

OpenMP を用いた並列計算（1）

谷口 隆晴

システム情報学研究科 計算科学専攻

2014 年 5 月 29 日

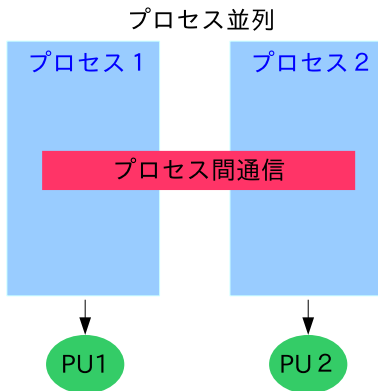
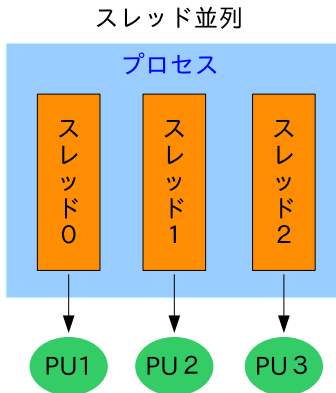
Open MP を使ってみる！

- 準備：並列計算に関する復習
- Hello World の並列化と並列計算機上での実行方法
- Do ループの並列化 (**omp do**)
- 配列代入の並列化 (**omp workshare**)
- 共有変数とプライベート変数 (**shared, private**)
- 宿題：リダクション変数と π の数値計算 (**reduction**)

共有メモリ型並列計算機における並列方式

【スレッド並列】

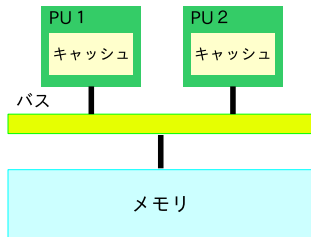
- スレッド：プロセス内の処理実行の流れ
- 同一プロセス内の各スレッドは同じメモリ空間にアクセス
- OpenMP によるプログラミングが標準的



共有メモリ型並列計算機（復習）

■ 構成

- 複数のプロセッサ（PU）がバスを通してメモリを共有
- どのPUも同じメモリ領域にアクセスできる



■ 特徴

- メモリ空間が単一のためプログラミングが容易
- PU の数が多すぎると、アクセス競合により性能が低下
→ 2~16 台程度の並列化が多い

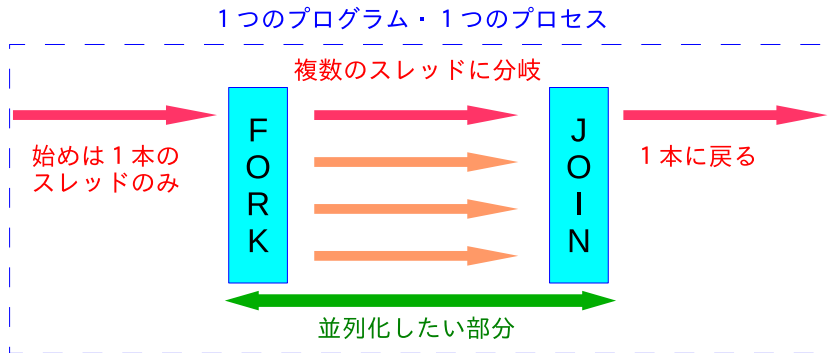
■ プログラミング言語

- OpenMP (FORTRAN/C/C++ + ディレクティブ (指示文)) を利用
- (後で学ぶ) MPI を利用することも可能

- 共有メモリ型並列計算機向け並列計算ライブラリ
 - 指示行を挿入するだけで並列化が可能
 - (始めから並列計算用プログラムを書くのではなく) 逐次コードを修正していく形でプログラミングが可能
 - 逐次プログラムとしても実行可
 - 比較的, デバッグが簡単
 - 移植性に優れる
 - プログラムを修正しなくても, 様々な共有メモリ型並列計算機で実行可
 - 解説書が豊富
 - 逐次実行部分が多くなりがち → 速くなりにくい
- 米国のコンパイラメーカーを中心に仕様を決定
 - 1997 FORTRAN Ver. 1.0 API
 - 1998 C/C++ Ver. 1.0 API
 - 2000 FORTRAN Ver 2.0 API
 - 2002 C/C++ Ver 2.0 API
 - 2005 FORTRAN C/C++ Ver 2.5 API
 - 2008 FORTRAN C/C++ Ver 3.0 API
 - 2013 Ver 4.0 Released! (アクセラレータ機能追加など)

