筑波大学計算科学研究センター
CCS HPCセミナー
「並列処理の基礎」

朴　泰祐
taisuke@cs.tsukuba.ac.jp
筑波大学大学院システム情報工学研究科
計算科学研究センター

「並列処理の基礎」内容

- 演算性能と通信性能のメトリック
- 並列処理とは？
- 並列処理の必要性
- 各種並列化手法
- 通信・同期
- 並列化効率とアムダールの法則
- 負荷バランス

演算性能と通信性能のメトリック

- 演算性能（主として浮動小数点演算）
  - FLOP: (number of) Floating point Operations
    その処理の中の浮動小数点演算数
    ex) for(i=0; i<100; i++) a[i] = b[i] * c + d[i];
    ⇒ 200FLOP
  - FLOPS: Floating point Operations Per Second
    1秒当たりの浮動小数点演算処理性能
    ex) 上の問題を2μ秒で処理した場合 ⇒ 100 MFLOPS
- 通信性能
  - B/s (Byte/sec):
    1秒当たりに通信処理できるデータ量
    ex) Infiniband 4xQDRの理論ピーク性能＝4 GB/s
    bps (bit per second) で表示される場合もある
    システムによって、必ずしも 1Byte=8bit でないことがあるので注意

並列処理とは？

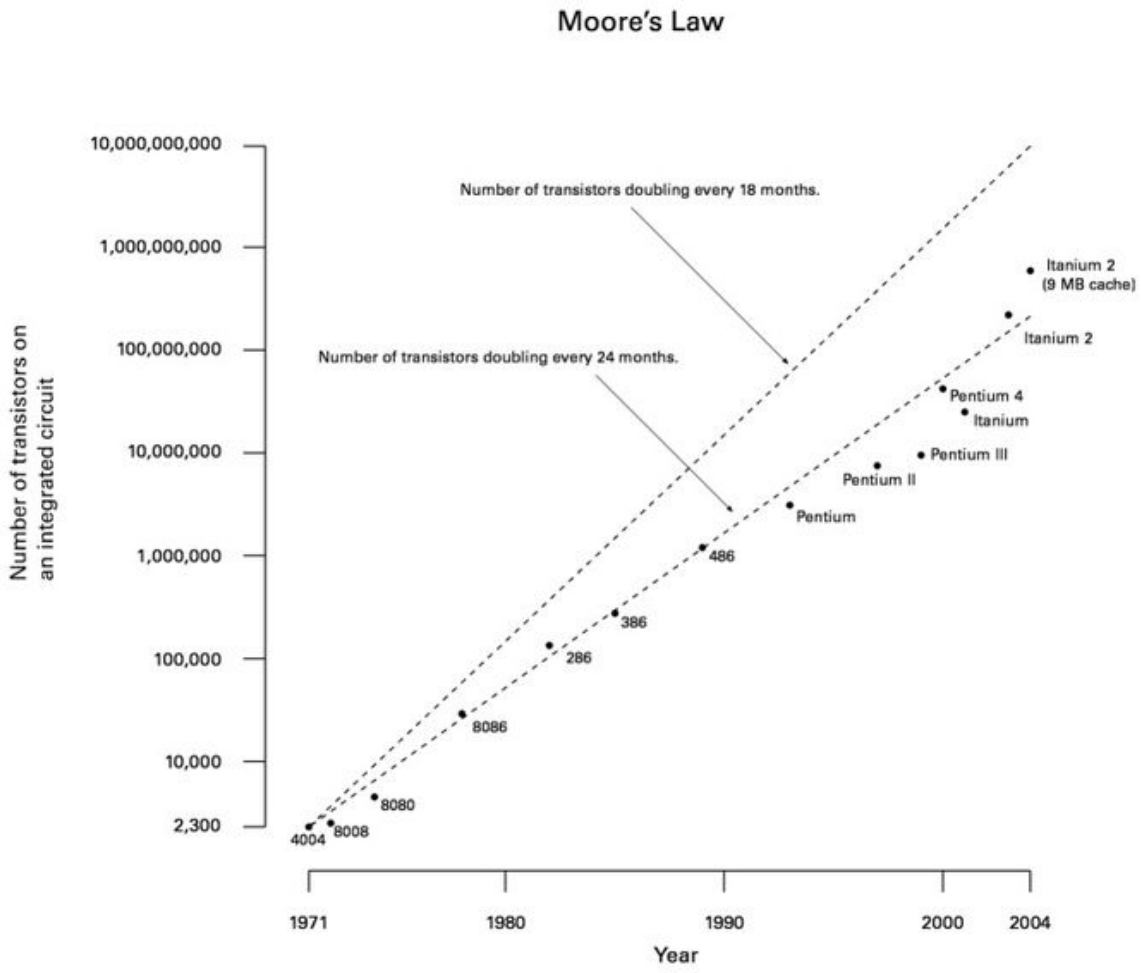
- 「*1つの問題を*何らかの方法で分割し、*複数の計算リソース*に割り当てることによって性能・規模等を向上させること」
  - 「1つの問題」を解く⇒多重処理とは違う
  - 「問題の分割」（並列化）⇒処理効率が高くなるように分割
  - 「向上」するもの⇒速度だけでなく、問題規模の増大、精度の向上等いろいろなメトリックがある
- parallel processing v.s. concurrent processing
  - 部分問題（並列化）されたものを「見かけ上同時実行」すること⇒concurrent processing(並行処理)
  - 部分問題を「物理的に同時実行」⇒parallel processing(並列処理)
- 並列処理に供されるリソース
  - CPU, memory, disk, network等、あらゆる計算リソースが並列化に貢献し得る
  - 複数のCPUに並列に割り当てられる処理単位を以後「並列プロセス」と定義

