2004 年 6 月 8 日

長谷川秀彦

情報処理演習 II
Parallel Computing on Shared Memory Machine

1. OpenMP

OpenMP は共有メモリ方式マルチプロセッサ上の（並列）プログラミングモデルであり、ベースとなる言語（ C や Fortran)に並列化の指示文 directive を加えることによりコンパイラが並列化を行う。 1997 年に Fortran の API（ Application Program Interface), 1998 年に C の API が決定された（ 詳細は http://www.openmp.org/)。

N=100 の行列積プログラムを 4 台の CPU で実行するとしよう。共有メモリでは、配列 a, b, c はすべてのプロセッサからアクセスできる。 j のループに対する並列処理では、CPU-0 が 0 から 24, CPU-1 が 25 から 49, CPU-2 が 50 から 74, CPU-3 が 75 から 99 というように 4 台の CPU で均等に分担すればよい。原型のプログラム：

```
for ( j = 0; j<100, j++)
  for ( k=0; k<100, k++)
    c[i][j] = c[i][j] + a[i][k] * b[k][j]
```

を分割して 4 台の CPU 用にプログラムを書き直すと

```
for ( j0 = 0; j0<25, j0++)        /*      for CPU-0     */
  for ( k0=0; k0<100, k0++)
    c[i][j0] = c[i][j0] + a[i][k0] * b[k0][j0]
for ( j1 = 25; j1<50 j1++)        /*      for CPU-1     */
  for ( k1=0; k1<100, k1++)
    c[i][j1] = c[i][j1] + a[i][k1] * b[k1][j1]
for ( j2 = 50; j2<75, j2++)       /*      for CPU-2     */
  for ( k2=0; k2<100, k2++)
    c[i][j2] = c[i][j2] + a[i][k2] * b[k2][j2]
for ( j3 = 75; j3<100, j3++)      /*      for CPU-3     */
  for ( k3=0; k3<100, k3++)
    c[i][j3] = c[i][j3] + a[i][k3] * b[k3] [j3]
```

のようになる。これらを 4 台の CPU に割り当てて実行させればよい。 j のループが 4 台の CPU 用に分割されたので、ループの動く範囲が変わり、それぞれ別の変数を割り当てなければならない（ CPU は 4 台でも、広いメモリ空間はひとつしかない)。同様の理由

で、その内側のループにも異なる変数を使う必要がある。このように CPU 毎に異なる変数を private variable という。このような機械的な置き換えをコンパイラに任せるのが OpenMP の発想で、

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

のように、分割する for ループの直前に #pragma omp parallel for と入れれば、コンパイラが自動的に複数の CPU で実行可能な機械語プログラムを生成してくれる。しかも CPU 台数は実行時に環境変数で与えればよい。

この演習では PGI C Compiler と Omni OpenMP Compiler Ver. 1.2 を用いる。 PGI C Compiler は The Portland Group (http://www.pgroup.com/) が販売している商用コンパイラで、 Omni OpenMP Compiler は RWCP (Real World Computing Partnership; かつてつくば三井ビルにあったが、現在は解散)が作成したフリーの OpenMP コンパイラである。これらのコンパイラの内部では POSIX threads 呼び出しを含む C ソースプログラムが生成され、それが C コンパイラ（ Omni OpenMP Compiler の場合は GNU gcc コンパイラを使用）によって実行形式が作られる。POSIX threads は並列処理に関連した Operating System の機能である。中間でどんな C 言語のプログラムを生成しているかを見るには、

```
clover% ompcc  -t  ソースファイル
```

として、 tmp.* というファイルを眺めて見ればよい。単純なプログラムなら、核となるループがどのように並列化されているかがわかるだろう（複雑なコードだと凡人の理解をこえる)。 PGI C Compiler の場合、 OpenMP の pragma を有効にするためには –mp オプションを指定する。さらに –P オプションを指定すると、 *.i というファイルに pragma に従って作られた並列コードが出力される（未確認)。

```
clover% pgcc –mp  -P ソースファイル
```

並列度は使用する threads 数を環境変数 OMP_NUM_THREADS に

```
clover% setenv OMP_NUM_THREADS 4
```

のように指定する。clover は 4 CPU なので、5 以上の値を指定すると 1 台の CPU で複数の threads が動作することになり、多くの場合、性能向上は望めない。 Omni OpenMP Compiler の使い方は ompcc –h とすれば表示される。 PGI C Compiler で OpenMP を使った並列プログラムを作成するには、コンパイラオプションに –mp を指定し、リンク時にも実行時ライブラリとして –mp を指定する（マニュアルには書いてないが、必須)。 PGI Compiler のマニュアルは http://www.slis.tsukuba.ac.jp/~hasegawa/DELL/ にある。

```
clover% ompcc  -h
clover% pgcc  –mp  ソースファイル  -mp
```

並列化効果を調べるには実行時間を測定する。複数の CPU が動いているからといって、すべての CPU の実行時間を合計するようなことはしない。よくやるのは、最初から最後まで動いている CPU にどれくらいかかったかを、壁時計（ Wall Clock time ）で測定する。並列処理では通信時間や待ち時間が生じるが、 CPU 時間にはそのような影響が間接的にしか反映しないため、占有使用の状態で経過時間を測定したほうがよい。測定の際は、混雑など、他の影響が入り込まないようにして、少なくとも 2,3 回の測定の平均をとるべきだろう。また、入出力は CPU での処理に比べて 1000 倍以上（もっと？）遅いので、時間計測の際には、「入出力を測定しない」ように注意しなければならない。

