AICS サマースクール　8/6 9:30-10:30

並列処理・HPCの基礎

佐藤　三久

理化学研究所　計算科学研究機構(AICS)

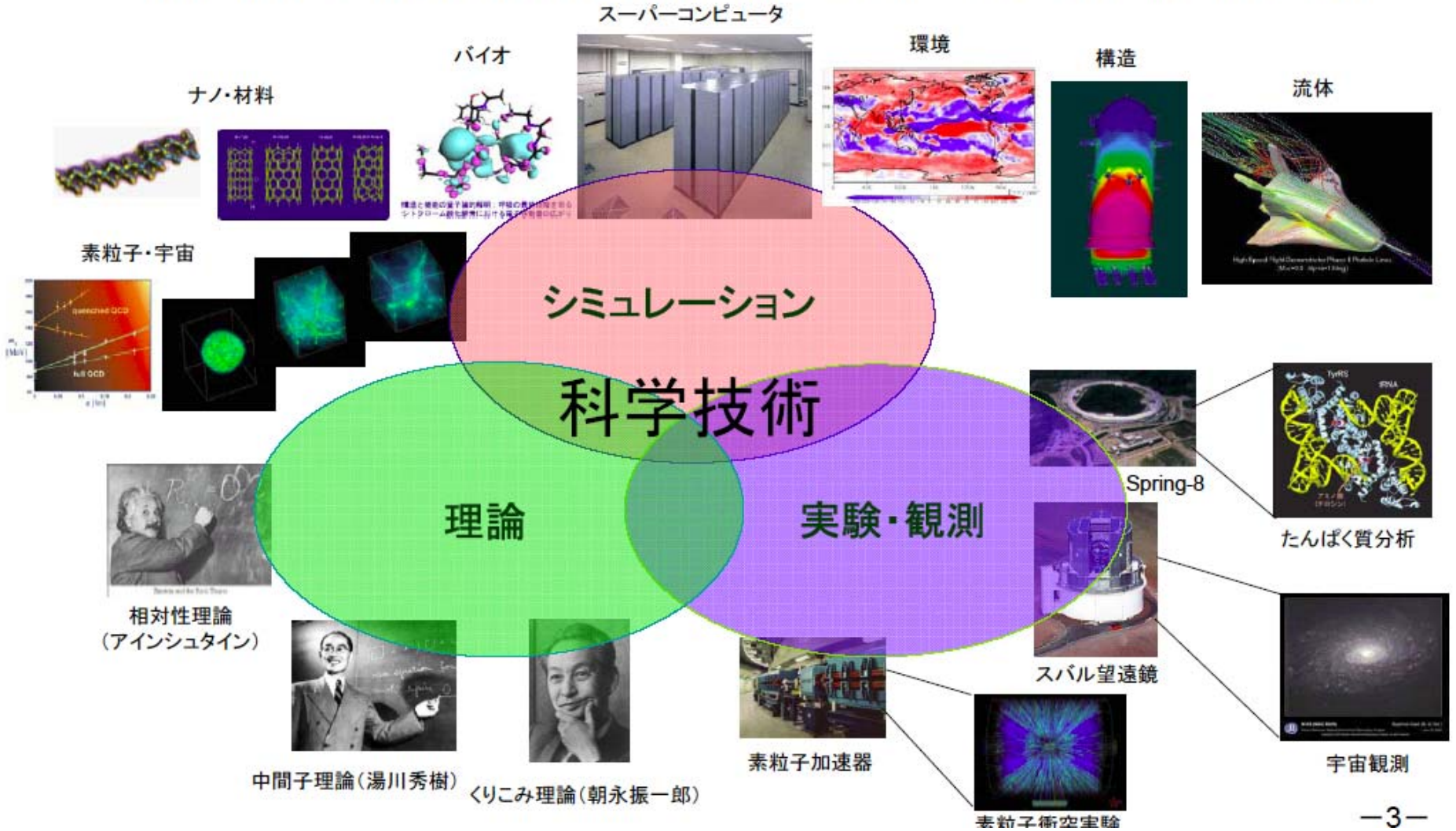
なぜ、並列処理なのか？

計算科学

スパコン（超高速計算機）を使った
シミュレーションで
科学の研究を行うこと

科学の三本柱としての計算科学

- 超高速計算機（スーパーコンピュータ）を用いた大規模シミュレーションを中心とした研究
- 科学技術の全分野で、実験・観測、理論と並ぶ、重要且つ最先端の研究手段



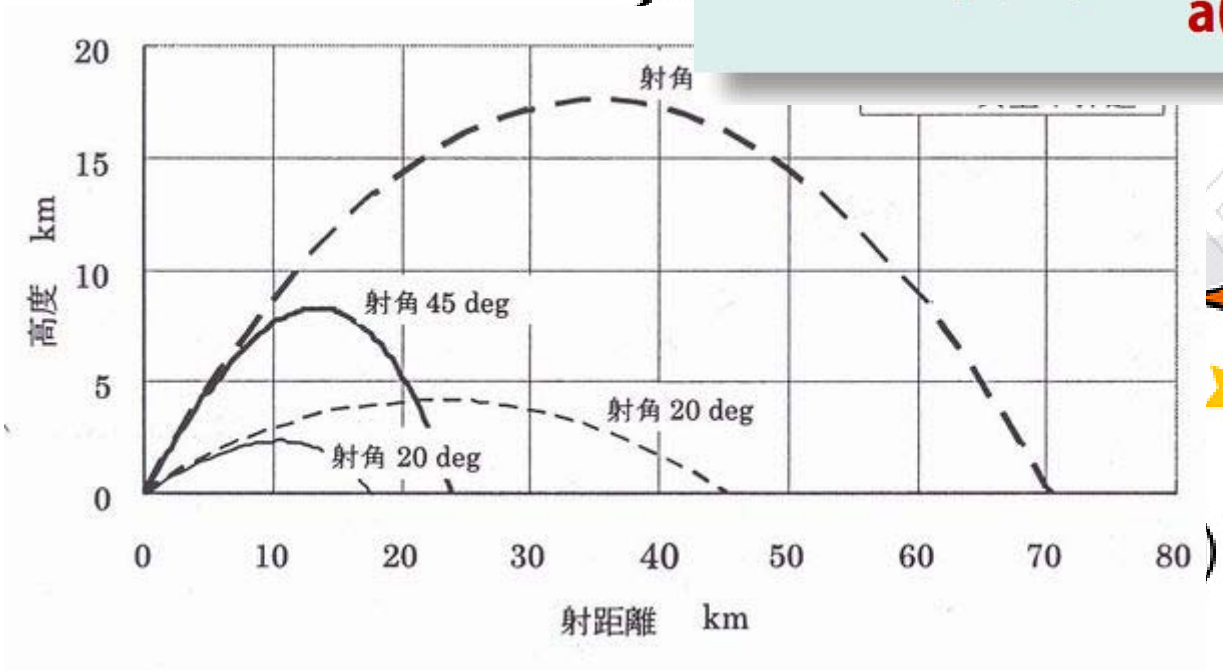
コンピュータのはじまりは？

- 弾道計算、暗号解読、…

ニュートンの運動方程式

$$F(力) = m(質量) \times a(加速度)$$

$$m(質量) = \frac{F(力)}{a(加速度)}$$



例えば、…

- Fがわかれば、速度V(t)（時刻tの時の速度）は

$$V(t+\Delta t) = V(t) + F(t) \times \Delta t$$

F(t)は時刻tの時の力

- 次々と、Vを計算していけば、同じようにそれぞれの時間でのVが求まる

- 速度が求まれば、位置p(t)も同じように

$$p(t+\Delta t) = p(t) + v(t) \times \Delta t$$

数値予報(天気予報)の場合は...

- 天気予報とは、晴れ、曇り、雨などの天気状態と、温度、湿度、風速、気圧などをある領域で、時間経過とともに予測すること。
  - 晴れ、曇り、雨などの天気状態は、温度や湿度、風速、気圧がわかれば、その状態として考えることができるはず。
  - 温度や湿度、風速、気圧は物理量なので、運動方程式からわかるはず。
  - V. ビァネークス(1904)「力学的、物理学的基礎に立つ問題」
- 空気に働く力＝気圧傾度力、コリオリ力、摩擦力
  - これをもって、運動方程式をとけばいい...

リチャードソンの夢

- 最初に数値シミュレーションによる予報実験を試みたのはイギリスのリチャードソンです。コンピュータの実用化以前の1920年頃、およそ水平200km間隔で鉛直５層の格子を用い、６時間予報を１か月以上かけて手計算で行いました。残念ながら、用いた数値計算に難点があり、非現実的な気圧変化を予測してしまい、野心的な試みは失敗に終りました。しかし、リチャードソンはその著書の中で、「６万４千人が大きなホールに集まり一人の指揮者の元で整然と計算を行えば、実際の時間の進行と同程度の速さで予測計算を実行できる」と提案しました。

気象庁のホームページから

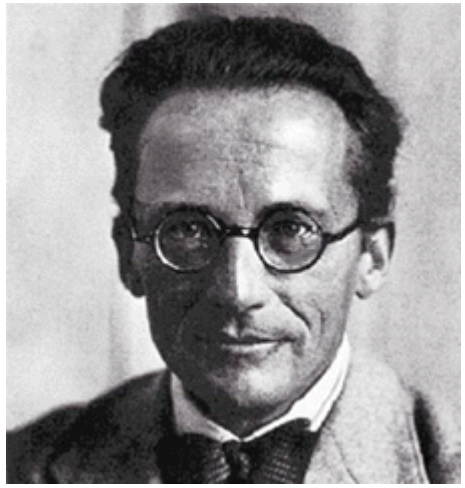
Figure 1.4 Lewis Fry Richardson (1881–1953). Photograph by Walter Stoneman, 1931, when Richardson was aged 50. (Copy of photograph courtesy of Oliver Ashford)

物質の原理をさぐるには、…

- 量子力学　シュレデンガー方程式

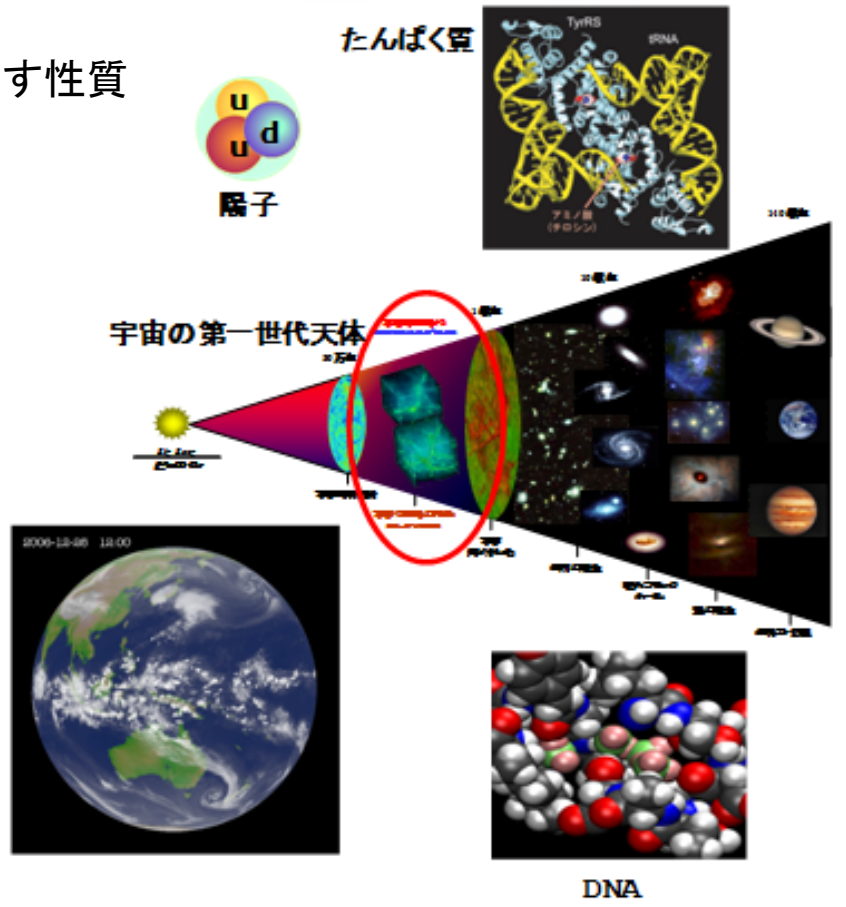
$$\mathbf{H}\psi = E\psi \qquad i\hbar\frac{\partial\psi}{\partial t} = \left[-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + V(x)\right]\psi$$

