スーパーコンピュータを使おう（3）

平　野　　靖

Ⅰ．はじめに

本連載 [1, 2] の 3 回目となる本稿では，スーパーコンピュータで効率的にプログラムを実行するための方法をいくつかご紹介します。ただし，本稿では概要をご紹介するにとどめます。詳細は適宜，参考文献をご参照ください。

名古屋大学情報連携基盤センターで運用しているスーパーコンピュータシステム HPC2500 は文献 [1 ～ 3] のように，24 ノードの大規模 SMP（Symmetric Multi Processor）マシンがクロスバネットワークによって接続されています。各ノードは 512GB のメインメモリを有し，64 個あるいは 128 個の CPU によってメインメモリが共有されています。このような形態の計算環境を大規模 SMP クラスタと呼びます（図 1）。また，いわゆる T2K オープンスパコン仕様（東京大学，京都大学，筑波大学が策定したスーパーコンピュータの仕様）では，市場でごく一般的に販売される部品で構成されるパソコンを多数つなげて PC クラスタを構成し（図 2），スーパーコンピュータを実現しようとしており，今後，このような形態での大規模計算環境が広まっていくものと考えられます。

大規模 SMP クラスタと PC クラスタの共通点は，筐体内にある複数の CPU あるいはコアがメインメモリを共有し，さらに複数の筐体が高速ネットワークで接続されている，ということです。このような計算環境で効率のよい計算を行うためには，複数の CPU を使った並列計算をすることです。そこで，本稿ではいくつかの並列化の方法をご紹介していきます。

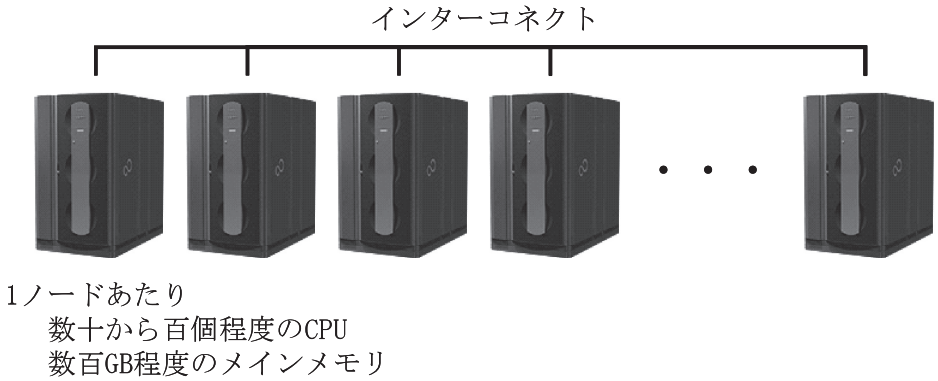


図 1　大規模 SMP クラスタ

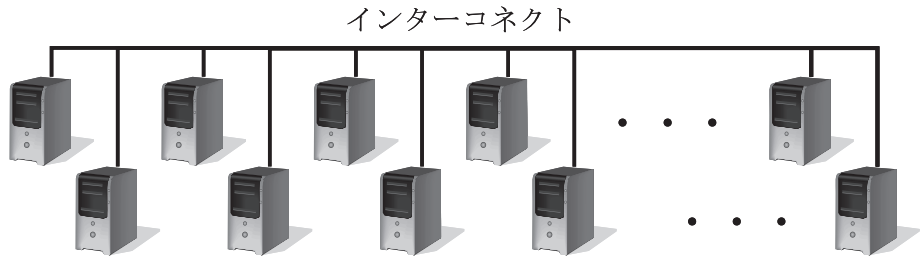


図２　PC クラスタ

Ⅱ．並列化

1．並列化とは？

ごく簡単に言えば，並列化とは複数の CPU を使って１つのプログラムを実行することです。これによって１個の CPU を使う場合よりも計算時間（正確に言うならば，経過時間）を短縮できます。どの程度の計算時間を短縮できるかは，プログラムに依存しますが，例えば，プログラムを N 個の完全に独立な部分に分割でき，それを N 個の CPU で並列的に実行すれば，理想的には１個の CPU で実行する場合の 1/N の時間で計算することができます。並列化することによってどの程度の高速化が達成できるかは下記の式で計算することができます。

$$(\text{計算時間}) = (\text{並列実行できない部分の計算時間}) + \frac{(\text{並列実行できる部分の計算時間})}{(\text{使用する CPU の数})}$$

