OpenMP チュートリアル

OpenMP は，共有メモリマルチプロセッサ上のマルチスレッドプログラミングのための API です．本稿では，OpenMP の簡単な解説とともにプログラム例を使って説明します．

詳しくは，OpenMP の規約を決めている OpenMP ARB の `http://www.openmp.org/` にある仕様書を参照して下さい．日本語訳は，`https://www.openmp.org/wp-content/uploads/OpenMP30spec-ja.pdf` で公開されています．

1. OpenMP の特徴と並列プログラミングモデル

OpenMP は，新しい言語ではありません．C や Fortran などの既存の逐次言語にプラグマ（`#pragma` で始まる C の指示文のこと）やコメント行（Fortran では`!$`で始まる行）で，指示を加えることにより，OpenMP の並列プログラミングモデルに従ったプログラミングをするための使用を定めたものです．

OpenMP では以下のような特徴があります．

(A) 既存の逐次プログラムをベースに並列プログラムを作ることができる．
(B) 指示文を使って，スレッドを生成，制御することができ，スレッドライブラリなどを使うよりも簡単にスレッドプログラミングができる．
(C) 徐々に指示文を加えることにより，段階的に並列化をすることができる．
(D) 基本的に，OpenMP の指示文を無視することにより，元の逐次プログラムになる（逐次の semantics を保持している）．従って，逐次と並列プログラムを同じソースで管理することができる．

このような特徴から，MPI のメッセージ通信のプログラミングに比べ非常に簡単に並列化することができます．

OpenMP の規約では，これらの要素を定義しています．

(A) 指示文
(B) 実行時ルーチン
(C) 環境変数

図 1 に OpenMP のアーキテクチャの概略について示します．OpenMP はこれらの要素を通じて，共有メモリのマルチプロセッサの並列プログラミングモデルを提供しています．共有メモリを使ったマルチスレッドプログラミングモデルでは，共有メモリ上でプロセッサによる複数の実行の流れを制御するプログラムを書きます．スレッドとは実行の流れのことで，OpenMP ではこの制御をコンパイラに対する指示文で行います．

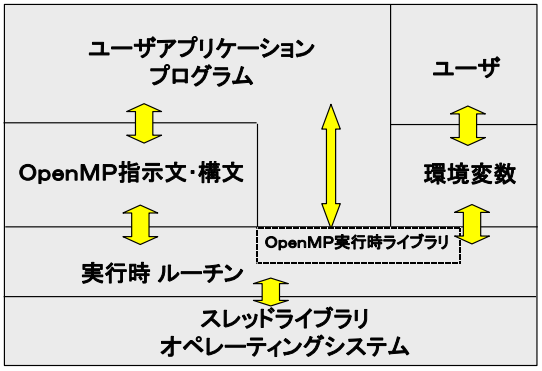

図 1　OpenMP のアーキテクチャの概略．

```
... A ...
#pragma omp parallel
{
   foo();  /* B */
}
... C ...
#pragma omp parallel
{
... D ...
}
... E ...
```

図 2　OpenMP のプログラム例.

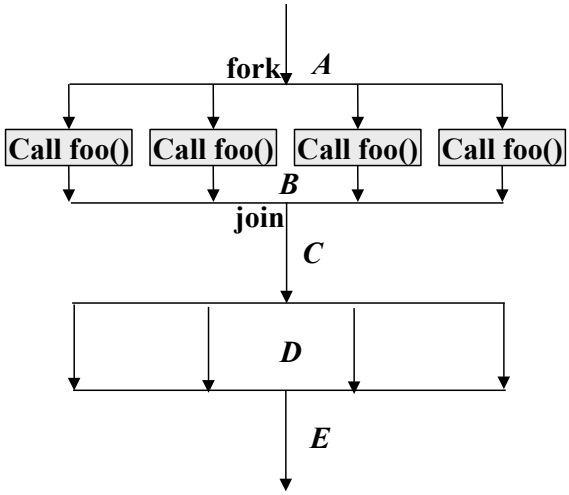

図 3　プログラムの動作の流れ.

OpenMP のプログラムは，通常の逐次プログラムと同じように main から始まります．#pragma で始まる行は指示文といいます．C 言語では#pragma omp で始まるプラグマを用います．

#pragma omp OpenMP 指示文 ...

