THE CHINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC4005

DISTRIBUTED AND PARALLEL COMPUTING

Report for CSC4005 Project 1

Author:

Feng Yutong

Student Number:

120090266

October 11, 2022

Contents

1 Introduction

This project implements two versions of odd-even transposition sort. One is sequential sort. The other is parallel sort by using MPI. The algorithm is as followed:

1. Insides each process, compare the odd element with the posterior even element in odd iteration, or the even element with the posterior odd element in even iteration respectively. Swap the elements if the posterior element is smaller.

2. If the current process rank is P, and there some elements that are left over for comparison in step 1, Compare the boundary elements with process with rank P-1 and P+1. If the posterior element is smaller, swaps them.

3. Repeat 1-2 until the numbers are sorted.

The report will compare the performance between two versions, MPI using different numbers of cores.

2 Method

2.1 Sequential version

Use *while* loop to repeat the comparison process. Record *rd* (round) and use *rd&1* to decide whether do odd-even or even-odd sort. The process stops if there is no swap in one round. (see Figure 1)

2.2 Parallel version

Step 1. MPI initialization. Read and store data in root process. Then call *MPI_Scatter* function to distribute the same amount of data to the sub-processes.

Step 2. Sorting. Comparison is similar to sequential version, but the program per-

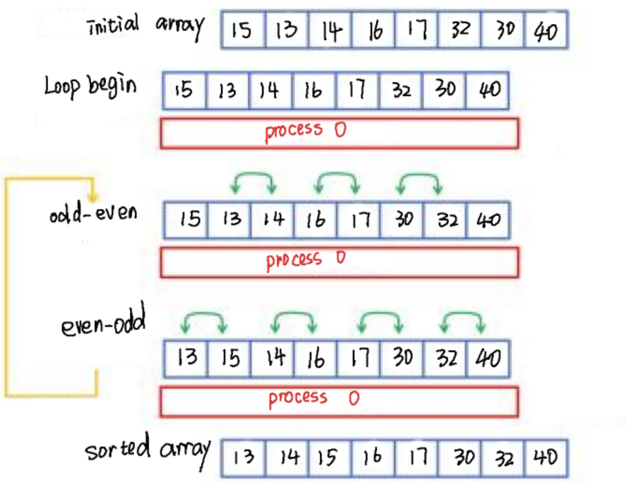


Figure 1: Sequential flow

form even-odd, (boundary), odd-even, (boundary) sort in one iteration. Boundary sort only occurs after even-odd or odd-even sort, so there may be a situation that the unsorted array has one iteration with no change due to no boundary sort. The boundary sort is done as followed: (1) Each process, say rank k, except the root process (rank = 0) sends their first element to their former process, say rank k-1. (2) Process k-1 compares the received element with its last element. If the received one is smaller, swap them. (3) Process k-1 sends the data back to process k and places it in the first position. Whether do boundary sort after odd-even or even-odd sort is dependent on the distributed array size.

Step 3. Root process call *MPI_Reduce* to calculate the OR of all local flags, which indicates whether there is a swap in one iteration. If all flags are false, the result will be false, indicating to end the loop. Then the root broadcast the result to all sub-processes.

Step 4. Root process gather all sorted data into a space and print out the result.