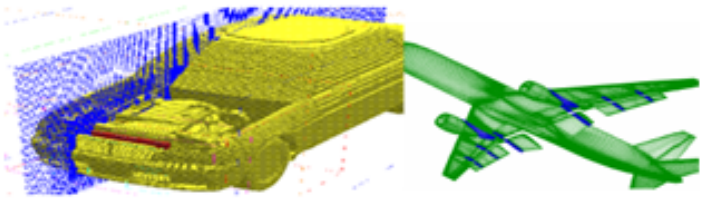
- これを用いて、物質のシミュレーションをすることを「第一原理計算」という。



計算科学の重要性：何に役立つのか

- 「紙と鉛筆」では解けないような複雑な現象の探求
  - 物質の根源である素粒子の成り立ち
  - DNAやたんぱく質等数百万個の原子の集団の示す性質
- 実験ができない現象の探求
  - 宇宙における最初の天体の起源
  - 地球規模の気候変動と温暖化予測
- 膨大な大規模データの探索
  - ゲノムインフォマティクス
- 実験の代替や開発コストの低減
  - 自動車の衝突シミュレーション
  - 航空機設計





第一原理的手法を使用すれば，実験不可能なことでも，シミュレーションによって解明される，であろうことが明らかになりつつある。
•バイオ，ナノテクノロジー
•現在の計算機リソースでは不可能なものも多い

コンピュータを速くするには、…

- 1秒間あたりの演算能力（足し算）
- MFLOPS: Millions of FLoating Point OPerationS.（1秒間に10^6回の浮動小数点処理）
- GFLOPS：10^9回，TFLOPS：10^{12}回，PFLOPS：10^{15}回

- ①　動作を速くする。
  - クロックを速くする
  （PCのプロセッサは2～3GHzの周波数）
  - 速いトランジスタ（回路）をつかう

©二宮知子

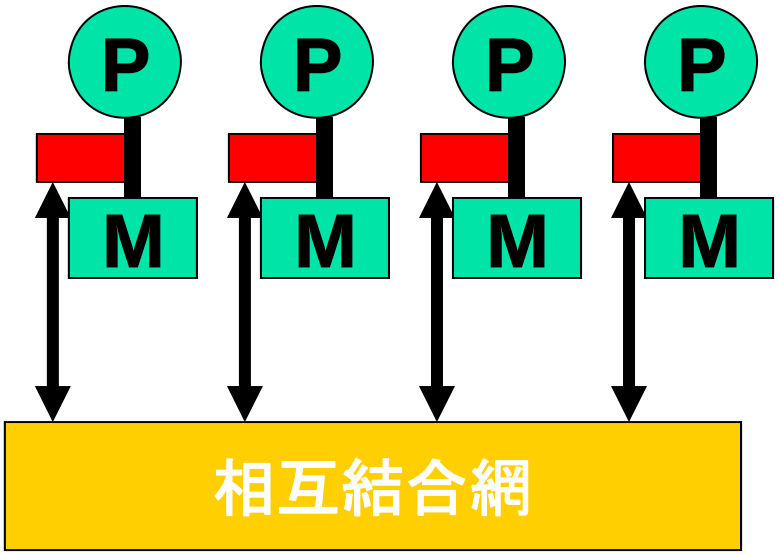
コンピュータを速くするには、…

- ②　コンピュータの中を工夫する
  - 一度に、たくさんの命令を実行できるようにする、など

- ③　たくさんのコンピュータを同時に使う。
  - 並列コンピュータ
  - 今のスパコンの主流はこれ！

  - PCでも、スマホでも2，3個のコンピュータがはいっている

分散メモリ型並列計算機



任意のプロセッサ間で
メッセージを送受信

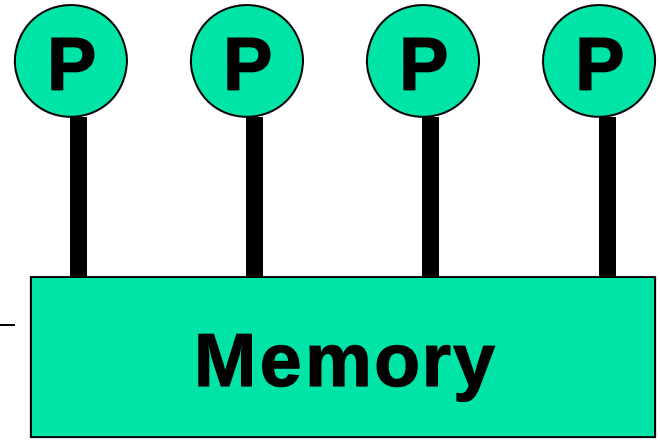
P ... Processor
M ... Memory
NIC (network interface controller)

●CPUとメモリという一つの計算機システムが、ネットワークで結合されているシステム

●それぞれの計算機で実行されているプログラムはネットワークを通じて、データ(メッセージ)を交換し、動作する

●比較的簡単に構築可能・拡張性(scalability) が高い
- ◆超並列計算機 (MPP: Massively Parallel Processor)
- ◆クラスタ型計算機

共有メモリ型計算機



複数のプロセッサからの同時アクセスを整理することが必要

●複数のCPUが一つのメモリにアクセスするシステム

●それぞれのCPUで実行されているプログラム（スレッド）は、メモリ上のデータにお互いにアクセスすることで、データを交換し、動作する

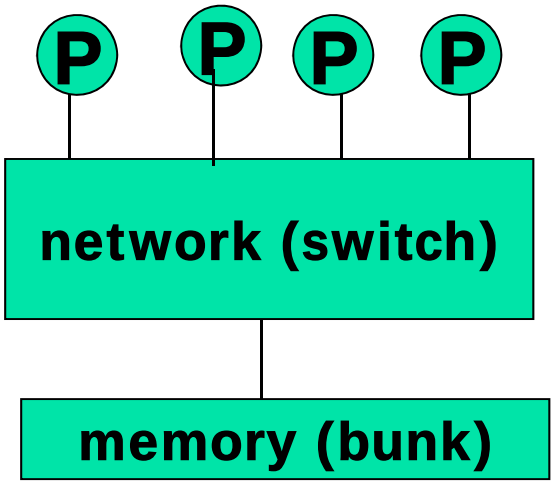
●大規模サーバ

●最近はプロセッサ１台が複数のプロセッサコアの共有メモリシステムになっている

●アーキテクチャ的にはさらにSMPとNUMAに分かれる（後述）

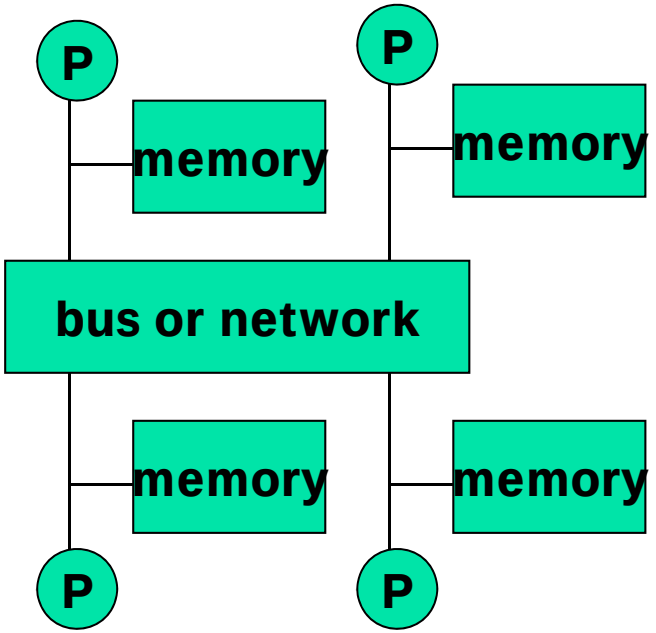
共有メモリアーキテクチャ:SMP

- SMP (Symmetric Multi-Processor)
  - 各プロセッサから見てどのmemory moduleへの距離も等しい
  - 構成としては、複数のプロセッサが共通のバスまたはスイッチを経由して、等しくmemory module(群)に接続されている
  - コモディティスカラプロセッサとしては、Intelプロセッサがこの方式
  - 大規模システムとしては富士通のHPC2500シリーズ、日立SR16000シリーズ等が該当する
  - coherent cacheとの併用が一般的
  - どのプロセッサからもデータが等距離にあるので偏りを心配しなくてよい
  - トラフィックが集中した場合に性能低下を防げない

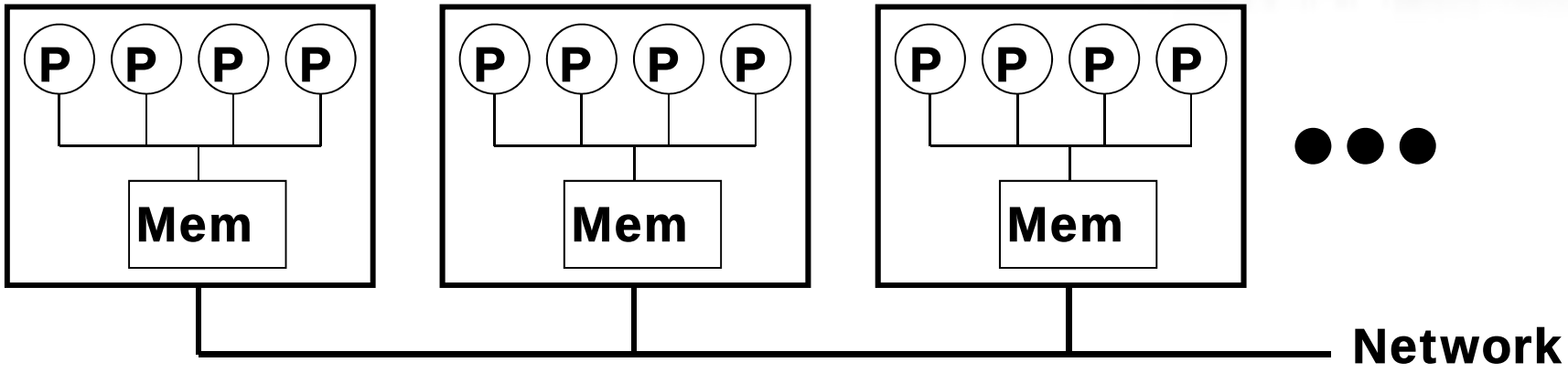


NUMA共有メモリアーキテクチャ

- NUMA (Non-Uniformed Memory Access)
  - CPUに付随して固有のmemory moduleがある
  - 共有バスまたはスイッチを介して、他のCPUのmemory moduleも直接アクセス可能
  - 遠距離memory moduleへのアクセスには時間が余計にかかる（non-symmetric）
  - コモディティスカラプロセッサとしてはAMD (Opteron)がこの方式
    ⇒ 最近、Intelも同様のアーキテクチャになった (Nehalem)
  - 大規模システムとしてはSGI Origin, Altixシリーズ等が該当
  - データをうまく分散し、参照の局所性が生かせれば性能を大幅に向上可能
  - 遠距離アクセス時の遅延時間増加に注意



