香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

CSC4005 Distributed and Parallel Computing - Assignment 1 Report

WEN Zhenduo

118010323

zhenduowen@link.cuhk.edu.cn

Contents

1 INTRODUCTION

This assignment report aims to realize and analysis the Odd-Even Transportation Sort based on MPI(Message Passing Interface) Standards. It also tries, via the implementation of Odd-Even Transportation Sort, to test and conclude certain phenomena in parallel computing taught in lectures. Besides, certain technical details will be discussed. In all, this report is both epagogic and heuristic in terms of parallel computing.

This introductory section provides a brief overview of Odd-Even Transportation Sort and MPI Standards. It also offers a preliminary knowledge of the essence of the sorting algorithm and how MPI communicates between processes.

1.1 OVERVIEW OF ODD-EVEN TRANSPORTATION SORT

Odd-Even Transportation Sort is a sorting algorithm based on the Bubble Sort Technique: it compares two adjacent terms and switches them is the first term is greater than the second term, so that an ascending order is maintained. The opposite case applies for the descending order sorting. As its name suggests, Odd-Even Transportation Sort operates in two different phases: Odd Phase and Even Phase. In both the phases, processes exchange numbers with their adjacent processes (See Figure 1):

Essentially, this sorting algorithm is the aggregation of a series of bubble sorts. The data elements are ”bubbling” in local arrays and if they emerge to the top or the bottom of the local array, they are sent and compared with adjacent local arrays. So we may derive the theoretical time complexity when the input data size is significantly larger than the number of processes:

$$\text{Theoretical Time Complexity} = \frac{\text{Sequential Sorting Time Complexity}}{\text{Number of Processes}} = \frac{\Theta(n^2)}{m} = \Theta(n^2)$$

The optimal theoretical time complexity is (when the input data size is near the number of processes):

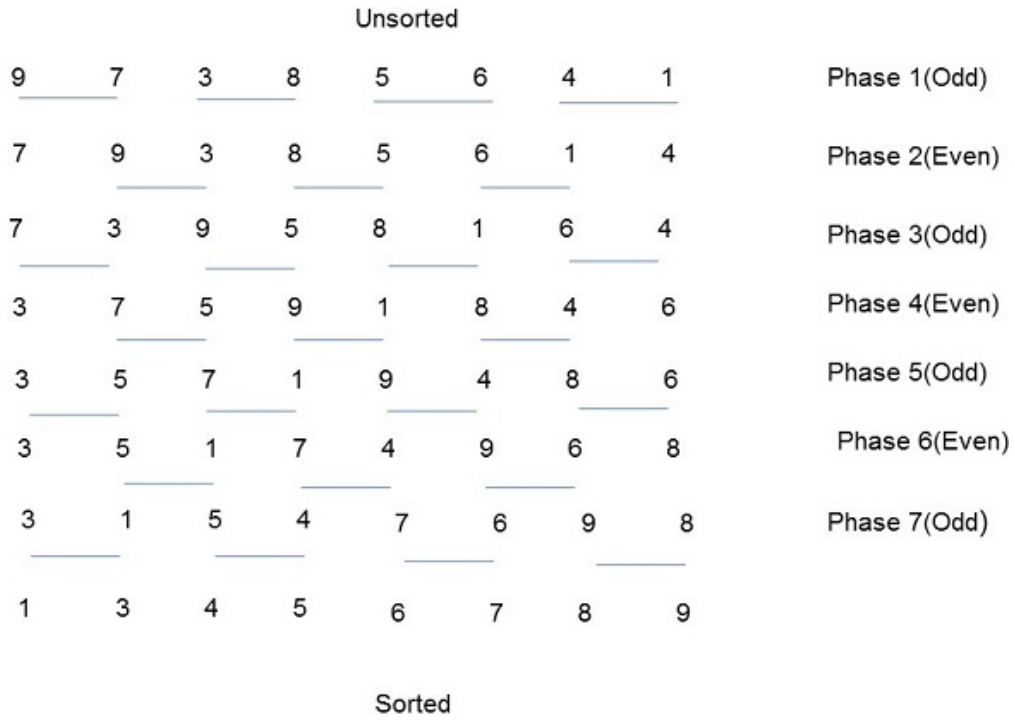

FIGURE 1: SIMPLE ILLUSTRATION OF ODD-EVEN TRANPOSRTATION SORT

$$\text{Theoretical Time Complexity} = \frac{\text{Sequential Sorting Time Complexity}}{\text{Number of Processes}} = \frac{\Theta(n^2)}{c \times n} = \Theta(n)$$

However, the theoretical formula for parallel program time complexity neglects the cost of communications among processes. In later Result Analysis section, this report will discuss computation and communication separately.