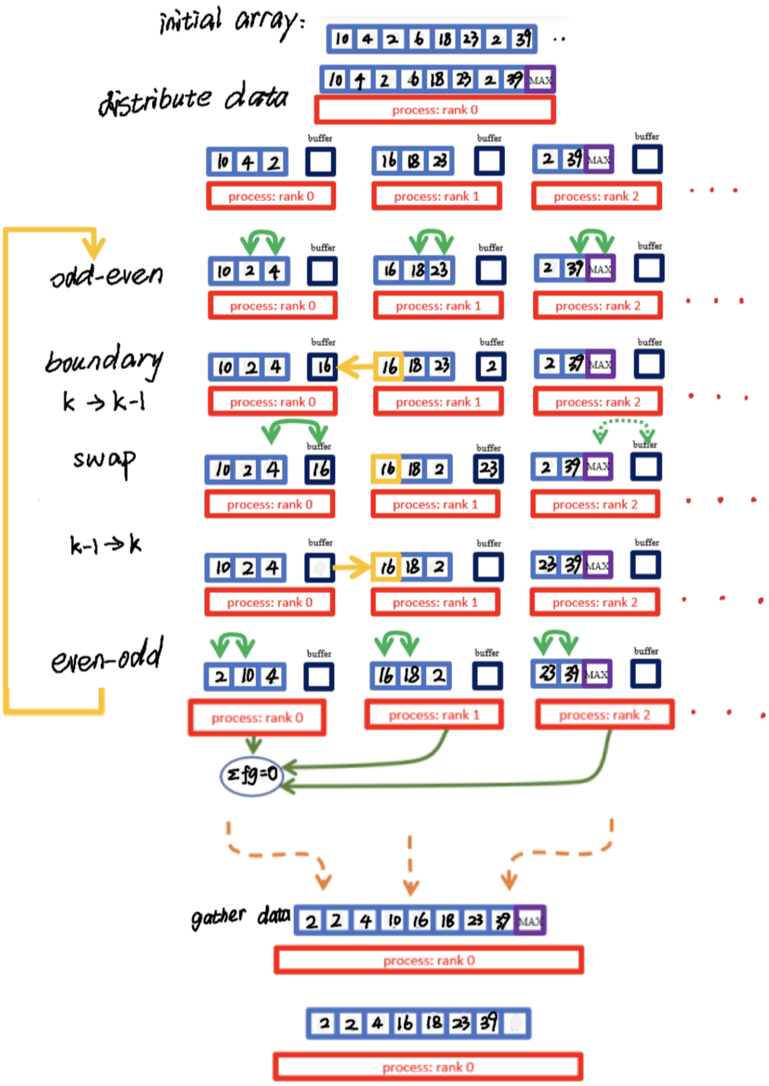


Figure 2: Parallel flow

2.3 Algorithm analysis

1. Correctness: The terminal condition is no swap occurring. Since we do odd-even and even-odd sort in iteration, if an element is smaller than its former element, there will be a swap. Therefore, no swap means a sorted array.

2. Time complexity:

For sequential version:

Average Case : $O(n^2)$

Best Case: $O(n)$

Worst Case: $O(n^2)$

For parallel version:

Assume we have n elements and p process, where each process stores $\frac{n}{p}$ elements. Each odd-even (even-odd) process takes $O(\frac{n}{p})$. Boundary transport takes $O(1)$ for each process. Assume we perform t iterations. So total time should be approximately $t \times (2O(\frac{n}{p}) + O(1))$, which can be seen as $O(\frac{n^2}{p})$. However, this is just theoretical analysis since communication time cannot be treated as $O(1)$ in real situation (e.g., waiting time). We will analyse real situation in the next part.

3 Result

3.1 Execution

1. Sequential Odd Even Transposition Sort

Compile:

