OpenMPによるスレッド並列計算

今村　俊幸
（理化学研究所　計算科学研究センター）
imamura.toshiyuki@riken.jp

KOBE HPC サマースクール（初級）
2023年9月18-22日

講義内容

1. スレッド並列とは
2. OpenMPによるループ処理の並列化
3. 差分化された偏微分方程式の並列化
4. アムダールの法則と並列化効率の評価

計算機とムーアの法則

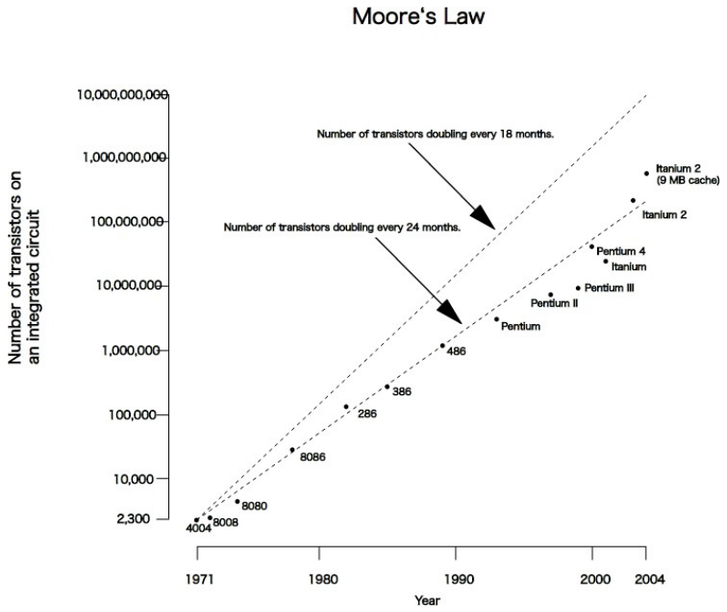
- スカラー計算機
  - 単一の実行ユニットが命令を１つ１つ、逐次処理を行う計算機。ただし近年では、複数のスカラー命令を同時に処理できるスー パースカラーが主流。一般的なパソコンやスマートフォンなど は、このスカラー計算機に分類される。
  - Intel系、AMD系、ARM系 (FUJITSU の富岳、FXシリーズ）など

- ベクトル計算機
  - ベクトル演算（同じ計算式を各配列要素に対してそれぞれ計算するといった処理）を一括して行うことができる計算機。一時期は『スーパーコンピュータ ≒ ベクトル機』と言われていたこともあった。
  - NEC-SX（地球シミュレータ[初代]）、GPUアクセラレータ等

計算機とムーアの法則

- **半導体の集積密度は１年半～２年ごとに倍になる**

- エンドユーザーからすれば、ただ待っているだけで計算機の性能が向上し続け、大規模計算が可能になるという”ありがたい話”



https://ja.wikipedia.org/wiki/%E3%83%A0%E3%83%BC%E3%82%A2%E3%81%AE%E6%B3%95%E5%89%87

しかし

近年、ムーアの法則の限界が囁かれるようになる

→そもそも原子サイズより小さな回路は実現不可能

→リーク電流、放熱の問題等

計算機とムーアの法則

- 大型計算機の主流はベクトルからスカラーへと変化してきた。また、そのスカラー計算機も、製造プロセスの微細化によるクロックの高 速化が頭打ちになっており、マルチコア化が図られるなど方針の転 換を図られてきた。近年ではGPU（アクセラレータ）を計算利用するなど、ハードウェアも多様化している。

→計算機の特性にあったプログラミングが必要

共有メモリ型

・複数のコアを持つCPU（マルチコア）、SGI UVシリーズ
⇒ **スレッド並列計算**（**OpenMP**、自動並列など）

分散メモリ型

・複数の計算機を通信システムで繋ぐ（クラスター）
⇒ **プロセス並列計算**（**MPI**、XcalableMPなど）

アクセラレータ

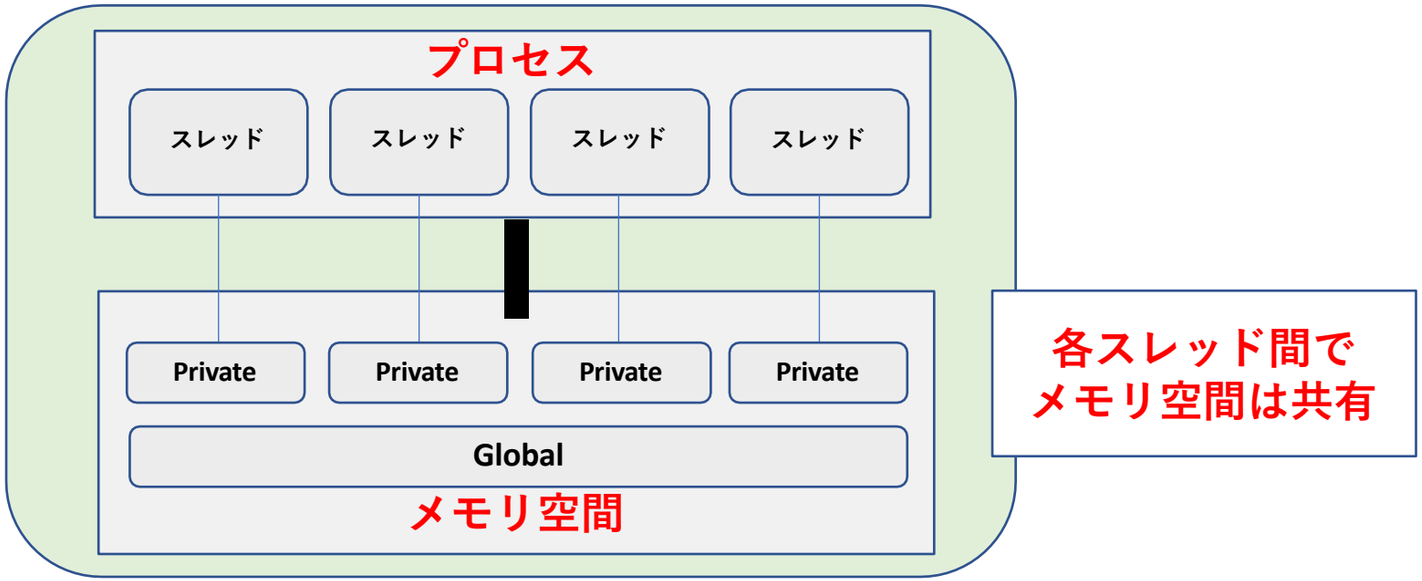
・メインのCPUに加え、GPUやXeon Phiなどを計算に用いる
⇒ **OpenACC**、OpenMP 4.0、OpenCL、CUDAなど

並列計算機に関して

- **共有メモリ型並列計算機**

  - 各演算ユニット間でメモリ空間が共有されている計算機

- 一般的なマルチコアCPU搭載のPCや、スパコンのノード １つ１つも、共有メモリ型の並列計算機と言える
- - **OpenMP**を用いた**スレッド並列計算**が可能



