2002 年 5 月 17 日

長谷川秀彦

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

1. 分散メモリ方式並列計算の基礎

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 Multiple Instruction Multiple Data type Parallel Computer という。これには複数の PC や Workstation をネットワークで結んで 1 台の並列コンピュータにしたようなものもあり、その場合は Cluster Computer という。一方、サーバのように複数の CPU が共有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、 Shared Memory Multiple Instruction Multiple Data type Parallel Computer あるいは Symmetric Multi-Processor (SMP) という。どちらの場合も各 CPU は他の CPU とは関係なしに、自分に与えられたプログラムを実行するので、全体としてみれば複数の CPU が別々のデータに対して別々の処理をしているので Multiple Instruction Multiple Data (MIMD 形式) という。 PC や Workstation は１時にひとつのデータしか処理できないので Single Instruction Single Data (SISD 形式) と呼ばれる。

分散メモリ方式並列コンピュータでも CPU に直結されたメモリには普通にアクセスできるが、別の CPU に接続されたメモリにはアクセスできない。他の CPU に接続されたメモリ上にあるデータは、相手の CPU にデータの取り扱いを依頼しなければならない。これには Message Passing Library と呼ばれる通信ライブラリを用いる。最近では MPI (Message Passing Interface) という de facto standard が広く使われている。データを要求したとき、相手が仕事を中断してその依頼に応えてくれたとしても、ネットワークの混雑などでデータが届くのに時間がかかって、自分の仕事が滞るかもしれない。送信と受信のタイミングが合わないとデッドロックが起こる。それらをうまくコントロールするのが分散並列プログラムのポイントである。基本は CPU 間でデータの移動を少なくし、各 CPU で独立な大きな仕事を実行させることである。 CPU の性能が向上し、多数の CPU を結合する高速ネットワーク技術も進歩しているので、数千台のプロセッサを結合して１台のコンピュータに仕上げるのが容易にできるようになり、 Top500 (http://www.top500.org/) に載るような大規模なスーパーコンピュータのほとんどは分散メモリ方式である。個々のプロセッサに PC を使えば、技術革新・量産の成果が取り入れやすく、きわめて経済的である。もちろん個々の CPU が SMP であってもよい。

本来なら Distributed Memory MIMD Parallel Computer nCUBE2 M5 （ホスト名 genie ）を用いるべきだが、サーバ室が工事中のため現在は使えないので、今回は SMP である DELL PowerEdge 6350 上で MPICH(http://www.mpi-forum.org/ ）を用いて分散並列プログラミングの初歩、特に Message Passing のしくみについて学ぶ。 MPICH はベンダが自社のマシンに MPI を実装する際にも参考にされるライブラリで、CPU が 1 台しかないワークステーションから稼働する。並列化性能は計算環境に依存し、今回の環境ではあまり高速化は期待できないが、そのままのコードを並列計算機で実行すれば高速に実行される（ことを期待しよう)。

本日の演習メニュー

1. MPICH に必要な以下の設定をホームディレクトリ直下のファイル .rhosts に入力する

    clover.ulis.ac.jp

2. 教材ファイルのコピーを行う

    clover% cp -rp ~hasegawa/IPP2/May17 .

3. まずは、サンプルプログラム Hello World! を入力し、コンパイル・実行してみよう。

```
clover% cat hello.c
#include <stdio.h>
main()
{
       printf("Hello World!¥n");
}
clover% cc -O hello.c
clover% a.out
Hello World!
```

4. MPI を用いてプログラムを並列実行させるためには、#include mpi.h の追加、main 関数への引数追加、並列処理の開始を宣言する MPI_Init 関数の挿入、並列処理の終了を宣言する MPI_Finalize 関数の挿入が必要である。これは約束事なので忠実に守ればよい。コンパイルには gcc コマンドの替わりに mpicc コマンドを、実行には mpirun コマンドを使う。 mpirun コマンドのパラメータ –np 4 は 4 つのプロセスで実行することを意味する。どんなプロセスが実行中かを調べるには top コマンド、暴走したプログラムを止め

るには CTL-C または CTL-D を入力する。なお、ここでの MPICH は UNIX の socket を利用して通信をしているため、プロセスの起動が遅い（忍耐がいる)。

```
clover% cat mhello.c
#include <stdio.h>
#include "mpi.h"
main(int argc, char* argv[])
{
        MPI_Init(&argc, &argv);
        printf("Hello World!¥n");
        MPI_Finalize( );
}
clover% mpicc -O mhello.c
clover% mpirun -np 4 a.out
Hello World!
Hello World!
Hello World!
Hello World!
```

