# 並列計算＋OpenMP 入門

原田健自

２０１９年５月１５日

## 1　並列計算

### 1.1　アムダールの法則

並列計算の効率に関する一般的な性質としてよく知られているのが、アムダールの法則である。

ある計算アルゴリズムを並列化することを考える。ただし、一般的にはどうしても逐次的に実行しないといけない部分があることが多い。今、実際に、問題を逐次的に実行した時、並列実行可能な部分の実行時間 $P$ と逐次実行が必要な部分の実行時間 $S$ がわかったとする。すると、トータルの計算時間 $T$ は、

$$T = S + P \tag{1}$$

そして、並列に実行できる計算プロセッサを $n$ 個使って、並列実行可能な部分の実行時間を $n$ 倍短縮できたとする。その場合、トータルの計算時間 $T_p$ は、

$$T_p = S + \frac{P}{n} \tag{2}$$

したがって、並列化効率は、

$$\frac{T}{T_p} = \frac{S+P}{S+\frac{P}{n}} = \frac{1}{(1-\alpha)+\frac{\alpha}{n}} \tag{3}$$

ここで、$\alpha = \frac{P}{T}$ で、並列化可能な部分の実行時間の割合である。このことから、例えば、$n$ を無限大にしても、並列化効率には上限があり、

$$\lim_{n\to\infty} \frac{T}{T_p} = \frac{1}{1-\alpha} \tag{4}$$

これらの考察をアムダールの法則と呼ぶ。並列化効率は理想的には $n$ のようになって欲しいが、アムダールの法則によりそうはならず上限がある。実際、図 1 のようにかなり大きな $\alpha$ でも理想とは程遠い並列化効率になってしまう。

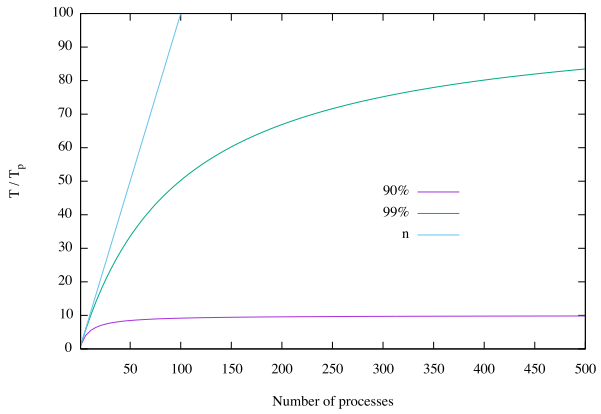


図 1: 並列化効率のアムダールの法則。$\alpha = 0.9, 0.99$ とした。

## 1.2 プロセスとスレッド

コンピュータに同時に実行すべき複数の計算がある時、限れられた計算資源をこれらの実行単位にわりつけ、場合によっては、時間ごとに切り替えるスケジューリングをオペレーションシステムが行なっている。

オペレーションシステムにおいて、これらの計算の実行単位をプロセス、または、スレッドと呼ぶ。その生成や消滅のコストはオーバーヘッドがあるが、もともと、スレッドはプロセスよりも軽めのオーバーヘッドをもつものとして導入された。

一方で、ハードウェアの方では、近年、1 つの CPU に単体で計算可能なコア（プロセッサエレメント：PE) が複数あり、かつ、 1 つのノードに複数の CPU がある場合が一般的になってきている。さらに、これらのノード間を高速のインタフェースで接続してシステム全体が構成されている場合もある。

現在では、通常のノート PC は 1CPU（ただし、複数コア）だが、デスクトップ PC は複数 CPU からなる事が多い。また、複数ノードからなるシステムはクラスター PC と呼ばれ、多くのスーパーコンピュータはこの構成になっている。最後のタイプの計算機は分散メモリ型並列計算機 (図 2) と呼ばれることもある。どちらにしても、どのコンピュータでも並列計算が実行可能な環境になってきている。

並列計算は 1 つの計算を複数のプロセスやスレッドを使って行うことである。そのため、上記のような現在の一般的なハードウェアの下では、システム内の総 PE 数以上にプロセス、スレッドを生成すれば、オペレーションシステムがこれらの切り替えを行ってしまう。したがって、プロセス数+スレッド数を総 PE 数以下に抑えなければコンピュータシステムの限界性能は出せない。

