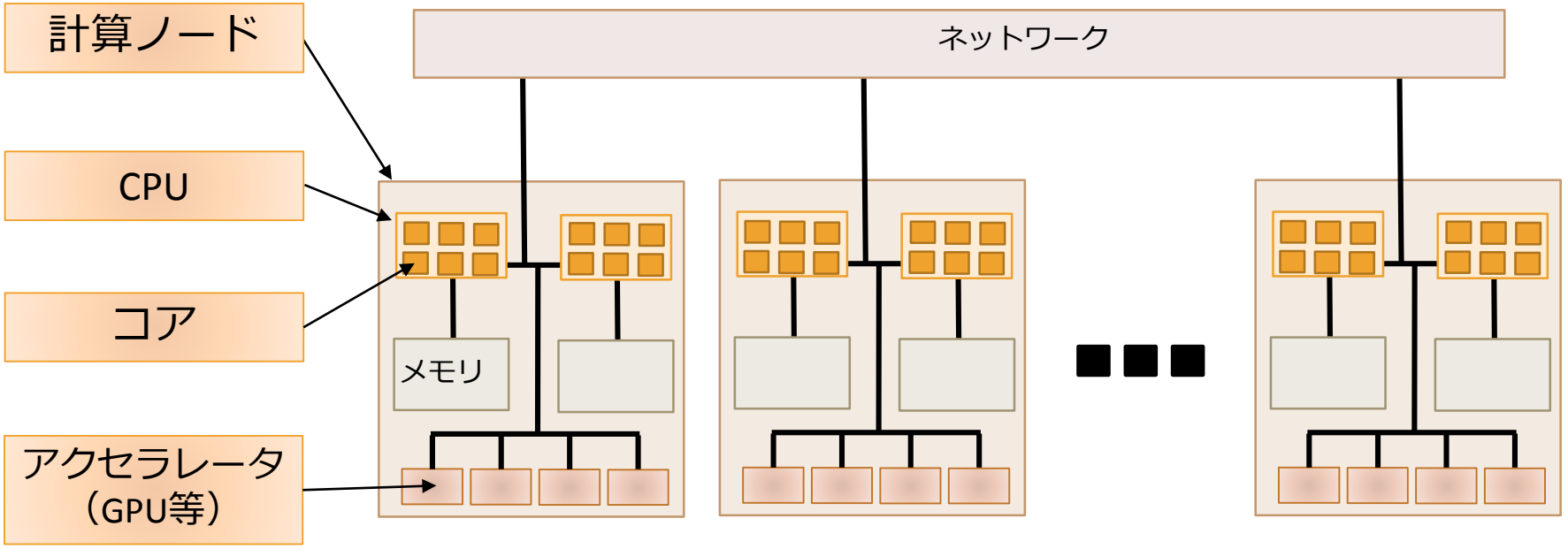
並列プログラミング超入門講習会 OpenMPコース

九州大学情報基盤研究開発センター

コアとOpenMPの関係？

並列計算機の構成

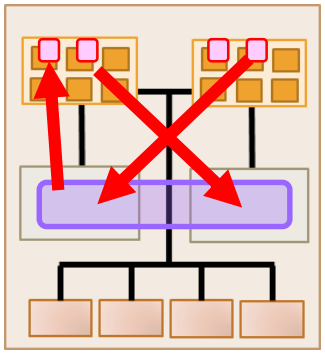


例： スーパーコンピュータシステム ITO サブシステム B

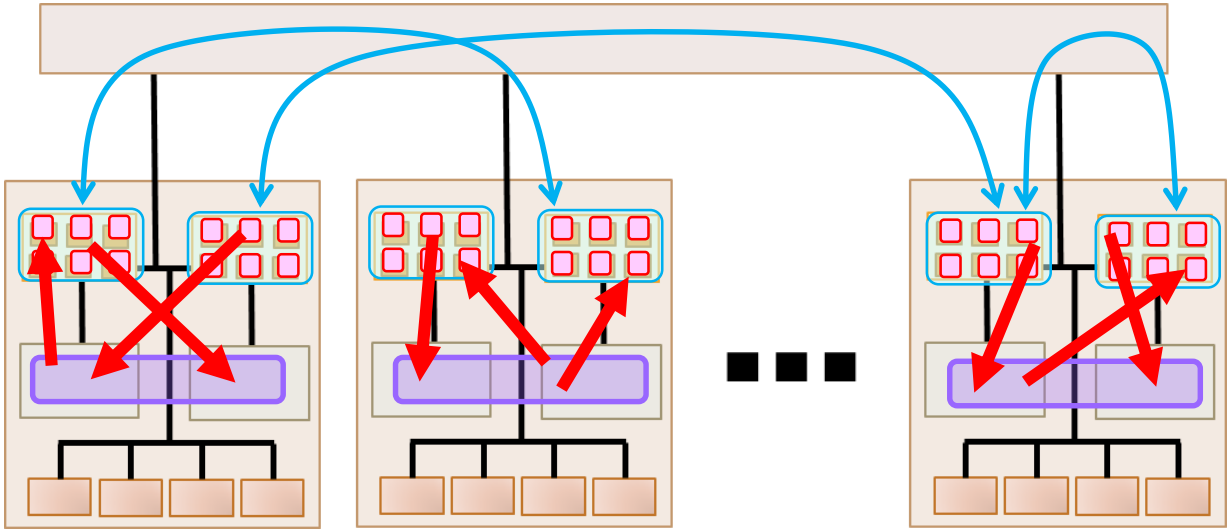
ノード数	CPU数 / ノード	コア数 / CPU	GPU数 / ノード
128	2	18	4

OpenMP

- 一つの計算ノード内の複数のコアで並列計算
  - 計算ノード内のメモリを直接読み書き



- MPIと組み合わせた利用も可能
  - MPI: 主に複数の計算ノードを用いた並列計算に利用
  - 詳しくは、MPIコースで



OpenMPコースの内容

- 第一部：
  - OpenMPプログラムの基本構成
  - コンパイルと実行
  - 実習 1
- 第二部：
  - 簡単なループの並列化
  - プライベート変数と共有変数
  - 並列プログラムの性能計測
  - 実習 2
- 第三部：
  - 並列計算結果の集計
  - 並列化出来ないプログラム
  - 実習 3
- 第四部：
  - より高度な OpenMPプログラミングに向けて

OpenMPによる並列プログラムの例

- 並列版 Hello World

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
include "omp_lib.h"