高性能計算と並列処理

- 数値シミュレーション等の科学技術計算、大量のデータ処理等において、問題規模の拡大要求は日増しに高まっている
- 問題規模を増大した時の処理量は多くの場合O(N)の増加では済まない
  - 3次元流体計算（気象予報等）
    空間の1次元をN点に分割して数値シミュレーションすると、3次元問題の演算量は$O(N^3)$。
  - 行列計算（連立一次方程式）
    直接法（消去法）で処理するとN変数の連立一次方程式解法の計算量は$O(N^3)$。
  - 多体問題（宇宙物理の重力計算等）
    N体の質点間の重力は相互作用の総当たり計算になるため、計算量は$O(N^2)$。
- 計算機の性能とデータ容量はいくら大きくても「大き過ぎる」ことはない
  ⇒大規模計算処理はもはや並列処理なしには成り立たない
- 大容量計算、大容量データ、大容量通信等の要求に対し、計算リソースを適切にマッピングし、効率的な並列処理を行う技術が必須になっている

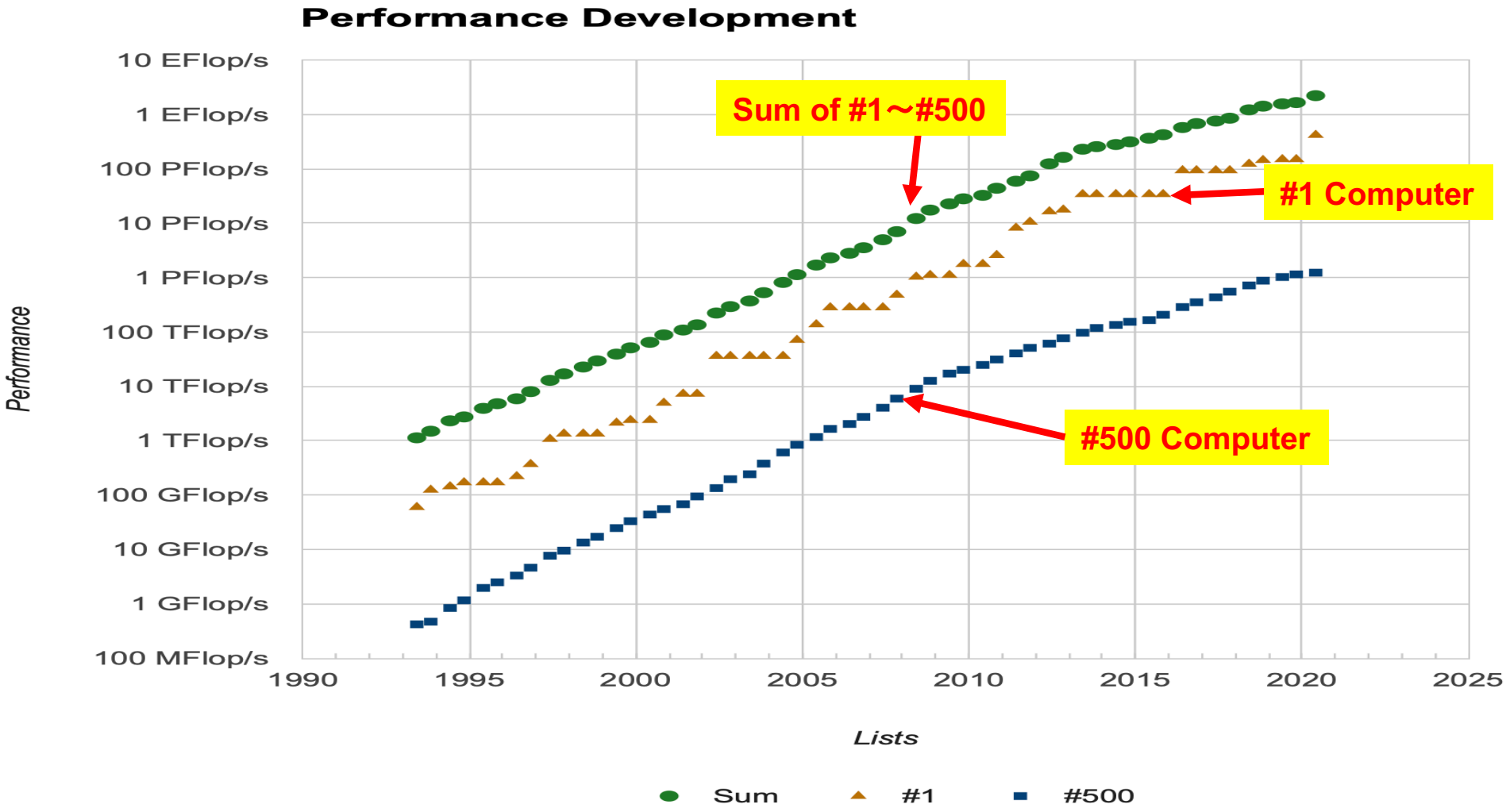
計算性能・データ容量の限界

- 半導体の集積度は1.5年で約2倍になる
  ⇒**ムーアの法則**
- もしIC上のトランジスタ数の増加を演算性能に転換できれば、「プロセッサの性能は1.5年で約2倍になる」と読み替えられる。
- 右図：Intelプロセッサのトランジスタ数の伸び
- メモリ等の容量の伸びもほぼこれに従う
- しかし、これだけでは爆発する性能向上要求に対応できない
  （ムーアの法則に従うプロセッサ単体の性能向上も限界に近づいている）
  ⇒**（超）並列化**



TOP500 List (Jun. 2020)

http://www.top500.org



TOP500 list on Jun. 2020 (#55)

#	Machine	Architecture	Country	Rmax (TFLOPS)	Rpeak (TFLOPS)	GFLOPS/W
1	**Fugaku, R-CCS**	MPP (Fujitsu, A64FX)	Japan	415,530	513,855	14.665
2	Summit, ORNL	Cluster (IBM, GPU V100)	USA	148,600	200,795	14.668
3	Sierra, LLNL	Cluster (IBM, GPU V100)	USA	94,640	125,712	12.724
4	TaihuLight, NSCW	MPP (Sunway, SW26010)	China	93,015	125,436	6.051
5	Tianhe-2A (MilkyWay-2A), NSCG	Cluster (NUDT, Xeon, Matrix2000)	China	61,445	100,679	3.325
6	**HPC5, Eni.S.P.A.**	Cluster (DELL, Xeon, GPU V100)	Italy	35,450	51,721	15.740
7	**Selene, NVIDIA**	Package-Cluster (NVIDIA, AMD, GPU A100)	USA	27,580	34,569	20.518
8	Frontera, TACC	Cluster (DELL EMC, Xeon)	USA	23,515	38,746	????
9	**Marconi-100, CINECA**	Cluster (IBM, GPU V100)	Italy	21,640	29,354	14.661
6	Piz Daint, CSCS	MPP (Cray, XC50: GPU P100)	Switzerland	21,230	27,154	8.905

