Program 2 Report

Yangxiao Wang

Contents

1 Documentation

In this project, we tried to build a two-dimensional wave diffusion program by using Schroedinger's Wave Dissemination formula. Since the formula requires heavy computational power, we proposed to use parallelization techniques to implement it. And in this report, I would demonstrate the code and time required for each of the following implementations: sequential, MPI, and hybrid form of MPI and OpenMPI.

The MPI methods I used is the two blocking communications *MPI_Send()* and *MPI_Recv()*. The advantage of using blocking communication is I do not need to handle the synchronization problem. The program can only continue to work until all communication is done (all data is sent and received). In addition, to exchange the boundary data, my approach was sending the boundary stripe at beginning of every loop of t (time). And for implementation of OpenMPI, I simply parallelized the Schroedinger's formula part. For each ranks, it has multiple threads to compute the formula.

2 Source code

2.1 Wave2D.cpp

```
#include <iostream>
#include "Timer.h"
#include <stdlib.h>   // atoi
#include <math.h>
#include <stdio.h>

int default_size = 100;  // the default system size
int defaultCellWidth = 8;
double c = 1.0;      // wave speed
double dt = 0.1;     // time quantum
double dd = 2.0;     // change in system

using namespace std;

int main(int argc, char *argv[]) {
   // verify arguments
   if (argc != 4) {
      cerr << "usage: Wave2D size max_time interval" << endl;
      return -1;
   }
   int size = atoi(argv[1]);
   int max_time = atoi(argv[2]);
   int interval = atoi(argv[3]);

   if (size < 100 || max_time < 3 || interval < 0) {
      cerr << "usage: Wave2D size max_time interval" << endl;
      cerr << "       where size >= 100 && time >= 3 && interval >= 0" << endl;
      return -1;
   }

   // create a simulation space
   double z[3][size][size];
   for (int p = 0; p < 3; p++)
      for (int i = 0; i < size; i++)
         for (int j = 0; j < size; j++)
            z[p][i][j] = 0.0; // no wave
```

```
37
38      // start a timer
39      Timer time;
40      time.start();
41
42      // time = 0;
43      // initialize the simulation space: calculate z[0][][]
44      int weight = size / default_size;
45      for (int i = 0; i < size; i++) {
46         for (int j = 0; j < size; j++) {
47            if (i > 40 * weight && i < 60 * weight &&
48               j > 40 * weight && j < 60 * weight) {
49               z[0][i][j] = 20.0;
50            } else {
51               z[0][i][j] = 0.0;
52            }
53         }
54      }
55
56      // time = 1
57      for (int i = 1; i < size - 1; i++) {
58         for (int j = 1; j < size - 1; j++) {
59            z[1][i][j] = z[0][i][j] + (pow(c, 2) / 2) * pow(dt / dd, 2) * (z[0][i +↩
                  1][j] + z[0][i - 1][j] + z[0][i][j + 1] + z[0][i][j - 1] - (4.0 * ↩
                z[0][i][j]));
60         }
61      }
62
63      // simulate wave diffusion from time = 2
64      for (int t = 2; t < max_time; t++) {
65         int time = t % 3;
66         for (int i = 1; i < size - 1; i++) {
67            for (int j = 1; j < size - 1; j++) {
68               int time_1, time_2;
69               //rotate z
70               if (time == 0) {
71                  time_1 = 2;
72                  time_2 = 1;
73               } else if (time == 1) {
74                  time_1 = 0;
75                  time_2 = 2;
76               } else {
77                  time_1 = 1;
78                  time_2 = 0;
79               }
80               //calculation
81               z[time][i][j] =
82               2.0 * z[time_1][i][j] - z[time_2][i][j] + (pow(c, 2) * pow(dt / dd, ↩
                     2)  * (z[time_1][i + 1][j] + z[time_1][i - 1][j] + z[time_1][i][↩
                     j + 1] + z[time_1][i][j - 1] - (4.0 * z[time_1][i][j])));
83            }
84         }
85         //print out
86         if (interval != 0 && t % interval == 0) {
87            cout << t << endl;
88            for (int j = 0; j < size; j++) {
89               for (int i = 0; i < size; i++) {
90                  cout << z[time][i][j] << " ";
91               }
92               cout << endl;
93            }
```

```
94              cout << endl;
95          }
96      } // end of simulation
97
98      // finish the timer
99      cerr << "Elapsed time = " << time.lap() << endl;
100     return 0;
101 }
```