  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
```

プログラム例の実行イメージ

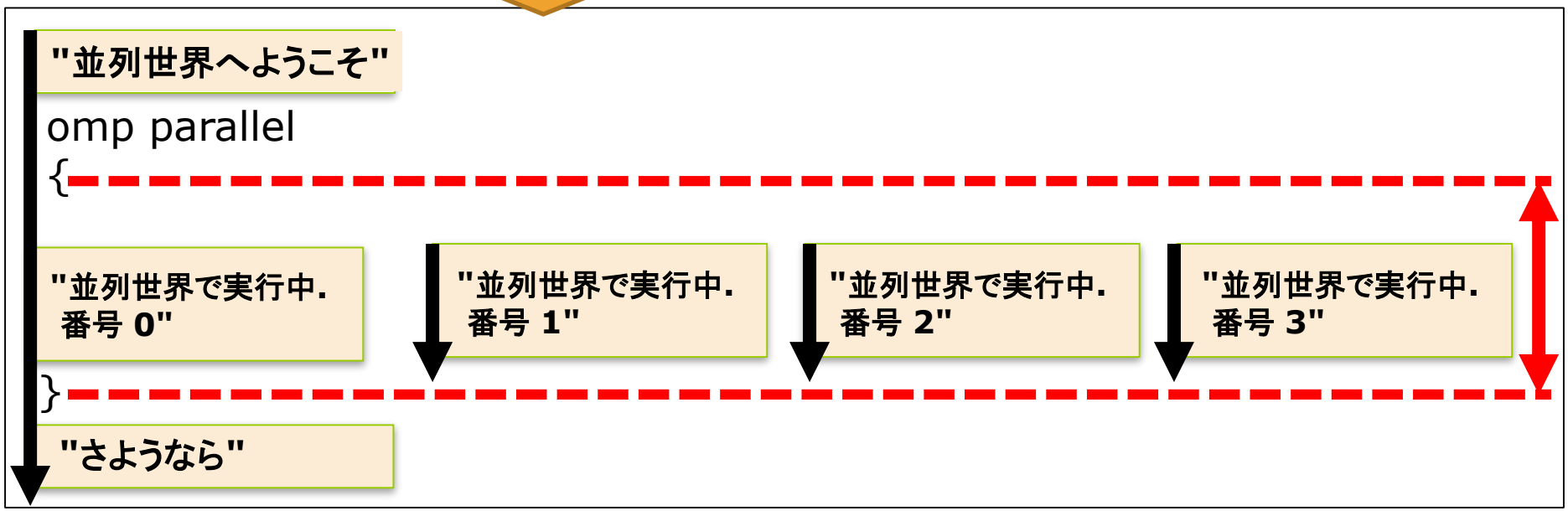
```
  printf("並列世界へようこそ¥n");
#pragma omp parallel
  {
    printf("並列世界で処理中．番号%d¥n",
           omp_get_thread_num());
  }
  printf("さようなら¥n");
```

非並列に実行する部分

並列に実行する部分

非並列に実行する部分

並列実行部分を、複数の流れで実行

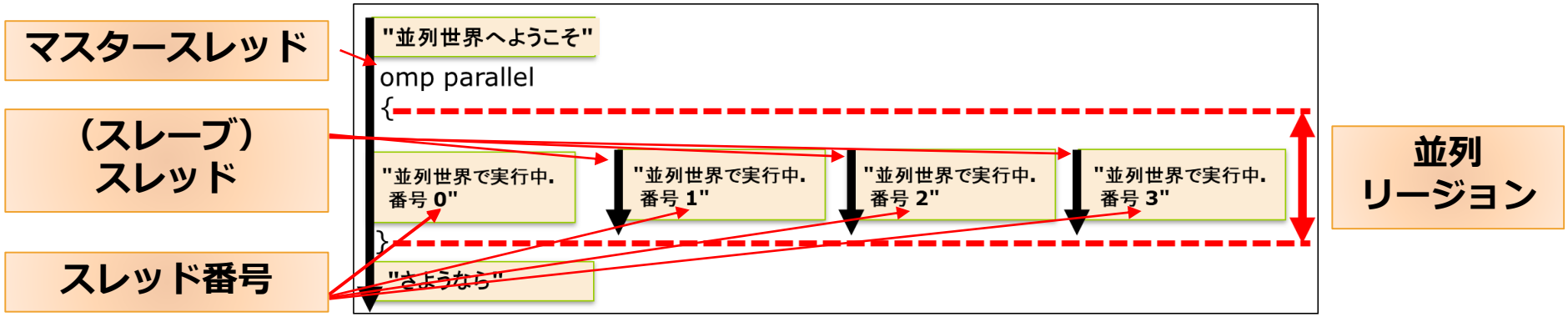


OpenMPのスレッドと並列リージョン

- スレッド = 一つの記憶空間上で実行される処理の流れ
  - 各スレッドは「スレッド番号」で識別
    - OpenMPでは、omp_get_thread_num() 関数で参照

- 並列リージョン = 複数のスレッドで実行する範囲

  - 逐次リージョン = 一つのスレッドで実行する範囲
  - マスタースレッド = 逐次リージョンを実行するスレッド
  - スレーブスレッド = 並列リージョンのみ実行するスレッド
  - チーム = 並列リージョン内のスレッド群



OpenMPプログラムの構成

- C/C++、Fortranのプログラムに
  「OpenMP指示文」、「OpenMPライブラリルーチン」
  を追加して並列化

```c
#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
include "omp_lib.h"

 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指示文

- 並列化の対象範囲や方法等を指示
  - C/C++: **#pragma omp ～**
  - Fortran: **!$omp ～**

- OpenMPに対応しないコンパイラには、コメント行として見える
  - その場合も、基本的に、計算結果はほぼ変わらない
    - 計算順序の違いによる丸め誤差の変化はある（後述）

```
#pragma omp parallel
  {
    printf("並列世界で処理中. 番号%d¥n",
           omp_get_thread_num());
  }
```

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

OpenMPの実行時ライブラリルーチン

- OpenMPプログラム内で呼び出す関数群
  - 主な関数

関数名	機能
omp_get_thread_num()	自分のスレッド番号取得
omp_get_max_threads()	並列リージョンで起動できるスレッド数取得
omp_get_num_threads()	並列リージョン内で、現在実行中のスレッド数取得
omp_get_wtime()	経過時間（秒）の取得

- 使用時はヘッダファイルを include
  - C/C++: **#include <omp.h>**
  - Fortran: **include "omp_lib.h"**
    - Fortran90 では **use omp_lib も可**

OpenMPプログラムのコンパイル

- コンパイラが提供する OpenMPオプションを指定
  - C / C++, Fortran共通

コンパイラ	OpenMPオプション
Intel	**-qopenmp**
GNU	**-fopenmp**
富士通	**-Kopenmp**

  - 例）

```
icc -qopenmp test.c -o test
```

OpenMPプログラムの実行

- 環境変数 OMP_NUM_THREADS でスレッド数を指定
  - 例）

```
export OMP_NUM_THREADS=16
./test
```

- スレッド数の決め方
  - 使用可能なコア数と、プログラムの性質に応じて設定

スレッド数	説明
スレッド数 = コア数	一般的な選択
スレッド数 < コア数	一定のスレッド数を超えると性能が上がらない、もしくは性能が低下する場合に選択
スレッド数 > コア数	計算よりもファイル I/Oや通信が主体で、コアが遊んでしまうようなプログラムの実行時に選択

実習１ OpenMPプログラムの実行

- ITOにログイン
- 実習用ファイル（/home/tmp/openmp/ompex-2018）を
  自分のホームにコピーし、ompex-2018ディレクトリに移動

```
cp -r /home/tmp/openmp/ompex-2018 .
cd ompex-2018
```

- Intelコンパイラの環境設定後、プログラムをコンパイル

```
module load intel/2017
icc -qopenmp ex1.c -o ex1
```

もしくは

```
module load intel/2017
ifort -qopenmp ex1.f90 -o ex1
```

- ログインノードで実行 （スレッド数 4）

```
export OMP_NUM_THREADS=4
./ex1
```

- ジョブスクリプトを確認し、ジョブ投入

```
cat ex1.sh
pjsub ex1.sh
```

- 実行結果確認

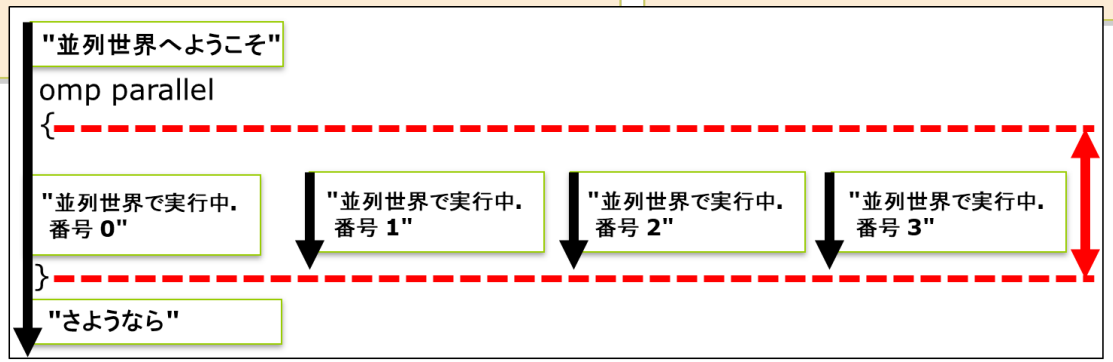
```
ls
cat ex1.sh.oジョブ番号
```

並列リージョンの指定

- parallel指示文
  - 並列リージョンを指定
  - 並列リージョン先頭でスレッド起動
  - 並列リージョン末尾で全スレッドの到着を待ってスレッド廃棄
    （使用するコンパイラ等によっては、廃棄せずに休止状態にする）

```
   ...
#pragma omp parallel
{
   ...
}
   ...
```

```
   ...
!$omp parallel
   ...
!$omp end parallel
   ...
```

