(財)計算科学振興財団、大学院GP「大学連合による計算科学の最先端人材育成」

第1回　社会人向けスパコン実践セミナー　資料

並列処理の基礎及びOpenMP演習

2009年2月17日　9:30～12:15

九州大学情報基盤研究開発センター

南里　豪志

あなたの周りにも並列計算機

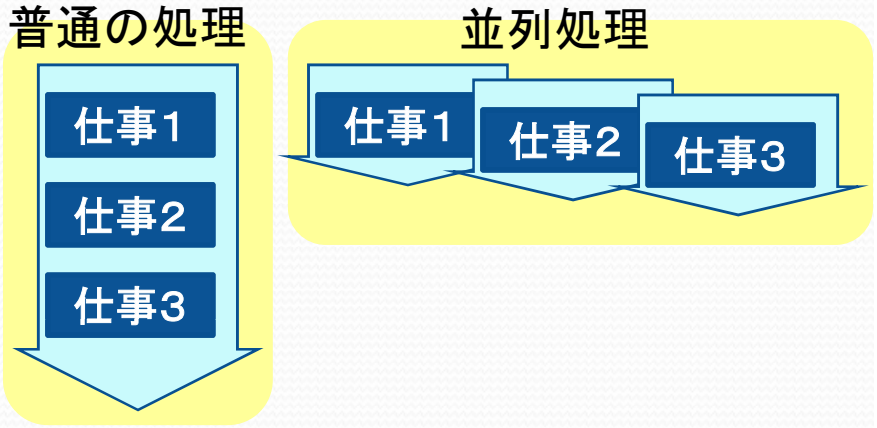
- **今や、スーパーコンピュータもパソコンも並列計算機**
  - 並列計算機　＝　CPUコアが複数搭載された計算機

- **IBM Roadrunner（米国 ロスアラモス国立研究所）　　122400 CPUコア**
  - 世界1位
- **Hitachi T2Kオープンスーパーコンピュータ（東京大学）　12288 CPUコア**
  - 世界 16位 (日本　1位)
- **Fujitsu PRIMERGY RX200S3（九州大学）　1536 CPUコア**
  - 世界172位(日本 11位)

- **パソコン**
  - MouseComputer Lm-A424B（39800円）　　**2CPUコア**
  - HP Pavilion Desktop s3720jp (64050円)　　**4CPUコア**

ようこそ”並列処理”の世界へ

- ”並列処理”とは？
  複数のCPUコアに仕事を分担して処理させること
  ⇒　並列計算機の性能を最大限に発揮



- 並列処理をするには？
  ”並列プログラム”を作成して、並列計算機の上で実行する

- ”並列プログラム”？
  仕事の分担の仕方等、並列処理に必要な事項が指示されたプログラム

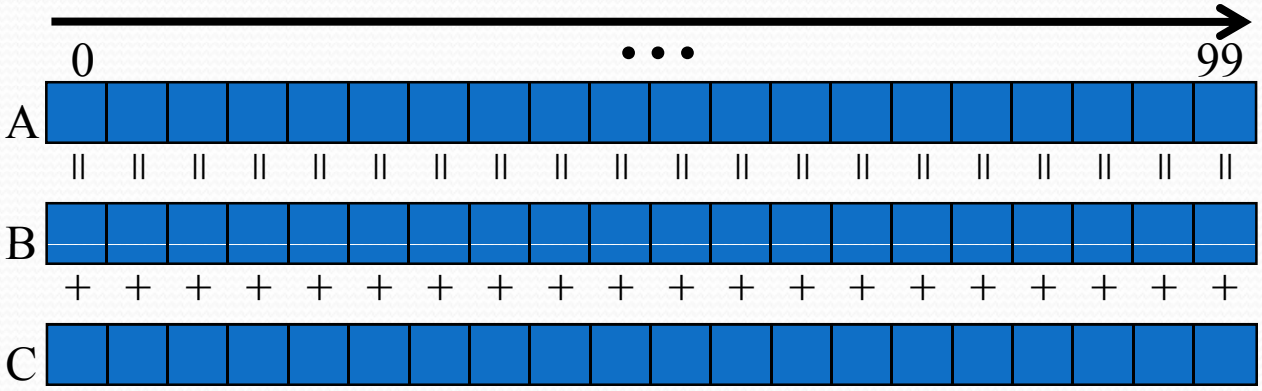
  普通のプログラム（並列じゃないプログラム）とどう違う?

講義の流れ

- **並列プログラムの概要**
  - 通常のプログラムと並列プログラムの違い
  - 並列プログラム作成手段と並列計算機の構造
- **OpenMPによる並列プログラム作成**
  - 処理を複数コアに分割して並列実行する方法
- **MPIによる並列プログラム作成 （午後）**
  - プロセス間通信による並列処理
  - 処理の分割　＋　データの配置　＋　通信

普通のプログラム(逐次プログラム)の例: ベクトル同士の和の計算

- **0番目の要素から順に99番目の要素まで計算**

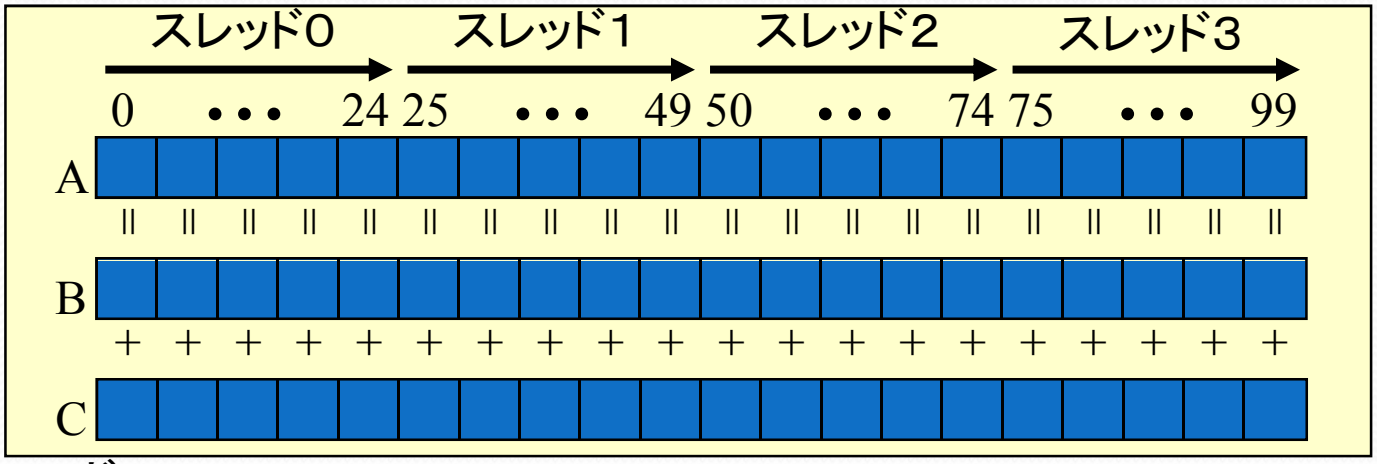


プログラム

```
double A[100], B[100], C[100];
   ...
for (i = 0; i < 100; i++)
    A[i] = B[i] + C[i];
```

