Chinese University of Hong Kong, Shenzhen

CSC4005

Report 1:
A Comprehensive Implementation and Analysis of Odd-Even Transposition Sort Implemented by MPI

Submitted by:

Changwen LI

118010134

Submission date:

12th of October 2021

Table of Contents

A Comprehensive Implementation and Analysis of Odd-Even Transposition Sort Implemented by MPI

1. Introduction:

This assignment requires students to implement odd-even transposition sort algorithm by MPI (Message Passing Interface). The report consists of 4 parts. In the first part, main concepts are introduced. The second part introduces the program architecture and the procedures to execute the program. In the third part, the analysis part, we test the programs with different number of cores and different length of data. In the fourth part, which is the discussion part, we compare odd-even transposition algorithm with odd-even merge algorithm.

1.1. Introduction to odd-even transposition sort

Odd-even transposition sort is a sorting algorithm which makes use of parallel computer architecture. The odd-even transposition sort is based on bubble sort which swaps two consecutive numbers if they latter one is larger than the former one (assuming the list of numbers is expected to be in ascending order). The algorithm consists of 2 phases. In the odd phase, the number with odd index compare with the next number with even index. Swap them if the odd index number is greater. In the even phase, the number with even index compare with the next number with odd index. Swap them if the even index number is greater. When the number of process n is less than the number of number m, each process will be assigned to m/n numbers. In each process, the numbers will be sorted in the way mentioned above. The leftover number (which is boundary number) will check with the neighboring left-out number and swap them to follow the ascending order. An example is illustrated in figure 1.1.1.

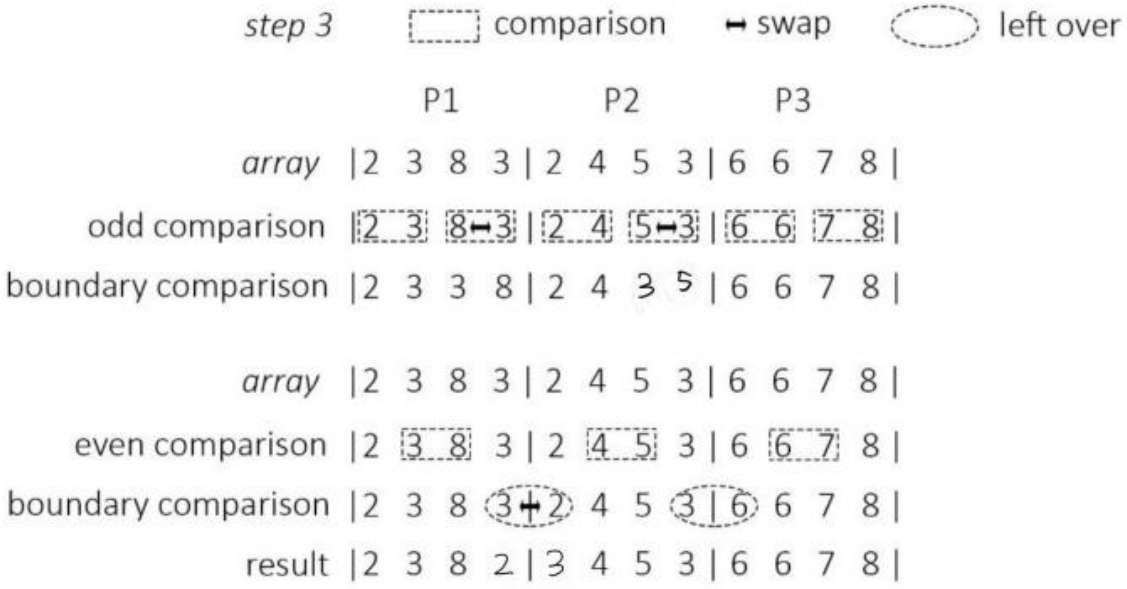

Figure 1.1.1: An example of odd-even transposition sort (cited from previous guideline)

1.2. Introduction to MPI

MPI (Message Passing Interface) is a protocol for passing messages between processes which is widely used in information transferring in parallel and distributed computing. In this project, the

block point-to-point communication is used. The blocking sending, receiving, broadcasting and other information passing schemes provided by MPI are used.

2. Methods

2.1. Program implementation

First, scatter the original data with size m into n processes. Each process will receive $\lfloor \frac{m}{n} \rfloor$ numbers.

