MESSAGE-PASSING PROGRAMMING COURSEWORK PART 1

B177252

November 5, 2020

INTRODUCTION

Percolate problem is about judging whether there's a path across empty squares from top to bottom in a grid. This is problem that can easily be solved by serial code. However, the performance could be improved by parallelism. This report compares a simple parallel solution with the serial solution in terms of performance and correctness, then analyzes many drawbacks of the simple parallelism and proposes a plan to improve this solution.

SERIAL SOLUTION VERSUS PARALLEL SOLUTION

The parallel program is implemented by MPI. It divides the map into N small maps, in which N equals the number of processes, then distributes the small maps across the processes using MPI_Scatter. Each process now has a small map which replaces the map in the serial code. Processes need to communicate with each other in order to swap halos. When calculating new maps, each process records the number of squares that are changed and submits the number to controller process using MPI_Reduce.
In order to check the correctness of parallel code and whether it improves the performance. We need to test the parallel code and compare with the serial one.
The first part is correctness. Firstly, we set the number of processes to be 4 and number of clusters to be 8. Then we change the random number seed to see whether the largest 8 clusters of serial and parallel version are equal. The table 1 below is the experimental results.

Correctness Test		
seed	serial(largest 8 clusters)	parallel(largest 8 clusters)
1000	30048, 16515, 9929, 4364, 4056, 2109, 1932, 1060	30048, 16515, 9929, 4364, 4056, 2109, 1932, 1060
2000	46439, 5946, 3868, 2994, 2663, 2238, 1918, 1738	46439, 5946, 3868, 2994, 2663, 2238, 1918, 1738
3000	419970, 10624, 7752, 7440, 6872, 3743, 3475, 3097	19970, 10624, 7752, 7440, 6872, 3743, 3475, 3097
4000	37978, 10165, 9474, 2374, 2305, 1757, 1672, 1203	37978, 10165, 9474, 2374, 2305, 1757, 1672, 1203
5000	31827, 5472, 5039, 4529, 2868, 2757, 2676, 2385	31827, 5472, 5039, 4529, 2868, 2757, 2676, 2385

Table 1: Correctness test

As is shown in the table, the results of serial and parallel versions are exactly the same. Therefore, we could conclude that the correctness of parallelism is proved. Next part is performance test. Again we use the test cases used in correctness test. In each test case, we measure the execution time of serial and parallel programs. We are not interested in serial part of the code, so we only measure the average execution time per step. Most importantly, measuring execution time requires high accuracy, so we need to run the parallel code in the compute nodes. Table 2 below shows the performance of parallel version versus serial version.

Performance Test		
seed	serial(average time per step(ms))	parallel(average time per step(ms))
1000	0.288	0.074
2000	0.287	0.074
3000	0.288	0.074
4000	0.288	0.075
5000	0.288	0.075

Table 2: Performance test

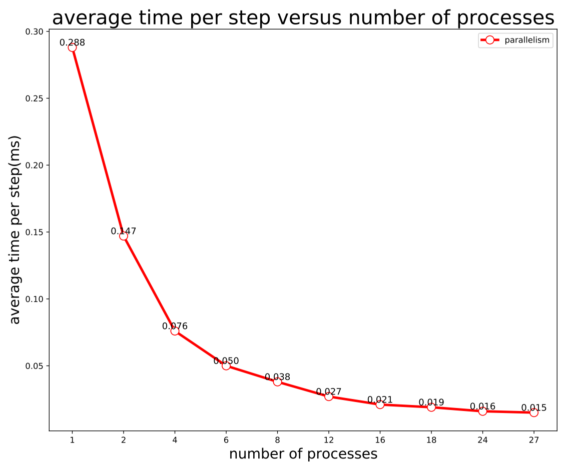
As we can see from the table, the performance gets largely improved after parallelism. Now we need to move a step further to see whether performance improves as the number of processes is increased. First, we set the seed to be 5000. Then we vary the number of processes and calculate the changes of average time per step. The figure 1 shows the execution time per step and speedup. Obviously, performance improves as the number of processes is increased. Actually, we can use Amdahl's Law to quantify the performance. The formula is presented below.