これはアムダールの法則と呼ばれ，並列化した際の性能向上の指針となります。また，逐次実行した場合の計算時間を並列実行した場合の計算時間で割って得られる値をスピードアップ（あるいはスピードアップ率）と言います。実際には，CPU 間の通信に必要な時間，並列化した部分の同期待ち，あるいは並列実行するための処理にかかるオーバヘッドなどが必要となるため，処理時間が 1/N になることはありません(注)。

自作のプログラムを並列化するためには，OpenMP [4] や MPI [5,6] を使うのが一般的です。計算環境によっては，独自の並列化機構（自動並列化）を持っている場合もあります。HPC2500 では，自動並列化のほかに，XPFortran による並列化をサポートしています。また，情報連携基盤センターのアプリケーションサーバにインストールされたアプリケーションには α-FLOW や Gaussian などのように並列計算に対応したものもありますので，ご利用ください。

（注）　場合によっては，スピードアップ率が CPU 数を上回る「スーパーリニア」という現象が見られることがありますが，これは扱うデータの量，キャッシュのサイズなどが複雑に関係しあって生じるため，一般的には，スピードアップ率は CPU 数未満となります．

2. 並列化のための準備

並列化を行ううえで，最も重要なことは，並列化できる部分を見つけ出すことです。上記の自動並列化を使えば並列化する部分（＝独立に計算できる部分）をユーザが明示的に指定することなく並列化することもできますが，並列化効率はそれほど高くありません。そこで，並列化の効果を高めるためには，どの部分をどのように並列化するのかを指定しなければなりません。

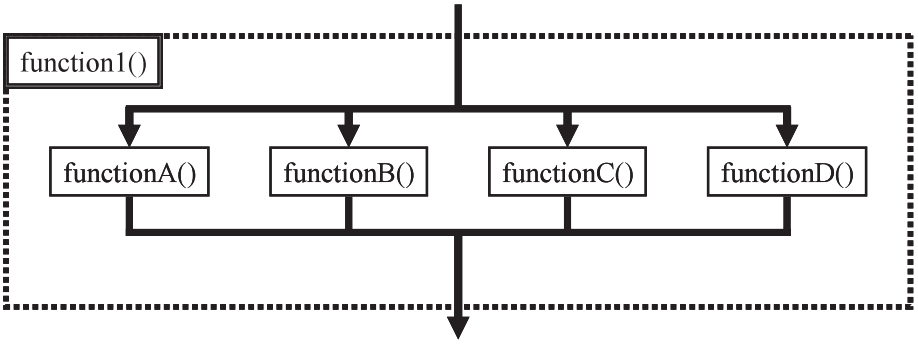
並列化の戦略としては，①独立な複数の関数やブロックを並列化する，②ループ変数について独立なループ内の関数やブロックを並列化する，という２つが考えられます。例えば，リスト１

```
int function1()
{
        :
    functionA();
    functionB();
    functionC();
    functionD();
        :
}
```

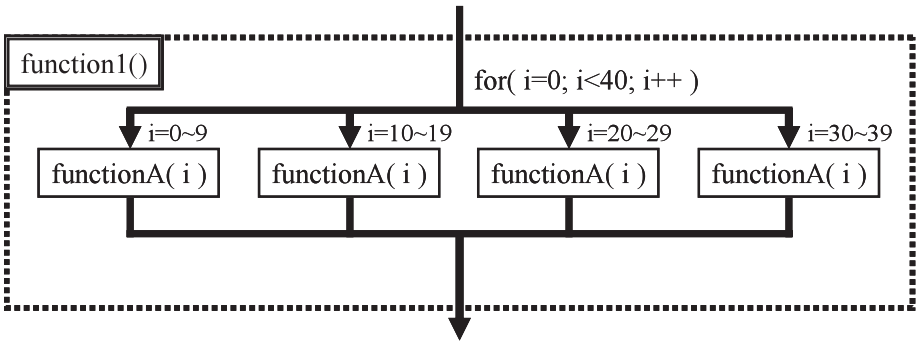
リスト１

```
int function1()
{
        :
    for(i=0;i<40;i++){
       functionA(i);
   }
        :
}
```

