# Message-passing Programming Coursework Part2

B177252

December 1, 2020

## 1 Introduction

In part 1, a specific plan was made to design, implement and test the 2-D decomposition parallel program for percolation problem. In this part, the new parallel program is successfully implemented and test of program is also conducted. This paper describes the details of changes to the proposal in part 1 as well as the correctness and performance test.

## 2 Implementation

### 2.1 Changes of proposal

In this section, we describe in detail about the changes of the initial plan in part 1.

The first change is the derived data type for halos. In practice, using struct to represent halos has many redundant procedures in which many unnecessary halos are sent to neighbour processes. Therefore, in part 2 we adopt vector to represent halos, which avoids redundant communication. Two vectors are defined for halos. One represents the halos up and down, another represents the halos in the left and right.

Second change is about the non-blocking communication when swapping halos. The proposed plan uses MPI_Irecv without MPI_Wait, which is a mistake. Hence we use blocking receive MPI_Recv instead, which has exactly the same effect as non-blocking receive. Moreover, in practice, there exists many sending requests in practice, so we use MPI_Waitall instead of MPI_Wait.

Last change is using periodic boundary conditions in the vertical direction. To implement this, we can simply set periods to be 0 and 1 when using MPI_Cart_create to create Cartesian topology.

### 2.2 Core problems solved

The first biggest problem is the swap of halos. To swap halos, send the halos to the neighbour processes whose data type is vector using MPI_Issend. At the same time, use MPI_Recv to receive the halos from neighbour processes. Finally, update the 'old' map for each process for the next turn of swap of halos.

Another problem is how to make problem size and number of processes flexible. In the solution, we make them arguments as inputs. Mapsize and number of processes are initiated in percolate.h file, so the first step is delete them. Then in percolate.c, we add a block of code which initializes mapsize and seed from input. When running the program, we need to initialize -s and -l, which represents seed and mapsize respectively. For example, when running the prohgram with mapsize to be 400 and seed to be 5000, we need to add -s 5000 -l 400 in the command line. For flexible number of processes, we also need to delete the block of code judging whether size of processes equals 4. Then we can use -n to set the number of processes.

Finally, we consider the case when mapsize is not an exact multiple of the number of processes. To avoid bugs, we need to make sure that 2-d decomposition does distribute all elements in the map to all processes. The solution is adding a ceiling for the width and height of small map. For example, if the mapsize is 10 and the dimension of topology is (4,3), then width of small map for each process is the ceiling of 10/3, which is 4. If the width is 3, then 9 is smaller than 10, there would be some parts of the map that are not computed. We might be concerned that the last process is supposed to get a width of 2 rather than 4. If do so, in the worst case it could possibly make map unable to provide enough space. To solve this problem, we initialize the width of the map to be L + size, in which L is mapsize and size is number of processes. The motivation is to make sure map can provide enough space in any case. In this way, there is no need for additional logic to judge the boundary conditions.

### 2.3 Additional and changed files

As the mapsize is accessed by input, the fixed L would all be deleted and the array 'map' and 'old' would be allocated dynamically. Here we create arralloc.c and arralloc.h to define the function to allocate space for 2-d arrays. More- over, the percio.c would be changed slightly because L is not fixed any more. The two core files which are percolate.h and percoalte.c are definitely mostly changed by adding many MPI routines and modularizing some parts of the code into functions which need to be pre-defined in percolate.h.

## 3 Correctness and performance test

### 3.1 Correctness Test

Correctness test is divided into two parts. For the first part, we use the test cases exactly the same as those in part 1 to test the basic correctness. Specifically, we set the mapsize to be 432, the number of processes to be 4 and the number of clusters to be 8. We vary the seed to see whether the parallel program has exactly the same clusters as the serial program. Moreover, we also need to test the cases when mapsize is not an exact multiple of number of processes. The mapsize we choose is 431 and seed is set to be 5000. Then we vary the number of processes to test the correctness. The reason why we choose 431 to be the map size is that 431 is a prime number. The results are shown in table 1 and 2 as well as figure 1. We can see that the improved parallel program passes all the tests and proves to be correct.

