演習1： 演習準備

2013年8月6日

神戸大学大学院システム情報学研究科
森下浩二

演習1の内容

- 神戸大FX10（π-Computer）について
  - システム概要
  - ログイン方法
  - コンパイルとジョブ実行方法

- OpenMPの演習（入門編）
  1. parallel 構文・実行時ライブラリ関数
  2. ループ構文
  3. shared 節・private 節
  4. reduction 節

神戸大FX10（π-Computer）について

神戸大FX10（π-Computer）

- 富士通PRIMEHPC FX10：1ラック
  - SPARC64™ IXfx プロセッサ x 96ノード
  - 総理論演算性能：20.2TFLOPS
  - 総主記憶容量：3TByte
- 1ノード諸元表（京との比較）

	FX10 (SPARC64™ IXfx)	京 (SPARC64™ VIIIfx)
コア数	16	8
L1キャッシュ（コア）	32KB(D)/32KB(I)	←
共有L2キャッシュ	12MB	6MB
動作周波数	1.65GHz	2.0GHz
理論演算性能	211.2GFlops	128GFlops
メモリ容量	32GB	16GB

FX10へのログイン方法

- 公開鍵認証によりログイン

- 手順の詳細は別紙を参照
  1. 鍵ペア（公開鍵・秘密鍵）の作成
  2. 仮の鍵ペアでログイン
  3. 自身の公開鍵を登録
  4. 自身の鍵ペアでログイン出来ることを確認
  5. 仮の公開鍵を削除

コンパイル方法

F : Fortran　C : C言語

- 逐次プログラム

```
$ frtpx sample.f90        F
$ fccpx sample.c          C
```

- OpenMP

```
$ frtpx -Kopenmp sample.f90        F
$ fccpx -Kopenmp sample.c          C
```

- 自動並列化（ノード内スレッド並列）

```
$ frtpx -Kparallel sample.f90      F
$ fccpx -Kparallel sample.c        C
```

ジョブ実行方法

- ジョブスクリプトの作成
  - single.sh：逐次ジョブ

```
#!/bin/sh
#PJM -L "rscgrp=school"
#PJM -L "node=1"
#PJM -L "elapse=10:00"
#PJM -j
#
./a.out
```

←シェルを指定
←利用リソースグループ名
←利用ノード数
←最大経過時間（hh:mm:ss）
←標準エラー出力をマージして出力
←プログラムの実行

- ジョブの投入

```
$ pjsub single.sh
```

ジョブ実行方法

- ジョブスクリプトの作成
  - thread_omp.sh：スレッド並列（OpenMP）ジョブ

```
#!/bin/sh
#PJM -L "rscgrp=school"
#PJM -L "node=1"
#PJM -L "elapse=10:00"
#PJM -j
#
export OMP_NUM_THREADS=16
./a.out
```

←シェルを指定
←利用リソースグループ名
←利用ノード数
←最大経過時間（hh:mm:ss）
←標準エラー出力をマージして出力
←OpenMP並列数を指定
←プログラムの実行

環境変数 OMP_NUM_THREADS にOpenMP並列数を設定

ジョブの管理

- ジョブの状態表示

```
$ pjstat [option]
```

  - “-v”オプション：詳細なジョブ情報を表示
  - “-H”オプション：終了したジョブ情報を表示
  - “-A”オプション：全ユーザのジョブ情報を表示
- ジョブのキャンセル

```
$ pjdel [JOB_ID]
```

  - [JOB_ID]は“pjstat”で表示されるものを指定
  - 例）[JOB_ID]が12345のジョブをキャンセル

```
$ pjdel 12345
```

ジョブ結果の確認

- バッチジョブの実行が終了すると、標準出力ファイルと標準エラー出力ファイルがジョブ投入ディレクトリに出力される
  - 標準出力ファイル：ジョブ名.oXXXXX
  - 標準エラー出力ファイル：ジョブ名.eXXXXX
    - デフォルトのジョブ名はジョブスクリプトのファイル名
    - XXXXXには[JOB_ID]が入る
    - ジョブスクリプト内で“#PJM -j”を指定した場合には、標準エラー出力はマージされ標準出力ファイルのみ出力される
  - 例）p.7の thread_omp.shを投入し、[JOB_ID]に12345が割り当てられた場合：thread_omp.sh.o12345 が出力

