OpenMP 入門 (2)

南里 豪志 *　　渡部 善隆 *

1　はじめに

OpenMP 入門 2 回目の本記事では, ループを並列化する方法について紹介します.

一般に, プログラム中で最も多くの処理時間を要するのはループ部分です. そのため, プログラムの並列化にあたっては, まず各ループについて並列化の可能性や並列化によって得られる効果を検討するのが定石になっています. 特に本センターのスーパーコンピュータやスカラー型並列計算機等で実行されるほとんどのプログラムで計算の大半を占める数値計算では, 処理時間のほぼ全てがループの処理で占められるので, ループの並列化によって処理時間の大幅な短縮が期待できます.

また, OpenMP では, 簡単なループであれば指示文を 1 行挿入するだけで並列化できます. さらに, 最初から全体を並列化するのではなく, まず一部のループだけ並列化して効果を調べ, その後他のループについても並列化する, といった段階的な並列化も可能です.

ただし, 計算結果がループの実行順序に依存する場合, そのまま並列化すると計算結果が変わってしまいます. このようなループは, 並列化を諦めるか, 並列化のためにプログラムを書き換える必要があります. また, 処理時間を多く費やしているループを優先的に並列化することも, 効率的な高速化には不可欠です. 処理時間の短いループの並列化は効果が少ないだけでなく, 並列化によって生じるコストのためにかえって処理速度が低下することがあります.

そこで本記事では, まず並列化の対象となるループの選択方法を紹介します. その後, ループ並列化のために挿入する OpenMP の指示行について, 利用方法を説明します. また, 並列化や高速化のためにプログラムの書き換えが必要となる例を紹介します. 最後に, OpenMP を用いて並列化されたいくつかの数値計算プログラムを紹介します.

なお, 本記事は前回の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)」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・実行する方法については前回の記事を参照して下さい. また前回の記事と同様, 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.

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

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 並列化するループの選択

2.1 ループの並列化による効果

2.1.1 ループの理想的な並列処理を妨げる要因

あるループを複数のスレッドが分担して処理する場合, そのループを並列ループと呼びます. また, あるループを並列ループとして指示することをループの並列化と呼びます. 前回の記事に書いたように, 基本的に並列ループを実行する各スレッドは指定された範囲の処理をそれぞれ独立に進めます. そのため, スレッド数を T とすると, ループの並列化によって, 理想的にはループの処理時間を $1/T$ に短縮できます.

しかし実際の並列ループでは, 以下のような理由によって処理時間が並列化前の $1/T$ より長くなります.

プロセッサへのスレッド割り当て

スレッドは仮想的な処理の流れであり, スレッドにプロセッサを割り当てることによって処理が進行します. この, スレッドにプロセッサを割り当てる方法は, 実行する環境によって異なります. 例えば並列ループのスレッド数より計算機のプロセッサ数が少ない場合や, 他のプロセスの影響で実行時に十分なプロセッサ数を確保できない場合, 多くの OS(Operating System) は複数のスレッドに 1 つのプロセッサを割り当てます. しかし, 1 プロセッサが同時に処理できるのは 1 スレッドのみなので, 割り当てられた全部のスレッドの処理に要する時間は, それらの処理を非並列に実行した場合に要する時間より短くなることはありません. そのため, 複数スレッドに 1 つのプロセッサが割り当てられる場合, 並列ループ全体の処理時間は $1/T$ より長くなります[1].

共有データへの排他的なアクセス

複数のスレッドが同じ記憶場所 (同じ変数や配列の同じ要素) を共有し, 並列ループの中でそれぞれ書き込みを行なう場合, スレッド間で競合が起こります. 競合が起こるプログラムで正しい計算を行なうためには, その場所に対して各スレッドが排他的にアクセスするよう, 制御する必要があります. その結果, あるスレッドが書き込みの権利を得ると, その書き込み処理が終了するまで他のスレッドは待つことになります. この待ち時間は, 並列化する前は必要なかったものなので, 並列ループ全体の処理時間が長くなります.

スレッドの生成と廃棄

OpenMP では, 並列リージョンの最初にスレーブスレッドを生成し, 並列リージョンの最後にスレーブスレッドを廃棄します. この生成と廃棄に要するコストも, 並列化する

[1]最近, 1 プロセッサ内で複数のスレッドを同時に処理するアーキテクチャが開発されています. このアーキテクチャを用いた計算機では, 複数のスレッドに 1 つのプロセッサが割り当てられた場合の影響が, 従来のアーキテクチャを用いた計算機より小さくなると期待できます.

前は必要無かったものなので, 結果的に並列ループ全体の処理時間は $1/T$ より長くなります.

不均等な負荷分散

ループの繰り返し数がスレッド数で割りきれない場合や, 各繰り返しで処理時間が異なる場合, スレッド毎の処理時間が不均等になります. その結果, 早く処理を済ませたスレッドが他のスレッドを待つので, 並列ループ全体の処理時間が長くなります.

これらの要因はループの構造によって影響の度合が異なります. すなわち, 並列化の効果が得られやすいループと得られにくいループが存在します. ループによっては, 上記のような要因が重なることによって, 並列ループの処理時間が並列化前の処理時間より長くなることもあり得ます. そのため, 並列化を施すループの選択は重要です.

2.1.2 ループの並列化によるプログラム高速化の効果

並列化を施すループの選択において重要なのは, ループの並列化によって得られる効果の見積もりです. あるループを並列化しても, そのループの処理時間がプログラム全体の処理時間に占める割合が少なければ, プログラム全体の並列化による効果は少ないか, もしくは並列化によって生じるコスト増のためにかえって全体の処理速度が低下します.

あるループの処理時間がプログラム全体の処理時間に占める割合を P $(0 < P \leq 1)$, 並列処理に参加するスレッド数を T とし, そのループが理想的に並列実行できると仮定すると, 並列化によってプログラム全体の処理時間は並列化前の $(1-P)+\frac{P}{T}$ 倍になります. このため, 並列化によって得られる性能向上率は $\frac{1}{(1-P)+\frac{P}{T}}$ になります. ただし, プログラム中の複数のループを並列化する場合は, 各ループの並列化前の処理時間の合計がプログラム全体の処理時間に占める割合を P とします.

例えばスレッド数 $T=8$ とし, P が 0.9, すなわち全体の処理時間の 90% を占めるループを理想的に並列化できた場合, 並列化の効果は 4.7 となります. 一方, P が 0.5, すなわち全体の処理時間の 50% を占めるループを並列化すると, 並列化の効果は 1.7 となります. このように, プログラム全体の処理時間に占める割合が多いループを並列化するほど, 並列化の効果が高くなります.

2.2 プログラムの性能分析

前節より, 並列化するループの選択において, そのループの処理時間がプログラム全体の処理時間に占める割合が重要であることが分かりました. そこで, 当センターのスカラー並列サーバと汎用 UNIX サーバでプログラムの性能を分析する方法を紹介します.

2.2.1 スカラー並列サーバ GS320 における性能分析

GS320 では, `pixie` コマンドによってプログラムの性能を分析できます.

まず, プログラムをコンパイルして実行ファイルを生成します. この際, プログラムが C 言語の場合は cc コマンドに-g オプションをつけます. 一方プログラムが Fortran の場合は通常通り f90 コマンドでコンパイルします.

```
kyu-ss% f90 example.f90 -o example
```

```
kyu-ss% cc -g example.c -o example
```

その後, 生成された実行ファイルに対して pixie コマンドを適用します. 例えば, example という実行ファイルを解析し, 結果を log というファイルに保存する場合, 以下のように実行します.

```
kyu-ss% pixie -procedures -lines example > log
```

この結果 log に格納される実行結果は図 1 のようになります.

「手続き毎の消費時間」の部分で重要なのは以下の 2 項目です.

cycles	各手続きの中で消費された時間
%cycles	その時間がプログラム全体の消費時間に占める割合

この例では, power_method 手続きで約 73%, sample_of_power_method 手続きで約 27%の時間が消費されています.

また, 「行毎の消費時間」の部分は予め消費時間の長い手続きの順番に並べ替えられています. この中では以下の項目を見ます.

line	行番号
cycles	各行の実行で消費された時間
%	その時間がプログラム全体の消費時間に占める割合

これらの情報をもとに消費時間の長いループを探し出し, 並列化を検討します.

