CMSI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

2014年4月10日

独立行政法人理化学研究所
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

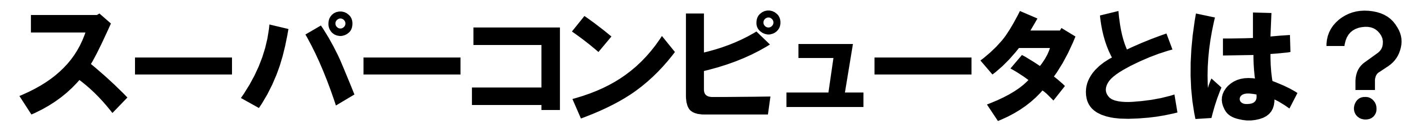
南 一生
minami_kaz@riken.jp

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 1.5 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1.5兆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

2011年　iPhone4S
LINPACK 140Mflops

≒

40倍

2002年　当時のパソコン
PentiumIV
6.4Gflops

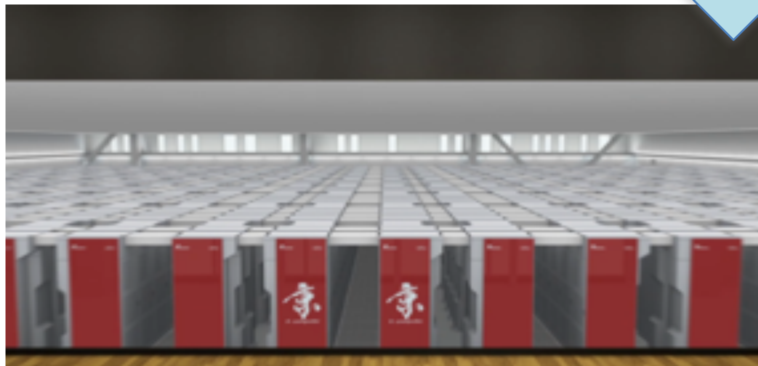
25万倍

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



10Pflop

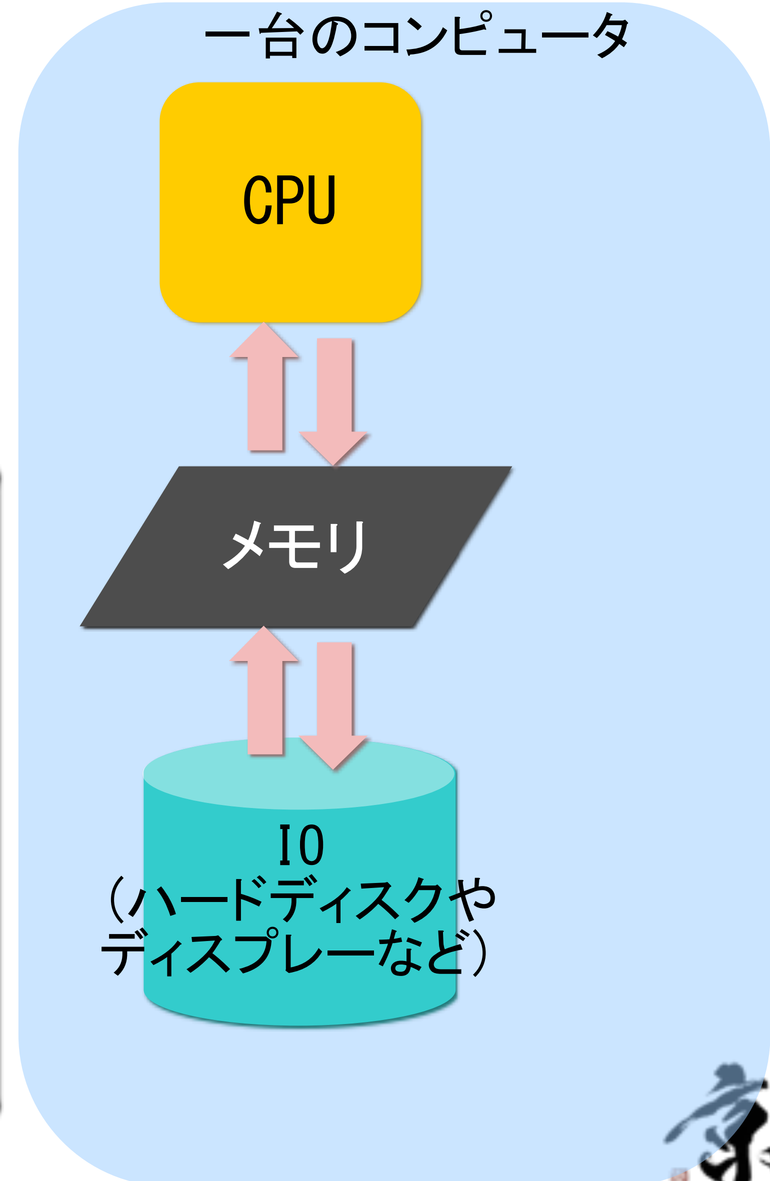
京 K computer

コンピュータとは？

ハードウェア

- CPU
- メモリ
- IO（入出力）

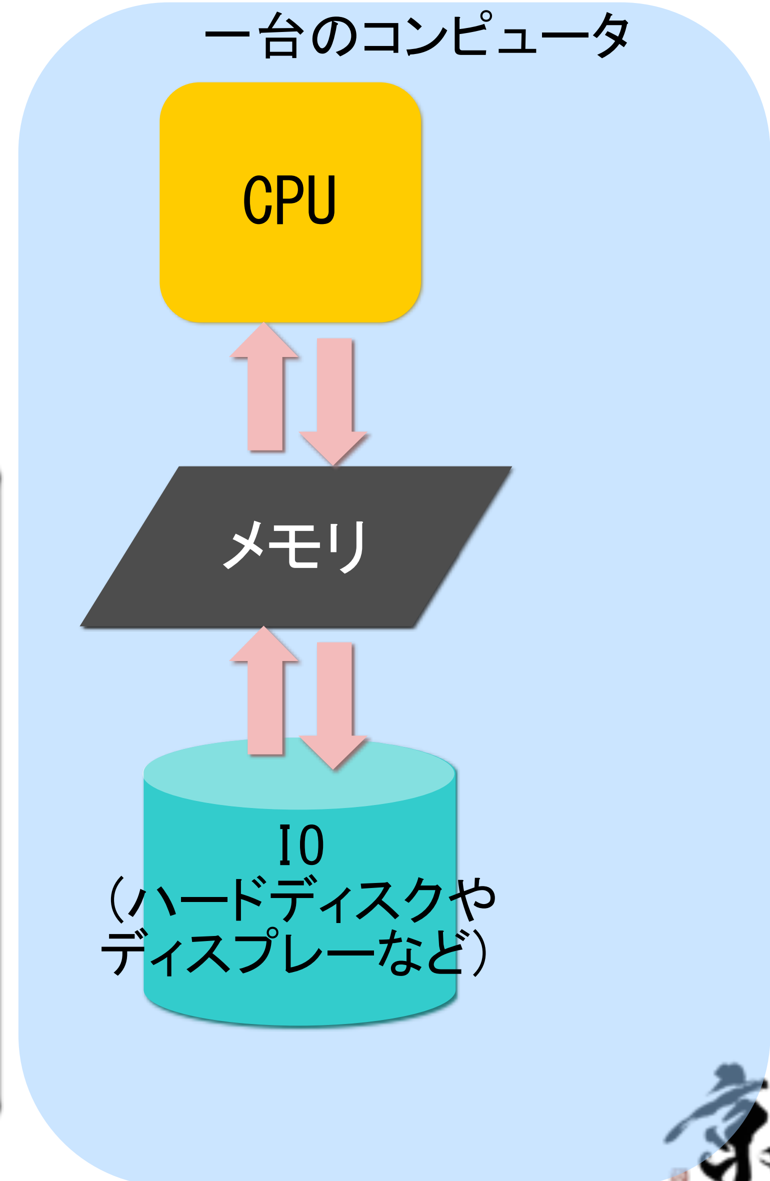
IOから受け取った（入力）データとプログラムをメモリに置き、CPUでプログラムに従ってデータの処理を行なって、メモリに書き戻し、それをIOに出す、書出す（出力）



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



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)

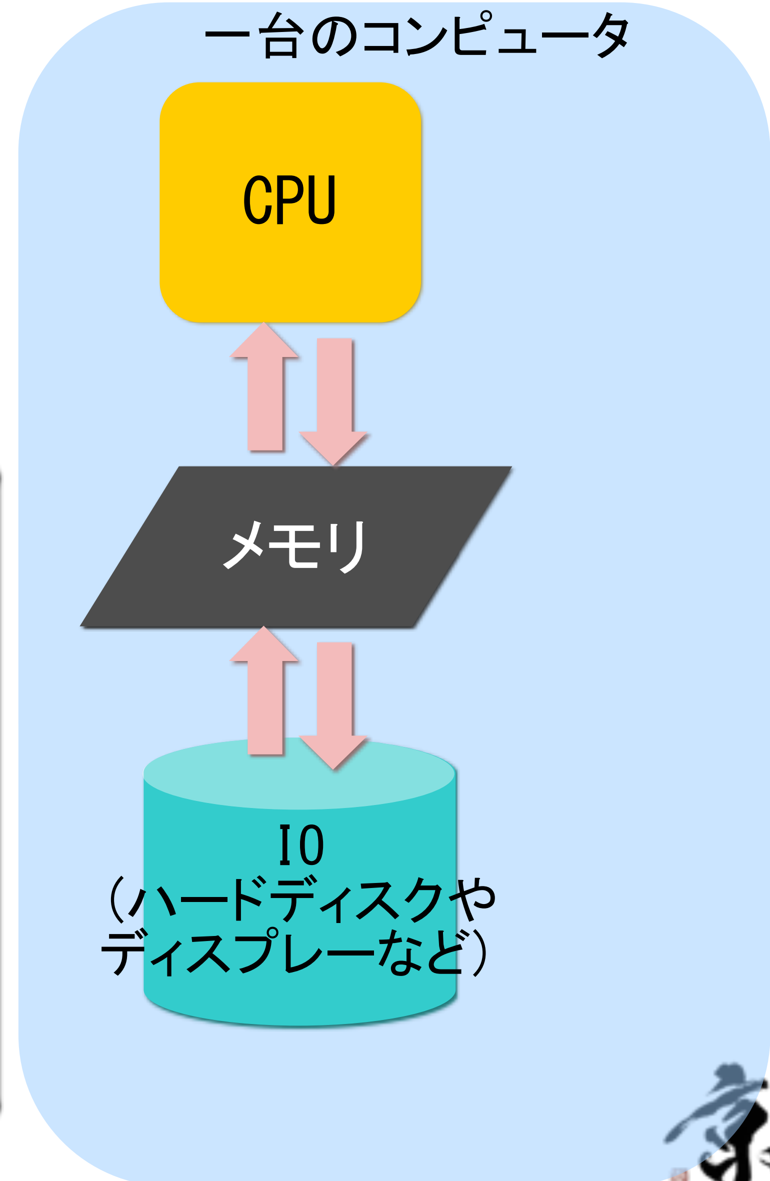
コンピュータとは？

ハードウェア
- 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

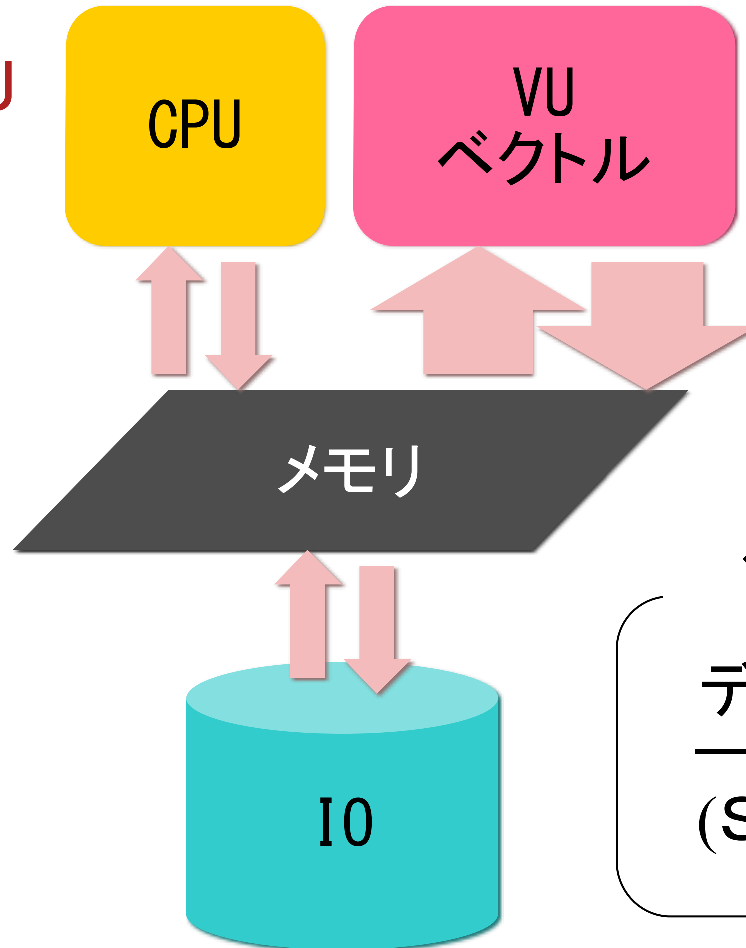
であった時代



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

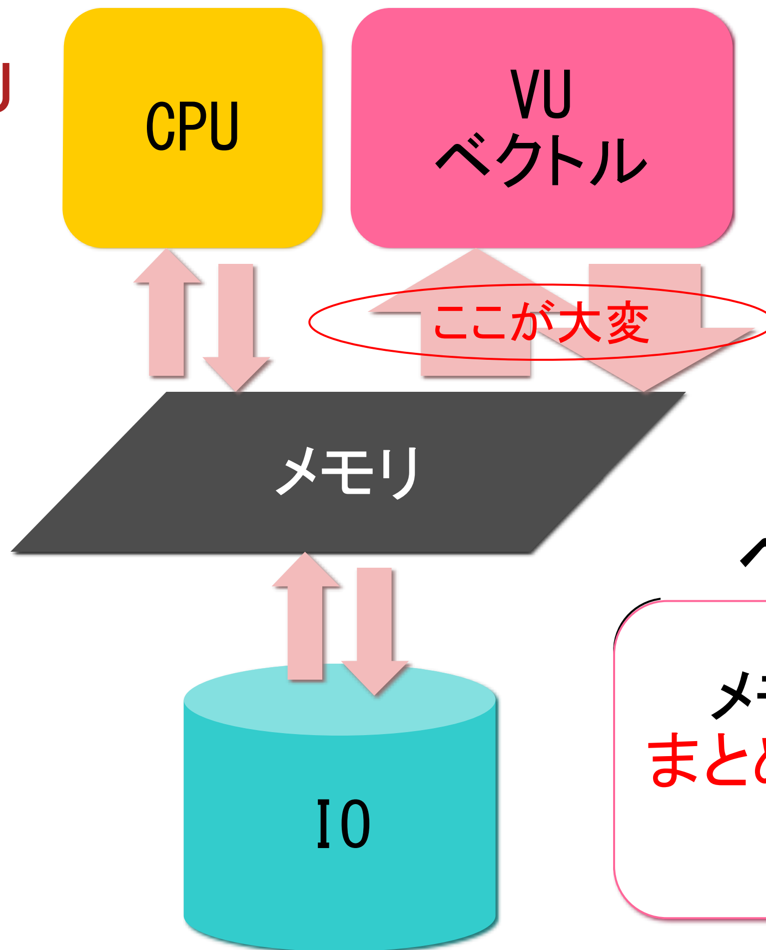
ベクトル機構

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

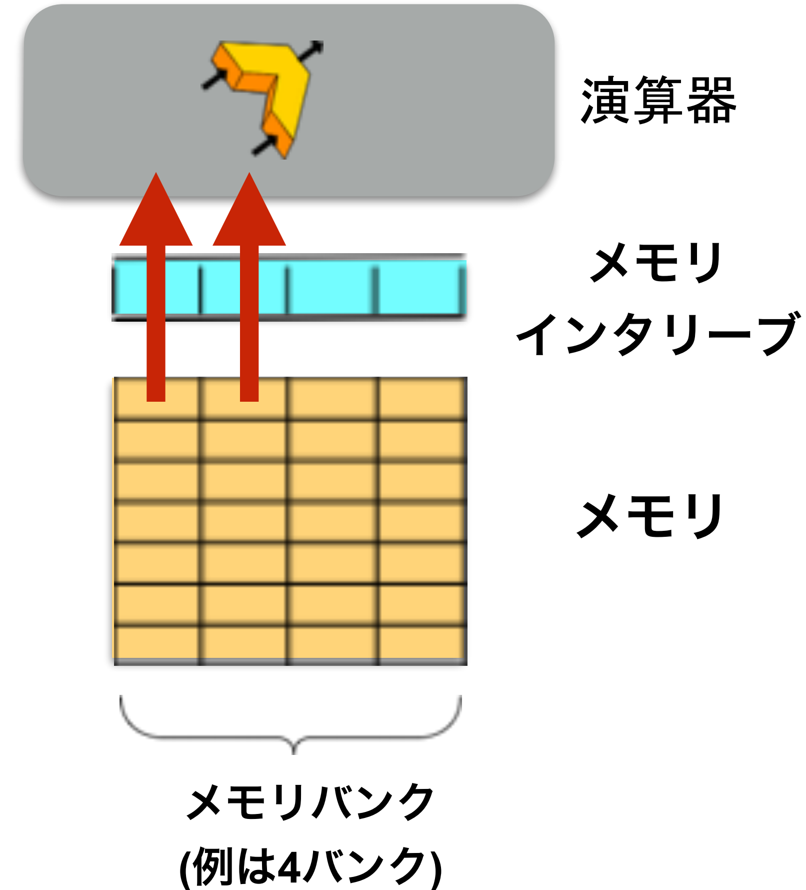
今もこの方式が無くなったわけではありませんが. . .

