平成 22 年 6 月 22 日

統計物理屋のための簡単 MPI 講座

渡辺宙志

東京大学情報基盤センター

概 要

MPI を用いた並列化について、基本的な概念と簡単な使い方を説明する。

目 次

---

1 はじめに

今後、計算機の計算能力の増大の大部分は、CPU 数を増やすことによって担われると思われる。従って、大規模な計算を行うためには並列化が必須となる。並列化とは、プログラム実行時に複数のプロセスが協調して動作することで、全体として一つの目的を達成するための作業であり、並列化プログラミングとはその実装手段である。

並列化プログラムにおいてまず重要なのは並列化効率である。並列化していないプログラムを一つの物理コアで実行した場合に比べて、同じ仕事を二つの物理コアで二つのスレッドで実行させた場合には計算時間が半分になることが望ましいが、並列化のプロセス間の同期コストなどで、実際には計算速度の向上

は 2 倍よりも小さくなることが多い。非並列実行時に T_1 だけ時間がかかったジョブを、N 個の物理コア上で n 個のスレッドで実行した際の実行時間 T_n であった時、並列化効率は

$$\alpha = \frac{T_1}{nT_n} \tag{1}$$

で定義される。$\alpha = 1$ の場合、並列化効率が 100%であると呼び、理想的に並列化できたことを意味する。一人でやったら 4 時間かかる仕事を 4 人でやったら 1 時間でおわるようなイメージであり、こういうことが可能なのは多くの場合単純作業だけで、一般の仕事ではそんなにはうまくいかないのは想像がつくであろう。

並列化ではプログラムの開発コストも非常に重要である。並列化することが目的ではなく、結果を早く得ることが目的である以上、実行時間を 5 時間短くするために一週間プログラムにかかるようでは意味がない。並列化には大きく分けて OpenMP によるループ分割と、MPI による明示的な通信によるものに大別される。OpenMP では、並列化したいループの直前にディレクティブと呼ばれる指示文を入れることで並列化を行う。ディレクティブは文法としてはコメントであり、OpenMP に対応していない処理系では無視される。従って、並列環境とそうでない環境で同じコードがつかえるというメリットがある。しかし、ディレクティブによる並列化で高い並列化効率が得られるのは単純なプログラムであることが多く、複雑なコードで高い並列化効率を得るためには明示的にプロセス間通信を記述する必要がある。そのプロセス間通信を記述する枠組みとして標準化されたのが MPI (Message Passing Interface) である。MPI はライブラリの動作を定めた仕様であり、その実装は MPICH や OpenMPI などが有名である[1]。

一般に、OpenMP よりも MPI の方がプログラムの作成コストが高いことが多い。しかし、統計力学においてはモンテカルロなど「サンプル数をとにかく稼ぎたい」場合が多い。このとき、単純に乱数の種だけ異なるプロセスを多数実行させるだけで良く、それは MPI を用いて簡単に書くことができる。こういう並列化を「単純並列 (trivial parallelization)」と呼ぶ。単純並列は別名「馬鹿パラ」と呼ばれ、文字通り馬鹿にされることが多いが、並列化効率 100%であり、もっとも効率的に計算資源を使っているため、その意義は大きい。何をするにせよ、まず馬鹿パラができないことには話にならない。本稿では、MPI の動作原理の簡単な説明と馬鹿パラのしかた、その他有用な技術について説明する。なお、動作環境は Mac OS X を想定している。

2 マルチスレッド動作の仕組み

まず、マルチスレッド実行というものを実感してみよう。コマンドラインにて、文字を表示させてみる。

```
$ echo Hello
Hello
```

これは単純に「Hello」という文字を表示させたものである。これを並列実行しよう。並列実行には mpirun というコマンドを用いる。

```
$ mpirun -np 2 echo Hello
Hello
Hello
```

「Hello」という文字列が二つ表示された。mpirun は、以下に続くコマンドを-np の後に指定された数だけプロセスを立ち上げて実行させるコマンドである。ここでは「-np 2」を指定しているため、二つのプロセスが立ち上がり、それぞれが独立に Hello という文字を表示している。「-np 4」を指定すれば 4 つのプロセスが実行され、「Hello」は 4 回表示される。これは立派な並列プログラムであるが、このままではそれ

[1] OpenMP と OpenMPI を混同しやすいので注意。OpenMPI は MPI という仕様の実装の一つであり、OpenMP とはまったく別物である。

ぞれのプロセスは同じ仕事しかできない。並列処理は、それぞれのプロセスに異なる仕事をさせなければならない。

MPI では、すべてのプロセスで同じプログラムが実行されるが、それぞれのプロセスに一意な識別番号が渡され、その番号により分岐することで異なる動作をさせることができる。このようなプログラム概念を **SPMD (Single Program Multiple Data)** と呼ぶ。

実際に並列プログラムを書いてみよう。以下のコードを mpitest.cc として保存する。

```
#include <stdio.h>
#include <mpi.h>
int
main(int argc,char **argv){
  int rank;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  printf("My rank = %d\n",rank);
  MPI_Finalize();
}
```