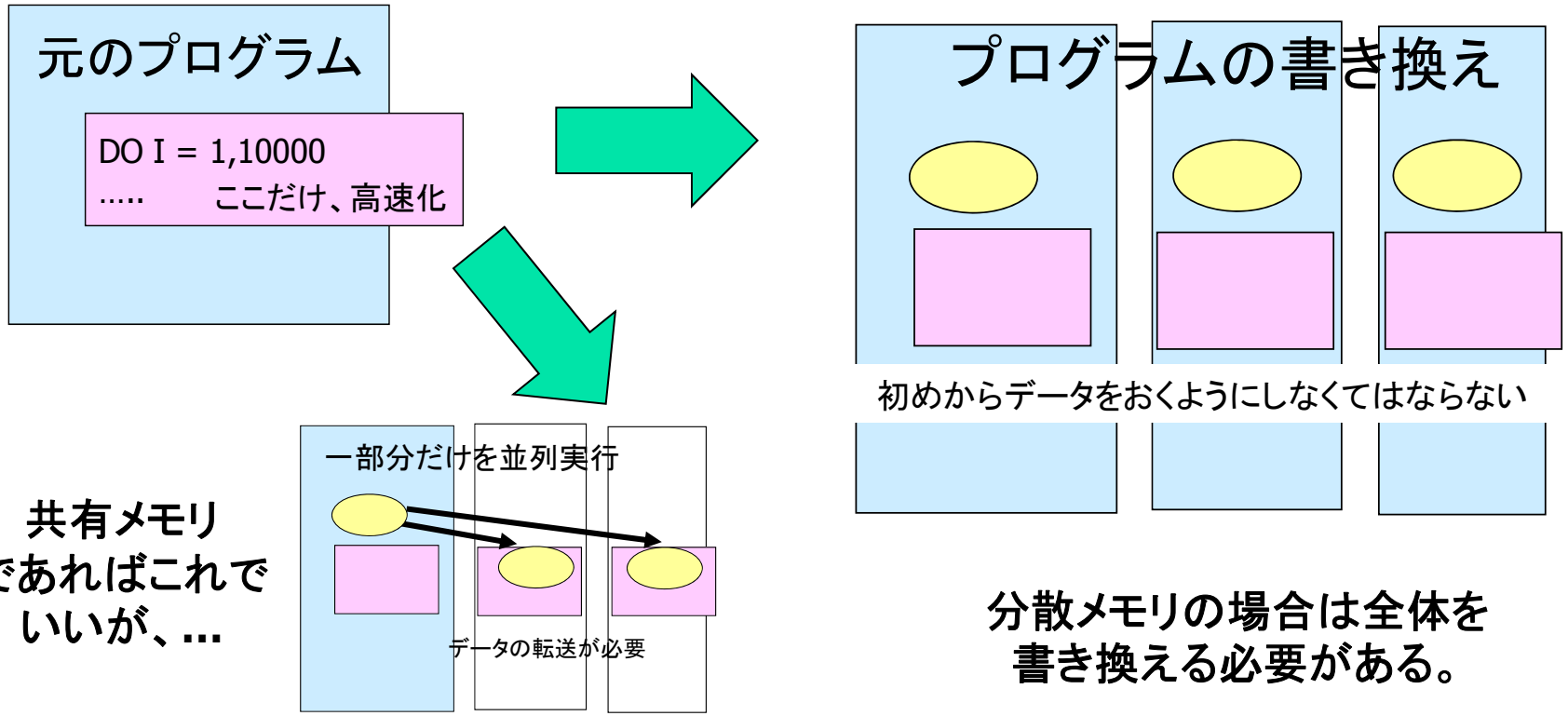
分散／共有メモリ・ハイブリッド



- 共有メモリと分散メモリの組み合わせ
- 分散メモリ型システムの各ノードがそれ自身共有メモリアーキテクチャになっている (SMP or NUMA)
- マイクロプロセッサ自体が1チップで共有メモリ構成（マルチコア）となっていることが大きな要因、近年のマルチコアプロセッサ普及により急激に主流となった

並列化と並列プログラミング

- 並列化：逐次プログラムを並列実行できるようにすること
  - 並列化の目的は高速化！
  - 並列処理の対象となる科学技術計算では、繰り返し文で書かれた部分が計算の大部分を占めるといわれる。
    - コードの全体の5%が、実行時間の95%を占めるともいわれる。この部分を見つけ、並列化することがまずは重要。

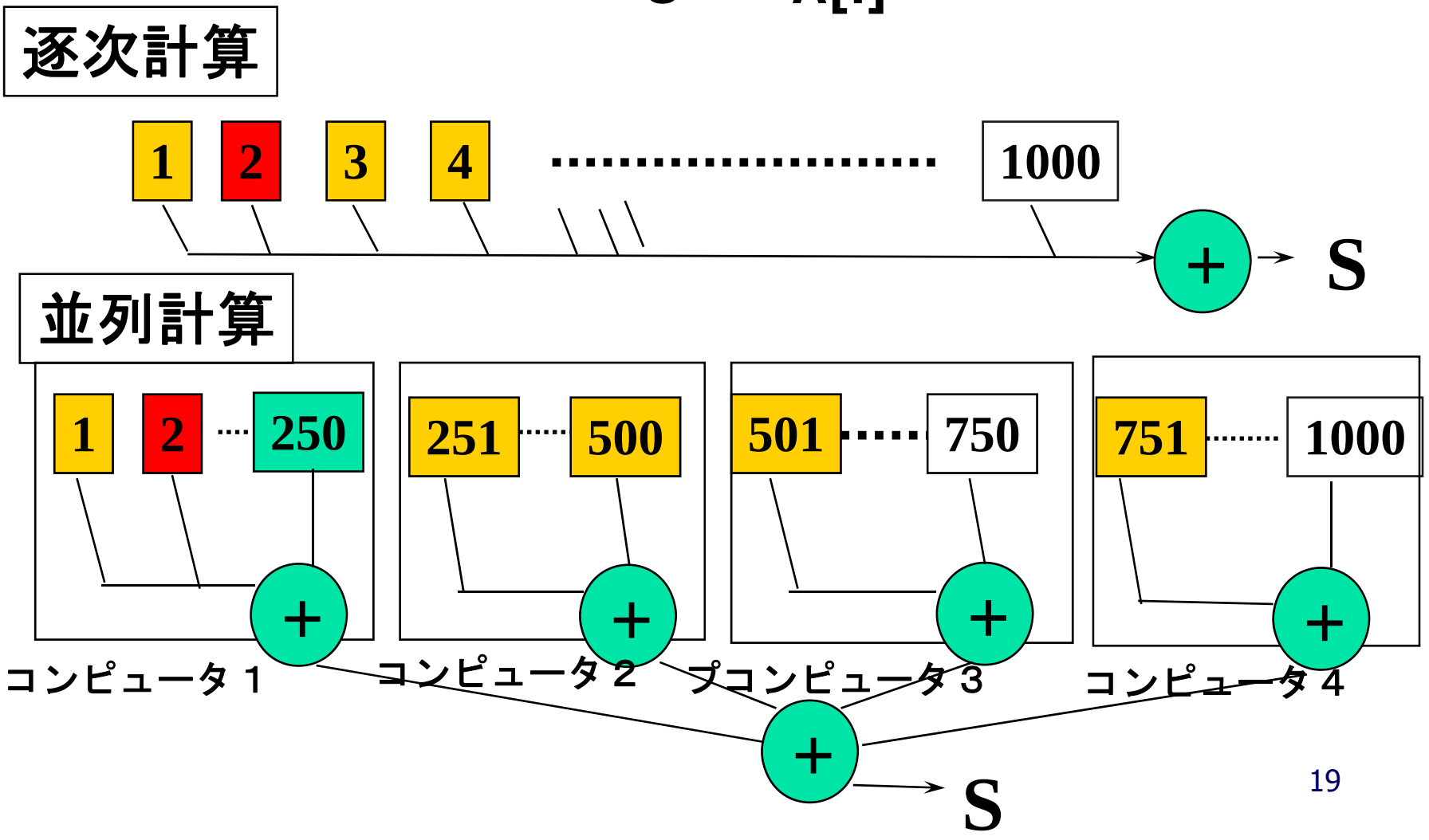


並列プログラミング・モデル

- メッセージ通信 (Message Passing)
  - メッセージのやり取りでやり取りをして、プログラムする
  - 分散メモリシステム（共有メモリでも、可）
  - プログラミングが面倒、難しい
  - プログラマがデータの移動を制御
  - プロセッサ数に対してスケーラブル
- 共有メモリ (shared memory)
  - 共通にアクセスできるメモリを解して、データのやり取り
  - 共有メモリシステム（DSMシステムon分散メモリ）
  - プログラミングしやすい（逐次プログラムから）
  - システムがデータの移動を行ってくれる
  - プロセッサ数に対してスケーラブルではないことが多い。

並列処理の簡単な例

```
for(i=0;i<1000; i++)
    S += A[i]
```



POSIXスレッドによるプログラミング

- スレッドの生成

Pthread, Solaris thread

```
for(t=1;t<n_thd;t++){
  r=pthread_create(thd_main,t)
}
thd_main(0);
for(t=1; t<n_thd;t++)
       pthread_join();
```

スレッド＝
プログラム実行の流れ

- ループの担当部分の分割
- 足し合わせの同期

```
int s;  /* global */
int n_thd; /* number of threads */
int thd_main(int id)
{ int c,b,e,i,ss;
  c=1000/n_thd;
  b=c*id;
  e=s+c;
  ss=0;
  for(i=b; i<e; i++) ss += a[i];
  pthread_lock();
  s += ss;
  pthread_unlock();
  return s;
}
```

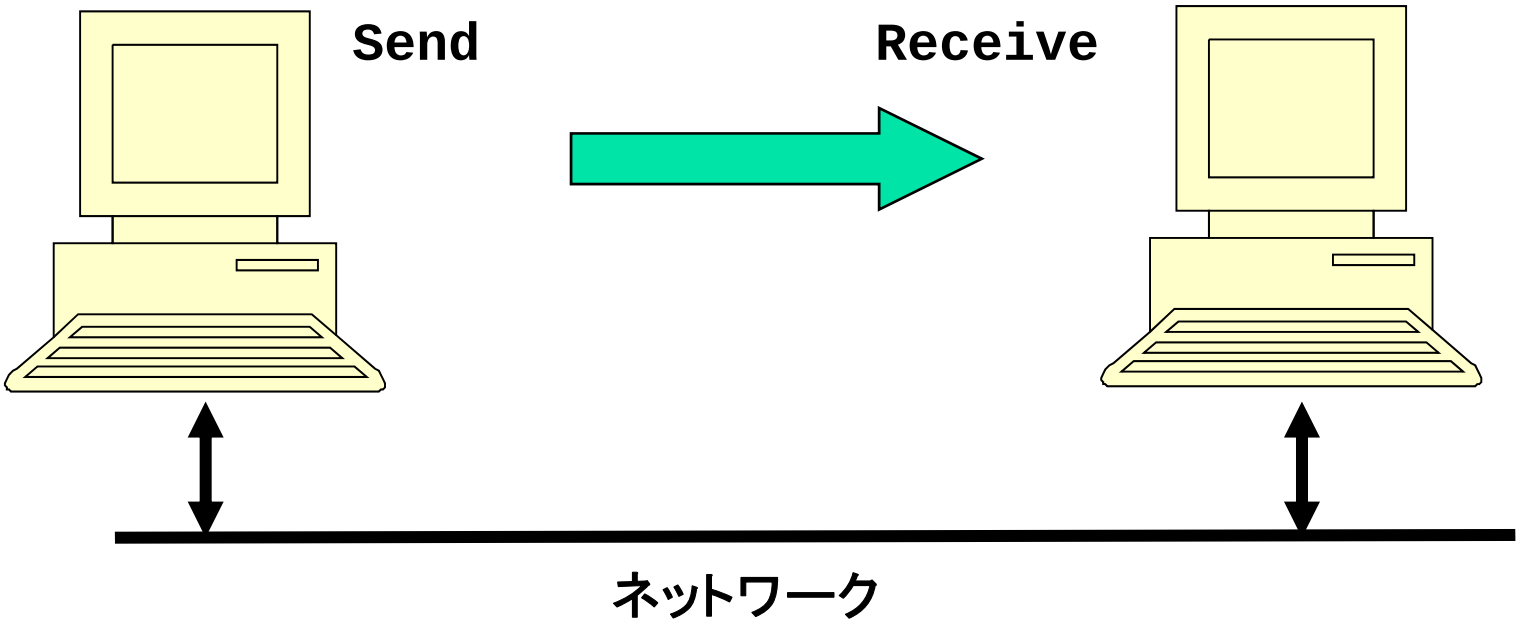
OpenMPによるプログラミング

これだけで、OK!

```
#pragma omp parallel for reduction(+:s)
      for(i=0; i<1000;i++) s+= a[i];
```

