# 古典場のシミュレーションにおける OpenMP の実装例

平松尚志

京都大学 基礎物理学研究所

**概要**

スレッド並列のライブラリである OpenMP の簡単な紹介を行い、宇宙論分野でしばしば行われる古典的実スカラー場のシミュレーションを例に、OpenMP の実装方法の違いによって、並列計算の実行時間に差がでることを示す。

## 1 序

これまでの並列動作するコードというのは、スーパーコンピューターや PC クラスターといった、非常に多くのノード[*1]で構成される巨大な計算機で実行することを前提とし、ノード間のデータのやりとりを制御するライブラリ MPI (Message Passing Interface)[*2]を用いて、プログラマー自身がコードの中で明示的にノード間通信を制御することで動作していた。MPI の取扱いは若干難しく、しかも大規模な計算機がなければそもそも実行できないという点で、専門家でも並列計算というのはハードルの高い存在であった。

しかし、一般向け CPU のクロック数が排熱処理の問題で頭打ちになった 2000 年代前半ころから、クロック数はではなく、CPU コアの数を増やすという流れが起き始めた。この流れの中で、2005 年に Intel, AMD が相次いでマルチコア CPU を発表して以来[*3]、一般向けのパソコンでも複数の CPU （実際は CPU コア）が存在する環境が急速に普及した。その結果、専門家の間でも、自分の持っているシミュレーションコードを並列化し、それを実行するということが比較的簡単に行えるようになってきた。

このような、複数の CPU コアが存在するようなコンピュータで、CPU コアの並列実行を制御するライブラリが OpenMP[*4] である [1]。OpenMP は、Linux や、Unix 系 OS である MacOS 等に標準で付属している GCC (GNU Compiler Collection) に含まれている C/C++ コンパイラや fortran コンパイラに初めから含まれているため[*5]、別途、ソフトウェアを導入する必要はない。また、Linux ならば無償で入手できる Intel C/C++/Fortran Compiler にも同梱されている[*6]。

本稿では、簡単な例を用いてこの OpenMP の 2 つの実装方法を紹介し、実装方法の違いで実行速度に違いが出ることを示す。次の 2 節では OpenMP の概要を、3 節では使用するモデル計算を

---

[*1] 物理的には CPU + メモリ + HDD + マザーボード + ノード間通信設備 (LAN) の一式。

[*2] ノード間の通信のことを「メッセージのやりとり」と捉える。

[*3] 当初は複数の CPU ダイを接続して 1 つのパッケージに収めるという形式だったが、最近では 1 つの CPU ダイ上に本当に複数の CPU コアが搭載されている形式のものが増えている。コア間のデータのやりとりやキャッシュメモリの共有などの面で、一般的には後者の方が有利。

[*4] 執筆時の最新は OpenMP v3.1 が利用可能。なお、ライブラリとはいっても、通常のライブラリのように関数群を集めたものというわけではない。

[*5] ただし GCC ver.4.2 以降。執筆時点の最新は ver.4.6.1。

[*6] ver.9 以降なら AMD の CPU にも対応。執筆時点の最新は ver.12.1。

説明し、4 節で 2 つの実装方法を紹介する。5 節では実際のテスト計算の結果を示し、6 節で本稿をまとめる。

# 2 OpenMP

## 2.1 概要

OpenMP は、共有メモリ型の並列計算機で使用される並列化ライブラリである。ライブラリとは言っても、ユーザーは特定の指示文（C/C++ ならば`#pragma omp` 指示文、fortran ならば`!\$OMP` 文）をわずかに挿入するだけで並列化を実現できるのが特徴で、MPI のように、明示的にメッセージ（データ）のやりとりを記述するわけではない。

ここで、共有メモリ型計算機について触れておく。「共有メモリ型」とは、複数の CPU もしくは CPU コアが、同じマザーボード上に設置されているメモリモジュールを共有して使用しているシステムを指す。冒頭にも述べたが、近年、個人が日常的に使用するパソコンにおいても、４つ程の CPU コアを搭載している CPU が普及しつつある[*7]。この場合、マザーボード上には４つのコアを持つ CPU が１つ、さらにいくつかのメモリモジュールが搭載されている。この形態を共有メモリ型計算機と言う。よりハイエンドのものになると、１つのマザーボード上に複数の CPU が搭載されることもある[*8]。この形態も共有メモリ型となる[*9]。