コンパイルには mpic++を用いる。

```
$ mpic++ mpitest.cc
```

コンパイルできたら実行してみよう。以下のような表示がでたら成功である。

```
$ mpirun -np 4 ./a.out
My rank = 0
My rank = 1
My rank = 2
My rank = 3
```

コードの内容を見てみよう。最初に mpi.h をインクルードする。MPI はライブラリの形で提供されているため、C/C++言語では、使う前にその宣言をしておく必要がある。プログラムでは、最初に MPI_Init を、最後に MPI_Finalize を呼び出している。これはオマジナイとして必ず実行するようにしよう。MPI_Init は引数として main 関数の引数を要求するので、それもそのまま渡す。なお、MPI のライブラリは、必ず頭に MPI_というプレフィックスがつく。このプログラムで最も重要なのは MPI_Comm_rank である。最初の MPI_COMM_WORLD はコミュニケータと呼ばれる識別子だが、これは後述する。次の引数で、整数型の変数のポインタを渡すと、その変数にプロセスごとに異なる値が代入される。これをプロセスの**ランク (Rank)** と呼び、プロセスの背番号のようなものである。各プロセスに通し番号をつけ、その番号を自分の背番号として、その背番号により処理を変えることにする。これが MPI による並列処理の基本的な発想である。この実行例ではランクの順番通りに表示されたが、実際には実行の度に順序が変わりうる。

なお、OpenMP はコンパイラが並列処理を記述するため、OpenMP 並列化に対応したコンパイラが必要となるが、MPI はライブラリであり、コンパイラがすることは単にリンクするだけである。したがって、正しくオプションを記述すれば、どのコンパイラでも MPI を実行できる。MPI プログラムをコンパイルするプログラムに mpic++(MacOS X) や mpCC(AIX IBM コンパイラ) などがあるが、これらは単にインクルードパスやライブラリパスを指定しているだけである。たとえば通常の mpic++では内部的に g++が呼ばれているが、ここでインテルコンパイラ icc、icpc などを使ってコンパイルしたいと思えば、

```
$ icpc mpitest.cc -I/usr/local/include -L/usr/local/lib -lmpich -lrt
```

などとパスやライブラリを正しく指定すればコンパイル、リンクができる (パスやリンクオプションは環境によって異なる)。

3 MPIによる単純並列

MPI はその名の通り、メッセージパッシング方式と呼ばれる方法を用いて並列処理を行う。メッセージパッシング方式はプロセス間通信の実装手段の一つで、各プロセスがメッセージを明示的にやりとりすることで並列処理をする。各プロセスが保持するメモリが物理的に同じ場所に存在しなくても良いため、分散メモリ型の並列処理マシンで広く使われている。多くの場合、MPI の学習はプロセス間の一対一通信、特にブロッキング通信と呼ばれるライブラリを使うことから始まるが、統計力学において馬鹿パラをする際には通信はほとんど必要ない。そこで、まずは馬鹿パラのやり方について述べる。

MPI による馬鹿パラの実装は、各プロセスに割り当てられた背番号であるランクを乱数の種とし、それぞれが独立にモンテカルロなり分子動力学法なりを実行すれば良い。まずは何も考えずに乱数の種と試行回数を受け取り、円周率を計算するコードを作ってみよう。

List 1: 円周率計算 (非並列版)

```
#include <stdio.h>
#include <stdlib.h>
//------------------------------------------------------------------------
double
myrand(void){
  return (double)rand()/(double)RAND_MAX;
}
//------------------------------------------------------------------------
double
calc_pi(int seed, int trial){
  srand(seed);
  int n = 0;
  for(int i=0;i<trial;i++){
    double x = myrand();
    double y = myrand();
    if(x*x + y*y < 1.0){
      n++;
    }
  }
  return 4.0*(double)n/(double)trial;
}
//------------------------------------------------------------------------
int
main(int argc, char **argv){
  double pi = calc_pi(1,1000000);
  printf("%f \n",pi);
}
//------------------------------------------------------------------------
```

このコードの `calc_pi` は、乱数の種 `seed` と試行回数 `trial` を受け取り、円周率を返す関数である。実行結果は以下のようになる。

```
$ ./a.out
3.142096
```

このコードを並列化するため、乱数の種としてランク番号を与えるように変更しよう。方針は、

1. `#include <mpi.h>`をつける
2. main 関数の最初と最後に `MPI_Init` と `MPI_Finalize` を書く。
3. `MPI_Comm_rank` によりランク番号を得る
4. `calc_pi` にランク番号を渡す。

とすればよい。こうして以下のようなコードとなる。

List 2: 円周率計算 (並列版その 1)

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
//------------------------------------------------------------------------
double
myrand(void){
  return (double)rand()/(double)RAND_MAX;
}
//------------------------------------------------------------------------
double
calc_pi(int seed, int trial){
  srand(seed);
  int n = 0;
  for(int i=0;i<trial;i++){
    double x = myrand();
    double y = myrand();
    if(x*x + y*y < 1.0){
      n++;
    }
  }
  return 4.0*(double)n/(double)trial;
}
//------------------------------------------------------------------------
int
main(int argc, char **argv){
  MPI_Init(&argc, &argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  double pi = calc_pi(rank,1000000);
  printf("rank=%d: pi = %f \n",rank,pi);
  MPI_Finalize();
}
//------------------------------------------------------------------------
```

これを 4 並列実行した場合の結果は以下のようになる。