- **#15⇨#19 Oakforest-PACS (U. Tsukuba + U. Tokyo) 13.555 PFLOPS (was #6, Nov. 2016)**
- **# 353⇨#402 Cygnus (U. Tsukuba) 1.582 PFLOPS (was #264, Jun. 2019)**

並列化手法

- 問題の並列化には様々な切り口がある
  - 1つの問題を分割
    （例）domain decomposition：問題を空間領域で切り分け、各部の処理を並列プロセスとする
  - 1つの処理を分割
    （例）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
- 並列化方法のいろいろ
  - EP (embarrassingly parallel)：並列化が自明なもの（parameter search等）、並列処理単位間の独立性が保証されている
  - data parallel：処理対象のデータを分割（domain decomposition等）
  - pipeline：流れ作業の中の各部分作業を並列リソースに割り当て
  - master/worker：処理すべきプロセスのプールを複数のワーカーに順次割り当て

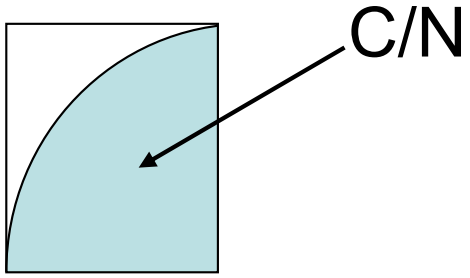
EPの例

- 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

N個の(x, y)対 $(0 \le x \le 1, 0 \le y \le 1)$をランダムに生成し、

$x^2 + y^2 < 1$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。

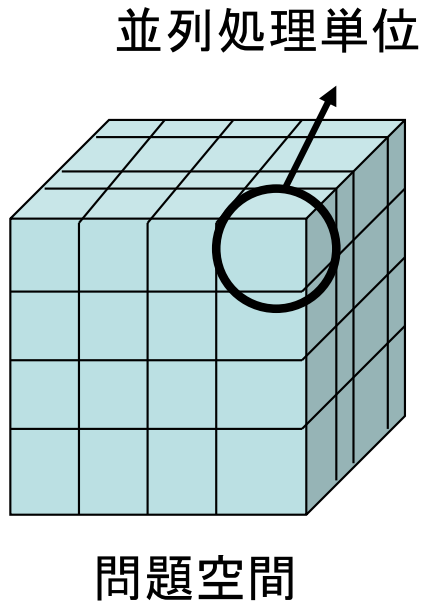
  - 各(x,y)対に対する単位円内点かどうかの判定は完全に独立に処理可能⇒独立処理（完全並列化）が可能
  - 一番最後にCの総和を求めればよい



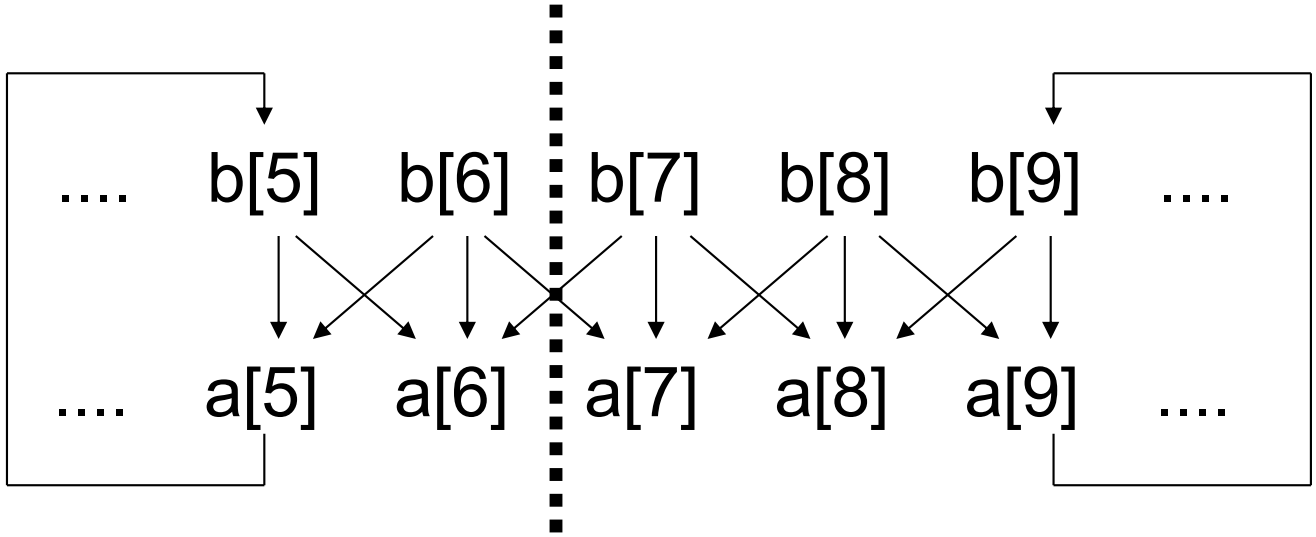
データ並列の例

- domain decomposition
  - 空間に一様な計算点が分布しており、これを適当な軸で分割し、並列処理
  - EP的な処理になる場合もあるが、一般的に隣接点等で何らかのinteractionが必要になる場合が多い
  - 例:

```
for(t=0; t < T; t++){
    for(i=1; i < N-1; i++)
        a[i] = b[i-1] + 2*b[i] + b[i+1];
    for(i=0; i < N; i++)
        b[i] = a[i];
}
```