並列プログラムの例1： ”スレッド”による並列処理

- スレッド並列： 同じメモリを使いながら仕事を分担する並列処理



スレッド0

```
double A[100], B[100], C[100];
  ...
for (i = 0; i < 25; i++)
    A[i] = B[i] + C[i];
```

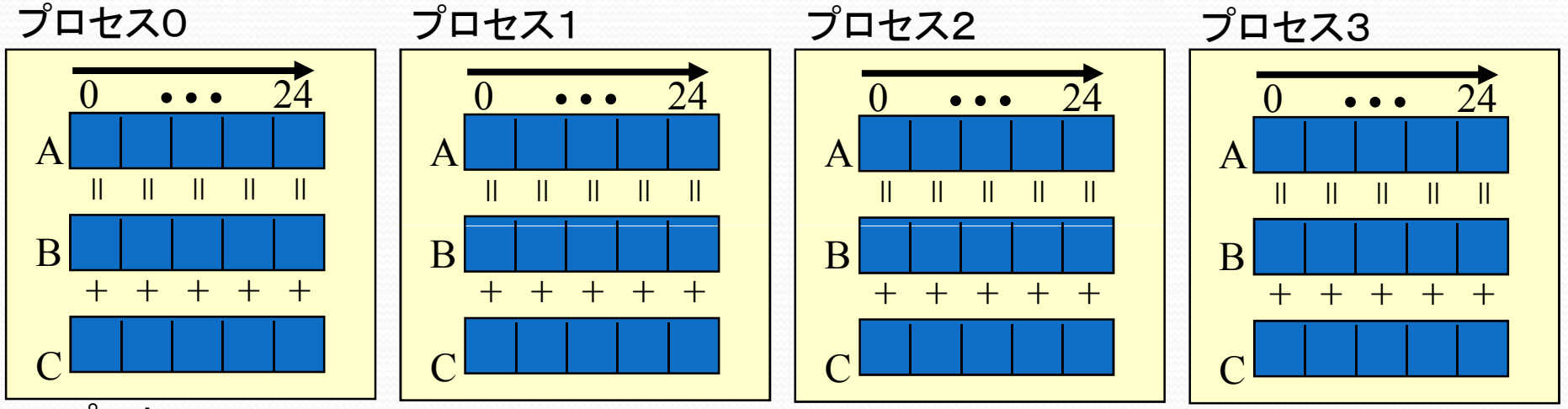
スレッド3

```
double A[100], B[100], C[100];
  ...
for (i = 75; i < 100; i++)
    A[i] = B[i] + C[i];
```

全部のスレッドで同じ配列を共有するので、逐次プログラムに近いイメージ

並列プログラムの例2：”プロセス”による並列処理

- プロセス並列： それぞれ独自のメモリを使いながら仕事を分担する並列処理



プロセス0

```
double A[25], B[25], C[25];
  ...
for (i = 0; i < 25; i++)
    A[i] = B[i] + C[i];
```

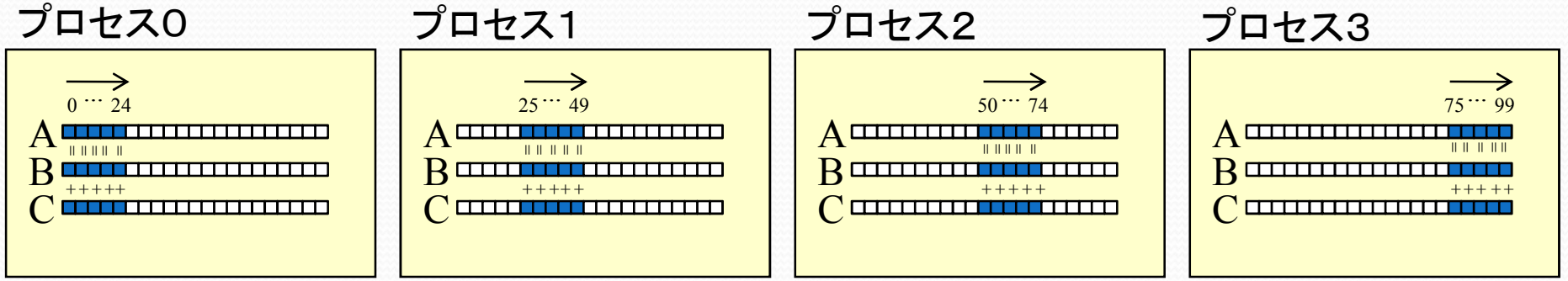
プロセス3

```
double A[25], B[25], C[25];
  ...
for (i = 0; i < 25; i++)
    A[i] = B[i] + C[i];
```

各プロセスに配列を分割するので、「データの配置」を意識する必要あり

”プロセス”による並列処理のもう一つの方法

- データの重複配置（各プロセスに配列の全要素を配置）



プロセス0

```
double A[100], B[100], C[100];
   ...
for (i = 0; i < 25; i++)
    A[i] = B[i] + C[i];
```

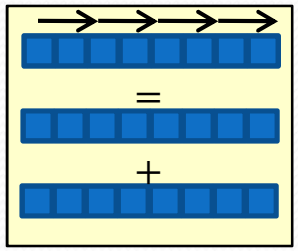
プロセス3

```
double A[100], B[100], C[100];
   ...
for (i = 75; i < 99; i++)
    A[i] = B[i] + C[i];
```

イメージは逐次プログラムに近くなるが、メモリを無駄に消費。
どちらにしても、他のプロセスの計算結果を直接参照することはできない。

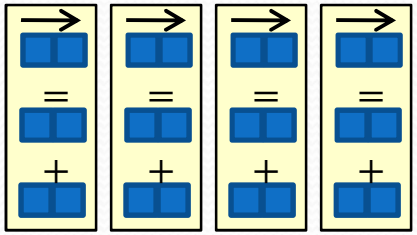
スレッド並列とプロセス並列，どちらがいい？

- スレッド並列： 全スレッドが同じメモリ空間でプログラムを実行



  - × 基本的に、“共有メモリ型”の並列計算機でしか利用できない
  - ◎ プログラムが簡単
    - 処理の分割の仕方だけを指示すれば良い
    - 逐次プログラムを自動的にスレッド並列プログラムに変換する“自動並列化コンパイラ” も利用可能。
      - 便利だが、必ずしも効率の良い並列プログラムに変換されるとは限らない

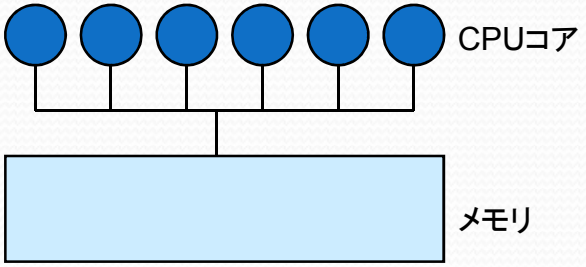
- プロセス並列： 各プロセスがそれぞれ独自のメモリ空間でプログラムを実行



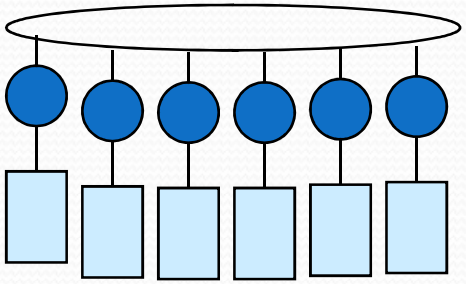
  - ◎ 基本的に、どんな形の並列計算機でも利用可能
  - × プログラムが複雑
    - データの配置を意識したプログラミングが必要
    - 他のプロセスの計算結果を参照するには通信が必要