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

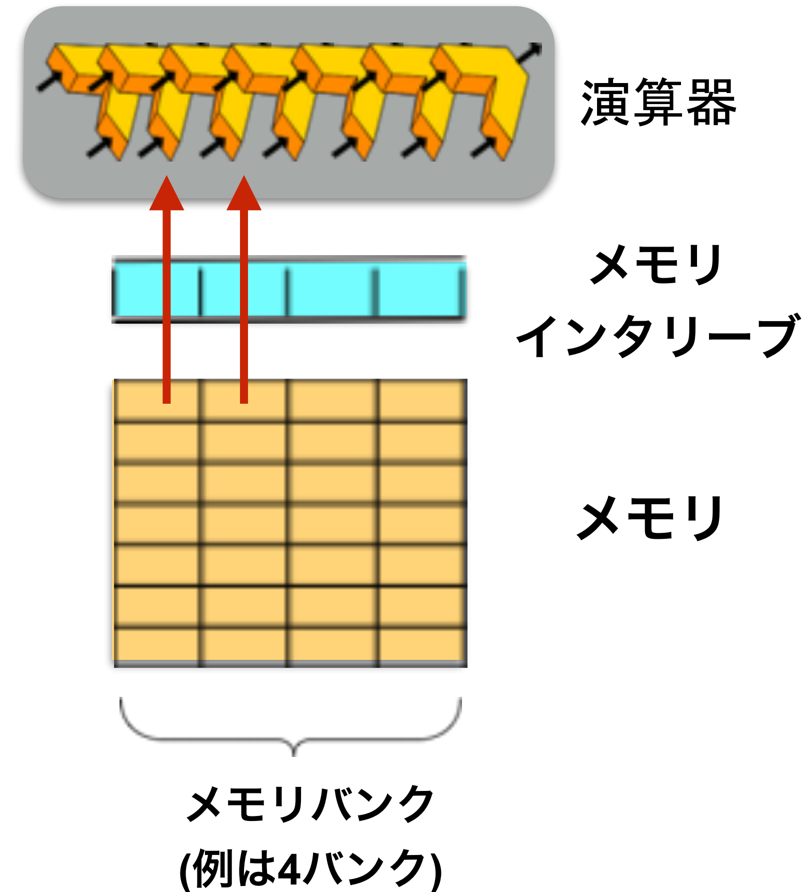
動作周波数が低い場合(昔)



シングルプロセッサの問題

メモリウォール問題

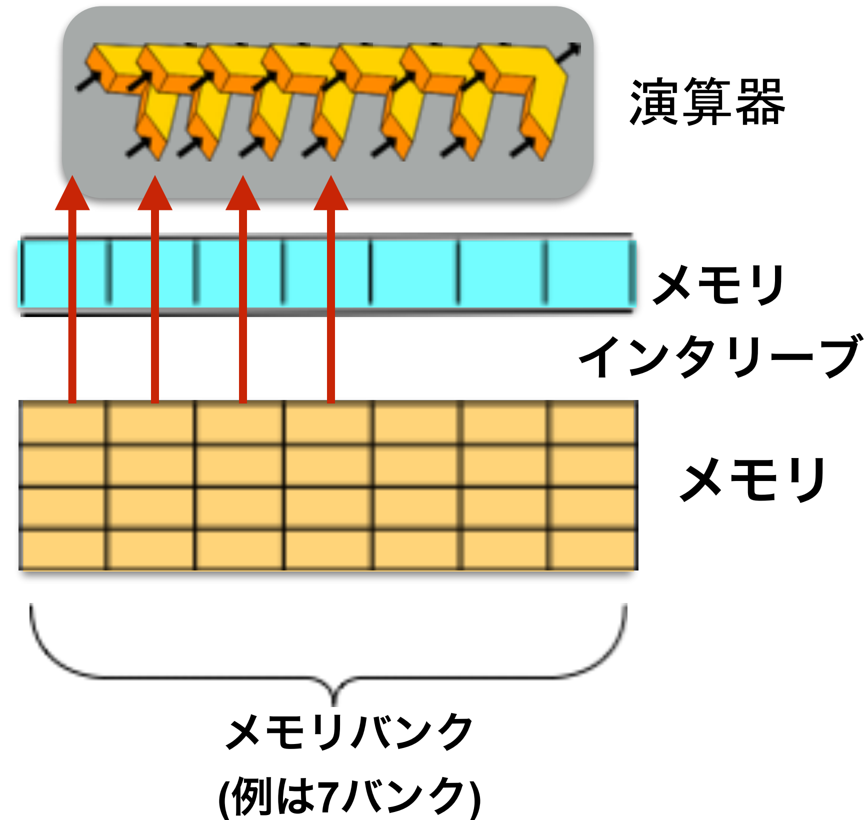
- メモリは動作周波数が高くなってくるともっと待つ事となる(100–200サイクル)
- メモリに比べて演算器は動作周波数が高くなると高速になった
- さらに半導体プロセスの微細化により演算器はCPUにたくさん搭載可能となった
- 結果的に演算器に比べメモリのデータ転送能力が低くなった