2.2 Wave2D mpi.cpp

```
#include <iostream>
#include "Timer.h"
#include <stdlib.h>   // atoi
#include <math.h>
#include <mpi.h>
#include <stdio.h>
#include <omp.h>

int default_size = 100;  // the default system size
int defaultCellWidth = 8;
double c = 1.0;      // wave speed
double dt = 0.1;     // time quantum
double dd = 2.0;     // change in system

using namespace std;

int main(int argc, char *argv[]) {
   int my_rank = 0;            // used by MPI
   // used by MPI

   // verify arguments
   if (argc != 5) {
      cerr << "usage: Wave2D size max_time interval n_thread" << endl;
      return -1;
   }
   int size = atoi(argv[1]);
   int max_time = atoi(argv[2]);
   int interval = atoi(argv[3]);
   int nThreads = atoi(argv[4]);
   int mpi_size;

   if (size < 100 || max_time < 3 || interval < 0 || nThreads <= 0) {
      cerr << "usage: Wave2D size max_time interval" << endl;
      cerr << "where size >= 100 && time >= 3 && interval >= 0 && mpi_size > 0"
         << endl;
      return -1;
   }

   // start MPI
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
   // change # of threads for openMP
   omp_set_num_threads(nThreads);

   // create a simulation space
   double z[3][size][size];
   for (int p = 0; p < 3; p++)
      for (int i = 0; i < size; i++)
         for (int j = 0; j < size; j++)
            z[p][i][j] = 0.0; // no wave

   // start a timer
   Timer time;
   time.start();

   // time = 0;
   // initialize the simulation space: calculate z[0][][]
```

```
58      int weight = size / default_size;
59      for (int i = 0; i < size; i++) {
60         for (int j = 0; j < size; j++) {
61            if (i > 40 * weight && i < 60 * weight && j > 40 * weight && j < 60 * ↩
                  weight) {
62               z[0][i][j] = 20.0;
63            } else {
64               z[0][i][j] = 0.0;
65            }
66         }
67      }
68
69      // time = 1, and parallelization
70      #pragma omp parallel for
71      for (int i = 1; i < size - 1; i++) {
72         for (int j = 1; j < size - 1; j++) {
73            z[1][i][j] = z[0][i][j] + (pow(c, 2) / 2) * pow(dt / dd, 2) * (z[0][i +↩
                  1][j] + z[0][i - 1][j] + z[0][i][j + 1] + z[0][i][j - 1] - (4.0 * ↩
                  z[0][i][j]));
74         }
75      }
76
77
78      int stripe = size / mpi_size;     // partitioned stripe
79
80      // simulate wave diffusion from time = 2
81      for (int t = 2; t < max_time; t++) {
82         //rotate z
83         int time = t % 3;
84         int time_1, time_2;
85         if (time == 0) {
86            time_1 = 2;
87            time_2 = 1;
88         } else if (time == 1) {
89            time_1 = 0;
90            time_2 = 2;
91         } else {
92            time_1 = 1;
93            time_2 = 0;
94         }
95
96         //exchange boundary data
97         if (my_rank == 0) {
98            MPI_Send(*(*(z + time_1) + stripe * (my_rank + 1) - 1), size, ↩
                  MPI_DOUBLE, my_rank + 1, 0, MPI_COMM_WORLD);
99
100           MPI_Status status;
101           MPI_Recv(*(*(z + time_1) + stripe), size, MPI_DOUBLE, my_rank + 1, 0, ↩
                  MPI_COMM_WORLD, &status);
102        } else if (my_rank == mpi_size - 1) {
103           MPI_Send(*(*(z + time_1) + stripe * my_rank), size, MPI_DOUBLE, my_rank↩
                  -     1, 0, MPI_COMM_WORLD);
104
105           MPI_Status status;
106           MPI_Recv(*(*(z + time_1) + stripe * my_rank - 1), size, MPI_DOUBLE, ↩
                  my_rank - 1, 0, MPI_COMM_WORLD, &status);
107        } else {
108           MPI_Send(*(*(z + time_1) + stripe * my_rank), size, MPI_DOUBLE, my_rank↩
                  - 1, 0,  MPI_COMM_WORLD);
109           MPI_Send(*(*(z + time_1) + stripe * (my_rank + 1) - 1), size, ↩
                  MPI_DOUBLE, my_rank + 1, 0, MPI_COMM_WORLD);
```

```
110
111             MPI_Status status;
112             MPI_Recv(*(*(z + time_1) + stripe * my_rank - 1), size, MPI_DOUBLE, ↩
                    my_rank - 1, 0, MPI_COMM_WORLD, &status);
113             MPI_Recv(*(*(z + time_1) + stripe * (my_rank + 1)), size, MPI_DOUBLE, ↩
                    my_rank + 1, 0, MPI_COMM_WORLD, &status);
114         }
115
116         //Parallelization for the Schroedinger's formula
117         #pragma omp parallel for
118         for (int i = my_rank * stripe; i < (my_rank + 1) * stripe; i++) {
119             if (i == 0 || i == size - 1) {
120                 continue;
121             }
122             for (int j = 1; j < size - 1; j++) {
123
124                 z[time][i][j] =
125                 2.0 * z[time_1][i][j] - z[time_2][i][j] + (pow(c, 2) * pow(dt / dd, ↩
                        2)
126                 * (z[time_1][i + 1][j] + z[time_1][i - 1][j] + z[time_1][i][j + 1] +
127                 z[time_1][i][j - 1] - (4.0 * z[time_1][i][j])));
128             }
129         }
130
131         //output if it's interval
132         if (interval != 0 && t % interval == 0) {
133             //Aggregate all results from all ranks
134             if (my_rank == 0) {
135                 for (int rank = 1; rank < mpi_size; ++rank) {
136                     MPI_Status status;
137                     MPI_Recv(*(*(z + time) + rank * stripe), stripe * size, ↩
                            MPI_DOUBLE, rank, 0, MPI_COMM_WORLD, &status);
138                 }
139
140                 cout << t << endl;
141                 for (int j = 0; j < size; j++) {
142                     for (int i = 0; i < size; i++) {
143                         cout << z[time][i][j] << " ";
144                     }
145                     cout << endl;
146                 }
147                 cout << endl;
148
149             } else {
150                 MPI_Send(*(*(z + time) + my_rank * stripe), stripe * size, ↩
                        MPI_DOUBLE, 0, 0,
151                 MPI_COMM_WORLD);
152             }
153         }
154     } // end of simulation
155
156     MPI_Finalize(); // shut down MPI
157
158     // finish the timer
159     if(my_rank == 0) {
160         cerr << "Elapsed time = " << time.lap() << endl;
161     }
162
163     return 0;
164 }
```

