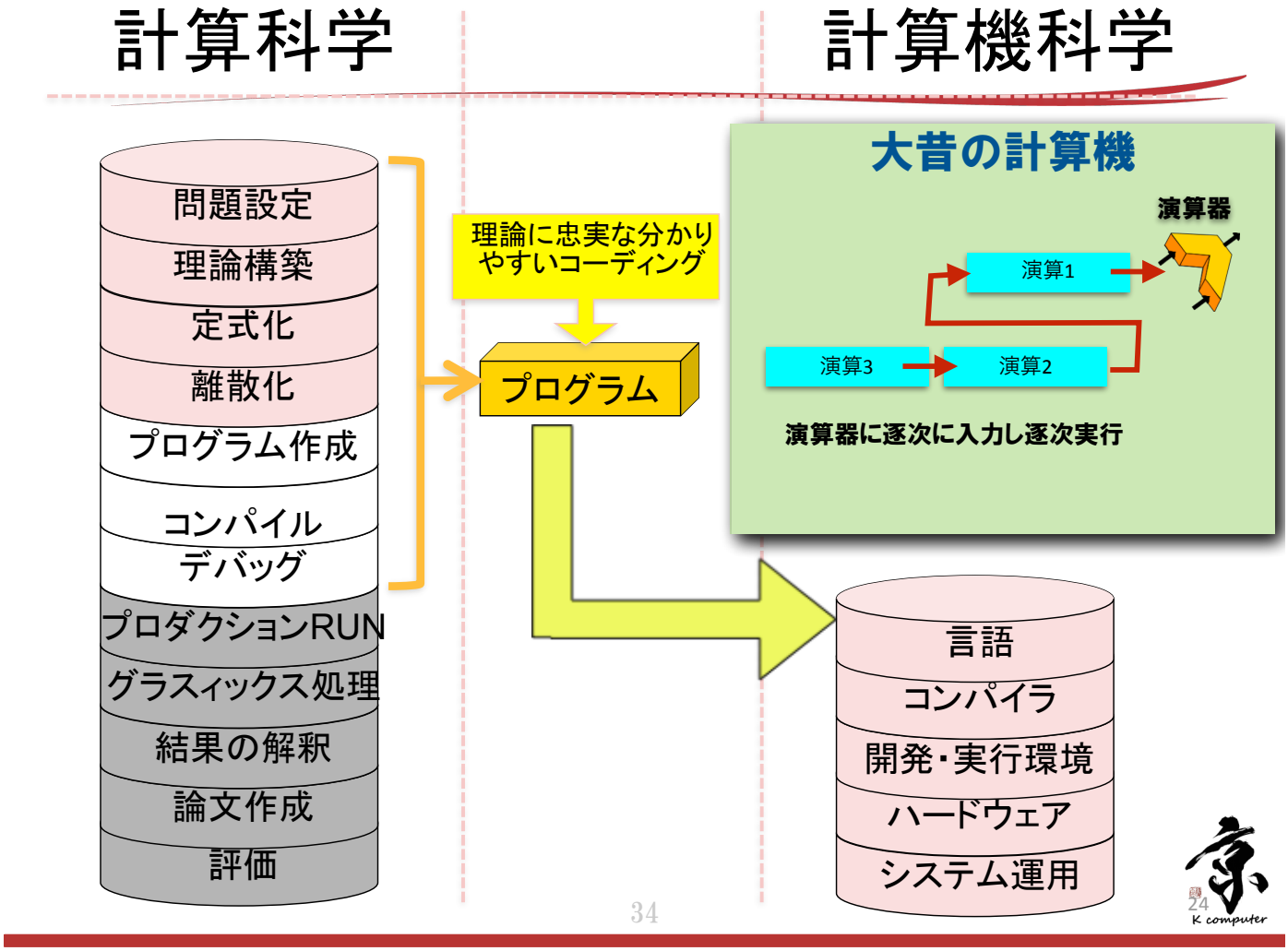


System Board






計算科学
計算機科学
問題設定
理論構築
定式化
離散化
プログラム作成
コンパイル
デバッグ
プロダクションRUN
グラスィックス処理
結果の解釈
論文作成
評価
理論に忠実な分かりやすいコーディング
プログラム
昔のプログラミング
言語
コンパイラ
開発・実行環境
ハードウェア
システム運用
33


計算科学
計算機科学
問題設定
理論構築
定式化
離散化
プログラム作成
コンパイル
デバッグ
プロダクションRUN
グラスィックス処理
結果の解釈
論文作成
評価
理論に忠実な分かりやすいコーディング
プログラム
大昔の計算機
演算器
演算1
演算3
演算2
演算器に逐次に入力し逐次実行
言語
コンパイラ
開発・実行環境
ハードウェア
システム運用
34