2.2.2 汎用 UNIX サーバ GP7000F における性能分析

GP7000F では, coll コマンドと samp コマンドによってプログラムの性能を分析できます. まず, プログラムをコンパイルして実行ファイルを生成します. この際, プログラムが C 言語の場合は fcc コマンドに-Kline オプションをつけます. 一方プログラムが Fortran の場合は通常通り frt コマンドでコンパイルします.

```
kyu-cc% frt example.f90 -o example
```

```
kyu-cc% fcc -Kline example.c -o example
```

次に coll コマンドによって性能情報を収集します. 例えば実行ファイル example の性能情報を samp.dat に格納するには以下のように実行します.

```
 iteration count=           5
 approximate eigenvalue=   2000.07950633774

Profile listing generated Sun Jan  6 16:35:35 2002 with:
   prof -pixie -all -procedures -lines ./a.out ./a.out.Counts

----------------------------------------------------------------------------
*  -p[rocedures] using basic-block counts;                                 *
*  sorted in descending order by the number of cycles executed in each     *
*  procedure; unexecuted procedures are excluded                           *
----------------------------------------------------------------------------

827878749 cycles (1.1329 seconds at 730.79 megahertz)

    cycles %cycles  cum %  seconds     cycles  bytes procedure (file)
                                        /call  /line

 603830365   72.94  72.94   0.8263  603830365    155 power_method_ (power_method.f90)
 224048065   27.06 100.00   0.3066  224048065     73 sample_of_power_method_ (power_method.f90)
       179    0.00 100.00   0.0000        179      ? __start (<a.out>)
        49    0.00 100.00   0.0000         49      ? __INIT_00_add_gp_range (<a.out>)
        38    0.00 100.00   0.0000         38      ? __FINI_00_remove_gp_range (<a.out>)
        21    0.00 100.00   0.0000         21     21 main (for_main.c)
        17    0.00 100.00   0.0000         17      ? __INIT_00_add_pc_range_table (<a.out>)
        15    0.00 100.00   0.0000         15      ? __FINI_00_remove_pc_range_table (<a.out>)

----------------------------------------------------------------------------
*  -p[rocedures] using invocation counts;                                  *
*  sorted in descending order by number of calls per procedure;            *
*  unexecuted procedures are excluded                                      *
----------------------------------------------------------------------------

8 invocations total

     calls  %calls    cum%    bytes procedure (file)

         1   12.50   12.50       60 __FINI_00_remove_pc_range_table (<a.out>)
         1   12.50   25.00      220 __INIT_00_add_gp_range (<a.out>)
         1   12.50   37.50       84 main (for_main.c)
         1   12.50   50.00      364 sample_of_power_method_ (power_method.f90)
         1   12.50   62.50      200 __start (<a.out>)
         1   12.50   75.00     5416 power_method_ (power_method.f90)
         1   12.50   87.50      184 __FINI_00_remove_gp_range (<a.out>)
         1   12.50  100.00       68 __INIT_00_add_pc_range_table (<a.out>)

----------------------------------------------------------------------------
*  -l[ines] using basic-block counts;                                      *
*  grouped by procedure, sorted by cycles executed per procedure;          *
*  '?' means that line number information is not available.               *
----------------------------------------------------------------------------

procedure (file)                           line bytes     cycles      %  cum %

power_method_ (power_method.f90)             25    68         13   0.00   0.00
                                             33   116         29   0.00   0.00
                                             37   372         93   0.00   0.00
                                             42   104      52013   0.01   0.01
                                             45     4          1   0.00   0.01
                                             46   100       4583   0.00   0.01
                                             47   212      12585   0.00   0.01
                                             49   148       6014   0.00   0.01
                                             50   204      13005   0.00   0.01
                                             54    48         14   0.00   0.01
                                              •
                                              •
                                              •
```

プログラムの出力

手続き毎の消費時間

手続きの呼出回数

行毎の消費時間

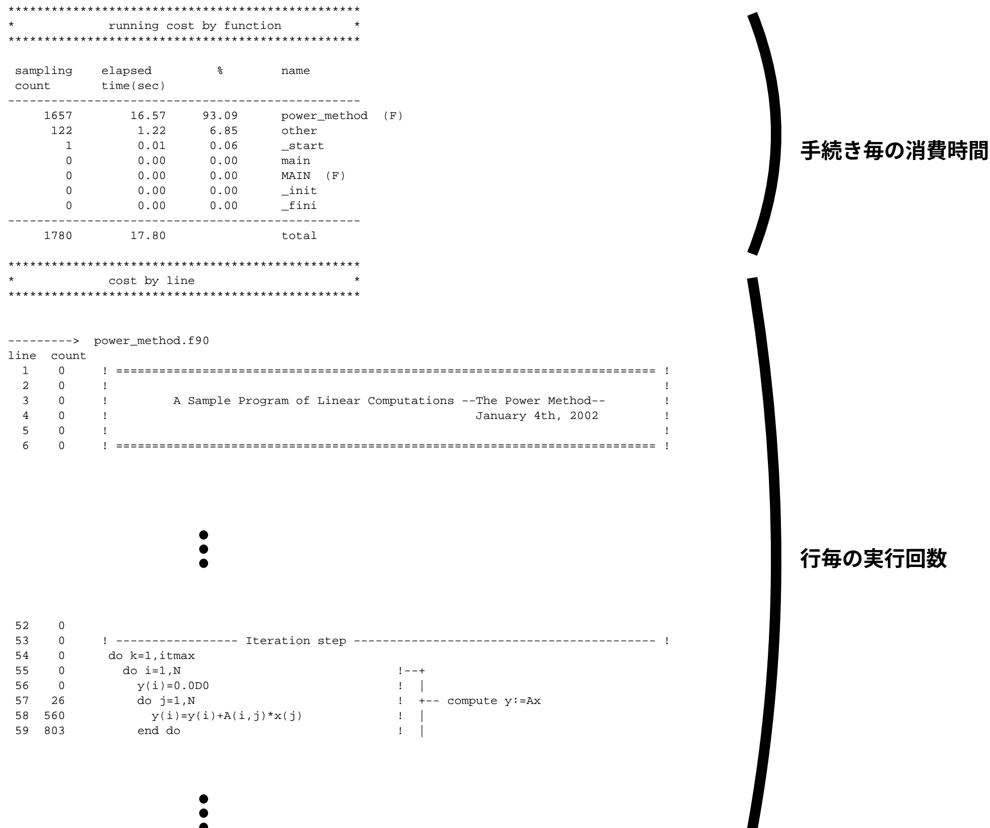
図 1: pixie コマンドの出力

```
kyu-cc% coll -d samp.dat example
```

その後, 得られた性能情報を samp コマンドで表示します. 性能情報を log という名前のファイルに格納したい場合は以下のように実行します.

```
kyu-cc% samp -d samp.dat -f -l example > log
```

この結果 log に格納される実行結果は図 2 のようになります.



```
************************************************
*            running cost by function          *
************************************************

 sampling    elapsed         %        name
 count       time(sec)
------------------------------------------------
     1657        16.57     93.09      power_method  (F)
      122         1.22      6.85      other
        1         0.01      0.06      _start
        0         0.00      0.00      main
        0         0.00      0.00      MAIN  (F)
        0         0.00      0.00      _init
        0         0.00      0.00      _fini
------------------------------------------------
     1780        17.80                total

************************************************
*            cost by line                      *
************************************************


--------->  power_method.f90
line  count
  1     0     ! ===================================================================== !
  2     0     !                                                                       !
  3     0     !         A Sample Program of Linear Computations --The Power Method--  !
  4     0     !                                              January 4th, 2002        !
  5     0     !                                                                       !
  6     0     ! ===================================================================== !

 52     0
 53     0     ! ---------------- Iteration step ----------------------------------------- !
 54     0      do k=1,itmax
 55     0        do i=1,N                             !--+
 56     0          y(i)=0.0D0                         !  |
 57    26          do j=1,N                           !  +-- compute y:=Ax
 58   560            y(i)=y(i)+A(i,j)*x(j)            !  |
 59   803          end do                             !  |
```


図 2: samp コマンドの出力

GS320 と同様に「手続き毎の消費時間」の部分で各手続きの中で消費された時間と, その時間がプログラム全体の消費時間に占める割合がそれぞれ出力されます. この例では, power_method 手続きで 93%の時間が消費されています. また, 「行毎の実行回数」によって各行の実行回数が分かります.

これらの情報をもとに実行回数の多いループを探し出し, 並列化を検討します.

2.3 入れ子構造のループの並列化

あるループの中に別のループが含まれる構造を「ループの入れ子構造」と呼びます．また,入れ子構造になっているそれぞれのループを, 相互の位置関係から相対的に外側ループ, 内側ループと呼びます．OpenMP でループの入れ子構造を並列化する際, 並列化されるループは並列化を明示的に指示したもののみ, すなわち, OpenMP の並列化指示文の直後のループのみです.

例えば図 3 のプログラムでは外側ループを並列化しています．この場合, 各スレッドがそれぞれ内側ループを最初から最後まで実行します．一方, 図 4 のプログラムでは内側ループを並列化しています．この場合, 外側のループはマスタースレッドが逐次的に実行します．マスタースレッドはスレーブスレッドを生成し, 内側ループを並列実行した後, スレーブスレッドを廃棄する繰り返しを, 外側ループの繰り返し回数分行ないます.

一般にループの入れ子構造を並列化する場合, なるべく外側のループを並列化した方が効率が良くなります．これは, スレッドの生成, 廃棄の回数を減らすことができるためです．ただし, 2.4 節で説明する依存関係のため, 外側のループを並列化できない場合があります．この場合は内側ループの並列化を検討するか, 並列化を諦めます．また, 5.2 節で説明するように, プログラム構造の変更によって外側ループを並列化できる場合もあります.

Fortran

```
subroutine sum_mat(a, b, c)
  real(kind(1.0D0),dimension(N,N),intent(in)  :: a, b
  real(kind(1.0D0),dimension(N,N),intent(out) :: c
  integer :: i, j

!$omp parallel do
  do i = 1, 100
    do j = 1, 100
      c(j, i) = a(j, i) + b(j, i)
    end do
  end do

end subroutine sum_mat
```

C 言語

```
void sum_mat(double a[][100], double b[][100], double c[][100])
{
  int i, j;

#pragma omp parallel for private(j)
  for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
      c[i][j] = a[i][j] + b[i][j];
}
```

図 3: 入れ子構造の外側ループの並列化

Fortran

```
subroutine sum_mat(a, b, c)
  real(kind(1.0D0),dimension(N,N),intent(in)  :: a, b
  real(kind(1.0D0),dimension(N,N),intent(out) :: c
  integer :: i, j

  do i = 1, 100
!$omp parallel do
    do j = 1, 100
       c(j, i) = a(j, i) + b(j, i)
    end do
  end do
  end subroutine sum_mat
```

C 言語

```
void sum_mat(double a[][100], double b[][100], double c[][100])
{
  int i, j;

  for (i = 0; i < 100; i++)
#pragma omp parallel for
    for (j = 0; j < 100; j++)
      c[i][j] = a[i][j] + b[i][j];
}
```

図 4: 入れ子構造の内側ループの並列化

2.4 並列化できないループ

2.4.1 ループの繰り返し間の依存関係

前回も紹介しましたが, 以下のループを並列化すると正しい結果が得られなくなります.