並列計算機の形

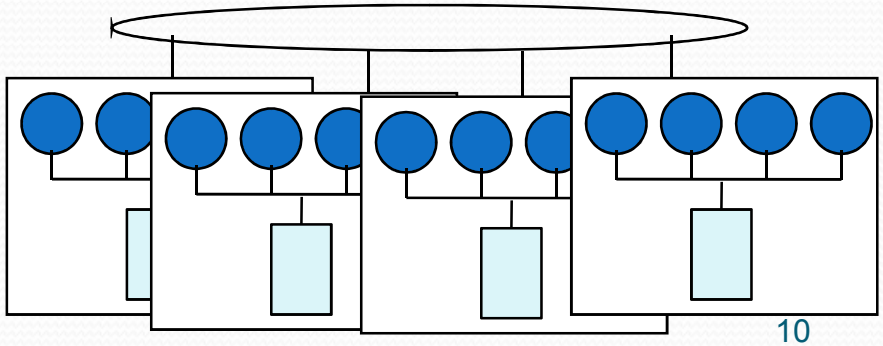
- **共有メモリ型並列計算機:**
  一つのメモリを複数のCPUコアが共有
  - 例) デュアルコアやクアッドコアのパソコン、
    小規模～中規模のサーバ
  - メモリへのアクセスが集中するので大規模化が困難



- **分散メモリ型並列計算機:**
  各CPUコアに一つずつメモリを配置
  - 例) (昔の)PCクラスタ
    - 複数のパソコンを接続して並列計算機として利用
  - メモリアクセスの集中が起きにくいので大規模化が可能

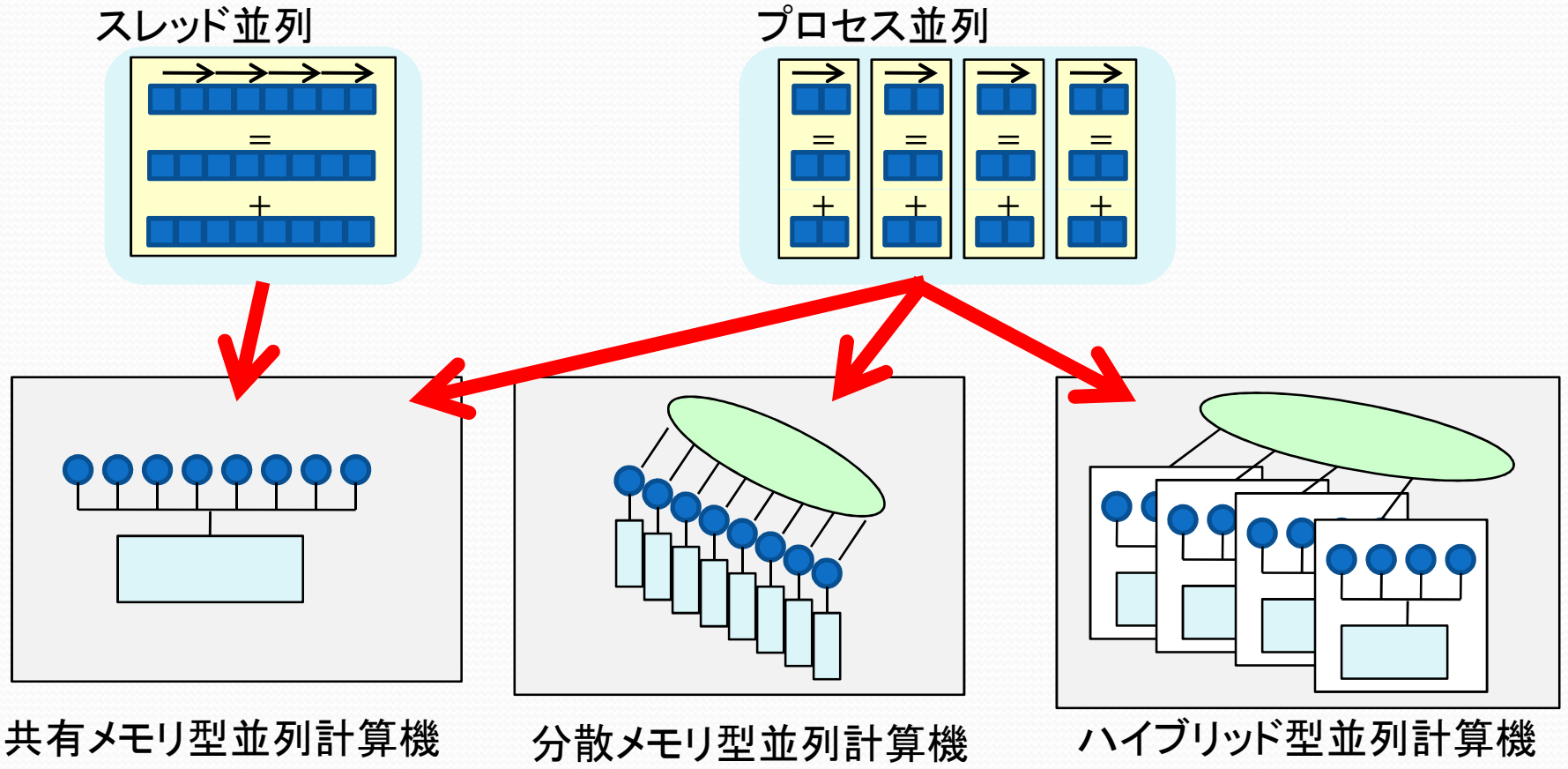


- **ハイブリッド型並列計算機:**
  複数の共有メモリ型並列計算機を接続
  - 例) (最近の)PCクラスタ、
    (最近の)スーパーコンピュータ
  - 両方のいいとこ取り (?)



並列処理の種類と並列計算機の関係

- プロセス並列は全ての並列計算機で実行可能
- スレッド並列は、基本的に共有メモリ型並列計算機だけ



実際のスレッド並列とプロセス並列

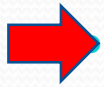
- **OpenMP … 現在最も普及しているスレッド並列のプログラミングモデル**
  - ほぼ全ての共有メモリ型並列計算機で、同じ OpenMPプログラムを利用可能
  - C, C++, Fortranのプログラムに"指示文"を追加するだけで並列化可能

- **MPI … 現在最も普及しているプロセス並列のプログラミングモデル**
  - ほぼ全ての並列計算機で、同じ MPIプログラム利用可能
  - C, C++, Fortranのプログラムから通信等の関数を呼び出す

- **ハイブリッド並列： プロセス並列の各プロセスをさらにスレッド並列化**
  - MPI + OpenMP もしくは MPI + 自動並列化 が主流
  - 特にハイブリッド型の並列計算機で利用