Amdahl's law:

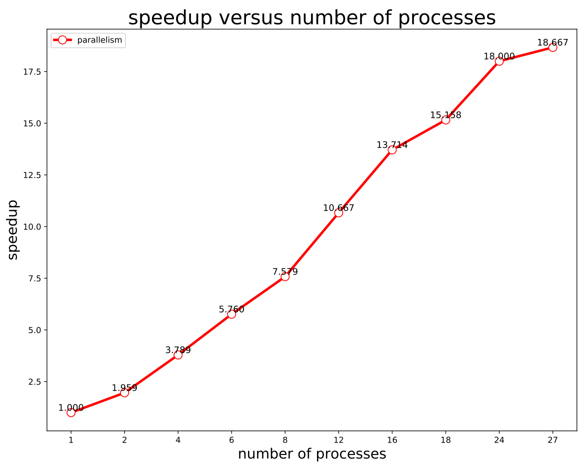
$$T_p = \alpha T_s + \frac{(1-\alpha)T_s}{P}$$

In this formula, T_p is the execution time of the parallel program. T_s is the execution of the serial program. α is the proportion of serial part. P is the number of processes. In this problem, as we only calculate the execution time per step of the parallel part, the formula can be changed into $T_p = \frac{(1-\alpha)T_s}{P}$. Time per step equals $\frac{T_p}{maxstep}$. Speedup equals $\frac{T_s}{T_p}$. We can also prove this formula from the figure 1. T_s is the execution time of the whole program when number of process equals 1. $(1-\alpha)T_s$ is the execution time of the calculation of all the steps in serial. $\frac{(1-\alpha)T_s}{maxstep}$ corresponds to the point when number of process equals 1 in figure 1, which is 0.288. When there are n processes, the time per step equals approximately $\frac{0.288}{n}$.

Another question we need to explore is how performance is affected by problem size. We change the size of the grid to see whether execution time changes. As is shown in figure 2, the



(a) Average time per step versus number of processes



(b) Speedup versus number of processes

Figure 1: Performance versus number of processes

execution time increases as the size of grid increase. As the grid is a square, so the relation between them can be described as $T_L = (\frac{L}{432})^2 T_{432}$.

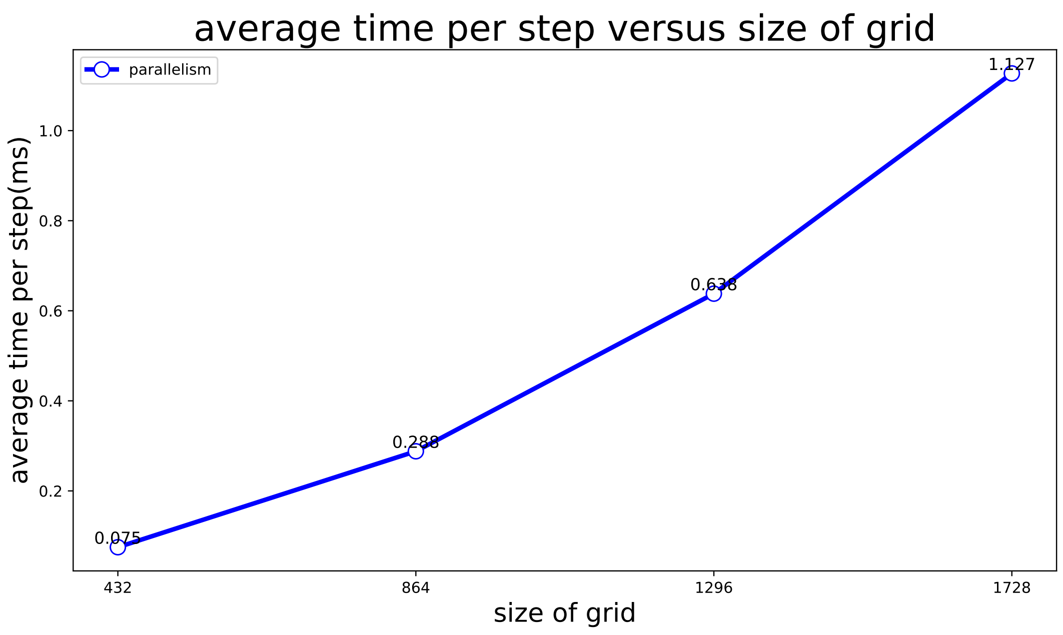


Figure 2: Performance versus size of grid

IMPROVEMENT OF PARALLEL PROGRAM

Although the parallel program proves to more efficient than the serial code, many aspects can still be improved.

