情報処理概論

第１３回　プログラムの高速化

情報基盤研究開発センター　谷本 輝夫

プログラムの実行を速くする方法

1. お任せコース
   - コンパイラの高速化機能を利用する
2. お手軽コース
   - キャッシュメモリを有効利用する
   - マルチコア計算機で並列処理
3. ヘビーコース
   - 大規模計算機を使って並列処理

高速化する前の確認

- 高速化の手間に見合った効果が得られそうなプログラムかどうか？
- 例えば
  - 数十分以内に終わるプログラム？
    ⇒　高速化の手間はあまりかけたくない
  - でも、今後何度も使う？
    ⇒　なら、もう少し手間をかけてももとがとれる

お任せコース

- コンパイラに任せてしまう
- コンパイラの役割
  - 「プログラムを翻訳する」　だけじゃない

人間向け（高級言語）

```
total = 0d0
do i = 1, 1000
  total = total + a(i)
end do
```

コンパイラ →

計算機向け（機械語）

```
pushq       %rbp
movq        %rsp, %rbp
pxor        %xmm0, %xmm0
movsd       %xmm0, t(%rip)
movl        $0, i(%rip)
movl        i(%rip), %eax
cmpl        $1000, %eax
jge         ..B1.4
...
```

コンパイラのもう一つの役割

- 最適化　（Optimization）
  - プログラムの意味を変えずに、より高速に実行できる形に変形する
- 使い方
  - コンパイラのオプションで指示
  - 例)

```
ifort -fast test.f90 -o test
```

```
gfortran -O test.f90 -o test
```

  - 指定できるオプションは、コンパイラ毎に異なる

コンパイラによる最適化の例（1）

- 無駄な計算の除去

```
s = 3.14159 * r * r
v = 4 * 3.14159 * r * r * r / 3
```

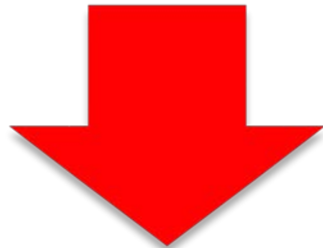
```
s = 3.14159 * r * r
v = 4 * s * r / 3
```

コンパイラによる最適化の例（2）

- より「軽い」計算への置き換え

```
do i = 1, 10000
  a(i) = a(i) / 3d0
```

１００００回の割り算



```
b = 1d0 / 3d0
do i = 1, 10000
  a(i) = a(i) * b
```

１回の割り算と
１００００回の掛け算

通常、割り算より掛け算の方が数倍高速

コンパイラにお任せコース

- オプションの指定だけで勝手にやってくれるので、とりあえず試してみて損は少ない。
- ただし、
  - かえって遅くなる　場合や、
  - 結果が微妙に違う　場合

  があるので注意。
- 性能的にまだ不足している場合は「お手軽コース」へ。

お手軽コース

- プログラムの一部を書き換えて高速化
- 対象
  - プログラムの中で時間のかかりそうな部分
- 手段
  - 計算機の特性に合わせて書き換え
    - 例（1）キャッシュメモリの有効利用
    - 例（2）マルチコアによる並列実行

プログラムの中で
時間のかかりそうな部分？

- プログラムの構造で判断：
  通常は、多重の繰り返し処理を行っている部分

```
do i = 1, 10000
  do j = 1, 10000
    do k = 1, 10000
      c(i, j) = c(i, j) + a(i, k) * b(k, j)
```

実際に所要時間を計測するとより正確

- 該当箇所の直前と直後で「現在時刻」を計測
  - Fortran では system_clockサブルーチン等を利用
    - system_clock: 現在時刻を取得する組み込みサブルーチン

```
integer(8) :: count1, count2, rate, mx
```

```
  call system_clock(count1, rate, mx)
  do i = 1, 10000
    do j = 1, 10000
      do k = 1, 10000
        c(i, j) = c(i, j) + a(i, k) * b(k, j)
      end do
    end do
  end do
```

