

OpenMP を用いた並列計算（２）

三宅 洋平

計算科学教育センター
システム情報学研究科 計算科学専攻

2017 年 6 月 8 日

本資料はH27年度担当の谷口先生の資料を
参考にさせていただいています。

今日の内容

応用編

- 演習 1 : 共有変数とプライベート変数
- 演習 2 : ループでのスレッド割り当て方法の指定 (**schedule**)
- 演習 3 : 各スレッドに異なる仕事を割り当てる方法 (**omp sections**)
↑ 宿題
- 単独のスレッドで実行 (**omp single, omp master**)
- 演習 4 : スレッドの同期と制御 (**barrier, critical, atomic**)
- 演習 5 : 不要な同期の削除 (**nowait**)
↑ 自由課題

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)
```

総和の計算の並列化

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**).

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

課題

- 今日の演習用のディレクトリ（例えば enshu-openmp2）を作成

```
mkdir enshu-openmp2  
cd enshu-openmp2
```

- 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.
- 【自由課題】計算結果と真の値 (3.1415926535897...) と比較せよ.

復習：実行の仕方

- 9 ページのようなスクリプトを作成し, **jscript.sh** などの名前で保存.
- その後,

```
pjsub jscript.sh
```

- jobname.o???? (???? は適当な番号) というファイルを確認.

演習1 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/openmp2/pi.f90 に置いてあります.

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 "elapse=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

同じものが /tmp/openmp2/jscript.sh においてあります。
前回利用したものを使いまわしてもかまいません。

ループでのスレッド割り当て方法の指定

例) 三角行列とベクトルの積

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\ & & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\ & & & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ & & & & a_{55} & a_{56} & a_{57} & a_{58} \\ & & & & & a_{66} & a_{67} & a_{68} \\ & & & & & & a_{77} & a_{78} \\ & & & & & & & a_{88} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix}$$

スレッド 0 は青の部分を, スレッド 1 が緑の部分を担当.

素直に 2 つに分割: 青の要素数 = 26 個, 緑の要素数 = 10 個.



緑の部分よりも青の部分のほうが計算が大変!

スレッド 0 の計算に時間がかかってしまい, 全体としても速くならない. ➡ なるべく負荷を均一にしたい

解決策の例) ブロックサイクリック分割

例) 三角行列とベクトルの積

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\ & & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\ & & & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ & & & & a_{55} & a_{56} & a_{57} & a_{58} \\ & & & & & a_{66} & a_{67} & a_{68} \\ & & & & & & a_{77} & a_{78} \\ & & & & & & & a_{88} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix}$$

スレッド 0 は青の部分を, スレッド 1 が緑の部分を担当.
2 行からなるブロックごとに分割: 青の要素数 = 22 個, 緑の要素数 = 14 個.

➡ ちょっと改善.

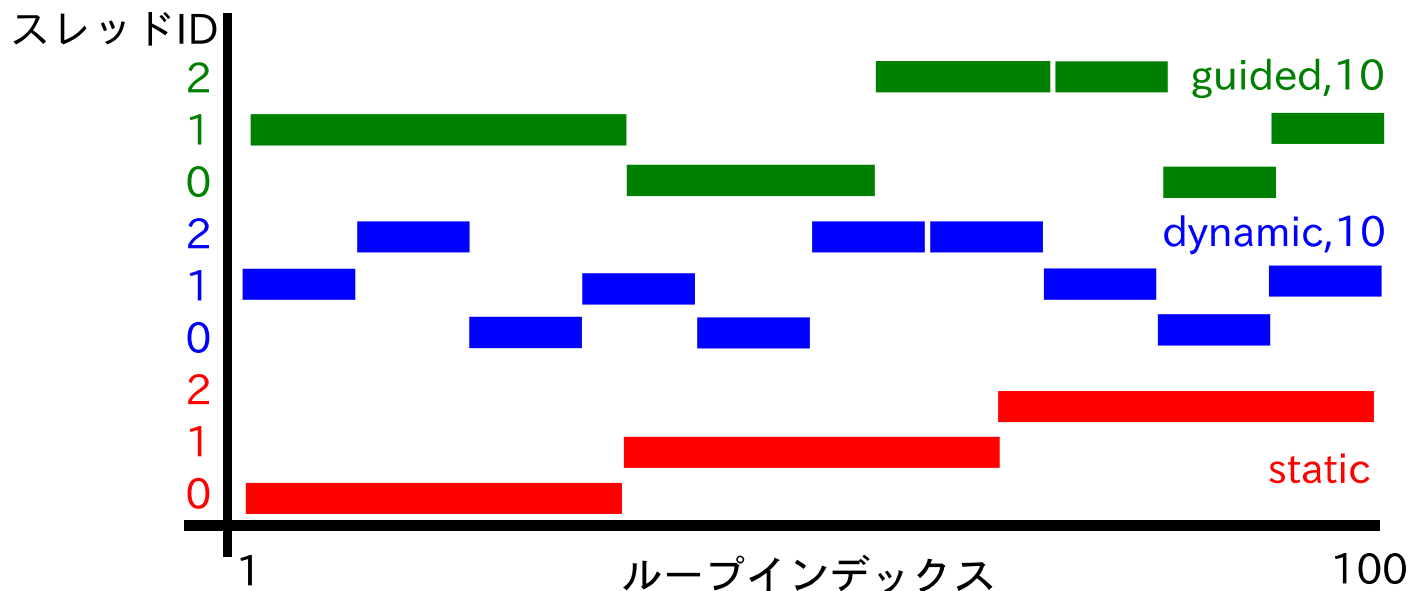
OpenMP では

