2004 年 6 月 1 日

長谷川 秀彦

情報処理演習 II -1
High Performance Computing

1. 高性能計算の基礎

複数の CPU が共通のメモリにアクセスするコンピュータを共有メモリ方式並列コンピュータ、正確には Shared Memory Parallel Computer あるいは SMP（ Symmetric Multiple Processor）という。一方、複数の CPU がそれぞれのメモリを持ち、独立に動作するコンピュータを分散メモリ方式並列コンピュータ、正確には Distributed Memory Parallel Computer という。今回は DELL Power Edge 6350（ CPU: Pentium III xeon 550MHz × 4, Memory: 2GB, O/S: RedHat Linux 6.1, ホスト名は clover）で並列プログラミングを行う。

例えば 4 台の CPU を持った共有メモリ方式のコンピュータで、あるプログラムを 1 台の CPU で実行すると 20 分かかるとしよう。CPU が 4 台あるから、4 人までは 20 分で結果を得ることができる。 5 人となると、 4 台の CPU が短い時間間隔で 5 人のプログラムを少しずつ実行するので、平均すると 25 分位かかるだろう。このような方式を Time Sharing System という。 UNIX ワークステーション、対話処理サーバ uni などはこの方式をとっており、利用者が少なければ早く結果が得られ、混雑していればその分だけ遅くなる。一般の C 言語プログラムは同時に複数の CPU を使用するようにはなっていないため、 CPU が 4 台あっても実行時間が 1/4 の 5 分になるわけではない。この演習では、複数の CPU を独占して実行時間を短縮する。

まず、 C 言語で書かれた行列積プログラムを考える。行列サイズ N を大きくすれば CPU 時間は単調に増加する。 CPU 時間で性能を評価するのは無意味なので、ここでは MFLOPS（Million Floating Operations Per Second）値を使う。 N x N の行列積プログラムの場合、必要な浮動小数演算数は約 $2N^3$ 回なので、演算数を実行に要した CPU 時間で割れば MFLOPS 値がでる。 MFLOPS 値は 1 秒間にどれだけの浮動小数演算を実行したかを示し、「実行スピード」の指標となる。 Intel 社の Pentium III xeon 550MHz は 1 クロックにひとつの浮動小数演算が実行できるので、プログラムが 550 MFLOPS で実行できれば、ハードウェアの性能を 100 % 出し切ったことになる。なお、最近では演算数 Floating Point Operations のことを FLOPS といい、 1 秒あたりの演算数を FLOP/S ということがある。文献などを見るときには注意がいる。

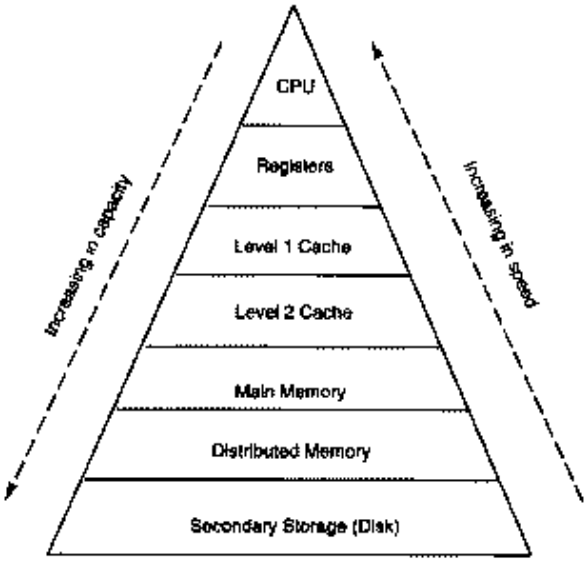

Figure 1 : A typical memory hierarchy.

