工学実験実習 V

—MPI—

学籍番号：16426
5 年 電子情報工学科 24 番
福澤 大地

提出日：2020 年 10 月 19 日

1　目的

MPI (Message Passing Interface) を用いて、複数の CPU 間で通信を行いながら、並列計算を行うプログラムを作成する。その上で、並列計算を行うと通常に比べどれほどの高速化を図れるか、並列計算に適したアルゴリズムとはどのようなものなのかなどを検証する。

2　実験環境

プログラムの開発、実行を行った環境を表 1 に示す。表 1 と同様の環境のコンピュータ 44 台が同一ネットワーク内に接続されており、公開鍵認証方式でこれらのコンピュータと SSH 通信を行える環境で実験を行った。

表 1　実験環境

CPU	Intel Core i5-6600 @ 3.3GHz
メモリ	8GB
OS	Ubuntu 14.04 LTS
システム	64bit
コンパイラ	GCC 4.8.4
MPI ライブラリ	Open MPI 1.10.2

3　MPI と Open MPI について

MPI とは、並列計算を行うために標準化された規格である。これを用いることにより、1 個の CPU で行っていた計算を複数の CPU で分散して行えるようになる。

Open MPI [1] は、MPI に準拠したライブラリの 1 つであり、Unix 上で利用できる。MPI のライブラリは他にも MPICH [2] などがあるが、本実験では Open MPI を使用する。

4　実行方法

プログラムのコンパルには `mpicc` コマンド、実行には `mpirun` コマンドを使用する。`mpicc` コマンドは `gcc` コマンドと同様の使い方ができ、`-Wall` オプションなどを利用することもできる。`mpirun` コマンドは、`-machinefile` オプションで使用するコンピュータの名前と CPU の数が記述されたファイル名を、`-np` オプションで使用する CPU の数を指定することで、コンパイルしたファイルを実行することができる。

例えば、“com001” ～ “com004” という名前のコンピュータの CPU を 1 つずつ使用する場合は、次のように記述されたテキストファイルを適当なファイル名で保存する。ここでは、ホームディレクトリに “mymachines” というファイル名で保存することとする。

```
com001 cpu=1
com002 cpu=1
com003 cpu=1
com004 cpu=1
```

そして、“program.c” というファイル名のプログラムをコンパイルし、先ほど指定した 4 個の CPU 実行する場合には次のようなコマンドを入力する。

```
$ mpicc program.c
$ mpirun -machinefile ~/mymachines -np 4 ./a.out
```

なお、今回の環境では次のようにエイリアスを設定することにより、-machinefile オプションを省略し実行できるようにしてある。

```
alias mpirun='mpirun -machinefile ~/mymachines'
```

5 MPI のプログラム

MPI を用いてプログラムを作成する際は、通常のプログラムとは違い、今実行している CPU の数はいくつなのか、自分はどの CPU なのかなどの情報を取得する必要がある。そのため、MPI のプログラムではリスト 1 のように、前処理を行うプログラムを記述する必要がある。なお、プログラムの終了時には、必ず MPI_Finalize 関数を呼び出さなければならない。

リスト 1　MPI のプログラム

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char **argv) {
5      int nsize;
6      int myrank;
7      int my_name_len;
8      char my_name[MPI_MAX_PROCESSOR_NAME];
9
10     MPI_Init(&argc, &argv);
11     MPI_Comm_size(MPI_COMM_WORLD, &nsize);
12     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
13     MPI_Get_processor_name(my_name, &my_name_len);
14
15     printf("nsize = %d\n", nsize);
16     printf("myrank = %d\n", myrank);
17     printf("my_name_len = %d\n", my_name_len);
18     printf("my_name = %s\n", my_name);
19
20     MPI_Finalize();
21
22     return 0;
23 }
```

リスト 1 のプログラムを 4 個の CPU で実行した結果を、リスト 2 に示す。リスト 2 を見ると、nsize に実行している CPU の数、myrank に自身の番号、my_name に自身のコンピュータ名が入っていることが分かる。実行結果が myrank の順番で表示されていないのは、プログラムが各 CPU 上で同時に実行されているためである。

リスト 2　MPI のプログラムの実行結果

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

```
myrank = 1
my_name_len = 6
my_name = ayu002
nsize = 4
myrank = 0
my_name_len = 6
my_name = ayu001
nsize = 4
myrank = 3
my_name_len = 6
my_name = ayu004
nsize = 4
myrank = 2
my_name_len = 6
my_name = ayu003
```

6 課題 1

6.1 課題内容

コマンドライン引数から数値 X を受け取り、$1 \sim X$ までの和を N 台の CPU で求めるプログラムを作成する。X と N は任意の自然数とする。

6.2 プログラムリスト

課題 1 のプログラムを、リスト 3 に示す。

リスト 3　課題 1 のプログラム

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4
5  typedef long long int ll;
6
7  int main(int argc, char **argv) {
8      int nsize;
9      int myrank;
10     int my_name_len;
11     char my_name[MPI_MAX_PROCESSOR_NAME];
12
13     ll i;
14     ll num;
15     ll sum = 0;
16     ll ans, max, min;
17
18     double start_t;
19     double finish_t;
20
21     MPI_Init(&argc, &argv);
22     MPI_Comm_size(MPI_COMM_WORLD, &nsize);
```

```
23      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
24      MPI_Get_processor_name(my_name, &my_name_len);
25
26      if (argc <= 1) {
27          if (myrank == 0)
28              printf("引数が与えられていません。\n");
29
30          MPI_Finalize();
31          return 0;
32      }
33
34      // 引数を整数型に変換
35      num = atoll(argv[1]);
36
37      if (num <= 0) {
38          if (myrank == 0)
39              printf("引数の値が小さすぎます。\n");
40
41          MPI_Finalize();
42          return 0;
43      }
44
45      if (num > 400000000011) {
46          if (myrank == 0)
47              printf("引数の値が大きすぎます。\n");
48
49          MPI_Finalize();
50          return 0;
51      }
52
53      // 処理開始時間を取得
54      start_t = MPI_Wtime();
55
56      // 演算
57      for (i = 1 + myrank; i <= num; i += nsize) {
58          sum += i;
59      }
60
61      // 集計
62      MPI_Reduce(&sum, &ans, 1, MPI_INTEGER8, MPI_SUM, 0, MPI_COMM_WORLD);
63      MPI_Reduce(&sum, &max, 1, MPI_INTEGER8, MPI_MAX, 0, MPI_COMM_WORLD);
64      MPI_Reduce(&sum, &min, 1, MPI_INTEGER8, MPI_MIN, 0, MPI_COMM_WORLD);
65
66      // 処理終了時間を取得
67      finish_t = MPI_Wtime();
68
69      // 結果表示
70      if (myrank == 0) {
71          printf("answer: %lld\n", ans);
72          printf("max: %lld\n", max);
73          printf("min: %lld\n", min);
74          printf("time: %.10f\n", finish_t - start_t);
75      }
76
77      MPI_Finalize();
78
```

```
79      return 0;
80  }
```