```
$ mpirun -np 4 ./a.out
rank=0: pi = 3.139268
rank=3: pi = 3.142288
rank=1: pi = 3.142096
rank=2: pi = 3.139256
```

それぞれのプロセスが異なる値を出力しているのがわかるであろう。また、非並列版と同じ種を与える rank=1 のプロセスが、同じ結果を出していることも確認しておきたい。一般に並列版プログラムのデバッグは難しいため、非並列版と比較できる場合にはなるべく比較してバグの混入を防ぐ。

さて、これでモンテカルロ法の馬鹿パラによる並列化が完了した。プロセスの数だけ円周率の推定値が出力されるので、実行後に平均を取れば、プロセス数の数だけサンプル数が稼げたことになる。次に、最初からプロセスの数で平均した値を出力するようにしたい。そのためには、「全体のプロセス数がどれだけあるか」を知る必要がある。これは関数は `MPI_Comm_size` という関数で得ることができる。先ほどのプログラムの `main` 関数を以下のように書き換えよう。

List 3: 円周率計算 (並列版その 2)

```
int
main(int argc, char **argv){
  MPI_Init(&argc, &argv);
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  double pi = calc_pi(rank,1000000);
  printf("rank=%d:/%d pi = %f \n",rank,size,pi);
  MPI_Finalize();
```

```
}
```

2 並列、4 並列の実行結果は次のようになる。

```
$ mpirun -np 2 ./a.out
rank=0/2 pi = 3.139268
rank=1/2 pi = 3.142096

$ mpirun -np 4 ./a.out
rank=1/4 pi = 3.142096
rank=0/4 pi = 3.139268
rank=2/4 pi = 3.139256
rank=3/4 pi = 3.142288
```

正しくプロセス数が動的に取得できていることがわかる。次に各プロセスが計算した円周率を平均することを考える。各プロセスが計算した値は、各プロセスが保持するメモリに格納されているため、他のプロセスの値を得るためには通信が必要となる。このような時、プロセス 1,2,3 番が 0 番に値を送り、0 番が集計して出力することもできるが、MPI で用意されている全体通信を使ったほうが手軽かつ実行も速い。

具体的には、MPI_Allreduce という関数を用いて以下のようなコードを書けば良い。

List 4: 円周率計算 (並列版その 3)

```
//------------------------------------------------------------------------
int
main(int argc, char **argv){
  MPI_Init(&argc, &argv);
  int rank, procs;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  double pi = calc_pi(rank,1000000);
  printf("rank=%d/%d pi = %f \n",rank,procs,pi);
  MPI_Barrier(MPI_COMM_WORLD);
  double sum = 0;
  MPI_Allreduce(&pi, &sum, 1, MPI_DOUBLE, MPI_SUM,MPI_COMM_WORLD);
  sum = sum / (double)procs;
  if (0==rank){
    printf("average = %f\n",sum);
  }
  MPI_Finalize();
}
//------------------------------------------------------------------------
```

実行結果は以下のようになる。

```
$ mpirun -np 4 ./a.out
rank=0/4 pi = 3.139268
rank=1/4 pi = 3.142096
rank=2/4 pi = 3.139256
rank=3/4 pi = 3.142288
average = 3.140727
```

データを集めたあと、ランク 0 番のプロセスが代表して平均値を表示している (表示が混ざらないように直前でバリア同期をかけている)。MPI_Allreduce という関数は以下のような引数を取る。

```
int MPI_Allreduce(void* senddata, void* recvdata, int count, MPI_Datatype datatype,
                  MPI_Op op, MPI_Comm comm)
```

senddata は送りたいデータの先頭アドレス、recvdata に受け取りデータ領域の先頭アドレス、count 送受信するデータの数、datatype は送受信するデータの型 (この場合は double なので MPI_DOUBLE)、op は演算のタイプ (この場合は総和なので MPI_SUM)、最後にコミュニケータを渡す。通信の際には必ずコミュニケータを指定する必要があるが、凝ったことをするのでなければ、プロセス全体を意味する MPI_COMM_WORLD を指定しておけばよい。MPI_Allreduce は、それぞれのプロセスのデータの総和を、すべてのプロセスに

ばらまく。この場合は pi という変数 (これはすべてのプロセスで異なる値を持つ) の総和を sum という変数に集める。関数の実行後、すべてのプロセスに置いて sum は共通の値を持つ。このように、すべてのプロセスが関与する通信を**集団通信**と呼ぶ。また、それぞれのプロセスが持つデータに、なんらかの演算を施した結果を集めるような操作を特に**大域的リダクション操作**と呼ぶ。リダクション操作はここで例に上げた総和の他、最大値や最小値の検索、論理演算、積などの計算を行うことができる。

同様な機能を持つ関数に MPI_Reduce がある。MPI_Reduce は MPI_Allreduce と異なり、「どのプロセスにデータを集めるか」を指定する。例えば先ほどのコードであれば、MPI_Reduce を用いて以下のようにも書ける。

List 5: 円周率計算 (並列版その 3)