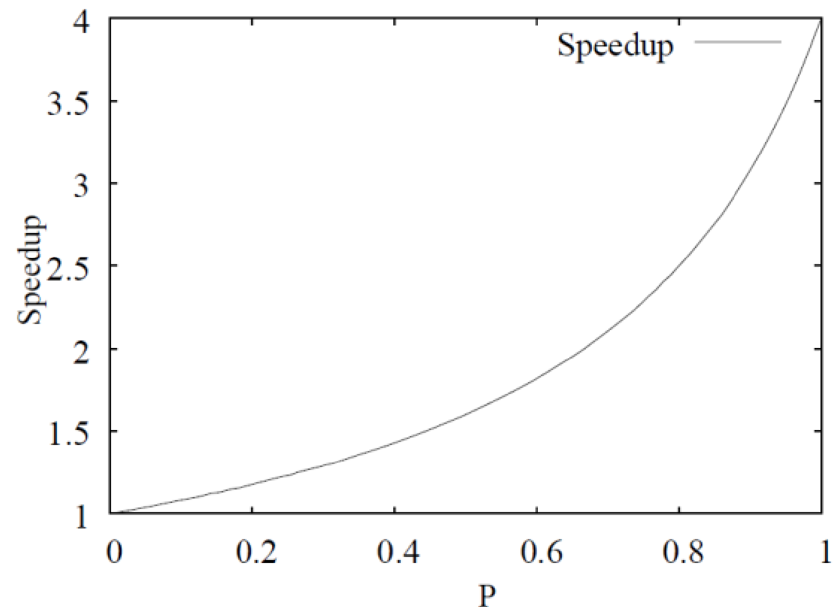
```
  call system_clock(count2)
  write(*, *) 'Time = ', dble(count2 - count1)/rate
```

system_clockサブルーチン

- 引数３個 (count, count_rate, count_max)
  - count　:　時間カウント数
  - count_rate : 1秒あたりのカウント数
  - count_max : 最大カウント数
- 引数は整数型
- 通常の integer でも良いが、より高い精度で計測したい場合は integer(8) を利用。

所要時間解析の重要性

- アムダールの法則：
  「改良した部分以外は速くならない」
- 例）プログラムのある部分を４倍高速化できたとすると
  全体の高速化率 = 1/((1-P)+P/4)
- P ＝
  高速化した部分が
  全所要時間に占めていた
  割合（高速化前）
- 例えば　8割を占めていた
  部分を４倍高速化できた
  ⇒　全体では 2.5倍