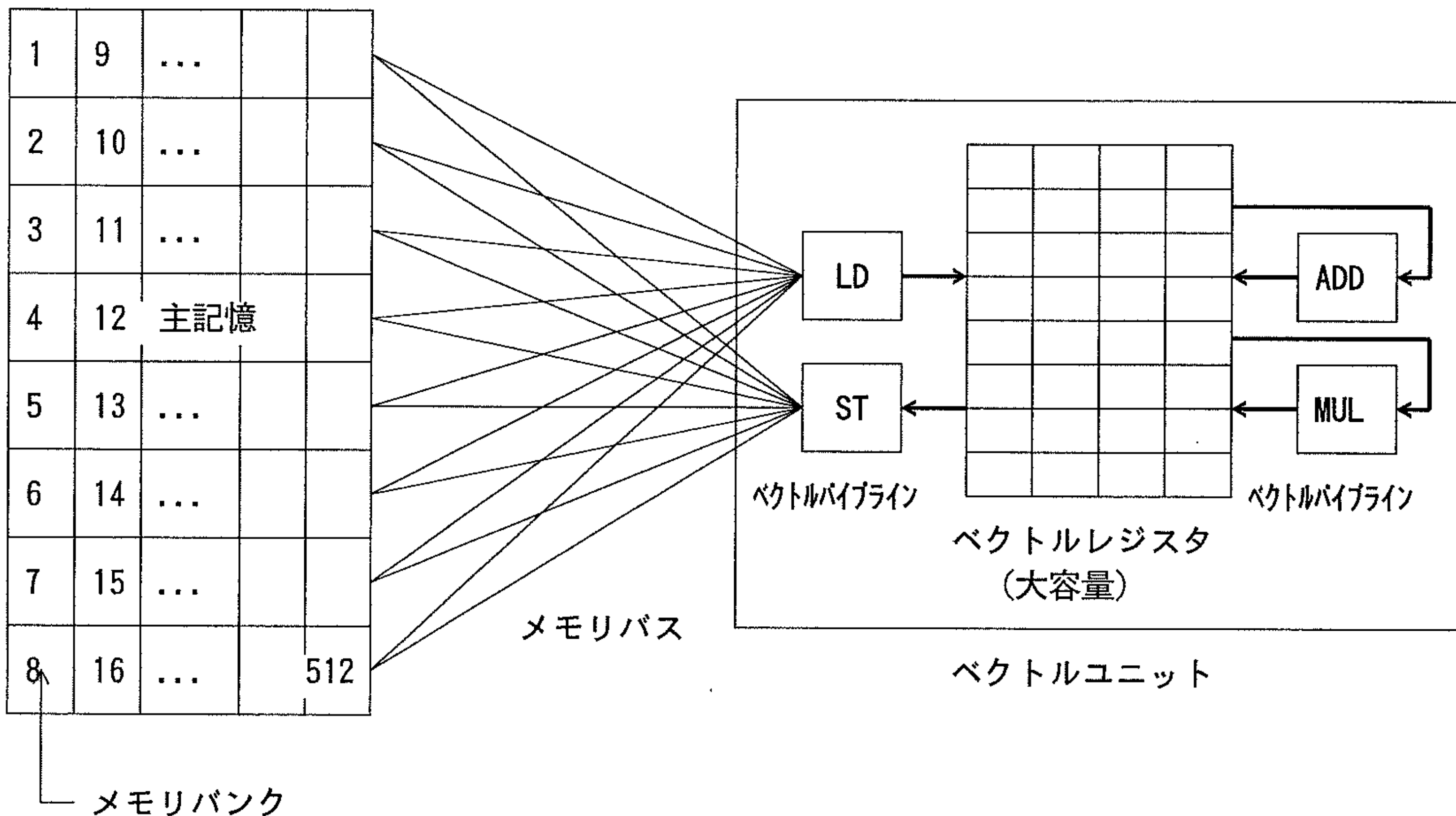
シングルプロセッサの問題

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



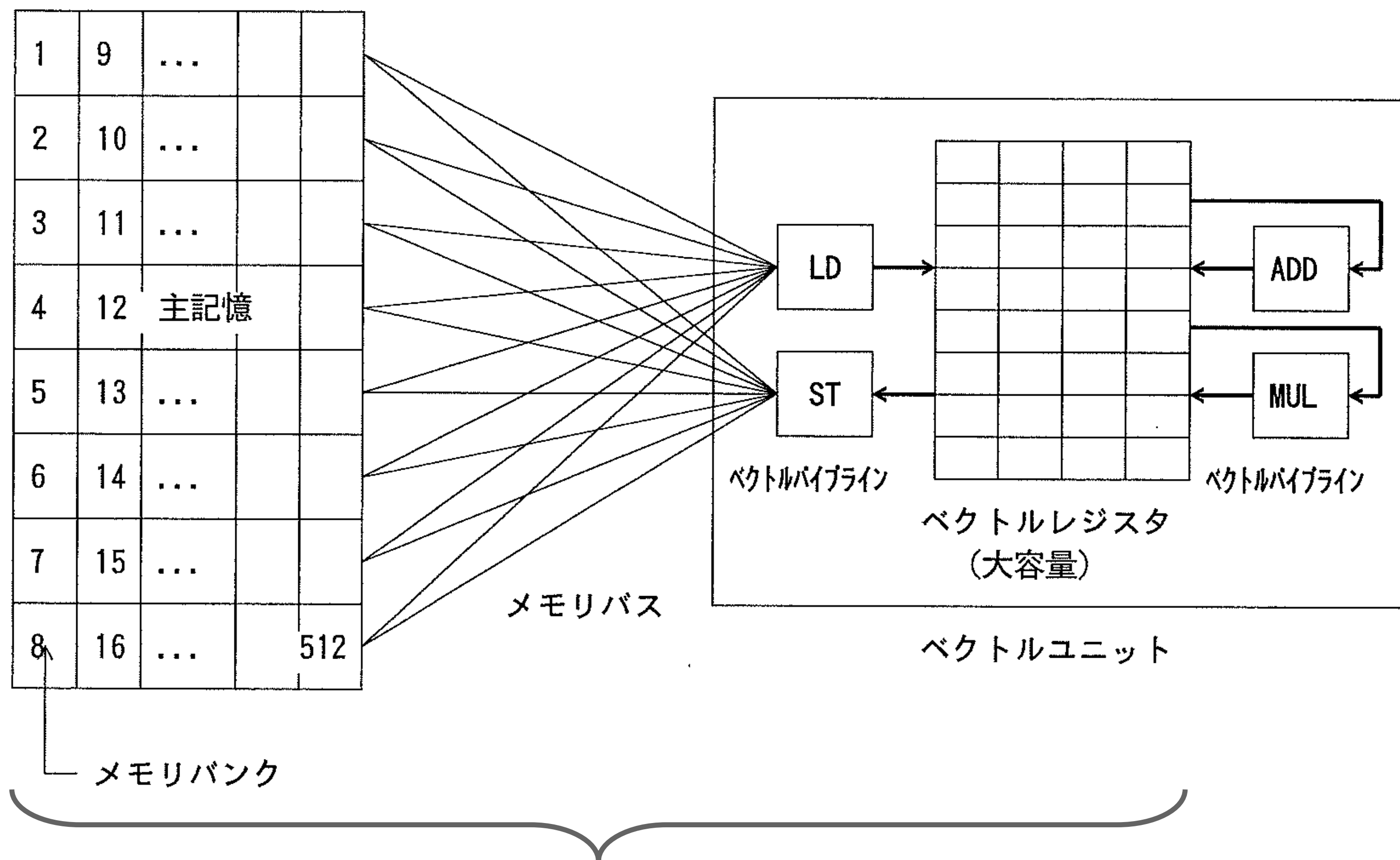
ベクトル機



特にこの部分がコスト・価格・電力面で高価になる

K computer

ベクトル機

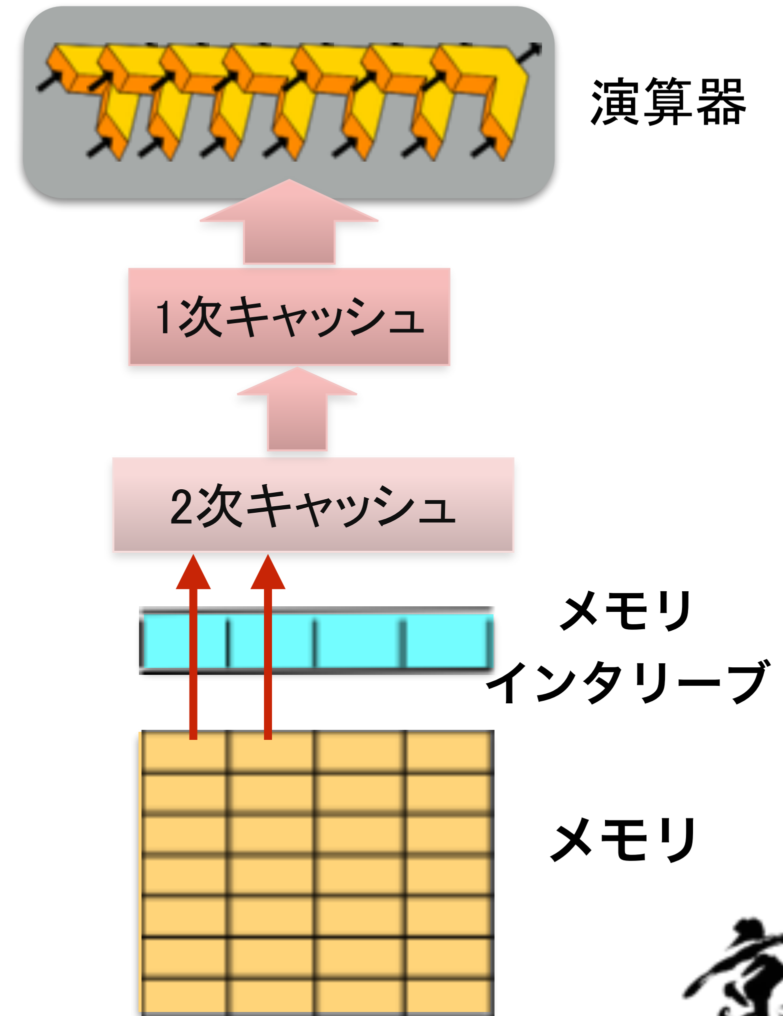


特にこの部分がコスト・価格・電力面で高価になる

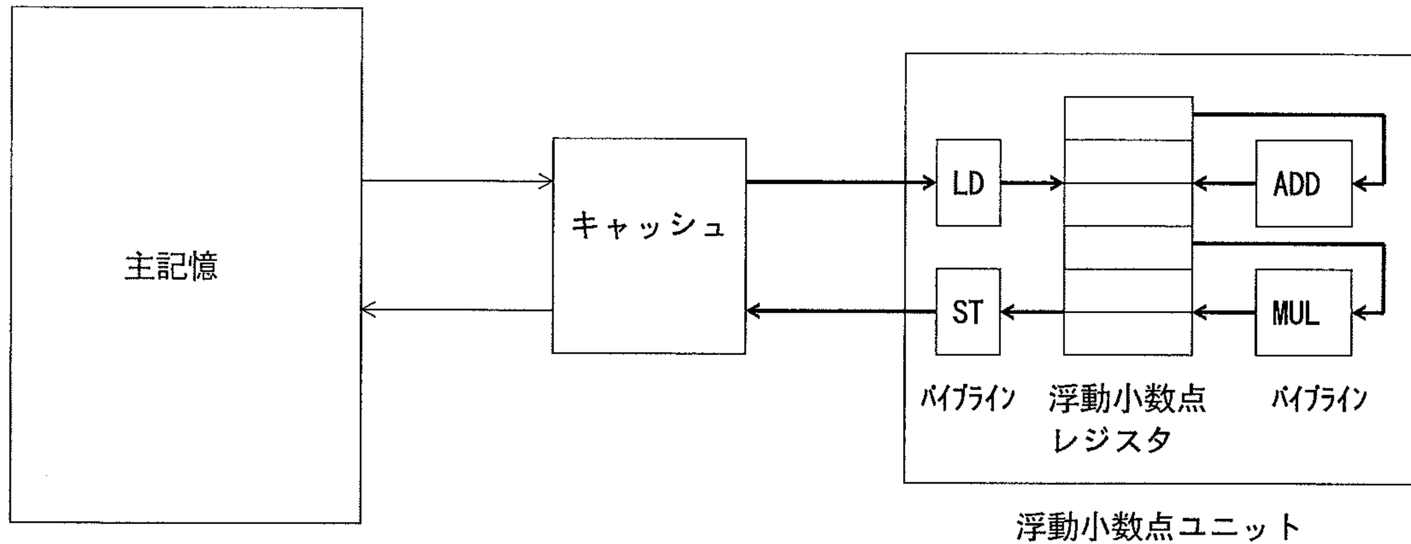
シングルプロセッサの問題

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

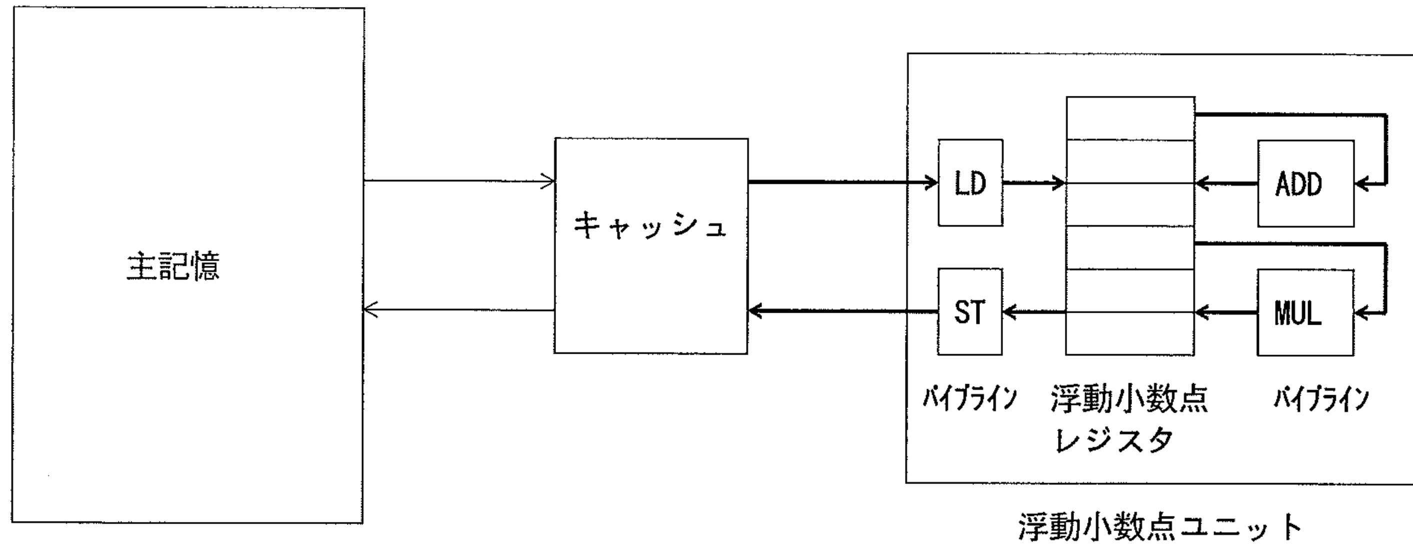


スカラー機



この部分がベクトル機に比べればシンプルになる

スカラー機



この部分がベクトル機に比べればシンプルになる

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

K computer

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

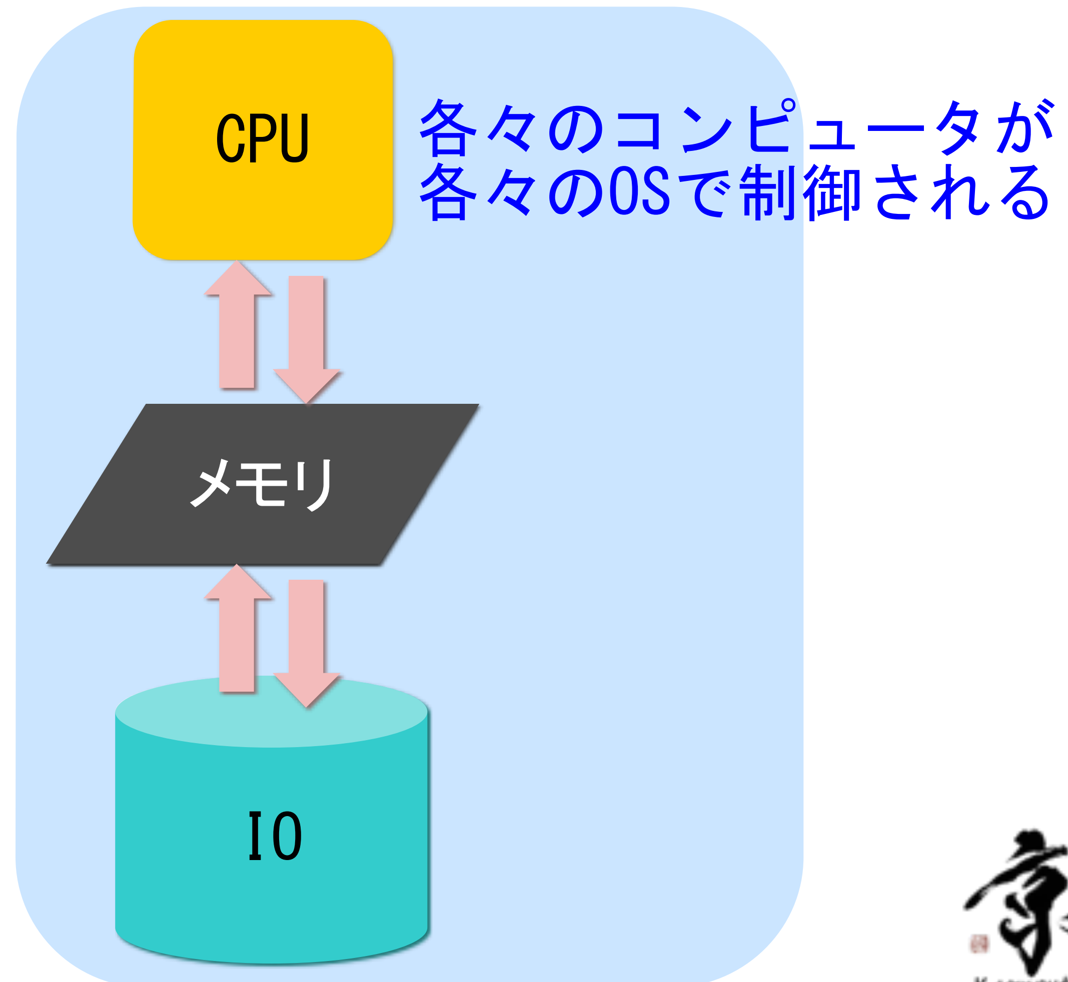
- 動作周波数を上げる事で電力が吹き上がる
- 動作周波数の限界を迎えている
- メモリウォール問題もあり一台のコンピュータの演算能力を上げてもメモリの能力が追いつかない

そこで！

K computer

最近のスーパーコンピュータ

- CPU
- メモリ
- IO(入出力)

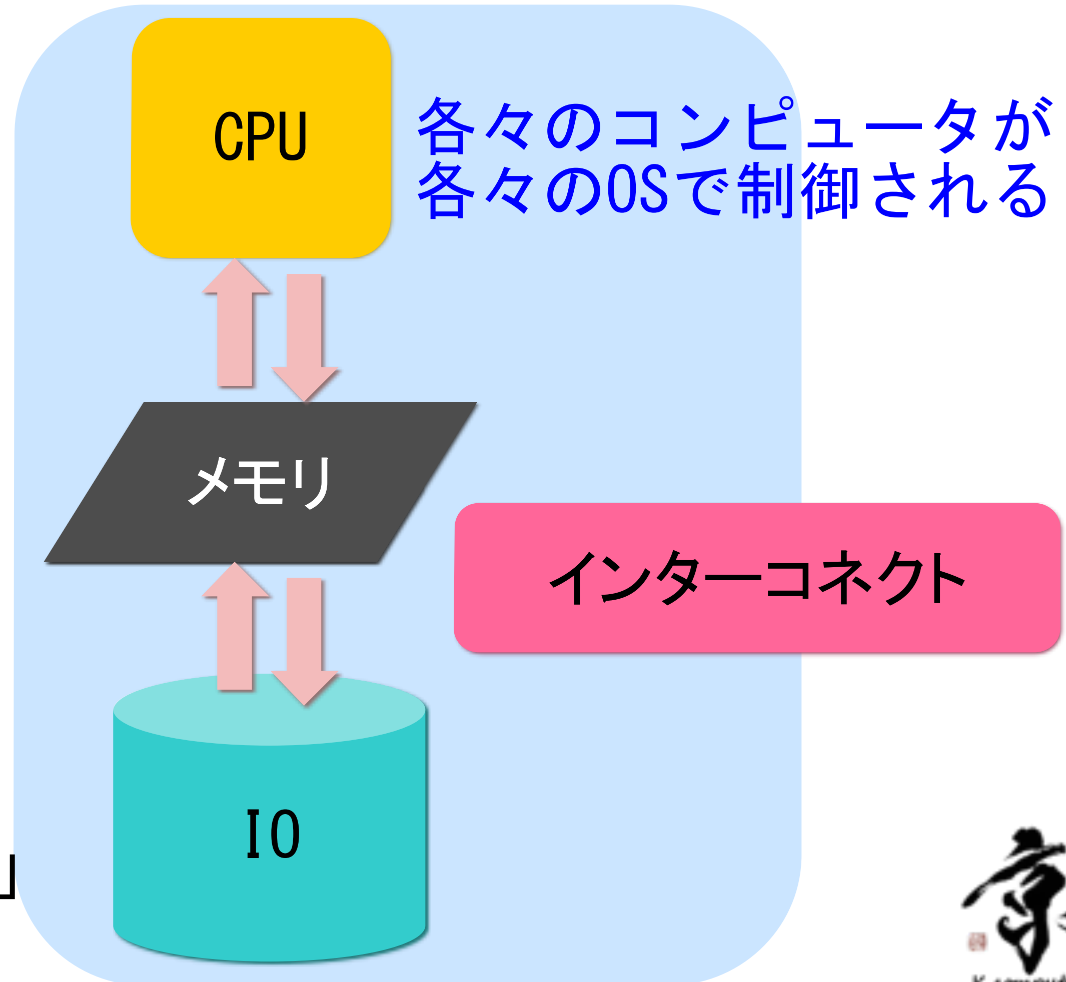


最近のスーパーコンピュータ

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)