計算科学

計算機科学

問題設定
理論構築
定式化
離散化
プログラム作成
コンパイル
デバッグ
プロダクションRUN
グラスィックス処理
結果の解釈
論文作成
評価

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

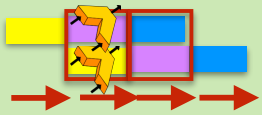
(1) そのまま実行すると6clock

演算1　演算2

(2) 演算を3つのステージに分割する

演算i

(3) 並列に計算できるようスケジューリング
(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

言語
コンパイラ
開発・実行環境
ハードウェア
システム運用

計算科学

計算機科学

問題設定
理論構築
定式化
離散化
プログラム作成
コンパイル
デバッグ
プロダクションRUN
グラスィックス処理
結果の解釈
論文作成
評価

理論に忠実な分かりやすいコーディング

プログラム

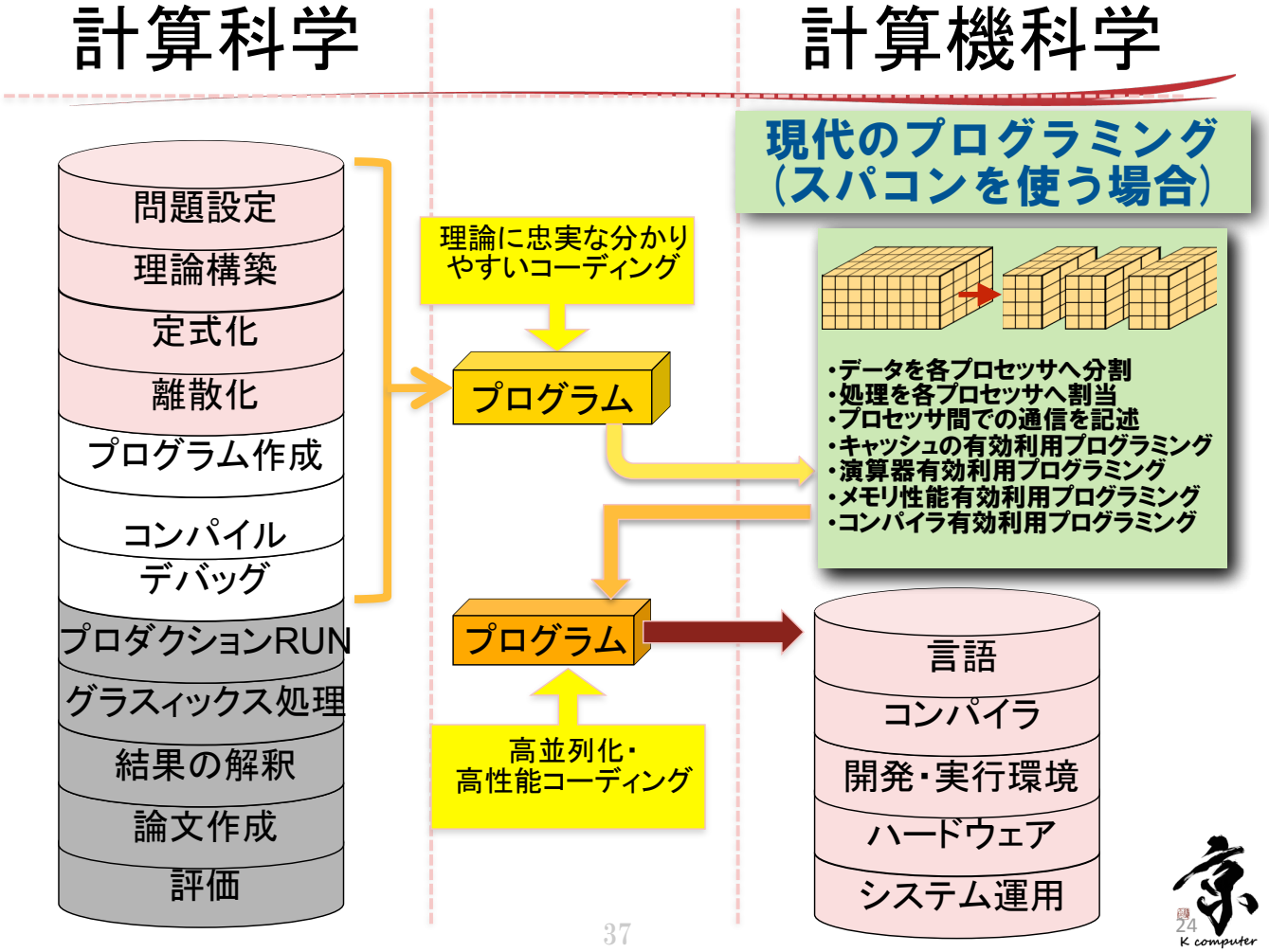
現代のプログラミング（スパコンを使う場合）

- 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

プログラム

高並列化・高性能コーディング

言語
コンパイラ
開発・実行環境
ハードウェア
システム運用

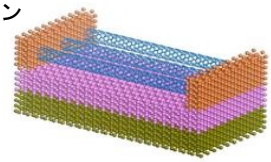


アプリケーションの性能についてまとめると

- 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

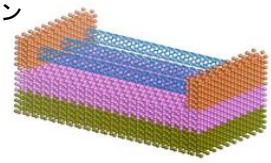