domain decomposition（続き）



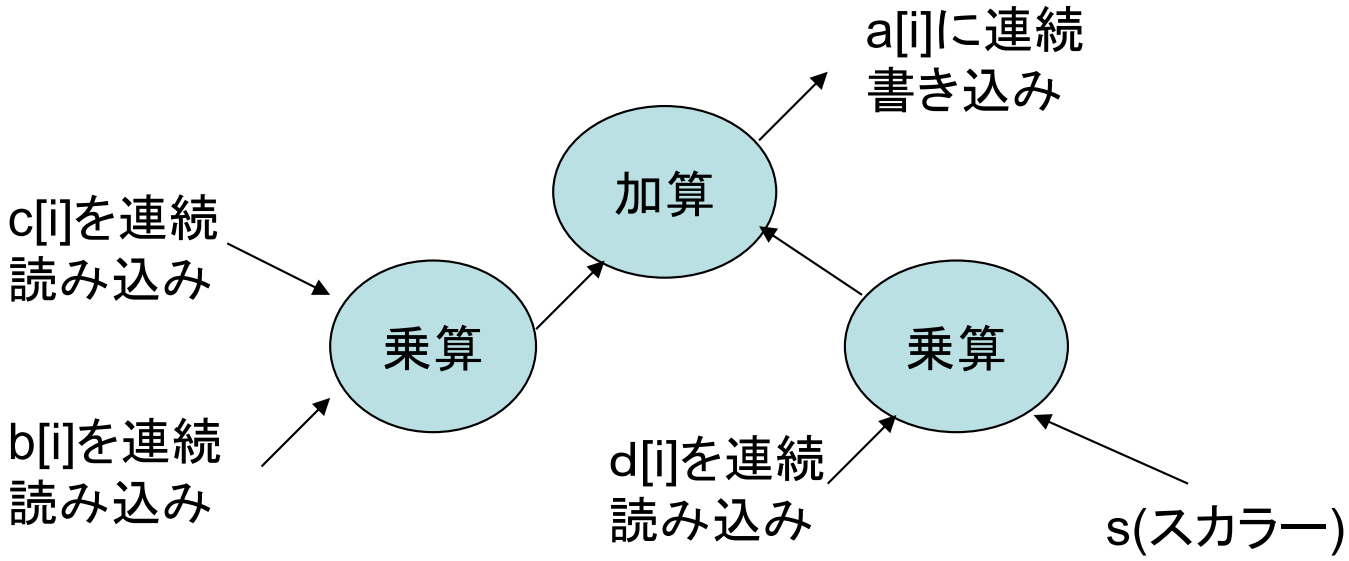
並列処理単位の切れ目

```
for(t=0; t < T; t++){
   for(i=1; i < N-1; i++)
      a[i] = b[i-1] + 2*b[i] + b[i+1];  // 更新時に通信あり
   for(i=0; i < N; i++)
     b[i] = a[i];                       // 更新時に通信なし
}
```

パイプライン並列

- 一連の処理が流れ作業で進む場合、各ステージの処理にリソースを割り当て、全体を並列に処理
- ベクトル処理の例

```
for(i=0; i < N; i++){
    a[i]=b[i] * c[i] + s * d[i];
}
```



master/worker型並列処理

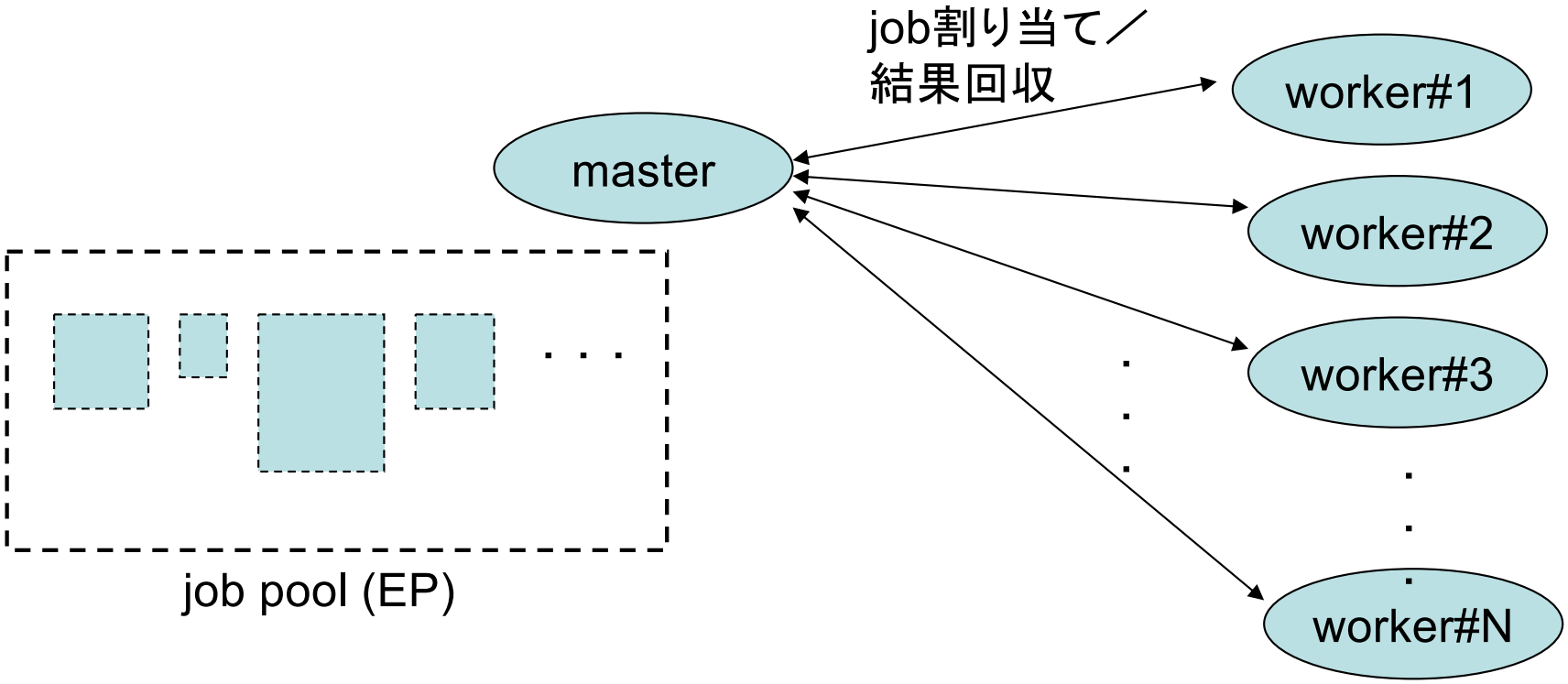
- 1つのmasterプロセスと複数のworkerプロセスがあり、masterが多数の独立な処理の「プール」を持つ（処理数>>worker数）
- masterはプールから処理すべき問題を取り出し、全workerに1つずつ与える
- workerは与えられた処理を行い、終了したら結果をmasterに返し、次の処理を割り当ててもらう

