

THE CHINESE UNIVERSITY OF HONG KONG,
SHENZHEN

CSC4005 PARALLEL PROGRAMMING

Homework 1 Report

Name: Xiang Fei

Student ID: 120090414

Email: xiangfei@link.cuhk.edu.cn

Date: 2022.10

Table of Contents

List of Figures	ii
List of Tables	iii
1 Introduction	1
2 Design and Method	2
2.1 Random number generator	2
2.2 Sequential Version odd-even sort	3
2.3 Parallel Version odd-even sort	4
2.3.1 Program Flow	4
2.3.2 Remainder Trick	6
2.4 Check Program	7
3 Execution	8
3.1 Test data generator	8
3.2 Sequential Odd Even Transposition Sort	9
3.3 Parallel Odd Even Transposition Sort	9
3.4 Check the correctness	9
3.5 Submit the job to sbatch	10
4 Result & Analysis	10
4.1 Correctness	10
4.2 Robustness	12
4.3 Comparison regarding to the number of cores used in the MPI program and the array size	13
4.4 Comparison between MPI and sequential case	15
5 Conclusion	17
Bibliography	19

Appendix	20
A Sequential Version Odd-even Sort Code	20
B Parallel Version Odd-even Sort Code	22
C The detailed result of comparison regarding to number of cores and array size .	25
D The detailed result of comparison between sequential and MPI	25

List of Figures

1 sequential sort	1
2 parallel sort	2
3 The main function of the data generator	2
4 The flow chart of sequential odd-even sort	3
5 The core code of sequential odd-even sort	3
6 The flow chart of parallel odd-even sort	4
7 The core code of parallel odd-even sort	5
8 The data transposition details in the array	6
9 The calculation of remainder and new length	6
10 Take the first n elements as the results	6
11 Load the data to the extended array	7
12 The main function of the check program	7
13 The cluster resource limit	8
14 The 20-dims unsorted array	10
15 Use sequential odd-even sort to get the 20-dims sorted array	11
16 Use parallel odd-even sort to get the 20-dims sorted array	11
17 Use parallel odd-even sort for the 10000-dims sorted array	11
18 Use parallel odd-even sort for the 20000-dims sorted array	12
19 The robustness test for 10000 elements and 7 processes case	12
20 The robustness test for 20000 elements and 13 processes case	13
21 The results of the small array size (1000) case	13

22	The results of the medium array size (10000) case	14
23	The results of the small array size (100000) case	14
24	The comparison between sequential and MPI: the small array size (1000) case .	16
25	The comparison between sequential and MPI: the medium array size (10000) case	16
26	The comparison between sequential and MPI: the large array size (100000) case	17
27	The detailed result of comparison regarding to number of cores and array size .	25
28	The detailed result of comparison between sequential and MPI	25

List of Tables

1 Introduction

In homework 1, we are required to write a parallel odd-even transposition sort by using MPI (Message Passing Interface). We also need to implement a sequential version used for comparison and analysis. For parallel version, MPI is used to implement the multi-process work. MPI is a standardized and portable messaging standard designed to run on parallel computing architectures. The MPI standard defines the syntax and semantics of library routines useful to a wide range of users writing portable messaging programs in C, C++, and Fortran. There are several open source MPI implementations that have fostered the parallel software industry and encouraged the development of portable and scalable massively parallel applications. MPI is still the dominant model used in HPC today. It is a parallel technology based on inter-process communication.

In computing, an odd-even sort or odd-even transposition sort (also known as brick sort or parity sort) is a relatively simple sorting algorithm originally developed for parallel processors with local interconnect. It is a comparison sort related to bubble sort and has many things in common with bubble sort. It works by comparing all odd/even indexed adjacent element pairs in the list, and if a pair is in the wrong order (the first is greater than the second), it switches the element. Next repeat this for even/odd index pairs (adjacent elements). Then it alternates between odd/even and even/odd steps until the list is sorted. The figure of sequential version shows as follows.

Indices:	0	1	2	3	4	5	6	7
Initial values:	G	H	F	D	E	C	B	A
After odd-even exchange:	G	F	H	D	E	B	C	A
After even-odd exchange:	F	G	D	H	B	E	A	C
After odd-even exchange:	F	D	G	B	H	A	E	C
After even-odd exchange:	D	F	B	G	A	H	C	E
After odd-even exchange:	D	B	F	A	G	C	H	E
After even-odd exchange:	B	D	A	F	C	G	E	H
After odd-even exchange:	B	A	D	C	F	E	G	H
After even-odd exchange:	A	B	C	D	E	F	G	H

Figure 1: sequential sort

According to Worsch, a unsorted array can be sorted in n pass using odd-even transposition sort. In fact, if we don't check the completion condition of this sorting algorithm, it will go through n passes, where n is the total number of elements in the input array. And each pass takes $O(n)$ steps, so the time complexity of this algorithm is $O(n^2)$.

On parallel processors, The algorithm extends efficiently to the case of multiple items per processor. Each processor sorts its own sublist at each step, using any efficient sort algorithm,

and then performs a transposition-merge, operation with its neighbor, with neighbor pairing alternating between odd-even and even-odd on each step. Initially, m numbers are distributed to n processes respectively, and the pair comparison in each process goes as mentioned before. On the boundary, the process needs to decide whether to pass or receive the element from the neighboring processes. The pair comparison occurred on the boundary requires message passing of processes. The figure of parallel version shows as follows.

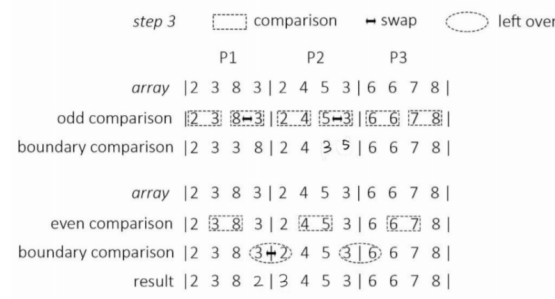


Figure 2: parallel sort

2 Design and Method

2.1 Random number generator

In order to generate random numbers, I just use the **test_data_generator.cpp** file which is provided in the csc4005 github template. When using the executable file, the user needs to specify the total number of elements and the name of the generated data file. The range of the number is between 1 and 99999999. The basic of this program is **rand()** function. The main function is showed as follows.

```
#define random(a, b) (rand() % (b - a) + a)

int main(int argc, char **argv){
    int num_elements; // number of elements to generate
    num_elements = atoi(argv[1]);

    std::ofstream out;
    out.open(argv[2], std::ios_base::out);

    srand((int)time(0));
    for (int i = 0; i < num_elements; i++){
        out << random(1, 99999999) << std::endl;
    }
    out.close();

    return 0;
}
```

Figure 3: The main function of the data generator

2.2 Sequential Version odd-even sort

The implementation of the sequential version odd-even sort is based on the csc4005 github template and follows the algorithm which is illustrated in introduction part. When using the executable file, the user needs to specify the total number of elements and the input data file. The input data will be loaded into a array. And in this program, we use **chrono** library to record the execution time. The flow chart is showed as follows.

