OpenMPによるスレッド並列計算

寺尾　剛史
（理化学研究所 計算科学研究センター）

27th Supercomputing Contest 2021

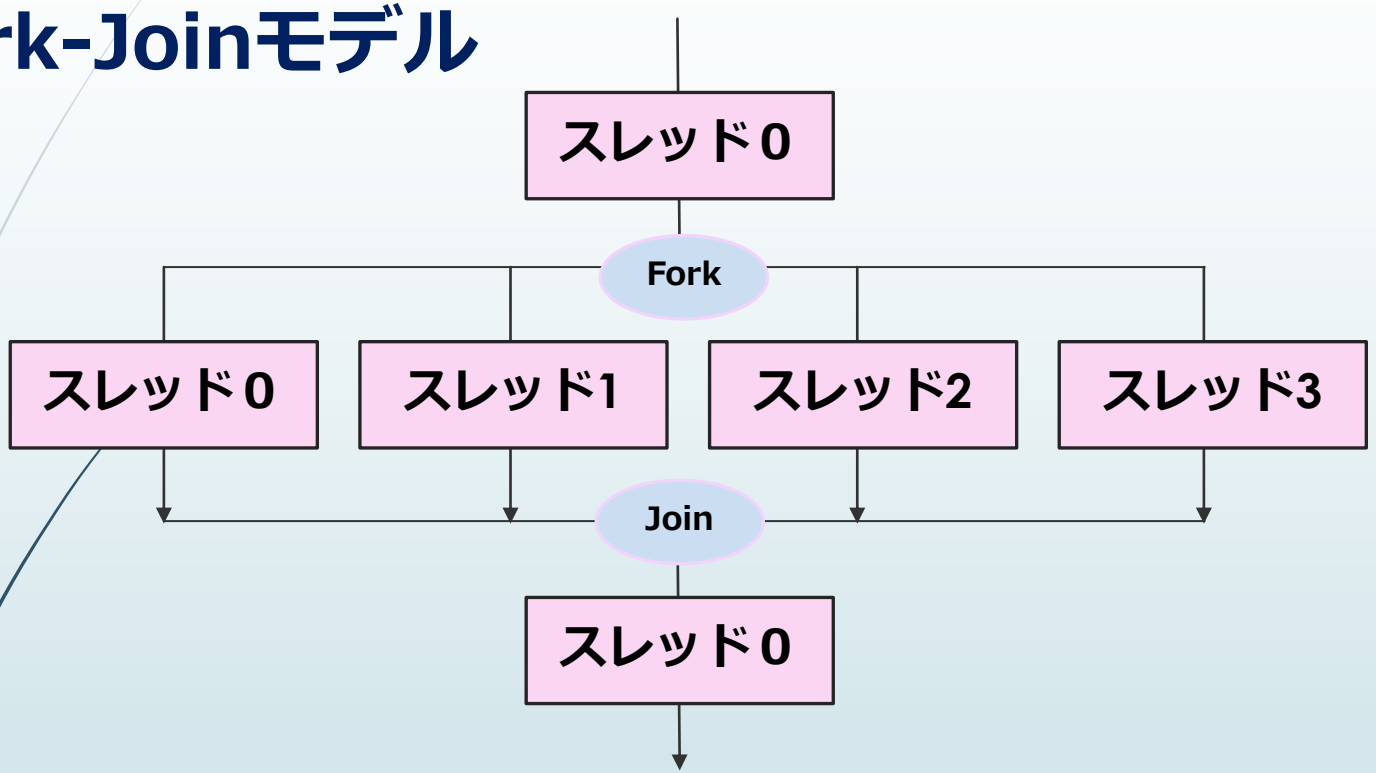
2021年9月23日

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

OpenMPによるスレッド並列

Fork-Joinモデル



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（Integer）

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

自スレッド番号取得（Integer）

[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）

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

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

F

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

C

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

コンパイル

・例）コンパイルオプションでOpenMPを有効にする

(trad) fccpx −Kopenmp −Nlibomp omp_ex.c

(clang) fccpx −Nclang −Kopenmp −Nlibomp omp_ex.c

詳しくは、以下を参照：