各々のコンピュータ＝「ノード」

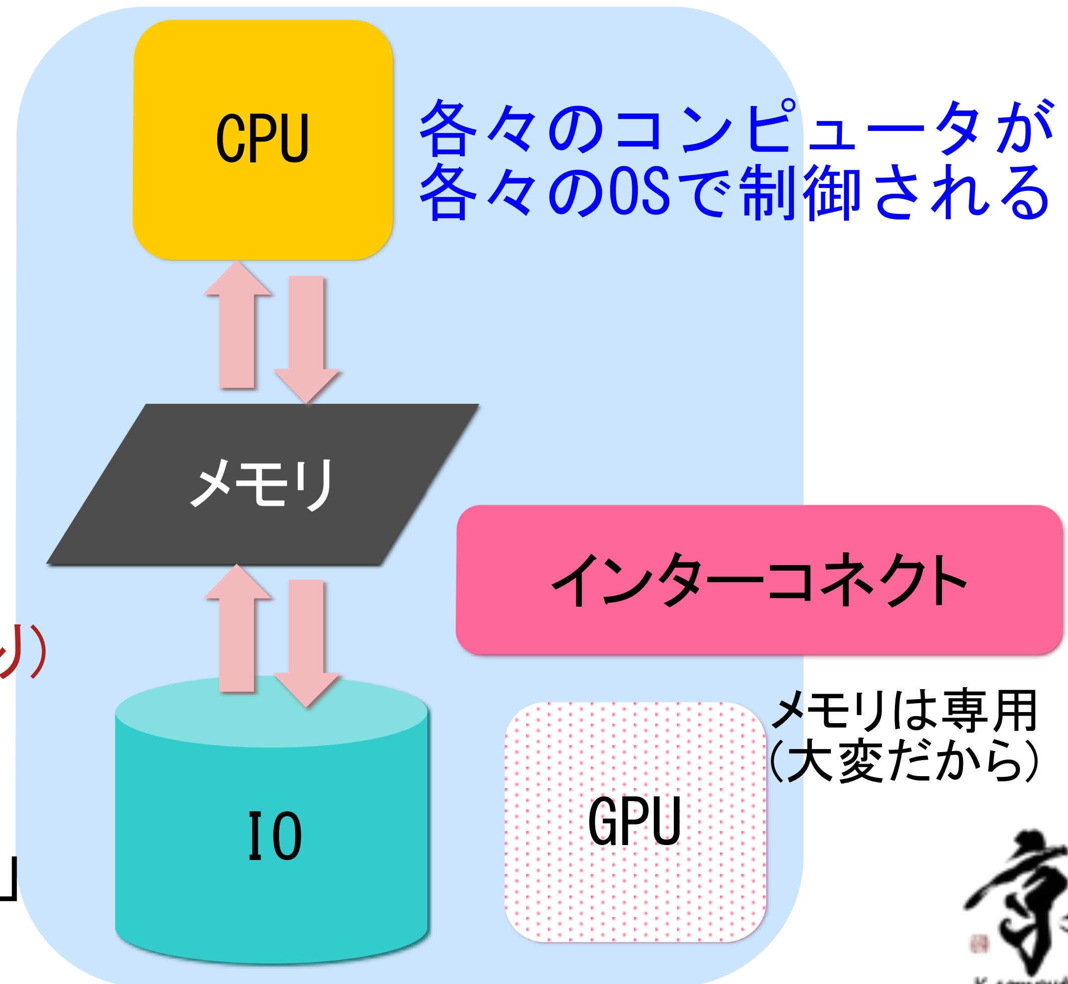
最近のスーパーコンピュータ

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

ノード

CPU

メモリ

インターコネクト

IO

K computer

コンピュータ(ノード)どうしをつなぐ

ノード

CPU

メモリ

IO

ソフトウェア
同士が連携して
一体として動く

インターコネクト

ノード

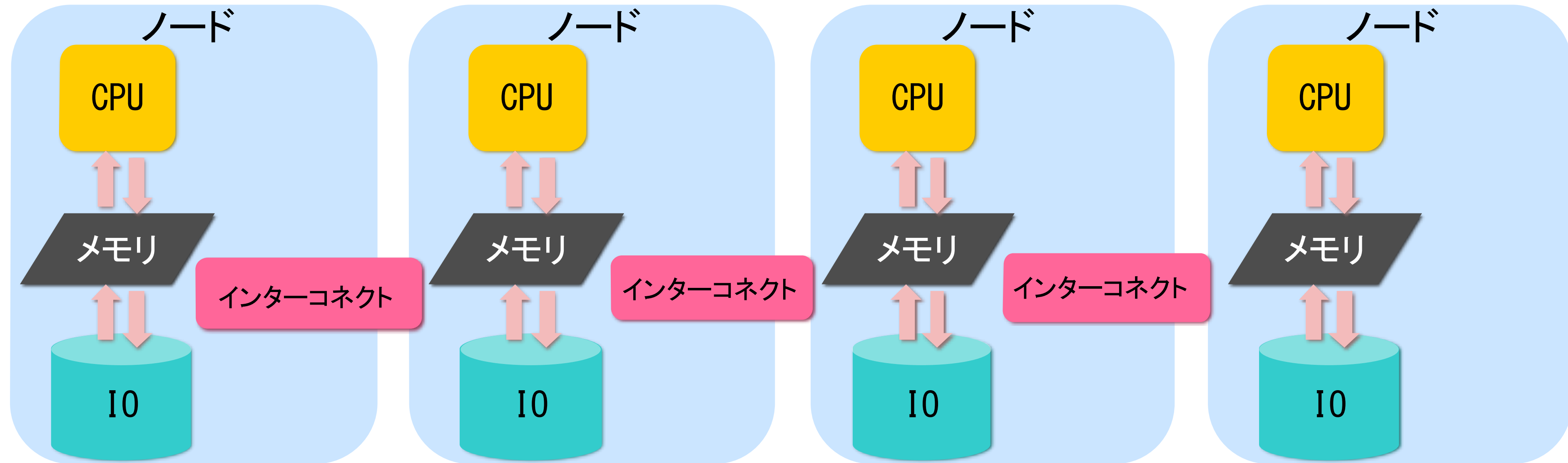
CPU

メモリ

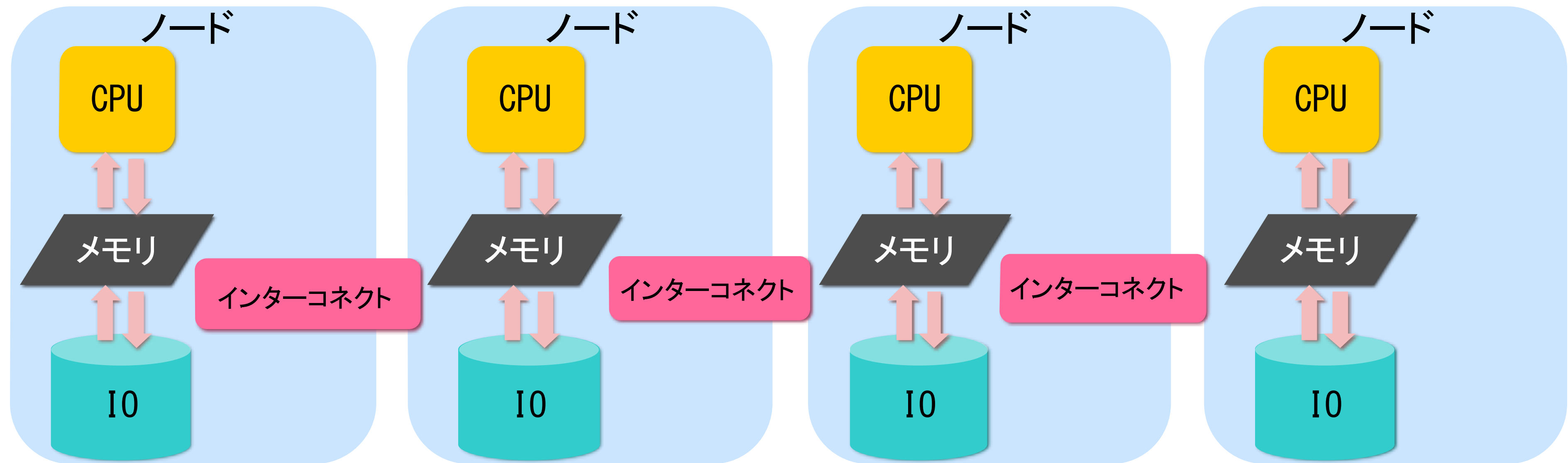
IO

K computer

コンピュータ（ノード）どうしをつなぐ



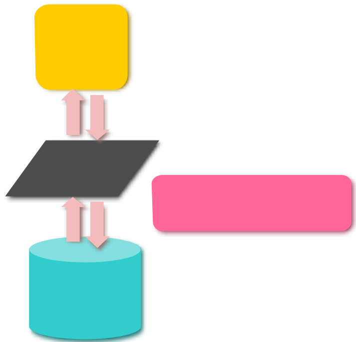
コンピュータ（ノード）どうしをつなぐ



システムとして一体で動く

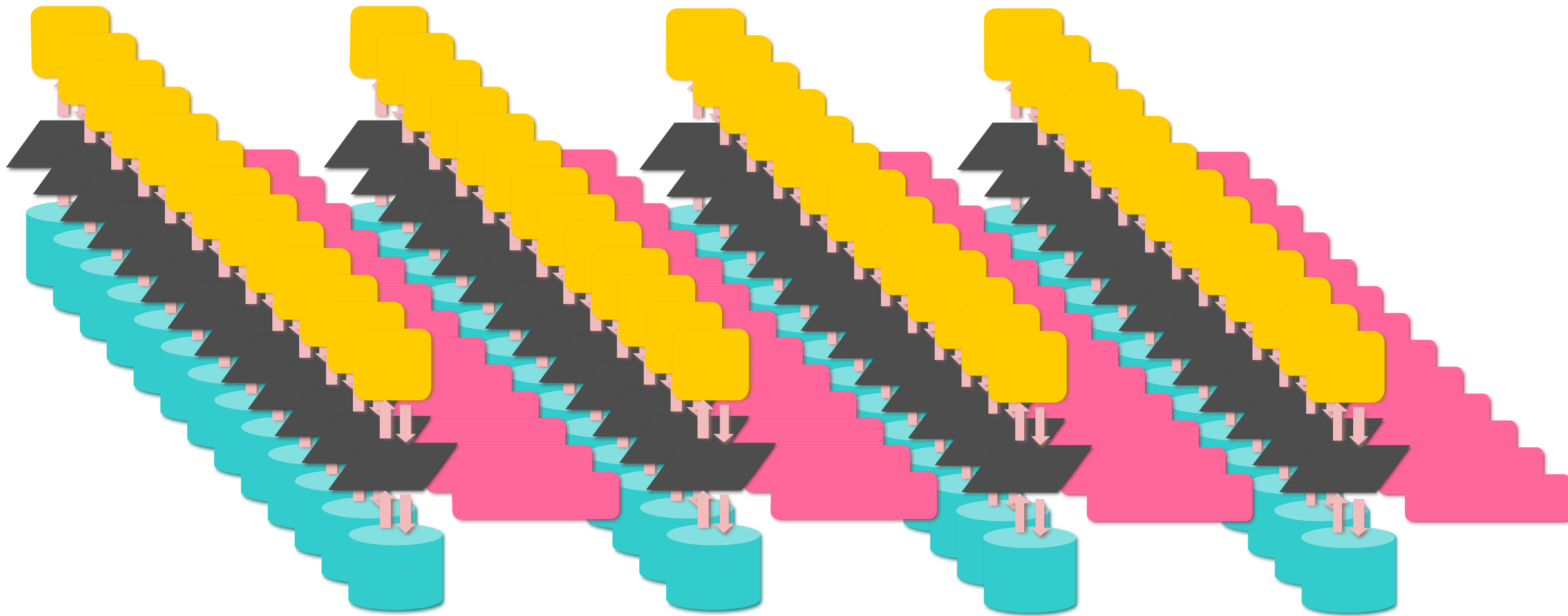
（一つの仕事をやる、たくさんの仕事は互いに連携して分担しあって片づける）

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



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

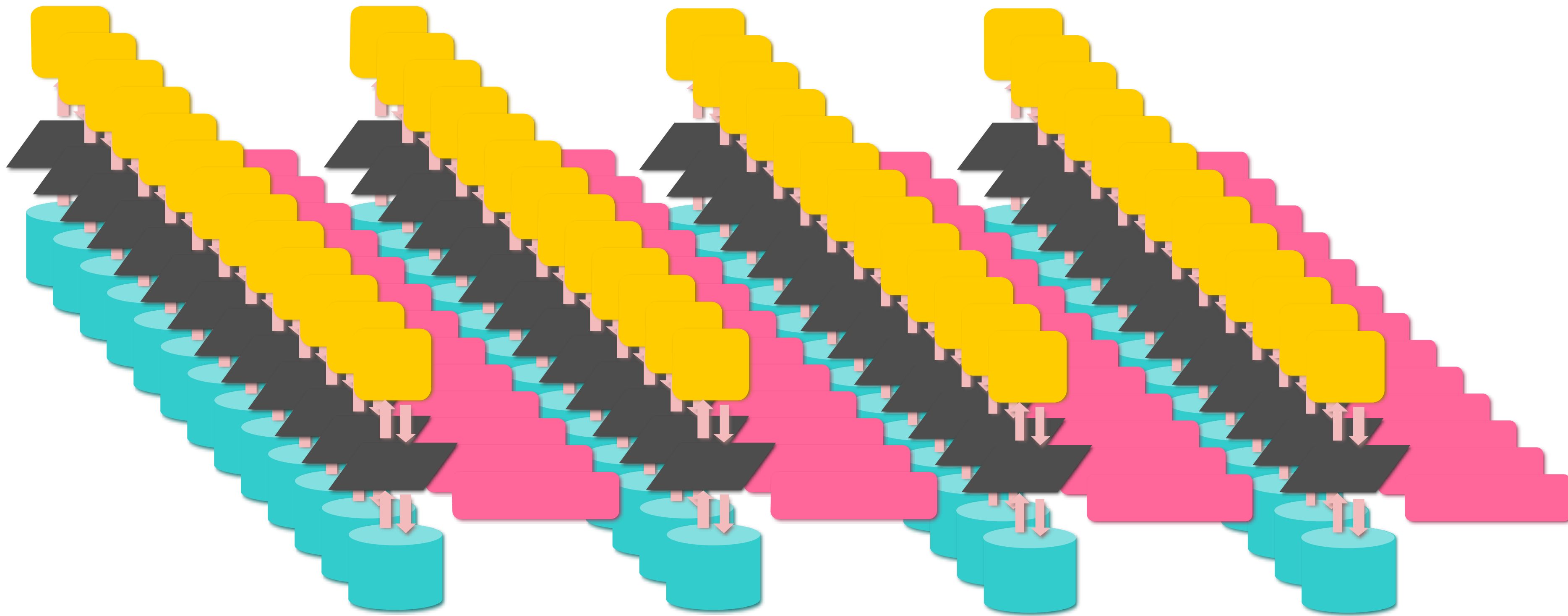


実際はハードディスクは、まとめる

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？

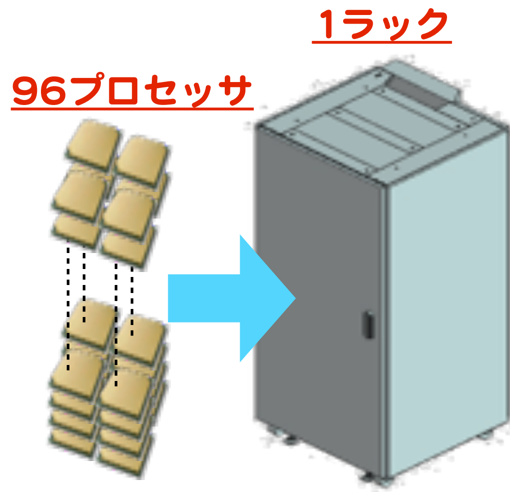
K computer

ちなみに「京」の場合は？

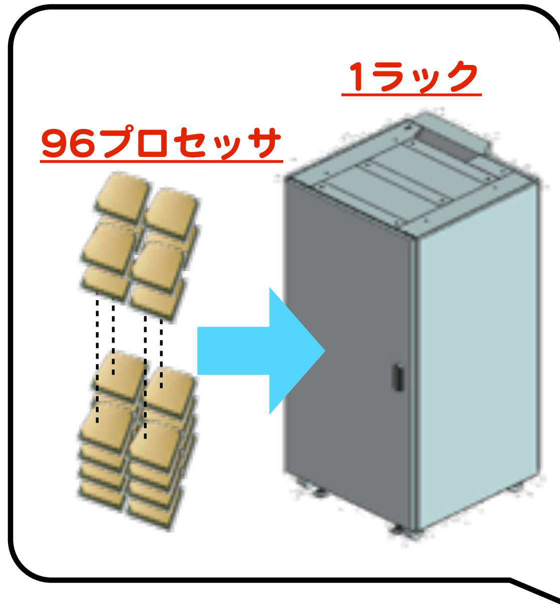
50m×60mの部屋に

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

864ラック
82944プロセッサ
66万コア

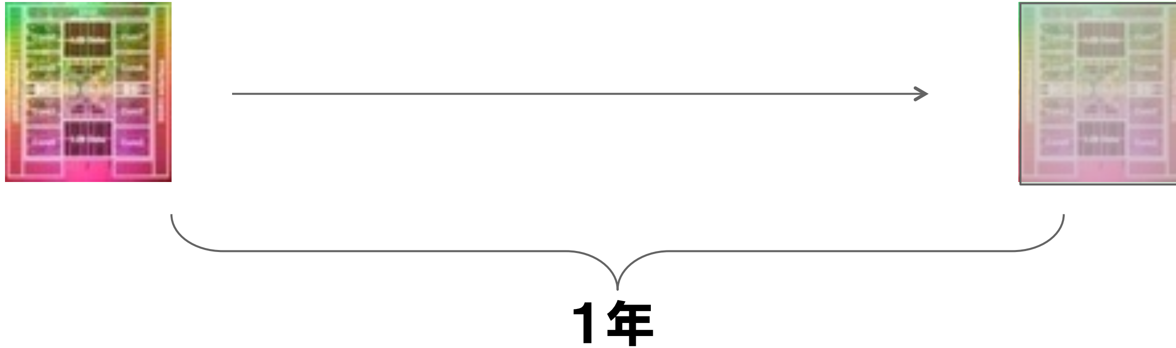
これだけ巨大システムでCPUが頻繁に故障すると・・・

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、

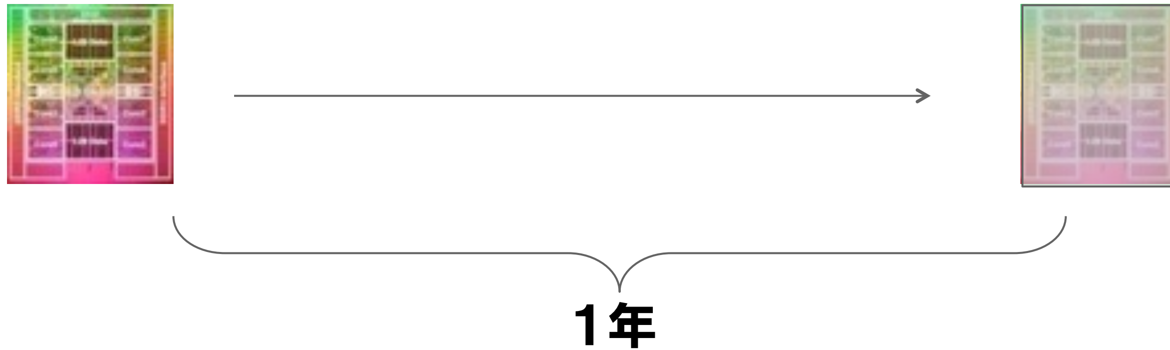
これだけ巨大システムでCPUが頻繁に故障すると・・・

✓ 仮にひとつのCPUが、1年に1回故障したとします



これだけ巨大システムでCPUが頻繁に故障すると・・・

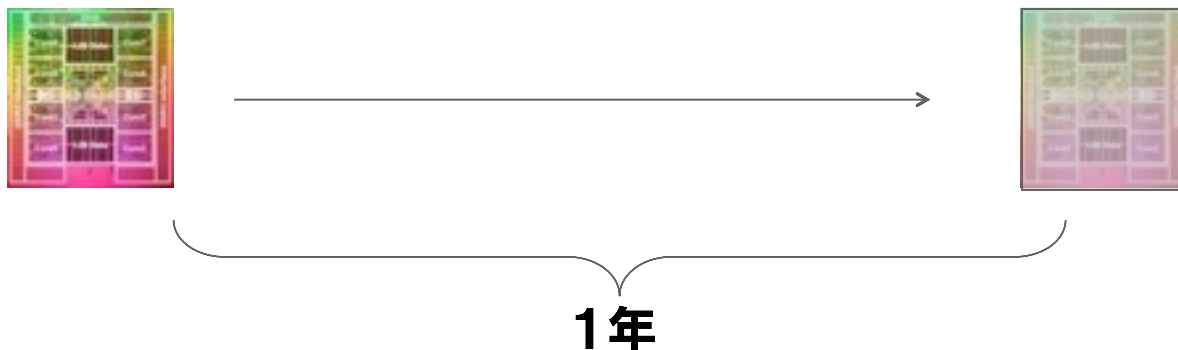
✓ 仮にひとつのCPUが、1 年に 1 回故障したとします



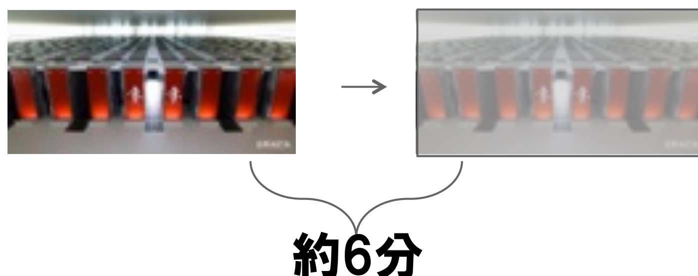
✓ システム全体で見ると、

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

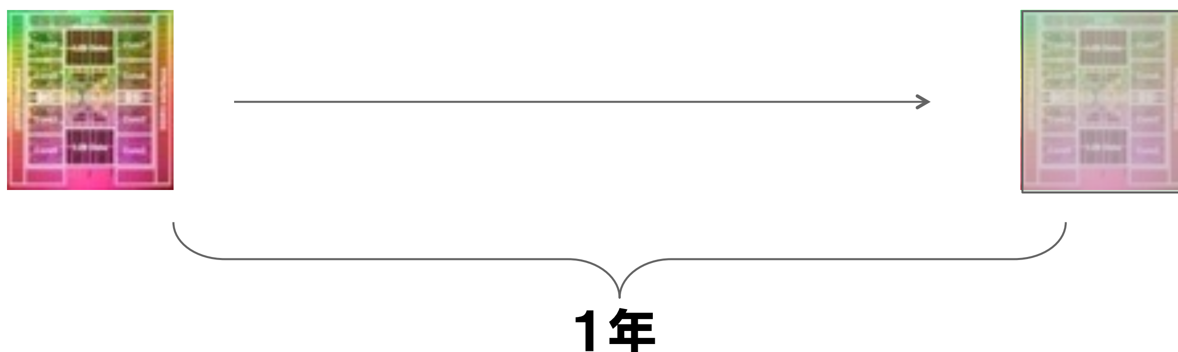


✓システム全体で見ると、　**約6分に1回**の故障・・・

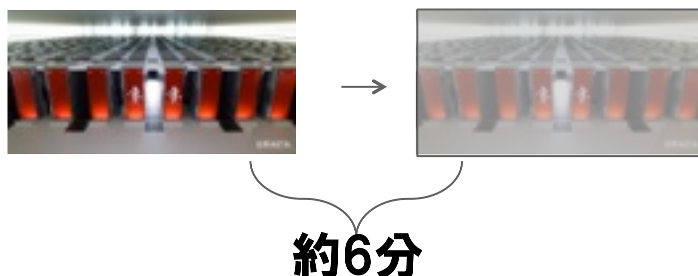


これだけ巨大システムでCPUが頻繁に故障すると・・・

✓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

K computer

アプリケーションの
性能とは？

科学について

理論

従来の
科学は

実験

スパコンはシミュレーションに使う

科学について

理論

実験

従来の
科学は

K computer

スパコンはシミュレーションに使う

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