g++ odd_even_sequential_sort.cpp -o ssort

To run it, use

./ssort $number_of_elements_to_sort $path_to_input_file

2. Parallel Odd Even Transposition Sort:

Compile odd_even_parallel_sort.cpp by

mpic++ odd_even_parallel_sort.cpp -o psort

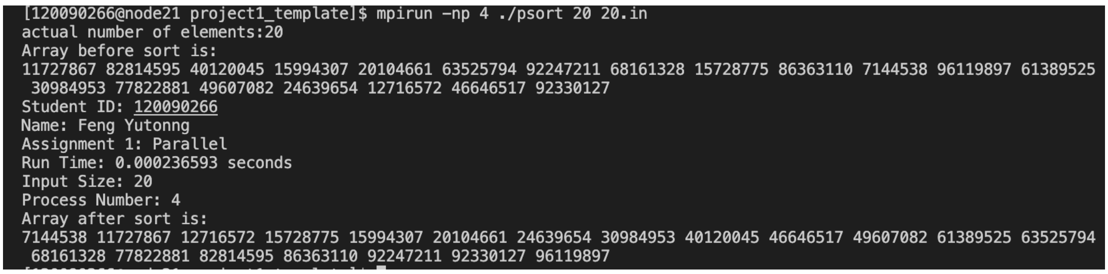
Run *salloc -n$num_of_core -p Project*

mpirun -np $num_of_core ./psort $number_of_elements_to_sort $path_to_input_file

Or you can **modify** the *run.sh* and use *qsub* to insert the process to the queue for running.

3.2 Demo

Here shows an example of 20-dims input array:



```
[120090266@node21 project1_template]$ mpirun -np 4 ./psort 20 20.in
actual number of elements:20
Array before sort is:
11727867 82814595 40120045 15994307 20104661 63525794 92247211 68161328 15728775 86363110 7144538 96119897 61389525
 30984953 77822881 49607082 24639654 12716572 46646517 92330127
Student ID: 120090266
Name: Feng Yutonng
Assignment 1: Parallel
Run Time: 0.000236593 seconds
Input Size: 20
Process Number: 4
Array after sort is:
7144538 11727867 12716572 15728775 15994307 20104661 24639654 30984953 40120045 46646517 49607082 61389525 63525794
 68161328 77822881 82814595 86363110 92247211 92330127 96119897
```

Figure 3: Demo

3.3 Performance analysis

1. Test Design

Array size: 10k, 50k, 100k, 300k, 600k

Process (Core) Number: 1, 2, 4, 8, 16

Data is the average of three runs.

2. Time

From Figure 4 and Table 1, we can see that the runtime decreases with the increasing of the process number as we analysed before. However, decreasing rate will decay when the process number is large. In addition, the case of 10k (Figure 5) shows a different trend. Its runtime first decrease and then increase. Above two phenomena are due to communication overhead. When process number is too large or the data size is too small, the communication overhead take the dominant effect and slower the whole process. We can amplify the impact by calculating speedup and efficiency.

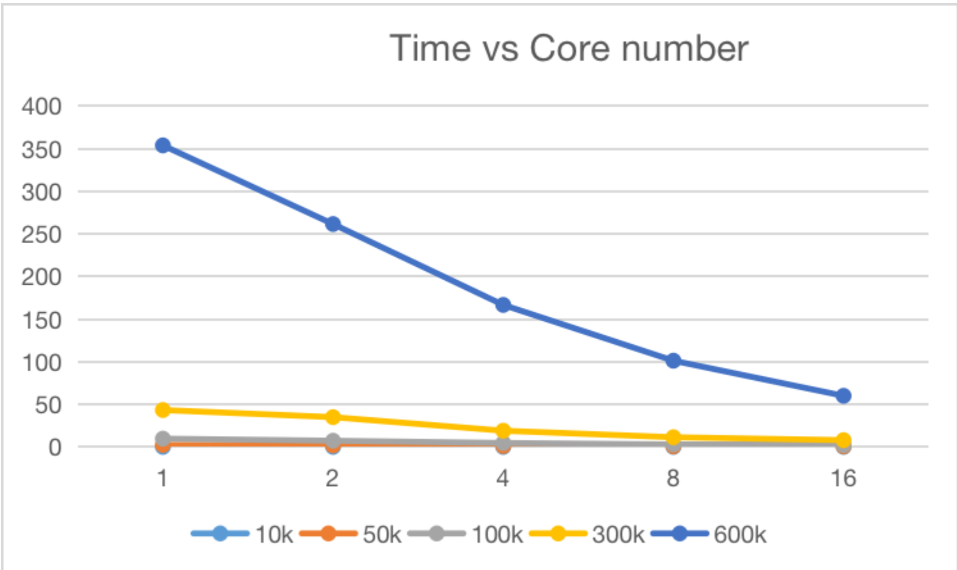


Figure 4: Time (s) based on different input size and core number

2. Speedup

The speedup is given as:

$$Speedup = \frac{\text{running time on one server}}{\text{running time on parallel server}}$$