https://www.fugaku.rccs.riken.jp/doc_root/ja/user_guides/lang_1.07/

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。

C

```
#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

・例）コンパイルオプションでOpenMPを有効にする

frtpx -Kopenmp -Nlibomp omp_ex.f90

詳しくは、以下を参照：
https://www.fugaku.rccs.riken.jp/doc_root/ja/user_guides/lang_1.07/

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。

Working Sharing構文

○ 複数のスレッドで分担して実行する部分を指定
○ 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

```
#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
  }
}
```

C

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

```
!$omp parallel
!$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

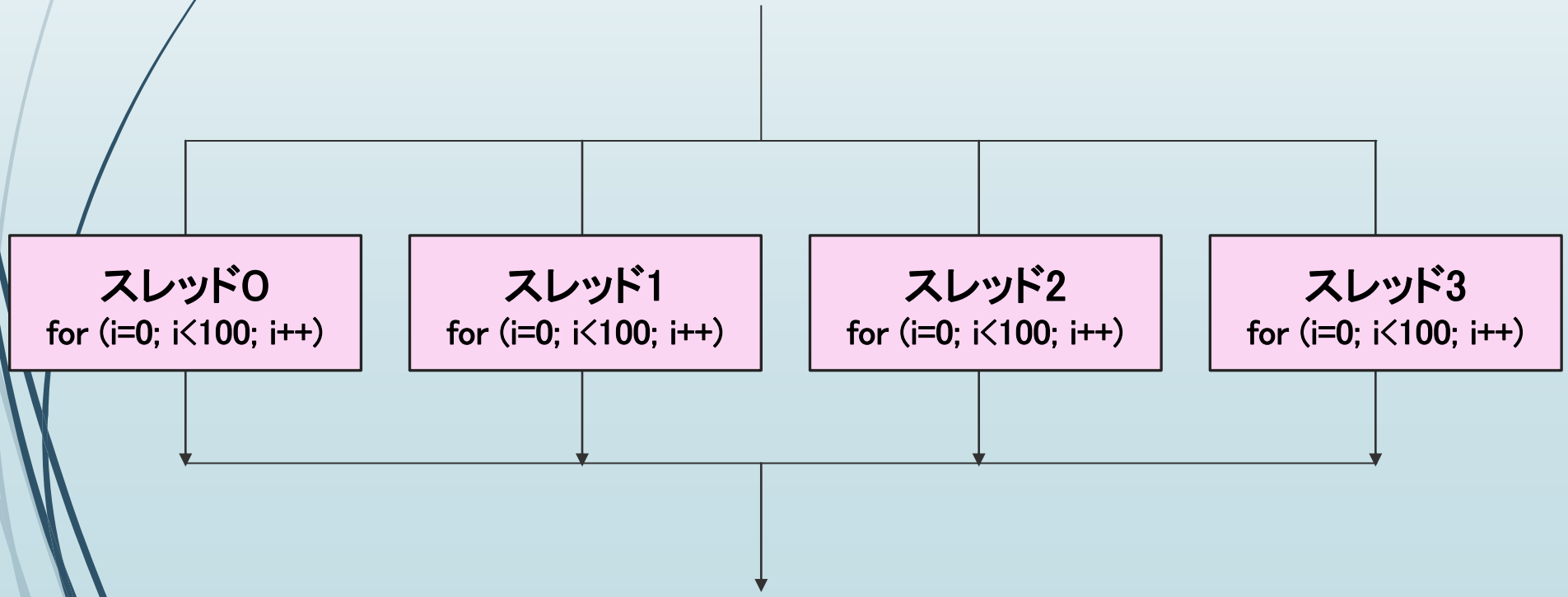
!$omp parallel でスレッドを生成しただけでは、全てのスレッドが全ループを計算してしまう