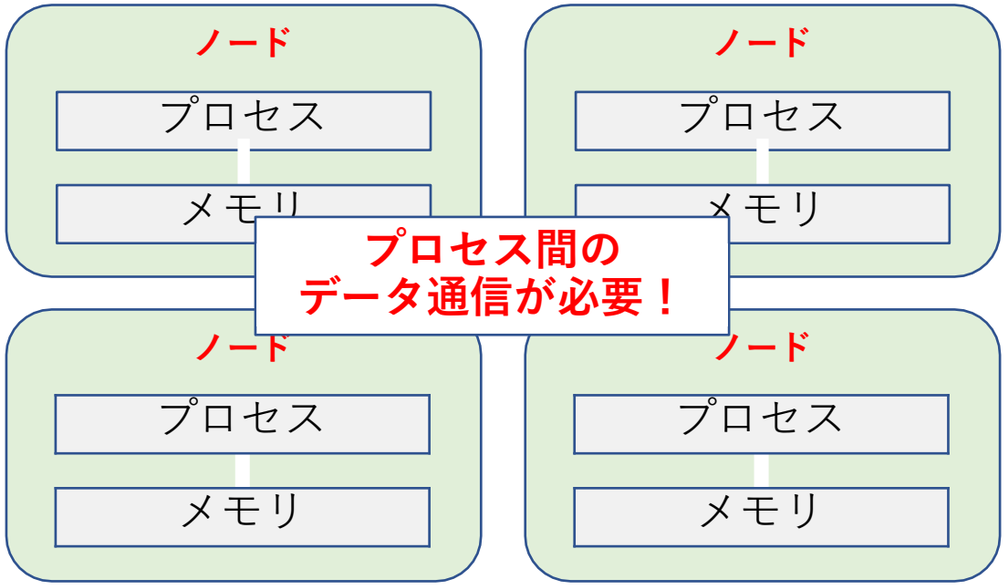
並列計算機に関して

- **分散メモリ型並列計算機**

- 独立したメモリを持つ計算ノード間を通信しながら並列動作させる計算機

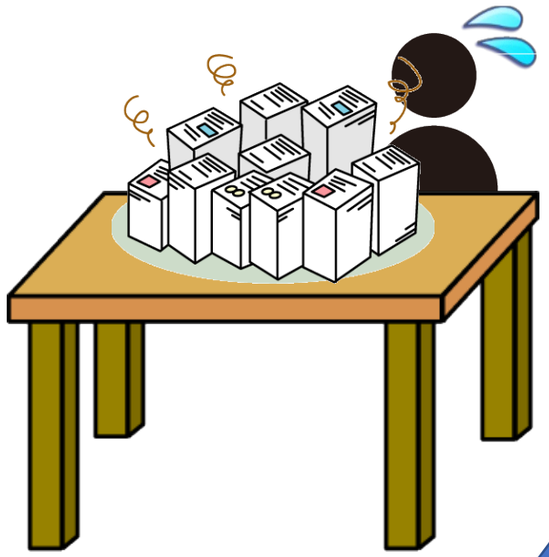
  - 富岳に代表される近代的なスーパーコンピュータの主流

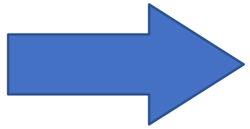
  - **MPI**などを用いた**プロセス並列計算**が必要となる



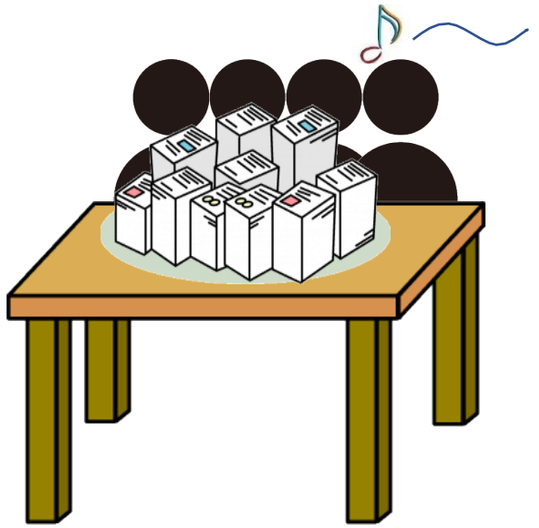
並列計算機のイメージ

非並列処理



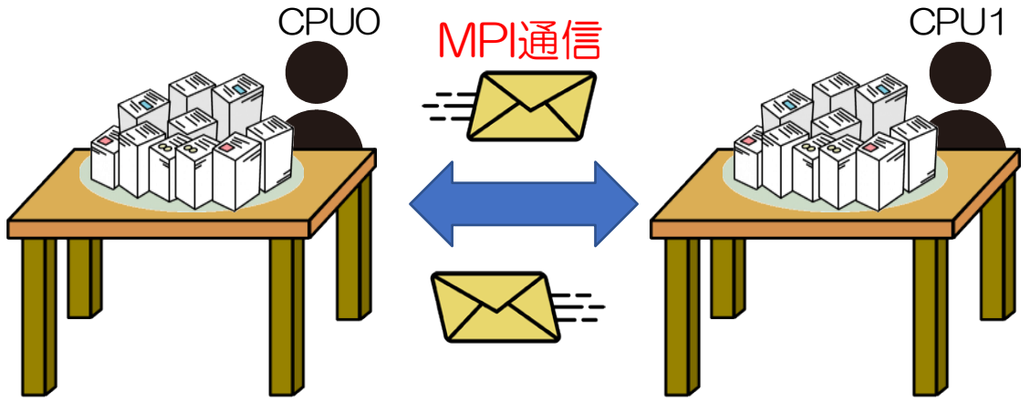


スレッド並列
（共有メモリ）



・各CPU（スレッド）は全てのデータにアクセス可能
・故にお互いが作業の邪魔してしまうこともある

プロセス並列
（分散メモリ）



・各CPU（プロセス）は自身のメモリ空間上のデータのみにアクセス可能
・他のプロセスのデータはMPI通信でアクセス

OpenMPとは

- Open Multi-Processing の略
  - 共有メモリ型計算機用の並列計算API（仕様）

  →**ノード内のスレッド並列**（ノード間は不可）
  - ユーザーが明示的に並列のための指示を与える

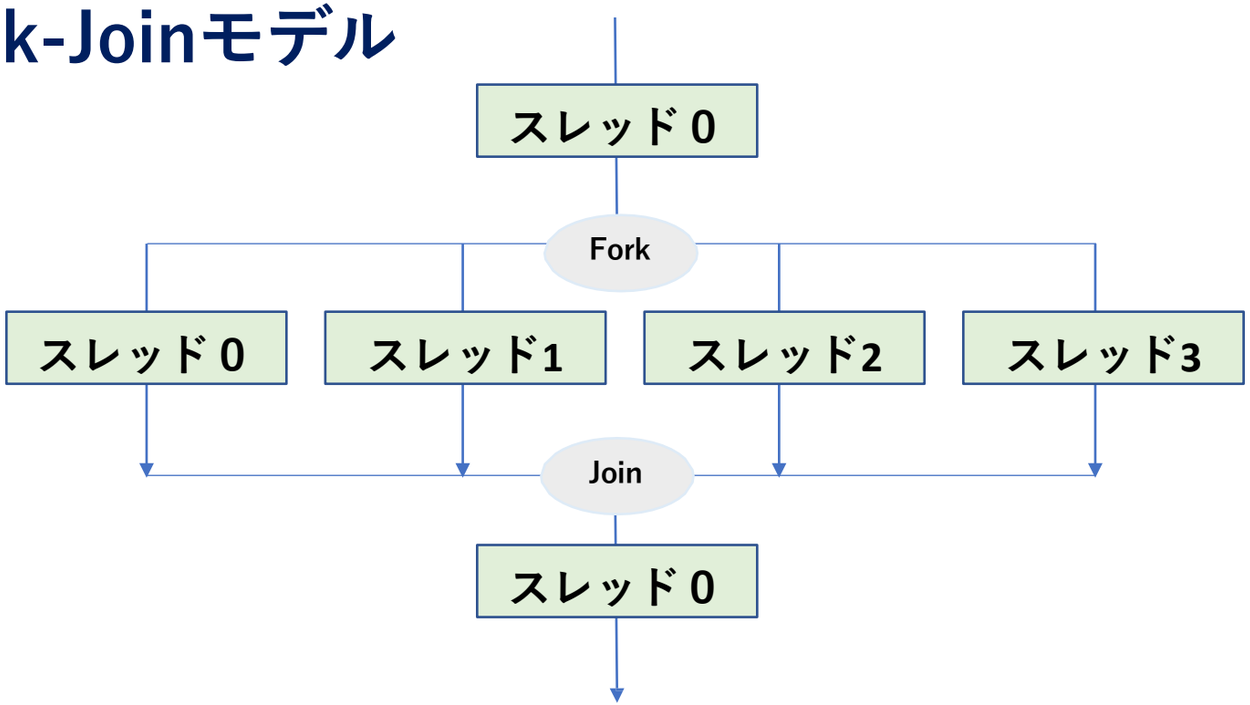
  →コンパイラの自動並列とは異なる
  - 標準化された規格であり、広く使われている
  - 指示行の挿入を行うことで並列化できる

  →既存の非並列プログラムに対し、元のコードの構造を大きく変えることなく並列化できるため、比較的手軽

- ちなみに “OpenMPI” というライブラリが存在するが、こちらはMPI（Message Passing Interface）の実装の１つであり、OpenMPとは全くの別物

OpenMPによるスレッド並列

Fork-Joinモデル



並列リージョン

F

```
...     非並列
!$omp parallel        Fork
...     並列リージョン
!$omp end parallel    Join
...     非並列
```

C

```
...     非並列
#pragma omp parallel        Fork
{
 ...    並列リージョン
}       Join
...     非並列処理
```

OpenMPの基本関数

- **OpenMPモジュール/ヘッダをロード**

[C] **#include <omp.h>**

[F] **use omp_lib**

***OpenMP関連の関数を使用するためのおまじない**

F

```
!$ use omp_lib
integer :: myid, nthreads

nthreads = omp_get_num_threads()
myid = omp_get_thread_num()
```

C

```
#include <omp.h>
int myid, nthreads;

nthreads = omp_get_num_threads();
myid = omp_get_thread_num();
```

OpenMPの基本関数

・最大スレッド数取得

[C][F] nthreads = **omp_get_max_threads()**

・並列リージョン内のスレッド数取得

[C][F] nthreads = **omp_get_num_threads()**

・自スレッド番号取得

[C][F] myid = **omp_get_thread_num()**

F

```
!$ use omp_lib
integer :: myid, nthreads

nthreads = omp_get_num_threads()
myid = omp_get_thread_num()
```

C

```
#include <omp.h>
int myid, nthreads;

nthreads = omp_get_num_threads();
myid = omp_get_thread_num();
```

OpenMPの基本関数

- **時間を測る（倍精度型）**
  - **ある時点からの経過時間を取得できるので、測定したい区間の前後に以下の関数を挟み込んで差を求めます。**

[F][C] time = **omp_get_wtime()**

F

```
!$ use omp_lib
real(8) :: dts, dte
dts = omp_get_wtime()
  ・・・ 処理 ・・・
dte = omp_get_wtime()
print *, dte-dts
```

C

```
#include <omp.h>
double dts;
double dte;
dts = omp_get_wtime();
  ・・・ 処理 ・・・
dte = omp_get_wtime();
```

なお、OpenMPモジュール（ヘッダ）のロードを忘れると、
これらの関数を使用できずコンパイルエラーになる

並列リージョンを指定 (C言語)

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

  #pragma omp single
  {
    …
  }

  #pragma omp for
  for (…)
  …..
  }
}
```

複数スレッドで処理（並列リージョン）

スレッドの起動～終結

[C] #pragma omp parallel {　}

括弧 { } 内が複数スレッドで処理される。

並列リージョンを指定 (Fortran)

```
!$omp parallel
!$omp do
do i = 1, 100
   a(i) = i
enddo
!$omp end do

!$omp single
call output(a)
!$omp end single

!$omp do
do i = 1, 100
.....
enddo
!$omp end do
!$omp end parallel
```

F

スレッドの起動

[F] !$omp parallel

スレッドの終結

[F] !$omp end parallel

複数スレッドで処理（並列リージョン）

演習準備

演習用ファイル（OpenMP）
rokko:/home/guest59/openmp

上記のファイルをホームディレクトリにコピーする

```
mkdir ~/openmp
cd ~/openmp
cp -r /home/guest59/openmp/ ./
```

演習1

演習1-1：omp_ex01_1.c

スレッドを４つ生成し、それぞれのスレッドで ″Hello, World!″ を出力せよ

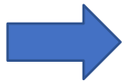
演習1-2：omp_ex01_2.c

スレッド４つで実行、それぞれ自スレッド番号を取得し出力せよ

omp_ex01_1.c

```c
#include <stdio.h>

int main(void){
  printf(″Hello, World!″);
}
```



```c
#include <stdio.h>

int main(void){
  #pragma omp parallel
  {
    printf(″Hello, World!″);
  }
}
```

演習1

演習1-1：omp_ex01_1.f90

スレッドを４つ生成し、それぞれのスレッドで ″Hello, World!″ を出力せよ

演習1-2：omp_ex01_2.f90

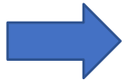
スレッド４つで実行、それぞれ自スレッド番号を取得し出力せよ

omp_ex01_1.f90