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



並列処理における通信と同期

- 各並列プロセスは何らかの関係を互いに持ち、情報を交換する必要がある
- 逐次処理では本来必要のない処理が追加される
  例）domain decompositionの例での境界部分のデータ参照
- 通信(**communication**)が引き起こす演算性能への影響
  - 通信そのものにかかる正味時間：
    逐次処理では存在しなかったオーバヘッドを引き起こす
  - 待ち合わせ（**同期:synchronization**）によって生じる処理の停滞：
    負荷不均衡（load imbalance）によって生じる、無駄な待ち時間。各プロセス内の処理が他のプロセスからのデータ待ちで停止
- 効率的な並列処理を行うためには、通信と同期のオーバヘッドを最小限に食い止める必要がある
  （並列処理効率の単元で後述）

並列処理における通信の形とコスト

- 並列処理アーキテクチャの様式に依存
  - 分散メモリ型アーキテクチャ:
    各プロセスは互いに明示的にデータを交換(送受信)し合う
    ⇒message passing (send, receive, ...)
  - 共有メモリ型アーキテクチャ:
    各プロセスは共有するメモリへのデータの読み書きで通信し合う
    ⇒shared memory access (write read, ...)
- 通信のコスト
  - message passing の場合、通信ペア同士がネットワーク上でどのような位置にいるかが重要
    ⇒「近接な通信」(物理的に近距離にいるプロセッサ同士の通信)は低コスト、かつシステム全体の通信性能に与えるインパクトが小さい
  - shared memory accessの場合、通信ペア同士とメモリの物理的な位置(必ずしも全てのメモリがプロセッサから等距離にあるとは限らない)が重要
  - いずれの場合も、底辺にあるハードウェアの性能に左右される

同期処理のコスト

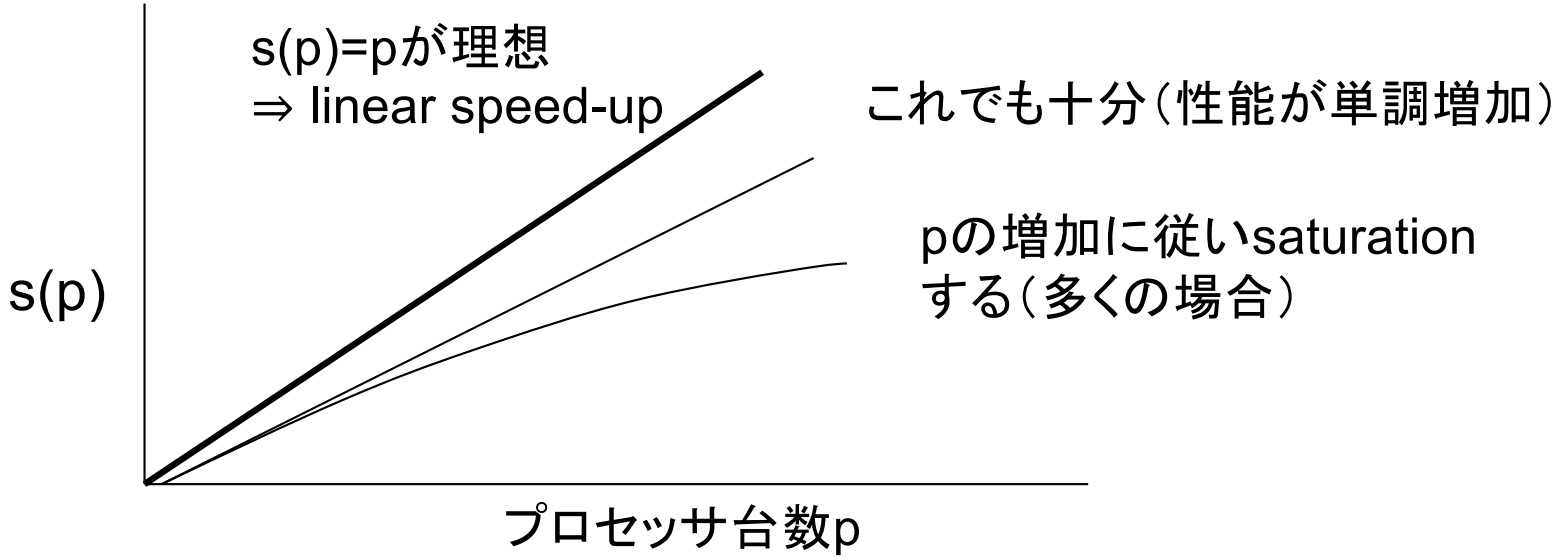
- 同期のオーバヘッド（分散メモリの場合）
  - 並列計算機を構成するネットワークの形状と同期処理のアルゴリズムに強く依存
  - システムの規模（並列プロセッサ台数）に強く依存
- 同期のオーバヘッド（共有メモリの場合）
  - 共有メモリアクセスにおける同期処理がハードウェアでサポートされているかどうか（ロック機構等）
  - 同期に関わる並列プロセス数に強く依存

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)