講義の流れ

- **並列プログラムの概要**
  - 通常のプログラムと並列プログラムの違い
  - 並列プログラム作成手段と並列計算機の構造



- **OpenMPによる並列プログラム作成**
  - 処理を複数コアに分割して並列実行する方法

- **MPIによる並列プログラム作成 （午後）**
  - プロセス間通信による並列処理
  - 処理の分割　＋　データの配置　＋　通信

OpenMPによる並列プログラムの例

- **並列版 Hello World**
  - 各スレッドがそれぞれ自分のスレッド番号付きでメッセージを表示

C言語

```
#include <stdio.h>
#include <omp.h>
int main()
{
  printf("並列世界へようこそ¥n");
#pragma omp parallel
  {
    printf("並列世界で処理中．番号%d¥n",
          omp_get_thread_num());
  }
  printf("さようなら¥n");

  return 0;
}
```

Fortran

```
program hello
implicit none
integer,external :: omp_get_thread_num
  print *, "並列世界へようこそ"

!$omp parallel
  print *, "並列世界で処理中．番号",  &
           omp_get_thread_num()
!$omp end parallel

  print *, "さようなら"
end program hello
```

OpenMPプログラムの基本構造

- 全スレッドが同じプログラムを実行
  - SPMD (Single Program Multiple Data)

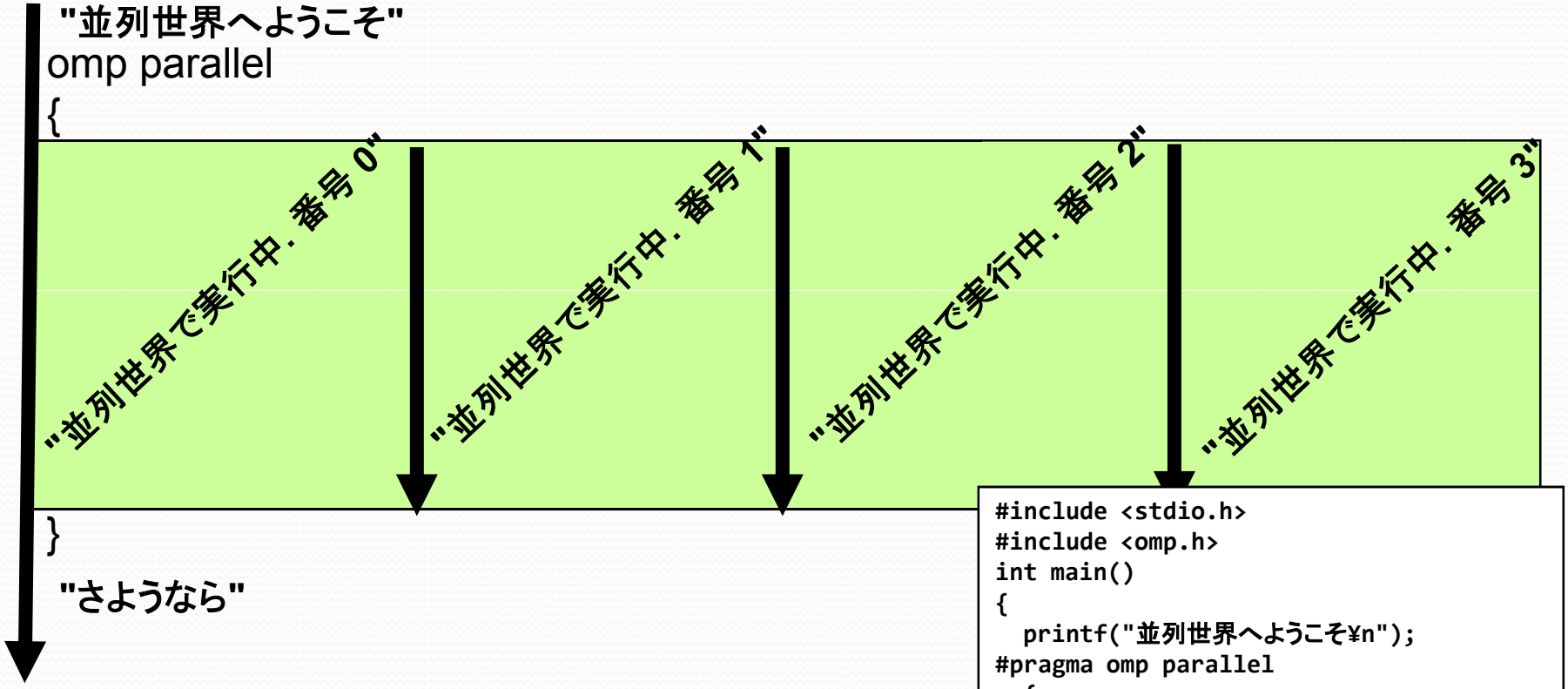
- 通常のプログラムに "指示文" を追加
  ⇒ 各スレッドへの処理の割り当て方
  等を指示

  例) parallel指示文
  - 並列実行開始
  - 直後のブロックを複数のスレッドで
  並列実行

```
#include <stdio.h>
#include <omp.h>
main()
{
  printf("並列世界へようこそ¥n");
#pragma omp parallel
  {
    printf("並列世界で処理中. 番号%d¥n",
          omp_get_thread_num());
  }
  printf("さようなら¥n");
}
```

- 並列処理を指示された場所(＝ 並列リージョン)だけが
複数スレッドで並列実行

OpenMPプログラムの実行イメージ




```
#include <stdio.h>
#include <omp.h>
int main()
{
  printf("並列世界へようこそ¥n");
#pragma omp parallel
  {
    printf("並列世界で処理中．番号%d¥n",
          omp_get_thread_num());
  }
  printf("さようなら¥n");
  return 0:
}
```

OpenMP指示文

- **プログラミング言語毎に規定**
  - C/C++: #pragma omp で始まる行
  - Fortran: !$omp で始まる行
- **OpenMPに対応していないコンパイラでは、注釈行として無視**
  - エラーにならない
- **構成： 指示文名，指示節**
  例)

```
#pragma omp for private(t), reduction(+: sum)
```

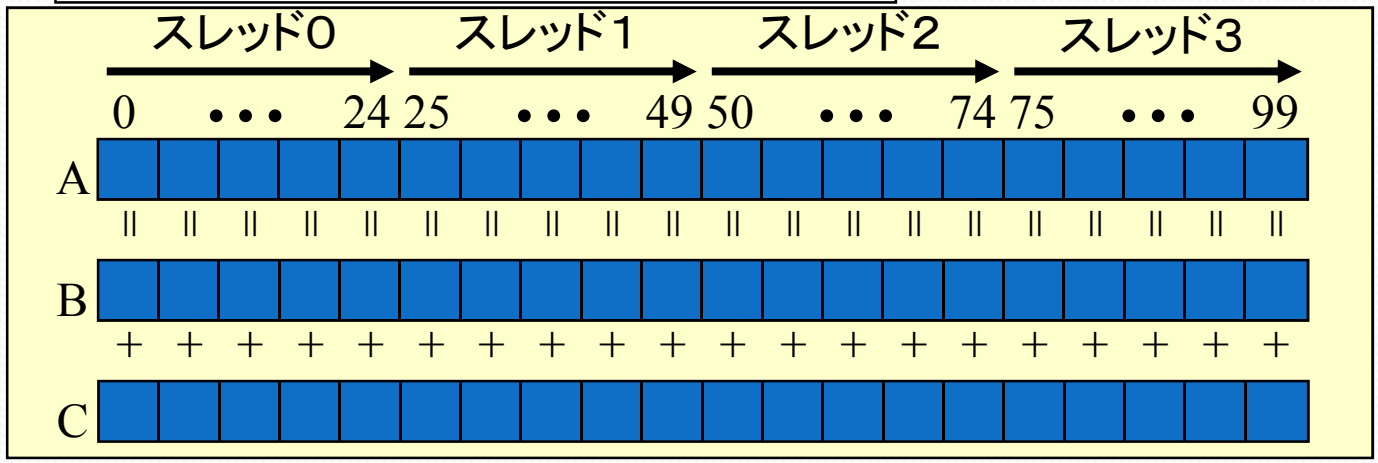
指示文名：主に並列化の手段を指示．
原則として1行に1つだけ．

指示節：変数の扱い等，細かい内容
を指示．1行に複数指定可．

for指示文(do指示文)