```
program omp_ex01_1

print*, ″Hello, World!″

end
```



```
program omp_ex01_1

!$omp parallel
print*, ″Hello, World!″
!$omp end parallel

end
```

環境の構築

- 本実験はインテルコンパイラを使用します

(GNUコンパイラでも可能ですが、今回はインテルコンパイラで統一)

- 適切なモジュールのロード
  - module load intel

    コマンド成功時には何もメッセージが返ってこないので、以下で確認

  - module list

Currently Loaded Modulefiles:
 1) intel/19.1.3

上記のようにモジュールが読み込まれていればOK. 問題があれば、講師やRAに相談してください。

コンパイル

・コンパイルオプションでOpenMPを有効にする

icc −qopenmp −o omp_ex01_1 omp_ex01_1.c

gcc −fopenmp −o omp_ex01_1 omp_ex01_1.c

コンパイルオプションを指定しない場合はOpenMPの指示行はコメントとして認識される。

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

指示行はCの場合は **#pragma omp …** という形式で記述する。

オプションを付けない場合、指示行は無視される

コンパイル

・コンパイルオプションでOpenMPを有効にする

ifort -qopenmp -o omp_ex01_1 omp_ex01_1.f90

gfortran -fopenmp -o omp_ex01_1 omp_ex01_1.f90

コンパイルオプションを指定しない場合はOpenMPの指示行はコメントとして認識される。

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

OpenMPで用いる指示行は、Fortranの場合 **!$OMP** から始まる。行頭に! がある行は通常、コメントとして処理される。

スレッド数の指定

・**シェルの環境変数で与える（推奨）**

export OMP_NUM_THREADS=4　（bashの場合）

setenv OMP_NUM_THREADS 4　（tcshの場合）

・**プログラム内部で設定することも可能**

F
```
!$ use omp_lib
call omp_set_num_threads(4)
```

C
```
#include <omp.h>
omp_set_num_threads(4);
```

ただしスレッド数を変えて実行する時など、毎回コンパイルが必要となってしまうため、今回は環境変数による指定を推奨する。

操作補足

ジョブスクリプト

run.sh

ジョブ投入方法はマシンの環境によって変わるため、実際にスパコンを使う時にはユーザーマニュアルを参考にすること

```
#!/bin/bash
#PBS -q S                          キューを指定
#PBS -l select=1:ncpus=4                リソース確保(4 コア)
#PBS -N omp_JOB                    ジョブ名
#PBS -j oe                         出力ファイル
                                   標準エラー出力と結合

source /etc/profile.d/modules.sh          moduleコマンドのための環境設定
module load intel                  Intelコンパイラ環境の読み込み

export  KMP_AFFINITY=disabled             AFFINITYをdisabledにする
export  OMP_NUM_THREADS=4                 スレッド並列数の設定
cd ${PBS_O_WORKDIR}                作業ディレクトリへ移動
./a.out                    実行
```

Working Sharing構文

〇複数のスレッドで分担して実行する部分を指定
〇 並列リージョン内で記述する
#pragma omp parallel { } の括弧範囲内

指示文の基本形式は

[C] **#pragma omp *xxx***
[F] **!$omp *xxx*** ~ **!$omp end *xxx***

◎for構文, do構文
ループを分割し各スレッドで実行
◎section構文
各セクションを各スレッドで実行
◎single構文
１スレッドのみ実行
◎master構文
マスタースレッドのみ実行

for構文 (C言語)

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

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

forループをスレッドで分割し、並列処理を行う

[F] #pragma omp for

・forループの前に指示行 #pragma omp for を入れる

#pragma omp parallel でスレッドを生成しただけでは、全てのスレッドが全ループを計算してしまう

#pragma omp for を入れることでループ自体が分割され、各スレッドに処理が割り当てられる

do構文 (Fortran)

```
!$omp parallel
!$omp do
do i = 1, 100
   a(i) = i
enddo
!$omp end do

!$omp do
do i = 1, 100
   b(i) = i
enddo
!$omp end do
!$omp end parallel
```

doループをスレッドで分割し、並列処理を行う

[F] !$omp do ～ !$omp end do

・do の直前に指示行 !$omp do を入れる

・enddo の直後に指示行 !$omp end do を入れる

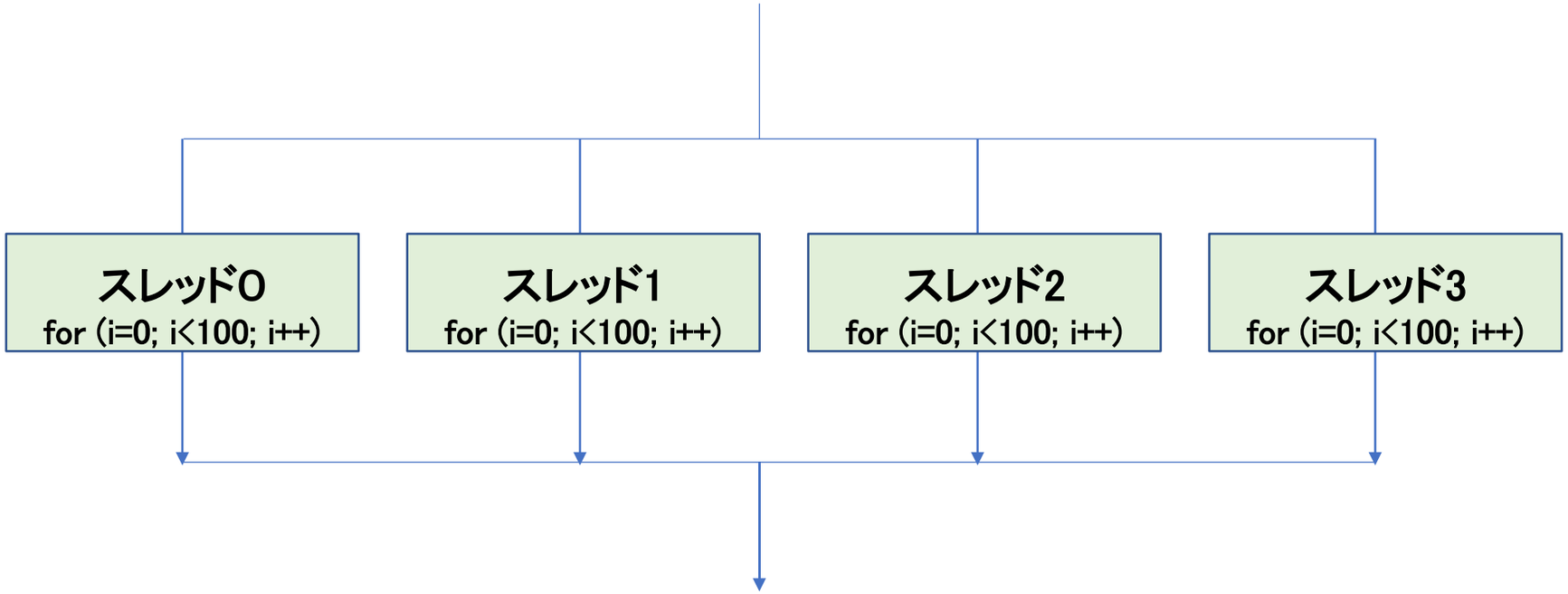
!$omp parallel でスレッドを生成しただけでは、全てのスレッドが全ループを計算してしまう

!$omp do を入れることでループ自体が分割され、各スレッドに処理が割り当てられる

OpenMPによるスレッド並列

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

スレッドを生成しただけでは、全スレッドが全ての処理を行ってしまい負荷分散にならない

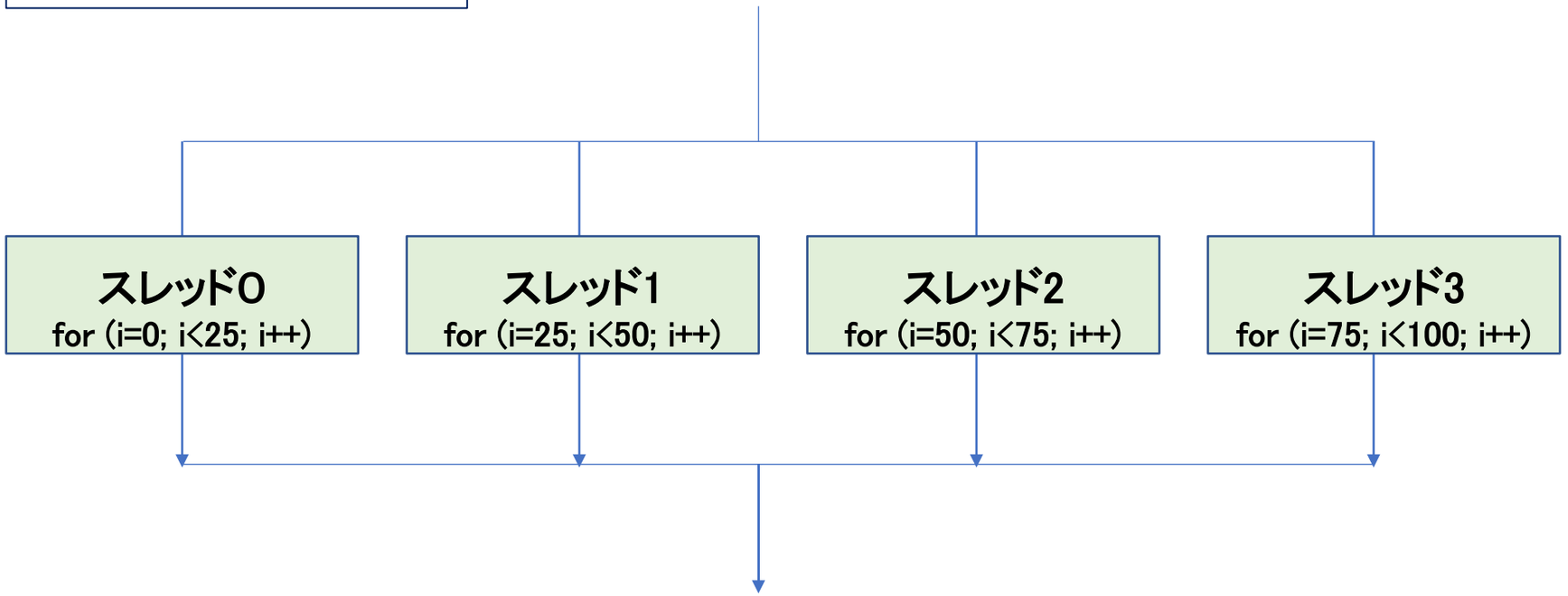


OpenMPによるスレッド並列

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

ワークシェアリング構文を入れることにより、
処理が分割され、正しく並列処理される。

#pragma omp for、!$omp do はループを自動的に
スレッド数で均等に分割する



OpenMPの基本命令(C言語)

スレッド生成とループ並列を１行で記述

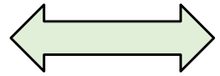
[C言語]

#pragma omp parallel { }

#pragma omp for

→#pragma omp parallel for と書ける

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



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

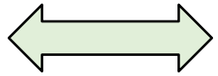
OpenMPの基本命令(Fortran)

スレッド生成とループ並列を１行で記述

[Fortran]

!$omp parallel
!$omp do
　→!$omp parallel do と書ける

