並列計算の概念（プロセスとスレッド）

長岡技術科学大学　電気電子情報工学専攻　出川智啓

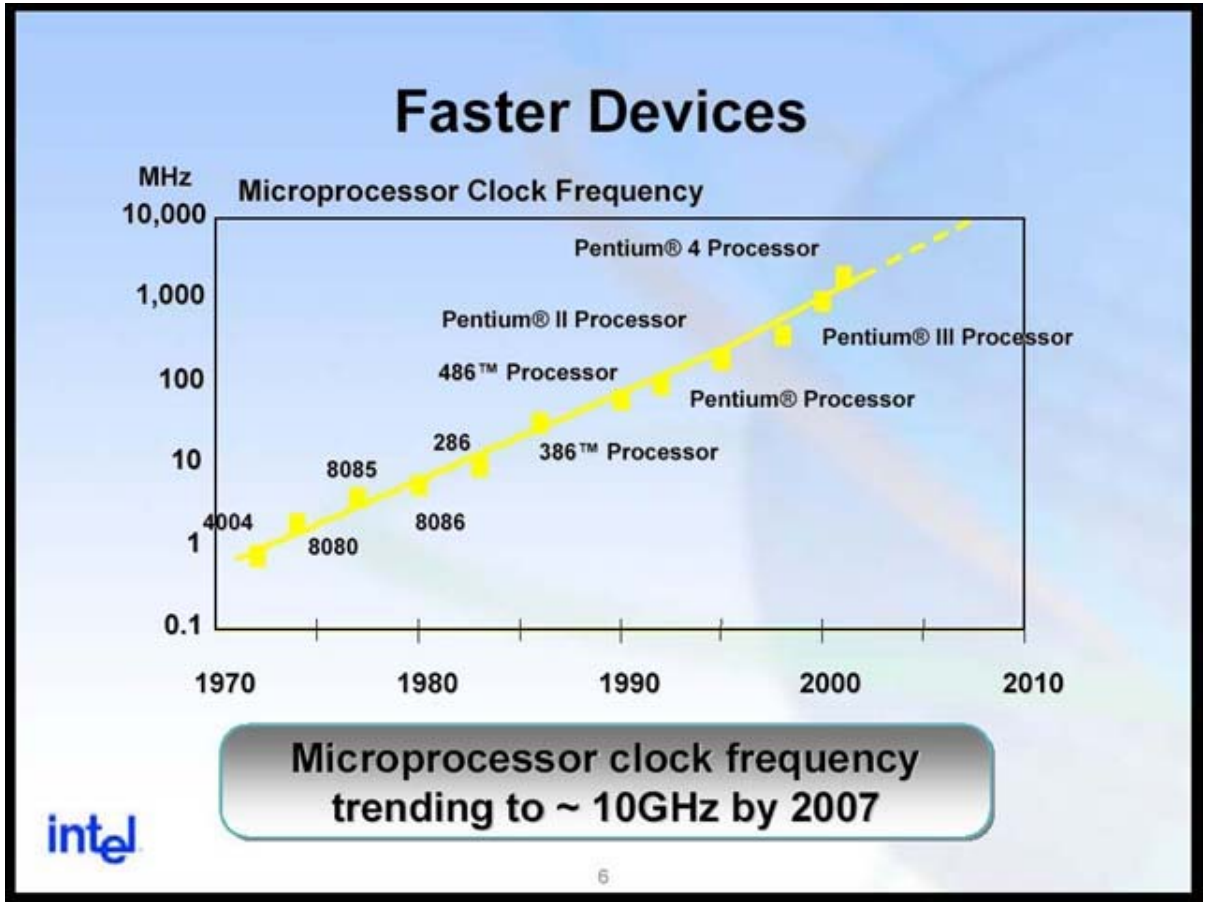
今回の内容

- 並列計算の分類
  - 並列アーキテクチャ
  - 並列計算機システム
  - 並列処理

- プロセスとスレッド
  - スレッド並列化
    - OpenMP
  - プロセス並列化
    - MPI

CPUの性能の変化

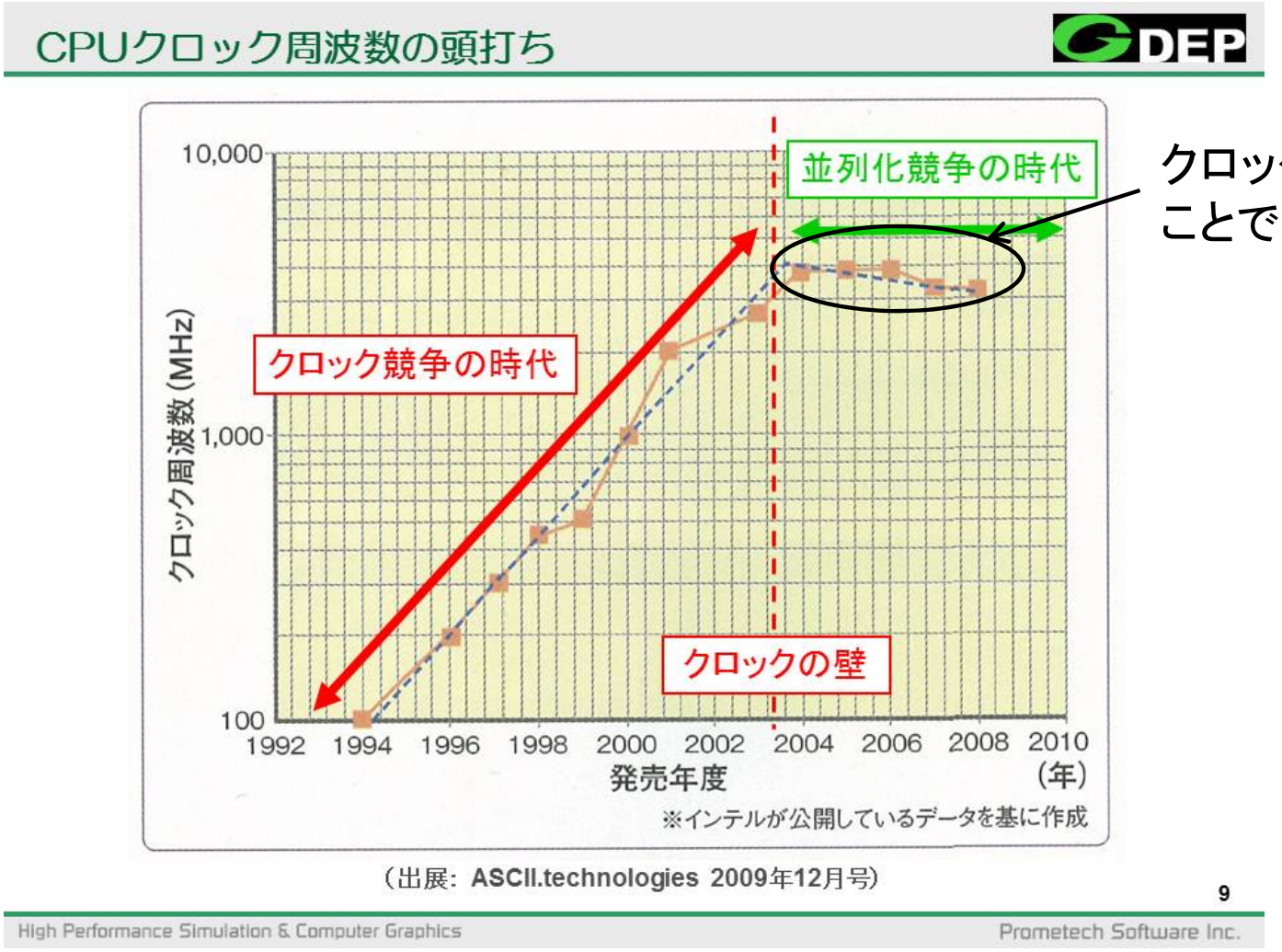
- 動作クロックを向上させることで性能を向上



http://pc.watch.impress.co.jp/docs/2003/0227/kaigai01.htmより引用

CPUの性能の変化

- 増加し続けるトランジスタ数をコアの増加に充てる



クロックを遅くすることで発熱を抑制

http://www.gdep.jp/page/view/248より引用

CPUの性能の変化

- FLOPS = 1コアの演算性能
  × コア数
  × CPUの動作周波数

- **1コアの演算性能の向上**
  - 演算器(トランジスタ)の増加

  } コンパイラの最適化を利用

- **コア数の増加**
  - トランジスタ数の増加

  } 複数のコアを使うように
  プログラムを書かないと
  速くならない

- **CPUの動作周波数**
  - 回路の効率化や印可電圧の向上

  } 劇的な性能向上は期待できない

CPUの性能の変化

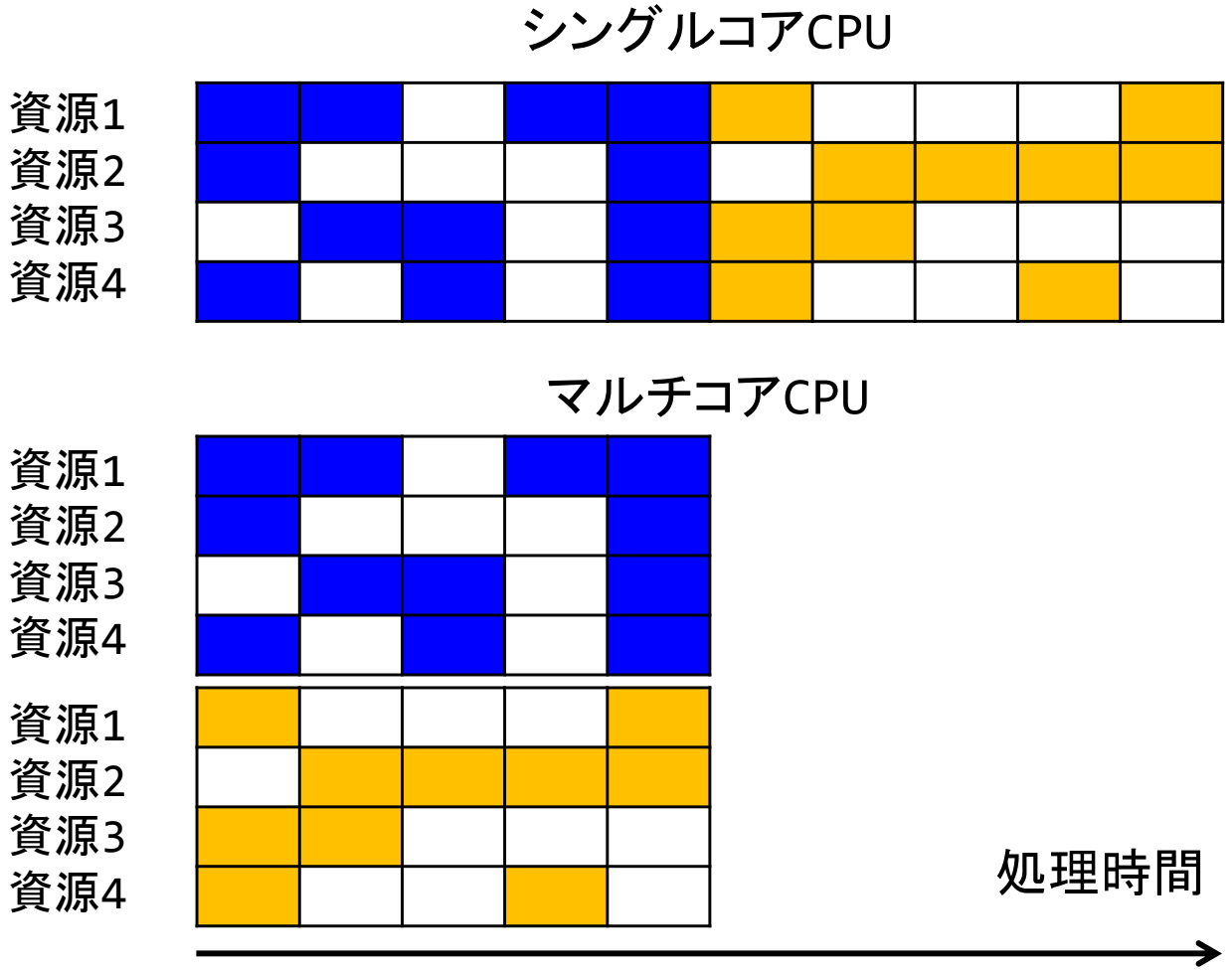
- シングルコア
  - 動作クロックを向上させることで性能を向上
  - CPUの動作は指数関数的に高速化
  - 遅いプログラムでもCPUを新しくすれば高速化

- マルチコア
  - 効率的な利用には並列計算を意識したプログラミングが必要
  - 待っていればCPUが勝手に速くなっていった時代はもう終わり

Free Lunch is Over
（タダ飯食いは終わり）

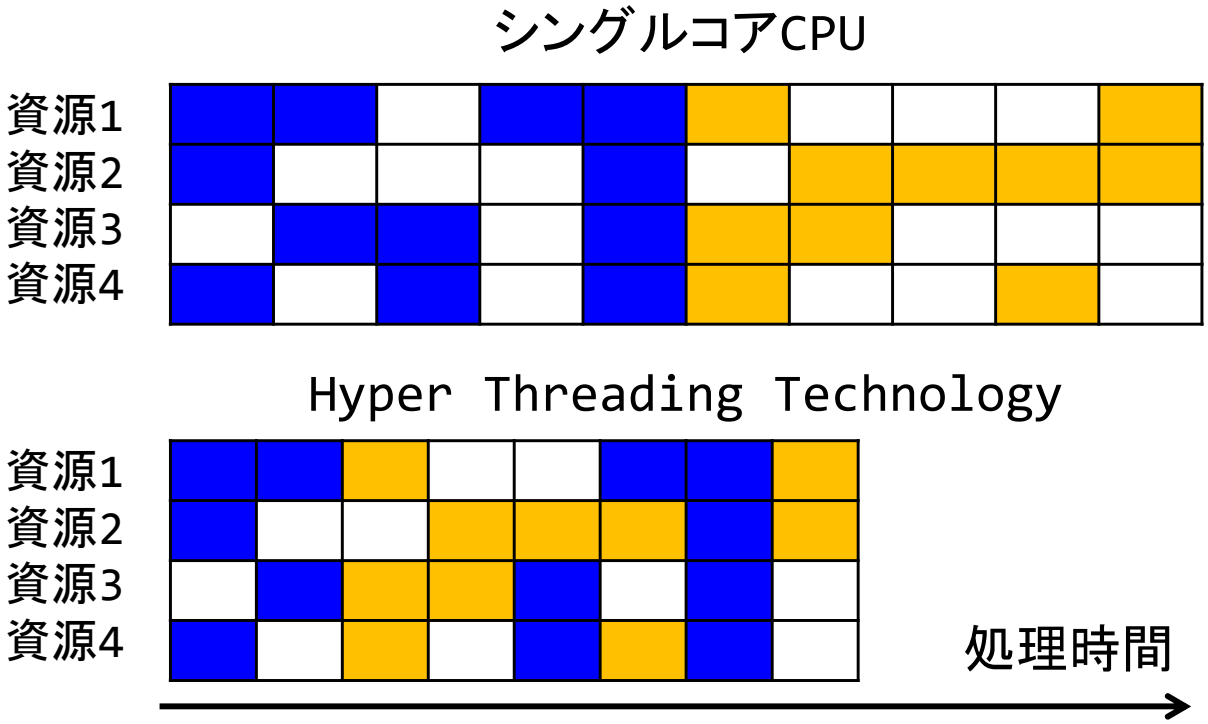
並列処理による高速化

- 処理をN個に分割して各コアが処理を分担
  - 実行時間が1/Nに高速化されると期待



並列処理による高速化

- Hyper Threading Technology
  - 一つの物理CPUを複数のCPUに見せる技術
  - CPU内のレジスタやパイプラインの空きを利用
  - 10～20%程度の高速化



並列計算の分類

- 並列アーキテクチャ
  - プロセッサレベルでの処理の並列化
  - データの処理と命令の並列性に着目
- 並列計算機システム
  - 計算機レベルでの処理の並列化
  - 計算機のメモリ構成に着目
- 並列処理
  - プログラムレベルでの処理の並列化
  - 処理の並列化の方法に着目

並列アーキテクチャの分類

- プロセッサの分類

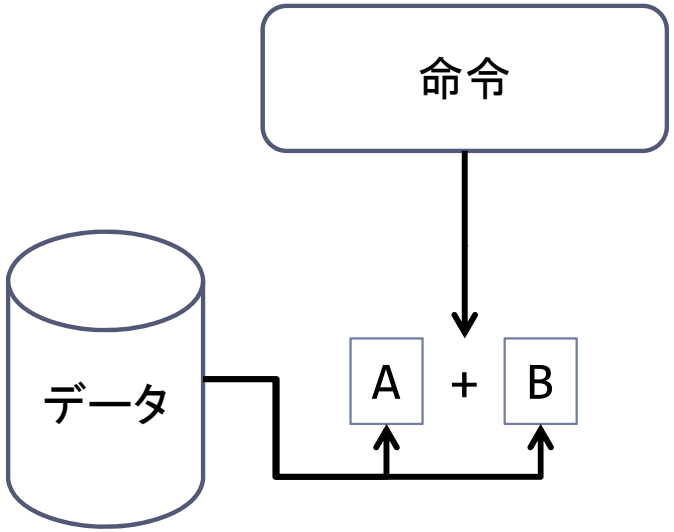
- Flynnの分類
  - 並列アーキテクチャのグループ分け
  - データの処理と命令の並列性に着目

