第6章　MPIプログラミングの初歩

いよいよ本章で，MPI(Message Passing Interface) プログラミングに触れることになる。MPI は多くの関数からなる規格であるが，前述の通り，本書では数値計算に必要なもののみ抜粋して紹介する。ここでは 1CPU/1 ノードの構成の PC Cluster における，MPICH[5] を用いた場合のプログラム及び実行例を見ていくことにする。

6.1　MPIの動作原理

まず，次のプログラムを実行してみよう。先頭が MPI_から始まる関数が，MPI で規定されている関数である。

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <math.h>
 4 :
 5 : #include "mpi.h"
 6 :
 7 : int main(int argc, char *argv[])
 8 : {
 9 :     MPI_Init(&argc, &argv);
10 :
11 :     printf("Hellow, MPI!\n");
12 :
13 :     MPI_Finalize();
14 :
15 :     return EXIT_SUCCESS;
16 : }
17 :
```

これをコンパイルする Makefile は次のようになる。このケースでは MPICH のライブラリを呼び出してリンクしている。

```
 1 : CC=mpicc
 2 : DEL=rm
 3 :
```

```
 4 : #LIB=-lmpich -lm
 5 : LIB=-lmpi -lm
 6 :
 7 : mpi1: mpi1.c
 8 :     $(CC) -o mpi1 mpi1.c $(LIB)
 9 :
10 : clean:
11 :     -$(DEL) mpi1
```

make すると mpi1 という実行ファイルが生成される。これを 1 ノードで実行するには

```
% mpirun -np 1 ./mpi1
```

とする。この場合は

```
Hellow, MPI!
%
```

という表示がなされる。次にこれを 2 ノード, 4 ノード , 8 ノードで並列実行してみよう。

```
% mpirun -np 2 ./mpi1
Hellow, MPI!
Hellow, MPI!
% mpirun -np 4 ./mpi1
Hellow, MPI!
Hellow, MPI!
Hellow, MPI!
Hellow, MPI!
% mpirun -np 8 ./mpi1
Hellow, MPI!
Hellow, MPI!
Hellow, MPI!
Hellow, MPI!
Hellow, MPI!
Hellow, MPI!
Hellow, MPI!
Hellow, MPI!
%
```

これからわかるように，MPI では一本のプログラムから複数のプロセス (ランク) で並列動作する元になる実行プログラムを生成し，それを mpirun コマンドによって複数プロセスで実行することになる (SPMD アプローチ)。prinf 関数のように，標準出力は全てランク 0 のプロセスに集められて表示することになる。

6.2 プロセス (ランク) 毎の動作

続いて，どのランクがどのノードで動作しているのかを表示するプログラムを実行してみよう。Makefile は前節のものを，ソース，実行ファイル名のみ変更して使えばよい。

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <math.h>
 4 :
 5 : #include "mpi.h"
 6 :
 7 : int main(int argc, char *argv[])
 8 : {
 9 :     int  namelen, num_procs, myrank;
10 :     char processor_name[MPI_MAX_PROCESSOR_NAME];
11 :
12 :     MPI_Init(&argc, &argv);
13 :
14 :     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
15 :     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
16 :     MPI_Get_processor_name(processor_name, &namelen);
17 :
18 :     printf("Hellow, MPI! at Process %d of %d on %s\n", myrank, nu
   m_procs, processor_name);
19 :
20 :     MPI_Finalize();
21 :
22 :     return EXIT_SUCCESS;
23 : }
24 :
```

これを 8 ノードで実行すると次のようになる。この結果は当然，使用する PC Cluster の各ノードのホスト名に依存する。

```
% mpirun -np 8 ./mpi2
Hellow, MPI!
```

```
Process 0 of 8 on cs-southpole
Hellow, MPI!
Process 4 of 8 on cs-room443-b04
Hellow, MPI!
Process 2 of 8 on cs-room443-b02
Hellow, MPI!
Process 6 of 8 on cs-room443-s03
Hellow, MPI!
Process 3 of 8 on cs-room443-b03
Hellow, MPI!
Process 7 of 8 on cs-room443-s04
Hellow, MPI!
Process 1 of 8 on cs-room443-b01
Hellow, MPI!
Process 5 of 8 on cs-room443-s02
%
```