一方で、マザーボード同士を外部の高速ネットワークで接続し、マザーボード間でデータのやり取りが行われるような形態の計算機を「分散メモリ型」と呼ぶ。この場合、あるノードの CPU が別のノードのメモリに入っているデータを参照する、というような状況が生まれる。こういったデータのやり取りは MPI で記述することになっている。

ソフトウェア上では処理の単位として「プロセス」と「スレッド」という言葉が用いられる。一般に、１つのプログラム（タスクと呼ばれる）は複数のプロセスで構成されている（通常は 1 プロセスで実行される）。分散メモリ型では、MPI 並列化したプログラムが実行されると、各ノードで MPI プロセスと呼ばれるものが立ち上がる。各プロセスは固有のメモリ空間を持つため、お互いを直接参照し合えないため、MPI でデータの受け渡しを明示的に示す必要がある[*10]。

さらに、１つのプロセスは一般に複数のスレッドで構成されている（通常は 1 プロセスに 1 スレッドが割り当てられる）。１つのプロセス内で生まれた各スレッドは、それらが所属しているプロセスの所持しているメモリ空間を共有しているため、スレッド間のデータはメモリ上で直接参照し合うことができる。このスレッドの生成・消滅を OpenMP で記述しているのである。複数のス

[*7] Intel Core i7 および i5 の上位、AMD Phenom/Athlon II X4 など。

[*8] Intel Xeon シリーズや AMD Opteron シリーズなど。また、4 CPU を越えるシステムでは、マザーボードに CPU カードが複数枚接続され、そのカード上に CPU が搭載される形態もある。

[*9] 執筆時点でもっとも CPU コアの総数が多い共有メモリ型の計算機は、Xeon E7-8800 シリーズを 8 基搭載した 80 コアモデル。

[*10] 共有メモリ型計算機でも MPI 並列化したコードは動かせる。その場合も各プロセスのメモリ空間は独立であるため、データの移動は MPI で記述する必要がある。

レッドは並列処理の間だけ存在しており、これが終了すると再び１つのスレッドに戻る[*11]。これが OpenMP による並列動作の流れとなる。図１に、プロセス、スレッド、OpenMP、MPI の関係を簡単に示した。