6.3 プログラムの説明

6.3.1 入力値チェック

26 ～ 47 行目では、与えられた引数が正しいものであるのかチェックを行っている。

本プログラムでは計算を long long int 型で行っている。long long int 型は 64 ビットであるため、表せる値の最大値は、$2^{63}-1$ である。最終的な計算結果がこの範囲に収まっている必要があるので、入力として許容できる最大値を n とすると、式 (1) のようにして求められる。

$$\begin{aligned} \sum_{k=1}^{n} k &= 2^{63}-1 \\ \frac{1}{2}n(n+1) &= 2^{63}-1 \\ \frac{1}{2}n^2+\frac{1}{2}n-2^{63}+1 &= 0 \\ n &\simeq \pm 4.3\times 10^9 \end{aligned} \tag{1}$$

式 (1) より、n が 4×10^9 以内であれば確実にオーバーフローが起こることはないため、これより大きい値が入力された際はエラーとしてプログラムを終了している。また、コマンドライン引数が与えられていなかった場合や、入力された値が 0 以下であった場合も同様にエラーとしてプログラムを終了している。

6.3.2 演算

演算は 51 ～ 53 行目で行っており、myrank+1 から始め、nsize 間隔で数字を足している。例えば、4 個の CPU で実行した場合には、各 CPU が担当する数字は次のようになる。このようにすることで、各 CPU で担当する数字の個数と合計のばらつきを少なくしている。

CPU 0　1, 5, 9, 13, 17, ...
CPU 1　2, 6, 10, 14, 18, ...
CPU 2　3, 7, 11, 15, 19, ...
CPU 3　4, 8, 12, 16, 20, ...

6.3.3 集計

各 CPU で行った計算結果の集計は、55 行目の MPI_Reduce 関数で行っている。このような記述を行うことで、全ての CPU の sum の合計を、ans に代入することができる。

第 4 パラメータには演算の種類を指定することができ、56 行目の MPI_MAX では最大値、57 行目の MPI_MIN では最小値を取得することができる。

6.3.4 処理時間の計測

MPI には、過去のある地点からの経過時間を取得する MPI_Wtime 関数が用意されている。この関数を処理の開始時と終了時に呼び出し、その差分を取ることで、処理に掛かった時間を計測することができる。本プログラムでは、49 行目で開始時間、59 行目で終了時間を取得し、65 行目でその差分を表示している。

6.4 実行結果

6.4.1 1 ～ 54321 の和

引数に 54321 を入力し、4 個の CPU で実行した場合の結果をリスト 4 に、8 個の CPU で実行した場合の結果をリスト 5 に示す。

リスト 4　引数に 54321 を入力した場合の実行結果

```
$ mpirun -np 4 ./a.out 54321
answer: 1475412681
max: 368873541
min: 368832800
time: 0.0005230904
```

リスト 5　8 個の CPU で実行した場合の実行結果

```
$ mpirun -np 8 ./a.out 54321
answer: 1475412681
max: 184450351
min: 184402820
time: 0.0002680087
```

6.4.2 入力値チェック

引数を与えなかった場合の結果をリスト 6, 0 以下の数を入力した場合の結果をリスト 7, 4×10^9 を上回る数を入力した場合の結果をリスト 8 に示す。

リスト 6　引数を与えなかった場合の実行結果

```
$ mpirun -np 4 ./a.out
引数が与えられていません。
```

リスト 7　0 以下の数を入力した場合の実行結果

```
$ mpirun -np 4 ./a.out -1
引数の値が小さすぎます。
```

リスト 8　最大値以上の数を入力した場合の実行結果

```
$ mpirun -np 4 ./a.out 5000000000
引数の値が大きすぎます。
```

6.5 考察

1 ～ 54321 の和は $\sum_{k=1}^{54321} k = 1475412681$ である。これはリスト 4 の計算結果と一致するため、正しく計算が行われている。また、各 CPU での計算結果の最大値と最小値を見比べると、大きな差は見受けられない。よって、狙い通り CPU による合計値のばらつきを少なくすることができた。

リスト 5 を見ると、4 個の CPU で実行したリスト 4 に比べ、処理時間が約半分になっている。このことから、正常に並列計算が行えていると言える。

さらに、リスト 6–8 を見ると、不正なコマンドライン引数に対して適切なエラーメッセージが表示されている。よって、正しく入力値のチェックが行えていることが分かる。

7　課題 2

7.1　課題内容

課題 1 で作成したプログラムを用いて、並列計算の効果を測定する。CPU の数 N を変えながら処理時間を測定し、その結果をグラフにして考察する。

7.2　実行結果

引数に 4×10^9 を与えた場合について、CPU の数 N を $1 \sim 30$ 個に増やしながら処理時間を測定した。全ての CPU が作業が行われていない状態で測定を行い、それぞれ 10 回の平均値を取った。

それぞれの CPU の個数についての処理時間についてまとめたものを表 2 に、グラフにプロットしたものを図 1 に示す。

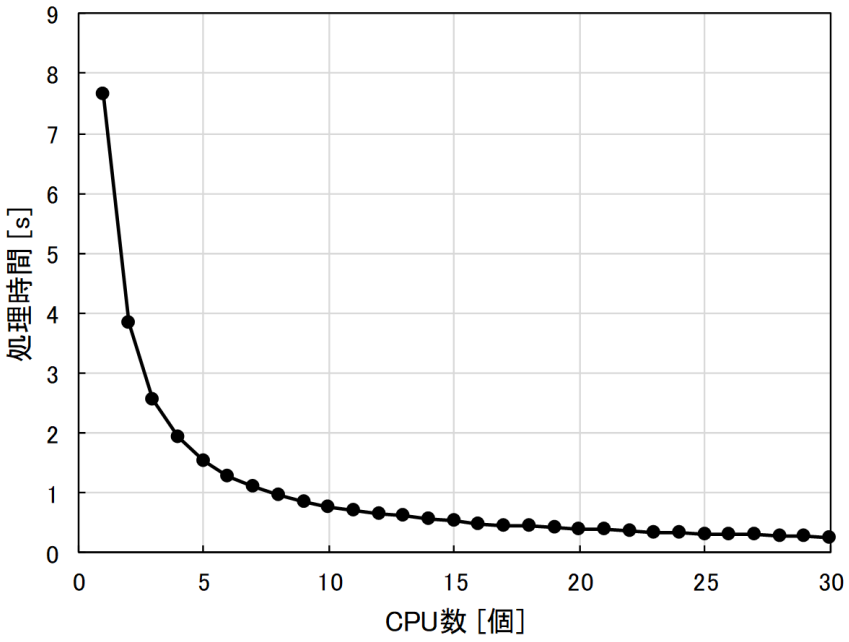

図 1　CPU の個数と処理時間の関係

7.3　考察

図 1 を見ると、反比例のグラフとなっている。また、表 2 の CPU1 個の処理時間に対する短縮率より、おおよそ CPU の数の分だけ処理時間の短縮がされていることが分かる。このことから、課題 1 のような、ほとんど通信を行わない単純な並列計算を行うと、投入した CPU の数の分だけ効率化が行えると言える。