```
!$omp parallel
!$omp do
do i = 1, 100
   a(i) = i
enddo
!$omp end do
!$omp end parallel
```



```
!$omp parallel do
do i = 1, 100
   a(i) = i
enddo
!$omp end parallel do
```

演習2

演習2-1：omp_ex02_1.c

サンプルのプログラムはループがスレッドで分割されていない。
指示文を挿入（もしくは修正）し、ループを正しく並列化せよ。

まずは omp_ex02_1.c をそのまま動かして挙動を確認しよう

omp_ex02_1.c

```
#include <stdio.h>
...
  #pragma omp parallel
  {
    for (i=0; i<10; i++) {
      printf("myid=%d, i=%d", omp_get_thread_num(), i);
    }
  }
}
```

演習2

演習2-1：omp_ex02_1.f90

サンプルのプログラムはループがスレッドで分割されていない。
指示文を挿入（もしくは修正）し、ループを正しく並列化せよ。

まずは omp_ex02.f90 をそのまま動かして挙動を確認しよう

omp_ex02_1.f90

```
program omp_ex02_1
...
!$omp parallel
do i=1, 10
  print*, 'myid =', omp_get_thread_num(), 'i =', i
enddo
!$omp end parallel
end
```

演習2

演習2-2：omp_ex02_2.c

演習2-1と同様に指示行を挿入し、ループを並列化せよ。また、計算結果（u）が、スレッド数を1,2,4と変えても変わらないことを確認せよ。

omp_ex02_2.c

```c
#include <stdio.h>
...
  #pragma omp parallel
  {
    for (i=0; i<100; i++) {
      u[i] = sin(2.0*pi*(double)(i+1)/100.0);
//       printf("myid=%d, i=%d", omp_get_thread_num(), i);
    }
  }
```

演習2

演習2-2：omp_ex02_2.f90

演習2-1と同様に指示行を挿入し、ループを並列化せよ。また、計算結果（u）が、スレッド数を1,2,4と変えても変わらないことを確認せよ。

omp_ex02_2.f90

```
program omp_ex02_2
...
!$omp parallel
do i=1, 100
  u(i) = sin(2.0*pi*dble(i)/100.0)
!   print*, 'myid =', omp_get_thread_num(), 'i =', i
enddo
!$omp end parallel
end
```

プライベート変数について

・OpenMPにおいて変数は基本的には共有（shared）であり、どのスレッドからもアクセス可能である。プライベート変数に指定した変数は各スレッドごとに値を保有し、他のスレッドからアクセスされない。

・並列化したループ演算の内部にある一時変数などは、プライベート変数に指定する必要がある。

・例外的に
[C]#pragma omp for
[F] !$omp parallel do
の直後のループ変数はプライベート変数になる

プライベート変数について

プライベート変数を指定

[C] #pragma omp parallel for private(a, b, ...)
[C] #pragma omp for private(a, b, ...)

```c
#pragma omp parallel
{
  #pragma omp for private(j, k)
  for (i=0; i<nx; i++) {
    for (j=0; j<ny; j++) {
      for (k=0; k<nz; k++) {
        f[i][j][k] = (double)(i * j * k);
      }
    }
  }
}
```

ループ変数の扱いに関して

並列化したループ変数は自動的にprivate変数になる。しかし多重ループの場合、内側のループに関しては共有変数のままである。

左の例の場合、i は自動的にprivateになるため必要ないが、j, k についてはprivate宣言が必要となる。

プライベート変数について

プライベート変数を指定

[F] !$omp parallel do private(a, b, ...)
[F] !$omp do private(a, b, ...)

```
!$omp parallel
!$omp do private(j, k)
do i = 1, nx
  do j = 1, ny
    do k = 1, nz
      f(k, j, i) = dble(i * j * k)
    enddo
  enddo
enddo
!$omp end do
!$omp end parallel
```

F

ループ変数の扱いに関して

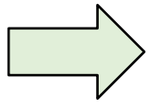
並列化したループ変数は自動的にprivate変数になる。しかし多重ループの場合、内側のループに関しては共有変数のままである。

左の例の場合、i は自動的にprivateになるため必要ないが、j, k についてはprivate宣言が必要となる。

プライベート変数について

起こりがちなミス

```
#pragma omp for
for (i=0; i<100; i++) {
  tmp = myfunc(i);
  a[i] = tmp;
}
```



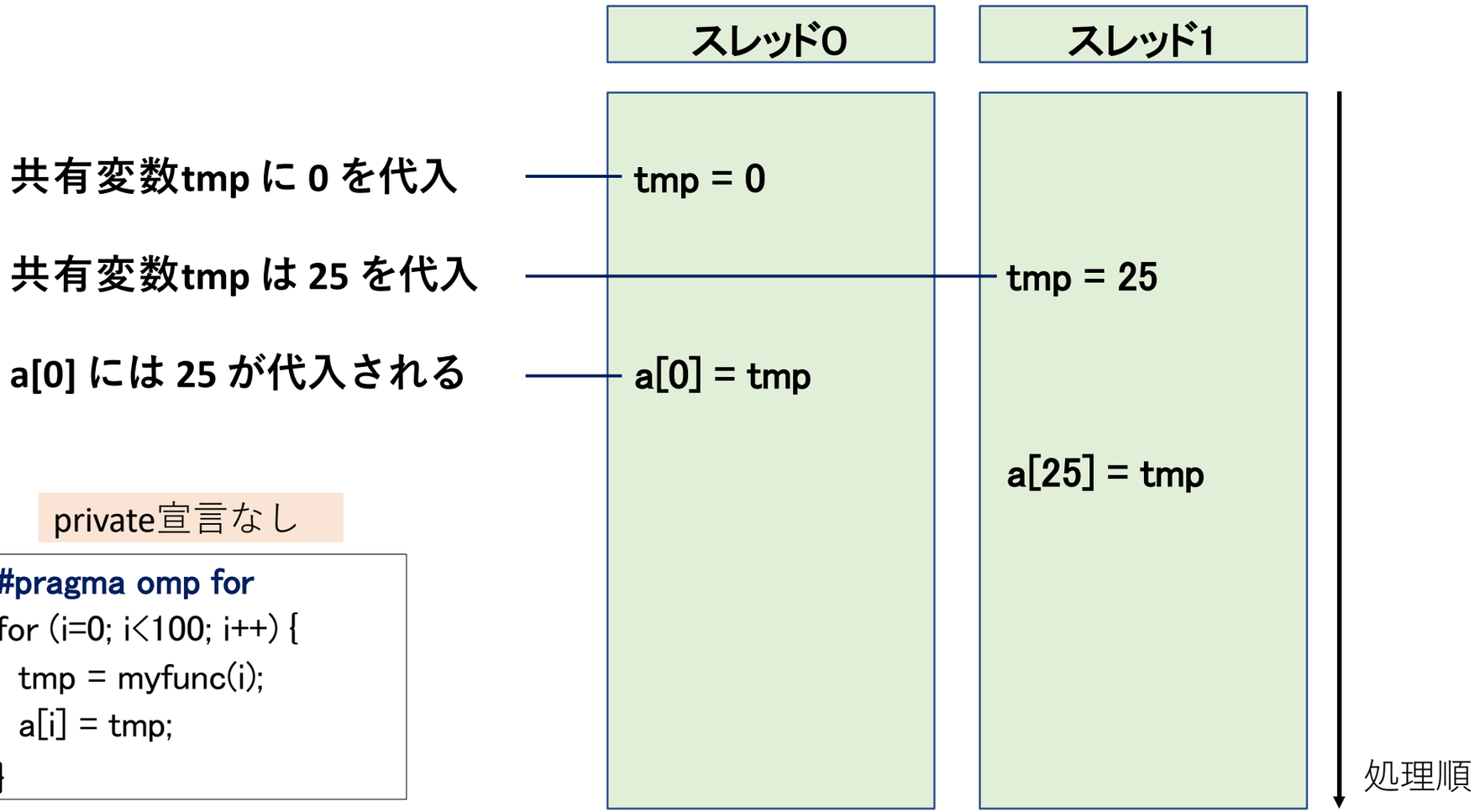
```
#pragma omp for private(tmp)
for (i=0; i<100; i++) {
  tmp = myfunc(i);
  a[i] = tmp;
}
```

tmpを上書きしてしまい、
正しい結果にならない

private宣言を入れる

並列化したループ内で値を設定・更新する場合は要注意
→privateにすべきではないか確認する必要あり

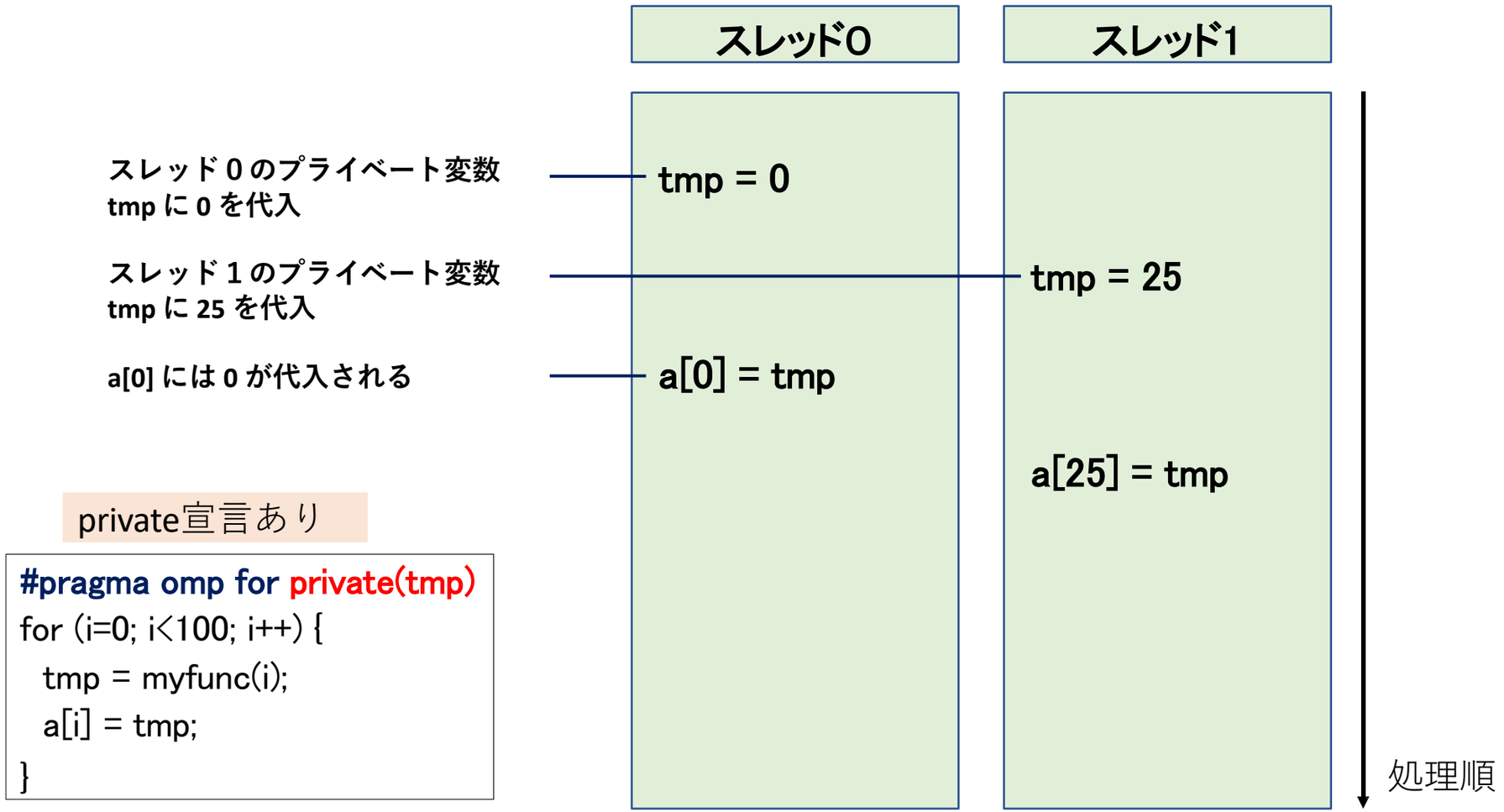
プライベート変数について



private宣言なし