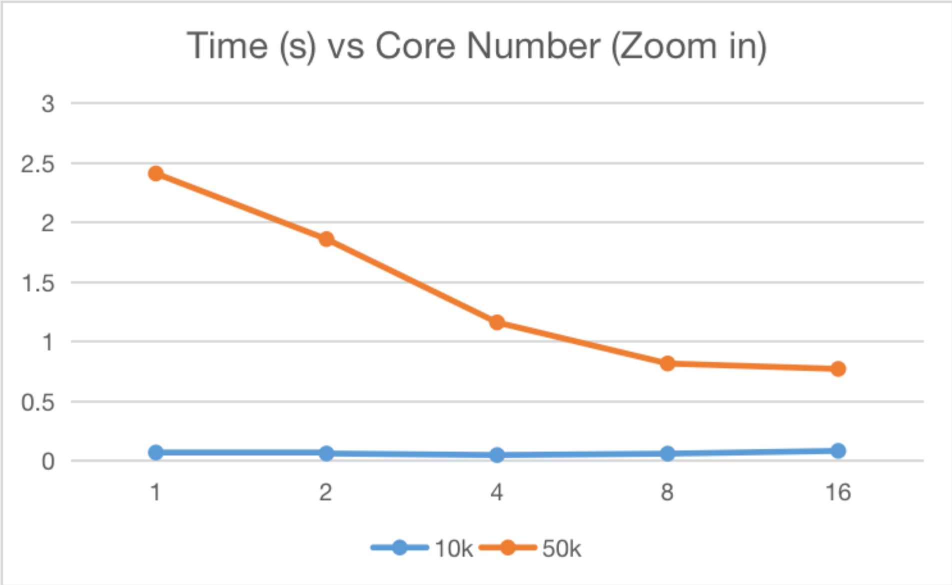


Figure 5: Time (s) based on smaller input size and core number

Size\Core number	1	2	4	8	16
10k	0.0714008	0.0629894	0.049866	0.0617178	0.0852042
50k	2.34183	1.7994	1.11316	0.756655	0.687626
100k	9.71557	7.28293	4.65404	2.93323	2.14633
300k	43.3193	34.9711	19.0314	11.3735	7.97791
600k	353.9	261.58	166.679	101.227	59.9998

Table 1: Time (s) based on different input data size and core number

Size\Core number	1	2	4	8	16
10k	1	1.133536754	1.431853367	1.156891529	0.837996249
50k	1	1.301450483	2.103767652	3.094977235	3.405674015
100k	1	1.334019413	2.087556188	3.312242818	4.526596563
300k	1	1.238717112	2.276201436	3.808792368	5.429905827
600k	1	1.352932181	2.399822413	3.951514912	6.666688889

Table 2: Speedup based on different input data size and core number

Figure 6 and Table 2 show that larger data size has a more obvious increment on speedup with the growth of the process number. Speedup of small data size (10k) first increase and then decrease. It's even smaller than one when communication overhead overwhelm the sorting time.