メッセージ通信プログラミング(1)

- sendとreceiveでデータ交換をする
  - MPI (Message Passing Interface)
  - PVM (Parallel Virtual Machine)



メッセージ通信プログラミング(2)

- 1000個のデータの加算の例

```
int a[250]; /* それぞれ、250個づづデータを持つ */

main(){     /* それぞれのプロセッサで実行される */
    int i,s,ss;
    s=0;
    for(i=0; i<250;i++) s+= a[i]; /*各プロセッサで計算*/
    if(myid == 0){    /* プロセッサ0の場合 */
        for(proc=1;proc<4; proc++){
            recv(&ss,proc);  /*各プロセッサからデータを受け取る*/
            s+=ss;    /*集計する*/
        }
    } else {   /* 0以外のプロセッサの場合 */
        send(s,0);  /* プロセッサ0にデータを送る */
    }
}
```

MPIによるプログラミング

- MPI (Message Passing Interface)
- 現在、分散メモリシステムにおける標準的なプログラミングライブラリ
  - 100ノード以上では必須
  - 面倒だが、性能は出る
    - アセンブラでプログラミングと同じ
- メッセージをやり取りして通信を行う
  - Send/Receive
- 集団通信もある
  - Reduce/Bcast
  - Gather/Scatter

MPIでプログラミングしてみると


```
#include "mpi.h"
#include <stdio.h>
#define MY_TAG 100
double A[1000/N_PE];
int main( int argc, char *argv[])
{
    int n, myid, numprocs, i;
    double sum, x;
    int  namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    fprintf(stderr,"Process %d on %s¥n", myid, processor_name);

    ....
```

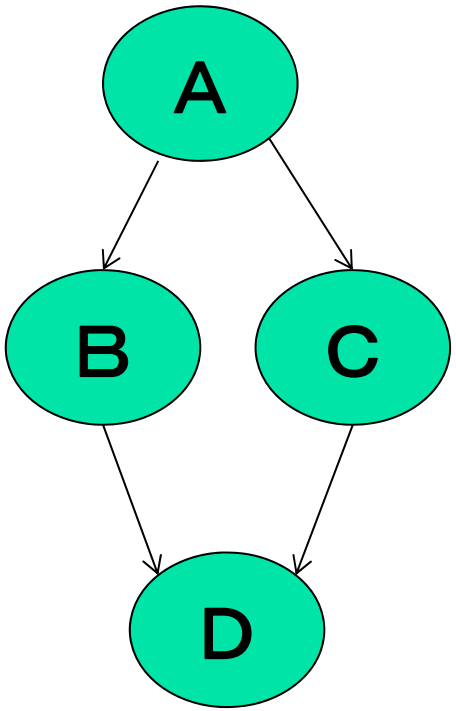
MPIでプログラミングしてみると


```
    sum = 0.0;
    for (i = 0; i < 1000/N_PE; i++){
       sum+ = A[i];
    }

    if(myid == 0){
       for(i = 1; i < numprocs; i++){
          MPI_Recv(&t,1,MPI_DOUBLE,i,MY_TAG,MPI_COMM_WORLD,&status)
            sum += t;
        }
    } else
            MPI_Send(&t,1,MPI_DOUBLE,0,MY_TAG,MPI_COMM_WORLD);
    /* MPI_Reduce(&sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0,
       MPI_COMM_WORLD); */
     MPI_Barrier(MPI_COMM_WORLD);
    ...
    MPI_Finalize();
    return 0;
}
```

並列化のパターン：タスク、粗粒度、細粒度

- 並列化する場合には、プログラムの処理を並列に実行する部分に分ける。この部分のことをタスクと呼ぶ。
- ある計算部分のタスクAの実行が終わらなければ、計算部分のタスクBが始められない場合に、Bは、Aに「依存」しているという。
  - タスクグラフ
  - 特に、Aの計算したデータをBの計算に用いる場合には、データ依存

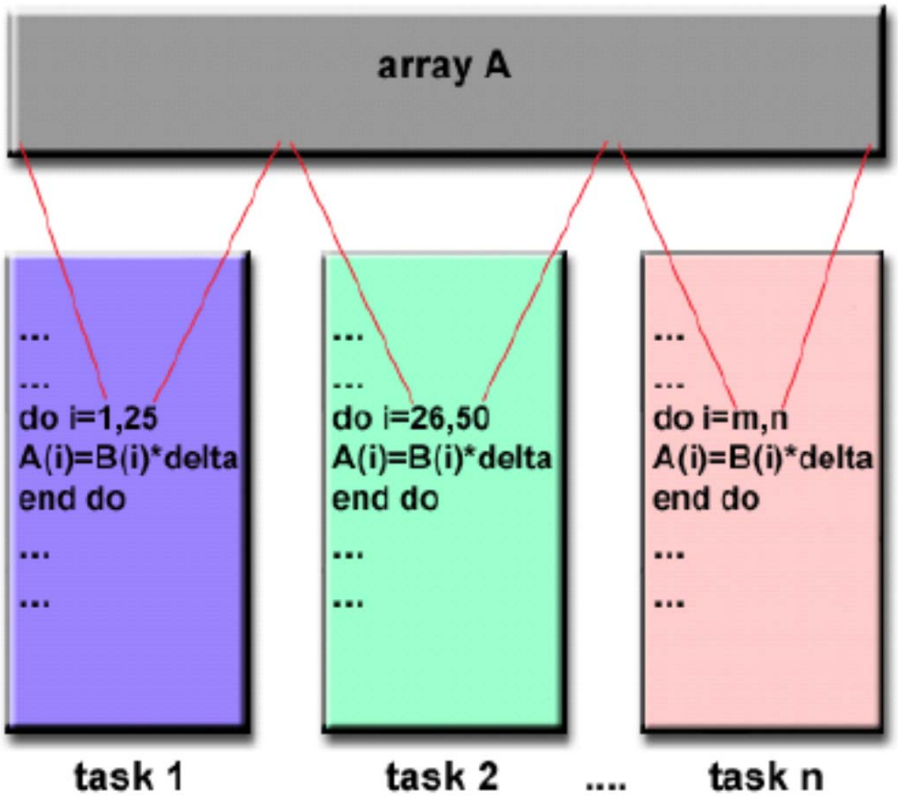


- タスクの実行時間がある程度長い場合には、粗粒度並列処理（Coarse-grain parallel processing）
  - タスクとして、関数ごとに分けたりする場合

- 数命令ごとのタスクに分けて並列処理する場合は細粒度並列処理（find-grain parallel processing）という。

並列化のパターン：データ並列

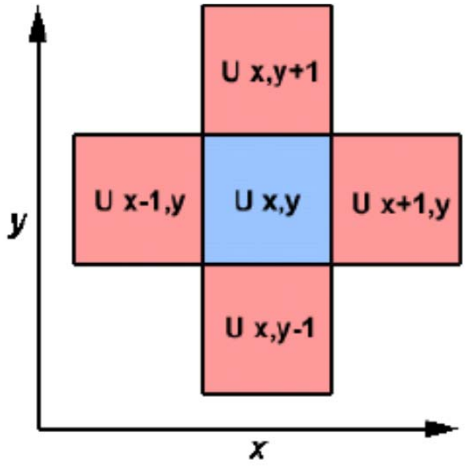
- データ並列：並列化の対象となることが多いのはループ
  - 繰り返し部分がそれぞれ独立に行える場合には各繰り返しを適当に分割して並列に実行できる。
  - ある配列の要素に同じ計算を行うループの場合はそれぞれの繰り返しに依存はない。このような並列計算のことをデータ並列計算という。

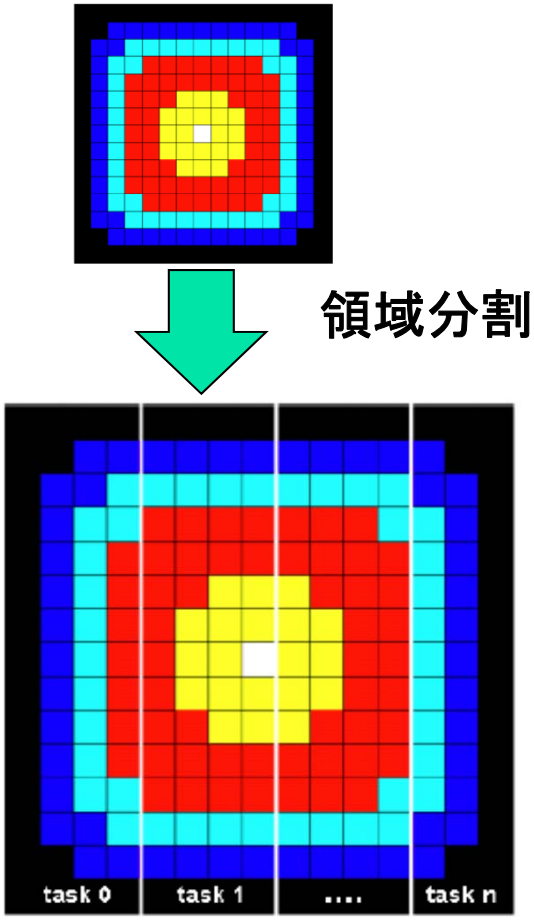


並列化のパターン：データ並列

- データ並列：領域分割(Domain decomposition)とは、解くべき問題に対応したデータを分割し、それぞれをタスクとする並列化
  - 簡単な熱拡散方程式を離散化し、陽解法で解くプログラムの１部分

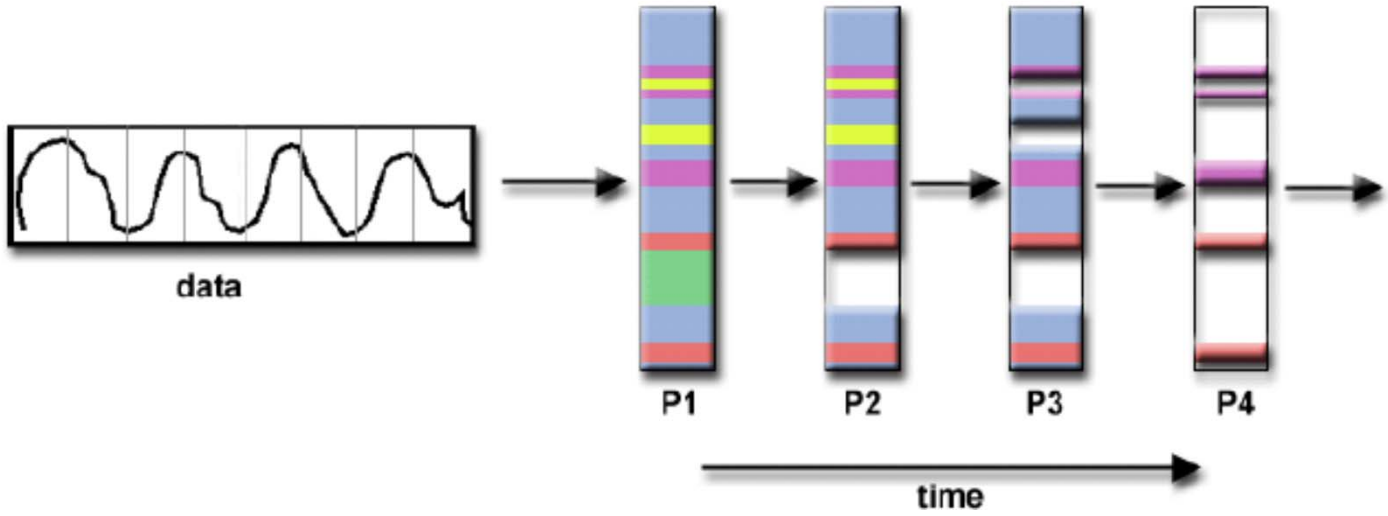
```
do iy = 2, ny-1
do ix = 2, nx-1
  u2(ix,iy) = u1(ix,iy)+
    cx*(u1(ix+1),y)+u1(ix+1,iy)-2*u1(ix,iy))+
    cy*(u1(ix,iy+1)+u1(ix,iy-1)-2*(ix,iy))
end do
end do
```