5. これでは、どこで実行されているかが実感しにくいだろうから、関数 MPI_COMM_rank と MPI_COMM_size を用いて全体のプロセス数と自分のプロセス番号を調べるプログラム phello.c を作ろう。関数 MPI_COMM_size は全プロセス数、MPI_COMM_rank プロセス番号を返す。MPI_COMM_WORLD は mpi.h に定義されている変数である。プロセス数を p とすると、プロセスは 0, 1, 2, ... , p-1 というプロセス番号を持つ。phello.c をコンパイルし、パラメータ np の値を変えて実行しよう。なお、結果の表示順序は一定ではない（なぜか？)。

MPI_COMM_rank(comm MPI_COMM_WORLD, int *rank)
MPI_COMM_size(comm MPI_COMM_WORLD, int *size)

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char* argv[])
{
       int    my_rank, p;

       MPI_Init(&argc, &argv);
```

```
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        MPI_Comm_size(MPI_COMM_WORLD, &p);
        printf("Hello World! I am %d of %d.¥n", my_rank, p );
        MPI_Finalize();
}
clover% mpicc -O phello.c
clover% mpirun -np 5 a.out
Hello World! I am 0 of 5.
Hello World! I am 2 of 5.
Hello World! I am 1 of 5.
Hello World! I am 3 of 5.
Hello World! I am 4 of 5.
```

6. プロセス番号が偶数か奇数かで異なった結果を表示するプログラム phello1.c を作り、次のような結果を出力させなさい。

```
clover% mpirun -np 4 a.out
Hello World! I am even 0.
Hello World! I am odd 3.
Hello World! I am even 2.
Hello World! I am odd 1.
```

7. MPI では別のプロセスとのデータ通信に関数 MPI_Send と MPI_Receive を使い、プロセス番号 source, dest とメッセージタイプ tagを指定して通信対象を識別する。*buf はデータが格納されている領域の先頭、 count はデータ数である。 datatype はデータの内容を表し、 MPI_CHAR（文字)、 MPI_DOUBLE（倍精度実数)、 MPI_FLOAT（単精度実数)、 MPI_INT（整数)、 MPI_LONG（長整数）などが mpi.h で定義されている。任意の source からのメッセージを受け取る MPI_ANY_SOURCE, 任意のメッセージタイプを受け取る MPI_ANY_TAG なども定義されている。これらを使用したとき、どこからどんなメッセージを受け取ったかを知るには

source = status.MPI_SOURCE

type = status.MPI_TAG

のように記述すればよい。

MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, comm MPI_COMM_WORLD)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,

comm MPI_COMM_WORLD, MPI_Status *status)

プロセス 0 から、1 から p-1 までのプロセスへメッセージ(整数)を送る例が message.c である。(他の番号でもよいが) プロセス 0 が特別な意味をもつ並列モデルを Host-Node モデル、 Master-Slave モデル、 Parent-Children モデルなどと呼ぶことがある。

```
clover%  cat message.c
#include <stdio.h>
#include "mpi.h"
main(int argc, char* argv[]){
             /*     Each Nodes recives any messages from Node 0.
                                  2002. 5. 14 H. Hasegawa        */
       int my_id, i, j, type, dest, source, nodes;
       MPI_Status status;

       MPI_Init(&argc, &argv);
       MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
       MPI_Comm_size(MPI_COMM_WORLD, &nodes);

       type = 777;
       if( my_id == 0 )
              for (i=1; i < nodes; i++){
                     j = 2*i; dest = i;
                     MPI_Send(      &j,
                                  1,
                                  MPI_INT,
                                  dest,
                                  type,
                                  MPI_COMM_WORLD);
                     printf(" Node %d sent a message %d to %d¥n", my_id, j, d
est );
              }
       else {
              source = 0;
              MPI_Recv(      &j,
                           1,
                           MPI_INT,
                           source,
```

