OpenMP入門(3)

南里, 豪志
九州大学情報基盤センター研究部

https://doi.org/10.15017/1470468

出版情報：九州大学情報基盤センター広報 : 全国共同利用版. 2 (3), pp.239-276, 2002-11. 九州大学情報基盤センター
バージョン：
権利関係：

OpenMP 入門 (3)

南里 豪志 *

1 はじめに

前回, 前々回の OpenMP 入門では, 主にループを対象とした並列化手法を紹介しました. 最終回である今回の記事では, まず 2 節で, これまでほとんど扱わなかったループ以外の処理の並列化手法について紹介します. これは, 並列化できないループをいくつか連続して実行するプログラムに有効な並列化手法です. 次に 3 節で, 複雑なプログラムの並列化において重要となるスレッド間の同期について説明し, OpenMP で提供されている同期の機能を紹介します. さらに 4 節で, コンパイラの自動並列化機能を活用した OpenMP プログラムの作成方法を紹介し, 5 節でプログラムの高速化に関するいくつかの手法を紹介します.

なお, 本記事は以前九州大学情報基盤センター広報 (全国共同利用版) に掲載した「OpenMP 入門 (1)」及び「OpenMP 入門 (2)」の内容を前提としています. 本センターの計算機上で OpenMP プログラムをコンパイル・実行する方法, ならびにループを並列化する方法についてはこれらの記事を参照して下さい. またこれらの記事と同様, 本記事でも主に Fortran プログラムを対象に説明を行ないますが, ほとんどのプログラム例について Fortran だけでなく C 言語によるプログラムも参考として掲載します. ただし, ループの順序や多次元配列の並びが C 言語のプログラムと本文では異なる場合があるので注意して下さい. また, Fortran のプログラム例は自由形式で表記していますが, 固定形式で OpenMP を利用することも可能です.

本記事の内容は主に Chandra らによる OpenMP の解説書 [1] と OpenMP Architecture Review Board による OpenMP 規格 [2] を参考にしています.

2 ループ以外の処理の並列化

前回, 前々回の記事にも書いた通り, OpenMP の指示文によって並列処理を指示された領域は "並列リージョン" と呼ばれ, 複数の "スレッド" と呼ばれる処理の流れによって並列に実行されます. また, 並列リージョン以外の部分は "マスタースレッド" と呼ばれる 1 つのスレッドによって実行されます.

これまで主に扱ってきた並列処理はループを並列リージョンとして指示するものでした. これは同じ処理を繰り返し行う仕事を複数のスレッドで分担させる並列処理です. 料理に例える

* 九州大学情報基盤センター 研究部
E-mail: nanri@cc.kyushu-u.ac.jp

と, たくさんの材料に同じ調理を施すもの, 例えばたくさんの海老フライを作る際に海老の皮剥きを何人かで分担して行うのに似ています.

一方, 御飯を炊いたり吸い物を作ったりする仕事は, それぞれの仕事の中で同時に出来る作業がほとんど無いため, 人数が多くてもそれほど役に立ちません. しかし, 一人が御飯を炊いている横でもう一人は吸い物を作り, 別の一人が海老フライを揚げる, というように, 同時に進行できる別々の仕事をそれぞれ別の人間に担当させると, 全体の調理時間の短縮を図ることができます.

本節では, このように同時に実行出来る別々の仕事をそれぞれ別のスレッドに割り当てることにより, 全体の処理時間を短縮する並列化手法について説明します.

2.1 別々の仕事を並列に実行させる指示文

OpenMP で別々の処理を図 1 に示すようにそれぞれ別のスレッドに割り当て並列に実行するには, parallel sections 指示文を利用します. 図 2,3 に parallel sections 指示文を利用したプログラム例を示します.

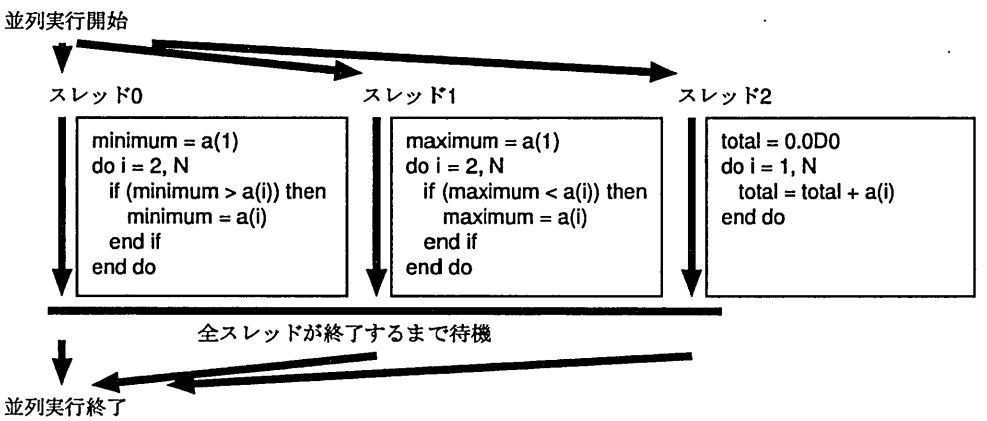

図 1: sections 指示文の実行イメージ

このプログラムでは, 配列 a の最小値, 最大値を探索し, 平均値を計算して, それぞれを表示します. この, “最小値を探索する”, “最大値を探索する”, “(平均値の計算に用いる) 合計値を計算する” という, 3 つの仕事を別のスレッドに割り当てて同時に実行させるために, parallel sections 指示文を用いています.

parallel sections 指示文は, 並列に実行させる領域の指定に利用します. Fortran では, 並列に実行させる領域を parallel sections 指示文と end parallel sections 指示文で囲みます. 一方 C 言語, C++言語では, 並列に実行させる領域の先頭に parallel sections 指示文を書いた後, 領域を{ } で囲みます. 並列に実行させる領域中で, 各スレッドに割り当てる個々の仕事は section 指示文で指定します. Fortran では, プログラム中で section 指示文と section 指示文の間に挟まれた部分がスレッドに割り当てられる仕事の単位となります.

一方C言語, C++言語では各スレッドに割り当てる仕事の先頭にsection指示文を書いた後, さらに割り当てる仕事全体を { } で囲みます.

このプログラムを実行すると, 図1のように配列aの最小値を探索する部分と最大値を探索する部分と合計を計算する部分がそれぞれ別のスレッドで並列に処理されます. 割り当てられた仕事を終えたスレッドは, 全スレッドが仕事を終了するまで待ち, その後マスタースレッドが並列リージョン以降の部分を処理します.

```
program session
implicit none
integer, parameter          :: N = 10000000
real(kind=8),dimension(N)   :: A
integer                     :: i
real(kind=8)                :: maximum, minimum, total

real(kind=8) rand
external  random

!$omp parallel do
do i = 1, N
   A(i) = rand(0)
end do

!$omp parallel sections
!$omp section
minimum = a(1)
do i = 2, N
   if (minimum > a(i)) then
      minimum = a(i)
   end if
end do

!$omp section
maximum = a(1)
do i = 2, N
   if (maximum < a(i)) then
      maximum = a(i)
   end if
end do

!$omp section
total = 0.0D0
do i = 1, N
   total = total + a(i)
end do
!$omp end parallel sections

print *, 'Minimum = ', minimum, '  Maximum = ', maximum, &
         '  Average = ', total / N

end program
```

図 2: 最大値の探索, 最小値の探索, 合計値の計算を同時に実行するプログラム (Fortran)

```
#include <stdio.h>
#include <omp.h>

#define N 10000000

main()
{
  int i;
  double A[N];
  double minimum, maximum, total;

#pragma omp parallel for
  for (i = 0; i < N; i++)
    A[i] = rand();

#pragma omp parallel sections
  {
#pragma omp section
    {
      minimum = A[0];
      for (i = 1; i < N; i++)
        if (minimum > A[i])
          minimum = A[i];
    }

#pragma omp section
    {
      maximum = A[0];
      for (i = 1; i < N; i++)
        if (maximum < A[i])
          maximum = A[i];
    }

#pragma omp section
    {
      total = 0.0;
      for (i = 0; i < N; i++)
        total = total + A[i];
    }
  }

  printf("Minimum = %e   Maximum = %e   Average = %e\n", minimum, maximum,
         total / N);
}
```

図 3: 最大値の探索, 最小値の探索, 合計値の計算を同時に実行するプログラム (C 言語)

2.2 並列化手法の選択

基本的に, プログラムを並列化する手法は前回の記事で紹介した「ループの並列化」と, 本節で紹介した「仕事単位の並列化」の 2 つに分類されます. 前回の記事にも書いた通り, ほとんどのプログラムにおいてループの処理に最も時間を要するので, 通常はまず「ループの並列化」を検討します.