図 1　コンピュータのメモリ階層

複数の CPU を使うときは、仕事を分散させるための処理も必要になるので、複数の CPU が消費した CPU 時間を合計すれば 1 台の CPU で実行したときと同じかそれ以上になる。このように、並列化 (parallelilzation) に伴って必要になった CPU 時間を並列化オーバーヘッドという。並列化によって短縮されるのは、結果が得られるまでの時間、すなわち経過時間である。経過時間のことを wallclock time ということもある。並列処理は資源浪費型の高速計算である。

2. 逐次プログラムの高速化

さて、ハードウェアの性能を出し切るために重要なのは、

（1） ハードウェアの特徴を生かすこと

（2） コンパイラオプションを活用すること

（3） プログラムのチューニング

である。

コンピュータのメモリ階層は図１のようになっており、上の階層は高速だが容量が小さく、下の階層はその逆である。階層間でのデータ移動は、演算スピードに比べると大幅に遅いので、その間 CPU は遊んでしまう。高速化の基本は余計なデータ移動を避けることである。ハードウェアは、プログラムのメモリ参照が局所性を持つことと、連続参照が多いことを前提に、なるべく安価な構成で高速性を発揮するように作られている。したがって、プログラムは連続的なメモリ参照を心がけることが望ましい。

さて、行列積プログラムの核となるのは

```
c[i][j] = c[i][j] + a[i][k] * b[k][j]
```

で、乗算１，加算１の２浮動小数演算である。３重のループの変数 i, j, k の順序は自由なので、６通りのプログラムが考えられる。

大規模なソフトウェアでは、メインプログラム（メイン関数）とサブプログラムを別々のファイルとして作成・管理することが多い。しかし C 言語の仕様には整合配列がないため、個々の関数に独立性を持たせるには、 2 次元配列を 1 次元化する必要がある。演習のプログラムでは、 a[N][N] で宣言された配列の a[i][j] 要素は N*i+j 番目の要素であることを利用して 1 次元配列に格納している。メモリ上では第 2 添字の方向が連続になる。なお、 Fortran では a(i,j) 要素は N*(j-1)+i 番目になり、メモリ上で第 1 添字の方向が連続になっている。こちらはコンパイラが自動的に変換してくれる。

コンパイルオプションのうち、実行性能に大きく関係するのは最適化オプションである。コンパイラは、ソースプログラムを機械語に翻訳する際、不要な命令を取り去ったり、命令の順序を入れ替えたりして、プログラムが小さなメモリで高速に実行できるよう最適化 (Optimize) する。特に Super Scalar Processor と言われる１クロックで複数の演算が実行できる RISC Processor の場合は、演算順序の入れ替えが重要で、命令の並べ替え方法の善し悪しで性能が大きく変わる。計算結果が正しいことは大前提なので、最適化はそう簡単なことではない。

コンパイラオプションを調べるには

```
uni% man cc
clover% man gcc
clover% man pgcc
```

のようにする。 uni の場合は HP の C コンパイラ、 clover の場合は GNU の gcc コンパイラと PGI の C コンパイラがある。最適化に関連した -O, +O2, +O4 などというオプションが見つかるだろう。その他、プログラムのデバッグとチューニングには、クロスリファレンスリストの出力、アセンブラリストの出力なども役立つ。なお、マシンが変わればコンパイラも変わり、オプションの指定方法も変わるので、マニュアルを確認する習慣をつけよう。

チューニングはいろいろな職人芸があり、一口では説明できないが、ここではデータの再利用の観点だけを考えてみよう。原型（実際は 1 次元化）は以下のとおりである：

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

これを、 k のループで不変な c[i][j] をレジスタにおくことを示すため、変数 s を用いて

```
for ( i = 0; i<N, i++)
```

```
for ( j = 0; j<N, j++)
  s = 0.0
  for ( k=0; k<N, k++)
    s = s + a[i][k] * b[k][j]
  c[i][j] = s
```

と書く。 k のループで c[i][j] の LOAD と STORE を繰り返すと、メモリ参照が増え、性能は大幅に劣化する。そこで j のループを一つおきにした次のプログラム：

```
for( j = 0; j<N, j=j+2 )      一つおきに
  s = 0.0; t = 0.0
  for ( k=0; k<100, k++)
    s = s + a[i][k] * b[k][j]
    t = t + a[i][k] * b[k][j+1]
  c[i][j] = s; c[i][j+1] = t
```