```
!$omp parallel do schedule(static,2)
```

【書き方】 **schedule(種類, サイズ)**

例) **!\$omp parallel do schedule(static, 4)**

- サイズは指定しなくても良い（指定しない場合、適切な値に自動設定）。
- 種類は次の中から指定。
 - **static** : 先ほどのブロックサイクリック分割。
 - **dynamic** : 1ブロックずつから始め、終わったスレッドが順次、次を実行。
 - **guided** : **dynamic** と同様だが、ブロックサイズを徐々に細かくしていく（最低でも指定サイズ）。
 - **runtime** : プログラムの実行時に環境変数 **OMP_SCHEDULE** で指定。



演習 2 : いろいろな分割方法を試してみよう !

- 1 次のスライドのプログラム (/tmp/openmp2/schedule.f90) について
並列化指示行

```
!omp parallel do schedule(,
```

の **schedule** の部分を, いろいろなサイズの **static, dynamic, guided** に設定.

- 2 プロセッサ数を 4 として計算時間を比較.

演習 2 のプログラム

```
program schedule
implicit none
integer, parameter :: SP = kind(1.0)
integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
integer, parameter :: n = 2000
integer :: i,j
real(DP), dimension(n) :: x, y
real(DP), dimension(n,n) :: A
real(DP) :: time0, time1, omp_get_wtime
x(:) = 2.0_DP
A(:, :) = 1.0_DP
time0 = omp_get_wtime()
!$omp parallel do schedule(,) default(none) private(i,j) shared(A,x,y)
do i=1,n
  y(i) = 0.0_DP
  do j=i,n
    y(i) = y(i) + A(i,j) * x(j)
  end do
end do
!$omp end parallel do
time1 = omp_get_wtime()
print *, time1-time0
end program
```

復習：ループの順番とキャッシュミス

演習 2 のプログラム

```
!$omp parallel do
do i=1,n
  y(i) = 0.0_DP
  do j=i,n
    y(i) = y(i) + A(i,j) * x(j)
  end do
end do
!$omp end parallel do
```

(復習) ループを回す順番でキャッシュミスの多さが変化 ➡ **変えてみよう!**

のループを入れ替えて

```
do j=1,n
  y(i) = 0.0_DP
  do i=1,j
    y(i) = y(i) + A(i,j) * x(j)
  end do
end do
```



