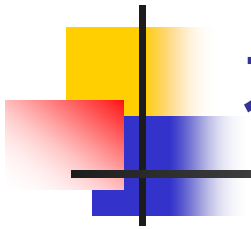
計算科学演習　第6回 講義
「OpenMP並列処理」

2010年6月3日

システム情報学研究科　計算科学専攻
臼井　英之



本講義の概要（2週分）

1．スレッド並列

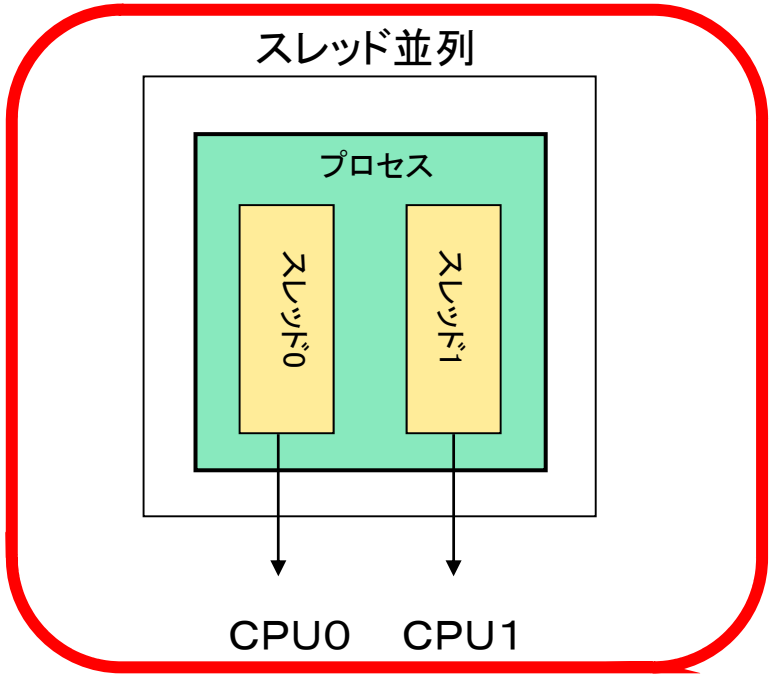
2．Scalarマシンでのバッチジョブについて

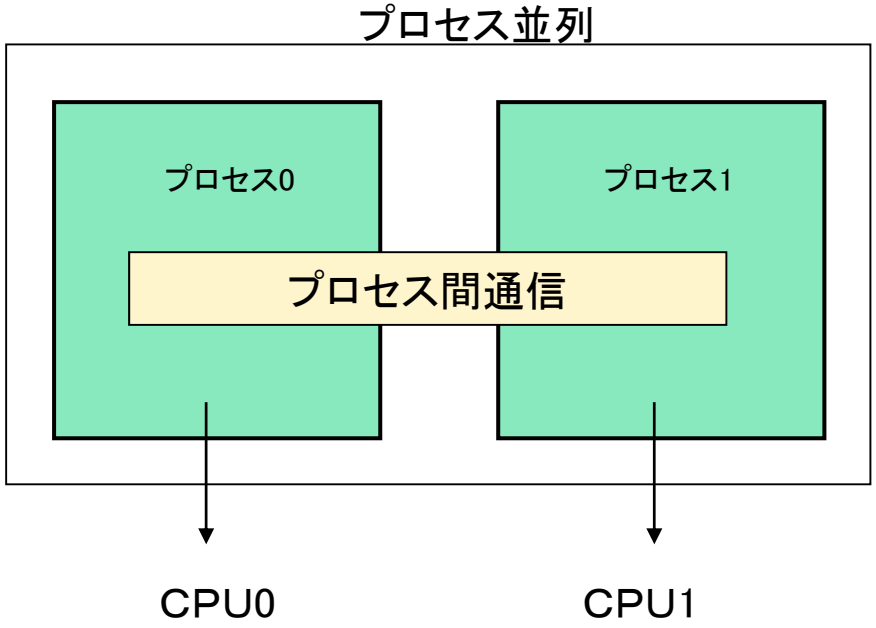
3．OpenMPの基礎と演習

- 簡単なDoループ
- Reduction演算（配列の中身を足していく）
- section 並列

1. スレッド並列

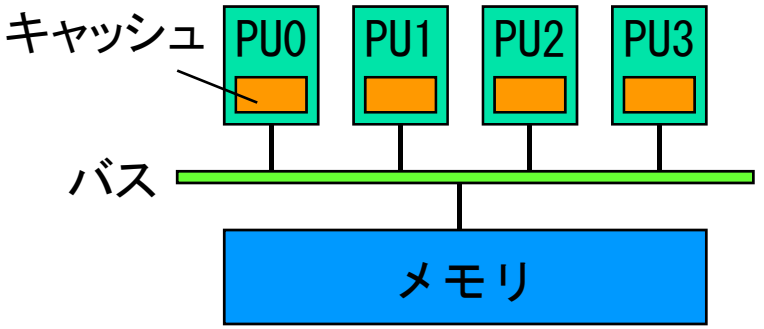
- 共有メモリ型並列計算機における並列方式
- 各スレッドは同じメモリ空間をアクセス
- OpenMPの利用





共有メモリ型並列計算機(復習)