OpenMP の実行モデル : Fork-Join モデル

- 1つのスレッド (マスタースレッド) でスタート
- 並列化部分の開始時 → 複数のスレッドに分岐 (Fork)
- 並列化部分の終了時 → マスタースレッドのみに戻る (Join)



- 元の (FORTRAN/C/C++ で書かれた) プログラム
- 指示文 (ディレクティブ)
 - 並列化すべき場所・並列化方法を指定
 - FORTRAN では **!\$omp** で開始
例)

```
!$omp parallel
```

- ライブラリ関数
例) 並列実行部分でスレッド数を取得する関数: **omp_get_num_threads()**
- 環境変数
 - 並列実行部分で使うスレッド数などを指定するのに利用
例) スレッド数を指定する環境変数: **OMP_NUM_THREADS.**

演習 1 (準備): Hello World を並列化してみよう!

まず, 逐次版プログラムを用意

- 今日の演習用のディレクトリ (例えば enshu-openmp1) を作成

```
% mkdir enshu-openmp1  
% cd enshu-openmp1
```

- emacs 等で, 以下のプログラムを作成し, hello.f90 などの名前で保存

```
program hello_world  
implicit none  
print *, "Hello World!"  
end program
```

- **frtpx** でコンパイル.

```
% frtpx hello.f90
```

- ./a.out では**実行できません**. 実行方法はこれから.

注意: 計算ノードで計算を行うのでコンパイル・実行方法が変わります

スパコンは共有財産！ ➡ ジョブの管理が必要

キューイングシステム

負荷状況・リソース使用量を監視し、ユーザが投入したジョブを適切な計算ノードに割り当て、実行するソフトウェア。

プログラム実行の流れ

- 1 ジョブスクリプトを作成
- 2 ジョブを投入
- 3 (ジョブの状態を確認)
- 4 結果を確認

```
% ./a.out
```

で実行するのでは ない

注) 自分のパソコンなどで実行する場合は ./a.out でOK.

演習 1 (続き): ジョブスクリプトの作成

演習で使うキュー

キュー名	最大ノード数	経過時間制限
small	12	10分

ジョブスクリプトの例 (OpenMP 版)

```
#!/bin/bash
#PJM -N "jobname"
#PJM -L "rscgrp=small"
#PJM -L "node=1"
#PJM -L "elapse=2:00"
#PJM -j
export OMP_NUM_THREADS=1
./a.out
```

シェルを指定
ジョブ名を指定
投入先のキュー名を指定
使用ノード数を指定

スレッド数を指定
実行プログラム名を指定

【演習】 上のスクリプトのジョブ名などを適切に修正し, hello.sh などの名前で保存.

演習 1 (続き) : ジョブの投入

■ ジョブの投入

```
pjsub (ジョブスクリプト名)
```

■ ジョブの状態確認

```
pjstat
```

JOB_ID	JOB_NAME	MD	ST	USER	START_DATE	ELAPSE_LIM	NODE_REQUIRE
17583	jobname	NM	RUN	user	(05/19 16:23)	0000:02:00	1
17584	jobname	NM	QUE	user	(05/19 16:33)	0000:02:00	1

■ ジョブのキャンセル

```
pjdel (ジョブ番号)
```

【演習】

■ Hello World のジョブを投入してみよう !

```
% pjsub hello.sh
```

[INFO] PJM 0000 pjsub Job 17583 submitted. などと表示
("17583" の部分がジョブ番号)

■ うまくいけば ジョブ名.o? (? はジョブ番号) というファイルが作成され, その中に "Hello World!" が出力されます (**cat** などで確認).