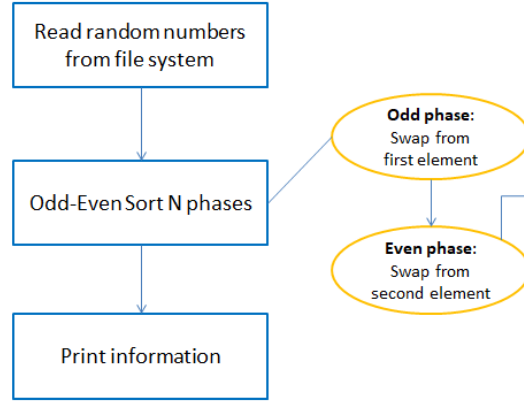


Figure 4: The flow chart of sequential odd-even sort

For the core part of sequential odd-even sort, first I use a while loop, which is ended when there is no swap during the odd-pass and even-pass. Inside the while loop, there are two for loop representing the odd-pass and even-pass, respectively. In each pass, when the left element is larger than the right one, then swap their position. Finally, personal information like name and student id as well we runtime profiling like process numbers, sort duration will print out. All the sorted data will be written to a new file. The core part is like the following.

```
bool hasBeenSorted;
while(!hasBeenSorted){
    hasBeenSorted=true;
    for(int i=0;i<num_elements-1;i+=2){
        if(elements[i]>elements[i+1]){
            std::swap(elements[i],elements[i+1]);
            hasBeenSorted=false;
        }
    }
    for(int i=1;i<num_elements-1;i+=2){
        if(elements[i]>elements[i+1]){
            std::swap(elements[i],elements[i+1]);
            hasBeenSorted=false;
        }
    }
}
```

Figure 5: The core code of sequential odd-even sort

2.3 Parallel Version odd-even sort

2.3.1 Program Flow

The implementation of the parallel version odd-even sort is based on the csc4005 github template and follows the algorithm which is illustrated in introduction part. The following figure is the program flow chart of parallel odd-even sort.

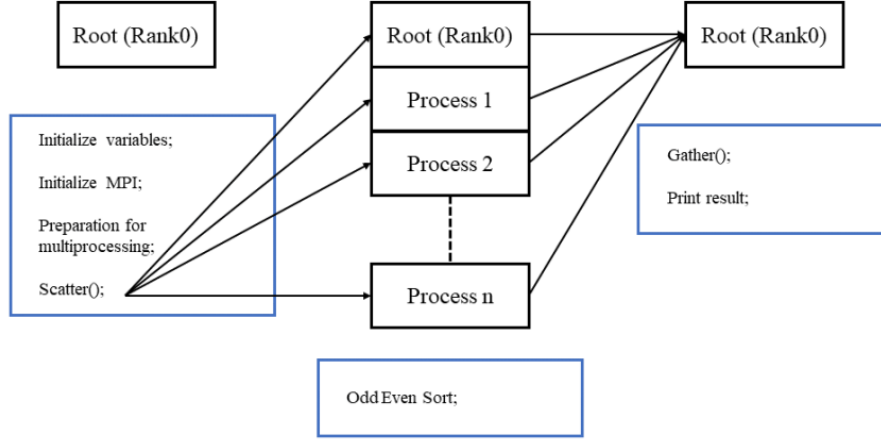


Figure 6: The flow chart of parallel odd-even sort

There are the main parts designed in the program:

First part:

In the main function, first, some variables will be initialized, including the total number of elements to be sorted, loading of the input data, etc. And then, it does the MPI initialization, including space allocation, specifying communicator, and get the rank & processor number. So the number of elements allocated to each processor can be computed and the local array for each process can be constructed. But if the number of elements and the process number are not divisible, we add the INT_MAX to the end ($processNumber - remainderValue$) times, make the length and the process number become divisible (after we finishing the sorting, we just need to take the first n elements, since the extended elements must stay at the tail of the array). The details of this trick will be discussed later. In the end of the first part, **MPI_Scatter** is used to distribute elements to each process.

Second part:

The second part is the odd-even sort part. In this part, we first construct three variables to represent the element send to left process, right process and the element received from other processes respectively. After that, a for loop is used which last n times, where n is the total number of elements to be sorted. Inside the for loop, there are three passes, first is odd pass, then is even pass, and the last is the boundary pass. In the odd pass and even pass, when the left element is larger than the right one, swap their positions. In the boundary

pass, since the compared elements are located in different process, so we need to communicate between different processes. For processes whose rank is not 0 (the leftest one), they send their first element to the left process and receive the last element of the left process. Then, if the received one is larger then their first element, let the first element of the local array becomes the received one. For processes whose rank is not the number of processes minus 1 (the rightest one), they send their last element to the right process and receive the first element of the right process. Then, if their last element is larger then the received one , let the last element of the local array becomes the received one. It should be pointed out that the boundary pass is implemented based on **MPI_Sendrecv** function. The core code to implement parallel odd-even sort is like the following.

```

for(int i=0;i<num_elements_extended;i++){
    if(i%2==0){
        for(int j=0;j<num_my_element-1;j+=2){
            if(my_element[j]>my_element[j+1]){
                std::swap(my_element[j],my_element[j+1]);
            }
        }
    }
    else{
        for(int j=1;j<num_my_element-1;j+=2){
            if(my_element[j]>my_element[j+1]){
                std::swap(my_element[j],my_element[j+1]);
            }
        }
        if(rank!=0){
            send_element_head = my_element[0];
            MPI_Sendrecv(&send_element_head,1,MPI_LONG,rank-1,0,&recv_element,1,MPI_LONG,rank-1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
            if(recv_element>my_element[0]){
                my_element[0]=recv_element;
            }
        }
        if(rank==world_size-1){
            send_element_tail = my_element[num_my_element-1];
            MPI_Sendrecv(&send_element_tail,1,MPI_LONG,rank+1,0,&recv_element,1,MPI_LONG,rank+1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
            if(my_element[num_my_element-1]>recv_element){
                my_element[num_my_element-1]=recv_element;
            }
        }
    }
}
}

```

Figure 7: The core code of parallel odd-even sort

Third part:

In the third part, the root process calls the **MPI_Gather** function to get the sorted array. Finally, personal information like name and student id as well we runtime profiling like process numbers, sort duration will print out. All the sorted data will be written to a new file.

The following graph shows the data transposition details in the array.