一方, 並列化できないループや並列化の効果が得られないほど実行回数の少ないループが多く存在するプログラム中では, 「仕事単位の並列化」を検討します. ただし, 仕事単位の並列化を適用するには, 並列実行させる各仕事が連続しており, さらにそれぞれの仕事をどんな順序で実行しても実行結果に影響が無いことが明らかでなければなりません. 例えば以下の例のように, いくつかの配列について同じ仕事を連続して行う場合, それぞれの配列について並列に実行することが出来ます.

Fortran

```
!$omp parallel sections
!$omp section
  call update(A)
!$omp section
  call update(B)
!$omp section
  call update(C)
!$omp end parallel sections
```

C 言語

```
#pragma omp parallel sections
  {
#pragma omp section
    {
      update(A);
    }
#pragma omp section
    {
      update(B);
    }
#pragma omp section
    {
      update(C);
    }
  }
```

仕事単位の並列化でもう一つ注意すべき点は, 各スレッドに割り当てる仕事量です. 割り当てる仕事量に偏りがあると, 仕事量の少ないスレッドが仕事量の多いスレッドを待つので, 並列化の効果が損なわれます. ループの並列化では, ループの実行回数で仕事を分配できたので仕事量の差を意識する必要はほとんどありませんでした. しかし仕事単位の並列化では, 特に図 2, 3 のように各スレッドが全く別の仕事をする場合, 仕事量に偏りがあるかどうかをプログラムの字面だけで判断するのは困難です. そのため, 前回の記事で紹介した性能解析ツールを

利用して事前にそれぞれの仕事で費やされる時間を調べておくと良いでしょう.

3 より複雑なプログラムの並列化

3.1 parallel 指示文

前々回の OpenMP 入門で紹介した通り, OpenMP のプログラムでは parallel 指示文によって並列実行が開始されます. 一方 parallel do(parallel for) 指示文でも並列実行が開始されますが, 実はこれは以下のように連続した parallel 指示文と do 指示文を繋げて記述したものです.

Fortran

```
!$omp parallel
!$omp do
  do i = 1, N
    ...
  end do
!$omp end parallel
```

C 言語

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

parallel sections 指示文も同様です.

また, parallel 指示文で指定された並列リージョン内で, 以下のように do(for) 指示文や sections 指示文を混在させることができます.

Fortran

```
!$omp parallel
!$omp sections
!$omp section
  ...
!$omp section
  ...
!$omp end sections

!$omp do
  do i = 1, N
    ...
  end do
!$omp end parallel
```

C 言語

```
#pragma omp parallel
  {
#pragma omp sections
    {
#pragma omp section
      {
        ...
      }
#pragma omp section
      {
        ...
      }
    }

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

このように一つの並列リージョン内で複数の並列処理を記述すると, スレッド生成, 廃棄に要するコストを削減出来るという利点があります. OpenMP では, 並列リージョンの開始時に並列処理に参加するスレッドを生成し, 終了時にスレッドを廃棄します. そのため, parallel do(parallel for) 指示文や parallel sections 指示文で並列リージョンを何度も指定すると, スレッドの生成, 廃棄コストによる影響が大きくなります (図 4). 一方, 一つの並列リージョン内で複数の並列処理を記述すると, このコストを生成, 廃棄それぞれ一回ずつに削減することができます (図 5).

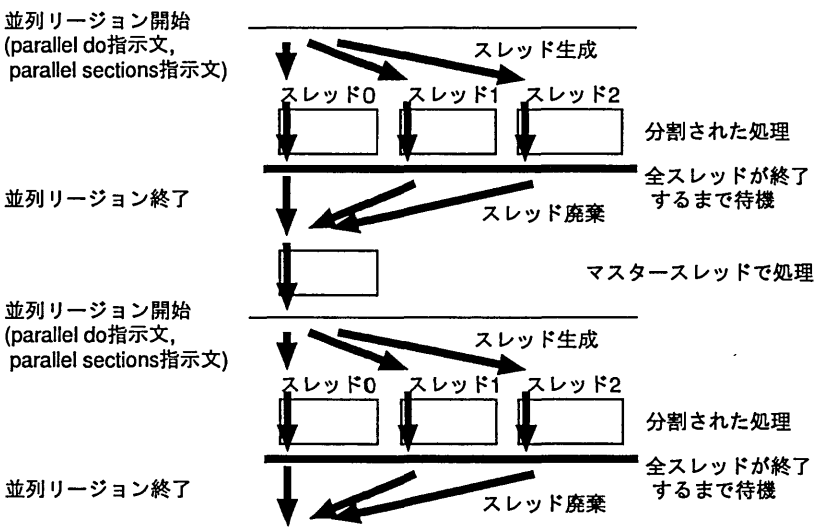

図 4: 並列リージョンを何度も指定する場合の動作

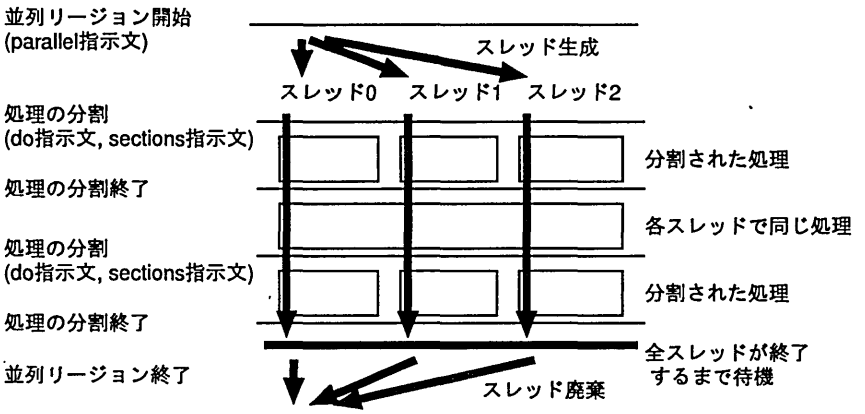

図 5: 一つの並列リージョンで複数の並列処理を記述する場合の動作

ただし, do(for) 指示文や sections 指示文で並列化を指定した場合に, どの番号のスレッドにどの処理が割り当てられるかを予測することはできません. 例えば, do(for) 指示文で並列化されたループの最初の繰り返しを実行したスレッドが, sections 指示文で並列化された仕事群の最初の仕事を実行するとは限りません. スレッドへの割り当てを細かく指定したい場合は, do(for) 指示文や sections 指示文を用いず, 次節で説明するライブラリ関数を用いて並列化します.

3.2 do 指示文や sections 指示文を使わない並列処理

do(for) 指示文や sections 指示文を使うと, 各スレッドへの処理の割り当てを内部で自動的に行ってくれるので, 簡単なプログラムの並列化では非常に便利です. しかし, プログラムが複雑で do(for) 指示文や sections 指示文ではうまく並列化できない場合があります. このような場合は, プログラマが明示的に各スレッドへの処理の割り当てを指示することにより, 効率の良い並列処理が行えないか検討します.

各スレッドへの処理の割り当ては, スレッド番号に応じて実行する処理の内容を変更することが基本となります. そこで, 以下のライブラリ関数を利用します.

omp_get_thread_num() :
: スレッド毎につけられた番号を返します. このスレッドの番号は 0 番から順につけられ, 複数のスレッドに同じ番号がつくことはありません. この番号をもとに, 各スレッドへの処理の割り当てを行います.

omp_get_num_threads() :
: 現在処理を行っているスレッドの数を返します. すなわち, 並列リージョン内では並列処理に参加しているスレッドの数を返し, 並列リージョン以外では 1 を返します.

omp_get_max_threads() :
: 並列処理に参加できるスレッドの数, すなわち, その時点で並列リージョンを開始した場

合に, その並列処理に参加するスレッド数を返します.

omp_set_num_threads(スレッド数) :
並列処理に参加するスレッドの数を指定します.

ここでは, これらの関数を用いてスレッドにプログラムの一部を割り当てる例と, ループを分割してスレッドに割り当てる例を紹介します.

ただし, これらのライブラリ関数は OpenMP の機能が無効の場合は利用できません. そのためこの例では, OpenMP の機能が無効の場合でも正しくコンパイルできるよう, 前々回の記事で紹介した条件コンパイルの機能を用いています.

3.2.1 スレッドへのプログラムの一部の割り当て

スレッドへのプログラムの割り当ては, Fortran や C 言語, C++言語の条件分岐を利用して記述できます. 以下の例では, sections 指示文による自動的な処理の割り当てとほぼ同様に仕事を各スレッドに割り当てています. ここで, スレッド毎につけられた番号を変数 myid に格納していますが, この値はスレッド毎に異なるのでプライベート変数として指示しています.

Fortran