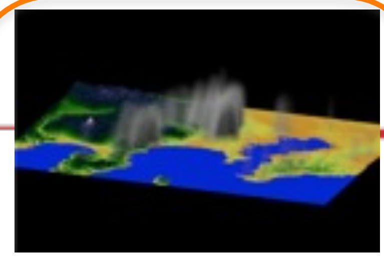
コンピュータ実験
（シミュレーション）

実験

京
K computer

具体的なアプリケーション

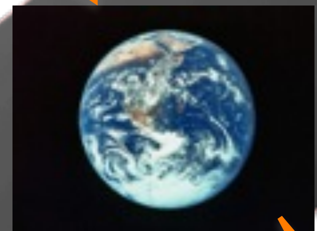
次世代のデバイス全体のシミュレーションによるエレクトロニクスへの貢献



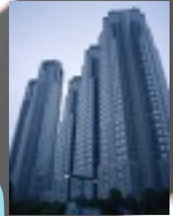
難しい大気現象の解明、また正確な台風の進路・強度の予測による気象への貢献

10^{21}m

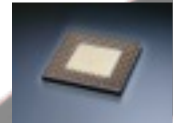
10^{7}m



10^{2}m



10^{0}m

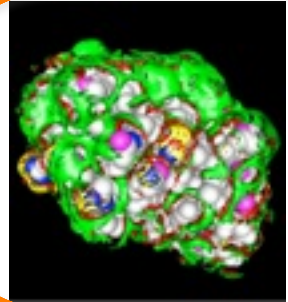


10^{-8}m

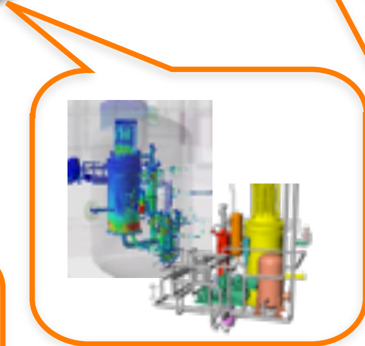
10^{-10}m

ポリオウィルス

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



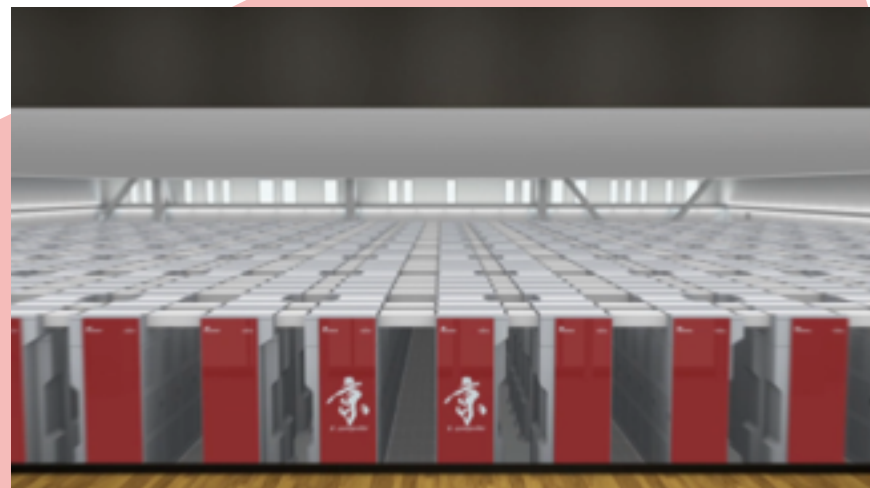
セルロース分解酵素のシミュレーションによる安価なバイオ燃料の提供等エネルギーへの貢献



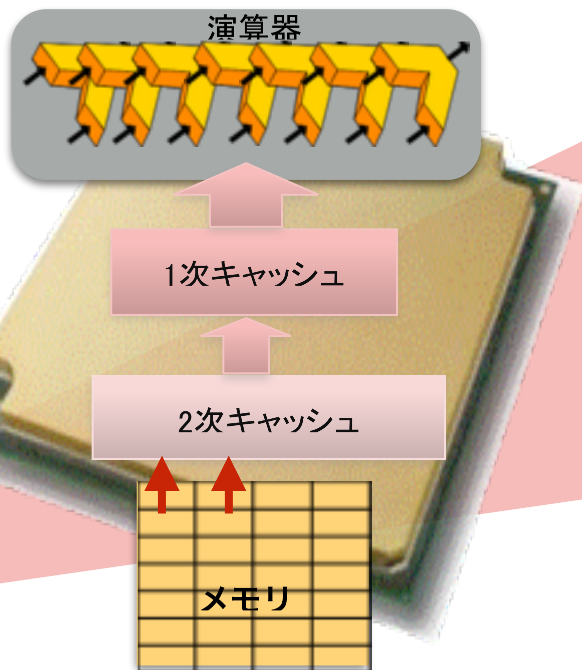
短周期地震波動の地震波シミュレーション、構造物の耐震シミュレーションを組み合わせた防災への貢献

現代のスパコン利用の難しさ

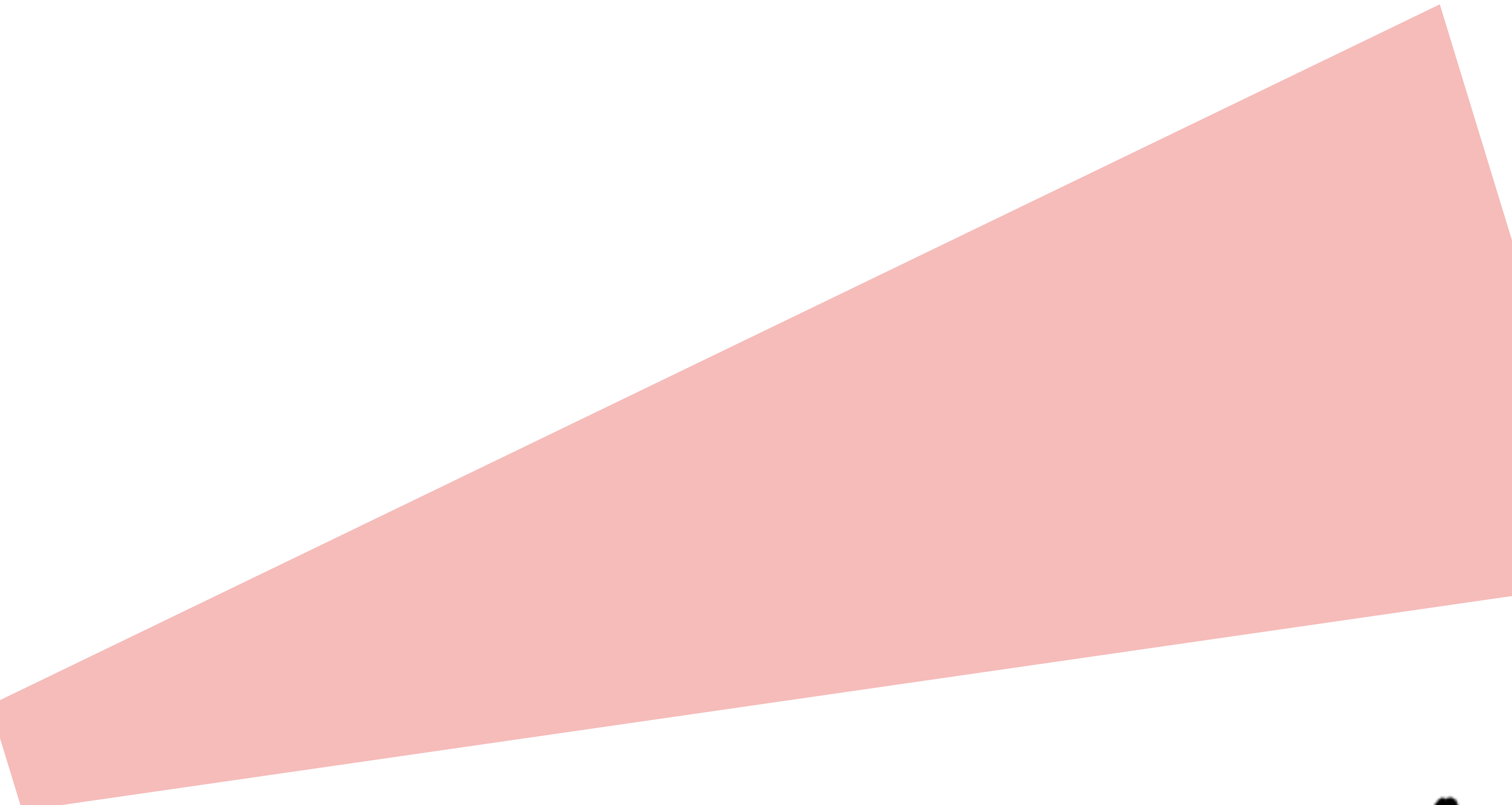
現代のスーパーコンピュータシステム



現代のプロセッサ

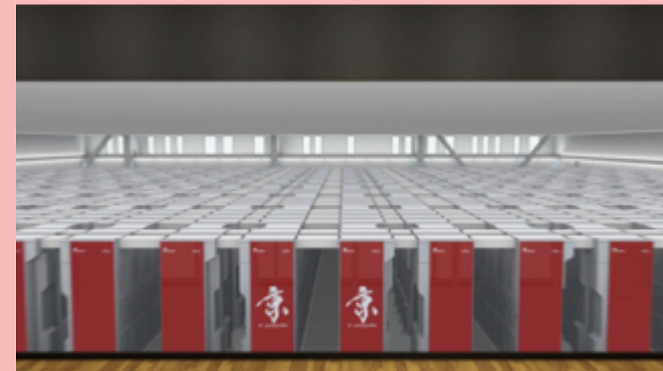


現代のスパコン利用の難しさ

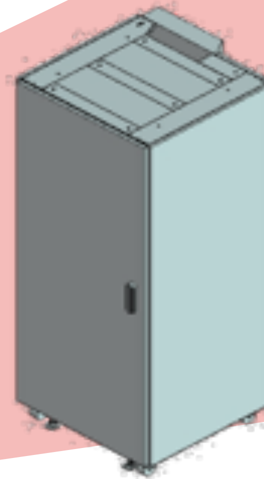


現代のスパコン利用の難しさ

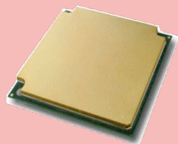
System



Rack

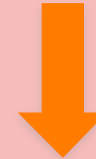


System Board

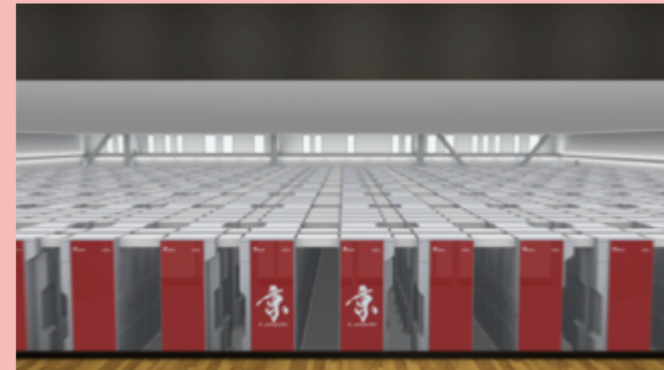


現代のスパコン利用の難しさ

アプリケーションの
超並列性を引き出す

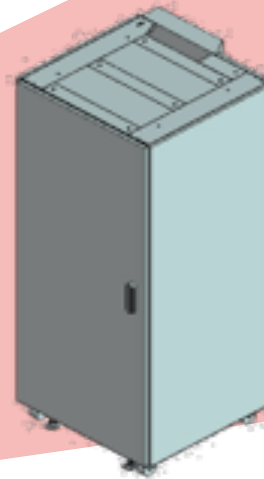


System

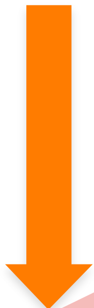


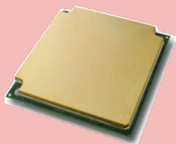
プロセッサの単体性能
を引き出す

Rack

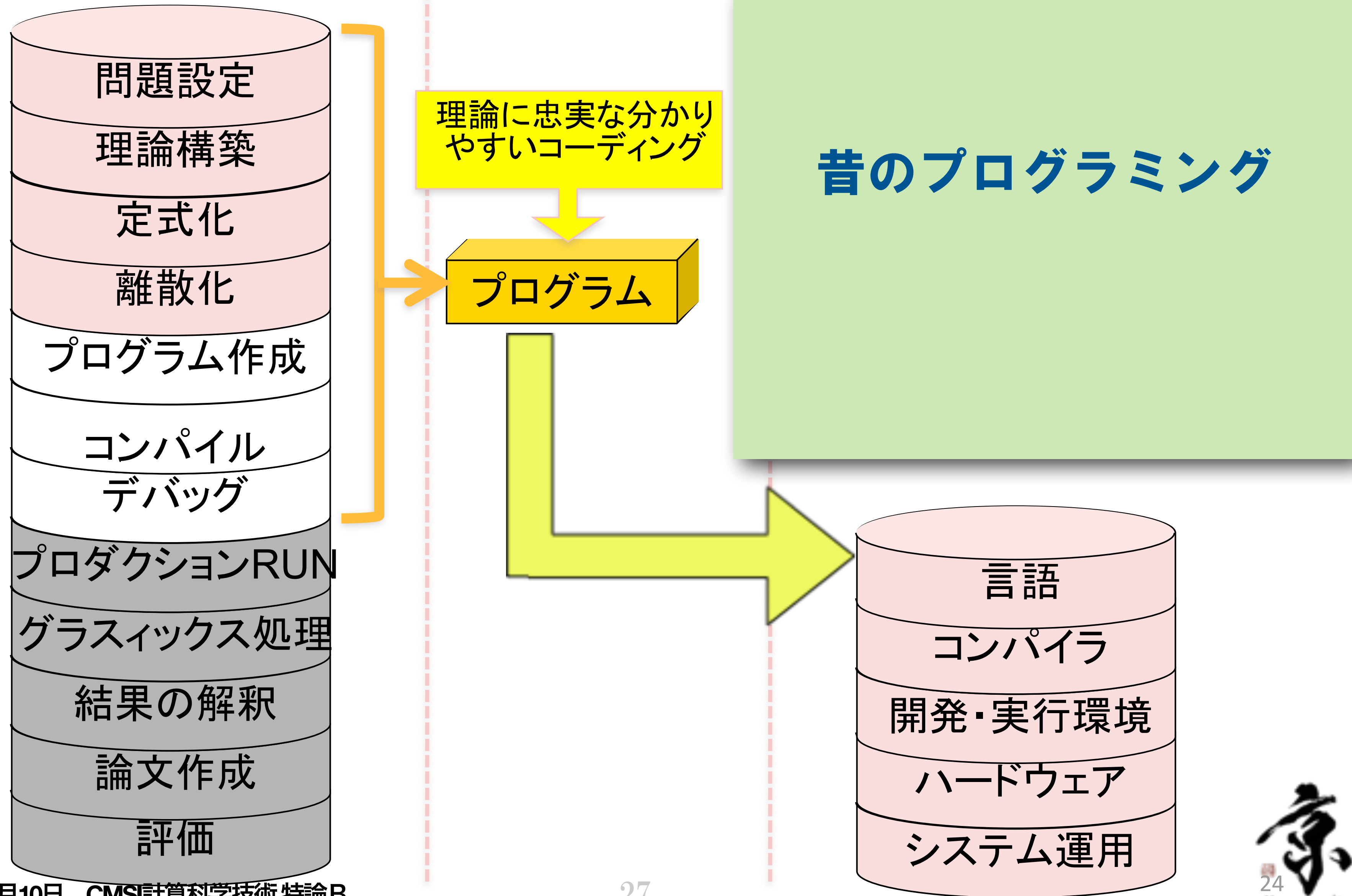


System Board



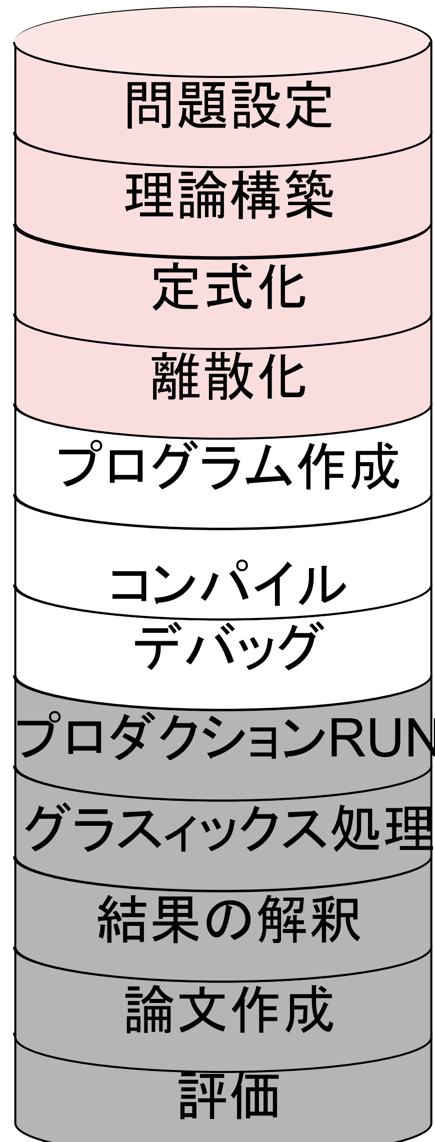


計算科学　　計算機科学



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

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

計算科学　　計算機科学

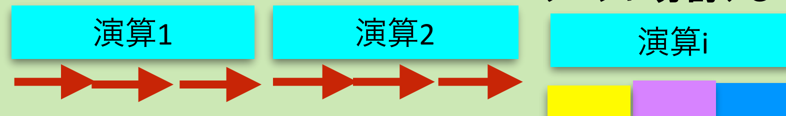
理論に忠実な分かりやすいコーディング

プログラム

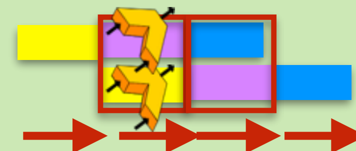
性能向上

(1) そのまま実行すると6clock

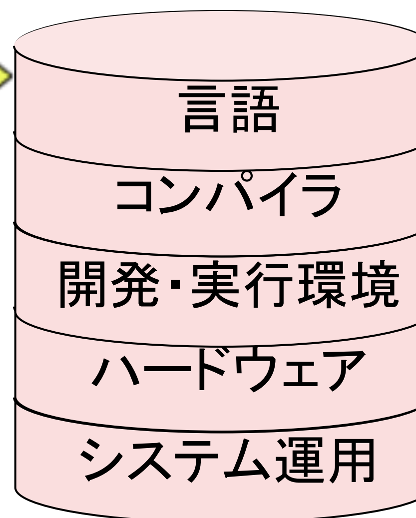
(2) 演算を3つのステージに分割する



(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) 処理や演算を細かく分割し複数の演算資源を並列に動作させ性能アップ