1.2 BRIEF INTRODUCTION OF MPI

Message Passing Interface (MPI) is a series of standards designed to function on parallel computing architectures. Both point-to-point and collective communication are supported. Some basic concepts of MPI are:

- Communicator: Objects connect groups of processes in the MPI session. Each Communicator gives inner processes an independent identifier called rank and arranges its contained process in an ordered sequence.
- Point-to-point Communication: A series of significant MPI functions involve

communication between two specific processes. This report mainly involves blocking send and receive **MPI_Send()** and **MPI_Recv()**.

- Collective Basics: Collective function involve communication among all processes in a communicator. This report utilizes Broadcasting **MPI_Bcast()**, Data Scattering **MPI_Scatter()**, and Data Gathering **MPI_Gather()**.

- Derived Data Types: Most MPI functions require specified data type defined in MPI. This is because MPI as an abstract standard aims to support heterogeneous environments where types might be represented differently. MPI predefines constants **MPI_INT, MPI_CHAR, MPI_DOUBLE** corresponding to **int,char,double**.

The sorting algorithm in this report is implemented on MPICH, a popular c++ realization of MPI Standards.

2 METHODOLOGY

This section covers the technical implementation details of the assignment, including the general Program Flow, the Data Structure Design, and some Special Cases involve boundary specialization and exceptional case.

2.1 PROGRAM FLOW

Figure 2 shows the overall view of the program flow.

From left to right, there are mainly three steps designed in the program:

- Initialization: The program goes into the main function on the root process, reads in the data from given file path, initialize variables including random array generation function's configuration, the storage used to contain global and local array, their lengths and other information including space allocation, specifying communicator, and get the rank and processor number. Then **MPI_Scatter** distributes the input array into the local array on each process. The segmentation order is based on the order of process ranks.