```
//------------------------------------------------------------------------
int
main(int argc, char **argv){
  MPI_Init(&argc, &argv);
  int rank, procs;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  double pi = calc_pi(rank,1000000);
  printf("rank=%d/%d pi = %f \n",rank,procs,pi);
  MPI_Barrier(MPI_COMM_WORLD);
  double sum = 0;
  MPI_Reduce(&pi, &sum, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
  if (0==rank){
    sum = sum / (double)procs;
    printf("average = %f\n",sum);
  }
  MPI_Finalize();
}
//------------------------------------------------------------------------
```

実行結果は全く同じとなる。ここで、受信プロセス (この場合はランク 0 番) 以外では、変数 sum の値は定義されていないことに注意したい。従って、平均処理もランク 0 番だけが行う必要がある。なお、MPI_Allreduce に限らず、MPI において値を受け取る場合はすべて変数をポインタ渡しで渡す。ここでは変数を一つしか受け渡さなかったが、多数のデータを受け渡しすることもできる。その場合は送信データとして配列へのポインタ[2]、送信サイズとしてデータの数を渡せば良い。

4 データ入出力

4.1 標準入力とブロードキャスト

並列処理においてデータの入出力は問題になりやすい。特に、シングルスレッドのプログラムにおいて標準入出力のリダイレクトを多用している場合、そのまま並列処理に書き換えられないことが多い。全てのプロセスに同じデータを渡したいなら標準入力ではなく、ファイルからパラメータを読み込むようにするのが簡単だが、リダイレクトなどで手軽にパラメータを渡したいということもあるだろう。そこで、データを標準入力から受け取るプログラムの並列化をしてみよう。

データを標準入力から受け取るプログラムとしては、たとえば C なら

List 6: 標準入力 (C 言語)

```
#include <stdio.h>
//------------------------------------------------------------------------
int
main(int argc, char **argv){
```

[2] C/C++言語では、もともと配列はポインタでアクセスするため、そのまま配列の識別子を渡せば良い (配列の場合は頭に&をつける必要はない)。Fortran ではすべて参照渡しなので識別子をそのまま渡せば良い (変数、配列の区別をする必要がない)。

```
  int value = 0;
  scanf("%d",&value);
  printf("value = %d\n",value);
}
//------------------------------------------------------------------------
```

C++なら

List 7: 標準入力 (C++言語)

```
#include <iostream>
//------------------------------------------------------------------------
int
main(int argc, char **argv){
  int value = 0;
  std::cin >> value;
  std::cout << "value = " << value << std::endl;
}
//------------------------------------------------------------------------
```

となるだろう。実行結果は以下のようになる。

```
$ ./a.out
123
value = 123
```

ただし 123 はユーザが入力した値である。このまま並列化しようとすると、プロセスの数だけ入力を用意しなければならないし、さらにどのプロセスがどの順番で読み込むかが分からないため不便である。

そこで、まずランク 0 番が代表して受け取り、全てのプロセスに配ることにする。コード例は以下の通り。

List 8: ブロードキャスト

```
#include <stdio.h>
#include <mpi.h>
//------------------------------------------------------------------------
int
main(int argc, char **argv){
  MPI_Init(&argc, &argv);
  int rank = 0;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  int value = 0;
  if(0 == rank){
    scanf("%d",&value);
  }
  MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD);
  printf("rank = %d: value = %d\n",rank, value);
  MPI_Finalize();
}
//------------------------------------------------------------------------
```

ここで使われている MPI_Bcast という関数は、以下のような引数を取る。

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
              MPI_Comm comm )
```

buffer はプロセスで共有したい変数 (もしくはデータ配列) のアドレス、count はデータの数、root は、送信元プロセス (この場合は 0 番)、最後がコミュニケータである。実行結果は以下の通り。123 はユーザによる入力である。ランク 0 番が代表して入力を受け取り、プロセス全員に正しく値が渡っていることがわかる。

```
$ mpirun -np 4 ./a.out
123
rank = 0: value = 123
```

```
rank = 1: value = 123
rank = 2: value = 123
rank = 3: value = 123
```

複数のデータを渡したい場合には、同じようなプロセスを繰り返せばよいが、パラメータが増えるたびにいちいち並列化コードを書き直すのは不便である。こういう場合は構造体を使うときれいに書ける。プログラムに乱数の種 (int 型) と温度 (double 型) をパラメータとして渡すことを考える。まず、その二つを含む構造体 parameter 型を作る。入力はその parameter 型のインスタンスに対して行い、その構造体を全プロセスにばら撒くことにする。コード例は以下の通り。

List 9: 構造体のブロードキャスト

```
#include <stdio.h>
#include <mpi.h>
//--------------------------------------------------------------------------
struct parameter{
  int seed;
  double temperature;
};
//--------------------------------------------------------------------------
int
main(int argc, char **argv){
  MPI_Init(&argc, &argv);
  int rank = 0;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  parameter param;
  if(0 == rank){
    scanf("%d",&param.seed);
    scanf("%lf",&param.temperature);
  }
  MPI_Bcast(&param, sizeof(param), MPI_BYTE, 0, MPI_COMM_WORLD);
  printf("rank = %d: seed = %d temperature = %f\n",rank, param.seed, param.
      temperature);
  MPI_Finalize();
}
//--------------------------------------------------------------------------
```