1. SISD 単一命令単一データ
2. SIMD 単一命令複数データ
3. MISD 複数命令単一データ
4. MIMD 複数命令複数データ

Single Instruction Single Data stream

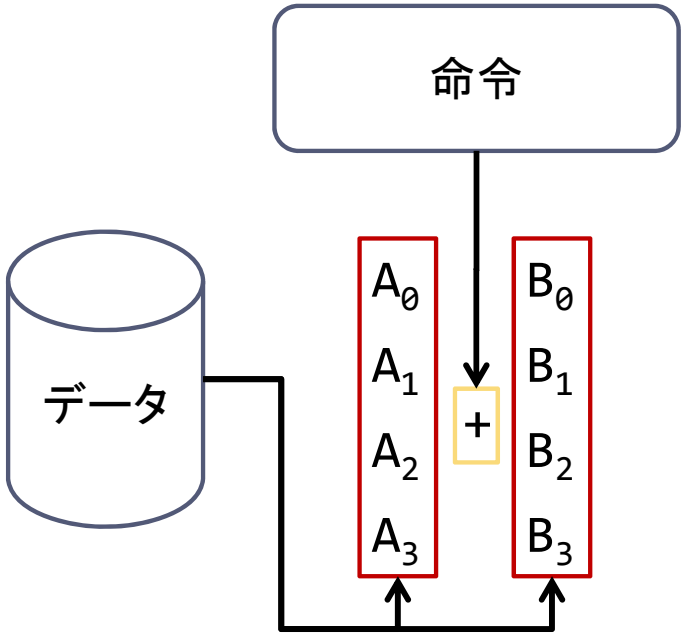
- 単一命令単一データ
  - 一つのデータに対して一つの処理を実行
  - 逐次アーキテクチャ，単一アーキテクチャ

- 一度に一つの命令を実行
  - パイプライン処理することで並列処理が可能



Single Instruction Multiple Data streams

- 単一命令複数データ
  - 複数のまとまったデータに対して同じ演算を同時に実行
  - 命令は一つ，その命令が同時に多くのデータに対して適用されるアーキテクチャ
- 数値シミュレーションに最適
- 数学のベクトルや配列計算の概念に一致
  - ベクトルプロセッサとも呼ばれる
- GPUも（一応）ここに分類



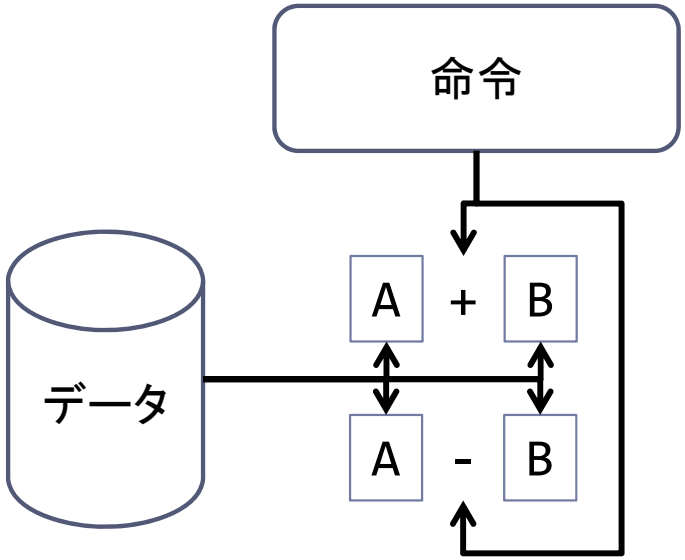
CPUのSIMD拡張命令セット

- SSE(Streaming SIMD Extensions)
  - 1命令で4個の単精度浮動小数点データを一括処理
    - インラインアセンブラで命令を記述
    - コンパイラの最適化で利用されることを期待

- AVX(Advanced Vector Extensions)
  - 1命令で8個の単精度浮動小数点データを一括処理
  - 1命令で4個の倍精度浮動小数点データを一括処理

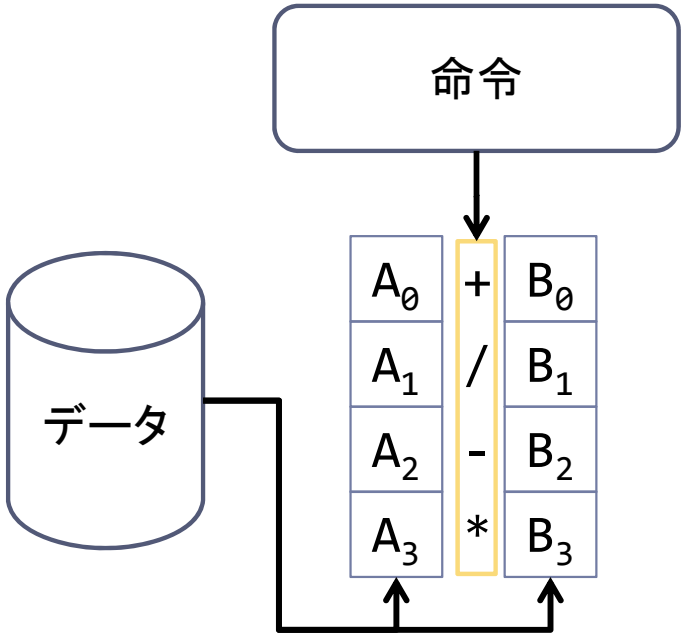
Multiple Instructions Single Data stream

- 複数命令単一データ
  - 複数のプロセッサが単一のデータに対して異なる処理を同時に実行
  - 形式上分類されているだけ
    - 実際にはほとんど見かけない
- 対故障冗長計算
  - 誤り訂正機能(ECC)を持たないGPU(Tesla以外)を2個同時に利用
  - 計算して結果を比較，異なっていればやり直し



Multiple Instructions Multiple Data streams

- 複数命令複数データ
  - 複数のデータに対して複数の処理を同時に実行
  - 一般的な並列コンピュータ
    - 複数のプロセッサがプログラマから見え，プロセッサごとに異なる命令を実行
- 対称マルチプロセッサ
  - 複数の同一種類のプロセッサを持つ
  - どのプロセッサでも同じようにプログラムを実行できる
- 非対称マルチプロセッサ
  - 複数のプロセッサを持つが，各プロセッサは同一種類ではない
  - プロセッサごとに処理の最適化ができる



その他の分類

- Single Program Multiple Data Streams
  - 単一プログラム複数データ
  - MIMDシステムのプログラミング手法
  - 現在のスーパーコンピュータやPCクラスタでプログラムを作る際の標準的な手法
  - 各PC用にプログラムを作らず，一つのプログラムの中で役割を認識

その他の分類

- Single Instruction Multiple Threads
  - 単一命令複数スレッド
  - GPUのアーキテクチャ
  - 一つの命令に対して，レジスタと演算器のペアがそれぞれデータを処理
    - CPUでは逐次処理が基本で，SIMD用の演算器を使って並列処理
    - SIMTでは並列処理が基本

並列計算の分類

- 並列アーキテクチャ
  - プロセッサレベルでの処理の並列化
  - データの処理と命令の並列性に着目

⇨ 並列計算機システム
  - 計算機レベルでの処理の並列化
  - 計算機のメモリ構成に着目

- 並列処理
  - プログラムレベルでの処理の並列化
  - 処理の並列化の方法に着目

並列計算機システム

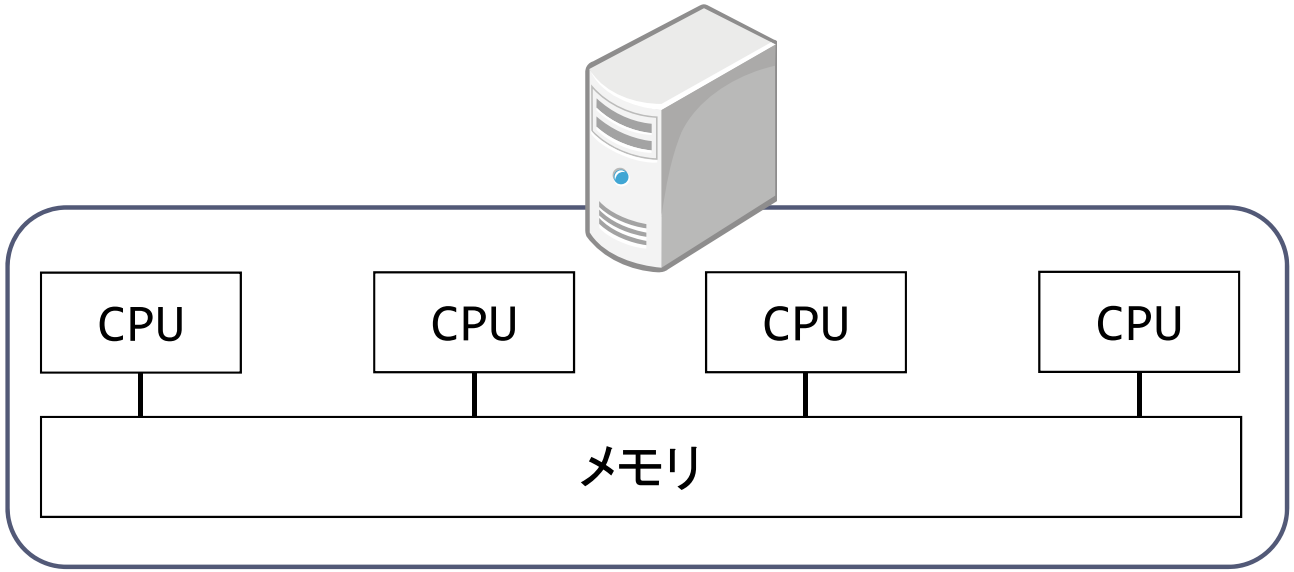
- 並列処理の基本
  - 処理を何らかの方法で分割
  - 分割した処理をプロセッサ（やコア）に割り当て同時に処理

- 並列計算機システム
  - 複数のプロセッサをもつ
  - 主にメモリに違いがある

1. 共有メモリシステム
2. 分散メモリシステム
3. ハイブリッドシステム

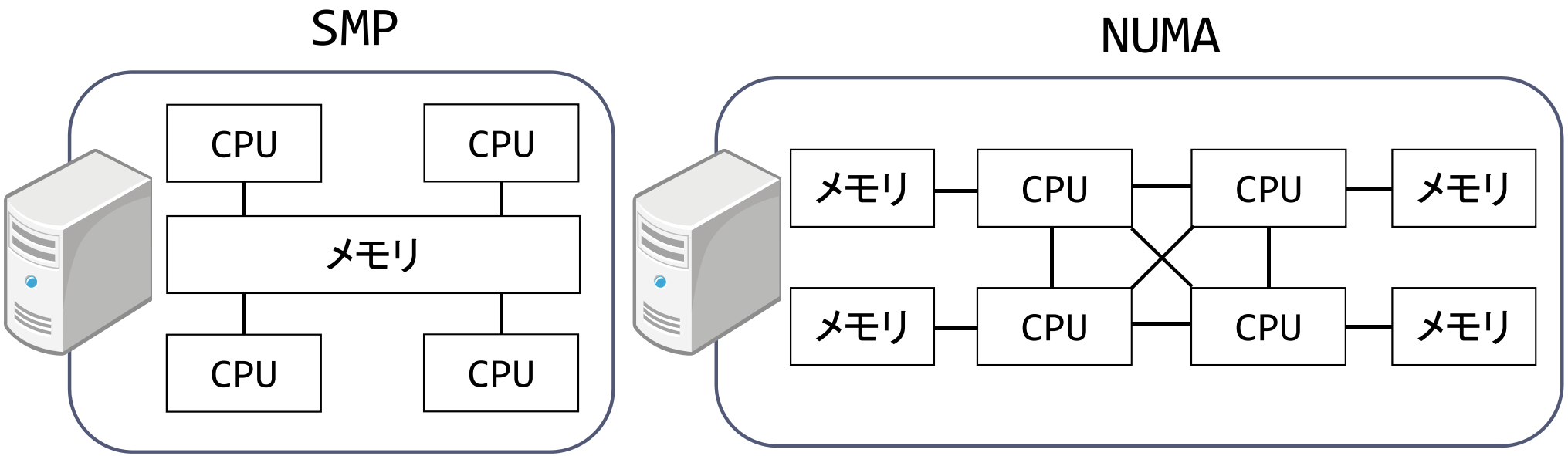
共有メモリシステム

- 複数のプロセッサがメモリ空間を共有
- 分割した処理は各プロセッサ上で並列的に処理
- 共有されたメモリ空間上の変数は全てのCPU(やコア)からアクセス(読み書き)可能
  - 他からアクセスされない変数を持つことも可能



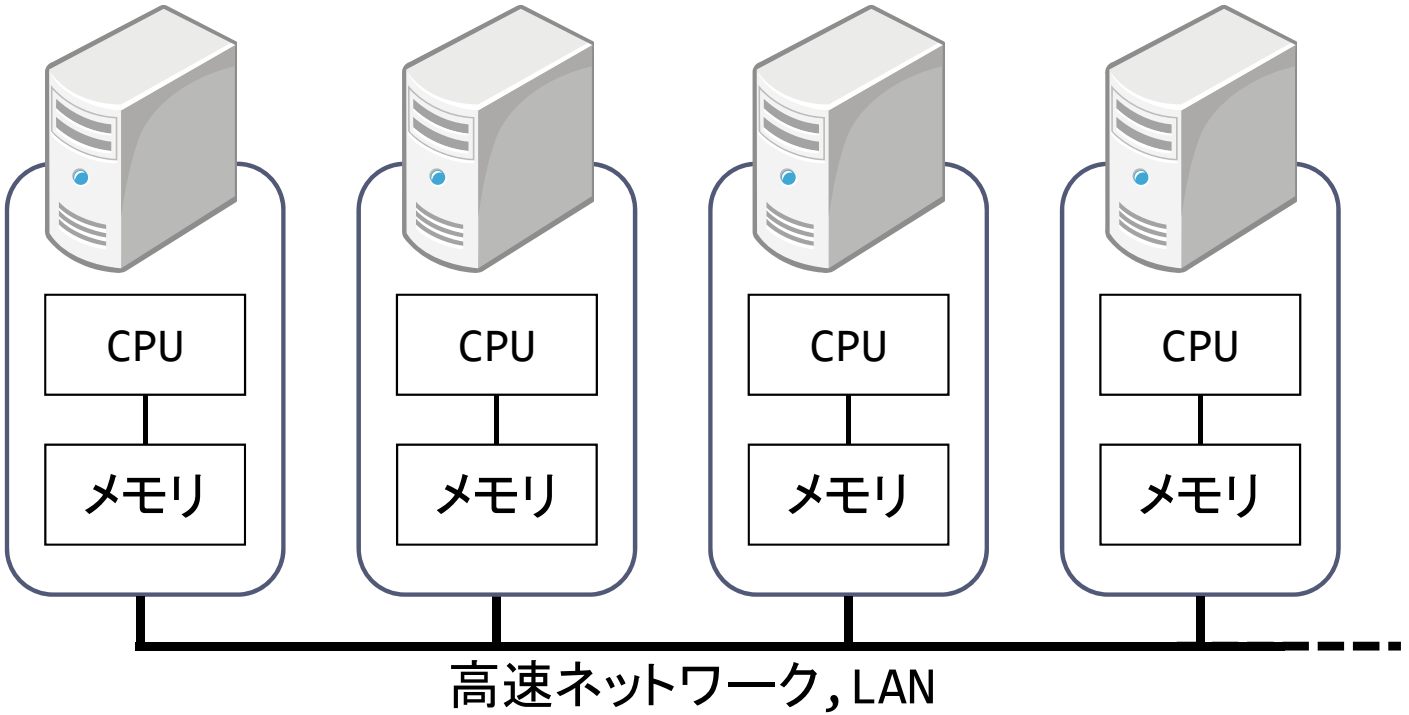
共有メモリシステム

- 各プロセッサが同等な条件でメモリにアクセス
  - SMP : Symmetric Multi-Processor
- 物理的に共有されていないがSMPに見えるシステム
  - NUMA : Non-Uniform Memory Access



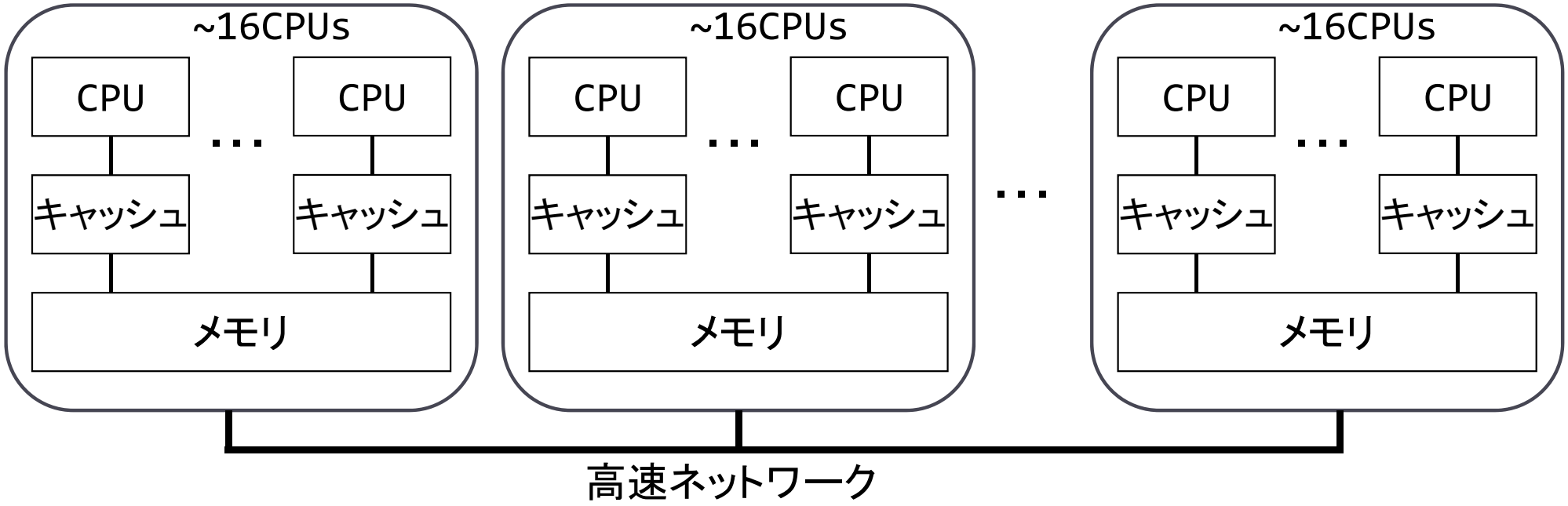
分散メモリシステム

- 複数のプロセッサが独立なメモリ空間を持つ
- 分割した処理は各PC上で並列的に処理
- 共有されたメモリ空間上の変数は全て共有されない
  - データの共有には通信を行う