表 2　CPU の個数と処理時間

CPU 数 [個]	処理時間 [ms]	CPU1 個の処理時間に対する短縮率
1	7663	1.00
2	3842	1.99
3	2572	2.98
4	1929	3.97
5	1544	4.96
6	1289	5.94
7	1113	6.89
8	971	7.89
9	862	8.89
10	780	9.83
11	711	10.78
12	652	11.76
13	613	12.49
14	576	13.29
15	536	14.29
16	492	15.58
17	466	16.46
18	440	17.42
19	415	18.48
20	397	19.29
21	391	19.61
22	370	20.71
23	346	22.17
24	328	23.36
25	319	24.05
26	309	24.83
27	300	25.51
28	296	25.91
29	282	27.19
30	266	28.77

8　課題 3

8.1　課題内容

N 個の CPU でモンテカルロシミュレーションを並列処理するプログラムを作成する。乱数の種は CPU ごとに異なるようにする。

8.2 プログラムリスト

課題 3 のプログラムを、リスト 9 に示す。

リスト 9　課題 3 のプログラム

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <math.h>
5  #include <mpi.h>
6
7  typedef long long int ll;
8  typedef long double ld;
9
10 int main(int argc, char **argv) {
11     int nsize;
12     int myrank;
13     int my_name_len;
14     char my_name[MPI_MAX_PROCESSOR_NAME];
15
16     unsigned int seed;
17     ll i;
18     ll n;
19     ll cnt = 0;
20     ll sum = 0;
21
22     double start_t;
23     double finish_t;
24
25     MPI_Init(&argc, &argv);
26     MPI_Comm_size(MPI_COMM_WORLD, &nsize);
27     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
28     MPI_Get_processor_name(my_name, &my_name_len);
29
30     if (argc <= 1) {
31         if (myrank == 0)
32             printf("引数が与えられていません。\n");
33
34         MPI_Finalize();
35         return 0;
36     }
37
38     // 引数を整数型に変換
39     n = atoll(argv[1]);
40
41     if (n <= 0) {
42         if (myrank == 0)
43             printf("引数の値が小さすぎます。\n");
44
45         MPI_Finalize();
46         return 0;
47     }
48
49     if (n % nsize != 0) {
50         if (myrank == 0)
```

```
51              printf("引数の値はCPUの個数の倍数を入力してください。\n");
52
53          MPI_Finalize();
54          return 0;
55      }
56
57      // 処理開始時間を取得
58      start_t = MPI_Wtime();
59
60      // シード値決定
61      srandom(time(NULL));
62      for (i = 0; i < myrank; i++)
63          random();
64
65      seed = random();
66      srandom(seed);
67
68      printf("CPU %02d seed: %u\n", myrank, seed);
69
70      // シミュレーション
71      for (i = 0; i < n / nsize; i++) {
72          ld x = (random() / ((ld)RAND_MAX + 1) + myrank) / nsize;
73          ld y = random() / ((ld)RAND_MAX + 1);
74
75          if (x * x + y * y < 1)
76              cnt++;
77      }
78
79      // 計算結果の統合
80      MPI_Reduce(&cnt, &sum, 1, MPI_INTEGER8, MPI_SUM, 0, MPI_COMM_WORLD);
81
82      printf("CPU %02d count: %lld\n", myrank, cnt);
83
84      if (myrank == 0) {
85          // 円周率を算出
86          ld ans = (ld)4.0 * sum / n;
87
88          // 処理終了時間を取得
89          finish_t = MPI_Wtime();
90
91          printf("answer: %.20Lf\n", ans);
92          printf("error: %.20Lf\n", ans - M_PI);
93          printf("time: %.10f s\n", finish_t - start_t);
94      }
95
96      MPI_Finalize();
97
98      return 0;
99  }
```

8.3 プログラムの説明

8.3.1 入力値チェック

30 ～ 55 行目では、与えられた引数が正しいものであるのかチェックを行っている。

本プログラムでは、円を分割したものを各 CPU で分担してシミュレーションを行うため、各 CPU でプロットする点は等しい必要がある。そこで、コマンドライン引数から受け取った値が CPU の個数の倍数でない場合は、エラーとしてプログラムを終了している。また、コマンドライン引数が与えられていなかった場合や、入力された値が 0 以下であった場合も同様にエラーとしてプログラムを終了している。

8.3.2 シード値の決定

ランダムな点をプロットするために、各 CPU でシード値を決定する必要がある。通常のプログラムであれば、現在時刻をシード値とするのが一般的であるが、並列計算のプログラムでその手法を取ると、各 CPU でシード値が同じとなる可能性がある。同じシード値となると各 CPU で全く同じ乱数列が生成されることとなってしまう。

そこで本プログラムでは、61 ～ 66 行目のように各 CPU について、時刻をシード値とし `myrank` の回数だけ乱数を生成したものを新たなシード値として採用することでこの問題を解決している。例えば、CPU0 と CPU1 が同じシード値となってしまったとする。この場合、CPU0 では生成された 1 つ目の乱数を新しいシード値として採用し、CPU1 では 2 つ目の乱数を新しいシード値として採用する。こうすることで、シード値として使う時刻が同じ値になってしまった場合にも、各 CPU で違うシード値を使用することができる。

8.3.3 シミュレーション

図 2 のような半径 r の円 1/4 の上にランダムな点を n 点プロットし、そのうち x 点が円の範囲内に入った場合、n と x の比は正方形と 1/4 の円の面積比となるため、(2) のような等式が成り立つ。これを変形すると、(3) のようになり、円周率 π の値が算出できる。

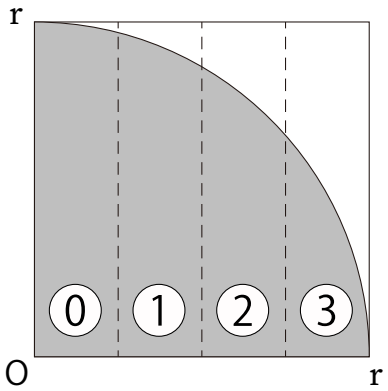

図 2　半径 r の円 1/4 と各 CPU の担当する部分

$$n : x = r^2 : \frac{\pi r^2}{4} \tag{2}$$

$$\frac{x}{n} = \frac{\pi}{4}$$

$$\pi = \frac{4x}{n} \tag{3}$$

本プログラムでは、1/4 の円を CPU の個数だけ縦に分割し、それぞれの CPU に担当させた。例えば、4 個の CPU で実行した場合は、図 2 のように分担することとなる。

これらの処理を 71 ～ 77 行目で行い、80 行目で各 CPU で数え上げた数の集計、86 行目で計算結果の算出を行っている。

8.4　実行結果

8.4.1　円周率の算出

コマンドライン引数に 2×10^8 を与え、4 個の CPU で実行した場合の結果をリスト 10 に示す。

リスト 10　2×10^8 の点でのシミュレーション結果