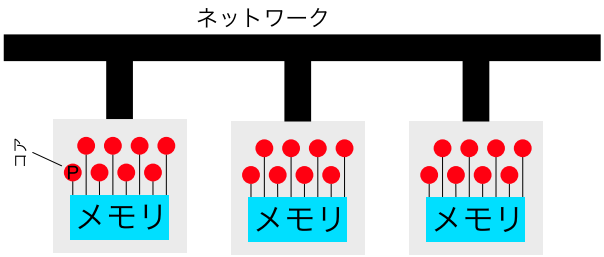


図 2: 分散メモリ型並列計算機。

## 1.3 OpenMP と MPI

一般的に、スレッド並列はメモリーを共有しているスレッド群で実行されるものを指し、プロセス並列は独立なメモリーを持つプロセスで実行されるとする。スレッド並列は共有メモリーを前提とするため、1 ノード内の総 PE 以下のスレッド数だけで実行する必要がある。現在では、以下に述べるように、スレッド並列では OpenMP、プロセス並列では MPI を使ってコードを書くことが多い。

### 1.3.1 OpenMP

C 言語、C++言語、又は、Fortran 言語にディレクティブ文を挿入することで、逐次的な処理を行うプログラムを並列計算のためのプログラムに変更し、共有メモリ型並列計算で実行可能にするための言語拡張。ディレクティブ文は、C 言語では、コンパイラに指示を与えるプラグマ文（#pragma omp で始まる）、fortran 言語では、特別なコメント文（!$omp で始まる）の形式。つまり、ディレックティブ文を理解しないコンパイラからは、普通の逐次実行のコードに見える。

### 1.3.2 Message Passing Interface(MPI)

明示的にメッセージの交換をプログラム中に記述することによって並列計算を行う仕組み。分散メモリ型並列計算機において並列計算を行うためには、OpenMP ではなく、MPI によりプログラムを記述しなければならない。全てのプロセスは同じコードから生成され、並列計算での役割分担は自分のプロセス番号を使って認識する。自分のプロセス番号の取得やメッセージ交換は、MPI ライブラリーの関数を呼ぶ形で記述する。

### 1.3.3 並列化の方針

どちらの方法を用いて並列計算を行えばよいかは、単純な話ではない、少なくとも、1CPU、2 コアのマシンでは、メモリを共有しているため、OpenMP を用いて、2 スレッドでの並列計算が可能である。同時に、MPI を動作させることのできるソフトウェアをインストールしているならば、2 プロセス並列の並列計算を行うことも可能である。又、両者を同時に使うハイブリット並列というのもよく行われる。

# 2 OpenMP 入門

OpenMP は、基本的には、ループ分割に基づく並列計算を容易に記述するための言語拡張である。

## 2.1 ループ分割に基づく並列計算

例えば、ベクトル $\vec{x}, \vec{y}$ の内積は並列計算可能である。

$$S = \sum_{i=1}^{N} x_i \times y_i = s_1 + s_2 + s_3 \tag{5}$$

$$s_1 = \sum_{i=1}^{\frac{1}{3}N} x_i \times y_i, \quad s_2 = \sum_{i=\frac{1}{3}N+1}^{\frac{2}{3}N} x_i \times y_i, \quad s_3 = \sum_{i=\frac{2}{3}N+1}^{N} x_i \times y_i \tag{6}$$

各 $s_i$ の計算は並列に可能。つまり、データ依存性のないループは、分割して並列に実行可能。

ループ分割による並列化が行えるように、データ依存性のあるループを、データ依存性のないループに書き換える。つまり、同じ問題に対する並列計算向きのデータ依存性がない別のアルゴリズムを考えることができると OpenMP を使いこなすことができる。ただし、新アルゴリズムのために、少しの追加計算の増加は許すとする。

この場合、以下のような OpenMP を使った C コードで並列計算が行われる。

```
#define N 100
int main(int argc, char **argv){
double x[N];
double y[N];
...
double sum = 0;
#pragma omp parallel for reduction(+:sum)
for(int i = 0; i < N; ++i)
{
  sum += x[i] * y[i];
}
...
```

以下では、C 言語の場合について、OpenMP の各ディレクティブや指示節の概略を説明する。

## 2.2 parallel ディレクティブ

並列化される範囲（パラレルリージョン）を指定する。例えば、以下のように挿入すれば、ブロック B の部分だけが複数スレッドで同時に実行される。

```
ブロック A
#pragma omp parallel
{
   ブロック B
}
ブロック C
```