ハイブリッドシステム

- 大規模な分散メモリシステム
  - MPP : Massively Parallel Processor
- 1ノードが共有メモリシステムからなるMPP
  - SMP/MPPハイブリッドシステム



並列計算の分類

- 並列アーキテクチャ
  - プロセッサレベルでの処理の並列化
  - データの処理と命令の並列性に着目

- 並列計算機システム
  - 計算機レベルでの処理の並列化
  - 計算機のメモリ構成に着目

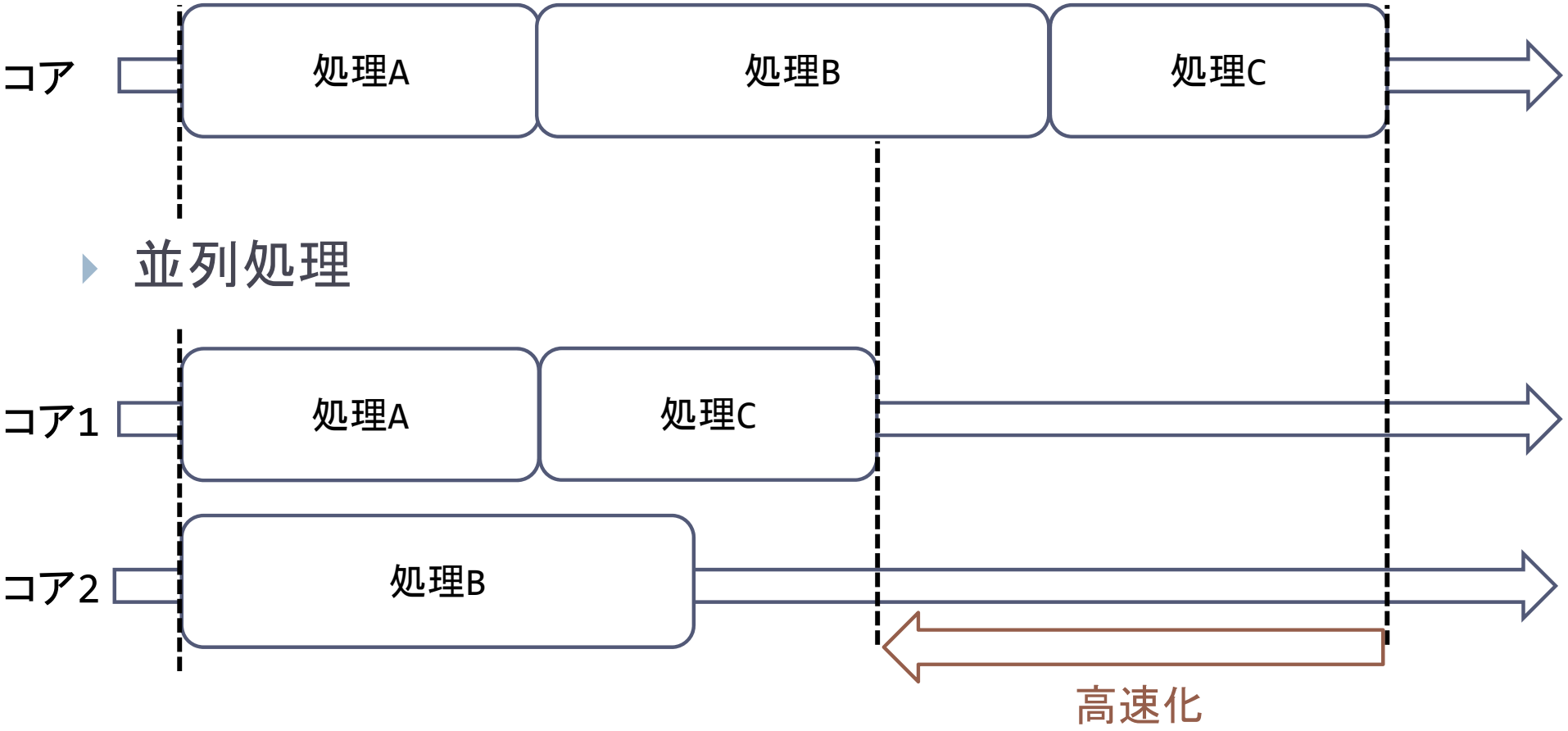
⇨ 並列処理
  - プログラムレベルでの処理の並列化
  - 処理の並列化の方法に着目

並列処理の分類

- タスク並列
  - 独立なタスクを異なるCPU,コアで同時に実行
- データ並列
  - 独立なタスクが処理するデータを分割し，異なるCPU,コアが参照し，処理を実行
- Embarrassingly parallel (perfectly parallel)
  - 各CPU,コアが同じタスクを異なるパラメータで実行
    - GPUが各ピクセルの色を決定し，ディスプレイに描画する処理
    - あるタスクに対してパラメータの影響を調査するような問題
    - 天気予報等での活用が期待されている

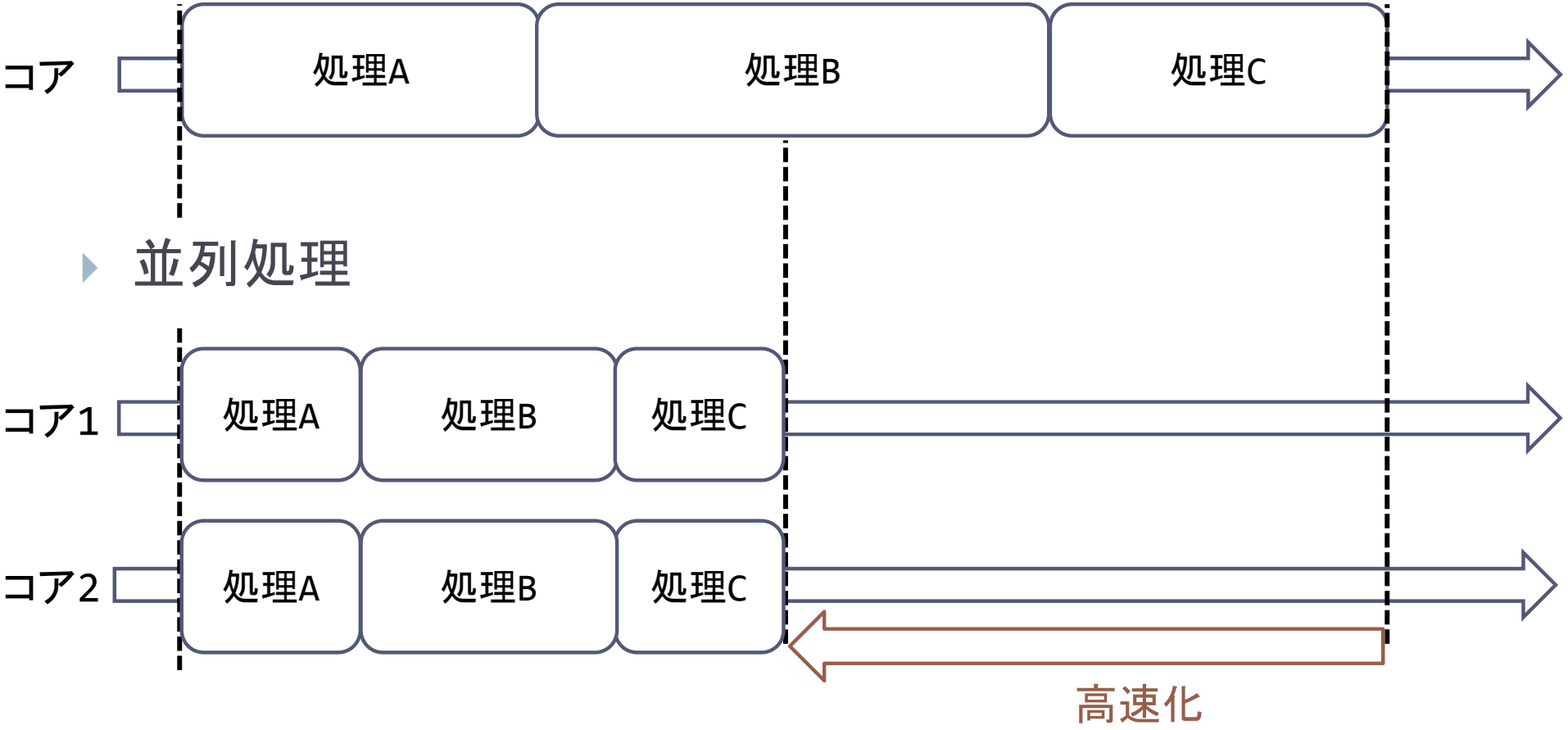
タスク並列

- 独立な処理A，B，Cを各CPU,コアが実行
  - 逐次処理
  - 並列処理



データ並列

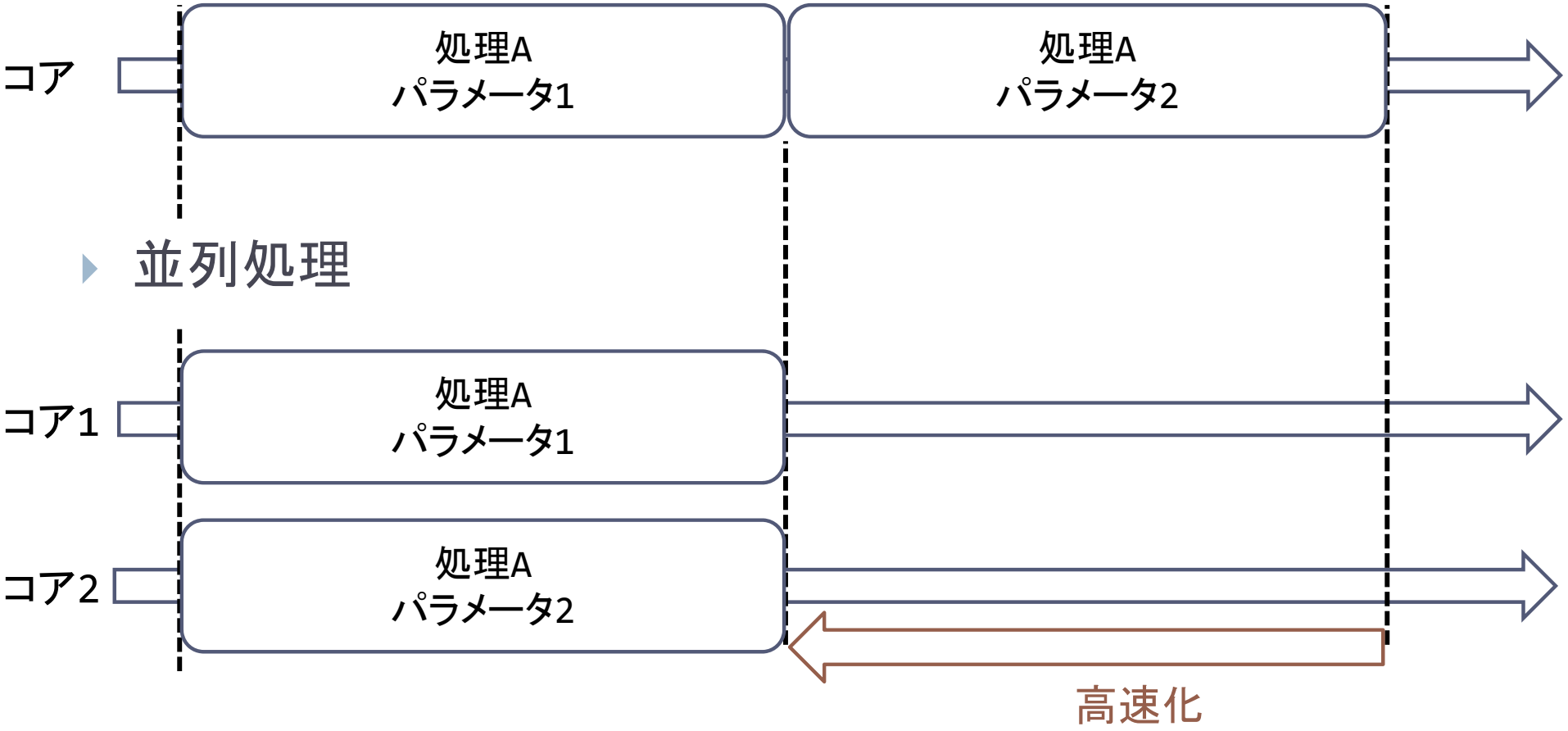
- 独立な処理A，B，Cが取り扱うデータを分割して実行
  - 逐次処理



  - 並列処理

Embarrassingly Parallel

- ある処理Aを複数の異なるパラメータで実行
  - 逐次処理
  - 並列処理



プロセスとスレッド

- プログラムの実行を抽象化した概念

- プロセス
  - OSから資源を割り付けられたプログラムの状態
  - 命令実行を行うスレッドとメモリ空間を含む

- スレッド
  - プロセスの中における命令実行を抽象化した概念

プロセス

- OSから資源を割り付けられ，実行状態（または待機状態）にあるプログラム
  - プログラムで定められた命令実行
  - 割り当てられたメモリの管理
- システムプロセス
  - OSの実行に関係するプログラム
- ユーザプロセス
  - ユーザ権限で実行されているプログラム

マルチプロセス

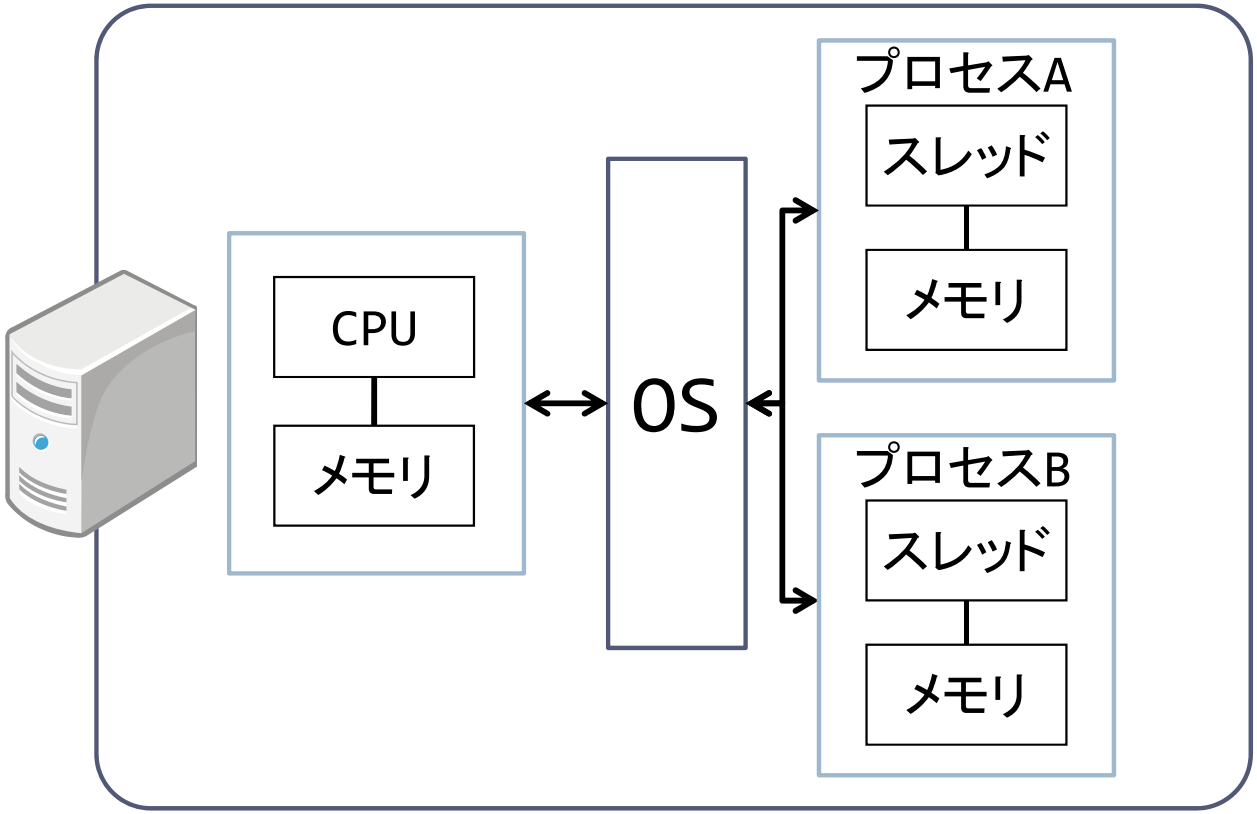
- 複数のプロセスが存在し，並列に実行
  - プロセスが一つのみ　　シングルプロセス
  - プロセスが二つ以上　　マルチプロセス