京 K computer

並列処理と依存性の回避

< スカラーパイプライン> **ハードウェア**

配列データ同士の足し算は

C(1)=A(1)+B(1)

C(2)=A(2)+B(2)

C(3)=A(3)+B(3)

:　　:

C(n)=A(n)+B(n)

と書き表せる。A,B ,CはそれぞれN個からなる配列データで ある。コンピュータが 足し算を行うときのプロセスを4つに分けて考える。

1.指数の比較　2.桁あわせ　3.加算　4.正規化

これら4つのプロセス を,1マシンサイクルのあいだに並行して 処理できる機構がパイ プラインである。つまり

指数比較　[1][2][3][4][5] …　→時間の流れ
桁あわせ　　[1][2][3][4][5] ...
加　算　　　[1][2][3][4][5] ...
正規化　　　　[1][2][3][4][5] …

この図ではあるタイミング、たとえば[4]の指数比較が行われると同時に[3]の桁あわせ、[2]の加算,[1]の正規化が行われている。これ らの[i]の 処理が スカラ 一命令で 駆動されるのがスカ ラーパ イプライン処理 である。(Fetch・Decode・EXecution・Memory Access・Write Backもパイプライン化されている)

並列処理と依存性の回避

< スーパースカラ方式> **ハードウェア**

- 並列実行可能な演算器を複数装備したプロセッサ

< レジスタリネーミング> **コンパイラ**

- r1=r0+r2
  r3=r1+r4
  r4=r5+r6
  r4=r7+r8

- 左側(1)(2)式は真の依存関係があるという
- 左側(2)(3)式は逆依存の関係があるという
- 左側(3)(4)式は出力依存の関係があるという
- このような依存関係がある場合、加算が1サイクルで実行可能な場合4サイクルかかる。
- そこで右側の式のようにレジスタを付け替えると依存関係はなくなる
- この場合は、(2)-(4)までは並列に実行可能であり、加算器を3つ以上持つスーパースカラプロセ ッサでは、1 サイクルで実行できる。
- したがって全体を2 サイクルで実行できる。
- このようなレジスタ番号の付け替えをレジスタリネーミングという。

並列処理と依存性の回避

< 内側ループアンローリング(1)> **コンパイラ**

・以下の様なコーディングの変更を内側ループアンローリングという。

```
do i=1,n
   x(i)=x(i)+a(i)*b+c(i)*d
end do

do i=1,n,2
   x(i)=x(i)+a(i)*b+c(i)*d
   x(i+1)=x(i+1)+a(i+1)*b+c(i+1)*d
end do
```

この変更により演算器を並列に使用できるようになる。

並列処理と依存性の回避

< 結合変換機能> **コンパイラ**

- x=a+b+c+d+e+fの形で計算する場合、各加算演算の間で依存性が発生する。
- しかしx=a+ b+ c+ (d+ e+ f)の形にすると前3つの加算と後3つの加算の間では依存がなくなる。
- この変換を結合変換という。

< 複数命令にまたがる依存性の消去>

- 以下のコーディング例で説明する。

```
do i=1,n
xx(i)=b1+a11*x(i)+a12*y(i)
yy(i)=b2+a21*x(i)+a22*y(i)
end do
```

- この中でb1+a11*x(i)とb2+a21*x(i)の計算は、依存性がない。
- このように依存性がない部分を見つけてやり依存性を消去する。

K computer

並列処理と依存性の回避

< アウトオブオーダー実行> **ハードウェア**

- 各命令間の依存関係をチェックしレジスタリネーミングによりレジスタ番号付けを行ったあと、これを効率よ く実行するためには、演算器を複数もった プロセッサを用意し動的に順序を入れ換えて命令を実行する必要がある。
- これをアウトオブオーダー実行という。

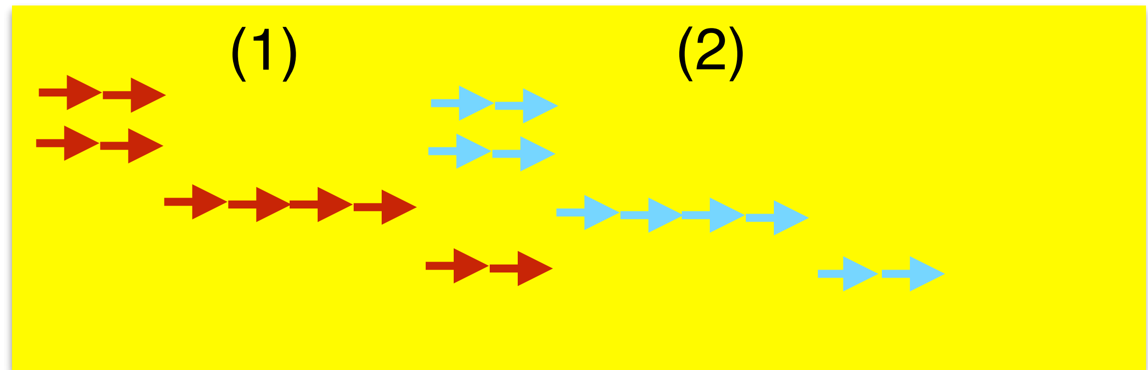
並列処理と依存性の回避

< ソフトウェアパイプラインニング> **コンパイラ**

<前提>
・ロード2つと演算とストアは同時実行可能
・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

例えば以下の処理をを考える。

```
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
```



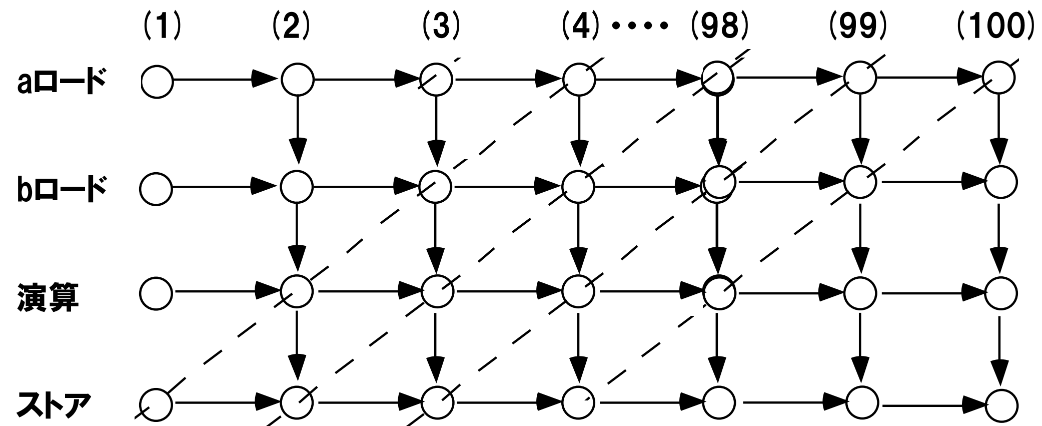
<実行時間>
・6クロック×100要素=600クロックかかる

並列処理と依存性の回避

< ソフトウェアパイプラインニング>

左の処理を以下のように構成し直す事を
ソフトウェアパイプラインニングという

```
a(1)a(2)a(3)のロード
b(1)b(2)のロード
(1)の演算
do i=3,100
      a(i+1)のロード
      b(i)のロード
      (i-1)の演算
      i-2番目の結果のストア
end do
b(100)のロード
(99)(100)の演算
(98)(99)(100)のストア
```



並列処理と依存性の回避

< ソフトウェアパイプラインニング>

左の処理を以下のように構成し直す事を
ソフトウェアパイプラインニングという

a(1)a(2)a(3)のロード

b(1)b(2)のロード

(1)の演算

do i=3,100

  a(i+1)のロード

  b(i)のロード

  (i-1)の演算

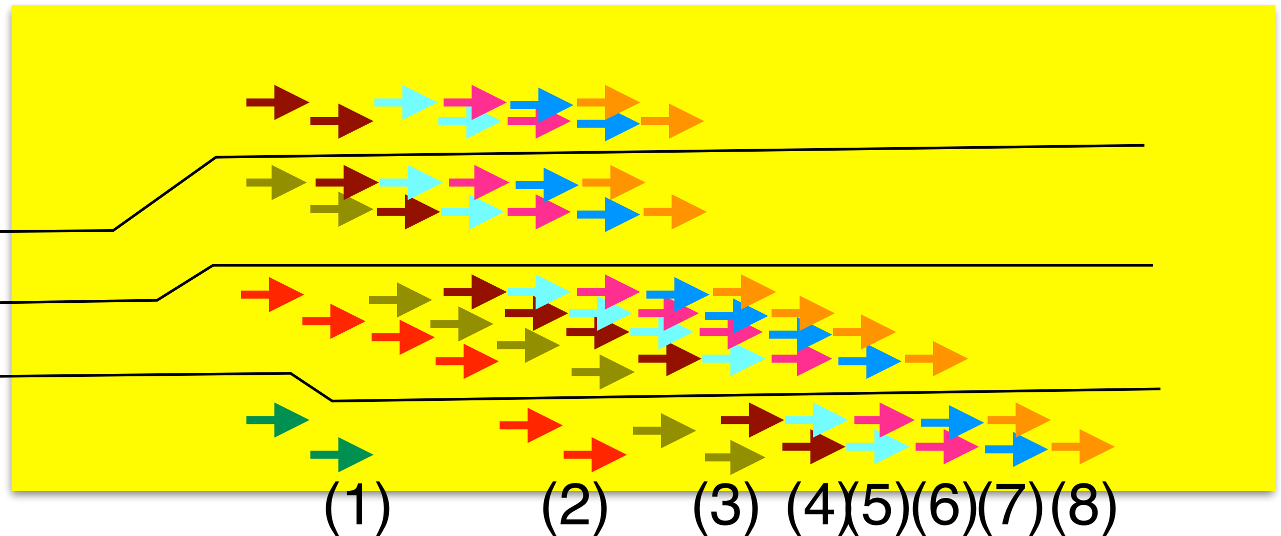
  i-2番目の結果のストア

end do

b(100)のロード

(99)(100)の演算

(98)(99)(100)のストア



<実行時間>
・前後の処理を及びメインループの立ち上がり部分を除くと
・1クロック×100要素=100クロックで処理できる

並列処理と依存性の回避

< 内側ループアンローリング(2)>

- 以下の様な2 つのコーディングを比較する。

```
do j=1,m
do i=1,n
   x(i)=x(i)+a(i)*b+a(i+1)*d
end do


do j=1,m do i=1,n,2
   x(i)=x(i)+a(i)*b+a(i+1)*d
   x(i+1)=x(i+1)+a(i+1)*b+a(i+2)*d
 end do
```

- 最初のコーディングの演算量は4 ,ロード/ストア回数は4 である.2 つ目のコ ーディン グの演算量は8 ,ロード/ストア回数は7 である.
- 最初のコーディングの演算とロード/ ストアの比は4/4,2つ目のコーディングの演算とロード/ストアの比は8/7となり良くなる.

並列処理と依存性の回避

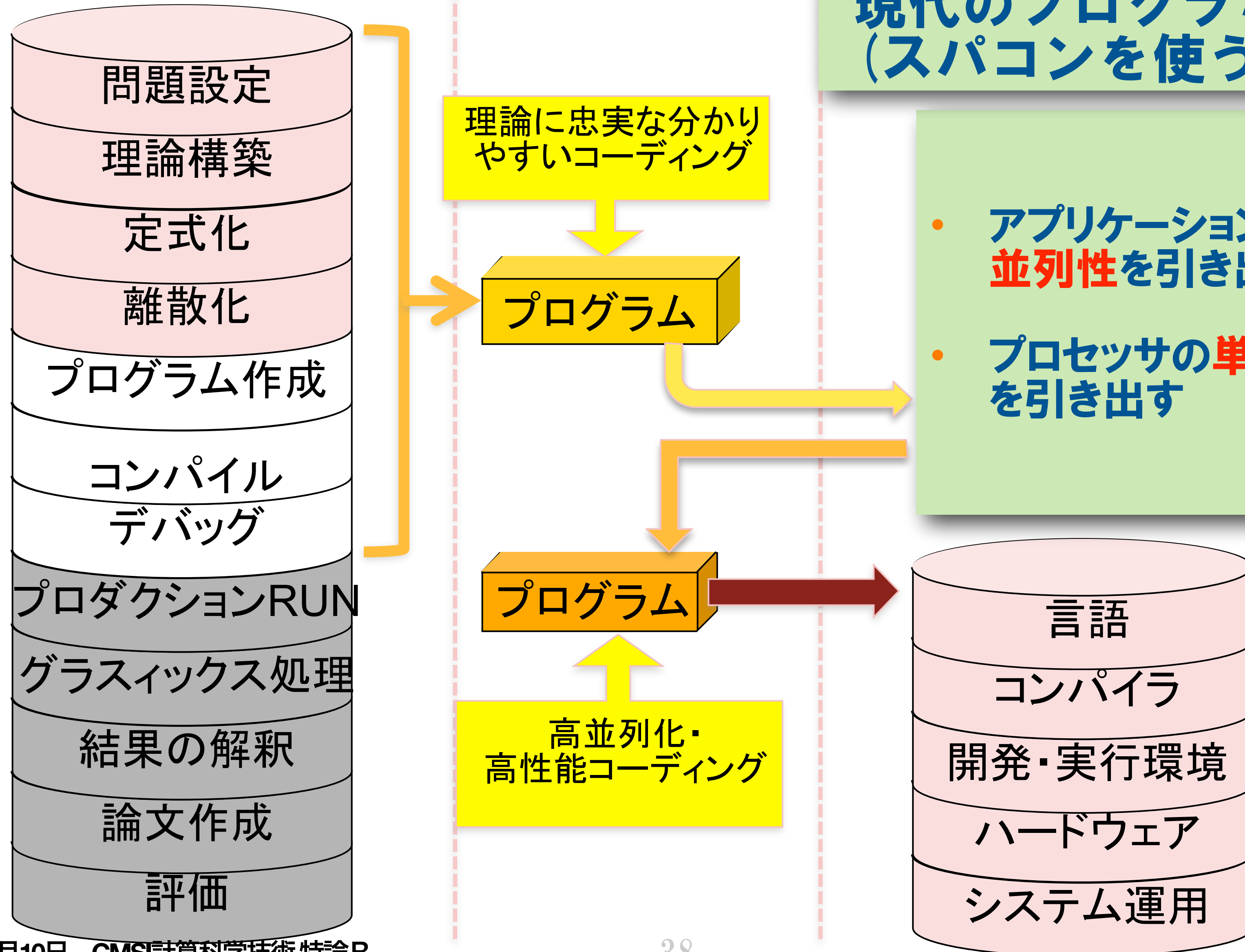
< ロード命令の先行実行>

(例題)

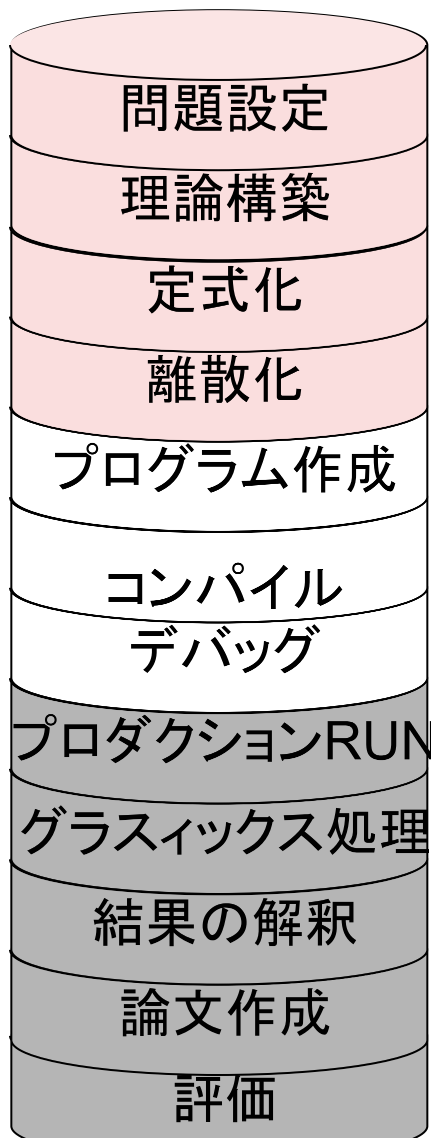
```
add——              load r1             …(1)
multiply——         add ——              …(2)
load  r1           multiply ——         …(3)
add   r3  r1  r2   add  r3  r1  r2     …(4)
----------------------------------------------------
```

- 左側の命令列でもし(3)においてキャッシュミスが生じると直後の命令でr1を使っているため待ちが生じてしまう.
- したがって右側のようにload命令をできるだけ先に実行してしまうような命令列に変更する.
- この命令列であれば、(1)でキャッシュミスが発生しても、そのデータを使う(4)までの間に(2)(3)は並行して実施できるため，キャッシュミスのペナルティを小さくできる.
- これをロード命令の先行実行という.