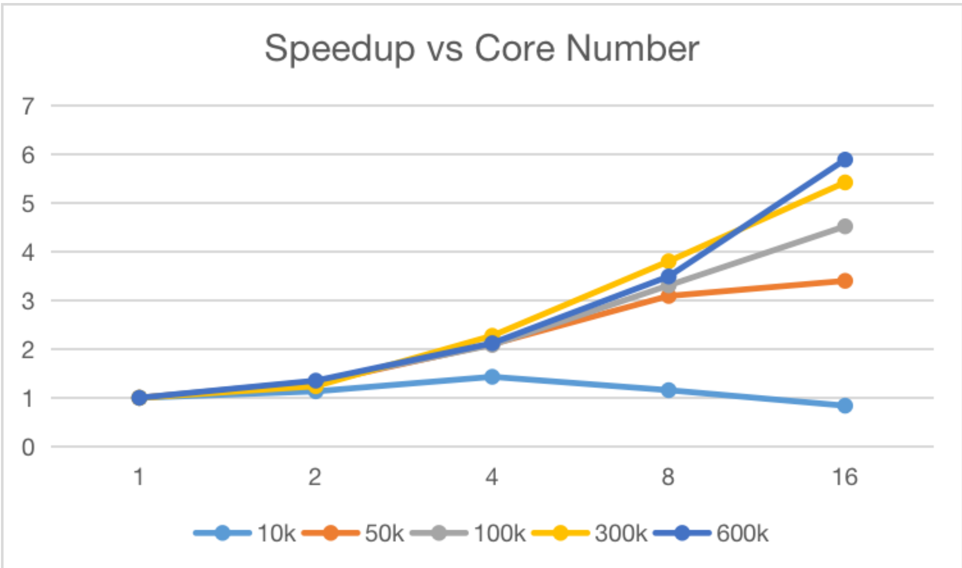


Figure 6: Speedup based on different input size and core number

3. Efficiency

Size\Core number	1	2	4	8	16
10k	1	0.566768377	0.357963342	0.144611441	0.052374766
50k	1	0.650725242	0.525941913	0.386872154	0.212854626
100k	1	0.667009706	0.521889047	0.414030352	0.282912285
300k	1	0.619358556	0.569050359	0.476099046	0.339369114
600k	1	0.676466091	0.599955603	0.493939364	0.416668056

Table 3: Efficiency based on different input data size and core number

Efficiency is given as

$$Efficiency = \frac{\text{Speedup}}{\text{process number}}$$

Figure 7 and Table 3 show that efficiency drops with the increasing of process number for all data size. One possible reason is that the data size distributed to sub-process decreases, so the real sorting proportion drops. More time is used to wait, e.g. MPI_Bcast and MPI_Reduce in the program. Therefore, efficiency drops.

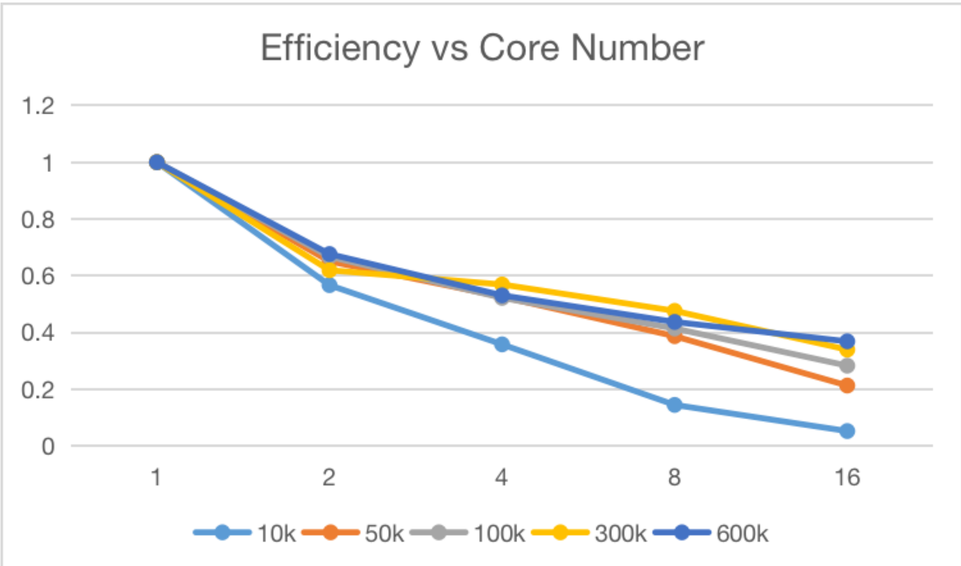


Figure 7: Efficiency based on different input size and core number

4. MPI vs Seqeuntial

Sequential is much slower than MPI. For example, Sequential uses 0.301582s while MPI uses 0.0714008s under input size of 10k. Sequential uses 7.60247s while MPI uses 2.34183s under input size of 10k. I guess this is due to different compile settings, which gives MPI high performance.