- 構成
  - 複数のプロセッサ(PU)がバスを通してメモリを共有
  - どのPUも同じメモリ領域にアクセスできる

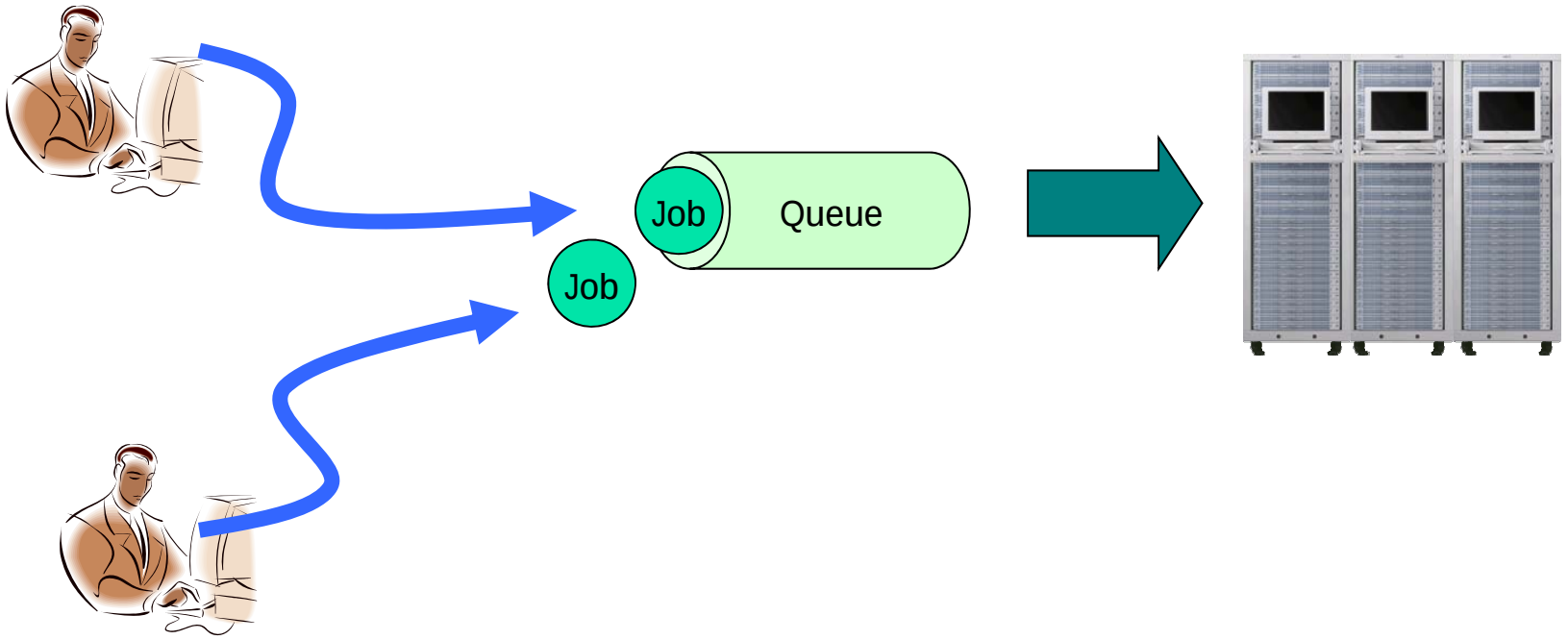


- 特徴
  - メモリ空間が単一のためプログラミングが容易
  - PUの数が多すぎると，アクセス競合により性能が低下
    → 2～16台程度の並列が多い

- プログラミング言語
  - OpenMP (FORTRAN/C/C++ ＋ 指示文)を使用
    - メモリ領域を分割し，MPI (次頁参照)を利用することも可能

2. バッチジョブ投入について

- プログラムの実行は、ジョブスクリプト（環境と実行すべきコマンド、プログラムが記述されている）をバッチキューイングシステムへ投入することで、順番制御された上で実行される。 NQS (Network Queuing System)の利用。



バッチジョブ投入について（OpenMP)

キュー名	ノード数	メモリ容量指定	同時実行 ジョブ数上限(本)
PCS-A	1-8	1GB/node	16
PCL-A	9-64	2GB/node	16

OpenMP利用のシェルスクリプト例

```
#PBS –l   cputim_job=00:01:00          ←   使用CPU時間を指定
#PBS –l   memsz_job=1gb          ←   使用メモリサイズを指定 (largeは2、small は1）
#PBS –l   cpunum_job=2           ←   使用CPU数を指定
#PBS –q   PCS-A                  ←   投入先のキュー名を指定
cd   作業ディレクトリ (例えば、/home/users/usui/OpenMP/test)
Setenv OMP_NUM_THREADS 2
実行プログラム （例えば、./a.out)
```

- サンプルシェルスクリプトはscalar:/tmp/100603/sample_OMP.sh
- 各自コピー後、編集して利用