```
$ mpirun -np 4 ./a.out 200000000
CPU 03 seed: 1424065441
CPU 00 seed: 1730146295
CPU 01 seed: 1044912555
CPU 02 seed: 1401709856
CPU 01 count: 46185718
CPU 02 count: 38758121
CPU 03 count: 22661399
CPU 00 count: 49473813
answer: 3.14158101999999999992
error: -0.00001163358979311608
time: 1.8063 s
```

8.4.2　並列化の効果の検証

引数に 2×10^8 を与えた場合について、CPU の数 N を 1~30 個に増やしながら処理時間を測定した。全ての CPU が作業が行われていない状態で測定を行い、それぞれ 10 回の平均値を取った。

それぞれの CPU の個数についての処理時間についてまとめたものを表 3 に、グラフにプロットしたものを図 3 に示す。

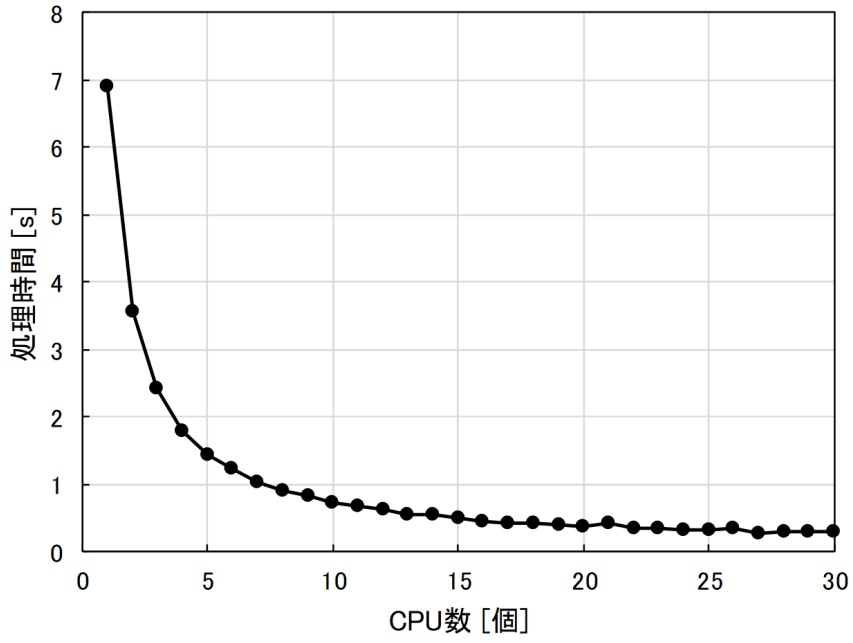

図 3　CPU の個数と処理時間の関係

表 3　CPU の個数と処理時間

CPU 数 [個]	処理時間 [ms]	CPU1 個の処理時間に対する短縮率
1	6895	1.00
2	3573	1.93
3	2422	2.85
4	1807	3.82
5	1438	4.79
6	1241	5.56
7	1027	6.72
8	903	7.63
9	837	8.24
10	727	9.48
11	680	10.15
12	632	10.91
13	562	12.26
14	555	12.42
15	504	13.67
16	460	14.98
17	436	15.81
18	430	16.02
19	394	17.49
20	381	18.12
21	432	15.95
22	340	20.27
23	348	19.82
24	338	20.43
25	323	21.36
26	359	19.20
27	280	24.64
28	296	23.27
29	309	22.34
30	305	22.60

8.5　考察

リスト 10 を見ると、各 CPU で違うシード値が採用されており、円周率が算出されているため、正しくモンテカルロ法によるシミュレーションが行われていると言える。

また、図 3 を見ると、反比例のグラフであるように見える。しかし、表 3 を見ると、CPU を 30 個使用した場合の時間の短縮率は 22.60 と、投入した分の効果が発揮されていない。このことから、実際に並列計算を行う場合は、投入した CPU の分だけ効率化が行えるとは限らないことが分かる。

9　課題 4

9.1　課題内容

以下の処理を実行するプログラムを作成する。

1. 整数配列 a[] = {3, 1, 4, 1, 5, 9} を CPU 0 で定義する。
2. CPU1 ～ 9 のそれぞれで適当な乱数 R を 1 個ずつ発生させる。
3. a[] を 9 台の CPU に MPI_Send で送信し、受信側では a[] のそれぞれの要素に手順 2 で発生させた R を加えた配列 b[] を作る。
4. 9 台の CPU からそれぞれが持っている R と b を CPU 0 に送り返す。
5. CPU 0 で CPU 番号とともに送り返されてきた R と b を表示する。

9.2　プログラムリスト

課題 4 のプログラムを、リスト 11 に示す。

リスト 11　課題 4 のプログラム

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <math.h>
5  #include <mpi.h>
6
7  #define N 6
8
9  int main(int argc, char **argv) {
10     int nsize;
11     int myrank;
12     int my_name_len;
13     char my_name[MPI_MAX_PROCESSOR_NAME];
14
15     unsigned int seed;
16     int i, j;
17
18     double start_t;
19     double finish_t;
20
21     MPI_Init(&argc, &argv);
22     MPI_Comm_size(MPI_COMM_WORLD, &nsize);
23     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
24     MPI_Get_processor_name(my_name, &my_name_len);
25
26     // 処理開始時間を取得
27     start_t = MPI_Wtime();
28
29     // シード値決定
30     srandom(time(NULL));
31     for (i = 0; i < myrank; i++)
32         random();
33
```

```
34      seed = random();
35      srandom(seed);
36
37      if (myrank == 0) {
38          int a[N] = {3, 1, 4, 1, 5, 9};
39
40          // aを送信
41          for (i = 1; i < nsize; i++)
42              MPI_Send(a, N, MPI_INTEGER, i, 100 + i, MPI_COMM_WORLD);
43
44          for (i = 1; i < nsize; i++) {
45              int R;
46              int b[N + 1];
47              MPI_Status status;
48
49              // Rとbを受信
50              MPI_Recv(&R, 1, MPI_INTEGER, i, 200 + i, MPI_COMM_WORLD, &status);
51              MPI_Recv(b, N + 1, MPI_INTEGER, i, 300 + i, MPI_COMM_WORLD, &status);
52
53              // 表示
54              printf("CPU %02d\n", i);
55              printf("R: %d\n", R);
56              printf("b:");
57
58              for (j = 0; j < N + 1; j++)
59                  printf(" %d", b[j]);
60              printf("\n\n");
61          }
62
63          // 処理時間を表示
64          finish_t = MPI_Wtime();
65          printf("time: %.10f s\n", finish_t - start_t);
66      } else {
67          int a[N];
68          int R;
69          int b[N + 1];
70          MPI_Status status;
71
72          // aを受信
73          MPI_Recv(a, N, MPI_INTEGER, 0, 100 + myrank, MPI_COMM_WORLD, &status);
74
75          // aにRを追加したものをbに格納
76          for (i = 0; i < N; i++)
77              b[i] = a[i];
78
79          R = random() % 100;
80          b[N] = R;
81
82          // bを送信
83          MPI_Send(&R, 1, MPI_INTEGER, 0, 200 + myrank, MPI_COMM_WORLD);
84          MPI_Send(b, N + 1, MPI_INTEGER, 0, 300 + myrank, MPI_COMM_WORLD);
85      }
86
87      MPI_Finalize();
88
89      return 0;
```

```
90 }
```