リスト２



（a）複数の関数やブロックの並列化



（b）ループ内の関数やブロックの並列化

図３　並列化の戦略

の function1（）から呼び出される functionA（），functionB（），functionC（），functionD（）が互いに独立であれば，4 つの CPU を使って並列計算をすることが可能です（図 3（a））。また，リスト 2 のように，ループ内の処理が独立であれば並列に実行することも可能です（図 3（b））。いずれの場合も，処理が分岐してから再び合流するまでの各 CPU の計算時間がなるべく等しくなるようにしないと，CPU が遊んでしまい，結果として多くの計算時間が必要となります。

3. 並列化手段

並列化する部分が決まったら，次はそれをどうやって並列化するかを考えます。上記のように，並列化するための手段としては，OpenMP，MPI，XPFortran などが考えられます。なお，OpenMP と MPI は C 言語，C++，Fortran のプログラムを並列化できます。また，XPFortran は名前から分かるように，Fortran をベースにした並列プログラミング言語です。これらの並列化手段は大きく分けると，スレッド並列とプロセス並列に分けることができます。それぞれの特徴を示します。

＜スレッド並列＞

- 一般的には 1 つの筐体（あるいはノード）内にある複数の CPU を使って並列化する方法。
- すべての CPU が同じメモリ空間を共有する。
- ある CPU が参照する変数を他の CPU が参照することが可能。
- プロセス並列プログラムのような CPU 間での通信を行う必要がない。
- OpenMP の場合，並列化したい部分にディレクティブを挿入するだけでよいため並列化が簡単。
- OpenMP と自動並列化（HPC2500 用の富士通コンパイラの場合）がこれに該当する。

＜プロセス並列＞

- 使用する CPU が複数の筐体にまたがってもかまわない。
- 複数の CPU がそれぞれ独自のメモリ空間を持つ。
- ある CPU が参照する変数を他の CPU が参照するには CPU 間の通信を行う必要がある。
- 並列化の方法と CPU 間通信の指示を明示的に行う必要がある。
- MPI と XPFortran がこれに該当する。

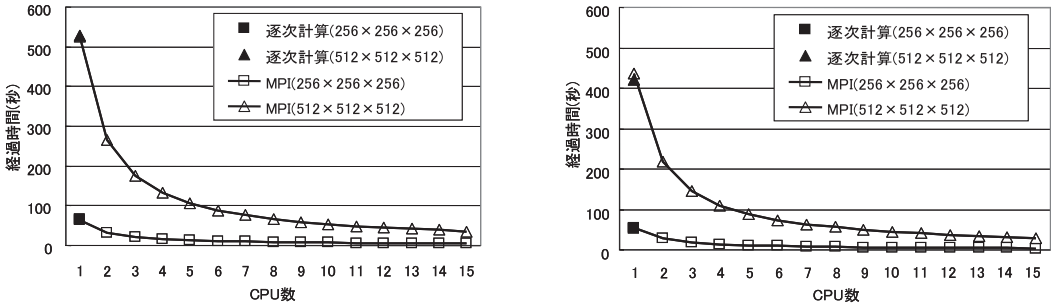
なお，スレッド並列とプロセス並列を組み合わせることもできます。どの方法で並列化するかは，必要とするメモリ量と 1 つの筐体内のメインメモリ量の関係，1 筐体内の CPU 数，並列化部分間での通信量などを勘案した上で決定してください。プログラミング方法は，OpenMP については文献 [7 ～ 9] などを，MPI については文献 [10] などを，XPFortran については文献 [11] などをそれぞれご参照ください。また，当センターでの実行方法は文献 [3] をご参照ください。

Ⅲ．MPI での実例

それでは，次に，MPI で並列化した場合にどの程度の効果が得られるのかを見ていきましょう。対象としたプログラムは 3 次元画像に対するメディアンフィルタで，マスクサイズは 3 画素× 3