```
y(:) = 0.0_DP
do j=1,n
  do i=1,j
    y(i) = y(i) + A(i,j) * x(j)
  end do
end do
```

【演習 2'】 ループを上のように修正して、実行時間を比較してみよ。

各スレッドに別々の仕事を割り当て（!\$omp sections）

例：質点の運動のシミュレーション

program

do while ($t < \text{必要な時間}$)

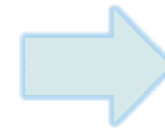
（ x 軸方向の更新）

（ y 軸方向の更新）

（ z 軸方向の更新）

end do

end program



!\$omp sections の特徴

- それぞれの **section** を別々のスレッドが実行.
- 他のスレッドは待機.
- 実行される順序は指定できない.

!\$omp parallel

!\$omp sections

!\$omp section

！（ x 軸方向の更新）

omp end section は書かない.

!\$omp section

！（ y 軸方向の更新）

!\$omp section

！（ z 軸方向の更新）

!\$omp end sections

!\$omp end parallel

演習 3 (宿題)

【課題】

- 1 プログラム `/tmp/openmp2/sierp.f90` を各自のディレクトリにコピー.
- 2 do ループ中の $x(i)$ の計算と $y(i)$ の計算は並列計算できるので **sections** を利用して並列化.
(**sections** を使えるように, うまくプログラムを書き換えること.)
- 3 1 スレッド, 2 スレッドで実行した場合について計算時間を比較.

【終わった人は】 結果を gnuplot で表示

- プログラム中の計算時間出力部分を消す. 最後 3 行のコメントを外して $x(i)$, $y(i)$ を出力.
- 1 プロセッサで実行.
- ~~一度, ログアウトして X サーバを有効にして再ログイン.~~
- gnuplot → plot "jobname.o???" → exit

MobaXterm であればログアウトの必要なし

(終わったらログアウトして, X サーバ無しで再ログイン)

演習 3 のプログラム

```
program sierpinski
implicit none
! 変数の定義など
do i=1,n
  call random_number(myrand(i))
end do

time0 = omp_get_wtime()
do i=1,n-1
  if (myrand(i) < 0.33_DP) then
    x(i+1) = x(i) * 0.5_DP + 1.0_DP
    y(i+1) = y(i) * 0.5_DP
  else if (myrand(i) > 0.66_DP) then
    x(i+1) = x(i) * 0.5_DP - 1.0_DP
    y(i+1) = y(i) * 0.5_DP
  else
    x(i+1) = x(i) * 0.5_DP
    y(i+1) = y(i) * 0.5_DP + 1.0_DP
  end if
end do
time1 = omp_get_wtime()

print *, time1-time0

!do i = 1,n
! print *, x(i), y(i)
!end do
end program
```

赤と青の部分を実行を別々のプロセッサで実行。

早めに終わった人は、出力部分を変更して gnuplot で表示。

課題の説明

プログラムは以下のランダムウォークの軌跡を計算：

- 確率 1/3 で

$$x^{(n+1)} = \frac{1}{2}x^{(n)} + 1, \quad y^{(n+1)} = \frac{1}{2}y^{(n)}.$$

- 確率 1/3 で

$$x^{(n+1)} = \frac{1}{2}x^{(n)} - 1, \quad y^{(n+1)} = \frac{1}{2}y^{(n)}.$$

- 確率 1/3 で

$$x^{(n+1)} = \frac{1}{2}x^{(n)}, \quad y^{(n+1)} = \frac{1}{2}y^{(n)} + 1.$$

実は、軌跡がシェルピンスキーギャスケットという絵になる。

宿題

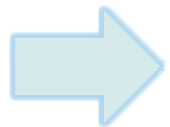
- 1 演習 3.
- 2 【自由課題】 終わった人は以下の自由課題も試してみてください.
- 3 プログラムと実行結果 (=スレッド数を変えたときの実行時間) を, 1つのテキストファイル (例えば result.txt) に入れて, その内容を下記 までメール.

【送り方】 `mail -s アカウント名 kobeuniv.compra1@gmail.com < result.txt`

【締切】 6/14(水)、午後5時.
なるべくこの時間に終わらせましょう！

一つのスレッドだけで実行 (!\$omp single)

```
!$omp parallel
val = 1.0_DP
!$omp do
do j=1,n
  do i=1,n
    A(i,j) = val
  end do
end do
!$omp end do
!$omp end parallel
```