指示文がなければ，通常の逐次プログラムと何ら変わりがありません．

図 2 のようなプログラムを考えてみましょう．このプログラムは，図 3 に示す流れで実行されます．まず，A は通常の逐次プログラムと同じように実行されます．次に，parallel 指示文#pragma omp parallel に続くブロック文が複数のスレッドで並列に実行されます．このブロック文の実行が終わると，全てのスレッドの終了を待って逐次に戻り，C の部分が逐次に実行されます．また，次の#pragma omp parallel があると，このブロック文が複数のスレッドで実行されます．逐次から複数のスレッドになることを fork，1 つのスレッドに戻ることを join といい，このような実行モデルは fork-join モデルといいます．B や D の parallel 指示文があると，この中の文は重複して実行されます．例では，B の関数呼び出しも含めて，それぞれのスレッドで実行されます．parallel 指示文で複数のスレッドで実行されるブロックを並列リージョンと呼びます．また，この並列リージョンを実行する複数のスレッドのことを team と呼びます．この team 内のスレッドは 0 から番号がつけられており，元の逐次部分を実行しているスレッドは 0 番になり，これをマスタースレッドと呼びます．

2.　Hello World：OpenMP による並列プログラミング

さて，具体的な例を使って説明していくことにしましょう．まずは，よく C のプログラムで初めに学習する Hello World のプログラムの OpenMP 版を考えることにします．図 4 にプログラム例を示します．ここでは，スレッドの番号（すなわち，0 から始まるスレッドの番号）を出すことにします．

プログラム中，#pragma で始まる行はコンパイラに対する指示文です．#で始まっているので，通常の C コンパイラにとってはコメント行です．pragma の後の omp キーワードにより，OpenMP コンパイラはこのコメント行がコンパイラに対する指示文であると認識します．parallel 指示文は，次に続く文あるいはブロック

```
#include <stdio.h>
#include <omp.h>
int main()
{
#pragma omp parallel
   {
      printf("Hello World from %d of %d\n",
         omp_get_thread_num(), omp_get_num_threads());
   }
}
```

図 4　Hello World を出力する OpenMP プログラム.

を並列に実行するコードを生成させます.

printf では，OpenMP 処理系の実行ライブラリである omp_get_threads_num 関数，および omp_get_num_threads 関数が呼ばれています．これらの関数はそれぞれ，スレッド番号，スレッド数を返す関数です．上記プログラムを GNU C コンパイラ GCC でコンパイルして実行してみましょう.

```
% gcc -fopenmp -o omphello omphello.c
% ./omphello
```

#pragma parallel で指定された部分がそれぞれのスレッドで実行され，4 コアをもつマシンでは次のような結果が得られるはずです.

```
Hello World from 0 of 4
Hello World from 2 of 4
Hello World from 1 of 4
Hello World from 3 of 4
```

OpenMP では，並列部分がいくつのスレッドで実行されるかは，プログラムでは指定しません．通常，共有メモリマシンで実行する場合には何個のコアがあるかを実行開始時に調べ，コア数と同じ数のスレッドが生成され，それぞれのコアでスレッドが実行されます．実験で用いるスーパーコンピュータ Cygnus は，1 ノードに 12 コアの CPU が 2 基搭載されており，総コア数が 24 コアなので,

```
Hello World from 0 of 24
Hello World from 1 of 24
Hello World from 2 of 24
...
Hello World from 23 of 24
```

となるはずです．確かめて下さい.

もしスレッド数を変えたい場合は，環境変数 OMP_NUM_THREADS で制御します．bash 環境では，以下の

ようにして環境変数をセットします．

```
% export OMP_NUM_THREADS=4
```

このスレッド数は，実際のコア数よりも多くても少なくても構いません．コア数よりもスレッド数が少ない場合には一部のコアしか使われません．コア数よりも多いスレッド数が指定された場合には，オペレーティングシステムのスケジューリングにより各スレッドにコアが適当にスレッドを割り当てられて，実行されます．この場合には，スレッド数をコア数よりも増やしたからといって，実行速度が速くなるわけではないことを注意して下さい（実際，遅くなることもあります）．

本当に CPU が並列に動いているのか．これを確かめるためには top コマンドを使います．Hello World のプログラムでは，あまりにも実行時間が短くて，ちょっとわかりにくいかもしれません．このコマンドは，同時に login しているユーザがプログラムを動かしているときにも表示されますから，他の人が使っていないことを確認するにも便利です．

3. ワークシェアリング指示文の使い方：ベクトル計算の並列化

OpenMP では，並列リージョンは全てのスレッドで同じコードが実行されます．スレッド番号を取得し明示的にマルチスレッドプログラミングをすることもできますが，ワークシェリング指示文を使うことによって，ループなどを簡単に並列化することができます．ワークシェアリング指示文とは，並列リージョンで team 内のスレッドで指示された文を分割して実行するための指示文です．前に，並列リージョンでは，同じ文を重複して実行すると述べましたが，ワークシェアリング指示文のところでは指示された部分を分割して実行します．