必ずしも，ランク番号の順に表示されているわけではないことが分かる。

では，ランクごとに異なる計算をするプログラムを作ってみよう。二つの IEEE754 倍精度変数の加減乗除をランク順に計算し，5 ランク以上では何もしないというものである。

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <math.h>
 4 :
 5 : #include "mpi.h"
 6 :
 7 : int main(int argc, char *argv[])
 8 : {
 9 :     int num_procs, myrank;
10 :     double a, b;
11 :
12 :     MPI_Init(&argc, &argv);
13 :
14 :     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
15 :     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
16 :
17 :     a = 1.0;
18 :     b = 3.14159;
19 :
20 :     switch(myrank % 4)
```

```
21 :     {
22 :         case 0: printf("%2d: %e + %e = %e\n", myrank, a, b, a + b
   ); break;
23 :         case 1: printf("%2d: %e - %e = %e\n", myrank, a, b, a - b
   ); break;
24 :         case 2: printf("%2d: %e * %e = %e\n", myrank, a, b, a * b
   ); break;
25 :         case 3: printf("%2d: %e / %e = %e\n", myrank, a, b, a / b
   ); break;
26 :         default: printf("%2d: No Computation\n", myrank); break;
27 :     }
28 :
29 :     MPI_Finalize();
30 :
31 :     return EXIT_SUCCESS;
32 : }
33 :
```

これを 5 ノードを使って実行すると

```
% mpirun -np 5 ./mpi3
1.000000e+00 + 3.141590e+00 = 4.141590e+00
1.000000e+00 * 3.141590e+00 = 3.141590e+00
1.000000e+00 - 3.141590e+00 = -2.141590e+00
1.000000e+00 / 3.141590e+00 = 3.183102e-01
No Computation
%
```

となる。

では最後に，ランクごと異なる値を用いて加算を行うプログラムを見ることにしよう。

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <math.h>
 4 :
 5 : #include "mpi.h"
 6 :
 7 : int main(int argc, char *argv[])
 8 : {
 9 :     int num_procs, myrank;
10 :     double a[128], b[128];
11 :
12 :     MPI_Init(&argc, &argv);
```

```
13 :
14 :     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
15 :     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
16 :
17 :     a[myrank] = num_procs - myrank;
18 :     b[myrank] = myrank;
19 :
20 :     printf("%e + %e = %e\n", a[myrank], b[myrank], a[myrank] + b[
    myrank]);
21 :
22 :     MPI_Finalize();
23 :
24 :     return EXIT_SUCCESS;
25 : }
26 :
```

これを8ノード使って実行すると

```
% mpirun -np 8 ./mpi4
8.000000e+00 + 0.000000e+00 = 8.000000e+00
6.000000e+00 + 2.000000e+00 = 8.000000e+00
4.000000e+00 + 4.000000e+00 = 8.000000e+00
2.000000e+00 + 6.000000e+00 = 8.000000e+00
7.000000e+00 + 1.000000e+00 = 8.000000e+00
5.000000e+00 + 3.000000e+00 = 8.000000e+00
3.000000e+00 + 5.000000e+00 = 8.000000e+00
1.000000e+00 + 7.000000e+00 = 8.000000e+00
%
```

となる。

6.3　プロセス間での1対1通信

複雑なプログラムを並列化しようとすると，それぞれのランクで計算した結果をやり取りする場面が出てくる。その基本となるのが1対1同期通信関数MPI_Send/MPI_Recvである。

この二つ関数は次のような引数を指定して使用する。

MPI_Send関数

```
MPI_Send(
```

(void *) 送信データ変数へのポインタ,

int データ数,

MPI_Datatype 変数のデータ型,

int 送信先ランク番号,

int 送信データに付加するタグ,

MPI_Comm コミュニケータ)

MPI_Recv 関数

MPI_Recv(

(void *) 受信データ変数へのポインタ,

int データ数,

MPI_Datatype 変数のデータ型,

int 受信元ランク番号,

int 受信データに付加されているタグ,

MPI_Comm コミュニケータ

MPI_Status * ステータス)