では、メモリから持ってきた a[i][k] が 2 回分の計算に利用できる。ここでは a[i][k] に対するメモリ参照や、ループの判定回数が半分になるなどのメリットがある。このようにループ内により多くの演算を詰め込むことをループアンローリング (loop unrolling) という。ループアンローリングはメモリからレジスタへのデータの LOAD/STORE 回数や分岐（ for 文の終了判定）が削減されるため、データの再利用が促進される。最近のプロセッサの性能向上は著しいが、メモリのアクセススピードの向上はプロセッサの性能向上に追いついていないため、高速化にはこのような手法が不可欠である。

3. 行列積のチューニング例

用いたコンピュータは SGI O2 で、メモリ 96 MB, CPU MIPS R5000SC 180MHz, Cache 32 KB である。この CPU は 3 段のパイプラインで倍精度浮動小数演算性能（カタログ性能）は 180 MFLOPS だが、実測による最高性能は約 120 MFLOPS である。

素朴な行列積 C = AB の中心部分は

```
c[i][j] = c[i][j] + a[i][k]*b[k][j]
```

となり、ループの取り方は i,j,k の組合せにより 6 通りある。gcc の最適化オプションを効かせた場合の実行性能を図２に示す。横軸の目盛は log スケールにしてある。これから、I-K-J または K-I-J を高速化すれば良さそうだということがわかる。大規模問題では、J-K-I と K-J-I はまったく性能がでない。これはメモリ参照が連続でないことが理由である。

図３、図４に、プログラム K-I-J と I-K-J にループ案ローリングを用いて高速化した結果を示す。ここでは、ループを二つ跳び、八つ跳びとし、一つのループ内により多くの演算を詰め込むことで、いったん CPU に持ってきたデータを再利用している。行列積の演算回数を減らすことは不可能だが、このようにデータの LOAD, STORE, 分岐などの命令を減らすことが実行時間の短縮につながる。このほか、コンパイラによる最適化がしやす