9.3 プログラムの説明

課題 3 までは、各 CPU で行った計算結果に対して総和を取ったり、最大値や最小値を取得する MPI_Reduce 関数を使用してきた。MPI には、これとは別に単純に値を送受信する MPI_Send, MPI_Recv 関数が用意されている。単一の値だけでなく、配列の送受信も行うことができ、これによりさらに自由度の高い通信が行えるようになる。

9.4 実行結果

課題 4 の実行結果をリスト 12 に示す。

リスト 12　課題 4 の実行結果

```
$ mpirun -np 10 ./a.out
CPU 01
R: 75
b: 3 1 4 1 5 9 75

CPU 02
R: 12
b: 3 1 4 1 5 9 12

CPU 03
R: 44
b: 3 1 4 1 5 9 44

CPU 04
R: 46
b: 3 1 4 1 5 9 46

CPU 05
R: 13
b: 3 1 4 1 5 9 13

CPU 06
R: 35
b: 3 1 4 1 5 9 35

CPU 07
R: 28
b: 3 1 4 1 5 9 28

CPU 08
R: 38
```

```
b: 3 1 4 1 5 9 38

CPU 09
R: 91
b: 3 1 4 1 5 9 91

time: 0.0016391277 s
```

9.5 考察

リスト 12 を見ると、各 CPU について、$3, 1, 4, 1, 5, 9$ に生成した R を加えた配列が CPU0 に送信され、表示されている。よって、課題内容のプログラムが正しく作成できた。

10 課題 5

10.1 課題内容

シュテルマーの公式 (4) を用いて円周率 π を小数点以下 1,000 桁以上並列計算するプログラムを作成する。

$$\pi = 24\tan^{-1}\left(\frac{1}{8}\right) + 8\tan^{-1}\left(\frac{1}{57}\right) + 4\tan^{-1}\left(\frac{1}{239}\right) \tag{4}$$

10.2 プログラムリスト

課題 5 のプログラムを、リスト 13 に示す。なお、多倍長整数の計算には 3 年次で作成したライブラリを使用しているが、そのプログラムリストは省略する。本プログラムで使っている多倍長整数ライブラリの関数とその説明をまとめたものを表 4 に示す。

リスト 13　課題 5 のプログラム

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4  #include "mulproc.h"
5
6  // 精度(桁数)
7  #define PREC 1000
8
9  int main(int argc, char **argv) {
10     int nsize;
11     int myrank;
12     int my_name_len;
13     char my_name[MPI_MAX_PROCESSOR_NAME];
14
15     nsize = 1;
16     myrank = 0;
17
18     int i, j;
19     int n;
20     double start_t;
21     double finish_t;
```

```
22      struct NUMBER tmp;
23
24      int a[] = {24, 8, 4};
25      int b[] = {8, 57, 239};
26
27      MPI_Init(&argc, &argv);
28      MPI_Comm_size(MPI_COMM_WORLD, &nsize);
29      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
30      MPI_Get_processor_name(my_name, &my_name_len);
31
32      if (argc <= 1) {
33          if (myrank == 0)
34              printf("引数が与えられていません。\n");
35
36          MPI_Finalize();
37          return 0;
38      }
39
40      // 引数を整数型に変換
41      n = atoi(argv[1]);
42
43      if (n <= 0) {
44          if (myrank == 0)
45              printf("引数の値が小さすぎます。\n");
46
47          MPI_Finalize();
48          return 0;
49      }
50
51      // 処理開始時間を取得
52      start_t = MPI_Wtime();
53
54      // 精度
55      struct NUMBER prec;
56      setInt(&prec, 1);
57
58      for (i = 0; i < PREC; i++) {
59          mulBy10(&prec, &tmp);
60          copyNumber(&tmp, &prec);
61      }
62
63      // 合計
64      struct NUMBER sum;
65      clearByZero(&sum);
66
67      for (i = 0; i < sizeof(a) / sizeof(a[0]); i++) {
68          struct NUMBER base;
69          struct NUMBER div;
70          struct NUMBER upd;
71          struct NUMBER bunshi;
72
73          setInt(&base, b[i] * b[i]);
74          setInt(&div, b[i]);
75          setInt(&upd, 1);
76
77          for (j = 0; j < myrank; j++) {
```

```
78              multiple(&div, &base, &tmp);
79              copyNumber(&tmp, &div);
80          }
81
82          for (j = 0; j < nsize; j++) {
83              multiple(&upd, &base, &tmp);
84              copyNumber(&tmp, &upd);
85          }
86
87          setInt(&bunshi, a[i]);
88          multiple(&bunshi, &prec, &tmp);
89          copyNumber(&tmp, &bunshi);
90
91          for (j = myrank; j < n; j += nsize) {
92              struct NUMBER bunbo;
93              struct NUMBER calc;
94
95              // 分母の計算
96              setInt(&bunbo, 2 * j + 1);
97              multiple(&bunbo, &div, &tmp);
98              copyNumber(&tmp, &bunbo);
99
100             // 分数の計算
101             divide(&bunshi, &bunbo, &calc, &tmp);
102
103             // 加減算
104             if (j % 2 == 0)
105                 add(&sum, &calc, &tmp);
106             else
107                 sub(&sum, &calc, &tmp);
108
109             copyNumber(&tmp, &sum);
110
111             // 割る数を更新
112             multiple(&div, &upd, &tmp);
113             copyNumber(&tmp, &div);
114         }
115     }
116
117     if (myrank == 0) {
118         MPI_Status status;
119
120         int recv_n[KETA];
121         int recv_sign;
122
123         struct NUMBER ans;
124         struct NUMBER recv;
125
126         copyNumber(&sum, &ans);
127
128         for (i = 1; i < nsize; i++) {
129             MPI_Recv(recv_n, KETA, MPI_INTEGER, i, 100 + i, MPI_COMM_WORLD, &status);
130             MPI_Recv(&recv_sign, 1, MPI_INTEGER, i, 200 + i, MPI_COMM_WORLD, &status);
131
132             for (j = 0; j < KETA; j++)
133                 recv.n[j] = recv_n[j];
```