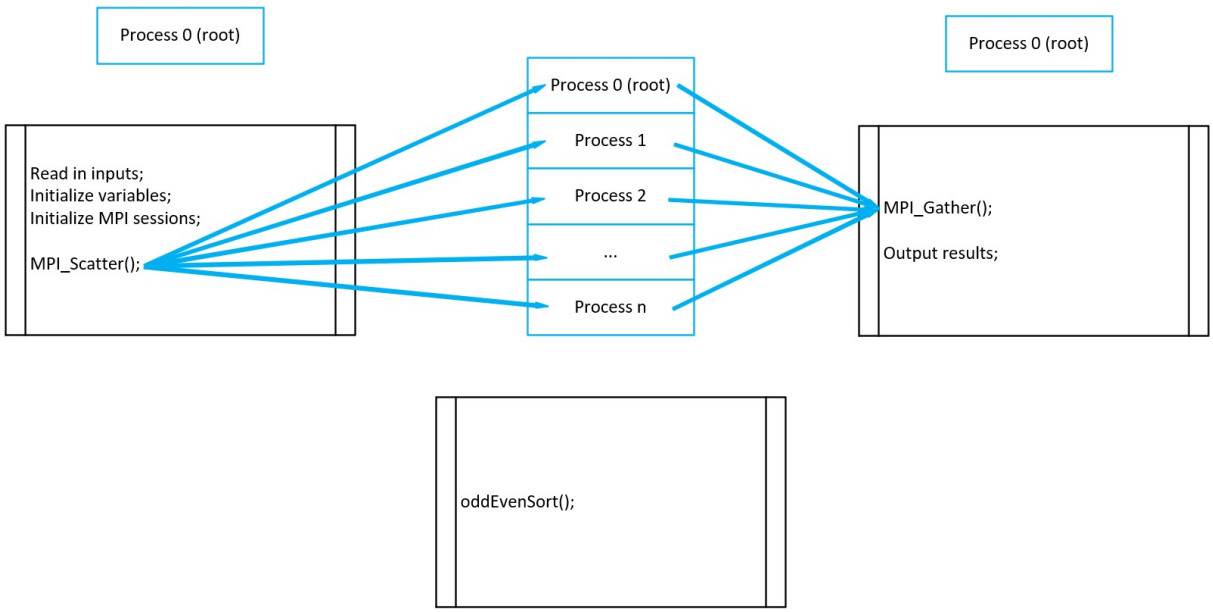

FIGURE 2: GENERAL PROGRAM FLOW

- Odd-Even Sort: p processes conduct the sorting algorithm together, the bubbling process happens in both inner-process and inter-process. The sorting function should not only implement the comparison algorithm, but also **MPI_Send** and **MPI_Recv** so that the processes can communicate with each other. Further details are shown in the **Parallel Data Structure Design** Section.

- The root process calls **MPI_Gather()**, all other processes return sorted array to the root. The program operation information then outputs.

2.2 PARALLEL DATA STRUCTURE DESIGN

Figure 3 shows the parallel data structure design, that is, how data are stored and operated by each process.

A local array is created on heap to contain the numbers. After receive the scattered array from the root process, the process will conduct local bubbling operation. And:

- Odd rank processes will send its smallest element to the right process (process with rank lesser exactly 1). The element pair will be switched if the former is smaller than the latter.

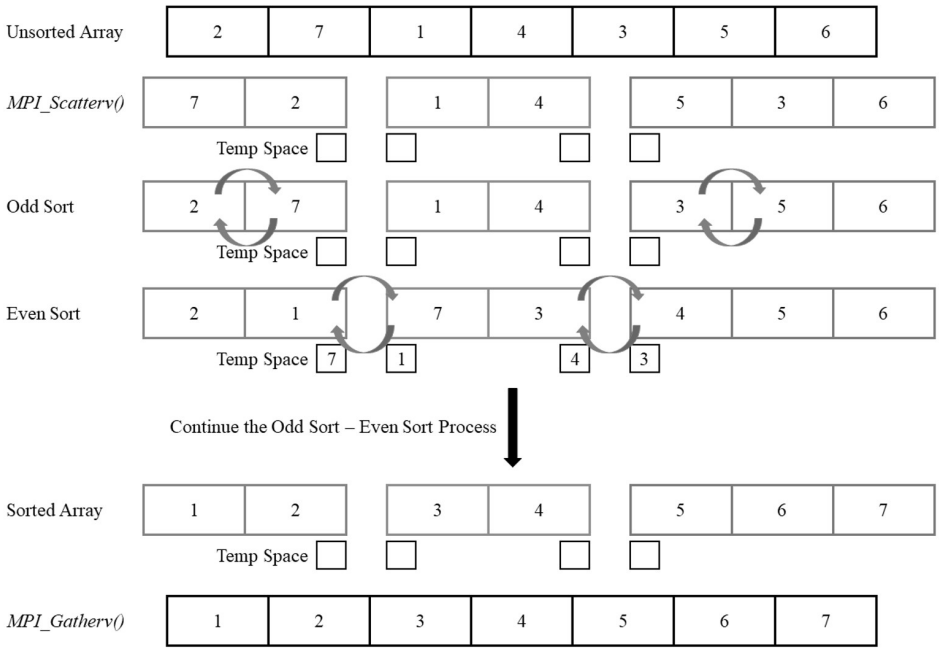

FIGURE 3: PARALLEL DATA STRUCTURE DESIGN

- Even rank processes (except the root process) will send its smallest element to the right process (process with rank lesser exactly 1). The element pair will be switched if the former is smaller than the latter.

The loop will continue until meets the stopping criteria, which is covered in the following subsections.