例えば次世代のデバイス
全体のシミュレーション



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

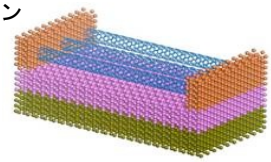
例えば次世代のデバイス
全体のシミュレーション



パソコンで100年かかる計算

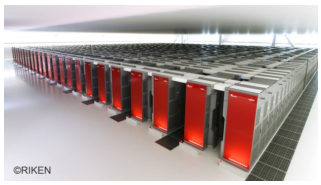
スーパーコンピュータの威力

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



K computer

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

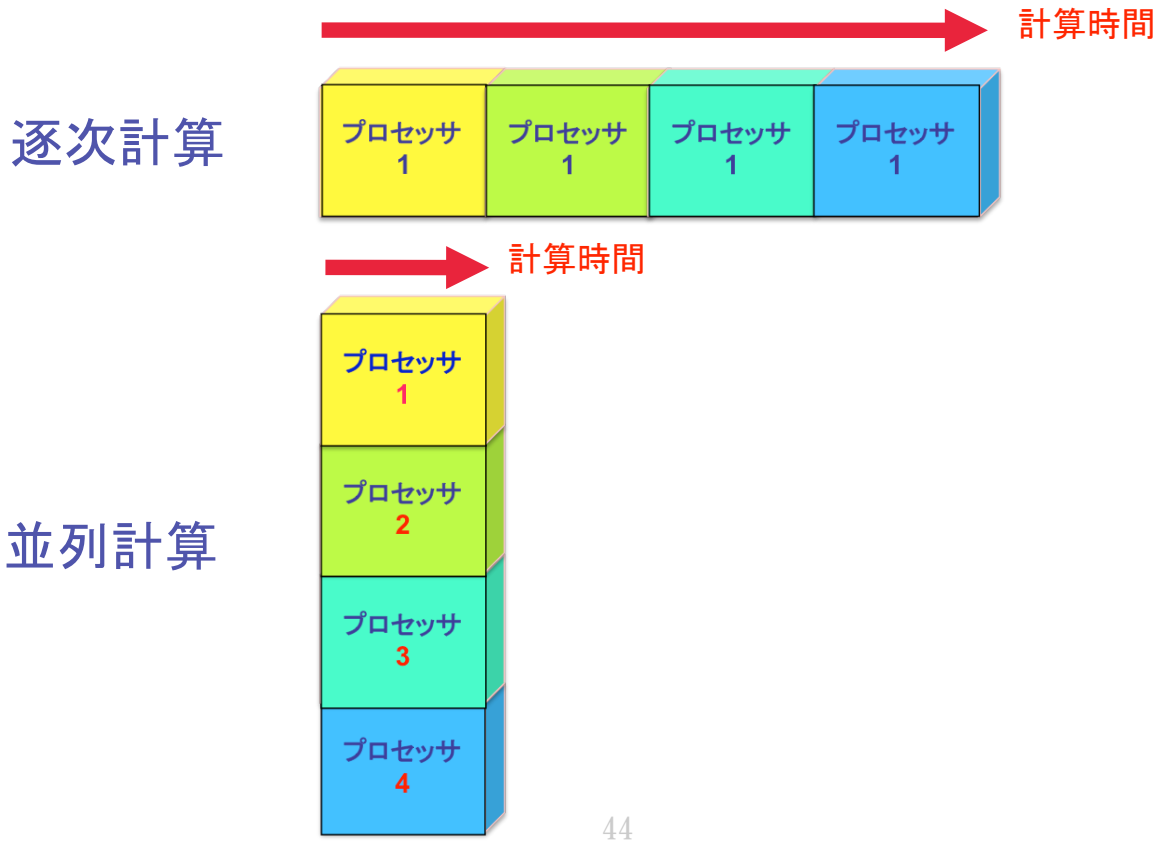