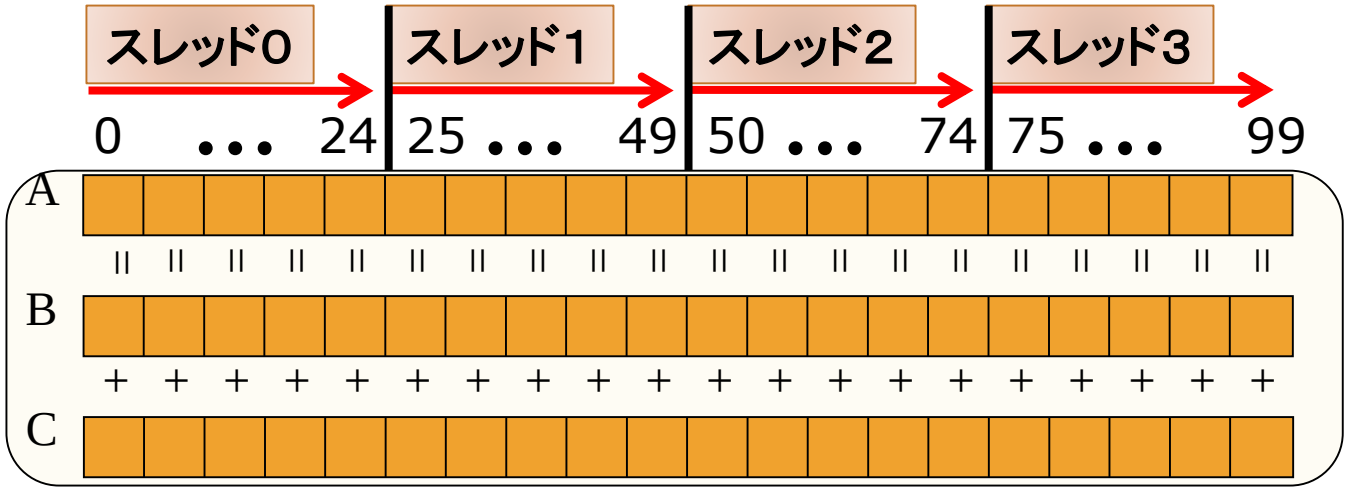


ループ並列化
for指示文、do指示文

- ループを分けてスレッドに割り当て、並列実行

```
#pragma omp parallel
{
  ...
#pragma omp for
  for (i = 0; i < 100; i++)
    a[i] = b[i] + c[i];
  ...
}
```

```
!$omp parallel
  ...
!$omp do
  do i = 0, 99
    a(i) = b(i) + c(i)
  end do
  ...
!$omp end parallel
```



parallel for (do) 指示文

- 並列リージョンの指定とループ並列化を併せて指定
  - 一つのループのみを並列化する場合に利用

```
double a[100], b[100], c[100];
  ...
#pragma omp parallel for
  for (i = 0; i < 100; i++)
    a[i] = b[i] + c[i];
```

{ } を省略可

```
real(8) :: a(0:99), b(0:99), c(0:99)
  ...
!$omp parallel do
  do i = 0, 99
    a(i) = b(i) + c(i)
  end do
```

!$omp end parallel do を省略可

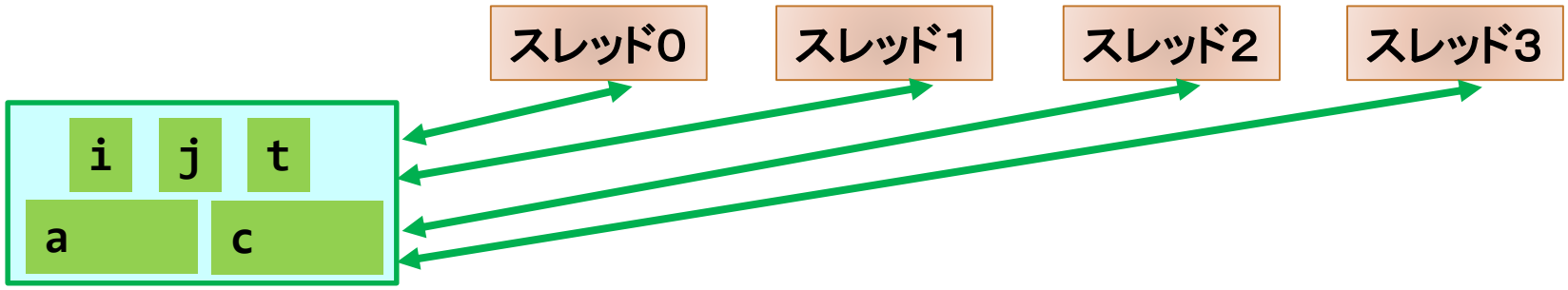
OpenMPプログラムの変数の扱い

- 基本的に、全スレッドで共有
  = 全スレッドが同じ値

- 以下のプログラムで、
  変数 i, j, t にはスレッド毎に別の値を持たせたい

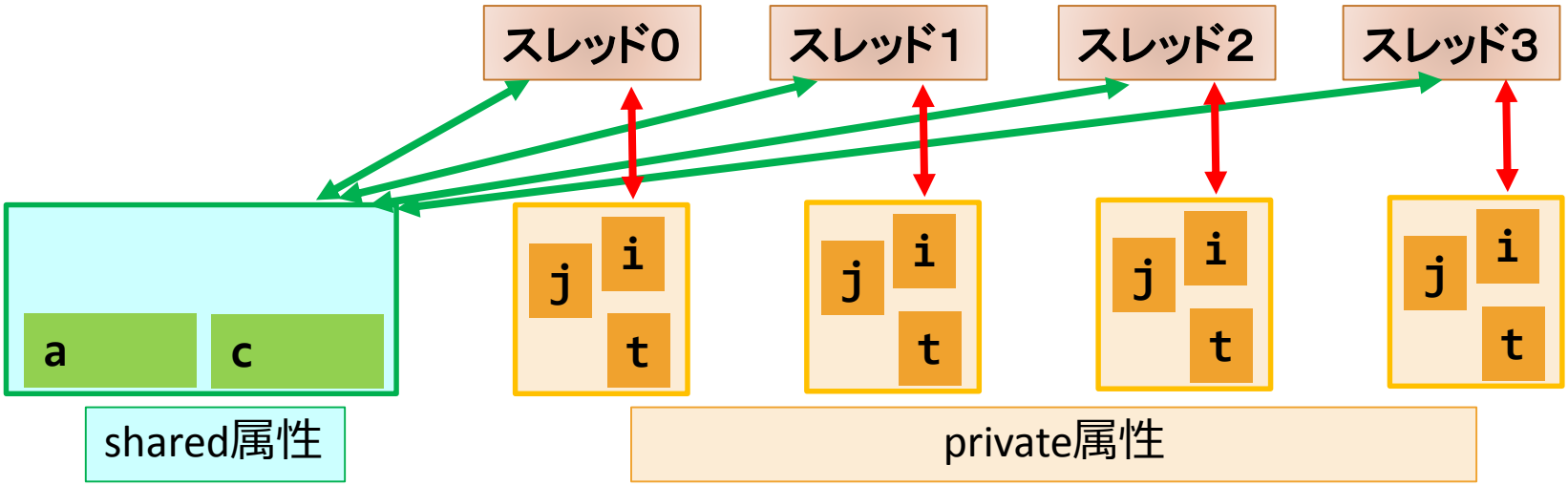
```
#pragma omp for
  for (i = 0; i < 100; i++){
    t = a[i];
    for (j = 0; j < 100; j++)
      c[i][j] = c[i][j] / t;
  }
```

```
!$omp do
  do i=0, 99
    t = a(i)
    do j=0, 99
      c(j,i) = c(j,i) / t
    end do
  end do
```