ポイントは、構造体の先頭アドレスを送受信データとして渡し、サイズを sizeof で、データ型を MPI_BYTE で指定しているところである。実行結果は以下のようになる。

```
$ mpirun -np 4 ./a.out
123
0.7
rank = 0: seed = 123 temperature = 0.700000
rank = 1: seed = 123 temperature = 0.700000
rank = 2: seed = 123 temperature = 0.700000
rank = 3: seed = 123 temperature = 0.700000
```

123 と 0.7 はユーザからの入力である。実際には input.cfg などのファイルを作っておき、リダイレクトで渡すのが良いだろう。

```
$ cat input.cfg
123
0.7

$ mpirun -np 4 ./a.out < input.cfg
rank = 0: seed = 123 temperature = 0.700000
rank = 1: seed = 123 temperature = 0.700000
rank = 2: seed = 123 temperature = 0.700000
rank = 3: seed = 123 temperature = 0.700000
```

C 言語版とほとんど同じだが、C++版についても念のためコード例も挙げておく。

List 10: 構造体のブロードキャスト (C++版)

```
#include <iostream>
#include <mpi.h>
//--------------------------------------------------------------------------
struct parameter{
  int seed;
  double temperature;
};
//--------------------------------------------------------------------------
int
main(int argc, char **argv){
  MPI_Init(&argc, &argv);
  int rank = 0;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  parameter param;
  if(0 == rank){
    std::cin >> param.seed;
    std::cin >> param.temperature;
  }
  MPI_Bcast(&param, sizeof(param), MPI_BYTE, 0, MPI_COMM_WORLD);
  std::cout << "rank = " << rank;
  std::cout << " seed = " << param.seed;
  std::cout << " temperature = " << param.temperature << std::endl;
  MPI_Finalize();
}
//--------------------------------------------------------------------------
```

4.2 ファイルの保存 (バリア同期型)

MPI は分散メモリ型のプログラムモデルであり、各プロセスが独立にメモリを持っている。このメモリをファイルに保存する際、もっとも簡単なのは、conf.000.dat,conf.001.dat··· のようにファイル名+プロセス番号のようにして、各プロセス独立にデータを吐いておき、後でまとめることである。しかし、何かのデータの時間発展を保存しようとする時、この方法ではステップごとにプロセス数だけのファイルが出力されて鬱陶しい。そこで、プロセスの保持するデータをまとめて保存する方法について述べる。

データが配列の形で保持されているとしよう。この配列をそのままファイルに保存するプログラムを考える。非並列版は、たとえば次のようなプログラムになるであろう。

List 11: ファイルの保存 (C 版)

```
#include <stdio.h>
int
main(void){
  const int SIZE = 10;
  int array[SIZE];
  for(int i=0;i<SIZE;i++){
    array[i] = i;
  }
  FILE *fp = fopen("data.dat","w");
  for(int i=0;i<SIZE;i++){
    fprintf(fp,"%d\n",array[i]);
  }
  fclose(fp);
}
```

List 12: ファイルの保存 (C++版)

```
#include <iostream>
#include <fstream>
int
main(void){
```

```
  const int SIZE = 10;
  int array[SIZE];
  for(int i=0;i<SIZE;i++){
    array[i] = i;
  }
  std::ofstream ofs("data.dat");
  for(int i=0;i<SIZE;i++){
    ofs << array[i] << std::endl;
  }
}
```

実行結果は次のようになる。

```
$ ./a.out

$ cat data.dat
0
1
2
3
4
5
6
7
8
9
```

このプログラムを並列化しよう。各プロセスは自分の処理結果を配列 array に出力し、それをプロセスの順番ごとに一つのファイル data.dat に保存する。簡単のため、各プロセスは配列の中身を自分のランク番号で埋めることにしよう。各プロセスのデータを一つのファイルにまとめるには、まず代表 (ランク 0 番) がファイルを作成し、その後プロセス数の数だけループをまわし、自分の番が来たらファイルをひらいてデータを追加すればよい。この方針をプログラムしたものが以下の例である (見易さのため、SIZE を 2 にしてある)。

List 13: 並列化ファイルの保存の (C 版)

```
#include <stdio.h>
#include <mpi.h>
int
main(int argc, char **argv){
  MPI_Init(&argc, &argv);
  int rank, procs;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  const int SIZE = 2;
  int array[SIZE];
  for(int i=0;i<SIZE;i++){
    array[i] = rank;
  }
  FILE *fp;
  if(0 == rank){
    fp = fopen("data.dat","w");
    fclose(fp);
  }
  for(int j=0;j<procs;j++){
    MPI_Barrier(MPI_COMM_WORLD);
    if(j != rank)continue;
    fp = fopen("data.dat","a");
    for(int i=0;i<SIZE;i++){
      fprintf(fp,"%d\n",array[i]);
    }
    fclose(fp);
  }
  MPI_Finalize();
```

```
}
```