京を使って
73000倍
の速度で実行

半日で終わる

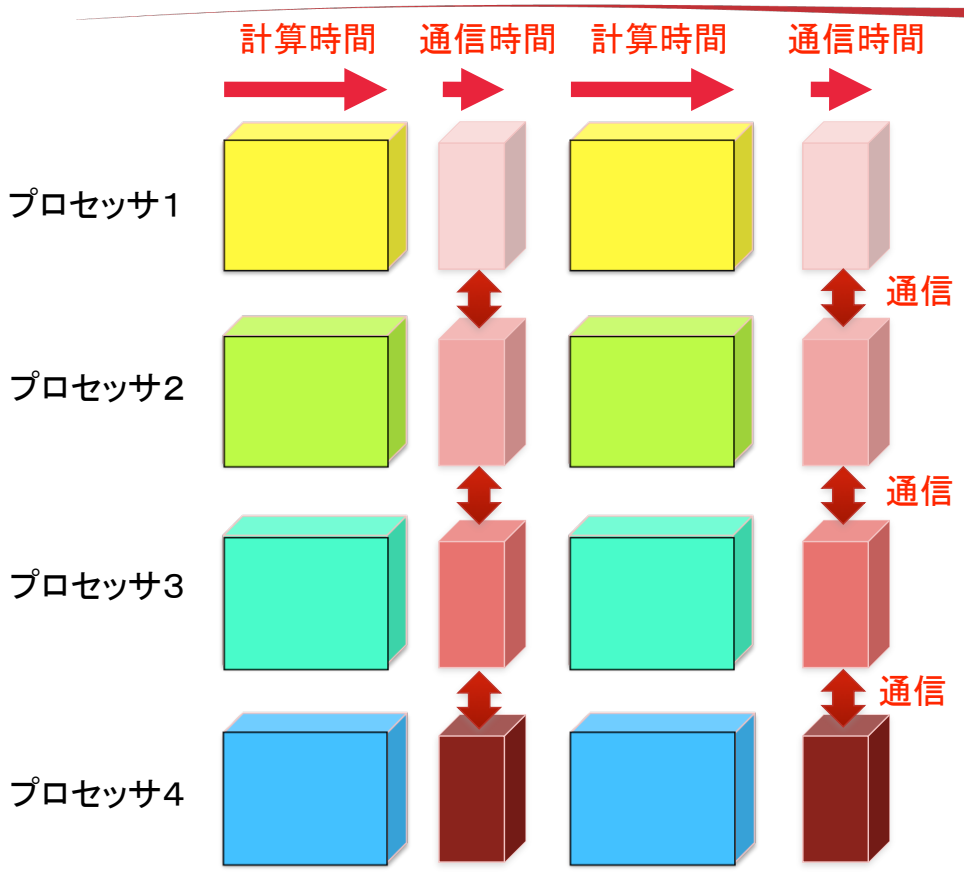
K computer

高並列化のための重要点

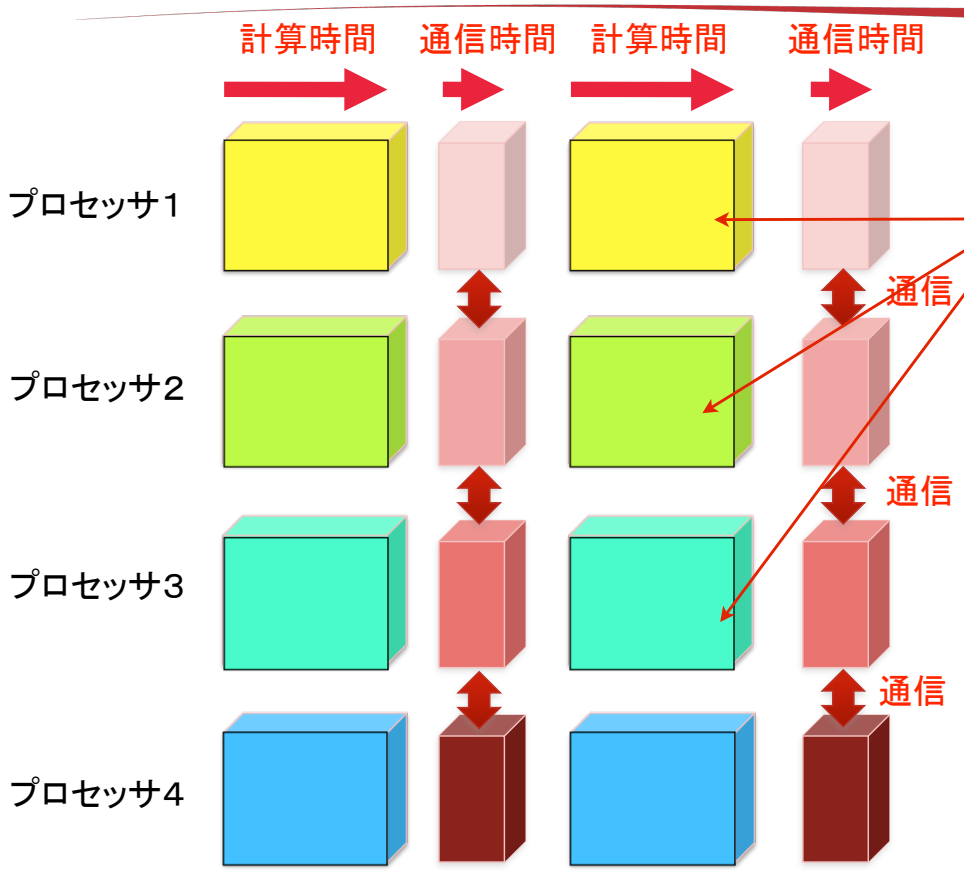
そもそも並列化とは？（1）



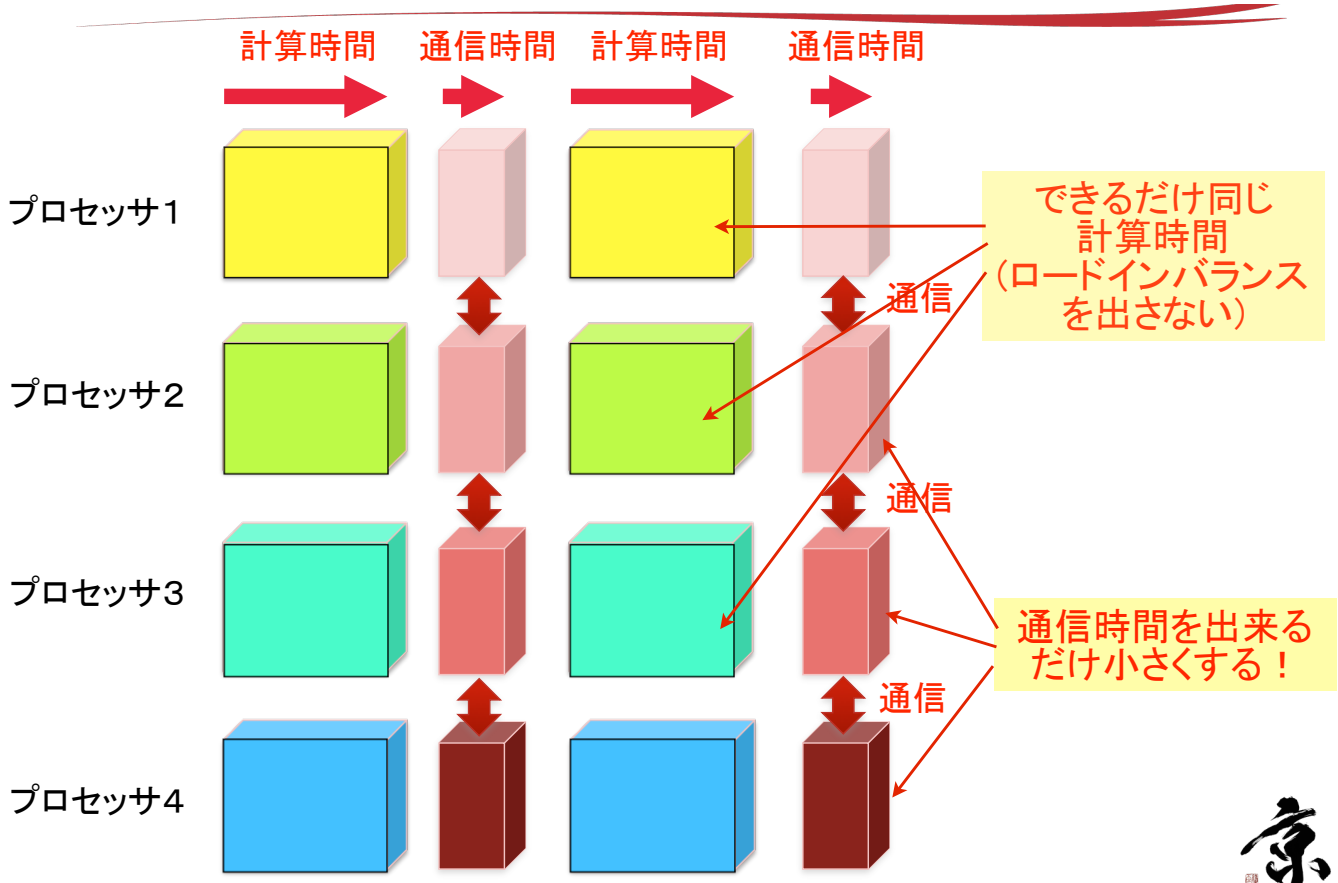
そもそも並列化とは？（2）



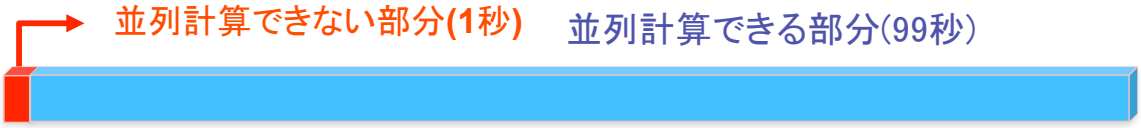
そもそも並列化とは？（2）



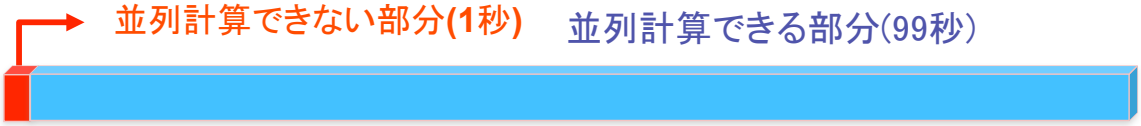
そもそも並列化とは？(2)

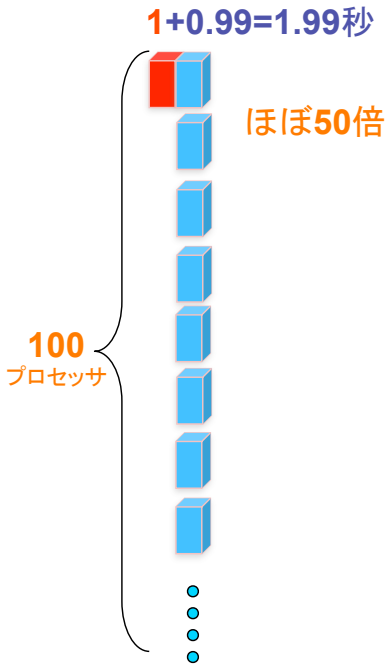


そもそも並列化とは？(3)