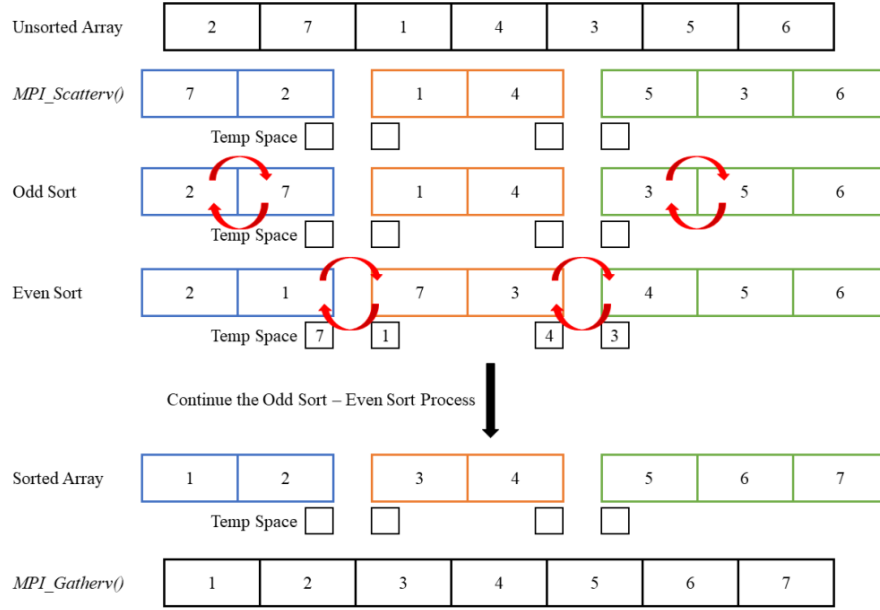


Figure 8: The data transposition details in the array

2.3.2 Remainder Trick

When the number of elements to be sorted (n) and the process number (p) are not divisible, I calculate the remainder r , and add `INT_MAX` to the end of the unsorted array $p - r$ times. Therefore, the length of the extended array and the process number are divisible. Then we do the sorting, and take the first n sorted element as the results, since the added elements must stay at the tail of the sorted extended array. The core code is showed as follows.

```
num_elements = atoi(argv[1]); // convert command line argument to num_elements
int num_my_element = num_elements / world_size; // number of elements allocated to each process
int remainder = num_elements - num_my_element*world_size;
int num_elements_extended = num_elements+world_size-remainder;

int elements_extended[num_elements_extended];
int sorted_elements[num_elements]; // store sorted elements
int sorted_elements_extended[num_elements_extended];
```

Figure 9: The calculation of remainder and new length

```
for(int i=0;i<num_elements;i++){
    sorted_elements[i] = sorted_elements_extended[i];
}
```

Figure 10: Take the first n elements as the results

```

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

```

Figure 11: Load the data to the extended array

2.4 Check Program

In order to check whether the sort is correct, I just use the **check_sorted.cpp** file which is provided in the csc4005 github template. When using the executable file, the user needs to specify the total number of elements and the name of the output sorted data file. The main function is showed as follows.

```

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
    std::ifstream input(argv[2]);
    int element;
    int i = 0;
    while (input >> element) {
        elements[i] = element;
        i++;
    }

    int unsort_count = 0;
    for (int i = 0; i < num_elements-1; i++) {
        if (elements[i] > elements[i + 1])
            unsort_count++;
    }

    if (unsort_count < 1) {
        std::cout << "Sorted." << std::endl;
    } else {
        std::cout << "Not Sorted. " << unsort_count << " errors." << std::endl;
    }
}

```

Figure 12: The main function of the check program

3 Execution

My code executes on the remote cluster. The cluster resource limits are shown in the following graph.

Cluster Resource Limit

Oct 1, 2022

partition	Debug	Project
MaxNodes (max number of nodes allocated to a job)	1	1
MaxCPUsPerNode (max number of cpus on a node allocated to a job)	8	40
Max number of total cpu cores allocated to a job	1*8=8	1*40=40
MaxTime (max running time of a job)	60min	10min
MaxJobsPerUser (max number of running jobs of a user at a given time)	2	1
MaxSubmitJobsPerUser (max number of jobs submitted of a user at a given time)	10	100
Total number of nodes in this partition (we will add more if not enough)	9	18

For time consuming jobs with a few cores, you can use `Debug` partition.

For jobs requiring many cores, you can use `Project` partition.

Figure 13: The cluster resource limit

3.1 Test data generator

'test_data_generator.cpp' is a test data generator.

compile:

```
1 g++ test_data_generator.cpp -o gen
```

this operation will produce an executable named as 'gen'.

execute:

Then, specify the number of elements to be sorted, and the file name.

```
1 ./gen $number_of_elements_to_sort $save_file_name
```

For example, to generate a dataset with '10000' elements and name it as './test_data/10000a.in',

```
1 ./gen 10000 ./test_data/10000a.in
```

Then you will find '10000a.in' in './test_data' directory.

You can generate many datasets and use them to test the program.

3.2 Sequential Odd Even Transposition Sort

Sequential Odd Even Transposition Sort is implemented in 'odd_even_sequential_sort.cpp'.

compile:

```
1 g++ odd_even_sequential_sort.cpp -o ssort
```

execute:

```
1 ./ssort $number_of_elements_to_sort $path_to_input_file
```

For example,

```
1 ./ssort 10000 ./test_data/10000a.in
```

The program will generate an output file called '10000a.in.seq.out' in './test_data'.

3.3 Parallel Odd Even Transposition Sort

Parallel Odd Even Transposition Sort is implemented in 'odd_even_parallel_sort.cpp'.

compile:

Use mpic++ to compile it .

```
1 mpic++ odd_even_parallel_sort.cpp -o psort
```

execute:

```
1 salloc -n8 -p Debug # allocate cpu for your task
2 mpirun -np 8 ./psort $number_of_elements_to_sort $path\_to\_input\_file
```

For example, to sort './test_data/10000a.in' generated before, we can use

```
1 salloc -n8 -p Debug # allocate cpu for your task
2 mpirun -np 8 ./psort 10000 ./test\_data/10000a.in
```

The program will generate an output file called '10000a.in.parallel.out' in './test_data'.

3.4 Check the correctness

'check_sorted.cpp' is a tool to check if the sorting result is correct.

compile:

```
1 g++ check_sorted.cpp -o check
```

execute:

```
1 ./check $number_of_elements_to_sort $path_to_output_file
```

For example, if we want to check the output file ‘./test_data/10000a.in.parallel.out’, you can use

```
1 ./check 10000 ./test_data/10000a.in. parallel .out
```

The output will be like (but not identical):

Not Sorted. 4983 errors.

3.5 Submit the job to sbatch

Firstly write a shell script called ‘sbatch_template.sh’,

```
1 #!/bin/bash
2 #SBATCH --job-name=your_job_name # Job name
3 #SBATCH --nodes=1                # Run all processes on a single node
4 #SBATCH --ntasks=20              # number of processes = 20
5 #SBATCH --cpus-per-task=1        # Number of CPU cores allocated to each process
6 #SBATCH --partition=Project      # Partition name: Project or Debug (Debug is default)
7
8 cd /nfsmnt/119010355/CSC4005_2022Fall_Demo/project1_template/
9 mpirun -np 4 ./psort 10000 ./test_data/10000a.in
```

Then use

```
1 sbatch xxx.sh
```

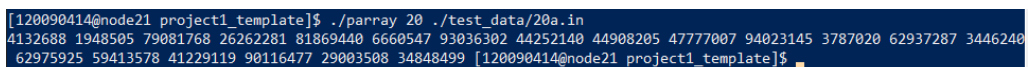
to submit the job. No need to use ‘salloc’ here.