- マルチプロセスに対応したOSが必要
  - 現在のOSはマルチプロセスに対応

- シングルコアCPU一つでもマルチプロセスが可能
  - OSが複数のプロセスを切替
  - 複数のプロセスが並列に実行されているように見せる

マルチプロセス

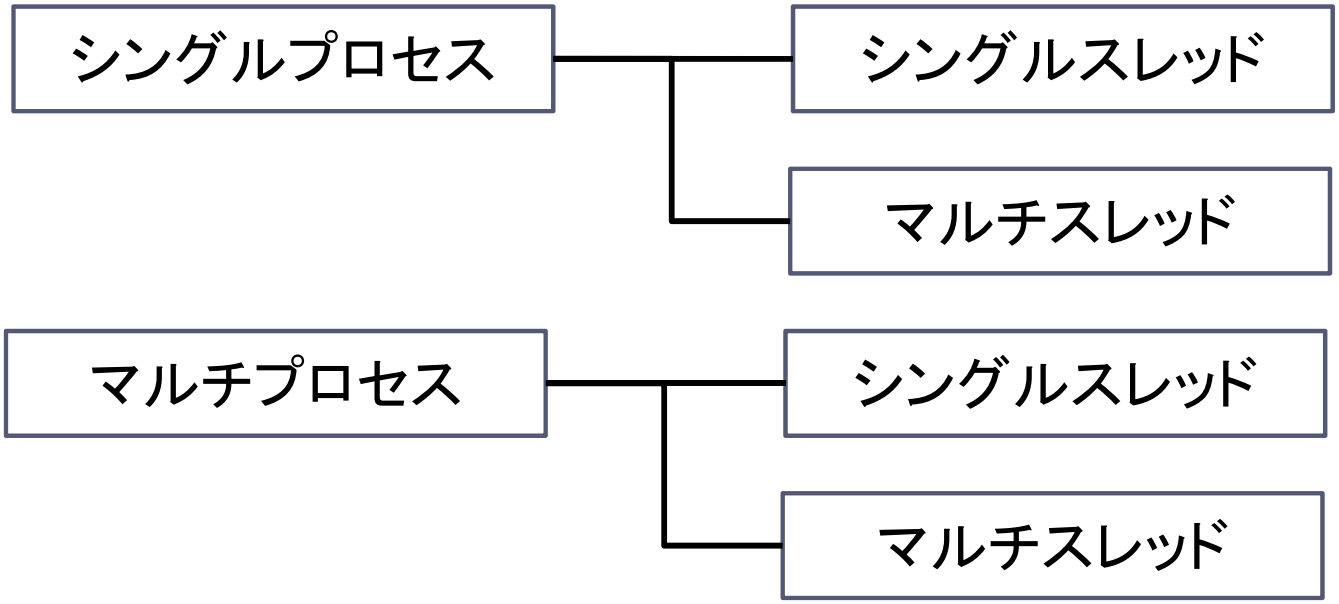
- 各プロセスに専用のメモリ領域を割り当て
  - CPUやメモリは複数のプログラムに割り当てられる
  - プログラムはCPUやメモリを独占しているように振る舞う



スレッド

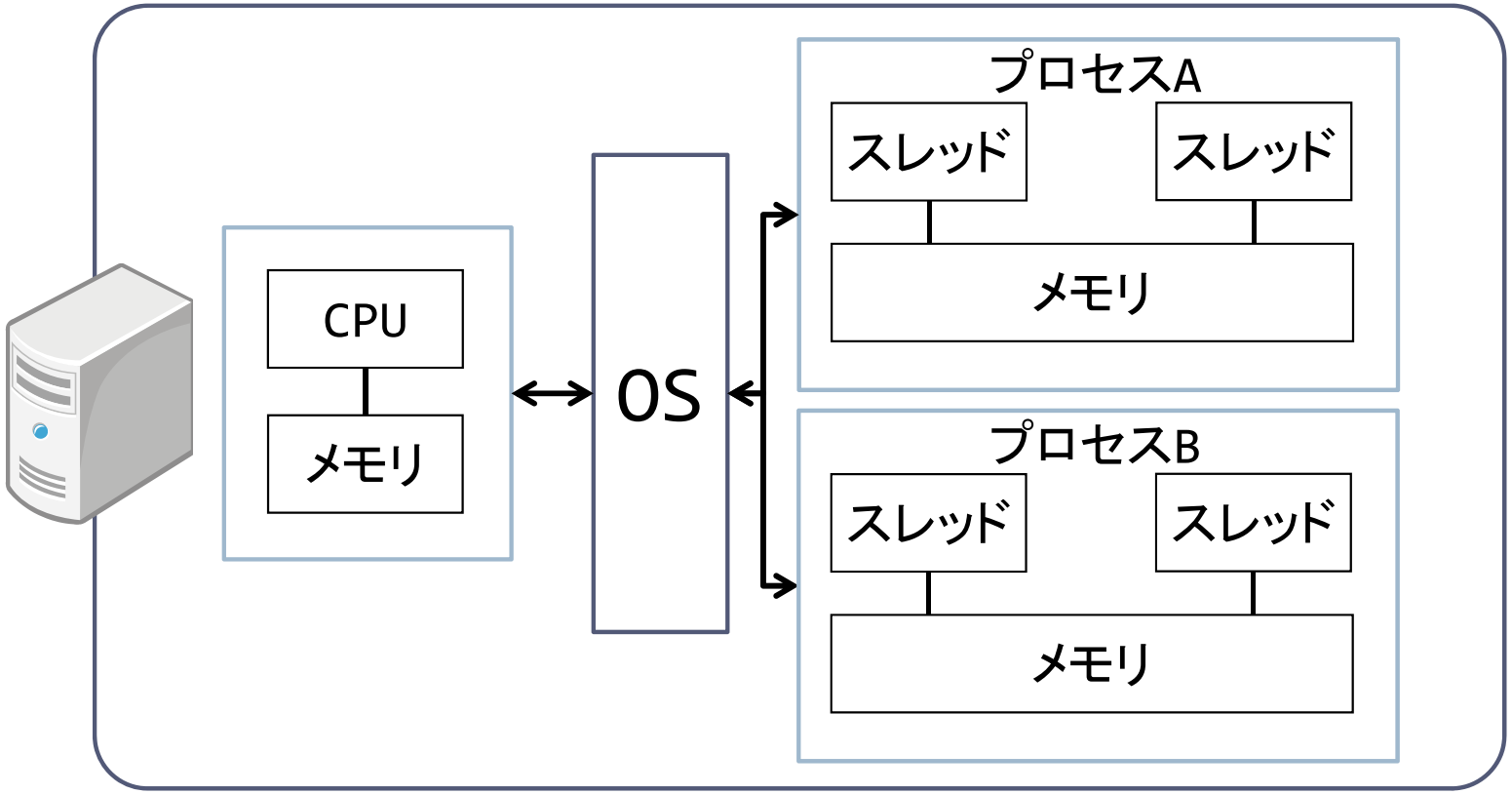
- プログラムの処理の最小実行単位

- プロセス内で複数のスレッドが存在
  - 1プロセスに一つ　　シングルスレッド
  - 1プロセスに二つ以上　　マルチスレッド



マルチスレッド

- 一つのプロセスに二つ以上のスレッドが存在
- 一つのプロセスには専用のメモリ領域が割り当てられる
  - プロセス内の複数のスレッドはメモリ領域を共有



並列処理と並行処理

- 並列処理　Parallel Processing
  - 一つの処理を複数の処理に分割
  - 複数の処理が協調しながら処理を実行
  - 真の並列処理　複数のプロセッサにそれぞれ一つのプロセス
  - 疑似並列処理　一つのプロセッサに複数のプロセス
- 並行処理　Concurrent Processing
  - 複数の処理を実行
  - 複数の処理は必ずしも協調していない

並列計算

- Parallel Computing

- 計算を複数に分割

- 各処理を計算機，プロセッサ，プロセス，スレッドが担当
  - 複数の計算機　　スーパーコンピュータ，PCクラスタ
  - 複数のプロセッサ　ワークステーション
  - 複数のコア　　マルチコアCPU，GPU

並列計算の方法

- マルチスレッド

- 計算を複数に分割し，一つの処理を一つのスレッドが担当
  - 複数のスレッドはメモリ領域を共有
  - 複数スレッドの協調処理は容易

- OpenMP (Open Multi-Processing)
  - プログラムに指示句（ディレクティブ）を挿入
  - ディレクティブで指定した箇所が自動で並列化
  - 非常にお手軽な並列化方法

並列計算の方法

- マルチプロセス

- 計算を複数に分割し，一つの処理を一つのプロセスが担当
  - 異なるプロセス同士はメモリの共有が不可能
  - 協調して処理を行うにはデータを共有する手段が必要

- MPI (Message Passing Interface)
  - 異なるプロセス間でデータを交換する仕組みを提供
  - 細かい命令を記述
  - 並列化に手間はかかるが高い性能を達成

マルチスレッド

対象

- 並列アーキテクチャ（SIMD）
  - CPUレベルでの処理の並列化
  - データの処理と命令の並列性に着目
- 並列計算機システム（共有メモリ計算機）
  - 計算機レベルでの処理の並列化
  - 計算機のメモリ構成に着目
- 並列処理（データ並列）
  - プログラムレベルでの処理の並列化
  - 処理の並列化の方法に着目

OpenMP

- 共有メモリシステムの並列処理に利用
- 標準化されたオープンな規格
- OpenMPをサポートしているコンパイラであれば同じ書き方が可能
- 並列化したい箇所をコンパイラに指示
  - ディレクティブを挿入
  - コンパイラが対応していなければコメントとして扱われる
  - 修正が最小限で済み，共通のソースコードで管理

OpenMPによる並列化

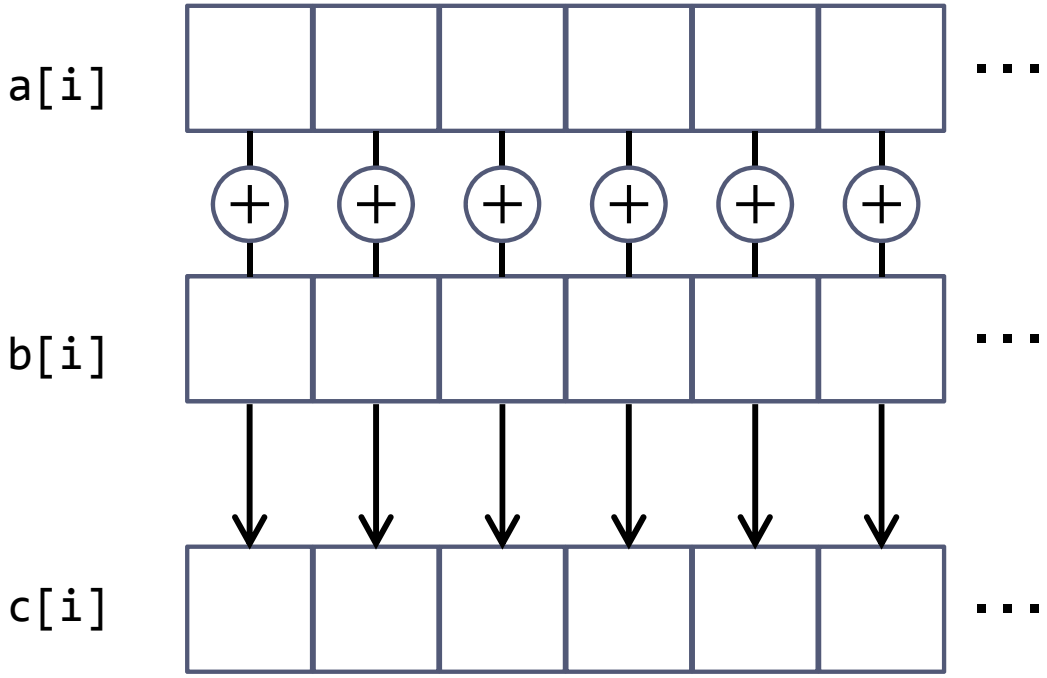
- 処理を並列実行したい箇所に指示句(ディレクティブ)を挿入

- for文の並列化
  - ディレクティブを一行追加(#pragma omp ～)

```
#pragma omp parallel for
for(int i=0; i<N; i++)
    C[i] = A[i] + B[i]
```

ベクトル和C=A+Bの計算

- 配列要素に対して計算順序の依存性がなく，最も単純に並列化可能



逐次（並列化前）プログラム

```
#include<stdio.h>

#define N (1024*1024)


int main(){
    float a[N],b[N],c[N];
    int i;




    for(i=0; i<N; i++){
        a[i] = 1.0;
        b[i] = 2.0;
        c[i] = 0.0;
    }
```

```
    for(i=0; i<N; i++)
        c[i] = a[i] + b[i];

    for(i=0; i<N; i++)
        printf("%f+%f=%f¥n",
                        a[i],b[i],c[i]);




    return 0;
}
```

vectoradd.c

逐次（並列化前）プログラム

```
#include<stdio.h>
#include<stdlib.h>
#define N (1024*1024)
#define Nbytes (N*sizeof(float))

int main(){
    float *a,*b,*c;
    int i;

    a = (float *)malloc(Nbytes);
    b = (float *)malloc(Nbytes);
    c = (float *)malloc(Nbytes);

    for(i=0; i<N; i++){
        a[i] = 1.0;
        b[i] = 2.0;
        c[i] = 0.0;
    }

    for(i=0; i<N; i++)
        c[i] = a[i] + b[i];

    for(i=0; i<N; i++)
        printf("%f+%f=%f¥n",
                        a[i],b[i],c[i]);

    free(a);
    free(b);
    free(c);
    return 0;
}
```

vectoradd_malloc.c

逐次（並列化前）プログラム

- malloc/free
  - 指定したバイト数分のメモリを確保/解放
  - stdlib.hをインクルードする必要がある

```
#include<stdlib.h>
main(){
    int *a;
    a = (int *)malloc( sizeof(int)*100 );
    free(a);
}
```

- sizeof
  - データ型1個のサイズ（バイト数）を求める

```
printf("%d, %d¥n", sizeof(float), sizeof(double));
実行すると4,8と表示される
```

並列化プログラム（スレッド並列）

```
#include<stdio.h>
#include<stdlib.h>
#define N (1024*1024)
#define Nbytes (N*sizeof(float))
#include<omp.h>
int main(){
    float *a,*b,*c;
    int i;

    a = (float *)malloc(Nbytes);
    b = (float *)malloc(Nbytes);
    c = (float *)malloc(Nbytes);

  //複数スレッドで処理する領域を指定
  //この指定だけだと全スレッドが同じ処理を行う
  #pragma omp parallel
  {
    #pragma omp for//for文を分担して実行
    for(i=0; i<N; i++){
        a[i] = 1.0;
        b[i] = 2.0;
        c[i] = 0.0;
    }
    #pragma omp for//for文を分担して実行
    for(i=0; i<N; i++)
        c[i] = a[i] + b[i];
  }

    for(i=0; i<N; i++)
        printf("%f+%f=%f¥n",
                        a[i],b[i],c[i]);

    free(a);
    free(b);
    free(c);
    return 0;
}
```

vectoradd.cpp

処理の並列化

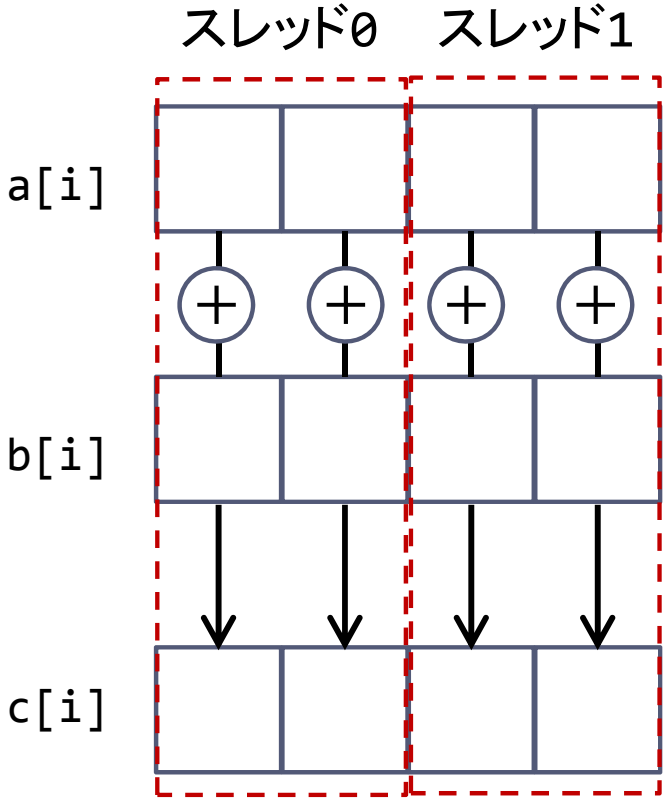
- データ並列
  - forループをスレッドの数だけ分割
  - 分割されたforループを各スレッドが実行
  - 実行時間は1/スレッド数になると期待