Then all processes will be executed in m phases (the meaning of phase here is similar to the meaning of iteration). This part of the codes is attached in figure 2.1.1. Then, 2 situations are under consideration.

The first situation is that the number of data in each process is even. When the phase number is even, the number with even index will swap with the next number with odd index (please do notice that the index refers to global index in original array). Only the number in local array (i.e. within the process) will be swapped in this stage. This is achieved through direct swap value in an array. When the phase number is odd, the number with odd index will swap with the next number with even index. After that, the neighbouring leftover will swap to ensure the ascending order. The neighbouring leftover number is swapped is achieved through MPI_Send and MPI_Recv. This part of the codes is attached in figure 2.1.2. For process with even rank, send the leftover number to the next process, and then receive number from the next process. For process with odd rank, receive the number from the former process, and then send the number to the next process.

```
Element *recv_buffer = new Element[1];
Element *send_buffer = new Element[1];
Element *local_data = new Element[len_proc];
MPI_Scatter(begin, len_proc*2, MPI_UNSIGNED, local_data, len_proc*2, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
```

Figure 2.1.1: Screenshot of codes about MPI_Scatter

```
if (len_proc%2==0 && phase%2==1){
    if (rank==0){
        if (right_proc!=-1){
            send_buffer[0] = local_data[len_proc-1];
            MPI_Send(send_buffer, 2, MPI_UNSIGNED, right_proc, 0, MPI_COMM_WORLD);
            MPI_Recv(recv_buffer, 2, MPI_UNSIGNED, right_proc, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            if (local_data[len_proc-1] > recv_buffer[0]){
                local_data[len_proc-1] = recv_buffer[0];
            }
        }
    }
```

Figure 2.1.1: Screenshot of codes about MPI_Send and MPI_Recv

The second situation is when the number of data in each process is odd. When the phase number is even, the number with even index will swap with the next number with odd index. After that, the neighbouring leftover will swap to ensure the ascending order. When the phase number is odd, the number with odd index will swap with the next number with even index. After that, the neighbouring leftover will swap to ensure the ascending order. Finally, gather the data from each process. Notice

that here some numbers might be left out if number m cannot be divided with n with no remainder. The remainder number and the sorted array will then be merged into the original array. A flow chart is drawn to demonstrate the basic architecture of this program in figure 2.1.3.