| Correctness Test 1 | | |
|---|---|---|
| seed | serial(largest 8 clusters) | parallel(largest 8 clusters) |
| 1000 | 32546, 19748, 9929, 5010, 4056, 1187, 1060, 1004 | 32546, 19748, 9929, 5010, 4056, 1187, 1060, 1004 |
| 2000 | 53883, 5946, 2994, 2663, 1918, 1738, 1473, 1444 | 53883, 5946, 2994, 2663, 1918, 1738, 1473, 1444 |
| 3000 | 37047, 10624, 7752, 3743, 3475, 3097, 1439, 977 | 37047, 10624, 7752, 3743, 3475, 3097, 1439, 977 |
| 4000 | 39771, 22920, 2305, 1757, 1672, 1203, 1102, 1085 | 39771, 22920, 2305, 1757, 1672, 1203, 1102, 1085 |
| 5000 | 39908, 5472, 5039, 4529, 4201, 2757, 2385, 1492 | 39908, 5472, 5039, 4529, 4201, 2757, 2385, 1492 |

Table 1: Correctness test 1

| Correctness Test 2 | |
|---|---|
| number of processes | parallel(largest 8 clusters) |
| 1 | 23934, 20417, 7701, 5739, 3339, 2674, 2297, 1314 |
| 2 | 23934, 20417, 7701, 5739, 3339, 2674, 2297, 1314 |
| 3 | 23934, 20417, 7701, 5739, 3339, 2674, 2297, 1314 |
| 4 | 23934, 20417, 7701, 5739, 3339, 2674, 2297, 1314 |
| 5 | 23934, 20417, 7701, 5739, 3339, 2674, 2297, 1314 |
| 6 | 23934, 20417, 7701, 5739, 3339, 2674, 2297, 1314 |
| 7 | 23934, 20417, 7701, 5739, 3339, 2674, 2297, 1314 |
| 8 | 23934, 20417, 7701, 5739, 3339, 2674, 2297, 1314 |
| 9 | 23934, 20417, 7701, 5739, 3339, 2674, 2297, 1314 |
| 10 | 23934, 20417, 7701, 5739, 3339, 2674, 2297, 1314 |