```
!$ integer :: myid, omp_get_thread_num
   ...
!$omp parallel private(myid)
!$ myid = omp_get_thread_num()
!$ if (myid == 0) then
    スレッド0に割り当てる処理
!$ else if (myid == 1) then
    スレッド1に割り当てる処理
!$ else if (myid == 2) then
    スレッド2に割り当てる処理
!$ end if
!$omp end parallel
```

C言語

```
#pragma omp parallel private(myid)
  {
#ifdef _OPENMP
    myid = omp_get_thread_num();
#endif
#ifdef _OPENMP
    if (myid == 0){
#endif
      スレッド0に割り当てる処理
#ifdef _OPENMP
    } else if (myid == 1){
#endif
      スレッド1に割り当てる処理
#ifdef _OPENMP
    } else if (myid == 2){
#endif
      スレッド2に割り当てる処理
#ifdef _OPENMP
    }
#endif
  }
```

ただしこの例では, 並列処理に参加するスレッド数が3以上でなければ正しく実行できません. 実行時にこのことを覚えていれば良いのですが, 念のために並列処理に参加するスレッド数を並列リージョンが開始される前に調べ, 必要に応じてスレッド数を変更するようプログラム中に記述しておいた方が無難です. このスレッド数の調査と変更にも OpenMP のライブラリ関数を利用します. 以下の例では, 必ずスレッド数が3以上となるように並列リージョンの前で調整を行っています.

Fortran

```
!$ integer :: maxthread, omp_get_max_threads
   ...
!  並列処理に参加できるスレッド数を取得する.
!$ maxthread = omp_get_max_threads()
!  もしスレッド数が 3 より小さければ, スレッド数を 3 に変更する.
!$ if (maxthread < 3) then
!$   call omp_set_num_threads(3)
!$ end if
!$omp parallel
   ...
!$omp end parallel
```

C 言語

```
#ifdef _OPENMP
/*  並列処理に参加できるスレッド数を取得する.  */
  maxthread = omp_get_max_threads();
/* もしスレッド数が 3 より小さければ, スレッド数を 3 に変更する. */
  if (maxthread < 3)
     omp_set_num_threads(3);
#endif
#pragma omp parallel
 {
   ...
 }
```

3.2.2　ループの分割とスレッドへの割り当て

次に, 上記の関数を使ってループを分割し, 各スレッドに割り当てる例を紹介します.

ここでは, 並列処理に参加するスレッド数が 3 以上とし, そのうちスレッド 0 とスレッド 1 にはそれぞれ単独の仕事を割り当て, スレッド 2 以降に N 回繰り返すループを分割して割り当てることにします.

並列処理に参加するスレッド数を T とすると, スレッド 0 とスレッド 1 はそれぞれ別の仕事が割り当てられるので, ループの並列処理に参加するスレッド数は T-2 です.

ここで, もし N が T-2 で割りきれるのであれば, 各スレッドには N/(T-2) 回の繰り返しが割り当てられるので, スレッド番号 myid のスレッドはループの (myid-2)*N/(T-2)+1 番目の繰り返しから N/(T-2) 回分の繰り返しを担当すれば良いことになります.

一方, N が T-2 で割りきれない場合, N/(T-2) の余りを nmod とすると, nmod 個のスレッドに一回ずつ多く繰り返しを担当させる必要があります. その結果, スレッド番号 myid のスレッドが担当するループの開始位置は, myid-2 が nmod 以下の場合 (myid-2)*(N/(T-2)+1)+1 番目となります. 一方 myid-2 が nmod より大きい場合は, (myid-2)*N/(T-2)+nmod+1 番目からの繰り返しを担当します. また, 繰り返しを担当する回数は, myid-2 が nmod 未満の場合 N/(T-2)+1 回で, myid-2 が nmod 以上の場合 N/(T-2) 回となります.

このように, ループの繰り返し回数がスレッド数で割りきれない場合を考慮して各スレッドへの処理の割り当てを明示的に記述すると以下のようになります. ただしこの例もスレッド数が 3 以上でなければ正常に動作しないので, 実行時に環境変数 OMP_NUM_THREADS を 3 以上にするか, もしくは前節で紹介した通り, プログラム中にスレッド数の調整のための処理を追加します.

Fortran

```
!$ integer :: nthreads, myid, nmod, ndiv, mylocalid
!$ integer :: omp_get_num_threads, omp_get_thread_num

! start, end, myid, mylocalid は各スレッドで値が異なるので
! private で宣言する
!$omp parallel private(start, end, myid, mylocalid)
!$  myid = omp_get_thread_num()
!$  if (myid == 0) then
      スレッド0の仕事
!$  else if (myid == 1) then
      スレッド1の仕事
!$  else
      start = 1
      end = N
!  ループを担当するスレッド数
!$    nthreads = omp_get_num_threads() - 2
!  ループを担当するスレッドの中での自分のスレッド番号
!$    mylocalid = myid - 2
!  繰り返し回数 N をスレッドで等分できない場合の余り
!$    nmod = mod(N, nthreads)
!  各スレッドに割り当てる最小限の繰り返し回数
!$    ndiv = floor(real(N/nthreads))
!$    if (nmod > mylocalid) then
!      ndiv + 1回を割り当てるスレッド
!$      start = mylocalid * (ndiv + 1) + 1
!$      end = start + ndiv
!$    else if (nmod == mylocalid) then
!      ndiv回を割り当てるスレッド（境界）
!$      start = mylocalid * (ndiv + 1) + 1
!$      end = start + ndiv - 1
!$    else
!      ndiv回を割り当てるスレッド
!$      start = mylocalid * ndiv + nmod + 1
!$      end = start + ndiv - 1
!$    end if
      do i = start, end
        スレッド2以降の仕事
      end do
!$  end if
!$omp end parallel
```

C 言語

```
#ifdef _OPENMP
  int myid, nthreads, nmod, ndiv, mylocalid;
#endif

/* start, end, myid, mylocalid は各スレッドで値が異なるので
 *  private で宣言する */
#pragma omp parallel private(start, end, myid, mylocalid)
  {
#ifdef _OPENMP
    myid = omp_get_thread_num();

    if (myid == 0){
#endif
      スレッド 0 の仕事
#ifdef _OPENMP
    } else if (myid == 1){
#endif
      スレッド 1 の仕事
#ifdef _OPENMP
    } else {
#endif
      start = 0; end = N-1;
#ifdef _OPENMP
/*  ループを担当するスレッド数 */
      nthreads = omp_get_num_threads() - 2;
/*  ループを担当するスレッドの中での自分のスレッド番号 */
      mylocalid = myid -2;
/*  繰り返し回数 N をスレッドで等分できない場合の余り */
      nmod = N % nthreads;
/*  各スレッドに割り当てる最小限の繰り返し回数 */
      ndiv = N / nthreads;
      if (nmod > mylocalid){
/*      ndiv + 1 回を割り当てるスレッド */
        start = mylocalid * (ndiv + 1);
        end = start + ndiv;
      } else if (nmod == mylocalid){
/*      ndiv 回を割り当てるスレッド (境界) */
        start = mylocalid * (ndiv + 1);
        end = start + ndiv - 1;
      } else{
/*      ndiv 回を割り当てるスレッド */
        start = mylocalid * ndiv + nmod;
        end = start + ndiv - 1;
      }
#endif
      for (i = start; i <= end; i++){
        スレッド 2 以降の仕事
      }
    }
  }
```

3.3 並列処理の細かい制御

3.3.1 バリア同期:全スレッドで足並みをそろえる

OpenMP には, 全スレッドの足並みをそろえる機能としてバリア同期が用意されています.

例えばスレッド 0 が計算した結果を共有変数 a に格納し, スレッド 1 がその格納された計算結果を参照する場合, スレッド 0 が計算結果を a に格納し終わった後でなければスレッド 1 は正しい値を参照できません. しかし, 並列リージョンでは基本的に各スレッドがそれぞれ独立して処理を進めるので, スレッド 1 からスレッド 0 の処理の様子を知ることはできません.

そこでバリア同期を使い, スレッド 0 が計算結果を共有変数 a に格納した後で全スレッドで足並みをそろえることにします. すると, このバリア同期の後であればスレッド 1 は共有変数 a に格納されたスレッド 0 の計算結果を参照することができます. また, a の値を用いずに行える処理があれば, バリア同期の前に実行しておくことができます.

バリア同期は barrier 指示文によって実行されます. ただし, バリア同期は "全ての" スレッドの足並みをそろえる機能なので, 全てのスレッドが barrier 指示文に到達しなければ先に進むことができません. ですから, バリア同期の利用にあたっては, スレッドによってバリア同期の実行回数が違う, ということが無いように注意して下さい. 通常は, 以下のように並列リージョンのうち全スレッドが実行する場所にバリア同期を記述します.

Fortran