図 5 の例を考えてみましょう．関数 sum は，n 個の数の和を逐次的に求める関数です（図 6 参照）．これを

```
#include <stdio.h>
int A[1000];
int sum(int *a, int n)
{
  int s, i;
  s = 0;
  for (i=0; i<n; i++) s += a[i];
  return s;
}
int main()
{
  int i;
  for (i=0; i<1000; i++) A[i] = i;
  printf("sum = %d\n", sum(A,1000));
}
```

図 5　n 個の数の和を求めるプログラム．

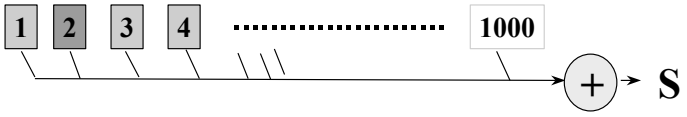

図 6　逐次処理の場合.

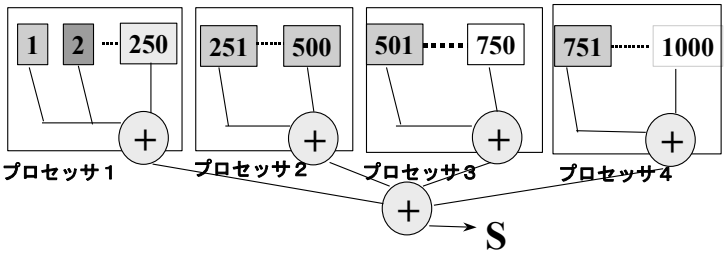

図 7　並列処理の場合.

並列化するためには，加算する配列を分割して，各スレッド（コア）がその部分を加算して，その結果を最終的に合計して，全体の加算をすればよいことになります（図 7 参照）.

このようなプログラミングの場合には，for 指示文が便利です．for 指示文は，ループを並列化するためのワークシェアリング指示文です．OpenMP で並列化された関数 sum を図 8 に示します．parallel 指示文で生成されたスレッドは，for 指示文により for ループの各部分を分担して実行します．for 指示文は，並列リージョンを実行する複数のスレッドで for 指示文の後にあるループを並列に実行します．例えば，4 スレッドで並列実行している場合には，図 8 の例では i が 0 から 249 まではスレッド 0，250 から 499 まではスレッド 1，... というように各スレッドで並列に実行します．この場合は均等にあらかじめ分割して実行しますが，ループの実行時間がばらつく場合などには動的にループを実行するなど，実行の仕方も指定することができます．for 指示文では，並列実行するループを全てのスレッドがそのループの実行が終了するまで，待ち合わせます.

並列リージョンに 1 つの for 指示文で指定される並列ループのみがある場合には，以下のように 1 つにする

```
int sum(int *a, int n)
{
  int s, i;
  s = 0;
#pragma omp parallel
{
#pragma omp for reduction(+:s)
  for (i=0; i<n; i++) s += a[i];
}
  return s;
}
```

図 8　OpenMP で並列化された関数 sum.

ことができます.

```
#pragma omp parallel for reduciton(+:s)
   for (i=0; i<n; i++) s += a[i];
```

さて，OpenMP の指示文にある reduction は何を示すのでしょうか？この文は，変数 s が共有されて加算される変数であることを指示します．このような指示句はデータスコープを指定するものです．例えば，次のような例を考えてみましょう.

```
#pragma omp parallel
   {
#pragma omp for private (t)
      for (i=0; i<1000; i++){
 t = ...;
 ... = ... t ...;
      }
      ...
   }
```

for 指示文の後にある private(t) は，ループ並列実行する場合に変数 t をそれぞれのスレッドで別々の変数をもつことを指定するもので，変数のスコープ属性の指定をするものです．通常，何も指定しない変数は全てのスレッドで共有されます．しかし，例にある変数 t のようにループ内で一時的に使われる変数の場合は，private(t) がないと並列実行しているスレッドが同じ変数に書き込んでしまうため，正しく並列化ができなくなります.

データスコープ属性には，以下の種類があります.

- shared(*var_list*)　構文内で指定された変数がスレッド間で共有される.
- private(*var_list*)　構文内で指定された変数が private.
- firstprivate(*var_list*)　private と同様であるが，直前の値で初期化される.
- lastprivate(*var_list*)　private と同様であるが構文の終了時に逐次実行された場合の最後の値を反映する.
- reduction(*op*:*var_list*)　reduction アクセスすることを指定する．実行中は private，構文終了後に反映.

4. その他の指示文

ワークシェアリング指示文には，ループを並列化する for 指示文の他に，1 つのスレッドのみで実行する single 指示文，異なる部分を別々のスレッドで実行する section 指示文があります.