並列化のパターン：タスク並列

- タスク並列：機能ごとに別のタスクとして実行するモデル
  - 機能分割は、機能ごとに分けた処理を並列に実行する場合
  - 例えば、大気海洋についてのシミュレーションを行う場合には、機能ごとに、大気をシミュレーションと海洋のシミュレーションについて、別々の解像度を用いて行うことが考えらえる。適当なステップでこれらの機能部分でデータを交換しながらシミュレーションが進んでいくことになる。
- パイプライン並列：機能分割の特別な形
  - 信号処理など。

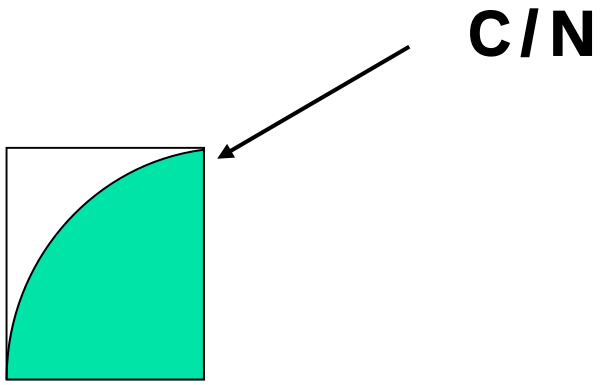


EP (Embarassingly Parallel)

- EP (Embarassingly Parallel)：並列化が自明なもの
  - parameter search：同じプログラムを多数の異なる入力パラメータで実行して統計値を得る
    ⇒1セットのパラメータを1つのプロセスで行い、それを管理・統合するプロセスを置く

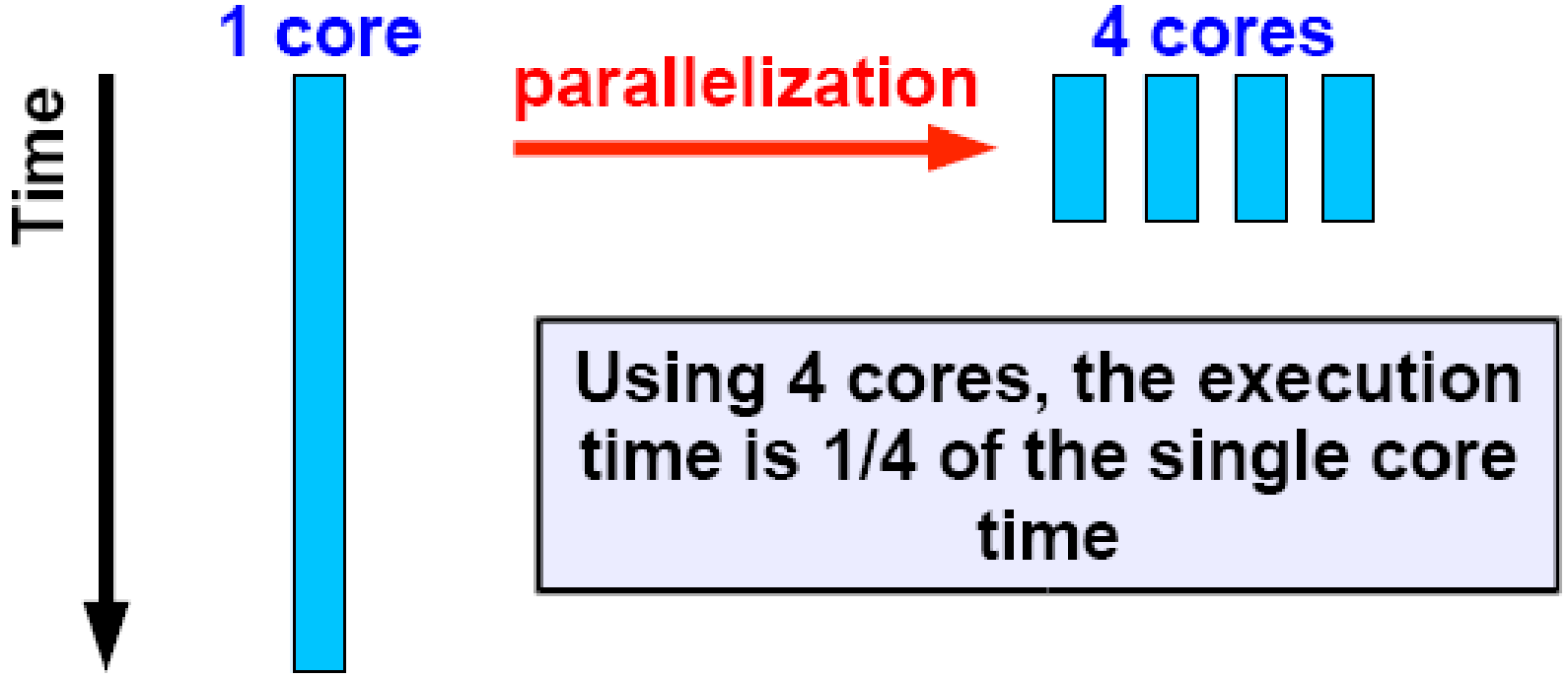
- モンテカルロ・シミュレーション
  - 多数のケースをランダムパラメータによって試行し、それぞれの結果を統計処理して最終的な結果を求める
  - 例：1/4単位円を用いたπの計算
    各(x,y)対に対する単位円内点かどうかの判定は完全に独立に処理可能
    ⇒独立処理（完全並列化）が可能
  - 一番最後にCの総和を求めればよい

N個の(x, y)対 $(0 \le x \le 1, 0 \le y \le 1)$、

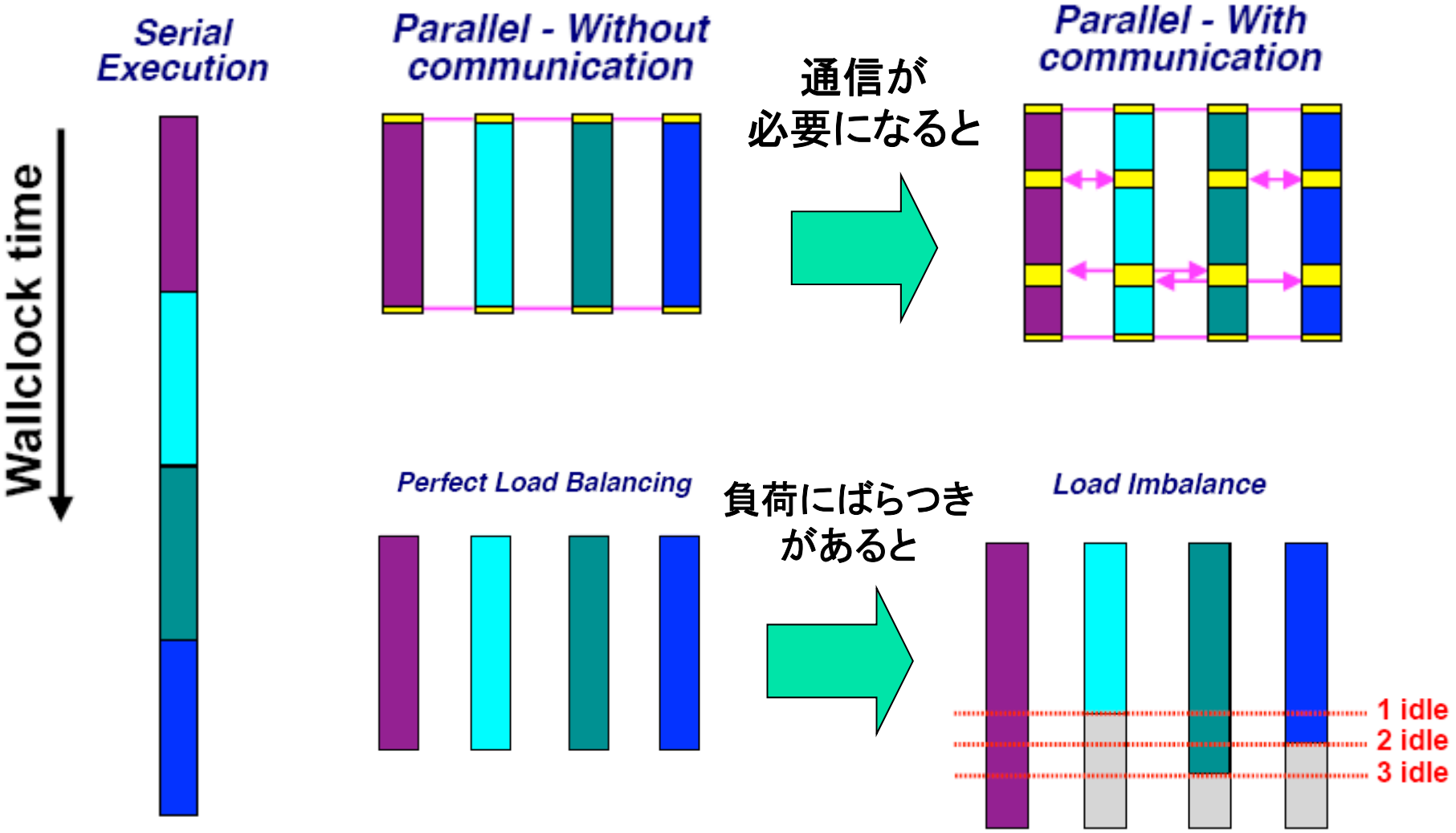


$x^2 + y^2 < 1$を満たす組の数をCとすると、$\frac{C}{N}$は$\frac{1}{4}\pi$に近づく。

なぜ、並列化するのか？
4つのコアがあれば、4倍！



並列化のオーバヘッド


Serial Execution
Wallclock time
Parallel - Without communication
通信が必要になると
Parallel - With communication
Perfect Load Balancing
負荷にばらつきがあると
Load Imbalance
1 idle
2 idle
3 idle