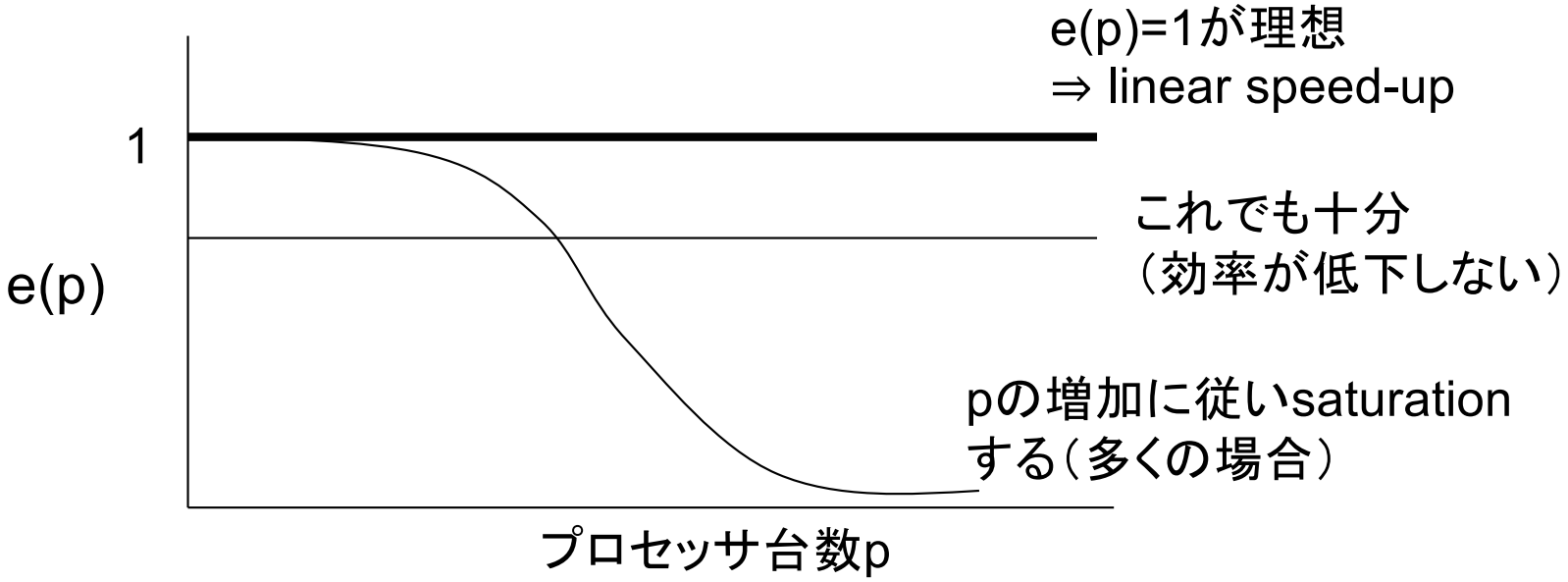
並列処理システムの性能指標(1)

- 速度向上率
  - 1プロセッサで実行した時の時間を*T1*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)=T1/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)



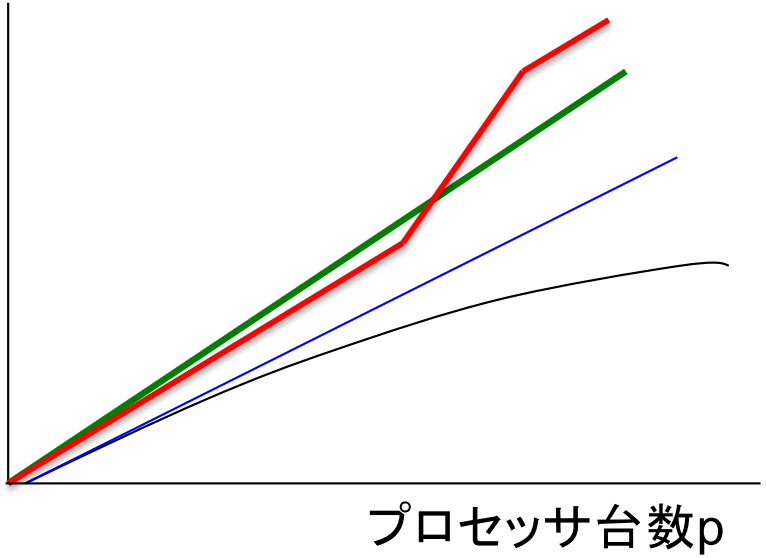
並列処理システムの性能指標(2)

- 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)



Ideal speed-up vs Super-linear

- 速度向上率カーブにおいて
  - 緑：100%効率の理想線形 (ideal speed-up)
  - 青：線形向上 (linear speed-up)
  - 黒：通信等にボトルネックがある場合（一般的）
  - 赤：超線形 (super linear)



- なぜ super linear が起きるか？
  - 基本的にidealまたはlinearである並列処理において
  - プロセッサ台数 p が増えても効率が落ちず
  - 全体の問題サイズが一定だと各プロセッサの割り当て領域サイズが小さいため
  - 主にキャッシュが効いて（例：全データがキャッシュに収まる）プロセッサ当たりの性能が大幅に向上
  - 結果的に1プロセッサの場合を基準とすると p 倍以上の性能になる

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*T1*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

$$T_1 = T_p + T_s$$

  - *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

$$T(p) = T_s + T_p / p$$

  - プロセッサ台数*p*を無限大にすると

$$\lim_{p\to\infty} T(p) = T_s$$

  - この時

$$\lim_{p\to\infty} e(p) = s(p)/p = (T_1/(T_s + T_p/p))/p = (T_s + T_p)/(T_s * p + T_p) = 0$$

従って、プロセッサ台数pをいくら増大しても、Ts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

Scalableな問題

- 実際、どんな問題にも必ずTsは存在し、並列化におけるscalabilityには限界がある
- 「大規模並列」「超並列」は無意味か？
  ⇒Tsが無視できるぐらいTpの大きな問題を想定すればよい
  ⇒**scalable(拡張性のある）問題**
- 大規模科学技術計算の多くがscalableな問題
  ただし、Tsが問題規模またはシステム規模(p)に応じて増大しないよう注意が必要
- 実問題上で、それがシステム規模に応じてscaleしているかどうかを見極めるための新しいメトリックが必要
  ⇒並列処理の**粒度（granularity）**