```
!$omp parallel private(myid) shared(a)
!$ myid = omp_get_thread_num()
!$ if (myid == 0) then
     処理 (計算し, 変数 a に結果を格納)
!$ else if (myid == 1) then
     処理 (変数 a の値を用いずに行える処理)
!$ else if (myid == 2) then
     処理 (スレッド 2 に割り当てる処理の前半)
!$ end if

!$omp barrier

!$ if (myid == 0) then
     処理 (残りの処理)
!$ else if (myid == 1) then
     処理 (変数 a の値を参照して行う処理)
!$ else if (myid == 2) then
     処理 (スレッド 2 に割り当てる処理の後半)
!$ end if
!$omp end parallel
```

C 言語

```
#pragma omp parallel private(myid) shared(a)
  {
#ifdef _OPENMP
    myid = omp_get_thread_num();
#endif
#ifdef _OPENMP
    if (myid == 0){
#endif
     処理 (計算し, 変数 a に結果を格納)
#ifdef _OPENMP
    } else if (myid == 1){
#endif
     処理 (変数 a の値を用いずに行える処理)
#ifdef _OPENMP
    } else if (myid == 2){
#endif
     処理 (スレッド 2 に割り当てる処理の前半)
#ifdef _OPENMP
    }
#endif

#pragma omp barrier

#ifdef _OPENMP
    if (myid == 0){
#endif
     処理 (残りの処理)
#ifdef _OPENMP
    } else if (myid == 1){
#endif
     処理 (変数 a の値を参照して行う処理)
#ifdef _OPENMP
    } else if (myid == 2){
#endif
     処理 (スレッド 2 に割り当てる処理の後半)
#ifdef _OPENMP
    }
#endif
  }
```

ここでスレッド 2 の処理を前半と後半に分けていますが, スレッド 2 の処理で変数 a の値を参照しないのであれば, この分け方に特に決まりはありません. ただし, バリア同期での待ち時間を少なくするため, スレッド 2 のバリア同期前後の処理量をスレッド 0, スレッド 1 のバリア同期前後の処理量にできるだけ近付けるようにします.

また, 以下のように各スレッドでそれぞれバリア同期を実行しても構いませんが, この場合はバリア同期を実行しないスレッドが無いよう, 気をつけて下さい.

Fortran

```
!$omp parallel private(myid) shared(a)
!$ myid = omp_get_thread_num()
!$ if (myid == 0) then
     処理 (計算し, 変数 a に結果を格納)
!$omp barrier
     処理 (残りの処理)
!$ else if (myid == 1) then
     処理 (変数 a の値を用いずに行える処理)
!$omp barrier
     処理 (変数 a の値を参照して行う処理)
!$ else
    スレッド2に割り当てる処理
!$omp barrier
!$ end if
!$omp end parallel
```

C言語

```
#pragma omp parallel shared(a)
  {
#ifdef _OPENMP
    myid = omp_get_thread_num();
#endif
#ifdef _OPENMP
    if (myid == 0){
#endif
     処理 (計算し, 変数 a に結果を格納)
#pragma omp barrier
     処理 (残りの処理)
#ifdef _OPENMP
    } else if (myid == 1){
#endif
     処理 (変数 a の値を用いずに行える処理)
#pragma omp barrier
     処理 (変数 a の値を参照して行う処理)
#ifdef _OPENMP
    } else{
#endif
      スレッド2に割り当てる処理
#pragma omp barrier
#ifdef _OPENMP
    }
#endif
  }
```

なお, do(for) 指示文もしくは sections 指示文による並列処理では, 特に指定しなければ処理の最後に暗黙的にバリア同期が実行されます. ですから, バリア同期を利用するのは do(for) 指示文や sections 指示文を用いずに並列処理を行う場合のみです.

3.3.2　他のスレッドの終了を待たずに次の処理に移る

do(for) 指示文もしくは sections 指示文による並列処理で, 処理の最後にバリア同期を実行させないようにするには, 以下のように指示文に nowait 指示節を追加します. 追加位置は, Fortran では end do 指示文, end sections 指示文の後, C 言語, C++言語では for 指示文, sections 指示文の後となります.

Fortran

```
!$omp parallel
!$omp sections
   ...
!$omp end sections nowait

!$omp do
   ...
!$omp end do nowait

!$omp end parallel
```

C 言語

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

#pragma omp for nowait
    for (...){
       ...
    }
  }
```

これにより, 自分に割り当てられた処理を済ませたスレッドは, 他のスレッドの終了を待たずに次の処理に移ることができます. ただし, この場合並列処理中に他のスレッドが作成するデータを, この並列処理の後で参照しないことが明らかでなければなりません. もし, 他のスレッドが生成するデータを並列処理の後で参照する可能性がある場合は, その直前にバリア同期を実行して全スレッドの足並みをそろえます.

do(for) 指示文や sections 指示文で, 特に指示をしないと並列処理の最後に暗黙的なバリア同期が実行されるのは, 多くのプログラムで, 並列処理中にあるスレッドが作成するデータを並列処理後に他のスレッドが参照するためです.

3.3.3　並列ループの一部を並列化前と同じ順序で実行する

並列ループでは複数のスレッドがそれぞれ自分に割り当てられた繰り返しを独立して実行していくので, 繰り返しの番号と実際に実行される順番が一致するとは限りません. 例えば並

列ループの3番目の繰り返しが10番目の繰り返しよりも先に実行されるという保証はありません.

そのため, 例えば並列ループ内に各繰り返しにおける計算結果を表示する命令が含まれている場合, 表示の順番が無秩序になるので, 見栄えがよくありません. このような場合に, 並列ループ中の一部の命令だけを強制的に元の繰り返しと同じ順番で実行させる ordered 指示文を利用すると, 表示の順番が並列化前の繰り返しと同じになるので, 見栄えを整えることができます.

利用例は以下の通りです. なお, 並列ループ中で ordered 指示文を利用する場合は, その並列ループの do(for) 指示文に ordered 指示節を追加する必要があります.

Fortran

```
!$omp parallel
!$omp do ordered
  do i = 1, N
   ...
!$omp ordered
   write(*, *) "i = ", i,  "  result = ", result(i)
!$omp end ordered
   ...
  end do
!$omp end parallel
```

C言語

```
#pragma omp parallel
  {
#pragma omp for ordered
    for (i = 0; i < N; i++){
      ...
#pragma omp ordered
      {
        printf("i = %d    result = %e\n", i, result[i]);
      }
      ...
    }
  }
```

ただしこの方法では, 繰り返しの度に ordered 指示文で指定された処理が順番通りに実行されるようスレッド間で待ち合わせを行うので, 一般に処理時間が増大すると予想されます. そのため, 可能であれば次節のように表示部分だけ並列ループの外に移して, 一つのスレッドに割り当てる方が効率的です.

3.3.4 一つのスレッドだけに処理を割り当てる

一つのスレッドだけに処理を割り当てるための指示文として OpenMP には single 指示文と master 指示文が用意されています.

single 指示文では, 指定した処理を不特定のどれか一つのスレッドに割り当てます. どのス

レッドに割り当てられるかを予測することはできません. また, single 指示文は nowait 指示節を追加しない限り最後に暗黙的なバリア同期を実行します. nowait 指示節は, Fortran の場合は end single 指示文の後に追加し, C 言語,C++言語の場合は single 指示文の後に追加します.

例えば以下の例では, 不特定の一つのスレッドで計算結果を表示させています. この例では nowait 指示節が記述されていないので, single 指示文で指定された領域の最後で全スレッドが待ち合わせます.