変数の属性

- private属性 = 各スレッドが独自に所持
  - スレッド毎に異なる値を持つ
- shared属性 = 全スレッドで共有
  - 全スレッドで同じ値を持つ



暗黙的に private属性になる変数

- 並列化されたループの制御変数
- （Fortranのみ）並列化されたループ内の非並列なループの制御変数

- 他の変数は、特に指定しなければ shared属性

```
#pragma omp for
  for (i = 0; i < 100; i++){
    t = a[i];
    for (j = 0; j < 100; j++)
      c[i][j] = c[i][j] / t;
  }
```

```
!$omp do
  do i=0, 99
    t = a(i)
    do j=0, 99
      c(j,i) = c(j,i) / t
    end do
  end do
```

属性の指定

- 暗黙的に決まる属性からの変更

- parallel指示文や for (do) 指示文等の指示文に「指示節」を追加
  - private指示節： private属性の変数
  - shared指示節： shared属性の変数

```
#pragma omp for private(j, t)
  for (i = 0; i < 100; i++){
    t = a[i];
    for (j = 0; j < 100; j++)
      c[i][j] = c[i][j] / t;
  }
```

```
!$omp do private (t)
  do i=0, 99
    t = a(i)
    do j=0, 99
      c(j,i) = c(j,i) / t
    end do
  end do
```

並列プログラムの性能計測

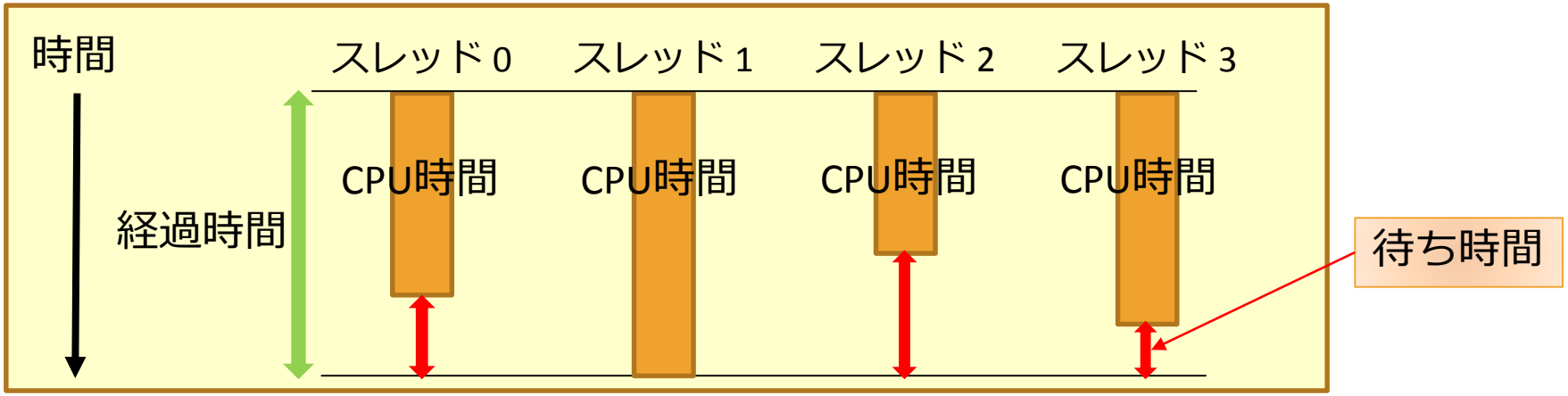
- 通常、「経過時間」で評価
  - omp_get_wtime()、gettimeofday()、system_clock()等を利用

```
double st, ft;

...

st = omp_get_wtime();

  計測対象のコード

ft = omp_get_wtime();

printf("Time: %e (sec)¥n",ft-st);
```

```
real(8) :: st, ft

...

st = omp_get_wtime()

  計測対象のコード

ft = omp_get_wtime()

print *,"Time: ",ft-st," (sec)"
```

経過時間と CPU時間

- 経過時間
  - wall clock time とも呼ばれる、実行に要した時間
- CPU時間
  - 実行中に CPUが動作した時間
  - OpenMPでは、全スレッドの CPU時間の合計値
  - 他のスレッドの完了を待つ時間など、
    スレッドが CPUコアを使えなかった時間は含まれない



実習２ ループの並列化と時間計測

- ex2.c もしくは ex2.f90 をもとに並列化したプログラム ex2-omp.c もしくは ex2-omp.f90を作成
  - 修正内容：
    - ２重ループ（行列の和）の外側ループを並列化
    - 並列リージョンの所要時間を計測して表示するコードを追加
- コンパイルし、実行ファイル ex2-ompを作成
- スレッド数を 1, 2, 4, 8, 16 と変えて実行し所要時間を確認
  - ログインノードで実行

```
export OMP_NUM_THREADS=スレッド数
./ex2-omp
```

  - ジョブ投入

```
pjsub ex2-omp.sh
```

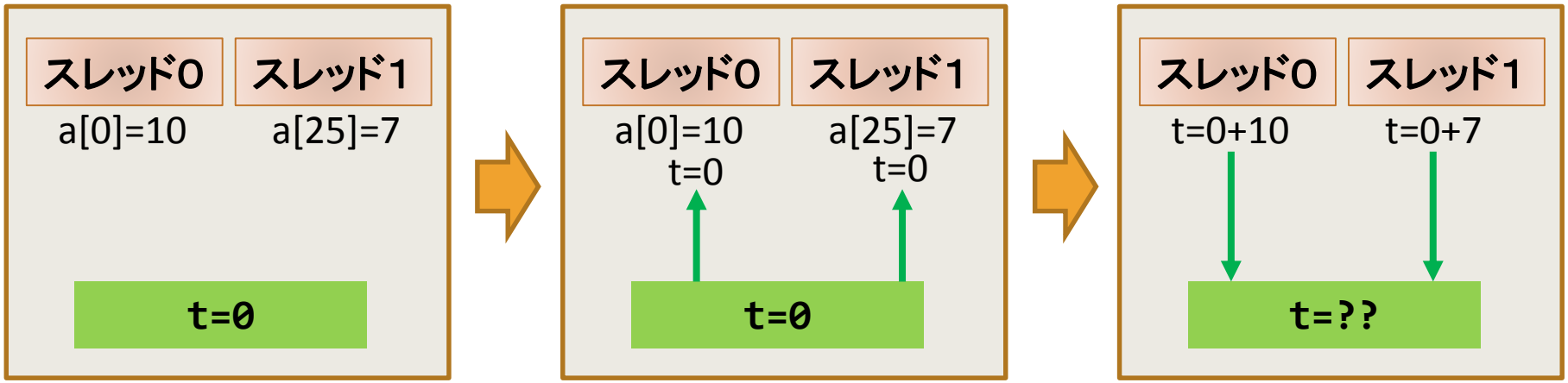
- 時間に余裕があれば、内側ループを並列化した場合と性能を比較

並列計算結果の集計

- 変数 tが shared属性だと、正しく集計できない
  - 複数のスレッドが同時に tを更新しようとするため

```
  t = 0.0;
#pragma omp for
  for (i = 0; i < 100; i++)
    t += a[i];
```

```
  t = 0.0
!$omp do
  do i = 0, 99
    t = t + a(i)
  end do
```