スレッド0

```
for(i=0; i<N/2-1; i++)
    c[i] = a[i] + b[i];
```

スレッド1

```
for(i=N/2; i<N; i++)
    c[i] = a[i] + b[i];
```



プログラムのコンパイルと実行

- コンパイル時にコンパイルオプションを付与

  `-fopenmp`

- `-fopenmp`を付けるとディレクティブを処理

- `-fopenmp`を付けないとディレクティブは処理されない

- grouseでは
  - ソースファイルの拡張子は`.c`ではなく`.cpp`
  - コンパイラは`cc`ではなく`g++`
  - `g++ -fopenmp vectoradd.cpp`

速度向上率

- N個のCPU･コア･PCで並列処理
  - 理想的には1個で処理した時のN倍高速化
- 処理時間に着目して評価
- 速度向上率

$$S = \frac{T(1)}{T(N)}$$

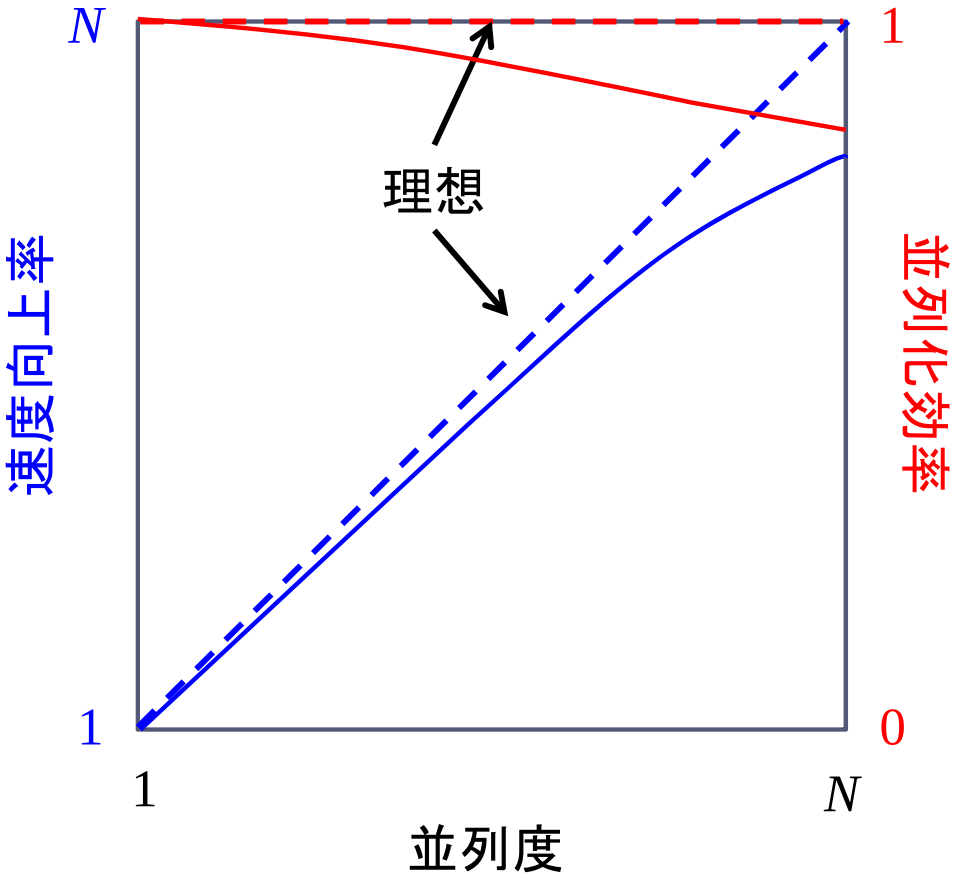
$T(1)$ ： 1個のCPU･コア･PCで処理した時間
$T(N)$ ： N個のCPU･コア･PCで処理した時間

- 並列化効率

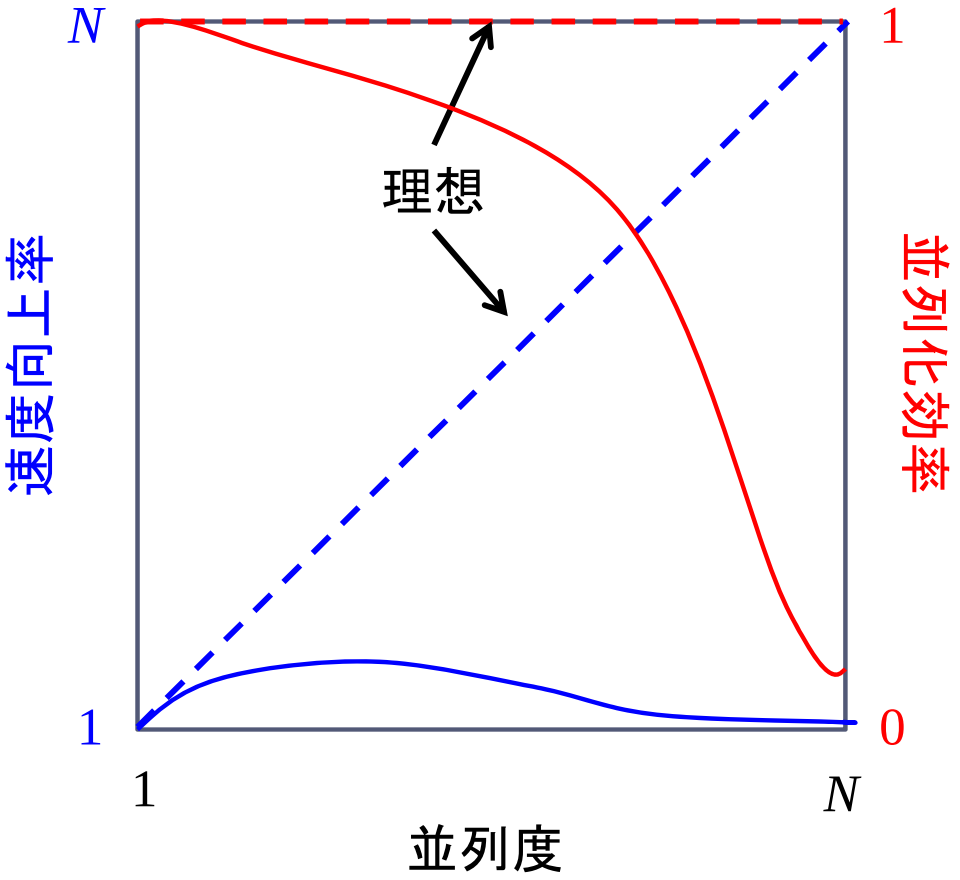
$$E = \frac{S(N)}{N}$$

速度向上率

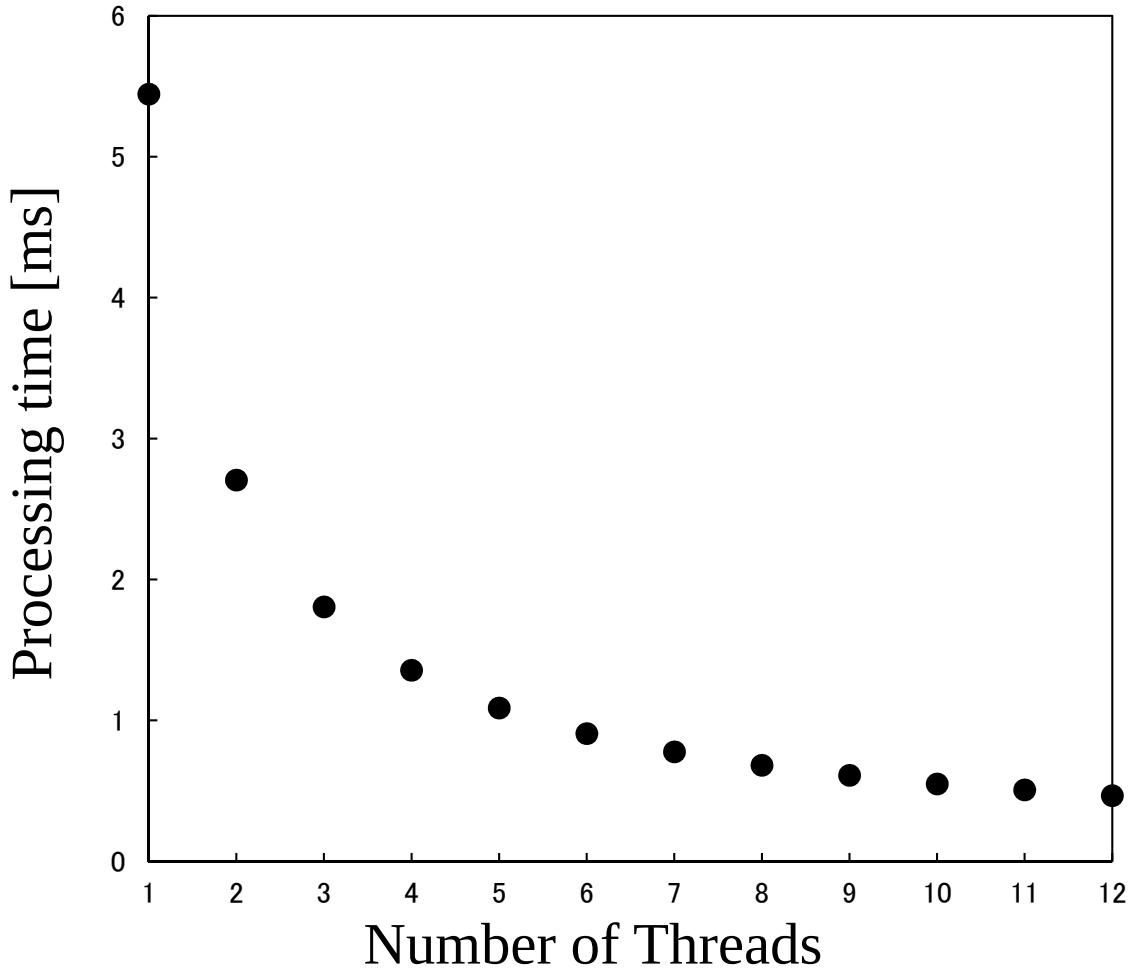
よい計算機，計算アルゴリズム，プログラム



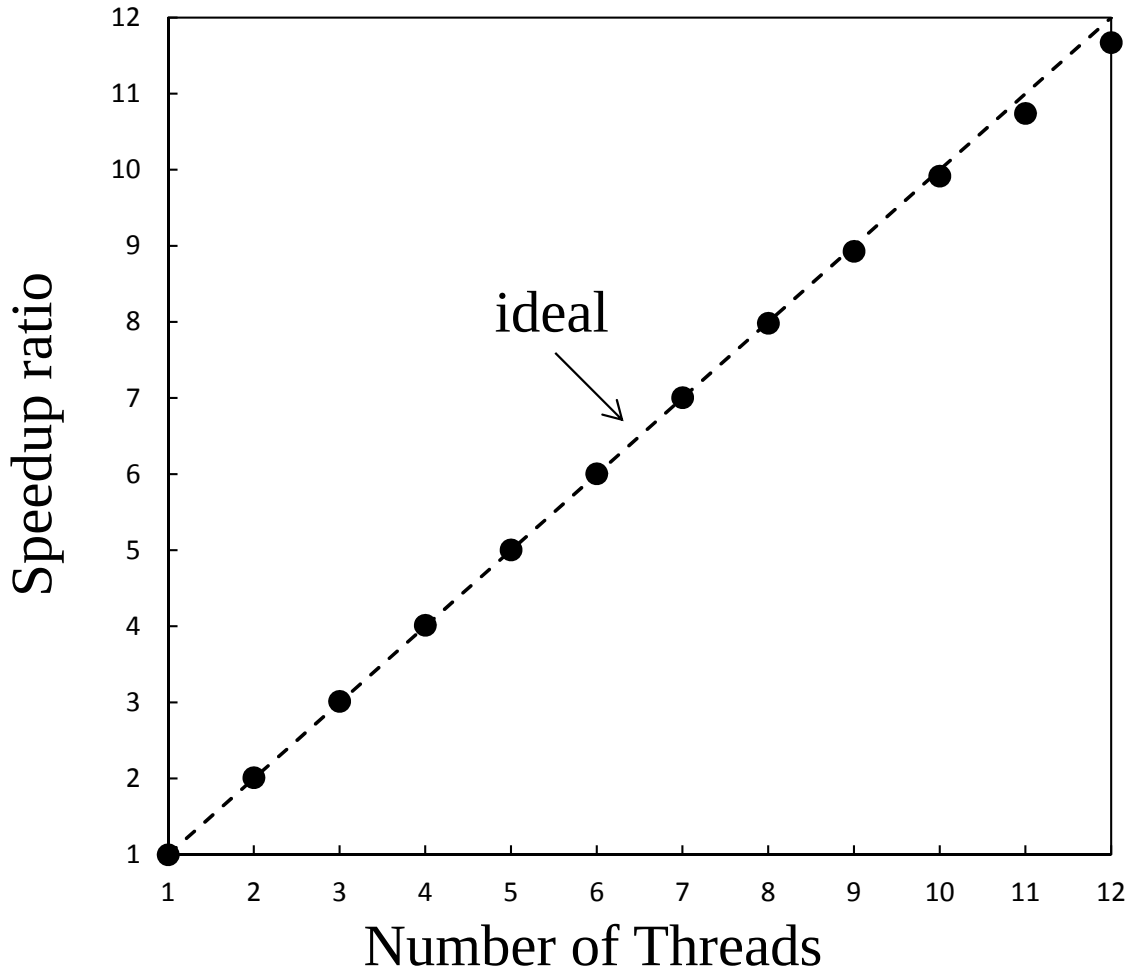
悪い計算機，計算アルゴリズム，プログラム



計算時間の変化


Processing time [ms]
0
1
2
3
4
5
6
1
2
3
4
5
6
7
8
9
10
11
12
Number of Threads

速度向上率



マルチプロセス

対象

- 並列アーキテクチャ（MIMD, SPMD）
  - CPUレベルでの処理の並列化
  - データの処理と命令の並列性に着目
- 並列計算機システム（分散メモリ計算機）
  - 計算機レベルでの処理の並列化
  - 計算機のメモリ構成に着目
- 並列処理（データ並列）
  - プログラムレベルでの処理の並列化
  - 処理の並列化の方法に着目

Single Program Multiple Data streams

- 単一プログラム複数データ
- MIMDシステムのプログラミング手法
- 現在のスーパーコンピュータやPCクラスタでプログラムを作る際の標準的な手法
- 各PC用にプログラムを作らず，一つのプログラムの中で役割を認識

MPI (Message Passing Interface)

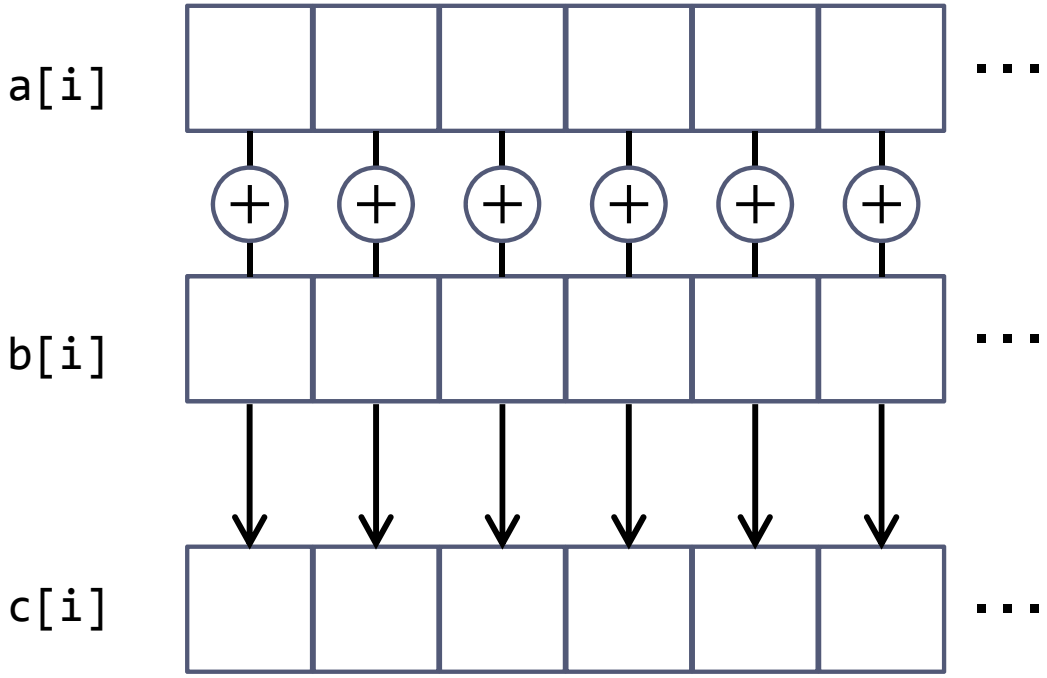
- メッセージ通信用ライブラリの標準規格の一つ
- 分散メモリシステムの並列処理に利用
  - 共有メモリシステムでも動作
- 共有できないメモリ内のデータを，ネットワークを介して送受信
  - 送受信されるデータをメッセージと呼ぶ

MPIの処理の概要

- 通信を行うノード（計算機，CPU，コアなど）の集合を定義
- 各ノードに0からn-1（nはノード総数）の番号を付与
- 各ノードでどのような処理を行うかを記述
  - 通常の四則演算や関数呼出
  - メッセージ通信
    - どのノードへデータを送るか，どのノードからデータを受け取るか
- 各ノード用にプログラムを書くのではなく，一つのプログラム内で各ノードが行う処理を書く