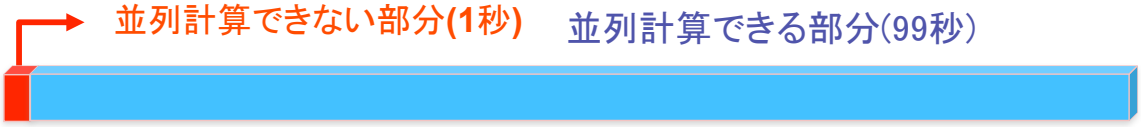


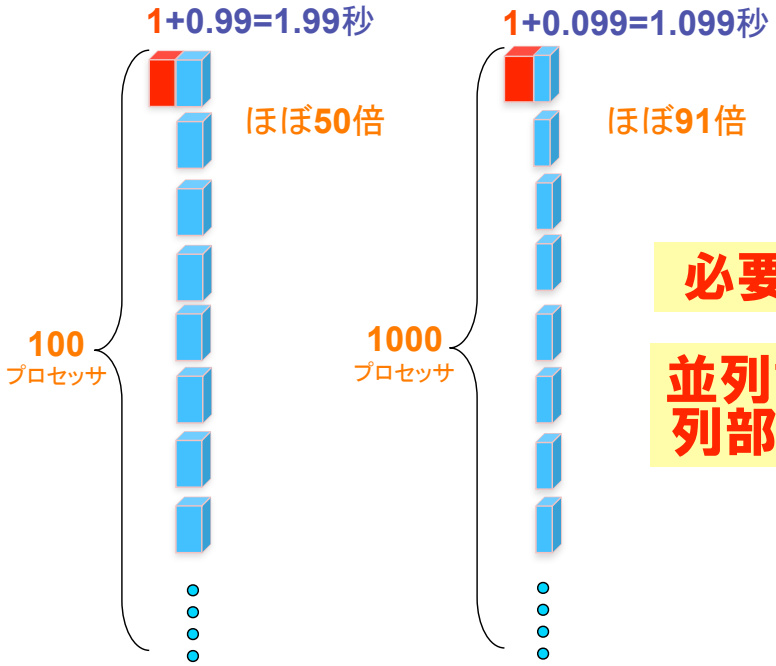
そもそも並列化とは？（3）





そもそも並列化とは？（3）



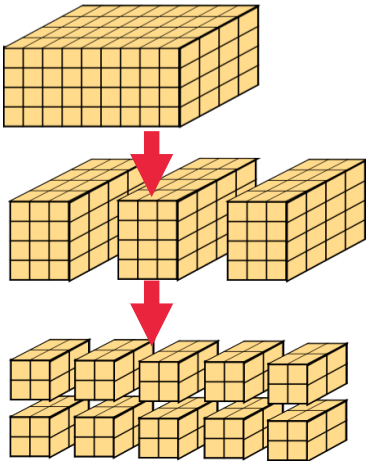


必要な並列度を確保する！

並列計算できない部分（非並列部）を限りなく小さくする！

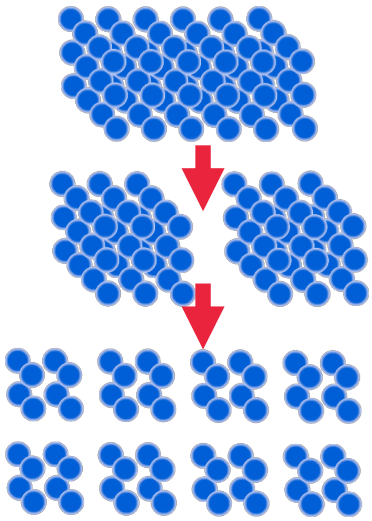
必要な並列度を確保するとは？

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

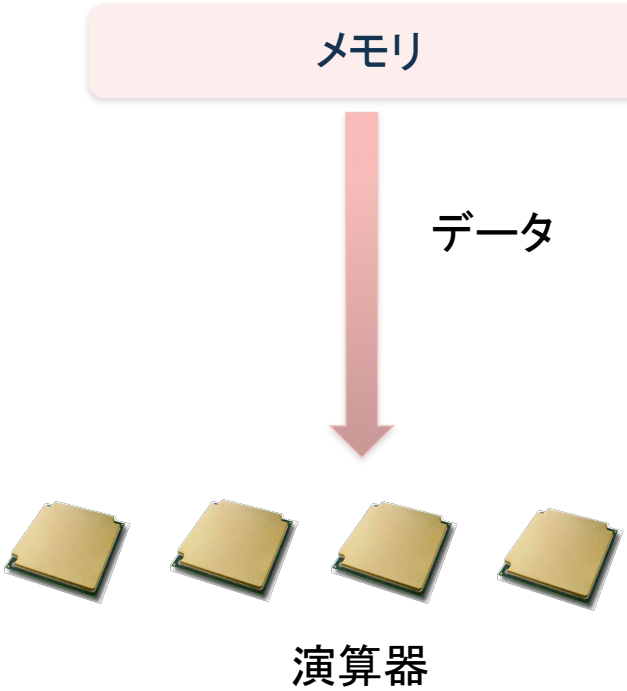
- 領域分割の場合，完全に領域分割されていれば非並列部が発生する場合は少ない．
- 領域分割されていな配列や処理が主要な計算部に残る場合あり．
- また初期処理に分割されていない処理が残っている場合もある．具体的には通信テーブルの作成等．
- 量子計算のアプリ等で領域分割でなくエネルギーバンド並列等を使用している場合は非並列部が残る場合がある．

```
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
```