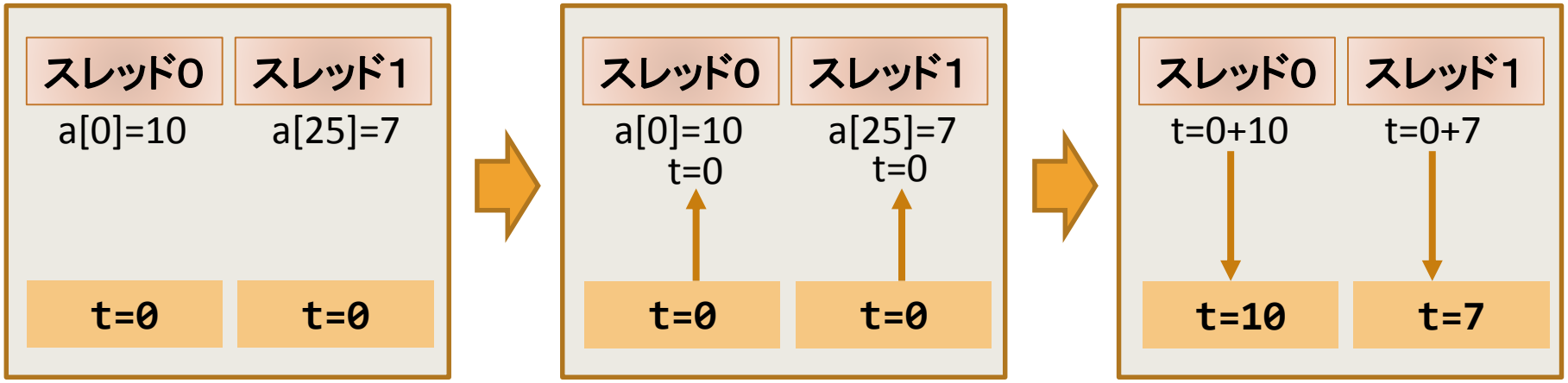


並列計算結果の集計（つづき）

- private属性にしても解決しない

```
  t = 0.0;
#pragma omp for private (t)
  for (i = 0; i < 100; i++)
    t += a[i];
```

```
  t = 0.0
!$omp do private (t)
  do i = 0, 99
    t = t + a(i)
  end do
```

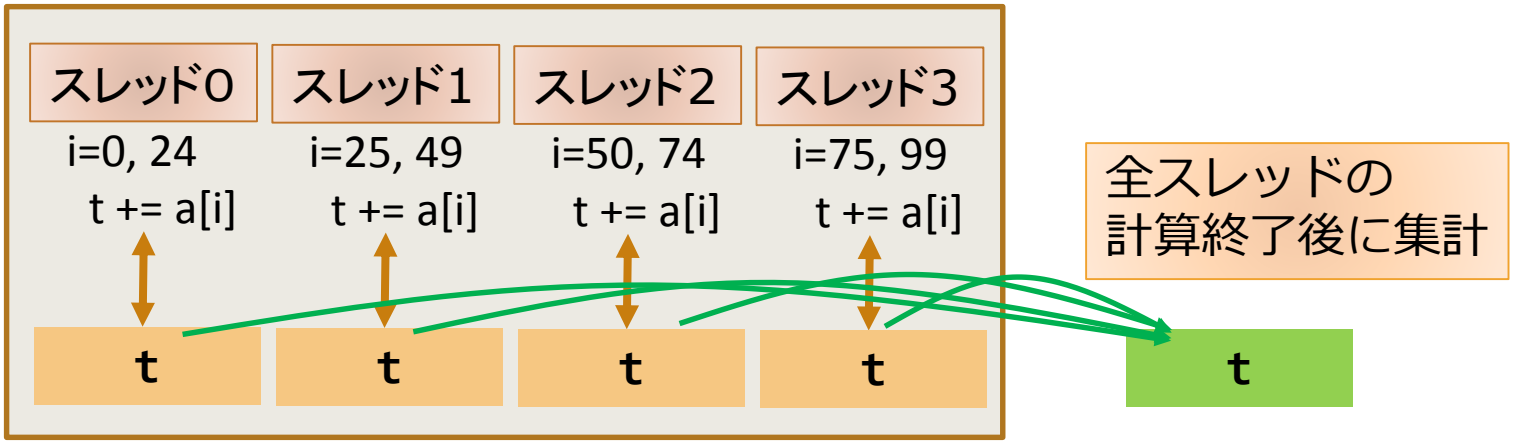


reduction指示節

- 全スレッドで集計処理をする変数を指定（複数指定可）
  - 並列計算中は一時的な private属性の変数で計算
  - 並列計算終了時、指定した演算で集約して格納（shared属性）
- 計算順序が非並列時と違う ⇒ **丸め誤差の変化に注意**

```
  t = 0.0;
#pragma omp for reduction (+: t)
  for (i = 0; i < 100; i++)
    t += a[i];
```

```
  t = 0.0
!$omp do reduction (+: t)
  do i = 0, 99
    t = t + a(i)
  end do
```



C/C++とFortranのreduction指示節の違い

- 集計演算の種類

C/C++	+, *, -, &, |, ^, &&, ||
Fortran	+, *, -, .and., .or., .eqv., .neqv., max, min, iand, ior, ieor

- 配列の集計の可否：
  Fortranは可
  C / C++は**不可** ⇒ 一時変数を使用

```
for (i = 0; i < 100; i++) {
  t = 0.0;
#pragma omp for reduction (+: t)
  for (j = 0; j < 100; j++)
    t += a[i];
  c[i] = t;
}
```

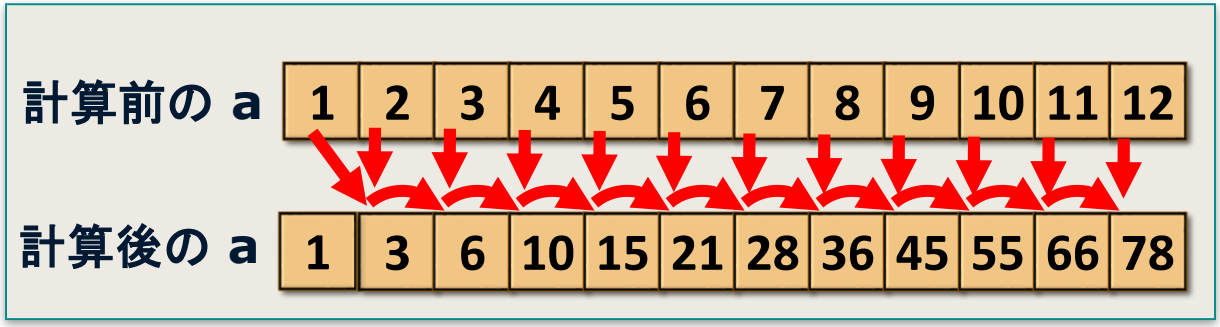
```
do i = 0, 99
!$omp do reduction (+: c)
  do j = 0, 99
    c(i) = c(i) + a(j, i)
  end do
  c(i) = t
end do
```

並列化すると結果が不正になるループ（並列化前）

- 前の計算結果をもとに、次の計算を実行

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

```
do i = 1, 11
  a(i) = a(i) + a(i-1)
```