Fortran

```
!$omp parallel
!$omp do
  do i = 1, N
   ...
  end do

!$omp single
  do i = 1, N
    write(*, *) "i = ", i,  "  result = ", result(i)
  end do
!$omp end single
   ...
!$omp end parallel
```

C 言語

```
#pragma omp parallel
  {
#pragma omp for
    for (i = 0; i < N; i++){
      ...
    }
#pragma omp single
    {
      for (i = 0; i < N; i++){
        printf("i = %d    result = %e\n", i, result[i]);
      }
    }
  }
```

一方 master 指示文は, マスタースレッドのみに処理を割り当てます. 利用法は single 指示文とほぼ同様ですが, 最後に暗黙的なバリア同期は実行されません. そのため, nowait 指示節は追加できません. もしマスタースレッドが master 指示文で指定された領域を実行し終わるまで他のスレッドに待たせたい場合は, その領域の直後にバリア同期を追加する必要があります.

3.3.5 複数のスレッドによる共有変数の同時更新の回避

第一回の OpenMP 入門でリダクション変数の説明を書いた際に, 並列リージョンで複数のスレッドが同時に同じ共有変数の値を更新すると, 異常な計算結果が得られる場合がある, という例を紹介しました. このような問題は, 合計値の計算のように簡単な並列処理であればリダクション変数で解決できますが, 複雑な並列処理の場合は, あるスレッドが共有変数に更新作業を行っている間, 他のスレッドにその共有変数の更新作業を行わせないよう, 明示的に制御する必要があります.

OpenMP では, このような制御を critical 指示文, もしくは atomic 指示文で行います. これらの指示文で指定した領域をあるスレッドが実行している間, 他のスレッドはその領域に到達しても実行を開始できず, 現在実行しているスレッドがその領域の実行を終了するまで待つことになります.

また, 同時更新される可能性のある共有変数が複数ある場合, それぞれの変数毎にスレッドの制御を行わなければ, 不必要に実行を待たされるスレッドが出てきます. そこで critical 指示文では, 同時実行を制限する領域に識別子を付けることができます. これにより, 同じ識別子を持つ領域を既に実行中のスレッドが他に存在しない限り, 待たされること無く指定された領域を実行することができます.

例えば, 1 から N までを加算するリダクション計算を, リダクション変数を用いず critical 指示文で記述すると以下のようになります. ここでは, 同時実行を制限する領域に c_x という識別子を付けています. この識別子には任意の名前をつけることができますが, プログラム中で利用している変数と同じ名前を使うことはできません.

Fortran

```
      x = 0
!$omp parallel
!$omp do
      do i = 1, N
!$omp critical(c_x)
         x = x + i
!$omp end critical(c_x)
      end do
!$omp end do
!$omp end parallel
```

C 言語

```
  x = 0;
#pragma omp parallel
  {
#pragma omp for
    for (i = 1; i <= N; i++){
#pragma omp critical(c_x)
      {
        x = x + i;
      }
    }
  }
```

一方 atomic 指示文は, 指示文の直後の一行だけを, 同時実行を制限する対象とします. また, 適用可能な命令は加減算もしくは Fortran の組み込み関数による共有変数の更新に限られています. さらに C 言語では, 以下のいずれかの形式でなければなりません.

```
  x 二項演算子= 式;
  x++;
  ++x;
  x--;
  --x;
```

ここで, x は変数です. また, "式" は x を参照しない計算式です. 二項演算子としては, +, *, -,/,&, ^,|,<<,>> を利用できます.

利用例は以下の通りです.

Fortran

```
!$omp atomic
  x = x + 2

!$omp atomic
  x = (a + 2) * b + x

!$omp atomic
  x = mod(x,10)
```

C 言語

```
#pragma omp atomic
  x -= 2;

#pragma omp atomic
  x += (a + 2) * b;

#pragma omp atomic
  x++;
```

atomic 指示文と同じ処理は critical 指示文でも記述できます. しかし, critical 指示文

に比べて制限が多い分 atomic 指示文の方が効率的に動作するよう実装されている可能性が高いので，可能な限り atomic 指示文を利用することをお勧めします．

4 自動並列化機能の活用

4.1 並列化の前準備としての自動並列化機能の利用

前回の記事に書いた通り GS320 には逐次型プログラムを OpenMP プログラムに変換する自動並列化コンパイラが用意されています．近年の自動並列化技術の進歩により，簡単なループの並列化であれば，この自動並列化だけで十分な性能が得られるようになりましたが，プログラムによってはうまく並列化できないものもあります．そこで本節では，自動並列化機能を足掛かりにプログラムの並列化を行う方法を紹介します．なお，GS320 の自動並列化機能を用いて作成した OpenMP プログラムも，そのまま GP7000F で利用することができます．

前回の記事で紹介したように，自動並列化のコマンドを利用すると並列化後のソースコードが作成されます．このソースコードは，OpenMP による並列プログラムなのですが，以下に示すように通常人間が作成するプログラムに比べるとかなり複雑です．

Fortran

```
C$OMP PARALLEL SHARED (K,Q,II4,A,II5) PRIVATE (DD4,DD5,DD6,DD7,DD13,DD14
C$OMP& ,DD15,DD16,J)
C$OMP DO
          DO J=K+1,997,4
           DD5 = Q * A(II4,J)
           DD6 = Q * A(II4,J+1)
           DD7 = Q * A(II4,J+2)
           DD4 = Q * A(II4,J+3)
           DD13 = A(II5,J) - DD5
           DD14 = A(II5,J+1) - DD6
           DD15 = A(II5,J+2) - DD7
           DD16 = A(II5,J+3) - DD4
           A(II5,J+3) = DD16
           A(II5,J+2) = DD15
           A(II5,J+1) = DD14
           A(II5,J) = DD13
          END DO
C$OMP END DO NOWAIT
C$OMP END PARALLEL
```

C 言語

```
#pragma omp for nowait
                    for ( _Kii24 = k + 1; _Kii24<=1996; _Kii24+=4 ) {
                        _Kdd32 = q * a[_Kii4][_Kii24];
                        _Kdd33 = q * a[_Kii4][_Kii24+1];
                        _Kdd34 = q * a[_Kii4][_Kii24+2];
                        _Kdd31 = q * a[_Kii4][_Kii24+3];
                        _Kdd35 = a[_Kii5][_Kii24] - _Kdd32;
                        _Kdd36 = a[_Kii5][_Kii24+1] - _Kdd33;
                        _Kdd37 = a[_Kii5][_Kii24+2] - _Kdd34;
                        _Kdd38 = a[_Kii5][_Kii24+3] - _Kdd31;
                        a[_Kii5][_Kii24+3] = _Kdd38;
                        a[_Kii5][_Kii24+2] = _Kdd37;
                        a[_Kii5][_Kii24+1] = _Kdd36;
                        a[_Kii5][_Kii24] = _Kdd35;
                    }
```

これは, このコマンドが自動並列化以外にも高速化に向けたプログラムの変形を行うためです. これでは, 自動並列化後のソースコードを改良するのは困難です.

そこで, ソースコードに改良を加える場合は, 自動並列化だけを行い, 他の変形を行わないよう, 以下のコマンドを利用します.

Fortran

```
kyu-ss% kf90 -fkapargs='-conc -scalaropt=0' example.f -o example
```

C 言語

```
kyu-ss% kcc -ckapargs='-conc -scalaropt=0' example.c -o example
```

これにより, 以下に示すように元のプログラムとほとんど同じ形で並列化を行えますので, 修正等が容易です. なお, 上記のコマンドではプログラムの変形がほとんど行われませんが, 同様の効果は OpenMP プログラムをコンパイルする際に以下のコマンドを利用することにより得られます.

Fortran

```
C$OMP DO
         DO J=K+1,1000
          A(IP(I),J) = A(IP(I),J) - Q * A(IP(K),J)
         END DO
C$OMP END DO NOWAIT
C$OMP BARRIER
```

C言語

```
#pragma omp for nowait
                for ( _Kii3 = k + 1; _Kii3<=1999; _Kii3++ ) {
                    a[ip[_Kii4]][_Kii3] -=  _Kdd2 * a[ip[k]][_Kii3];
                }
#pragma omp barrier
```

4.2 自動並列化によって生成される OpenMP プログラムの特徴

自動並列化によって生成される OpenMP プログラムには, いくつかの特徴があります.