4 Result & Analysis

4.1 Correctness

- **Print out 20-dims array:**

The input data:



```
[120090414@node21 project1_template]$ ./parray 20 ./test_data/20a.in
4132688 1948505 79081768 26262281 81869440 6660547 93036302 44252140 44908205 47777007 94023145 3787020 62937287 3446240
62975925 59413578 41229119 90116477 29003508 34848499 [120090414@node21 project1_template]$
```

Figure 14: The 20-dims unsorted array

Use sequential odd-even sort:

```
[120090414@node21 project1_template]$ ./ssort 20 ./test_data/20a.in
actual number of elements:20
Student ID: 120090414
Name: Xiang Fei
Assignment 1
Run Time: 8.514e-06 seconds
Input Size: 20
Process Number: 1
[120090414@node21 project1_template]$ ./check 20 ./test_data/20a.in.seq.out
Sorted.
[120090414@node21 project1_template]$ ./parray 20 ./test_data/20a.in.seq.out
1948505 3446240 3787020 4132688 6660547 26262281 29003508 34848499 41229119 44252140 44908205 47777007 59413578 62937287
62975925 79081768 81869440 90116477 93036302 94023145 [120090414@node21 project1_template]$
```

Figure 15: Use sequential odd-even sort to get the 20-dims sorted array

We can see that the array is successfully sorted.

Use parallel odd-even sort (4 processes as an example):

```
[120090414@node21 project1_template]$ salloc -n4 -p Debug
salloc: Pending job allocation 7211
salloc: job 7211 queued and waiting for resources
salloc: job 7211 has been allocated resources
salloc: Granted job allocation 7211
salloc: Waiting for resource configuration
salloc: Nodes node08 are ready for job
[120090414@node21 project1_template]$ mpirun -np 4 ./psort 20 ./test_data/20a.in
actual number of elements:20
Student ID: 120090414
Name: Xiang Fei
Assignment 1
Run Time: 0.000280577 seconds
Input Size: 20
Process Number: 4
[120090414@node21 project1_template]$ ./check 20 ./test_data/20a.in.parallel.out
Sorted.
[120090414@node21 project1_template]$ ./parray 20 ./test_data/20a.in.parallel.out
1948505 3446240 3787020 4132688 6660547 26262281 29003508 34848499 41229119 44252140 44908205 47777007 59413578 62937287
62975925 79081768 81869440 90116477 93036302 94023145 [120090414@node21 project1_template]$
```

Figure 16: Use parallel odd-even sort to get the 20-dims sorted array

We can see that the array is successfully sorted.

- Use check program to check 10000-dims and 20000-dims case as examples

10000-dims case:

```
[120090414@node21 project1_template]$ mpirun -np 4 ./psort 10000 ./test_data/10000a.in
actual number of elements:10000
Student ID: 120090414
Name: Xiang Fei
Assignment 1
Run Time: 0.0416805 seconds
Input Size: 10000
Process Number: 4
[120090414@node21 project1_template]$ ./check 10000 ./test_data/10000a.in.parallel.out
Sorted.
[120090414@node21 project1_template]$
```

Figure 17: Use parallel odd-even sort for the 10000-dims sorted array

We can see that the array is successfully sorted.

20000-dims case:

```
[120090414@node21 project1_template]$ mpirun -np 4 ./psort 20000 ./test_data/20000a.in
actual number of elements:20000
Student ID: 120090414
Name: Xiang Fei
Assignment 1
Run Time: 0.136073 seconds
Input Size: 20000
Process Number: 4
[120090414@node21 project1_template]$ ./check 20000 ./test_data/20000a.in.parallel.out
Sorted.
[120090414@node21 project1_template]$
```

Figure 18: Use parallel odd-even sort for the 20000-dims sorted array

We can see that the array is successfully sorted.

4.2 Robustness

Consider the case that the number of elements to be sorted and the process number are not divisible.

10000 elements, 7 processes:

```
[120090414@node21 project1_template]$ salloc -n7 -p Debug
salloc: Pending job allocation 7241
salloc: job 7241 queued and waiting for resources
salloc: job 7241 has been allocated resources
salloc: Granted job allocation 7241
[120090414@node21 project1_template]$ mpirun -np 7 ./psort 10000 ./test_data/10000a.in
actual number of elements:10000
Student ID: 120090414
Name: Xiang Fei
Assignment 1
Run Time: 0.0332787 seconds
Input Size: 10000
Process Number: 7
[120090414@node21 project1_template]$ ./check 10000 ./test_data/10000a.in.parallel.out
Sorted.
```

Figure 19: The robustness test for 10000 elements and 7 processes case

We can see that the program can successfully sort the array in this case.

20000 elements, 13 processes:


```

[120090414@node21 project1_template]$ salloc -n13 -p Project
salloc: Pending job allocation 7258
salloc: job 7258 queued and waiting for resources
salloc: job 7258 has been allocated resources
salloc: Granted job allocation 7258
[120090414@node21 project1_template]$ mpirun -np 13 ./psort 20000 ./test_data/20000a.in
actual number of elements:20000
Student ID: 120090414
Name: Xiang Fei
Assignment 1
Run Time: 0.0765829 seconds
Input Size: 20000
Process Number: 13
[120090414@node21 project1_template]$ ./check 20000 ./test_data/20000a.in.parallel.out
Sorted.

```

Figure 20: The robustness test for 20000 elements and 13 processes case

We can see that the program can successfully sort the array in this case.

4.3 Comparison regarding to the number of cores used in the MPI program and the array size

Array size: 1000, 10000, 100000;

Process number: 1 - 20.

For every case, I run five times and calculate the average time.

The detailed result table can be seen in Appendices.

Small array size (1000):

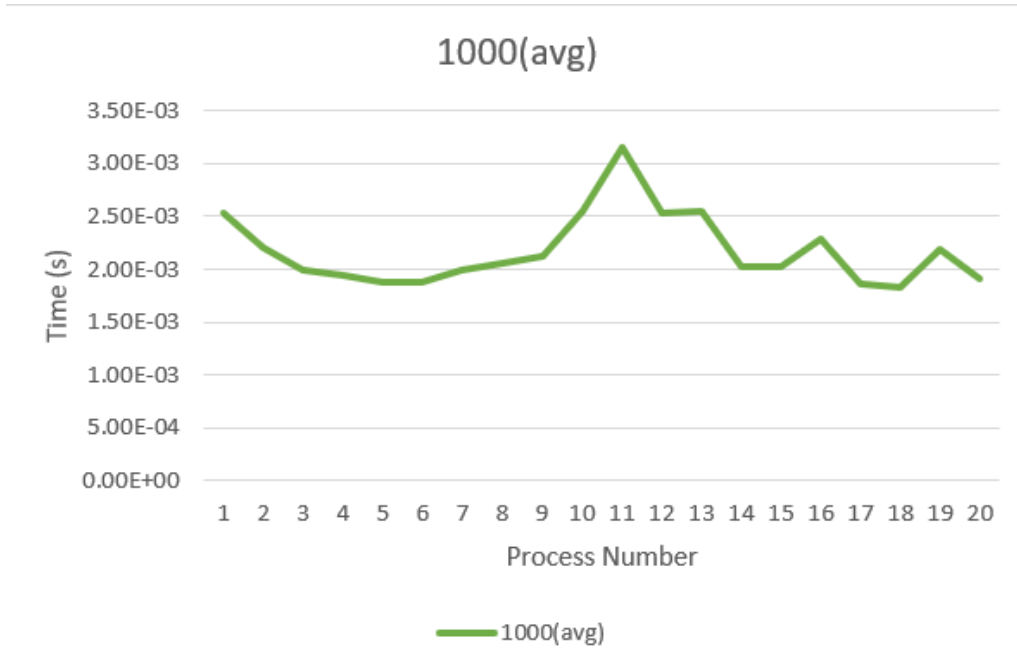


Figure 21: The results of the small array size (1000) case

Medium array size (10000):