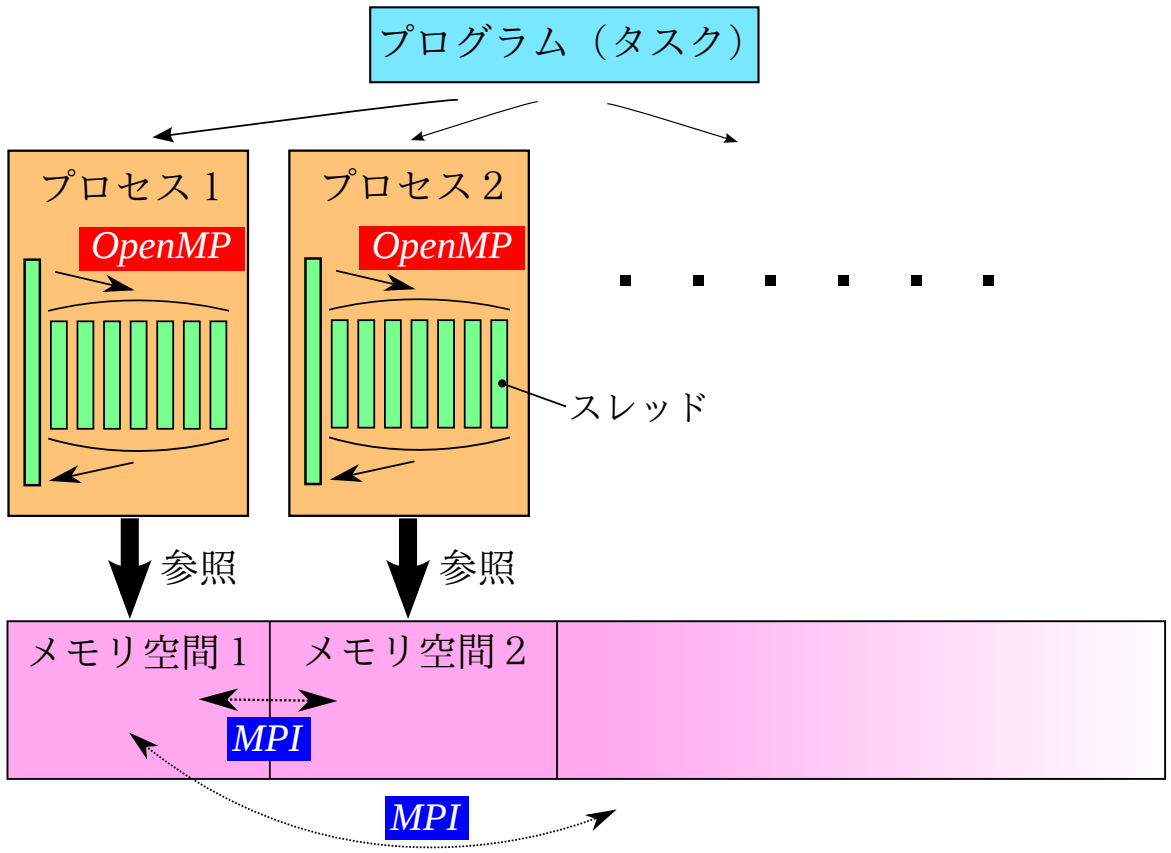


図１　実際の計算はスレッド内で行われる。MPI がメモリ空間の間のデータのやりとりを記述するのに対して、OpenMP はスレッドの生成と消滅を記述する。

さて、共有メモリ型の場合、分散メモリ型と比較していくつかの有利な点がある。一つは、外部ネットワークを使用しないことだ。一般に、ネットワークカードを経由して接続される外部のネットワーク（LAN）は、CPU とメモリの間の通信よりも遅いため、InfiniBand 等の高速なネットワークを使用しないと、データのやり取りがボトルネックになる可能性が高い。また、MPI で並列化すると、各プロセスがメモリ空間を共有していないため、たとえ同一ノード内で実行したとしてもデータの移動にオーバーヘッドが生じる。一方で共有メモリ型（OpenMP）の場合、データのやり取りは物理的にはすべてマザーボード上の CPU-メモリ間のリンクのみで行われ、さらにメモリ空間も共有するため、データの相互参照時に余計なオーバーヘッドは生じない（ただしスレッドの生成・消滅にオーバーヘッドが生じる）。

二つ目の有利な点は、コードの並列化のハードルが低いことだ。先ほども述べたように、OpenMP

[*11] このような実行形態は Fork-Join モデルと呼ばれる。

を使った並列化は、わずか数行の指示文の挿入だけで行える。一方の MPI は、ノード間のデータのやり取りを事細かに記述することが必要で、初心者には若干ハードルが高い。

加えて、昨今の CPU 事情の変化も考慮に入れるべきだろう。前にも触れたように、CPU コアの増加が一つのトレンドとなっている。本稿執筆時の Intel 系の CPU では、Xeon 7500 シリーズが 8 つの CPU コアを持っており、さらに最大 10 コアを持つ Xeon E7 シリーズが登場した。AMD 系では、Opteron 6100 (Magny-Cours) シリーズが最大 12 コアを搭載しており、2-way 仕様のマザーボードを使用すれば、最大 24 コアのマシンが通常のパソコンより一回り大きい匡体（SSI-EEB など）で実現可能になっている。このような事情から、HPC の分野でも「多コア＋多ノード」の構成が主流になってきており、計算機の性能を最大限に引き出すには、OpenMP と MPI を融合させたプログラミングが必要となってきている。

## 2.2 簡単な例

本稿では C++ における OpenMP による並列化を行う。最低限必要な指示文は以下の通りである。

- `#pragma omp parallel`：並列化の指示（スレッドの生成）
- `#pragma omp for`　　：for ループのワークシェアリング[*12]
- `#pragma omp barrier`　：明示的な同期の指示

この中で、上 2 つは `#pragma omp parallel for` とまとめて 1 行で書くこともできる。

例として、以下のコードを考える。

並列化していないコード

```
for( int j=0; j<1000; j++ ) {
  for( int i=0; i<1000; i++ ) {
    （i,j を使った処理）
  }
}
```

このコードの for ループを OpenMP で並列化させるには、

for 指示文でループを並列化

```
#pragma omp parallel for
for( int j=0; j<1000; j++ ) {
  for( int i=0; i<1000; i++ ) {
    （i,j を使った処理）
  }
}
```

とする（以後、並列化のために追加した部分を赤く表示する）。これで、1000 回繰り返す $j$ のループが、スレッドの数で分割されて、各々が処理を並行して行う。たとえば、10 スレッドを使用す

[*12] ループをブロックに分割して分担作業することをワークシェアリングと呼ぶ。

る場合、それぞれのスレッドが 100 個ずつ受け持ち、同時に処理を行う[*13]。そして、$j$ のループが終わると並列処理に使われたスレッドが消滅する。なお、スレッドの生成と消滅にはそれなりの時間がかかるため、上記のような多重ループの場合、一番外のループのみを並列化するのが普通である。また、この例は以下のようにも書くことができる。