- **直後の for ループ (doループ)を並列化**
  - ループをスレッド数で等分して分担処理
  - 並列リージョン内で使用
  - parallel指示文（並列実行開始）と for指示文を合わせた parallel for指示文（parallel do指示文）もある

```
double A[100], B[100], C[100];
  ...
#pragma omp for
  for (i = 0; i < 100; i++)
    A[i] = B[i] + C[i];
```



実行時ライブラリルーチン

- **OpenMPプログラムの実行を補助するルーチン**

  例)

  - スレッド番号の参照
    omp_get_thread_num()
  - スレッド数の参照
    omp_get_num_threads()
  - スレッド数の指定
    omp_set_num_threads()
    - 次回の並列リージョンから適用
  - 経過時間の測定
    omp_get_wtime()

```
#include <stdio.h>
#include <omp.h>
main()
{
  printf("並列世界へようこそ¥n");
#pragma omp parallel
  {
    printf("並列世界で処理中. 番号%d¥n",
          omp_get_thread_num());
  }
  printf("さようなら¥n");
}
```

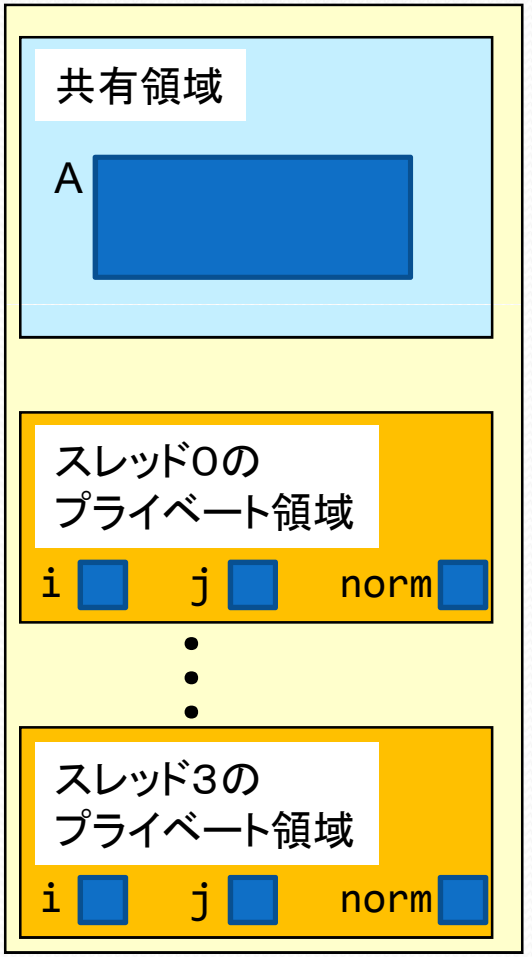
- **ヘッダファイル omp.h の中で定義**

OpenMPの記憶空間：共有変数とプライベート変数

- **共有変数**
  - 全スレッドで共有する領域を利用する変数
  - OpenMPでは、特に指示しなければ全て共有変数
  - ただし、並列化されたループの制御変数はプライベート変数
- **プライベート変数**
  - 各スレッドが独自に持つプライベート領域を利用する変数
  - private指示節等で指定

プログラム例:
配列Aの各行について、ユークリッドノルムで正規化

```
double A[N][M], norm;
integer i, j;
  ...
#pragma omp for private(j, norm)
for (i = 0; i < N; i++){
  norm = 0.0;
  for (j = 0; j < M; j++)
    norm += A[i][j]*A[i][j];
  for (j = 0; j < M; j++)
    A[i][j] = A[i][j] / sqrt(norm);
}
```



プライベート変数の利用

- **スレッド毎に独立して値を設定するべき変数**
  ＝　他のスレッドに書き換えられると
  実行結果が変わってしまう変数

```
double A[N][M], norm;
integer i, j;
  ...
#pragma omp for private(j, norm)
for (i = 0; i < N; i++){
  norm = 0.0;
  for (j = 0; j < M; j++)
    norm += A[i][j]*A[i][j];
  for (j = 0; j < M; j++)
    A[i][j] = A[i][j] / sqrt(norm);
}
```

例)

- 並列化されたループの制御変数
  - 特に指示しなくてもプライベート変数になる
- 並列リージョン中のループ制御変数
  - 特殊な場合を除き、ループ制御変数が勝手に書き変えられると困る
    - ちなみにFortranでは、この場合も指示無しでプライベート変数になる
- その他、並列リージョン内で値が変化する変数に注意
  - 複数のスレッドが同時に同じ変数（配列の場合は同じ要素）に対して書き込む可能性がある場合プライベート変数にする
    - もしくは、OpenMPのロックルーチン（本講義では説明を割愛）を使って排他制御する
  - 配列は、アクセスする範囲がスレッド毎に異なれば共有変数のままで良い（例中の配列 A）
  - 配列以外の変数は、プライベート変数にしないといけない場合が多い（例中の norm）

リダクション変数

- **並列処理終了後に全スレッドの値を集約させることを指示**
  - 集約時の操作: 総和(+), 最大値(MAX)等.
  - 内部では、各スレッドでの計算中はプライベート変数として扱い、最後に全スレッドで集約して共有変数となる.