この関数を使用した例を次に示す。これはランク 0(PE0) からランク 1(PE1) へ IEEE754 倍精度のデータを一個分送信している例である。

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <math.h>
 4 :
 5 : #include "mpi.h"
 6 :
 7 : int main(int argc, char *argv[])
 8 : {
 9 :     int num_procs, myrank;
10 :     double a, b;
11 :     int tag = 0;
12 :     MPI_Status status;
```

```
13 :
14 :     MPI_Init(&argc, &argv);
15 :
16 :     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
17 :     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
18 :
19 :     a = 0;
20 :     b = 0;
21 :     if(myrank == 0)
22 :     {
23 :         a = 1.0;
24 :         MPI_Send((void *)&a, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORL
   D);
25 :     }
26 :     else if(myrank == 1)
27 :     {
28 :         MPI_Recv((void *)&b, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORL
   D, &status);
29 :     }
30 :
31 :     printf("Process %d: a = %e, b = %e\n", myrank, a, b);
32 :
33 :     MPI_Finalize();
34 :
35 :     return EXIT_SUCCESS;
36 : }
37 :
```

これをコンパイルして，実行ファイル mpi-sr を得て，2 ノード使って実行すると

```
% mpirun -np 2 ./mpi-sr
Process 0: a = 1.000000e+00, b = 0.000000e+00
Process 1: a = 0.000000e+00, b = 1.000000e+00
%
```

となる。この実行推移を図 6.1 に示す。

6.4　多倍長浮動小数点数を用いた MPI プログラム

mpi-sr.c を多倍長浮動小数点数で実行してみよう。このために MPIBNCpack をリンクして使用する。この場合の Makefile は

```
1 : CC=mpicc
```

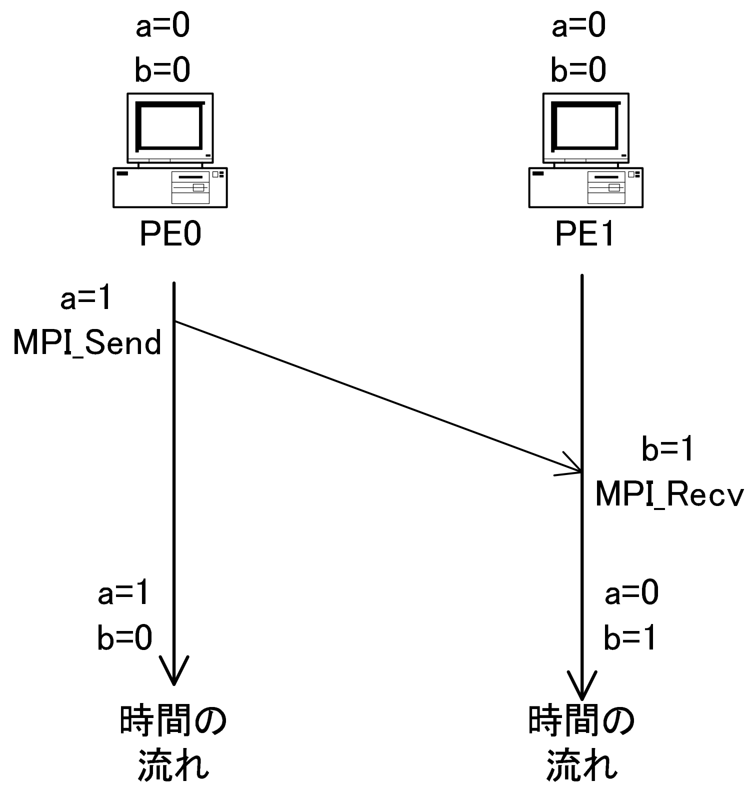

図 6.1: mpi-sr.c の送受信処理