画素× 3 画素です。メディアンフィルタとは，入力画像中のある画素（注目画素と言います）を中心としてその周囲（ここでは, 3 画素× 3 画素× 3 画素の範囲）の画素の値のうち, 中央値（メディアン）を出力画像中の注目画素と同じ位置の画素の値とする処理で，スパイクノイズの除去などに使われます。この処理を入力画像のすべての画素について行います。また, メディアンフィルタは各画素に対して独立に実行することができます。

メディアンフィルタを実現するプログラムを C 言語と Fortran で作成し，画像サイズが 256 画素× 256 画素× 256 画素の画像と 512 画素× 512 画素× 512 画素の画像に対して，使用する CPU の数を変化させて経過時間を測定した結果を図 4 に示します。比較のために，MPI を使わない逐次プログラムで計測した結果も同図に示します。いずれの場合も，計算環境は本センターの HPC2500 で，NQS によるバッチ処理として実行しました。なお，付録に逐次計算，及び並列計算のための C 言語プログラムの一部を示します。



（a）C 言語による実装　　（b）Fortran による実装

図 4　MPI による並列化の効果

本プログラムでは，メディアンフィルタの適用のみを行っており，プログラムのほとんどすべての部分が並列化可能です。したがって，逐次計算した場合と MPI によって並列化したプログラムで 1 個の CPU のみを用いた場合で，経過時間はほとんど同じでした。また，C 言語による実装でも, Fortran による実装でも, 経過時間はほぼ CPU 数に反比例していることが分かります。このように，複数の CPU を使って並列計算をすることによって計算時間を短縮することが可能です。

なお，Fortran による実装の方が，C 言語による実装に比べて経過時間が短いのは，コンパイラの性能の差によります。Fortran は文法的な制約により，C 言語よりも自由度が少ないですが，その分，コンパイラによるプログラムの解析を単純化でき，無駄の少ない実行プログラムを生成することが可能な場合が多いようです。

Ⅳ．まとめ

本稿では，現在，及び今後の大規模計算環境で必要とされる並列計算の概要をご紹介しました。並列計算の方法はさまざま存在しますし，今後，新たな並列化手法が登場してくるかもしれません。しかしスレッド並列とプロセス並列の概念をしっかりと抑えておけば応用は利くと思います。

これを機会にぜひ並列プログラミングに挑戦してみてください。

参考文献

[1] 石井克哉：スーパーコンピュータを使おう（1），名古屋大学情報連携基盤センターニュース，Vol.6, No.3, pp.263-267, 2007.8
http://www2.itc.nagoya-u.ac.jp/pub/pdf/pdf/vol06_03/263_267kaisetu01.pdf

[2] 永井亨：スーパーコンピュータを使おう（2），名古屋大学情報連携基盤センターニュース，Vol.6, No.4, pp.355-360, 2007.11
http://www2.itc.nagoya-u.ac.jp/pub/pdf/pdf/vol06_04/355_360kaisetu01.pdf

[3] 津田知子：新スーパーコンピュータ（HPC2500）利用のしおり，名古屋大学情報連携基盤センターニュース，Vol.4, No.2, pp.104-113, 2005.5
http://www2.itc.nagoya-u.ac.jp/pub/pdf/pdf/vol04_02/104_113system01.pdf

[4] http://www.openmp.org/drupal/

[5] http://www-unix.mcs.anl.gov/mpi/

[6] http://www.mpi-forum.org/

[7] 南里豪志，天野浩文：OpenMP 入門（1），九州大学情報基盤センター広報 全国共同利用版，Vol.1, No.3, pp. 186-215, 2001.10

[8] 南里豪志，渡部善隆：OpenMP 入門（2），九州大学情報基盤センター広報 全国共同利用版，Vol.2, No.1, pp.1-40, 2002.3

[9] 南里豪志：OpenMP 入門（3），九州大学情報基盤センター広報 全国共同利用版，Vol.2, No.3, pp.239-276, 2002.11

[10] P. パチェコ：MPI 並列プログラミング，培風館，東京，2001

[11] 永井亨：XPFortran 入門，名古屋大学情報連携基盤センターニュース，Vol.5, No.2, pp.129-168
http://www2.itc.nagoya-u.ac.jp/pub/pdf/pdf/vol05_02/129_168kaisetu02.pdf

