

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

2011 年 5 月 26 日

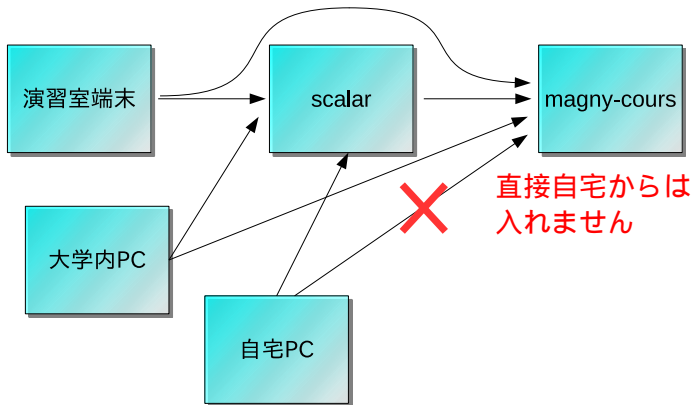
出席の確認のため、端末を立ち上げて
scalar にログインしておいて下さい。

Open MP を使ってみる！

- 準備：今回，次回で利用する計算機の準備
- 演習 1：Hello World の並列化と並列計算機上での実行方法
- 演習 2：Do ループの並列化 (**omp do**)
- 演習 3：配列代入の並列化 (**omp workshare**)
- 演習 4：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 の計算 (**reduction**)

今回・次回の演習で使うマシン

今回と次回は、いつもの scalar マシンではなく、別のマシン（ホスト名：magny-cours, 48 コア）で演習を行います。



ただし出席は **scalar** でとります！ 必ず **scalar** にもログインして下さい。

ターミナルを新しく起動し、以下のコマンドを実行

- 1 公開鍵を転送

```
% scp .ssh/authorized_keys 133.30.112.246
```

- 2 48 コアマシンへログイン

```
% ssh 133.30.112.246 # 今日の授業中のみパスワードログイン可
```

- 3 公開鍵を配置

```
% mkdir .ssh  
% mv authorized_keys .ssh
```

- 4 アクセス権を設定

```
% chmod 600 .ssh/authorized_keys
```

- 5 パスワードを変更

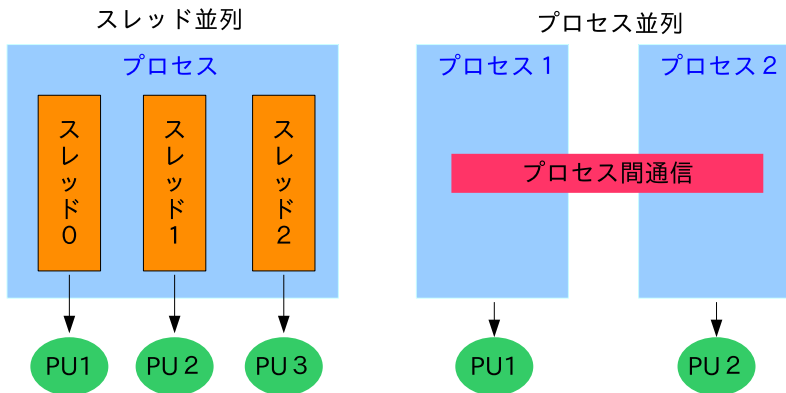
```
% passwd
```

今後は `scalar` と同様の方法 (ただし `scalar.scitec.kobe-u.ac.jp` を `133.30.112.246` におきかえる) でログインできます。

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】

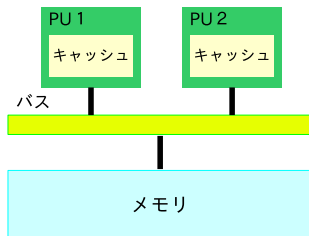
- スレッド：プロセス内の処理実行の流れ
- 同一プロセス内の各スレッドは同じメモリ空間にアクセス
- OpenMP によるプログラミングが標準的



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

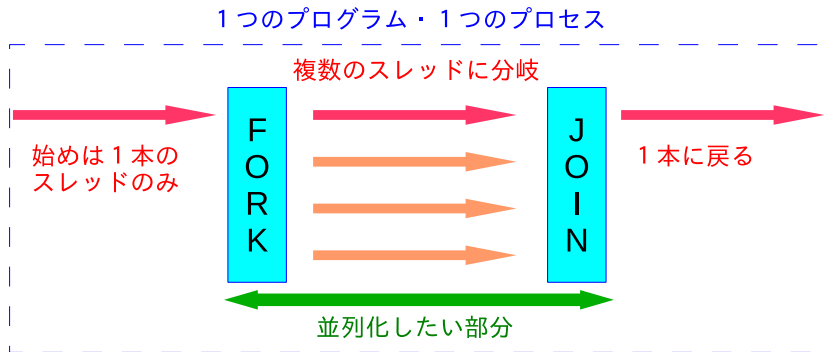
● プログラミング言語

- **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
 - (始めから並列用コードを書くのではなく) 逐次コードを修正していく形でプログラミングが可能
 - 並列プログラム・逐次プログラムのコードを共通にできる
 - 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できる)
 - 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

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

- 48 コアマシンで, 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・コンパイル

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

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．
- この演習では TORQUE Resource Manager を使用．

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 (十分な数の PU 等が確保できないことがある)

注) 自宅で実行する場合などは ./a.out で OK .