OpenMPの演習（入門編）

OpenMP復習

- （ノード内）スレッド並列のためのAPI仕様
- 逐次プログラムに指示文を挿入
- parallel 構文
  - 複数のスレッドにより並列実行するリージョンを指定

F

```
!$ omp parallel
    並列リージョン
!$ omp end parallel
```

C

```
# pragma omp parallel
{
    並列リージョン
}
```

OpenMP演習1:実行時ライブラリ関数

- 並列計算を行う際にプログラムの管理がしやすくなる
- 関数を利用するには以下の文を追加

F
```
include "omp_lib.h"
  もしくは
use omp_lib
```

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

- 実行時ライブラリ関数の例

関数名	戻り値
omp_get_thread_num()	自身のスレッド番号(整数)
omp_get_num_threads()	総スレッド数(整数)
omp_get_wtime()	経過時間(秒:倍精度実数)

OpenMP演習1:実行時ライブラリ関数

- プログラム:omp1.f90

F

```
program omp1
   use omp_lib
   implicit none
   real(8) :: t1, t2

   t1 = omp_get_wtime()
!$omp parallel
   print *, "#ID:",omp_get_thread_num(),"of",omp_get_num_threads()
!$omp end parallel
   t2 = omp_get_wtime()
   print *, "#time:", t2-t1
end program omp1
```

演習1

上記のプログラムを作成し、コンパイル・ジョブ実行し、結果を確認してください

OpenMP演習1:実行時ライブラリ関数

- プログラム:omp1.c

C

```c
#include <stdio.h>
#include <omp.h>
int main(void){
   double t1, t2;
   t1 = omp_get_wtime();
#pragma omp parallel
   {
   printf("#ID: %d of %d ¥n",omp_get_thread_num(),omp_get_num_threads());
   }
   t2 = omp_get_wtime();
   printf("#time: %le¥n",t2-t1);
   return 0;
}
```

演習1

上記のプログラムを作成し、コンパイル・ジョブ実行し、結果を確認してください

OpenMP演習1:実行時ライブラリ関数

- 実行結果の例

```
#ID: 11 of 16
#ID: 13 of 16
#ID: 9 of 16
#ID: 1 of 16
#ID: 10 of 16
#ID: 0 of 16
#ID: 5 of 16
#ID: 15 of 16
#ID: 4 of 16
#ID: 6 of 16
#ID: 7 of 16
#ID: 8 of 16
#ID: 2 of 16
#ID: 12 of 16
#ID: 3 of 16
#ID: 14 of 16
#time: 1.557683944702148E-02
```

OpenMP演習２：ループ構文

- doループおよびforループの並列処理

F

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

C

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

- 又は、まとめて

F

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

C

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

OpenMP演習2：ループ構文

- プログラム：omp2.f90

F

```
program omp2
   implicit none
   integer, parameter :: n = 100000000
   real(8) :: x(n), y(n), pi, dx
   integer :: i

   pi = 4.0d0*atan(1.0d0)
   dx = 2.0d0*pi/n
   do i = 1, n
      x(i) = dx*i
      y(i) = sin(x(i))
   end do
   print *, y(1), y(n)
end program omp2
```

演習2

1. 上記プログラムのdoループ部分をOpenMPで並列化する
2. doループ部分の計算時間を計測・出力し、並列数を変えたときに計算時間がどう変わるかを確認する

OpenMP演習2：ループ構文

- プログラム：omp2.c

C

```
#include <stdio.h>
#include <math.h>
int main(void){
   int n = 100000000, i;
   double x[n], y[n], pi, dx;
   pi = 4.0*atan(1.0);
   dx = 2.0*pi/n;
   for(i = 0; i < n; i++){
      x[i] = dx*(i+1);
      y[i] = sin(x[i]);
   }
   printf("%le %le¥n", y[0], y[n-1]);
   return 0;
}
```

演習2

1. 上記プログラムのforループ部分をOpenMPで並列化する
2. forループ部分の計算時間を計測・出力し、並列数を変えたときに計算時間がどう変わるかを確認する