CPU 台数を増やしたとき、１台（並列版が 1 台では実行できない場合は 2 台）に対する時間短縮率の逆数を並列化効率という。台数が増えたとき、並列化効率が傾き１の直線にそって良くなるのが理想的な並列プログラムで、並列化効率が線形であるという（傾きが０に近いと恥ずかしい！）。他のユーザが同時に使っていたときには、複数の threads がひとつの CPU で実行され、混雑していた CPU に割り当てられた threads の実行時間によって全体の実行時間が決まってしまう。だれがどんなプログラムを実行しているかは、top や ps w コマンドで調べられる。プログラムが暴走した場合は CTL-C または CTL-D を入力する。

```
clover% top
clover% ps
```

一般の（非並列化）コンパイラでは #pragma 行をコメントとみなすので、並列化したソースプログラムはコメントが増えただけでコンパイル結果には（実行結果にも）影響がない。 OpenMP では逐次プログラムを徐々に並列化できる。OpenMP コンパイラは「並列化してもいい」ように、余計なコードを生成するから、まったく並列化していないソースコードを OpenMP コンパイラにかけるのは得策でない。

OpenMP の directive はいろいろあるが、今回の演習では 2 種類だけを使う。 2000 年の情報処理演習 II のレポートとして、先輩が作ってくれたとてもわかりやすい説明が http://www.slis.tsukuba.ac.jp/~hasegawa/DELL/ にある。並列化の結果は（コンパイラにバグがない限り）すべてがプログラマの責任なので、プログラムの並列化にはじゅうぶんな注意がいる。実行するたびに結果が変わる並列プログラムは正しい並列プログラムとは言わない。

#pragma omp parallel for あるいは #pragma omp parallel for private(s,t)

直後の for ループを並列化する。 private(s,t) はループ変数以外に s と t が private variable であることを宣言する。ループ変数は自動的に private variable になる。ループは一定の stride で 論理演算子は <. <=, >, >= など、 break などによるループ外への飛

び出しがないことなどの条件がある。

#pragma omp parallel for reduction(+:s)

総和計算 s += a[i] などを並列実行する。 s は各 CPU で部分和を求め、最後にそれを足し合わせる必要がある。数値に敏感な計算では、部分和を取ってから総和をとることで結果が異なることもある。 + の代わりに、 -, *, max, min なども使える。

#pragma omp parallel

直後のブロック {} を並列実行させる。 for 文以外から構成されるブロックに使われる。

#pragma omp atomic

parallel のブロック中で、この部分だけは逐次に実行することを指示する。

わからないことは早めに明らかにするように。問題に出くわしたときは、本やマニュアルを調べたり、知人に聞いたり、メールで hasegawa@slis.tsukuba.ac.jp に聞いたり、 211 研究室を訪れるなり（不在のことが多い！）、時間に余裕をもって色々と試行錯誤してほしい。

参照文献

1) 湯浅太一、安村道晃、中田登志之編．はじめての並列処理，共立出版， 1998, 244p. bit 別冊
2) Bil Lewis, and Daniel J. Berg. 岩本信一訳．マルチスレッドプログラミング入門，アスキー出版局， 1996, 327p. ISBN 4-7561-1682-5
3) Chandra, R., et al. Parallel Programming in OpenMP, Morgan Kaufmann Publishers, 2001

Ⅰ．　OpenMP を用いた高速化

1. 教材ファイルのコピーを行う

clover% cp　-rp　~hasegawa/IPP2/June08　.　ピリオド

カレントディレクトリにディレクトリ June08 が作られるので、まずは June08 内のファイルが何をしているかを調べてみよう

2. Makefile を使ってコンパイルした並列コードを実行してみよう

3. A というファイルを作成し、Omni OpenMP コンパイラに必要な設定を入力する

setenv CLASSPATH　~hasegawa/jdk1.1.3/lib/classes.zip:.

set path=(~hasegawa/bin ~hasegawa/jdk1.1.3/bin $path .)

4. source コマンドを実行し、 printenv で設定が変わったことを確認する

clover% printenv

clover% source　A

clover% printenv

5. Makefile 中のコンパイラの指定を ompcc に変えて実行してみよう

6. openmp.c を gcc コンパイラでコンパイルして実行する

clover% gcc –O openmp.c

7. openmp.c を ompcc コンパイラでコンパイルする

8. 環境変数 OMP_NUM_THREADS に設定してある threads 数を 1, 2, 3, 4, 8 と変えて openmp.c の実行形式を実行し、どのステップがどれだけ高速化されるかを観察する

clover% setenv OMP_NUM_THREADS 4

clover% a.out

9. 同様に openmp1.c を条件を変えて並列実行しなさい

10. コンパイラを PGI Ccompiler に変えて並列実行しなさい

課題　行列積プログラムを OpenMP を用いて高速化しなさい。性能は MFLOPS 値で評価する（元々高速なコード、高速なコンパイラを使用するのが早道)。この結果を OpenMP についての実験結果とともに簡潔にまとめて 6 月 15 日に提出しなさい。

（注意） CPU 台数は限られているので、測定は利用者のいない時間帯にしよう。 threads 数を変えて同じ実行形式を実行するなら、コマンド列をファイル B に入れておく。Chmod コマンドでパーミッションを変えて、このファイル名を入力すれば、中のコマンド列が実行できる。これを shell script という。

clover% chmod　a+x　B

clover% B