```
134             recv.sign = recv_sign;
135
136             add(&ans, &recv, &tmp);
137             copyNumber(&tmp, &ans);
138         }
139
140         // 処理終了時間を取得
141         finish_t = MPI_Wtime();
142
143         // 表示
144         dispNumber(&ans);
145         printf("time: %.4f s\n", finish_t - start_t);
146     } else {
147         MPI_Send(sum.n, KETA, MPI_INTEGER, 0, 100 + myrank, MPI_COMM_WORLD);
148         MPI_Send(&sum.sign, 1, MPI_INTEGER, 0, 200 + myrank, MPI_COMM_WORLD);
149     }
150
151     MPI_Finalize();
152
153     return 0;
154 }
```

表 4　多倍長整数ライブラリの関数の説明

関数名	説明
copyNumber	値をコピーする
clearByZero	値を 0 に初期化する
setInt	多倍長整数変数に int 型変数の値を設定する
mulBy10	値を 10 倍する
add	加算を行う
multiple	乗算を行う
divide	除算を行う
dispNumber	多倍長整数変数の値を表示する

10.3　プログラムの説明

(4) を計算するには、$\tan^{-1} x$ の値が計算できれば良い。$\tan^{-1} x$ をテイラー展開すると (5) のようになる。

$$\tan^{-1} x = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1} \quad \text{for} \quad |x| < 1 \tag{5}$$

(5) の各項を各 CPU に分担させ、計算することで並列計算を行っている。

10.4　実行結果

課題 5 の実行結果をリスト 14 に示す。

リスト 14　課題 5 の実行結果

```
31415926535897932384626433832795028841971693993751058209749445923078164062862089
```

```
98628034825342117067982148086513282306647093844609550582231725359408128481117450
28410270193852110555964462294895493038196442881097566593344612847564823378678316
52712019091456485669234603486104543266482133936072602491412737245870066063155881
74881520920962829254091715364367892590360011330530548820466521384146951941511609
43305727036575959195309218611738193261179310511854807446237996274956735188575272
48912279381830119491298336733624406566430860213949463952247371907021798609437027
70539217176293176752384674818467669405132000568127145263560827785771342757789609
17363717872146844090122495343014654958537105079227968925892354201995611212902196
08640344181598136297747713099605187072113499999983729780499510597317328160963185
95024459455346908302642522308253344685035261931188171010003137838752886587533208
38142061717766914730359825349042875546873115956286388235378759375195778185778053
21712268066130019278766111959092164201 98
```

次に、CPU の数 N を 1~30 個に増やしながら処理時間を測定した。全ての CPU が作業が行われていない状態で測定を行い、それぞれ 5 回の平均値を取った。

それぞれの CPU の個数についての処理時間についてまとめたものを表 5 に、グラフにプロットしたものを図 4 に示す。

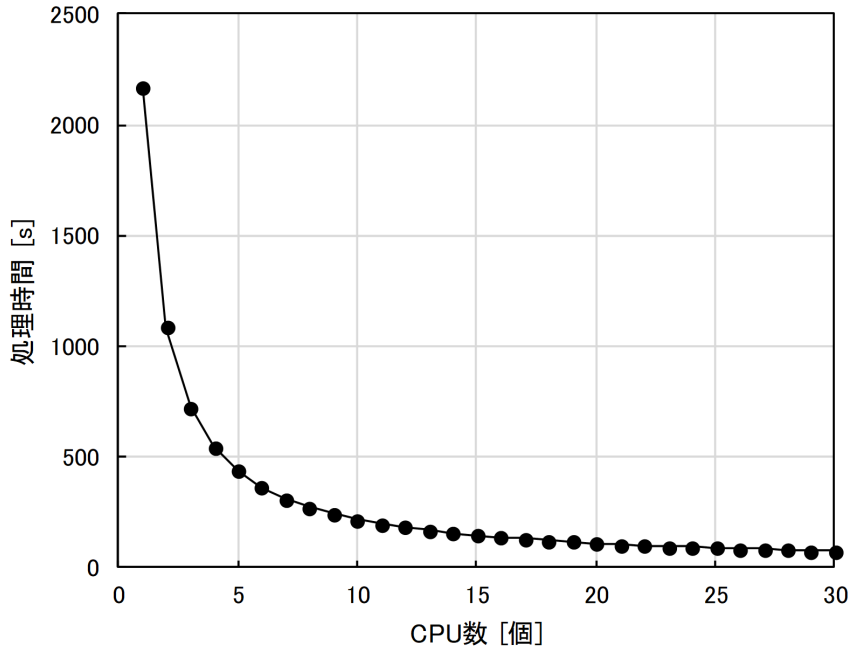

図 4　CPU の個数と処理時間の関係

10.5　考察

まずは、正しく計算が行われていることを確認するためにマチンの公式 (6) を計算するプログラムを作成した。それをリスト 15 に示す。このプログラムの実行結果とリスト 14 は一致したため、正しく円周率を計算できていると言える。

$$\pi = 16\tan^{-1}\left(\frac{1}{5}\right) - 4\tan^{-1}\left(\frac{1}{239}\right) \tag{6}$$

表 5　CPU の個数と処理時間

CPU 数 [個]	処理時間 [s]	CPU1 個の処理時間に対する短縮率
1	2165.54	1.00
2	1083.79	2.00
3	722.74	3.00
4	542.74	3.99
5	435.03	4.98
6	362.66	5.97
7	311.22	6.96
8	272.53	7.95
9	242.42	8.93
10	218.28	9.92
11	198.77	10.89
12	182.46	11.87
13	168.42	12.86
14	156.55	13.83
15	146.30	14.80
16	137.34	15.77
17	129.36	16.74
18	122.20	17.72
19	115.87	18.69
20	110.15	19.66
21	105.00	20.62
22	100.43	21.56
23	96.16	22.52
24	92.32	23.46
25	88.63	24.43
26	85.27	25.40
27	82.16	26.36
28	79.34	27.30
29	76.66	28.25
30	74.12	29.22

リスト 15　マチンの公式を計算するプログラム

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4 #include "mulproc.h"
5
6 // 精度(桁数)
7 #define PREC 1000
8
9 int main(int argc, char **argv) {
```

```
10      int nsize;
11      int myrank;
12      int my_name_len;
13      char my_name[MPI_MAX_PROCESSOR_NAME];
14
15      nsize = 1;
16      myrank = 0;
17
18      int i, j;
19      int n;
20      double start_t;
21      double finish_t;
22      struct NUMBER tmp;
23
24      int a[] = {24, 8, 4};
25      int b[] = {8, 57, 239};
26
27      MPI_Init(&argc, &argv);
28      MPI_Comm_size(MPI_COMM_WORLD, &nsize);
29      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
30      MPI_Get_processor_name(my_name, &my_name_len);
31
32      if (argc <= 1) {
33          if (myrank == 0)
34              printf("引数が与えられていません。\n");
35
36          MPI_Finalize();
37          return 0;
38      }
39
40      // 引数を整数型に変換
41      n = atoi(argv[1]);
42
43      if (n <= 0) {
44          if (myrank == 0)
45              printf("引数の値が小さすぎます。\n");
46
47          MPI_Finalize();
48          return 0;
49      }
50
51      // 処理開始時間を取得
52      start_t = MPI_Wtime();
53
54      // 精度
55      struct NUMBER prec;
56      setInt(&prec, 1);
57
58      for (i = 0; i < PREC; i++) {
59          mulBy10(&prec, &tmp);
60          copyNumber(&tmp, &prec);
61      }
62
63      // 合計
64      struct NUMBER sum;
65      clearByZero(&sum);
```

```
66
67      for (i = 0; i < sizeof(a) / sizeof(a[0]); i++) {
68          struct NUMBER base;
69          struct NUMBER div;
70          struct NUMBER upd;
71          struct NUMBER bunshi;
72
73          setInt(&base, b[i] * b[i]);
74          setInt(&div, b[i]);
75          setInt(&upd, 1);
76
77          for (j = 0; j < myrank; j++) {
78              multiple(&div, &base, &tmp);
79              copyNumber(&tmp, &div);
80          }
81
82          for (j = 0; j < nsize; j++) {
83              multiple(&upd, &base, &tmp);
84              copyNumber(&tmp, &upd);
85          }
86
87          setInt(&bunshi, a[i]);
88          multiple(&bunshi, &prec, &tmp);
89          copyNumber(&tmp, &bunshi);
90
91          for (j = myrank; j < n; j += nsize) {
92              struct NUMBER bunbo;
93              struct NUMBER calc;
94
95              // 分母の計算
96              setInt(&bunbo, 2 * j + 1);
97              multiple(&bunbo, &div, &tmp);
98              copyNumber(&tmp, &bunbo);
99
100             // 分数の計算
101             divide(&bunshi, &bunbo, &calc, &tmp);
102
103             // 加減算
104             if ((j % 2 == 0) ^ (i != 0))
105                 add(&sum, &calc, &tmp);
106             else
107                 sub(&sum, &calc, &tmp);
108
109             copyNumber(&tmp, &sum);
110
111             // 割る数を更新
112             multiple(&div, &upd, &tmp);
113             copyNumber(&tmp, &div);
114         }
115     }
116
117     if (myrank == 0) {
118         MPI_Status status;
119
120         int recv_n[KETA];
121         int recv_sign;
```

```
122
123         struct NUMBER ans;
124         struct NUMBER recv;
125
126         copyNumber(&sum, &ans);
127
128         for (i = 1; i < nsize; i++) {
129             MPI_Recv(recv_n, KETA, MPI_INTEGER, i, 100 + i, MPI_COMM_WORLD, &status);
130             MPI_Recv(&recv_sign, 1, MPI_INTEGER, i, 200 + i, MPI_COMM_WORLD, &status);
131
132             for (j = 0; j < KETA; j++)
133                 recv.n[j] = recv_n[j];
134             recv.sign = recv_sign;
135
136             add(&ans, &recv, &tmp);
137             copyNumber(&tmp, &ans);
138         }
139
140         // 処理終了時間を取得
141         finish_t = MPI_Wtime();
142
143         // 表示
144         dispNumber(&ans);
145         printf("time: %.4f s\n", finish_t - start_t);
146     } else {
147         MPI_Send(sum.n, KETA, MPI_INTEGER, 0, 100 + myrank, MPI_COMM_WORLD);
148         MPI_Send(&sum.sign, 1, MPI_INTEGER, 0, 200 + myrank, MPI_COMM_WORLD);
149     }
150
151     MPI_Finalize();
152
153     return 0;
154 }
```