OpenMP演習３：shared節・private節

- データ共有属性
  - 並列リージョン内での変数の共有属性
  - shared 節
    - 例）
```
!$omp parallel shared(a)
```
F
```
#pragma omp parallel shared(a)
```
C
      - 変数 a はスレッド間で共有される（共有変数）
  - private 節
    - 例）
```
!$omp parallel private(b)
```
F
```
#pragma omp parallel private(b)
```
C
      - 変数 b は各スレッド固有となる（プライベート変数）

デフォルトではループ制御変数は private 、それ以外は shared

OpenMP演習３：shared節・private節

- プログラム：omp3.f90

F

```
program omp3
   implicit none
   integer, parameter :: n = 100000000
   real(8) :: x, y(n), pi, dx
   integer :: i

   pi = 4.0d0*atan(1.0d0)
   dx = 2.0d0*pi/n
   do i = 1, n
      x = dx*i
      y(i) = sin(x)
   end do
   print *, y(1), y(n)
end program omp3
```

演習３

1. 上記プログラムのdoループ部分をOpenMPで並列化する
2. その際、shared, private を明示的に指定してみる
3. 不適切な指定をした場合、どうなるかやってみる

OpenMP演習３：shared節・private節

- プログラム：omp3.c

C

```c
#include <stdio.h>
#include <math.h>
int main(void){
   int n = 100000000, i;
   double x, y[n], pi, dx;
   pi = 4.0*atan(1.0);
   dx = 2.0*pi/n;
   for(i = 0; i < n; i++){
      x = dx*(i+1);
      y[i] = sin(x);
   }
   printf("%le %le¥n", y[0], y[n-1]);
   return 0;
}
```

演習３

1. 上記プログラムのdoループ部分をOpenMPで並列化する
2. その際、shared, private を明示的に指定してみる
3. 不適切な指定をした場合、どうなるかやってみる

OpenMP演習４：reduction節

- データ共有属性
  - reduction 節
    - 例）

```
!$omp parallel reduction(+:c)        F
#pragma omp parallel reduction(+:c)  C
```

      - 変数 c は private と同様各スレッド固有であるが、並列リージョン終了時に全スレッドについて総和（+）がとられる
    - 指定した演算で各スレッドの値を集計した結果が代入される
      - 演算子： +, -, * 等
      - 一部組み込み関数（Fortran）： max, min 等

OpenMP演習4：reduction節

- プログラム：omp4.f90

F

```
program omp4
   implicit none
   integer, parameter :: n = 100000000
   real(8) :: x(n), y(n), pi, dx, z, l1, l2
   integer :: i

   pi = 4.0d0*atan(1.0d0)
   dx = 2.0d0*pi/n
   do i = 1, n
      x(i) = dx*i
      y(i) = sin(x(i))
   end do
   l1 = 0.0d0
   l2 = 0.0d0
   do i = 1, n
      z = x(i)*y(i)
      l1 = l1 + z*dx
      l2 = l2 + z*(1.0d0-z)*dx
   end do
   print *, l1, l2
end program omp4
```

演習4

1. 上記プログラムの2つのdoループ部分をOpenMPで並列化する
2. reduction の指定が不適切な場合、どうなるかやってみる

OpenMP演習4：reduction節

- プログラム：omp4.c

C

```c
#include <stdio.h>
#include <math.h>
int main(void){
   int n = 100000000, i;
   double x[n], y[n], pi, dx, z, l1, l2;
   pi = 4.0*atan(1.0);
   dx = 2.0*pi/n;
   for(i = 0; i < n; i++){
      x[i] = dx*(i+1);
      y[i] = sin(x[i]);
   }
   l1 = 0.0;
   l2 = 0.0;
   for(i = 0; i < n; i++){
      z = x[i]*y[i];
      l1 += z*dx;
      l2 += z*(1.0-z)*dx;
   }
   printf("%le %le¥n", l1, l2);
   return 0;
}
```

演習4

1. 上記プログラムの2つのforループ部分をOpenMPで並列化する
2. reduction の指定が不適切な場合、どうなるかやってみる