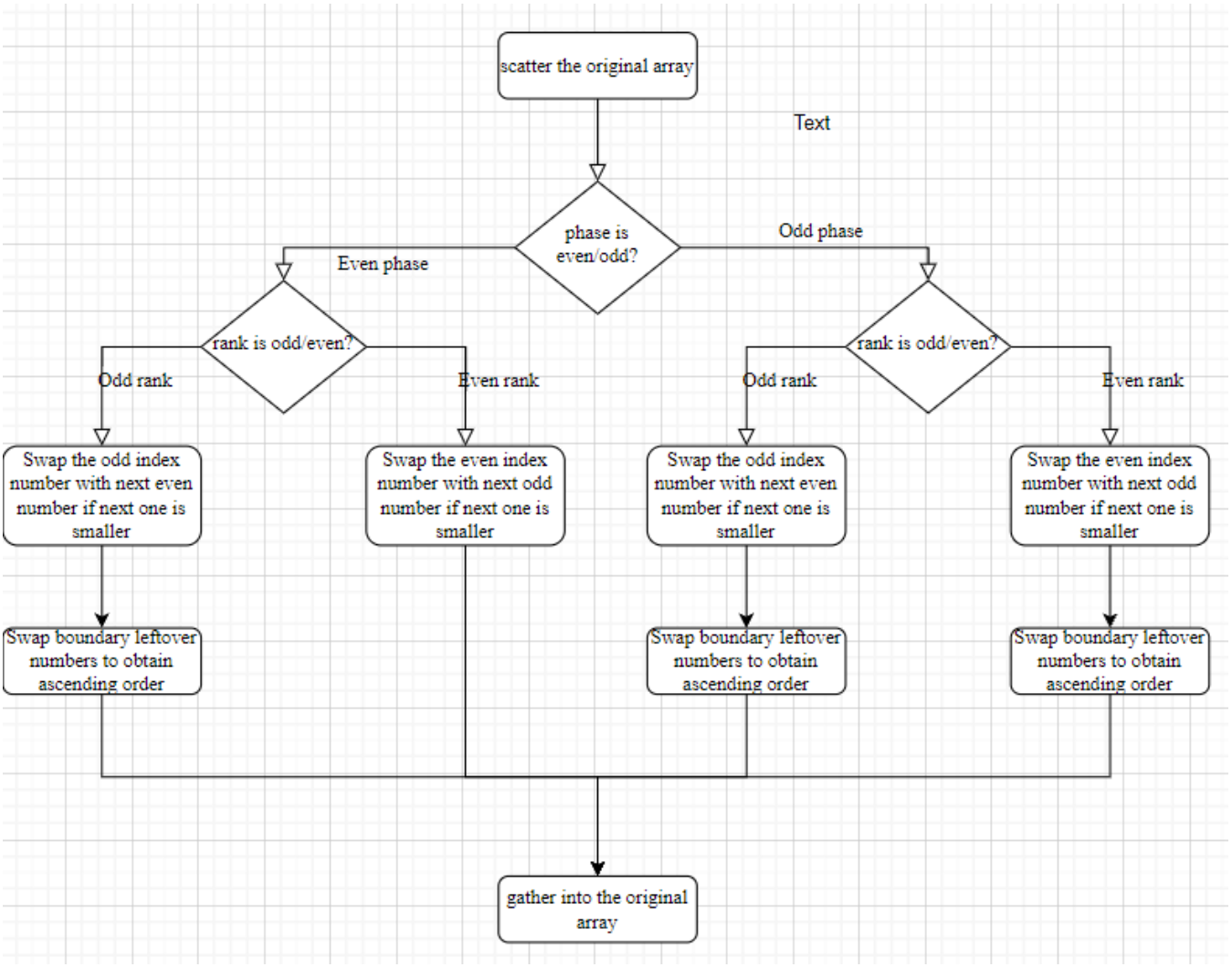

Figure 2.1.1: Flowchart of basic programming architecture

2.2. Procedures to execute the program

The project is based on templates provided by TAs and it is already compiled. There are 2 executables to execute. Both of them are in build directory under the root of my submitted project. The first one is main. This execute takes a file as input, and sort the number in the input file, and then output an output file with sorted number. The gtest_sort tests if the project gives the correct results. The method to execute the project is as same as the methods provided by TAs.

If you are using a localhost, the command to run the main is $mpirun -np num_of_process main path_of_input_files path_of_output_files$. To execute the gtest_sort executable, simply use command $mpirun -np num_of_process gtest_sort$.

If you are using Slurm, then command approach is not the best approach to execute them on Slurm. The recommended one is using batch approach. There are 2 batch files in the build folder (run_main.sh and run_gtest_sort.sh). Modify the parameters in the batch file to suit your requirements. The number after ntasks indicates the number of processors, the number after nodes

indicates the number of nodes you want. You can also modify the information after output to denote the name of the file storing the printed information. The command starts with “mpirun” is the same as the commands in localhost (except there is no need to state the number of processes). Modify it if it is necessary for you. An example of batch file is shown in figure 2.2.1. After modifying the batch file, use command $sbatch batch_name.sh$ to run the batch files (You may need to wait in queue).

```
#!/bin/bash
#SBATCH --account=csc4005
#SBATCH --partition=debug
#SBATCH --qos=normal
#SBATCH --ntasks=10
#SBATCH --nodes=1
#SBATCH --output test_result_10_100000.out
#SBATCH --time=10

echo "mainmode: " && /bin/hostname
mpirun /pvfsmnt/118010134/assignment/csc4005-assign
ment-1/build/main /pvfsmnt/118010134/assignment/csc
4005-assignment-1/input_data_100000.txt /pvfsmnt/11
8010134/csc4005-assignment-1/output_data.txt
~
~
~
~
~
~
```

Figure 2.2.1: An example of batch file for main executable

2.3. Methods of experiments and data collection

The project tests the performance of the algorithm with different number of processes and different length of data. There are mainly three set of experiments conducted in this project.

In the first set of experiment, the performance of different data size with the same number of processes is tested. The fixed number of processes is 32, with data size of 10000 numbers, 20000 numbers, 30000 numbers, 40000 numbers, 50000 numbers, 60000 numbers, 70000 numbers, 80000 numbers, 90000 numbers and 100000 numbers. In the second set of experiment, different number of cores are used to test on same size of data. 10 processes, 20 processes, 30 processes, 40 processes and 50 processes are used to test on 100000 numbers. In the third set of experiment, parallel computing and sequential computing are being compared. 100000 numbers are sorted by 1 core and 10 cores.