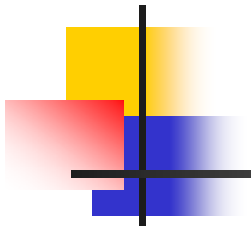
バッチジョブ投入について(MPI)

キュー名	ノード数	メモリ容量指定	同時実行ジョブ数上限(本)
PCS-A	1-8	1GB/node	16
PCL-A	9-64	2GB/node	16

MPI利用のシェルスクリプト例

```
#PBS –l   cputim_job=00:05:00                    使用CPU時間を指定
#PBS –l   memsz_job=2gb      ←  使用メモリサイズを指定 (largeは2、small は1)
#PBS –l   cpunum_job=1       ←  使用CPU数を指定
#PBS –T   vltmpi             ←  Voltaire MPIを指定
#PBS –b   16                 ←  16ノードを指定
#PBS –q   PCL-A              ←  投入先のキュー名を指定  (largeはPCL-A, small はPCS-A)
cd  作業ディレクトリ (例えば、/home/users/usui/MPI/test)
mpirun_rsh  -np 16 ${NQSII_MPIOPTS} 実行プログラム(たとえば ./a.out )
```

- サンプルシェルスクリプトはscalar:/tmp/100603/sample_mpi.sh
- 各自コピー後、編集して利用



バッチジョブ関連コマンド

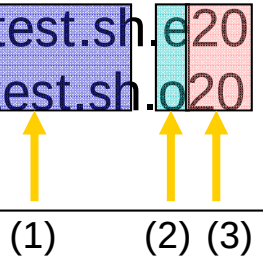
- ジョブ投入： qsub　ジョブスクリプトファイル
  - 例えば、ジョブスクリプトファイルが、test.shなら、qsub test.shでシステムにジョブ投入
- ジョブの状態表示： qstat
  - 投入したジョブの状態を表示（キュー状態か、Runか終了か。）
- 投入ジョブのキャンセル： qdel ジョブ番号
  - ジョブ番号は、qstatで表示されるRequestIDに相当