```
#pragma omp for
for (i=0; i<100; i++) {
  tmp = myfunc(i);
  a[i] = tmp;
}
```

プライベート変数について

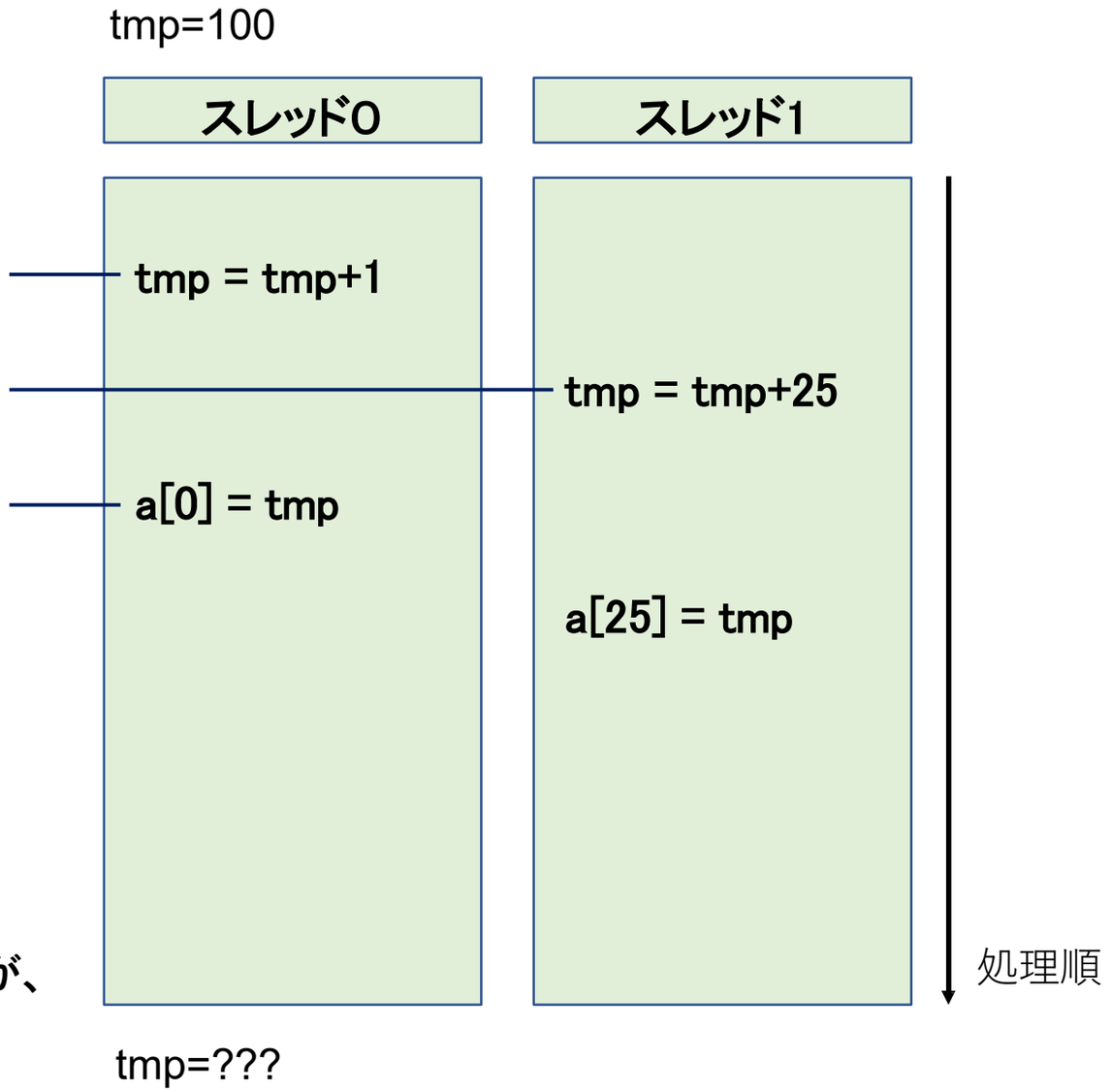


private宣言あり

```
#pragma omp for private(tmp)
for (i=0; i<100; i++) {
  tmp = myfunc(i);
  a[i] = tmp;
}
```

OpenMP外で定義されたプライベート変数

- firstprivate/lastprivate



private変数はあくまでも
各スレッドのfork時に生成された変数であり、
初期値は不定。**マスタースレッドのみ**
OpenMP外側で定義されたものと同等

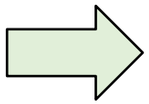
全てのスレッドでfork時にOpenMP外側での値とする場合はfirstprivateの属性が必要

Join時、マスタースレッドの値が保持されるが、シーケンシャルな動作と同等にしたい場合はlastprivate属性の指定が必要

多重ループに関して

良くない例

```
for (i=0; i<nx; i++) {
  for (j=0; j<ny; j++) {
    #pragma omp parallel for private(k)
    for (k=0; k<nz; k++) {
      f[i][j][k] = (double)(i * j * k);
    }
  }
}
```



改善案

```
#pragma omp parallel private(i, j, k)
{
  for (i=0; i<nx; i++) {
    for (j=0; j<ny; j++) {
      #pragma omp for
      for (k=0; k<nz; k++) {
        f[i][j][k] = (double)(i * j * k);
      }
    }
  }
}
```

OpenMPを用いた並列化では、内側ループ、外側ループのどちらを並列化しても良い。ただし、内側ループを並列化する場合、毎回fork-joinしてしまうとスレッド生成回数がものすごいことになる。
（上記の例では１つの３次元ループで nx * ny 回）

なお、並列化するループを変えたり、ループの計算順序を変更する可能性があるため、private宣言にはループ変数も書いた方が無難。

共有変数について

共有（shared）変数を指定

[C] #pragma omp parallel shared(a, b, ...)
[C] #pragma omp for shared(a, b, ...)

[F] !$omp parallel shared(a, b, ...)
[F] !$omp do shared(a, b, ...)

・指定しなければ基本的に共有変数であるため、省略可能。

スレッドの同期

nowait を明示しない限り、ワークシェアリング構文の終わりに自動的に同期処理が発生

スレッドの同期待ちをしない

[C] #pragma omp for nowait

[F] !$omp do ~ !$omp end do nowait

スレッドの同期をとる

[C] #pragma omp barrier

[F] !$omp barrier

Section構文

スレッドごとに処理を分岐させる

[C] #pragma omp sections {#pragma omp section, ...}
[F] !$omp sections, !$omp section, ... , !$omp end sections

F