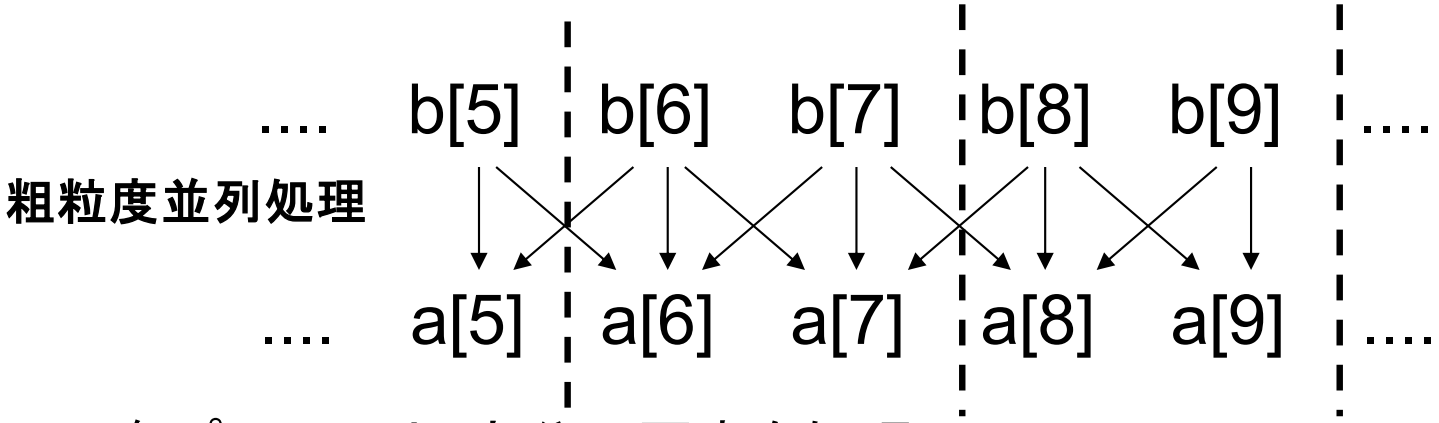
Scalability: Strong vs Weak

- 並列システムの規模を大きくする際、対象問題のサイズをどう考えるか？
  - 全体の問題サイズは変わらず、システムの並列度を上げる
    ⇒ 各プロセッサに割り当てられる問題領域は小さくなる
    ⇒ 通信等のオーバヘッドは相対的に大きくなる
    ⇒ Strong Scaling
  - プロセッサ当たりの問題領域は変わらず、システムの並列度を上げる
    ⇒ 全体の問題サイズがプロセッサ台数に応じて（比例して）大きくなる
    ⇒ 通信等のオーバヘッドは相対的に（ほぼ）一定
    ⇒ Weak Scaling
- strong scaling は weak scaling に比べ遙かに難しい
  - strong scaling = 一定問題をいかに速く解くか (time-to-solution)
  - weak scaling = 一定時間でいかに大規模な問題を解くか (scalable)

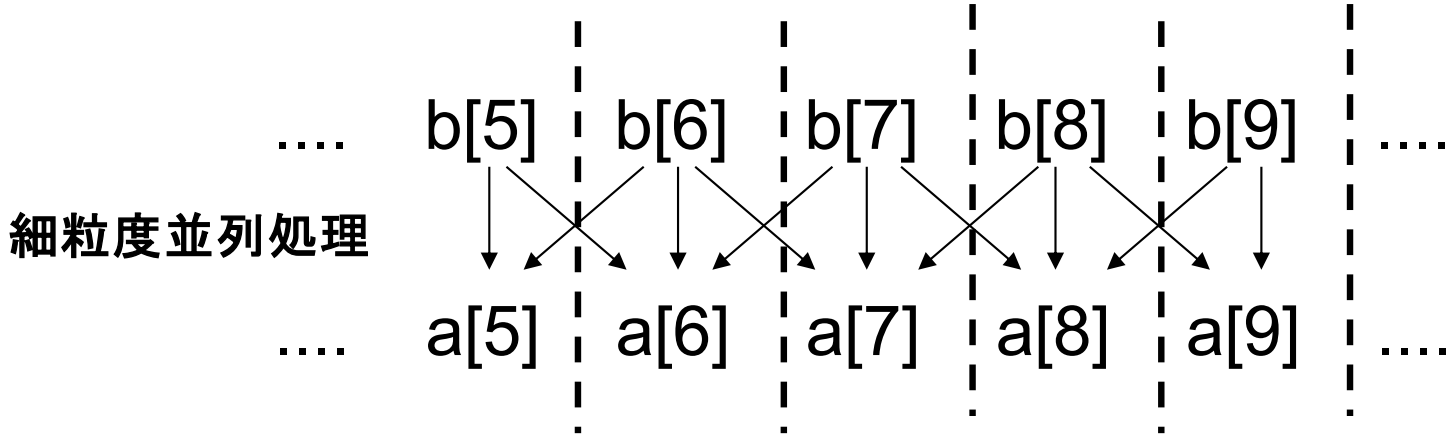
並列処理の粒度（granularity）

- 並列処理性能を低下させる最大要因は通信と同期のオーバヘッド
  - 逐次処理にはなかった**余計な時間**が発生
  - 同期により**演算も通信もしていない待ち時間**が発生
- 一定時間、通信も同期も発生しない状況は逐次処理を行っているのと等価である
  ⇒この時間が長いほど処理効率が高い
- 通信・同期を伴わない処理部分の相対的評価
  - この部分が長い⇒**「粒度が粗い」(coarse grain)**
  - この部分が短い⇒**「粒度が細かい」(fine grain)**
- 一般に、問題規模を固定のままシステムの並列度を上げると、並列処理の粒度が細かくなる
  ⇒ strong scaling が難しい最大の理由

domain decompositionの例



- 各プロセスは2点分の要素を処理
- 隣接プロセスに対し2点分のデータを送り、2点分のデータを受け取る



- 各プロセスは1点分の要素を処理
- 隣接プロセスに対し2点分のデータを送り、2点分のデータを受け取る

並列処理性能を左右する要因

- システム並列度を上げた場合、それに応じて問題規模を大きくすれば粒度を下げずに済む (weak scaling)
- 粒度の概念は相対的なものなので、各アプリケーションにおいてこれを計る尺度を設けておくべき
  ⇒例えば通信の前後に計時機能を組み込む等
- システム並列度を上げた場合、常に並列処理効率を観測し、リソースの有効利用に注意を払うことが望ましい
- 性能ボトルネックの最も大きな要因が通信と同期のオーバヘッド、すなわち処理の細粒度化
- アプリケーション自体が本質的に細粒度である可能性があるが、これはアルゴリズムの修正、プログラミングの工夫等で対応しなければならない
- 粒度を上げるために、従来と異なる通信パターンが発生してしまうような場合は注意が必要
  ⇒システムの通信・同期性能自体が変わってしまう