ジョブ実行結果

- ジョブの実行が終了すると、標準出力/標準エラー出力がそれぞれ通常ファイルとして出力される。

test.sh というジョブスクリプトを投入した場合

```
[ss099@scalar pcc]$ ls -l test.sh.?20
-rw-r--r--  1  ss099  ss2008   244  Mar 19  2008  test.sh.e20
-rw-r--r--  1  ss099  ss2008  3285  Mar 19  2008  test.sh.o20
```



(1) ジョブスクリプトファイル名

(2) 標準出力(o)、標準エラー出力(e)

(3) ジョブのリクエストIDの数字部

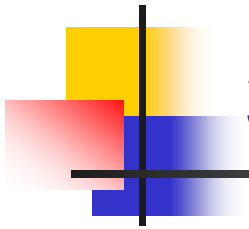
練習1

・ジョブ投入の練習のために、すでにMPIプログラム（パイの計算）をコンパイルした実行形式のファイルを使います。（ファイル名はpi_test.)
・利用するCPU数を変えて、演算時間の違いを見てみます。
（CPU数を増やすと、速く終わるはず。例えば、2，4，8，16，32と試してみる。）
・具体的なMPIプログラムの説明は別途、山本先生の実習講義で行います。

1. scalarマシンにログイン　(scalar.scitec.kobe-u.ac.jp)
2. mkdirコマンドで、適当な作業ディレクトリを作成　（たとえば、” mkdir　test” でtestという名前のディレクトリを作成）
3. そのディレクトリに移動　(“ cd test”)
4. 必要なファイル(large.sh, small.sh, pi_test)をコピー

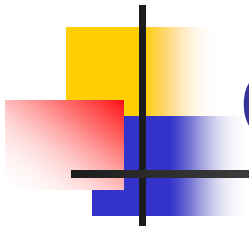
   (“ cp /tmp/nqs_test/large.sh . “, “ cp /tmp/nqs_test/pi_test . “,)
5. large (small). shの中身を確認・編集　（利用ノード数の変更など
6. ジョブ投入　（” 例えば qsub large.sh”)
7. ステータスの確認　(“qstat”)
8. ジョブ終了後、出力の確認（” more　large.sh.o***”)

PCL-Aのキューで8以下のCPUを使うジョブも打てるが、キューが混むので、
8以下のCPUの場合、PCS-Aのキュー（small.shの利用）でジョブ投入



3. OpenMP の基礎

- OpenMP とは
- OpenMP プログラムの構成要素
- 簡単な OpenMP プログラムの例
- ループへのスレッド割り当ての指定
- セクション型の並列化
- 参考文献