つまり、以下のような順番で実行される。

```
ブロック A
ブロック B
        スレッド 0, スレッド 1, ..., スレッド p
ブロック C
```

ただし、スレッド数 $p$ は、環境変数 OMP_NUM_THREADS で指定する。

そして、基本的に全ての変数は各スレッドで共有されている。特に同じ変数に書き込むと衝突が起きるので注意する必要がある。

また、parallel ブロックの開始と終了には、スレッド生成、消滅のためのオーバーヘッドがある上に、（指定がなければ）暗黙に全スレッドがブロックを抜けるまで待つという同期が入る。したがって、連続した parallel ブロックは副作用がなければ 1 つにしたほうが良い。

## 2.3 for ディレクティブ

以下のように分割する for の前に書くと、ループの実行をスレッドに分割して並列に実行する。

```
#paragma omp parallel
{
  ...
  #pragma omp for
  for(int i = 0;i < N; ++i)
  {
    ...
  }
  ...
}
```

この場合、ループを、ループ変数 $i$ に関して、$N/p$ で割った単位ごとに分割して、各スレッドで実行される。したがって、ループ分割が行われても、実行結果がおかしくならないようなループだけにこのディレクティブを使わないといけない。

## 2.4 parallel for ディレクティブ

特定の for 文をパラレルリージョンに指定して、かつ、その for 文の実行を各スレッドに分割する。つまり、parallel ディレクティブ内に for ディレクティブがついた for 文と同値。

### 2.4.1 parallel for ディレクティブの好ましくない使い方

```
#paragma omp parallel for
for(int i = 0;i < N1; ++i)
{
  ...
}
#paragma omp parallel for
for(int i = 0;i < N2; ++i)
{
  ...
}
#paragma omp parallel for
for(int i = 0;i < N3; ++i)
{
  ...
}
```

理由は、parallel for ブロックごとに、スレッド並列実行の準備のためのオーバーヘッドや同期が入るから。

## 2.5 sections ディレクティブ

構造化されたブロックの実行を、各スレッドに分割する。

```
#pragma omp parallel
{
  ...
  #pragma omp sections
  {
    #pragma omp section
    {
      ブロック A
    }
    #pragma omp section
    {
      ブロック B
    }
  }
  ...
}
```

この例では、二つのスレッドが生成され、それぞれがブロック A、B を実行する。実行スレッド数がセクション数よりも少ない場合は、セクション実行が終わったスレッドが、まだ実行されていないセクションを順次実行していく。実行スレッド数のほうが多い場合は、実行をしないスレッドがいる。

## 2.6 single ディレクティブ

パラレルリージョン内で、1 つのスレッドだけに特定のコードを実行させる。ただし、どのスレッドが担当するかはわからない。

```
#pragma omp parallel
{
  ...
  #pragma omp single
  {
    ブロック A
  }
  ...
}
```

この例では、ブロック A がある 1 つのスレッドで実行され、それ以外のスレッドはその実行が終わるまで待つ。

## 2.7 barrier ディレクティブ

スレッド間で同期を取る。barrier ディレクティブは、parallel、for、sections、single ディレクティブの終了時に暗黙的に含まれる。

```
#pragma omp parallel
{
  ブロック A
  #pragma omp barrier
  ブロック B
}
```

全てのスレッドで、ブロック A の実行が終了するまで待つ（同期を取る）。

## 2.8 critical ディレクティブ

スレッド間で排他制御を行う。

```
#pragma omp parallel
{
  ...
  #pragma omp critical
  {
    ブロック A
  }
  ...
}
```

ブロック A の実行は同時にはスレッド 1 つだけ。

## 2.9 atomic ディレクティブ

スレッド間で次の 1 文の排他制御を行う。

```
#pragma omp parallel
{
  ...
  #pragma omp atomic
  x++
  ...
}
```

atomic の後の文を実行できるのは、同時に 1 スレッドのみ。ここでは、変数 x に対する計算式を実行。ただし、x++の他に、++x, x--, --x, さらに、x += y 等の形の “二項演算子=” が指定できる。

## 2.10 reduction 指示節

reduction 演算の実行を指示する。指示節は for ディレクティブ等のディレクティブの後につけて使う。一般に、指示節の種類によって、どのディレクティブと組み合わせることができるかは変わる。

```
reduction(演算子:変数名)
```

演算子の種類は、+, *, -, &, ^, |, &&, ||が使える。

以下が代表的な使い方である。

```
#paragma omp parallel
{
  double total = 0;
  #pragma omp for reduction(+:total)
  for(int i = 0;i < N; ++i)
  {
    total += ary[i];
  }
}
```