以下のコードでは，single 指示文で指定されたブロック分は 1 つのスレッドでしか実行されません．

```
#pragma omp single
{
    ... statements ...
}
```

この指示文では，全てのスレッドが到着するまで待ち合わせをします．

section 指示文では，`#pragma omp sections` で囲まれたブロックの中で，`#pragma omp section` で指示された部分は別々のスレッドで実行されます．これを用いて，いわゆるタスク並列のプログラミングを行うことができます．

```
#pragma omp sections
{
#pragma omp section
 { ... section1 ... }
#pragma omp section
 { ... section2 ... }
}
```

また，この指示文でも全てのスレッドはこの指示文を実行するまで待ち合わせます．

`#pragma omp parallel` で複数スレッドで実行させるときに，全てのスレッドを待ち合わせる操作がバリア操作です．この操作を行う指示文がバリア指示文です．

```
#pragma omp barrier
```

なお，OpenMP の指示文は複数のスレッドで実行されている場合にしか有効でありません．つまり，`#pragma omp parallel` で指定される並列リージョン以外では無効になります（ただし，並列リージョン内から呼び出された関数でも，複数のスレッドで同時に実行されていますから，有効になることがあります）．

5. Laplace 方程式

OpenMP で並列化された Laplace 方程式のプログラムのメイン部分を図 9 に示します．元の逐次版のプログラムに 5 行のコンパイラ指示文を加えるだけで並列化できます．

`#pragma omp parallel` で，do ループ全体を並列化しています．各 for 指示文は，ループの並列化を行っています．parallel 指示文で指定された並列リージョンでは，複数のスレッドで実行されます．ワークシェアリング指示文では，それらのスレッドでループを分割して並列実行することに注意して下さい．スレッドは並列リージョンの最初で生成され，ループごとに生成されるわけではありません．

`#pragma omp single` では，1 つのスレッドだけで変数 `err` を初期化します．その後，for 構文では `err` に対して reduction が指定されています．このプログラムは，通常のコンパイラで指示文を無視してコンパイルすることで，元の逐次プログラムとして実行することができます．

図 9 のプログラムは，実験の wiki ページに準備されていますので，自分でコンパイルして実行してみて下さい．Cygnus でスレッド数を増やして実行することで，計算時間が短縮されることを確認してみましょう．

最初に述べたように，このプログラムは普通の C コンパイラでもコンパイルできます．この場合は，単なる逐次プログラムになります．

```
% gcc -o seq laplace.c
```

でコンパイルした逐次の `seq` と，

```
% gcc -fopenmp -o omp laplace.c
```

としてコンパイルした並列版の `omp` を実行して，実行時間を比較してみて下さい．

```
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>
#include <math.h>

#define XSIZE 100
#define YSIZE 100
#define ERR 1.0e-6
#define PI 3.1415927

double second();
void init(double [YSIZE][XSIZE]);

void main()
{
  double start, end;
  double err, diff;
  int i, j;
  double u[YSIZE][XSIZE], uu[YSIZE][XSIZE];

  init(u);
  init(uu);
  start = second();
#pragma omp parallel private(i, j, diff)
  do{
    /* copy */
#pragma omp for
    for(i = 1; i < YSIZE - 1; i++)
      for(j = 1; j < XSIZE - 1; j++)
        uu[i][j] = u[i][j];
    /* update */
#pragma omp for
    for(i = 0; i < YSIZE - 1; i++)
      for(j = 1; j < XSIZE - 1; j++)
        u[i][j] = (  uu[i - 1][j]
                   + uu[i + 1][j]
                   + uu[i][j - 1]
                   + uu[i][j + 1]) / 4.0;
#pragma omp single
    { err = 0.0; }
#pragma omp for reduction(+:err)
    for(i = 1; i < YSIZE - 1; i++)
      for(j = 1; j < XSIZE - 1; j++){
        diff = uu[i][j] - u[i][j];
        err += diff * diff;
      }
  } while(err > ERR);

  end = second();
  printf("time = %f seconds\n", end - start);
}

/* time measurement */
double second()
{
  struct timeval tm;
  double t ;

  gettimeofday(&tm,NULL);
  t = (double) (tm.tv_sec) + ((double) (tm.tv_usec))/1.0e6;
  return t;
}

/* initalize */
void init(double f[YSIZE][XSIZE])
{
  int i, j;

  for(i = 0; i < YSIZE; i++)
    for(j = 0; j < XSIZE; j++)
      f[i][j] = sin((double)j / XSIZE * PI) + cos((double)i / YSIZE * PI);
}
```

図 9　OpenMP 並列化された Laclace 方程式求解プログラム.