Fortran

```
do i = 2, 100
   z(i) = z(i) + z(i - 1)
end do
```

C 言語

```
for (i = 1; i < 100; i++)
   z[i] = z[i] + z[i - 1];
```

このループでは, `z(i)` の要素を計算するために, `i` ループの一つ前の繰り返しで算出した値 `z(i-1)` を利用するので, 並列化によって繰り返しの実行順序が変わると, 並列化前と異なる値が得られるようになってしまうのです.

一般に, プログラム中のある二つの処理の実行順序を入れ換えることによってプログラムの実行結果が変化する場合, これら二つの処理の間に「依存関係がある」といいます. 通常, プログラム中のある二つの処理がメモリ上の同じ場所に対してアクセスをしており, 少なくとも一方の処理が書き込みである場合, これら二つの処理の間には依存関係があります.

このうち, あるループの異なる繰り返しで実行される二つの処理の間の依存関係を, 「ループの繰り返し間の依存関係」と呼びます. この, ループの繰り返し間の依存関係がないループは, OpenMP の並列化指示文を挿入するだけで並列化することができます. 逆に, ループの繰り返し間の依存関係があるループは, 並列化できないか, 並列化可能であっても多少の工夫が必要になります. そのため, 簡単に並列化できるループだけ OpenMP を使ってとりあえず並列化したい, という場合は, このようなループの並列化は後回しにした方がいいでしょう.

2.4.2 依存関係の判別

ループの繰り返し間の依存関係は, ループ内でアクセスされる変数への書き込みと参照の順序を調べることによって判別します. ループのある繰り返しで書き込まれた値が他の繰り返しで読まれる場合と, 異なる繰り返しで同じ場所に値が書き込まれる場合, そのループには繰り返し間の依存関係が存在します.

ただし, ループの構造が複雑で依存関係の有無を判定するのが難しい場合, 自動並列化機能を持つコンパイラを利用して判定する, という方法があります. 最近のコンパイラには, プログラムを解析して自動的に並列化を行なう機能を有するものが増えています. 特に本センターのスカラー並列サーバ GS320 に用意されている自動並列化コンパイラを利用すると, 非並列のプログラムを解析し, 並列化可能なループに OpenMP の指示文を自動的に挿入することが出来ます. また, このコンパイラは OpenMP の指示文を挿入したソースプログラムだけでなく, プログラムの解析結果も出力します. そこで, この機能を使ってループの並列化が可能か

否かを判別する方法を紹介します.

なお, この機能はプログラム並列化の前処理としても利用できます. ただし, インデクス配列を介した間接アクセスを含むループや, 外部の関数の呼び出しを含むループは, 基本的に自動並列化できません. また, このコンパイラの指示文や指示節の挿入方法が必ずしも最適であるとは限りません. 自動並列化機能を用いた場合との性能比較は次回の記事で行ないます.

GS320 の自動並列化コンパイラは, 以下のように kf90 コマンドで利用できます.

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

```
kyu-ss% kcc -fkapargs='-conc' example.c -o example
```

その結果, 並列化後のソースコード example.cmp.f (C 言語の場合, example.cmp.c), プログラムの解析結果 example.out, 及び並列化後のソースコードをコンパイルして得られる実行コード example が生成されます.

このうち, example.out の先頭に以下のような解析結果が出力されます. この中で, Actions 欄に並列化可能か否かが表示されるので, 並列化をしたいループについてこの欄を参照します.

```
 KAP/Tru64_U_F90                 4.3 k3105171 000607       HELLO             S
ource          13- 1-2002  15:45:54      Page   1

 Footnotes      Actions        DO Loops    Line

                                               1         program hello
                                               2         integer i
                                               3         double precision a(0:99)
                                               4
 1              SO             +---------      5         do i = 1, 10
 2              DD             !               6            a(i) = a(i) + a(i - 1)
                               !_________      7         end do
                                               8         end



 Abbreviations Used
  DD       data dependence
  SO       scalar optimization
```

もしループの最初の行に C もしくは NC と出力されていれば, そのループには繰り返し間の依存関係はありません. すなわち, そのループを並列化することができます. また, 最初の行に C と出力されているループは, example.cmp.f の中で並列化されています. ちなみに最初の行に NC と出力されているループは, 繰り返し回数が少なすぎて並列化の効果が得られそうにない, 等の理由で並列化が行なわれなかったループです.

一方, ループの最初の行に C や NC が出力されておらず, ループの範囲内に DD と出力されている行がある場合は, そのループに繰り返し間の依存関係が無いとは判断できなかったことを示しています. 例えばインデクス配列を介した間接アクセスを含むループは, インデクス配列の中身が分からなければ依存関係の有無を判別できないため, ループの最初の行には C や NC

は出力されません.

2.5 並列化を指示できないループ

OpenMP では, コンパイラの処理を容易にするため, 「実行時, ループの処理に入る前にループ全体の繰り返し数を計算できる」ようなループに限り並列化の指示を許可しています. これは, ループ全体の繰り返し数が分からなければスレッドへのループ割り当てが行なえないためです.

具体的には, OpenMP では正規形であるループ, すなわち, ループを制御する変数を `var` とすると, 以下の形であるループのみ, 並列化を指示することができます.

Fortran

```
do var = 初期値, 終了値 [, ストライド]
```

C/C++

```
for (var = 初期値; var 条件 終了値; 増分式)
```

ただし, ループを制御する変数, 初期値, 終了値及びストライドは全て整数でなければなりません. また, 初期値, 終了値, ストライドは変数を含む式でも構いませんが, ループの実行中に値が不変でなければなりません.

それから, 終了値に到達する前に中断されることのあるループも並列化できません. ただし, ループのある繰り返しで Fortran の cycle や C/C++ の continue によって次の繰り返しに移ったり, Fortran の stop や C/C++ の exit によってプログラム全体を中断するのは構いません. また, C++ でループ実行中に同じループ内の try ブロックに catch されるような例外処理を raise しても構いません.

なおこれらの条件は, 並列ループをスレッドに分割するために必要な情報が得られるかどうかを判別するもので, そのループに依存関係があるか否かを判別するものではありません. OpenMP では依存関係に関する判別は行ないませんので, 注意して下さい. 依存関係をチェックするのは, 前節で紹介した自動並列化機能を持つコンパイラのみです.

3 データの扱いの指示

前回の記事に書いたように, 並列実行中にアクセスされる変数は共有変数かプライベート変数として扱われます. 共有変数は, あるスレッドが書き込んだ値を他の全スレッドが参照できます. 一方プライベート変数は各スレッドがそれぞれ別の値を持つ変数であり, あるスレッドが書き込んだ値を他のスレッドが直接参照することはできません.

共有変数はスレッド間の情報交換に利用できるため便利なのですが, 全ての変数を共有変数にすれば良いと言うわけではありません. 各変数の用いられ方に応じて適切にプライベート変数か共有変数かを決定しなければ, 正しく並列実行できないので注意して下さい.

OpenMP では変数の扱われ方をスコープ指示節で指示します．スコープ指示節には以下の種類があります．

- private(3.3 節)
- firstprivate(3.4 節)
- lastprivate(3.5 節)
- shared(3.6 節)
- default(3.7 節)
- reduction(3.8 節)
- copyin(3.9 節)

これらの指示節は, `parallel for` 等の OpenMP の並列化指示文の後に追加することにより, その並列リージョン内での変数の扱われ方を指示します．1 つの指示文中に複数のスコープ指示節があっても構いませんが, 変数名の重複があってはなりません．各変数とも高々一つの指示節にのみ含まれるようにします．ただし, firstprivate 指示節と lastprivate 指示節については両方に同じ変数を記述しても構いません．また, スコープ指示節の指示が適用されるのは並列ループ指示文で並列化される領域に直接書かれているアクセスのみです．並列化される領域内で呼び出されるサブルーチンの中のアクセスには適用されません．

3.1 スコープ指示節に含まれない変数の扱い

OpenMP では, スコープ指示節に含まれない変数が並列リージョンの中でアクセスされる場合, 基本的に共有変数として扱われます．ただし, 4 節で説明する並列ループの制御変数は, 何も指定しなくてもプライベート変数となります．また, 並列リージョンから呼び出される手続きの中で宣言される変数も, 基本的にプライベート変数となりますが, Fortran の save 属性や common 文, C/C++ の static で定義された変数は共有変数となります．

なお, C/C++ の並列リージョンから呼び出される手続きに値渡しで渡される引数は, 指示がなければプライベート変数となります．一方 Fortran の関数に渡される引数や C/C++のポインタで渡される引数のように参照渡しで渡される引数は, 呼び出し側の変数の扱われ型を引き継ぎます．また, C/C++ では並列リージョンの途中で変数を定義することができますが, このような変数もプライベート変数となります．

3.2 threadprivate 指示文

前節で説明した通り, 基本的に OpenMP では, スコープ指示節で扱いを指示しない変数は共有変数となります．しかし, 大域変数や common ブロックをプライベート変数として扱いたい場合, 並列リージョン開始の度にスコープ指示節を書くのは面倒です．特に MPI で書かれた並列プログラムを OpenMP で書き換える場合, ほとんどの大域変数や common ブロックをプライベート変数として扱うように指示する必要があります．

そこで OpenMP には, 大域変数や common ブロックを一括してプライベート変数として扱うよう指示する threadprivate 指示文が用意されています．threadprivate 指示文の利用法は以下の通りです．なお, 名前 1, 名前 2, ... には Fortran の場合は common ブロック名を, C/C++

の場合は大域変数名を記述します.