ベクトル和C=A+Bの計算

- 配列要素に対して計算順序の依存性がなく，最も単純に並列化可能



逐次（並列化前）プログラム

```
#include<stdio.h>
#include<stdlib.h>
#define N (1024*1024)
#define Nbytes (N*sizeof(float))

int main(){
    float *a,*b,*c;
    int i;

    a = (float *)malloc(Nbytes);
    b = (float *)malloc(Nbytes);
    c = (float *)malloc(Nbytes);

    for(i=0; i<N; i++){
        a[i] = 1.0;
        b[i] = 2.0;
        c[i] = 0.0;
    }

    for(i=0; i<N; i++)
        c[i] = a[i] + b[i];

    for(i=0; i<N; i++)
        printf("%f+%f=%f¥n",
                        a[i],b[i],c[i]);

    free(a);
    free(b);
    free(c);
    return 0;
}
```

vectoradd_malloc.c

並列化プログラム（プロセス並列）

```
#include<stdlib.h>
#include<mpi.h>
#define N (1024*1024)

int main(int argc, char* argv[]){
    float *a, *b, *c;
    int i, nsize, rank, nproc, bytes;

    //MPIのライブラリを初期化して実行準備
    MPI_Init(&argc, &argv);
    //全ノード数を取得
    MPI_Comm_size(MPI_COMM_WORLD,
                                  &nproc);
    //各ノードに割り振られた固有の番号を取得
    MPI_Comm_rank(MPI_COMM_WORLD,
                                  &rank);
    //各ノードで処理するデータサイズを設定
    nsize = N/nproc;
    if(rank==nproc-1)nsize+=N%nproc;
    bytes = sizeof(float)*nsize;
    a=(float *)malloc(bytes);//各ノード
    b=(float *)malloc(bytes);//でメモリを
    c=(float *)malloc(bytes);//確保
    //各ノードで配列の初期化と加算を実行
    for(i=0;i<nsize;i++){
        a[i]=1.0;
        b[i]=2.0;
        c[i]=0.0;
    }
    for(int i=0;i<nsize;i++)
        c[i] = a[i]+b[i];

    //全ノードの同期を取る
    MPI_Barrier(MPI_COMM_WORLD);
    free(a);
    free(b);
    free(c);
    MPI_Finalize();//MPIのライブラリを終了
    return 0;
}
```

vectoradd_mpi.cpp

MPIライブラリの内容

- MPI_Init
  - MPIのライブラリを初期化
  - MPI_COMM_WORLDという名前のノードの集合を用意
- MPI_Comm_rank(MPI_COMM_WORLD, &rank)
  - 各ノードに割り振られた固有の番号を取得
- MPI_Comm_size(MPI_COMM_WORLD, &nproc)
  - 全ノード数を取得
- MPI_Barrier(MPI_COMM_WORLD)
  - 各ノード間の同期をとる
- MPI_Finalize
  - MPIのライブラリを終了

処理の並列化

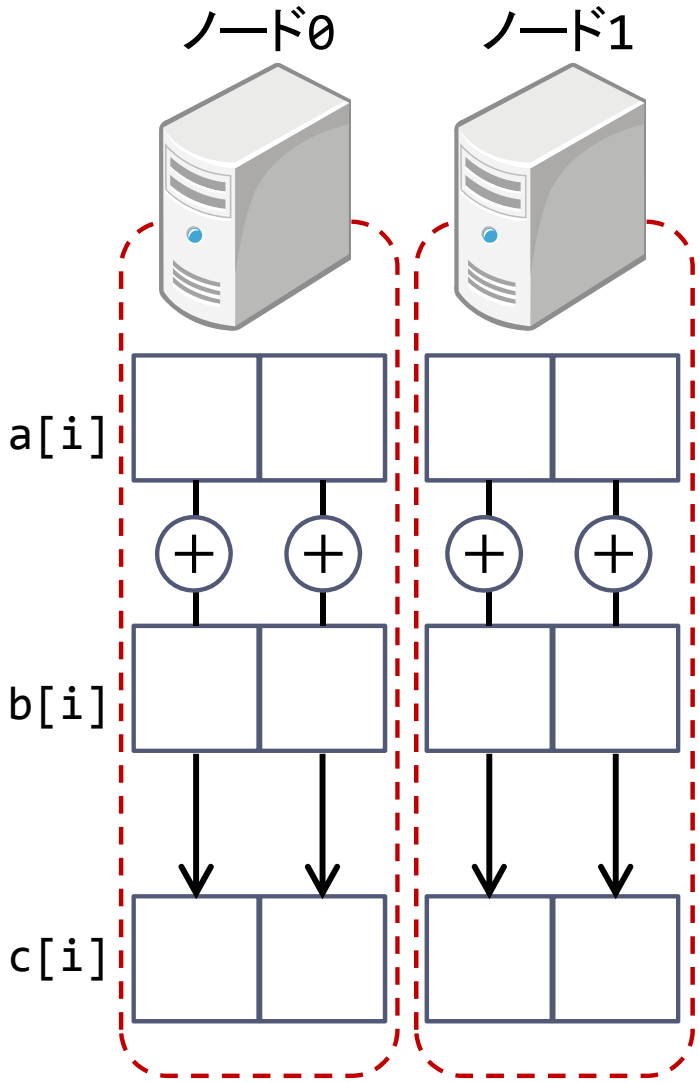
- データ並列
  - forループをスレッドの数だけ分割
  - 分割されたforループを各ノードが実行
  - 実行時間は1/スレッド数になると期待

ノード0

```
for(i=0; i<N/2-1; i++)
    c[i] = a[i] + b[i];
```

ノード1

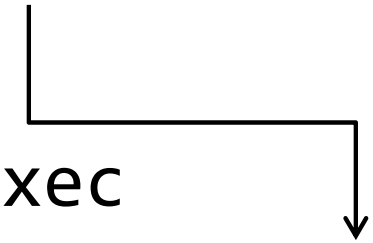
```
for(i=0; i<N/2-1; i++)
    c[i] = a[i] + b[i];
```



プログラムのコンパイルと実行

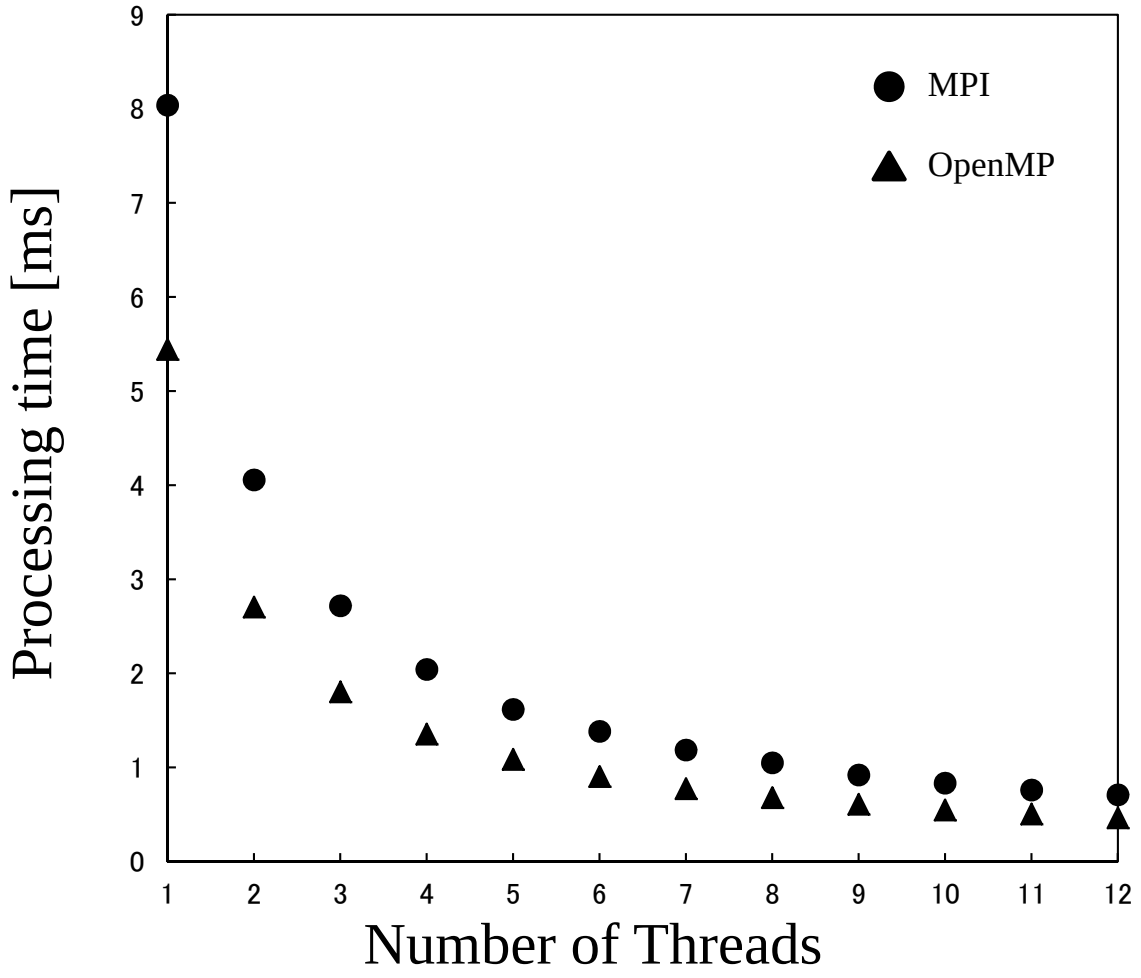
- コンパイル，実行にはMPI用のコンパイラや実行プログラムを利用

- コンパイル　mpic++
  - mpic++ vectoradd_mpi.cpp
  - 標準で a.out という実行ファイルを作成（これまでと同じ）

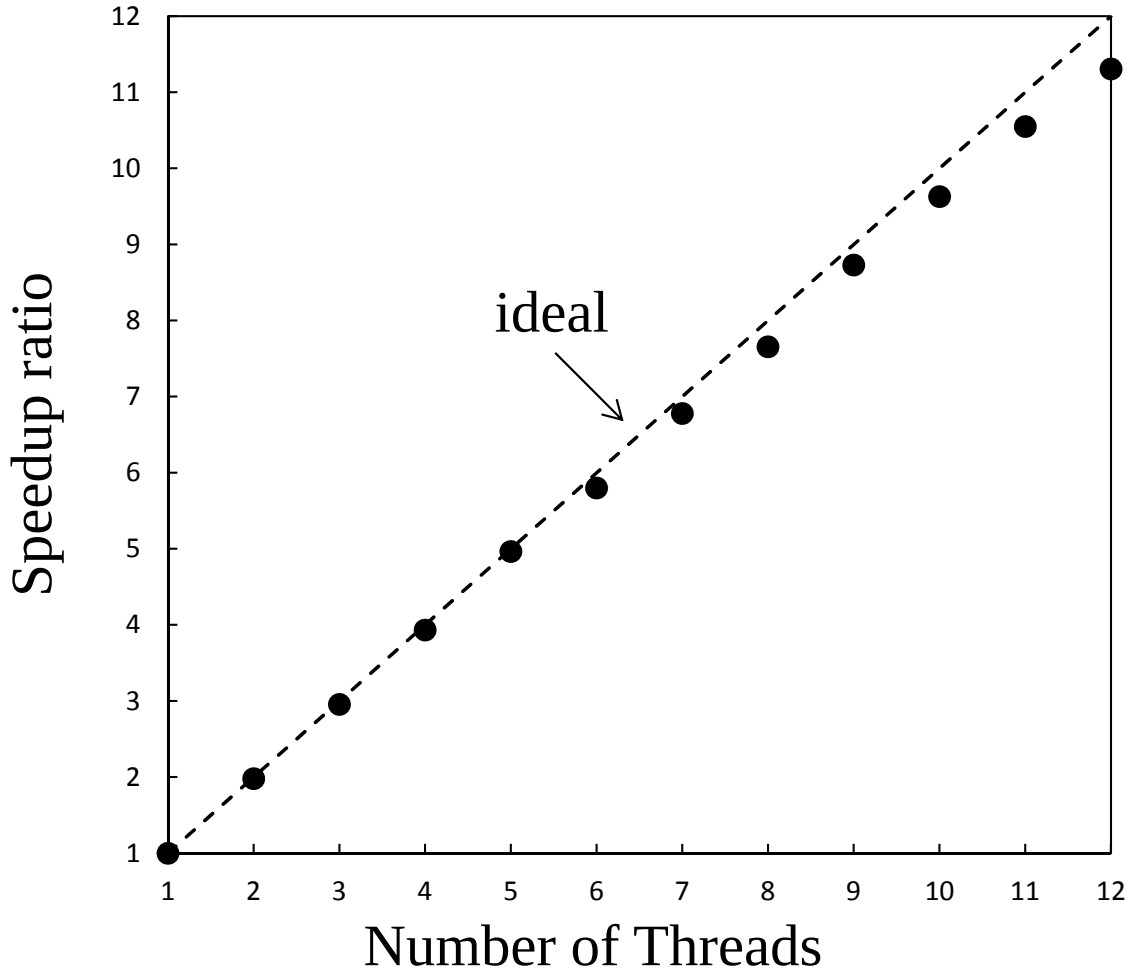


- 実行 mpiexec
  - mpiexec –n 4 ./a.out
  - -n 4　　プロセスの数を4で実行
  - grouseでは -mca btl ^openib というオプションも必要

計算時間の変化


MPI
OpenMP
Processing time [ms]
Number of Threads
0
1
2
3
4
5
6
7
8
9
1
2
3
4
5
6
7
8
9
10
11
12

高速化率

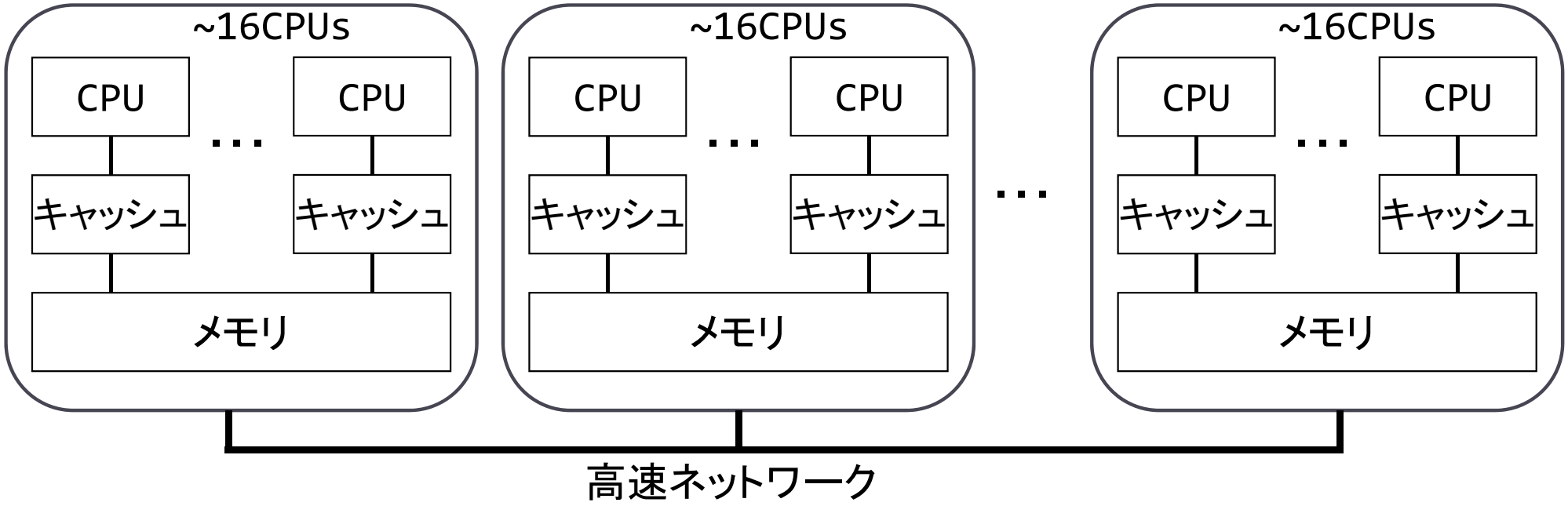


OpenMPとの比較

- OpenMPよりも若干実行に時間がかかる
  - MPIを利用するための準備が必要
- OpenMPの並列化の上限は1ノード内のCPUのコア数
- MPIの並列化の上限は計算機システム上のCPUの全コア数
- 大規模な計算をするにはMPIしか選択肢がない

ハイブリッドシステム

- 大規模な分散メモリシステム
  - MPP : Massively Parallel Processor
- 1ノードが共有メモリシステムからなるMPP
  - SMP/MPPハイブリッドシステム



ハイブリッドシステムでのプログラミング

- 大規模な計算ではMPIしか選択肢がない

- 全てMPIでプログラムを作成(Flat MPI)
  - メモリが共有できないノード間ではMPI
  - メモリが共有できるノード内もMPI

- MPIとOpenMPでプログラムを作成(Hybrid MPI)
  - メモリが共有できないノード間ではMPI
  - メモリが共有できるノード内はOpenMP

並列処理の性能評価

- 速度向上率
- アムダールの法則(Amdahl's law)
- スケーリング
  - 強いスケーリング(strong scaling)
  - 弱いスケーリング(weak scaling)