付録：プログラムの例

リスト 3 に 3 次元画像に対するメディアンフィルタを実行するためのプログラムの一部を示します。メディアンフィルタの主要部分は xx, yy, 及び zz の 3 重ループの部分で，ここで処理対象画像 image の注目画素（x, y, z）の近傍にある画素（x+xx, y+yy, z+zz）の値を配列 list に格納し，その中央値を出力画像 median に代入します。これを x, y, 及び z の 3 重ループで繰り返して計算することによって画像全体にメディアンフィルタを施します。なお，image_size には画像の一辺の画素数（今回は 256 あるいは 512）が，mask_size には中央値の計算に使う範囲（今回は 3）があらかじめ代入されているものとします。

このプログラムでは処理を高速化するため 3 次元画像は 1 次元配列として扱われています。なお，アルゴリズムを工夫することによってさらなる高速化も可能ですが，プログラムが煩雑にな

るため，ここではもっとも単純なアルゴリズムによる実装を示しています。

```
/* 画像内のすべての画素に対するループ */
for(z=mask_size/2; z<image_size-mask_size/2; z++){
 for(y=mask_size/2; y<image_size-mask_size/2; y++){
  for(x=mask_size/2; x<image_size-mask_size/2; x++){

    /* マスク内の画素の画素値を配列 list に格納 */
    i=0;
    for(zz=-mask_size/2; zz<=mask_size/2; zz++){
     for(yy=-mask_size/2; yy<=mask_size/2; yy++){
      for(xx=-mask_size/2; xx<=mask_size/2; xx++){
       list[i]=image[(x+xx)+(y+yy)*image_size+(z+zz)*image_size*
       image_size];
       i++;
      }
     }
    }

    sort( list, i );  /* 配列 list を昇順に並べ替え */

    /* 中央値を配列 median に格納 */
    median[x+y*image_size+z*image_size*image_size]=list[mask_size*
    mask_size*mask_size/2];
  }
 }
}
```

リスト3　逐次計算用プログラム

次に，逐次計算用プログラムを MPI によって並列化したものをリスト 4 に示します。太字の部分が並列化のために追加した部分です。

どんなプログラムであっても MPI を使うためには，mpi.h のインクルード，MPI_Init（&argc, &argv）及び MPI_Finalize（）の呼び出しが必要となります。その他の関数はプログラムによっては必要ありませんが，覚えておくとよいものとして MPI_Comm_size，MPI_Comm_rank，MPI_Bcast，MPI_Gather，MPI_Barrier などがあり，とりあえずこれらの 7 つの関数を覚えておけば，MPI による並列化ができます。それぞれの意味はリスト 4 内のコメントや文献 [10] などをご参照ください。

MPI で並列化すると，同じプログラムが複数の CPU（正確には，プロセス）によって実行されます。各 CPU には rank と呼ばれる番号がつけられています。全部でいくつの CPU が使われているのかは MPI_Comm_size 関数を呼び出すことによって，自分の rank は MPI_Comm_rank

関数を呼び出すことによって知ることができます。

一般的に，OpenMP を使う場合でも，MPI を使う場合でも，ループ部分を並列化する際には，一番外側のループを並列化します。これは，このようにすることによって不必要な通信やオーバヘッドを削減することができ，効率的な計算ができる場合が多いからです。今回のプログラムでは，x, y, あるいは z の 3 重ループのいずれを並列化しても計算時間は変わりませんが，メモリ上の画像の配置のされ方を考慮し，やはり一番外側のループ(z)を並列化することにします。ループの並列化では，ブロック分割，あるいはサイクリック分割によって各 CPU に処理を分配します。一般に，サイクリック分割を行った方が，各 CPU に分配される仕事量が均一になる傾向がありますが，どちらを選ぶかは行いたい処理によって異なります。今回はサイクリック分割で分配しました。下記に並列化したループ部分を示します。

```
for(z=mask_size/2+myrank; z<image_size-mask_size/2; z+=nprocs)
```