演習 1 (これで最後): OpenMP を用いた Hello World の並列化

- Hollow World プログラムに赤文字部分を追加して (追加するだけで) 並列化

```
program hello_world
implicit none
integer :: omp_get_thread_num
!$omp parallel
print *, "My id is ", omp_get_thread_num(), "Hello World!"
!$omp end parallel
end program
```

- OpenMP を用いていることを明示してコンパイル

```
% frtpx -Kopenmp hello.f90
```

- 2 スレッドで実行: hello.sh の OMP_NUM_THREADS の値を 2 に書きかえて

```
% pjsub hello.sh
```

■ 並列リージョン

- 2つの指示文 `!$omp parallel` と `!$omp end parallel` で囲まれた部分を **並列リージョン** という。
- 並列リージョン内では (OMP_NUM_THREADS) 個のスレッドが同じコードを実行。
- 各スレッドは固有のスレッド番号をもつ。これを用いて、各スレッドに異なる処理を行わせることができる。
- スレッド番号は `omp_get_thread_num()` によって取得できる。

```
program hello
implicit none
!$omp parallel
...
...   並列リージョン
...
!$omp end parallel
end program
```

■ 変数・配列の参照・更新

- すべてのスレッドが同じ変数・配列を参照できる。
- 複数のスレッドが同時に同じ変数を更新しないよう、注意が必要。
(同じ配列の、異なる要素を同時に更新するのはOK.)

OpenMP 並列化プログラムの基本構成例

```
program main  
implicit none
```

(逐次実行部分)

```
!$omp parallel
```

(並列化したい部分)

```
!$omp end parallel
```

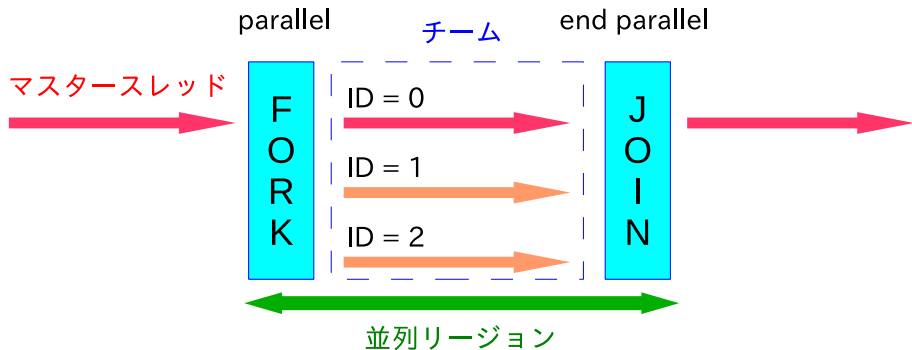
(逐次実行部分)

```
end program
```

} 並列リージョン：
複数のスレッドにより、
並列実行される部分

マルチスレッドでの実行のイメージといくつかの用語

- プログラム実行開始時はマスタースレッドのみ
- PARALLEL 指示文により複数のスレッドを生成
 - スレッド ID : 0 ~ OMP_NUM_THREADS - 1 に値をもつ, 各スレッドに割り振られる固有の番号.
 - チーム : 並列実行を行うスレッドの集まり.
 - スレッド生成後, 全てのスレッドで冗長実行
- END PARALLEL 指示文によりマスター以外のスレッドが消滅



計算の並列化 (Work-Sharing 構造)

- チーム内のスレッドに仕事 (Work) を分割 (Share)
- Work-Sharing 構文：チームに仕事を割り振るための指示文
 - DO ループの分割 (!\$OMP DO, !\$OMP END DO)
 - 別々の処理を各スレッドが分担 (!\$OMP SECTIONS, !\$OMP END SECTIONS)
 - 配列に対する操作の分割 (FORTRAN のみ, !\$OMP WORKSHARE, !\$OMP END WORKSHARE)