実行結果は以下のようになる。

```
$ mpirun -np 4 ./a.out

$ cat data.dat
0
0
1
1
2
2
3
3
```

ここで、最初にランク 0 番がファイルを作成し、すぐに閉じている。これは、プログラムの実行前に既に data.dat が存在した場合、中身をクリアするためである。この作業をしておかないとプログラムを再実行した際に、前のジョブの結果に追記してしまう。この種の間違いは意外とありがちなので覚えておきたい。また、他のプロセスがファイルを開いている間に別のプロセスがファイルを触りにいかないようにループの最初にバリア同期をしている。なお、C++版は以下の通り。

List 14: 並列化ファイルの保存 (C++版)

```
#include <iostream>
#include <fstream>
#include <mpi.h>
int
main(int argc, char** argv){
  MPI_Init(&argc, &argv);
  int rank, procs;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  const int SIZE = 2;
  int array[SIZE];
  for(int i=0;i<SIZE;i++){
    array[i] = rank;
  }
  if(0==rank){
    std::ofstream ofs("data.dat");
    ofs.close();
  }
  for(int j=0;j<procs;j++){
    MPI_Barrier(MPI_COMM_WORLD);
    if(j!=rank)continue;
    std::ofstream ofs("data.dat",std::ios::app);
    for(int i=0;i<SIZE;i++){
      ofs << array[i] << std::endl;
    }
    ofs.close();
  }
  MPI_Finalize();
}
```

4.3 ファイルの保存 (Allreduce 型)

プロセスの数だけループをまわし、自分の番になったときにファイルを追記書き込みする方法は汎用的だが、プロセスの数だけバリア同期する必要があるため、プロセス数が増えたら遅くなる。もしメモリが許すのであれば、一度代表プロセスにデータを集めて、一気にファイルを書いてしまう方が速い。また、

リスタートなどのため、データをバイナリで吐いておきたいことも多いだろう。ここでは、各プロセスが持つデータをルートプロセスにまとめた上で、バイナリで一気にファイルに出力する方法について述べる。

粒子の座標や運動量など、多くの場合バイナリで出力したいのは double 型であろう。そこで、まず double 型の配列をバイナリ保存するコードを考える。

List 15: バイナリファイルの保存 (C 版)

```
#include <stdio.h>

int
main(int argc, char **argv){
  const int SIZE = 10;
  double data[SIZE];
  for(int i=0;i<SIZE;i++){
    data[i] = (double)i;
  }
  FILE *fp = fopen("data.dat","wb");
  fwrite(data,sizeof(double),SIZE,fp);
  fclose(fp);
}
```

実行すると data.dat というバイナリファイルができるので、hexdump コマンド[3]で内容を出力しよう。

```
$ ./a.out

$ hexdump -v -e '"%f\n"' data.dat
0.000000
1.000000
2.000000
3.000000
4.000000
5.000000
6.000000
7.000000
8.000000
9.000000
```

この並列版を考える。各プロセスがデータを配列 data に保持しているので、それを一度ランク 0 番にまとめ、0 番が責任をもって保存する。コード例は以下の通り。

List 16: データの Gather(C 版)

```
#include <stdio.h>
#include <mpi.h>

int
main(int argc, char **argv){
  MPI_Init(&argc, &argv);
  int rank,procs;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  const int SIZE = 2;
  double data[SIZE];
  for(int i=0;i<SIZE;i++){
    data[i] = (double)rank;
  }
  double *buf;
  if(0==rank){
    buf = new double[SIZE*procs];
  }
  MPI_Gather(data, SIZE, MPI_DOUBLE, buf, SIZE, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  if(0==rank){
```

[3]hexdump は所定のフォーマットでバイナリファイルを出力する。フォーマットは c 言語の printf 文と同じなので使い方はすぐにわかるであろう。知っておくとデバッグ時に便利なコマンドの一つ。なお、hexdump は同じ出力内容が続くと省略して*(アスタリスク) を出力するので、これを防ぐために-v オプションをつける。

```
    FILE *fp = fopen("data.dat","wb");
    fwrite(buf,sizeof(double),SIZE*procs,fp);
    fclose(fp);
    delete [] buf;
  }
  MPI_Finalize();
}
```

まず、ランク 0 番が全てのデータを保存できるだけのバッファ用にメモリを確保している。その後、MPI_Gather 関数を用いて data の中身を受信用のバッファbuf に集める。MPI_Gather は以下のような引数を持つ。

```
int MPI_Gather(void *sendbuffer, int sendcount, MPI_Datatype sendtype,
               void *recvbuffer, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm )
```

引数の意味はわかるであろう。通常は sendcount と recvcount、sendtype と recvtype は同じはずである。受信用バッファは、root で指定されたプロセスだけが用意しておけば良く、他のプロセスでメモリが確保されていなくても問題は起きない。実行結果は以下の通り。

```
$ mpirun -np 4 ./a.out

$ hexdump -v -e '"%f\n"' data.dat
0.000000
0.000000
1.000000
1.000000
2.000000
2.000000
3.000000
3.000000
```

それぞれのプロセスのデータが二個ずつ、ただしく順序通りに保存されていることがわかる。

念のため C++版のコード例も挙げておく。

List 17: バイナリファイルの保存 (C++版)

```
#include <iostream>
#include <fstream>

int
main(int argc, char **argv){
  const int SIZE = 10;
  double data[SIZE];
  for(int i=0;i<SIZE;i++){
    data[i] = (double)i;
  }
  std::ofstream ofs("data.dat", std::ios::binary);
  ofs.write((char*)data,sizeof(double)*SIZE);
}
```