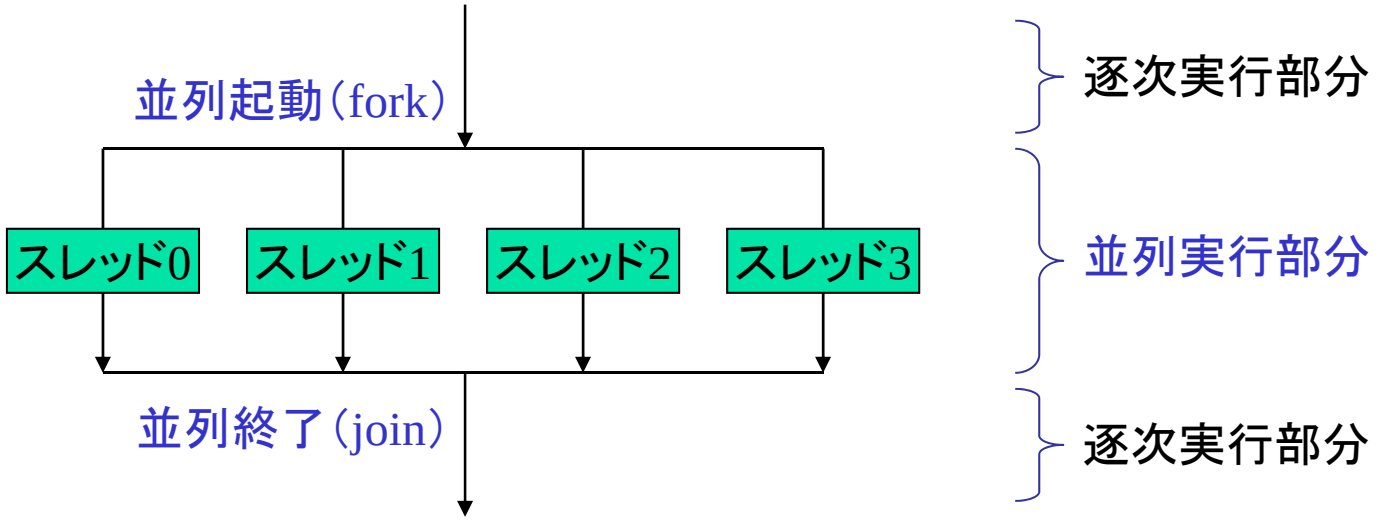
OpenMPとは

- 共有メモリ型並列計算機上での並列プログラミングのためのプログラミング規格
  - ベース言語（FORTRAN/C/C++）をディレクティブ（指示文）により並列プログラミングができるように拡張

- 米国のコンパイラメーカーを中心に仕様を決定
  - 1997/10　FORTRAN Ver. 1.0 API
  - 1998/10　C/C++ Ver. 1.0 API
  - 2000/11　FORTRAN Ver. 2.0 API
  - 2002/3　C/C++ Ver. 2.0 API
  - 2007/10　FORTRAN C/C++ Ver. 2.5 API
  - 2007/10　FORTRAN C/C++ Ver. 3.0 API Draft

OpenMPの実行モデル

- Fork-joinモデル
  - 並列化を指定しない部分は逐次的に実行
  - 指示文で指定された部分のみを複数のスレッドで実行



  - 各スレッドは同じメモリ空間をアクセス

簡単な OpenMP プログラムの例 (1)

- 環境変数 OMP_NUM_THREADS を 2 に設定
  (コマンドラインで、“ setenv OMP_NUM_THREADS 2”)
- 下記のプログラムを自分で書いて保存し、コンパイル・実行

```
program hello
integer :: omp_get_thread_num
print*,  ‘program start.’
!$omp parallel
print*,  ‘My thread number =’, omp_get_thread_num()
!$omp end parallel
print*,  ‘program end.’
end
```

スレッド番号を取得するライブラリ関数

指示文: 並列実行部分の開始

指示文:並列実行部分の終了

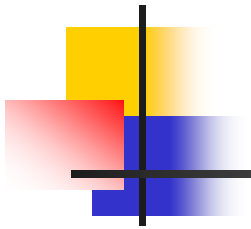
ただし、コンパイルするには、

’ pgf90 -o [実行形式ファイル名] -mp [プログラム名] ‘

実行形式ファイル(たとえば、hello)を走らせる。(./hello)

- 実行結果

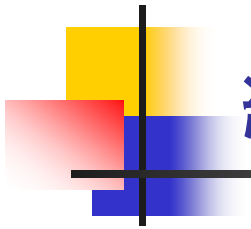
```
My thread number = 0
My thread number = 1
```



プログラムの解説

- 並列リージョン
  - 2つの指示文 !$omp parallel と !$omp end parallel で囲まれた部分を並列リージョンと呼ぶ
  - 並列リージョン内では，OMP_NUM_THREADS で指定した個数のスレッドが同じコードを実行する
  - 各スレッドは，関数 omp_get_thread_num によって取得できる固有の番号（スレッド番号）を持つ。スレッド番号を用いて，各スレッドに異なる処理を行わせることができる