4 Conclusion

This project implements two version of odd-even transposition sort. The report analyses performance by theory and experiment. Runtime will decrease with the increase of process number when data input is large. However, too large process core or too small data size will weaken the effect due to communication overhead, which is indicated by speedup and efficiency. Therefore, it is important to find the balance between data size and process number so that we can improve performance and save resources. For future improvement, we can extend the buffer size during communication or improve the algorithm by using the waiting time to do some local calculation.

5 Source code

5.1 Sequential version

```
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <chrono>
#pragma GCC optimize(2)
```

```

int main (int argc, char **argv){

    int num_elements; // number of elements to be sorted
    num_elements = atoi(argv[1]); // convert command line argument to num_elements

    int elements[num_elements]; // store elements
    int sorted_elements[num_elements]; // store sorted elements

    std::ifstream input(argv[2]);
    int element;
    int i = 0;
    while (input >> element) {
        elements[i] = element;
        i++;
    }
    std::cout << "actual number of elements:" << i << std::endl;

    std::chrono::high_resolution_clock::time_point t1;
    std::chrono::high_resolution_clock::time_point t2;
    std::chrono::duration<double> time_span;
    t1 = std::chrono::high_resolution_clock::now(); // record time
```

```

    int rd = 0, fg = 1;
    while(fg){
        fg = 0;
        int j = (rd & 1) + 1;
        for(; j < i; j += 2)
            if(elements[j] < elements[j - 1]){
                fg = 1;
            std::swap(elements[j], elements[j - 1]);
            }
        rd += 1;
    }
    for (int i = 0; i < num_elements; i++) {
        sorted_elements[i] = elements[i];
    }
    t2 = std::chrono::high_resolution_clock::now();
    time_span = std::chrono::duration_cast<std::chrono::duration<double>>(t2 - t1);
    std::cout << "Student ID: " << "120090266" << std::endl; // replace it with your student id
    std::cout << "Name: " << "Feng Yutong" << std::endl; // replace it with your name
    std::cout << "Assignment 1" << std::endl;
    std::cout << "Run Time: " << time_span.count() << " seconds" << std::endl;
    std::cout << "Input Size: " << num_elements << std::endl;
```

```
    std::cout << "Process Number: " << 1 << std::endl;

    std::ofstream output(argv[2]+std::string(".seq.out"), std::ios_base::out);
    for (int i = 0; i < num_elements; i++) {
        output << sorted_elements[i] << std::endl;
    }

    return 0;
}
```

5.2 Parallel version

```
#include <mpi.h>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <chrono>


int main (int argc, char **argv){

    MPI_Init(&argc, &argv);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