```
 2 : DEL=rm
 3 :
 4 : INC=-I/usr/local/include
 5 : LIBDIR=-L/usr/local/lib
 6 :
 7 : #LIB=-lmpibnc -lmpich -lbnc -lmpfr -lgmp -lm
 8 : LIB=$(LIBDIR) -lmpibnc -lmpi -lbnc -lmpfr -lgmp -lm
 9 :
10 : mpi-sr-gmp: mpi-sr-gmp.c
11 :     $(CC) $(INC) -o mpi-sr-gmp mpi-sr-gmp.c $(LIB)
12 :
13 : clean:
14 :     -$(DEL) mpi-sr-gmp
```

となる。
ソースファイルは次のようになる。

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <math.h>
 4 :
 5 : #include "mpi.h"
 6 :
 7 : #define USE_GMP
 8 : #define USE_MPFR
 9 : #include "gmp.h"
10 : #include "mpfr.h"
11 : #include "mpi_bnc.h"
12 :
13 : main(int argc, char *argv[])
14 : {
15 :     int num_procs, myrank;
16 :     mpf_t a, b;
17 :     void *buf;
18 :     int tag = 0;
19 :     MPI_Status status;
20 :
21 :     MPI_Init(&argc, &argv);
22 :
23 :     _mpi_set_bnc_default_prec_decimal(50, MPI_COMM_WORLD);
24 :     commit_mpf(&(MPI_MPF), ceil(50/log10(2.0)), MPI_COMM_WORLD);
25 :
26 :     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
27 :     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
28 :
29 :     mpf_init_set_ui(a, 0);
```

```
30 :     mpf_init_set_ui(b, 0);
31 :     if(myrank == 0)
32 :     {
33 :         mpf_set_ui(a, 1);
34 :         buf = allocbuf_mpf(mpf_get_prec(a), 1);
35 :         pack_mpf(a, 1, buf);
36 :         MPI_Send(buf, 1, MPI_MPF, 1, tag, MPI_COMM_WORLD);
37 :     }
38 :     else if(myrank == 1)
39 :     {
40 :         buf = allocbuf_mpf(mpf_get_prec(b), 1);
41 :         MPI_Recv(buf, 1, MPI_MPF, 0, tag, MPI_COMM_WORLD, &status
   );
42 :         unpack_mpf(buf, b, 1);
43 :     }
44 :
45 :     printf("Process %d: a = ", myrank);
46 :     mpf_out_str(stdout, 10, 0, a);
47 :     printf(", b = ");
48 :     mpf_out_str(stdout, 10, 0, b);
49 :     printf("\n");
50 :
51 :     mpf_clear(a);
52 :     mpf_clear(b);
53 :     free_mpf(&(MPI_MPF));
54 :
55 :     MPI_Finalize();
56 :
57 :     return EXIT_SUCCESS;
58 : }
59 :
```

見て分かるように，多倍長変数を送受信するには幾つかの特別な処理を行う必要がある。実際に行っていることを図にすると図6.2のようになる。

これを実行すると次のように10進50桁の多倍長データが正しく送受信されていることが分かる。

```
% mpirun -np 2 ./mpi-sr-gmp
--------------------------------------------------------------------------------
BNC Default Precision    : 167 bits(50.3 decimal digits)
BNC Default Rounding Mode: Round to Nearest
--------------------------------------------------------------------------------
Process 0: a = 1.00000000000000000000000000000000000000000000000000, b = 0
Process 1: a = 0, b = 1.00000000000000000000000000000000000000000000000000
```

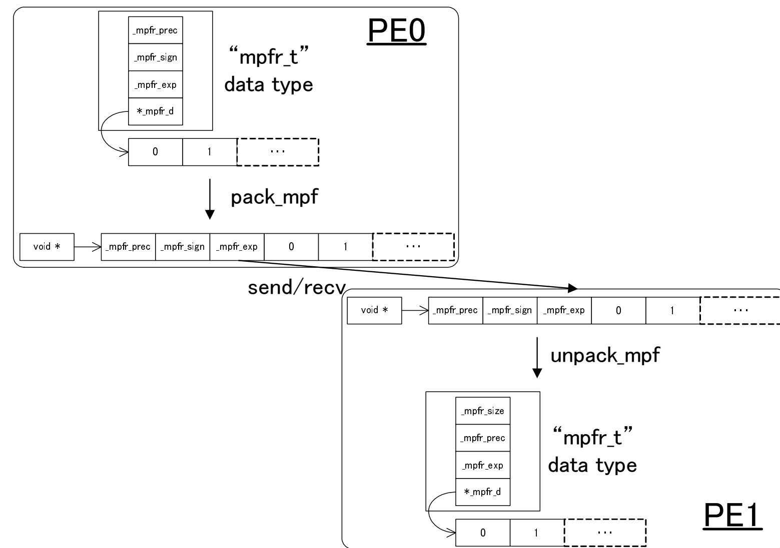

図 6.2: 多倍長 FP 数の送受信

演習問題

1. mpi4.c を改良し，整数の乱数 (rand() 関数を使用) をランクごとに生成し，それぞれの平方根を IEEE754 倍精度で計算して表示するプログラムを作れ。