```
!$omp parallel
  !$omp sections
    !$omp section
      処理A
    !$omp section
      処理B
  !$omp end sections
!$omp end parallel
```

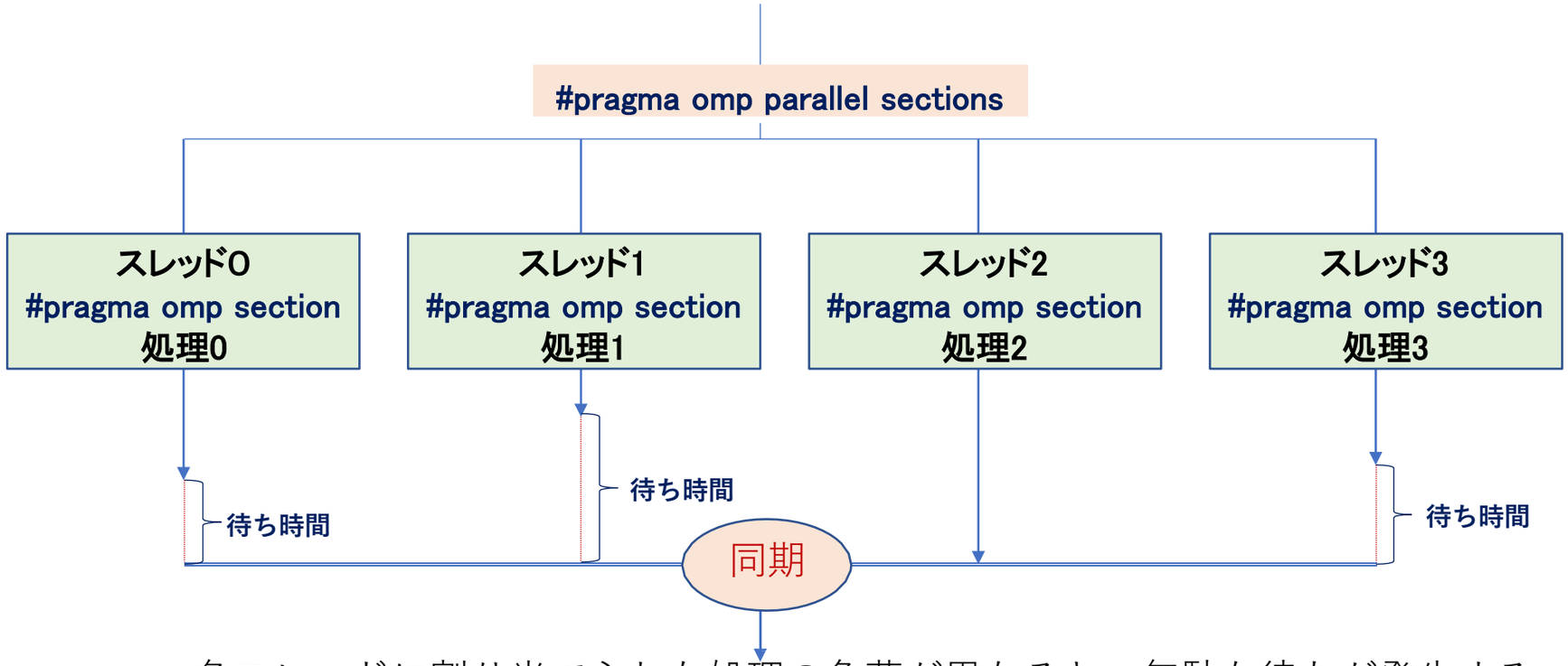
C

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
      処理A
    #pragma omp section
      処理B
  }
}
```

処理の分岐

Section構文

スレッドごとに処理を分岐させる



・各スレッドに割り当てられた処理の負荷が異なると、無駄な待ちが発生する
・ロードバランスに注意

演習3

演習3-1：omp_ex03.c

演習2で行ったループ演算の並列化は、SECTION構文を用いて分割することも可能である。未完成のサンプルコード（omp_ex03.c）を完成させ、同じ処理を行っていることを確認せよ。なおスレッド数は４にすること。

omp_ex02_2.c

```
#include <stdio.h>
...
for (i=0; i<100; i++) {
  u[i] = sin(2.0*pi*･･･);
}
```

omp_ex03.c

```
#include <stdio.h>

for (i=0; i<25; i++) {
  u[i] = sin(2.0*pi*･･･);
}

for (i=25; i<50; i++) {
  u[i] = sin(2.0*pi*･･･);
}
```

演習3

演習3-1：omp_ex03.f90

演習2で行ったループ演算の並列化は、SECTION構文を用いて分割することも可能である。未完成のサンプルコード（omp_ex03.f90）を完成させ、同じ処理を行っていることを確認せよ。なおスレッド数は４にすること。

omp_ex02.f90

```
program ex02
...
do i=1, 100
  u(i) = sin(2.0*pi*...)
enddo

end
```

omp_ex03.f90

```
program ex03
...
do i=1,25
  u(i) = sin(2.0*pi*...)
enddo

do i=26,50
  u(i) = sin(2.0*pi*...)
enddo
```

１スレッドのみで処理

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

  #pragma omp single
  {
    output(a);
  }

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

[C] #pragma omp single { }

一般的に、スレッドの立ち上げ回数は極力減らすほうがオーバーヘッドが少なくなるため良いとされる。

逐次処理やデータの出力のような処理が入る場合

#pragma omp single { }
とすると１スレッドのみで処理される

１スレッドのみで処理

```
!$omp parallel
!$omp do
do i = 1, 100
   a(i) = i
enddo
!$omp end do

!$omp single
  call output(a)
!$omp end single

!$omp do
do i = 1, 100
.....
enddo
!$omp end do
!$omp end parallel
```

[F] !$omp single ~
!$omp end single

一般的に、スレッドの立ち上げ回数は極力減らすほうがオーバーヘッドが少なくなるため良いとされる。

逐次処理やデータの出力のような処理が入る場合、

!$omp single
とすると１スレッドのみで処理される

１スレッドのみで処理

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

  #pragma omp master
  {
    output(a);
  }

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

[C] #pragma omp master { }

一般的に、スレッドの立ち上げ回数は極力減らすほうがオーバーヘッドが少なくなるため良いとされる。

逐次処理やデータの出力のような処理が入る場合

#pragma omp master { }
とすると**マスタースレッド**のみで処理される。
終了時にスレッド間同期は行われない。

１スレッドのみで処理

```
!$omp parallel
!$omp do
do i = 1, 100
   a(i) = i
enddo
!$omp end do

!$omp master
  call output(a)
!$omp end master

!$omp do
do i = 1, 100
.....
enddo
!$omp end do
!$omp end parallel
```

[F] !$omp master ~
!$omp end master

一般的に、スレッドの立ち上げ回数は極力減らすほうがオーバーヘッドが少なくなるため良いとされる。

逐次処理やデータの出力のような処理が入る場合、

!$omp master
とすると**マスタースレッド**のみで処理される
終了時にスレッド間同期は行われない。

排他処理

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

  #pragma omp critical
  {
    a[0]=func(a,b,100);
  }

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

[C] #pragma omp critical { }

共有メモリではshared変数の書き込みと読み込みのタイミングによる意図しない動作がおこります(競合状態)。

ブロック内の処理を行うスレッドが１つだけになるよう排他処理する場合

#pragma omp critical { }
とすると排他処理される

排他処理

```
!$omp parallel
!$omp do
do i = 1, 100
    a(i) = i
enddo
!$omp end do

!$omp critical
    a(1)=func(a,b,100)
!$omp end critical

!$omp do
do i = 1, 100
    .....
enddo
!$omp end do
!$omp end parallel
```

[F] !$omp critical ~
!$omp end critical

共有メモリではshared変数の書き込みと読み込みのタイミングによる意図しない動作がおこります(競合状態)。

ブロック内の処理を行うスレッドが1つだけになるよう排他処理する場合

!$omp critical
とすると排他処理される

リダクション変数を指定

reduction(演算子:変数)

・並列計算時はそれぞれのスレッドで別々の値を持ち、並列リージョン終了時に各スレッドの値が足し合わされる（総和）shared変数

・総和の他、積などを求めることも可能

F

```
integer :: i, sum
sum = 0
!$omp parallel do reduction(+:sum)
do i=1, 10000
  sum = sum + 1
enddo
!$omp end parallel do
```

C

```
int i, sum;
sum = 0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i<10000; i++) {
    sum = sum + 1;
}
```

演習4

演習4-1：omp_ex04.f90 / omp_ex04.c

1から100までを足し合わせるプログラムをOpenMPで並列化せよ。

reduction変数の指定を忘れると正しく動かないため注意。

omp_ex04.f90

F

```
program ex04
...

do i=1,100
  a = a + i
enddo

print*, a
```

omp_ex04.c

C

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

for (i=1; i<=100; i++) {
  a = a + i;
}

printf("%d¥n", a);
```

並列化できないプログラム

・どのループを並列化するか（可能か）という判断は全てプログラマが行う

並列化できないプログラム