All the data used in the experiment are generated by a program written by me. To test these data, use main executable to input the test data, and the file to store print out information is mentioned above.

3. Analysis

3.1. Algorithm analysis:

Assume that the data size is m (i.e. there are m numbers), and the assigned process is n. The time complexity for the odd-even transposition algorithm is $O(\frac{m^2}{n})$. The odd-even transposition algorithm

is merely a bubble sort when it is run in sequential architecture. Thus, the time complexity is O(m^2). The parallel version is to distribute the algorithm into several parts, and execute them together simultaneously. Thus, the time complexity for the odd-even transposition algorithm is O($\frac{m^2}{n}$).

3.2. Experiment result analysis

3.2.1. Performance of different data size with the same number of processes

In this experiment set, the number of processes is assigned to be a fixed number. The number of assigned processes is 32. Then, test data with 10000 numbers, 20000 numbers, 30000 numbers, 40000 numbers, 50000 numbers, 60000 numbers, 70000 numbers, 80000 numbers, 90000 numbers and 100000 numbers are tested. The raw data is shown in figure 3.2.1.1. According to the time complexity of this algorithm, when the number of cores is fixed, the time complexity should be O(m^2). The graph shown below basically fits this time complexity. For example, when data size is 20000, the execution time is 1.46s. When data size is 40000, the execution time is 6.07s, which is roughly 4 times of the execution time when the data size is 20000. The experiment results fit the algorithm time complexity.

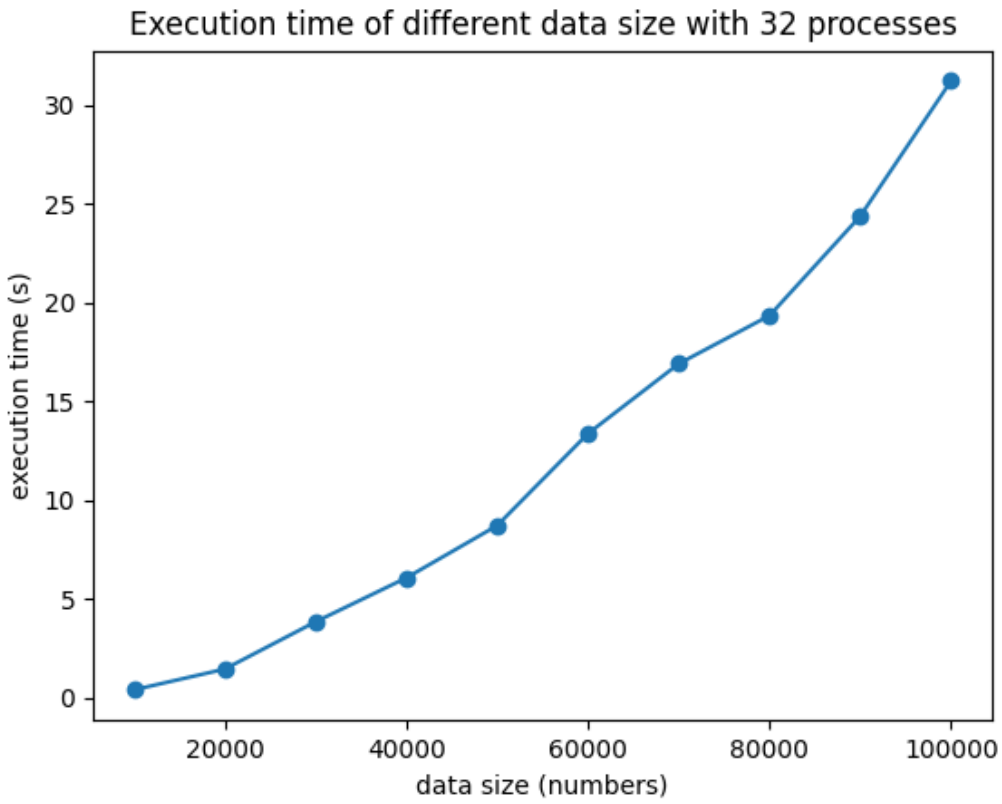

Figure 3.2.1.1: Execution time of different data size with 32 processes

3.2.2. Performance of different number of processes with the same data size

In this experiment set, the data size is assigned to be a fixed number. The data size is 100000. Then, different number of cores are used to test on same size of data. 10 processes, 20 processes, 30 processes, 40 processes and 50 processes are used to test on 100000 numbers. The raw data is shown in figure 3.2.2.1. The figure 3.2.2.1 demonstrates strange result when the number of processes