計算科学　計算機科学



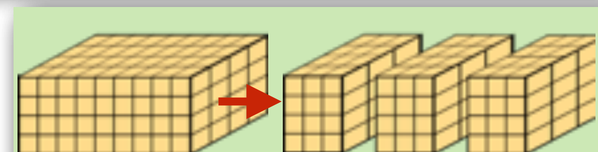
計算科学 計算機科学



理論に忠実な分かりやすいコーディング

プログラム

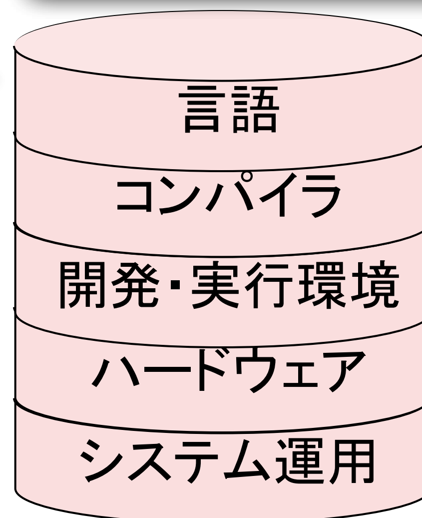
現代のプログラミング（スパコンを使う場合）



- 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

プログラム

高並列化・高性能コーディング



アプリケーションの性能についてまとめると

- 従来，単体プロセッサの時代からプログラムに内在する並列性をコンパイラで解釈しハードウェアに装備された並列処理機構を利用することによりことにより，高速処理を実現してきた．

- しかし現代のスパコンプログラマーは，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%の性能しかでない．