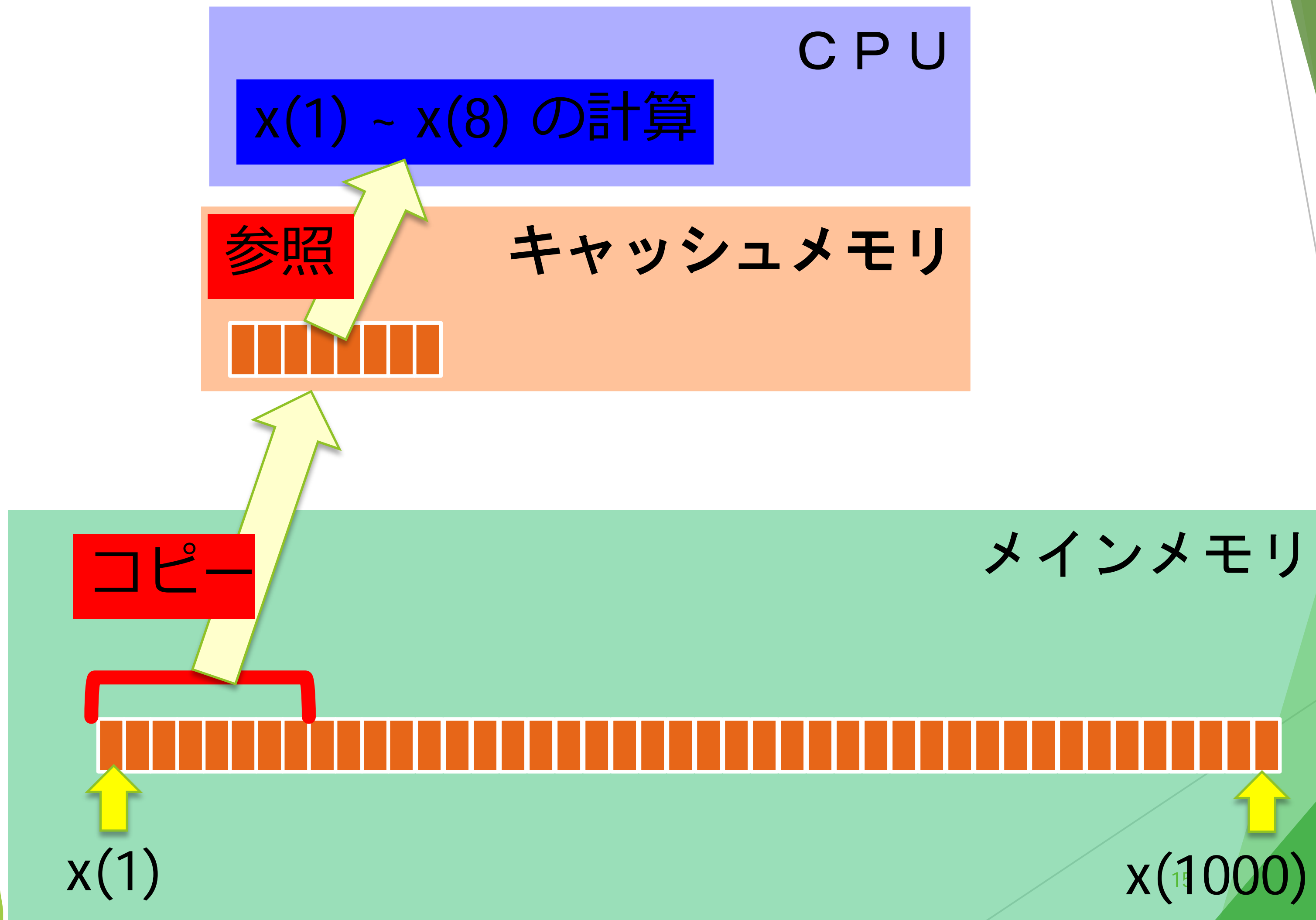


より時間のかかる部分を優先して高速化したほうが良い

お手軽コース例（１）：キャッシュメモリをうまく使う

- キャッシュメモリ
  - データの一時的な保存場所
- ＣＰＵの速度に近い高速動作が可能
- 高価なので、容量が非常に小さい

キャッシュメモリの動作



キャッシュメモリの動作

- ＣＰＵはキャッシュにコピーされたデータしか参照できない
- コピーは「キャッシュライン」単位
  - 例）Intel Core2 の2次キャッシュ：　64バイト単位
- 一度コピーしたデータは何度でも参照できる
- ただし、キャッシュが一杯になった場合、古いコピーを新しいコピーと入れ替え

近接するデータに連続して参照するとキャッシュを有効に活用できる

キャッシュメモリの有効利用（１）

- より高速に動作するプログラムはどっち？

```
do i = 1, 10000
  a(i) = a(i) * 2.0
  b(i) = b(i) * 2.0
  c(i) = c(i) * 2.0
  ...
  z(i) = z(i) * 2.0
end do
```

```
do i = 1, 10000
  a(i) = a(i) * 2.0
end do
do i = 1, 10000
  b(i) = b(i) * 2.0
end do
 ...
do i = 1, 10000
  z(i) = z(i) * 2.0
end do
```

こっち

キャッシュメモリの有効利用（２）

- アクセスの順番に注意
- 特に多次元配列：
  Fortran では一番左側の次元から配置

a(1, 1) | a(2, 1) | … | a(100, 1) | a(1, 2) | … | a(100,100)

⇒　一番左側の次元を連続参照すると高速

- どちらが高速？

```
do i = 1, 10000
  do j = 1, 10000
    a(i, j) = a(i, j) + 1.0
```

```
do j = 1, 10000
  do i = 1, 10000
    a(j, i) = a(j, i) + 1.0
```

こっち

キャッシュメモリを有効利用するプログラム

- なるべく連続したデータを参照する
  - データの配置を思い浮かべながら作る
- なるべく連続して計算だけを行う
  - 出来るだけ do文の中で if 文等の条件分岐をしない
  - 計算以外の処理をすると、せっかくキャッシュにコピーしたデータが無駄になる可能性
- キャッシュ有効利用の効果はほぼ全ての計算機で得られ、場合によっては10倍以上高速化