指示文を分割して記述

```
#pragma omp parallel
{
  #pragma omp for
  for( int j=0; j<1000; j++ ) {
    for( int i=0; i<1000; i++ ) {
      (i,j を使った処理)
    }
  }
}
```

今の例では並列化したいループが 1 つしかないが、並列化したいループが複数ある場合は、全体を {} でブロック化して、まとめて parallel 指示文を指定することもできる[*14]。このブロックの中では最初に生成されたスレッドがブロックの終端まで生き続けるため、先ほど述べたようなスレッドの立ち上げ・後始末にかかるオーバーヘッドを最小限に抑えることができるので、上記のような記法は、複数の for ループを並列化したい場合などには効果的になる場合がある。

上の for 指示文で並列化したループの最後には、スレッド間の同期（すべてのスレッドが計算を終えるまで先に進まない）が自動的に取られるようになっている（暗黙のバリア）。OpenMP でも、スレッド間の同期には注意を払わなければならない。もし正しく同期が取られていないと、実行するたびに答えが変わるなどの悪影響が現れる。上の例では必要ないので現れていないが、明示的にスレッド間の同期を取る指示文が barrier 指示文である。これは 4 節で使われる。

また、4.2 節紹介するが、インクルードファイル omp.h をインポートすれば、生成するスレッドの総数をあらかじめ指定する関数 omp_set_num_threads(n) や、並列ブロック内でスレッドの番号を得る関数 omp_get_thread_num() などが使用できる。これらの関数は頻繁に利用するので、OpenMP を本格的に利用する場合には、使い方を調べておく必要がある。また、上で挙げた以外にも様々な指示文が存在し、スレッドの動作をかなりきめ細かく設定できるようになってい るので、合わせて目を通しておいてもらいたい。

# 3 モデル計算

宇宙論分野では、様々な場面で古典場のシミュレーションが行われている。例えば、極めて初期の宇宙に対するプローブとなることが期待されているドメインウォールや宇宙ひもといった位相欠陥や、超対称標準模型で現れる Q-ball などのシミュレーションには、3 次元の複素スカラー場な

[*13] ループの分割の仕方も指定できる。

[*14] parallel ブロックの中に複数の for 指示文が入る形になる。

どのシミュレーションが欠かせない。それぞれの具体的な事例は [2, 3, 4] などを参照されたい。これらの研究では、現在開発中の汎用双曲型偏微分方程式解法コードジェネレータ Freja から生成されたシミュレーションコードを用いている。

ここでは簡単のため、3 次元の古典場のシミュレーションの例として、ポテンシャルの無い実スカラー場 $\phi(t,x,y,z)$ を取り上げる [5]。発展方程式は、

$$\frac{\partial^2\phi}{\partial t^2} - \nabla^2\phi = 0. \tag{1}$$

で与えられる。この問題を、2 次の中心差分と 2 次の Runge-Kutta 法 [6] で解くことを考える。そのために、まずは補助変数 $\chi$ を導入し、方程式を以下のように 2 つの時間に関する一階微分方程式に分解する。

$$\frac{\partial\mathbf{y}}{\partial t} = \mathbf{f}(t,\mathbf{y}), \quad \mathbf{y}(t) = \begin{pmatrix}\phi\\ \chi\end{pmatrix}, \quad \mathbf{f}(t,\mathbf{y}) = \begin{pmatrix}\chi\\ \nabla^2\phi\end{pmatrix}. \tag{2}$$

中心差分は空間微分 $\nabla^2\phi$ の部分に適用し、

$$\nabla^2\phi(x_i,y_j,z_k) \approx \frac{\phi_{i-1,j,k}+\phi_{i+1,j,k}+\phi_{i,j-1,k}+\phi_{i,j+1,k}+\phi_{i,j,k-1}+\phi_{i,j,k+1}-6\phi_{i,j,k}}{h^2}, \tag{3}$$

という標準的な公式を使用する。ここで $h$ はグリッドの間隔（一様なグリッドを使用）、$\phi_{i,j,k}$ は $\phi(x_i,y_j,z_k)$ を省略表記したものである。

2 次の Runge-Kutta 法（修正オイラー法）は、以下のように定義される。

$$\mathbf{y}(t_{n+1}) = \mathbf{y}(t_n) + \mathbf{k}_2\Delta t, \tag{4}$$

$$\begin{cases}\mathbf{k}_1 = \mathbf{f}(t_n,\mathbf{y}(t_n)),\\ \mathbf{k}_2 = \mathbf{f}(t_n+\Delta t/2,\mathbf{y}(t_n)+\mathbf{k}_1\Delta t/2).\end{cases} \tag{5}$$

