2004 年 6 月 15 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 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。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。いっぽう 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し、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 Parallel Computer nCUBE2 M5 （ホスト名 genie ）を用いるべきだが、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 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でも実行でき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）。

演習メニュー - I

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/June15 . ピリオド！

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. このプログラムでは、どの CPU 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、source, dest にプロセス番号、tag にメッセージタイプを指定して通信を行う。*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 から始めて次々と隣に値を渡し、端のノード p-1 までいったらプロセス 0 に渡す（リング ring ができる）。隣から渡された値に自分のプロセス番号を加えて隣へ渡せば、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. 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、全体は Binary Tree になっていなければならない。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 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。

2. 放送

MPI の場合、プロセス 0 からそれ以外のすべてのプロセスに情報を送るのが MPI_Bcast (broadcast; 放送) である。放送に対して、ふつうのメッセージ交換を「通信」という。 MPI_Bcast は細かく相手を指定することはできないが、システムが提供するメッセージパッシングライブラリに効率よく情報を届けるアルゴリズムが組み込まれている。broadcast と同様の機構を用いれば、部分和から総和を計算する MPI_Reduce, 最大値・最小値の計算など各プロセスの情報が効率よく集約できる。

MPI_Bcast, MPI_Reduce を用いて数値積分を実行する例が Integral.c である。これは、f120 翫さんのレポートを手直ししたもので、関数 4/(1+x^2) を0から1まで積分して円周率πを得る(本当かな？)。このプログラムでは、 [0, 1] 区間を n 等分し、細かく区切られた区間を一つずつ順に演算を担当するプロセスに割り当てる。このような割り当て方法を cyclic distribution という。実際は、プログラムに仕組みが組み込まれているので、MPI_Bcast で分割数 n だけを全プロセスに伝えればよい。プロセスは与えられた小区間に対して、関数の下側の領域を近似する矩形の面積を加え合わせる。各プロセスで部分和が計算できたら、 MPI_Reduce を用いて総和を計算する。 bnode, mask, allnds などの使い方に注意してプログラムをながめてほしい。

このプログラムでは、分割数 n を大きくして区間を細かくすればするほど(限度はあるが)正しい値に近づく。並列処理を用いることで、時間をかけることなく高品質の解が得られる。一般の逐次プログラムとは、部分和を取ってから総和をとるという違いがあるので、数値に敏感な計算の場合は注意がいる。とはいっても、部分和をとってから総和をとるほうが、丸め誤差の影響がでにくいので精度はよくなるはずである。

3. 時間測定

並列化効果を調べるには、最初から最後まで動いているプロセス（多くの場合はプロセス０）がどれくらいかかったかを、壁時計（ Wall Clock time ）で測定する。あるいは、仕事の開始時点と終了時点に同期（ synchronization ）を入れてその間の実行時間を測定する。複数プロセスが動いているからといって、すべてのプロセスの実行時間を合計するようなことはしない。並列処理ではメッセージの通信時間や待ち時間（オーバーヘッド）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。測定の際は、混雑の影響がでないように十分注意し、少なくとも 2,3 回の測定の平均をとる。

オーバーヘッドのため、個々のプロセスの CPU 時間を合計したものは、逐次プログラムより CPU 時間がかかるはずである。台数を増やしたとき、１台あるいは２台（並列版

のプログラムが１台では実行できない場合）に対する時間の短縮率の逆数を並列化効率という。台数が増えたとき、傾きが１の直線にそって並列化効率がよくなるも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)。

時間計測の例 timing.c はプロセス０から右回りに最終プロセスまでメッセージを送り、プロセス０がそのメッセージを受け取って終了するというリングを N 周繰り返して、隣のプロセスへメッセージを送るのに必要な平均時間を求めている。当然ながら、プロセス数が増えれば１周に必要な時間は長くなる。プログラムからもわかるように、ほとんどがメッセージ待ちの時間になる。