まずループですが, 必ず以下のように nowait 指示節を指定し, その直後に barrier 指示文で足並みをそろえるよう, 記述されます.

Fortran

```
C$OMP DO
         DO J=K+1,1000
          A(IP(I),J) = A(IP(I),J) - Q * A(IP(K),J)
         END DO
C$OMP END DO NOWAIT
C$OMP BARRIER
```

C言語

```
#pragma omp for nowait
                for ( _Kii3 = k + 1; _Kii3<=1999; _Kii3++ ) {
                    a[ip[_Kii4]][_Kii3] -=  _Kdd2 * a[ip[k]][_Kii3];
                }
#pragma omp barrier
```

これは, 以下のようにバリア同期を明示しない場合と同じ意味です.

Fortran

```
C$OMP DO
         DO J=K+1,1000
          A(IP(I),J) = A(IP(I),J) - Q * A(IP(K),J)
         END DO
C$OMP END DO
```

C言語

```
#pragma omp for
                for ( _Kii3 = k + 1; _Kii3<=1999; _Kii3++ ) {
                    a[ip[_Kii4]][_Kii3] -=  _Kdd2 * a[ip[k]][_Kii3];
                }
```

また, リダクション演算を行う際は必ず以下のようにループの外で critical 指示文により計算を行うよう, 記述されます.

Fortran

```
C$OMP DO
        DO J=K+1,1000
         S1 = S1 - A(IP(K),J) * X(IP(J))
        END DO
C$OMP END DO NOWAIT
C$OMP CRITICAL (II4)
        S = S + S1
C$OMP END CRITICAL (II4)
C$OMP BARRIER
```

C 言語

```
#pragma omp for nowait
            for ( _Kii6 = _Kii5 + 1; _Kii6<=1999; _Kii6++ ) {
                _Kdd3 -=  a[ip[_Kii5]][_Kii6] * x[ip[_Kii6]];
            }
#pragma omp critical
            {
                s +=  _Kdd3;
            }
#pragma omp barrier
```

これも, 以下のように reduction 指示節を指定した場合と同じ意味です.

Fortran

```
C$OMP DO REDUCTION(+:s)
        DO J=K+1,1000
         S = S - A(IP(K),J) * X(IP(J))
        END DO
C$OMP END DO
```

C 言語

```
#pragma omp for reduction(+:s)
            for ( _Kii6 = _Kii5 + 1; _Kii6<=1999; _Kii6++ ) {
                s -=  a[ip[_Kii5]][_Kii6] * x[ip[_Kii6]];
            }
```

これらは, 意味としては同じでも内部での実装のされ方が違う場合があり, 計算機によっては処理時間に影響がある可能性もあります. ただ, 九州大学情報基盤センターの GP7000F と GS320 では性能に差は見られませんでした.

4.3　自動で並列化できないプログラムの対処

GS320 の自動並列化コンパイラでは, 並列化可能かどうかが簡単に分かるループについてのみ, 並列化を行います. しかし, プログラムの作成者には並列化可能であることが分かっているループでも, 自動並列化コンパイラがそのように判断できない場合があります. 例えば, 以

下のプログラム中の2番目のループにあるような間接アクセスを含むプログラムは, 自動的に並列化できません.

Fortran

```
do i = 1, N/2
  a(i) = i
  ip(i) = i
end do

do i = N/2 + 1, N
  a(i) = a(ip(i - N/2))
end do
```

C 言語

```
for(i = 0; i < N/2; i++){
  a[i] = i * 1.0;
  ip[i] = i;
}

for(i = N/2; i < N; i++){
  a[i] = a[ip[i - N/2]];
}
```

このプログラムは, 2番目のループで間接アクセスが行われています. ここで, もし ip(i-N/2) が N/2+1 から N の間になることがあれば, そのインデクスで参照される a(ip(i-N/2)) はこのループの別の繰り返しで生成されるため, このループを並列化すると繰り返しの順序が崩れ, 正しい計算結果が得られなくなってしまいます.

しかし実際には ip は 1 番目のループで初期化されている通り 1 から N/2 の間のどれかの値しか取らないので, 問題無く並列化することができるのです. これはプログラムの作成者には分かっていることですが, コンパイラはそこまで解析できないため, 並列化不可能と判断してしまいます.

そこで, このようなループが自動並列化適用後に残ってしまったら, その出力コードをさらに変更することによって自分で並列化します. 並列化は以下のように do(for) 指示文を追加するだけで行えます.

Fortran

```
  do i = 1, N/2
    a(i) = i
    ip(i) = i
  end do

!$omp parallel do
  do i = N/2 + 1, N
    a(i) = a(ip(i - N/2))
  end do
```

C 言語

```
  for(i = 0; i < N/2; i++){
    a[i] = i * 1.0;
    ip[i] = i;
  }

#pragma omp parallel for
  for(i = N/2; i < N; i++){
    a[i] = a[ip[i - N/2]];
  }
```

一方正しい順序に繰り返さないと正しい結果が得られないループについては, 当然自動並列化はできません. しかし, 前回の記事でもいくつか紹介した通り, プログラムを書き換えることによって並列化できるようになる場合があります. また, 並列化できないループを複数連続して実行するプログラムでは, 各スレッドに一つずつループを割り当てて並列処理することが可能である場合があります.

例えば以下のループは, 図 6 に示すように, ある要素の値の計算に上下左右の要素の値を参照します. このうち上の要素と左の要素は計算が終わってなければなりません. すなわち, ある繰り返しで生成された値を後の繰り返しの計算に利用するため, このループは並列化できません.

Fortran

```
  do i = 2, n - 1
    do j = 2, n - 1
      a(i, j) = (a(i - 1, j) + a(i + 1, j) + a(i, j - 1) + a(i, j + 1)) / 4.0D0
    end do
  end do
```

C 言語

```
  for (i = 1; i< n - 1; i++)
    for (j = 1; j< n - 1; j++)
      a[i][j] = (a[i - 1][j] + a[i + 1][j] + a[i][j - 1] + a[i][j + 1]) / 4.0;
```

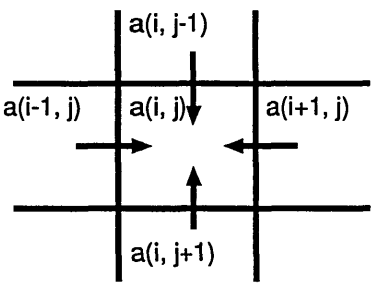

図 6: 上下左右の要素の参照

しかしこのようなループは値を更新する順番を変えると, 並列化できるようになります. 例えば a(2,2) の値を生成すると a(3,2), a(2,3) の値を計算できますが, これらはどちらを先に計算しても構いません. また, これらの計算が終わると, a(4,2), a(3,3), a(2,4) の値を計算できますが, これらもどれを先に計算しても構いません. すなわち, 値を更新する順番を図 7 に示すように変更すると, 並列に処理を進めることができるようになりました.

並列化した例は以下の通りです.

Fortran

```
  do i = 2, 2 * (n - 2)
     w = i + 2
     s = min(n - 1, i)
     e = max(2, w - n + 1)
!$omp parallel do
     do j = s, e, - 1
        a(j, w-j) = (a(j-1, w-j) + a(j+1, w-j) &
                     + a(j, w-j-1) + a(j, w-j+1)) / 4.0D0
     end do
  end do
```

C 言語

```
  for (i = 1; i< 2 * (n - 2); i++){
    w = i + 1;
    s = ((n-2)<i) ? (n-2) : i;
    e = (1>(w-n+2))? 1 : (w-n+2);
#pragma omp parallel for
    for (j = s; j >= e; j--)
      a[j][w-j] = (a[j-1][w-j] + a[j+1][w-j]
                   + a[j][w-j-1] + a[j][w-j+1]) / 4.0;
  }
```

ただしこのプログラムは, スレッド毎に割り当てられる仕事量にばらつきがあるため, 並列化の効果はそれほど高くないと予想されます.

一方, 並列化できないループが複数存在し, それらを連続して実行するプログラムで, 各ループの間で実行順序に依存が無い場合, すなわちどのループを先に実行しても結果に影響が無い場合, それぞれのループを別のスレッドで行うことにより, 並列処理することができます. これには 2 節で紹介した sections 指示文を利用します.

例えば以下のプログラムでは, 先ほどのループと同じ計算を 2 つの配列に対してそれぞれ行います.