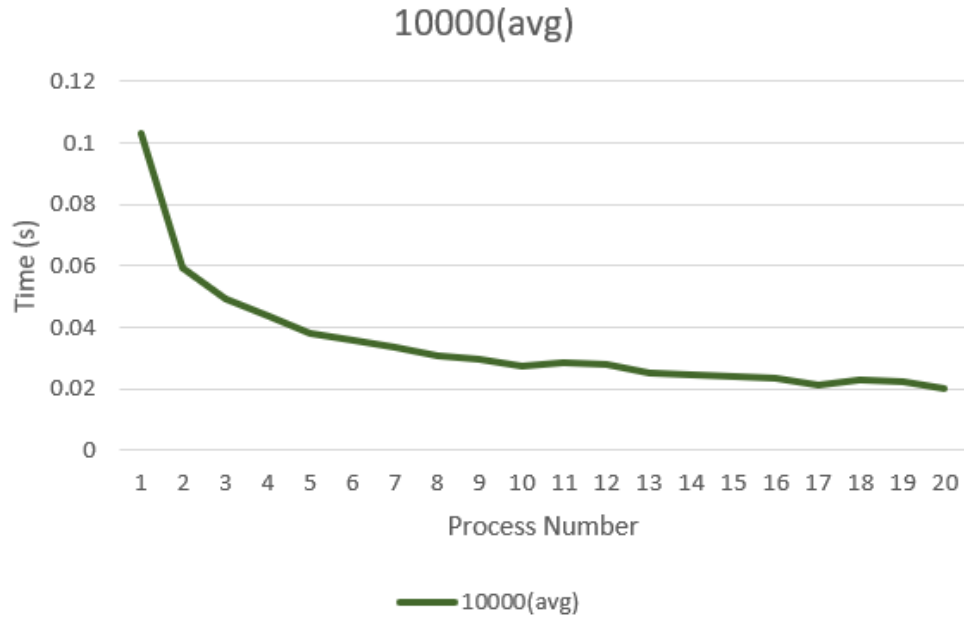


Figure 22: The results of the medium array size (10000) case

Large array size (100000):

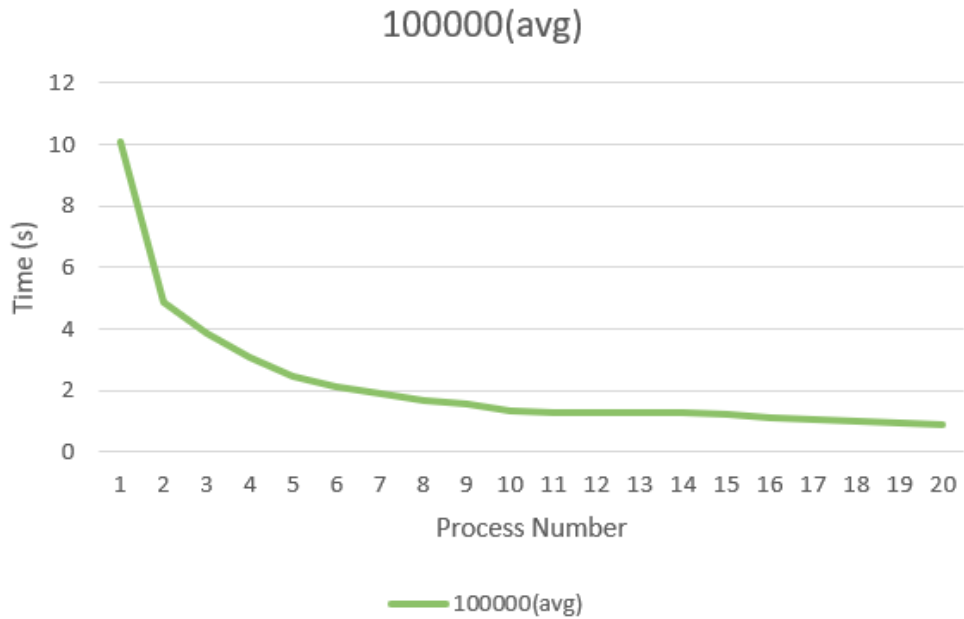


Figure 23: The results of the small array size (100000) case

From the small array size case graph, we can see that the time does not decrease significantly with the increase of the number of processes, and it fluctuates. This is because when the size is small, the time gap is small, and the time of communication between different processes is

comparable with the running time. Besides, there is another reason that we can only measure with limited times, so there is some experimental error, and when the array is small, the error time and the running time are also comparable. And in fact, when the number of elements to be sorted and the process number are not divisible, there are some extra loop to make sure that the sort can be successful, it can also cause time differences. When the array size is small, the use of multi-processing is not obvious for the improvement of the effect, which will cause a waste of resources. Fewer processes are recommended when array size is small to fully utilize computing resources.

From the medium array size case and large array size case graphs, we can find that as the number of processes increases, the running time decreases significantly, but the effect of improvement decreases as the number of processes increases. The improvement is most noticeable when going from single process to dual process. For arrays of these sizes, it is recommended to appropriately increase the number of processes to greatly improve the running speed and make full use of computing resources.

4.4 Comparison between MPI and sequential case

Array size: 1000, 10000, 100000;

Process number: sequential case, and multi-processing case (2 - 20).

For every case, I run five times and calculate the average time.

The detailed result table can be seen in Appendices.

Small array size (1000):

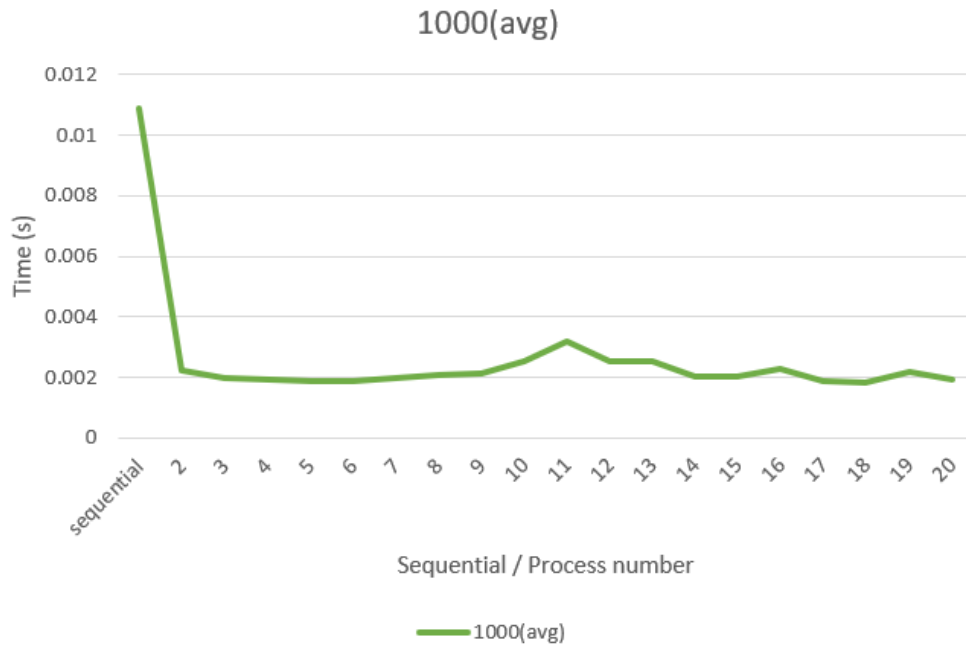


Figure 24: The comparison between sequential and MPI: the small array size (1000) case