2.3 SEQUENTIAL IMPLEMENTATION IN C++

The following code block is the sequential implementation of Odd-Even Sort in C++:

```
// Sequential Odd-Even Sort
bool is_sorted = false;
while (!is_sorted){
    is_sorted = true;

    // Odd-Phase
    for (int i = 1; i <= global_length - 2; i = i + 2){
```

```
            if (begin[i] > begin[i+1]){
                std::swap(begin[i], begin[i+1]);
                is_sorted = false;
            }
        }
        // Even-Phase
        for (int i = 0; i <= global_length - 2; i = i + 2){
            if (begin[i] > begin[i+1]){
                std::swap(begin[i], begin[i+1]);
                is_sorted = false;
            }
        }
}
```

The stopping criteria used in this sequential implementation of Odd-Even Sort is the bool flag variable **is_sorted**. If **is_sorted** is true after the bubbling process of odd phase and even phase, then no switch happens in this iteration, which means the input array is sorted.

2.4 PARALLEL IMPLEMENTATION BY MPICH

The same flag variable technique can be easily transplanted onto the parallel version implementation. We add a new variable called **is_global_sorted** which is the mathematical aggregation of **is_sorted** on every process. If **is_global_sorted** is greater than zero, then some processes' local array has not been sorted yet, the loop then continues. The drawback is that processes have to frequently communicate with each other in each iteration such that this idea leads to higher communication cost.

But we can use a simple way to reduce the communication sort based on the rationale of bubbling sort. The idea is to focus on the outer loop of bubbling technique. Recall that the outer loop of bubbling sort iterates by the size of input data (to "bubble" each element up). Naturally

we reach a statement:

> After n iterations of the outer loop in bubbling, the original array must be sorted.

where n is the input data size. The proof is simple: Suppose there are some unsorted elements in the array after n iterations, since the outer loop has already looped n times, n elements has been sorted. The existence of these unsorted elements indicate that the size of input array is greater than n, which is a contradiction.

The same statement stands in the parallel case. Essentially the Odd-Even Bubbling Sort is a series of processes doing bubbling sort together. So after n outer iterations, the array must be sorted. This stopping criteria gives a $O(n)$ upper bound of the program.

We can further extend this statement: If there are in total p processes and each process is already locally sorted. Then after p loops of the outer iteration (p times of mutually communication), the original array must be sorted. This means that if we used a local sorting algorithm with time complexity $O(nlogn)$, the overall time complexity is $O(pnlogn) = O(nlogn)$ when p is significantly smaller than n. The only problem is that this sorting algorithm is no longer the Odd-Even Transportation Sort.

The inner bubbling process is the same with the sequential version of implementation. Here we focus on the inter-process communication among processes (We only show the odd phase communication here since the even phase situation is similar):

Sender side:

```
// Odd-Phase Communication
local_buff = local_array[0];
MPI_Send(&local_buff, 1, MPI_LONG, rank - 1, 0, MPI_COMM_WORLD);
MPI_Recv(&local_buff, 1, MPI_LONG, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
if (local_buff > local_array[0]){
    std::swap(local_buff, local_array[0]);
}
```

Receiver side:

```
if (rank != num_of_proc - 1){
    MPI_Recv(&local_buff, 1, MPI_LONG, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    if (local_buff < local_array[local_length - 1]){
        std::swap(local_buff, local_array[local_length - 1]);
    }
    MPI_Send(&local_buff, 1, MPI_LONG, rank + 1, 0, MPI_COMM_WORLD);
}
```

The flow of point-to-point communication is illustrated in Figure 4:

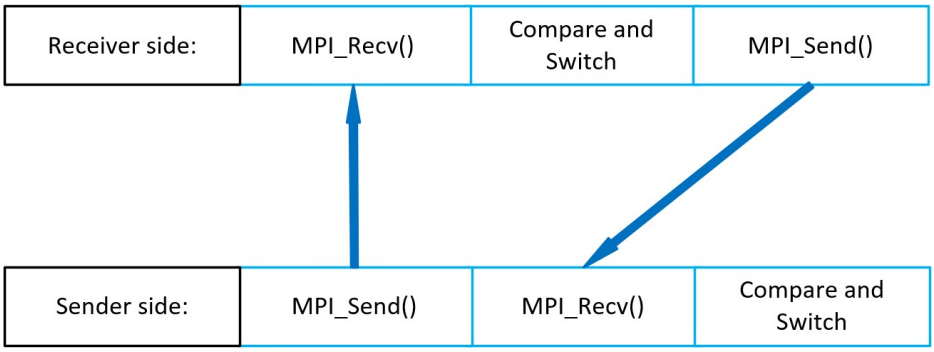

FIGURE 4: POINT-TO-POINT COMMUNICATION FLOW

2.5 ROBUSTNESS ON SPECIAL CASES

There are some special cases that require special treatment:

- when the input data size is smaller than the number of processes
- when the input data size cannot be divided equally by the number of processes

This report proposes two strategies to achieve robustness in the above two scenarios:

- Augmentation: We add certain among of element whose value is the maximum of **int_64** such that the augmented input data size is a multiple of the number of processes. The augmented data then can be perfectly distributed to each process. In the end, remove the augmented elements before output.

- Residualization: When the input data size cannot be equally divided, left the residuals at the end. When other data items are sorted and returned, insert the residuals back to the original array according to their value. When the input data size is smaller than the number of processes, conduct the sort on the root process only.

The first strategy is easier to implement. However, we test that it has slightly poorer performance than the second strategy. The final robustness implementation is based on the second strategy.

3 RESULT EXPERIMENT AND ANALYSIS

This section covers a series of experiment on Running Time, Parallel Speed-Up and Efficiency.

3.1 EXPERIMENT ON RUNNING TIME AND REBOUND EFFECT

The Experiment Design Information:

- Data Size: 20K, 40K, 80K, 160K, 320K
- Process Number: 1, 2, 4, 8, 16, 32
- Running Time count in nanoseconds

Figure 5 shows the running time versus the number of processes on different data scale. Generally, despite the length of the input data size, increasing the number of threads can reduce the running time. However, when the amount of processes is larger than 16, the decrease in running time is limited.

Figure 6 is the scaled version of Figure 5, focusing on the smaller data size in range 20K to 80K. We see an increase in running time. Although there might be some fluctuations in the running time, the tendency of increased running time is verified on all three experiments.

The reason behind this **rebound effect** is the communication overhead. When we apply more processes into the program, the communication costs increased linearly: our