お手軽コース（2）
マルチコアの計算機で並列処理

- 並列計算機
  - 複数のCPUコアに、仕事を分担させて高速化することができる計算機
- 数万円で買えるパーソナルコンピュータも並列計算機
  - デュアルコア　（＝　2 CPUコア）
  - クアッドコア　（＝　4 CPUコア）

マルチコア計算機で並列処理

- 「並列処理」とは？
  複数のCPUコアに仕事を分担させて高速化

- 並列処理をするには？
  「並列プログラム」を作成

並列プログラム

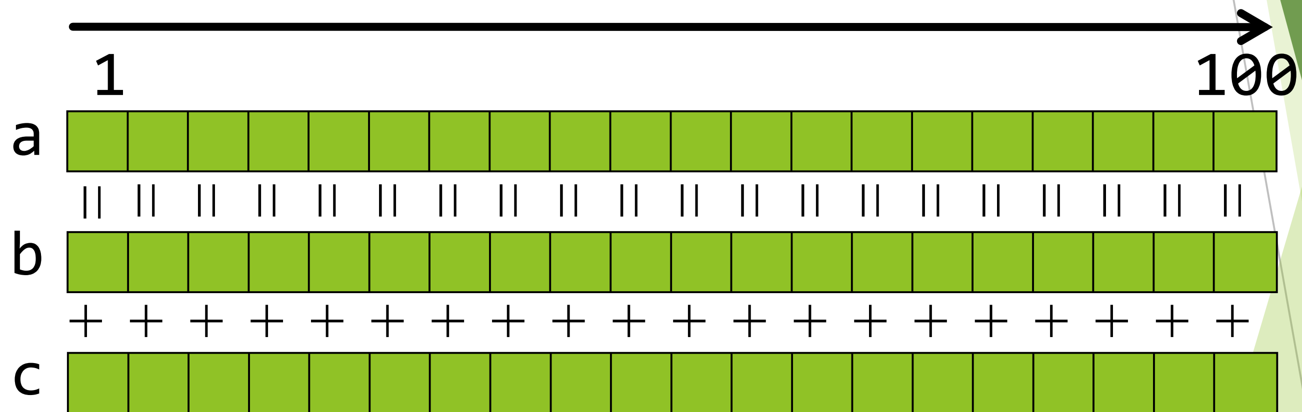
- 並列処理に必要な事項を含むプログラム
  - 各ＣＰＵコアへの仕事の分担のさせ方、
  - 相互の情報交換、
  - CPUコアの間の同期

  等

普通のプログラム（＝並列じゃないプログラム）とどう違う?