```
a[0] = 0
for (i=1; i<=100; i++) {
    a[i] = a[i−1] + 1
}
```

このプログラムはi 番目の計算を行うためにはi-1 番目の計算結果が必要であり（**データの依存関係**）、並列化できないプログラムである。
しかしOpenMPの指示行を入れれば、コンピュータは無理やりこのプログラムを並列化し、間違った計算を行う。

演習5

演習5-1：

熱伝導問題のプログラム（laplace.c / laplace.f90）を OpenMPでスレッド並列化して計算せよ（ただし並列化可能かも調べよ）

基礎方程式

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

↓ 中心差分（2次精度）

$$T_{i,j} = \frac{1}{4}\left(T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1}\right)$$

演習5

演習5-1：

熱伝導問題のプログラム（laplace.c / laplace.f90）を OpenMPでスレッド並列化して計算せよ（ただし正しく並列化されているかも調べよ）

SOR反復法

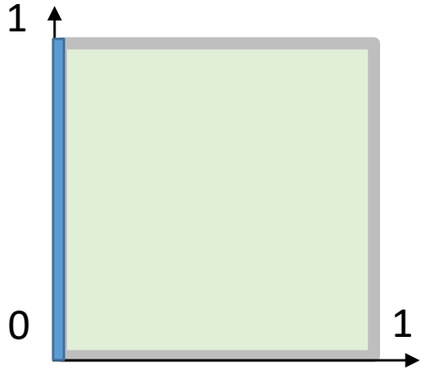
$$\rho_{i,j} = \frac{1}{4}\left(T_{i-1,j}^{k+1} + T_{i+1,j}^{k} + T_{i,j-1}^{k+1} + T_{i,j+1}^{k}\right)$$

$$T_{i,j}^{k+1} = T_{i,j}^{k} + \omega\left(\rho_{i,j} - T_{i,j}^{k}\right)$$

境界条件

$$T_{i,0} = T_{N+1,j} = T_{i,N+1} = 0$$

$$T_{0,j} = \sin(\pi * i/n)$$



SOR法の補足

・基本は、ガウス・ザイデル法（$\omega = 1$ のとき同じ式になる）

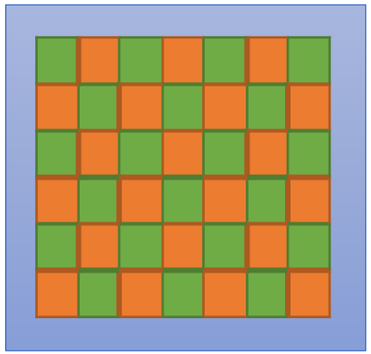
$$\rho_{i,j} = \frac{1}{4}\left(T_{i-1,j}^{k+1} + T_{i+1,j}^{k} + T_{i,j-1}^{k+1} + T_{i,j+1}^{k}\right)$$

$$T_{i,j}^{k+1} = T_{i,j}^{k} + \omega\left(\rho_{i,j} - T_{i,j}^{k}\right)$$

・注意せずに単純にOpenMPでパラレルforなどの指定をすると「競合状態」に陥り正しく計算ができないことや、結果が実行毎に異なる場合があります。

・一般に、オーダリングを用いて並列化（ハイパープレーン、２色など）

・２色オーダリングの場合：



1．■ ブロックを先に計算

$$\rho_{i,j} = \frac{1}{4}\left(T_{i-1,j}^{k} + T_{i+1,j}^{k} + T_{i,j-1}^{k} + T_{i,j+1}^{k}\right)$$

2．次に、■ ブロックを計算

$$\rho_{i,j} = \frac{1}{4}\left(T_{i-1,j}^{k+1} + T_{i+1,j}^{k+1} + T_{i,j-1}^{k+1} + T_{i,j+1}^{k+1}\right)$$

演習5

演習5-1：

熱伝導問題のプログラム（ laplace.c / laplace.f90）

をOpenMPでスレッド並列化して計算せよ

補足（laplace.c / laplace.f90）

n：縦、横それぞれのグリッド数

ITMAX : 最大反復回数

eps：反復ベクトルの差のノルムの閾値

（どこまで値を収束させるか）

演習5

gnuplotを使って結果をプロット

username@rokko:~/khpc2019/omp> gnuplot

gnuplot> set pm3d

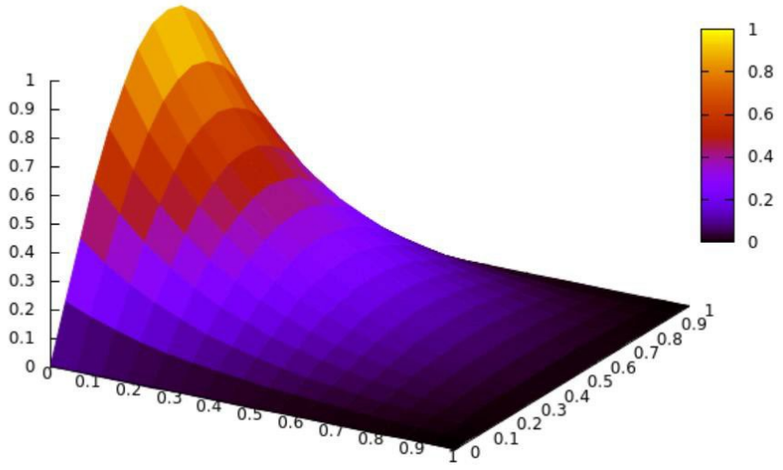
gnuplot> set ticslevel 0

gnuplot> ~~set cbrange[0:1]~~ カラーバーの値の範囲指定

gnuplot> ~~set palette defined (0 “blue”, 1 ″red″)~~ カラーバーの色設定

gnuplot> splot “data.d″ with pm3d

カラーバーの値の範囲等は
計算のパラメータによって
適宜変更すること



演習5

演習5-2：

スレッド並列化した熱伝導問題のプログラムを、並列数を
1, 2, 4, 8, 16, … と変えて実行し、計算時間を計測せよ

並列数はジョブスクリプトにて（**8スレッドの場合**）

CPU確保：　　**#PBS -l select=1:ncpus=8**

スレッド数：　**export OMP_NUM_THREADS=8**

の２か所を書き換える

演習5

演習5-2：

時間計測（簡易版）：
　ジョブスクリプトの実行で time を使用

time dplace ./a.out

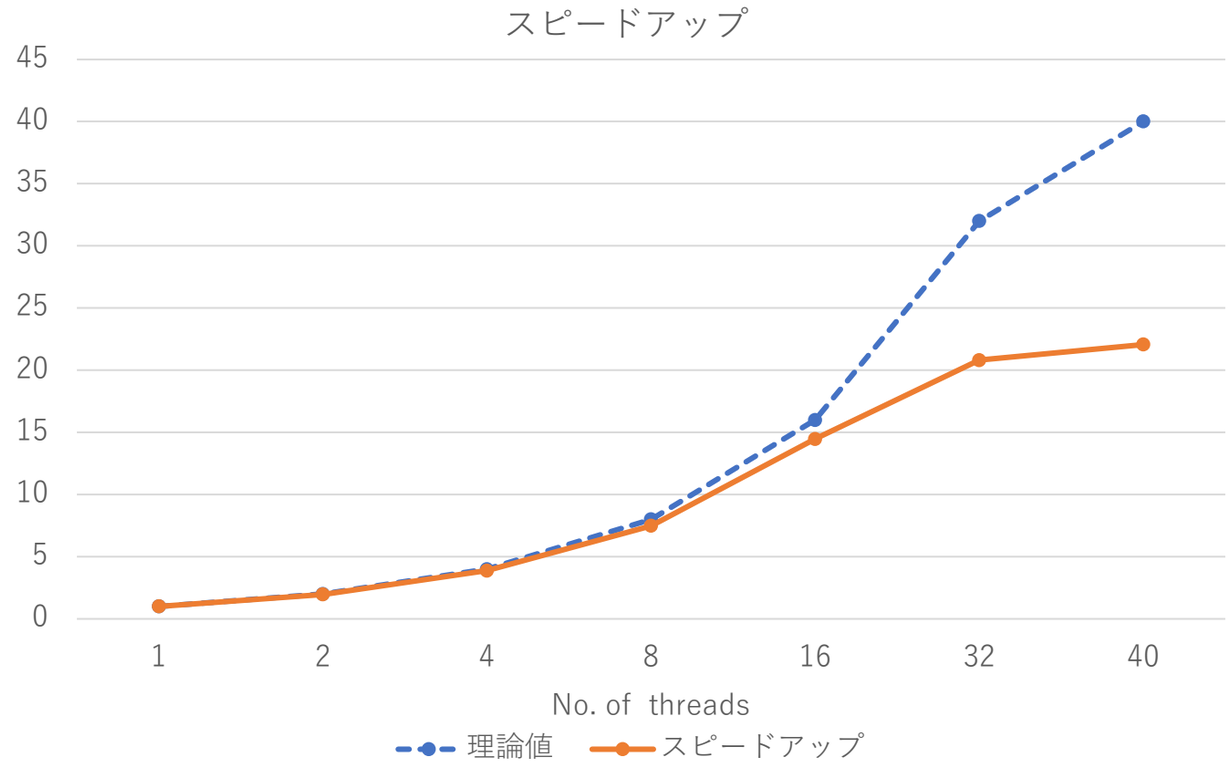
時間計測（関数）：
　omp_get_wtime 関数を使用

計算開始時に
　dts = **omp_get_wtime()**

計算終了時に
　dte = **omp_get_wtime()**

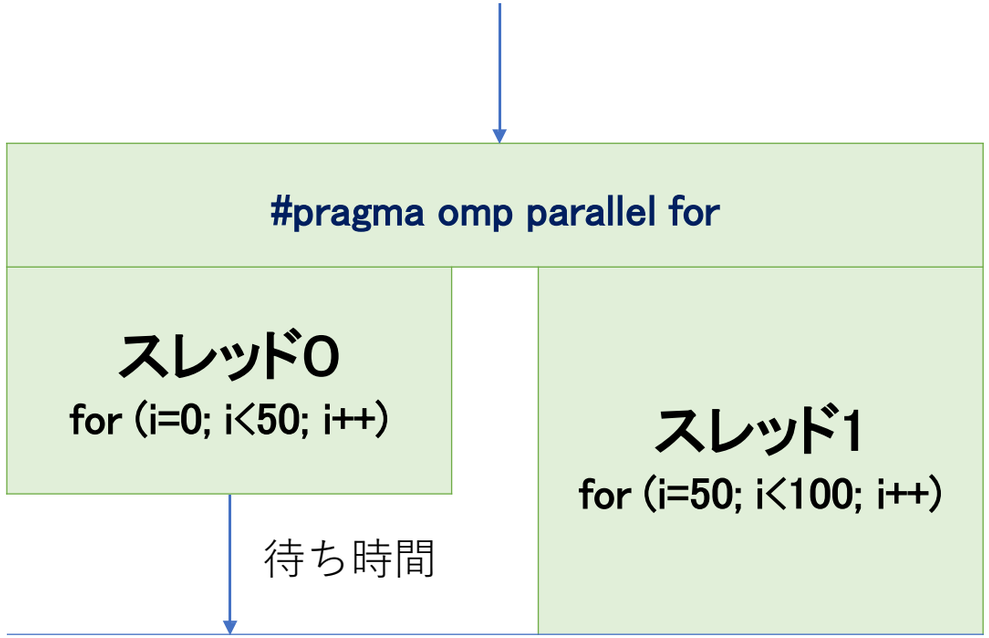
とし、 dte - dts を出力する

並列数と計算性能



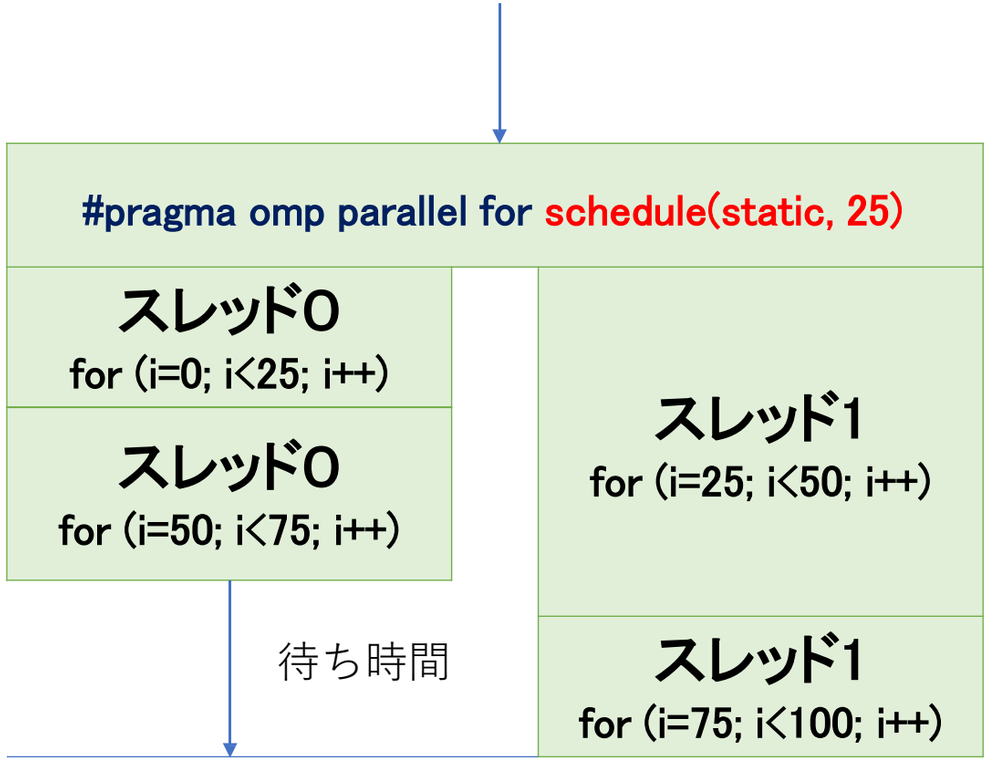
並列数	1	2	4	8	16	32	40
実測値(秒)	2.56E+00	1.30E+00	6.61E-01	3.42E-01	1.77E-01	1.23E-01	1.16E-01
スピードアップ値	1.00E+00	1.97E+00	3.87E+00	7.49E+00	1.45E+01	2.08E+01	2.21E+01

スケジューリング



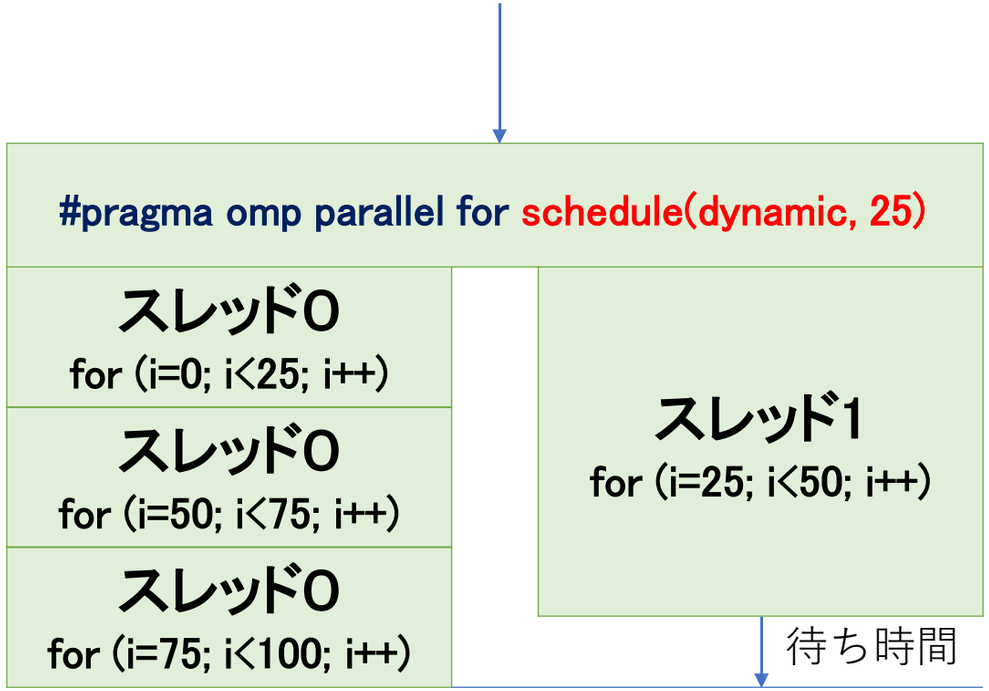
・各スレッドに割り当てられた処理の負荷が異なると、無駄な待ちが発生する
・ロードバランスに注意

スケジューリング



・各スレッドに割り当てられた処理の負荷が異なると、無駄な待ちが発生する
・ロードバランスに注意

スケジューリング



・各スレッドに割り当てられた処理の負荷が異なると、無駄な待ちが発生する
・ロードバランスに注意

スケジューリング

- 例：（上三角行列）＊（ベクトル）
- $y = A * x, \ \ (a_{ij} = 0, i > j)$