- 変数・配列の参照・更新
  - すべてのスレッドが同じ変数・配列を参照できる
  - 複数のスレッドが同時に同じ変数を更新しないよう，注意が必要
    - 同じ配列の異なる要素を同時に更新するのは問題なし

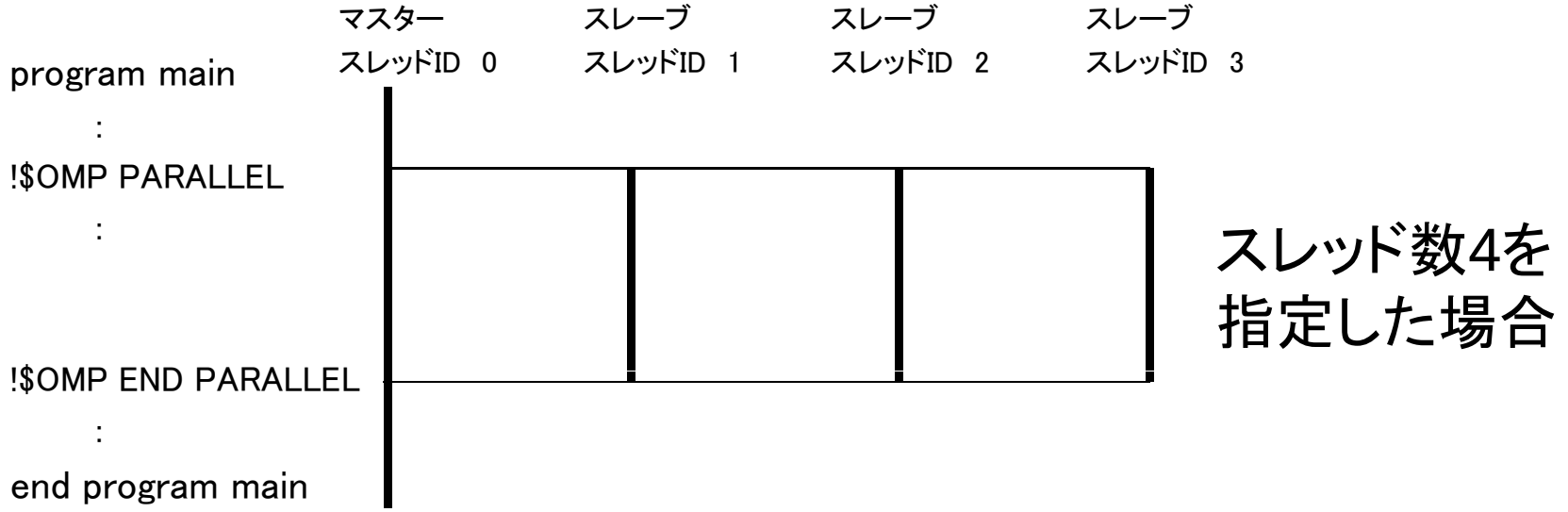


練習２

- バッチジョブ投入により、前頁のプログラムを走らせる。
- OMP_NUM_THREADS を1の場合と2の場合で違いを見る。

マルチスレッドでの実行イメージ

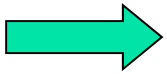
- プログラム実行開始時はマスタースレッドのみ
- PARALLELディレクティブによりスレーブスレッドを生成
  - スレッドID：　マスタースレッドは0、スレーブスレッドは1～
  - チーム：　並列実行を行うスレッドの集団
  - スレッド生成後、全てのスレッドで冗長実行
- END PARALLELディレクティブによりスレーブスレッドが消滅



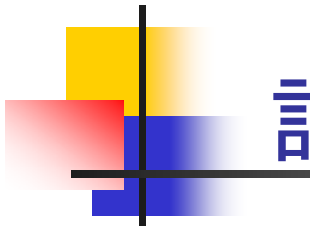
スレッド数4を
指定した場合

OpenMP並列化プログラムの基本構成例

```
program main
integer :: i,j
...
...
!$omp parallel
...
...
...
!$omp end parallel...
...
end
```