Table 2: Correctness test 2

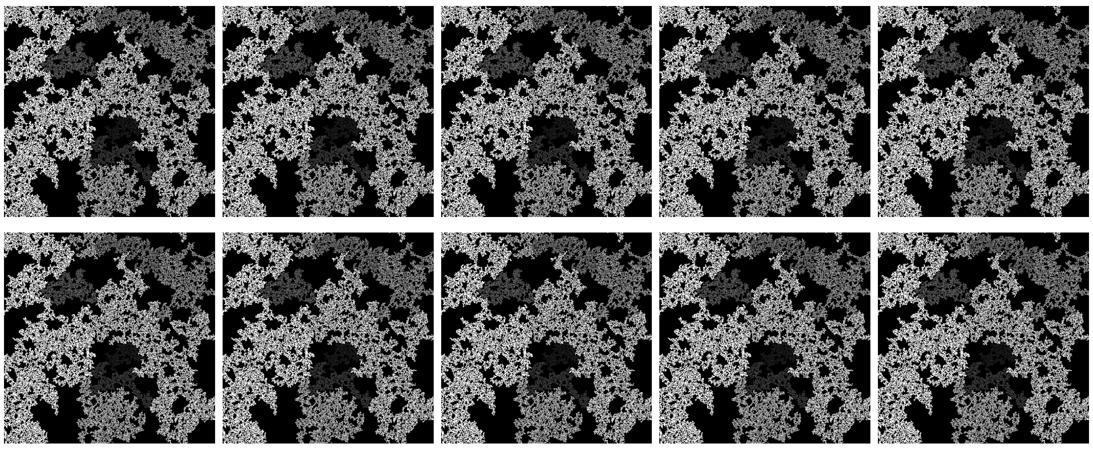


Figure 1: Pictures of map in correctness test 2

## 3.2 Performance Test

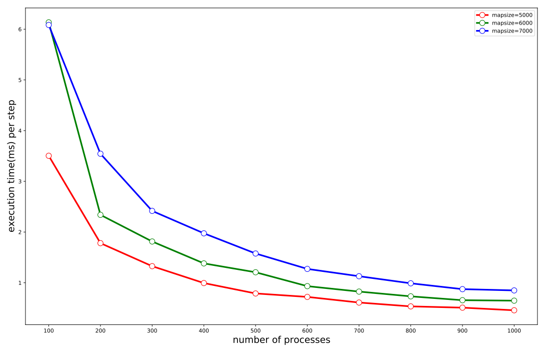
Performance test includes the test of execution time per step and speedup versus number of processes and execution time per step versus mapsize. The reason why we choose to measure execution time per step is clarified in part 1. For the first part of test, we set the mapsize to be 5000, 6000, 7000 respectively and seed to be 6543. Then we vary the number of processes to get performance data. For the second part, we set number of processes to be 100, 200 and 300 respectively and seed to be 6543. Then vary the mapsize to get performance data. In order to make accurate timing, test is conducted in multiple compute nodes. Therefore we need to write a SLURM script. For each case, we test the time for 10 times and use the average time as the final result. The results are shown in table 3 and 4 figure 2 and 3.

| Execution time(ms) per step versus number of processes | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| p m | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | 1000 |
| 5000 | 3.504 | 1.781 | 1.327. | 0.994 | 0.789 | 0.721 | 0.610 | 0.534 | 0.509 | 0.457 |
| 6000 | 6.135 | 2.339 | 1.814 | 1.382 | 1.207 | 0.933 | 0.826 | 0.732 | 0.656 | 0.647 |
| 7000 | 6.084 | 3.544 | 2.417 | 1.975 | 1.577 | 1.273 | 1.129 | 0.989 | 0.874 | 0.849 |

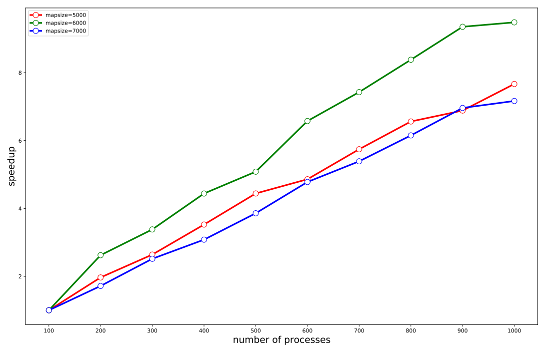
Table 3: Execution time per step versus number of processes

| Execution time(ms) per step versus mapsize | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| m p | 500 | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 |
| 100 | 0.066 | 0.174 | 0.359 | 0.607 | 0.914 | 1.210 | 1.641 | 2.194 | 2.579 | 3.246 |
| 200 | 0.053 | 0.096 | 0.203 | 0.350 | 0.510 | 0.733 | 1.014 | 1.231 | 1.509 | 1.764 |
| 300 | 0.049 | 0.0762 | 0.139 | 0.250 | 0.363 | 0.501 | 0.647 | 0.891 | 0.990 | 1.211 |