くなるよう、スカラー変数を用いてループ内で不変であることを明示したり、１次元配列を用いて明示的にデータの再利用をさせるなど、涙ぐましい努力をしている。

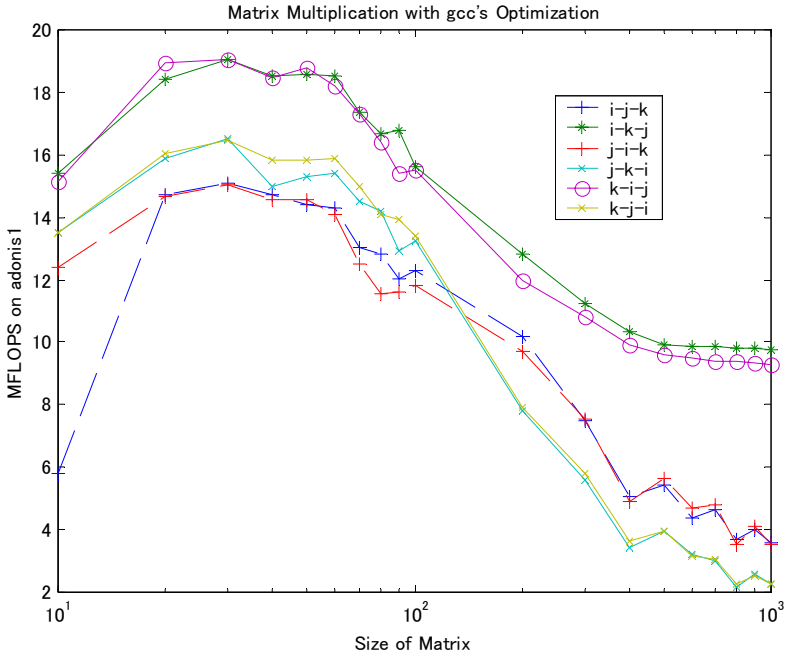

図　２　ループ順序の違いによる行列積プログラムの性能

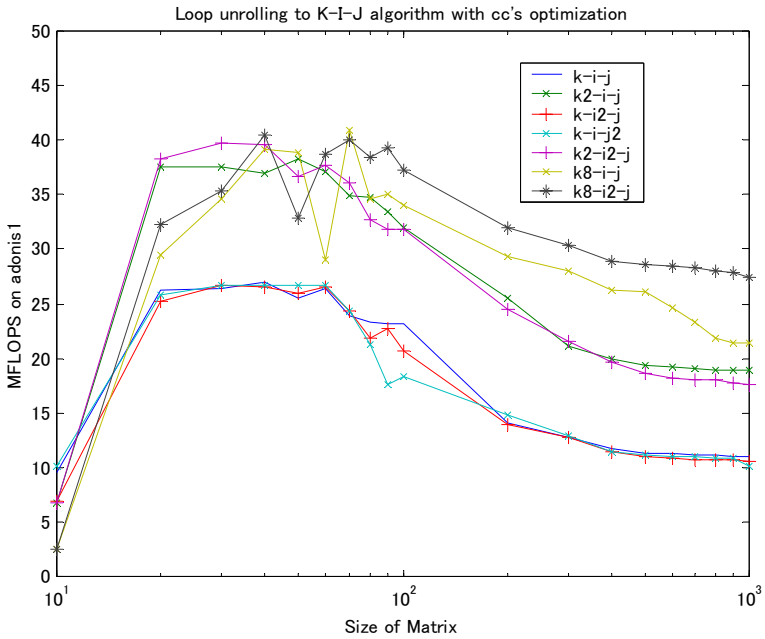

図　３　Ｋ－Ｉ－Ｊプログラムの高速化

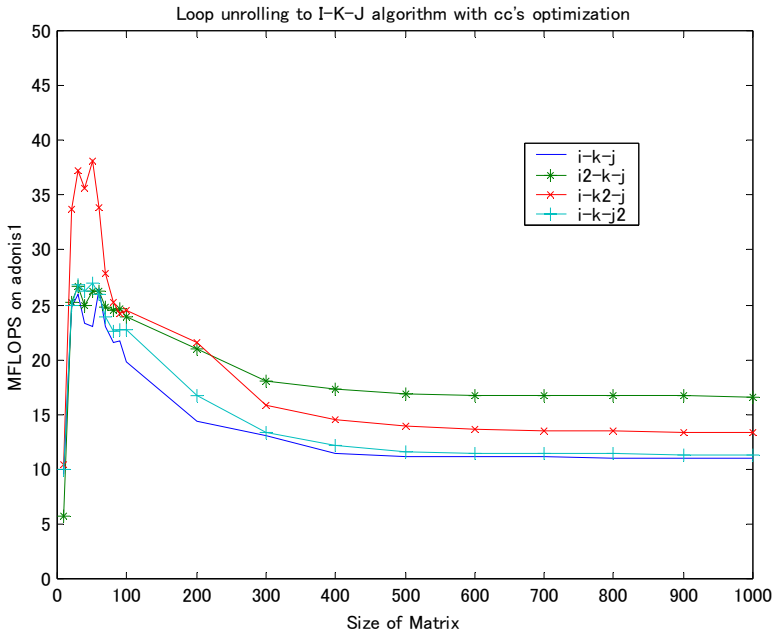

図 4　Ｉ－Ｋ－Ｊプログラムの高速化

4.　Makefile の使い方

大規模ソフトウェアでは、ソースプログラムを一つのファイルにまとめるようなことはしない。一部の関数が他のプログラムでも使われたり、多くの場合、修正や変更は局所的に行われることがその理由である。一般には、関数あるいは機能ごとにファイルを分け、それぞれのコンパイル結果をオブジェクトファイルとして保存し、修正があったときは修正のあったファイルだけをリコンパイルする。そして、それらをリンケージエディタで実行形式にまとめあげる。これは、プログラムをきちんと管理し、無駄なコンパイルをしないという意味も持つ。このような手法を分割コンパイルという。