速度向上率

- N個のCPU･コア･PCで並列処理
  - 理想的には1個で処理した時のN倍高速化
  - N倍の基準は？　処理時間？FLOPS？

- 速度向上率
  - 処理時間に着目して評価

$$S = \frac{T(1)}{T(N)}$$

$T(1)$ ： 1個のCPU･コア･PCで処理した時間
$T(N)$ ： N個のCPU･コア･PCで処理した時間

- 並列化効率

$$E = \frac{S(N)}{N}$$

アムダールの法則

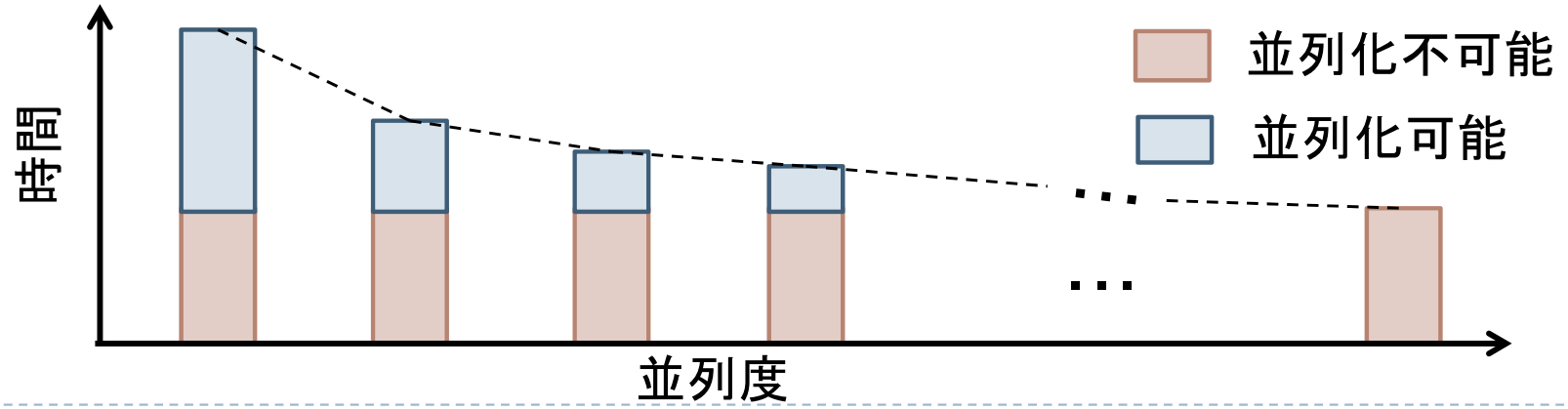
- 並列度を大きくした際の性能向上の予測
  - 問題のサイズは固定
  - 並列化が不可能な箇所の割合によって性能向上の上限が決定

$$S = \frac{T(1)}{T(N)}$$

$T(1)$ ： 1個のCPU･コア･PCで処理した時間
$T(N)$ ： N個のCPU･コア･PCで処理した時間

$$T(N) = \left[(1-\alpha) + \frac{\alpha}{N} \right] \times T(1)$$

α ： プログラム中で並列化可能な箇所の割合



スケーリング（scaling）

- いくつかの条件を固定したとき，速度向上率が並列度に比例すること
- スケーラビリティ(scalability)
  - スケーリングの度合い

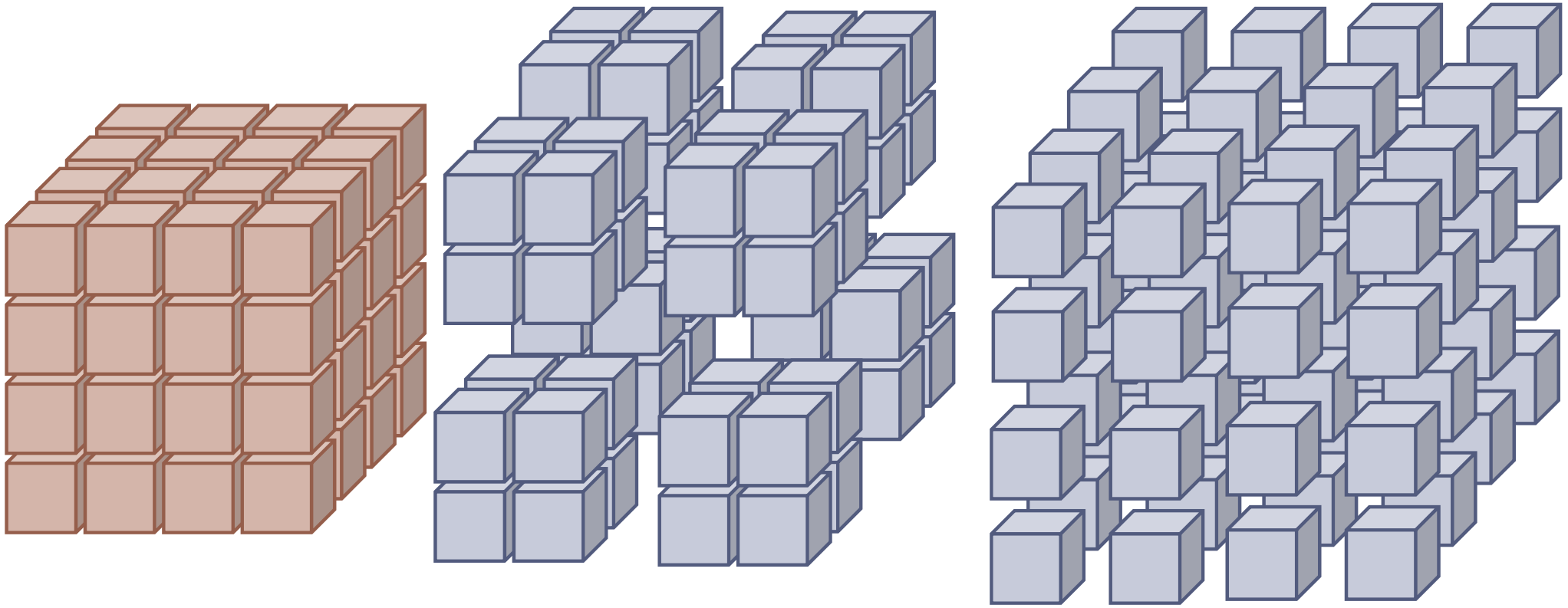
- ストロング（強い）スケーリング(strong scaling)
  - 問題の大きさを固定したときのスケーラビリティ

- ウィーク（弱い）スケーリング(weak scaling)
  - 各CPU，コア，PCの負荷を固定したときのスケーラビリティ
  - 問題の大きさは並列度に比例して大規模化

ストロング（強い）スケーリング

- 問題の大きさを固定して並列度を増加
  - 並列度の増加に伴って一つのCPU，コア，PCの負荷が減少

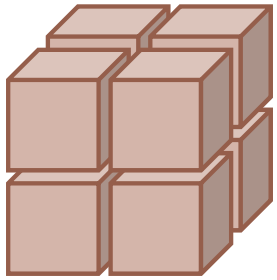
並列度0　並列度8　並列度64



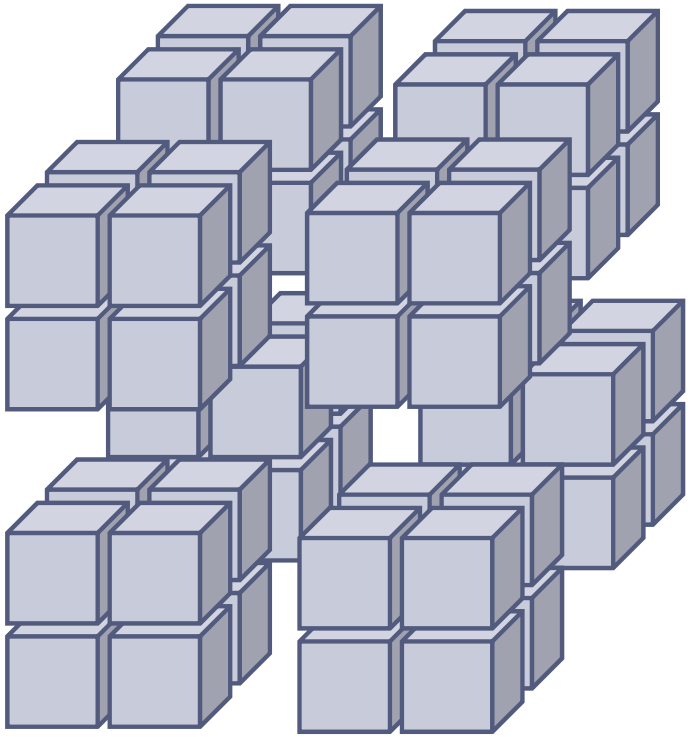
ウィーク（弱い）スケーリング

- 各CPU，コア，PCの負荷を固定したときのスケーラビリティ
  - 並列度の増加に伴って問題が大規模化

並列度0



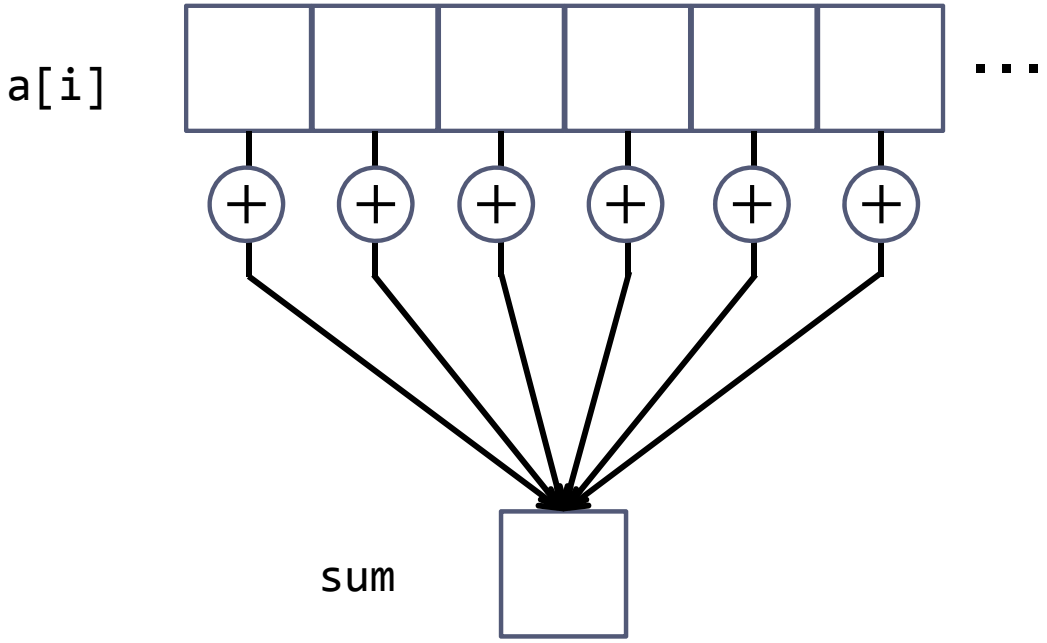
並列度8



付録

OpenMPによる総和計算の並列化

- 計算順序に依存性はないが，出力結果に依存性がある



逐次（並列化前）プログラム

```
#include<stdio.h>
#include<stdlib.h>

#define N 100
#define Nbytes (N*sizeof(float))

int main(){
    float *a,sum=0.0f;
    int i;

    a = (float *)malloc(Nbytes);

    for(i=0; i<N; i++)
        a[i] = (float)(i+1);

    for(i=0; i<N; i++)
        sum += a[i];

    printf("%f¥n",sum);

    return 0;
}
```

sum_serial.c

並列化プログラム（スレッド並列）

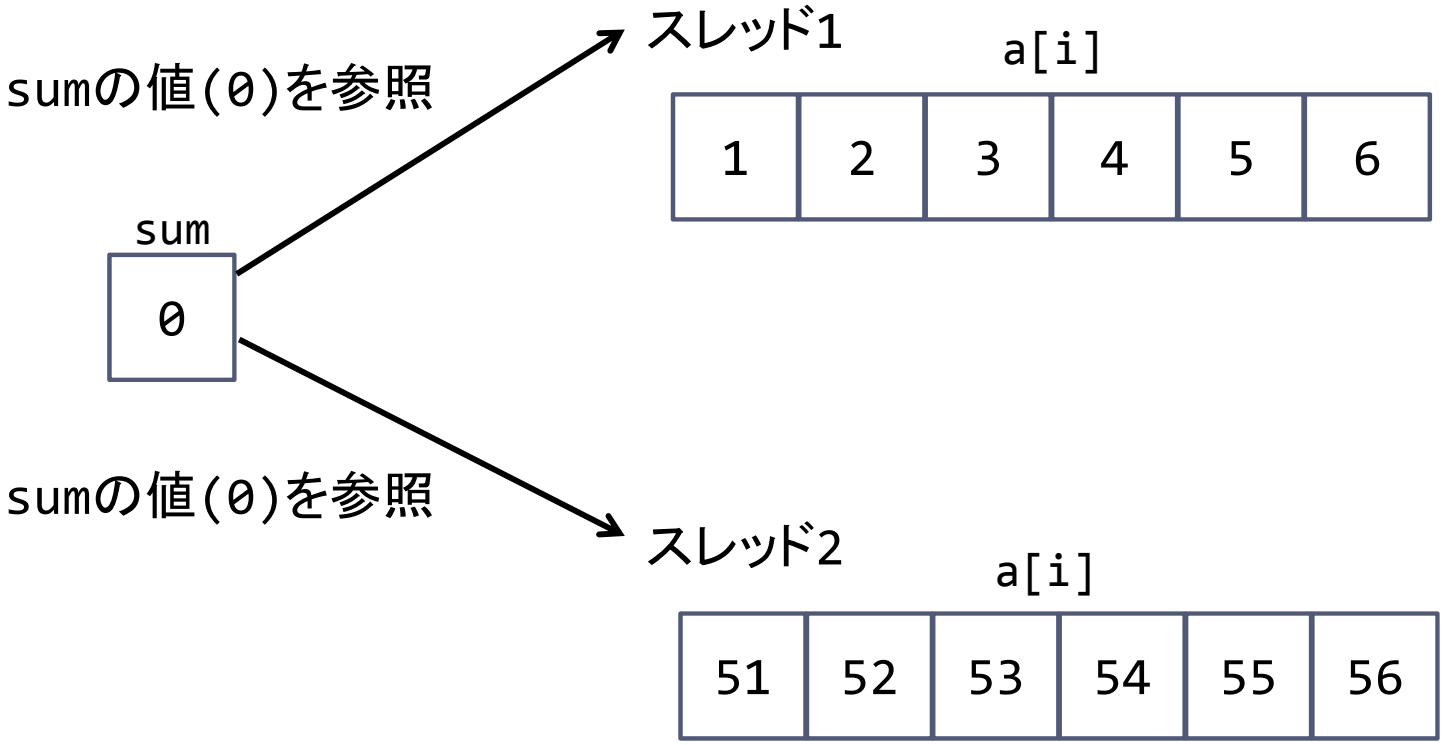
```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
#define N 100
#define Nbytes (N*sizeof(float))

int main(){
    float *a, sum=0.0f;
    int i;

    a = (float *)malloc(Nbytes);
    #pragma omp parallel for
    for(i=0; i<N; i++)
        a[i] = (float)(i+1);

    #pragma omp parallel for ¥
    num_threads(4) reduction(+:sum)
    for(i=0; i<N; i++)
        sum += a[i];

    printf("%f¥n",sum);

    return 0;
}
```

```
//num_threads()でスレッド数を設定
//reduction(+:sum)でsumに値の合計を集約
```