Fortran

```
!$omp threadprivate(名前 1[, 名前 2 ...])
```

C/C++

```
#pragma omp threadprivate(名前 1[, 名前 2 ...])
```

threadprivate 指示文に指示する common ブロックや大域変数はその前に定義されていなければなりません. また, threadprivate 指示文に指示した common ブロックや大域変数を, copyin 指示節以外のスコープ指示節に記述することはできません.

3.3 private 指示節

前回の記事で紹介した通り, private 指示節は並列リージョン内でプライベート変数として扱う変数を指示します.

private 指示節の利用法は以下の通りです. 名前 1, 名前 2, ... には変数名や common ブロック名を記述します.

```
private(名前 1[, 名前 2 ...])
```

private 指示節に含まれる変数は, 並列リージョン実行中各スレッドがそれぞれ別の値を持ちます. また, private 指示節に書かれた変数が並列ループ開始時に取る初期値は不定ですし, 並列実行終了後, マスタースレッドでこれらの変数が取る値も不定です.

プライベート変数の初期値を設定したい場合は, firstprivate 指示節 (3.4 節) を, 並列実行終了時の値を逐次リージョンで参照したい場合は, lastprivate 指示節 (3.5 節) を利用します.

3.4 firstprivate 指示節

3.3 節で述べた通り, プライベート変数の初期値は不定です. これは, データの移動に要するコストをなるべく少なくするという点で有効です. しかし, プライベート変数の初期値として直前の逐次リージョンにおけるマスタースレッドでの値を使いたい場合があります. そのような場合にこの firstprivate 指示節を用います.

firstprivate 指示節の利用法は以下の通りです. 名前 1, 名前 2, ... には変数名や common ブロック名を記述します.

```
firstprivate(名前 1[, 名前 2 ...])
```

firstprivate 指示節に記述された変数はプライベート変数として扱われ, マスタースレッドにおける並列リージョン直前の値で各スレッドの変数が初期化されます. 基本的にある変数は高々 1 つのスコープ指示節にしか記述できませんが, 例外的にこの firstprivate 指示節と, 3.5

節で紹介する lastprivate 指示節の双方に同じ変数を記述しても構いません．この場合, その変数は初期化と後処理の両方が行なわれます．

ただし firstprivate 指示節による初期化は, 各スレッド毎に並列リージョンの最初の一回だけ行なわれます．例えば, 並列ループ指示文 (4 節) の firstprivate 指示節に記述された変数が並列ループ内で書き換えられると, そのスレッドのそれ以降の繰り返しでは初期値を参照できません．

例えばループ中で全ての要素に書き込みが行われる配列を firstprivate 指示節に指示した場合, どのスレッドが初期値を参照するかを予測することは困難です．これはループ中で書き込みが行われる変数についても, 同じことが言えます．一方, ループ中で書き込みが行われない配列や変数は, プライベート変数にする必要がありません．そのため並列ループで firstprivate 指示節を利用するのは, ループ中で一部の要素にのみ書き込みが行なわれる配列に対してであることがほとんどです．

なお C++では, firstprivate 指示節にオブジェクトが記述された場合, そのオブジェクトのコンストラクタを用いて初期化します．

3.5 lastprivate 指示節

あるプライベート変数について, 並列ループ終了後の逐次リージョンを実行するマスタースレッドにループの最後の繰り返し実行後の値をコピーしたい場合があります．このような場合に lastprivate 指示節を用います．

lastprivate 指示節の利用法は以下の通りです．名前 1, 名前 2, ... には変数名や common ブロック名を記述します．

```
lastprivate(名前 1[, 名前 2 ...])
```

lastprivate 指示節に記述された変数もプライベート変数として扱われ, 並列ループ実行後, 最後の値をマスタースレッドの同名の変数にコピーします．この, マスタースレッドにコピーされる値は, ループを逐次実行した場合の最後の繰り返しを担当するスレッドにおける, 並列ループ実行後の同名の変数の値です．

なお, lastprivate 指示節に配列や構造体が記述された場合, 最後の繰り返しで値が書き込まれる要素以外は値が不定となります．また C++ で, lastprivate 指示節に指定されたクラスのデータ型を持つ変数は, アクセス可能で曖昧でないコピーコンストラクタを持つ必要があります．

3.6 shared 指示節

前回の記事で紹介した通り, shared 指示節は並列リージョン内で共有変数として扱う変数を指示します．

shared 指示節の利用法は以下の通りです．名前 1, 名前 2, ... には変数名や common ブロック名を記述します．

```
shared(名前1[, 名前2 ...])
```

shared 指示節に含まれる変数は, 共有変数として並列実行中全スレッドから共有されます. 共有変数は逐次リージョンと並列リージョンでメモリ上の格納場所が変化しないので, 格納された値も継続して利用できます. ただし OpenMP では, 3.1 節で説明したようにほとんどの変数は特に指示がなければ共有変数として扱われるため, この指示節を用いる必要がある状況はそれほど多くありません.

なお, もし C/C++ のプログラム中でポインタ変数が共有される場合, ポインタ変数に格納されたアドレス値は共有されますが, そのアドレス値によって指定されるメモリ上の場所が各スレッドで同じであるとは限りません.

3.7 default 指示節

default 指示節の利用法は以下の通りです.

```
default(none または private または shared)
```

スコープ指示節で扱いが指示されていない変数の扱いを変更したい場合, default 指示節を用います. Fortran では `default (private)` 指示節をプログラムに記述すると, それ以降の並列リージョンにおいて, 指示のない変数は全てプライベート変数として扱われます. これは, プログラム中の大部分の変数をプライベート変数として扱いたい場合に便利です. 例えば MPI プログラムを OpenMP に移植する場合, 基本的に変数をプライベート変数にした方がプログラムの書き換えが容易になります. ただし C/C++ では, 標準関数の内部で利用される大域変数がプライベート変数として扱われると誤動作に繋がるため, `default (private)` 指示節を用いることはできません.

一方 C/C++, Fortran 双方で `default (none)` 指示節を付けると, スコープ指示節に含まれない変数を並列リージョン内で利用できなくなります. ただし並列ループを制御する変数だけは, スコープ指示節に含まれていなくてもプライベート変数として扱われます. この指示節は, 全ての変数が正しい扱いになっていることを確認したい場合に用います.

3.8 reduction 指示節

ベクトルの総和や最大値の探索のように, 複数の値から一つの値を算出する計算をリダクション計算と呼びます. このリダクション計算を行なっている部分を並列化する際は, 前回の記事に書いた通り, 最終的な結果を格納する変数をリダクション変数として指示します. このリダクション変数の指示には reduction 指示節を利用します.

リダクション変数は並列リージョン中ではプライベート変数として扱われます. 並列リージョンで各スレッドが生成したリダクション変数の値は, 演算の種類に応じてループの最後に取りまとめられ, 逐次リージョンの同名の変数に格納されます.

reduction 指示節は以下の形で用います.

```
reduction(演算子: 変数名１ [, 変数名２ ... ])
```

reduction 指示節で指示できる演算子は表 1 に示す通りです. 表中の初期値とは, reduction 指示節で指示された変数に並列ループ開始時に格納される値です.

これらの演算子の中からリダクション計算の内容に合った演算子を選択します. 通常, reduction 指示節は並列ループ中で以下の形の計算を行なう場合に用いるので, その演算子を指示します.

Fortran

```
x = x 演算子 式
x = 式 演算子 x (except for subtraction)
x = 組み込み手続き (x,expr)
x = 組み込み手続き (expr, x)
```

C/C++

```
x = x 演算子 式
x 演算子= 式
x = 式 演算子 x
x++
++x
x--
--x
```

また, 変数名にはスカラー変数を記述します. すなわち, 配列変数やポインタは記述できません. もし, ある配列要素についてリダクション計算を行なうループを並列化する場合は, 一旦スカラー変数を用いてリダクション計算を行ない, その結果を目的の要素に代入するよう, プログラムを変更する必要があります.

なお, リダクション計算を行なうループを並列化する場合, 丸め誤差に注意して下さい. 実数のリダクション計算を行なうループを並列化した場合, 計算結果が並列化前と異なる場合があります. これは, 計算順序が変化することによって丸め誤差が変化するためです. 並列化の前後で生じる計算結果の差が大きすぎる場合, アルゴリズムを変更するか, 並列化を諦める必要があります.