ただし、空間の添字 $i,j,k$ は略した。

これを C/C++ で実装すると、主要な部分は大まかに以下のようになる。

メインループの実装例

```
for( int n=0; n<nmax; n++ ) {
  double t = dt*n;
  evaluate(k1,phi,t);
  rkupdate(phi,k1,.5*dt);
  evaluate(k2,phi,t+dt*.5);
  rkupdate(phi,k2,*dt);
}
```

ここで、`nmax` は総ステップ数、`dt` は時間刻みを表し、`phi` と `k1`, `k2` は全空間のデータを保持する配列として確保されているものとする。なお、実際の問題には、ここに挙げた以外にも様々な引数を必要とする場合があるが、ここでは説明の都合上、必要最低限のもののみ考える。また、`phi` の初期条件や配列の後片付け（メモリの解放）なども省略する。

# 4　実装

ここでは以下の２種類の実装の仕方を比較する。

- 方法１：手間をかけたくないが、実行速度が犠牲になる
- 方法２：手間はかかるが、実行速度のロスを抑える

一つ目は for ループが現れる度に `parallel for` 指示文を明示する方法で、ユーザーが追加すべきコードの量は少ない。しかしその分、ループの出入り口に到達するたびにスレッドの生成・消滅が行われ、実行速度を幾分犠牲にすることになる。二つ目は、`parallel` 指示文はコード内で一度だけ明示し、for ループの分割、および同期のタイミングをユーザーが直接指定する方法である。この方法はユーザーが追加するコード量が比較的多い上に、同期のタイミングを間違えると、間違った答えを出力する場合がある。しかしその労力の分、シミュレーションの規模によっては計算機の性能をフルに引き出すことも可能な方法になっている。

それぞれの実行時のスレッドのイメージを図示すると、図 2 のようになる。

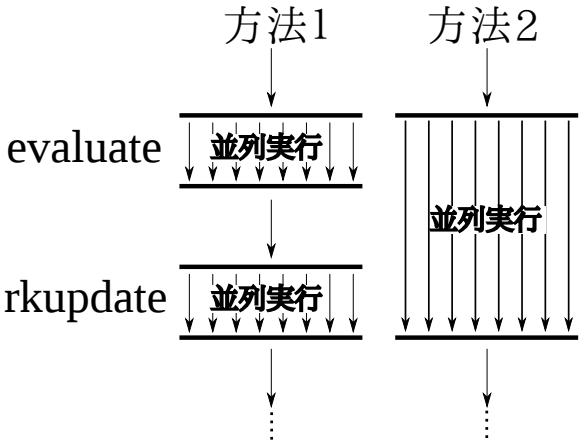


図 2　矢印が各スレッドを、太線がスレッドの生成・消滅のタイミングを表す。

図 2 では、プログラムは上から下へ向かって実行され、各矢印はその時点で動作しているスレッドを表し、太線はスレッドの生成と消滅のタイミングを表している。この図が示すように、負荷の大きいスレッドの生成・消滅作業の回数が、方法２では最小限に抑えられていることがわかる[*15]。

## 4.1　方法１：手間をかけない方法

方法１を実装したコードは以下の通り。

[*15] なお、拙作の Freja では方法２が実際に使われている。

手間をかけない方法

```
void evaluate(省略) {

  #pragma omp parallel for
  for( int k=0; k<N; k++ ) { // z
    for( int j=0; j<N; j++ ) { // y
      for( int i=0; i<N; i++ ) { // x
        (k を計算する処理)
      }
    }
  }
}

void rkupdate(省略) {

  #pragma omp parallel for
  for( int k=0; k<N; k++ ) { // z
    for( int j=0; j<N; j++ ) { // y
      for( int i=0; i<N; i++ ) { // x
        (次のステップの phi を計算する処理)
      }
    }
  }
}

int main {
(中略)
  for( int n=0; n<nmax; n++ ) { // time step
    double t = dt*n;
    evaluate(k1,phi,t);
    rkupdate(phi,k1,.5*dt);
    evaluate(k2,phi,t+dt*.5);
    rkupdate(phi,k2,dt);
  }
(中略)
  return 0;
}
```

**■概略**

- parallel 指示文は rkupdate と evaluate 内の全ての for ループに対して記述する。同時に for 指示文で for ループをワークシェアリングするように指示する。