master/worker型並列処理

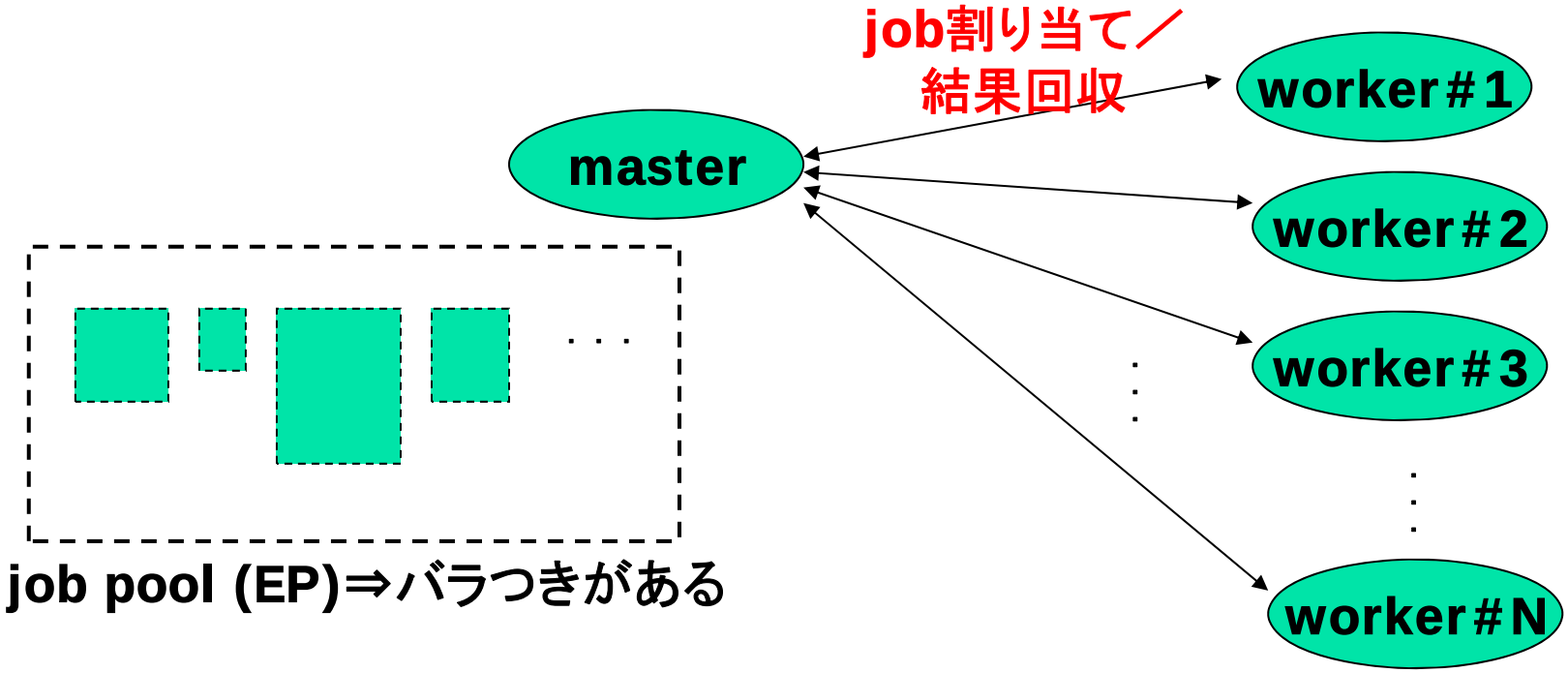
- 1つのmasterプロセスと複数のworkerプロセスがあり、masterが多数の独立な処理の「プール」を持つ（処理数>>worker数）
- masterはプールから処理すべき問題を取り出し、全workerに1つずつ与える
- workerは与えられた処理を行い、終了したら結果をmasterに返し、次の処理を割り当ててもらう

```
master::
// give a job to each worker
while(1){
    // receive a worker's result
    // give the next job to that worker
}
```

```
worker::
while(1){
        // receive a job from master
    // process the job
    // send the result to master
        }
```

master/worker（続き）

- 特に各処理の重さが異なり、負荷分散が難しい場合に有効
- 各処理は基本的にEPである必要がある

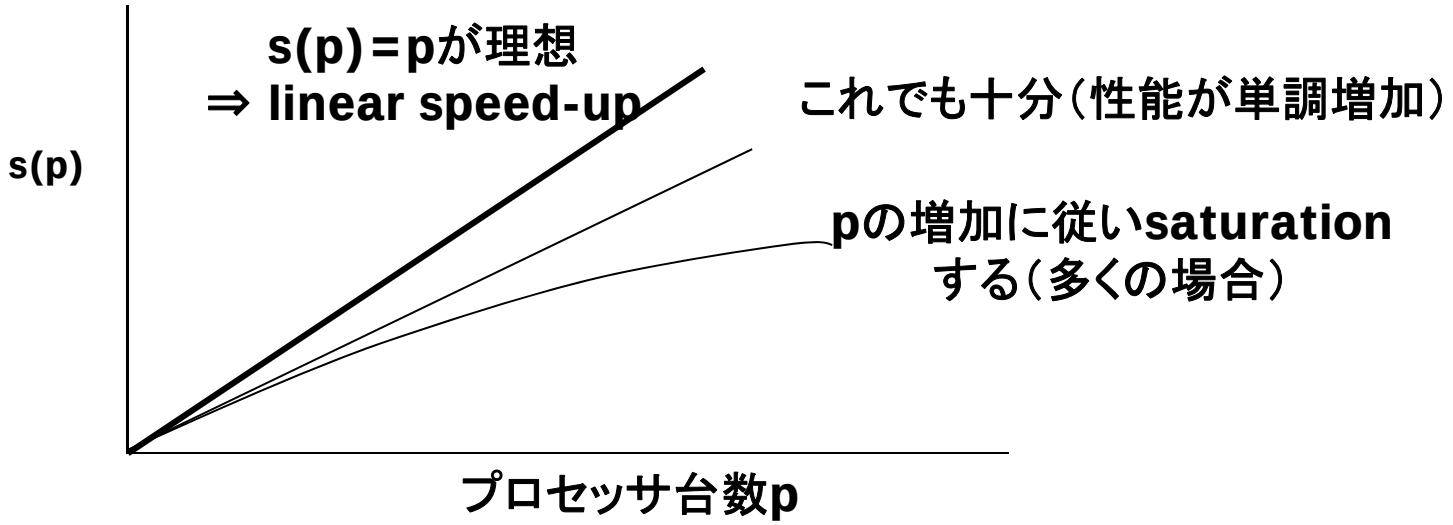


並列処理の性能メトリック

- 並列処理の目的は速度、問題規模(メモリ＆ディスク)、精度等様々だが、最も本質的なのは速度向上
- 並列処理を行った結果、当然総演算処理時間が短縮されることが期待される．．．が、
- 実際の速度が思ったほど上がらないことがしばしば起こる
- 特にシステム規模(プロセッサ数)の増大が性能に結びつかない場合が大きな問題となる
  ⇒並列処理のscalability(拡張性)
- 並列処理による性能向上を正しく測定するメトリックが必要
- 並列度(degree of parallelism)の定義
  - 問題の持つ並列度:問題の中に並列処理可能部分がどれくらいあるか(並列性とも呼ぶ)
  - システムの持つ並列度:システムの並列リソース数(一般的にはプロセッサ数)

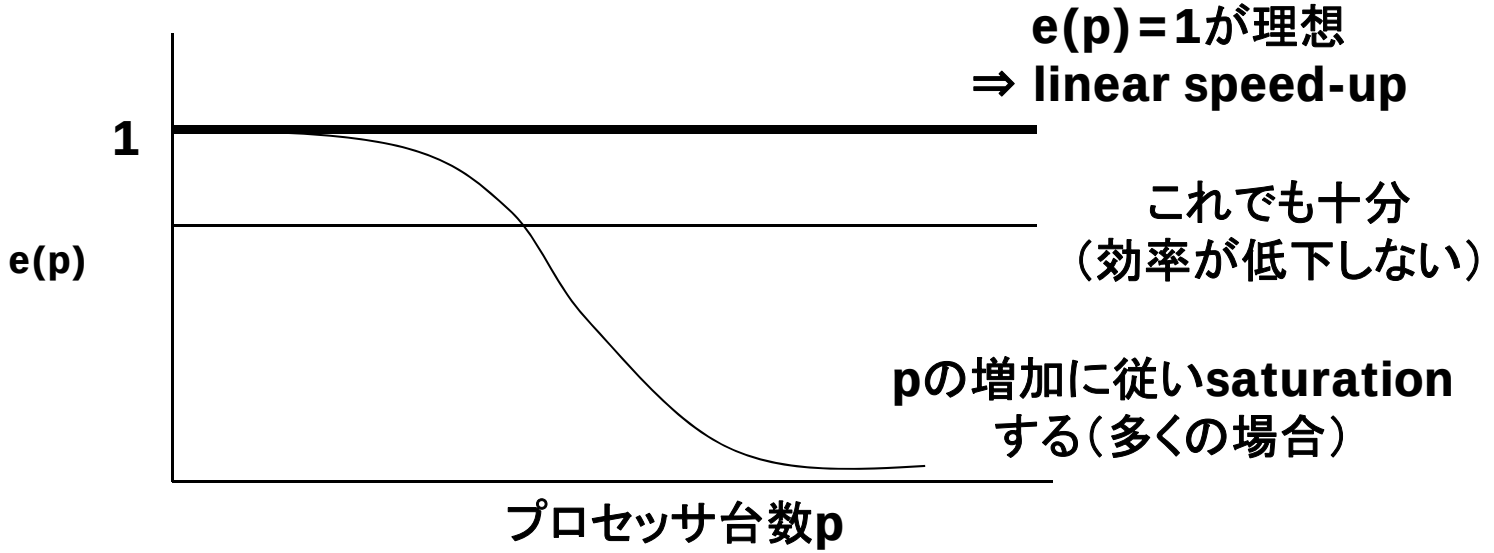
並列処理システムの性能指標（1）

- 速度向上率
  - 1プロセッサで実行した時の時間を*T*とする。
  - *p*プロセッサで実行した時の時間を*T(p)*とする。
  - *s(p)=T/T(p)*
    *s(p)*を「プロセッサ台数*p*台の速度向上率」と呼ぶ。*s(p)*が1以上であれば速度が上がったことになる。
  - 理想的には*s(p)=p*（*p*台のプロセッサを投入した結果、*p*倍の速度が得られた）



並列処理システムの性能指標(2)

- 並列化効率
  - 速度向上率*s(p)*は*p*に依存するので指標として不便
  - 「*s(p)=p*が理想的」ということに着目し、実際にはそれがどれくらい達成できたかを「効率」として考える
  - *e(p)=s(p)/p*
    *e(p)*は*p*に寄らず、1に近いほど理想的(通常は1以下)



アムダールの法則と並列処理効率

- アムダールの法則
  - 「処理効率はそれを構成する個々の要素の平均効率で決まるのではなく、一部の非効率部分によって律速される」
- 並列処理におけるアムダールの法則
  - 逐次処理における実行時間*T*が並列処理可能部分*TP*と並列処理不可能部分（逐次処理のみ可能）*TS*から成ると仮定

    $T=TP+TS$
  - *TP*部分について、*p*台のプロセッサで理想的な並列化ができるとすると、*p*プロセッサ投入時の実行時間*T(p)*は

    $T(p)=TS+TP/p$
  - プロセッサ台数*p*を無限大にすると

    $T(p, \lim p\to\infty) = TS$

    この時

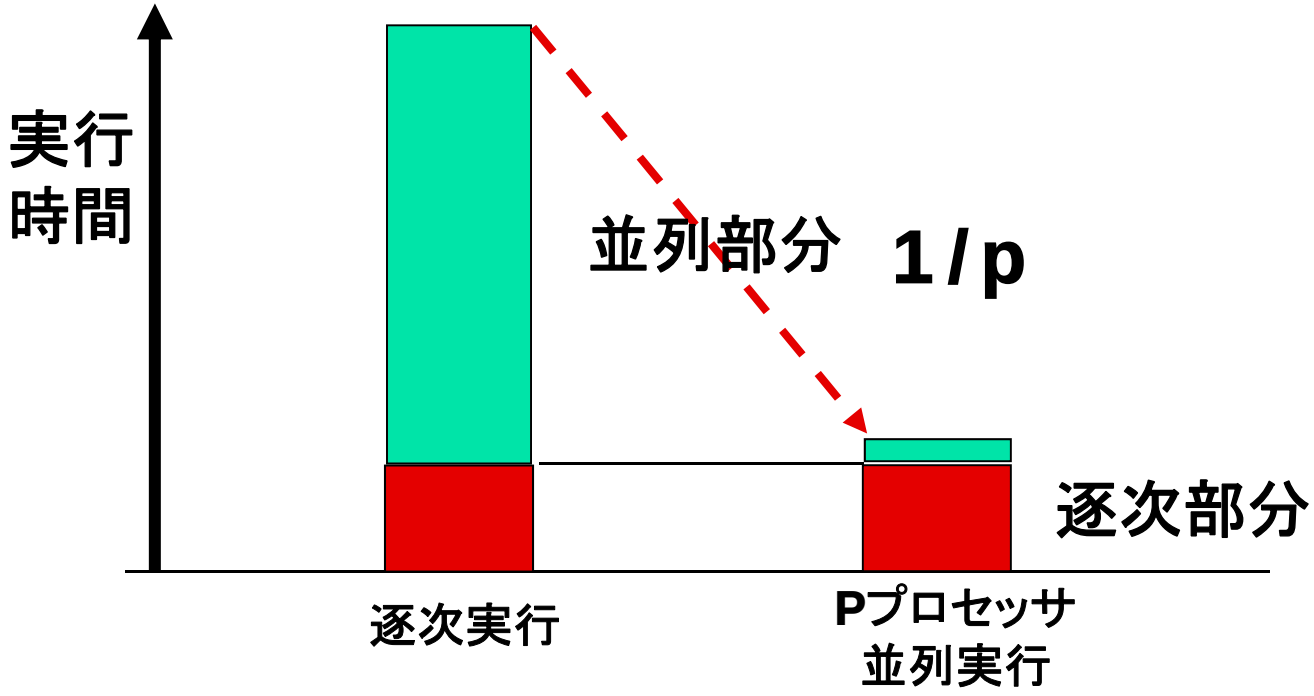
    $e(p) = s(p) / p = TS / p \ \ (p\to\infty) = 0$