それから, `-`, `&&`, `||` を演算子とするリダクション計算を並列実行する場合も注意が必要です. まず `-` は, `+` や `*` と違ってそのままではリダクション計算になりません. プログラムの意味に応じて並列ループ中の演算子を `+` に変更する等の調整が必要です. また, `&&` や `||` で連結された論理式は, 途中までの評価で最終的な結果が判明することがあります. 例えば論理式 `(a && b)` の値は, `a` が 0 であれば `b` の値に関わらず 0 です. このように途中で論理式全体の値が判明する場合, C/C++ では以降の評価を行ないません. しかしこのような論理式を用いるリダクション計算を並列化すると, 評価の順番が変わるため, 並列化前には評価されなかった式まで評価される可能性があります. この, 並列化によって評価されるようになった式が副作用のある処理を含む場合, 並列化の前後で実行結果が変わります. そのため, 副作用を含む論理式のリダクション計算は, 副作用を含まない形に変更してから並列化するか, 並列化を諦

表 1: reduction 指示節で指示できる演算子

Fortran 演算子	初期値	C/C++ 演算子	初期値
+	0	+	0
*	1	*	1
-	0	-	0
.AND.	.TRUE.	&	全ビット 1
.OR.	.FALSE.	|	0
.EQV.	.TRUE.	^	0
.NEQV.	.FALSE.	&&	1
MAX	表現可能な最小の値	||	0
MIN	表現可能な最大の値		
IAND	全ビット 1		
IOR	0		
IEOR	0		

める必要があります.

3.9 copyin 指示節

threadprivate 指示文で指示された変数は通常初期値を持ちませんが, copyin 指示節によって初期値を指定することができます.

copyin 指示節の利用法は以下の通りです. 名前 1, 名前 2, ... には変数名や common ブロック名を記述します.

```
copyin(名前 1[, 名前 2 ...])
```

threadprivate 指示文で指示された変数名や common ブロック名を copyin 指示節で指示すると, それらのマスタースレッドにおける並列リージョン開始時の値が初期値として全スレッドにコピーされます.

ただし, copyin 指示節に記述する変数や common ブロックは予め threadprivate 指示文に記述されており, さらに並列リージョンの実行前にマスタースレッドにおける値が確定していなければなりません.

4 並列ループ指示文の利用法

前回の記事にも書いたように OpenMP では並列化ループ指示文の直後に書かれているループを並列実行します. Fortran のプログラムで用いる並列ループ指示文 `parallel do` と C/C++

で用いる並列ループ指示文 parallel for の構文を図 5 に示します. なお, [ ] で囲まれた部分はオプションです.

Fortran

```
!$omp parallel do [指示節 [,] [指示節 ...]]
do ループ
```

C/C++

```
#pragma omp parallel for [指示節 [指示節 ...]]
for ループ
```

図 5: 並列ループ指示文の構文

3 節で紹介したスコープ指示節で扱いを指示しない限り, 並列ループを制御する変数はプライベート変数として扱われます. これは, それぞれのスレッドが独立にループを進行させるためです. さらに Fortran では, ループの入れ子構造を並列化する際, 特に指示がなければ並列ループの内側のループの制御変数についてもプライベート変数として扱います. 一方 C/C++ では, 特に指示がなければ並列ループの内側のループを制御する変数は共有変数として扱われてしまいます. これは, C/C++では for 文を複雑に書くことが可能であり, どの変数をプライベート変数にすれば良いかをコンパイラが判断するのが難しいためです. 通常, 並列ループの内側にあるループの制御変数が共有変数だと正しく動作しないため, C/C++ では private 指示節 (3.3 節) を用いて明示的に指示する必要があります.

指示文の後には以下の指示節を追加することができます.

スコープ指示節
: 並列ループ内での変数の扱われ方を指示します (3 節).

schedule 指示節
: ループの繰り返しをスレッドに割り当てる方法を指示します (4.1 節).

if 指示節
: ループを並列実行するか否かの条件を付けます (4.2 節).

ordered 指示節
: 完全な並列化ができないループにおいて, 一部だけ逐次実行を行なわせる ordered 指示文がループ中に存在することを宣言します. 詳細は次回の記事で説明する予定です.

copyin 指示節
: 指示した private 変数の値を並列ループの先頭で初期化します. 詳細は次回の記事で説明する予定です.

スコープ指示節と copyin 指示節は, 指示の対象となる変数が重複しなければ, 一つの並列ループ指示文中にそれぞれ複数存在しても構いません. 一方 schedule 指示節, if 指示節, ordered 指示節は, 1 つの並列ループ指示文中にそれぞれ高々 1 回しか記述できません.

4.1 schedule 指示節

parallel do (parallel for) 指示文によって指示されたループは, 実行時にいくつかの繰り返し毎に分割されて各スレッドに割り当てられます. この, 繰り返しをスレッドに割り当てることをスケジュールと呼びます. OpenMP ではスケジュールの方法を schedule 指示節によって指示することができます.

今まで出てきた例のように schedule 指示節がない場合, ループの繰り返し数をスレッド数で均等に分割し, そのひとまとまり毎にスレッドに割り当てます. このスケジュール方法は, ループの各繰り返しで仕事量が均等である場合は, 一般的に最も効率の良い並列実行を行なえる割り当て方法です.

しかし各繰り返しで仕事量にばらつきがある場合, 今までの方法ではスレッド毎に仕事量の違いが生じます. すなわち, 割り当てられた仕事量が少ないスレッドは早く仕事を済ませ, 仕事量の多いスレッドを待ちます. その間, そのスレッドは何も仕事をしないので, システム全体の並列処理の効果が低下します. このような場合, 各スレッドになるべく均等に仕事を割り当てるため, schedule 指示節を利用します.

schedule 指示節の利用法は以下の通りです.

`schedule(` *kind*,*chunk* `)`

kind にはスケジュール方法を指示します. また, *chunk* にはチャンクの大きさを指示します. チャンクとは, スレッドに一度に割り当てる繰り返しの数です. すなわち 1 回の割り当てで, 指示された数の連続した繰り返しが一つのスレッドに割り当てられます.

schedule 指示節で指示できるスケジュール方法は以下の 4 つです.

`static`

chunk が指示されていればループの繰り返しを *chunk* 回ずつまとめて各スレッドに割り当てます. 最初のチャンクをスレッド 0 に, 次のチャンクをスレッド 1 に, というように割り当ててゆき, 最後のスレッドまで割り当ててもチャンクが残っていればまたスレッド 0 から割り当ててゆきます.

例えば 100 回の繰り返しを行なうループをチャンクの大きさ 10 でスレッドに割り当てる場合, プログラムは以下のようになります.

Fortran

```
!$omp parallel do schedule(static,10)
do i = 1, 100
  a(i) = b(i) * c(i)
enddo
```

C 言語

```
#pragma omp parallel for schedule(static, 10)
for (i = 0; i < 100; i++)
  a[i] = b[i] * c[i];
```

このループを 5 スレッドで並列実行すると, スレッド 0 はループの 1〜10 番目と, 51〜60 番目の繰り返しを担当します. また, スレッド 1 はループの 11〜20 番目と, 61〜70 番目の繰り返しを担当します.

もし *chunk* が指示されなければ, ループの繰り返し回数をスレッド数で等分したものを *chunk* として, 1 スレッドに 1 チャンクずつ割り当てます. なお, 繰り返しがスレッド数で割りきれない場合の端数の処理については OpenMP の仕様に規定されて無いので, 実行する計算機によって異なります.

`dynamic`

chunk 回の繰り返しをチャンクとして, 各スレッドに 1 チャンクずつ割り当ててゆきます. 割り当て分の処理が済んだスレッドには, さらに未処理の繰り返しが *chunk* 分割り当てられます. *chunk* が指示されていなければ 1 が指示されたものと見なされます.

`guided`

最初大きなチャンクを割り当てておき, 処理が進むにつれてチャンクの大きさを減らしてゆきます. *chunk* が指示されると, その大きさまで徐々にチャンクを減らします. 指示がなければ 1 が指示されたものと見なされます.

`runtime`

実行時の環境変数 `OMP_SCHEDULE` によってスケジュール方法を決定します. この場合, schedule 指示節にはチャンクの大きさを指示できません. `OMP_SCHEDULE` には, 以下のようにスケジュールの方法とチャンクの大きさを指示します.

csh

```
setenv OMP_SCHEDULE "guided.5"
```

sh

```
OMP_SCHEDULE="guided.5"
export OMP_SCHEDULE
```

`OMP_SCHEDULE` が定義されていない場合の動作は計算機によって異なります.

`dynamic` や `guided` でスケジューリングする場合は, 実行中にスレッド間で調整しながら仕事を割り当てるためのコストが必要です. 一方 `static` でスケジューリングする場合は, 実行前に割り当てを決定できるのでそのようなコストが不要です. すなわち, もし事前に仕事量が

分かっていて static で均等に分割できるのであれば, static でスケジューリングを行なうべきです.

一方, 事前に仕事量が分かっていない場合は dynamic や guided の利用を考えます. この場合, なるべく少ない調整回数で各スレッドの仕事量がなるべく均等になるように行なうのが理想です. guided は, 最初はおおざっぱに割り当てておいて最後の方で微調整することにより, 割り当て単位が固定の dynamic より効率的に, かつ均等に仕事を割り当てることを目的としています.

また, 最適なスケジューリング方法が入力データなどによって異なる場合は runtime としておいて, 実行時に環境変数でスケジューリング方法を指示する方が便利です.

4.2 if 指示節

OpenMP には, 並列実行するか否かを実行時に決定する if 指示節が用意されています. この指示節は, 並列実行の効果が得られそうな場合のみ並列実行したい場合や, 依存関係が実行時にならないと判明しない場合に利用します.

並列実行の効果が得られるか否かを厳密に判断するのは通常は困難ですが, プログラムの処理量からある程度予測することが可能なプログラムもあります. 例えばパラメータを変更することによってサイズの異なる行列を処理できるプログラムでは, 予めいくつかのパラメータ値で並列実行の効果を調べておき, 並列化の効果が得られる閾値を定めておいて, その閾値を用いた if 指示節を記述します.

以下の例は, 変数 N が 1000 以上の場合にのみ並列実行を行なう if 指示節の利用法です.