**■長所**

- ユーザーが新たに追加すべきコード量が非常に少ない（この例の場合わずか 2 行)。
- 同期はしかるべき箇所（この場合 k ループの最後）で自動的に行われるため、ユーザーが気にする必要はない

**■短所**

- 負荷が非常に大きいスレッドの生成と消滅を頻繁に行うため、並列させるスレッドの数を増やしても、実行時間があまり短くならない場合が多い。

## 4.2 方法２：手間をかける方法

方法２を実装したコードは以下の通り[*16]。

手間をかける方法

```
#include <omp.h>

void evaluate(int thread_num, int nthreads, 省略) {

  int kstart = N*thread_num/nthreads;
  int kend = N*(thread_num+1)/nthreads;
  for( int k=kstart; k<kend; k++ ) { // z
    for( int j=0; j<N; j++ ) { // y
      for( int i=0; i<N; i++ ) { // x
        (k を計算する処理)
      }
    }
  }
}

void rkupdate(int thread_num, int nthreads, 省略) {

  int kstart = N*thread_num/nthreads;
  int kend = N*(thread_num+1)/nthreads;
  for( int k=kstart; k<kend; k++ ) { // z
    for( int j=0; j<N; j++ ) { // y
      for( int i=0; i<N; i++ ) { // x
        (次のステップの phi を計算する処理)
      }
    }
  }
}

int main {
(中略)
  int nthreads = 24;

  #pragma omp parallel
  {
    int thread_num = omp_get_thread_num();
    for( int n=0; n<nmax; n++ ) { // time step
      double t = dt*n;
      evaluate(thread_num,nthreads,k1,phi,t);
      #pragma omp barrier
      rkupdate(thread_num,nthreads,phi,k1,.5*dt);
      #pragma omp barrier
      evaluate(thread_num,nthreads,k2,phi,t+dt*.5);
      #pragma omp barrier
      rkupdate(thread_num,nthreads,phi,k2,dt);
```

[*16] ちなみに、この実装方法は筆者自身の試行錯誤の結果であるため、これより良い方法が存在するかもしれない。

```
      #pragma omp barrier
    }
  }
(中略)
  return 0;
}
```

**■概略**

- parallel 指示文は最初の一回だけ。
- 同期は barrier 指示文を使って明示する。
- omp.h にある omp_get_thread_num() 関数を使ってスレッド番号 thread_num を取得し、それぞれのスレッドが rkupdate と evaluate を実行する[*17]。
- 各スレッドの担当領域 kstart, kend を決める。

**■長所**

- スレッド生成・消滅のためのオーバーヘッドを最小限に抑えられるため、並列化効率が高まる場合が多い。

**■短所**

- ユーザー自身が、スレッド番号から各スレッドの担当領域を明示する必要がある。
- ユーザー自身が同期のタイミングを明示する必要がある。ただし不用意に同期を取りすぎると速度低下に繋がるので注意する[*18]。

ちなみにこの例では総スレッド数 nthreads を 24 で固定しているが、現在の総スレッド数は omp.h にある omp_get_num_threads() 関数で取得でき、さらに、omp_set_num_threads(n) 関数で、総スレッド数を n に設定できる。方法 1 でも同じことができる。なお、総スレッド数は、OS 側で制限をかけない限り、そのシステムに存在している CPU コアの数（ハイパースレッディングが有効の場合はさらに倍）がデフォルト値になっている。

# 5 計算時間の比較

テスト計算の実行環境は以下の通り。

**CPU** AMD Opteron 6172 (Magny-Cours, 12 cores) × 2

[*17] parallel ブロック内で定義された変数 int thread_num は各スレッドで独立に扱われる。一方で、ブロックの外で定義された変数は、すべてのスレッドで共有される。これらを明示的に表す指示句 private(), shared() もある。

[*18] 一見すると evaluate の直後の barrier は必要ないように見える。しかし、方程式にラプラシアンが入っており、一部の k は隣のスレッドの管轄である phi を必要とする。したがって、その phi が予期しないタイミングで更新されてしまうのを防ぐ必要があるため、この barrier は外せない。

**memory** 64GB(8GB×8)
**MB** ASUS KGPE-D16 (2CPUs, 8×2 mem slots)
**HDD** 7TB

コンパイラは Intel C++ for Linux ver. 11.1 を使用した。コンパイルの方法は以下の通り。

```
$ icpc -O3 -xHost -static-intel -openmp test.cpp -o test
```