when to stop

The calculation only stops after doing a fixed number of steps. However, the code can stop as long as there are no changes. So this part could be changed by adding a block of judgement code.

flexible problem size and number of processes

The size of problem and number of processes are fixed in a program called percolate.h. We can make it flexible by making the problem size and number of processes parameter as inputs.

overlap communication and calculation

In the case study example, communication between halos is implemented by MPI_Sendrecv function, which will block until the sending message has been sent and received message has been received. However, the calculation only requires the halos received but does not need the halos to be sent, which means that waiting for the sending message to be sent before the calculation is a waste of time. Therefore, we could use non-blocking send and receive before the calculation and continue to execute the calculation block. Before updating the old map, we use MPI_Wait to wait for the sent message to be received. The pseudo code can be expressed as algorithm 1.

Algorithm 1 Overlap communication and calculation

```
1: while Grid changes do
2:     MPI_Issend(...) MPI_Irecv(...) non-blocking send and receive.
3:     Do calculation.
4:     MPI_Wait(...) block until the sent message is received.
5:     Update the old map.
```

derived data types

In the case study code, the original map is distributed across all the processes to small map by collective communication pattern MPI_Scatter. And then copy small map to old map. Actually, we do not necessarily need it. Instead, defining a derived data type is a better way of distribution. MPI_Type_Vector is the best MPI routine in this problem. So we define a M*N subsection of the original map using MPI_Type_Vector. Process 0 is responsible for distributing the original map across all the processes directly to old map using MPI_Ssend. The message is a M*N subsection array, the data type of which is the defined derived data type. What's more, we can also define a derived data type for halos. Halos of each process are the squares surrounding its small map. So we can define a struct datatype using MPI_Type_create_struct which contains the 1-Dimensional arrays representing the top and bottom surrounding halos. So we can send the new struct datatype to swap halos between processes.

2-dimensional decomposition

Decomposition of the original map in the case study is 1-Dimensional decomposition. However, we can move a step further to use 2-Dimensional decomposition. To achieve this goal, we only need to change the size of old map because previously we define a derived data type which is M*N subsection array. In terms of the struct data type representing halos, we need to modify the data type by adding halos in the left and right side of each rectangle.

uneven distribution

In the case study, the distribution is even, however, if problem size is not an exact multiple of the number of processes, bug exists. So we need to take uneven distribution into consideration. In order to solve this, we can ignore the last process temporarily, and distribute evenly across the other (P-1) processes and then fill the last process. The details of how to implement this will be included in part 2.

topology

If we use 2-Dimensional decomposition, it would be better to use Cartesian Topology to connect processes using MPI_Cart_create. There are many benefits of using topology. Firstly, it's easier to get the ranks of neighbour processes for each process using MPI_Cart_shift. Another benefit is that it's similar to 2-D decomposition in geometry so we can get the coordinate of each process using MPI_Cart_coords to distribute the original map to processes more easily.

TEST PLAN

Test plan of the improved version is basically the same as the plan in the case study described above which includes correctness and performance test. But the difference is that we need to add comparison between the case study and the improved code. What we want to test is whether the improved code is correct especially in some bug cases in the case study and whether the performance improves. What's more, how number of processes and problem size affect the performance will also be tested using exactly the same test cases in the test of case study.

PROGRAM STRUCTURE

The whole structure of program is presented in algorithm 2.

Algorithm 2 Program Structure

```
 1: Create Cartesian Topology and create derived data type vector for map and struct for
    halos.
 2: Controller process does 2-D decomposition to distribute original map directly to old map
    in each process.
 3: while Grid changes do
 4:     MPI_Issend(...) MPI_Irecv(...) non-blocking send and receive.
 5:     Do calculation.
 6:     MPI_Wait(...) block until the sent message is received.
 7:     Update the old map.
 8: Each process sends old map to controller process.
 9: Controller process judges whether the cluster percolates.
10: return
```