簡単な例を作ろう。行列積プログラムは 4 つの関数からなっているので、それぞれ関数名を名前に持つファイル main.c, input_array.c, multiply_array_ijk.c に入れる。まずはコンパイルするために

```
clover% gcc -O -c main.c
clover% gcc -O -c input_array.c
clover% gcc -O -c multiply_array_ijk.c
clover% gcc -O -c second.c
```

を実行する。 -c オプションはコンパイルだけで実行形式を作らないことを意味する。コンパイル結果がオブジェクトファイル main.o, input_array.o, multiply_array_ijk.o, second.o に作られる。 ls コマンドと file コマンドで確認しておこう。これらを一つにま

とめて実行形式 a.out を作るには

clover% gcc　main.o input_array.o multiply_array_ijk.o　second.o

とする。実行形式の名前を multiply とするには、cc の次に -o multiply を指定する。ここでは同じ gcc コマンドがリンケージエディタとして使われている。

行列積プログラムのチューニングでは、 multiply_array_ijk.c をいろいろ作っては測定することの繰り返しになるだろうから、 main.c, input_array.c, second.c は最初に 1 回だけコンパイルしておけば十分である。更新されたファイルをきちんとコンパイルするとか、実行形式が最新版になるように管理するのは結構手間がかかる。このような管理を手助けしてくれる UNIX 上の仕組みが Makefile である。

まずは以下の内容を Makefile というファイルに作る（すでに用意してある）。

```
#                    A sample of Makefile

CC = gcc
CFLAGS = -O -Wall
OBJS = main.o input_array.o multiply_array_ijk.o second.o

all: multiply

multiply: $(OBJS)
        $(CC) -o $@ $(OBJS)

.c.o:
        $(CC) $(CFLAGS) -c $<

clean:
        rm -f multiply *.o core
```

先頭部分はマクロの定義で、変数 CC に gcc, CFLAGS に -O -Wall, OBJS に main.o と input_array.o と multiply_array_ijk.o と second.o が割り当てられる。最初の all: では multiply という仕事（ターゲット）を実行すると指定してあるので、 multiply: と書いてある部分に実行が移る。 multiply: は変数 OBJS, すなわち main.o, input_array.o, multiply_array_ijk.o, second.o から構成されている。ここで使われるオブジェクトファイル main.o, input_array.o, multiply_array_ijk.o, second.o の作成時刻がソースファイルの修正時刻より前（昔）なら自動的にリコンパイルが行われる（分単位）。コンパイル方法は .c.o ターゲットが参照される。オブジェクトファイルの準備が完了したときに実行されるコマンドは

$(CC) -o $@ $(CFLAGS) $(OBJS)

である。 $@ には ターゲット名 multiply が割り当てられるので、実際のコマンドは

gcc -o multiply　main.o input_array.o multiply_array_ijk.o second.o

となる。

.c.o: では、 .c ファイル（ソースプログラム）から .o ファイル（オブジェクトプログラム）を作成するコマンドを定義している。 $< には .c が展開される。

$(CC) $(CFLAGS) -c $<

つまり、

gcc -O -Wall -c xxx.c

が、必要なオブジェクトファイル xxx.o に対応するソースファイル xxx.c に対して繰り返し適用される。コンパイル規則は繰り返し参照されるので、ソースファイル名は変数として記述され、内容は参照時に決まる。最後は不要なファイルの削除で、ターゲット clean を実行すると、 rm -f multiply *.o core によって、実行ファイル multiply, オブジェクトファイル *.o, core ファイルが強制的に消去される。文頭の空白は Tab であることに注意してほしい。見た目では区別がつかないが、空白だと無視される。# 以降はコメントである。