sum.cpp

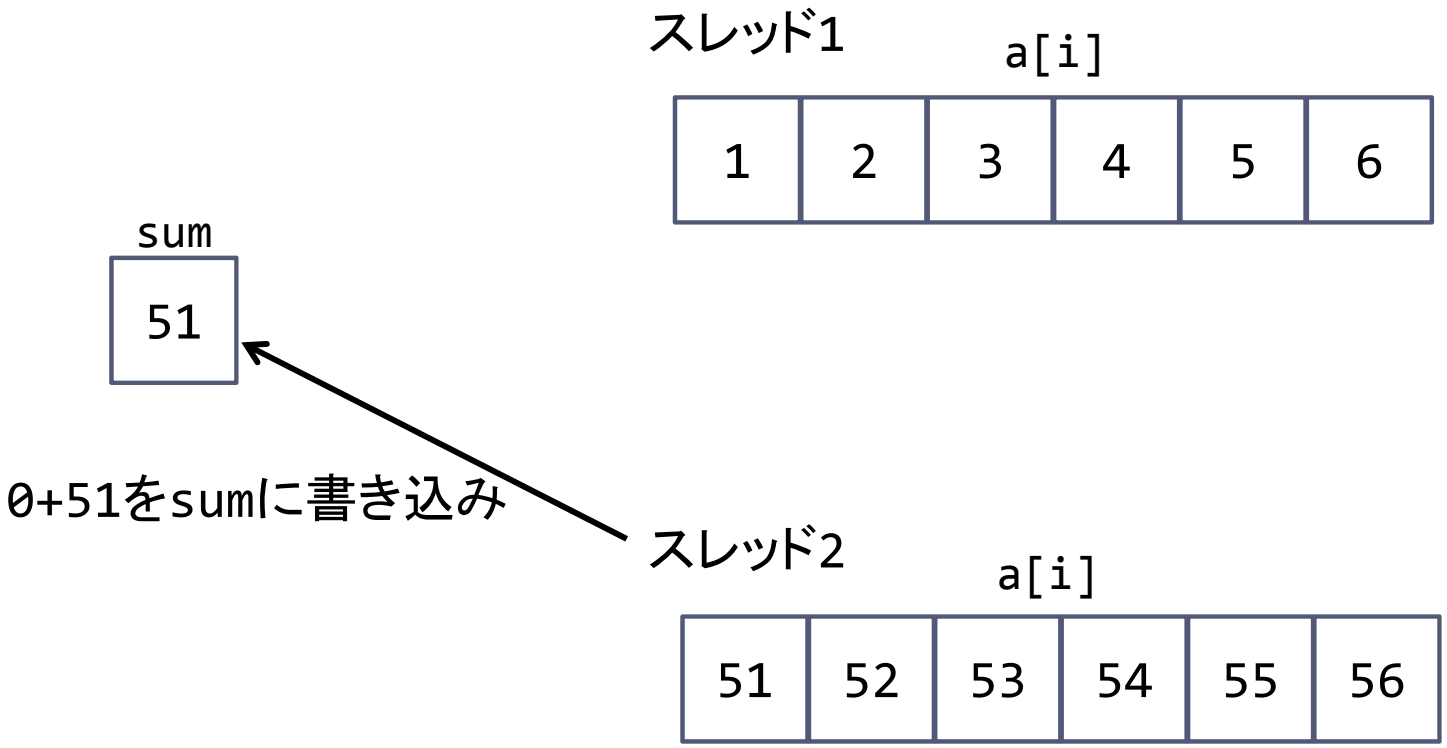
データ競合

- reduction()を付けないとどうなるか
  - 実行結果が毎回変わる
  - データ競合（データレース）が発生



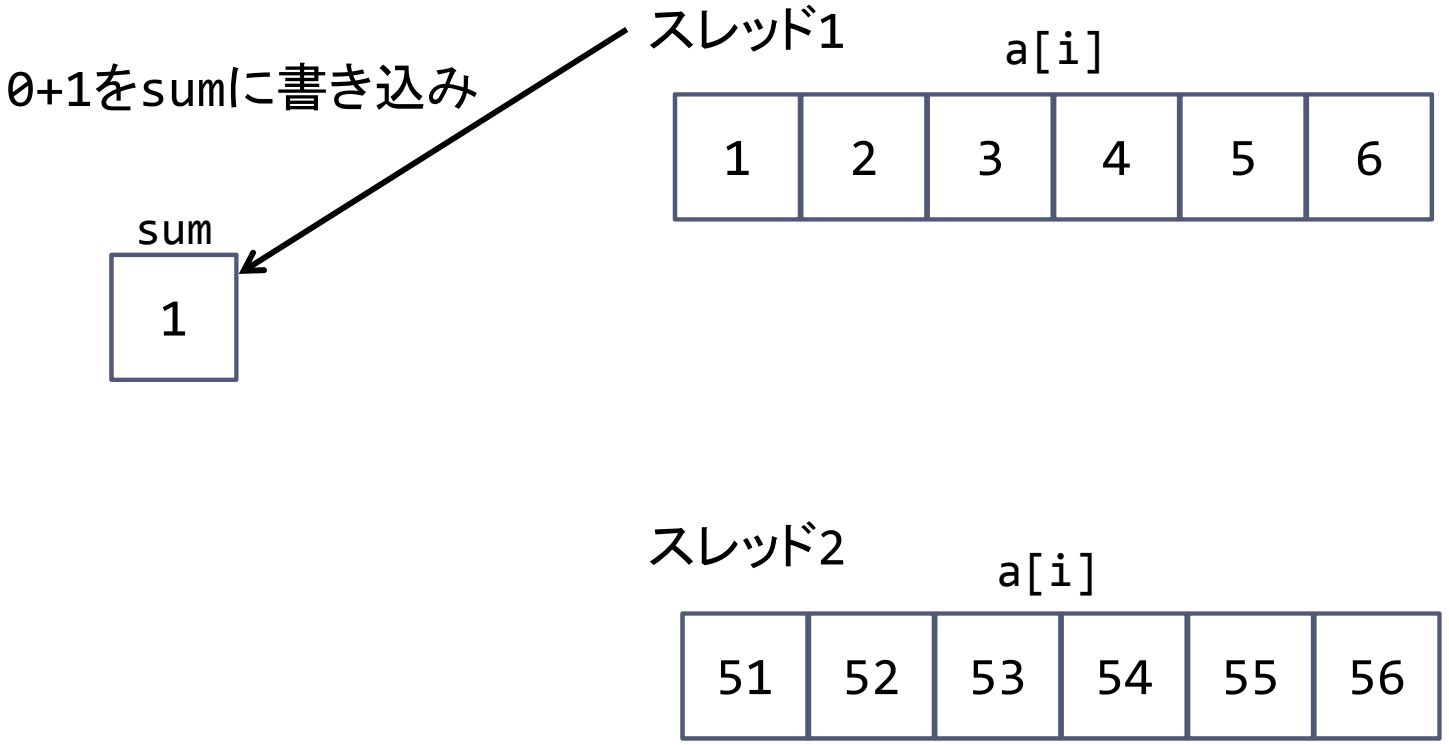
データ競合

- reduction()を付けないとどうなるか
  - 実行結果が毎回変わる
  - データ競合（データレース）が発生



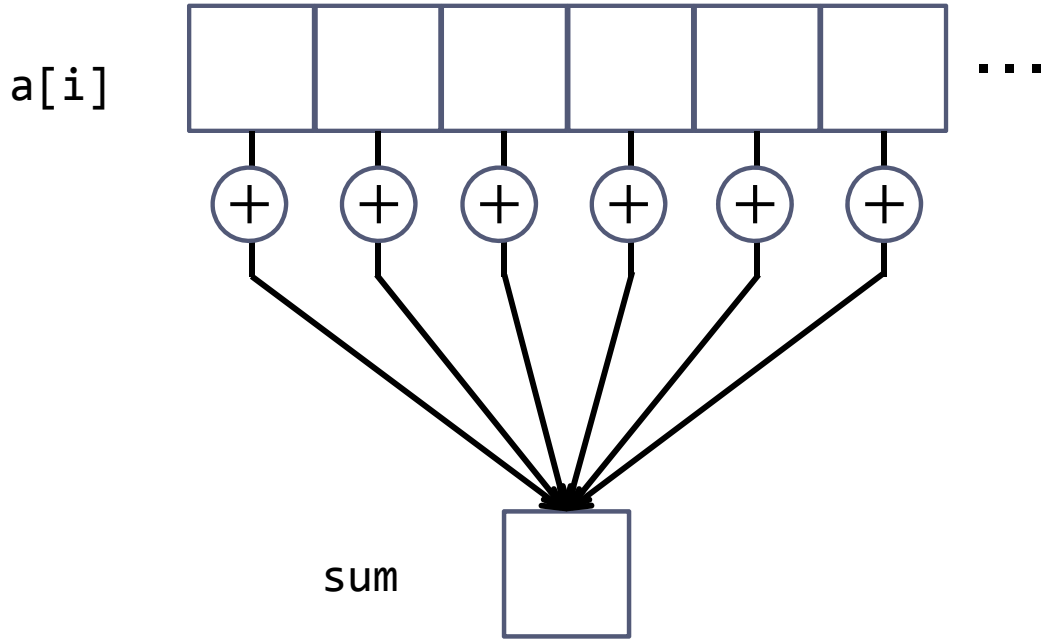
データ競合

- reduction()を付けないとどうなるか
  - 実行結果が毎回変わる
  - データ競合（データレース）が発生



MPIによる総和計算の並列化

- 計算順序に依存性はないが，出力結果に依存性がある
  - OpenMPの時は計算順序が問題
  - MPIの時はノード間通信が必要



並列化プログラム（プロセス並列）

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>

#define ROOT 0 //rank 0のノードは特別扱い
#define N 100

int main(int argc, char* argv[]){
    float *a,sum;
    int i, nsize, rank, nproc,bytes;
    int src, dst,tag;
    float sum_rcv;
    MPI_Status stat;

    //MPIのライブラリを初期化して実行準備
    MPI_Init(&argc, &argv);
    //全ノード数を取得
    MPI_Comm_size(MPI_COMM_WORLD,
                                  &nproc);
```

```
    //各ノードに割り振られた固有の番号を取得
    MPI_Comm_rank(MPI_COMM_WORLD,
                                  &rank);

    //各ノードで処理するデータサイズを設定
    nsize = N/nproc;
    if(rank==nproc-1) nsize+=N%nproc;

    //各ノードでメモリを確保
    bytes = sizeof(float)*nsize;
    a=(float *)malloc(bytes);

    //値の設定
    for(i=0;i<nsize;i++)
       a[i]=(float)(i+rank*N/nproc+1);

    MPI_Barrier(MPI_COMM_WORLD);
```

sum_mpi.cpp

並列化プログラム（プロセス並列）

```
//各ノードで配列の総和を求める
sum=0.0;
for(i=0;i<nsize;i++)
    sum += a[i];

//全ノードの同期を取る
MPI_Barrier(MPI_COMM_WORLD);

//各ノードの総和の値をrank0に送る
if(rank != ROOT){
    //rank0以外のノードはrank0に
    //sumの値を送信
    dst=ROOT;
    tag=0;
    MPI_Send(&sum, 1, MPI_FLOAT,
        dst, tag, MPI_COMM_WORLD);
}else{
    //rank0はそれ以外の全ノードから
    //送られてくる値を受信し，総和を計算
    for(src=ROOT+1; src < nproc;
                                src++){
        tag=0;
        MPI_Recv(&sum_rcv, 1,
            MPI_FLOAT, src, tag,
            MPI_COMM_WORLD, &stat);
        sum +=sum_rcv;
    }
}

if(rank == ROOT)
    printf("%f¥n",sum);

free(a);
MPI_Finalize();//MPIのライブラリを終了
return 0;
}
```

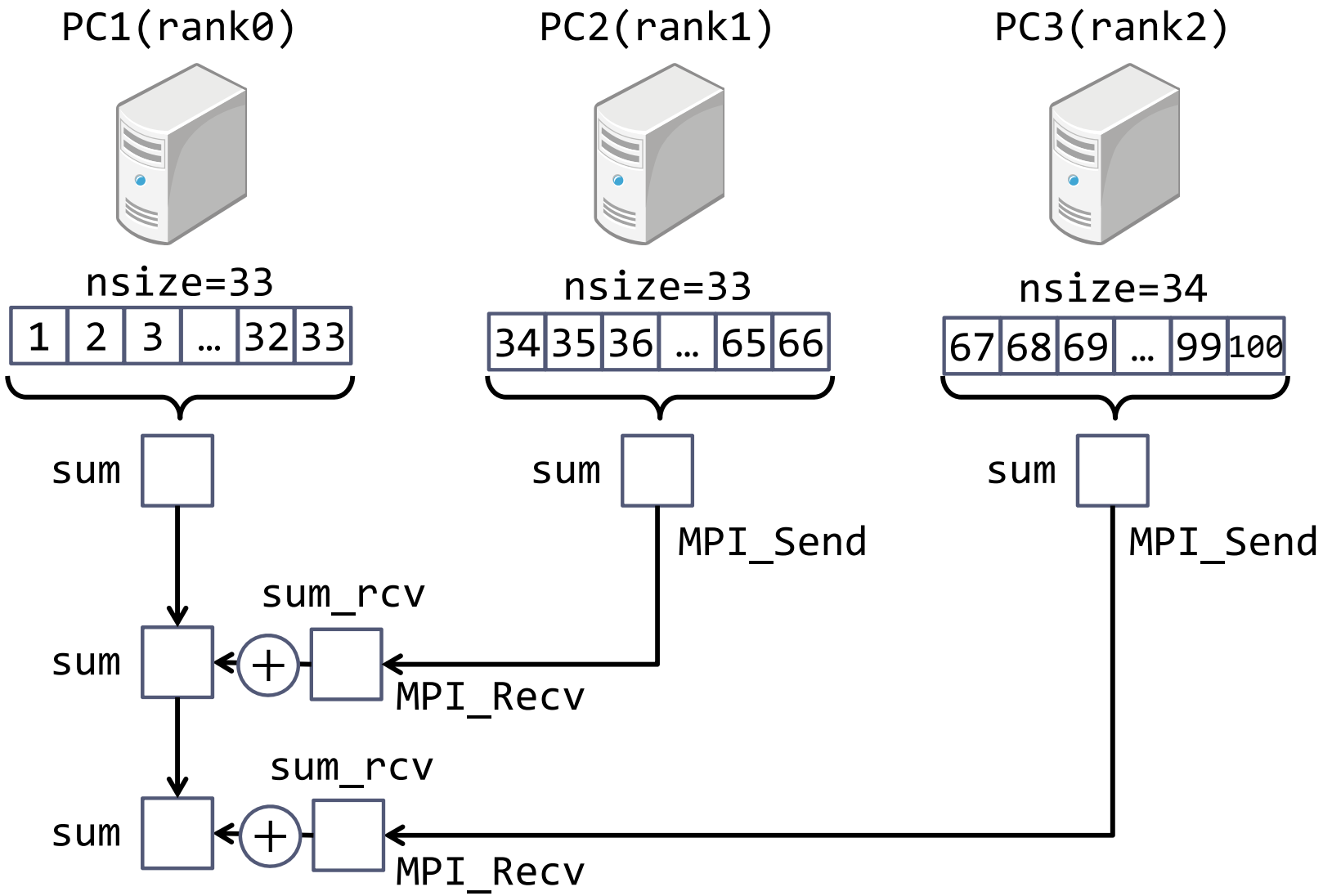
sum_mpi.cpp

MPIライブラリの内容

- MPI_Send
  - 指定したノードへメッセージを送信
  - 送信したメッセージが正常に受信されるまで停止（同期実行）

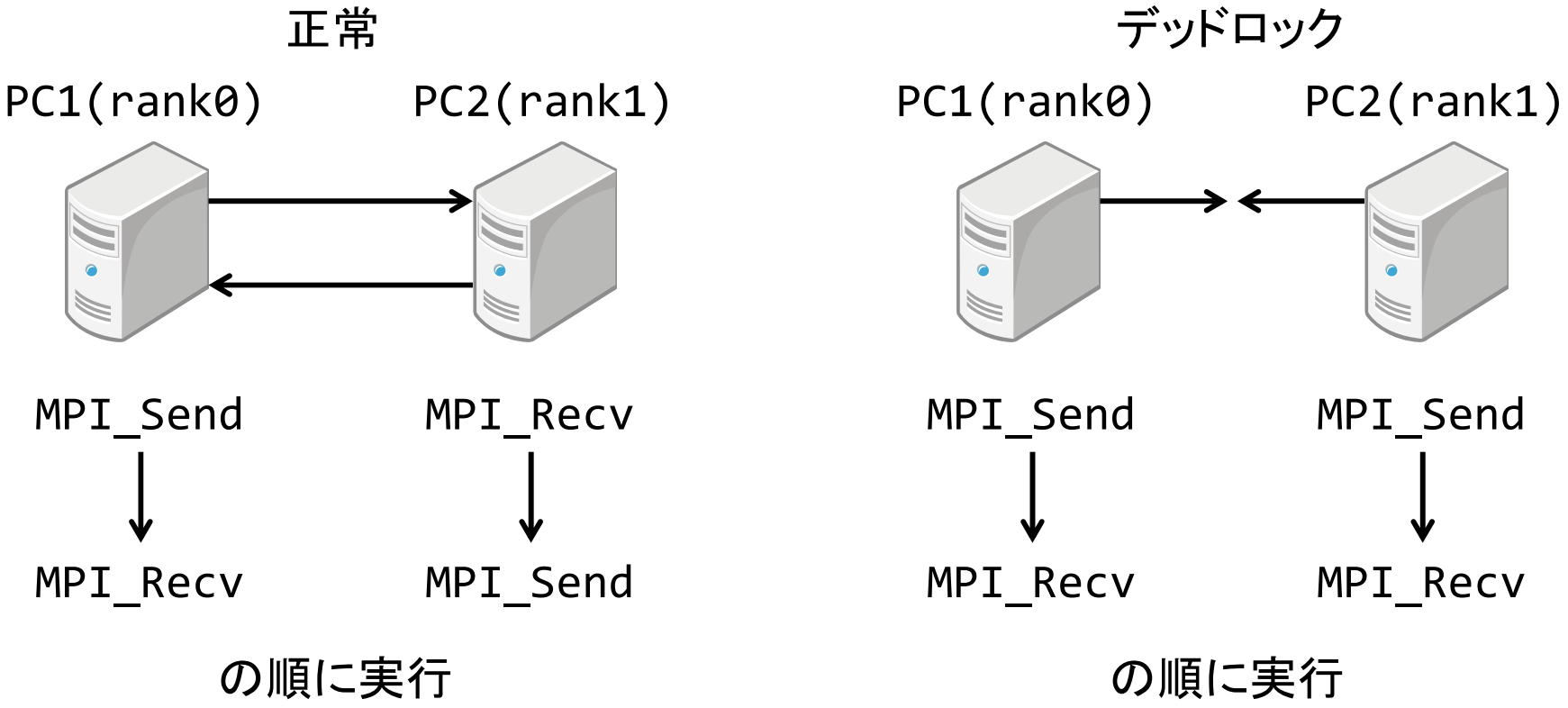
- MPI_Recv
  - 指定したノードからメッセージを受信
  - 送信された（されてくるはずの）メッセージを正常に受信するまで停止（同期実行）

実行イメージ（1から100までの総和）



デッドロック

- ノード間のデータ交換
  - 処理の順序を間違えるとプログラムが停止



デッドロック

- ノード間のデータ交換
  - 処理の順序を間違えるとプログラムが停止