Medium array size (10000):

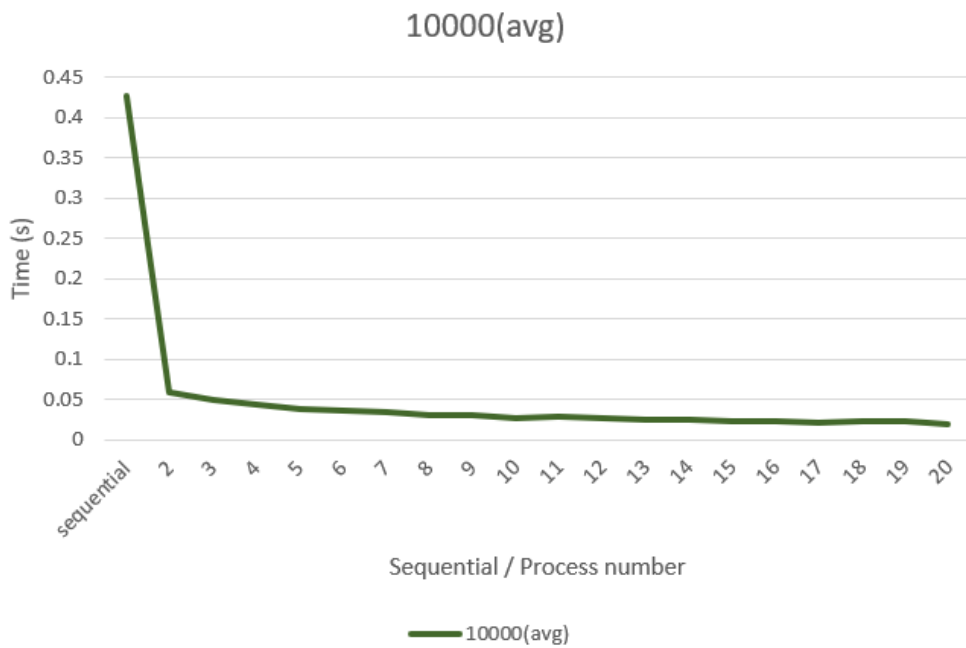


Figure 25: The comparison between sequential and MPI: the medium array size (10000) case

Large array size (100000):

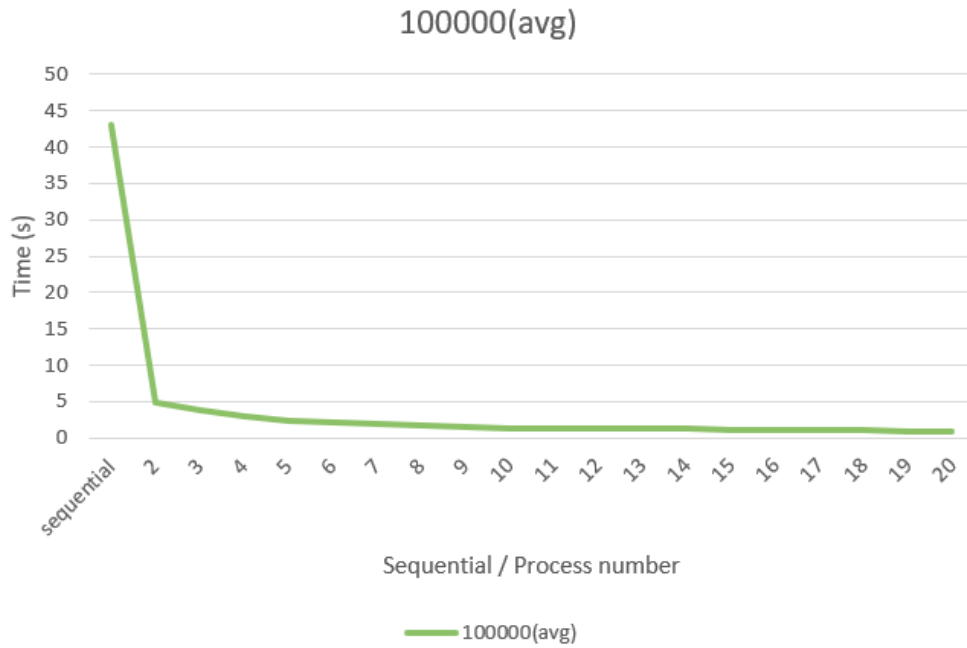


Figure 26: The comparison between sequential and MPI: the large array size (100000) case

From the above graph, we can see that for both size, the running time of MPI cases are smaller than that of the sequential case, and when the number of process increases, the running time decreases, but the effect of improvement decreases as the number of processes increases. The improvement is most noticeable when going from single process to dual process. We also need to pay attention that some time differences may because of the different implementation of these two algorithm, for sequential case, I use a while loop and a flag to check whether there is a swap, to determine the end condition. But in MPI case, I use a big for loop to implement it.

5 Conclusion

To sum up, this Homework 1 experiments can support the relationship between the parallel computation process and the operating rate mentioned in the lecture. According to the experiment results, we can find that for the small array, the running time does not decrease significantly with the increase of the number of processes, and it fluctuates. This is because when the size is small, the time gap is small, and the time of communication between different processes is comparable with the running time. So when the array size is small, the use of multi-processing is not obvious for the improvement of the effect, which will cause a waste of resources. Fewer processes are recommended when array size is small to fully utilize computing resources.

From the results medium array size case and large array size case, we can find that as the number of processes increases, the running time decreases significantly, but the effect of improve-

ment decreases as the number of processes increases. The improvement is most noticeable when going from single process to dual process. For arrays of these sizes, it is recommended to appropriately increase the number of processes to greatly improve the running speed and make full use of computing resources.

From the result of the comparison between sequential case, we can see that for both size, the running time of MPI cases are smaller than that of the sequential case, and when the number of process increases, the running time decreases, but the effect of improvement decreases as the number of processes increases. The improvement is most noticeable when going from single process to dual process. We also need to pay attention that some time differences may because of the different implementation of these two algorithm, for sequential case, I use a while loop and a flag to check whether there is a swap, to determine the end condition. But in MPI case, I use a big for loop to implement it.