プログラム例:
ベクトルの総和

```
double total(x)
double x[];
{
  int i;
  double t;
  t = 0.0;
#pragma omp parallel for reduction(+:t)
  for (i = 0; i < 100; i++)
    t += x[i];
  return t;
}
```

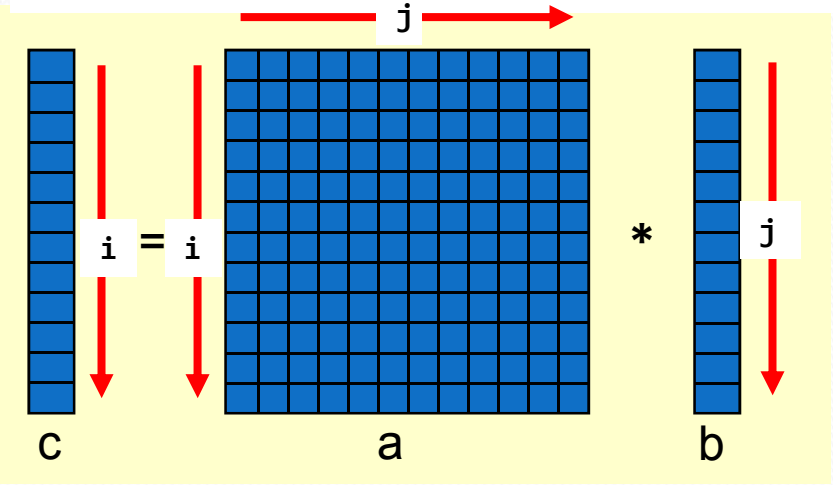
OpenMPの特徴

- **高い移植性**
  - C,C++,Fortranの文法はそのまま
  - 世界共通の規格（各計算機メーカーが準拠）
- **簡単で使いやすい**
  - OpenMP非対応のコンパイラではコメント文として無視される
    → 同じプログラムを並列版，非並列版どちらにも利用可能
  - ほとんどの場合，並列化したいループの直前に一行追加するだけ．
- **基本的に共有メモリ型並列計算機でのみ利用可能**
  - PCクラスタ等の分散メモリ型並列計算機での利用は困難
- **並列化は自己責任**
  - 並列化の可否や並列化による効果に関するチェックは行わない

プログラムの並列化事例
対象： 行列・ベクトル積プログラム

並列化前のプログラム

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

#define N 100
double a[N][N], b[N], c[N];

int main(int argc, char *argv[])
{
  int i, j;

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

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

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

  for (i = 0; i < N; i++)
    printf("(%d: %.2f) ", i, c[i]);
  printf("¥n");

  return 0;
}
```



プログラムを並列化する前に検討すること

- そのプログラムを並列化する価値はあるか？
- そのプログラムは並列化できるか？
- どのように処理を各スレッド（プロセス）に分担させるか？

注意）
必ずしもこの順番に進めるとは限らないし、いくつかを合わせて検討する場合もある。

そのプログラムを並列化する価値はあるか？

- **並列化に要する時間に見合った速度向上が得られそうか?**
  - プログラムの性能分析が重要:
    1. 1回の実行に何時間くらいかかりそうか？
    2. そのうち何割くらいを並列化できそうか？
    3. そのプログラムは今後何回くらい実行しそうか?
    - 1回しか実行しないプログラムを並列化するのは、もったいないかもしれない。
  - OpenMPの場合は、比較的短時間で並列化できるので、
    とりあえず試してみても良いかもしれない。

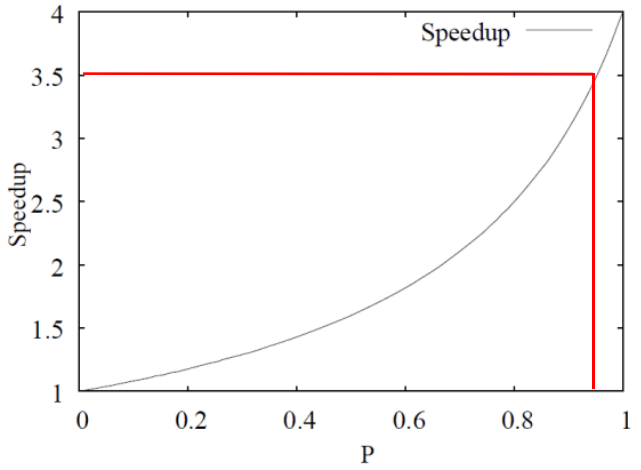
- **並列化の他に、もっと楽な高速化の手段が無いか?**
  - コンパイラの最適化オプションの吟味
    - 適切に選ぶと性能が数倍向上する場合もある
  - 数値計算ライブラリの利用
    - 特に密行列の計算（行列積、ベクトル行列積、連立一次方程式、固有値等）は
      高速な関数が多数存在する。
    - スーパーコンピュータ上の数値計算ライブラリは、
      その計算機向けにチューニングされているので、特に高速

ちょっと寄り道：
アムダールの法則

- **並列化による効果の理論的な限界**
  「プログラム中の並列化対象部分が全処理時間に占める割合をPとすると、その並列化対象部分をN個のスレッド（又はプロセス）で理想的に並列実行できたとしても、並列化による性能向上率は
  1/((1−P)+P/N)
  である。」
  - 難しく見えるかもしれないが、よく見るととても単純：
    1. 並列化できる部分 P は N倍まで速くできる　(P/N)
    2. 並列化できない部分 1−P はそのまま　(1−P)

- **実は、並列化の効果を得るのは結構難しい。**
  - 例えば N=4 の場合、3.5倍以上高速化するためにはプログラムの処理時間の95%以上が並列化できなければならない.
  - さらに、並列化することによって新たに発生するコストがある。
    - スレッド（プロセス）間の待ち時間や、プロセス間データ転送に要する通信時間等。

プロセス数4（N=4）の場合の性能向上率



今回のプログラムの場合：何割くらいを並列化できそうか?

- **まず、多重ループに注目する**
  - 多重ループの処理時間はNのべき乗に比例するので、特に割合が大きい。
  - 今回のプログラムでは、2重ループが2か所（右図　A，B）
- **例えば、以下の計算機の場合、処理時間の比率は 全体に対して　Aが 35.8% 、Bが 61.5%**
  - Fujitsu PRIMEQUEST（CPU: Intel Itanium2 1.6GHz）
  - これらを効率よく並列化できれば、十分な高速化が期待できる。
- **今回は　Bの並列化のみ紹介**
  - Aはデータの初期値の設定部分なので、通常のループとは事情が少し違う。
    - Aの初期設定は最初に一回だけ実行される。一方Bは、実用プログラムでは何度も実行されることが多い。
      - Bのみの並列化でも十分な高速化が見込める。
    - 実用プログラムでは、初期データをファイルから読み出す場合等、単純な並列化ができない場合もある。

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

#define N 100
double a[N][N], b[N], c[N];

int main(int argc, char *argv[])
{
  int i, j;

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

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

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

  for (i = 0; i < N; i++)
    printf("(%d: %.2f) ", i, c[i]);
  printf("¥n");

  return 0;
}
```

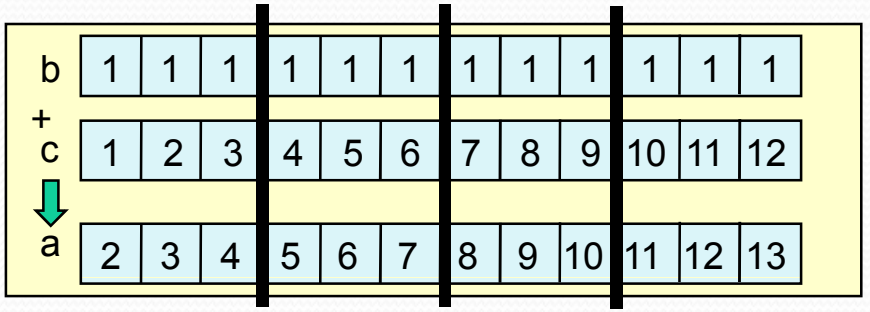
… A

… B

そのプログラムは並列化できるか？

- **並列処理 ＝ 複数のCPUコアに仕事を分けて、同時進行で処理を行う**
  - 仕事の順番が不確定
- **並列化できるプログラム**
  ＝ 実行の順番が変わっても
  結果が変わらないプログラム。
  - 例

```
for (i = 0; i < 12; i++)
    a[i] = b[i] + c[i];
```