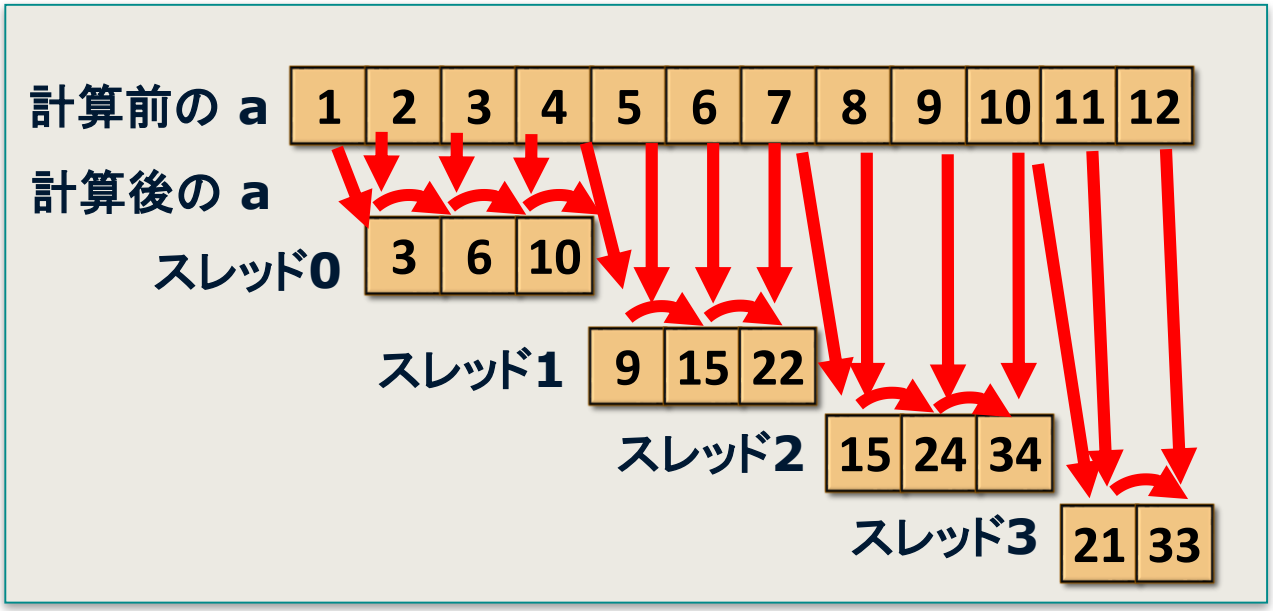
並列化すると結果が不正になるループ（並列化後）

- 各スレッドが前のスレッドの計算を待たずに計算開始

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

```
!$omp do
do i = 1, 11
  a(i) = a(i) + a(i-1)
```

4スレッドが同時に計算を開始した場合



並列化により計算結果が変わるループの例

- ループ中で、以前の値を参照した後、上書きする場合

```
for (i = 0; i < 11; i++)
    a[i] = a[i] + a[i+1];
```

```
do i = 0, 10
  a(i) = a(i) + a(i+1)
```

- 間接参照しているプログラムで、結果が同じ場所に書き込まれる可能性がある場合
  - 例えば以下のプログラムで b[i] == b[9] の場合

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

```
do i = 0, 11
  a(b(i)) = c(i) + d(i)
```

注意：
コンパイラは、計算結果が変わるかどうかチェックしない
⇒プログラマの責任で並列化

コンパイラによるループの解析

- Intelコンパイラの最適化リポート

```
icc -opt_report -qopenmp test.c -o test
```

```
ifort -opt_report -qopenmp test.f90 -o test
```

- 生成されるリポートファイル ～.optrpt で、
  並列化したいループの行番号のメッセージ確認
  - 以下のように、依存関係に関するメッセージがある場合、
    並列化によって結果が変わる可能性があるので、要注意

```
ループ はベクトル化されませんでした: ベクトル依存関係がベクトル化を妨げています。
```

実習 3

- ex3-1.c もしくは ex3-1.f90 を並列化し、ex3-1-omp.c もしくは ex3-1-omp.f90 を作成
  - 1重ループを reduction指示節付きで並列化
- プログラムをコンパイルし ex3-1-omp を作成
- スレッド数を変化させながらログインノード、およびジョブで実行し、実行時間を確認
  - 時間があれば、reduction指示節を外したものと結果を比較

- ex3-2.c もしくは ex3-2.f90 を並列化し、ex3-2-omp.c もしくは ex3-2-omp.f90を作成
  - １重ループを並列化
- プログラムをコンパイルし ex3-2-omp を作成
- スレッド数を変化させながらログインノード、およびジョブで実行し、結果を確認

より高度なOpenMPプログラミングに向けて

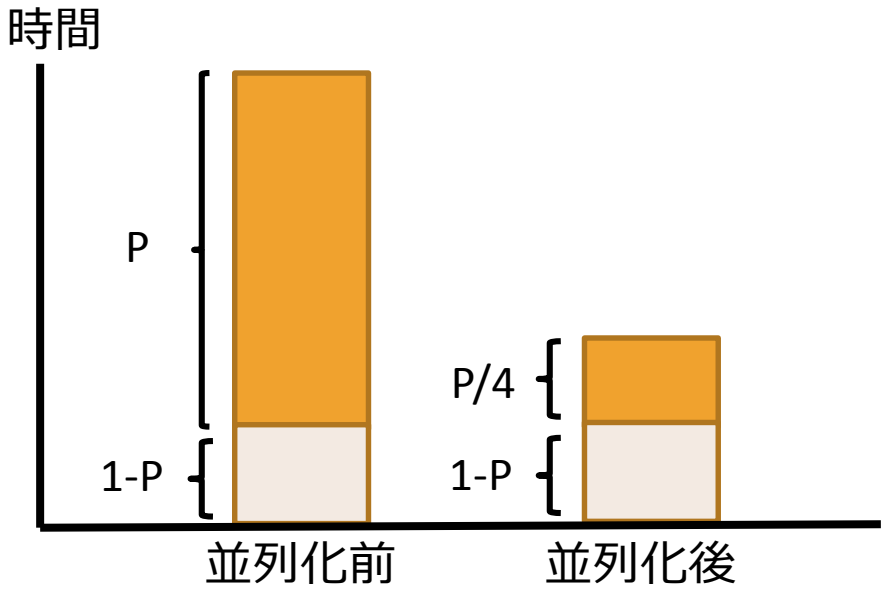
- 効率的な並列計算に向けたテクニック
  - 並列化すべきループの選択
  - 繰り返し毎の計算量が違うループの並列化
  - スレッドの待ち合わせ場所の調整
- 同じファイルで並列版と非並列版を切り替える方法
- 複雑なプログラムの並列化手段
  - 多重ループの並列化
  - ループ以外の並列化
  - 細かいスレッド制御

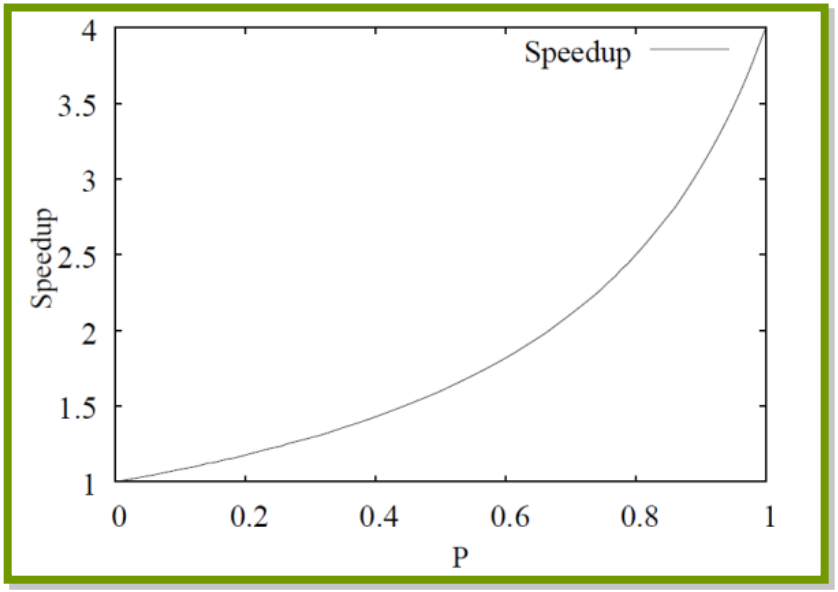
並列化するループの選択

- 並列化しても結果が変わらないループ
- **より時間のかかっているループ**
- **より外側のループ**

時間のかかるループから並列化

- アムダールの法則：
  「改良した部分以外は速くならない」
  - 例）ある部分を並列化で４倍高速化できたとすると
    高速化率　＝ 1 / ((1-P) + P/4)
    - P = 並列化できた部分の割合