```
                        MPI_ANY_TAG,
                        MPI_COMM_WORLD,
                        &status);
            printf(" Node %d received a message %d from %d¥n", my_id, j, sou
rce );
      }
      MPI_Finalize();
}
clover% mpicc -O –o message message.c
clover% mpirun -np 4 message
 Node 0 sent a message 2 to 1
 Node 0 sent a message 4 to 2
 Node 0 sent a message 6 to 3
 Node 3 received a message 6 from 0
 Node 1 received a message 2 from 0
 Node 2 received a message 4 from 0
```

8. 1 から p-1 までのプロセスからプロセス 0 へメッセージ（整数）を送るプログラム message1.c を作りなさい。

9. プロセス 0 から、 1 から p-1 までのプロセスへメッセージ（整数 x ）を送り、各プロセスで 2^x を計算しプロセス 0 に送り返す。プロセス 0 で送り返された値の総和をとり表示するプログラムを message2.c として作成せよ。実行形式 message2 参照。

```
clover% mpirun -np 8 message2
 Sum of 2^k ( k = 0, 7 ) is 255
 Node 3 sent 8 ( = 2^3 )
中略
 Node 7 sent 128 ( = 2^7 )
 Node 1 sent 2 ( = 2^1 )
```

10. プロセス 0 から始めて次々と右隣に値を渡し、右端ノード p-1 までいったらプロセス 0 に渡す（リング ring ができる)。左隣から渡された値に自分のプロセス番号を加えて右隣へ渡せば、最後には 1 から p-1 までの和が計算できる。このプログラムを message3.c として作成せよ。実行形式 message3 参照。

```
clover% mpirun -np 8 message3
 Node 0 received 28 from 7
 Node 2 received 1 from L and passed 3 to R
中略
 Node 7 received 21 from L and passed 28 to R
 Node 3 received 3 from L and passed 6 to R
```

11. Binary Tree の例を考えよう。まず、奇数番のプロセスから左隣の偶数番のプロセスに値を送る。今度は偶数番のプロセスに 0 から連番をふり、最初と同じ操作を行う。一連の操作を繰り返すことにより、最終的には 0 番プロセスに情報を集約できる。これがうまくいくためには、プロセス数は 2 のべき乗 2^k でなければならない。このプログラムを message4.c として作成せよ。実行形式 message4 参照。

```
clover% mpirun -np 8 message4
 Number of valid nodes is 8
 Sum on Node 0 is 28
 Node 4 sent 22 to 0
 Sum on Node 4 is 22
中略
 Sum on Node 5 is 5
 Node 1 sent 1 to 0
 Sum on Node 1 is 1
```

12. 実行結果をファイル OUTPUT.txt に保存するには `script` コマンドを使って次のようにする。

clover% script OUTPUT.txt

この間の操作・出力すべてが OUTPU.txt に格納される

clover% exit

このファイルを uni に ftp で送って加工やプリントすればよい。 uni でのプリントは

uni% a2ps テキストファイル名 | lp –d プリンタ名

uni% pr テキストファイル名 | lptext –d プリンタ名

uni% pr テキストファイル名 | a2ps | lp –d プリンタ名

などとすればよい。 a2ps, pr, lp, lptext などがどんなコマンドかは man で調べてみよう。

課題： 9, 10, 11(11 はオプションでもよい）のプログラムを作成し、ソースリスト・実行結果に簡単な説明・コメントを書き加えて次回の演習時に持参せよ。特に、ソースプログラムにはコメント・インデントをつけるのは常識だということをお忘れなく。

並列化の結果は（コンパイラにバグがない限り）すべてプログラマが責任を負う必要がある。並列プログラムではデッドロックや、データの待ち続けなどが起こりやすいので、じゅうぶんな注意が必要である。まあ、楽しんで！

並列処理とか高性能計算のことを HPC (High Performance Computing), あるいは通信まで含めて HPCC (High Performance Computing and Communication) という。この分野の情報は PHASE (Parallel and High-performance Application Software Exchange, http://phase.hpcc.jp/), Netlib (http://www.netlib.org/) などの Web ページを見るのが定石である。ここには、 MPI や OpenMP などの仕様書の日本語訳、基本行列演算の自動チューニングライブラリ ATLAS (Automatic Tuning for Linear Algebra Softwares?) などもある。蛇足を言えば、これこそが Network Library であり、Digital Library である。

参考文献

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) William Gropp, Ewing Lusk, and Anthony Skjellum. Using MPI, Second Edition, The MIT Press, 1999

3) Peter S. Pacheco. Parallel Programming with MPI, Morgan Kaufmann Publishers, 1997(実は日本語訳がでている)

4) Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. MPI: The Complete Reference, The MIT Press, 1996