```
for (i=0; i<n; i++) {
    for (j=i; j<n; j++) {
        y[i] += a[i][j]*x[j];
}}
```

- チャンクサイズが性能に大きく影響する。
- 負荷バランスとシステムのオーバーヘッドのトレードオフ。
- チューニングコストが増加する。

並列化率に関して

CPUをN個使って並列計算した時、計算速度がN倍になるのが理想だが・・・

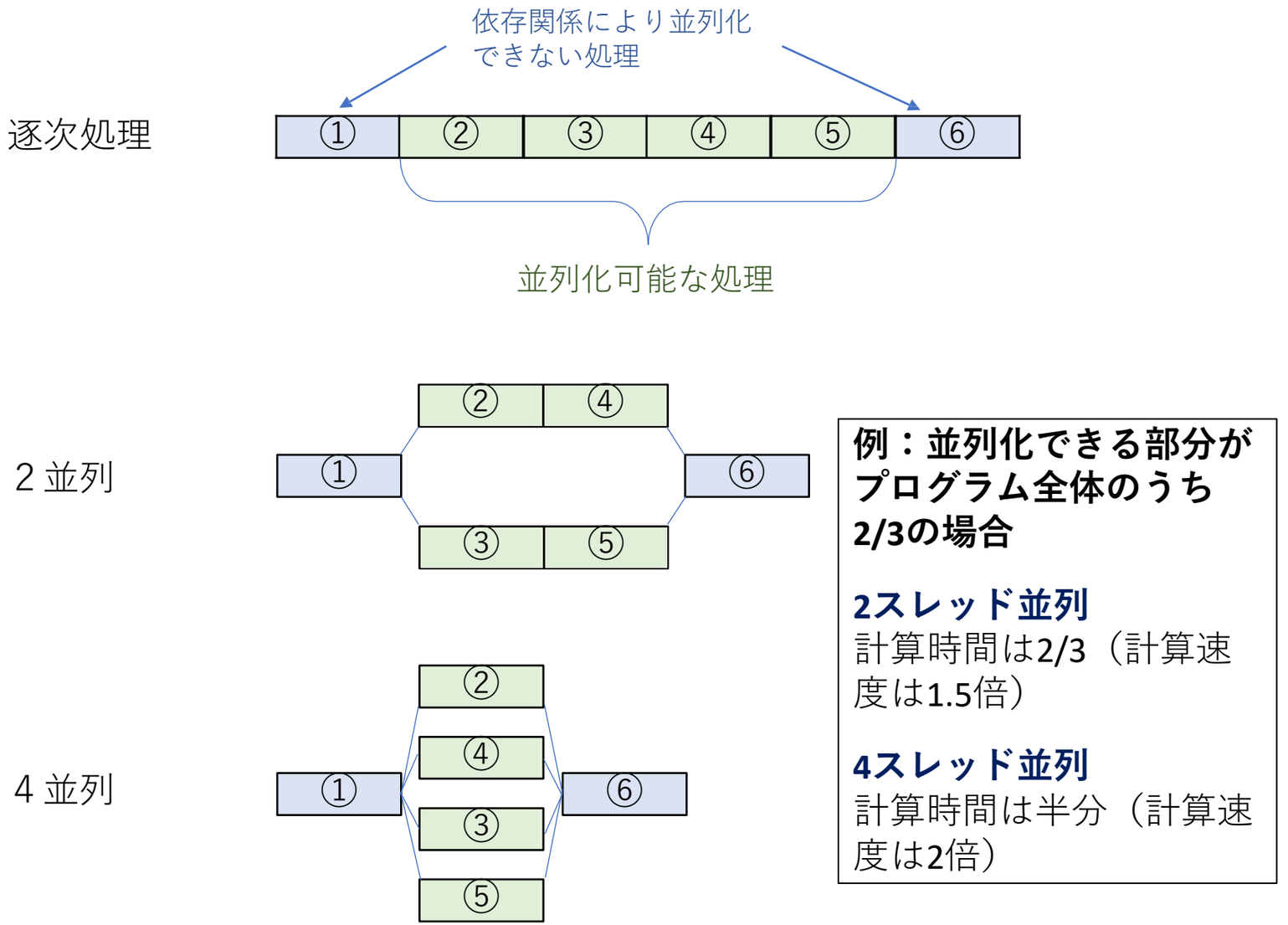
並列化率の問題

プログラム内に並列化できない処理が含まれていると、その部分が並列計算におけるボトルネックになる（アムダールの法則）

並列計算を行うためのコスト

並列化を行うことで、逐次実行には不要 だった処理が増えることがある（スレッド起動、MPIのプロセス間通信等）

並列化率に関して



例：並列化できる部分がプログラム全体のうち2/3の場合

2スレッド並列
計算時間は2/3（計算速度は1.5倍）

4スレッド並列
計算時間は半分（計算速度は2倍）

並列化率に関して

アムダールの法則

プログラムの並列化できる割合をP とし、プロセッサ数をn とすると、並列計算した時の性能向上率は

$$\frac{1}{(1-P)+\frac{P}{n}}$$

で与えられる。

これを**アムダールの法則**と呼ぶ。

並列化率に関して

- **アムダールの法則**

- 例えば、プログラム全体の９割は並列化できるが１割は逐次処理が残ってしまうような場合、どれだけプロセッサを投入しても計算速度は１０倍以上にはならない。

- 富岳のような大型計算機を用いる上では、如何にして並列化率を上げるかが重要である。