!$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

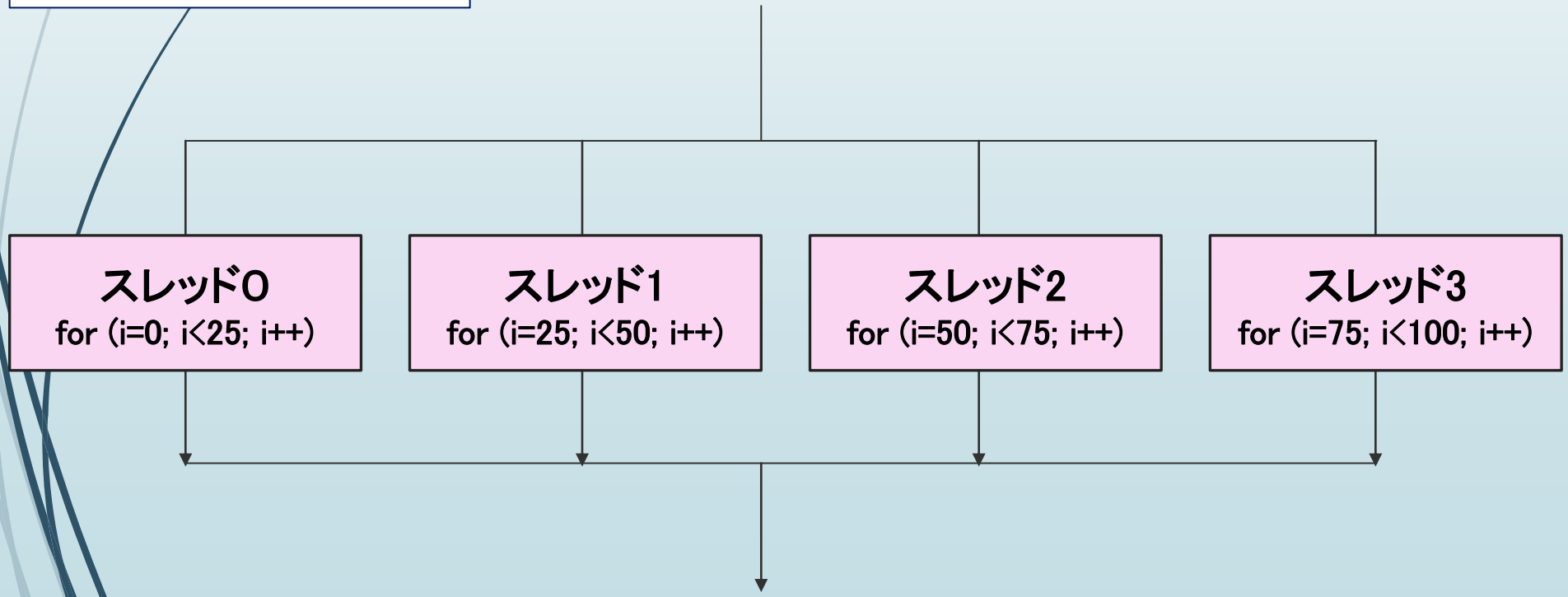


OpenMPによるスレッド並列

```
#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

スレッド生成とループ並列を１行で記述

[C言語]

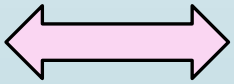
#pragma omp parallel { }

#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;
  }
}
```

C



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

C

OpenMPの基本命令

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
```

F

⟺

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

F

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, ...)

```
#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 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
```

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。

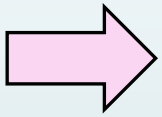
左の例の場合、i は自動的にprivateになるため必要ないが、j, k についてはprivate宣言が必要となる。

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;
}
```