3 Execution output

3.1 Output analysis

1. The performance improvement with four machines: 5721050 / 1575955 = 3.63 times
2. The performance improvement with four machines with multithreading: 5721050 / 874590 = 6.54 times

3.2 Execution output

Check if output is correct

```
[wyxiao_css534@cssmpi1 prog2]$ ./Wave2D 576 500 50 > reS.txt
Elapsed time = 7313114
[wyxiao_css534@cssmpi1 prog2]$ mpirun -n 4 ./Wave2D_mpi 576 500 50 4 > reF.txt
Elapsed time = 6991195
[wyxiao_css534@cssmpi1 prog2]$ diff reF.txt reS.txt
[wyxiao_css534@cssmpi1 prog2]$
```

Check output the performance improvement with four machines: 5721050 / 1575955 = 3.63 times

```
[wyxiao_css534@cssmpi1 prog2]$ ./Wave2D 576 500 0
Elapsed time = 5721050
[wyxiao_css534@cssmpi1 prog2]$ mpirun -n 4 ./Wave2D_mpi 576 500 0 1
Elapsed time = 1575955
```

Check output the performance improvement with four machines with multithreading: 5721050 / 874590 = 6.54 times

```
[wyxiao_css534@cssmpi1 prog2]$ mpirun -n 4 ./Wave2D_mpi 576 500 0 4
Elapsed time = 874590
```

4 Discussions

I noticed the cost of communication is not ideal, especially the aggregation of all ranks' data. Therefore I should keep the numbers of aggregation as low as possible, the aggregation only happens when the output is required.

To improve the current implementation, the best way I could think of is using unblocking communication (*MPI_Isend* and *MPI_Irecv*) when aggregation happens. Then the program would not stop when perform aggregation. And add a *MPI_Wait* statement before $t = t_{current} + 3$ to make sure previous communication is finished when changing the values. Since the aggregation is the most time consuming part.

5 Lab Sessions 2

Lab 2 we parallelize a programs that compute the result of multiplication of matrix using sequential and MPI. As the result shown, four MPI ranks decrease significant amount of time that required to finish the program. However, if the size of matrix is too small, the performance would decrease as shown in the first execution output.