Fortran

```
!$omp parallel do if(N .ge. 1000)
```

C/C++

```
#pragma omp parallel for if(N >= 1000)
```

また, 例えばインデクス配列を介した間接アクセスを行なうループで, インデクス配列の中身によって並列化可能か否かが判定できる場合, ループの直前に判定のためのプログラムを記述し, その判定結果によって並列実行するかどうかを決定するよう, 並列化指示文に if 指示節を記述します.

5 プログラムの書き換えを伴うループの並列化

5.1 ループの繰り返し間の依存関係の除去

2.4 節で書いたように, ループの繰り返し間の依存関係がある場合, そのままでは並列化できません. ただし, ループの書き換えを行なうことによって, 依存関係を除去することが可能な場合があります. このうち, 本節では簡単なものについてのみ紹介します. 依存関係の解析や

除去の詳細については [1] や [3] 等を参照して下さい.

Fortran

```
!$omp parallel do
do i = 1, 100
   b(i) = a(i)
end do

!$omp parallel do
do i = 1, 99
   a(i) = b(i + 1) + d(i)
end do
```

C 言語

```
#pragma omp parallel for
for (i = 0; i < 100; i++){
    a[i] = b[i];
}

#pragma omp parallel for
for (i = 0; i < 99; i++){
    a[i] = b[i + 1] + d[i];
}
```

図 6: 繰り返し間の依存関係を含むループの並列化 (1)

例えば以下のプログラムでは `a(i)` の書き換えに `a(i + 1)` の値を参照しています.

Fortran

```
do i = 1, 99
   a(i) = a(i + 1) + d(i)
end do
```

C 言語

```
for (i = 0; i < 99; i++){
    a[i] = a[i + 1] + d[i];
}
```

このループをこのまま並列化すると, 例えば `i = 50` を実行するスレッドが `a(51)` の値を参照するより先に `i = 51` を実行するスレッドが `a(51)` に値を書き込む可能性があります. そこで,

図 6 に示すように別の配列 b を用意し, 予め a の値をコピーしておくよう, プログラムを変更します. それに合わせて, a(i) に書き込む値を a(i + 1) + d(i) ではなく b(i + 1) + d(i) とすれば, このループを並列実行することができます.

次に以下のループを見てみます.

Fortran

```
do i = 1, 100
   x = a(i) * b(i)
   c(1) = d(i) * x
end do

r = x + c(1) + c(2)
```

C 言語

```
for (i = 0; i < 100; i++){
   x = a[i] * b[i];
   c[0] = d[i] * x;
}
r = x + c[0] + c[1];
```

このループを単純に並列化すると, マスタースレッドが i = 99 の繰り返しを実行するとは限らないので, ループを実行した後の x と c(1) の値が並列化する前と異なる場合があります.

このうち x については, x を lastprivate 節に記述すれば除去できます (3.5 節参照). 一方 c(1) については, 同様の方法では除去できません. 配列の要素だけを lastprivate 節に記述することはできませんし, 配列 c 全体を lastprivate 節に記述すると, このループの最後の繰り返しで書き込みが行なわれる要素 (c(1)) 以外の値が並列実行後に不定になってしまいます. そこで, 図 7 に示すように, 変数 c1 (C/C++ では c0) を用います. この変数を仲介することにより, 出力依存を除去し, ループを並列化することができます.

5.2　性能改善

次にループの入れ子構造の並列化を見てみます. 図 8 のループの入れ子構造において, ループ j は繰り返し間の依存関係を含んでいるので並列化できません. 一方ループ i は繰り返し間の依存関係を含みません. そのためループ i の直前に並列化指示文を記述して並列化しています. この場合, ループ j の各繰り返しで並列ループ i が実行されます. しかしこのように内部ループを並列化すると, 2.1.1 節で紹介した「スレッドの生成」と「スレッドの廃棄」がそれぞれ N-1 回ずつ行なわれるため, 並列実行の効率は低くなります.

ところで, 図 8 のプログラムは, ループ i とループ j を入れ換えても結果には影響ありません. そこで図 9 のように, ループ i を外側にしたループの入れ子構造にしてみます. すると, 並列ループ i は一回しか実行されず, 「スレッドの生成と廃棄」も一回ずつしか行なわれないので, 並列実行の効率が向上されそうに見えます.

Fortran

```
!$omp parallel do lastprivate(x, c1)
do i = 1, 100
   x = a(i) * b(i)
   c1 = d(i) * x
end do

c(1) = c1
r = x + c(1) + c(2)
```

C 言語

```
#pragma omp parallel for lastprivate(x, c0)
for (i = 0; i < 100; i++){
   x = a[i] * b[i];
   c0 = d[i] * x;
}
c[0] = c0;
r = x + c[0] + c[1];
```

図 7: 繰り返し間の依存関係を含むループの並列化 (2)

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];
   }
}
```

図 8: 繰り返し間の依存関係を含むループの入れ子構造の並列化

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];
   }
}
```

図 9: ループの入れ換えによる性能改善

しかし, ループを入れ換えるとメモリの参照順序が変化します. Fortran の場合, 2 次元配列の列方向に連続してアクセスすると, メモリの連続した領域にアクセスが行なわれます. 一方 C/C++の場合は, 2 次元配列の行方向に連続してアクセスすると, 同様のアクセスとなります. 現在使われているほとんどの計算機は, このようにメモリの連続した領域にアクセスした方が高速に実行できるようになっています. ところが図 9 のプログラムは, ループを入れ換えたためにメモリの連続した領域にアクセスできないようになってしまっています.

これを解決する有効な方法としては, 2 次元配列の行と列を入れ換えることが考えられます. しかし, プログラム全体を書き換える必要がある上, その書き換えによってプログラムの他の部分で性能が低下する可能性もあります. また, 図 10,11 のように転置行列を格納する配列 `ta` を利用する方法もあります. これは, 転置行列との間のコピーによるコストの増加量よりも, 転置行列を使うことによるコストの削減量の方が多い場合に有効です.

```fortran
!$omp parallel do
do i = 1, N
   do j = 1, N
      ta(i, j) = a(j, i)
   enddo
enddo

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

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

図 10: 図 9 のプログラムを行と列を入れ換えて並列化 (Fortran)

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

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

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

図 11: 図 9 のプログラムを行と列を入れ換えて並列化 (C 言語)

6　ループ並列化の例

本節では, 以下の線形計算について, OpenMP を用いた並列プログラムの例を紹介します.

- 密行列の行列積
- 密行列の LU 分解
- 冪乗法による密行列の最大固有値計算
- 疎行列のベクトル行列積ルーチン

6.1　密行列の行列積

ここでは, 乱数で初期化された配列 `a`, `b` の積を配列 `c` に格納する行列積プログラムを扱います. このプログラムは, 配列 `c` の各要素の計算において依存関係がありません. すなわち, `c` の列方向に進むループ `i`, 行方向に進むループ `j` のどちらでも並列化が可能です. 5.2 節で書いた通り, ループの入れ子構造を並列化する場合は, なるべく外側のループを並列化した方が並列化にともなうコストを削減できるので, このプログラムではループ `j` を並列化しました. Fortran のプログラムでは, 並列ループ指示文 `parallel do` を挿入するだけで並列化できます. 一方 C 言語のプログラムでは, 3.7 節で説明した通り, ループの入れ子構造を並列化する際, 何も指示しなければ並列化されたループの内側のループ変数が共有変数となり, 正しい動作が行なえません. そこで, private 指示節を用いて内側のループ変数 `i,k` をプライベート変数に指示する必要があります.

なお, このプログラムのループ `k` は, 繰り返し間の依存関係があるので注意して下さい. この依存関係は `a(i, j)` のリダクション計算によるものなので, もしループ `k` を並列化したい場合は, 一旦別の変数を用いてリダクション計算をした後, `a(i, j)` に代入するよう, プログラムを書き換えて下さい.

6.1.1　並列行列積プログラム (Fortran)

```
! //////////////////////////////////////////////////////////////////////////// !
!         A Sample Program of Linear Computations --Matrix Multiplication--    !
! //////////////////////////////////////////////////////////////////////////// !
  program MatrixMultiplication
  implicit none
  integer,parameter            :: N=1000
  real(kind=8),dimension(N,N)  :: a, b, c
  integer                      :: i, j, k
  real(kind=8)                 :: rand

!------- Initiallize --------!
!$omp parallel do
  do j = 1, N
    do i = 1, N
      a(i, j) = rand(0)
```

```
      b(i, j) = rand(0)
      c(i, j) = 0.0D0
    end do
  end do

!------- Matrix Multiplication --------!
!$omp parallel do
  do j = 1, N
    do k = 1, N
      do i = 1, N
        c(i, j) = c(i, j) + a(i, k) * b(k, j)
      end do
    end do
  end do
end program MatrixMultiplication
```

6.1.2 並列行列積プログラム (C 言語)

```
/* ////////////////////////////////////////////////////////////////////////// */
/*     A Sample Program of Linear Computations --Matrix Multiplication--     */
/* ////////////////////////////////////////////////////////////////////////// */
#include <stdio.h>
#include <math.h>
#define N 1000

int main(void);

int main(void)
{
  double a[N][N], b[N][N], c[N][N];
  int i, j, k;

/* ------- Initiallize -------- */
#pragma omp parallel for private(j)
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++){
      a[i][j] = rand();
      b[i][j] = rand();
      c[i][j] = 0.0;
    }