例) 配列の各要素に対する加算

$$a(1:n) = a(1:n) + 1$$

の並列化

- 1 スレッドのみで実行 (!\$OMP SINGLE, !\$OMP END SINGLE)
- Work-Sharing 構文以外にも
 - マスタスレッドのみで実行 (!\$OMP MASTER, !\$OMP END MASTER)

DO ループの分割 (!\$omp do)

```
program main
implicit none
integer, parameter :: SP = kind(1.0)
integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
real(DP), dimension(100000) :: a, b
integer :: i
```

```
!$omp parallel
!$omp do
do i=1,100000
  b(i) = a(i)
end do
!$omp end do
!$omp end parallel
```

直後の DO ループを複数のスレッド
で分割して実行せよ、という意味
(!\$omp end do は省略可)

```
end program
```

2 スレッドで実行した場合

スレッド 0

```
do i=1,50000
```

```
  b(i) = a(i)
```

```
end do
```

スレッド 1

```
do i=50001,100000
```

```
  b(i) = a(i)
```

```
end do
```

(分割の仕方はコンパイラ依存)

演習 2 : omp do を使ってみよう !

課題

- 次のスライドのプログラムを作成.
- スレッド数を 1, 2 と変えてみて経過時間を計測.

【時間計測の方法】 `omp_get_wtime` 関数を利用.

- 倍精度で `omp_get_wtime`, `time0`, `time1` を定義し,
- 測定したい部分を `time0=omp_get_wtime()` と `time1=omp_get_wtime()` ではさむ (今回は `!$omp parallel` の前と `!$omp end parallel` の後に挿入).
- `time1 - time0` が経過時間 (秒単位).
- 時間計測用のジョブスクリプトは, 2つ後のスライドに掲載してあります.

```
time0=omp_get_wtime()
!$omp parallel
! (時間計測する部分)
!$omp end parallel
time1=omp_get_wtime()
print *, time1-time0
```

演習 2 のプログラム

```
program axpy
implicit none
integer, parameter :: SP = kind(1.0)
integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
real(DP), dimension(100000) :: x, y, z
real(DP):: a
integer :: i
!
! a, x, y の値を各自で自由に設定.
!
!$omp parallel
!$omp do
do i = 1, 100000
    z(i) = a*x(i) + y(i)    ベクトルの加算  $z = ax + y$ 
end do
!$omp end do
!$omp end parallel
!
! 経過時間の確認
!
end program
```

時間計測用ジョブスクリプトの例

```
#!/bin/bash
#PJM -N "jobname"
#PJM -L "rscgrp=small"
#PJM -L "node=1"
#PJM -L "elapsed=2:00"
#PJM -j

export FLIB_CNTL_BARRIER_ERR=FALSE

for opn in 1 2 4
do
export OMP_NUM_THREADS=$opn
./a.out
done
```

opn を変えながら do 内を実行

スレッド数を opn に設定
実行プログラム名を指定

/tmp/openmp1/jscript.sh に置いてあります。

```
% cp /tmp/openmp1/jscript.sh ./
```

として、コピーして利用してください。

演習 3 : !\$omp parallel do

- do ループの並列化は, parallel と do をまとめて

```
!$omp parallel
```

```
!$omp do
```

```
do i=1,100000
```

```
  b(i) = a(i)
```

```
end do
```

```
!$omp end do
```

```
!$omp end parallel
```



```
!$omp parallel do
```

```
do i=1,100000
```

```
  b(i) = a(i)
```

```
end do
```

```
!$omp end parallel do
```

のように書いても良い.

- !\$omp end parallel do は省略しても良い.

【演習 3】 演習 2 のプログラムを parallel do を使って書き換えてみる.