演習 1 (続き): ジョブスクリプトの作成

48 コアマシンで用意されているキュー

キュー名	最大 PU 数	ノード数	同時利用可能ユーザ数
default	48	1	64

ジョブスクリプトの例 (OpenMP 版)

```
#!/bin/bash
#PBS -N jobname
#PBS -l nodes=1
#PBS -l ncpus=1
#PBS -q default
#PBS -j oe
cd /home/username/enshu-openmp1/
export OMP_NUM_THREADS=1
./a.out
```

シェルを指定
ジョブ名を指定
使用ノード数を指定
使用プロセッサ数を指定
投入先のキュー名を指定
作業ディレクトリを指定
ncpus と同じ値を指定
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・作業ディレクトリ・実行プログラム名を適切に修正し, hello.sh などの名前で作成。

演習 1 (続き): ジョブの投入

- ジョブの投入

```
qsub (ジョブスクリプト名)
```

- ジョブの状態確認

```
qstat
```

RequestID	ジョブ名	ユーザ名	実行時間	状態 (R: 実行中, Q: 実行待ち)	キュー名
Job id	Name	User	Time Use	S	Queue
5.localhost	hello_world	yaguchi	0	R	default

- ジョブのキャンセル

```
qdel (ジョブ番号)
```

【演習】

ジョブ番号は qstat で表示される RequestID のもの .

- Hello World のジョブを投入してみよう！

```
% qsub hello.sh
```

168.magny-cours.localdomain などと表示

➡ "168" の部分が RequestID

- うまくいけばジョブ名.o? (? は RequestID) 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

```
% pgf95 -mp hello.f90
```

- 2 スレッドで実行: hello.sh の ncpus, OMP_NUM_THREADS の値を (両方共) 2 に書きかえて

```
% qsub 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
```

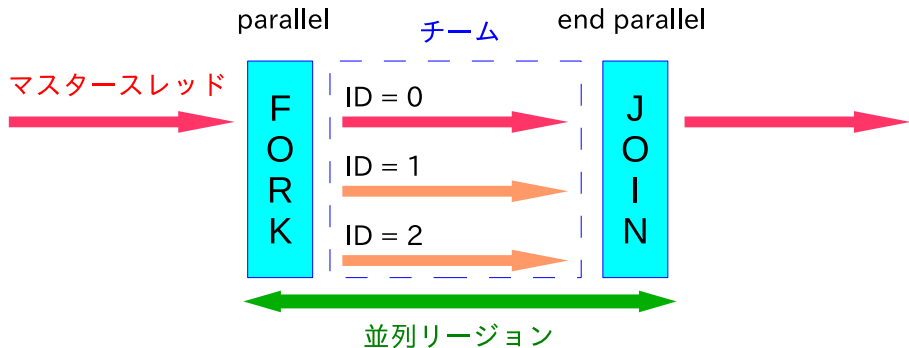
(逐次実行部分)

```
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



- 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
#PBS -N jobname
#PBS -l nodes=1
#PBS -l ncpus=2
#PBS -q default
#PBS -j oe
cd /home/username/enshu-openmp1/
for opn in 1 2
do
export OMP_NUM_THREADS=$opn
./a.out
done
```

最大使用プロセッサ数を指定

作業ディレクトリを指定
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(作業ディレクトリは変更が必要)。

!\$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.

— 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 : 9 9

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 3 : omp workshare

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)
```

のように書くと並列化してくれるコンパイラもある .

```
!$omp end workshare
```

【演習 3】 演習 2 のプログラムを workshare を使って書き換えみよう !

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 shared(a, b) private(i,j,k)
```

課題

次のスライドのプログラムは

配列 a と配列 b の内容を入れ替える．
ただし，配列 a については偶数番目を
0 にした上で入れ替える．また，入れ
替えた後の配列 b の和を表示する．

ように作成した**つもり**のプログラム．

- これを 6 スレッドで実行．
- 正しい結果にならないはずなの
で，適切に private / shared を指
定し，正しいプログラムに修正し
た上で再実行．

注) 偶然，正しい結果になるときもあります．

元の値

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

【正しい結果】

```
1 9
2 0
3 7
4 0
5 5
6 0
7 3
8 0
9 1
10 0
sum of b: 25
```

演習 4 のプログラム

```
program swap
implicit none
integer, parameter :: SP = kind(1.0)
integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
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)
!$omp parallel do (ここに shared, private 節を適切に追加)
do i=1,n
  tmp = a(i)
  if(mod(i,2)==0)then
    tmp = 0
  end if
  a(i) = b(i)
  b(i) = tmp
end do
!$omp end parallel do
! 結果の表示
write (6, '(2i4)') (a(i),b(i),i=1,n)
write (6,'(a11,i4)') 'sum of b: ', sum(b)
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**）．

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．
- プログラムと時間計測の結果（全ての場合について）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】48コアマシン上で

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

【締切】5月31日，午後5時．

分からない場合は yaguchi@pearl.kobe-u.ac.jp までメールを下さい．

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
```

- 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.