また、図 4 を見ると、反比例のグラフとなっている。さらに、表 5 の CPU1 個の処理時間に対する短縮率より、おおよそ CPU の数の分だけ処理時間の短縮がされていることが分かる。このように並列化の効果が強く出ているのは、本プログラムが通信に対して計算の比率が大きいためと考えられる。このことから、効率的な並列計算を行うには、できるだけ通信を少なくすることが重要であることが分かる。

11 課題 6

11.1 課題内容

ライフゲームを並列処理で実行するプログラムを作成する。

11.2 プログラムリスト

課題 6 のプログラムを、リスト 16 に示す。

リスト 16　課題 6 のプログラム

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
```

```
4  #include <unistd.h>
5  #include <mpi.h>
6
7  #define H 10
8  #define W 2
9
10 int main(int argc, char **argv) {
11     int nsize;
12     int myrank;
13     int my_name_len;
14     char my_name[MPI_MAX_PROCESSOR_NAME];
15
16     MPI_Init(&argc, &argv);
17     MPI_Comm_size(MPI_COMM_WORLD, &nsize);
18     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
19     MPI_Get_processor_name(my_name, &my_name_len);
20
21     int i, j;
22     int x, y;
23
24     srand(time(NULL) + myrank * 100);
25
26     // マップの状態
27     int p_cells[H + 2][H];
28     int cells[W][H] = {{0}};
29
30     // ランダムに配置
31     for (y = 0; y < H; y++) {
32         for (x = 0; x < W; x++) {
33             cells[x][y] = rand() % 2;
34         }
35     }
36
37     int gen = 0;
38
39     while (1) {
40         // 前回の状態を保存
41         for (y = 0; y < H; y++) {
42             for (x = 0; x < W; x++) {
43                 p_cells[x + 1][y] = cells[x][y];
44             }
45         }
46
47         // のりしろ部分の通信
48         MPI_Status status;
49
50         int l_rank = (myrank - 1 + nsize) % nsize;
51         int r_rank = (myrank + 1 + nsize) % nsize;
52
53         MPI_Send(cells[0], H, MPI_INTEGER, l_rank, 100 + myrank, MPI_COMM_WORLD);
54         MPI_Send(cells[W - 1], H, MPI_INTEGER, r_rank, 200 + myrank, MPI_COMM_WORLD);
55
56         MPI_Recv(p_cells[W + 1], H, MPI_INTEGER, r_rank, 100 + r_rank, MPI_COMM_WORLD, &
       status);
57         MPI_Recv(p_cells[0], H, MPI_INTEGER, l_rank, 200 + l_rank, MPI_COMM_WORLD, &status)
       ;
```