— 注意 —

omp do は並列実行できない場合も自動的に分割してしまう！

```
program invl
implicit none
integer, parameter :: n = 100
integer, dimension(n) :: a
integer :: i

a(1) = 0
!$omp parallel do
do i=2,n
a(i) = a(i-1) + 1
end do
!$omp end parallel do

print *, a(n)

end program
```

2 スレッド
で実行



```
スレッド 0
do i=1,50
  a(i) = a(i-1) + 1
end do

スレッド 1
do i=51,100
  a(i) = a(i-1) + 1
end do
```

本当は a(50) の結果が
ないと実行できない！

正しい結果：99

do ループの並列化のまとめ

- do ループを並列化するには並列化したいループの前に `!$omp parallel do` を置けば良い.
 - ループ変数の動く範囲が `OMP_NUM_THREADS` 個に分割され,
 - 各ブロックはそれぞれ1スレッドにより実行.
 - 分割のされ方はコンパイラ依存. 同じプログラムの中でも変わり得る.
- ただし, 並列化してよいループかどうかはプログラマが判断.

並列化してはダメなループの例) 再帰参照を含むループ

```
do i=1,100
  x(i) = a*x(i-1) + b
end do
```

1つ前に計算した要素の値を使って, 現在の要素を計算.

ただし, 一見ダメそうでも, よく考えれば並列化できる場合も.

演習 2 のプログラムは次のように書いてもよい :

```
!$omp parallel
!$omp do
do i=1,100000
  z(i) = a*x(i) + y(i)
end do
!$omp end do
!$omp end parallel
```



```
!$omp workshare
  z(:) = a*x(:) + y(:)
!$omp end workshare
```

(!\$omp end workshare は省略不可)

- FORTRAN のみで使える書き方.
- コンパイラによっては matmul (行列積) なども並列化してくれる.

```
!$omp workshare
  C = matmul(A, B)
!$omp end workshare
```

のように書くと並列化してくれるコンパイラもある.

共有変数とプライベート変数

■ 共有変数

- どのスレッドからも参照・更新が可能な変数.
- OpenMP では、いくつかの例外を除き、**変数はデフォルトで共有変数**.

■ プライベート変数

- 各スレッドが独自の値を保持する変数.
- **並列化終了時に値は破棄**される.
- 例) ループインデックス変数

```
do i=1,100
  ! do something
end do
```

を2スレッドで動かす場合、**i**を2つのスレッドで共有してはダメ.

スレッド0

スレッド1

変数 **i** は

```
do i=1,50
  ! do something
end do
```

```
do i=51,100
  ! do something
end do
```

- スレッド0では1~50を、
- スレッド1では51~100を
動いて欲しい.

変数の共有・プライベートの指定

■ デフォルトの設定

- 何も指示しなければ**基本的に共有変数**.
- **並列化されたループのインデックス変数**などは、特に指定しなくても**プライベート変数**となる。

【注意】多重ループの場合は注意が必要

```
!$omp parallel do
do i=1,100
  do j=1,100
    !
    ! do something
    !
  end do
end do
!$omp end parallel do
```

左の例で、

- **i** はプライベート変数になるが
 - **j** がプライベート変数になるかは C か FORTRAN かで異なる。
- ➡ デフォルトの設定は複雑。
明示的に指定したほうが安全。

- 共有変数の指定：並列化指示文の後に **shared** 節を追加。
- プライベート変数の指定：並列化指示文の後に **private** 節を追加。
(例)

```
!$omp parallel do default(none) shared(a, b) private(i,j,k)
```