Makefile の内容を実行するには、以下のように指定する。なお、 Makefile 以外のファイル名を使うときは -f ファイル名の指定がいる。

clover% make

clover% make -f ファイル名

なお Makefile 中のコンパイラオプションを変えたとか、コンパイラ名を変更したなど、ソースファイルの修正時刻に変更が加えられない場合は最新版とならない。その場合は

clover% make clean

で、いったんすべてのファイルを消去し、その後 make を実行する。特定の機能を実行するにはターゲット名を指定すればよい。なお、実行形式を保存するときはが mv コマンドで名前を変えて消されないようにする。不要なファイルは消去しておく習慣をつけよう。

参照文献

1) Kevin Dowd. High Performance Computing, O'Reilly & Associates, Inc., 1993, 371p. ISBN 1-56592-032-5 (第１版の翻訳あり、英語版は第２版がある)
2) Jack Dongarra, Iain Duff, Danny Sorensen, and Henk van der Vorst. Numerical Linear Algebra for High-Performance Computers, SIAM, 1998, 342p. ISBN 0-89871-428-1
3) Jack Dongarra, Iain Duff, Danny Sorensen, and Henk van der Vorst. Solving Linear Systems on Vector and Shared Memory Computers, SIAM, 1991, 256p. ISBN 0-89871-270-X (小国力訳、コンピュータによる連立一次方程式の解法、丸善。この本の改訂版が文献 2 であり、内容はだいぶ古くなっているが日本語が救いか？)
4) 寒川光． RISC 超高速化プログラミング技法、共立出版、 1995, 214p. ISBN 4-320-02750-7
5) 中沢喜三郎．計算機アーキテクチャ構成方式、朝倉書店、 1995, 570p. ISBN 4-254-12100-8
6) Google で Linpack ベンチマーク、 BLAS の高速化を検索してみよう

6 月 1 日の演習メニュー

1. clover のアカウントにパスワードをつける（パスワードを考えてね）
2. clover に login し、教材ファイルのコピーを行う

    clover% cp -rp ~hasegawa/IPP2/June01　.　ピリオド！

とすれば、カレントディレクトリに June01 というディレクトリが作られる

3. どのようなファイルがあるのかをながめてみよう（中身も）
4. Makefile を実行してみよう

    clover% make

5. multiply を実行してみよう。multiply を 3 回実行して実測結果を記録する

    clover% multiply

6. man コマンドで gcc の最適化オプションを調べる
7. Makefile 中の最適化オプションを変更して同様に実行しなさい
8. コンパイラを PGI の C コンパイラに替えて実行する
9. コンパイラとコンパイラオプションを変えたとき、性能がどう変わるかを調べる

A. ループの順序を 6 通り変化させて性能を比較しよう

B. multiply_array_*** を作成しなさい。*** には、一番高速だったループの順序、たとえば kji が入る（試してから）

C. multiply_array_*** を実行できるように main.c と Makefile を変更する

D. multiply_array_*** の性能を実測する

E. multiply_array_*** をチューニングして、高速化版 fastest_multiply_array.c を作成しなさい

F. 高速化版 fastest_multiply_array.c の性能を実測する

最適化オプション、ループの順序、プログラムのチューニングによって性能がどのように変化するかを体験するのが今回の目的である。その結果を A4 版 1-2 枚の簡単なレポートにまとめて、8 日に提出しなさい。結果をグラフにまとめるとなお良い。大事なことは、正しいプログラム、きれいなプログラム、速いプログラムである。なお、 N=1000 のときに 200MFLOPS 以上をだすプログラムの作者には「ごほうび」をだす（＝ 200MKLOPS はそう簡単には達成できない)。

（注意） N をあまりにも大きな値にして、他の利用者に迷惑をかけないように気をつけること。学外から直接 clover に login することはできないが、 uni を経由すればいつでも学外から login できる。利用する時間帯など、利用者相互で譲り合うことも必要。