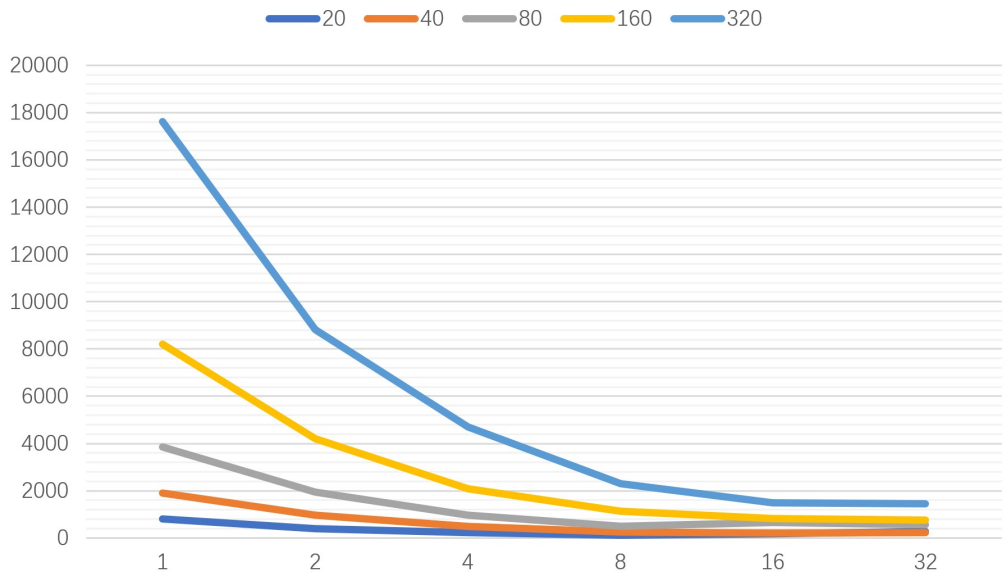

FIGURE 5: RUNNING TIME VERSUS PROCESSES NUMBER

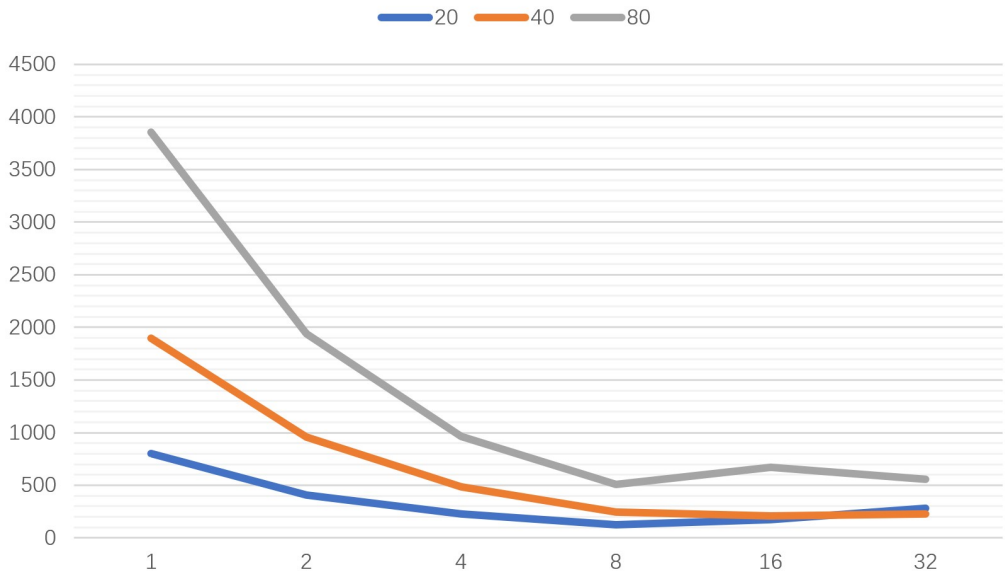

FIGURE 6: COMMUNICATION OVERHEAD

communication design shown previously in Figure 4 indicates that doubling the number of processes results in doubling times of inter-process communication. If we increase the process number to 64 or 128, the rebound effect will be even more serious. We study further on this rebound phenomena in terms of the speedup and efficiency formula.

3.2 CHECK SPEED-UP AND EFFICIENCY ON REBOUND EFFECT

The speed-up factor is calculated by:

$$\text{Speed-Up} = 1 - \frac{1}{\text{SerialTime} + \frac{\text{ParallelTime}}{\text{p}}}$$

Figure 7 shows the speed-up factor versus process number on different data size from 200K to 3200K (for processes number greater than 8, the result has larger variance, so the information showed here is the average of 3 identical experiments' result): For smaller scale of data size