演習4：共有変数・プライベート変数

課題

次のスライドのプログラムは
「配列 a と配列 b の内容を入れ替えて
表示する」

ように作成したプログラム。

- そのままではコンパイルできないので、適切に `private / shared` を指定し、コンパイルせよ。
- 4 スレッドで実行せよ。

元の値

9 1
8 2
7 3
6 4
5 5
4 6
3 7
2 8
1 9
0 10

【結果】

1 9
2 8
3 7
4 6
5 5
6 4
7 3
8 2
9 1
10 0

演習4のプログラム

```
program swap
implicit none
integer, parameter :: n = 10
integer :: i, tmp
integer, dimension(n) :: a, b
! 入れ替え前の値を設定
do i=1,n
  a(i) = n-i
  b(i) = i
end do
! 配列の中身の入れ替え (並列実行)
!$omp parallel do default(none) (ここに shared, private 節を適切に追加)
do i=1,n
  tmp = a(i)
  a(i) = b(i)
  b(i) = tmp
end do
!$omp end parallel do
! 結果の表示
write (6, '(2i4)') (a(i),b(i),i=1,n)
end program
```

このプログラムは /tmp/openmp1/swap.f90 に置いてあります。

例) 2つのベクトルの内積

```
c = 0.0_DP
do i=1,n
  c = c + a(i) * b(i)
end do
```

を並列化したい!

変数 **c** は共有変数? プライベート変数?

- 共有変数にすると...
 - ➡各スレッドが **c** を同時に更新しようとし、正しく計算できない。
- プライベート変数にすると...
 - ➡並列化終了時に、各スレッドにおける値が破棄されてしまう。

どちらとも違う種類の変数に設定 ➡ **リダクション変数**

リダクション変数

リダクション変数：

- 並列実行時にはプライベート変数で、
- 並列終了時にある演算によって一つの値に集約されるような変数.
- 演算としては **+**, *****, **.and.**, **.or.**, **max**, **min** などが利用可能.

```
c = 0.0_DP
```

```
!$omp parallel do reduction(+:c)
```

↑ 変数と最後に適用する演算を指定

```
do i=1,n
```

```
  c = c + a(i) * b(i)
```

```
end do
```

```
!$omp end parallel do
```

変数 **c** は

- 並列実行時には、各スレッドで独立した値をもち、
- 並列終了時には **+** 演算で一つの値に結果をまとめる (**総和をとる**) .

宿題： $\pi = 4 \int_0^1 (\tan^{-1}(x))' dx$ の数値計算

課題

- 次のスライドのプログラムを並列化.
 - **omp parallel, omp do, omp parallel do** などを適切な場所に挿入.
 - **shared, private, reduction** などを適切に指定.
 - 時間測定のための記述を適切に挿入.
- 1, 2, 4 スレッドを用いた場合の3通りについて計算時間を測定.
- 【選択課題】 計算結果と真の値 (3.1415926535897...) と比較せよ.
- プログラムと時間計測の結果 (全ての場合について), 真の値と比較して気づいたことを1つのテキストファイル (例えば result.txt) に入れて, その内容を yaguchi までメール.

【メールの送り方】

```
% mail yaguchi < result.txt
```

【締切】 6月4日 (水), 午後5時.

宿題のプログラム

```
program pi
implicit none
integer, parameter :: SP = kind(1.0)
integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
integer, parameter :: n = 1000000
integer :: i
real(DP) :: x, dx, p

dx = 1.0_DP/real(n, DP)

p = 0.0_DP
do i = 1,n
x = real(i, DP) * dx
p = p + 4.0_DP/(1.0_DP + x**2)*dx
end do

print *, p

end program
```

このプログラムは /tmp/openmp1/pi.f90 に置いてあります。

- 南里豪志, 天野浩文. OpenMP 入門 (1), (2), (3),
<http://www.cc.kyushu-u.ac.jp/scp/system/library/OpenMP/OpenMP.html>.
- 黒田久泰. C 言語による OpenMP 入門,
http://www.cc.u-tokyo.ac.jp/publication/kosyu/03/kosyu-openmp_c.pdf.
- 北山洋幸. OpenMP 入門- マルチコア CPU 時代の並列プログラミング, 秀和システム, 2009.
- Barbara Chapman, Gabriele Jost and Ruud van der Pas (Foreword by David J. Kuck). Using OpenMP –Portable Shared Memory Parallel Programming–, The MIT Press, 2007.