負荷バランスと並列処理効率

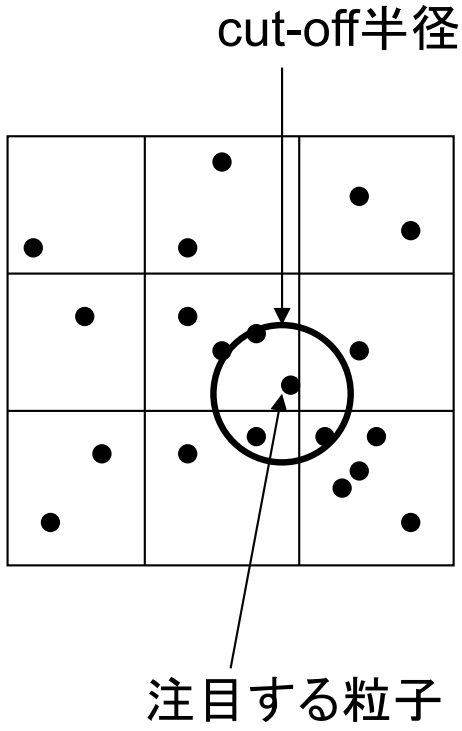
- 負荷バランス：各並列プロセスに与えられる処理の量は均一であることが望ましい
  例）一斉同期を取る場合にボトルネックとなるプロセスが生じると処理が停止する
- 均質なデータに対する単純なデータ並列では、基本的に負荷バランスは暗黙のうちに保たれる
- 負荷バランス問題が生じる場合
  - 不均質なdomainに対する分割
  - parameter searchにおいて個々の処理量がparameterに依存する場合
  - 問題空間が一様でなく、「境界」「内点」等が存在する場合
    （例：周期境界条件でない問題）

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

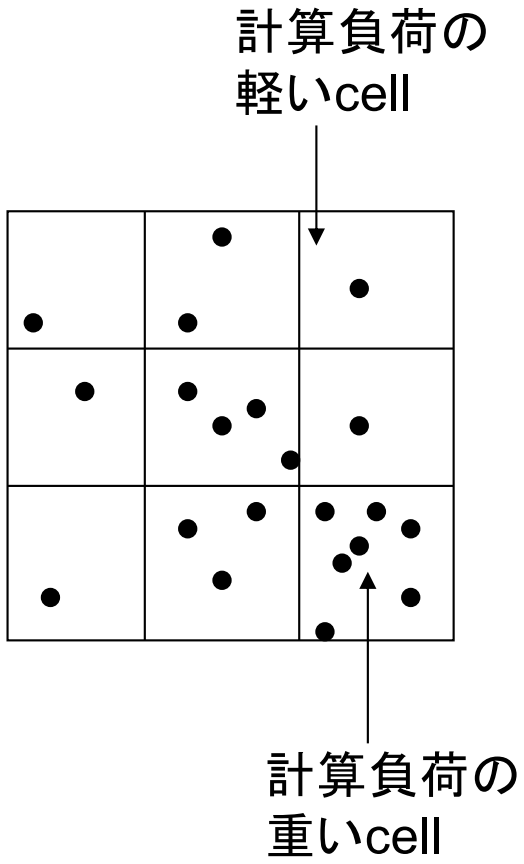
具体例：cut-off付きMD

- 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

まとめ

- 大規模科学技術計算における並列処理の重要性
- 並列処理の典型的な手法、並列化方法
- 並列処理効率の概念
- アムダールの法則
- 並列処理のscalabilityと処理粒度の関係
- 負荷分散問題

レポート課題

[1] 並列処理性能とその効率について以下の設問に答えよ。
ある処理を1台のプロセッサで実行した時、*T1* [sec]で終了する。この処理が完全並列化可能（逐次部分は無視できる）である時、*p*台のプロセッサでの実行を考える。この処理は完全並列化は可能であるが、*p*台のプロセッサで並列処理する際、*Tcomm(p)* [sec]だけの通信時間がオーバヘッドとして発生するとする（通信時間は*p*の関数である）。

(a) この問題を*p*台のプロセッサで処理する際の処理時間*T(p)*, 速度向上率*s(p)*, 並列化効率*e(p)*をそれぞれ式で表せ。

*(b) Tcomm(p)*がp^2に比例し、$Tcomm(p)=\alpha p^2$と表せたとする。並列化効率を90%以上に保つには、*α*はどのような値でなければならないか。*T1*と*p*を用いて表せ。

(c) EPな問題の場合、(2)におけるαはどのようにみなせるか。また、T1とαの関係と並列処理性能の関係について定性的に述べよ。

レポート課題（続き）

[2] 2次元の物理問題領域のサイズがN^2であるとする。これをn^2台のプロセッサで並列処理する（ただし、$N^2 >> n^2$であるとする）。並列処理手法としてdomain decompositionを用いる場合、問題のプロセッサへの割り当て方法として、(a)問題領域を特定の1つの次元方向でのみで分割する方法と、(b)それぞれの次元をプロセッサのそれぞれの次元で分割する方法の2通りが考えられる。講義資料27ページにあるように、全要素の値を更新する際、互いに隣り合う要素の値を参照するとする（ただし資料では1次元空間だがここでは2次元空間とする）。
領域分割において、各点における演算量を6[FLOP]、各点当たりの隣接点との通信量を8[Byte]とした時、分割方法(a)と(b)について、並列処理時の演算量と通信量の比率についてどちらが一般的に有利であるか、定量的に論ぜよ。ただし、問題領域の外周部分についての通信量は無視し、式を簡略化してよい（周期境界条件を想定してよい）。