/* ------- Matrix Multiplication -------- */
#pragma omp parallel for private(j, k)
  for (i = 0; i < N; i++)
    for (k = 0; k < N; k++)
      for (j = 0; j < N; j++)
        c[i][j] += a[i][k] * b[k][j];

}
```

6.2 密行列の LU 分解

ここでは, $N \times N$ 行列 a 及び N 次元ベクトル b について $a \times x = b$ となるベクトル x を求める LU 分解プログラムを扱います. ただし, 行列 `a` を

$$a_{ij} = \sqrt{\frac{2}{N+1}} sin\left(\frac{ij\pi}{N+1}\right)$$

ベクトル b を

$$b_i = \sum_{j=1}^{N} a_{ij}$$

で定義します. すると連立一次方程式 $a \times x = b$ の解は理論的に

$$x \equiv 1$$

となります. このプログラムでは, LU 分解によって計算した x の誤差を

$$\max_{1 \leq i \leq N} |x_i - 1|$$

で測定します.

このプログラムで最も処理時間を要するのは forward elimination 内のループ `i` とループ `j` で構成される入れ子構造です. 基本的にループ `i`, ループ `j` とも, 各繰り返しで配列 `a` の別の要素を更新するので, どちらのループも並列化可能です. そこで入れ子構造の外側であるループ `i` を並列化しています.

厳密には, 配列 `a` はインデクス配列 `ip` を介してアクセスされるため, このループを見ただけでは並列化可能かどうかを判断することはできません. もし配列 `ip` の各要素の値に重複がある場合, ループ `i` の複数の繰り返しで同じ要素に書き込みを行なうため, ループ `i` を並列化できません. しかし, プログラム全体を見ると

- 配列 `ip` に値を書き込んでいるのは冒頭部分と partial pivotting 部分だけである.
- プログラムの冒頭で配列 `ip` の要素毎に別の値を代入している.
- partial pivotting においても 2 要素間で値を入れ換えているだけである.

ということが分かるので, 配列 `ip` の各要素の値に重複が無く, ループ `i` を並列化しても問題無いことが確認できます.

またこのループでは, ピボット行の要素 `a(ip(i),k)` を一時的に変数 `q` に格納して, 実行時間の節約を図っています. この変数 `q` はループ `i` の各繰り返しで別の値を持ちます. すなわち, ループ `i` を並列化する場合, 変数 `q` はプライベート変数で扱う必要があります. しかし OpenMP では, 指示のない変数は基本的に共有変数となるので, 何も指示しないと正しい結果が得られなくなります. そこで, private 指示節を用いて変数 `q` をプライベート変数に指示します. C 言語のプログラムでは, `q` に加えて, 前節の行列積のプログラムと同じ理由から, 内側ループを制御する変数 `j` も private 指示節に記述します.

なお, このプログラムの partial pivoting の部分では k 列の最大要素をピボット行とするためのループを実行しています. これはリダクション計算なのですが, 最大要素が格納されている行の番号も知る必要があり, reduction 指示節を用いた単純な方法では並列化できないのでここでは非並列に実行しています.

6.2.1 並列 LU 分解プログラム (Fortran)

```
! ////////////////////////////////////////////////////////////////////////// !
!         A Sample Program of Linear Computations --LU Decomposition--        !
! ////////////////////////////////////////////////////////////////////////// !
  program LU
  implicit none
  integer,parameter           :: N=2000
  real(kind=8),dimension(N,N) :: a
  real(kind=8),dimension(N)   :: b, x
  integer,dimension(N)        :: ip
  integer                     :: i, j, k, l, lv
  real(kind=8)                :: error, Pi, p, q, s, al
  Pi = atan(1.0D0)*4.0D0

! ---------------- Initiallize ---------------------------------------------!
!$omp parallel do
  do i = 1, N
    ip(i) = i
    b(i) = 0.0d0
    do j = 1, N
      a(i, j) = sqrt(2.0D0/dble(N+1))*sin(dble(i)*dble(j)*Pi/dble(N+1))
      b(i) = b(i) + a(i, j)
    end do
  end do

! ---------------- Decompose Matrix ----------------------------------------!
  do k = 1, N - 1

! ---------------- Partial Pivotting ---------------------------------------!
    l = k
    al = abs(a(ip(l), k))
    do i = k + 1, N
      if (abs(a(ip(i), k)) .gt. al) then
        l = i
        al = abs(a(ip(l), k))
      end if
    end do
    if (l .ne. k) then
      lv = ip(k)
      ip(k) = ip(l)
      ip(l) = lv
    end if

! ---------------- Forward Elimination -------------------------------------!
    p = a(ip(k), k)
!$omp parallel do
```

```
    do j = k + 1, N
      a(ip(k), j) = a(ip(k), j) / p
    end do
    b(ip(k)) = b(ip(k)) / p

!$omp parallel do private(q)
    do i = k + 1, N
      q = a(ip(i), k)
      do j = k + 1, N
        a(ip(i), j) = a(ip(i), j) - q * a(ip(k), j)
      end do
      b(ip(i)) = b(ip(i)) - q * b(ip(k))  ! Forward Substitution
    end do
  end do

! ----------------- Backward Substitution ------------------------------------!
  x(ip(N)) = b(ip(N)) / a(ip(N), N)
  do k = n - 1, 1, -1
    s = b(ip(k))
    do j = k + 1, N
      s = s - a(ip(k), j) * x(ip(j))
    end do
    x(ip(k)) = s
  end do

! ----------------- Compute Error --------------------------------------------!
  error = 0.0d0
  do i = 1, N
     s = abs(x(i) - 1.0d0)
     if (s .ge. error) error = s
  end do

  print *, 'error = ', error
end program LU
```

6.2.2 並列 LU 分解プログラム (C 言語)

```
/* ////////////////////////////////////////////////////////////////////////// */
/*     A Sample Program of Linear Computations --LU Decomposition--           */
/* ////////////////////////////////////////////////////////////////////////// */
#include <stdio.h>
#include <math.h>

#define N 2000

int main(void);

int main(void)
{
  double a[N][N];
  double b[N], x[N];
  int ip[N];
  int i, j, k, l, lv;
```

```
  double error, Pi, p, q, s, al;

  Pi = atan(1.0) * 4.0;

/* ---------------- Initiallize -------------------------------------------- */
#pragma omp parallel for private(j)
  for (i = 0; i < N; i++){
    b[i] = 0.0;
    for (j = 0; j < N; j++){
      a[i][j] = sqrt(2.0/(N+1)) * sin(1.0 * (i + 1) * (j + 1) * Pi / (N+1));
      b[i] = b[i] + a[i][j];
    }
    ip[i] = i;
  }

/* ---------------- Decompose Matrix --------------------------------------- */
  for (k = 0; k < N - 1; k++){

/* ---------------- Partial Pivotting -------------------------------------- */
    l = k;
    al = fabs(a[ip[l]][k]);
    for (i = k + 1; i < N; i++){
      if (fabs(a[ip[i]][k]) > al){
        l = i;
        al = fabs(a[ip[i]][k]);
      }
    }
    if (l != k){
      lv = ip[k];
      ip[k] = ip[l];
      ip[l] = lv;
    }

/* ---------------- Forward Elimination ------------------------------------ */
    p = a[ip[k]][k];
#pragma omp parallel for
    for (j = k + 1; j < N; j++)
      a[ip[k]][j] = a[ip[k]][j] / p;
    b[ip[k]] = b[ip[k]] / p;
#pragma omp parallel for private(j, q)
    for (i = k + 1; i < N; i++){
      q = a[ip[i]][k];
      for (j = k + 1; j < N; j++){
        a[ip[i]][j] -= q * a[ip[k]][j];
      }
      b[ip[i]] -= q * b[ip[k]];           /* Forward Substitution */
    }
  }

/* ---------------- Backward Substitution ---------------------------------- */
  x[ip[N - 1]] = b[ip[N - 1]] / a[ip[N - 1]][N - 1];
  for (k = N - 2; k >= 0; k--){
    s = b[ip[k]];
    for (j = k + 1; j < N; j++)
```

```
      s = s - a[ip[k]][j] * x[ip[j]];
    x[ip[k]] = s;
  }

/* ---------------- Compute Error ----------------------------------------- */
  error = 0.0;
  for (i = 0; i < N; i++){
    s = fabs(x[i] - 1.0);
    if (s > error)
      error = s;
  }
  printf("error = %e\n", error);
  return(0);
}
```

6.3 冪乗法による密行列の最大固有値計算

ここでは, $a_{ij} = i + j$ で初期化された行列 a の最大固有値を, 冪乗法を用いて求めるプログラムを扱います. アルゴリズムは以下の通りです.

1. x(0) を適当に決める.
2. 適当なノルムで x(0) を正規化する. (x(0) = x(0) / |x(0)|)
3. k = 1, 2, ... 収束するまで以下を繰り返す.
   (収束判定条件は反復前後の相対誤差)
   (a) y(k) = a * x(k) を計算
   (b) x(k) = y(k) / |y(k)| で正規化

このプログラム中で, ノルムの計算を行なうループはリダクション計算なので, reduction 指示節を用いて並列化することができます. また, 相対誤差の最大値を求めるループは, Fortran のプログラムでは reduction 指示節に MAX 演算子を用いて並列化できますが, C 言語の OpenMP 規格には MAX 演算子がないため並列化できません.

6.3.1 並列冪乗法プログラム (Fortran)