List 18: データの Gather(C++版)

```
#include <iostream>
#include <fstream>
#include <mpi.h>

int
main(int argc, char **argv){
  MPI_Init(&argc, &argv);
  int rank,procs;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  const int SIZE = 2;
  double data[SIZE];
```

```
  for(int i=0;i<SIZE;i++){
    data[i] = (double)rank;
  }
  double *buf;
  if(0==rank){
    buf = new double[SIZE*procs];
  }
  MPI_Gather(data, SIZE, MPI_DOUBLE, buf, SIZE, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  if(0==rank){
    std::ofstream ofs("data.dat",std::ios::binary);
    ofs.write((char*)buf,sizeof(double)*SIZE*procs);
    delete [] buf;
  }
  MPI_Finalize();
}
```

5 一対一通信

5.1 ブロッキング通信

本気で並列化を行うためには、各プロセス間で一対一通信を行う必要がある。データの送信は MPI_Send、受信は MPI_Recv 関数で行うことができる。以下は、0 番から 1 番に整数を一つ送る例である。

List 19: 一対一通信

```
#include <stdio.h>
#include <mpi.h>

int
main(int argc, char **argv){
  MPI_Init(&argc, &argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  int send_value = rank;
  int recv_value = -1;
  const int TAG = 0;
  MPI_Status st;
  if(0==rank){
    MPI_Send(&send_value, 1, MPI_INT, 1, TAG, MPI_COMM_WORLD);
  }else if(1==rank){
    MPI_Recv(&recv_value, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD,&st);
  }
  printf("rank = %d: recv_value = %d\n",rank, recv_value);
  MPI_Finalize();
}
```

実行結果は以下の通り。

```
$ mpirun -np 2 ./a.out
rank = 0: recv_value = -1
rank = 1: recv_value = 0
```

MPI_Send、MPI_Recv はそれぞれ以下のような引数を取る。

```
int MPI_Send(void *sendbuffer, int sendcount, MPI_Datatype sendtype,
             int sendtag, int dest, MPI_Comm comm )
int MPI_Recv(void *recvbuffer, int recvcount, MPI_Datatype recvtype,
             int recvtag, int src, MPI_Status *st, MPI_Comm comm )
```

sendbuffer や recvbuffer の意味は良いだろう。それぞれ送信および受信データの先頭アドレスである。sendcount と recvcount、sendtype と recvtype は送信側と受信側で一致している必要がある。tag と

いうのは、データ送信の際につけるタグのことで、同じプロセスに複数のデータを送りたいとき、それぞれのデータを区別するのに使う。送信側と同じタグの値を指定しなければ受信できないが、複雑なことをするのでなければ、全て 0 にしておけば問題ない。受信側では MPI_Status 構造体を引数に渡す必要がある。ここには送信プロセスの情報が入るが、いまは気にしなくて良い。MPI_Send や MPI_Recv は、送受信にかかわるプロセス以外は無視される。たとえば先ほどのコードを 4 並列で実行した場合、

```
$ mpirun -np 4 ./a.out
rank = 0: recv_value = -1
rank = 1: recv_value = 0
rank = 2: recv_value = -1
rank = 3: recv_value = -1
```

と、通信にかかわらない 2 番と 3 番でなにも起きていないことがわかるだろう。

さて、0 番と 1 番の値を交換しようとして、以下のようなコードを書くとデッドロックすることがある[4]。

List 20: デッドロック例

```
#include <stdio.h>
#include <mpi.h>

int
main(int argc, char **argv){
  MPI_Init(&argc, &argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  int send_value = rank;
  int recv_value = -1;
  const int TAG = 0;
  MPI_Status st;
  if(0==rank){
    MPI_Send(&send_value, 1, MPI_INT, 1, TAG, MPI_COMM_WORLD);
    MPI_Recv(&recv_value, 1, MPI_INT, 1, TAG, MPI_COMM_WORLD,&st);
  }else if(1==rank){
    MPI_Send(&send_value, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD);
    MPI_Recv(&recv_value, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD,&st);
  }
  printf("rank = %d: recv_value = %d\n",rank, recv_value);
  MPI_Finalize();
}
```

これは、MPI_Send や MPI_Recv が**ブロッキング通信**だからである。ブロッキング通信とは、送信なら送信が完了するまでその関数の次に処理が進まないタイプの通信であり、先ほどの例ではプロセス 0 番と 1 番が両方送信を試み、お互いが受信できないため通信がいつまでたっても完了せずにデッドロックする。これを防ぐためには、送信と受信の順番をどちらかで入れ替えればよい。

List 21: デッドロック解消例

```
#include <stdio.h>
#include <mpi.h>

int
main(int argc, char **argv){
  MPI_Init(&argc, &argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  int send_value = rank;
  int recv_value = -1;
  const int TAG = 0;
  MPI_Status st;
  if(0==rank){
    MPI_Send(&send_value, 1, MPI_INT, 1, TAG, MPI_COMM_WORLD);
```