OpenMP を使用するには `-openmp` オプションが必要になる[*19]。また、実行時には `numactl` を使ってメモリーアロケーションの方法を明示した。

```
$ numactl --interleave=all ./test
```

計算は、グリッド数 $128^3, 256^3, 512^3$ に対してそれぞれ 3 回行い、各々の CPU 時間の平均を実行時間とした。方法 1 の実行時間を $T_1$、方法 2 の実行時間を $T_2$ とし、それぞれの実行時間の比 $T_2/T_1$ を図 3（左）に図示した。全ての場合において、方法 2 の方が実行時間が半分以下になっているのが分かる。特に、スレッド数が増えると 1/3 程度まで差が広がり、$N = 128$ の場合は 1/4 まで実行時間が短縮できている。

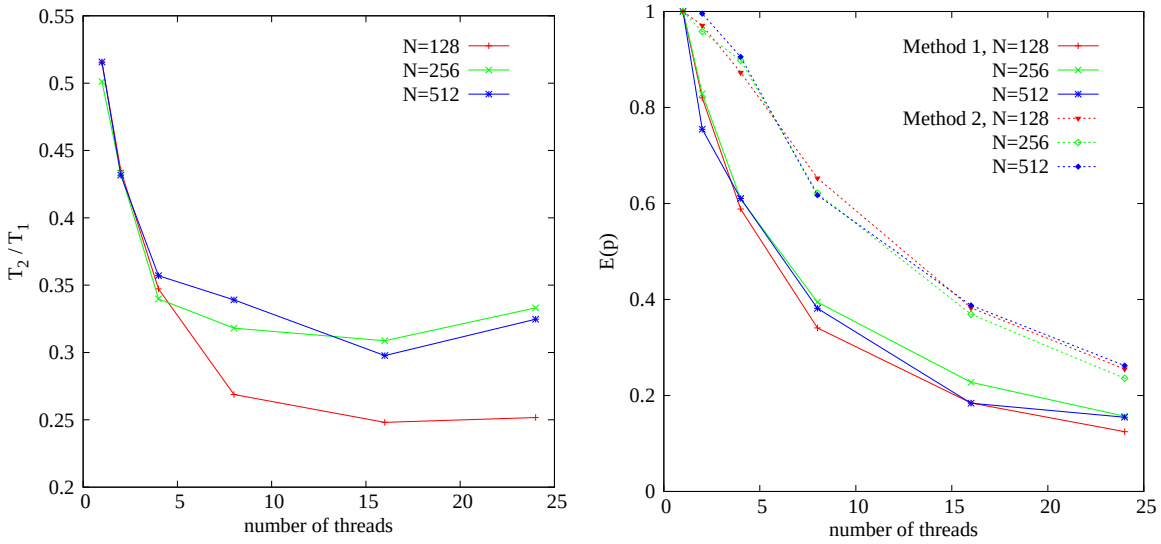


図 3　左：方法 1 の実行時間（$T_1$）と方法 2 の実行時間（$T_2$）の比。右：並列化効率 $E(p)$。横軸はスレッド数。

図 3 の右は、並列化効率 $E(p)$ を示している。$E(p)$ は、以下のように定義される。まずは、$p$ スレッドを使用したときの計算時間を $T(p)$ とすると、シングルスレッド（$p = 1$）の場合に対してどれだけ計算速度が向上したかを示す指標として、$S(p) \equiv T(1)/T(p)$ という量がしばしば使用され

[*19] GCC の場合は `-fopenmp`。

| #. of | $N = 128$ | | $N = 256$ | | $N = 512$ | |
|---|---|---|---|---|---|---|
| threads | 1 | 2 | 1 | 2 | 1 | 2 |
| 1 | 33.93 | 17.48 | 278.03 | 139.34 | 2228.40 | 1149.22 |
| 2 | 20.68 | 9.00 | 167.85 | 72.70 | 1388.41 | 599.33 |
| 4 | 14.41 | 5.00 | 114.15 | 38.79 | 908.69 | 324.38 |
| 8 | 12.45 | 3.35 | 88.14 | 28.04 | 705.94 | 239.35 |
| 16 | 11.51 | 2.86 | 77.14 | 23.59 | 648.70 | 193.11 |
| 24 | 11.35 | 2.86 | 73.94 | 24.63 | 590.48 | 191.73 |