Bibliography

- Wikipedia (2022a). *Message Passing Interface*. URL: https://en.wikipedia.org/wiki/Message_Passing_Interface (visited on 26th Sept. 2022).
- (2022b). *Odd–even sort*. URL: https://en.wikipedia.org/wiki/Odd%E2%80%9393even_sort (visited on 19th Mar. 2022).
- Worsch, Thomas (2017). *Five Lectures on CA*. URL: <http://linwww.ira.uka.de/~thw/vl-hiroshima/slides-4.pdf> (visited on 30th July 2017).

Appendix

A Sequential Version Odd-even Sort Code

```
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <chrono>

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

    /* TODO BEGIN
       Implement sequential odd even transposition sort
       Code in this block is not a necessary.
       Replace it with your own code.
    */

    bool hasBeenSorted;
    while(!hasBeenSorted){
        hasBeenSorted=true;
        for(int i=0;i<num_elements-1;i+=2){
            if(elements[i]>elements[i+1]){
```

```

        std::swap(elements[i],elements[i+1]);
        hasBeenSorted=false;
    }
}
for(int i=1;i<num_elements-1;i+=2){
    if(elements[i]>elements[i+1]){
        std::swap(elements[i],elements[i+1]);
        hasBeenSorted=false;
    }
}
}

for (int i = 0; i < num_elements; i++) {
    sorted_elements[i] = elements[i];
}
/* TODO END */

t2 = std::chrono::high_resolution_clock::now();
time_span = std::chrono::duration_cast<std::chrono::duration<double>>(t2
→ - t1);
std::cout << "Student ID: " << "120090414" << std::endl; // replace it
→ with your student id
std::cout << "Name: " << "Xiang Fei" << 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;
}

```

B Parallel Version Odd-even Sort Code

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

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 num_my_element = num_elements / world_size; // number of elements
    ↪ allocated to each process
    int remainder = num_elements - num_my_element*world_size;
    int num_elements_extended = num_elements+world_size-remainder;

    int elements_extended[num_elements_extended];
    int sorted_elements[num_elements]; // store sorted elements
    int sorted_elements_extended[num_elements_extended];

    if (rank == 0) { // read inputs from file (master process)
        std::ifstream input(argv[2]);
        int element;
        int i = 0;
        while (input >> element) {
            elements_extended[i] = element;
            i++;
        }
    }
```

```

    for(int i=num_elements;i<num_elements_extended;i++){
        elements_extended[i] = INT_MAX;
    }
    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
}

/* TODO BEGIN
    Implement parallel odd even transposition sort
    Code in this block is not a necessary.
    Replace it with your own code.
    Useful MPI documentation: https://rookiehpc.github.io/mpi/docs
*/

num_my_element = num_elements_extended/world_size;
int my_element[num_my_element]; // store elements of each process
MPI_Scatter(elements_extended, num_my_element, MPI_INT, my_element,
↪ num_my_element, MPI_INT, 0, MPI_COMM_WORLD); // distribute elements to
↪ each process

int send_element_head;
int send_element_tail;
int recv_element;

for(int i=0;i<num_elements;i++){
    if(i%2==0){
        for(int j=0;j<num_my_element-1;j+=2){
            if(my_element[j]>my_element[j+1]){
                std::swap(my_element[j],my_element[j+1]);
            }
        }
    }
    else{
        for(int j=1;j<num_my_element-1;j+=2){
            if(my_element[j]>my_element[j+1]){
                std::swap(my_element[j],my_element[j+1]);
            }
        }
    }
}

```

```

        }
    }
    if(rank!=0){
        send_element_head = my_element[0];
↪ MPI_Sendrecv(&send_element_head,1,MPI_LONG,rank-1,0,&recv_element,1,MPI_LONG,rank-1,0,MPI_COMM_WORLD);
        if(recv_element>my_element[0]){
            my_element[0]=recv_element;
        }
    }
    if(rank!=world_size-1){
        send_element_tail = my_element[num_my_element-1];
↪ MPI_Sendrecv(&send_element_tail,1,MPI_LONG,rank+1,0,&recv_element,1,MPI_LONG,rank+1,0,MPI_COMM_WORLD);
        if(my_element[num_my_element-1]>recv_element){
            my_element[num_my_element-1]=recv_element;
        }
    }
}

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

for(int i=0;i<num_elements;i++){
    sorted_elements[i] = sorted_elements_extended[i];
}