普通のプログラムの例：２つのベクトルの和を計算

- １番目から１００番目までの要素を順に計算

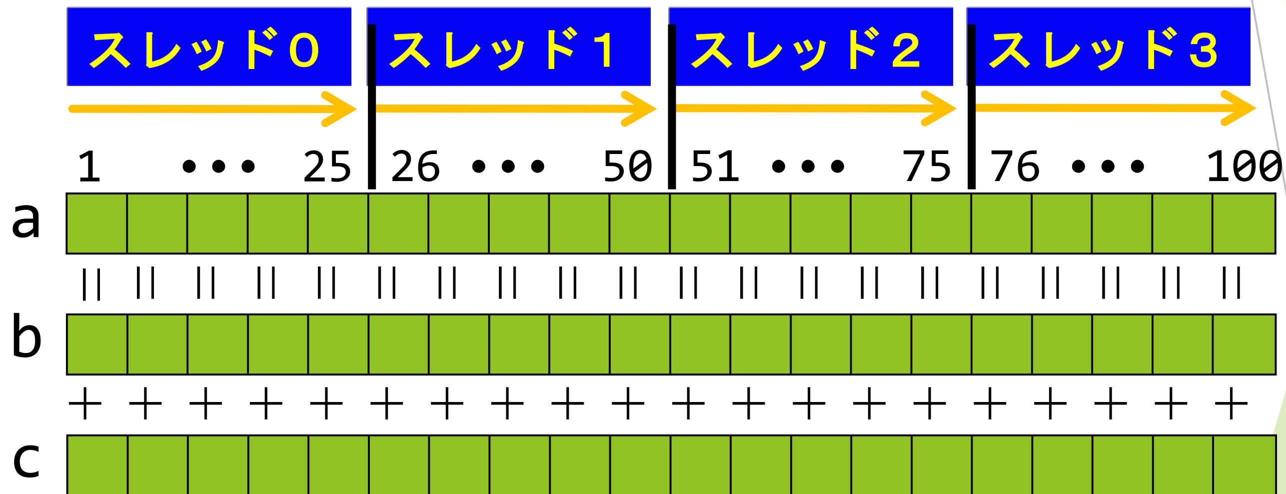


プログラム

```
do i = 1, 100
  a(i) = b(i) + c(i)
```

並列プログラムの例：
複数の「スレッド」で並列処理

- スレッド
  - 同じ記憶空間で進行するプログラムの流れ



スレッド0
```
do i = 1, 25
  a(i) =
```

スレッド1
```
do i = 26, 50
  a(i) = b
```

スレッド2
```
do i = 51, 75
  a(i) =
```

スレッド3
```
do i = 76, 100
  a(i) = b(i) + c(i)
```

スレッド並列プログラムの作り方

- １）コンパイラにおまかせ
  - 並列化しても問題ない部分を、自動的に変形して仕事をスレッドに分担させる
- ２）自分でプログラムを書き換える
  - 仕事の分担のさせ方を、プログラム中に明記する

コンパイラによる並列化

- コンパイラの自動並列化機能
  ＝　「お任せコース」の最適化の一部
  - 最近はほとんどのコンパイラで利用可能
  - だが、gfortranは持っていない…
  - ifort の場合：

```
ifort -fast -parallel test.f90 -o test
```

さらに実行時に使用スレッド数を指定

```
export OMP_NUM_THREADS 4
./test
```

- 簡単なプログラムでは、それなりの効果

複雑なプログラムの並列化は人間の助けが必要

「OpenMP」による並列化

- OpenMP
  - 簡単にスレッドへの仕事の割り当て方を記述するための手法

- 基本的に「並列化指示行」を追加するだけ
  - 例）　前出のスレッド並列処理をOpenMPで記述

```
!$omp parallel do
do i = 1, 100
  a(i) = b(i) + c(i)
```

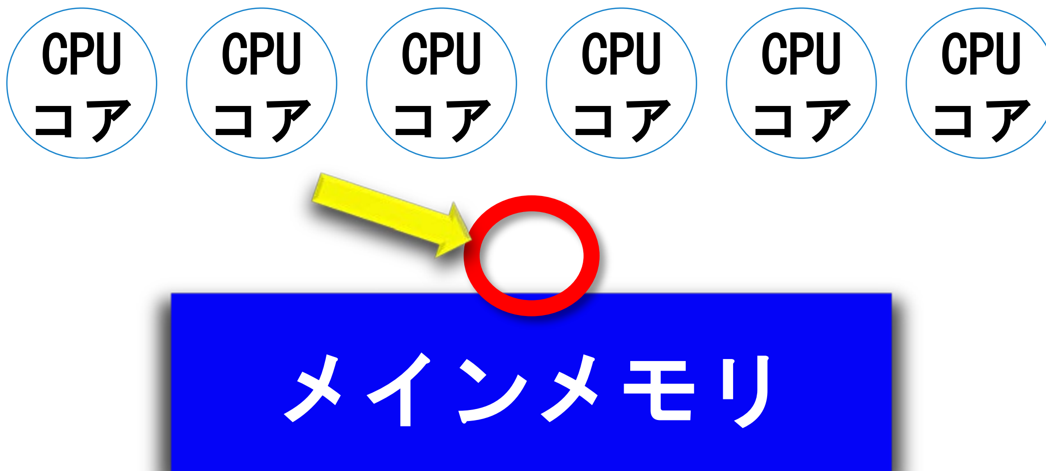
並列化指示行

スレッドによる並列化の利点と欠点

- 利点：　簡単に並列化
  - コンパイラにお任せ
    もしくはOpenMPの指示行を追加するだけ
- 欠点：
  基本的に「共有メモリ型並列計算機」向き
  ⇒　大規模な計算機で利用できない