太字の部分が並列化する際に追加した部分です。myrank は自分の rank，nprocs は全 CPU 数が格納された変数です。ループ変数の初期値と終了条件に mask_size/2 という値がありますが，これは確保していないメモリ領域にアクセスするのを防ぐためです。並列化を理解するためには，

```
for(z= myrank; z<image_size; z+=nprocs)
```

とすると簡単に分かると思います。つまり，各 CPU はそれぞれの rank に対応した z 座標を持つ 2 次元画像（3 次元画像の z 座標を固定すると xy 平面になります）に対する処理を行い，次に全 CPU 数と同じだけ離れた 2 次元画像に対する処理を行います。これを z 座標が image_size よりも小さい間繰り返すと，それぞれの CPU がほぼ同じ枚数の 2 次元画像の処理を行うことになります（図 5）。

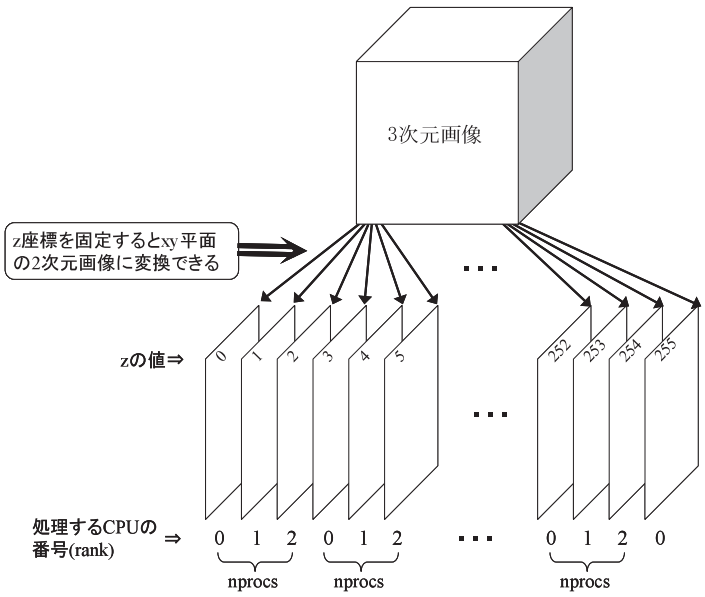


図 5　3 次元画像の 2 次元画像への変換とサイクリック分割

```
#define<mpi.h> /* MPI を使うためのヘッダーファイルをインクルード */
       :
int nprocs; /* 使用する CPU 数 */
int myrank; /* 自分が何番目であるか（rank）を格納する変数 */
       :
MPI_Init(&argc, &argv);  /* MPI 環境の初期化 */
       :
MPI_Comm_size( MPI_COMM_WORLD, &nprocs ); /* 使用する CPU 数の取得 */
MPI_Comm_rank( MPI_COMM_WORLD, &myrank ); /* rank の取得 */
       :
/* rank 0 の CPU から他の CPU への処理対象画像の送信 */
MPI_Bcast(image,image_size*image_size*image_size,MPI_INT,0,
MPI_COMM_WORLD);

/* 画像内のすべての画素に対するループ */
for(z=mask_size/2+myrank; z<image_size-mask_size/2; z+=nprocs){
 for(y=mask_size/2; y<image_size-mask_size/2; y++){
  for(x=mask_size/2; x<image_size-mask_size/2; x++){
  （メディアンフィルタの部分は省略）
  }
 }
}

MPI_Barrier( MPI_COMM_WORLD );  /* 各 CPU の同期待ち */

/* 各 CPU での処理結果を rank 0 の CPU に送信 */
for(z=mask_size/2; z<image_size-mask_size/2; z+=nprocs){
 MPI_Gather((void*)(median+(z+myrank)*image_size*image_size),
            image_size*image_size, MPI_INT,
           (void*)(median+z*image_size*image_size),
           image_size*image_size, MPI_INT, 0,
           MPI_COMM_WORLD );
 }

MPI_Barrier( MPI_COMM_WORLD ); /* 各 CPU の同期待ち */
       :
MPI_Finalize();  /* MPI 環境の終了 */
       :
```

リスト 4　MPI による並列計算用プログラム

（ひらの やすし：名古屋大学情報連携基盤センター）