時間計測の際は、 MPI_Wtime を使い、２回の MPI_Wtime 呼び出しをすればその間の時間がわかる。また、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。入出力は CPU での処理に比べて 1000 倍以上（もっと？）遅いので、MPI_Wtime の間に入出力があると何を計測したのかわからない結果となる。

4. データ分散方法とプログラミングスタイル、負荷分散

処理時間がかかる例として、２種類の行列積プログラム MatMul1.c, MatMul2.c を作った。わかり易さを優先したため、インターフェースや性能には問題がある。このプログラムでは、 C = AB を計算する際、行列 B をすべてのプロセスに持たせ、 A を横切りにした BLOCK × n の小行列 A_1, A_2, …, A_l を各プロセスに送って計算させ、結果の BLOCK × n の小行列 C_1, C_2, …, C_l をプロセス０に集めている。プロセス間で A と C の連続的な領域が受け渡しできるよう、行ブロックとしてデータを分散させている。このようなデータ分割を block distribution という。高速計算のためにも、よけいなデータ移動を避けるためにも、メモリの連続アクセスが基本である。一方、これまでの例では cyclic distribution を使った。

二つのプログラムはまったく同じ計算をしているが、 MatMul1.c では

```
if( my_id == 0 ) {          プロセス 0 の処理
}
else {                      その他のプロセスの処理
}
```

が全体を通して１組になっているのに対し、 MatMul2.c では何組も出現している。MatMul1.c ではプロセス０の仕事とその他の仕事という形式で記述されているが、MatMul2.c ではある send に対応した receive の組という形式で記述されている。後者のやり方のほうがバグを作り込みにくくなるためか、複雑なプログラムでは後者の書き方をすることが多いように思う。どちらのやり方でも全く同じことが記述できるので、作者と自分の考え方を対応づけながらプログラムを読むことが大事である。並列処理では（意味

的に）プログラムが複数になる。複数のプログラムを作るとメンテナンスも大変なので、一つのプログラムで複数のプロセス用の機能を実現したSPMD（Single Program Multiple Data）形式のプログラムが好まれるが、プログラムが複雑になるなら、 SPMD などやめて別プログラムにしてしまえということだってありである。いずれにせよ、メンテナンスの問題を含めて、並列化の結果はすべてプログラマが責任を負う必要があり、並列化にはじゅうぶんな注意が必要である。

この行列積プログラムは行列サイズ N とプロセス数の組み合わせによって、プロセス１台がまるまる空いてしまったり、極端に仕事が少なくなったりする。このような場合、負荷分散（ load balancing ）に問題があるという。よい並列プログラムは、すべてのプロセスを均等に働かせることと、計算と通信を同時に実行して通信を隠蔽する。この例では、最適なブロックサイズ BLOCK をどう決めるとか、プロセス０がデータの送信・受信ばかりでなく積の計算に加わるとか、データを行ごとに cyclic に送るとか、いろいろと検討の余地がある

演習メニュー - II

(1) Integral.c をそのまま実行して正しく動作することを確認せよ

(2) √（1 - x^2）を 0 から 1 まで積分すると π/4 が得られる。これをプログラムとして実現しなさい

(3) 「関数の下側の領域を近似する矩形の面積を加え合わせる」代わりに「近似する台形の面積を加え合わせる」ようなプログラムを作りなさい

(4) 被積分関数、分割数、矩形か台形かなどを変えて、結果がどう変わるかを調べなさい。πの正確な値はインターネットで探そう（時間に余裕があるときに)。

(5) 条件を変えて timing.c を実行せよ

(6) プロセス 0 で printf を実行し、どのくらい遅くなるかを調べなさい

(7) 前回の Binary Tree の例を（無意味だけど）繰り返し、その間の実行時間を２種類の方法で測定しなさい

これまでのプリントをゆっくりと理解してほしい。色々と試行錯誤することは必要だが、ゆっくり考えれば、必ずできる！