共有メモリ型並列計算機

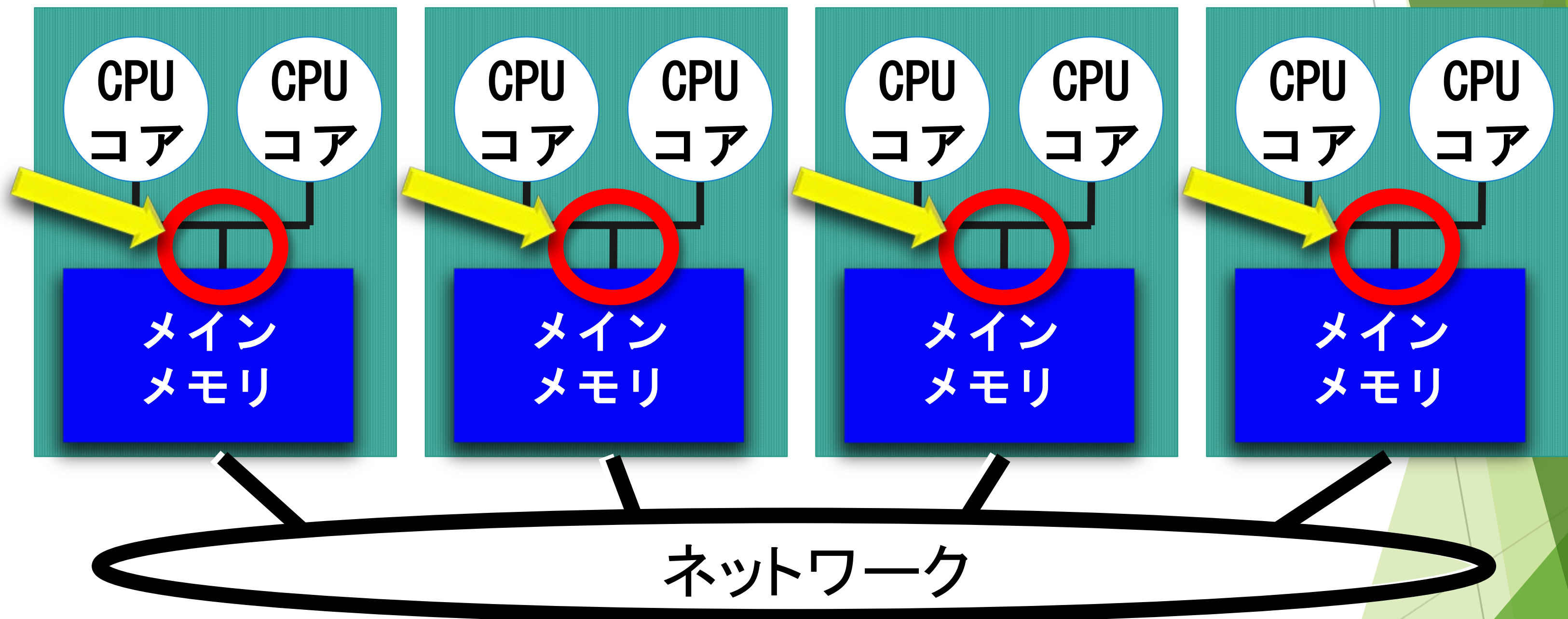
- １つのメインメモリを複数のＣＰＵコアで共有
  - 例）マルチCPUコアのＰＣ等