Table 4: Execution time per step versus mapsize



(a) Execution time versus number of processes



(b) Speedup versus number of processes

Figure 2: Performance versus number of processes

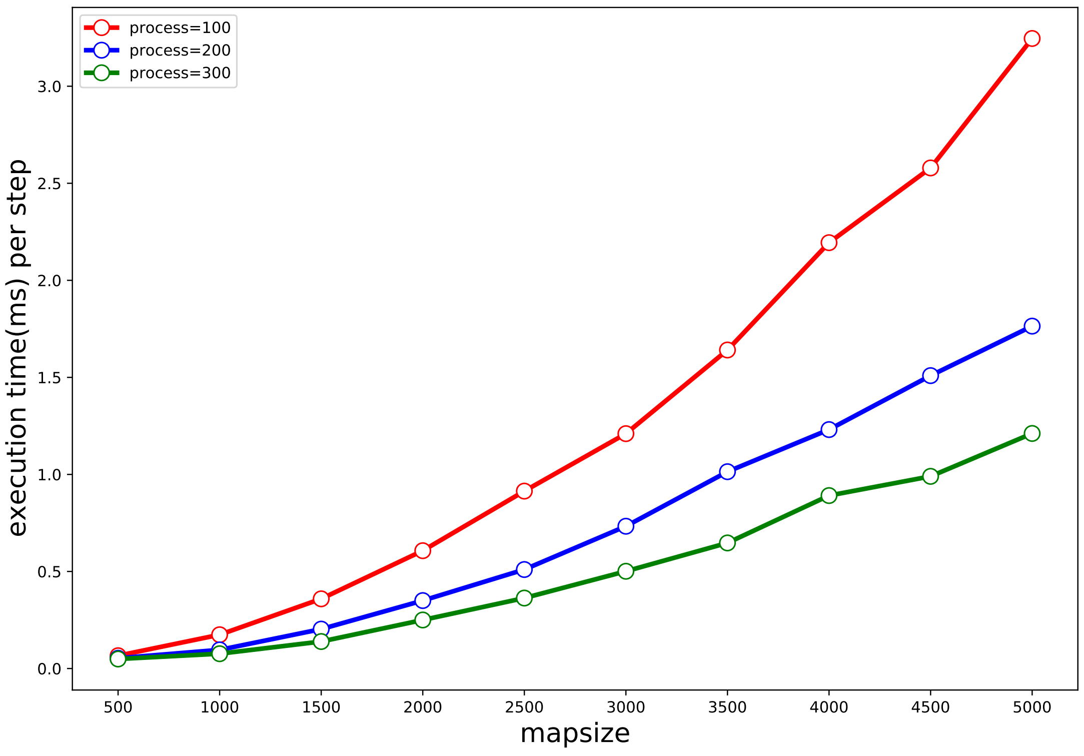


Figure 3: Performance versus mapsize

## 4 Discussion and Analysis

This part mainly discusses the results of performance test. As is shown in figure 2, performance improves with number of processes. Firstly, we can use Amdahl's Law to quantify the scaling. Amdahl's Law can be represented as below.

$$T_p = \alpha T_s + \frac{(1-\alpha)T_s}{P}$$

$T_p$, $T_s$, $\alpha$ and $P$ refer to the execution time after parallelism, execution time of serial program, proportion of serial part and number of processes respectively. In this problem, as we only focus on the execution time per step of the parallel part, $\frac{(1-\alpha)T_s}{P}$ is what we are concerned about. As is discussed in part 1, suppose the execution time per step in the serial and parallel code to be $T'_s$ and $T'_p$ and the number of processes to be $p$, the relation between them can be presented as $T'_p = \frac{T'_s}{p}$. The speedup can be presented as $\frac{T'_s}{T'_p}$, which equals $p$. We can check that the curve in the figure basically matches this formula but not very well. For example, when mapsize is 5000, $\frac{3.504}{1.781} = 1.967$, $\frac{3.504}{1.327} = 2.641$ etc. However, as the number of processes increases, the figure seems less and less accurate. This is because the time of communication between processes also increases, which can not be ignored.

When it comes to figure 3, execution time per step increases with mapsize. As we discussed in part 1, the relation between execution time per step and mapsize can be represented as $T_L = (\frac{L}{L_0})^2 T_{L_0}$. Here L refers to mapsize and $L_0$ refers to the first mapsize which is 300. Due to the same reason in the first test, the figure does not match the formula well but basically matches the trend that we predict.

From the analysis above, although we quantify the performance based on Amdahl's Law and inference, the performance data does have many deviations in reality. However, it does not mean the formula is wrong because communication of halos can not be ignored in practice. The point of quantification is to predict the trend rather than give an accurate prediction.

## 5 Conclusions

In conclusion, this paper successfully implemented the 2-d decomposition parallel solution of percolation problem and proved its correctness. Based on the previous 1-d decomposition version, the new program does not only make decomposition 2-d but also make mapsize and number of processes flexible as well as solve the problem of uneven distribution and make the boundaries of vertical directions periodic. In terms of performance, it generally increases with number of processes and decreases with mapsize. The relations between them can be represented as formulas that we derive. However, in practice, the figure does not perfectly match the quantification due to swap of halos.

However, there are also some improvement that can be expected in future work. For example, the definition of derived data type vector is very complicated. We can possibly find better MPI routines for 2-d decomposition. Moreover, we can also study how performance is related to seed or density.