複数のスレッドにより
並列実行される部分



計算の並列化（Work-Sharing構造）

- チーム内のスレッドに仕事（Work）を分割(Share)する。
- Work-Sharing構造の種類
  - DOループを各スレッドで分割 (!$OMP DO , !$OMP END DO)
  - 別々の処理を各スレッドが分担(!$OMP SECTIONS, !$OMP END SECTIONS)
  - 1スレッドのみ実行(!$OMP SINGLE, !$OMP END SINGLE)
- Work-Sharing構造ではないが・・・
  - マスタスレッドでのみ実行(!$OMP MASTER, !$OMP END MASTER)

OMP DO （1）

```
implicit none
integer, parameter ::  SP = kind(1.0)
integer, parameter ::  DP = selected_real_kind(2*precision(1.0_SP))
real(DP), dimension(100) :: a, b
integer :: i
!$omp parallel
!$omp do
do i=1,100
    b(i)=a(i)
enddo
!$omp end do
!$omp end parallel
end
```

直後のdoループを複数のスレッドで分割して実行せよ という 指示

2スレッドの場合：

スレッド0

```
    do i=1,50
        b(i)=a(i)
    enddo
```

スレッド1

```
    do j=51,100
        b(j)=a(j)
    enddo
```

OMP DO （2）

注意） !$OMP DO はdoループの中身が並列実行可能かどうかは関知せず、必ず分割してしまう。

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

2スレッドの場合：

スレッド0

```
    do i=1,50
        b(i)=a(i)+b(i-1)
    enddo
```

スレッド1

```
    do j=51,100
        b(j)=a(j)+b(j-1)
    enddo
```

→ b(50)の結果がないと本来実行できない

簡単な OpenMP プログラムの例 (2)

```
program axpy
 implicit none
  integer, parameter ::  SP = kind(1.0)
  integer, parameter ::  DP = selected_real_kind(2*precision(1.0_SP))
  real(DP), dimension(100) :: x, y, z
   real(DP):: a
  integer :: i
!
! a, x, yの値を各自設定 (なにか定数を設定するとか。)
!
!$omp parallel
!$omp do
do i = 1, 100
  z(i) = a*x(i) + y(i)
end do
!$omp end do
!$omp end parallel
(z(100)の値を表示)
End
```

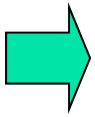
ベクトルの加算 $z = ax + y$

実行結果: 1≦ i ≦50 がスレッド0, 51≦ i ≦100 がスレッド1で計算される。

簡単な OpenMP プログラムの例 (2a)

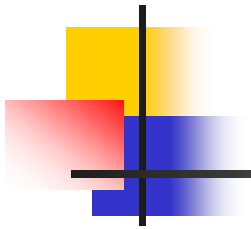
- 先ほどのDo並列化は以下のようにも書ける。

```
!$omp parallel
!$omp do
      do i=1,n
            :
      end do
!$omp end do
!$omp end parallel
```



```
!$omp parallel do
      do i=1,n
            :
      end do
!$omp end parallel do
```

（!$OMP END PARALLEL DOは省略可）



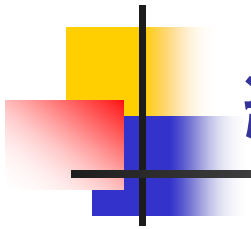
プログラムの解説

- do ループの並列化
  - do ループを並列化するには，並列化したい do 文の直前に指示文 !$omp parallel do を置けばよい
  - すると，ループ変数の動く範囲が OMP_NUM_THREADS 個にブロック分割され，各ブロックがそれぞれ1スレッドにより実行される
  - 並列化してよいループかどうかは，プログラマが判断する必要あり

- 並列化してはいけないループの例
  - 再帰参照を含むループ

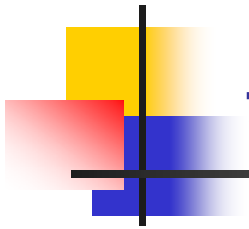
```
do i = 1, 100
  x(i) = a*x(i-1) + b
end do
```