外側のループから並列化

- 並列ループ開始、終了処理に時間を要するため
- 例）
  - 外側ループを並列化：

```
#pragma omp for private(j, t)
  for (i = 0; i < 100; i++){
    t = a[i];
    for (j = 0; j < 100; j++)
      c[i][j] = c[i][j] / t;
  }
```

```
!$omp do private (t)
  do i=0, 99
    t = a(i)
    do j=0, 99
      c(j,i) = c(j,i) / t
    end do
  end do
```

  - 内側ループを並列化：

```
   for (i = 0; i < 100; i++){
    t = a[i];
#pragma omp for private(t)
    for (j = 0; j < 100; j++)
      c[i][j] = c[i][j] / t;
  }
```

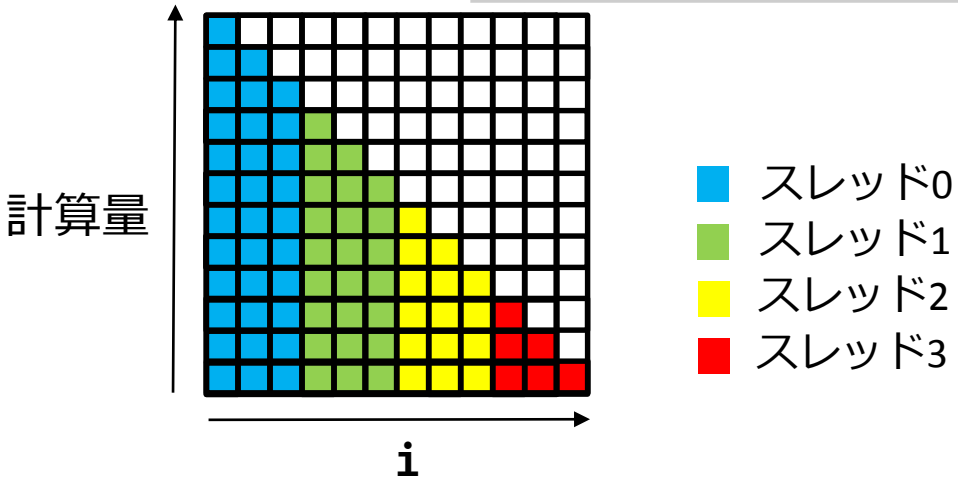
```
  do i=0, 99
    t = a(i)
!$omp do private (t)
    do j=0, 99
      c(j,i) = c(j,i) / t
    end do
  end do
```

繰り返し毎の計算量が違うループの並列化

- 例えば以下のループは、i に応じて計算量が減るため、スレッド毎の計算量に大きな差

```
#pragma omp for
  for (i = 0; i < 12; i++){
    t = a[i];
    for (j = i; j < 12; j++)
      c[i][j] = c[i][j] / t;
  }
```

```
!$omp do
  do i=0, 11
    t = a(i)
    do j=i, 11
      c(j,i) = c(j,i) / t
    end do
  end do
```

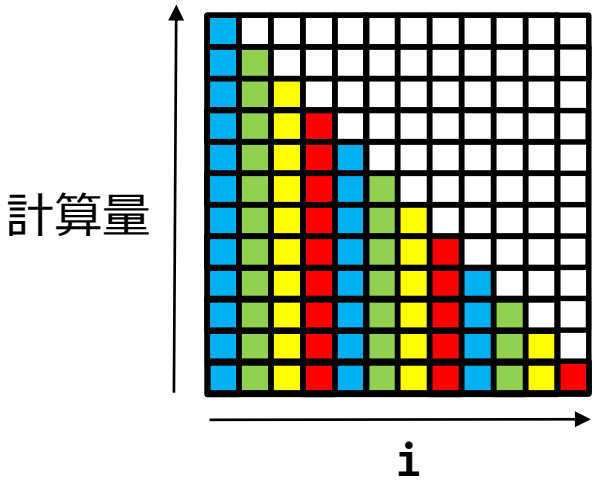


schedule指示節による割り当て調整 static

- 割り当て単位を小さくする
  - 例） 繰り返し1回ずつ割り当て

```
#pragma omp for schedule(static, 1)
  for (i = 0; i < 12; i++){
    t = a[i];
    for (j = i; j < 12; j++)
      c[i][j] = c[i][j] / t;
  }
```

```
!$omp do schedule(static, 1)
  do i=0, 11
    t = a(i)
    do j=i, 11
      c(j,i) = c(j,i) / t
    end do
  end do
```

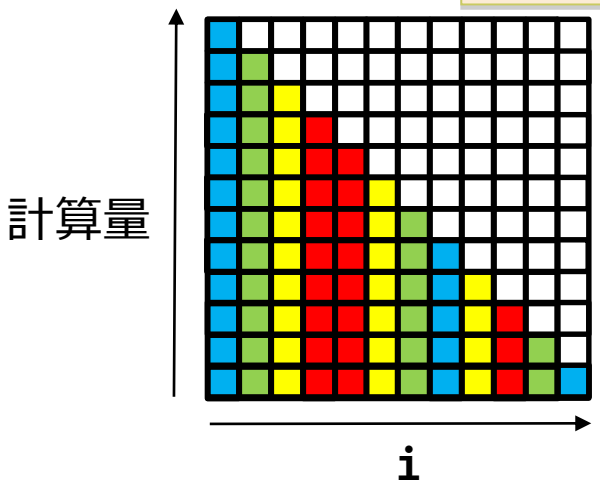


schedule指示節による割り当て調整 dynamic

- 実行中に、速いスレッド順に割り当てる
  - 例）繰り返し1回分ずつ、速い順に割り当て
  - より均等に割り当て可能だが、staticより割り当て処理に時間を要する

```
#pragma omp for schedule(dynamic, 1)
  for (i = 0; i < 12; i++){
    t = a[i];
    for (j = i; j < 12; j++)
      c[i][j] = c[i][j] / t;
  }
```

```
!$omp do schedule(dynamic, 1)
  do i=0, 11
    t = a(i)
    do j=i, 11
      c(j,i) = c(j,i) / t
    end do
  end do
```