increases from 30 to 40. It is known that the program should run faster when there are more cores used (assume the data size is the same). However, it is not the case when the number of processes increases from 30 to 40. A possible explanation derive from the architecture of the nodes. The GPU used in this experiment only consists of 32 cores. Thus, when there are more cores, more than one node will be used. The data then need to transform from one node to another node. If the number of cores is smaller than 32, data will only transfer within a node. The transfer speed between nodes is surely much slower than the transfer speed within a node. This can be proven by viewing the data throughput. When the process number is 20, the throughput is 2.68058e-05 GB per second. However, when the process number is 40, the throughput is 2.34841e-05 GB per second. The throughput decreases when more nodes are used.

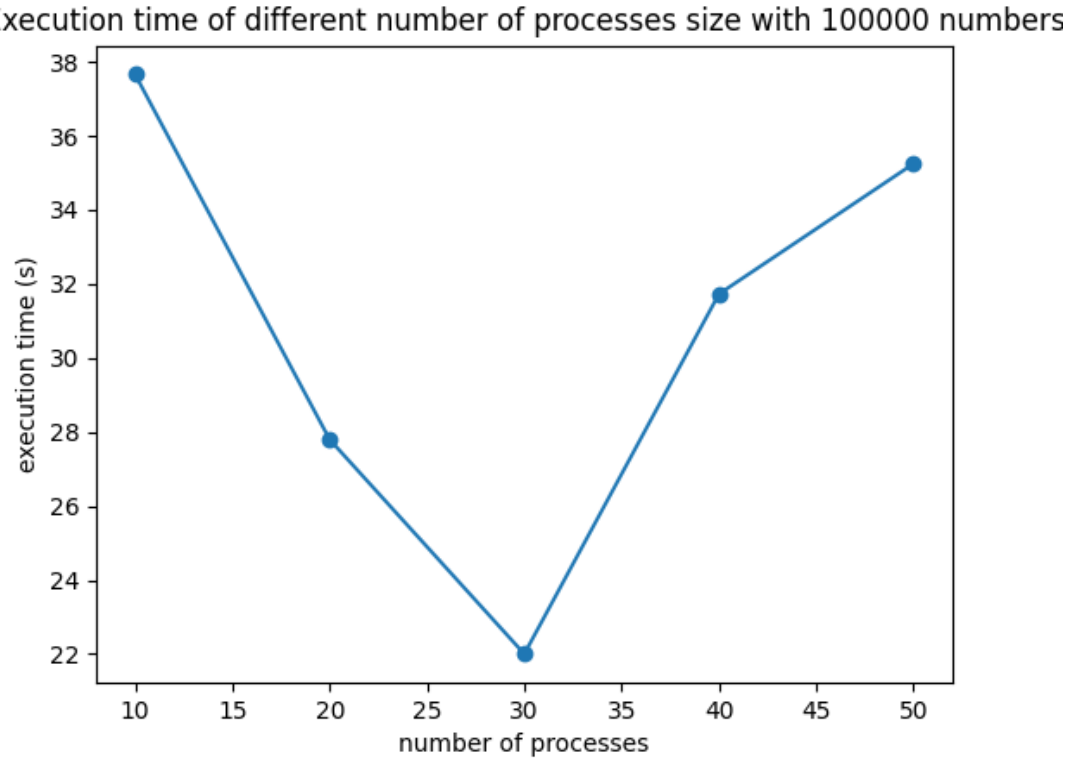

Figure 3.2.2.1: Execution time of different number of processes size with 100000 numbers

The time complexity of the algorithm should be $O(\frac{1}{n})$. To eliminate the influence of the communication time between different nodes, another experiment is conducted. The number of processes is chosen to be 10, 15, 20, 25. Figure 3.2.2.2 demonstrates the experiment result. The experiment results fit the algorithm time complexity.