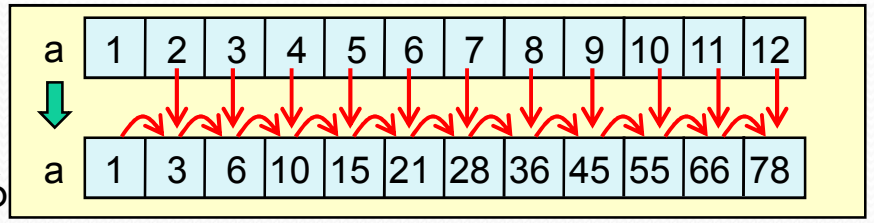


- **並列化が難しいプログラム**
  ＝ 実行の順序によって結果が変わる
  プログラム
  - 前の繰り返しで計算した値を参照して計算する
  - 例

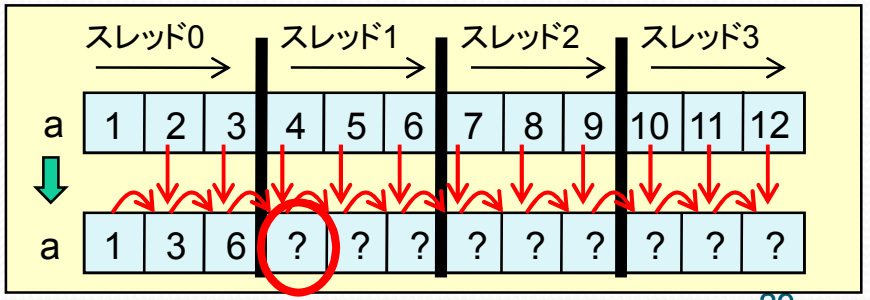
```
for (i = 1; i < 12; i++)
    a[i] = a[i] + a[i-1];
```

  - 例えば右図でスレッド0がa[2]を計算する前にスレッド1がa[3]の計算をすると正しい計算ができない。

並列化前



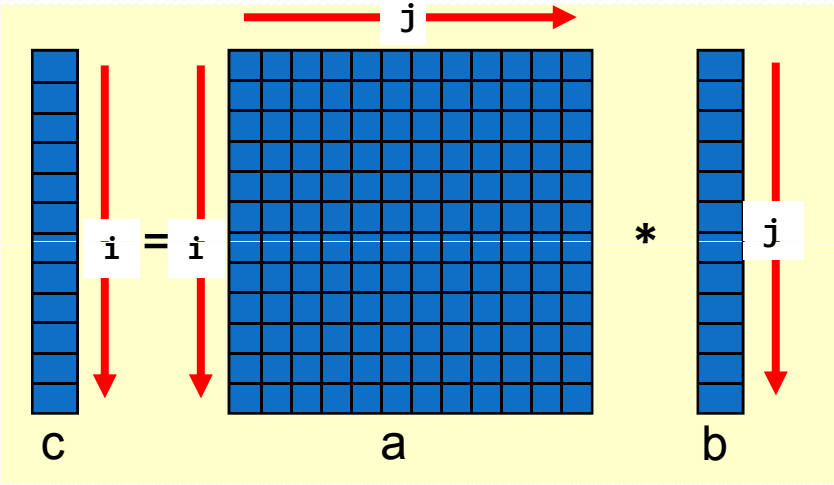
並列化後



今回のプログラムの場合

- 2重ループの内側と外側のどちらを並列化の対象とするかで状況が違う。

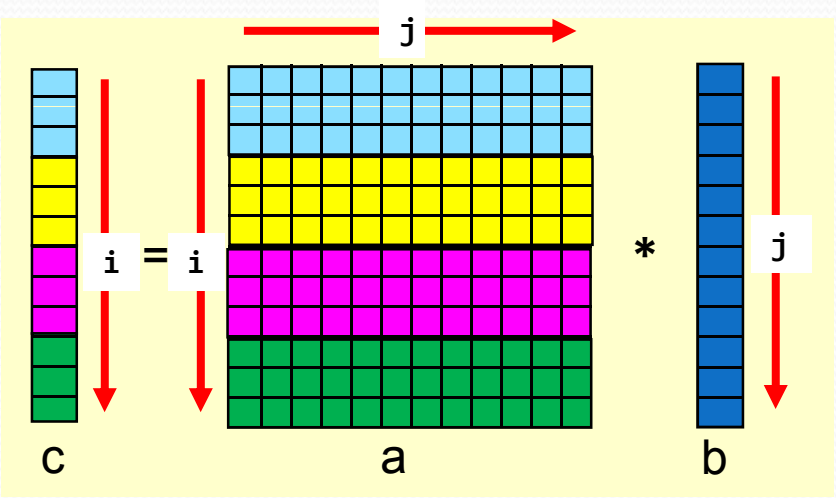
```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    c[i] += a[i][j] * b[j];
```



- 外側（ループi）: i = 0, … , N-1 それぞれについて、c[i] の値を別々に計算
  - 明らかに並列化可能
- 内側（ループj）: c[i] は j = 0, …, N-1の計算値の総和
  - ループ毎の計算結果の総和が必要

外側のループを並列化する場合

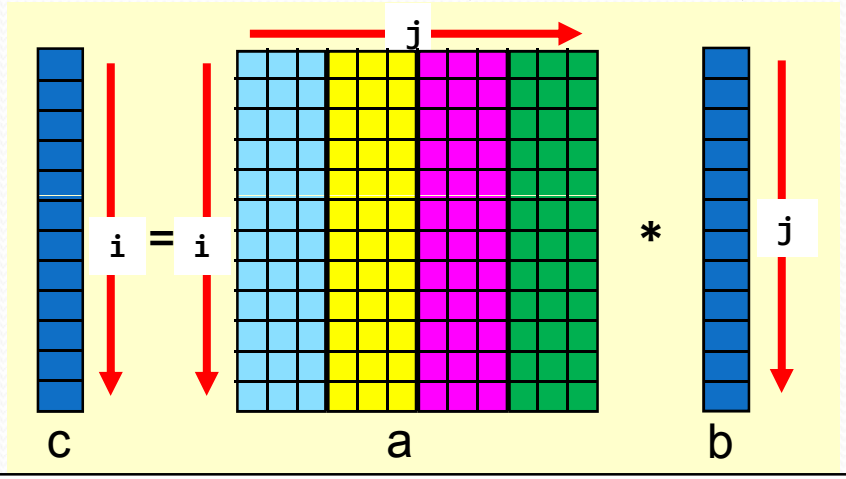
- 外側（ループ i）: i = 0, … , N-1 それぞれについて、c[i] の値を別々に計算
  - ⇒ 前の繰り返しでの計算値を使って計算することは無い。
  - ⇒ 明らかに並列化可能




```
#pragma omp parallel for private(j)
   for (i = 0; i < N; i++)
     for (j = 0; j < N; j++)
       c[i] += a[i][j] * b[j];
```