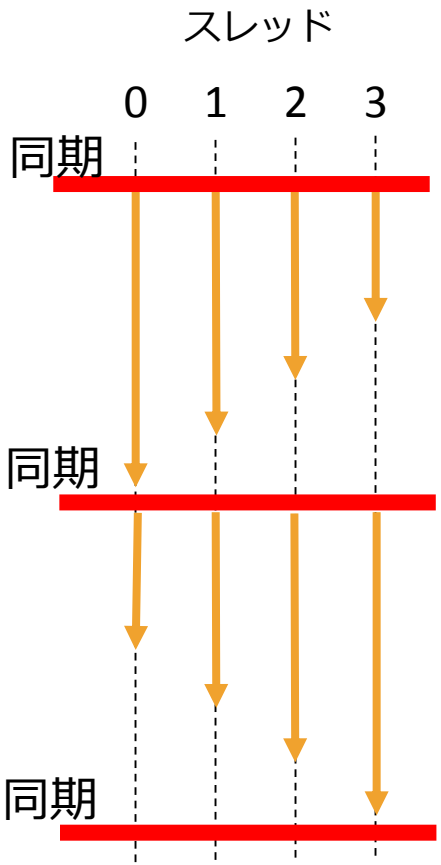


スレッドの待ち合わせ場所の調整

- スレッドの待ち合わせ = 同期
- 並列ループ等の後、暗黙的に同期を実行

```
#pragma omp parallel
{
#pragma omp for
  for (i = 0; i < 12; i++){
    t = a[i];
    for (j = i; j < 12; j++)
      c[i][j] = c[i][j] / t;
  }
#pragma omp for
 for (i = 0; i < 12; i++){
    t = a[i];
    for (j = 0; j < i; j++)
      c[i][j] = c[i][j] / t;
  }
}
```

```
!$omp parallel

!$omp do
  do i=0, 11
    t = a(i)
    do j=i, 11
      c(j,i) = c(j,i) / t
    end do
  end do
!$omp do
  do i=0, 11
    t = a(i)
    do j=0, i
      c(j,i) = c(j,i) / t
    end do
  end do
!$omp end parallel
```

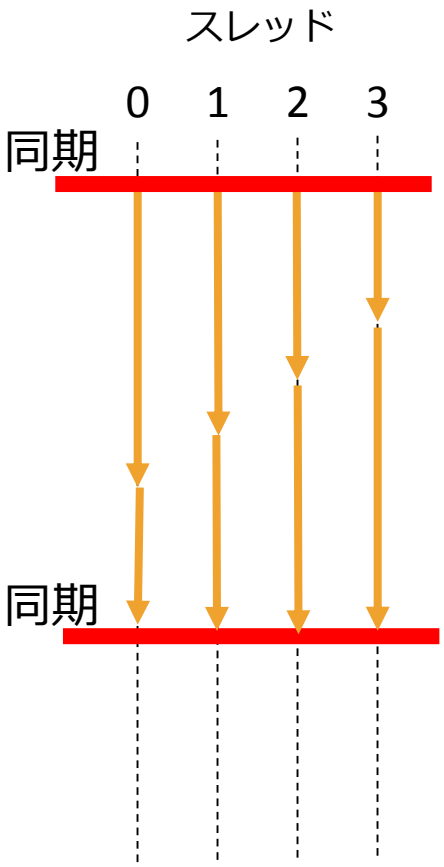


不要な同期を実施しない

- nowait指示節を追加

```
#pragma omp parallel
{
#pragma omp for nowait
  for (i = 0; i < 12; i++){
    t = a[i];
    for (j = i; j < 12; j++)
      c[i][j] = c[i][j] / t;
  }
#pragma omp for
 for (i = 0; i < 12; i++){
    t = a[i];
    for (j = 0; j < i; j++)
      c[i][j] = c[i][j] / t;
  }
}
```

```
!$omp parallel

!$omp do
  do i=0, 11
    t = a(i)
    do j=i, 11
      c(j,i) = c(j,i) / t
    end do
  end do
!$omp end do nowait
!$omp do
  do i=0, 11
    t = a(i)
    do j=0, i
      c(j,i) = c(j,i) / t
    end do
  end do
!$omp end parallel
```



多重ループの並列化

- 並列ループの中で、さらに並列化
  - 例）4 x 4 = 16スレッドによる 2重ループの並列化

```
#pragma omp parallel for num_threads(4)
  for (i = 0; i < 12; i++){
    t = a[i];
#pragma omp parallel for num_threads(4)
    for (j = 0; j < 12; j++)
      c[i][j] = c[i][j] / t;
  }
```

```
!$omp parallel do num_theads(4)
  do i=0, 11
    t = a(i)
!$omp parallel do num_theads(4)
    do j=0, 11
      c(j,i) = c(j,i) / t
    end do
  end do
```

- 実行時に環境変数 OMP_NESTED を trueに設定

```
export OMP_NUM_THREADS=16
export OMP_NESTED=true
./a.out
```

同じファイルで並列版と非並列版を切り替え

- 条件付きコンパイル
  - コンパイル時にOpenMPが有効な場合だけ翻訳される行を指定

```
#ifdef _OPENMP
st = omp_get_wtime();
#endif

#pragma omp parallel for
  for (i = 0; i < 100; i++)
    a[i] = b[i] + c[i];

#ifdef _OPENMP
ft = omp_get_wtime();
#endif
```

```
!$ st = omp_get_wtime()

!$omp parallel do
  do i = 0, 99
    a(i) = b(i) + c(i)
  end do

!$ ft = omp_get_wtime()
```

ループ以外の並列化

section

- スレッド毎に別の処理を割り当て
  - 例）4個のルーチンを並列に実行

```
#pragma omp sections
{
#pragma omp section
  work1();
#pragma omp section
  work2();
#pragma omp section
  work3();
#pragma omp section
  work4();
}
```

```
!$omp sections
!$omp section
  call work1()
!$omp section
  call work2()
!$omp section
  call work3()
!$omp section
  call work4()
!$omp end sections
```

ループ以外の並列化

task

- 実行中に新しい仕事を割り当て
  - 再帰処理の並列化を簡潔に記述可能
  - 例）再帰ルーチンの並列化

```
void traverse( struct node *p )
{
  if (p->left)
#pragma omp task
    traverse(p->left);
  if (p->right)
#pragma omp task
    traverse(p->right);
#pragma omp taskwait
  process(p);
}
```

```
recursive subroutine traverse (p)

  if (associated(p%left)) then
!$omp task
    call traverse(p%left)
!$omp end task
  endif
  if (associated(p%right)) then
!$omp task
    call traverse(p%right)
!$omp end task
  endif
!$omp task wait
  call process(p)
end subroutine
```

from "OpenMP Application Program Interface Version 3.0", May 2008

細かいスレッド制御

- 実行するスレッドの指定

指示文	機能
single	指定した範囲を一つのスレッドで実行
master	指定した範囲をマスタースレッドで実行

- 実行順序の制御

指示文	機能
barrier	全スレッドが到着するまで待つ
critical	指定した範囲を同時に一つのスレッドだけが実行
atomic	指定した演算を同時に一つのスレッドだけが実行
ordered	並列ループ中の特定の範囲を、非並列時と同じ順序で実行
flush	それまでのメモリ書き込みが完了したことを確認

変数の属性に関する制御

指示文	機能
threadprivate	Fortranのcommonブロックやsave属性の変数をprivate属性とする

指示節	機能
firstprivate	並列リージョン開始時にマスタースレッドの値を全スレッドにコピー
lastprivate	並列リージョン終了後のマスタースレッドの値を、非並列実行時の最後の値とする
copyin	threadprivateの変数の値を、並列リージョン開始時のマスタースレッドの値で初期化
copyprivate	single指示文で指定された範囲を実行後、値を全スレッドにコピー
default	指示文や指示節で属性が指定されていない変数の暗黙的な属性を指定

OpenMPに関する情報

- OpenMPの仕様
  https://www.openmp.org/specifications/

- 書籍
  - 「並列プログラミング入門: サンプルプログラムで学ぶOpenMPとOpenACC」
    片桐 孝洋、東大出版会 (2015)
  - 「C/C++プログラマーのためのOpenMP並列プログラミング」
    菅原 清文、カットシステム (2012)
  - 「OpenMP入門マルチコアCPU時代の並列プログラミング」
    北山 洋幸、秀和システム (2009)
  - 「OpenMPによる並列プログラミングと数値計算法」
    牛島 省、丸善 (2006)