    従って、

プロセッサ台数*p*をいくら増大しても、*TS*部分が律速が存在する限り並列処理効率の極限値は常に0になってしまう

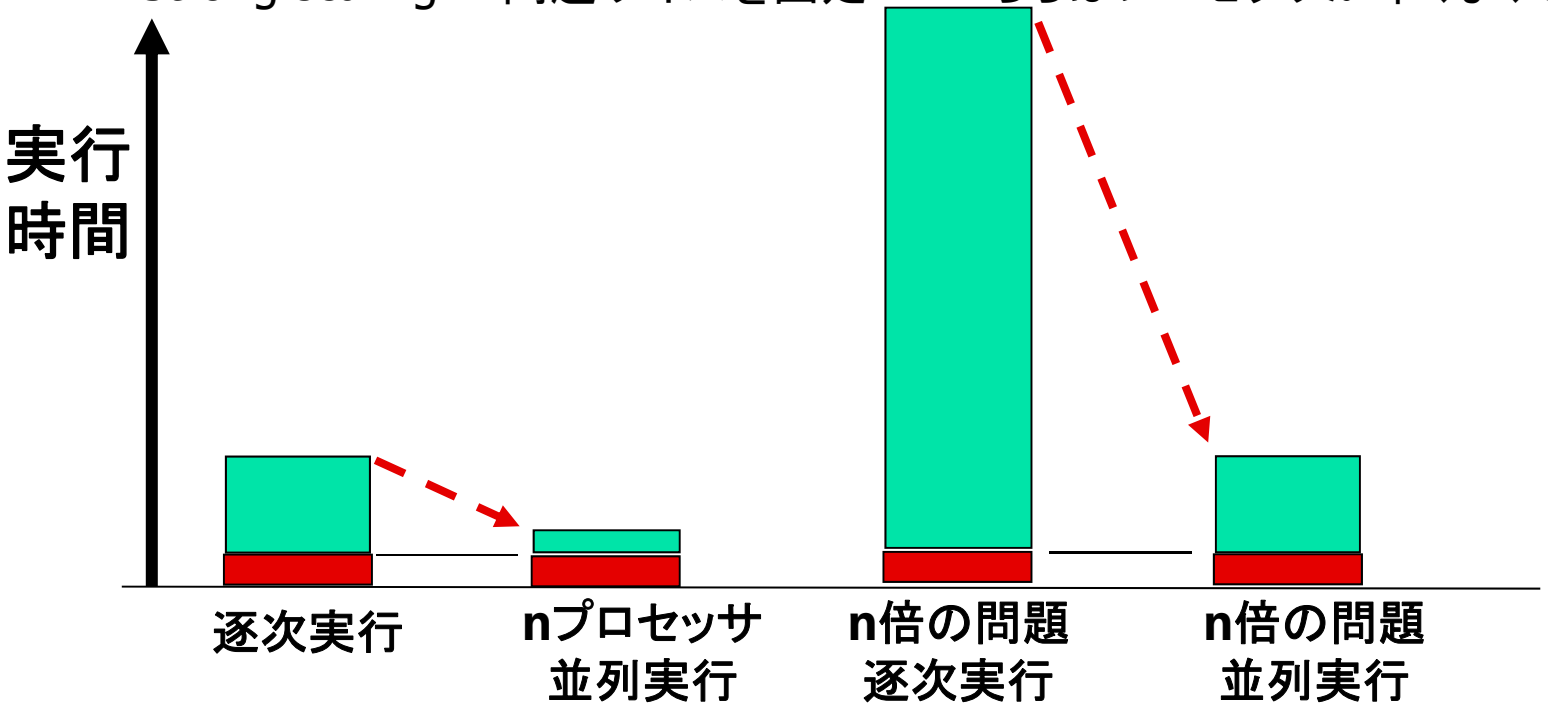
並列処理の問題点:「アムダールの法則」の呪縛

- アムダールの法則
  - 逐次処理での実行時間をT_1, 逐次で実行しなくてはならない部分の比率がαである場合、pプロセッサを用いて実行した時の実行時間(の下限)T_pは、$T_p = \alpha * T_1 + (1-\alpha) * T_1/p$
  - つまり、逐次で実行しなくてはならない部分が10%でもあると、何万プロセッサを使っても、高々10倍にしかならない。



並列処理の問題点:「アムダールの法則」の呪縛

- 「Gustafsonの法則」:では実際のアプリではどうか？
  - 並列部分は問題規模によることが多い
  - 例えば、ノード数nの場合、n倍の大きい問題を解けばよい。n倍の問題は、計算量がnになると、並列処理部分は一定
  - Weak scaling – プロセッサあたりの問題を固定←大規模化は可能
  - Strong scaling －問題サイズを固定　←こちらはプロセッサが早くなくてはならない。



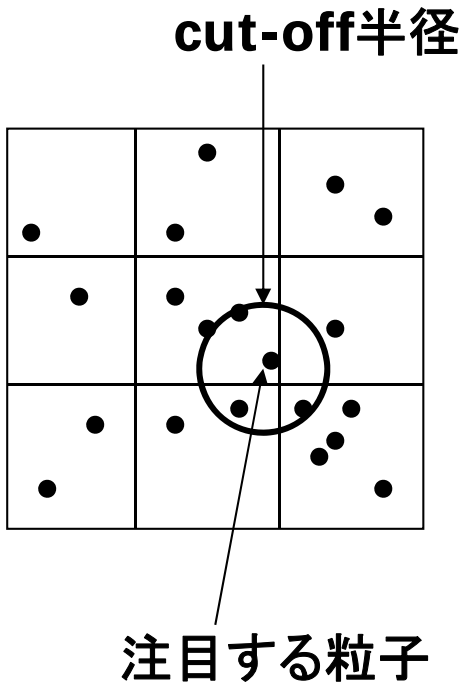
負荷バランスを考慮した問題分割

- domain decompositionでは、問題空間をなるべく粗く分割（隣接点を1つのプロセスに閉じ込める）することが理想
- 問題空間が不均質な場合、この方針では負荷バランスが崩れることがある
- 問題空間の形状を無視し、並列プロセス間の処理量が均等になるよう分割を変更することも必要
  ⇒ただし通信が近接でなくなることがあるので要注意！
  ⇒さらに処理粒度が低下する可能性もある

具体例：cut-off付きMD

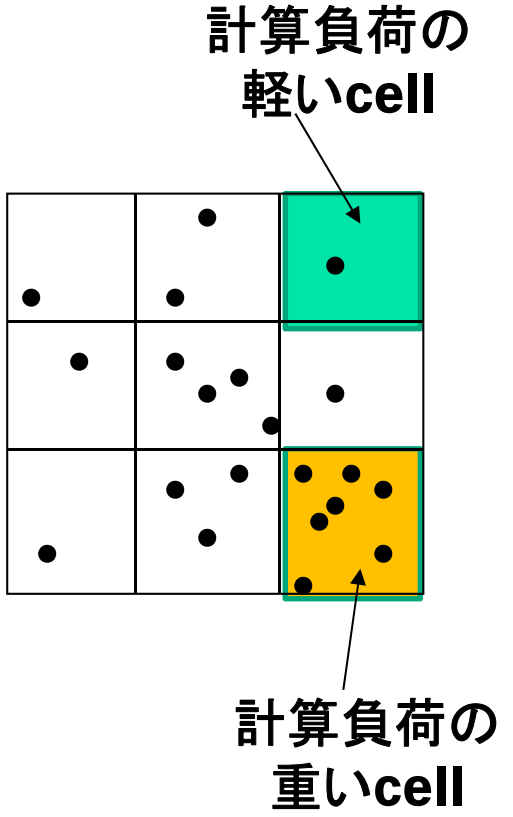
- MD (Molecular Dynamics)等の実例
  - n次元空間上にP個の粒子があり、粒子間力の相互作用をシミュレーションする
  - クーロン力のようななだらかなポテンシャルではなく、距離に応じて急激に縮小するポテンシャル（井戸型等）を持つため、力の影響する空間範囲にcut-off半径が存在する

- 「全対全」のinteractionではないため、通信コストを削減するためにデータ交換（通信）範囲をcut-off半径内の粒子（を持つプロセッサ）に限定したい
  - 空間をdomain decompositionし、プロセスに割り当てられた部分空間（cell）内の粒子を処理対象とする
    ⇒cellサイズをcut-off半径以上にすれば、隣接cellを受け持つプロセスとのみ通信すればよい（cell mapping method）



cut-off付きMD（続き）

- 粒子は他の粒子との力の相互作用により時間と共に移動するので、右図のように特定cellに固まる可能性がある
- cell単位で並列プロセスにマッピングすると負荷バランスが崩れる
- 負荷バランスを保つには、プロセス内の粒子数（cell数ではなく）をなるべく均一にする必要がある
- 方法
  - 1) 粒子数/cellの密度に応じ、プロセスに割り当てるcell数を一定時間毎に再調整する
  - 2) cell数がプロセス数より遥かに多い場合、cellをblockでプロセスにマップせず、cyclicにマップする
  - 3) cellマッピングを諦め、粒子単位での管理を行う



cut-off付きMD（続き）

- 方法1)
  cellとプロセスの割り当てを変更するには大量のデータを頻繁に移動しなければならない。また、cell間の通信は隣接するとは限らない（非定型形状）。
- 方法2)
  cyclic分割により比較的負荷分散が容易にできる。ただし、隣接cellが隣接プロセスにマップされなくなるため、通信距離が長くなる。
- 方法3)
  cell法を捨てるため、通信相手プロセスがどこにいるかを、毎回テーブル（粒子とプロセスの関係管理）引きによって求め、かつ通信距離も長くなる。

  ⇒決定的な方法はない
- 問題の特性（粒子の固まり易さ、ポテンシャルの性質等）に依存するため、一般解はない
- 極端な負荷非均衡が生じている場合、通信コストを犠牲にしても負荷バランスを保つ価値があるかもしれない

スパコンのハードウェアの歴史

- 1983年：1 GFLOPS，1996年：1 TFLOPS…

- 1990年以前は、特別なスパコン（ベクトル型）が主流
- 1990年代以降は、多数のコンピュータを結合した並列計算機が主流に。
- PCに使われているマイクロプロセッサ（1つのチップでできたコンピュータ）の急激な進歩
  - 1.5年に2倍の割合でトランジスタの集積度が増加（ムーアーの法則）
  - 4004(世界初、1971年、750KHz) 8008(1972年、500KHz、インテル) 8080(1974年、2MHz、インテル)
  - Pentium 4 (2000年、～3.2GHz)

  - 30年間で、1MHzから1GHz、1000倍の進歩

スパコンのハードウェアの歴史

- 2000年以降は、PCに使われてマイクロプロセッサを使ったが並列計算機（PCクラスタ）が主流に。
- 2008年にはIBM RoadRunner, 1Peta Flops を達成
- そして、「京」が世界1に！