1つ前に計算した要素の値を使って現在の要素を計算



練習　３

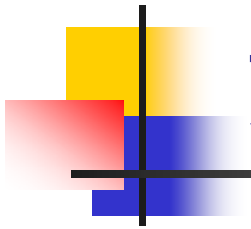
- 前述のプログラム（２）を作成し、scalarマシンで走らせて、１スレッドの場合と、２スレッドの場合とで経過時間を比較せよ。
- 時間計測には、omp_get_wtime関数を用いる。
  - 倍精度でomp_get_wtime, time0, time1, wall timeを定義し
  - 測定したい計算ブロックをtime0 = omp_get_wtime()　と time1 = omp_get_wtime() ではさむ。
  - 経過時間（秒単位）は、walltime = (time1 - time0)　として得られる。



共有変数とプライベート変数

- 共有変数
  - OpenMP のプログラミングモデルでは，基本的にすべての変数は共有変数（どのスレッドからも参照・更新が可能）

- プライベート変数
  - ループインデックス変数 i については，スレッド 0 と 1 で共有すると，正しい制御ができない
    - スレッド0では 1≦ i ≦50，1では 51≦ i ≦100 の範囲を動いて欲しい
  - そこで，各スレッドがそれぞれ別の変数を持つ必要がある
  - このような変数をプライベート変数と呼ぶ



変数の共有指定

- デフォルト値
  - 何も指示をしない変数については，**基本的に共有変数**となる
  - しかし，並列化された**ループのインデックス変数**のように，明らかにプライベート変数でなければならない変数については，**特に指示をしなくてもプライベート変数**となる
    - 多重ループの場合は並列化対象ループのインデックス変数のみ
- 共有変数の指定（通常は不要）
  - 並列化指示文の後に，shared 節を追加する。
- プライベート変数の指定
  - 並列化指示文の後に，private 節を追加する。
- 例(2)のプログラムで，指示を省略せずに書く場合
  - `!$omp parallel do shared(a, x, y, z) private(i)`

変数の共有指定の例

- 2重ループの並列化（行列ベクトル積）

```
program gemv
implicit none
  integer, parameter ::  SP = kind(1.0)
  integer, parameter ::  DP = selected_real_kind(2*precision(1.0_SP))
  real(DP), dimension(100,100) :: a
  real(DP), dimension(100) :: x, y
  integer :: i, j
（a, xの値を設定）
!$omp parallel do private(j)
do i = 1, 100
  y(i) = 0.0_DP
  do j = 1, 100
    y(i) = y(i) + a(i,j) * x(j)
  end do
end do
（y(100)の値を表示）
stop
end
```

j をプライベート変数に指定

a, x, y は自動的に共有変数となる。
i は自動的にプライベート変数となる。
j はプライベート変数とすべきだが，
自動的にはそうならない（並列化対象ループのインデックス変数ではない）ので指定が必要

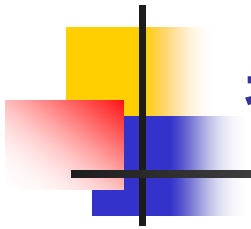
OMP DO （3）

分割を規定する

```
!$omp parallel
!$omp do schedule(static, 4)
do i=1,100
    b(i)=a(i)
enddo
!$omp end do
!$omp end parallelend
```

1～100を4つづつのchunkにわけて、それをサイクリックに各スレッドに割り当てる

4スレッド実行時
マスタスレッド担当行：
1,2,3,4,17,18,19,20,



参考文献

- 南里豪志，天野浩文：“OpenMP入門 (1), (2), (3)”，
  http://www.cc.kyushu-u.ac.jp/scp/system/library
  /OpenMP/OpenMP.html

- 黒田久泰：“C言語によるOpenMP入門”，
  http://www.cc.u-tokyo.ac.jp/publication/kosyu
  /03/kosyu-openmp_c.pdf

- 北山 洋幸：
  “OpenMP入門 - マルチコアCPU時代の並列プログラミング ”，
  秀和システム，2009.