2. mpi-sr.c を改良し，受信した値を b へ代入し，それを 1 だけ増やして a に代入して次のランクへ送信するようにせよ (図 6.3 参照)。

   これを実行すると次のような結果を得る。

```
% mpirun -np 8 ./mpi-sr1
Process 0: a = 1.000000e+00, b = 0.000000e+00
Process 2: a = 3.000000e+00, b = 2.000000e+00
Process 3: a = 4.000000e+00, b = 3.000000e+00
Process 4: a = 5.000000e+00, b = 4.000000e+00
Process 7: a = 0.000000e+00, b = 7.000000e+00
Process 6: a = 7.000000e+00, b = 6.000000e+00
Process 1: a = 2.000000e+00, b = 1.000000e+00
Process 5: a = 6.000000e+00, b = 5.000000e+00
%
```

3. mpi-sr-gmp.c を改良して，上記実行例と同じ動作を行うようにせよ。

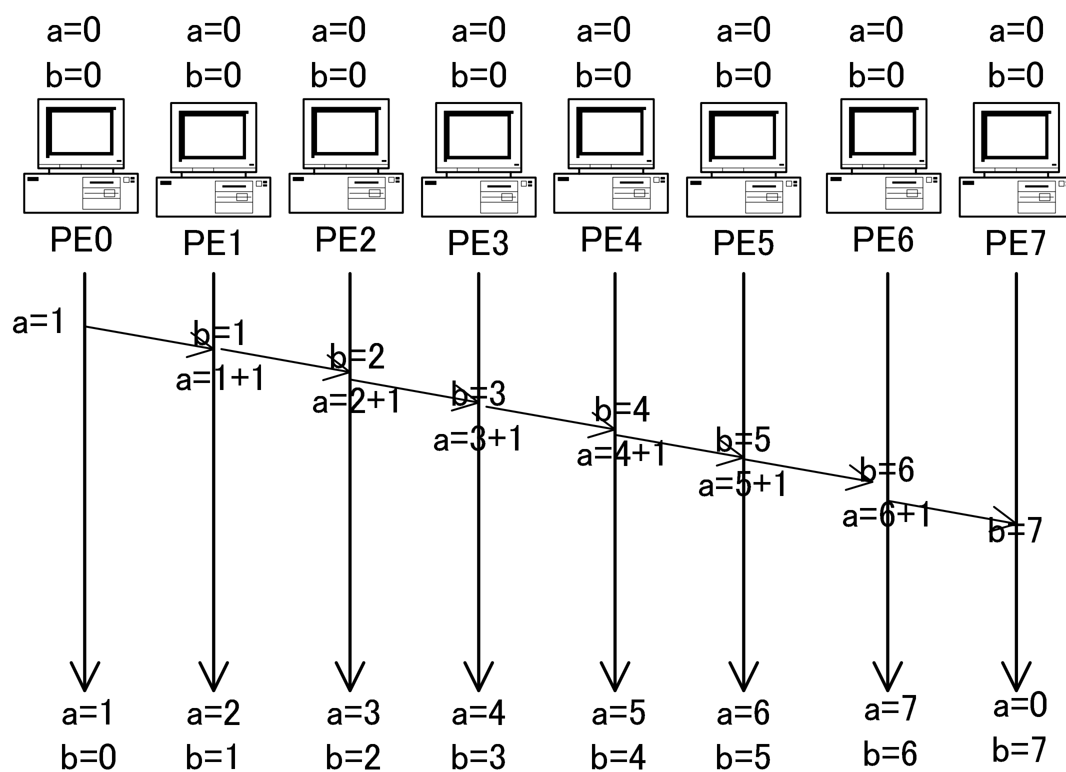

図 6.3: mpi-sr1.c の送受信処理