/* TODO END */

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: " << "120090414" << std::endl; // replace
↪ it with your student id
    std::cout << "Name: " << "Xiang Fei" << 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: " << 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;
}

```

C The detailed result of comparison regarding to number of cores and array size

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1000(1)	2.61E-03	0.002289	0.002006	0.00184	0.001911	0.001882	0.00189	0.001887	0.001865	0.002313	0.003035	0.002287	0.002426	0.001979	0.00199	0.002616	0.001506	0.001972	0.002412	0.002091
1000(2)	2.61E-03	0.002237	0.001918	0.001613	0.002042	0.002014	0.002291	0.001972	0.002203	0.002769	0.003999	0.001953	0.002277	0.002379	0.001756	0.002228	0.001788	0.002061	0.001472	0.002911
1000(3)	2.24E-03	0.002159	0.001899	0.002192	0.001687	0.001885	0.001742	0.002049	0.002106	0.002596	0.002568	0.002862	0.002604	0.002222	0.001703	0.002671	0.001895	0.00153	0.002245	0.001516
1000(4)	2.60E-03	0.002172	0.001999	0.002194	0.001943	0.001698	0.002095	0.002127	0.002284	0.002435	0.002625	0.002433	0.003315	0.00176	0.002066	0.001851	0.002287	0.002053	0.002256	0.00146
1000(5)	2.60E-03	0.002173	0.002112	0.001882	0.001807	0.001882	0.001947	0.00229	0.002163	0.002596	0.00358	0.003087	0.002132	0.00176	0.002587	0.001998	0.001857	0.001555	0.002521	0.001542
1000(avg)	2.53E-03	2.21E-03	1.99E-03	1.94E-03	1.87E-03	1.87E-03	1.99E-03	2.07E-03	2.12E-03	2.54E-03	3.16E-03	2.52E-03	2.55E-03	2.02E-03	2.02E-03	2.28E-03	1.86E-03	1.83E-03	2.18E-03	1.90E-03
10000(1)	0.100663	0.060165	0.048392	0.042229	0.037917	0.034205	0.03359	0.029803	0.028592	0.02909	0.027966	0.026237	0.025514	0.022896	0.024547	0.021776	0.022392	0.022867	0.020196	0.019567
10000(2)	0.103101	0.06084	0.049179	0.043242	0.036332	0.038991	0.03359	0.030519	0.030307	0.028102	0.029976	0.026709	0.02574	0.027186	0.022318	0.024998	0.02161	0.023458	0.022976	0.019081
10000(3)	0.103361	0.060238	0.049903	0.041155	0.039561	0.03547	0.03267	0.031952	0.028301	0.02885	0.026247	0.027496	0.025774	0.025049	0.025139	0.024791	0.019697	0.023895	0.022242	0.019051
10000(4)	0.103316	0.058402	0.04912	0.044814	0.038963	0.035815	0.03351	0.027433	0.031039	0.026281	0.029344	0.028943	0.022593	0.0225	0.023555	0.022343	0.020173	0.023239	0.024313	0.022235
10000(5)	0.103951	0.058507	0.048772	0.045597	0.037206	0.034604	0.035503	0.033181	0.031039	0.026029	0.028906	0.029256	0.026133	0.026175	0.023836	0.023448	0.022545	0.020442	0.022866	0.019742
10000(avg)	0.1028784	0.05963	0.049073	0.043407	0.037996	0.035817	0.033772	0.030577	0.029855	0.02767	0.028488	0.027728	0.025151	0.024761	0.023879	0.023471	0.021283	0.02278	0.022519	0.019935
100000(1)	10.0153	4.86717	3.85372	3.06942	2.40748	2.17782	1.9038	1.73233	1.51409	1.30604	1.21616	1.341	1.23732	1.28823	1.16866	1.03045	1.10259	1.01221	0.99043	0.818067
100000(2)	10.0466	4.87983	3.94366	3.034	2.44272	2.10785	1.94082	1.70043	1.55662	1.3138	1.23351	1.26817	1.29624	1.19436	1.2158	1.00709	1.0816	1.03679	0.8887	0.842308
100000(3)	10.1415	4.86889	3.88751	3.01944	2.47182	2.11288	1.9008	1.72046	1.53653	1.32145	1.27701	1.23275	1.26821	1.31335	1.18429	1.17897	1.06448	1.02643	0.879465	0.892166
100000(4)	10.0466	4.86192	3.83677	3.06314	2.50018	2.17367	1.90213	1.68438	1.579	1.43329	1.37527	1.23408	1.29008	1.22173	1.27533	1.04751	1.06197	1.04048	0.987252	0.969959
100000(5)	10.2156	4.914	3.84015	3.15263	2.42524	2.12791	1.90508	1.67293	1.51498	1.33443	1.3368	1.34562	1.32865	1.2605	1.2139	1.20133	1.10705	1.01139	0.950853	0.904319
100000(avg)	10.09312	4.878362	3.872362	3.067726	2.449488	2.140026	1.910526	1.702106	1.540244	1.341802	1.28775	1.284324	1.2841	1.255634	1.211596	1.09307	1.083538	1.01818	0.93934	0.885364

Figure 27: The detailed result of comparison regarding to number of cores and array size

D The detailed result of comparison between sequential and MPI

	sequential	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1000(1)	0.0112232	0.002289	0.002006	0.00184	0.001911	0.001882	0.00189	0.001887	0.001865	0.002313	0.003035	0.002287	0.002426	0.001979	0.00199	0.002616	0.001506	0.001972	0.002412	0.002091
1000(2)	0.0108909	0.002237	0.001918	0.001613	0.002042	0.002014	0.002291	0.001972	0.002203	0.002769	0.003999	0.001953	0.002277	0.002379	0.001756	0.002228	0.001788	0.002061	0.001472	0.002911
1000(3)	0.0101204	0.002159	0.001899	0.002192	0.001687	0.001885	0.001742	0.002049	0.002106	0.002596	0.002568	0.002862	0.002604	0.002222	0.001703	0.002671	0.001895	0.00153	0.002245	0.001516
1000(4)	0.0107941	0.002172	0.001999	0.002194	0.001943	0.001698	0.002095	0.002127	0.002284	0.002435	0.002625	0.002433	0.003315	0.00176	0.002066	0.001851	0.002287	0.002053	0.002256	0.00146
1000(5)	0.0114871	0.002173	0.002112	0.001882	0.001807	0.001882	0.001947	0.00229	0.002163	0.002596	0.00358	0.003087	0.002132	0.00176	0.002587	0.001998	0.001857	0.001555	0.002521	0.001542
1000(avg)	0.0109031	2.21E-03	1.99E-03	1.94E-03	1.87E-03	1.87E-03	1.99E-03	2.07E-03	2.12E-03	2.54E-03	3.16E-03	2.52E-03	2.55E-03	2.02E-03	2.02E-03	2.28E-03	1.86E-03	1.83E-03	2.18E-03	1.90E-03
10000(1)	0.423745	0.060165	0.048392	0.042229	0.037917	0.034205	0.03359	0.029803	0.028592	0.02909	0.027966	0.026237	0.025514	0.022896	0.024547	0.021776	0.022392	0.022867	0.020196	0.019567
10000(2)	0.42634	0.06084	0.049179	0.043242	0.036332	0.038991	0.03359	0.030519	0.030307	0.028102	0.029976	0.026709	0.02574	0.027186	0.022318	0.024998	0.02161	0.023458	0.022976	0.019081
10000(3)	0.426196	0.060238	0.049903	0.041155	0.039561	0.03547	0.03267	0.031952	0.028301	0.02885	0.026247	0.027496	0.025774	0.025049	0.025139	0.024791	0.019697	0.023895	0.022242	0.019051
10000(4)	0.425889	0.058402	0.04912	0.044814	0.038963	0.035815	0.03351	0.027433	0.031039	0.026281	0.029344	0.028943	0.022593	0.0225	0.023555	0.022343	0.020173	0.023239	0.024313	0.022235
10000(5)	0.43184	0.058507	0.048772	0.045597	0.037206	0.034604	0.035503	0.033181	0.031039	0.026029	0.028906	0.029256	0.026133	0.026175	0.023836	0.023448	0.022545	0.020442	0.022866	0.019742
10000(avg)	0.426802	0.05963	0.049073	0.043407	0.037996	0.035817	0.033772	0.030577	0.029855	0.02767	0.028488	0.027728	0.025151	0.024761	0.023879	0.023471	0.021283	0.02278	0.022519	0.019935
100000(1)	42.7391	4.87983	3.94366	3.034	2.44272	2.10785	1.94082	1.70043	1.55662	1.3138	1.23351	1.26817	1.29624	1.19436	1.2158	1.00709	1.0816	1.03679	0.8887	0.842308
100000(2)	42.6981	4.86889	3.88751	3.01944	2.47182	2.11288	1.9008	1.72046	1.53653	1.32145	1.27701	1.23275	1.26821	1.31335	1.18429	1.17897	1.06448	1.02643	0.879465	0.892166
100000(3)	42.6981	4.86889	3.88751	3.01944	2.47182	2.11288	1.9008	1.72046	1.53653	1.32145	1.27701	1.23275	1.26821	1.31335	1.18429	1.17897	1.06448	1.02643	0.879465	0.892166
100000(4)	43.6321	4.86192	3.83677	3.06314	2.50018	2.17367	1.90213	1.68438	1.579	1.43329	1.37527	1.23408	1.29008	1.22173	1.27533	1.04751	1.06197	1.04048	0.987252	0.969959
100000(5)	42.8832	4.914	3.84015	3.15263	2.42524	2.12791	1.90508	1.67293	1.51498	1.33443	1.3368	1.34562	1.32865	1.2605	1.2139	1.20133	1.10705	1.01139	0.950853	0.904319
100000(avg)	43.00816	4.878362	3.872362	3.067726	2.449488	2.140026	1.910526	1.702106	1.540244	1.341802	1.28775	1.284324	1.2841	1.255634	1.211596	1.09307	1.083538	1.01818	0.93934	0.885364

Figure 28: The detailed result of comparison between sequential and MPI