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



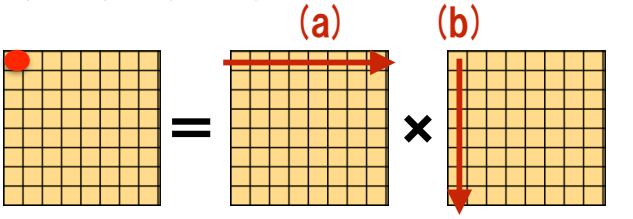
メモリウォール問題

✓ 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

B/F値が小さい計算

行列行列積の計算

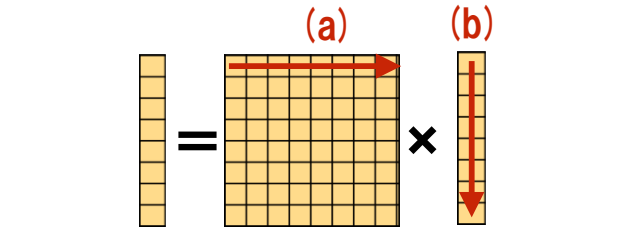


B/F値
=移動量(Byte)/演算量(Flop)
=$2N^2/2N^3$
=1/N

原理的にはNが大きい程小さな値

B/F値が大きい計算

行列ベクトル積の計算

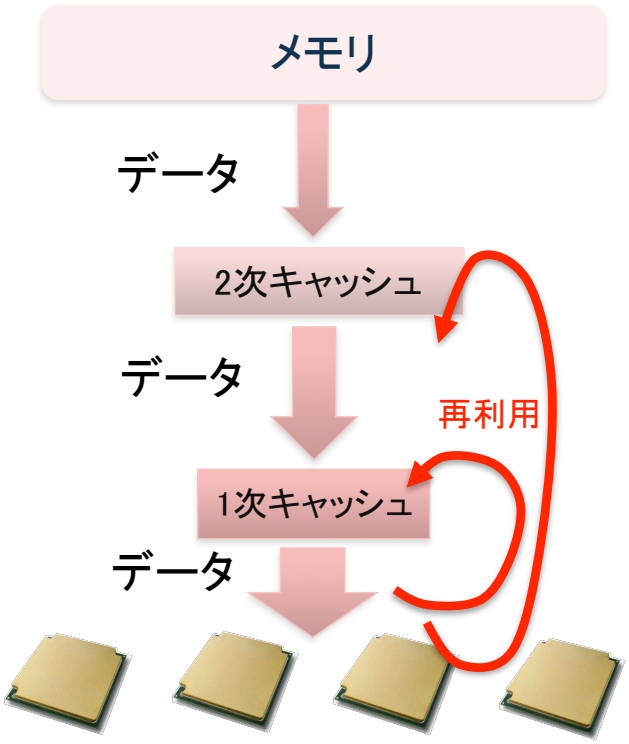


B/F値
=移動量(Byte)/演算量(Flop)
=$(N^2+N)/2N^2$
≒1/2

原理的には1/Nより大きな値

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)

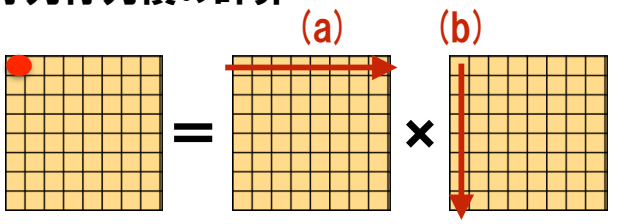
例えば行列行列積
(2N²個のデータで2N³個の演算可)

(*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

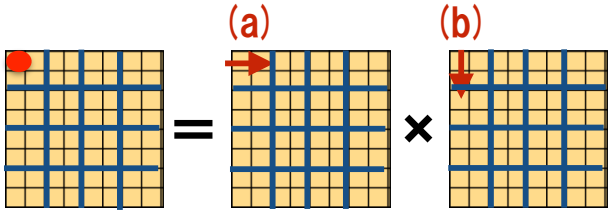
プロセッサの単体性能を引き出す (2)

行列行列積の計算



B/F値
=移動量(Byte)/演算量(Flop)
=2N²/2N³
=1/N
原理的にはNが大きい程小さな値