表 1　計算時間。単位は秒。

る。例えば、2 スレッドで実行して計算時間がちょうど半分になったとすると $S(2) = 2$ になる。この量を使用したスレッド数で割ったものが、並列化効率 $E(p) = S(p)/p$ である。この量は、p スレッドで実行した場合、各スレッドがどれだけ効率良く使用されたかを示している。例えば先の例では $E(2) = S(2)/2 = 1$ なので、もっとも効率よく並列計算が行われたことを示している。一方で、$S(2) = 1$、すなわち 2 スレッドで実行しても計算時間がまったく向上しなかった場合は、$E(2) = 0.5$ となって、各スレッドは 50% しか使われていないことを示している。

図 3 の右は、4 スレッドで比較すると、方法 2 の方は $E(p) \approx 0.9$ と高い効率を示しているにも関わらず、方法 1 の方は $E(p) \approx 0.6$ まで効率が落ちている。つまり、方法 2 ではシングルスレッド時に比べて 3.6 倍の速度向上があった一方で、方法 1 では 2.4 倍に止まっていることを示している。

# 6　まとめと考察

今回は、並列化が比較的簡単に行える OpenMP を使用し、その実装方法として 2 つの例を挙げた。たしかに、OpenMP は誰でも簡単に実装できるように設計されてはいるが、計算機の性能を十分に発揮させるには、やはりそれなりの労力が必要であることを示した。

今回示した実装方法は、例えば有名な高速フーリエ変換ライブラリ FFTW [7] に対しても適用できる。FFTW では、スレッド並列のインターフェース完備しているが、これは今回の方法 1 に対応する方式を採用しているため、スレッドが増えると並列化効率があまり良くない。そこで、3 次元のフーリエ変換を 1 次元のフーリエ変換に分解し、それを方法 2 に従って並列化したところ計算時間が改善した。

また今回使用した例では、効率の面に関してはそれほど注力していないため、図 3 の右が示しているように、8 スレッド以上ではあまり並列化効率が良くない結果を得た。シミュレーションの種類や規模にもよるが、OpenMP の場合には、キャッシュミスの大きさなどが計算時間を大きく左右することがある。個人的な感想だが、非常に多くのスレッドを使用して高い並列化効率を実現するには、CPU のキャッシュ構造やマザーボード上のメモリと CPU ソケットとの結合関係をある

程度把握した上でプログラミングする必要があると感じている。

さらに、今回の結果はあくまで当方のテスト環境における結果であり、ユニバーサルな結果ではないことに注意されたい。実際、Core i7-975X（3.3GHz, 4cores）で同様のテストを行ってみたところ、4 スレッドの場合で 30% 程度しか速度が改善しなかった。このように、方法 2 の方が速くなるという事実は変わらないが、その改善の度合いは、使用する計算機のアーキテクチャに強く依存しているものと言える。

また、各スレッド内の負荷が非常に大きくなり、ループ内の計算時間が、スレッドの生成・消滅にかかる時間よりもはるかに長くなるような場合は、方法 1 と 2 との差はずっと小さくなるものと考えられる。したがって、手間のかかる方法 2 のような実装方法は、開発者への負担の度合いと、得られる恩恵とのバランスを考慮して行うべきものと考えている。

このように、手軽である OpenMP でも、計算速度の改善を深く追求しようとすればいくらでも複雑になり得る。とはいうものの、巨大な計算機を必要とせず、手近のマルチコアの計算機でも気軽に並列化を実感することをできる OpenMP は、単に並列化のスタート地点としての役割だけでなく、MPI との連携などとも合わせて、今後ますます多方面で浸透していくものと思う。

# 参考文献


[1] 日本語の参考書では、菅原 清文 ”C/C++ プログラマーのための OpenMP 並列プログラミング” (カットシステム) など。

[2] T. Hiramatsu, M. Kawasaki, F. Takahashi, JCAP **1006** (2010) 008. [arXiv:1003.1779 [hep-ph]].

[3] T. Hiramatsu, M. Kawasaki and K. Saikawa, JCAP **1108** (2011) 030 [arXiv:1012.4558 [astro-ph.CO]].

[4] T. Hiramatsu, M. Kawasaki, T. Sekiguchi, M. Yamaguchi and J. Yokoyama, Phys. Rev. D **83** (2011) 123531 [arXiv:1012.5502 [hep-ph]].

[5] 場の理論の標準的な教科書としては Peskin, Schroeder ”And Introduction to Quantum Field Theory” (Perseus Books); Ryder ”Quantum Field Theory” (Cambridge) などを参照。

[6] ネルセット, ヴァンナー, ハイラー ” 常微分方程式の数値解法　I” (シュプリンガー, 邦訳版); 三井 斌友 ” 常微分方程式の数値解法” (岩波書店)

[7] `http://www.fftw.org`