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

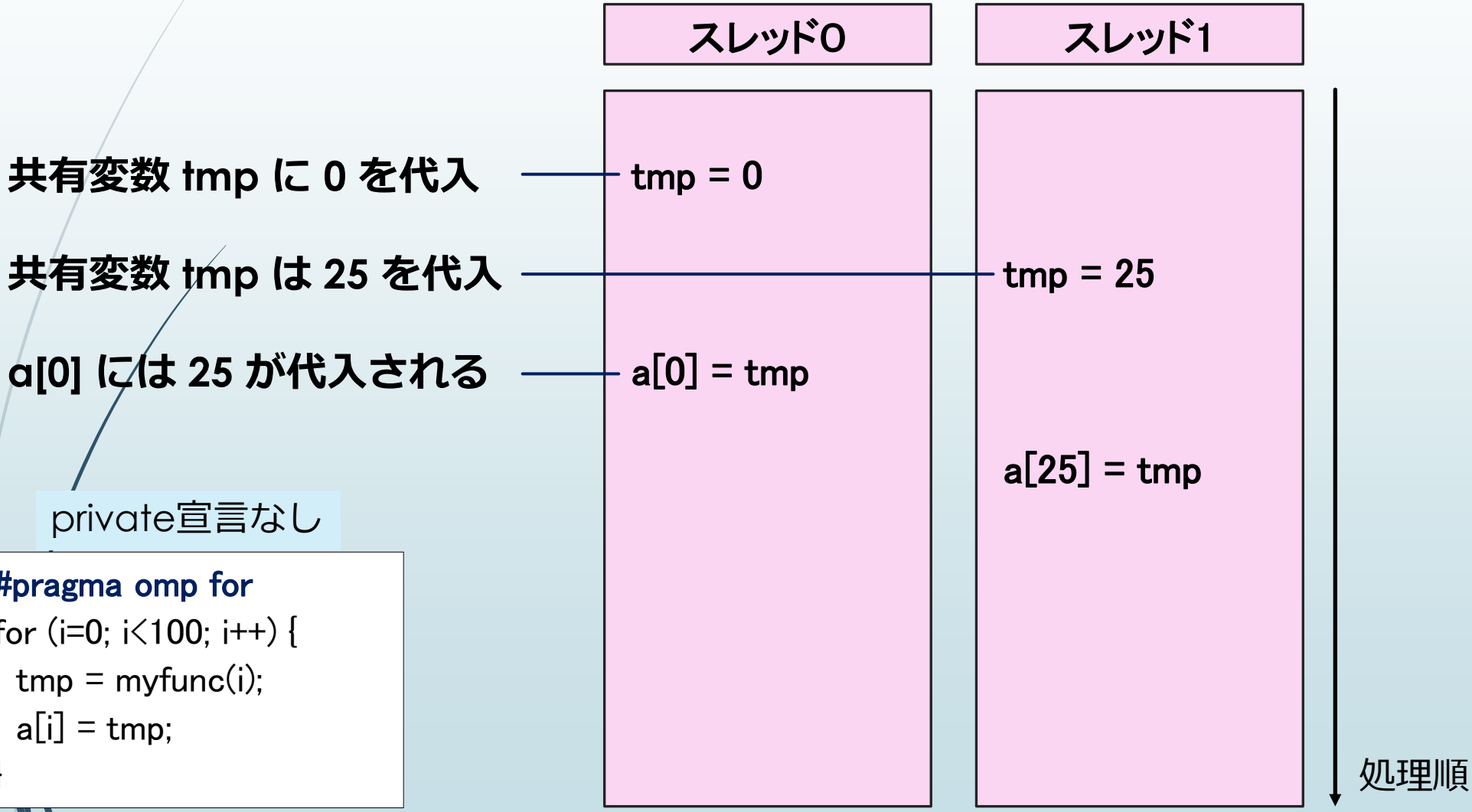
```
#pragma omp for private(tmp)
for (i=0; i<100; i++) {
  tmp = myfunc(i);
  a[i] = tmp;
}
```