```
58
59          // 更新
60          for (y = 0; y < H; y++) {
61              for (x = 0; x < W; x++) {
62                  // 周囲の生存セルをカウント
63                  int cnt = 0;
64
65                  for (i = -1; i <= 1; i++) {
66                      for (j = -1; j <= 1; j++) {
67                          if (i == 0 && j == 0) continue;
68                          if (p_cells[x + j + 1][(y + i + H) % H]) cnt++;
69                      }
70                  }
71
72                  if (cnt == 3) {
73                      // 誕生
74                      cells[x][y] = 1;
75                  } else if (cnt <= 1 || cnt >= 4) {
76                      // 死滅
77                      cells[x][y] = 0;
78                  }
79              }
80          }
81
82          // 集約して表示
83          if (myrank == 0) {
84              int all_cells[W * nsize][H];
85
86              for (y = 0; y < H; y++) {
87                  for (x = 0; x < W; x++) {
88                      all_cells[x][y] = cells[x][y];
89                  }
90              }
91
92              for (i = 1; i < nsize; i++) {
93                  for (x = 0; x < W; x++) {
94                      MPI_Recv(all_cells[i * W + x], H, MPI_INTEGER, i, 300 + i,
    MPI_COMM_WORLD, &status);
95                  }
96              }
97
98              printf("\033[;H\033[2J");
99              printf("Generation: %d\n", gen);
100
101             for (y = 0; y < H; y++) {
102                 for (x = 0; x < W * nsize; x++) {
103                     if (all_cells[x][y])
104                         printf("@");
105                     else
106                         printf(".");
107                 }
108
109                 printf("\n");
110             }
111         } else {
112             for (x = 0; x < W; x++)
```

```
113                 MPI_Send(cells[x], H, MPI_INTEGER, 0, 300 + myrank, MPI_COMM_WORLD);
114
115             usleep(100000);
116         }
117
118         gen++;
119         MPI_Barrier(MPI_COMM_WORLD);
120     }
121
122     MPI_Finalize();
123
124     return 0;
125 }
```

11.3 プログラムの説明

本プログラムでは、ライフゲームのセルを CPU の個数だけ縦に分割している。こうすることで、各 CPU について、通信を行うのは左右の CPU だけで済む。

11.4 実行結果

グライダーを初期配置として実行した結果をリスト 17–21 に示す。

リスト 17　グライダーの第 0 世代

```
Generation: 0
..........
...@......
.@.@......
..@@......
..........
..........
..........
..........
..........
..........
```

リスト 18　グライダーの第 1 世代

```
Generation: 1
..........
..@.......
...@@.....
..@@......
..........
..........
..........
..........
..........
```

```
..........
```

リスト 19　グライダーの第 2 世代

```
Generation: 2
..........
...@......
....@.....
..@@@.....
..........
..........
..........
..........
..........
..........
```

リスト 20　グライダーの第 3 世代

```
Generation: 3
..........
..........
..@.@.....
...@@.....
...@......
..........
..........
..........
..........
..........
```

リスト 21　グライダーの第 4 世代

```
Generation: 4
..........
..........
....@.....
..@.@.....
...@@.....
..........
..........
..........
..........
..........
```

次に、ランダムなセルの状態から 10000 世代実行するプログラムで、CPU の数 N を 1~30 個に増やしながら処理時間を測定した。全ての CPU が作業が行われていない状態で測定を行い、それぞれ 10 回の平均値を取った。

それぞれの CPU の個数についての処理時間についてまとめたものを表 6 に、グラフにプロットしたものを図 5 に示す。

表 6　CPU の個数と処理時間

CPU 数 [個]	処理時間 [s]	CPU1 個の処理時間に対する短縮率
1	154.46	1.00
2	77.82	1.98
3	52.15	2.96
4	39.82	3.88
5	32.37	4.77
6	27.01	5.72
7	23.35	6.62
8	20.57	7.51
9	18.60	8.31
10	16.80	9.19
11	15.49	9.97
12	14.34	10.77
13	13.25	11.66
14	12.34	12.51
15	11.60	13.32
16	11.07	13.95
17	10.62	14.55
18	10.21	15.13
19	9.68	15.95
20	9.21	16.77
21	8.91	17.34
22	8.65	17.86
23	8.31	18.59
24	8.03	19.24
25	7.78	19.84
26	7.58	20.39
27	7.31	21.13
28	7.15	21.61
29	6.91	22.34
30	6.75	22.90

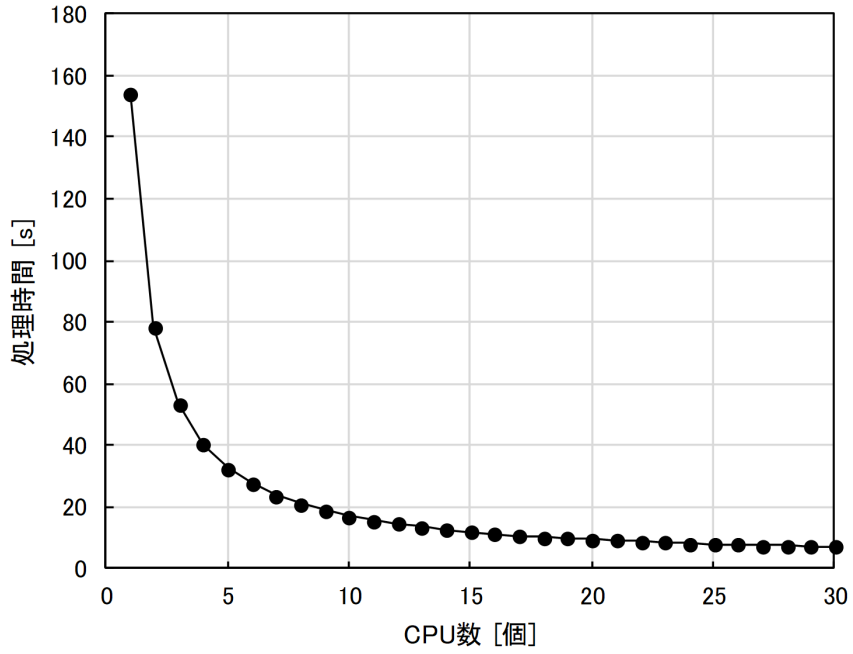

図 5　CPU の個数と処理時間の関係

11.5　考察

リスト 17–21 を見ると、4 世代後に初期と同じ形状になり、初期地点から 1 つ右下に移動していることが分かる。よって、正しくグライダーを飛ばすことができたため、ライフゲームのプログラムを作成することができた。

また、図 6 を見ると、反比例のグラフであるように見える。しかし、表 6 を見ると、使用した CPU の個数に対してあまり効率化は行われていない。このことから、本プログラムのような通信が多い並列処理では、並列処理の恩恵を受けづらいと言える。

参考文献

[1] OpenMPI: Open Source High Performance Computing, `https://www.open-mpi.org/`
[2] MPICH | High-Performance Portable MPI, `https://www.mpich.org/`