[4]MPI の処理系によってはデッドロックしないことがあるようなので、逆に注意が必要である。普段からデッドロックしないように心がけて組むことが望ましい。

```
    MPI_Recv(&recv_value, 1, MPI_INT, 1, TAG, MPI_COMM_WORLD,&st);
  }else if(1==rank){
    MPI_Recv(&recv_value, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD,&st);
    MPI_Send(&send_value, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD);
  }
  printf("rank = %d: recv_value = %d\n",rank, recv_value);
  MPI_Finalize();
}
```

これは無事に実行され、以下のような結果が出力される。

```
$ mpirun -np 2 ./a.out
rank = 0: recv_value = 1
rank = 1: recv_value = 0
```

さて、ほとんどの並列プログラムにおいて、送信のみ、受信のみが必要ということはなく、一般にはプロセス間で送受信が両方必要になる。この時、送受信を同時に行う関数として MPI_Sendrecv が用意されている。MPI_Sendrecv は以下のような引数を取る。

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int src, int recvtag, MPI_Comm comm, MPI_Status *status )
```

それぞれの引数の意味は MPI_Send、MPI_Recv と同じである。MPI_Sendrecv を用いると、先ほどのコードは以下のようにかける。

List 22: Sendrecv の例

```
#include <stdio.h>
#include <mpi.h>

int
main(int argc, char **argv){
  MPI_Init(&argc, &argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  int send_value = rank;
  int recv_value = -1;
  const int TAG = 0;
  MPI_Status st;
  if(0==rank){
    MPI_Sendrecv(&send_value, 1, MPI_INT, 1, TAG,
                 &recv_value, 1, MPI_INT, 1, TAG, MPI_COMM_WORLD,&st);
  }else if(1==rank){
    MPI_Sendrecv(&send_value, 1, MPI_INT, 0, TAG,
                 &recv_value, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD,&st);
  }
  printf("rank = %d: recv_value = %d\n",rank, recv_value);
  MPI_Finalize();
}
```

なお、MPI_Sendrecv の送信先と受信先は別のプロセスでも良い。たとえば、複数のプロセスがサイクリックに通信するようなコードを以下のように書くことができる。

List 23: Sendrecv の例

```
#include <stdio.h>
#include <mpi.h>

int
main(int argc, char **argv){
  MPI_Init(&argc, &argv);
  int rank, procs;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

```
  MPI_Comm_size(MPI_COMM_WORLD,&procs);
  int send_value = rank;
  int recv_value = -1;
  int dest_rank = (rank+1)%procs;
  int src_rank = (rank-1+procs)%procs;
  const int TAG = 0;

  MPI_Status st;
  MPI_Sendrecv(&send_value, 1, MPI_INT, dest_rank, TAG,
               &recv_value, 1, MPI_INT, src_rank , TAG, MPI_COMM_WORLD,&st);
  printf("rank = %d: recv_value = %d\n",rank, recv_value);
  MPI_Finalize();
}
```

ソースに if 文がないことにも注意したい。プロセス番号により if 文で分岐するようなコードはバグが入りやすいため、なるべく if 文を使わずに組むようにしたい。実行結果は以下の通り。

```
$ mpirun -np 4 ./a.out
rank = 0: recv_value = 3
rank = 1: recv_value = 0
rank = 2: recv_value = 1
rank = 3: recv_value = 2
```

たとえば 2 番のプロセスが、3 番にデータを送信するのと 1 番からデータを送信するのを一つの関数で行っている。一般に、MPI_Sendrecv はもっともチューニングされていることが多く、これを使えばデッドロックの心配もなく、また MPI_Send、MPI_Recv を分けて使うよりも速いことが多い。なるべく MPI_Sendrecv を使うようにしたい。

5.2 ノンブロッキング通信

MPI_Send や MPI_Recv といったブロッキング通信の関数に対し、MPI_Isend や MPI_Irecv がノンブロッキング通信を行う。これは相手が受信完了しなくても関数の処理が戻ってくるため、通信時間の隠蔽に使われる。ただし、大量にノンブロッキング通信を行うと、「MPI_REQUEST_MAX を使い切った」というエラーが出ることがある。これはノンブロッキング通信が特殊なリソースを用いるため、大量にノンブロッキング通信を行うとそのリソースを使い切ることにより発生するエラーである。これはデバッグ時に小さい系では発生しないが、プロダクトラン用に大きなジョブを投げたときに発生する、たちの悪いエラーである。対策としてはリソースを大きく確保する、リソースを使い切らないようにこまめに MPI_Request_free を呼ぶなどがある。ノンブロッキング通信を行う際には記憶の片隅に入れておくとよい。また、一般論だがノンブロッキング通信はブロッキング通信よりもレイテンシが大きい。したがって、上手に通信を隠蔽できないとかえって実行速度が遅くなることがあるので注意したい。

6 最後に

以上、駆け足で MPI 並列化について述べた。並列化は簡単なことは意外に簡単にできるが、凝ったことをやろうとするととたんに大変になる。特に並列化プログラムのデバッグは非常に難しいので、シリアル版で十分にデバッグしておくことが大事である。