C/C++の場合

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

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

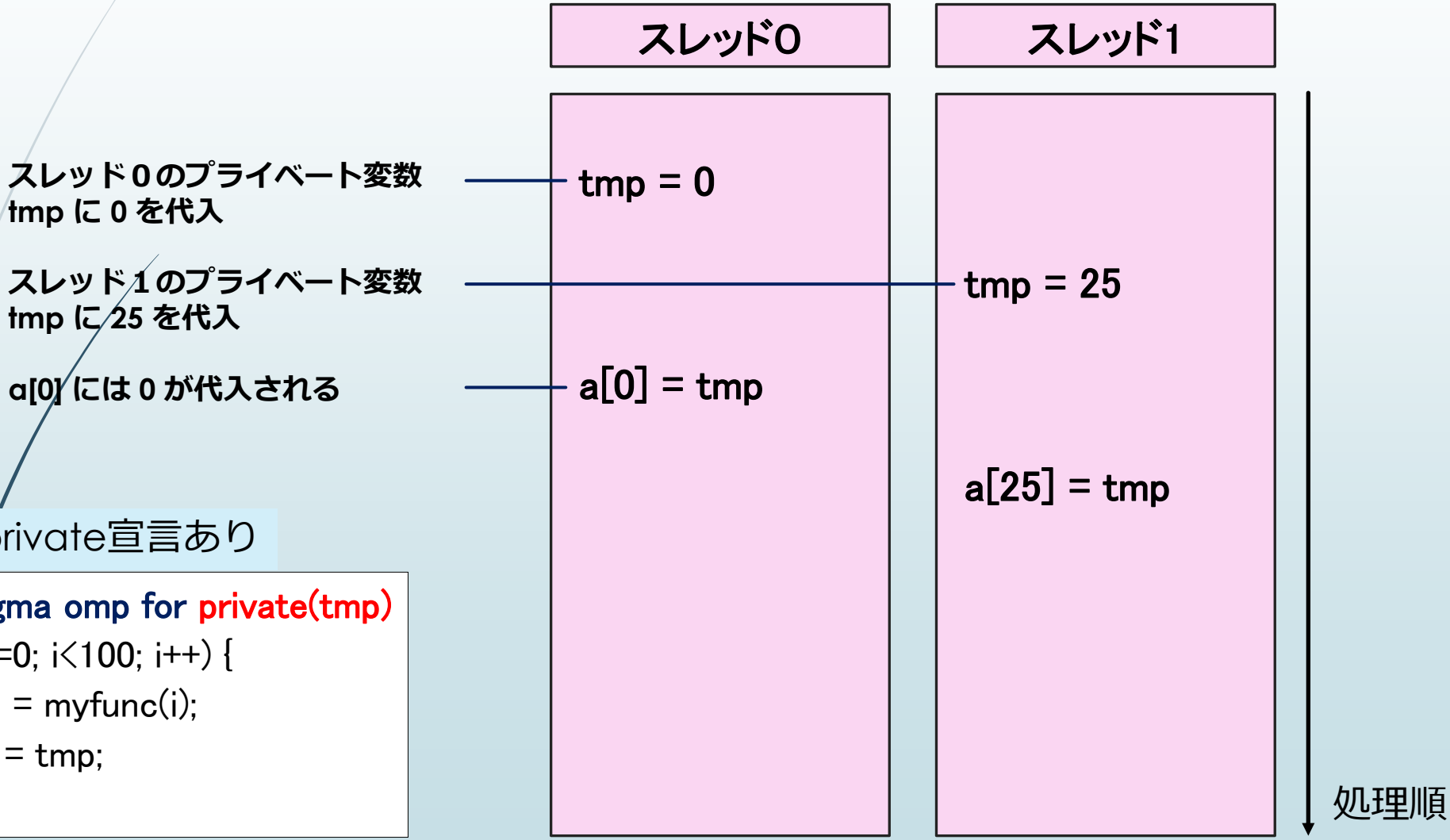
プライベート変数について



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;
}
```

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

その他のよく用いる指示文

Schedule指示節

[C] #pragma omp parallel for schedule(type[,size])

・各スレッドの仕事量が均等でない場合などに、チャンクサイズを指定して並列化する

・チャンクサイズやスケジューリングのタイプによっては、性能が悪化するケースがある

Reduction指示節

[C] #pragma omp parallel for reduction(op: var_list)

・各スレッドで計算された変数を、オプションで指定した演算でリダクションする