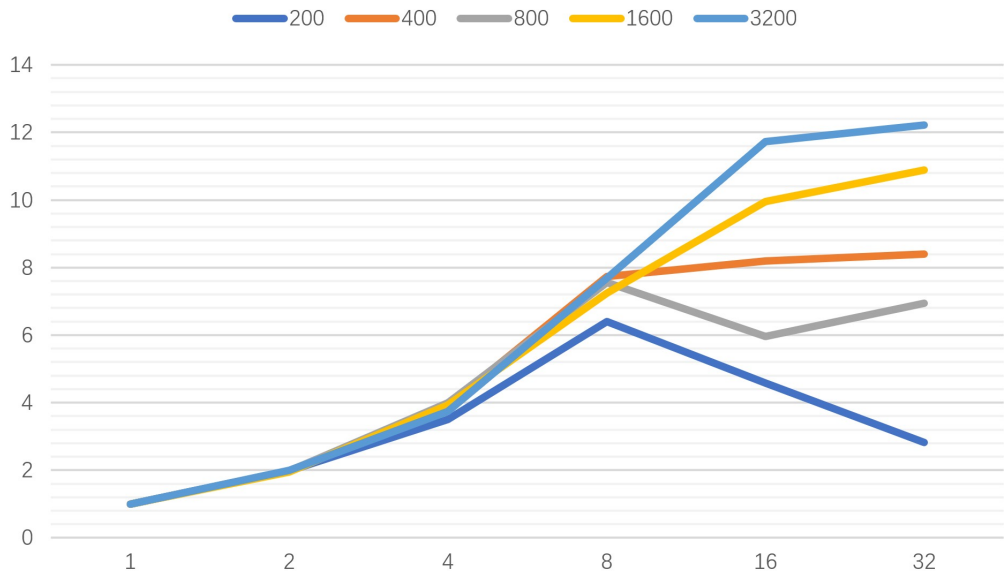

FIGURE 7: SPEED-UP VERSUS PROCESSES NUMBER

(200K to 800K), the speed-up decreases after processes number reaches near 8 to 16. Yet if the scale is stiller larger (1600K or 3200K), the speed-up slightly increases after 16 processes.

The parallel efficiency is calculated by:

$$\text{Efficiency} = \frac{\text{SerialTime}}{\text{p} \times \text{ParallelTime}} \times 100\%$$

Figure 8 shows the efficiency versus process number on different data size from 200K to 3200K. In the experiment of efficiency we see the similar case. The efficiency drops fast after the process number reaches 8.

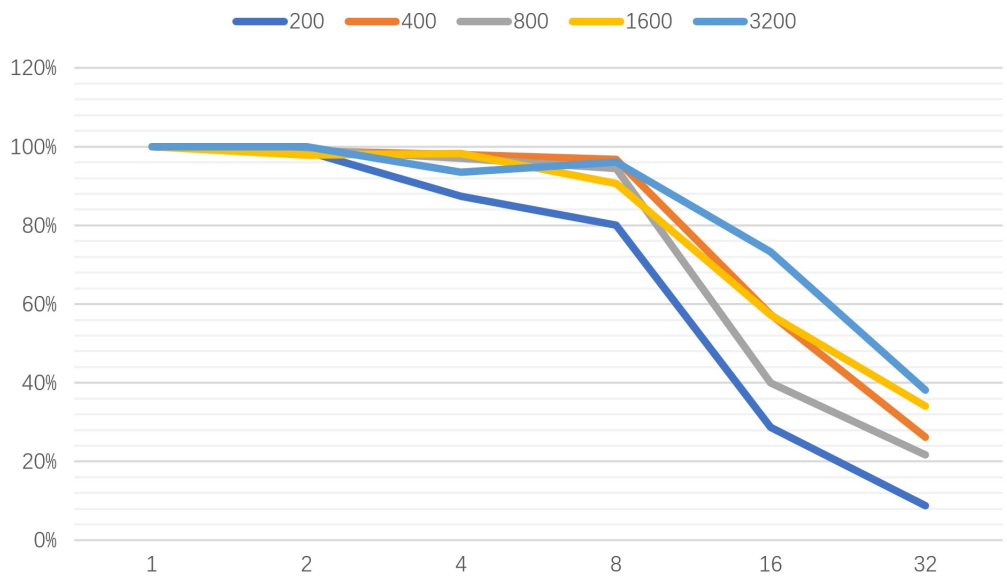

FIGURE 8: EFFICIENCY VERSUS PROCESS NUMBER

4 CONCLUSION

This report utilizes the knowledge on MPI to implement an parallel version of Odd-Even Transportation Sort. Some technical obstacles are encountered and overcome to achieve desirable performance and required robustness: we design and show the program flow and parallel data structure operation. Both sequential and parallel implementation of Odd-Even transportation sort are shown. We also propose two strategy: Augmentation and Residualization to achieve the required robustness. In the end, a series of experiments are conducted and verify the relationship between program performance and the number of processes being used. And the experiments support the relationship between the number of processes and the overall running time mentioned in the lecture. In all, the initial targets listed at the beginning of the report are majorly achieved.

5 APPENDIX: HOW TO RUN THE SUBMITTED CODE?

The source code is implemented on the given template. So it shouldn't be a problem in terms of Reproducibility. Compile the project with:

```
cmake --build . -j4
```

And run the executable file with four processes as an example(input.txt and output.txt need to be listed in the build file):

```
mpirun -n 4 main input.txt output.txt
```

Or use the Google test:

```
mpirun gtest_sort
```