```
! //////////////////////////////////////////////////////////////////////// !
!         A Sample Program of Linear Computations --The Power Method--        !
! //////////////////////////////////////////////////////////////////////// !
program sample_of_power_method
  implicit none
  integer,parameter :: N=1000
  real(kind(1.0D0)),dimension(N,N) :: a
  real(kind(1.0D0)) :: eigenvalue

  external init_a
  external power_method

  call init_a(a, N)
  call power_method(a,N,eigenvalue)

  write(*,*) "approximate eigenvalue=",eigenvalue
end program sample_of_power_method

! //////////////////////////////////////////////////////////////////////// !
subroutine init_a(a, N)
  integer,intent(in) :: N
  real(kind(1.0D0)),dimension(N,N),intent(out) :: a
  integer :: i, j

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

```
     end do
  end do
end subroutine init_a

! ////////////////////////////////////////////////////////////////////////// !
subroutine power_method(a,N,eigenvalue)
  integer,intent(in) :: N
  real(kind(1.0D0)),dimension(N,N),intent(in) :: a
  real(kind(1.0D0)),intent(out) :: eigenvalue

  integer :: i,j,k,itmax
  real(kind(1.0D0)),dimension(:),allocatable :: x,y
  real(kind(1.0D0)) :: epsilon,s,ss

  intrinsic max,abs,sqrt,random_number
  allocate(x(N),y(N))

! ---------------- set parameters ---------------------------------------- !
  epsilon=1.0D-6        ! stopping criterion
  itmax=1000            ! maximum iteration number
  call random_number(x) ! initial vector

! ---------------- normalize the initial vector---------------------------- !
  s=0.0D0
!$omp parallel do reduction (+:s)
  do i=1,N
     s=s+x(i)**2
  end do
!$omp parallel do
  do i=1,N
     x(i)=x(i)/sqrt(s)
  end do

! ---------------- Iteration step ---------------------------------------- !
  do k=1,itmax
!$omp parallel do
     do i=1,N                          !--+
        y(i)=0.0D0                     !  |
        do j=1,N                       !  +-- compute y:=Ax
           y(i)=y(i)+a(i,j)*x(j)       !  |
        end do                         !  |
     end do                            !--+

     s=0.0D0                           !--+
!$omp parallel do reduction (+:s)
     do i=1,N                          !  +-- compute 2-norm of y as |y|
        s=s+y(i)**2                    !  |
     end do                            !--+

     ss = sqrt(s)
!$omp parallel do
     do i=1,N                          !--+
        y(i)=y(i)/ss                   !  +-- set y:=y/|y|
     end do                            !--+
```

```
     s=0.0D0                                !--+
!$omp parallel do reduction (MAX:s)
     do i=1,N                               !  +-- compute error |x-y|/|y| as
        s=max(abs(x(i)-y(i))/abs(y(i)),s)   !  |   the maximum norm sense
     end do                                 !--+

     if(s<epsilon) then                     !----- check the convergence
        write(*,*) "iteration count=",k
!$omp parallel do
        do i=1,N                            !--+
           x(i)=0.0D0                       !  |
           do j=1,N                         !  +-- compute x:=Ay
              x(i)=x(i)+a(i,j)*y(j)         !  |
           end do                           !  |
        end do                              !--+

        eigenvalue=0.0D0                    !--+            T
!$omp parallel do reduction (+:eigenvalue)
        do i=1,N                            !  +-- compute y x as eigenvalue
           eigenvalue=eigenvalue+y(i)*x(i)  !  |
        end do                              !--+
        exit
     else                                   !----- if not converge
!$omp parallel do
        do i=1,N
           x(i)=y(i)
        end do
        if(k==itmax) write(*,*) "power method does not converge"
     end if
  end do
end subroutine power_method
```

6.3.2 並列冪乗法プログラム (C 言語)

```
/* ////////////////////////////////////////////////////////////////////// */
/*      A Sample Program of Linear Computations --The Power Method--        */
/* ////////////////////////////////////////////////////////////////////// */
#include <stdio.h>
#include <math.h>

#define N 1000

int main(void);
void init_a(double [][N], int);
void power_method(double [][N], int, double *);

int main(void)
{
  double a[N][N];
  double eigenvalue;

  init_a(a, N);
```

```
  power_method(a, N, &eigenvalue);

  printf(" approximate eigenvalue=%.9f\n", eigenvalue);
  return(0);
}

/* ////////////////////////////////////////////////////////////////////////// */
void init_a(double a[][N], int n)
{
  int i, j;

#pragma omp parallel for private(j)
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      a[i][j] = i + j + 2;
}

/* ////////////////////////////////////////////////////////////////////////// */
void power_method(double a[][N], int n, double *eigenvalue)
{
  int i, j, k, itmax;
  double *x, *y;
  double epsilon, s, ss, ev;
  x = (double *)malloc(n * sizeof(double));
  y = (double *)malloc(n * sizeof(double));

/* ----------------- set parameters ---------------------------------------- */
  epsilon = 0.000001;
  itmax = 1000;

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

/* ----------------- normalize the initial vector-------------------------- */
  s = 0.0;
#pragma omp parallel for reduction (+:s)
  for (i = 0; i < N; i++)
    s += x[i] * x[i];

  ss = sqrt(s)
#pragma omp parallel for
  for (i = 0; i < N; i++)
    x[i] = x[i] / ss;

/* ----------------- Iteration step ---------------------------------------- */
  for (k = 0; k < itmax; k++){
#pragma omp parallel for private(j)
    for (i = 0; i < N; i++){
      y[i] = 0.0;
      for (j = 0; j < N; j++){
        y[i] += a[i][j] * x[j];              /* computer y:=Ax */
      }
    }
```

```
    s = 0.0;
#pragma omp parallel for reduction(+:s)
    for (i = 0; i < N; i++)
      s += y[i] * y[i];                        /* computer 2-norm of y as |y|

    ss = sqrt(s);
#pragma omp parallel for
    for (i = 0; i < N; i++)
      y[i] = y[i] / ss;                        /* set y:=y/|y| */

    s = 0.0;
    for (i = 0; i < N; i++){                   /* compute error |x-y|/|y| as  */
      ss = fabs(x[i] - y[i])/fabs(y[i]);       /* the maximum norm sense      */
      if (ss > s)
        s = ss;
    }

    if (s < epsilon){                          /* check the convergence  */
      printf(" iteration count=%d\n", k+1);
#pragma omp parallel for private(j)
      for (i = 0; i < N; i++){
        x[i] = 0.0;
        for (j = 0; j < N; j++)
          x[i] += a[i][j] * y[j];              /* compute x:=Ay */
      }
      ev = 0.0;
#pragma omp parallel for reduction(+:ev)
      for (i = 0; i < N; i++)
        ev += y[i] * x[i];                     /* compute y x as eigenvalue */
      *eigenvalue = ev;
      return;
    } else{
#pragma omp parallel for
      for (i = 0; i < N; i++)
        x[i] = y[i];
      if (k == itmax)
        printf(" power method does not converge\n");
    }
  }
}
```

6.4　疎行列のベクトル行列積ルーチン

ここではハーウェル ボーイング (Compressed Column Storage の名でも呼ばれています) 形式で格納された実数非対称疎行列と実数ベクトルの積を計算するルーチンを扱います.

対象とする疎行列は、row_start, col_idx, a の 3 つの 1 次元配列から構成されています. 配列 col_idx は, 疎行列の非ゼロ要素の列番号を格納しています. 配列 a は, その位置の非ゼロ要素の値を格納します. 配列 row_start は, 疎行列の各行について最初の非ゼロ要素の場所を格納した配列です. こうして、i 行目の非ゼロ要素に対する列番号は, col_idx(row_start(i)) から col_idx(row_start(i+1)-1)) に格納されています. それに対応する疎行列の非ゼロ要素の値は, a(row_start(i)) から a(row_start(i+1)-1) に格納されています。

このルーチン中のループ入れ子構造では, ループ i の各繰り返しで配列 y の別の要素に書き込みを行なうので並列化可能です. ただし, ループ i の各繰り返しで一時的に値を格納する変数 t, start, end をプライベート変数として指示する必要があります.

6.4.1　並列疎行列ベクトル行列積ルーチン (Fortran)

```
! ////////////////////////////////////////////////////////////////////////// !
subroutine matvec(a, row_start, col_idx, x, y, nn, nnz)
  integer,intent(in) :: nn, nnz
  real(kind(1.0D0)),dimension(nnz),intent(in) :: a
  real(kind(1.0D0)),dimension(nn),intent(in)  :: x
  real(kind(1.0D0)),dimension(nn),intent(out) :: y
  integer,dimension(nn+1),intent(in)          :: row_start
  integer,dimension(nn),intent(in)            :: col_idx
  integer           :: i, j, start, end
  real(kind(1.0D0)) :: t

!$omp parallel do private(t, start, end)
  do i = 1, nn
    start = row_start(i)
    end = row_start(i + 1)
    t = 0.0D0
    do j = start, end - 1
      t = t + a(j) * x(col_idx(j))
    end do
    y(i) = t
  end do
end subroutine matvec
```

6.4.2　並列疎行列ベクトル行列積ルーチン (C 言語)

```
/* ////////////////////////////////////////////////////////////////////////// */
void matvec(double a[], int row_start[], int col_idx[], double x[],
    double y[], int nn)
{
  int i, j, start, end;
  double t;
```

```
#pragma omp parallel for private(j, t, start, end)
  for(i=0; i< nn; i++){
    start = row_start[i];
    end = row_start[i+1];
    t = 0.0;
    for(j= start; j< end; j++)
      t += a[j] * x[col_idx[j]];
    y[i] = t;
  }
}
```

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.

[3] 中田 育男: “コンパイラの構成と最適化,” 朝倉書店, 1999 年.