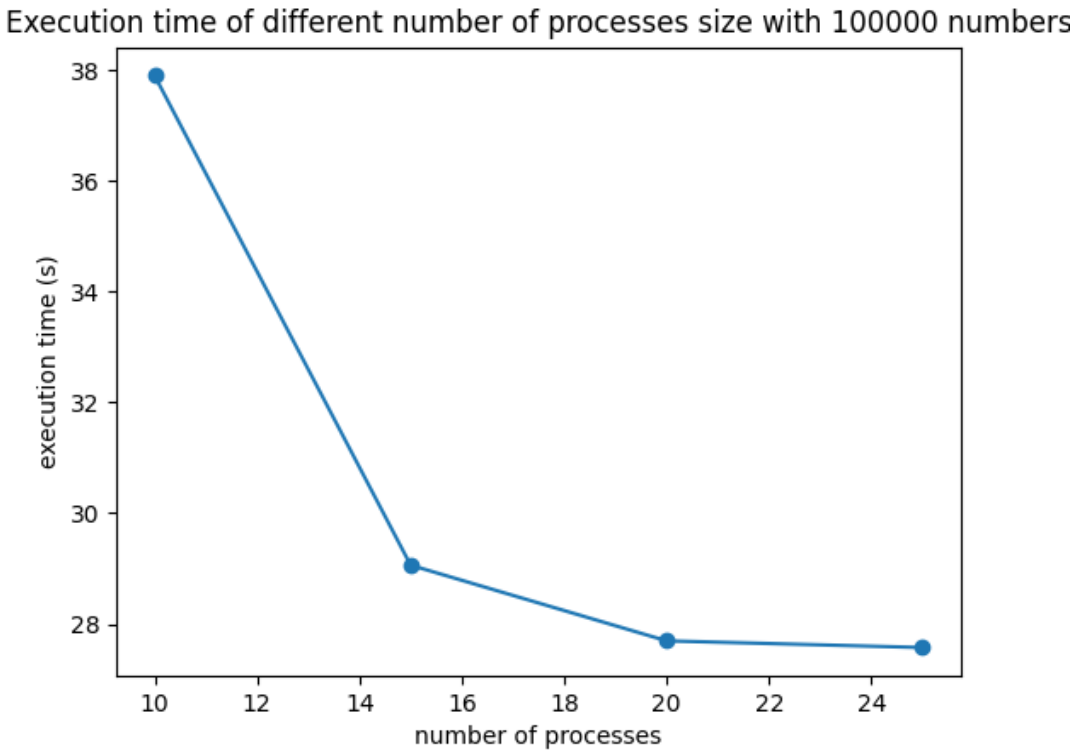

Figure 3.2.2.2: Execution time of different number of processes size with 100000 numbers

3.2.3. Performance comparison between sequential programming and parallel programming
If the program is executed in sequential way, we only need to consider it as a parallel program running with only 1 process. Thus, it will be easy to do the experiment. The sequential test is configured with 1 process and 100000 data. The parallel test is configured with 10 process and 100000 numbers. The execution time of the sequential test is 284.231s, the execution time of parallel test is 37.896s. The execution time for sequential program is 7.5 times of the parallel one. The reason why execution time for sequential program is less than 10 times of the parallel one is perhaps the MPI instruction takes more time than the common sequential instruction since most of the MPI instructions used in this project is bocking communication.

4. Discussion

In discussion, the odd-even merge sort is also implemented. The code id attached in the "src" directory, with name "odd-even-merge-sort.cpp". The performance of odd-even merge sort and odd-even transposition sort is compared. The odd-even merge sort also separates the original data into several fractions. In each fraction, each process will sort the data locally using quicksort. In odd phase, the odd rank process will merge with the next even rank process, then put the smaller half to the smaller rank of process and put the bigger half to the larger rank process. Similar procedure for the even phase. The time complexity for this algorithm is O($\left(\frac{m}{n} log \frac{m}{n} + 2\frac{m}{n}\right) n$), which is O(m $log \frac{m}{n}$) when the number of process *n* is much smaller than the data length *m.* The performance comparison can be verified by experiment. Use 4 cores, 100000 numbers to execute both algorithms. The execution time for odd-even transposition algorithm is around 50.995s. While for the odd-even

merge sort, the execution time is 0.117s. The performance of odd-even transposition sort is significantly worse than odd-even merge sort.

5. Conclusion

The project implements odd-even transposition sort. The report has analyzed the performance of it with different configurations. Generally speaking, the experimental results fit the theoretical analysis. Besides the odd-even transposition sort, odd-even merge sort is implemented as well. The comparison between them can be seen in discussion.