内側のループを並列化する場合

- 内側（ループｊ）: c[i] は j = 0, …, N-1の計算値の総和
  - ⇒ reduction指示節を利用
    C言語では reduction指示節に配列を使えないので、一時変数 t を利用

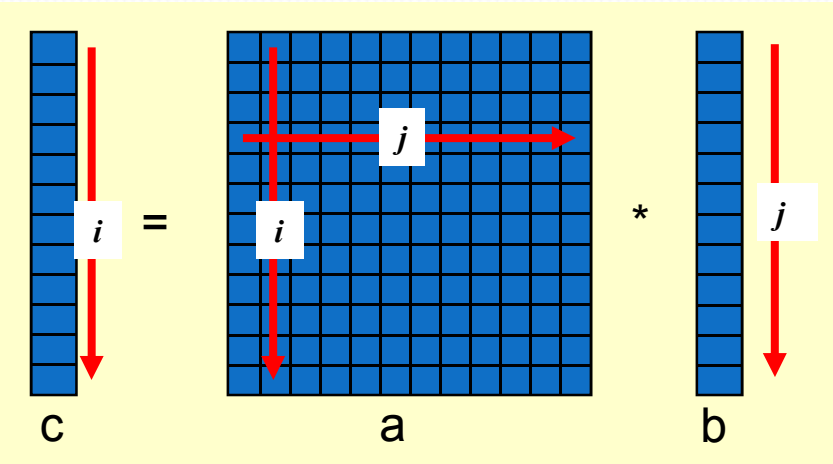



```
  for (i = 0; i < N; i++){
    t = 0.0;
#pragma omp parallel for reduction(+: t)
    for (j = 0; j < N; j++)
      t += a[i][j] * b[j];
    c[i] = t;
  }
```

では、どのループを並列化するか?

- 基本的な考え方:
  1. なるべく外側のループを並列化
     - 並列ループの開始処理と終了処理に余分なコストが必要
  2. reductionは出来るだけ使わない
     - 一般に、reductionの処理は時間がかかる
- 今回は、外側のループを並列化した方が効率が良さそう。
- 実際には一長一短な場合があり、やってみないと分からないことも多い。
  - 計算機によっても違う場合がある。

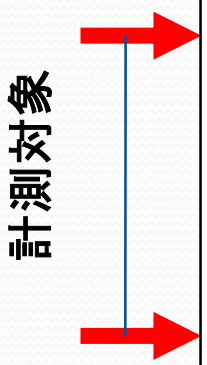
```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    c[i] += a[i][j] * b[j];
```



OpenMPプログラムの時間計測
実行時ルーチン omp_get_wtime()

- **現在時刻(秒)を実数で返す関数**
  - 経過時間 (wall clock time) の計測に利用

- **利用例**




```
#include <omp.h>
...
double t1, t2;
...
  t1 = omp_get_wtime();
#pragma omp parallel for private(j)
   for (i = 0; i < N; i++)
     for (j = 0; j < N; j++)
       c[i] += a[i][j] * b[j];
  t2 = omp_get_wtime();
  printf("Time: %e sec¥n", t2 – t1);
```

経過時間とCPU時間

- **経過時間： 実際に経過した時間**
  - 計算機の状態（CPUコアが働いていたか、休んでいたか）によらない。
  - 並列プログラムでは、他のスレッドやプロセスを待つ時間も含めて評価するので、経過時間を用いる場合が多い。

- **CPU時間： CPUコアが、そのプログラムの実行のために働いた時間**
  - 待ち時間などは含まない
  - スレッド並列の場合、全スレッドで使用した CPU時間の合計
  - 経過時間と比較することにより、並列プログラムの実行効率を評価できる

OpenMPプログラムのコンパイル、実行

- **コンパイル：コンパイラの　OpenMPオプションを指定する**
  - PGI コンパイラ：　-mp
  - Intel コンパイラ：　-openmp
  - その他：　マニュアルで確認する

- **実行：　予め、使用するスレッド数を指定してから実行**
  - 実行コマンドは、通常のプログラムと同じ

- **使用するスレッド数の指定方法**
  `setenv OMP_NUM_THREADS スレッド数`
  （sh，bash等では `export OMP_NUM_THREADS=スレッド数`）
  - 一度設定したら、ログアウトするか、別のスレッド数で再度指定するまで有効

演習0　OpenMPプログラムの実行

- 演習用の計算機にログインして、以下を実行。

```
$ tar xf /tmp/test.tar.gz
$ cd test
$ cat vm-omp.c
$ pgcc –mp –fast –O3 vm-omp.c –o vm-omp
$ setenv OMP_NUM_THREADS 1
$ ./vm-omp
$ setenv OMP_NUM_THREADS 2
$ ./vm-omp
$ setenv OMP_NUM_THREADS 4
$ ./vm-omp
```

演習1　バッチ処理による OpenMPプログラムの実行

- **バッチ処理:　多数の利用者に共有された計算機でのプログラム実行**
  - プログラムを直接実行するのではなく、ジョブ管理システムに実行を依頼
  - ジョブ管理システムは、空いている計算機にプログラムの実行を依頼
  - 実行が終わったら、画面に出力されるはずだった結果をファイルに保存

- **続けて以下を実行。**

```
$ cat test1-omp.sh
$ qsub test1-omp.sh
Request 7129.pcj submitted to queue: PCL-A.
$ qstat
```

ジョブの受付番号

何度か qstat を実行して、自分が投入したジョブが消えてから

```
$ ls
test1-omp.sh.e???? と test1-omp.sh.o???? というファイルができていることを確認(????はジョブの番号)

$ cat test1-omp.sh.o????
```

演習2　並列化手法による性能比較

- **講義資料を参考に、vm-omp2.c のベクトル・行列積計算部分について内側ループを並列化してみる**
  - vm-omp2.c の中身は vm-omp.c と同じ
- **必要に応じて、一時変数も宣言して利用する**
- **コンパイル、実行して、元のプログラムと所要時間を比較する**
  - 特にスレッド数を変化させた場合の所要時間の違いを確認
  - 最後に
    Check Done
    と表示されるのを必ず確認する
    - 間違った並列化をして計算ミスを起こしていると、表示されない

演習3　行列積の並列化

- **行列積計算プログラム mm.c を並列化してみる**
- **並列化するループや並列化手段は自由に選択**
- **コンパイル、実行して、所要時間を確認**

  `pgcc -mp -fast -O3 mm.c -o mm`

  - ジョブの投入には test2-omp.sh を利用
  - 最後に
    Check Done
    と表示されていることを確認する
    - 間違った並列化をして計算ミスを起こしていると、表示されない

まとめ

- 並列計算機の能力を発揮させるには並列プログラムが必要

- 並列プログラムの書き方:
  スレッド並列　…　OpenMP
  プロセス並列　…　MPI

- OpenMPによる並列プログラム作成の手順:
  0. 並列化するかどうかの吟味
  1. 並列化の対象部分を選定
  2. OpenMP指示文(必要に応じて実行時ルーチンも)を追加
  3. データの扱い方(共有変数 or プライベート変数)の選択

OpenMPに関する資料

- **Intel社**
  http://www.intel.com/jp/business/japan/feature/hpc/case_documents.htm

- **OpenMP仕様（日本語訳）**
  http://www.openmp.org/mp-documents/OpenMP30spec-ja.pdf