まず、スレッドごとに変数 total をプライベートとして生成し（初期値はコピーされる)、最後に、それらのプライベート変数 total の和をとって、共有している変数 total に入れるという風に実行される。

### 2.10.1 reduction 指示節における注意

C 言語では、reduction 演算に指定できる対象は変数のみだが、Fortran では、reduction 演算に指定できる対象は変数と配列の両方が許される。

## 2.11 private 指示節

変数をプライベート化する。

```
private(変数名のリスト)
```

この指示節も for ディレクティブなどと一緒に使える。

```
#paragma omp parallel
{
  double tmp = 0;
  #pragma omp for private(tmp)
  for(int i = 0;i < N; ++i)
  {
    tmp = ...
    ary[i] = f(tmp)
  }
}
```

変数 tmp は各スレッドごとにプライベート化されるので、ループ文内で代入しても、他のスレッドと衝突することはない。

### 2.11.1 データスコープの指定

これ以外にも、データの有効範囲を指定する様々な指示節がある。

**shared(list)**
: list で指定された変数を、全てのスレッドで共有する。

**firstprivate(list)**
: list で指定された変数について、構文前に存在している元の実体の値で初期化した上で、スレッド固有な変数にする。

**lastprivate(list)**
: 領域の最後の作業単位を実行するスレッドに属している対応するスレッドのプライベートの変数値が外部変数に引き継がれる。sections 構造の場合、最後の作業単位は最後に現れる section である。for 文では、ループのインデックス変数で生成される 1 つ 1 つの計算の塊を作業単位とする。

## 2.12 nowait 指示節

`#pragma omp for` の後には、暗黙の同期が含まれる。しかし、nowait 指示節をつけると、その同期を取らなくなる。

```
#pragma omp parallel
{
  #pragma omp for nowait
  for(int i = 0; i < N1; ++i)
  {   ...
  }
  #pragma omp for
  for(int i = 0; i < N2; ++i)
  {   ...
  }
}
```

上と下の for に依存関係がない場合には、上の`#pragma omp for` の後に、nowait 指示節を入れると、上のループの終了時に同期を取らないので、早く終わったスレッドはすぐに下のループに実行に取りかかれる。下の for に nowait と書いても意味はない。parallel を抜けるときにも、暗黙の同期が含まれるからである。`#pragma omp single` の後にも、暗黙の同期が含まれる。しかし、

```
#pragma omp parallel
{
  #pragma omp single nowait
  {   ...
  }
}
```

とすると、single のブロックを担当するスレッド以外は、single のブロックを飛ばして、先に進むようになる。

## 2.13 OpenMP の実行環境関数

以下の実行環境ルーチンの関数を使用するときには、ヘッダーファイル omp.h のインクルードが必要。

**omp_get_num_threads()**
　スレッド数を取得。

**omp_set_num_threads(int num)**
スレッド数を変更。

**omp_get_thread_num()**
スレッド ID を取得。

**omp_get_max_threads()**
最大のスレッド数を返す。パラレルリージョンの外でも呼び出せる。

**omp_get_wtime()**
時刻を返す。単位は秒。

### 2.13.1 omp_get_max_threads() を利用する上での注意

4CPU × 4 コア=16 コアの共有メモリ型並列計算機で並列計算を行う場合、各コア毎に、100 個の倍精度浮動小数点数を working area として必要とすることを想定する。逐次計算領域内で、
`malloc(sizeof(double)*100*omp_get_max_threads())`
というように、プログラムを書いてはいけない。並列計算領域内で、個別に、それぞれのコアが `malloc(sizeof(double)*100)` として、領域を確保すること。そして、working area を使い終わったら、free すること。

## 2.14 #pragma omp for の本当の価値

`#pragma omp for` の後には、次のような語句を続けられる。

```
#pragma omp for schedule(dynamic,chunk)
```

通常、chunk は 1 を指定する。例えば、

```
#pragma omp parallel
{
  #pragma omp for
  for(int i = 0; i < N; ++i)
  {   ...
  }
}
```

for の後に何も書かない場合には、3 スレッドで実行する場合の仕事の割り振りは、スレッド番号 0 は 0 から $N/3$ 未満を担当、スレッド番号 1 は $N/3$ から $2N/3$ 未満を担当、スレッド番号 2 は $2N/3$ から $N$ 未満を担当する。