CPUコアからメインメモリへの経路が共有
⇒　規模（＝CPUコア数）に限界

分散メモリ型並列計算機

- 複数の独立したメインメモリで構成



規模に応じて経路の数も増加
⇒　大規模化が比較的容易

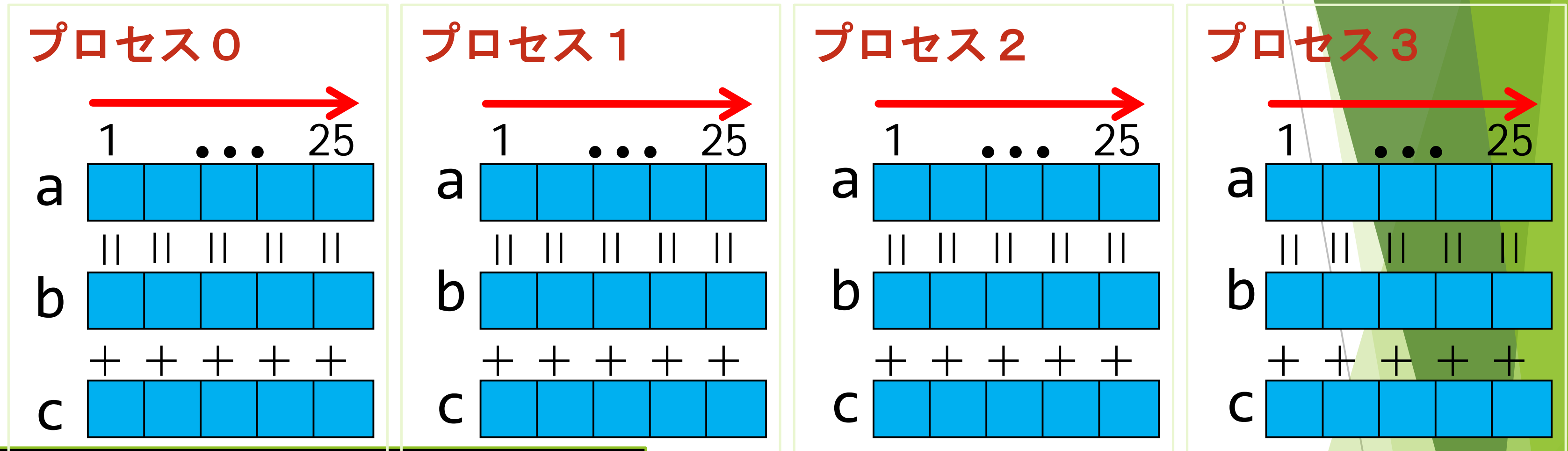
ヘビーコース

- 分散メモリ型並列計算機の有効利用
  ＝「プロセス並列」プログラム

- プロセス並列
  - 独立した記憶空間をもつ「プロセス」を単位とした並列処理

「プロセス並列」プログラムの特徴（１）

- 処理だけでなくデータも分割




```
real(8),dimension(25)::a, b, c
  ...
do i = 1, 25
  a(i) = b(i
```

プロセス０

```
real(8),dimension(25)::a, b, c
  ...
do i = 1, 25
  a(i) = b(i)
```

プロセス１

```
real(8),dimension(25)::a, b, c
  ...
do i = 1, 25
  a(i) = b(i)
```

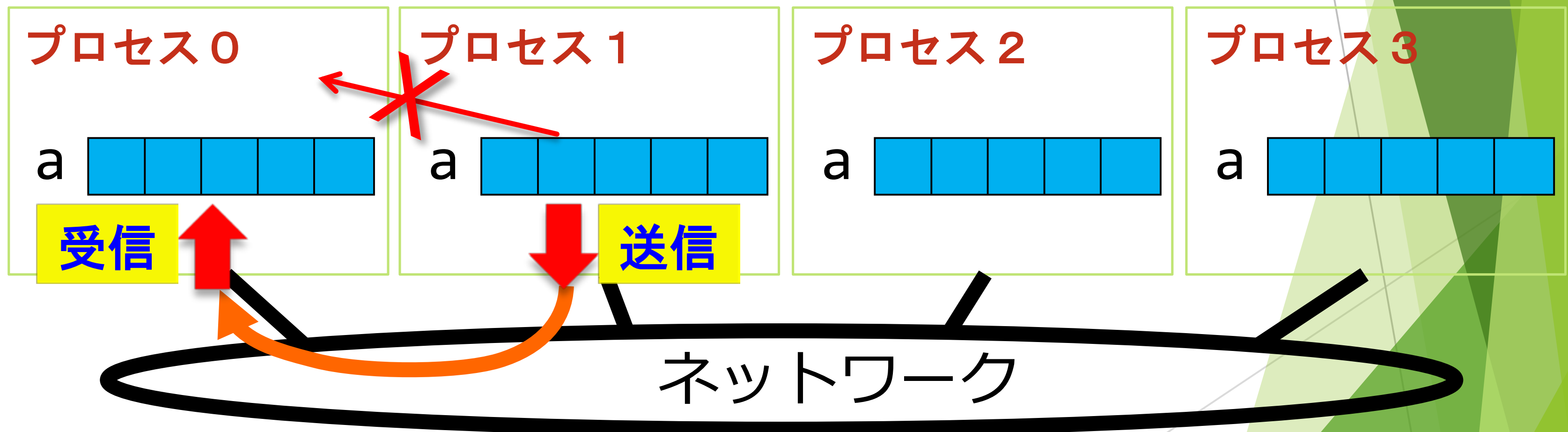
プロセス２

```
real(8),dimension(25)::a, b, c
  ...
do i = 1, 25
  a(i) = b(i) + c(i)
```