計算機はどのくらい早くなったか

- 能力はどうやって計るのか？
  - １秒あたりの演算可能回数
  - Top500
- マイクロプロセッサの発展
  - クロックスピードにほぼ比例して早くなる
- スーパーコンピュータは並列処理の時代へ
  - 一つのコンピュータでは限界！
  - たくさんのコンピュータをつないで並列処理
  - スーパーコンピュータは並列処理により早くなっている

TOP 500 List　スパコン・ランキング
http://www.top500.org/

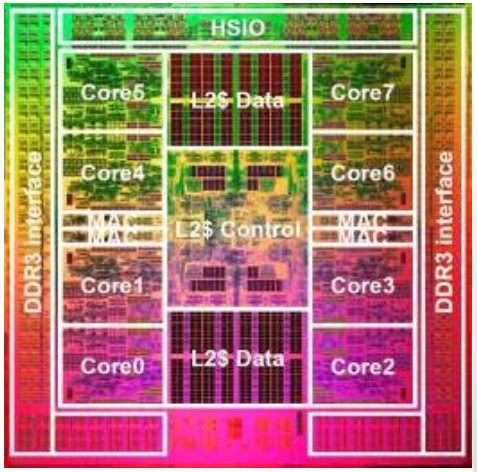
- LINPACKと言われるベンチマークプログラムの性能を性能の基準とする。
  - 超大規模な連立一次方程式を解く
  - 1千万次元の連立1次方程式
- 実際のアプリケーションの性能とは違う
  - 実際のアプリケーションではこれほどの性能は出ない
- 2008年から、電力消費量を表示するようになった
  - これからのスパコンは、電力が大切

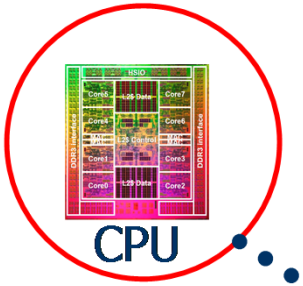
June / 2011

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect / 2011 Fujitsu	548352	8162.00	8773.63	9898.56
2	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C / 2010 NUDT	186368	2566.00	4701.00	4040.00
3	DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60
4	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU / 2010 Dawning	120640	1271.00	2984.30	2580.00
5	GSIC Center, Tokyo Institute of Technology Japan	TSUBAME 2.0 - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU Linux/Windows / 20 NEC/HP				
6	DOE/NNSA/LANL/SNL United States	Cielo - Cray XE6 8- Cray Inc.				
7	NASA/Ames Research Center/NAS United States	Pleiades - SGI Altix Xeon HT QC 3.0/Xe Infiniband / 2011 SGI				
8	DOE/SC/LBNL/NERSC United States	Hopper - Cray XE6 Cray Inc.				
9	Commissariat a l'Energie Atomique (CEA) France	Tera-100 - Bull bull S6010/S6030 / 201 Bull SA				
10	DOE/NNSA/LANL United States	Roadrunner - Blade Cluster, PowerXCe 1.8 GHz, Voltaire In IBM				

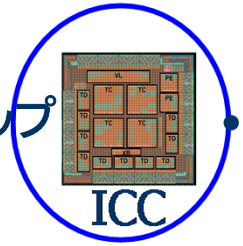
京コンピュータの構成

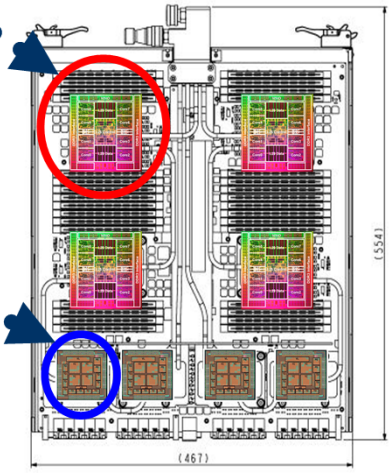
- 1つのチップに8個のコンピュータ(コア)
- 1つのコンピュータの性能は、16GFLOPS (2GHz), チップあたり、128GFLOPS
- PCとかわらない？

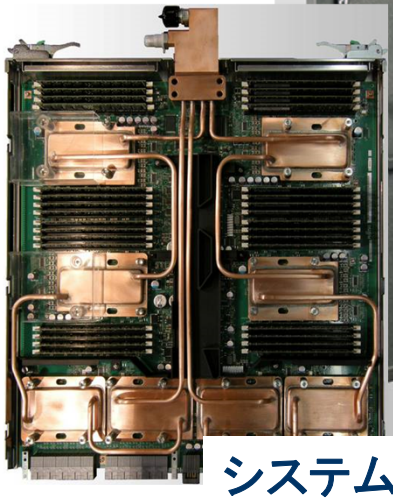




通信チップ





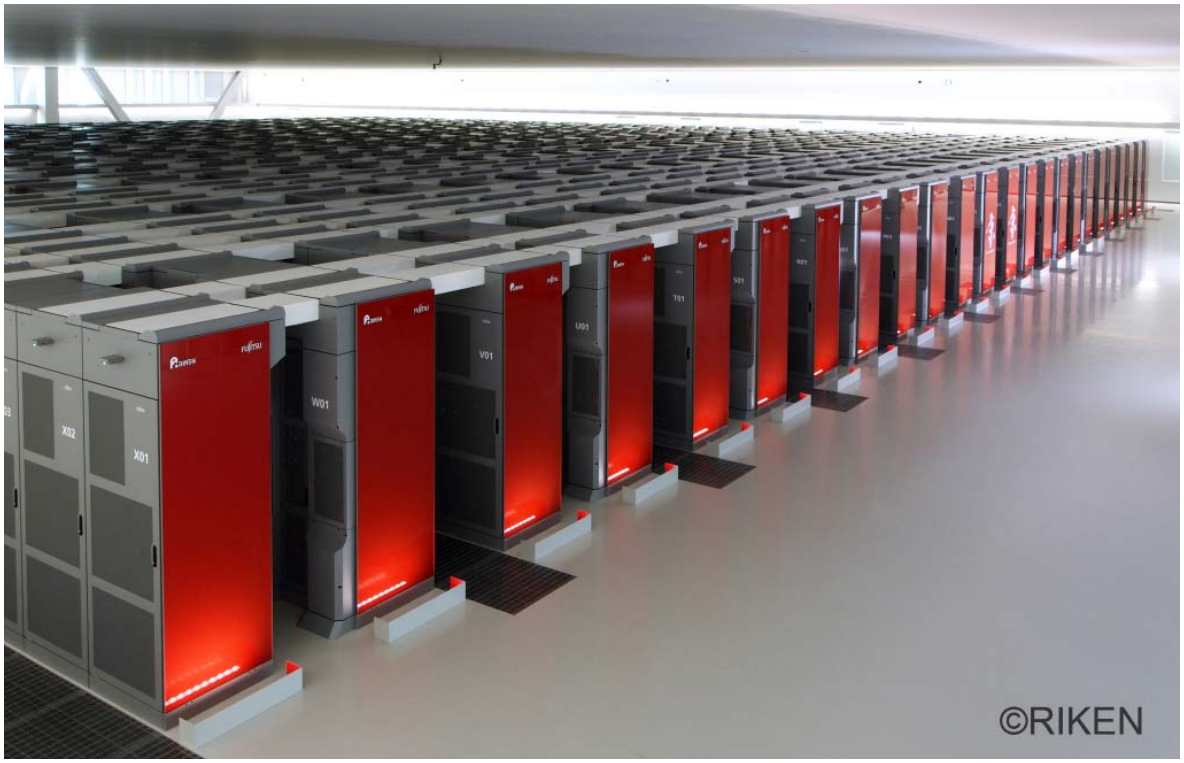


システムボード

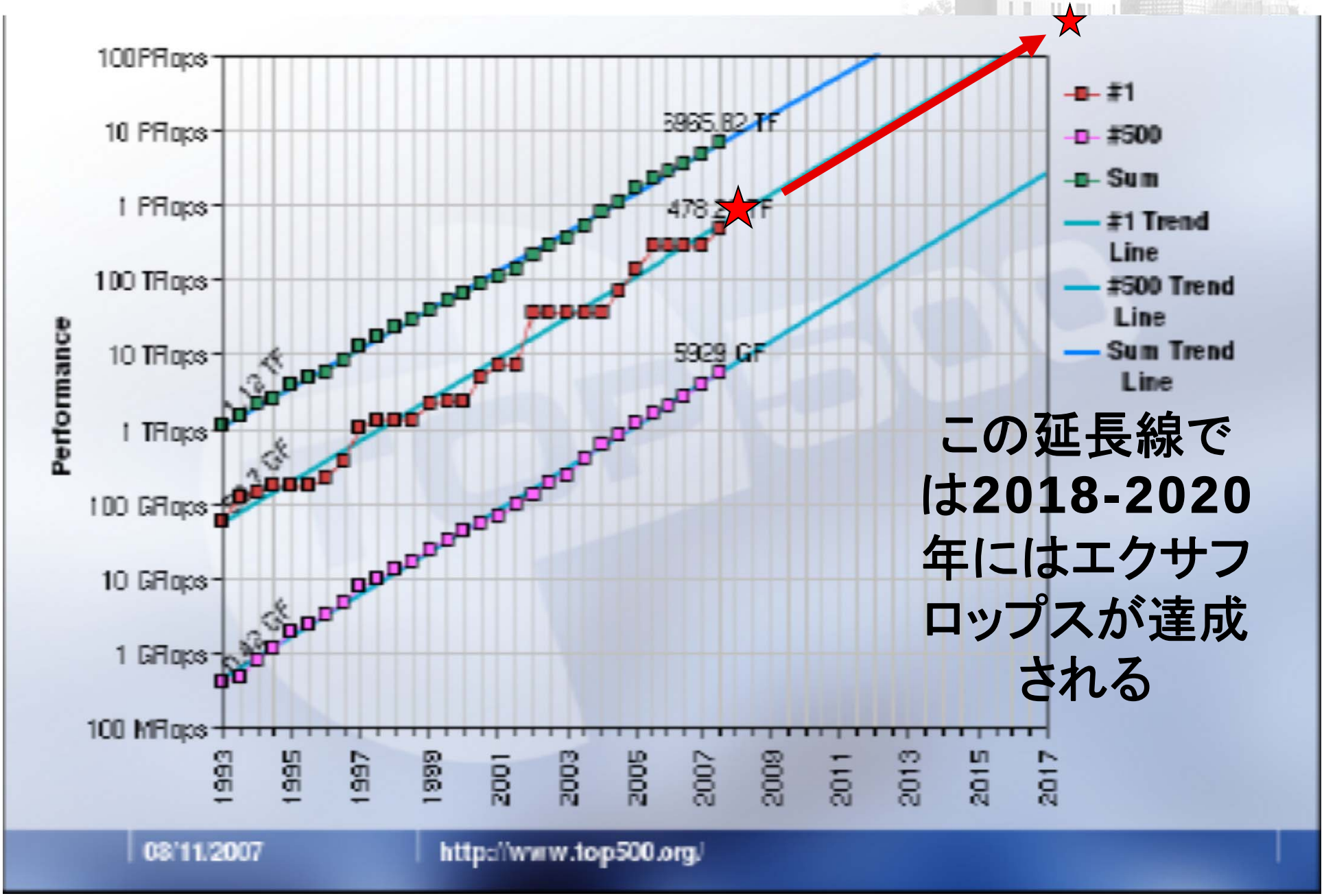
Courtesy of FUJITSU Ltd.

京コンピュータ　全体のデータ

- 筐体数　864
- チップ数:　82,944
- コンピュータ数:　663,552
- 性能　Linpack
  10.51PF
  (電力12.66MW)
  2011/11月



これからのトレンド



この延長線では2018-2020年にはエクサフロップスが達成される

いま、最先端のスパコンを作る時の問題は、...

- いまのスパコンの性能は、並列処理から
- すなわち、コンピュータ数
- ということは、性能は結合するコンピュータの数を増やせばいい
- が、電力が限界