```

```
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int num_elements; // number of elements to be sorted

    num_elements = atoi(argv[1]); // convert command line argument to num_elements

    int elements[num_elements]; // store elements
    int sorted_elements[num_elements]; // store sorted elements

    if (rank == 0) { // read inputs from file (master process)
        std::ifstream input(argv[2]);
        int element;
        int i = 0;
        while (input >> element) {
            elements[i] = element;
            i++;
        }
        std::cout << "actual number of elements:" << i << std::endl;
    }

    std::chrono::high_resolution_clock::time_point t1;
```

```
    std::chrono::high_resolution_clock::time_point t2;
    std::chrono::duration<double> time_span;
    if (rank == 0){
        t1 = std::chrono::high_resolution_clock::now();
            // record time
    }
    int num_my_element = num_elements / world_size; //
       number of elements allocated to each process
    int my_element[num_my_element]; // store elements
       of each process

    MPI_Scatter(elements, num_my_element, MPI_INT,
       my_element, num_my_element, MPI_INT, 0,
       MPI_COMM_WORLD); // distribute elements to each
       process
    int rd = 0;
    bool fg = 0, tot_flag = 0;
    int a;
    while(!tot_flag){
        fg = 1;
        tot_flag = 1;


        // even-odd
        for(int j = 1; j < num_my_element; j += 2)
            if(my_element[j] < my_element[j - 1]){
```

```
                    std::swap(my_element[j], my_element[j -
                        1]);
                    fg = 0;
                }

        if (num_my_element & 1){
            if(rank){
                MPI_Send(my_element, 1, MPI_INT, rank
                    -1, 0, MPI_COMM_WORLD);
                MPI_Recv(my_element, 1, MPI_INT, rank
                    -1, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
            }
            if(rank < world_size - 1){
                MPI_Recv(&a, 1, MPI_INT, rank+1, 0,
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                if(a < my_element[num_my_element-1]) {
                    std::swap(a, my_element[
                        num_my_element - 1]);
                    fg = 0;
                }
                MPI_Send(&a, 1, MPI_INT, rank+1, 0,
                    MPI_COMM_WORLD);
            }
        }
        // odd-even
        for(int j = 2; j < num_my_element; j += 2)
```

```
                if(my_element[j] < my_element[j - 1]){
                    std::swap(my_element[j], my_element[j -
                        1]);
                    fg = 0;
                }

        if (!(num_my_element & 1)){
            if(rank){
                MPI_Send(my_element, 1, MPI_INT, rank
                    -1, 0, MPI_COMM_WORLD);
                MPI_Recv(my_element, 1, MPI_INT, rank
                    -1, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
            }
            if(rank < world_size - 1){
                MPI_Recv(&a, 1, MPI_INT, rank+1, 0,
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                if(a < my_element[num_my_element-1]) {
                    std::swap(a, my_element[
                        num_my_element - 1]);
                    fg = 0;
                }
                MPI_Send(&a, 1, MPI_INT, rank+1, 0,
                    MPI_COMM_WORLD);
            }
        }

```

```
            MPI_Reduce(&fg, &tot_flag, 1, MPI_C_BOOL,
                MPI_LAND, 0, MPI_COMM_WORLD);
            MPI_Bcast(&tot_flag, 1, MPI_C_BOOL, 0,
                MPI_COMM_WORLD);
            rd += 1;

        }
        MPI_Gather(my_element, num_my_element, MPI_INT,
            sorted_elements, num_my_element, MPI_INT, 0,
            MPI_COMM_WORLD); // collect result from each
            process

        if (rank == 0){ // record time (only executed in
            master process)
            t2 = std::chrono::high_resolution_clock::now();
            time_span = std::chrono::duration_cast<std::
                chrono::duration<double>>(t2 - t1);
            std::cout << "Student ID: " << "120090266" <<
                std::endl; // replace it with your student
                id
            std::cout << "Name: " << "Feng Yutonng" << std
                ::endl; // replace it with your name
            std::cout << "Assignment 1: Parallel" << std::
                endl;
            std::cout << "Run Time: " << time_span.count()
                << " seconds" << std::endl;
```

```
        std::cout << "Input Size: " << num_elements << std::endl;
        std::cout << "Process Number: " << world_size << std::endl;
    }

    if (rank == 0){ // write result to file (only executed in master process)
        std::ofstream output(argv[2]+std::string(".parallel.out"), std::ios_base::out);
        for (int i = 0; i < num_elements; i++) {
            output << sorted_elements[i] << std::endl;
        }
    }

    MPI_Finalize();

    return 0;
}
```