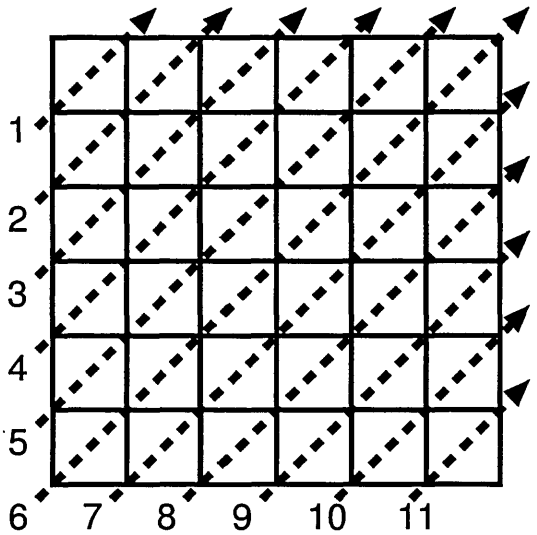

図 7: 処理の進行方向を斜めにした例

Fortran

```
do i = 2, n - 1
  do j = 2, n - 1
    a1(i, j) = (a1(i-1, j) + a1(i+1, j) + a1(i, j-1) + a1(i, j+1)) / 4.0D0
  end do
end do

do i = 2, n - 1
  do j = 2, n - 1
    a2(i, j) = (a2(i-1, j) + a2(i+1, j) + a2(i, j-1) + a2(i, j+1)) / 4.0D0
  end do
end do
```

C 言語

```
for (i = 1; i< n - 1; i++)
  for (j = 1; j< n - 1; j++)
    a1[i][j] = (a1[i-1][j] + a1[i+1][j] + a1[i][j-1] + a1[i][j+1]) / 4.0;

for (i = 1; i< n - 1; i++)
  for (j = 1; j< n - 1; j++)
    a2[i][j] = (a2[i-1][j] + a2[i+1][j] + a2[i][j-1] + a2[i][j+1]) / 4.0;
```

sections指示文を利用して並列化すると以下のようになります.

Fortran

```
!$omp parallel sections
!$omp section
    do i = 2, n - 1
      do j = 2, n - 1
        a1(i, j) = (a1(i-1, j) + a1(i+1, j) + a1(i, j-1) + a1(i, j+1)) / 4.0D0
      end do
    end do

!$omp section
    do i = 2, n - 1
      do j = 2, n - 1
        a2(i, j) = (a2(i-1, j) + a2(i+1, j) + a2(i, j-1) + a2(i, j+1)) / 4.0D0
      end do
    end do
!$omp end parallel sections
```

C言語

```
#pragma omp parallel sections
{
#pragma omp section
  {
    for (i = 1; i< n - 1; i++)
      for (j = 1; j< n - 1; j++)
        a1[i][j] = (a1[i-1][j] + a1[i+1][j] + a1[i][j-1] + a1[i][j+1]) / 4.0;
  }

#pragma omp section
  {
    for (i = 1; i< n - 1; i++)
      for (j = 1; j< n - 1; j++)
        a2[i][j] = (a2[i-1][j] + a2[i+1][j] + a2[i][j-1] + a2[i][j+1]) / 4.0;
  }
}
```

この並列化は, 各スレッドがそれぞれ同じ仕事量を受け持つので, スレッド数に対して高い並列化効果が得られると期待できます. ただし, 並列処理に参加するスレッド数は同時に実行できる仕事の数で制限されます.

5 高速化に向けて

5.1 数値計算ライブラリの利用

本節では, OpenMPプログラムの並列リージョンから呼び出し可能な数値計算ライブラリを紹介します.

5.1.1 OpenMP プログラムの並列リージョンで利用できる数値計算ライブラリ

GP7000F, GS320 には, いくつかの数値計算関数が用意されています. これらは計算機の特性に合わせてチューニングしてあるため, 容易に高い性能を得ることができます. しかし通常の関数は, OpenMP プログラムの並列リージョンから呼び出すと結果に異常が生じます. これは, 関数の中で使用する作業領域を複数のスレッドが同時にアクセスするためです.

そこで OpenMP の並列リージョン内で用いる関数としては, 予めこのような問題が生じないよう設計されたものを選択します. GP7000F, GS320 で利用できる数値計算ライブラリの内, OpenMP の並列リージョンで用いても問題のない関数を提供するものは以下の通りです.

- **BLAS, LAPACK**: (GP7000F, GS320 共通)
  米国のテネシー大学, オークリッジ国立研究所等で開発された数値計算ライブラリで, 主に密行列と特殊な疎行列についての線形計算の関数が提供されています. このライブラリはソースコードが無料で公開されているため多くの計算機で利用可能です. ただし GP7000F, GS320 で利用可能なライブラリは各計算機向けにチューニングされたものです. GP7000F 用のライブラリについては, チューニング後のソースコードは公開されていませんが, GS320 用のライブラリは Compaq 社よりソースコードが公開されています. 利用法の詳細は以下の Web ページを参照して下さい.
  `http://www.cc.kyushu-u.ac.jp/scp/system/library/LAPACK/LAPACK.html`

- **SSL II**: (GP7000F のみ)
  富士通株式会社による数値計算ライブラリで, 連立 1 次方程式, 固有値計算のような線形計算だけでなく, 数値積分やフーリエ変換等の関数も提供されています. 利用法の詳細は以下の Web ページを参照して下さい.
  `http://www.cc.kyushu-u.ac.jp/scp/system/library/SSL2/SSL2.html`

- **CXML**: (GS320 のみ)
  Compaq 社による数値計算ライブラリで, 上記の BLAS, LAPACK に加えてフーリエ変換等の関数が提供されています. 利用法の詳細は以下の Web ページを参照して下さい.
  `http://www.cc.kyushu-u.ac.jp/scp/system/manual/cxml_webpages/dxml.3dxml.html`
  (英語)

この他に GP7000F には NAG, IMSL という数値計算ライブラリが用意されています. これらは上記のライブラリに含まれていない数値計算の関数をいくつか提供しています. ただしこれらのライブラリの関数は, C 言語用の IMSL ライブラリの関数を除き, OpenMP プログラムの並列リージョンから呼び出すことはできません. これらのライブラリの利用法については以下の Web ページを参照して下さい.

- **NAG**:
  `http://www.cc.kyushu-u.ac.jp/scp/system/library/NAG/NAG.html`

- **IMSL**:
  `http://www.cc.kyushu-u.ac.jp/scp/system/library/IMSL/IMSL.html`

5.1.2 逐次版と並列版

BLAS, LAPACK, SSL II, CXML には, それぞれ逐次版と並列版の関数が用意されています. OpenMP の並列リージョンで逐次版を呼ぶと, 各スレッドがそれぞれその関数を全部実行します. そこで, スレッドによって関数に渡す配列やパラメータを変えることにより, 複数の関数呼び出しを並列に実行することができます.

一方並列版は, 関数自体が並列化されています. すなわち OpenMP を使わなくてもこの関数部分で複数のスレッドが生成され, 並列に実行されます.

また, OpenMP の並列リージョン内で並列版の関数を呼び出すと, 各スレッドからさらに複数のスレッドが生成されて並列実行されます. スレッドが生成されるときのスレッド数は環境変数 OMP_NUM_THREADS で指定された数ですから, もし全スレッドがそれぞれ並列版の関数を同時に呼び出すと, 全部で OMP_NUM_THREADS * OMP_NUM_THREADS のスレッドが作成されます.

5.1.3 GP7000F における数値計算ライブラリを用いたプログラムのコンパイルと実行

コンパイル方法は以下の通りです. C 言語の場合, 事前にプログラムの main() ルーチンを MAIN__() に変更しておきます.

Fortran

```
frt -o test test.c -KOMP,V9FMADD -SSL2
```

C 言語

```
fcc -c test.c -KV9FMADD
frt -o test test.o -KOMP,V9FMADD -SSL2
```

他の OpenMP プログラムと同様, 実行前に並列実行に参加するスレッド数を以下の通り指定します.

```
setenv OMP_NUM_THREADS スレッド数
```

また, 関数呼び出し時にさらにスレッドを生成してそれぞれの関数を並列に処理したい場合, 以下の指定も行います.

```
setenv OMP_NESTED TRUE
```

その後, 通常のプログラムと同様に実行します.

```
./test
```

5.1.4 GS320 における数値計算ライブラリを用いたプログラムのコンパイルと実行

逐次版のコンパイル方法は以下の通りです.