```
!$omp parallel
!$omp single
val = 1.0_DP ← (1 スレッドだけで実行, 他は待機)
!$omp end single
!$omp do
do j=1,n
  do i=1,n
    A(i,j) = val
  end do
end do
!$omp end do
!$omp end parallel
```

左側のコードでは **val** の値に全てのスレッドが同時に書き込む

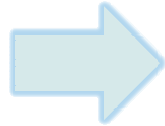
- 理論的には大丈夫だが,
- 同じアドレスに同時にアクセス ➡ パフォーマンスの低下

マスタースレッドだけで実行 (!\$omp master)

single を利用

master を利用

```
!$omp parallel
val = 1.0_DP
(何か別の処理)
!$omp do
do j=1,n
  do i=1,n
    A(i,j) = val
  end do
end do
!$omp end do
!$omp end parallel
```



```
!$omp parallel
!$omp single
val = 1.0_DP
!$omp end single
(何か別の処理)
!$omp do
do j=1,n
  do i=1,n
    A(i,j) = val
  end do
end do
!$omp end do
!$omp end parallel
```

```
!$omp parallel
!$omp master
val = 1.0_DP
!$omp end master
(何か別の処理)
!$omp do
do j=1,n
  do i=1,n
    A(i,j) = val
  end do
end do
!$omp end do
!$omp end parallel
```

- !\$omp master: マスタースレッドだけで実行. 他は待たない.
- 次の処理を進められる場合に有効.
(何か別の処理)の部分では val を使ってはいけない (更新が終わっていないため). val にアクセスする際は, 次に説明する barrier が必要.

スレッドの同期と制御

- **!\$omp barrier** : 全てのスレッドがここに来るまで待機.
 - **!\$omp end do**, **!\$omp end single** などの後には, 自動的に **barrier** が設置される.
 - **!\$omp end do nowait** などとすることで, 設置しないようにもできる.
- **!\$omp critical** : 同時に2つ以上のスレッドが実行しないようにする.
- **!\$omp atomic** : 同時書き込みの禁止 (スカラー値の更新のみ).

```
real(DP) :: sval, pval
real(DP), dimension(n) :: svec
!$omp parallel shared(sval, svec) private(pval)
  (pval の値を各スレッドで計算)
  !$omp critical
    svec(:) = pval * svec(:)
  !$omp end critical
  !$omp atomic
    sval = sval + pval
    omp end atomic は書かない
!$omp end parallel
```

演習 4 : reduction を使わない総和計算（自由課題）

【自由課題】 下記のプログラムを次の3通りに修正し，6スレッドで実行。

- 1 そのまま実行。
- 2 **omp atomic** の部分を削除して実行。
- 3 **omp atomic** の代わりに **omp critical, omp end critical** を用いたプログラムを作成し，実行。

何回か

```
program summation
integer, parameter :: SP=kind(1.0)
integer, parameter :: DP=selected_real_kind(2*precision(1.0_SP))
integer, parameter :: n=1000
real(DP) :: sval, pval
real(DP), dimension(n) :: svec
svec(:) = 1.0_DP
sval = 0.0_DP
!$omp parallel shared(sval, svec) private(pval)
pval=0.0_DP
!$omp do
do i=1,n
pval = pval + svec(i)
end do
!$omp end do
!$omp atomic
sval = sval + pval
!$omp end parallel
print *, sval
end program
```

ソースファイルは /tmp/openmp2/sum.f90 に置いてあります。

演習 5 : nowait を使うことによる高速化 (自由課題)

omp end do の後には **barrier** が置かれるが, ここで全員が揃うまで待っている必要はない → **nowait** を挿入することで **barrier** を除去.

```
!$omp parallel shared(sval, svec) private(pval)
pval=0.0_DP
!$omp do
do i=1,n
pval = pval + svec(i)
end do
!$omp end do nowait
!$omp atomic
sval = sval + pval
!$omp end parallel
```

【自由課題】 演習 4 のプログラムについて **nowait** を入れた場合, 入れない場合の実行速度 (6 スレッド) を比較.

参考文献

- 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.

質問は y-miyake@eagle.kobe-u.ac.jp まで.