5.1 Source Code

```
#include "mpi.h"
#include <stdlib.h> // atoi
#include <iostream> // cerr
#include "Timer.h"

using namespace std;

void init(double *matrix, int size, char op) {
   for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++)
         matrix[i * size + j] = (op == '+') ? i + j : ((op == '-') ? i - j : 0);
}

void print(double *matrix, int size, char id) {
   for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++)
         cout << id << "[" << i << "][" << j << "] = " << matrix[i * size + j] ↩
            << endl;
}

void multiplication(double *a, double *b, double *c, int stripe, int size) {
   for (int k = 0; k < size; k++)
      for (int i = 0; i < stripe; i++)
         for (int j = 0; j < size; j++)
            // c[i][k] += a[i][j] * b[j][k];
            c[i * size + k] += a[i * size + j] * b[j * size + k];
}

int main(int argc, char *argv[]) {
   int my_rank = 0;            // used by MPI
   int mpi_size = 1;           // used by MPI
   int size = 400;             // array size
   bool print_option = false;  // print out c[] if it is true
   Timer timer;

   // variables verification
   if (argc == 3) {
   if (argv[2][0] == 'y')
      print_option = true;
   }

   if (argc == 2 || argc == 3) {
      size = atoi(argv[1]);
   } else {
      cerr << "usage:   matrix size [y|n]" << endl;
      cerr << "example: matrix 400   y" << endl;
      return -1;
   }

   MPI_Init(&argc, &argv); // start MPI
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

   // matrix initialization
   double *a = new double[size * size];
   double *b = new double[size * size];
   double *c = new double[size * size];

```

```
58      if (my_rank == 0) { // master initializes all matrices
59         init(a, size, '+');
60         init(b, size, '-');
61         init(c, size, '0');
62
63         // print initial values
64         if (false) {
65            print(a, size, 'a');
66            print(b, size, 'b');
67         }
68
69         // start a timer
70         timer.start();
71      } else {                // slavs zero-initializes all matrices
72         init(a, size, '0');
73         init(b, size, '0');
74         init(c, size, '0');
75      }
76
77      // broadcast the matrix size to all.
78      MPI_Bcast(&size, 1, MPI_INT, 0, MPI_COMM_WORLD);
79
80      int stripe = size / mpi_size;     // partitioned stripe
81
82      // master sends each partition of a[] to a different slave
83      // master also sends b[] to all slaves
84      if (my_rank == 0) {
85         for (int rank = 1; rank < mpi_size; ++rank) {
86            MPI_Send(a + rank * stripe * size, size * stripe, MPI_DOUBLE, rank, 0, ↩
                  MPI_COMM_WORLD);
87            MPI_Send(b, size * size, MPI_DOUBLE, rank, 0, MPI_COMM_WORLD);
88         }
89      } else {
90         MPI_Status status;
91         MPI_Recv(a, size * stripe, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
92         MPI_Recv(b, size * size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
93      }
94
95      multiplication(a, b, c, stripe, size); // all ranks should compute ↩
          multiplication
96
97      // master receives each partition of c[] from a different slave
98      if (my_rank == 0) {
99         for (int rank = 1; rank < mpi_size; ++rank) {
100           MPI_Status status;
101           MPI_Recv(c + rank * stripe * size, size * stripe, MPI_DOUBLE, rank, 0, ↩
                  MPI_COMM_WORLD,
102           &status);
103        }
104     } else {
105        MPI_Send(c, stripe * size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
106     }
107
108     if (my_rank == 0)
109        // stop the timer
110        cout << "elapsed time = " << timer.lap() << endl;
111
112     // results
113     if (print_option && my_rank == 0)
114        print(c, size, 'c');
115
```

```
116         MPI_Finalize(); // shut down MPI
117     }
```

5.2 Execution output

Check output is correct

```
1   [wyxiao_css534@cssmpi1 lab2]$ ./matrix 100 y > maxS.txt
2   [wyxiao_css534@cssmpi1 lab2]$ mpirun -n 4 ./matrix_mpi 100 y > maxM.txt
3   [wyxiao_css534@cssmpi1 lab2]$ diff maxS.txt maxM.txt
4   1c1
5   < elapsed time = 5185
6   ---
7   > elapsed time = 7133
8   [wyxiao_css534@cssmpi1 lab2]$
```

Check output the performance improvement: 774693 / 258639 = 2.9952 times

```
1   [wyxiao_css534@cssmpi1 lab2]$ ./matrix 500
2   elapsed time = 774693
3   [wyxiao_css534@cssmpi1 lab2]$ mpirun -n 4 ./matrix_mpi 500
4   elapsed time = 258639
5   [wyxiao_css534@cssmpi1 lab2]$
```