Fortran

```
f90 -omp -o test test.f90 -lcxml
```

C 言語

```
cc -omp -o test test.c -lcxml
```

一方, 並列版のコンパイル方法は以下の通りです.

Fortran

```
f90 -omp -o test test.f90 -lcxmlp
```

C 言語

```
cc -omp -o test test.c -lcxmlp
```

他の OpenMP プログラムと同様, 実行前に並列実行に参加するスレッド数を以下の通り指定します.

```
setenv OMP_NUM_THREADS スレッド数
```

また, コンパイル時に並列版を選択した場合で, 関数呼び出し時にさらにスレッドを生成してそれぞれの関数を並列に処理したい場合, 以下の指定も行います.

```
setenv OMP_NESTED TRUE
```

その後, 通常のプログラムと同様に実行します.

```
./test
```

5.2 並列化とメモリの参照順序

前回の記事で, ループの入れ子構造を並列化する例として以下のようなプログラムを紹介しました.

Fortran

```
do j = 2, N
!$omp parallel do
   do i = 1, N
      a(i, j) = a(i, j) + a(i, j - 1)
   enddo
enddo
```

C言語

```
for (j = 1; j < N; j++){
#pragma omp parallel for
   for (i = 0; i < N; i++){
      a[j][i] = a[j][i] + a[j - 1][i];
   }
}
```

このプログラムは内側の i のループで並列化することができます．しかしこの例ではスレッドの生成と廃棄を N 回実行します．そこで，以下のように i のループと j のループを入れ替えて，外側のループで並列化すると，スレッドの生成と廃棄の回数を 1 回に減らすことができます．このように，ループの入れ替えによって並列化によるコストを低くすることが出来る，ということも書きました．

Fortran

```
!$omp parallel do
do i = 1, N
   do j = 2, N
      a(i, j) = a(i, j) + a(i, j - 1)
   enddo
enddo
```

C言語

```
#pragma omp parallel for private(j)
for (i = 0; i < N; i++){
   for (j = 1; j < N; j++){
      a[j][i] = a[j][i] + a[j - 1][i];
   }
}
```

しかし，実際にこの二つのプログラムを実行してみたときにどちらが短時間で処理できるかは，実行する計算機におけるスレッドの生成，廃棄に必要なコストと，ループを入れ替えること

によって発生するメモリの参照順序の変更による性能へ影響との兼ね合いを考える必要があります.

多次元配列を参照するプログラムでは, どの次元に沿って連続した参照が行われるかが性能に大きく影響します. Fortran では, なるべく左側の次元に沿って参照を行うと最も効率が良くなります. 一方 C 言語, C++言語では, なるべく右側の次元に沿って参照を行うと最も効率が良くなります. 上記の例ではループの入れ替え前は効率の良い参照順序になっていましたが, ループを入れ替えることによって効率の悪い参照順序になってしまいました.

この影響は, 計算機によってまちまちですが, GP7000F, GS320 では, スレッドの生成, 廃棄のコストが非常に低く, 一方メモリの参照順序による速度の低下が大きいため, ループを入れ替えないプログラムの方が高速に実行されます. 前回のプログラムでは, 2 次元配列の行と列を入れ替える方法も紹介しましたが, GP7000F, GS320 に限って言えば, そのような方法に頼る必要もありません.

すなわち, まず最も効率の良いメモリアクセスの参照順序となるようにプログラムを作成し, その順序を乱さないように並列化する, という方針が, GP7000F, GS320 における高速化では有効のようです.

5.3 GS320 における OpenMP の性能

密行列の LU 分解や行列積で実行時間を比較したところ, GS320 の 1CPU による計算速度は GP7000F の 2 倍程度です. そのため, 全体の処理時間も基本的には GS320 の方が短くなるのですが, 並列化の効果を考えると, GS320 の方が低くなってしまいます.

並列化の効果とは, 用いた台数に応じた速度の向上が得られたかどうか, ということです. CPU の速度が 2 倍速いと, 同じプログラムでも計算時間が 1/2 になります. しかし並列化によって必要となるスレッド生成, 廃棄や同期のようなコストは, CPU の速度と関連しているとは限りません. もし双方の計算機で並列化によって必要となるコストが同じだとすると, 全体の処理時間に占める並列化によるコストの割合は, CPU 速度の遅い GP7000F の方が低くなってしまいます. 実際に GP7000F と GS320 で使用 CPU 数と処理速度の関係を見てみると, プログラムによっては, 1〜2CPU の時は 2 倍程度だった速度の比が, 4〜8CPU になると 1.5 倍程度になり, 16CPU でほとんど同じくらいの処理速度になります.

さらに GS320 は, 並列化によるコストが高くなりやすい構造になっています. GS320 には 32 個の CPU が用意されていますが, 内部では 4CPU ずつの 8 ブロックで構成されています. これらのブロックにメモリが 8GB ずつ配置され, システム全体で 64GB のメモリとなっています. 各ブロックは内部の高速ネットワークで接続されており, プログラム上は, このブロックの境界を気にすること無く, どこに配置されたデータでもアクセスできます. しかし, データへのアクセス時間を比較すると, ブロック内のデータへのアクセス時間はブロック外のデータへのアクセスの時間の約 1/3 です. すなわち, GS320 における並列化効果の向上には, なるべくブロック外へのアクセスが発生しないようにデータを配置することが重要となります.

そこで GS320 の Fortran コンパイラには, データの配置をプログラム内で指定するための特殊な機能として, `distribute` 指示文が用意されています. 例えば 2 次元配列 `x` を図 8 のように 2 次元目で分割して各 CPU に割り当てたい場合, `distribute` 指示文を用いて以下のよ

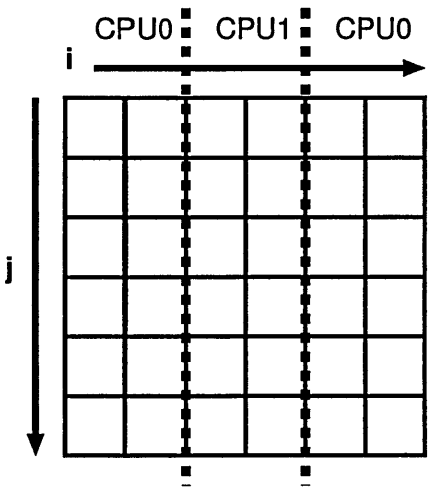

図 8: 配列の分割

うに指定します.

```
   ...
real(8), dimension(N, N) :: x
!dec$ distribute (*, block) :: x
   ...
!$omp parallel do
do i = 1, N
  do j = 1, N
     x(j, i) = i + j
  end do
end do
```

このプログラムはループ i で並列化されており, i は配列 x の 2 次元目のインデクスに利用されているので, distribute 指示文で 2 次元目を分割して各 CPU に割り当てることにより, 並列処理に参加した CPU は自分に割り当てられたデータに対してのみ書込みを行います.

この distribute 文を用いたプログラムは, 以下のコマンドでコンパイルします.

```
f90 -o test -omp -numa -numa_memories ブロック数 -numa_tmp 4 test.f90
```

ここでブロック数には, 実行時に利用するスレッド数を 4 で割って小数点以下を切り上げたものを指定します. ですから, このプログラムを実行する際に, コンパイル時に指定したブロック数に応じたスレッド数を環境変数 OMP_NUM_THREADS で指定することにより, 最も良い性能が得られます.

6　おわりに

これまで 3 回にわたって OpenMP による並列プログラミングの方法を御紹介してきました. 最初の回に書いた通り, OpenMP は共有メモリ型並列計算機で簡単に並列プログラムを作成する手段として普及しつつあります. また, 計算機アーキテクチャの流れを見ても共有メモリ型並列計算機は, 既存のベクトル型計算機の代替となるスーパーコンピュータとして導入されるなど, 高性能計算機の主流として定着しており, OpenMP の重要度は今後さらに増すことが予想されます. 本記事で OpenMP に関心を持つ方が増え, たくさんの OpenMP プログラムが本センターの計算機 (もしくは研究室等の計算機) で利用されるようになることを願っています.

参考文献

[1] Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D. and McDonald, J.: *"Parallel Programming in OpenMP,"* Morgan Kaufmann Publishers, 2000.

[2] OpenMP Architecture Review Board: *"OpenMP Fortran Application Program Interface,"* `http://www.openmp.org/specs/mp-documents/fspec10.pdf`, October 1997.