ちょっと考えると、$i$ が変化する毎に仕事の内容が変化するため、$i$ 毎に計算時間がばらつくほうが自然なように感じるが、世の中のプログラムのほとんどが $i$ が変化するにも関わらず $i$ 毎の実行時間が等しいものが多い。よって、多くのプログラムでは for の後に何も書かない場合の動作のように、仕事の配分は均等配分でよい。

しかし、どこの世界にも例外というものは必ず存在する。例えば、$i$ が変化すると、問題の性質から $i$ 毎の実行時間がバラつく場合、入力データにより $i$ 毎の実行時間が変化し実行してみるまで実行時間の予測が立たない場合など。そういった場合に、

```
#pragma omp for schedule(dynamic,1)
```

とすると、始めに、各スレッドはまだ処理が終わっていない仕事を担当して取り掛かる。担当した仕事が速く終ったスレッドは $i$ の残っている番号の仕事に取りかかる。以降、仕事が終わるスレッドが出るたびに、残っている仕事のキューから仕事を取り出して実行する。chunk は、$i$ の取りうる値の何個を 1 塊の仕事と考えるかを指定する。

## 2.15 並列化の実際

OpenMP による並列化は遅くはならないと思うかもしれないが、しばしば、返って遅くなることがある。

その原因の一つは、スレッド生成のオーバーヘッドである。スレッドはプロセスよりも生成は軽いと言っても、やはり、生成にはオーバーヘッドがある。そのため、ループ分割によって実行される計算が軽いとスレッド生成のオーバーヘッドの方が実行時間がかかってしまうことがよくある。

その他の遅くなる原因の一つは、共有変数に対する衝突である。OpenMP はメモリー共有なので、同じ変数を同時に見に行く事もあるし、同じ変数に同時に書き込もうとする事もある。そのような共有変数に対する衝突が起きると、後のスレッドが待たされることがあり、効率的な並列化が難しくなることがある。

効率の問題以外にも OpenMP による気をつけなければならない問題点は、スレッドで並列実行しても問題が起きないアルゴリズムもしくは対策が取られているかと言うことである。そのようなコードをスレッドセーフであると呼ぶ。ただ、スレッドセーフでも内部状態を共有していると、共有変数の衝突も起きてしまうので、効率的な並列化が難しくなることがある。

例えば、乱数生成ルーチンは、通常、乱数生成の為の内部状態を必要とし、それを保持している。この内部状態変数への衝突を避けないとスレッドセーフにはならない。しかし、例え、スレッドセーフにしても、つまり、衝突が起きてもスレッド間で排他的に書き込みを行うようにしてしまうと、並列化

効率は下がる。そのため、スレッド並列化効率を上げるためには、スレッド同志で独立に内部状態の管理を行う必要がある。以下の例は、乱数生成ルーチン xor128 の内部状態を乱数生成ルーチンに明示的に渡すことでスレッドセーフにしたものである。

```
void init_xor128(uint32_t s0, uint32_t *s) {
  for (uint32_t i = 0; i < 4; ++i){
    s0 = 1812433253U * ( s0 ^ ( s0 >> 30 ) ) + i;
    s[i] = s0;
  }
}

uint32_t xor128(uint32_t *s) {
  uint32_t t = ( s[0] ^ ( s[0] << 11 ) );
  s[0] = s[1];
  s[1] = s[2];
  s[2] = s[3];
  s[3] = ( s[3] ^ ( s[3] >> 19 ) ) ^ ( t ^ ( t >> 8 ) )
  return s[3];
}

double ur(uint32_t *s){
  return ((double) xor128(s))/4294967296e0;
}
```

以上のことを考えて、スレッド並列化の方針を決め、コードを書く必要がある。例えば、モンテカルロ法ではサンプル生成を繰り返すが、２つの並列化の方針が立てられる。１番目は、サンプル１つの生成の並列化。ただし、この場合、並列化効率を上げるには、サンプル１つの生成がオーバーヘッドよりも十分実行時間がかかることが必要である。２番目は、モンテカルロ法で必要とするサンプルリングを分割して行う並列化。これは、モンテカルロ法がサンプルの平均値で期待値を計算すると性質を生かしたものである。こちらも、各スレッドでのサンプル生成に十分実行時間がかかれば並列化効率が上がる。

以上のようにアルゴリズム、計算の特性を考慮して、並列化の方針を立てることが重要である。