プロセス３

「プロセス並列」プログラムの特徴（２）

- 他のプロセスのデータは直接参照できない
- 必要に応じてプロセス間通信



ＭＰＩ
（Message Passing Interface）

- 並列プログラム用に用意された通信関数群
  - 例）プロセス０からプロセス１にデータを転送

```
call MPI_Comm_rank(MPI_COMM_WORLD, myid, err)
   ...
if (myid == 0) then
  call MPI_Send(a[5], 1, MPI_INTEGER, 1, &
                 0,  MPI_COMM_WORLD, err)
else if (myid == 1) then
  call MPI_Recv(a[3], 1, MPI_INTEGER, 0, &
                0, MPI_COMM_WORLD, status, err)
end if
```

自分のプロセス番号を取得

プロセス１に送信

プロセス０から受信

並列化の手段と並列計算機

	共有メモリ型	分散メモリ型
自動並列化、OpenMP	○	×
MPI	○	○

MPIプログラムは、
作るのに苦労するがどこでも実行できる

並列化できる？できない？

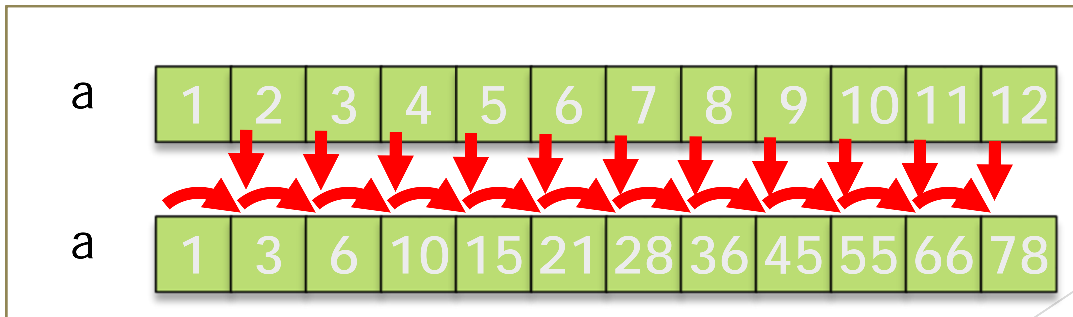
- 並列処理
  ＝　仕事を分担し、同時進行で処理
  ＝　仕事の順番が不確定

- 並列化可能な仕事
  ＝　実行の順番が変わっても結果が変わらない仕事

並列化できないプログラムの例

- 以下のプログラムを複数の仕事に分担させても良いか？

```
real(8), dimension(12) :: a
  ...
do i = 2, 12
  a(i) = a(i) + a(i-1)
```



まとめ

- プログラムの書き方は何通りもある
- プログラムの高速化
  ＝　計算機に応じた適切な書き方の選択
- 主な高速化の手法
  - コンパイラに任せる
  - キャッシュの効果的利用
  - スレッドによる並列化
  - プロセスによる並列化

演習

- p. 18 の 2 つの例を実際に動くプログラムにする。
  - 変数の宣言、aの値の初期化など
- p. 11 の時間計測サブルーチンを使って、 2 つの例の実行時間を表示させ、その違いを確認する

```
program Speed_Check
  implicit none
  integer :: i, j
  real(8), dimension(10000,10000) :: a
  integer(8) :: count1, count2, rate, mx

  a = 0
  call system_clock(count1, rate, mx)
  do i = 1, 10000
     do j = 1, 10000
        a(i, j) = a(i, j) + 1.0
     end do
  end do
  call system_clock(count2)
  write(*, '(A14F7.3)') 'Type 1 Time = ', dble(count2 - count1)/rate

  a = 0
  call system_clock(count1, rate, mx)
  do i = 1, 10000
     do j = 1, 10000
        a(j, i) = a(j, i) + 1.0
     end do
  end do
  call system_clock(count2)
  write(*, '(A14F7.3)') 'Type 2 Time = ', dble(count2 - count1)/rate

end program Speed_Check
```