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

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

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

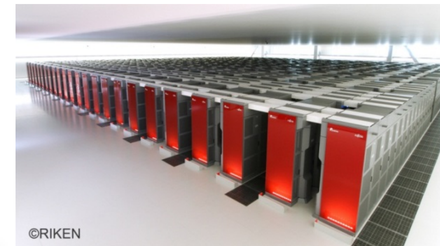
パソコンで100年かかる計算

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

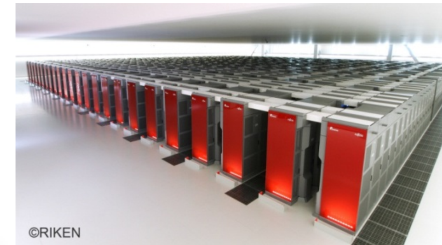


スーパーコンピュータの威力

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

京
K computer

高並列化のための重要点

K computer

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？（1）

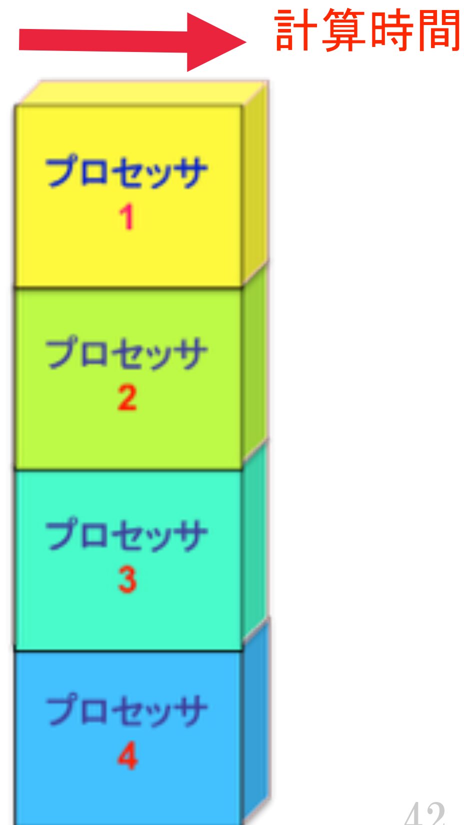
計算時間

逐次計算

並列計算

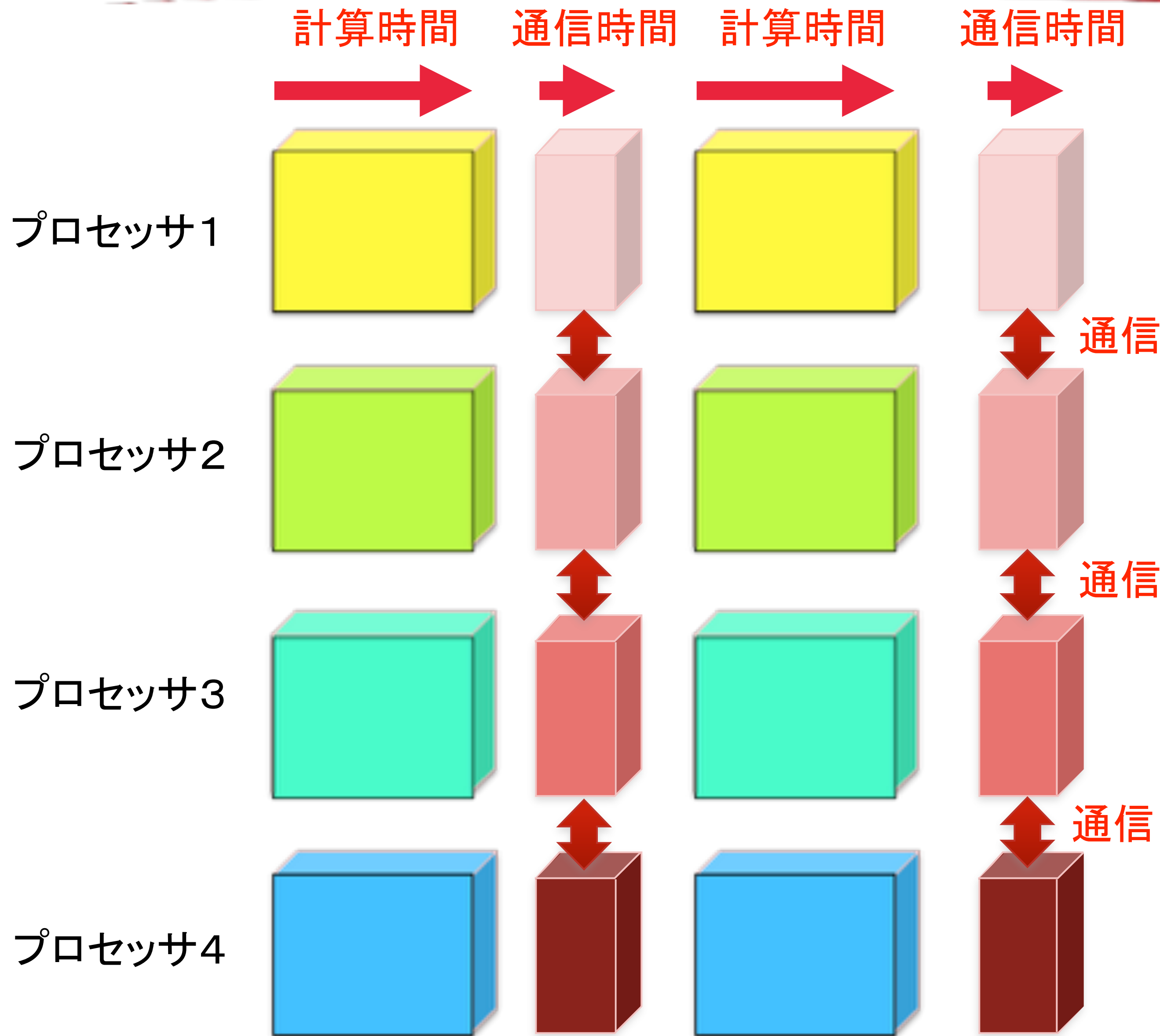
そもそも並列化とは？（1）

逐次計算

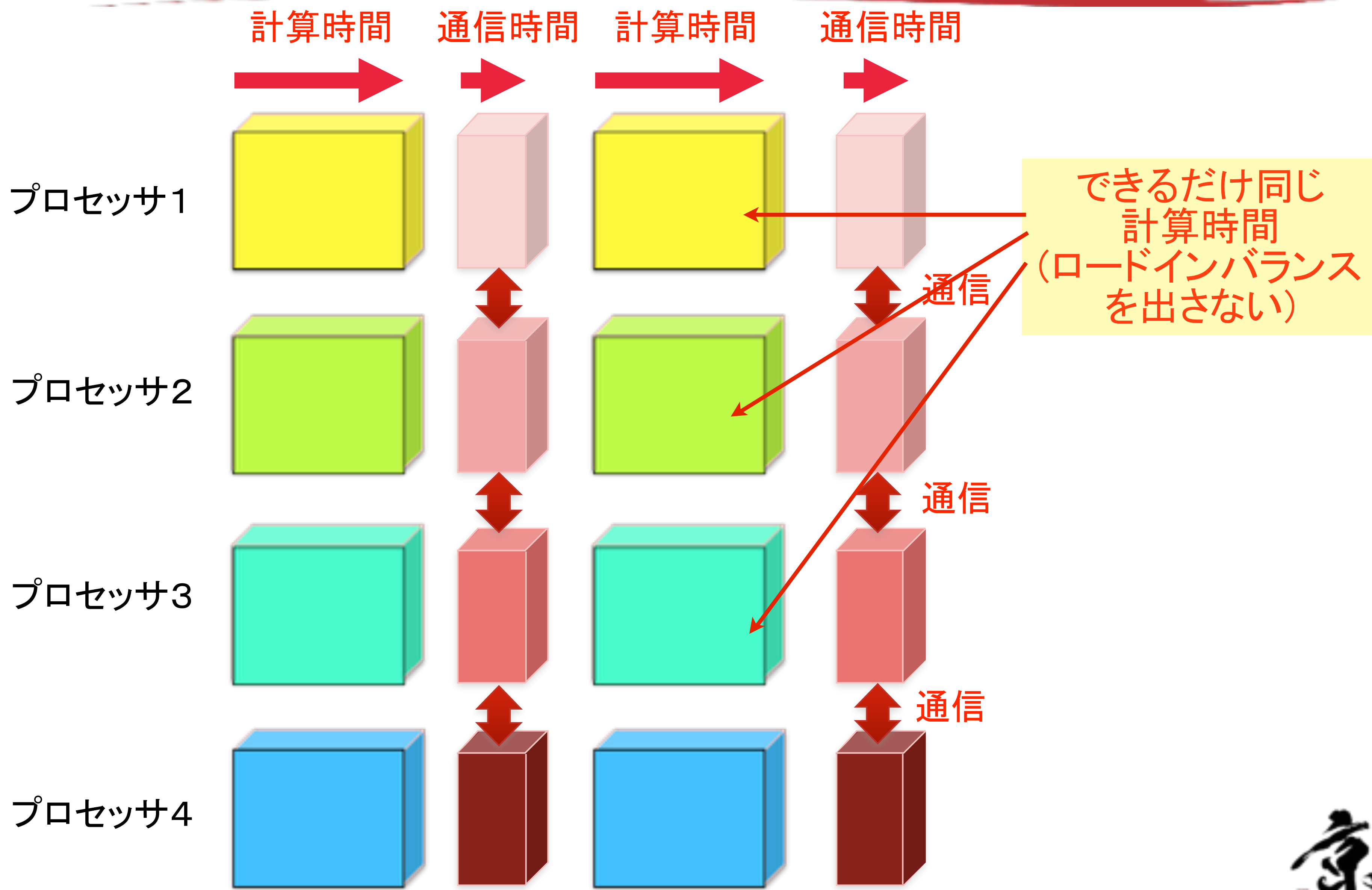


並列計算

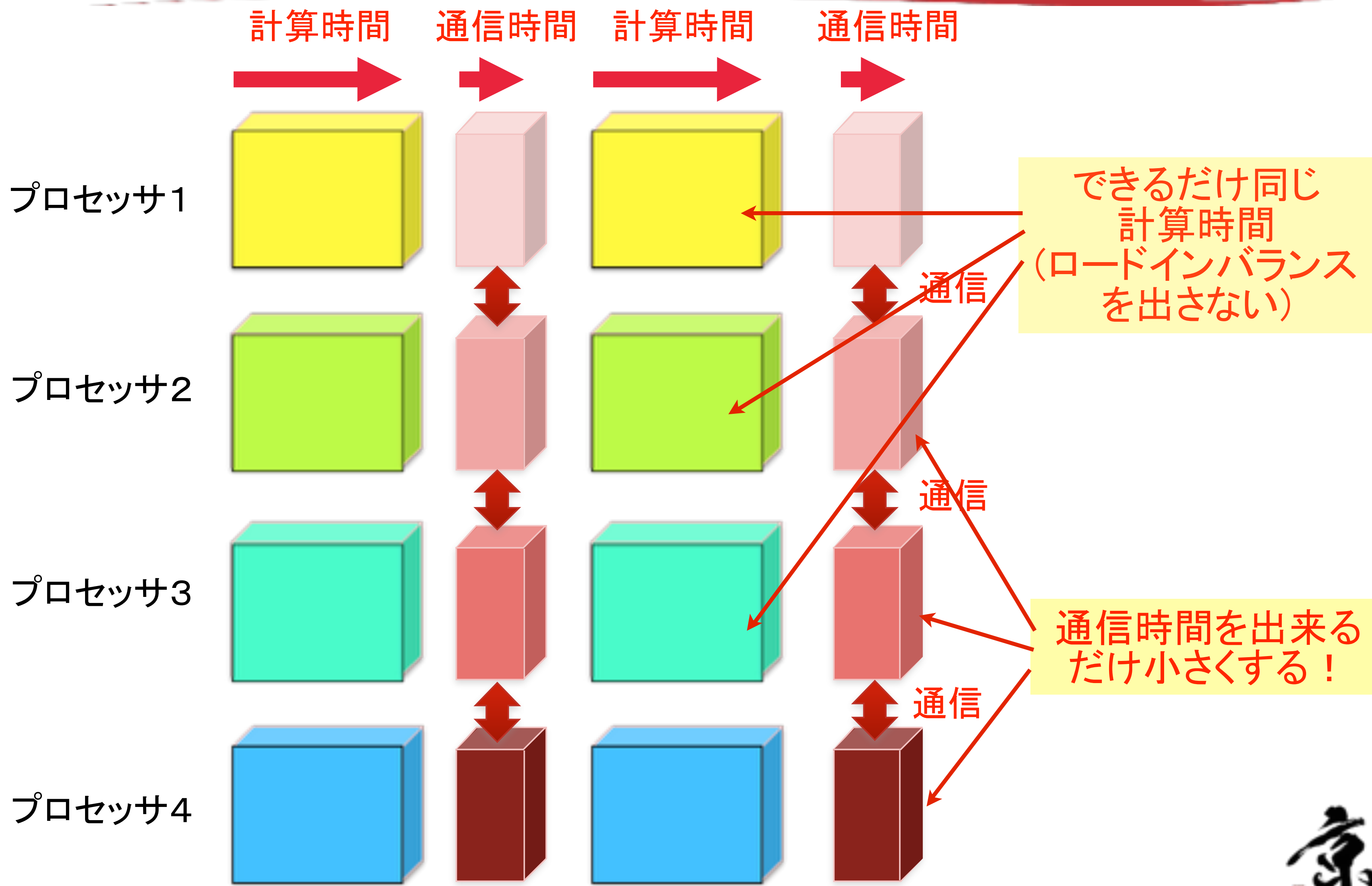
そもそも並列化とは？（2）



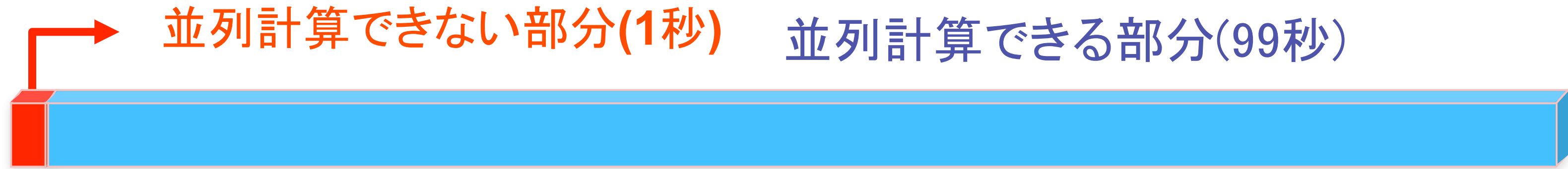
そもそも並列化とは？（2）



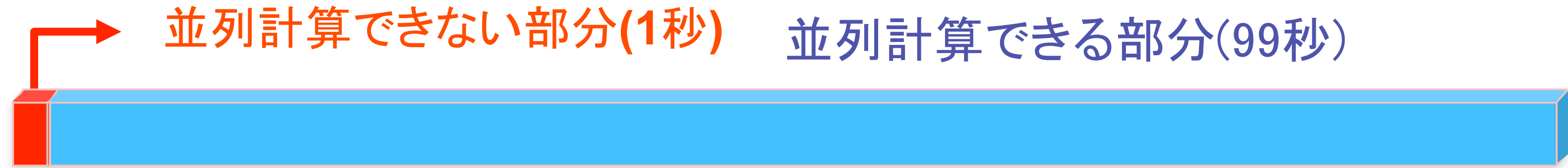
そもそも並列化とは？（2）



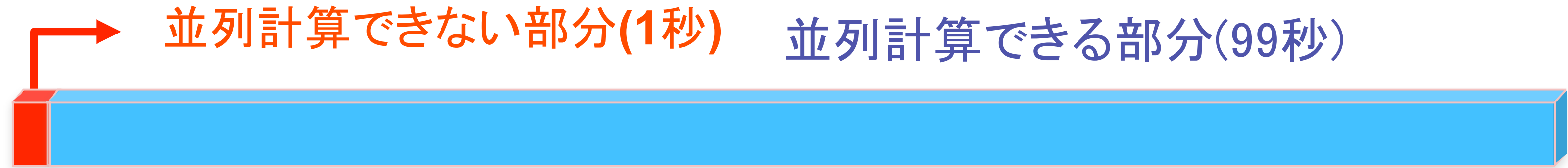
そもそも並列化とは？（3）

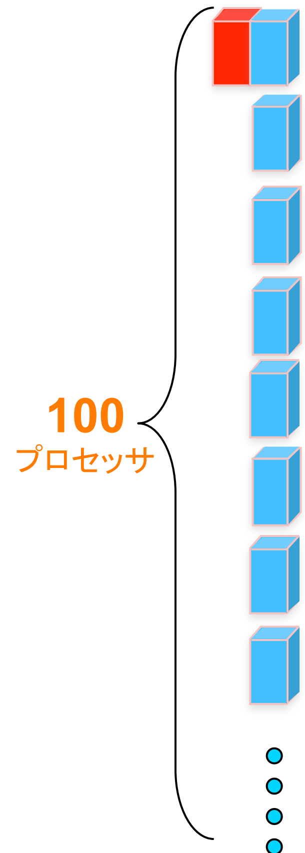


そもそも並列化とは？（3）



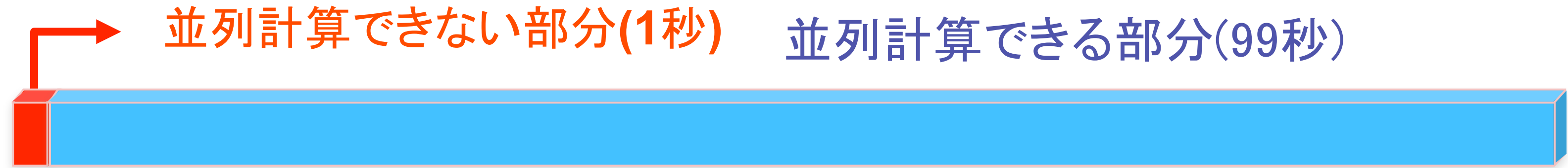
そもそも並列化とは？（3）



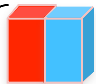


そもそも並列化とは？（3）

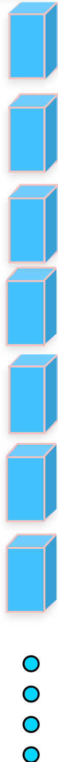
並列計算できない部分(1秒)　並列計算できる部分(99秒)



1+0.99=1.99秒

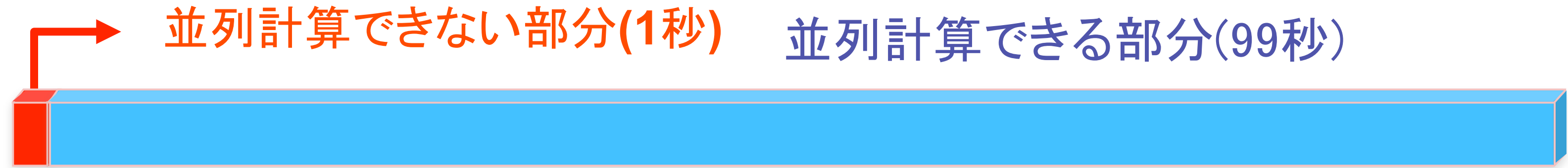


100
プロセッサ

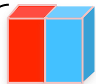


そもそも並列化とは？（3）

並列計算できない部分(1秒)　並列計算できる部分(99秒)

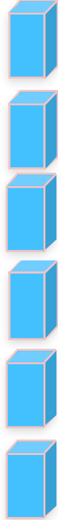


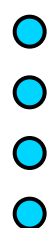
1+0.99=1.99秒



ほぼ50倍

100
プロセッサ





そもそも並列化とは？(3)

1+0.99=1.99秒

1+0.099=1.099秒

そもそも並列化とは？（3）

1+0.99=1.99秒

1+0.099=1.099秒

ほぼ50倍

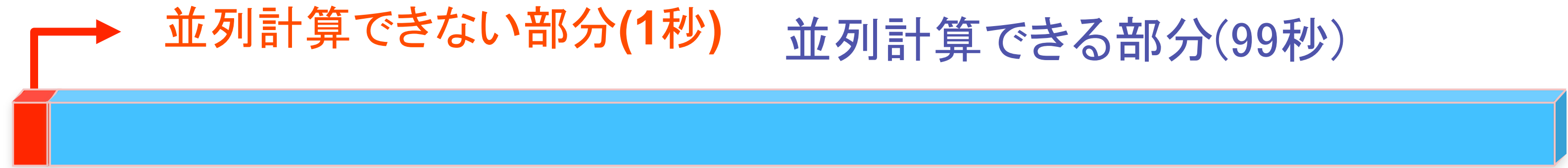
ほぼ91倍

100
プロセッサ

1000
プロセッサ

必要な並列度を確保する！

そもそも並列化とは？（3）



1+0.99=1.99秒

1+0.099=1.099秒

ほぼ50倍

100
プロセッサ

ほぼ91倍

1000
プロセッサ

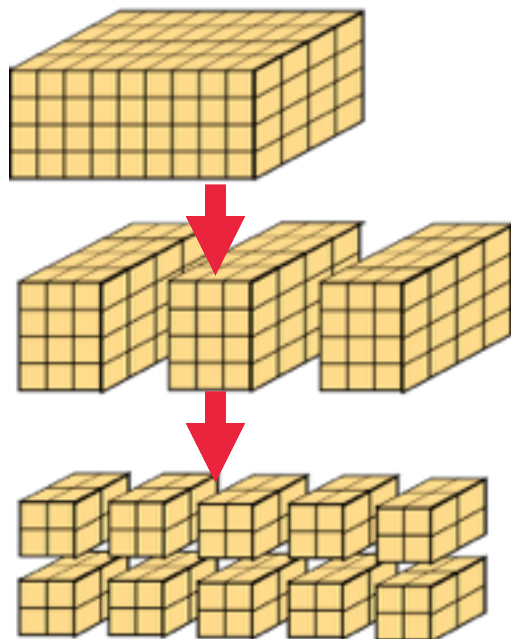
必要な並列度を確保する！

並列計算できない部分（非並列部）を限りなく小さくする！

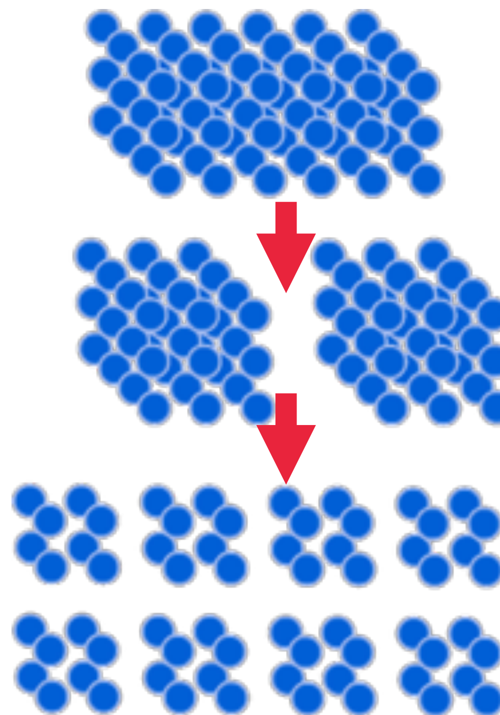
K computer

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
```


K computer

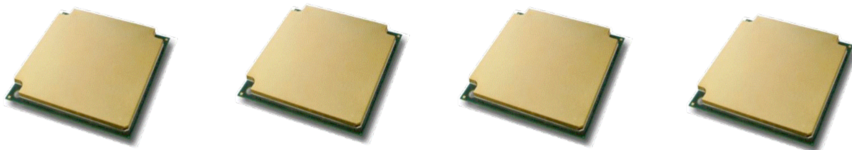
単体性能向上のための重要点

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

データ



演算器

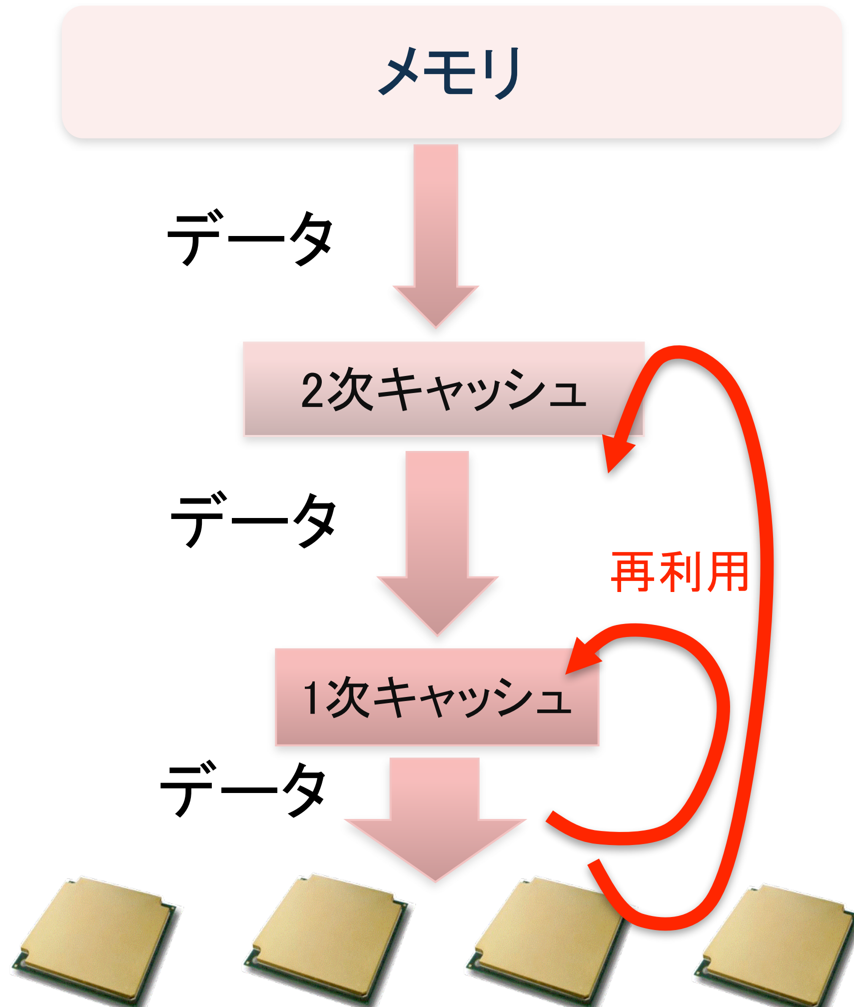
メモリウォール問題

✓ 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

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)

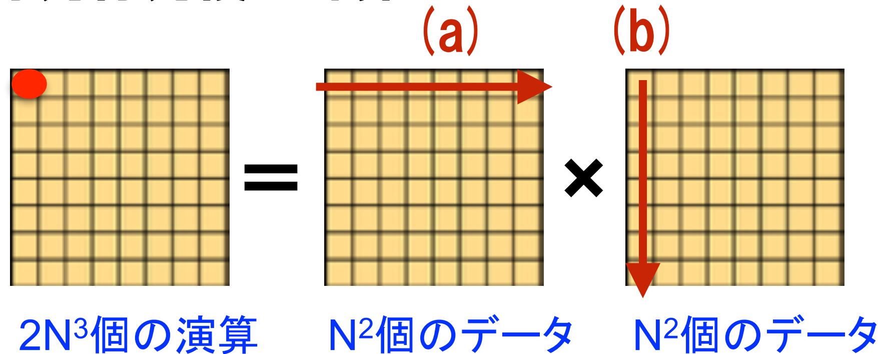
(*1)演算量(Flop) に比べ
データの移動量(Byte)が小さい計算

キャッシュの有効利用

演算器

もう少し詳しく説明すると

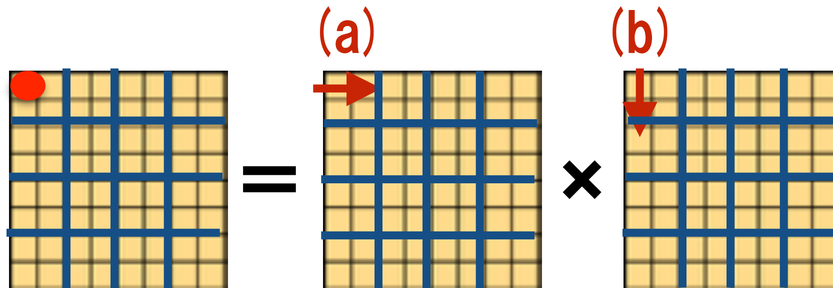
行列行列積の計算



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**

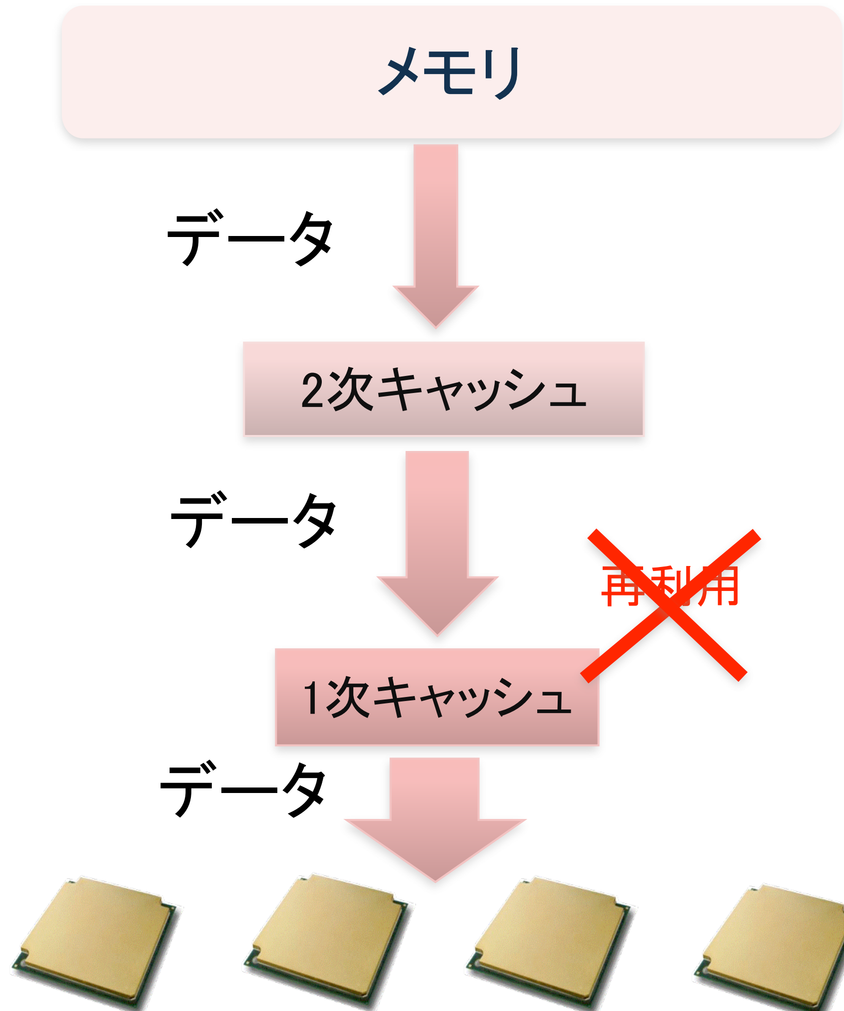


- **そこで行列を小行列にブロック分割し (a) も (b) もキャッシュに乗るようにしてキャッシュ上のデータだけで計算するようにする事で性能向上を実現する.**

K computer

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)

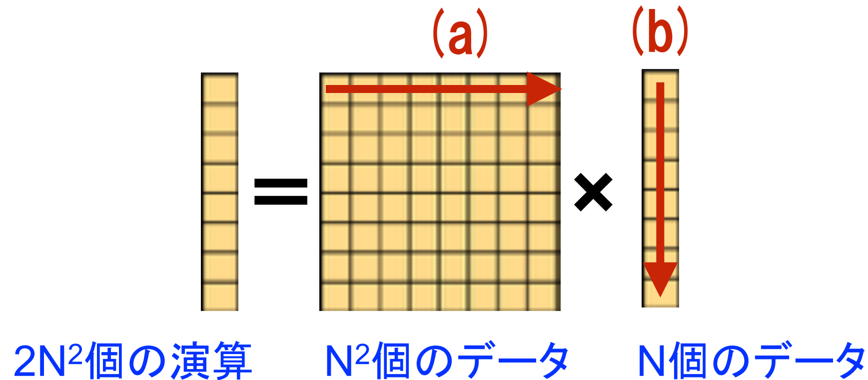
(*2)演算量(Flop) に比べデータの移動量(Byte)が大きい計算

キャッシュの有効利用が難しい

演算器

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

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

- **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

p7.p8,p14-p17については RIKEN AICS井上統括役の資料である．

K computer