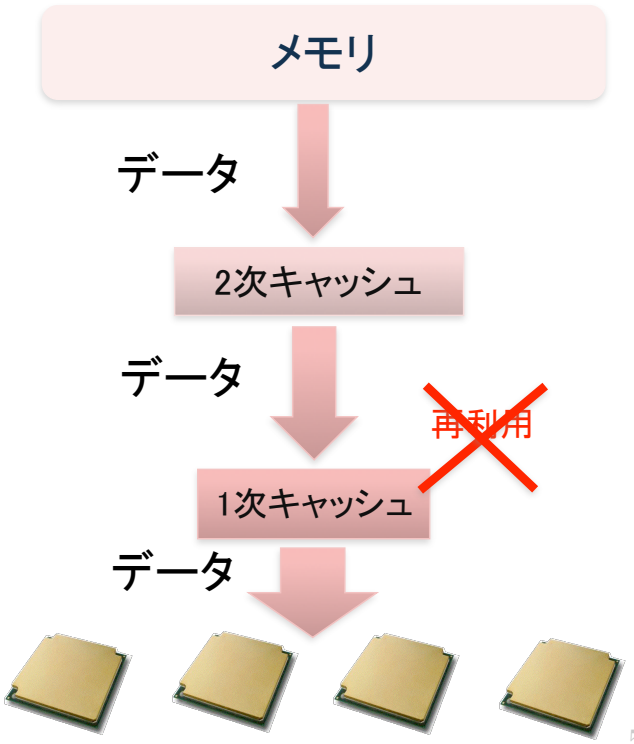
- **現実のアプリケーションではNがある程度の大きさになるとメモリ配置的には (a) はキャッシュに乗っても (b) は乗らない事となる**



- **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ッ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.**

プロセッサの単体性能を引き出す（3）

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

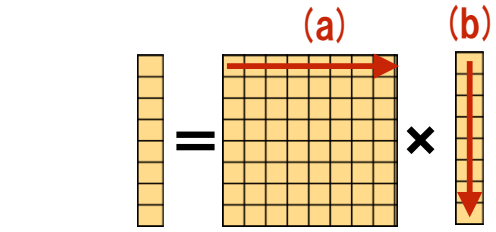
「アプリケーションが要求するByte/Flop値が高い」タイプの計算(*2)

(*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

プロセッサの単体性能を引き出す（3）

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

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．**
- **一度は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．**

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

p7-p10,p17-p21については元RIKEN AICS井上統括役の資料である.

K computer